KATHOLIEKE UNIVERSITEIT LEUVEN
FACULTEIT WETENSCHAPPEN
FACULTEIT TOEGEPASTE WETENSCHAPPEN
DEPARTEMENT COMPUTERWETENSCHAPPEN
Celestijnenlaan 200 A — 3001 Leuven

ITERATIVE VERSIONSPACES WITH AN APPLICATION IN INDUCTIVE LOGIC PROGRAMMING

Jury :
Prof. Dr. ir. Y.D. Willems, voorzitter
Prof. Dr. ir. M. Bruynooghe, promotor
Prof. ir. M. Gobin, promotor
Prof. Dr. L. De Raedt
Prof. Dr. ir. K. De Vlaminck
Prof. Dr. N. Lavrač (J.S.I., Slovenia)
Prof. Dr. C. Mellish (Univ. of Edinburgh, UK)

Proefschrift voorgedragen tot het behalen van het doctoraat in de Informatica

door

Gunther SABLON

U.D.C. 681.3*I26

May 1995

Iterative Versionspaces with an application in Inductive Logic Programming

Gunther Sablon
Department of Computer Science, K.U.Leuven

Abstract

This thesis consists of two parts: in the first part we develop a language-independent framework for efficiently solving concept learning problems; in the second part we apply this framework to Inductive Logic Programming.

We view concept learning as a search problem. In the well-known framework of Versionspaces a bi-directional depth-first search algorithm that learns a maximally general and maximally specific concept representation is presented, and contrasted to the breadth-first approach of Mellish's Description Identification algorithm. In this context, we identify redundant information elements, in order to reduce the memory needed for storing information elements. We describe how automatically generated information elements can replace less informative ones.

Next, we extend this framework to describe the more complex versionspaces that originate from introducing disjunctions. To be practically useful, we gradually restrict these disjunctive versionspaces by imposing preference criteria, based on notions of minimality. This leads to extensions of the non-disjunctive algorithms to the disjunctive case.

In the second part of the thesis we show in detail how this general framework can be instantiated to Inductive Logic Programming. In this respect we also discuss the integration of machine learning in a planning system based on Horn clause logic. This illustrates that the use of a logical representation allows a smooth integration of Machine Learning and Problem Solving.

In summary, the thesis contributes to the understanding and the development of search algorithms for concept learning in general, by developing a language independent framework, and by introducing several novel and generally applicable concept learning techniques. The application in the second part of the thesis shows that the framework is practically useful, and that it contributes to the field of Inductive Logic Programming.

Keywords: Machine Learning, Concept Learning, Inductive Logic Programming

Acknowledgements

I would like to thank many people with whom I have worked together and who have contributed to this work in one way or another.

First, I wish to thank Maurice Bruynooghe, for his advice, his confidence and his support during six years. He guided my research all that time, and in that showed how to do scientific research. He always insisted on precision and clarity, which often made things easier to understand.

I am also grateful to Marc Gobin, for being co-promotor of my thesis, and for his interest in the subject. I also want to thank all members of the Jury, for their valuable comments on the first version of the thesis.

For six years I have also been a teaching assistant at this Department, mainly for Yves Willems. I wish to thank him for his confidence in me in fulfilling this task. I also want to thank him for his interest in my research. And I am grateful he gave me the opportunities of sharing our knowledge with less gifted universities.

At that time I was called "the Namibian connection" by Luc De Raedt. He was the closest colleague during these six years. First of all, I want to thank him for starting Machine Learning research at our Department, and for making me interested in the subject. He has strongly influenced my research all the time (and he still does) through the many fruitful discussions and the intense cooperation. I also thank him for reading this work several times, each time coming up with more improvements. He as well often insisted on clarity and structure, and on relating our work to other work.

I also want to thank the other members of the Machine Learning group (Hilde Adé, Wim Van Laer, Luc Dehaspe and Hendrik Blockeel) for the many discussions, and for reading parts of this work. I also want to thank the people that visited our group for broadening my views in many respects.

I am grateful to Lode Missiaen for introducing me to the Event Calculus, which has for a while strongly influenced my research; to the members of the Logic Programming and Artificial Intelligence group, for making me interested in Logic Programming, and for the subsequent influence on my understanding of Inductive Logic Programming.

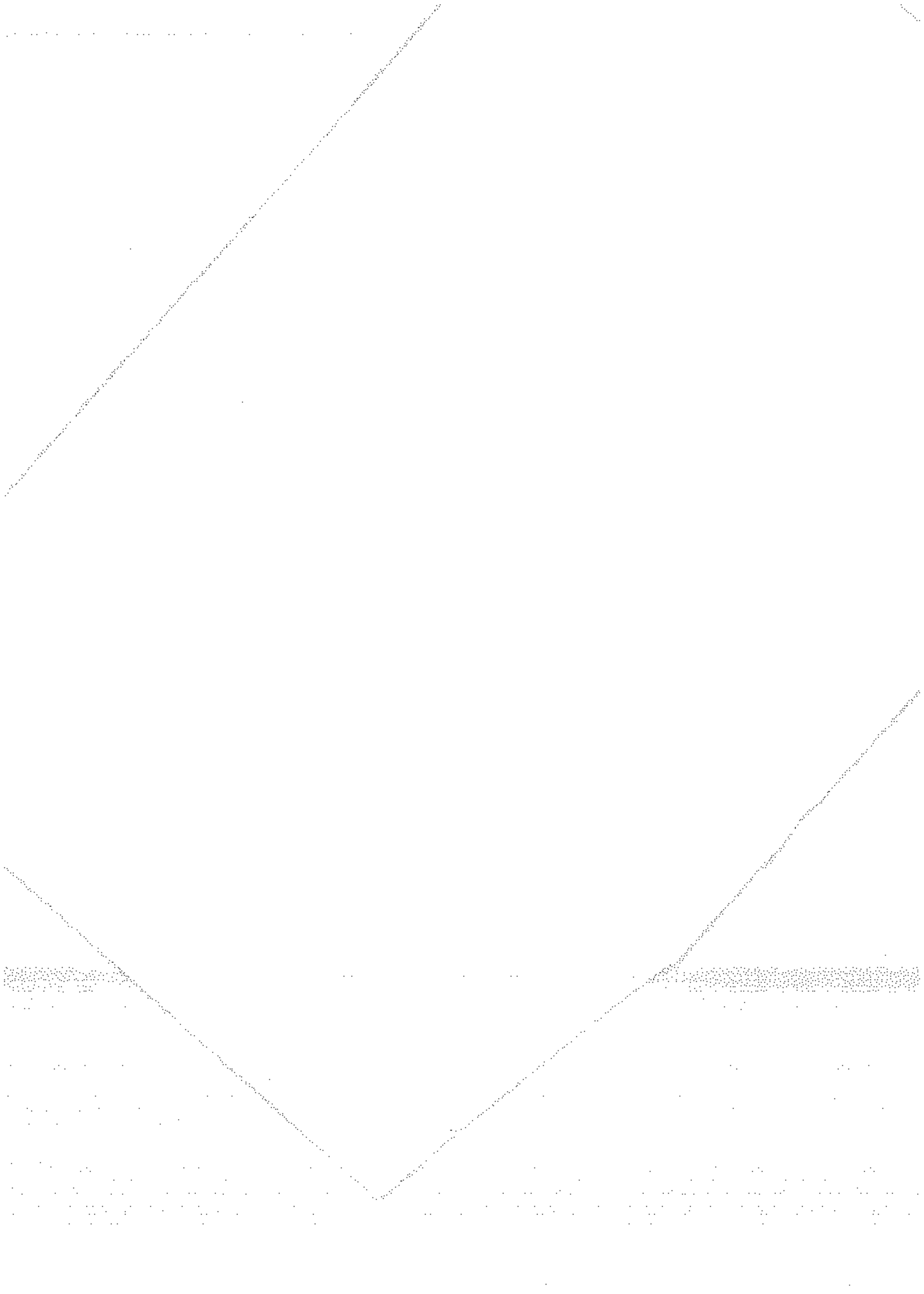
Regarding organizational aspects, I am grateful to Denise Brams for having disposition of a separate office during the writing of my thesis; to Lode Missiaen and Veroniek Dumortier for borrowing me their TeX and LaTeX constructions, and to Hilde Adé for helping me whenever LaTeX was making trouble again.

I want to thank the K.U.Leuven, the Belgian National Fund for Scientific Research, the European Community (Esprit BRA 6020 and MLNet) for the financial support in doing research and in attending conferences and workshops.

I also want to thank my parents, for their great support, my sister and many friends. Finally, I wish to thank Hilde, once more, for her love: she looked after me when I was writing this thesis during all those long evenings.

Heverlee, May 11, 1995.

Contents



List of Figures

List of Algorithms

List of Definitions

List of Symbols

iff: if and only if
$\Box$: end of proof
$\diamond$: end of example

Sets and lists

$\in$: ... is an element of ...
$\notin$: ... is no element of ...
$\subseteq$: ... is a subset of ...
$\not\subseteq$: ... is no subset of ...
$\subset$: ... is a proper subset of ...
$\not\subset$: ... is not a proper subset of ...
$\cup$: set union
$\setminus$: set difference
$\emptyset$: the empty set, or the empty list
$\mid S \mid$: the cardinality of the set S
$\mathcal{P}(S)$: the powerset of the set S
$Max\ S$: the maximal elements of S (w.r.t. $\preccurlyeq$)
$Min\ S$: the minimal elements of S (w.r.t. $\preccurlyeq$)
$[c]_{\equiv}$: the equivalence class of c w.r.t. the equivalence relation $\equiv$
ρ^{tc}: the transitive closure of the function ρ
$[1, 2, 3, \ldots]$: non-empty list of the elements $1, 2, 3, \ldots$
$\uplus$: appends two lists
$A[j]$: j-th element of the array A

Concept Learning

$\mathcal{I}$: the set of all instances
$\mathcal{C}$: the set of all concepts
$\mathcal{L}_I$: the language of instance representations
$\mathcal{L}_C$: the language of concept representations
$\mathcal{U}$: the set of unit concept representations
$\mathcal{B}$: the background knowledge
R_I: maps instance representations to instances
R_C: maps concept representations to concepts
t: the target concept
$cover$: the cover of a concept representation

covers: … covers …
$\preccurlyeq$: … is more specific than …
$\succcurlyeq$: … is more general than …
$\prec$: … is strictly more specific than …
$\succ$: … is strictly more general than …
$\trianglelefteq$: … is a refinement of …
$\sim$: … is consistent with …
$\top$: the top element of $\mathcal{L}_C$
$\bot$: the bottom element of $\mathcal{L}_C$
I_s: the set of all s-bounds in I (in the algorithms represented as an array)
I_g: the set of all g-bounds in I (in the algorithms represented as an array)
$I_s|J_s$: the set of all elements (i.e., s-bounds) in I_s that have an index in J_s
I_s^+: set of positive lowerbounds in I_s
$\mathcal{I}nf$: infostream
mub: minimal upperbounds
mlb: maximal lowerbounds
mgs: most general specializations
msg: most specific generalizations
$\mathcal{S}_I$: the set of maximally specific elements of $\mathcal{L}_C$ consistent with I
$\mathcal{G}_I$: the set of maximally general elements of $\mathcal{L}_C$ consistent with I
$\mathcal{VS}_I$: the set of elements of $\mathcal{L}_C$ consistent with I

Disjunctive Concept Learning

$\mathcal{DL}_C$: the language of reduced disjunctive concept representations
$cover_d$: the cover of a disjunctive concept representation
$covers_d$: … covers … in the disjunctive case
$\preccurlyeq_d$: … is more specific than … in the disjunctive case
$\succcurlyeq_d$: … is more general than … in the disjunctive case
$\prec_d$: … is strictly more specific than … in the disjunctive case
$\succ_d$: … is strictly more general than … in the disjunctive case
mub_d: minimal upperbounds in the disjunctive case
mlb_d: maximal lowerbounds in the disjunctive case
mgs_d: most general specializations in the disjunctive case
msg_d: most specific generalizations in the disjunctive case
$\mathcal{DG}_I$: the set of maximally general elements of $\mathcal{DL}_C$ consistent with I
$\mathcal{DS}_I$: the set of maximally specific elements of $\mathcal{DL}_C$ consistent with I
$\mathcal{DVS}_I$: the set of elements of $\mathcal{DL}_C$ consistent with I
$\mathcal{DG}_{\sqsubseteq,I}$: the set of subdisjunctions of the maximally general element of $\mathcal{DL}_C$ consistent with I
$\mathcal{ADS}_{\sqsubseteq,I}$: the set of almost maximally specific elements of $\mathcal{DL}_C$ under an element of $\mathcal{DG}_{\sqsubseteq,I}$
$\mathcal{ADVS}_{\sqsubseteq,I}$: the set of disjunctions between an element of $\mathcal{DG}_{\sqsubseteq,I}$ and an element of $\mathcal{ADS}_{\sqsubseteq,I}$
MS: minimal set preference criterion
ML: minimal length preference criterion
$[d]_r$: the reduction of the disjunction d
$\#d$: the length of the disjunction d

Complexity analysis

$\bar{s}$: the size of the specific-to-general search space
$\bar{g}$: the size of the general-to-specific search space
b_s: the branching factor of the specific-to-general search space
b_g: the branching factor of the general-to-specific search space
$\bar{\mathcal{S}}$: the size of $\mathcal{S}$
$\bar{\mathcal{G}}$: the size of $\mathcal{S}$
c_i: space complexity of an information element
c_c: space complexity of a concept representation
c_{ind}: space complexity of an index to an s-bound
$c_{\preccurlyeq}$: time complexity of a $\preccurlyeq$-test
c_{gen}: time complexity of a generalization operation
c_{spec}: time complexity of a specialization operation

Logic and Logic Programming

$\vee$: logical disjunction
$\wedge$: logical conjunction
$\mathcal{V}$: set of variables
$\mathcal{F}$: set of functors
$\mathcal{P}$: set of predicates
$\mathcal{T}_{\mathcal{F},\mathcal{V}}$: the set of all terms built with $\mathcal{F}$ and $\mathcal{V}$
$\mathcal{A}_{\mathcal{P},\mathcal{F},\mathcal{V}}$: the set of all atoms built with $\mathcal{P}$, $\mathcal{F}$ and $\mathcal{V}$
$\mathcal{CL}_{\mathcal{P},\mathcal{F},\mathcal{V}}$: the set of all clauses built with $\mathcal{P}$, $\mathcal{F}$ and $\mathcal{V}$
$Sat(\ c\ )$: the saturation of clause c
lgg: least general generalization
lss: least specific specialization
:-) : smile
$\ominus$: removes a set of literals from a clause
$\oplus$: adds a set of literals to a clause
$\leftarrow$: implication in clauses
$c = h \leftarrow b$ is to be read as $c = (\ h \leftarrow b\ )$ (i.e., c is the clause with head h and body b)
$\preccurlyeq_\theta$: ... θ-subsumes ... w.r.t. $\mathcal{B}$
$\equiv_\theta$: ... is equivalent to ... w.r.t. θ-subsumption
$\preccurlyeq_P$: ... subsumes ... w.r.t. program P
$\equiv_P$: ... is equivalent to ... w.r.t. program P
$\preccurlyeq_\mathcal{B}$: ... subsumes ... w.r.t. $\mathcal{B}$
$\equiv_\mathcal{B}$: ... is equivalent to ... w.r.t. $\mathcal{B}$
$\Box$: the empty clause

$\ll$: ... is before ... (time relation)
$\leqslant$: ... is before or equal to ... (time relation)

Algorithms

- **procedure** *Name*($Arg_1 : Type_1$, ..., $Arg_n : Type_n$) **returns** $Type'_1$, ..., $Type'_m$: procedure or function heading
- **if** *Condition* **then** *Statement* **else** *Statement*: the usual if-then-else construct
- **if** *Condition* **then** *Statement*: the usual if-then construct (the **else** part is empty)
- **repeat** *Statement* **until** *Condition*: the usual repeat loop
- **while** *Condition* **do** *Statement* **endwhile**: the usual while loop
- **for all** *Selector* **do** *Statement* **endfor**: executes *Statement* for each element returned by *Selector*
- **select all** *Element* **from** *Structure* **with** *Condition*(*Element*): returns all elements *Element* from *Structure* such that *Condition*(*Element*) returns *true*
- **select one** *Element* **from** *Structure* **with** *Condition*: returns one element *Element* from *Structure* such that *Condition*(*Element*) returns *true*
- **return** X_1 , ..., X_m: halts the procedure and return the values X_1 , ..., X_m
- *failure* : halts the surrounding constructs, the procedure, and the calling procedures until a call is found which is followed by a "**if** *failure* **then**" statement
- **endproc**: end of procedure or function text
- X'_1 , ..., $X'_m := p(X_1 , \ldots , X_n)$: the procedure or function p, which has n arguments, is called with X_1 , ..., X_n as actual values. The parameter mechanism is in principle "call by value". Sometimes we explicitly stress this by copying the argument at the beginning of the procedure. Often the "call by reference" can be used instead. The call returns m values, which are assigned to X'_1 , ..., X'_m respectively
- { *Comment* }: comments are surrounded by curly braces

Chapter 1

Introduction

Machine Learning (ML) originated in the context of *Artificial Intelligence (AI)*, mainly motivated by the observation that no intelligence is possible without learning (e.g., [Rich and Knight, 1991]). Consequently, machines that are to behave intelligently must be able to learn. Many researchers have tried to give a definition for learning. The key point in all definitions is that learning systems *adapt themselves*. The stimulus for the adaptation can be external (e.g., the observation of changes in the environment), or internal (e.g., a reasoning process that leads to new insights). [Michalski, 1986] distinguishes two elements that can be adapted: *knowledge* and *skills*. Hence, two forms of learning can be distinguished: *knowledge acquisition* and *skill acquisition*. The term "knowledge" refers to pieces of information that the intelligent system uses to infer solutions for the problems it has to solve. The term "skills" refers to perceptive, motoric and cognitive abilities that are to be *trained* in order to improve them. Knowledge is used for reasoning; skills are used for acting and reacting. Classical AI has been concerned with knowledge-based problem solving, e.g., in the development of rule-based systems, planning, theorem proving, game-playing, ...Recent research areas such as reinforcement learning and reactive planning show that automated skill acquisition also belongs to the domain of AI. In this thesis we are concerned with Machine Learning in the knowledge acquisition sense.

A *knowledge-based system* or *problem solver* uses its knowledge to solve problems using one or more inference mechanisms. From our initial point of view we can say that such a system can only be considered intelligent if it also contains capabilities to acquire new knowledge or to update its current knowledge. A knowledge-based system can contain several types of knowledge: domain-specific knowledge, knowledge about how to solve particular (types of) problems, knowledge about how to solve particular (types of) problems *efficiently*, knowledge about solutions to previous problems, knowledge about itself, etc. To acquire this kind of knowledge the knowledge-based system could apply several learning techniques: learning by induction on several problem solutions, learning by analogy with previous problem solutions, learning from experience from previous problem solutions, etc. In general a knowledge based system can thus use several kinds of problem solving inference mechanisms, it could contain several types of knowledge and it could acquire knowledge by means of several different learning mechanisms. This gives rise to a problem of *integration*: how to let problem solving and learning cooperate on several types of knowledge? Or, if one views learning as just another problem solving task: how to let several kinds of inference mechanisms cooperate in general?

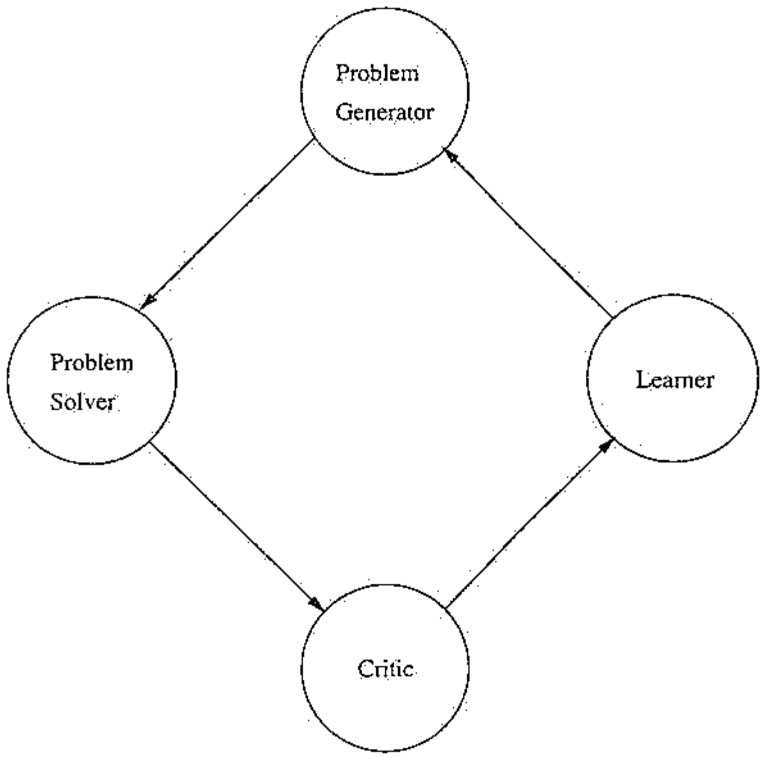


Figure 1.1 Learning in between solving problems

1.1 Integrating Machine Learning and Problem Solving

In the field of machine learning, integrating several kinds of learning mechanisms gave rise to a new area of research called *Multistrategy Learning* [Michalski and Tecuci, 1994]. In this thesis we will rather focus on the problem of integrating one type of problem solving mechanism with one type of learning mechanism. We will start from the point of view of having a problem solving system that acquires new knowledge in between solving problems. The knowledge it acquires must be processed (i.e., must be learned), such that it can be used later for solving more problems. The system learns *incrementally* in the sense that new information is processed as it comes, and once processed, it can immediately be used by the problem solver. At the same time, we let the system be curious: it can invent new problems for which a solution would be relevant in the learning process, and thus for future problem solving. This means that the system learns *interactively*: an interactive process between the learner and the problem solver provides new relevant information.

The basic cycle of a system of this kind is represented in Figure 1.1. This diagram originated in the context of LEX [Mitchell *et al.*, 1983]. Problems are solved by the problem solver. The solutions are evaluated by the critic. During the evaluation of a proposed solution the solution might turn out to be incorrect, due to erroneous knowledge in the system; or, the solution might at the same time solve some other problems, an effect that was unexpected. The result of the evaluation is used by the learner to update the knowledge, in order to find better solutions for future problems. The problem generator generates new problems; the feedback of solving and evaluating these problems can give relevant information to learn from.

To focus on an example, suppose the problem solver is a planning system. Given a certain goal, the planner makes a plan. To evaluate the plan, the plan is executed. The execution of certain actions in the plan might fail, or might not have the desired outcome. Others might have unexpected effects. Yet others have exactly the expected outcome. Given this feedback, the learner updates the knowledge, if necessary, to improve the planning process on future problems. Then the problem generator designs a new planning problem, the solution of which will be executed, and so on. In LEX, this design was used to learn heuristics for solving symbolic integration problems.

The learning system assimilates all knowledge it receives from the critic, knowledge about particular solutions (or even parts of solutions) together with their evaluation. The learned knowledge then ought to be useful, not only for solving the same problem again, but also for solving other, more or less similar, problems. [Michalski, 1986] distinguishes three criteria for evaluating the quality of the learned knowledge w.r.t. the problem solver: the knowledge's validity (i.e., its degree with which it corresponds to reality), its effectiveness (i.e., its degree of usefulness of the knowledge in achieving the goals of the problem solver), and its abstraction level (i.e., its scope and degree of detail). The quality of the learned knowledge along these three dimensions in a sense also determines the quality of the learner.

1.2 Integration in a logical representation

Whereas the previous section describes the problem of integrating a problem solver and a learner in terms of knowledge in general, we will now take a look at a lower, more symbolic, level, on which we are concerned with representation formalisms.

In this thesis we will concentrate on one particular type of learning: learning of concepts from examples. In the context of the previous section, examples are particular solutions to problems. These solutions could have been correct solutions, or not; this is to be determined by the critic. Concepts are sets of correct solutions in which common *general rules* for solving the problem are applied. The idea behind concept learning is to induce these general rules, from the particular solutions, together with their evaluation class.

The three criteria *validity*, *effectiveness*, and *abstraction* can also be evaluated on this level. First, it is important for the validity of the learned knowledge that the learner and the problem solver interpret the representation in the same way. Second, it is important that both mechanisms can use the chosen representation. This supposes that for both components the representation can at least be transformed to a representation that can be used by the component. Third, it is necessary that both systems use the same abstraction level of symbols, or again can transform their knowledge to a common level of abstraction.

We argue that on each of these points the use of a *logical* representation formalism common to the problem solver and the learner is a natural way of integration. Because of its mathematical well-foundedness, a logical representation allows a clear view on the semantics of the used representation. Furthermore logic provides a well-defined framework to describe the relations between problems and their solutions. The expression of these relations in a logical representation formalism can be used for solving problems in many problem domains. One such domain is, for instance, planning where the combined use of *deduction* and *abduction* proves to be useful (see Chapter 6). If we view the learner as a problem solver itself, another such domain is Inductive Logic Programming, which solves the learning task in a logical framework (see Chapter 5). Finally, by using a common logical language, the problem solver and the learner can use knowledge of the same abstraction level.

Another important feature of a logical representation formalism is its expressiveness, and therefore its applicability in a wide number of domains. A logical representation also allows a declarative view on knowledge: knowledge can be seen independent of the way or the system it is going to be used in, it is modular, and it allows a declarative reading by humans (see [Omar, 1994]).

Some people criticize the use of logical representation formalisms, arguing that it is not feasible to implement logic-based systems on a large scale. Another critique is that the nature of knowledge is not logical and declarative, but procedural, and that it must be compiled to be effective. We will not argue this might not be so, but even if the latter critiques would turn out to be fundamental in implementing AI-systems, we believe it is still useful to use logical representations to study these systems, and to study the fundamental problems with these systems. In particular, on the problem of integrating problem solving and learning, if the use of logic allows a straightforward representation for problem solving and for learning, effort can be put in the study of other problems of the integrated architecture, such as the problem of controlling when to solve problems, when to learn, when to generate new problems, etc.

In this we want to take the same viewpoint as [Buntine, 1987] takes on the use of Horn Clause representations in machine learning:

> We believe that having a good understanding of inductive learning and its applications in the context of Horn Clauses, we can improve our understanding of comparable problems in more extensive knowledge representations, for instance, ones incorporating uncertainty or using a different subset of first-order logic - as often found in knowledge-based systems.

We argue that, even if there are technical difficulties in using logical representation formalisms effectively in knowledge-based systems, it is still useful to describe the problem of integration and possible solutions in a logical representation, because of the clear mathematical framework logic provides.

1.3 Integrating ML and Problem Solving in practice

Let us take a look at how the integration of one problem solving strategy and one learning strategy is realized in existing systems.

Adapting the knowledge base of knowledge-based systems in dynamic domains is a tedious task if it must be done by hand. In the above sense such a system would not be called intelligent. A more advanced solution is to build up and change the knowledge base using intelligent knowledge acquisition and knowledge base validation and verification tools. These tools can be seen as a part of the knowledge-based system, which would therefore become intelligent. Moreover, the more intelligent the behavior of the tool, the more the whole system could be considered intelligent, and the less effort is required from the user of the system. At the extreme, one would like to just tell the new knowledge to the system, or even better, the intelligent tool would know how and where to acquire new knowledge by itself. In general, this kind of systems could be called *learning apprentice systems* [Smith *et al.*, 1985], [Mitchell *et al.*, 1985], [Tecuci and Kodratoff, 1990].

More specific examples[1] are the applications in tools for

- acquiring expert systems knowledge [Bratko *et al.*, 1989],
- inductive engineering [Bratko *et al.*, 1989],
- planning and scheduling [Minton, 1993],
- knowledge elicitation and knowledge acquisition [Morik *et al.*, 1993],
- knowledge base validation and verification [De Raedt *et al.*, 1991],
- program testing [Bergadano *et al.*, 1993],
- program construction and verification [Bratko and Grobelnik, 1993],
- analyzing large databases [Van Laer *et al.*, 1994],

By investigating more applications of machine learning in other areas, the problem of integration has become more apparent. The integration is often not straightforward: machine learning tools may be too general (i.e., not domain-specific enough) to be applied in realistic applications. On the other hand, domain-specific approaches are often not easily transferable to other domains. Therefore there is a need for research on the integration aspect itself, to find the requirements and the properties of a good integration.

Example

In this context we introduce the example that is going to be used throughout the thesis.

Recently software tools for personal assistance also tend to integrate machine learning techniques, e.g., smart mailboxes or search engines [Etzioni, 1993b], [Indermaur, 1995]. A smart mailbox is an autonomous agent that manages the user's electronic communication. At first it observes how the user processes all types of electronic information, and after a while offers to create some standard rules for handling future incoming information. These applications are assisting the computer user in managing his or her computer system, and are therefore (still) relying on the user for confirmation of the rules they learn and of the decisions they have no evidence for yet. For these systems an interactive concept learning system such as CLINT [De Raedt, 1992] would be well-suited. On the other hand, in the future there will also be a need for tools that can work autonomously, i.e., non-interacting with a user, in case there is no person available that can answer these questions. These

[1]This list is only illustrative, and in no way to be interpreted exhaustively.

autonomous agents will have to collect the evidence for their knowledge by themselves. One could think of a tutoring program for a windowing system that can automatically adapt to newer versions of the windowing system, or of WWW-indexing-agents that search and index information found on the World Wide Web, and meanwhile find regularities in the huge amount of Web pages.

We will elaborate on the example of an autonomous tutor. Suppose a novice has to learn how to use a certain computer program. For this reason computer producers include tutorials, in the early days on paper, nowadays more and more as separate programs, called *tutors*. A tutor typically explains how to perform certain tasks in the domain, gives the novice some analogous tasks, evaluates the solutions, explains questions of the student, and so on. Incrementally the tutor *teaches* the novice how to use the program. More and more tutoring systems behave intelligently in their strategy to guide the learning process of the novice, by learning the behavior of the novice (see Intelligent Tutoring Systems [Sleeman and Brown, 1982] [Clancey, 1987]). Now the question we want to address is: *how did the tutor acquire the knowledge about the computer program?* Most probably, somebody who knows the program very well has programmed his or her knowledge into the tutor. But then, each time new versions of the program or similar programs are released, the tutor has to be adapted by humans. In the future intelligent tutors will probably have learning capabilities themselves.

To focus attention on a particular example, suppose we are using an operating system with a graphical user interface. Graphical user interfaces for operating systems in general tend to look alike, but then again have very different functionalities for, e.g., opening and closing documents by double-clicking, or moving them towards another folder and so on. Although there are emerging common features, differences remain. Moreover, in future versions of the graphical user interface some features may have been changed or added. This motivates the extension of this tutor with autonomous learning capabilities, so that it can adapt to future versions without having to be reprogrammed.

On the other hand, using a graphical user interface requires some *planning* capabilities. In order to open a document, one typically has to go to the folder (or directory) the document is in, and, for instance, double-click the icon of the document. To remove a document from a disk, again one has to find the icon of the document, and drag it into the trash. In order to recover a document from the trash, one has to open the trash, and select the "Put Back" command in one of the menus. These are planning problems the tutor must be able to solve, for if it asks the novice to perform a certain task in a particular situation, the tutor must be able to solve the task itself. Furthermore, it must be able to evaluate the novice's solution. If the novice, for instance, needed to remove a certain file, and had to open ten or more folders before finding the file, the tutor might advise the novice to first use the menu-command "Find File", which will help to search for the particular file. To conclude, this motivates the need for the tutor to have planning capabilities.

How would the tutor then automatically acquire its knowledge? Given a new (release of a) graphical user interface, the autonomous tutor can try to find out whether or not some features, necessary for its tutoring, have changed. It could for instance try to solve the tasks it would give to a novice in several situations itself. It could try and set up some experiments, and try to find out how the graphical user interface behaves in well-chosen situations. In short, the tutor gets to *play* with a graphical user interface, just as most humans familiar with similar environments would try out new programs. Of course this does not exclude any additional capability of the autonomous tutor to *be told* some things by humans. Neither does it exclude the possibility that the autonomous tutor would not be able to capture some of the new features and that it would report this.

To summarize: in this example the autonomous tutor basically has two states: a state in which it is tutoring a novice, and a state in which it is acquiring its own domain knowledge. It is on the learning process in the latter state that we will focus in our examples.

1.4 Overview of the thesis and Main contributions

In the previous sections, we have outlined the context of the learning problems we want to solve: we aim at an autonomous learning system that is suitable for an interactive integration with a problem solver. Therefore, we will need an *incremental* learning system, adapting its knowledge in between problem solving. Furthermore, we want to use the learning system on logical representations (in particular on first order logic language representations), because this facilitates the integration with problem solving. In the learning task, we will from the start adopt two major restrictions: we learn only one concept at a time, assuming the other concepts are completely known. Furthermore, we assume there are no errors in the examples from which the concepts have to be induced.

Although we are mainly interested in applying Inductive Logic Programming (i.e., learning first order logic representations), the first three chapters do not specifically concentrate on Inductive Logic Programming. Inductive Logic Programming is one possible instantiation of more general frameworks for concept learning (e.g., Versionspaces [Mitchell, 1982] and GenCol [De Raedt and Bruynooghe, 1992b]). Therefore Chapter 2 to Chapter 4 adopt a language independent point of view in describing solution spaces and general algorithms for concept learning.

In Chapter 2 we briefly introduce the major themes of concept learning to put the rest of the thesis in the right perspective. Some of these themes will not be discussed in the rest of the thesis. They will be touched upon when relevant, though. In Chapter 2 we also set up a basic formalization of concept learning based on the notion of *coverage*. We describe when it is sound to represent concepts and instances of concepts in a *concept representation language* $\mathcal{L}_C$, resp. an *instance representation language* $\mathcal{L}_I$. As such, we can reason about concepts and instances by means of their representation in $\mathcal{L}_C$, resp. $\mathcal{L}_I$.

In Chapter 3, the notion of coverage is used to define the notion of generality. This allows to structure $\mathcal{L}_C$, and to view concept learning as a search problem. In the framework of Versionspaces [Mitchell, 1982], which was extended by [Mellish, 1991], we describe the Iterative Versionspaces algorithm. The Iterative Versionspaces algorithm has an efficient worst case space and worst case time complexity. The algorithm implements an efficient check for maximality, and implements optimal search. The algorithm's bi-directionality allows to use the concept in a maximally specific or a maximally general way, and allows interaction with the user or the environment by generating relevant questions. Within the same framework, we also develop a theory to identify redundant information elements in concept learning, and to generate new information elements which make others redundant. We apply these extensions to the Iterative Versionspace algorithm. One of the major contributions of this chapter, is that all these achievements are *language independent*. Moreover, the ideas and representations are general and flexible enough to be applicable in other concept learning algorithms as well.

In Chapter 4 we allow concepts to be represented by disjunctions of concept representations of $\mathcal{L}_C$. This increases the expressivity of the language, but on the other hand also the computational complexity of the problem. We identify a sufficient condition for reducing the search for disjunctions to a search for individual disjuncts in $\mathcal{L}_C$. Based on this reduction, we develop a framework that describes *disjunctive versionspaces*. Because unrestricted disjunctive versionspaces contain too much undesired disjunctions, we enhance this framework by adopting additional preference criteria. Preference criteria express which solutions are most preferred, e.g., solutions with a minimal number of disjuncts. In a next step, we apply some of these criteria to disjunctive versionspaces to obtain the versionspaces of most preferred solutions. These ideas are implemented in two algorithms: a disjunctive extension of the Description Identification algorithm of [Mellish, 1991], and a disjunctive extension of the Iterative Versionspaces algorithm. As in Chapter 3 all these notions and the algorithms are formalized in a language independent way. The main contribution of this chapter is that it describes disjunctive versionspaces, with and without preference criteria, in terms of the elements of $\mathcal{L}_C$, and all this independently of the choice of $\mathcal{L}_C$.

In Chapter 5 we show how the frameworks developed in Chapter 3 and Chapter 4 can be applied to *Inductive Logic Programming*. On the one hand, this shows that the results obtained in Chapter 3 and Chapter 4 are applicable to particular concept representation languages. On the other hand, it formally shows how Inductive Logic Programming fits in the framework of Versionspaces and Iterative Versionspaces. As such, this chapter also contributes to the understanding of the search problems in Inductive Logic Programming. Moreover, the language independent notions and methods can be applied to ILP.

In Chapter 6 we present an autonomous agent architecture, in which Machine Learning and Planning are integrated. Planning is considered as typical AI problem solving. The agent's knowledge is represented in the *Event Calculus*, a formalism which allows to express temporal knowledge in Horn clause logic. Chapter 6 shows that the logical approach of the Event Calculus is suitable for integrating planning and learning in the autonomous agent: an existing planning technique that uses abduction is combined with Inductive Logic Programming as described in Chapter 5. The use of an *interactive* Inductive Logic Programming method allows the system to learn from experiments.

Chapter 7 concludes the thesis, and describes some directions for future research.

All algorithms described in this thesis, except the ones of Section 3.9, are currently implemented in ProLog by BIM.

A brief guide for the reader

The sections that contain rather technical material are marked [T] and printed in a slightly smaller font than the other text. Technical sections contain, for instance, algorithms, correctness proofs, complexity analyses. Each technical section begins with a paragraph that summarizes the most important results of the section. Having read the summary, the reader who is not interested in technical details, can skip the section.

Note that Page vi and following contain a list of figures, a list of algorithms, a list of definitions, and a list of symbols and algorithm notations.

Chapter 2

Concept Learning

2.1 Introduction

In Chapter 1 we have presented the context of the learning problems we want to consider. Essentially these problems are *concept learning problems*. The basic idea of concept learning is to induce general rules from specific examples.

In this chapter we describe and discuss the essence of concept learning. We also introduce the basic notions and assumptions needed for this description. The chapter is merely meant to put the following chapters in perspective. Starting from an example in Section 2.2, we identify the distinct components of the problem in Section 2.3. In Section 2.4 we build up a framework to describe the problem, which results in a formal problem description. In Section 2.5 we discuss several aspects of concept learning in more detail.

2.2 A Concept Learning Problem

Example 2.1 Let us first give an example of a concept learning problem, in the context of the autonomous tutor introduced in Section 1.3. The problem solving task of the tutor is to instruct novices how to use a graphical windowing system. However, in order to instruct humans, the tutor must first acquire knowledge of the functionalities of the graphical user interface itself. If the tutor is not to be (re)programmed it should be able to learn these functionalities autonomously. This is the learning task of the tutor we focus on in this example.

If the tutor is to act properly, it will need to have *an idea* of when it is able to perform particular actions in the graphical user interface, and when it is not, and what the effects of these actions are. Possible actions are, e.g., dragging a certain document from one folder to another, double clicking a document, clicking on the close-box of a window, and so on. Its *learning task* consists of forming itself an idea of successful actions and their effects *in general*, by reviewing *particular instances* of successful and failing dragging actions. The tutor might try to drag a particular document D from a folder F_1 to another folder F_2, in a particular situation where both folders are open, and it might succeed in doing so. It might try to drag a particular document D from a folder F_1 to another folder F_2, in a particular situation where F_2's *parent* is

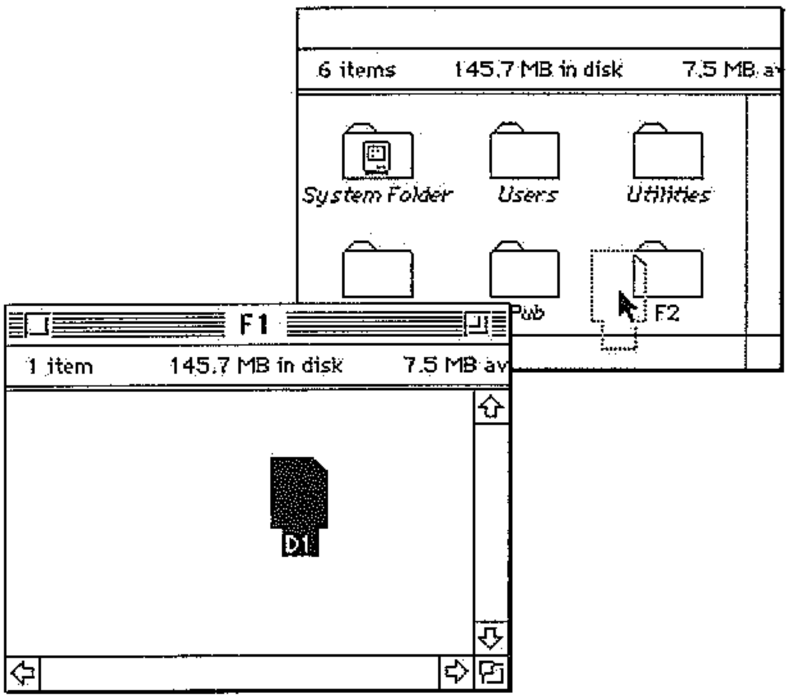

6 items 145.7 MB in disk 7.5 MB a
System Folder
Users
Utilities
Pub
F2
F1
1 item 145.7 MB in disk 7.5 MB av
D1

Figure 2.1 Successfully dragging D_1 from F_1 to F_2

open (see Figure 2.1), and also succeed in doing so[1]. It might try to drag a particular document D from a folder F_1 to another folder F_2, in a particular situation where F_2 is not visible, and fail in doing so[2]. It might try to drag documents of different types[3] from one folder to another, or documents that are opened and others that are closed, and each time observe the result. From these successes and failures of dragging documents, it might induce *the concept* of "dragging a document from one folder to another is successful", e.g.,

> dragging a document D_1 from one folder F_1 to another folder F_2 is successful, iff at the time of the action D_1 resides in F_1, F_1 is open, and F_2 is visible.

The concept is expressed by means of a condition on particular situations for being an instance of the concept. This condition is expressed as a relation between the objects that are part of that situation. The particular successful trials of dragging a document from one folder to another are *instances* belonging to that concept. The unsuccessful trials are instances which do not belong to the concept. Actually, the

[1]A document's *parent folder* is the folder the document resides in.
[2]We call a folder *visible* if it is open, or if its parent folder is open.
[3]Types of documents are, e.g., text files or executable files.

concept encloses ***all possible*** situations in which it is possible to drag the document from one folder to another, and *none* of the situations in which it is not possible. Once the tutor has learned the concept, it can use it to *predict* for future instances whether or not they will be successful.

The *effect* of a successful dragging action is that the document no longer resides in the first folder but now resides in the second folder. Similarly as the tutor must know the concept of a successful dragging action, it must know the concept of the effect of a successful dragging action. In this case, successfully dragging the document from the first folder to the second one, will *always* have the effect of the document residing in the second folder afterwards, and not residing in the first folder. From these successful trials the tutor's concept of "a successful action of dragging a document from one folder to another has the effect of the document residing in the second folder afterwards, and no longer residing in the first folder" might be

> the effect of successfully dragging a document D_1 from a folder F_1 to another folder F_2, is always that D_1 resides in F_2, and D_1 does not reside in F_1.

This concept encloses all instances of successfully dragging a document from one folder to another.

In general it is not said where the particular instances from which a concept learner learns originate. They might be observations made by the tutor itself. They might as well come from an instructor, "telling" the tutor that in this particular situation a drag action would fail, and in that particular situation a drag action would succeed, and have a particular effect. A concept learning program would just, *given* the instances and non-instances of the concept, try to find the common features and relationships between the instances, and generalize this into a general concept. This general concept can be used to predict for future actions whether or not they will be successful, and what their effect will be.

◇

We will now identify the components of a concept learning problem more generally.

2.3 A description of concept learning

The goal of concept learning is to find concept representations for particular concepts. Concepts are groups of instances. In the previous example these instances were *situations*. Other examples are for instance the concepts *bird*, *ostrich* and *penguin*. In talking about penguins the concept allows us to make abstraction of the complete set of penguins. The *concept representation* sums up features that are common to all instances that belong to the concept, and that are not common to the other instances. In other words, the concept representation presents conditions characterizing instances belonging to the concept. These conditions describe the instances in a way which is typical for the instances of the concept. The conditions can be expressed on several levels of abstraction, determined by the choice of a particular *concept language*. Birds might be described by the way they look (*they have feathers*, or *they have wings*), or by the way they behave (*they fly*, or *they lay eggs*). In order to describe birds by means of their behavior, one needs to possess knowledge about what flying and laying eggs is, i.e., the concepts *flying* and *laying eggs* must already be known. These representation building-blocks are actually in turn concepts, and could as well be described using more detailed concepts. To describe the concept *bird* in terms of those building-blocks, these building-blocks should have been learned in advance. Therefore knowledge about the concepts *flying* and *laying eggs* are said to belong to the *background knowledge* needed for learning the concept *bird*.

A representation of a concept can then be used to classify unseen instances as belonging to the concept or not. In this sense the concept representation could carry much more information than the original instances it was derived from. Classifying all animals that have feathers and wings as birds, allows to recognize animals one has never seen before as birds. In Example 2.1, it is impossible for the tutor to observe *every* situation in which the drag action is successful, yet it should find a general rule for this concept. Representing concepts by a description in a certain language (i.e., representing the concept *intensionally*), rather than as a set of all its instances (i.e., *extensionally*), is therefore a way of compacting information and knowledge.

In the next section we formalize this basic description of the concept learning problem.

2.4 A formalization of concept learning

We will now formalize the problem setting for learning a concept given some instances belonging to the concept and some instances not belonging to the concept. We will start from the set of *all* instances.

Notation 2.2 The set of all instances is denoted by $\mathcal{I}$.

In Example 2.1 an instance is, e.g., the situation in which document D_1 and folder F_2 both are in the open folder F_1. In the example of the birds, instances are particular birds. In the autonomous tutor example, instances are particular states of the windowing environment.

Definition 2.3 (Concepts) A concept is a set of instances.

Notation 2.4 The set of all concepts is denoted by $\mathcal{C}$.

By definition, $\mathcal{C} \subseteq \mathcal{P}(\mathcal{I})$ [4]. Concepts will be named: examples of concept names are *bird*, *flying*, and "dragging a document D from a folder F_1 to another folder F_2 is successful".

To reason and talk about instances and concepts, we represent them in a certain language. Actually, instead of reasoning about instances and concepts, we will be reasoning about *instance representations* and *concept representations*. Concept representations can in turn be built up from other concept representations, which are supposed to be in the *background knowledge*. We will now set up a framework in which it is sound to focus on representations only.

Notation 2.5 The language of instance representations is denoted by $\mathcal{L}_I$; the language of concept representations is represented by $\mathcal{L}_C$. The background knowledge is represented by $\mathcal{B}$.

[4] $\mathcal{P}(S)$ denotes the *powerset* of S, i.e., the set of all subsets of S.

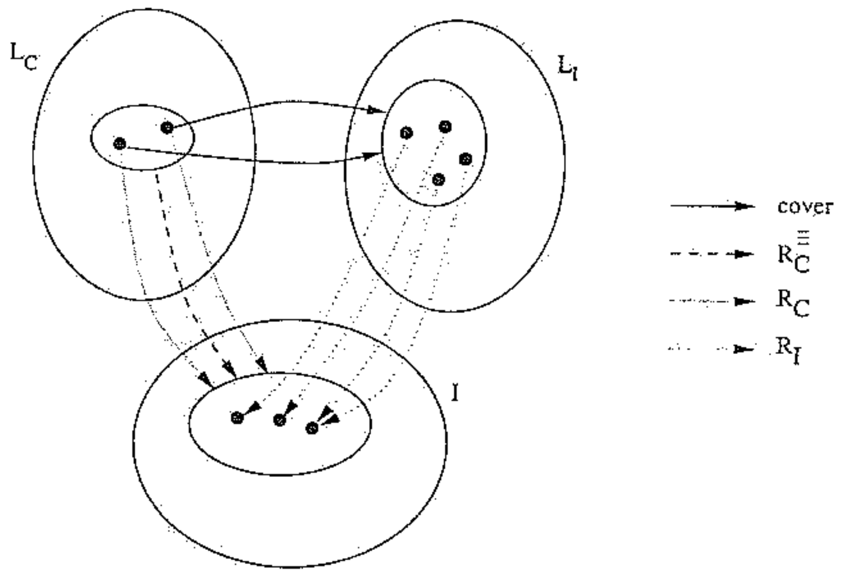


Figure 2.2 *cover*, R_I, R_C, and $R_C^{\equiv}$

Definition 2.6 (Representing instances and objects) The function $R_I : \mathcal{L}_I \to \mathcal{I}$ maps an instance representation to the instance it represents. Similarly the function $R_C : \mathcal{L}_C \to \mathcal{C}$ maps a concept representation to the concept it represents.

Requiring that R_I is a function means that no two distinct instances can have the same representation. Similarly, requiring that R_C is a function means that no two distinct concepts can have the same representation. Intuitively it is even acceptable to assume R_C and R_I are both *mappings*, i.e., R_C is defined for every element of $\mathcal{L}_C$, and R_I is defined for every element of $\mathcal{L}_I$. However, in general they are not necessarily injective (i.e., one concept can have several representations; see further), nor surjective (i.e., some concepts might have no representation).

Concepts are related to instances by the membership relation. This membership relation must be projected in a sound way to a relation between concept representations and instance representations. We will call this relation *covers*. Intuitively a concept representation c should cover an instance representation i, iff the instance represented by i is a member of the concept represented by c. Whether c covers i could depend on the concept representations c is built up with, and consequently on the background knowledge. Whenever we want to represent this dependency explicitly we will write $covers_{\mathcal{B}}$. However, most often we shall omit the index $\mathcal{B}$. We will define *covers* in terms of the related mapping *cover* : $\mathcal{L}_C \to \mathcal{P}(\mathcal{L}_I)$. The mapping *cover*(c) defines for each concept representation c the cover in $\mathcal{L}_I$, i.e., intuitively it defines, *on the representation side*, the set of instances that belong to the concept (see Figure 2.2). The mapping *cover* depends on the chosen languages $\mathcal{L}_I$ and $\mathcal{L}_C$, and can therefore not be defined here.

Definition 2.7 (*covers* **in terms of** *cover*) *covers* : $\mathcal{L}_C \times \mathcal{L}_I \rightarrow \{\, true,\ false \,\}$ is defined by: *covers*(c , i) iff $i \in cover(\ c\ )$.

Although *cover* is language dependent, we can specify a soundness constraint on it, which reflects our intuitive idea of its meaning.

Constraint 2.8 (Soundness and completeness of *cover*)

$$\forall i \in \mathcal{L}_I, c \in \mathcal{L}_C : i \in cover(\ c\ ) \text{ iff } R_I(\ i\ ) \in R_C(\ c\ ).$$

This constraint expresses that if a concept representation covers an instance representation, the corresponding instance must belong to the corresponding concept (soundness), and if an instance belongs to a concept then *every* representation of that instance must be covered by *every* representation of the concept (completeness). Under this constraint, it is safe to reason only about representations of instances and concepts instead of reasoning about the instances and concepts themselves.

2.4.1 Unique concept and instance representations [T]

SUMMARY: having multiple concept representations for one concept is inconvenient because there is no way to discriminate between them. Under Constraint 2.8 one can partition $\mathcal{L}_C$ into equivalence classes, based on the *cover*. Then we can represent concepts by equivalence classes, and assume there is only one representation for each concept. A similar approach applies for instance representations. In the remainder of the thesis we assume that each instance has at most one instance representation, and each concept has at most one concept representation. How this is accomplished in Inductive Logic Programming is discussed in Chapter 5.

It is often inconvenient to have several representations for one concept[5]. The main reason will become clear in Chapter 3, where $\mathcal{L}_C$ will be *searched* in a systematic way. Having multiple representations for one concept may lead to never-terminating processes, because there is no way to discriminate between them. In order to avoid multiple representations one can divide $\mathcal{L}_C$ into equivalence classes, such that all elements of one class represent the same concept. If one can identify exactly one element per equivalence class, this element can serve as a representant or a canonical form of the whole class, and therefore as a representation of the corresponding concept. We will now work out this idea, and show that, under Constraint 2.8, this is a sound approach.

Lemma 2.9 $\forall c_1, c_2 \in \mathcal{L}_C$:

$$R_C(\ c_1\ ) = R_C(\ c_2\ ) \text{ implies } cover(\ c_1\ ) = cover(\ c_2\ ).$$

Proof Take $i \in cover(\ c_1\ )$. Then $R_I(\ i\ ) \in R_C(\ c_1\ )$ by Constraint 2.8. Now if $R_C(\ c_1\ ) = R_C(\ c_2\ )$, we have that $R_I(\ i\ ) \in R_C(\ c_2\ )$. Consequently $i \in cover(\ c_2\ )$. □

Given this result, we can define an equivalence relation on $\mathcal{L}_C$.

Definition 2.10 ($\equiv$ on $\mathcal{L}_C$) $\equiv$: $\mathcal{L}_C \times \mathcal{L}_C \rightarrow \{\, true,\ false \,\}$ is defined as follows:

$$c_1 \equiv c_2 \text{ iff } R_C(\ c_1\ ) = R_C(\ c_2\ ).$$

[5]Distinct representations for one concept are called *syntactic variants*.

The relation $\equiv$ is an equivalence relation[6] on $\mathcal{L}_C$, because $=$ is an equivalence relation on $\mathcal{C}$. Let $\mathcal{L}_C/\equiv$ denote the set of all equivalence classes[7] of $\mathcal{L}_C$ w.r.t. $\equiv$, and $[c]_\equiv$ the equivalence class of $c \in \mathcal{L}_C$. Then $[c]_\equiv$ can be used to represent $R_C(\ c\ )$, with the mapping $R_C{}^\equiv : \mathcal{L}_C/\equiv \to \mathcal{C}$. (see Figure 2.2). $R_C{}^\equiv$ is well-defined, since every element of an equivalence class represents the same concept. Every concept has at most one representation in $\mathcal{L}_C/\equiv$, because there is at most one equivalence class whose elements represent (with R_C) a given concept. Now define $cover_\equiv : \mathcal{L}_C/\equiv \to \mathcal{P}(\mathcal{L}_I)$ as $cover_\equiv(\ [c]_\equiv\ ) = cover(\ c\ )$. $cover_\equiv$ is well-defined because each two elements of one class have the same *cover* (Lemma 2.9). By definition *cover* and $cover_\equiv$ then map representations of the same concept to the same set of instance representations. Consequently, we can represent concepts by elements of $\mathcal{L}_C/\equiv$, and assume that each concept has at most one representation in $\mathcal{L}_C$.

We can define a similar equivalence relation on $\mathcal{L}_I$.

Definition 2.11 ($\equiv'$ on $\mathcal{L}_I$) $\equiv'$: $\mathcal{L}_I \times \mathcal{L}_I \to \{\ true,\ false\ \}$ is defined as follows:

$$i_1 \equiv' i_2 \text{ iff } R_I(\ i_1\ ) = R_I(\ i_2\ ).$$

By following the same process as above the equivalence classes according to $\equiv'$ can be used to represent instances. To each instance then corresponds at most one representation.

For the remainder of the thesis we assume that each instance has at most one instance representation, and each concept has at most one concept representation. In general this is not a trivial assumption. In Inductive Logic Programming, for instance, determining equivalence is *in general* undecidable (see Chapter 5).

2.4.2 The single representation trick [T]

SUMMARY: in some application areas it is useful to consider concept representations covering only one instance. This allows instance representations to be represented by concepts, such that $\mathcal{L}_I \subset \mathcal{L}_C$. This is called *the single representation trick* [Cohen and Feigenbaum, 1981]. In general we will *not* assume the single representation trick.

We first introduce concept representations that cover only one instance representation.

Definition 2.12 (Unit concept representation) $c \in \mathcal{L}_C$ is a unit concept representation iff $\exists i \in \mathcal{L}_I : cover(\ c\ ) = \{\ i\ \}$.

Notation 2.13 Let $\mathcal{U}$ denote the set of all unit concept representations in $\mathcal{L}_C$.

Under Constraint 2.14 the single representation trick can be applied, because each instance corresponds to a unit concept.

Constraint 2.14 (Single cover constraint) For each instance representation $i \in \mathcal{L}_I$ there exists a unit concept representation $c \in \mathcal{L}_C$ such that $cover(\ c\ ) = \{\ i\ \}$.

Note that the assumption of having unique concept representations (see Section 2.4.1) implies that for a given i, there can be *at most one* unit concept representation c such that $cover(\ c\ ) = \{\ i\ \}$.

Under Constraint 2.14 $\mathcal{U}$ can be chosen as instance representation language instead of $\mathcal{L}_I$ itself. First let $SRT : \mathcal{L}_I \to \mathcal{U}$ be the mapping from an instance representation to the corresponding unit concept representation. SRT is a bijection: to every unit concept representation

[6]An equivalence relation is *reflexive*, *symmetric* and *transitive*.

[7]The equivalence class of $c \in \mathcal{L}_C$ is the set of all $c' \in \mathcal{L}_C$ such that $c \equiv c'$. Each element of $\mathcal{L}_C$ belongs to exactly one equivalence class. The set of all equivalence classes is denoted $\mathcal{L}_C/\equiv$.

corresponds an instance representation such that $cover(\ c\ ) = \{\ i\ \}$ (by Definition 2.12); to every instance representation corresponds a unit concept representation such that $cover(\ c\ ) = \{\ i\ \}$ (by Constraint 2.14). The mapping $R_I{}' : \mathcal{U} \rightarrow \mathcal{I}$, defined as $R_I{}' = R_I \circ SRT^{-1}$, maps each unit concept representation to the instance it represents. The mapping $cover' : \mathcal{L}_C \rightarrow \mathcal{P}(\mathcal{U})$, defined as

$$cover'(\ c\ ) = \{\ u \in \mathcal{U} \mid SRT^{-1}(\ u\ ) \in cover(\ c\ )\ \},$$

maps each concept representation to the unit concept representations it covers.

In general we will not assume the single representation trick. Where we do, we will mention it explicitly.

2.5 Problem specification

Given:

- a language $\mathcal{L}_I$ of instance representations;
- a language $\mathcal{L}_C$ of concept representations;
- background knowledge $\mathcal{B}$;
- a relation $covers_{\mathcal{B}} : \mathcal{L}_C \times \mathcal{L}_I \rightarrow \{\ true,\ false\ \}$;
- a set $P \subseteq \mathcal{L}_I$ of positive examples w.r.t. an unknown target concept t, and a set $N \subseteq \mathcal{L}_I$ of negative examples w.r.t. t; the elements of $P \cup N$ are called examples;

Find: $h \in \mathcal{L}_C$ (called a *hypothesis*) such that

- $\forall p \in P : covers_{\mathcal{B}}(\ h\ ,\ p\ ) = true$
- $\forall n \in N : covers_{\mathcal{B}}(\ h\ ,\ n\ ) = false$

The two conditions are called *the consistency requirement.*

Problem 2.1 The Concept Learning problem

Formally, the concept learning problem can now be formulated as in Problem 2.1. Given a language to represent instances and a language to represent concepts, given the background knowledge, and given a set of positive examples and a set of negative examples, the aim is to find a representation for an unknown target concept. In order to do so, we have to seek a hypothesis in the language of concept representations which covers all positive examples and which covers none of the negative examples. We will now highlight some aspects of Problem 2.1 in more detail.

Sufficient and perfect data

Problem 2.1 is an idealized problem in two respects: it assumes *sufficient data* and *perfect data.*

Problem 2.1 assumes sufficient data: in practice P and N will not always contain a sufficient number of elements to identify the target concept t. In case the number of examples is not sufficient, the resulting hypothesis h can be *any* concept representation fulfilling the consistency requirement. In general one could define a *success criterion* [Lavrač and Džeroski, 1994], and accept any element of $\mathcal{L}_C$ that fulfills this criterion as a solution. In this thesis we will always adopt the consistency requirement as a criterion for success. If more than one concept representation fulfills the success criterion, an additional *preference criterion* can specify which of them is preferred as a solution. In this thesis we will have to specify particular preference criteria for disjunctive versionspaces in Chapter 4.

Problem 2.1 assumes perfect data: in practice P and N could contain *noise*. This means that P might contain some examples that belong to N, or vice versa. It also means there is a certain degree of unreliability in the elements of P and N, so that we cannot require full consistency (if we would, we would not find t). In case noise is expected among the examples, the consistency requirement for h will have to be relaxed, in order to avoid *overfitting*. An alternative success criterion would for instance admit a certain percentage of elements of P for which $covers(\ h\ ,\ p\ ) = false$ and elements of N for which $covers(\ h\ ,\ n\ ) = true$, or would take into account *classification accuracy*, i.e., the accuracy in classifying unseen examples as positive or negative. In this thesis, we will not handle the problem of imperfect data (see [Hirsh, 1990],[Lavrač and Džeroski, 1994]). This thesis is merely meant as a theoretical basis for describing and searching sets of solutions for concept learning, and largely independent of the chosen success criterion.

Bias

The concept learning problem is often viewed as a search problem. Also in this thesis, we try to find h by means of search algorithms. The search is specified by the chosen *bias*. Bias determines which parts of the language will be searched, which parts will be pruned, what will be searched first, etc. A very strong bias will lead to a solution in an almost straightforward manner. On the contrary, a bias that is too weak will underconstrain the search problem and will become computationally unfeasible.

There are several ways to specify bias (see also Chapter 5 for a discussion in the context of Inductive Logic Programming). *Language bias* determines which part of the language will be searched. Most often language bias is chosen at the start and cannot change during the concept learning process. In these cases one can actually view the concept learning problem constrained by a language bias as a concept learning problem within the sublanguage defined by the bias. Extensions of this scheme have been proposed to dynamically adjust language bias, called *shift of bias* [Utgoff, 1986] (see Section 3.10). Therefore it is still useful to make a distinction between the concept representation language and a language bias. We will denote the choice of a particular language bias $*$ within the concept representation language $\mathcal{L}_C$ as $\mathcal{L}_C^*$. In the Inductive Logic Programming context of Chapter 5, for instance, $\mathcal{L}_C$ would be the set of all Horn Clauses that can be constructed with a given set of predicates, a given set of functors, and a given set of variables. A language bias $*$ on $\mathcal{L}_C$ could, e.g., restrict the search to clauses with 0, 1 or 2 existential variables only, or to clauses not containing a certain predicate p/n (see Chapter 5).

The way *how* the language is searched is called the *search bias*. Search bias includes for instance the choice of a search strategy, the choice of particular heuristics and pruning strategies. Search bias is also closely related to the preference criterion, because it might search the most preferred parts of $\mathcal{L}_C$ first. The preference criterion is sometimes called *preference bias*.

Incremental vs. batch

An important distinction must be made between *batch* and *incremental* concept learning problems. The former assume P and N are completely known in advance. The latter however do not assume P and N to be completely known, and therefore in fact basically implement an *updating* procedure. This procedure starts from a *current hypothesis*, which is a solution for the problem w.r.t. the known examples. Then it updates the current hypothesis each time the concept learning problem changes because more examples become available. Our starting point outlined in Section 1.1 is an inherent incremental one. It is very important to realize that at each moment the current hypothesis can be used as a working hypothesis for the problem solver. Whenever new examples, inconsistent with the current hypothesis become available, the latter is to be updated.

Interactive concept learning

Interactive concept learning systems automatically generate relevant questions, i.e., elements of $\mathcal{L}_I$ for which a classification as positive or negative example is relevant in their search process. This kind of feedback is provided by a critic, in this context called *an oracle*. In most cases the oracle is a human, but the answers to these questions could also be obtained by observation (see Chapter 6).

Multiple concept learning

So far we limited the discussion to learning a single concept, given a concept learning language. Elements of this language are based on other concepts, the representation of which is in the background knowledge. For our problems we assume the background knowledge to be correct. However, the concept representations in the background knowledge could have been learned as well. If this learning process is not completely finished, the fact that an example is not consistent with the current concept representation might have its cause in an inconsistent concept representation in the background knowledge. This suggests that the background knowledge should be updated, rather than the current hypothesis. Also the learning system first has to decide which concept representation is to be adapted. This decision could be made by incorporating an intelligent debugger. In [Shapiro, 1983] and [De Raedt, 1992] such an intelligent debugger relies on the presence of an oracle. In [De Raedt *et al.*, 1993] the decision is based on the notions of *local* and *global* inconsistency.

In general the problem of learning multiple concepts at the same time is solved by *theory revision systems* [De Raedt and Bruynooghe, 1992a], [Wrobel, 1993], [Adé *et al.*, 1994]. No preference is given to the concept representation of the inconsistent example w.r.t. the concept representations of the background knowledge: the whole set of all concept representations is considered as one theory. For each inconsistent example of any of the concepts in the theory, a theory revision system typically selects a concept representation that is to blame, and updates it. During the update, the other concept representations are considered to be correct. This means that from our point of view they are considered as the background knowledge at that moment. It is very well possible that the update leads to an inconsistency for another concept representation depending on the updated one. Consequently, the theory revision loop might start again from the beginning.

2.6 Conclusion

In this chapter we have introduced the concept learning problem, and briefly touched upon several important aspects. The basic notion of the introduced framework is the *cover* relation. In the next chapter we will extend the basic framework of this chapter by introducing ***versionspaces***. The framework of versionspaces is built on the structure of the search space induced by the *cover* relation. The extended framework therefore allows us to formally describe the search space of the concept learning problem, and to formulate algorithms that systematically search this space.

Chapter 3

Iterative Versionspaces

3.1 Introduction

In this chapter we will give a theoretical description of the concept learning problem and basic algorithms for solving it. On the one hand, the concept learning problem of Chapter 2 will be extended towards handling other types of information. On the other hand, the concept learning problem will be seen as a *search problem*. With the Candidate Elimination algorithm (CE) Mitchell did not only present an algorithm for concept learning, but, more important, introduced the theoretical framework of Versionspaces [Mitchell, 1978]. This framework has proven to be very useful in reasoning about the concept learning problem. The framework allows to represent Versionspaces by means of their boundaries. CE, and its successor, the Description Identification algorithm (DI) of [Mellish, 1991], represent the *versionspace* of all solutions by means of its maximally specific and its maximally general elements. They effectively compute these sets, basically in a bi-directional breadth-first way. They are therefore criticized, because the boundary representation is not always useful in practice: its size can be exponential in the number of information elements presented to the algorithm [Haussler, 1988].

Therefore [Hirsh, 1992b], among others, explicitly uses alternative representations, which are more efficient in language-specific contexts. We want to continue this line of research, but in a language-independent direction: within the framework of Versionspaces, we develop an alternative representation that turns out to be very efficient w.r.t. CE for its space complexity. We present the Iterative Versionspaces[1] algorithm (ITVS), based on this representation [Sablon *et al.*, 1994]. We prove the correctness and the completeness of this approach, and analyze its complexity. We also characterize redundancy in the input data and extend ITVS to discard redundant data. Finally, we show that for incremental use ITVS can generate relevant information elements.

One of the major contributions of this work is its independency from any specific concept or instance representation language. The general framework therefore has a wide application potential. In Chapter 5 we will for instance apply the framework in an Inductive Logic Programming context. The language independence character also makes the work fundamental as a study of the nature and the complexity of concept learning in general.

This chapter is structured as follows: first we reformulate the concept learning problem

[1][Cohen and Feigenbaum, 1981] also mentions an "iterative versionspace", which is however not directly related to our approach.

as a search problem, and extend it by introducing new forms of information (Section 3.2). Having reformulated and extended the problem, we then investigate how we can describe the *versionspace* of all solutions (Section 3.3). In Section 3.4 we introduce the setting for constructing this versionspace by searching $\mathcal{L}_C$ through the discussion of search operators and search strategies. Taking a bi-directional breadth-first strategy leads us to the Description Identification algorithm (Section 3.5). As opposed to this breadth-first approach, we present the depth-first approach of the Iterative Versionspaces algorithm in Section 3.6. Section 3.7 describes some examples of this algorithm, and Section 3.8 discusses its properties. The next three sections describe some useful extensions in the context of the Iterative Versionspaces framework. First Section 3.9 discusses the problem of avoiding storage of redundant information elements. Section 3.10 briefly touches upon the problem of shift of bias in relation to redundant information elements. Section 3.11 discusses the generation of relevant information elements in the context of a bi-directional search strategy. Finally, Section 3.12 concludes this chapter.

3.2 Reformulation of the Concept Learning problem

Mitchell was one of the first to formulate the Concept Learning problem as a search problem [Mitchell, 1978], [Mitchell, 1982]. The *search space* is the set of all possible concept representations of $\mathcal{L}_C$ that are allowed by the chosen language bias. We denote this by $\mathcal{L}_C^*$ (see Chapter 2). Most of the time we work with a fixed language bias within $\mathcal{L}_C$. Therefore we will omit the superscript $*$ when no confusion is possible.

To search $\mathcal{L}_C$ in an organized way, we have to structure $\mathcal{L}_C$. Because concept learning is concerned with finding a concept representation that covers a given set of positive examples and that does not cover a given set of negative examples (cf. Problem 2.1), it is natural to order $\mathcal{L}_C$ according to the instances covered. Therefore the $\subseteq$ relation on $\mathcal{P}(\mathcal{L}_I)$ is used to induce an order on $\mathcal{L}_C$. In Chapter 2 we introduced the notation $cover_{\mathcal{B}}(\ c\ )$ to denote the set of instances covered by the concept c. The subscript $\mathcal{B}$ points to the fact that the cover might depend on the background knowledge. However, when no confusion is possible about which background knowledge is meant, the index $\mathcal{B}$ from $cover_{\mathcal{B}}$ will often be omitted.

We will now relate concept representations by comparing their *cover*.

Definition 3.1 (More specific than) $\preccurlyeq_{\mathcal{B}} : \mathcal{L}_C \times \mathcal{L}_C \rightarrow \{\ true,\ false\ \}$ is defined by

$$\preccurlyeq_{\mathcal{B}}(\ c_1\ ,\ c_2\ ) \text{ iff } cover_{\mathcal{B}}(\ c_1\ ) \subseteq cover_{\mathcal{B}}(\ c_2\ ).$$

Notation 3.2 $\preccurlyeq_{\mathcal{B}}(\ c_1\ ,\ c_2\ )$ is denoted by $c_1 \preccurlyeq_{\mathcal{B}} c_2$ or $c_2 \succcurlyeq_{\mathcal{B}} c_1$. It reads as "$c_1$ is more specific than c_2", resp. "c_2 is more general than c_1". $c_1 \prec_{\mathcal{B}} c_2$, or $c_2 \succ_{\mathcal{B}} c_1$, denotes that $c_1 \preccurlyeq_{\mathcal{B}} c_2$ and $c_1 \neq c_2$, and reads as "c_1 is strictly more specific than c_2", resp. "c_2 is strictly more general than c_1". As for *cover* the index $\mathcal{B}$ will be omitted when no confusion is possible.

The relation $\preccurlyeq$ is illustrated in Figure 3.1.

Proposition 3.3 If the relation *covers* is sound, $\preccurlyeq$ is a partial order[2].

[2]A partial order is reflexive, anti-symmetric and transitive.

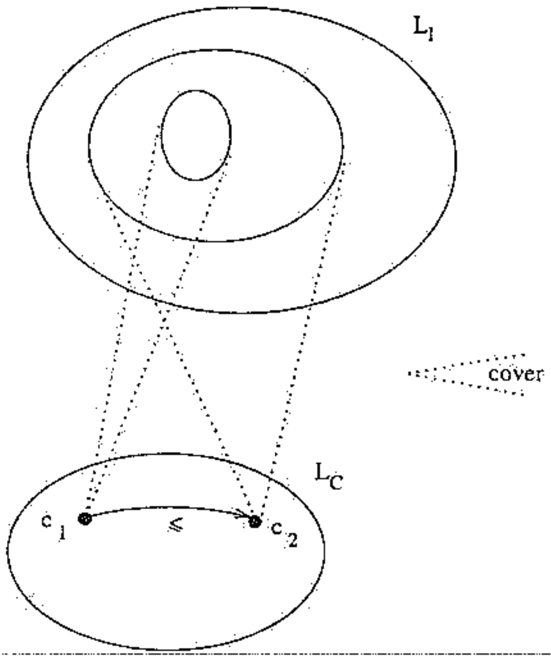


Figure 3.1 c_1 is more specific than c_2

Proof This follows from Lemma 2.9 and from $\subseteq$ being a partial order on $\mathcal{P}(\mathcal{L}_I)$. □

Note that, since it is not necessarily a total order[3], $c_1 \preccurlyeq c_2$ does not imply $c_2 \prec c_1$.

Example 3.4 Consider the following concept of Example 2.1, which we will call C.

> C:
> dragging a document D from a folder F_1 to another folder F_2 is successful,
> iff at the time of the action D resides in F_1,
> F_1 is open, and F_2 is visible.

Possible specializations of C are for instance:

- dragging a document D from a folder F_1 to another folder F_2 is successful,
  iff at the time of the action D resides in F_1,
  F_1 is open, F_2 is visible,
  and F_1 is unlocked.
- dragging a document D from a folder F_1 to another folder F_2 is successful,
  at the time of the action D resides in F_1,
  iff F_1 is open, F_2 is visible,
  and F_2 is unlocked.

[3]In a *total order* we have $c_1 \preccurlyeq c_2$ or $c_2 \preccurlyeq c_1$ for all $c_1, c_2 \in \mathcal{L}_C$.

- dragging a document D from a folder F_1 to another folder F_2 is successful, iff at the time of the action D resides in F_1,
  F_1 is open, F_2 is visible,
  F_1 is a high-priority folder, and F_2 is unlocked.

These are specializations because by adding more conditions, less instance representations are covered.

A possible generalization of C is for instance:

> dragging a document D from a folder F_1 to another folder F_2 is successful, iff at the time of the action D resides in F_1,
> and F_2 is visible.

This is a generalization because by dropping conditions, more instance representations will be covered. ◇

Further on we will discuss how to derive specializations and generalizations of a concept representation by means of *refinement operators* (Section 3.4.1). In Chapter 5 we will describe specialization and generalization in the context of Inductive Logic Programming.

Lemma 3.5 If the single representation trick holds, then

$$\forall c \in \mathcal{L}_C, i \in \mathcal{L}_I : covers(\ c\ ,\ i\ )\ \text{ iff }\ i \preccurlyeq c.$$

Proof This follows from the definition of *covers* and $\preccurlyeq$. □

If the single representation trick does not hold, then $i \preccurlyeq c$ is not defined for $i \in \mathcal{L}_I$ and $c \in \mathcal{L}_C$. Therefore we introduce the following notation.

Notation 3.6 $\forall i \in \mathcal{L}_I, c \in \mathcal{L}_C$: let $i \preccurlyeq c$ denote $covers(\ c\ ,\ i\ )$. Instead of saying that "c covers i", we can also say that "i is more specific than c".

Because of Lemma 3.5, in case the single representation trick does hold, this notation does not introduce conflicts .

Using this notation, Problem 2.1 can be reformulated, stating that all positive examples have to be more specific than a solution h, and no negative examples must be more specific than h.

The problem can be extended, in that there are not only elements of $\mathcal{L}_I$ which are related to the target concept, but also that there can be elements of $\mathcal{L}_C$ that are related to the target concept. Requiring that $c_1 \in \mathcal{L}_C$ is more specific than c_2, then means that *all instance representations* covered by c_1 must be covered by c_2 (see Figure 3.2.a; c_1 is called a positive lowerbound for c_2). Requiring that c_1 is not more specific than c_2, then means that *at least one instance representation* covered by c_1 is not covered by c_2 (see Figure 3.2.b; c_1 is called a negative lowerbound for c_2). Other relationships can be considered: there can be elements of $\mathcal{L}_C$ that are *more general* than the target concept or are not *more general* than the target concept (see [Mellish, 1991]). Requiring that c_1 is more general than c_2, then means that *all instance representations* covered by c_2 are covered by c_1 (see Figure 3.2.c; c_1 is called a positive upperbound for c_2). Requiring that c_1 is not more general than c_2, means that *at least one instance representation* covered by c_2 must not be covered by c_1 (see Figure 3.2.d; c_1 is called a negative upperbound for c_2). That the latter kind of information is meaningful, is illustrated in the next example.

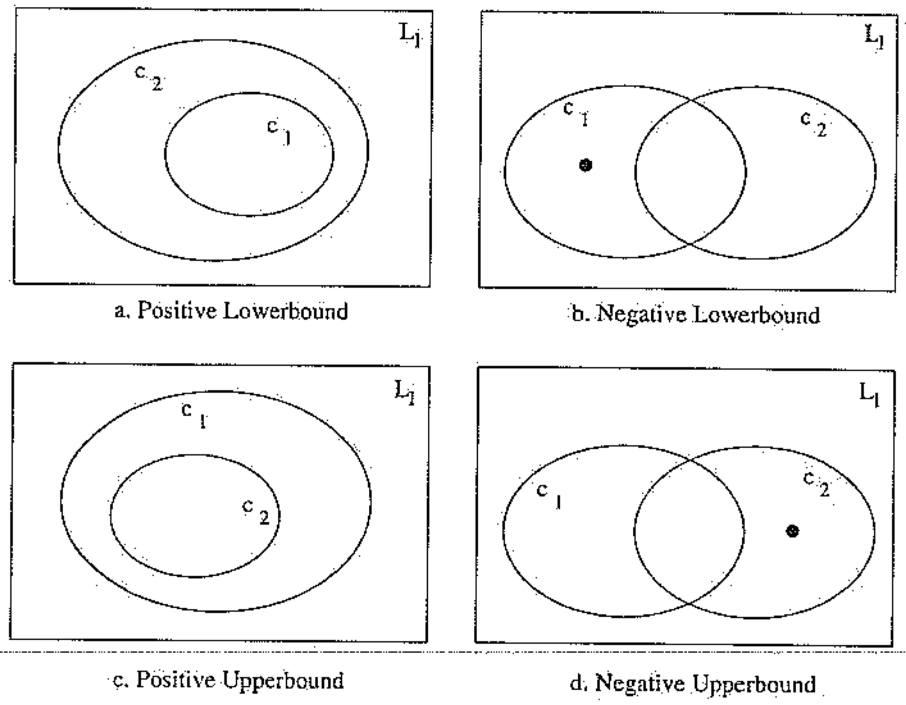


Figure 3.2 Information elements; c_1 is a bound for c_2

Example 3.7 In the example of Example 2.1 we assumed the tutor's only source of information was observation. There could be other information sources, e.g., a human teacher. A human could *tell* the tutor for instance that dragging a document to a folder will erase the document, if that folder is the trash. It might be important the tutor does not have to discover this condition by itself, because it might have erased some important documents before it discovers the exact condition. Instead of having to "reprogram" the tutor, taking into account the tutor's current hypothesis for a successful drag action, the human could tell the tutor that the concept representation

> successfully dragging a document D to folder T will erase D,
> iff T is the trash.

is an upperbound of the target concept, i.e., every possible hypothesis has to be a specialization of this concept. The concept representation

> successfully dragging a document D to folder T will erase D,
> iff T is the trash, and D is unlocked.

is for instance such a specialization. This illustrates that *positive upperbounds* can be used to avoid overly general concepts, because they introduce conditions that always have to be present in the concept representation.

Similarly overly specific concept representations can be avoided by giving *negative upperbounds* to the learning system. Consider for instance a very cautious tutor, applying the drag action only under the too restricted condition that the document to be dragged must be a text file. A human could tell the tutor that the target concept is not more specific than the concept representation

> dragging a document D from a folder F_1 to another folder F_2 is successful, iff at the time of the action D is a text file.

i.e., that the condition that D is a text file does not appear in the target concept. In this case the tutor's current hypothesis should be general enough as not to include this condition, and, e.g., also allow executable files to be dragged.

$\diamond$

We will now formally define these information elements. Using the extended notion of information elements, information elements can come from $\mathcal{L}_I$ as well as from $\mathcal{L}_C$.

Definition 3.8 (Lowerbounds and upperbounds) Given a concept representation $c \in \mathcal{L}_C$, an information element $i \in \mathcal{L}_I \cup \mathcal{L}_C$ is called a

- positive lowerbound for c iff $i \preccurlyeq c$;
- negative lowerbound for c iff $\neg(i \preccurlyeq c)$;
- positive upperbound for c iff $c \preccurlyeq i$;
- negative upperbound for c iff $\neg(c \preccurlyeq i)$.

Positive lowerbounds and negative upperbounds will be called *s-bounds* as they constrain the specificity of candidate hypotheses; negative lowerbounds and positive upperbounds will be called *g-bounds* as they constrain the generality of candidate hypotheses.

As will become clear throughout this Chapter and Chapter 4, the notion of an s-bound is in fact a generalization of the notion of a positive example; similarly, providing g-bounds to the learning system is a more general way of providing it with negative examples. As positive examples, s-bounds will be used to search specific-to-general (through generalization); as negative examples, g-bounds will be used to search general-to-specific (through specialization).

From the above properties of upper- and lowerbounds, other possibly useful types of information elements can be derived. They are shown in Figure 3.3.

1. *All instances* covered by c_1 are not covered by c_2 (i.e., c_1 and c_2 are disjoint; see Figure 3.3.a).

2. *At least one instance* covered by c_1 is covered by c_2 (i.e., c_1 and c_2 are not disjoint; see Figure 3.3.b).

3. *All instances* not covered by c_1 are covered by c_2 (i.e., c_1 and c_2 are complementary; see Figure 3.3.c).

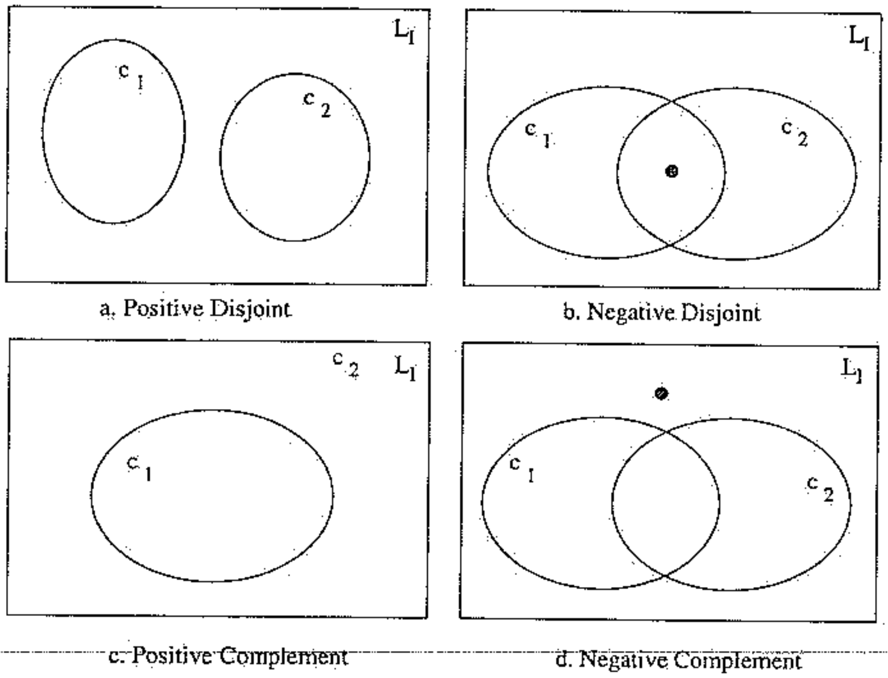


Figure 3.3 More information elements

4. *At least one instance* not covered by c_1 is not covered by c_2 (i.e., c_1 and c_2 are not complementary; see Figure 3.3.d).

Unlike the earlier kinds of information elements, the latter cannot be implemented with a $\preccurlyeq$ test between c_1 and c_2. However, if for each $c \in \mathcal{L}_C$ there exists $\overline{c} \in \mathcal{L}_C$ (the *negation* of c), such that $cover(\ \overline{c}\ ) = \mathcal{L}_I \setminus cover(\ c\ )$, then they can be implemented using $\preccurlyeq$ tests. We would then have the following four tests:

1. "c_1 and c_2 are disjoint" is expressed by $c_2 \preccurlyeq \overline{c_1}$.

2. "c_1 and c_2 are not disjoint" is expressed by $\neg(\ c_2 \preccurlyeq \overline{c_1}\ )$.

3. "c_1 and c_2 are complementary" is expressed by $\overline{c_1} \preccurlyeq c_2$.

4. "c_1 and c_2 are not complementary" is expressed by $\neg(\ \overline{c_1} \preccurlyeq c_2\ )$.

If $\mathcal{L}_C$ does not contain negations, these four types of information elements require tests on an instance-by-instance basis (in a Logic Programming context one would call this "extensionally"), which would be a costly operation.

Clearly these kind of information elements give information about the negation of a concept. Therefore, if we assume all negations are in $\mathcal{L}_C$, the negation of the target concept is also in $\mathcal{L}_C$, so we can use this information to learn the negation of the target concept. Disjoints and complements for the target concept can then be considered as a lowerbounds and upperbounds for the negation of the target concept. That this kind of information would be useful for the target concept as well (because of the *tertium non datur* principle $t \vee \bar{t}$) is shown in the following example.

Example 3.9 We will again take the example of the tutor. Suppose the concept of a successful action of dragging a document is to be learned.

- Negative disjoint: suppose the tutor is not able to observe all important properties. In that case, it would for instance only know, that it was able to drag a low-priority text file from one folder to another, but not whether or not these folders were open. On the one hand this will not allow to find the relevant conditions for a successful action (in this case the condition that the first folder must be open), but on the other hand it will allow to reject hypotheses such as

  C_1:
  dragging a document D from a folder F_1 to another folder F_2 is successful, iff at the time of the action D resides in F_1,
  F_2 is visible, D is a high-priority document, and D is an executable file.

  because this hypothesis has no instance in common with the partial observation. Consequently, a negative disjoint behaves as positive lowerbounds and negative upperbounds, in that it avoids overly specific hypotheses.

- Positive disjoint: this is the opposite case of a negative disjoint. The tutor observes some properties which hold at the time that a drag action does not succeed. Positive disjoints behave as negative lowerbounds and positive upperbounds, in that they avoid overly general hypotheses.

- Positive complement: when the tutor observes that dragging a document which is *not* an executable file is successful, it has another source for generalization. For instance, C_1 would have to be generalized in order to cover this (again only partially described) instance.

- Negative complement: this is the opposite case of a positive complement. The tutor observes some properties do *not* hold at the time a drag action does *not* succeed. Negative complements can be used for specialization.

◇

The example shows that these information elements also seem to be useful for handling incomplete observations. Nevertheless, we will not include them in the description of the extended concept learning problem, but assume they can be used to learn the negation of the target concept.Some Inductive Logic Programming systems can explicitly handle the representation of a certain concept together with a representation of the negation of that concept, by introducing a multi-valued logic (see e.g., CLINT [De Raedt, 1992], or MOBAL [Morik *et al.*, 1993]).

So far $\preccurlyeq$ is only defined between two concept representations ('...is more specific than ...') (see Definition 3.1) and between a concept representation and an information element ('... covers ...') (see Notation 3.6). As we have illustrated, the elements of $\mathcal{L}_C$ as well as elements of $\mathcal{L}_I$ can be used as information elements. For the sake of simplicity of some definitions (a.o., Definition 3.29 to Definition 3.32, and in Section 3.9) and of the formulation of Problem 3.1, we will trivially extend the definition of $\preccurlyeq$ towards instance representations mutually, and towards concept representations and instance representations.

Definition 3.10 (Extension of $\preccurlyeq$ to $\mathcal{L}_I \cup \mathcal{L}_C \times \mathcal{L}_I \cup \mathcal{L}_C$)

- For all $i_1, i_2 \in \mathcal{L}_I$: $i_1 \preccurlyeq i_2$ iff $i_1 = i_2$.
- For all $i \in \mathcal{L}_I$ and for all $c \in \mathcal{L}_C$: $c \preccurlyeq i$ iff $c = i$.

In case the single representation trick is applied and Constraint 2.14 holds, $\preccurlyeq$ was already defined, because $\mathcal{L}_I \subseteq \mathcal{L}_C$. Because of Constraint 2.14 Definition 3.10 is defined consistently.

The extended concept learning problem can now be formulated as in Problem 3.1. In

Given:

- a language $\mathcal{L}_C$ of concept representations ;
- a language $\mathcal{L}_I$ of instance representations;
- background knowledge $\mathcal{B}$;
- a relation $\preccurlyeq_{\mathcal{B}}\ : (\mathcal{L}_I \cup \mathcal{L}_C \times \mathcal{L}_I \cup \mathcal{L}_C) \rightarrow \{\ true,\ false\ \}$;
- four sets of *information elements*:

$$PLB, NLB, PUB, NUB \subseteq \mathcal{L}_I \cup \mathcal{L}_C,$$

  which are, respectively, positive lowerbounds, negative lowerbounds, positive upperbounds and negative upperbounds for an unknown *target concept representation* t

Find: an element $h \in \mathcal{L}_C$, if there exists one, such that

- $\forall i \in PLB : i \preccurlyeq_{\mathcal{B}} h$, i.e., i is a positive lowerbound for h;
- $\forall i \in NLB : \neg(\ i \preccurlyeq_{\mathcal{B}} h\ )$, i.e., i is a negative lowerbound for h;
- $\forall i \in PUB : h \preccurlyeq_{\mathcal{B}} i$, i.e., i is a positive upperbound for h;
- $\forall i \in NUB : \neg(\ h \preccurlyeq_{\mathcal{B}} i\ )$, i.e., i is a negative upperbound for h.

h is called a *hypothesis*.

Problem 3.1 The Extended Ideal Concept Learning problem

the rest of the thesis we will always shorten "i is a positive lowerbound for t" to "i is a positive lowerbound", and similarly for the other types of information elements.

Notation 3.11 For a given set I of information elements, we will denote the set of all s-bounds in I by I_s, and the set of all g-bounds in I by I_g. We also assume that each information element I is implicitly qualified as positive/negative lower-/upperbound.

In short, four sets of information elements w.r.t. an unknown target concept t are given. The idea is to identify t by searching $\mathcal{L}_C$ for an element c that is *consistent* with all information elements.

Definition 3.12 (Consistency) $c \in \mathcal{L}_C$ is consistent with

1. a positive lowerbound $i \in \mathcal{L}_I \cup \mathcal{L}_C$, iff $i \preccurlyeq c$;
2. a negative lowerbound $i \in \mathcal{L}_I \cup \mathcal{L}_C$, iff $\neg(i \preccurlyeq c)$;
3. a positive upperbound $i \in \mathcal{L}_I \cup \mathcal{L}_C$, iff $c \preccurlyeq i$;
4. a negative upperbound $i \in \mathcal{L}_I \cup \mathcal{L}_C$, iff $\neg(c \preccurlyeq i)$.

$c \in \mathcal{L}_C$ is consistent with $I \subseteq \mathcal{L}_I \cup \mathcal{L}_C$ if c is consistent with all elements of I. If c is not consistent with i, resp. I, we call c inconsistent with i, resp. I.

Notation 3.13 $c \in \mathcal{L}_C$ is consistent with $i \in \mathcal{L}_I \cup \mathcal{L}_C$ is denoted by $c \sim i$. $c \in \mathcal{L}_C$ is consistent with $I \subseteq \mathcal{L}_I \cup \mathcal{L}_C$ is denoted by $c \sim I$.

The following theorem describes how $\preccurlyeq$ is related to $\sim$.

Theorem 3.14
If i is a g-bound, and $x, y \in \mathcal{L}_C$ such that $x \preccurlyeq y$, then $y \sim i$ implies $x \sim i$.
If i is an s-bound, and $x, y \in \mathcal{L}_C$ such that $x \preccurlyeq y$, then $x \sim i$ implies $y \sim i$.

Proof This follows immediately from the definition of consistency and the transitivity of $\preccurlyeq$. □

By contraposition, this theorem will be used to prune the search for a consistent concept representation (see further): if i is a g-bound and $x \preccurlyeq y$, then $\neg(x \sim i)$ implies $\neg(y \sim i)$. Similarly if i is an s-bound and $x \preccurlyeq y$, then $\neg(y \sim i)$ implies $\neg(x \sim i)$.

3.3 Versionspaces

We will first investigate some characteristics of the search space and the set of solutions, before discussing search algorithms.

In concept learning the set of all consistent concept representations is often described by means of the set of its *maximally general* elements w.r.t. $\preccurlyeq$ and the set of its *maximally specific elements* w.r.t. $\preccurlyeq$.

Definition 3.15 (Maximal Generality and Maximal Specificness) $\forall S \subseteq \mathcal{L}_C$:

1. $m \in S$ is maximally general in S iff $\neg \exists s \in S : m \prec s$;
2. $m \in S$ is maximally specific in S iff $\neg \exists s \in S : s \prec m$;
3. $Max\ S = \{ m \in S \mid m$ is maximally general in $S \}$;
4. $Min\ S = \{ m \in S \mid m$ is maximally specific in $S \}$.

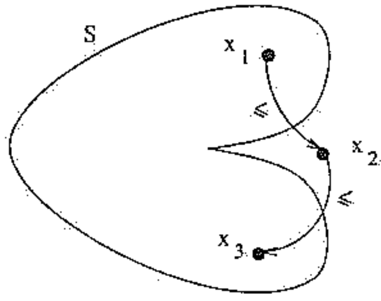


Figure 3.4 S is not convex

When no confusion is possible, we will shorten "maximally general (resp. specific) in S" to "maximally general (resp. specific)". Note that maximal specificity corresponds to minimality for $\preccurlyeq$ (see Notation 3.2).

Describing a set by means of its maximal and minimal elements is not possible for arbitrary sets. We will need to restrict ourselves to convex and bounded sets.

Definition 3.16 (Convex set) A set $C \subseteq \mathcal{L}_C$ is convex iff for all c_1, c_2, c_3 with $c_1, c_3 \in C$, $c_1 \preccurlyeq c_2 \preccurlyeq c_3$ implies that $c_2 \in C$.

The set S in Figure 3.4 is not convex because $x_1 \preccurlyeq x_2 \preccurlyeq x_3$, where x_1 and x_3 are in S and x_2 is not. However, given a set I of information elements, the set of all concept representations consistent with I must be convex because of Theorem 3.14: if $c_1 \preccurlyeq c_2 \preccurlyeq c_3$ and c_1 and c_3 are consistent with I, then $c_1 \preccurlyeq c_2$ implies that c_2 is consistent with all s-bounds, and $c_2 \preccurlyeq c_3$ implies that c_2 is consistent with all g-bounds.

Definition 3.17 (Bounded set) A set $C \subseteq \mathcal{L}_C$ is bounded iff for all $c \in C$ there exists a g maximally general in C and an s maximally specific in C such that $s \preccurlyeq c \preccurlyeq g$.

If there exists in $C, \preccurlyeq$ an infinite chain which is not closed above or not closed below, then C is not bounded. Figure 3.5 shows an infinite chain $C = \{c_1, c_2, c_3, \ldots\}$ of concept representations, such that $c_1 \preccurlyeq c_2 \preccurlyeq c_3 \preccurlyeq \ldots$ which is not closed above: C is convex but $c \notin C$, so there is no element in C which is maximally general.

[Hirsh, 1990] proves that bounded convex sets can always be represented by means of their maximal and minimal elements. Consequently all bounded infinite chains are closed above and closed below. Furthermore, if we want to describe the set of all consistent concept representations by means of its maximally general and maximally specific elements we have to impose Constraint 3.18 on $\mathcal{L}_C$.

Constraint 3.18 (Boundedness Constraint) For every possible set of information elements I, the set of concept representations consistent with I is bounded.

The admissibility constraint of [Mitchell, 1978] is stronger in the sense that it requires *every* subset of $\mathcal{L}_C$ to be bounded. [Hirsh, 1990] argues that this requirement can be

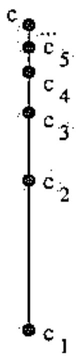


Figure 3.5 C is an infinite chain not closed above

weakened: only those subsets to be described by the learning system have to be bounded. This is also the requirement used in [Mellish, 1991].

On the other hand for algorithms that compute the sets of maximally general and maximally specific elements completely, these sets need to be finite.

Constraint 3.19 (Finiteness Constraint) For every possible set of information elements I, the set of maximally general, resp. maximally specific, concept representations consistent with I is finite.

[Mellish, 1991] discusses several approaches to fulfill this constraint. One could restrict the types of information elements in I, or, in some cases, choose a particular order of providing the information elements of I. Or one could restrict $\mathcal{L}_C$ and $\mathcal{L}_I$ in some way. If $\mathcal{L}_C$ were enumerable, the latter approach could lead to a kind of *Iterative Broadening* [Ginsberg and Harvey, 1992] in the construction of the sets of maximally general and maximally specific concept representations consistent with I. Iterative broadening would introduce a breadth cutoff[4]. As long as no solution is found with the given cutoff, the value of the cutoff could be increased, thus allowing more and more parts of the search space to be included. In this case, if there is no solution, termination cannot be guaranteed. In CLINT this approach leads to a series of finite languages with an increasing expressiveness [De Raedt, 1992]. To guarantee termination CLINT uses a finite series of languages.

For the rest of the thesis we will assume that Constraint 3.18 and Constraint 3.19 are fulfilled.

We will also assume the existence of a *top* element $\top$ and a *bottom* element $\bot$ in $\mathcal{L}_C$.

Constraint 3.20 (Top and bottom)

$$\exists \top, \bot \in \mathcal{L}_C : \forall c \in \mathcal{L}_C : \bot \preccurlyeq c \preccurlyeq \top.$$

Example 3.21 In the tutor example of Example 2.1 the maximally general concept of "dragging a document D from folder F_1 to another folder F_2 is successful" would be

[4]By analogy to the more widely known *iterative deepening* which is a depth-first search with a depth cutoff [Korf, 1985].

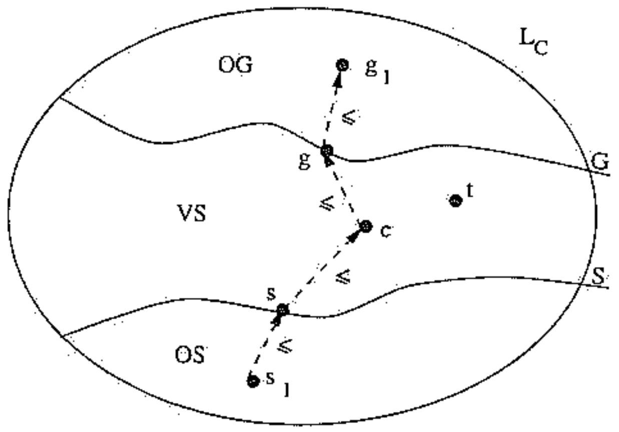


Figure 3.6 Overly general, overly specific and consistent parts of $\mathcal{L}_C$

dragging document D from folder F_1 to another folder F_2 is *always* successful.

The maximally specific concept would be:

dragging document D from folder F_1 to another folder F_2 is *never* successful.

◇

Now that we have determined the conditions for describing sets by means of their boundary sets in general, we will investigate how we can describe the set of all consistent concept representations by means of its boundary sets.

Using $\preccurlyeq$, $\mathcal{L}_C$ can be divided in three parts (see Figure 3.6) : the first part (OG) containing the concept representations inconsistent with some g-bounds, the second part (OS) containing those inconsistent with some s-bounds, and the third part $(\mathcal{VS})$ those consistent with all g-bounds and all s-bounds. Clearly, the third part is disjoint from the other two, and must contain the target concept, if the latter is in $\mathcal{L}_C$. Part OG contains all concept representations that are *overly general*; part OS contains those that are *overly specific*. The interesting part is the *border of the third part* $\mathcal{VS}$. Any concept representation g_1 strictly more general than an element on the border of $\mathcal{VS}$ at OG's side (called $\mathcal{G}$) is overly general; any concept representation s_1 strictly more specific than an element on the border of $\mathcal{VS}$ at OS's side (called $\mathcal{S}$) is overly specific. Only those concept representations c_1 more specific than an element of $\mathcal{G}$ and more general than an element of $\mathcal{S}$, are consistent.

We will now formalize these notions, which were introduced by [Mitchell, 1978]:

Definition 3.22 ($\mathcal{G}$ and $\mathcal{S}$) For a given set I of information elements,

- $\mathcal{S}_I = Min \{ s \in \mathcal{L}_C \mid s \sim I \}$, and
- $\mathcal{G}_I = Max \{ g \in \mathcal{L}_C \mid g \sim I \}$.

S_I determines a "lowerbound" on the set of all concept representations consistent with I. Analogously $\mathcal{G}_I$ is an "upperbound". Together, S_I and $\mathcal{G}_I$ determine a versionspace $\mathcal{VS}_I$.

Definition 3.23 (Versionspace) $\mathcal{VS}_I = \{\ c \in \mathcal{L}_C \mid \exists g \in \mathcal{G}_I, \exists s \in S_I : s \preccurlyeq c \preccurlyeq g\ \}$.

When no confusion is possible, the index I will be omitted from $\mathcal{VS}_I$, S_I and $\mathcal{G}_I$.

Theorem 3.24 (Adapted from [Mitchell, 1978]) $\mathcal{VS}_I$ is the set of *all* concept representations consistent with I.

Proof ($\subseteq$) Take $c \in \mathcal{VS}_I$. Then there exist $g \in \mathcal{G}_I$ and $s \in S_I$ such that $s \preccurlyeq c \preccurlyeq g$. Since $s \in S_I$ is consistent with all s-bounds, and c is more general than s, c is consistent with all s-bounds (Theorem 3.14); similarly c is more specific than $g \in \mathcal{G}_I$, and therefore consistent with all g-bounds.

($\supseteq$) Suppose c is consistent with I. Take s a maximally specific element of $C = \{\ x \in \mathcal{L}_C \mid x \preccurlyeq c \text{ and } x \sim I\ \}$. Such s exists, because C is not empty (c belongs to it) and because of Constraint 3.18 (the Boundedness Constraint). s is maximally specific in C and therefore also an element of S_I. A similar argument holds for choosing an element of $\mathcal{G}_I$. □

If the target concept representation t is in $\mathcal{L}_C$, we can *correctly* classify some elements of $\mathcal{L}_I \cup \mathcal{L}_C$ as positive or negative lower- or upperbounds, even if we have not yet fully identified t. The idea is that if all consistent concept representations of $\mathcal{L}_C$ classify a certain instance representation in the same way, then the target concept representation, whichever it is, must also classify it that way.

Theorem 3.25 (Adapted from [Mitchell, 1978]) An element $i \in \mathcal{L}_I \cup \mathcal{L}_C$ is a

1. positive lowerbound if $\forall s \in S : i \preccurlyeq s$;
2. negative lowerbound if $\forall g \in \mathcal{G} : \neg(\ i \preccurlyeq g\ )$;
3. positive upperbound if $\forall g \in \mathcal{G} : g \preccurlyeq i$;
4. negative upperbound if $\forall s \in S : \neg(\ s \preccurlyeq i\ )$.

Proof This follows immediately from Theorem 3.14. □

It should be noted that in practice this classification method is not used by the problem solver for solving problems. As described in Chapter 2 the current hypothesis (i.e., a maximally preferred element of $\mathcal{VS}$) is used for that purpose.

Apart from correctly classifying elements of $\mathcal{L}_I \cup \mathcal{L}_C$, [Hirsh, 1992b] also identifies other useful operations on versionspaces, some of which are already discussed above: checking whether a versionspace is empty, checking whether a versionspace has converged to a singleton, updating a versionspace with a new information element, checking whether a concept representation belongs to a versionspace, checking whether one versionspace is a subset of another versionspace, making the union of two versionspaces, and making the intersection of two versionspaces. The last operation gave rise to the *Incremental Versionspace Merging* algorithm [Hirsh, 1990]. All but the last two operations were already described by [Mitchell, 1978]. [Hirsh, 1992b] calls the versionspace representation *epistemologically adequate* for a set of operations and a class of concept representation languages if all the operations can be implemented using the representation.

3.4 Searching the Search Space

3.4.1 Search Operators

Now that we have defined a structure on $\mathcal{L}_C$ by means of $\preccurlyeq$, we can define search operators on $\mathcal{L}_C$. We employ two kinds of search operators: *specialization operators* (also called downward refinement operators) and *generalization operators* (also called upward refinement operators). The term *refinement operator* refers to specialization operators as well as generalization operators.

Definition 3.26 (Specialization operator)
$\sigma : \mathcal{L}_C \rightarrow 2^{\mathcal{L}_C}$ is a specialization operator iff $\forall d \in \sigma(\ c\ ) : d \preccurlyeq c$.
The elements of $\sigma(\ c\ )$ are called *direct specializations* of c w.r.t. σ.

Definition 3.27 (Generalization operator)
$\gamma : \mathcal{L}_C \rightarrow 2^{\mathcal{L}_C}$ is a generalization operator iff $\forall d \in \gamma(\ c\ ) : d \succcurlyeq c$.
The elements of $\gamma(\ c\ )$ are called *direct generalizations* of c w.r.t. γ.

Usually refinement operators are defined on $\mathcal{L}_C$ only, and not necessarily on $\mathcal{L}_I$, except when the single representation trick applies. Because we have extended the definition of $\preccurlyeq$ towards instance representations mutually in Definition 3.10, the refinement operators we will introduce in Definition 3.29 to Definition 3.32 can be defined on $\mathcal{L}_I \cup \mathcal{L}_C$. This will be especially useful for a uniform treatment of instance representations and concept representations when automatically generating new information elements in Section 3.9.

Example 3.28 In Example 2.1, given a certain concept representation, a specialization operator could return the set of all possible specializations obtained by adding one extra condition. Similarly, a generalization operator could return all possible generalizations of a given concept representation obtained by dropping a condition.

In Chapter 5 we will describe particular operators for Inductive Logic Programming.

$\diamond$

With a specialization operator one can search $\mathcal{L}_C$ in a *general-to-specific* way: starting from $\{\ \top\ \}$ as initial queue, iteratively one or more elements of the queue are selected to be specialized (depending on the search strategy - see further). Then the selected element(s) are replaced by their direct specializations, and the queue is pruned. This process continues until an element of the queue is a solution, i.e., is consistent with all information elements. The pruning step typically removes those elements of the queue that are not maximally general in the queue or inconsistent with some given s-bounds (using the contraposition of Theorem 3.14). Dually, using a generalization operator, one can search $\mathcal{L}_C$ in a *specific-to-general* way.

In an *incremental* learning system, information elements are provided to the learning system one by one. In this case the queue is updated as above (through generalization or specialization) for each information element i separately. In a general-to-specific search, and in case i is a negative lowerbound, for instance, the candidate hypothesis selected from the queue will have to be specialized so that it does not cover i. Dually, in a specific-to-general search, and in case i is a positive lowerbound, for instance, the selected candidate hypothesis will have to be generalized so that it covers i. This gives rise to the four following operations.

Definition 3.29 (Minimal Upperbounds) For $c_1, c_2 \in \mathcal{L}_I \cup \mathcal{L}_C$, the set of minimal upperbounds of c_1 and c_2 is

$$mub(\ c_1\ ,\ c_2\ ) = Min\ \{\ c \in \mathcal{L}_C \mid c_1 \preccurlyeq c \text{ and } c_2 \preccurlyeq c\ \}.$$

Definition 3.30 (Maximal Lowerbounds) For $c_1, c_2 \in \mathcal{L}_I \cup \mathcal{L}_C$, the set of maximal lowerbounds of c_1 and c_2 is

$$mlb(\ c_1\ ,\ c_2\ ) = Max\ \{\ c \in \mathcal{L}_C \mid c \preccurlyeq c_1 \text{ and } c \preccurlyeq c_2\ \}.$$

Definition 3.31 (Most Specific Generalizations) For $c_1, c_2 \in \mathcal{L}_I \cup \mathcal{L}_C$, the set of most specific generalizations of c_1 not covered by c_2 is

$$msg(\ c_1\ ,\ c_2\ ) = Min\ \{\ c \in \mathcal{L}_C \mid c_1 \preccurlyeq c \text{ and } \neg(\ c \preccurlyeq c_2\ )\ \}.$$

Definition 3.32 (Most General Specializations) For $c_1, c_2 \in \mathcal{L}_I \cup \mathcal{L}_C$, the set of most general specializations of c_1 not covering c_2 is

$$mgs(\ c_1\ ,\ c_2\ ) = Max\ \{\ c \in \mathcal{L}_C \mid c \preccurlyeq c_1 \text{ and } \neg(\ c_2 \preccurlyeq c\ )\ \}.$$

These operations refine a given information element c_1 *relative to another given information element* c_2, and therefore depend on the information elements presented to the concept learning algorithm. Given c_2 we can consider the mappings $c \mapsto mgs(\ c\ ,\ c_2\ )$, $c \mapsto msg(\ c\ ,\ c_2\ )$, $c \mapsto mub(\ c\ ,\ c_2\ )$ and $c \mapsto mlb(\ c\ ,\ c_2\ )$ as refinement operators. Therefore we will often also call mgs, msg, mub and mlb refinement operators.

The result of searching $\mathcal{L}_C$ can only be guaranteed to be successful if the search operators used are of a certain quality. In the rest of the thesis we will always work with refinement operators that are *complete*[5].

Definition 3.33 (Completeness of a search operator)

- A specialization operator ρ is complete, iff for all $c_1, c_2 \in \mathcal{L}_C$, $c_1 \preccurlyeq c_2$ implies that $c_1 \in \rho^{tc}(\ c_2\ )$[6].
- A generalization operator ρ is complete, iff for all $c_1, c_2 \in \mathcal{L}_C$, $c_1 \preccurlyeq c_2$ implies that $c_2 \in \rho^{tc}(\ c_1\ )$.

[5]Sometimes this kind of completeness is called *local* completeness.

[6]ρ^{tc} denotes the transitive closure of ρ.

Note that *mub*, *mlb*, *msg* and *mgs* are complete: they return *all* concept representations in $\mathcal{L}_C$ fulfilling the specified condition. The completeness of the search algorithms of the following sections will be based on the completeness of the operators.

We also introduce the following definition for refinement operators. It will be used in Chapter 5 when introducing refinement operators for Inductive Logic Programming.

Definition 3.34 (Locally finite) A refinement operator ρ is locally finite iff for all $c \in \mathcal{L}_C$, $\rho(\ c\ )$ is finite.

As noted, refinement operators could be defined on $\mathcal{L}_I \cup \mathcal{L}_C$ instead of $\mathcal{L}_C$. In that case Definition 3.33 and Definition 3.34 ought to be extended towards $c_1 \in \mathcal{L}_I \cup \mathcal{L}_C$ as well.

3.4.2 Search Strategies

Concerning search strategies, two aspects can be distinguished when developing concept learning algorithms. On the one hand one can choose the *direction* of the search: searching general-to-specific, searching specific-to-general or bi-directionally. On the other hand within a chosen direction, one can still apply several strategies, e.g., depth-first [Mitchell, 1982], [Sablon *et al.*, 1994], breadth-first [Mitchell, 1982], beam search [Michalski, 1983] or other heuristic search strategies.

In incremental concept learning the goal is usually to have one current consistent concept representation. This concept representation is used during problem solving to determine whether certain instance representations belong to the concept or not. The choice of a search strategy is in the first place determined by the way the resulting concept representation is going to be used. The more specific the resulting concept representation is, the less instance representations it will cover. This means that few errors in classifying non-instances as a member will be made, (also called *errors of commission* [Bundy *et al.*, 1985]), but probably also many errors in classifying instances as non-members (also called *errors of omission* [Bundy *et al.*, 1985]). Dually, the more general the resulting concept representation is, the more errors of commission and the less errors of omission will be made. In order to reduce the number of errors, *maximally specific* or *maximally general* concept representations are preferred.

Preferring specific concept representations will lead to using a specific-to-general strategy; preferring general concept representations will lead to a general-to-specific strategy. However, there are several advantages in having a *bi-directional* strategy. On the one hand, this allows to make a choice between using a general concept representation g or a specific concept representation s dynamically, i.e., at the time the concept representation is going to be used, instead of at the time the concept learning is initiated. A bi-directional strategy can also allow to use some middle strategy, by using a concept representation in between s and g. On the other hand, a bi-directional approach allows to generate new *relevant* upper- and lowerbounds automatically. A lowerbound i more specific than g, but not more specific than s is relevant, because if its truthvalue (positive or negative) is known, either g or s are inconsistent with i, and should be updated. Dually an upperbound i more general than s but not more general than g is also relevant. For more details on generating relevant upper- and lowerbounds we refer to Section 3.11.

Determining whether a concept representation g is *maximally* general requires in principle that g is compared to all other candidates for maximal generality. Candidates can be pruned using Theorem 3.14, leaving g to be compared only to all elements of $\mathcal{G}$. This means that all maximally general concept representations have to be stored, which would suggest some kind of breadth-first search, or else recomputed. Dually, testing for maximal specificity of a concept representation s would in principle require comparison to all elements of $\mathcal{S}$. Searching breadth-first bi-directionally is done in the *Description Identification algorithm* ([Mellish, 1991]). We will present the Description Identification algorithm in Section 3.5.

As opposed to this breadth-first strategy, we propose an alternative representation for $\mathcal{S}$ and $\mathcal{G}$ in the ITVS framework ([Sablon *et. al.*, 1994]). ITVS will be presented in Section 3.6.3. This representation allows to search *depth-first bidirectionally* and to identify a maximally general and maximally specific concept representations without having to compute or store $\mathcal{S}$ and $\mathcal{G}$.

3.5 The Description Identification algorithm

In this section we describe and discuss the Description Identification algorithm (DI) [Mellish, 1991].

3.5.1 The algorithm [T]

SUMMARY: we first describe the algorithm.

The Description Identification algorithm (DI) ([Mellish, 1991]; see Algorithm 3.1 and Algorithm 3.2) is an extension of the Candidate Elimination algorithm (CE) in the sense that it also accepts upperbounds apart from lowerbounds. There is one parameter to DI: the stream of information elements $\mathcal{Inf}$. The stream of information elements is a sequence of information elements, with a pointer to the current element in the stream. A read operation reads the current element in the stream, and sets the pointer to the next element in the stream. If the pointer points to the element *eos* (end of stream), a read operation on the stream will fail.

Algorithm 3.1 is incremental: for each information element i read from $\mathcal{Inf}$, it updates the sets G (which represents $\mathcal{G}$) and S (which represents $\mathcal{S}$). Since $\mathcal{G}$ as well as $\mathcal{S}$ are computed, a bi-directional breadth-first search is used. Initially G only contains $\top$, and S only contains $\bot$. If $\mathcal{L}_C$ is known to contain the target concept representation, the search procedure may stop whenever $G = S = \{ c \}$ (i.e., when $\mathcal{G}$ and $\mathcal{S}$ have *converged* to a singleton). Then c must be the target concept. In that case it is also not necessary to test for a collapse of either $\mathcal{S}$ or $\mathcal{G}$ (see further).

We will first discuss the case of the information element read (i) being an s-bound. First G is pruned (see Step 3.1): if an element of G is not consistent with i, then none of its specializations can be consistent with i (because of the contraposition of Theorem 3.14). Whenever G is empty, the search has *collapsed*, meaning that the concept representation cannot be in $\mathcal{L}_C$. In that case the algorithm fails and stops (see Step 3.2). Otherwise all elements of S are generalized if necessary (see Algorithm 3.2). The generalizations are gathered in S'. If i is a positive lowerbound and s an element of S, all minimal upperbounds (*mub*) of s and i have to be added to S' (see Step 3.9). In case s is consistent with i, the only minimal upperbound is s itself. If i is a negative upperbound, all most specific generalizations (msg) of s and i are added to S' (see Step 3.10). Not all elements in S' are necessarily consistent with all g-bounds, or are necessarily maximally specific, however. Therefore, after the result of *generalize* has been assigned to S (see Step 3.3), only those elements

```
procedure DI($\mathcal{I}nf$: stream of info ) returns set of concept, set of concept
  $S := \{ \perp \}$; $G := \{ \top \}$
  while $\mathcal{I}nf$ is not empty
     do $i := read( \mathcal{I}nf )$
       if $i$ is an $s$-bound
       then G := select all $g$ from $G$ with $g \sim i$  {3.1}
            if $G = \emptyset$
            then fail  {3.2}
            $S := generalize\_all( S , i )$  {3.3}
            $S$ := select all $s$ from $S$
                 with $\exists g \in G : s \preccurlyeq g$ and $\neg\exists s' \in S : s' \preccurlyeq s$  {3.4}
       else {$i$ is a $g$-bound }
            S := select all $s$ from $S$ with $s \sim i$  {3.5}
            if $S = \emptyset$
            then fail  {3.6}
            $G := specialize\_all( G , i )$  {3.7}
            $G$ := select all $g$ from $G$
                 with $\exists s \in S : s \preccurlyeq g$ and $\neg\exists g' \in G : g \preccurlyeq g'$  {3.8}
  endwhile
  return $G$, $S$
endproc
```

Algorithm 3.1 Description Identification (DI)

```
procedure generalize_all ( $S$: set of concept;$i$: $s$-bound )
          returns set of concept
   $S' := \emptyset$
   if $i$ is a positive lowerbound
   then for all $s \in S$
              do $S' := S' \cup mub( s , i )$   {3.9}
        endfor
   else  { $i$ is a negative upperbound}
        for all $s \in S$
              do $S' := S' \cup msg( s , i )$   {3.10}
        endfor
   return $S'$
endproc
procedure specialize_all ( $G$: set of concept;$i$: $g$-bound )
          returns set of concept
   $G' := \emptyset$
   if $i$ is a positive upperbound
   then for all $g \in G$
              do $G' := G' \cup mlb( g , i )$   {3.11}
        endfor
   else  { $i$ is a negative lowerbound}
        for all $g \in G$
              do $G' := G' \cup mgs( g , i )$   {3.12}
        endfor
   return $G'$
endproc
```

Algorithm 3.2 Generalization and Specialization in DI

in S more specific than some $g \in G$ (and therefore consistent with all g-bounds) and not more general than some other $s' \in S$ are retained in S (see Step 3.4).

The case of i being a g-bound is dual. First S is pruned (see Step 3.5): if an element of S is not consistent with i, then none of its generalizations can be consistent with i (because of the contraposition of Theorem 3.14). Whenever S is empty, the search has *collapsed* also, and the algorithm fails and stops (see Step 3.6). Otherwise all elements of G are specialized if necessary (see Algorithm 3.2). The specializations are gathered in G'. If i is a positive upperbound and g an element of G, all maximal lowerbounds (mlb) of g and i have to be added to G' (see Step 3.11), otherwise all most general specializations (mgs) of g and i are added to G' (see Step 3.12). After the result of *specialize* has been assigned to G (see Step 3.7), only those elements in G more general than some $s \in S$ (and therefore consistent with all s-bounds) and not more specific than some other element of G are retained in G (see Step 3.8).

This concludes the discussion of Algorithm 3.1 and Algorithm 3.2.

3.5.2 Discussion

To prune $\mathcal{G}$ and $\mathcal{S}$ and for checking maximal generality and maximal specificity, DI and CE make use of $\mathcal{G}$ and $\mathcal{S}$, and do not need to store previous information elements. Because of Theorem 3.25 DI and CE can correctly classify some unseen information elements. However, storing $\mathcal{G}$ and $\mathcal{S}$ is often very expensive, because the size of these sets can grow exponentially in the number of examples (see [Haussler, 1988]). [Korf, 1985] argues that exponential breadth-first search often exhausts the available memory long before an appreciable amount of time is used. When using very expressive descriptive languages (as in Inductive Logic Programming), memory efficiency becomes extremely important. Therefore many concept learning programs do not compute $\mathcal{S}$ and $\mathcal{G}$ completely, but rather only one maximally general element and/or one maximally specific element of $\mathcal{L}_C$. In general the question is then how to test for maximal specificity or maximal generality, when the other elements of $\mathcal{S}$ and $\mathcal{G}$ are not known.

Depth-first search (see [Mitchell, 1982]) does not have the same memory shortcoming: the main advantage of depth-first algorithms is their linear space complexity. However, without searching the *complete* search space (i.e., without recomputing $\mathcal{S}$ and $\mathcal{G}$) depth-first algorithms are in general unable to check maximal specificity and maximal generality, they can neither detect convergence, nor classify unseen information elements correctly. [Korf, 1985] discusses depth-first search versus breadth-first search in general, and presents depth-first iterative deepening as a search strategy with linear space complexity, and, in an exponential search space, the same worst case time complexity as breadth-first search. The underlying idea is that depth-first search avoids the memory problems of breadth-first search at the expense of recomputation.

Motivated by this general result, we developed the Iterative Versionspace algorithm (ITVS). As DI algorithm, ITVS can handle upperbounds as well as lowerbounds. ITVS is an incremental algorithm to compute one maximally specific and one maximally general concept representation using an adaptation of bi-directional depth-first search. Apart from backtracking, the backtrack information will also allow to check maximal specificity and maximal generality without having to recompute the complete search space. Convergence still cannot be detected and no unseen information elements can be classified correctly without searching the complete search space.

3.6 The Iterative Versionspaces algorithm

In this section we describe the *Iterative Versionspaces* framework and the Iterative Versionspaces algorithm. We start by introducing an alternative representation for $\mathcal{S}$ and $\mathcal{G}$ which does not store these sets, but yet allows to compute them completely if necessary. This representation is described by the datastructures used in the Iterative Versionspaces algorithm (Section 3.6.1), and by the invariants on these datastructures (Section 3.6.2). In Section 3.6.3 we then describe the Iterative Versionspaces algorithm, and prove its correctness.

3.6.1 The datastructures

The Iterative Versionspace algorithm combines general-to-specific and specific-to-general depth-first search.

We use the following datastructures:

- s is the current maximally specific concept representation, g is the current maximally general concept representation.
- the array I_s containing all s-bounds, and I_g containing all g-bounds. Lower- and upperbounds are needed for checking consistency while backtracking. n_s is the total number of elements in I_s, n_g is the total number of elements in I_g.
- the stack B_s containing triplets (ind , s_{ind} , alt_{ind}), called choicepoints, where ind is an index in I_s, s_{ind} is a concept representation, and alt_{ind} is a non-empty list of concept representations. These triplets are used to organize the search for s and to test maximal specificity of candidates for s. Similarly, the stack B_g contains choicepoints (ind , g_{ind} , alt_{ind}), where ind is an index in I_g, g_{ind} is a concept representation, and alt_{ind} is a non-empty list of concept representations. These triplets are used to organize the search for g and to test maximal generality of candidates for g.

In [Sablon *et al.*, 1994] s_{ind} was not used, because it is not necessary for the basic algorithms of ITVS. It is only needed for the algorithms of Section 3.9. However, to prove the invariants s_{ind} is involved in, we will already introduce s_{ind} here. This will have no major consequences for the complexity analysis of Section 3.8.2.

3.6.2 Invariants on the datastructures

We have the following invariants on the components of each versionspace:

- **Invariant 3.6.1.** (The maximal specificity invariant for s) $s \in \mathcal{S}_I$, with $I = I_g[1..n_g] \cup I_s[1..n_s]$.
- **Invariant 3.6.2.** (The maximal generality invariant for g) $g \in \mathcal{G}_I$, with $I = I_g[1..n_g] \cup I_s[1..n_s]$.
- **Invariant 3.6.3.** (The soundness invariant for B_s)
  For all choicepoints (ind_1 , s_1 , alt_1) on B_s:

- $s_1 \preccurlyeq s$, and $\neg(a_1 \preccurlyeq s)$ for all $a_1 \in alt_1$;
- s_1 and all elements of alt_1 are consistent with the elements of J and are maximally specific in $\mathcal{S}_J$, where $J = I_g[1..n_g] \cup I_s[1..ind_1]$, i.e., J contains *all* g-bounds and the first ind_1 s-bounds;
- for every choicepoint (ind_2 , s_2 , alt_2) closer to the top of B_s: $ind_1 < ind_2$, $s_1 \prec s_2$, $s_1 \prec a_2$ and $\neg(a_1 \preccurlyeq a_2)$ for every a_1 in alt_1 and for every a_2 in alt_2.

- **Invariant 3.6.4.** (The soundness invariant for B_g)
  For all choicepoints (ind_1 , g_1 , alt_1) on B_g:
  - $g \preccurlyeq g_1$, and $\neg(g \preccurlyeq a_1)$ for all $a_1 \in alt_1$;
  - g_1 and all elements of alt_1 are consistent with the elements of J and are maximally general in $\mathcal{G}_J$, where $J = I_s[1..n_s] \cup I_g[1..ind_1]$, i.e., J contains *all* s-bounds and the first ind_1 g-bounds;
  - for every choicepoint (ind_2 , g_2 , alt_2) closer to the top of B_g: $ind_1 < ind_2$, $g_2 \prec g_1$, $a_2 \prec g_1$ and $\neg(a_2 \preccurlyeq a_1)$ for every a_1 in alt_1 and for every a_2 in alt_2.

- **Invariant 3.6.5.** (The completeness invariant for B_s) For all $c \in \mathcal{L}_C$, consistent with I, s or an alternative for s on B_s[7] is more specific than c.

- **Invariant 3.6.6.** (The completeness invariant for B_g) For all $c \in \mathcal{L}_C$, consistent with I, g or an alternative for g on B_g is more general than c.

In the following we will shorten "x is consistent with the elements of J and is maximally specific in $\mathcal{S}_J$, where $J = I_g[1..n_g] \cup I_s[1..ind]$" to "x is maximally specific and consistent with $I_s[1..ind]$", and "x is consistent with the elements of J and is maximally general in $\mathcal{G}_J$, where $J = I_s[1..n_s] \cup I_g[1..ind]$" to "x is maximally general and consistent with $I_g[1..ind]$". In Figure 3.7 some of the invariants on B_s are illustrated. On the figure, dashed arrows are in the relation $\preccurlyeq$. Dashed arrows with a cross are *not* in the relation $\preccurlyeq$. s_1 and all elements of alt_1 are maximally specific and consistent with $I_s[1]$. s_2 and all elements of alt_2 are more general than s_1, and maximally specific and consistent with $I_s[1]$ and $I_s[2]$. s_3 and all elements of alt_3 are more general than s_2, and maximally specific and consistent with $I_s[1]$ to $I_s[3]$. Each of the consistent concept representations c_1, c_2 and c_3 is more general than s or more general than some alternative on B_s. Elements of alt_3 are not more general than elements of alt_2 or alt_1.

It is important to keep in mind that the alternatives on B_s are actually *the roots* of the search subtrees that are still to be searched. Consequently checking $a_1 \preccurlyeq a_2$ for the roots a_1 and a_2 of two search subtrees t_1 and t_2 corresponds to checking whether t_1 is a subtree of t_2. From this point of view s_{ind} is "the root" of all alternatives on B_s with index larger than *ind*.

This also means that the conditions $\neg(a_1 \preccurlyeq s)$ and $\neg(a_1 \preccurlyeq a_2)$ of Invariant 3.6.3 guarantee that each $c \in \mathcal{L}_C$ will not be generalized more than once. This restriction therefore implements an *optimal*, but still complete (because of Invariant 3.6.5), generalization

[7]With "an alternative for s on B_s" we mean "an element of alt_{ind} for a choicepoint (ind , s_{ind} , alt_{ind}) on B_s". Also, with "all alternatives for s on B_s" we mean "all elements of alt_{ind} for all choicepoints (ind , s_{ind} , alt_{ind}) on B_s".

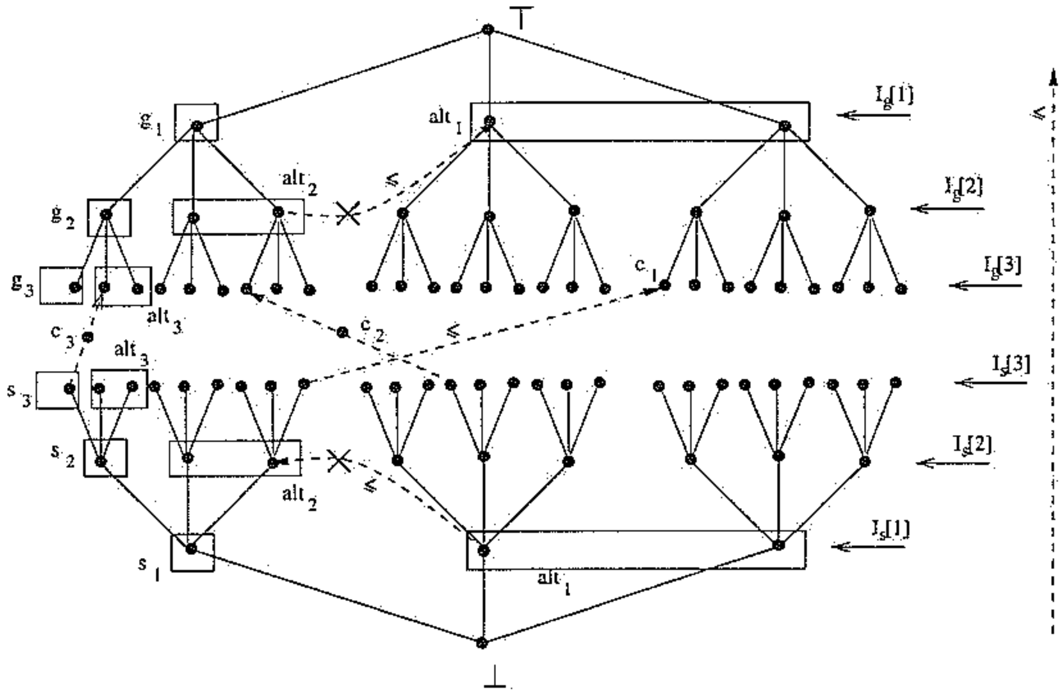


Figure 3.7 Datastructures and Invariants of ITVS

operator. By definition, a generalization operator is optimal if each $c \in \mathcal{L}_C$ will not be generalized more than once. Therefore optimal refinement operators avoid searching parts of the search space more than once.

Apart from implementing an optimal refinement operator, a second advantage of these invariants is the fact that if s is consistent with I, s is maximally specific and consistent with I. This is because all consistent $c \in \mathcal{L}_C$ are more general than s or than an alternative for s on B_s (Invariant 3.6.5). Because of the invariant $\neg(\ a_1 \preccurlyeq s\ )$, all c consistent with I and more specific than s, are not more specific than an alternative for s on B_s, which means that s is maximally specific. As a consequence there is no need for s to be compared with all other elements of S_I. The advantage lies in the fact that the computational cost of keeping these invariants invariant is *linear* in the number of information elements (see Theorem 3.40).

Dually, the conditions $\neg(\ g \preccurlyeq a_1\ )$ and $\neg(\ a_2 \preccurlyeq a_1\ )$ of Invariant 3.6.4 guarantee that each $c \in \mathcal{L}_C$ will not be specialized more than once. This restriction therefore implements an *optimal*, but still complete (because of Invariant 3.6.6), specialization operator.

The second advantage can also be dualized: if g is consistent with I, g maximally general and consistent with I. In this case there is no need to compare g with all other elements of $\mathcal{G}_I$. Again the computational cost of keeping the invariants satisfied is linear in the number of information elements.

As a drawback of implementing optimal refinement operators, solutions can only be reached through one path in the search tree. By implementing optimal refinement operators dynamically, solutions will only be found when no other path in the search tree to the

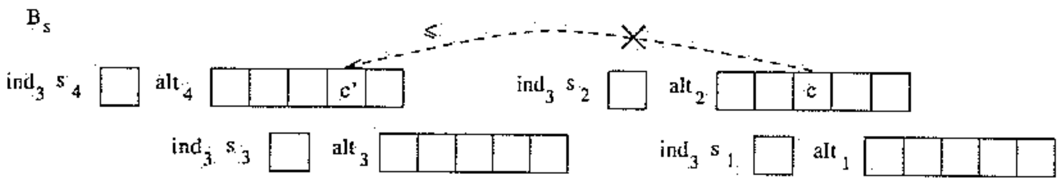


Figure 3.8 ITVS's representation with breadth-first search

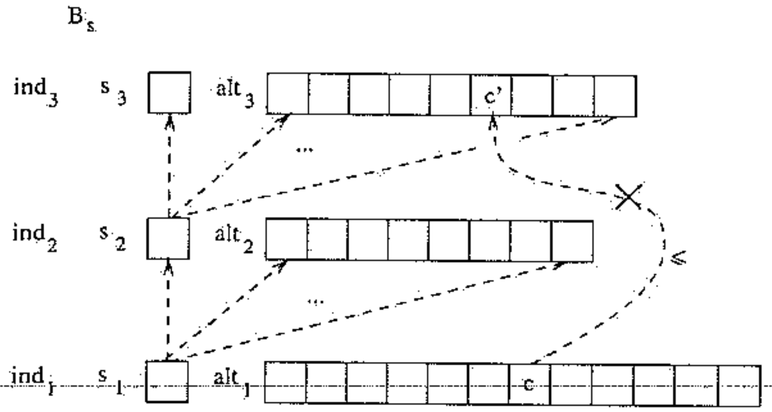


Figure 3.9 ITVS's representation with depth-first search

solution is left over. This means that the price to pay for having an optimal refinement operator, and for obtaining a maximally specific or a maximally general solution without having to search the complete search space, is that all paths to a solution c will have to be explored before c will be recognized as a solution.

In Section 3.8.2 we will consider using ITVS to generate $\mathcal{G}$ and $\mathcal{S}$ completely by means of backtracking, in order to compare it to DI. When backtracking is used to find more solutions, testing for maximal specificity and maximal generality will still include comparing candidate solutions to the solutions already found.

The application of these invariants is not restricted to ITVS. The whole spectrum of classical search methods from depth-first search (ITVS) to breadth-first search (DDI) can be described in the ITVS framework: the versionspace is represented by a collection of maximally specific representations together with an indication with which s-bounds they are consistent, and a collection of maximally general representations together with an indication with which g-bounds they are consistent, and this together with all information elements. In our algorithm these collections are represented by stacks to implement a depth-first search. All concept representations consistent with all information elements are more specific than some element in each collection, so that $\mathcal{S}$ and $\mathcal{G}$ can be (re)computed whenever necessary. Testing maximal specificity will range from linear in the number of information elements (as in ITVS) to linear in the number of elements in $\mathcal{S}$ or $\mathcal{G}$ (as in the Disjunctive version of DI; see further). In Figure 3.8, Figure 3.9 and Figure 3.10 these collections of maximally specific elements are represented for three different kinds of search

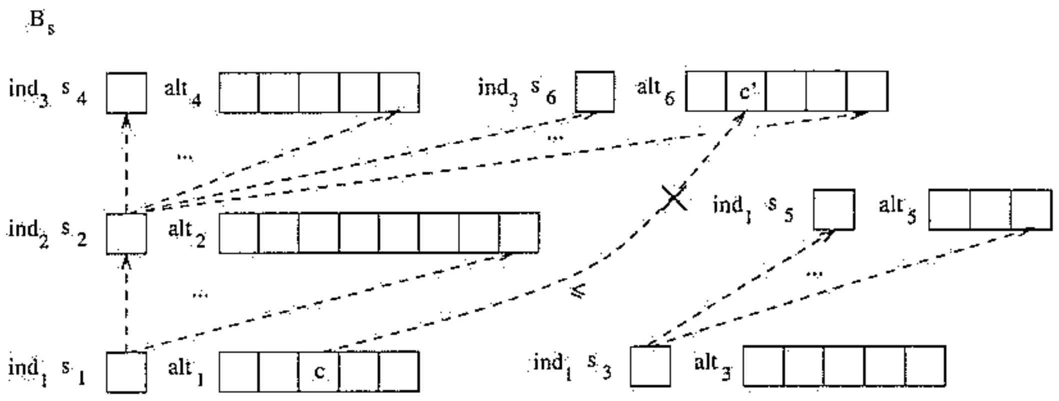


Figure 3.10 ITVS's representation with another search method

(resp. breadth-first search, depth-first search, and a search method in between). These figures illustrate that in this framework one cannot have $c \in alt_j$ and $c' \in alt_k$, such that $j < k$ and $c \preccurlyeq c'$ (see Invariant 3.6.3 and Invariant 3.6.4).

For several search strategies the worst case *space* complexity can be kept *linear* in the number of information elements. Theorem 3.38 proves this for ITVS, which uses a depth-first strategy. However, a hill-climbing strategy, retaining the ability to backtrack to retain completeness, can use the same backtrackstack as ITVS. Only the order the search space is searched will be altered. Similarly this can be extended to beam-search (again retaining the ability to backtrack), keeping m current hypotheses g and s instead of one. This approach would still have a worst case space complexity linear in the number of information elements.

At this point one can wonder whether this representation is epistemologically adequate for the operations on versionspaces listed at the end of Section 3.3. Since the representation of a versionspace by means of its $\mathcal{S}$ and $\mathcal{G}$ is epistemologically adequate for all these operations under Constraint 3.18 (the Boundedness Constraint) [Hirsh, 1992b], and since our datastructures allow to reconstruct $\mathcal{S}$ and $\mathcal{G}$ through backtracking, our representation is epistemologically adequate for all these operations under Constraint 3.18.

3.6.3 The Iterative Versionspaces algorithm [T]

SUMMARY: in this section we describe ITVS, and prove the invariants of Section 3.6.2 are satisfied.

The main algorithm

Consider Algorithm 3.3. The input to the algorithm is, as in DI, the stream $\mathcal{I}nf$ of information elements. First s is initialized to $\perp$, g to $\top$, B_s and B_g to the empty stack, and n_s and n_g to 0. These initializations make all invariants valid. The main loop of the algorithm processes the information elements one by one in the given order. The way i is processed depends on i being an s-bound or a g-bound.

We will first explain the case where i is an s-bound. First i is stored in I_s. Then all alter-

```
procedure ITVS ( $\mathcal{I}nf$: stream of info )
          returns concept,stack,array,index,concept,stack,array,index
  $s := \bot$; $g := \top$; $B_s := \emptyset$; $B_g := \emptyset$; $n_s := 0$; $n_g := 0$;
  while $\mathcal{I}nf$ is not empty
      do $i := read(\mathcal{I}nf)$
        if $i$ is an $s$-bound
        then $n_s := n_s + 1$; $I_s[n_s] := i$
             $B_g := prune\_stack(B_g, i)$
             if $\neg(g \sim i)$ {3.13}
             then $g, B_g, ind := select\_alternative(\emptyset, B_g, n_g)$
                  $g, B_g := specialize(g, B_g, ind)$ {3.14}
             $s, B_s := generalize(s, B_s, n_s - 1)$ {3.15}
        else {$i$ is a $g$-bound}
             $n_g := n_g + 1$; $I_g[n_g] := i$
             $B_s := prune\_stack(B_s, i)$
             if $\neg(s \sim i)$ {3.16}
             then $s, B_s, ind := select\_alternative(\emptyset, B_s, n_s)$
                  $s, B_s := generalize(s, B_s, ind)$ {3.17}
             $g, B_g := specialize(g, B_g, n_g - 1)$ {3.18}
  endwhile
  return $s$, $B_s$, $I_s$, $n_s$, $g$, $B_g$, $I_g$, $n_g$
endproc
```

Algorithm 3.3 The Iterative Versionspaces algorithm(ITVS)

natives for g on B_g not consistent with i are removed from B_g with the procedure **prune_stack**[8] (Algorithm 3.6). Indeed, if an alternative c on B_g is not consistent with i, then certainly none of its specializations will be consistent with i (because of Theorem 3.14), so c can be deleted from B_g without affecting Invariant 3.6.6. This pruning step ensures that all alternatives on B_g are consistent with $I_s[1..n_s]$.

If g is not consistent with i (see Step 3.13), an alternative for g is popped from B_g using the procedure **select_alternative**[9] (Algorithm 3.6). Because g was not consistent with i anyway, and because an alternative for g on B_g is removed from B_g and assigned to g, Invariant 3.6.6 is not affected.

In general *select_alternative*(*alt* , B_g , n_g) (with *alt* a list of elements in $\mathcal{L}_C$, B_g fulfilling Invariant 3.6.4 and Invariant 3.6.6, and n_g an index in I_g) returns

- a maximally general g, consistent with all s-bounds and the elements of $I_g[1..ind]$,
- a B_g which fulfills Invariant 3.6.4 and Invariant 3.6.6, and
- the index *ind* up to where g is consistent with the g-bounds.

The procedure call *select_alternative*(*alt* , B_g , n_g) fails if no such g, B_g and *ind* exist.

Then all information elements on I_g from *ind* up to n_g are reprocessed (see Step 3.14). Given that g is already maximally general and consistent with the first *ind* information elements in I_g and with all information elements in I_s, and given that B_g fulfills Invariant 3.6.4, *specialize*(g , B_g , *ind*) returns a maximally general g consistent with I, and a stack B_g fulfilling Invariant 3.6.4 and Invariant 3.6.6, or fails if no such g and B_g exist. Since g is maximally general and consistent with I, $g \in G_I$ (Invariant 3.6.2).

In the next step, given that s is maximally specific and consistent with the first $n_s - 1$ information elements of I_s and with all information elements in I_g, and given that B_s fulfills Invariant 3.6.3 and Invariant 3.6.5, *generalize*(s , B_s , $n_s - 1$) returns a maximally specific s consistent with I, and a stack B_s fulfilling Invariant 3.6.3 and Invariant 3.6.5, or fails when no such s and B_s exist. Therefore $s \in S_I$ (Invariant 3.6.1). Consequently, all invariants will hold after Step 3.15.

In case i is a g-bound, it is stored in I_g. Then all alternatives for s on B_s not consistent with i are removed from B_s, because if an alternative c on B_s is not consistent with i, then certainly none of its generalizations will be consistent with i (because of Theorem 3.14), so c can be deleted from B_s without affecting Invariant 3.6.5. This pruning step ensures that all alternatives on B_s are consistent with $I_g[1..n_g]$.

If s is not consistent with i (see Step 3.16), an alternative for s is popped from B_s using the procedure **select_alternative**. Because s was not consistent with i anyway, and because an alternative for s on B_s is removed from B_s and assigned to s, Invariant 3.6.5 is not affected. In general *select_alternative*(*alt* , B_s , n_c) (with *alt* a list of elements in $\mathcal{L}_C$, B_s fulfilling Invariant 3.6.3 and Invariant 3.6.5, and n_c an index in I_s such that all elements of alt_{ind} are consistent with $I_s[1..n_c]$) returns

- a maximally specific s, consistent with all g-bounds and the elements of $I_s[1..ind]$,
- a B_s which fulfills Invariant 3.6.3 and Invariant 3.6.5, and

[8] We use the operation *push*(*ind* , s_{ind} , alt_{ind} , B_c) to put the choicepoint (*ind* , s_{ind} , alt_{ind}) on top of stack B_c, *pop*(B_c) to remove the top choicepoint from B_c, and *is_empty*(B_c) to test whether B_c is empty.

[9] The assignment g, B_g, *ind* := *select_alternative*($\emptyset$, B_g , n_g) assigns the first returned value of *select_alternative*($\emptyset$, B_g , n_g) to g, the second one to B_g, and the third one to *ind*.

- the index *ind*, up to where s is consistent with s-bounds,

or fails if no such s, B_s, and *ind* exist. The s returned by `select_alternative` is an element of *alt*, if *alt* is not empty, or else an alternative of the top choicepoint of B_s, if B_s is not empty.

Then all information elements on I_s from *ind* up to n_s still have to be reprocessed (see Step 3.17). Given that s is already maximally specific and consistent with the first *ind* information elements in I_s and with all information elements in I_g, and given that B_s fulfills Invariant 3.6.3, *generalize*(s , B_s , *ind*) returns a $s \in S_I$ (Invariant 3.6.1), and a stack B_s fulfilling Invariant 3.6.3 and Invariant 3.6.5.

In the next step, given that g is maximally general and consistent with the first $n_g - 1$ information elements of I_g and with all information elements in I_s, and given that B_g fulfills Invariant 3.6.4 and Invariant 3.6.6, *specialize*(g , B_g , $n_g - 1$) returns $g \in G_I$, and a stack B_g fulfilling Invariant 3.6.4 and Invariant 3.6.6. Consequently, all invariants will also hold after Step 3.18.

Since all invariants hold after Step 3.15 and Step 3.18, they will also hold at the end of the while loop.

Note that backtracking on s and g is completely independent, in the sense that returning to the last choicepoint for s undoes all consequences for s, but not those for g: e.g., when returning to a choicepoint for s, values for g that were rejected after the choicepoint for s was created, are still rejected when other choices for s are made.

After processing an information element, convergence can only be detected by exhaustive backtracking on B_s and B_g, i.e., by checking that $g = s$ and that there are no consistent alternatives on B_s or B_g. Since this might be time consuming (see the complexity analysis in Section 3.8.2), Algorithm 3.3 does not detect convergence. A less time consuming, but only sufficient, condition for convergence is testing whether $g = s$ and $B_g = B_s = \emptyset$.

Generalization in ITVS

In this section we will explain how the procedure `generalize` works (see Algorithm 3.4).

We have to show that, given that s is maximally specific and consistent with the first n_c information elements of I_s and with all information elements in I_g, and given that B_s fulfills Invariant 3.6.3 and Invariant 3.6.5, *generalize*(s , B_s , n_c) returns a maximally specific s consistent with I, and a stack B_s fulfilling Invariant 3.6.3 and Invariant 3.6.5.

When $n_c = n_s$, s is maximally specific and consistent with *all* elements of I_s, so the procedure ends. Otherwise, after having incremented n_c with 1, s is generalized such that it is consistent with $I_s[n_c]$ (if it was not consistent already). The generalizations of s w.r.t. the s-bound $I_s[n_c]$ are computed in `generalizations`. If the s-bound is a positive lowerbound, consistent generalizations are all minimal upperbounds of s and the s-bound (see Step 3.21). Otherwise the s-bound is a negative upperbound, and the consistent generalizations are all most specific generalizations of s that are not more specific than the s-bound (see Step 3.22). Note that all consistent generalizations of s are also consistent with $I_s[1..n_c - 1]$ because of Theorem 3.14. In Step 3.19 only those generalizations c also consistent with all g-bounds, maximally specific and not more general than an alternative on B_s are selected (for `all_consistent` see Algorithm 3.6). This does not affect the completeness for B_s (Invariant 3.6.5):

- if there exists a c' on B_s such that $c' \preccurlyeq c$, then c will be considered for generalization on backtracking. This is guaranteed by the completeness of `msg` and `mub`. Consequently, there is no need to consider the generalizations of c at this point, without affecting completeness. In terms of search subtrees: if the subtree rooted by c is a subtree of the tree rooted by c', the tree of c will be explored when c' is being explored, so it does not have to be explored at this point.

```
procedure generalize( $s$: concept; $B_s$: stack; $n_c$: index ) returns concept, stack
    while $n_c \neq n_s$
        do $n_c := n_c + 1$
            if $\neg( s \sim I_s[n_c] )$
            then $gens := generalizations( s , I_s[n_c] )$
                 $gens :=$ select all $c$ from $gens$
                           with $all\_consistent( c , I_g , n_g )$
                            and $max\_specific( c , B_s )$  {3.19}
                 $s, B_s, n_c := select\_alternative( gens , B_s , n_c )$  {3.20}
    endwhile
    return $s, B_s$
endproc

procedure generalizations( $c$: concept; $i$: $s$-bound ) returns list of concept
    if $i$ is positive lowerbound
    then $gens := mub( c , i )$  {3.21}
    else  {$i$ is negative upperbound}
          $gens := msg( c , i )$  {3.22}
    return $gens$
endproc

procedure max_specific( $c$: concept; $B_s$: stack ) returns boolean
    $B_c := copy( B_s )$
    $max\_specific := true$
    while $\neg$ $is\_empty( B_c )$  and  $max\_specific$
        do $ind, s_{ind}, alt_{ind}, B_c := pop( B_c )$
            $max\_specific := (\neg \exists c' \in alt_{ind} : c' \preccurlyeq c)$
    endwhile
    return $max\_specific$
endproc
```

Algorithm 3.4 Generalization in ITVS

- if no c' exists on B_s such that $c' \preccurlyeq c$, then, because of transitivity of $\preccurlyeq$, consistent generalizations of any of the alternatives on B_s can neither be strictly more specific than c (and thus c belongs to $\mathcal{S}$) nor be equal to c (and thus the generalization operator is optimal for c).

From this and from Invariant 3.6.3 follows that every $c \in \mathcal{L}_C$, consistent with I, is more general than one of the selected generalizations or than an alternative for s on B_s. Then `select_alternative` (see Step 3.20 and Algorithm 3.6) selects a next candidate s, the corresponding B_s, and the index up to where s is maximally specific and consistent with I_s.

The call *max_specific*(c , B_s) checks whether c is maximally specific and not more general than an alternative on B_s. First `max_specific` (see Algorithm 3.4) copies the parameter B_s to B_c in order not to change B_s. Then *max_specific* is initialized to *true*. In the while-loop all choicepoints are popped from B_c, until B_c is empty, or until a choicepoint has been found containing an alternative which is more specific than c. If such a choicepoint is found, *max_specific* becomes *false*. Finally *max_specific* is returned.

Specialization in ITVS

The procedure `specialize` (see Algorithm 3.5) is dual to `generalize`. When $n_c = n_g$, g is maximally general and consistent with *all* elements of I_g, so the procedure ends. Otherwise, after having incremented n_c with 1, g is specialized such that it is consistent with $I_g[n_c]$ (if it was not consistent already). The specializations of g w.r.t. the g-bound $I_g[n_c]$ are computed in `specializations`. If the g-bound is a negative lowerbound, consistent specializations are all most general specializations of g not covering the g-bound (see Step 3.25). Otherwise the g-bound is a positive upperbound, and the consistent specializations are all maximal lowerbounds of g and the g-bound (see Step 3.26). Again all consistent specializations of g are also consistent with $I_g[1..n_c - 1]$ because of Theorem 3.14. In Step 3.23 only those specializations also consistent with all s-bounds and maximally general are selected. This does not affect the completeness for B_g (Invariant 3.6.6):

- if there exists a c' on B_g such that $c \preccurlyeq c'$, then c will be considered for specialization on backtracking. This is guaranteed by the completeness of mgs and mlb. Consequently, there is no need to consider the generalizations of c at this point, without affecting completeness.

- if no c' exists on B_g such that $c \preccurlyeq c'$, then, because of transitivity of $\preccurlyeq$, consistent specializations of any of the alternatives on B_g can neither be strictly more general than c (and thus c belongs to $\mathcal{G}$) nor be equal to c (and thus is the specialization operator optimal for c).

From this and from Invariant 3.6.4 follows that every $c \in \mathcal{L}_C$, consistent with I, is more specific than one of the selected specializations or than an alternative for g on B_g. Then `select_alternative` (see Step 3.24 and Algorithm 3.6) selects a next candidate g, the corresponding B_g, and the index up to where g is maximally general and consistent with I_g.

The call `max_general` is dual to `max_specific`. *max_general*(g , B_g) checks whether g is maximally general, and not more general than an alternative on B_g.

Auxiliary procedures in ITVS

The procedure call *all_consistent*(c , I_g , n_g) checks whether $c \sim I_g[1..n_g]$ and is straightforward. If B_g is empty, this test can be replaced by the test $c \preccurlyeq g$. This is more efficient for languages in which $\mid \mathcal{G} \mid$ is always equal to 1.

```
procedure specialize( $g$: concept; $B_g$: stack; $n_c$: integer ) returns concept, stack
    while $n_c \neq n_g$
        do $n_c := n_c + 1$
            if $\neg( g \sim I_g[n_c] )$
            then $specs := specializations( s , I_g[n_c] )$
                  $specs :=$ select all c from $specs$
                            with $all\_consistent( c , I_s , n_s )$
                            and $max\_general( c , B_g )$ {3.23}
                  $g, B_g, n_c := select\_alternative( specs , B_g , n_c )$ {3.24}
    endwhile
    return $g, B_g$
endproc

procedure specializations( $c$: concept; $i$: $g$-bound ) returns list of concept
    if $i$ is positive upperbound
    then $specs := mlb( c , i )$ {3.25}
    else {$i$ is negative lowerbound}
          $specs := mgs( c , i )$ {3.26}
    return $specs$
endproc

procedure max_general( $c$: concept; $B_g$: stack ) returns boolean
    $B_c := copy( B_g )$
    $max\_general := true$
    while $\neg$ $is\_empty( B_c )$ and $max\_general$
        do $ind, s_{ind}, alt_{ind}, B_c := pop( B_c )$
            $max\_general := (\neg \exists c' \in alt_{ind} : c \preccurlyeq c')$
    endwhile
    return $max\_general$
endproc
```

Algorithm 3.5 Specialization in ITVS

```
procedure all_consistent( c: concept; $I_c$: array; $n_c$: index ) returns boolean
    return $\forall ind,\ 1 \leq ind \leq n_c :\ c \sim I_c[ind]$
endproc

procedure prune_stack( $B_c$: stack; $i$: info ) returns stack
    if $is\_empty( B_c )$
    then return $\emptyset$
    else $ind,\ s_{ind},\ alt_{ind},\ B_c := pop( B_c )$
         $alt_{ind}$ := select all $c$ from $alt_{ind}$ with $c \sim i$
         $B_c := prune\_stack( B_c\ , i )$
         if $alt_{ind} \neq \emptyset$
         then $B_c := push( ind\ , s_{ind}\ , alt_{ind}\ , B_c )$
    return $B_c$
endproc

procedure select_alternative( $alt$: list; $B_c$: stack; $n_c$: index )
            returns concept, stack, index
    if $alt = \emptyset$
    then if $is\_empty( B_c )$
         then failure {3.27}
         else $n_c,\ s,\ alt,\ B_c := pop( B_c )$ {3.28}
    $c := head( alt )$ {3.29}
    if $tail( alt ) \neq \emptyset$
    then $B_c := push( n_c\ , c\ , tail( alt )\ , B_c )$ {3.30}
    return $c,\ B_c,\ n_c$
endproc
```

Algorithm 3.6 Auxiliary procedures in ITVS

The procedure `prune_stack` is written recursively. If the given stack is empty, the procedure returns an empty stack. Otherwise, it pops the top choicepoint of the stack and removes those elements that are inconsistent. After pruning the rest of the stack, the pruned choicepoint is pushed back onto the stack, if it has not become empty.

Given a list *alt* of concept representations, a stack B_c, and an index n_c in I_c, the procedure `select_alternative` works as follows: it selects an element c of *alt* (see Step 3.29); if *alt* is initially empty, it first pops a choicepoint from B_c (see Step 3.28). If *alt* and B_c are empty, no c consistent with I can exist (because of Invariant 3.6.5 and Invariant 3.6.6), so `select_alternative` announces failure and halts ITVS (see Step 3.27). Otherwise, the choicepoint $\langle\, n_c\ ,\ c\ ,\ alt\,\rangle$ is pushed onto B_c (see Step 3.30), and it returns c, B_c and n_c, such that c is maximally general and consistent with the elements of $I_c[1..n_c]$. Therefore, initially n_c must be such that all elements of *alt* are maximally general and consistent with the elements of $I_c[1..n_c]$, in case B_c is B_g, and maximally specific and consistent with the elements of $I_c[1..n_c]$, in case B_c is B_s. Also, depending on B_c being B_g, resp. B_s, it will fulfill Invariant 3.6.4 and Invariant 3.6.6, or resp. Invariant 3.6.3 and Invariant 3.6.5. If B_c is B_g, we should therefore have initially that Invariant 3.6.4 holds, and that for every $c \in \mathcal{L}_C$, consistent with I, an element of *alt* or an alternative for g on B_g is more general than c. Similarly, if B_c is B_s, we should have initially that Invariant 3.6.3 holds, and that for every $c \in \mathcal{L}_C$, consistent with I, an element of *alt* or an alternative for s on B_s is more specific than c.

3.7 Examples

In this section we will give some examples of the Iterative Versionspaces algorithm.

Example 1

The first example is based on the lattice $\mathcal{M}$ (see Figure 3.11) of [Mellish, 1991]. Figure 3.12 shows the consecutive stages of the example session with ITVS [10]. The target concept is represented as t. Initially g is $\top$, and s is $\bot$. The first information element is a negative upperbound and is stored in $I_s[1]$. With respect to this negative upperbound s is generalized to *inanimate*, and the alternatives *female* and *male* are put on B_s. The second information element is a positive lowerbound, and therefore stored in $I_s[2]$. It is not consistent with *inanimate*, so *inanimate* should be generalized. The concept representation *inanimate* can only be generalized to $\top$. The concept representation $\top$ is more general than *female* on B_s, so it can be skipped at this point. The most recent alternative on B_s (i.e., *female*) is assigned to s, and only *male* remains on B_s. The concept representation *female* is consistent with $I_s[2]$. The third information element is a negative lowerbound that forces g to be specialized to *animate*, which is maximally general, since there are no alternatives on B_g. The fourth information element is a positive upperbound, according to which *male* should be pruned from B_s and g should be specialized. The only consistent maximally general specialization is *female*, which is also consistent with the other information elements. The resulting s and g is *female*. Since $g = s$ and B_g and B_s are empty, *female* must be the target concept.

[10] In the tables we have shortened "animate" to "anim" and "inanimate" to "inanim".

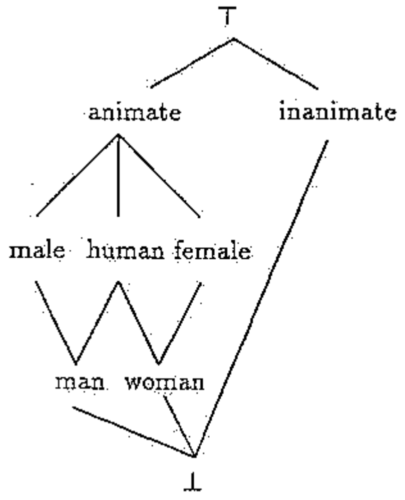


Figure 3.11 Taxonomy $\mathcal{M}$

New Information	Stored in	g	B_g	s	B_s
		$\top$	$\emptyset$	$\bot$	$\emptyset$
$\neg(t \preccurlyeq human)$	$I_s[1]$	$\top$	$\emptyset$	$inanim$	$[(1, inanim, [female, male])]$
$woman \preccurlyeq t$	$I_s[2]$	$\top$	$\emptyset$	$female$	$[(1, female, [male])]$
$\neg(inanim \preccurlyeq t)$	$I_g[1]$	$anim$	$\emptyset$	$female$	$[(1, female, [male])]$
$t \preccurlyeq female$	$I_g[2]$	$female$	$\emptyset$	$female$	$\emptyset$

Figure 3.12 Example 1

Example 2

	New Information	Stored In
1	$\neg(t \preccurlyeq c(human, \top))$	$I_s[1]$
2	$t \preccurlyeq c(\top, anim)$	$I_g[1]$
3	$\neg(c(woman, woman) \preccurlyeq t)$	$I_g[2]$
4	$t \preccurlyeq c(anim, anim)$	$I_g[3]$
5	$\neg(t \preccurlyeq c(woman, \top))$	$I_s[2]$
6	$c(woman, \bot) \preccurlyeq t$	$I_s[3]$
7	$\neg(c(man, \bot) \preccurlyeq t)$	$I_g[4]$
8	$c(woman, man) \preccurlyeq t$	$I_s[4]$
9	$\neg(t \preccurlyeq c(\top, human))$	$I_s[5]$
10	$\neg(c(man, man) \preccurlyeq t)$	$I_g[5]$

Figure 3.13 Information elements of Example 2

	g	B_g
	s	B_s
	$c(\top, \top)$	$\emptyset$
	$c(\bot, \bot)$	$\emptyset$
1	$c(\top, \top)$	$\emptyset$
	$c(inanim, \bot)$	$[(1, c(inanim, \bot), [c(female, \bot), c(male, \bot)])]$
2	$c(\top, anim)$	$\emptyset$
	$c(inanim, \bot)$	$[(1, c(inanim, \bot), [c(female, \bot), c(male, \bot)])]$
3	$c(inanim, anim)$	$[(2, c(inanim, anim), [c(male, anim), c(\top, male)])]$
	$c(inanim, \bot)$	$[(1, c(inanim, \bot), [c(female, \bot), c(male, \bot)])]$
4	$c(male, anim)$	$[(2, c(male, anim), [c(\top, male)])]$
	$c(female, \bot)$	$[(1, c(female, \bot), [c(male, \bot)])]$
5	$c(male, anim)$	$[(2, c(male, anim), [c(\top, male)])]$
	$c(female, \bot)$	$[(1, c(female, \bot), [c(male, \bot)])]$
6	$c(anim, male)$	$\emptyset$
	$c(female, \bot)$	$[(1, c(female, \bot), [c(male, \bot)])]$
7	$c(female, male)$	$\emptyset$
	$c(female, \bot)$	$\emptyset$
8	$c(female, male)$	$\emptyset$
	$c(female, man)$	$\emptyset$
9	$c(female, male)$	$\emptyset$
	$c(female, male)$	$\emptyset$

Figure 3.14 States of g, B_g, s and B_s in Example 2

In the second example $\mathcal{L}_C$ is the direct product of the lattice $\mathcal{M}$ with itself. Concepts are couples $c(X , Y)$ with $X,Y \in \mathcal{M}$. In the direct product the relation $\preccurlyeq$ is defined by

$$c(X_1 , X_2) \preccurlyeq c(Y_1 , Y_2) \text{ iff } (X_1 \preccurlyeq Y_1 \text{ and } X_2 \preccurlyeq Y_2).$$

The top element of $\mathcal{L}_C$ is then $c(\top , \top)$ and the bottom element is $c(\bot , \bot)$.

The concept representation $c(X , Y)$ can be interpreted as a couple of creatures, X on the left, Y on the right. The target concept is a particular subset of the set of all couples of creatures. The latter set is represented by $c(\top , \top)$. $c(X , \bot)$ represents a creature X on the left, with nothing at the right. Dually $c(\bot , Y)$ represents a creature Y on the right, with nothing at the left. The aim is to find a representation of the target concept, i.e., to find which creature must be on the right, and which creature must be on the left. Figure 3.13 shows the consecutive information elements given to ITVS, and Figure 3.14 shows the respective stages of the example session with ITVS. Note that some of the information elements actually do not change s, g, B_s or B_g, e.g., $\neg(t \preccurlyeq c(woman, \top))$, the fifth information element. Also note that if the order of the information elements were different, still other information elements would not change s, g, B_s or B_g, e.g., $t \preccurlyeq c(\top, animate)$ (the second information element) and $t \preccurlyeq c(animate, animate)$ (the seventh information element). In these particular cases, this is a consequence of a particular relation between the information elements themselves. This will be discussed in Section 3.9. There we will also discuss how, after transformation of s, g, B_s and B_g, the second and fifth information element can even be omitted.

After the ninth information element, search has converged to $s = g = c(female, male)$, and $B_s = B_g = \emptyset$.

Example 3

	New Information	Stored In
1	$\neg(t \preccurlyeq c(human, \top))$	$I_s[1]$
2	$\neg(c(woman, woman) \preccurlyeq t)$	$I_g[1]$
3	$c(woman, man) \preccurlyeq t$	$I_s[2]$
4	$c(man, woman) \preccurlyeq t$	$I_s[3]$

Figure 3.15 Information elements of Example 3

In this example (see Figure 3.15 and Figure 3.16), the target concept is not in $\mathcal{L}_C$. The resulting s after processing the third information element is $c(female, \bot)$. This s cannot be generalized to cover $c(man, woman)$, without covering $c(woman, woman)$. In Chapter 4 we will discuss how to extend the concept representation language in order to be able to represent the target concept of this example, by introducing disjunctions.

3.8 Properties of ITVS

3.8.1 Completeness, soundness and finiteness

Theorem 3.35 ITVS fails iff there exists no $c \in \mathcal{L}_C$ consistent with I.

	g	B_g
	s	B_s
	$c(\top,\top)$	$\emptyset$
	$c(\bot,\bot)$	$\emptyset$
1	$c(\top,\top)$	$\emptyset$
	$c(inanim,\bot)$	$[(1, c(inanim,\bot), [c(female,\bot), c(male,\bot)])]$
2	$c(\top,inanim)$	$[(1, c(\top,inanim), [c(\top,male), c(male,\top), c(inanim,\top)])]$
	$c(inanim,\bot)$	$[(1, c(inanim,\bot), [c(female,\bot), c(male,\bot)])]$
3	$c(\top,male)$	$\emptyset$
	$c(female,\bot)$	$[(1, c(female,\bot), [c(male,\bot)])]$
4	FAILURE	

Figure 3.16 States of g, B_g, s and B_s in Example 3

Proof ITVS only fails (see Step 3.27) when s is inconsistent with I, and no alternatives for s on B_s exist, and also fails when g is inconsistent with I, and no alternatives for g on B_g exist. From Invariant 3.6.5 and Invariant 3.6.6 follows that this only happens when there exists no $c \in \mathcal{L}_C$ consistent with I. □

Theorem 3.36 If ITVS does not fail, $s \in \mathcal{S}_I$ and $g \in \mathcal{G}_I$.

Proof This follows from Invariant 3.6.1 and Invariant 3.6.2. □

Theorem 3.37 For every given finite set I of information elements, ITVS halts in finite time.

Proof From Constraint 3.19 (the finiteness constraint) follows that the search spaces for the specific-to-general search and the general-to-specific search are both finite for a finite number of information elements. ITVS implements a depth-first search in both search spaces, and is therefore finite. □

3.8.2 Complexity Analysis [T]

SUMMARY: in this section we analyze the computational complexity of ITVS. The main result of this section is that the worst case space complexity of ITVS is *linear* in the number of information elements. Testing maximal specificity and maximal generality of candidate hypotheses is in the worst case also linear in the number of information elements. If ITVS is used to compute $\mathcal{S}$ and $\mathcal{G}$ completely, its worst case time complexity is a linear factor worse than the worst case time complexity of DI. However, if the size of $\mathcal{S}$ or the size of $\mathcal{G}$ is exponential in the number of information elements, ITVS realizes an exponential improvement in space requirements w.r.t. DI.

We analyze the worst case time and space complexity as in [Hirsh, 1992a]. For the time complexity analysis we count the number of $\preccurlyeq$-tests and the number of applications of the refinement operators (*mub*, *msg*, *mlb* and *mgs*), as a function of the number of information elements. For the space complexity we count the number of elements of $\mathcal{L}_C$ stored. Other factors that play a role (such as the chosen concept representation language, the branching of $\preccurlyeq$ in $\mathcal{L}_C$ and the order of the information elements) will be averaged, assuming a uniform distribution of positive and negative lower- and upperbounds over I.

The space complexity of the elements of $\mathcal{L}_C$ and the time complexity of $\preccurlyeq$-tests and generalization and specialization operations are language dependent. As in [Hirsh, 1992a] we will assume they are constant. The space complexity of instance representations (i.e., elements of $\mathcal{L}_I$) is denoted by c_i; the space complexity of concept representations (i.e., elements of $\mathcal{L}_C$) is denoted by c_c. The time complexity of $\preccurlyeq$ is denoted by $c_{\preccurlyeq}$; the time complexity of the generalization operators is denoted by c_{gen}, and of the specialization operators by c_{spec}. Note that w.r.t. [Mitchell, 1982] and [Sablon *et al.*, 1994] we additionally count the number of specialization and generalization operations, because these operations might have a complexity of the same order as testing $\preccurlyeq$.

Let $\bar{s}$ be the size of the specific-to-general search space, and $\bar{g}$ the size of the general-to-specific search space. Also let b_s be the average upward branching factor in $\mathcal{S}_I$, and b_g the average downward branching factor in $\mathcal{G}_I$. The average upward branching factor is the average number of generalizations (i.e., most specific generalizations and minimal upperbounds) that pass the test Step 3.19 in algorithm 3.4. The average downward branching factor is the average number of specializations (i.e., most general specializations and maximal lowerbounds) that pass the test Step 3.23 in algorithm 3.5. Then b_g and b_s are finite because of Constraint 3.19 (the finiteness constraint). The search spaces (and therefore also b_s and b_g) are completely determined by the elements of I and their order. Because we implement an *optimal* generalization and specialization operator (see Section 3.6.2), we can use the same branching factor as in DI (instead of the *edge branching factor* of [Korf, 1985]): in DI searching parts of the search space more than once is avoided by using a set representation for $\mathcal{S}$ and $\mathcal{G}$ and hence implicitly removing doubles (see Algorithm 3.1 and Algorithm 3.2). The resulting complexity analysis shows that optimal operators are useful, and can be implemented efficiently (w.r.t. time and space) in the context of versionspaces.

Worst case space complexity

Theorem 3.38 ITVS has a worst case space complexity of

$$\mathcal{O}(\ (n_s + n_g) \times c_i + (n_s \times b_s + n_g \times b_g) \times c_c\ ).$$

Proof ITVS stores all n_s s-bounds in I_s (this yields the term $n_s \times c_i$). Furthermore B_s contains in the worst case one concept representation s_{ind} and a list alt_{ind} of $b_s - 1$ alternatives per s-bound (this yields the term $n_s \times b_s \times c_c$).

Dually I_g contains all n_g g-bounds (this yields the term $n_g \times c_i$), and B_g contains in the worst case g_{ind} and a list alt_{ind} of $b_g - 1$ alternatives per g-bound (this yields the term $n_g \times b_g \times c_c$). □

This means that the worst case space complexity of ITVS is *linear* in the number of information elements.

We can compare this to the worst case space complexity of DI.

Theorem 3.39 (Adapted from [Mitchell, 1982]). DI has a worst case space complexity of

$$\mathcal{O}(\ (\bar{\mathcal{S}} + \bar{\mathcal{G}}) \times c_c\ ).$$

Proof DI stores the sets $\mathcal{S}$ and $\mathcal{G}$ completely, but does not store any information elements. □

In case the size of $\mathcal{G}$ or $\mathcal{S}$ is exponential in the number of g-bounds or s-bounds, ITVS realizes an exponential improvement w.r.t. DI.

Worst case time complexity

Theorem 3.40

- For each s-bound, updating s, B_s, g and B_g in case no backtracking is needed, has a worst case time complexity of
$$\mathcal{O}(\ (\ n_g \times (b_g + b_s - 1) + n_s \times b_s^2\ ) \times c_{\preccurlyeq} + 1 \times c_{gen}\ ).$$
- For each g-bound, updating s, B_s, g and B_g in case no backtracking is needed, has a worst case time complexity of
$$\mathcal{O}(\ (\ n_s \times (b_s + b_g - 1) + n_g \times b_g^2\ ) \times c_{\preccurlyeq} + 1 \times c_{spec}\ ).$$

Proof In case no backtracking is needed, the update w.r.t. an s-bound (the then part in Algorithm 3.3) requires one generalization operation. W.r.t. a g-bound an update without backtracking (the else part in Algorithm 3.3) requires one specialization operation.

We will now count the number of $\preccurlyeq$-tests. In case of an s-bound i, during the pruning step consistency of i has to be checked w.r.t. all alternatives on B_g. This gives $n_g \times (b_g - 1)$ tests in the worst case. Then, for each of the b_s generalizations of s we have n_g tests to check consistency with I_g (yielding the term $\mathcal{O}(\ n_g \times b_s\ )$), and $n_s \times b_s$ tests to check maximal specificity (yielding the term $\mathcal{O}(\ n_s \times b_s^2\ )$).

In case of an g-bound, i has to be checked w.r.t. all alternatives on B_s. This gives $n_s \times (b_s - 1)$ tests in the worst case. Then, for each of the b_g specializations of g we have n_s tests to check consistency with I_s, and $n_g \times b_g$ tests to check maximal generality. □

The importance of this theorem lies in the fact that we can update each element of $\mathcal{G}$ and $\mathcal{S}$ in a time *linear* in n_g and n_s. The main time complexity factor will therefore be the computation of an alternative element of $\mathcal{G}$, resp. $\mathcal{S}$, in case the current g, resp. s, is not consistent with I. Backtracking could be reduced by changing the depth-first algorithm into a hill-climbing strategy, retaining the ability to backtrack, and therefore retaining the completeness property. Step 3.29 in `select_alternative` (see Algorithm 3.6) would then not just return an element of the list alt_{ind}, but rather select the best element of alt_{ind} according to the heuristic. The worst case space complexity would still be linear in the number of information elements. Similarly we could extend this to a kind of beam-search (again retaining the ability to backtrack), in which we keep m current hypotheses g and s instead of one. This approach would still have a worst case space complexity linear in the number of information elements. The worst case time complexity is also linear, when no backtracking is needed. The use of the optimal refinement operator will make sure no parts of the search space are searched more than once, and that the solution found is maximally specific or maximally general.

Theorem 3.41 To compute a maximally specific concept representation s and a maximally general concept representation g, ITVS has a worst case time complexity of

$$\mathcal{O}(\quad \bar{s} \times c_{gen} + \bar{g} \times c_{spec} + \\ (\bar{s} \times (n_g + n_s \times b_s) + \bar{g} \times (n_s + n_g \times b_g) + n_s \times n_g \times (b_s + b_g)) \times c_{\preccurlyeq}.\quad)$$

Proof In the worst case the specific-to-general and general-to-specific search spaces have to be searched completely.

W.r.t. the number of specialization operations, *all* $\bar{s}$ elements s of the specific-to-general search space have to be generalized once. This gives the term $\bar{s} \times c_{gen}$. Similarly, *all* $\bar{g}$ elements g of the general-to-specific search space have to be specialized once. This gives the term $\bar{g} \times c_{spec}$.

We will now count the number of $\preccurlyeq$-tests. In the specific-to-general case, for all $\bar{s}$ elements s of the search space, all n_g g-bounds may have to be reexamined for consistency. Guaranteeing maximal specificity of s requires s to be compared to all alternatives on B_s. There are at most $n_s \times (b_s - 1)$ alternatives on B_s. This gives a number of $\preccurlyeq$-tests in the order of $\bar{s} \times (n_g + n_s \times b_s)$. Guaranteeing consistency and maximal generality of g gives another $\bar{g} \times (n_s + n_g \times b_g)$ $\preccurlyeq$ tests.

The pruning steps are done only once for each information element. For B_g this results in $n_g \times (b_g - 1)$ $\preccurlyeq$-tests for each of the n_s s-bounds. Similarly pruning of B_s will give another $n_g \times n_s \times b_s$ $\preccurlyeq$-tests. □

To compare ITVS with DI, we will also compute the worst case time complexity for ITVS to recompute $\mathcal{S}$ and $\mathcal{G}$ completely. As noted in Section 3.6.3 this *does* require the elements of $\mathcal{S}$ and $\mathcal{G}$ to be stored (albeit temporarily) to check maximal specificity and maximal generality for the consecutive elements. The worst case time complexity will therefore depend on the sizes of $\mathcal{S}$ and $\mathcal{G}$. We will denote the size of $\mathcal{S}$ by $\bar{\mathcal{S}}$, and the size of $\mathcal{G}$ by $\bar{\mathcal{G}}$.

Theorem 3.42 To compute $\mathcal{S}$ and $\mathcal{G}$ *for each new information element*, or to detect convergence, ITVS has a worst case time complexity of

$$\mathcal{O}(\quad (n_s + n_g) \times (\bar{s} \times c_{gen} + \bar{g} \times c_{spec} + (\bar{s} \times (n_g + n_s \times b_s + \bar{\mathcal{S}}) + \bar{g} \times (n_s + n_g \times b_g + \bar{\mathcal{G}})) \times c_{\preccurlyeq}) + n_s \times n_g \times (b_s + b_g) \times c_{\preccurlyeq}. \quad)$$

Proof To compute $\mathcal{S}$ from s and B_s and $\mathcal{G}$ from g and B_g, the parts of the specific-to-general and general-to-specific search spaces that are not yet pruned, have to be recomputed completely. Hence, the worst case time complexity of Theorem 3.41 must be multiplied by $n_s + n_g$, the total number of information elements. Additionally, each element of the specific-to-general search space might have to be compared to each element of $\mathcal{S}$ collected so far, which yields an extra term $(n_s + n_g) \times \bar{s} \times \bar{\mathcal{S}} \times c_{\preccurlyeq}$. Dually an extra term $(n_s + n_g) \times \bar{g} \times \bar{\mathcal{G}} \times c_{\preccurlyeq}$ must be added because each element of the general-to-specific search space might have to be compared to each element of $\mathcal{G}$ collected so far.

Note that the terms originating from pruning B_s and B_g are not multiplied by $n_s + n_g$, since pruning still happens only once for each information element.

After having computed $\mathcal{S}$ and $\mathcal{G}$, detecting convergence is just checking whether $\mathcal{S}$ and $\mathcal{G}$ are equal and singletons. □

Note that to classify an unseen information element i correctly, each element of $\mathcal{S}$ and $\mathcal{G}$ will also have to be computed, and compared to i.

Theorem 3.43 (Adapted from [Mitchell, 1982] and [Hirsh, 1992a]).
DI has a worst case time complexity of

$$\mathcal{O}(\bar{s} \times c_{gen} + \bar{g} \times c_{spec} + (\bar{s} + \bar{g}) \times (\bar{\mathcal{S}} + \bar{\mathcal{G}}) \times c_{\preccurlyeq}).$$

Proof Each element of the specific-to-general search space will be generalized once. This gives a term $\mathcal{O}(\ \bar{s} \times c_{gen}\ )$. Dually, each element of the general-to-specific search space will be specialized once, which gives a term $\mathcal{O}(\ \bar{g} \times c_{spec}\ )$.

W.r.t. $\preccurlyeq$-tests, each element s in the specific-to-general search space, has to be compared to all $\bar{\mathcal{G}}$ elements in $\mathcal{G}$ to check consistency, and to all $\bar{\mathcal{S}}$ elements in $\mathcal{S}$ to check maximal specificity. Also each element g in the general-to-specific search space, has to be compared to all elements in $\mathcal{S}$ to check consistency, and to all elements in $\mathcal{G}$ to check maximal generality. □

We can now compare DI to the version of recomputing $\mathcal{S}$ and $\mathcal{G}$ for each information element (Theorem 3.42).

Discussion

When $b_s > 1$ and $b_g > 1$, ITVS basically is a linear factor worse than DI in time to compute $\mathcal{S}$ and $\mathcal{G}$ completely, because in the exponential case

$$\bar{s} = 1 + b_s + b_s^2 + \ldots + b_s^{n_s} = \frac{b_s^{n_s+1} - 1}{b_s - 1} \approx b_s^{n_s},$$

and $\bar{\mathcal{S}} = b_s^{n_s}$. Dually $\bar{g} \approx b_g^{n_g} = \bar{\mathcal{G}}$. The terms $\mathcal{O}(\ \bar{s} \times \bar{\mathcal{G}}\ )$ and $\mathcal{O}(\ \bar{g} \times \bar{\mathcal{S}}\ )$ of the worst case time complexity of Theorem 3.43 can be considered of the same order as $\mathcal{O}(\ \bar{s} \times \bar{\mathcal{S}}\ )$ and $\mathcal{O}(\ \bar{g} \times \bar{\mathcal{G}}\ )$. The linear factor appears as well in the number of $\preccurlyeq$-tests, as in the number of generalization and specialization operations. In that sense this result extends the result of [Sablon *et al.*, 1994], where only the number of $\preccurlyeq$-tests were counted. Basically the linear factor arises because ITVS needs to recompute the search space once for each information element. On the other hand, ITVS is not intended to compute $\mathcal{S}$ and $\mathcal{G}$ completely, but rather only one maximally general element and one maximally specific element of it. To do so, its space requirements are linear, whereas DI is still exponential as soon as $b_s > 1$ or $b_g > 1$. As [Korf, 1985] we argue that this is an important improvement for concept learning, since combinatorial explosion of space requirements is much more critical than explosion in time. To conclude we could say that, in case the learning system needs to find only one maximally specific or maximally general concept representation, ITVS shows an exponential time improvement over DI; for correct classification however, time is a linear factor worse.

More related work

Under strong restrictions, earlier incremental approaches already presented better complexity results than DI. In particular, [Smith and Rosenbloom, 1990] and [Hirsh, 1992b] work with conjunctive attribute-value languages with k features. In this case, $b_s = 1$ and $b_g = k$, so ITVS is still exponential in time when backtracking is needed to update s or g.

Incremental Non-Backtracking Focusing (INBF) of [Smith and Rosenbloom, 1990] does not have an exponential behavior by avoiding backtracking (as [Bundy *et al.*, 1985] did non-incrementally). INBF employs only one maximally general concept representation (*upper*). It specializes *upper* only when there exists a single consistent specialization, i.e., in case of specialization w.r.t. a near-miss (see [Winston, 1975]). INBF relies on having a sufficient number of positive examples to identify near-misses. As long as a negative example is not identified as a near-miss, it could be covered by *upper*, which therefore is not necessarily consistent. INBF can be extended to be consistent at any point by processing the remaining far-misses in the way CE does. This could also lead to a $\mathcal{G}$ -set exponential in size. The advantage over CE is, however, that the positive examples and the near-misses were processed first; in this way $\mathcal{G}$ is kept as small as possible.

[Hirsh, 1992b] only represents S and the set of all negative examples, thus avoiding exponential explosion for computation or storage of $\mathcal{G}$ in case of a tree-structured conjunctive language. Hirsh notes that the explosion is very much language dependent. For disjunctive languages for instance, S could be exponential as well. Hirsh also notes that in certain applications maximally general representations are preferred over maximally specific ones. Indeed, in the application of ITVS for the integration of Planning and Learning (see Chapter 6), taking an element of S as precondition could restrict the application of the action so much, that the agent would almost never apply it. Hence, a general approach should be symmetric in S and $\mathcal{G}$.

So the reason why [Smith and Rosenbloom, 1990] and [Hirsh, 1992b] are not exponential, is basically because they do not compute a consistent element of $\mathcal{G}$. If a maximally general consistent concept representation is needed, their algorithms will have to be extended, and will show an exponential behavior as well.

3.9 Compacting information elements in ITVS

3.9.1 Motivation

One of the major drawbacks of ITVS w.r.t. DI, is the fact that ITVS has to store all information elements, also when the sizes of S and $\mathcal{G}$ are smaller than the size of I. In this case DI would still have a better memory usage because the information given by all information elements can be compressed without any loss of information. In this section we will investigate how, in general, to reduce the memory needed by ITVS. The particular case when the size of S and $\mathcal{G}$ are smaller than the size of I, will be a special case. Because ITVS's worst case space complexity is linear in the number of information elements, we will reduce the number of information elements. In this section we will therefore on the one hand characterize redundant information elements and on the other hand introduce automatically generated information elements. These will replace two or more information elements without loss of information content, and thus reduce the number of information elements.

In general, incremental concept learning algorithms maintaining consistency with all information elements have to store *all* previous information elements as soon as any backtracking is involved. Exceptions are, for instance, DI because it searches breadth-first, and algorithms searching specific-to-general in a conjunctive tree-structure language, as Incremental Non-Backtracking Focusing [Smith and Rosenbloom, 1990]. Bundy et al. argue that for learning disjunctive concepts, all data will have to be stored anyway [Bundy *et al.*, 1985]. Hirsh even prefers a representation storing all negative examples together with S over storing S and $\mathcal{G}$ in case $\mathcal{G}$ can grow exponentially or can be infinite [Hirsh, 1992b].

One of the goals of concept learning is *compaction* of the information provided to the algorithm. Therefore, in all cases where all information elements have to be kept, preferably no redundant information should be stored. As [Sebag, 1994] and [Sebag and Rouveirol, 1994] do for negative examples in a conjunctive tree-structure, resp. first order logic language, we will remove redundant information elements in a language independent way by *partially ordering* them using $\preccurlyeq$, according to their information contents. We only have to store information elements maximal w.r.t. $\preccurlyeq$, while forgetting those with less information content. However, we will have to take care that ITVS does not lose any solutions, does not search previously discarded parts of the search space again, and keeps its most interesting properties. Ordering information elements using $\preccurlyeq$ will be possible because we extended $\preccurlyeq$ towards instance representations mutually in Definition 3.10.

Although we will develop this idea in the framework of the ITVS algorithm, we nevertheless argue that it has a much wider application potential. The theory is formulated independently from any concept learning algorithm or search strategy and independently from the chosen concept representation language. Identifying and removing redundant information elements could be used in any incremental algorithm that stores all information elements, and even in a preprocessing phase of a non-incremental concept learning algorithm, to reduce its actual processing time.

We will make sure that the main properties of ITVS will be maintained: the worst case space complexity, and the worst case time complexity of testing candidate solutions for maximal generality or maximal specificity should remain linear in the number of information elements. The global cost of extending the ITVS algorithm will be an increase in time complexity quadratic in the number of information elements. The gain is twofold: on the one hand storing less information elements will reduce the memory needed by the algorithm. On the other hand, if a search with a branching factor b is exponential in the number of information elements, reducing the number of information elements with a factor k, would reduce the time complexity with a factor b^k.

3.9.2 Redundant Information Elements

We first define the information elements we are going to focus on. Then we prove they are redundant.

Definition 3.44 (*s*-prunable and *g*-prunable)

- $i_1 \in I_s$ is s-prunable w.r.t $i_2 \in I_s$ iff
  - i_1 and i_2 are both positive lowerbounds such that $i_1 \preccurlyeq i_2$, or
  - i_1 and i_2 are both negative upperbounds such that $i_1 \preccurlyeq i_2$, or
  - i_1 is a negative upperbound and i_2 is a positive lowerbound such that $\neg(\ i_2 \preccurlyeq i_1\ )$.
- $i_1 \in I_s$ is s-prunable in I_s iff $\exists i_2 \in I_s$ such that i_1 is s-prunable w.r.t. i_2.
- $i_1 \in I_g$ is g-prunable w.r.t $i_2 \in I_g$ iff
  - i_1 and i_2 are positive upperbounds such that $i_2 \preccurlyeq i_1$, or
  - i_1 and i_2 are negative lowerbounds such that $i_2 \preccurlyeq i_1$, or
  - i_1 is a negative lowerbound and i_2 is a positive upperbound such that $\neg(\ i_1 \preccurlyeq i_2\ )$.
- $i_1 \in I_g$ is g-prunable in I_g iff $\exists i_2 \in I_s$ such that i_1 is g-prunable w.r.t. i_2.

Theorem 3.45 For any $i_1, i_2 \in I_g$ such that i_1 is g-prunable w.r.t i_2, or for any $i_1, i_2 \in I_s$ such that i_1 is s-prunable w.r.t i_2, and for any $c \in \mathcal{L}_C$, $i_2 \sim c$ implies $i_1 \sim c$.

Proof Actually, the proof is a straightforward application of the transitivity of $\preccurlyeq$ and the definition of $\sim$. First suppose $i_1, i_2 \in I_s$ and $i_2 \sim c$. In all three cases of i_1 being s-prunable w.r.t. i_2, this implies $i_1 \sim c$:

- $i_1 \preccurlyeq i_2$ and $i_2 \preccurlyeq c$ implies $i_1 \preccurlyeq c$.
- $i_1 \preccurlyeq i_2$ and $\neg(c \preccurlyeq i_2)$ implies $\neg(c \preccurlyeq i_1)$.
- $\neg(i_2 \preccurlyeq i_1)$ and $i_2 \preccurlyeq c$ implies $\neg(c \preccurlyeq i_1)$.

If $i_1, i_2 \in I_g$ and $i_2 \sim c$, in all three cases of i_1 being g-prunable w.r.t. i_2, this implies $i_1 \sim c$:

- $i_2 \preccurlyeq i_1$ and $c \preccurlyeq i_2$ implies $c \preccurlyeq i_1$.
- $i_2 \preccurlyeq i_1$ and $\neg(i_2 \preccurlyeq c)$ implies $\neg(i_1 \preccurlyeq c)$.
- $\neg(i_1 \preccurlyeq i_2)$ and $c \preccurlyeq i_2$ implies $\neg(i_1 \preccurlyeq c)$.

□

Note that the six cases in the proof of Theorem 3.45 are the only six possibilities of instantiating $\{ A , B , C \}$ with a permutation of $\{ i_1 , i_2 , c \}$ in the transitivity rule

$$A \preccurlyeq B \text{ and } B \preccurlyeq C \text{ implies } A \preccurlyeq C$$

and the equivalent rules

$$\neg(A \preccurlyeq C) \text{ and } B \preccurlyeq C \text{ implies } \neg(A \preccurlyeq B)$$

and

$$A \preccurlyeq B \text{ and } \neg(A \preccurlyeq C) \text{ implies } \neg(B \preccurlyeq C),$$

such that i_2 does not appear in the consequence of the rule (i.e., such that the rule concludes something about i_1).

As a consequence of Theorem 3.45, we do not have to store *all* information elements, but rather only the non s-prunable ones and the non g-prunable ones. Whenever we detect that a previously stored s-bound i_1 is s-prunable w.r.t. a newly provided one i_2, we will replace i_1 by i_2 in I_s.

In ITVS, a naive method to update B_s would be to reprocess all s-bounds with an index in I_s larger than i_1's. However, this could lead to computing and generalizing previously discarded elements of $\mathcal{L}_C$. Therefore, we will try to *update* all alternatives on B_s, instead of recomputing them, while respecting B_s's invariants.

A dual argument holds for B_g.

So far we assumed all information elements involved were *provided* to the concept learning algorithm. However, under certain conditions new information elements with an information content equivalent to the provided ones can be *automatically created*. Moreover, the automatically created information elements will enforce that earlier provided information elements will become s-prunable or g-prunable, so that less information elements have to be stored. We will now describe some conditions under which such information elements can be automatically generated.

Lemma 3.46 Given $c \in \mathcal{L}_C$, and

- two positive lowerbounds i_1 and i_2 such that $mub(i_1 , i_2) = \{ i \}$, or

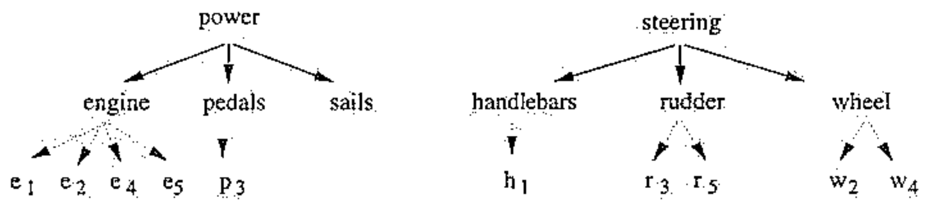


Figure 3.17 Two tree-structure taxonomies

- two positive upperbounds i_1 and i_2 such that $mlb(\ i_1\ ,\ i_2\ ) = \{\ i\ \}$,

then $c \sim i$ iff ($c \sim i_1$ and $c \sim i_2$).

Proof First consider two positive lowerbounds with $mub(\ i_1\ ,\ i_2\ ) = \{\ i\ \}$. On the one hand each $c \in \mathcal{L}_C$ more general than i will also be more general than i_1 and more general than i_2. On the other hand, each element of $\{\ c \in \mathcal{L}_C \mid i_1 \preccurlyeq c$ and $i_2 \preccurlyeq c\ \}$ is more general than a minimal element of this set, which is exactly $mub(\ i_1\ ,\ i_2\ ) = \{\ i\ \}$.

Now consider two positive upperbounds with $mlb(\ i_1\ ,\ i_2\ ) = \{\ i\ \}$. On the one hand each $c \in \mathcal{L}_C$ more specific than i will also be more specific than i_1 and more specific than i_2. On the other hand, each element of $\{\ c \in \mathcal{L}_C \mid c \preccurlyeq i_1$ and $c \preccurlyeq i_2\ \}$ is more specific than a maximal element of this set, which is exactly $mlb(\ i_1\ ,\ i_2\ ) = \{\ i\ \}$. □

This means that whenever two positive lowerbounds have only one minimal upperbound they may be *replaced* by this one minimal upperbound without loss of information, and whenever two positive upperbounds have only one maximal lowerbound, they may be replaced by this one maximal lowerbound. In a conjunctive tree-structure attribute-value language [Smith and Rosenbloom, 1990], where each tree is augmented with a bottom element (because of Constraint 3.20), the minimal upperbound of two positive lowerbounds and the maximal lowerbound of two positive upperbounds are unique. This is because the tree-structure augmented with a bottom element forms a lattice, and the direct product of all these lattices is also a lattice [Birkhoff, 1979].

Example 3.47 Figure 3.17 shows two tree-structure taxonomies, adapted from the *vehicle* example of [Murray, 1987a]. The set of concept representations is the direct product of the first taxonomy with the second one (cf. Section 3.7). A concept representation consists of a non-leaf of each taxonomy, e.g., [*engine*, *steering*] represents the concept of all vehicles with an engine, and with any steering mechanism. Instance representations consist of one leaf of each taxonomy with the same index, e.g., $[e_1, h_1]$ is the instance consisting of engine e_1 and handlebars h_1. To introduce a bottom element we assume in both trees a bottom element $\perp$ more specific than each non-leaf of the taxonomy[11]. The bottom concept representation of the direct product is then $[\perp, \perp]$.

[11]These bottom elements are not shown in Figure 3.17 in order not to overload the figure.

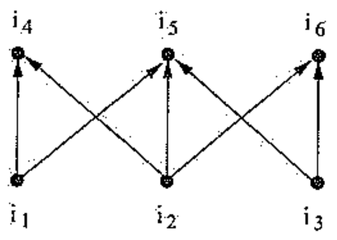


Figure 3.18 Taking minimal upperbounds is order-dependent

$\preccurlyeq$ is defined as in Section 3.7:

$$[X_1, X_2] \preccurlyeq [Y_1, Y_2] \text{ iff } (X_1 \preccurlyeq Y_1 \text{ and } X_2 \preccurlyeq Y_2).$$

Suppose $[e_1, h_1]$ and $[e_2, w_2]$ are positive lowerbounds. Then the unique minimal upperbound of both is $[engine, steering]$. Suppose $[power, handlebars]$ and $[pedals, handlebars]$ are both positive upperbounds, then $[pedals, handlebars]$ is their unique maximal lowerbound.

◇

In Chapter 5 we will show that also in Inductive Logic Programming using θ-subsumption the minimal upperbound and the maximal lowerbound are always unique.

Theorem 3.48

- Given two positive lowerbounds i_1 and i_2 such that $mub(\ i_1\ ,\ i_2\ ) = \{\ i\ \}$. Then i_1 is s-prunable w.r.t. i.
- Given two positive upperbounds i_1 and i_2 such that $mlb(\ i_1\ ,\ i_2\ ) = \{\ i\ \}$. Then i_1 is g-prunable w.r.t. i.

In both cases we also have that: given a set I of information elements such that $i_1 \in I$, the set of concept representations consistent with $I \cup \{\ i\ \}$ is the set of concept representations consistent with $I \cup \{\ i_2\ \}$.

Proof This is an immediate consequence of Lemma 3.46, and of the definition of s-prunable and g-prunable. □

In particular, in the case of a positive lowerbound i_2, for instance, i can be provided to ITVS instead of i_2 without losing any solutions. Then, because i_1 is s-prunable w.r.t. i, i_1 becomes redundant. Whenever a minimal upperbound i replaces a positive lowerbound i_1, all other s-bounds will have to be checked whether they are not s-prunable, or whether they have more than one minimal upperbound with i. Unfortunately, the result depends on the order in which the information elements are provided.

Example 3.49 Suppose i_1, i_2 and i_3 are positive lowerbounds (see Figure 3.18). Suppose

- $mub(\ i_1\ ,\ i_2\ ) = \{\ i_4\ ,\ i_5\ \}$,
- $mub(\ i_2\ ,\ i_3\ ) = \{\ i_5\ ,\ i_6\ \}$, and
- $mub(\ i_1\ ,\ i_3\ ) = \{\ i_5\ \}$.

If the positive lowerbounds are provided in the order i_1, i_2, i_3, then i_1 and i_3 have a unique minimal upperbound i_5. Thus, i_5 can be provided to ITVS instead of i_3. Then i_1 is s-prunable, and can be replaced by i_5. But then i_2 is also s-prunable and can be removed from I. However, suppose instead that $mub(\ i_1\ ,\ i_2\ ) = \{\ i_4\ \}$ and $mub(\ i_2\ ,\ i_6\ ) = \{\ i_6\ \}$. If the order of the positive lowerbounds is i_1, i_2, i_3, then i_4 and i_3 will remain in I. If the order of the positive lowerbounds is i_2, i_3, i_1, then i_6 and i_1 will remain in I. This shows the resulting set of information elements is order dependent.

◇

Negative information elements cannot be generalized or specialized in the same way. However, there is a special kind of negative information elements that can be transformed to positive information elements.

Definition 3.50 (Lower and upper near-miss)

- A lower near-miss w.r.t. $c \in \mathcal{L}_I \cup \mathcal{L}_C$ is a negative lowerbound i_n such that $\{\ x \in mgs(\ \top\ ,\ i_n\ ) \mid c \preccurlyeq x\ \}$ is a singleton $\{\ i_u\ \}$.
- An upper near-miss w.r.t. $c \in \mathcal{L}_I \cup \mathcal{L}_C$ is a negative upperbound i_n such that $\{\ x \in msg(\ \bot\ ,\ i_n\ ) \mid x \preccurlyeq c\ \}$ is a singleton $\{\ i_l\ \}$.

By definition

$$\{\ x \in mgs(\ \top\ ,\ i_n\ ) \mid c \preccurlyeq x\ \} = Max\ \{\ x \in \mathcal{L}_C \mid \neg(\ i_n \preccurlyeq x\ ) \text{ and } c \preccurlyeq x\ \},$$

and

$$\{\ x \in msg(\ \bot\ ,\ i_n\ ) \mid x \preccurlyeq c\ \} = Min\ \{\ x \in \mathcal{L}_C \mid \neg(\ x \preccurlyeq i_n\ ) \text{ and } x \preccurlyeq c\ \}.$$

Given a lower near-miss i_n w.r.t. a positive lowerbound i_p, the target concept must be more general than i_p to be consistent with i_p, but also more specific than the corresponding i_u to be consistent with i_n. In other words, i_u is a positive upperbound constraining the search space in the same way as i_n does. If we replace each lower near-miss w.r.t. a positive lowerbound by its equivalent positive upperbound, we can apply Theorem 3.48 when appropriate. This situation is illustrated in Figure 3.19. Two of the three elements in the upper half of the figure that are more general than i_p are also more general than i_n. This is not the case for i_u; consequently i_n is a lower near-miss and i_u is a positive upperbound.

Similarly, given an upper near-miss i_n w.r.t. a positive upperbound i_p with corresponding i_l, the target concept must be more specific than i_p and more general than i_l. In this case i_l is a positive lowerbound constraining the search space in the same way as i_n does. If we replace each upper near-miss w.r.t. a positive upperbound by its equivalent positive lowerbound, we can again apply Theorem 3.48 when appropriate.

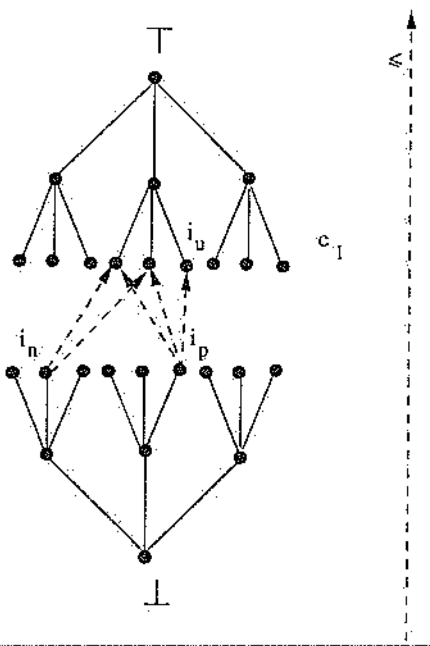


Figure 3.19 Replacing a lower near-miss by a positive upperbound

Example 3.51 Consider again the taxonomies of Figure 3.17, and the concept representation language introduced in Example 3.49. Given two negative lowerbounds $[p_3, r_3]$ and $[e_4, w_4]$, and a positive lowerbound $[e_5, r_5]$. Then both $[p_3, r_3]$ and $[e_4, w_4]$ are near-misses w.r.t. $[e_5, r_5]$. The positive upperbound $[engine, steering]$ corresponds to $[p_3, r_3]$, and the positive upperbound $[power, rudder]$ corresponds to $[e_4, w_4]$. Their unique maximal lowerbound is $[engine, rudder]$. Consequently, the two negative lowerbounds can be replaced by the one positive upperbound $[engine, rudder]$. ◇

The definition of a lower near-miss is consistent with the usual definition of a near-miss[12] in a conjunctive tree-structure language, since $mgs(\top, i_n)$ will only be a singleton if i_n and s differ in only one attribute. Theorem 3.48 explains that providing the only consistent maximally specific concept representation s as positive lowerbound is equivalent to providing all actual positive lowerbounds. In this case all lower near-misses can be replaced by exactly one positive upperbound, since the corresponding positive upperbounds have only one maximal lowerbound (mlb). This corresponds to the result of [Smith and Rosenbloom, 1990].

In Section 3.9.3, we will also need the Lemma 3.52 to Lemma 3.55 to prove correctness of the given algorithms. These lemmas express that the generalization operators *mub* and *msg*, and the specialization operators *mlb* and *mgs* in a sense *preserve* the relation $\preccurlyeq$. Readers that wish to skip technical sections, can also skip these lemmas. Figure 3.20

[12]The notion of a near-miss was introduced by [Winston, 1975].

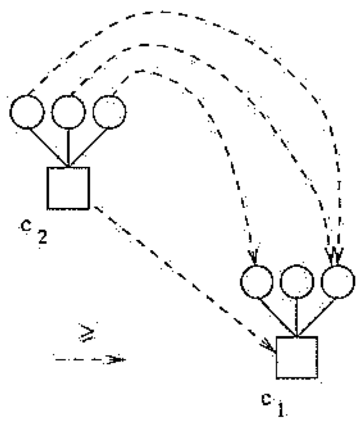


Figure 3.20 Preservation of $\preccurlyeq$ by *mub*

illustrates Lemma 3.52. c_1 and c_2 are represented by boxes; their minimal upperbounds w.r.t. i are represented by circles.

Lemma 3.52 Given $c_1, c_2 \in \mathcal{L}_C$ with $c_1 \preccurlyeq c_2$ and a positive lowerbound i:

$$\forall x_2 \in mub(\ c_2\ ,\ i\ ) : \exists x_1 \in mub(\ c_1\ ,\ i\ ) : x_1 \preccurlyeq x_2.$$

Proof Assume the conclusion is false, i.e.,

$$\exists x_2 \in mub(\ c_2\ ,\ i\ ) : \forall x_1 \in mub(\ c_1\ ,\ i\ ) : \neg(\ x_1 \preccurlyeq x_2\ ).$$

Since $c_2 \preccurlyeq x_2$, we also have $c_1 \preccurlyeq x_2$. Because of Constraint 3.18 (the Boundedness Constraint), the set $V = \{\ x \mid c_1 \preccurlyeq x \text{ and } i \preccurlyeq x\ \}$ is bounded. Moreover, $x_2 \in V$. Consequently, there exists a $x' \in V$ such that $x' \preccurlyeq x_2$ and x' is minimal in V. Consequently $x' \in mub(\ c_1\ ,\ i\ )$, which contradicts the assumption. □

Lemma 3.53 Given $c_1, c_2 \in \mathcal{L}_C$ with $c_1 \preccurlyeq c_2$ and a negative upperbound i:

$$\forall x_2 \in msg(\ c_2\ ,\ i\ ) : \exists x_1 \in msg(\ c_1\ ,\ i\ ) : x_1 \preccurlyeq x_2.$$

Proof Assume the conclusion is false, i.e.,

$$\exists x_2 \in msg(\ c_2\ ,\ i\ ) : \forall x_1 \in msg(\ c_1\ ,\ i\ ) : \neg(\ x_1 \preccurlyeq x_2\ ).$$

Since $c_2 \preccurlyeq x_2$, we also have $c_1 \preccurlyeq x_2$. Because of Constraint 3.18, the set $V = \{\ x \mid c_1 \preccurlyeq x \text{ and } \neg(\ x \preccurlyeq i\ )\ \}$ is bounded. Moreover, $x_2 \in V$. Consequently, there exists a $x' \in V$ such that $x' \preccurlyeq x_2$ and x' is minimal in V. Consequently $x' \in msg(\ c_1\ ,\ i\ )$, which contradicts the assumption. □

Lemma 3.54 Given $c_1, c_2 \in \mathcal{L}_C$ with $c_1 \preccurlyeq c_2$ and a positive upperbound i:

$$\forall x_1 \in mlb(\ c_1\ ,\ i\ ) : \exists x_2 \in mlb(\ c_2\ ,\ i\ ) : x_1 \preccurlyeq x_2.$$

Proof Assume the conclusion is false, i.e.,

$$\exists x_1 \in mlb(\ c_1\ ,\ i\ ) : \forall x_2 \in mlb(\ c_2\ ,\ i\ ) : \neg(\ x_1 \preccurlyeq x_2\ ).$$

Since $x_1 \preccurlyeq c_1$, we also have $x_1 \preccurlyeq c_2$. Because of Constraint 3.18, the set $V = \{\ x \mid x \preccurlyeq c_2 \text{ and } x \preccurlyeq i\ \}$ is bounded. Moreover, $x_1 \in V$. Consequently, there exists a $x' \in V$ such that $x_1 \preccurlyeq x'$ and x' is maximal in V. Consequently $x' \in mlb(\ c_2\ ,\ i\ )$, which contradicts the assumption. □

Lemma 3.55 Given $c_1, c_2 \in \mathcal{L}_C$ with $c_1 \preccurlyeq c_2$ and a negative lowerbound i:

$$\forall x_1 \in mgs(\ c_1\ ,\ i\ ) : \exists x_2 \in mgs(\ c_2\ ,\ i\ ) : x_1 \preccurlyeq x_2.$$

Proof Assume the conclusion is false, i.e.,

$$\exists x_1 \in mgs(\ c_1\ ,\ i\ ) : \forall x_2 \in mgs(\ c_2\ ,\ i\ ) : \neg(\ x_1 \preccurlyeq x_2\ ).$$

Since $x_1 \preccurlyeq c_1$, we also have $x_1 \preccurlyeq c_2$. Because of Constraint 3.18, the set $V = \{\ x \mid x \preccurlyeq c_2 \text{ and } \neg(\ i \preccurlyeq x\ )\ \}$ is bounded. Moreover, $x_1 \in V$. Consequently, there exists a $x' \in V$ such that $x_1 \preccurlyeq x'$ and x' is maximal in V. Consequently $x' \in mgs(\ c_2\ ,\ i\ )$, which contradicts the assumption. □

3.9.3 The algorithm [T]

SUMMARY: this section describes how to adapt ITVS for handling s-prunable and g-prunable information elements. In particular this approach imposes two extra invariants on ITVS's datastructures: I_s does not contain s-prunable elements, and I_g does not contain g-prunable elements.

In the previous section we have determined which information elements are s-prunable or g-prunable, and hence redundant. We will now modify the ITVS algorithm (Algorithm 3.3) such that no redundant information elements are stored. We will only discuss the case of adding an s-bound; as usual the case of a g-bound is dual.

The following two extra invariants will enforce that no redundant information elements are stored:

- **Invariant 3.9.1.** (Unprunability of I_s) No element in I_s is s-prunable, and
- **Invariant 3.9.2.** (Unprunability of I_g) No element in I_g is g-prunable.

Note that Invariant 3.9.1 and Invariant 3.9.2 are expressed solely in terms of I_s, resp. I_g, and are therefore independent of any search strategy or concept language: they only constrain the set of information elements that is stored, and can therefore be used in any system that stores all information elements.

Given these invariants, the question is how to update ITVS's datastructures once a redundant information element is detected?

Adding an s-bound could only imply other s-bounds to be discarded, and this affects the way B_s is built up. Therefore we only have to replace the parts of the algorithm where I_s and B_s are adapted. In order not to interfere with the call to `specialize` (see Step 3.14 in Algorithm 3.3) I_s is updated as before. Replacing the call in Step 3.15 in Algorithm 3.3 by a call $manage\_s\_bound(\ s\ ,\ B_s\ ,\ i\ )$ will be the only change.

```
procedure manage_s_bound( $s$: concept; $B_s$: stack; $i$: info ) returns concept, stack
    if $i$ is an upper near-miss with corresponding positive lowerbound $i_l$

    then $i := i_l$  {3.31}
    $n_c := 1$
    while $n_c \leq n_s$ and then $I_s[n_c]$ is not $s$-prunable w.r.t. $i$ and
        $i$ is not $s$-prunable w.r.t. $I_s[n_c]$  {3.32}
         do if $i$ and $I_s[n_c]$ are positive lowerbounds and $mub( i , I_s[n_c] ) = \{ i_p \}$ {3.33}
            then $i := i_p$  {3.34}
                 $n_c := 1$  {3.35}
            else  $n_c := n_c + 1$ {3.36}
    endwhile
    if $n_c = n_s$  {3.37}
    then $s, B_s := generalize( s , B_s , n_s - 1 )$  {3.38}
    else  $n_s := n_s - 1$  {3.39}
          if $I_s[n_c] \preccurlyeq i$
          then $I_s[n_c] := i$  {3.40}
               $s, B_s, ind := generalize\_stack( n_c , s , B_s , i )$  {3.41}
               $s, B_s := generalize( s , B_s , ind )$
    return $s, B_s$
endproc
```

Algorithm 3.7 Managing s-bounds in ITVS

Algorithm 3.7 first checks whether is an upper near-miss with corresponding positive lowerbound i_l, i.e., it checks whether there exists a positive upperbound i_p in I_g, such that the set $\{ x \in msg(\perp , i) \mid x \preccurlyeq c \}$ is a singleton $\{ i_l \}$. If this is the case, the positive lowerbound i_l constrains the search in the same way as i does. Therefore i_l is processed instead of i, by assigning i_l to i (Step 3.31).

Then Algorithm 3.7 checks whether Invariant 3.9.1 on I_s is still fulfilled. This is done by checking for each of the existing s-bounds $I_s[n_c]$ whether $I_s[n_c]$ is s-prunable w.r.t. i or i is s-prunable w.r.t. $I_s[n_c]$ (Step 3.32). Also, if i and $I_s[n_c]$ are positive lowerbounds, Step 3.33 checks whether they have a unique minimal upperbound i_p. If they have, Theorem 3.48 allows to process i_p instead of i. Therefore i_p is assigned to i (Step 3.34). At least one s-bound will now be s-prunable w.r.t. i, namely $I_s[n_c]$. However, other s-bounds in I_s might be s-prunable w.r.t. i_p as well, therefore the check for s-prunable elements is restarted from the front of I_s (Step 3.35). If none of the existing s-bounds is s-prunable, i itself is not s-prunable, and Theorem 3.48 could not be applied (see Step 3.37), ITVS continues as before (i.e., with the call to `generalization` in Step 3.15 in Algorithm 3.3). If i is found to be s-prunable w.r.t. $I_s[n_c]$, it is removed from I_s (Step 3.39) and no call to `generalization` is needed. Otherwise, $I_s[n_c]$ is s-prunable w.r.t. i. In that case i is also removed from I_s, and then replaces $I_s[n_c]$ (Step 3.40). Then B_s must be updated to fulfill Invariant 3.6.3 and Invariant 3.6.5 (soundness and completeness of B_s; see Step 3.41). This is explained in Algorithm 3.8. The result is a new maximally specific concept representation s, the updated B_s, and an index ind such that s is consistent with all g-bounds and with the first ind s-bounds. Finally, a new maximally specific concept representation consistent with *all* information elements must be computed using the procedure `generalize`. This call will fulfill Invariant 3.6.1 (maximally specificity of s), and keep Invariant 3.6.3 and Invariant 3.6.5 unchanged. In Algorithm 3.7 and Algorithm 3.8 neither g, nor B_g, nor I_g, nor n_g are changed, so that Invariant 3.6.4 and Invariant 3.6.6 (soundness and completeness of B_g) and the maximality of g (Invariant 3.6.2) are preserved all the time.

In general terms `generalize_stack` works as follows: it first pops the choicepoints of B_s that are to be generalized from B_s and pushes them onto a temporary stack B_h. It then generalizes these choicepoints one by one w.r.t. the new information element i. The result is a new B_s fulfilling Invariant 3.6.3 and Invariant 3.6.5. Simultaneously, all information elements i' s-prunable w.r.t. i are removed from I_s (by shifting the information elements following i' towards the front). We will now discuss `generalize_stack` in more detail.

Let us first discuss the meaning of the local variables used in `generalize_stack`, and the way they are initialized.

- B_s and B_h: since I_s has been changed in position n_c, the alternatives to be generalized are the ones with index greater than or equal to n_c. Therefore the stack B_s is first split in two stacks, assigned to B_s and B_h. The former initially contains all choicepoints of B_s with index less than n_c and in the same order as on B_s. The latter contains all choicepoints of B_s with index greater than or equal to n_c, and in *reversed order* w.r.t. B_s. Also, the bottom element of B_h must have s as s_{ind}, in order to handle s as any other s_{ind}. This implies that B_h initially is not empty. The initialization of these variables is actually done in procedure `init_gen_stack` (see Algorithm 3.9).

- new_s_{ind}: for each ind, new_s_{ind} will replace s_{ind} on the generalized B_s. At the end, new_s_{ind} will also be returned as the resulting s. Note that new_s_{ind} will always be more general than s_{ind} of the top choicepoint of B_s;

- new_{ind}: new_{ind} is the index in I_s up to where the s-bounds of the original I_s are checked for being s-prunable, and up to where new_s_{ind} is consistent with all s-bounds.

```
procedure generalize_stack ( $n_c$: index; $s$: concept; $B_s$: stack; $i$: info )
          returns concept,stack,index
   $B_s$, $B_h$, $new\_s_{ind}$, $new\_ind$ := $init\_gen\_stack( B_s , n_c )$
   $n'_s := n_c$
   $new\_ind := n_c$
   $prune\_B_h := false$
   repeat
        $ind$, $s_{ind}$, $alt_{ind}$, $B_h := pop( B_h )$
        $gens_1 := generalizations( s_{ind} , i )$ {3.42}
        $gens_1$ := select all c from $gens_1$ {3.43}
               with $all\_consistent( c , I_g , n_g )$ and $max\_specific( c , B_s )$ and
               ( $\neg\exists c' \in alt_{ind} : c' \preccurlyeq c$ ) and $exists\_more\_general( c , s , B_h )$
        $gens_2 := \bigcup_{a \in alt_{ind}} generalizations( a , i )$ {3.44}
        $gens_2$ := select all c from $gens_2$ {3.45}
               with $all\_consistent( c , I_g , n_g )$ and $max\_specific( c , B_s )$ and
               ( $\neg\exists c' \in gens_1 : c' \preccurlyeq c$ ) and ( $\neg\exists c' \in gens_2 : c' \prec c$ )
        if $gens_1 \neq \emptyset$
        then $n'_s, I_s := shift( n'_s , new\_ind , ind , i , I_s )$ {3.46}
        else $prune\_B_h := true$
             $n'_s, I_s := shift( n'_s , new\_ind , n_s , i , I_s )$ {3.47}
        $new\_ind$, $new\_s_{ind}$, $B_s := select\_alternative( gens_1 \cup gens_2 , B_s , ind )$ {3.48}
   until $B_h = \emptyset$ or $prune\_B_h$ {3.49}
   $n_s := n'_s$ {3.50}
   return $new\_s_{ind}$, $B_s$, $new_{ind}$
```

Algorithm 3.8 Generalizing the part of B_s with index larger than or equal to n_c

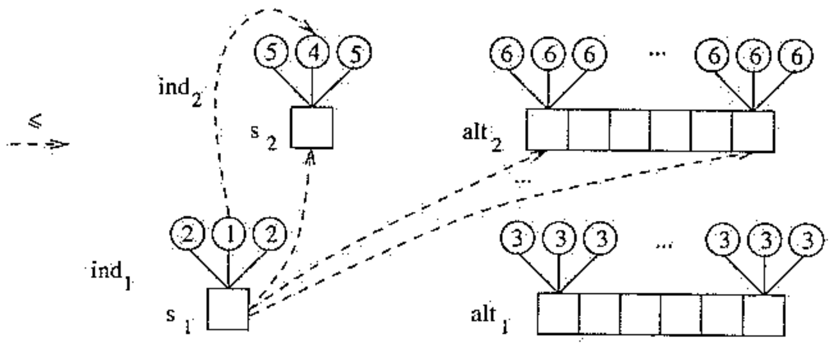


Figure 3.21 Generalizing B_s

- n'_s: n'_s is the index up to where I_s contains the non s-prunable information elements of $I_s[1..n_c]$. Initially, $n'_s = n_c$, since $I_s[n_c]$ was the first information element in I_s s-prunable w.r.t. i. Elements with index larger than n_c are not yet checked.
- *prune_B*$_h$: *prune_B*$_h$ will be true iff the rest of B_h cannot be generalized consistently. It is initialized to *false*.
- $gens_1$ and $gens_2$: these variables are used to contain the generalizations of s_{ind}, resp. alt_{ind}.

We will now explain what happens inside the repeat-loop. As always we explain the algorithm in general, but to fix the attention, one can take a look at what happens on Figure 3.21. Figure 3.21 shows two consecutive choicepoints (ind_1 , s_1 , alt_1) and (ind_2 , s_2 , alt_2) of B_s. As for any two consecutive choicepoints we have that $ind_1 < ind_2$, s_2 and the elements of alt_2 are more general than s_1, maximally specific and consistent with all g-bounds and with $I_s[1]$ to $I_s[ind_2]$. The squares in Figure 3.21 represent the state of the two choicepoints before the call to `generalize_stack`. The dashed arrows represent $\preccurlyeq$. The circles on the figure are generalizations of the square they are connected to. The numbers inside the circles are labels. During the explanation of the repeat-loop, suppose that choicepoint (ind_1 , s_1 , alt_1) has already been generalized: the new s_1 is labeled 1, the elements of the new alt_1 are labeled 2 and 3, and suppose that choicepoint (ind_2 , s_2 , alt_2) is popped from B_h.

If the procedure `generalize_stack` fails to generalize B_s, it halts ITVS. We will prove that in case `generalize_stack` does not fail to generalize B_s the following three invariants hold after each cycle in the repeat-loop:

A. *new*_s_{ind} is more general than each s_{ind} on B_s;

B. Invariant 3.6.3 (the soundness of B_s);

C. if B_h is not empty, and *prune_B*$_h$ is false, each $c \in \mathcal{L}_C$ consistent with I is more general than s, or more general than some alternative on B_s, or more general than some alternative on B_h; otherwise, each $c \in \mathcal{L}_C$ consistent with I is more general than *new*_s_{ind}, or more general than some alternative on B_s,

Invariant B holds before the repeat-loop because `init_gen_stack` left all choicepoints with index smaller than n_c on B_s in the same order. Invariant C also holds before the repeat-loop: B_h is not empty, and *prune_B*$_h$ is false. Moreover, from Invariant 3.6.5 before B_s is split in two parts follows that each concept representation consistent with the *original* set I (i.e., without i) is more general than s or more general than some alternative a on the original B_s with index ind_a. Consequently, each concept representation consistent with the *new* I (i.e., where i replaces $I_s[n_c]$) is also more general than s or than some alternative on the *original* B_s. Therefore, after splitting the *original* B_s in two parts, invariant C is fulfilled. Also, by construction of B_h, and because B_h is only popped inside the repeat-loop, the relations between the choicepoints on B_h induced by Invariant 3.6.3 remain valid.

First s_2 is generalized (Step 3.42): $gens_1$ should contain all maximally specific consistent generalizations of s_2, not reachable from an alternative on B_s and not yet explored or discarded before. To compute $gens_1$ all minimal generalizations of s_2 are computed: if i is a positive lowerbound it is the set $mub(\ s_2\ ,\ i\ )$; if i is a negative upperbound it is the set $msg(\ s_2\ ,\ i\ )$. On Figure 3.21 these generalizations are labeled 4 and 5. Of this set of generalizations, generalizations that are

1. not consistent, or

2. more general than some alternative on B_s, or

3. more general than some alternative in alt_2, or

4. not more general than s or than an alternative on B_h

are *removed* (Step 3.43). Condition 2 and Condition 3 implement, as in ITVS, a check for maximal specificity as well as an optimal generalization operator. This shows the use of an optimal generalization operator is still possible in the extended ITVS. Condition 4 is needed for the following reason: first note that all elements in $gens_1$ are generalizations of s_2. Also note that all alternatives more general than s_2 and still to be explored are the alternatives on B_h, together with s. Therefore Condition 4 rejects those generalizations not more general than s or than some alternative on B_h. Further on we prove that Invariant 3.6.5 is established, i.e., that this step does not remove any solutions.

The remaining elements are assigned to $gens_1$. In $gens_1$ all elements are more general than new_s_{ind}, since new_s_{ind} is a generalization of s_1 (see invariant A of the repeat-loop), since each minimal generalization of s_2 is more general than some generalization of s_1 (see Lemma 3.52 and Lemma 3.53) and since the minimal generalizations more general than the elements labeled 2 (which are on B_s already) are not selected for $gens_1$.

Then all elements of alt_2 are generalized (Step 3.44): $gens_2$ should contain all maximally specific consistent generalizations of the elements of alt_2, not reachable from another alternative on B_s. To compute $gens_2$ all minimal generalizations are computed: if i is a positive lowerbound it is the union of all sets $mub(\ a\ ,\ i\ )$, with $a \in alt_2$; if i is a negative upperbound it is the union of all sets $msg(\ a\ ,\ i\ )$, with $a \in alt_2$. On Figure 3.21 this union consists of the circles labeled 6. Of this union, generalizations that are

1. not consistent, or

2. more general than some other alternative on B_s, or

3. more general than some other alternative in $gens_1$, or

4. more general than another generalization in $gens_2$

are removed (Step 3.45). Condition 4 removes the elements of $gens_2$ not maximally specific w.r.t. $gens_2$. Condition 2 and Condition 3 again implement maximal specificity and an optimal generalization operator. The remaining elements are assigned to $gens_2$. Like for $gens_1$, all elements of $gens_2$ are more general than new_s_{ind}.

If $gens_1$ is empty, no generalization of s_2 consistent with i exists that is not more general than an alternative on B_s. Therefore it is not necessary to generalize the other alternatives on B_h, since they are all generalizations of s_2, so $prune\_B_h$ is set to $true$. Of all information elements $I_s[new\_ind+1]$ up to $I_s[n_s]$ only the ones not s-prunable are shifted in I_s towards the front, thus removing the s-prunable ones (Step 3.47).

Otherwise, if $gens_1$ is not empty, only all non s-prunable information elements of $I_s[new\_ind+1]$ up to $I_s[ind_2]$ are shifted in I_s towards the front (Step 3.46). Then `select_alternative` (see Algorithm 3.6) selects new_s_{ind}:

- from $gens_1$, if $gens_1$ is not empty;
- from $gens_2$, if $gens_1$ is empty, and $gens_2$ is not[13];
- from the top choicepoint of B_s, if $gens_1$ and $gens_2$ are both empty, and B_s is not.

Otherwise, if $gens_1$, $gens_2$ and B_s are empty, `select_alternative` fails, and halts ITVS. This means there exists neither a generalization of s nor of any alternative on B_s consistent with i. Together with new_s_{ind} `select_alternative` returns the number of s-bounds new_s_{ind} is consistent with, new_{ind}, and the updated B_s, fulfilling invariant B of the repeat-loop: if a choicepoint has been added to B_s, its index ind_2 is the same as the old choicepoint; new_s_{ind} is consistent with all g-bounds and ind_2 s-bounds; all elements of $gens_1 \cup gens_2$ are more general than s_I, and not more general than some alternative on B_s. Invariant A of the repeat-loop also remains fulfilled. We will now show that invariant C of the repeat-loop is also preserved, i.e., that no solutions are lost during the update of B_s.

We will first consider the case where a concept representation c consistent with the new I is more general than some alternative on B_s. Since B_s is only changed by pushing something on top, this case is trivial. The same is true when c is more general than some alternative on B_h which is not in the top choicepoint of B_h, since only one choicepoint is popped from B_h during one cycle of the repeat-loop.

Otherwise, suppose c is more general than s. Since the bottom choicepoint on B_h always has s as s_{ind}, s is generalized as in Step 3.43, after popping the last choicepoint from B_h. Now c must be more general than a minimal generalization c' of s w.r.t. i, because of Constraint 3.18 (the Boundedness constraint). However, c' could be discarded from $gens_1$, because of Step 3.43. Even then, c will still be more general than *some* alternative on B_s (see Figure 3.22: the boxes represent concept representations not yet generalized w.r.t. i, the circles represent those already generalized):

- if c' is not consistent with all g-bounds, then c would not have been consistent with all g-bounds;
- if c' is more general than another alternative a_1 on B_s, c is also more general than a_1;
- if c' is more general than an element a_2 of alt_{ind}, c is more general than a_2; how elements of alt_{ind} are handled is discussed hereafter;
- the last case is not applicable, since c' is more general than s.

[13]It is important to select from $gens_1$ first, because if $gens_1$ is not empty, the generalizations on B_h still have to be generalized w.r.t. i. They will be generalizations of an element of $gens_1$. This is not the case for $gens_2$. With the implementation of `select_alternative` as in Algorithm 3.6, this order is preserved.

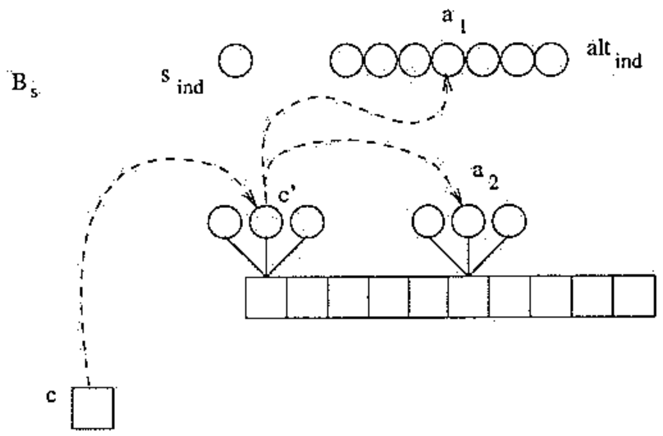


Figure 3.22 Proving completeness when $s \preccurlyeq c$

If c' is not discarded, c is more general than an element of $gens_1$ (of which all elements are put on B_s by `select_alternative`), except when c' is chosen as new_s_{ind}. However, then B_h is empty, so that invariant C of the repeat-loop is still fulfilled.

Now assume c does not fit any of the above cases, but c is more general than some a in the top choicepoint of B_h. In that case c must be more general than a minimal generalization c' of a in Step 3.44 (because of Constraint 3.18; the Boundedness Constraint). c' could be discarded from $gens_2$ for four reasons (Step 3.45), but in any case c is more general than *some* element in $gens_1$, in $gens_2$ or on B_s(see Figure 3.23):

- if c' is not consistent with all g-bounds, then c would not have been consistent with all g-bounds;
- if c' is more general than another alternative a_1 on B_s, c is also more general than a_1;
- if c' is more general than an element a_2 in $gens_1$ not discarded from $gens_1$, c is also more general than a_2. Since all elements in $gens_1$ are more general than s or than an alternative on B_h, this would imply that c is more general than s, or than an alternative on B_h. However, we assumed that this was not the case;
- if c' is more general than a minimal generalization a_3 of another element of alt_{ind} which is not discarded, c is more general than a_3.

If c' is not discarded, c is more general than an element of $gens_2$ (of which all elements are put on B_s by `select_alternative`), except when c' is chosen as new_s_{ind}. In the latter case, $prune\_B_h$ is false, so that invariant C of the repeat-loop is still fulfilled.

After the loop n'_s is assigned to n_s (Step 3.50), since it is the index up to where I_s contains all non s-prunable s-bounds. Finally, new_s_{ind}, consistent with new_ind s-bounds, the updated B_s and new_ind are returned. Because of their assignment to s, B_s and ind, Invariant 3.6.3 follows from invariant B, and Invariant 3.6.5 follows from invariant C.

The auxiliary procedures for managing s-bounds are grouped in Algorithm 3.9. The procedure `init_gen_stack` just initializes B_s and B_h as discussed above. Given a concept representation c, a concept representation s and a stack B_s, `exists_more_general` checks whether c is more general

```
procedure init_gen_stack($B_s$: stack; $n_c$: index ) returns stack, stack

    $ind := n_c$
    $B_h := \emptyset$
    if $B_s = \emptyset$
    then $B_h := push( n_s , s , \emptyset , B_h )$
    else $ind, s_{ind}, alt_{ind}, B_s := pop( B_s )$
         if $ind \neq n_s$
         then $B_h := push( n_s , s , \emptyset , B_h )$
         while $B_s \neq \emptyset$ and $n_c \leq ind$
              do $B_h := push( ind , s_{ind} , alt_{ind} , B_h )$
                 $ind, s_{ind}, alt_{ind}, B_s := pop( B_s )$
         endwhile
    if $n_c > ind$ then  $B_s := push( ind , s_{ind} , alt_{ind} , B_s )$
    return $B_s$, $B_h$
endproc


procedure exists_more_general ( $c$: concept; $s$: concept; $B_s$: stack )
          returns boolean
    $B_c := copy( B_s )$
    $exists\_more\_general := s \preccurlyeq c$
    while $\neg is\_empty( B_c )$ and $\neg exists\_more\_general$
        do $ind, s_{ind}, alt_{ind}, B_c := pop( B_c )$
           $exists\_more\_general := ( \exists c' \in alt_{ind} : c' \preccurlyeq c )$
    endwhile
    return exists_more_general
endproc

procedure shift ( $to$: index; $from$: index; $upto$: index; $i$: info; $I_c$: array )
          returns index,array
    while $from \neq upto$
        do $from := from + 1$
           if $I_c[from]$ is not $c$-prunable w.r.t. $i$
           then $to := to + 1$
                if $from \neq to$ then $I_c[to] := I_c[from]$
    endwhile
    return $to$, $I_c$
endproc
```

Algorithm 3.9 Auxiliary procedures for managing s-bounds

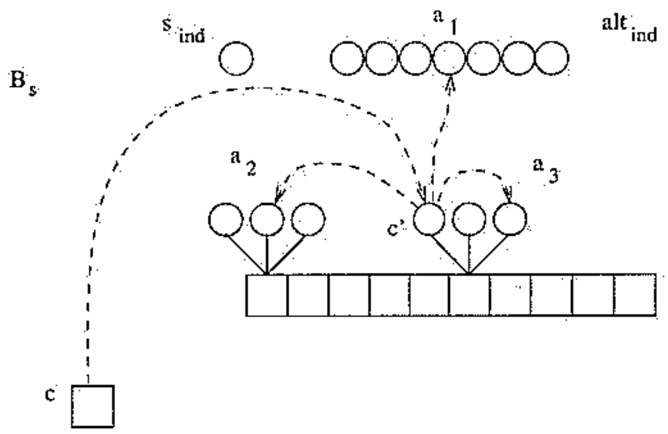


Figure 3.23 Proving completeness when $a \preccurlyeq c$

than s, or more general than some alternative on B_s. Finally, given three indexes *to*, *from* and *upto*, an information element i and an array I_c (which is I_s or I_g), such that $to \leq from < upto$, the procedure `shift` shifts all elements from $I_c[from+1..upto]$ not c-prunable w.r.t. i to $I_c[to+1], I_c[to+2], \ldots$ c-prunable in this context means s-prunable if $I_c = I_s$ and g-prunable if $I_c = I_g$.

3.9.4 Example

We will show on the example of Section 3.7 what is the result of the integration of **manage_s_bound** in ITVS.

The set of information elements is repeated in Figure 3.24. The first three information

	New Information	Stored In
1	$\neg(t \preccurlyeq c(human, \top))$	$I_s[1]$
2	$t \preccurlyeq c(\top, anim)$	$I_g[1]$
3	$\neg(c(woman, woman) \preccurlyeq t)$	$I_g[2]$
4	$t \preccurlyeq c(anim, anim)$	$I_g[1]$
5	$\neg(t \preccurlyeq c(woman, \top))$	
6	$c(woman, \bot) \preccurlyeq t$	$I_s[2]$
7	$\neg(c(man, \bot) \preccurlyeq t)$	$I_g[3]$
8	$c(woman, man) \preccurlyeq t$	$I_s[2]$
9	$\neg(t \preccurlyeq c(\top, human))$	$I_s[3]$
10	$\neg(c(man, man) \preccurlyeq t)$	$I_g[3]$

Figure 3.24 Information elements

elements are neither s-prunable nor g-prunable. Therefore B_s and B_g are constructed as before. When the positive upperbound $c(animate, animate)$ element is provided to ITVS,

	g	B_g
	s	B_s
	$c(\top,\top)$	$\emptyset$
	$c(\bot,\bot)$	$\emptyset$
1	$c(\top,\top)$	$\emptyset$
	$c(inanim,\bot)$	$[(1, c(inanim,\bot), [c(female,\bot), c(male,\bot)])]$
2	$c(\top, anim)$	$\emptyset$
	$c(inanim,\bot)$	$[(1, c(inanim,\bot), [c(female,\bot), c(male,\bot)])]$
3	$c(inanim, anim)$	$[(2, c(inanim, anim), [c(male, anim), c(\top, male)])]$
	$c(inanim,\bot)$	$[(1, c(inanim,\bot), [c(female,\bot), c(male,\bot)])]$
4		$t\preccurlyeq c(anim, anim)$ replaces $t\preccurlyeq c(\top, anim)$
	$c(male, anim)$	$[(2, c(male, anim), [c(anim, male)])]$
	$c(female,\bot)$	$[(1, c(female,\bot), [c(male,\bot)])]$
5		$\neg(t\preccurlyeq c(woman,\top))$ is s-prunable w.r.t. $\neg(t\preccurlyeq c(human,\top))$
6	$c(anim, male)$	$\emptyset$
	$c(female,\bot)$	$[(1, c(female,\bot), [c(male,\bot)])]$
7	$c(female, male)$	$\emptyset$
	$c(female,\bot)$	$\emptyset$
8		$c(woman, man)\preccurlyeq t$ replaces $c(woman,\bot)\preccurlyeq t$
	$c(female, male)$	$\emptyset$
	$c(female, man)$	$\emptyset$
9	$c(female, male)$	$\emptyset$
	$c(female, male)$	$\emptyset$

Figure 3.25 States of g, B_g, s and B_s

the positive upperbound $c(\top, \mathit{animate})$ in $I_g[1]$ is g-prunable and is therefore *replaced* by $c(\mathit{animate}, \mathit{animate})$. In the same step g and B_g are to be specialized to be consistent with the new $I_g[1]$: g cannot be specialized consistently, and is replaced by the first alternative on the stack, i.e., $c(\mathit{male}, \mathit{animate})$. Note that $c(\mathit{male}, \mathit{animate})$ is consistent with $I_g[1]$. Furthermore the other alternative $c(\top, \mathit{male})$ on B_g is specialized to $c(\mathit{animate}, \mathit{male})$ to be consistent with $I_g[1]$. The fifth information element $\neg(t \preccurlyeq c(\mathit{woman}, \top))$ is s-prunable w.r.t. $\neg(t \preccurlyeq c(\mathit{human}, \top))$ and is therefore not included in I_s. The sixth and seventh information elements are included in I_s and I_g, respectively. However, then the positive lowerbound $c(\mathit{woman}, \bot)$ is s-prunable w.r.t. the positive lowerbound $c(\mathit{woman}, \mathit{man})$; the latter therefore replaces the former in $I_g[2]$. Since B_s is empty, it does not have to be generalized. However, s has to be generalized w.r.t. the new $I_g[2]$. Finally with the ninth information element the search converges again to $c(\mathit{female}, \mathit{male})$. Note that the last information element, the negative lowerbound $c(\mathit{man}, \mathit{man})$, can still replace $I_g[3]$. Consequently, 4 of the 10 original information elements were s-prunable or g-prunable and have been discarded.

3.9.5 Complexity analysis [T]

SUMMARY: in this section we discuss what are the costs of extending ITVS with Algorithm 3.7: the worst case space complexity of ITVS remains linear in the number of information elements. The worst case time complexity has increased with a term quadratic in the number of information elements.

Lemma 3.56 Given are the s-bounds $i_1, i_2, \cdots, i_n, i$ such that i_1 is s-prunable w.r.t. i and $i_2, \cdots, i_n$ and i are not s-prunable. The two sequences $S = i_1, i_2, \cdots, i_n$ and $S' = i, i_2, \cdots, i_n, i$ will be presented to the extended ITVS. Suppose that:

- the initial value of s and B_s before any of these s-bounds is presented to ITVS is s_0 and $B_{s,0}$;
- there exists $c \in \mathcal{L}_C$ consistent with I with $s_0 \preccurlyeq c$;
- for all $1 \leq k \leq n$, $B_{s,k}$ denotes the value of B_s after the sequence $i_1, i_2, \cdots, i_k$ has been presented to the extended ITVS, starting from s_0 and $B_{s,0}$. Let s_k denote the corresponding value of s (see Figure 3.27). The choicepoints of the final stack $B_{s,n}$ are denoted (k , s_k , alt_k). Similarly, $B'_{s,k}$ denotes the value of B_s after the sequence $i, i_2, \cdots, i_k$ has been presented to the extended ITVS, starting from s_0 and $B_{s,0}$. s'_k is the corresponding value of s. The choicepoints of $B'_{s,n}$ are denoted (k , s'_k , alt'_k);
- for each choicepoint on $B_{s,n}$ s_{ind} has been chosen such that $s_{ind} \preccurlyeq c$.

Assuming that the index of i_k in I_s is k, let $new\_B_{s,k}$ denote the value of B_s in `generalize_stack` after generalizing up to choicepoint (k , s_k , alt_k).
Then we have the following result: when the sequence S' is presented to the extended ITVS, there exists for each k, $1 \leq k \leq n$, a choice for s'_k such that $B'_{s,n} = new\_B_{s,n}$.

Proof We will prove the lemma by induction on n. First we prove the lemma for $n = 1$ (see Figure 3.26: the boxes depict concept representations of B_s, the circles depict generalizations of the boxes w.r.t. i). When presenting S (containing only i_1), there are two possibilities: s_0 could be consistent with i_1 or not:

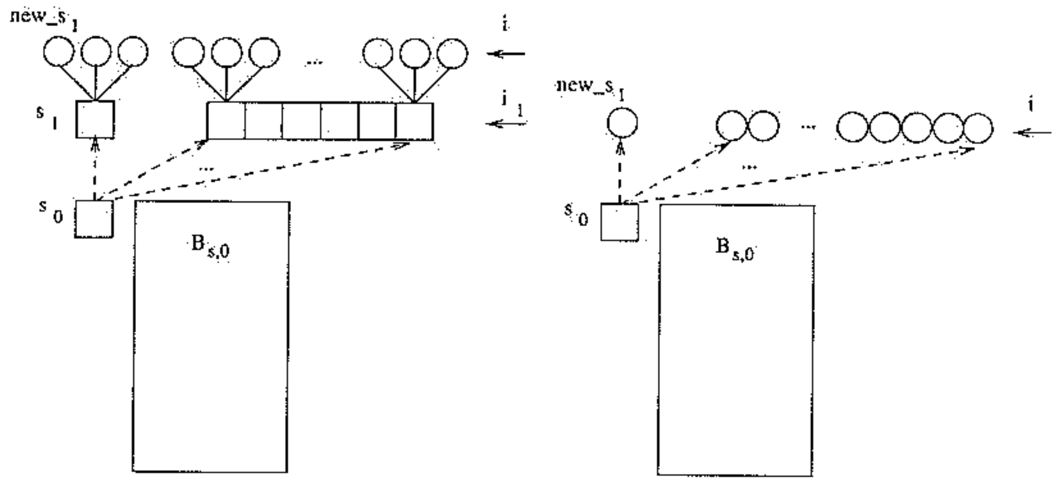


Figure 3.26 Case $n = 1$

- if it is not, the choicepoint (1 , s_1 , alt_1) will be on B_h.
- if it is, $B_{s,0} = B_{s,1}$ and $s_1 = s_0$. In this case, init_gen_stack will make sure that choicepoint (1 , s_0 , []) is on B_h.

In both cases generalize_stack will generalize s_1 minimally to $gens_1$, alt_1 to $gens_2$. Then $gens_1 \cup gens_2$ contains the minimal generalizations of s_0 w.r.t. i that are not discardable w.r.t. $B_{s,0}$. Then new_s_1 is chosen as one of those (since there exists one consistent with all elements of I_g), new_alt_1 contains the others. Consequently when presenting S' (containing only i), ITVS will compute exactly the minimal generalizations of s_0 w.r.t. i, select one of them to be $new\_s'_1$, the rest being new_alt_1. Consequently, $new\_s'_1$ could be chosen equal to new_s_1, and $new\_alt'_1$ would then be new_alt_1, and thus $B'_{s,n} = new\_B_{s,n}$.

Now suppose the lemma holds for all k from 1 up to $n-1$. We will prove it holds for n (see Figure 3.27). Since generalize_stack incrementally generalizes B_s with increasing index ind, and because of the induction hypothesis, there exists for each k from 1 up to $n-1$, a choice for s_k, such that $B'_{s,k} = new\_B_{s,k}$.

Again there are two cases:

- First, suppose there exists a choicepoint on $B_{s,n}$ with index n. Then all elements in $\{ s_n \} \cup alt_n$ are minimal generalizations of s_{n-1} consistent with i_n and not discarded w.r.t. $B_{s,n-1}$. The procedure generalize_stack generalizes s_n minimally to $gens_1$, alt_n to $gens_2$. Note that all elements of $gens_1$ and of $gens_2$ are more specific than new_s_{n-1} (see Section 3.9.3). Thus $gens_1 \cup gens_2$ contains the maximally specific generalizations of new_s_{n-1} consistent with i and i_n and not discardable w.r.t. $B_{s,n-1}$. Then new_s_n is chosen as one of those (since there exists one consistent with all elements of I_g), new_alt_n contains the others. Now, since $new\_B_{s,n-1} = B'_{s,n-1}$, we have that $new\_s_{n-1} = s'_{n-1}$. So the minimal generalizations of s'_{n-1} w.r.t. i_n is the same set $gens_1 \cup gens_2$. So $new\_s'_n$ can be chosen equal to new_s_n, and $new\_alt'_n$ is then new_alt_n.
- Now suppose there is no choicepoint on $B_{s,n}$ with index n, i.e., $B_{s,n} = B_{s,n-1}$. On the one hand, there is no choicepoint to generalize, so $new\_B_{s,n} = new\_B_{s,n-1}$. On the

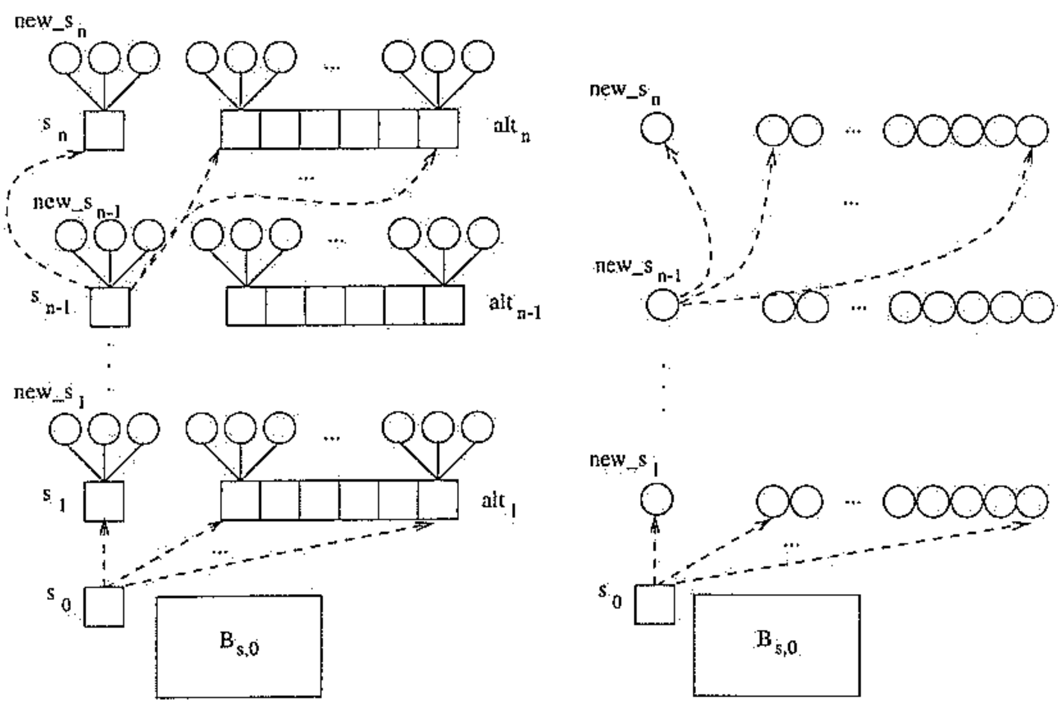


Figure 3.27 Induction step

other hand, s_{n-1} must be consistent with i_n, so its generalization new_s_{n-1} is also consistent with i_n. Consequently $B'_{s,n} = B'_{s,n-1}$.

□

Theorem 3.57 The original ITVS and the extended ITVS have the same worst case space complexity; they are linear in the number of information elements.

Proof The previous lemma shows that the sequence S presented to the extended ITVS results in the same B_s as the sequence S', in case there is no backtracking involved. The latter sequence does not contain any s-prunable information elements, and is therefore handled exactly the same as in the original ITVS. Involving backtracking would only *remove* elements from B_s. Therefore the *worst case* space complexity of the extended ITVS cannot be worse than, and is thus equal to, the worst case space complexity of the original ITVS. □

We can now explain the reason for the change in representation of B_s and B_g w.r.t. [Sablon *et al.*, 1994] (see Section 3.6.1). During the update of B_s we will have to be able to generalize each s_{ind} in order to put these generalizations all but one in a list of alternatives on B_s. If s_{ind} were not known, all alternatives on B_s with index larger than *ind* would be generalized instead, which could lead to a combinatorial explosion of the size of B_s.

For the worst case time complexity, we will again count the number of $\preccurlyeq$-tests, the number of generalization operations and the number of specialization operations (see Section 3.8.2).

Theorem 3.58 The worst case time complexity of the extended ITVS has an extra term of

$$\mathcal{O}(\ (\ (n_s + n_g)^2 + (b_s^2 + b_g^2) \times n_s \times n_g + b_s^3 \times n_s^2 + b_g^3 \times n_g^2\ ) \times c_{\preccurlyeq}\ ),$$

i.e., quadratic in the number of information elements.

Proof On the one hand, no parts of the search space are explored more than once in the extended ITVS, because of Invariant 3.6.3, and because no information elements are reprocessed during the generalization of B_s. On the other hand, as a consequence of Theorem 3.57, the worst case time complexity of the tests for maximal generality and maximal specificity remain linear in the number of information elements. Consequently, we only have to *add* the overhead of testing information elements for being s-prunable or g-prunable, and the overhead of updating B_s and B_g.

There is no overhead in specialization or generalization operations, since all elements of the specific-to-general search space are still specialized only once: if an element is generalized during the update of B_s in **generalize_stack**, it is removed from B_s (for being overly specific), and it will never be considered again. Dually all elements of the general-to-specific search space are still generalized only once. The only overhead to be counted is therefore an overhead in the number of $\preccurlyeq$-tests.

In case no s-prunable information elements are provided to the extended ITVS, it will have an overhead w.r.t. the original ITVS of comparing each new s-bound to all previous s-bounds, and comparing each new g-bound to all previous g-bounds. Furthermore, for detecting near-misses and their dual counterpart, s-bounds will also be compared to g-bounds, and vice versa. This gives an extra term of $\mathcal{O}(\ (n_s + n_g)^2 \times c_{\preccurlyeq}\ )$.

Then, if an information element is found to be s-prunable, for all n_s choicepoints $(\ ind\ ,\ s_{ind}\ ,\ alt_{ind}\ )$ on B_s, s_{ind} must be compared to:

- all n_g g-bounds;
- all alternatives in each, already generalized, choicepoint on B_s;
- s and all alternatives in each element in alt_{ind};
- all alternatives in each remaining choicepoint on B_h.

Together the latter three cases give $b_s \times n_s$ (the worst case size of B_s) $\preccurlyeq$-tests. Consequently, this is an operation of $\mathcal{O}(\ (\ n_s \times n_g + b_s \times n_s^2\ ) \times c_{\preccurlyeq}\ )$.

Similarly for all n_s choicepoints $(\ ind\ ,\ s_{ind}\ ,\ alt_{ind}\ )$ on B_s, all b_s generalizations of each of the $(b_s - 1) \times n_s$ elements in alt_{ind} must be compared to:

- all n_g g-bounds;
- all b_s alternatives in each of the $b_s \times n_s$ already generalized choicepoints on B_s;
- each of the b_s elements of $gens_1$;
- each of the $b_s \times b_s$ elements of $gens_2$.

The major term is $\mathcal{O}(\ (\ b_s^2 \times n_s \times n_g + b_s^3 \times n_s^2\ ) \times c_{\preccurlyeq}\ )$.

In total an s-prunable element gives in the worst case an overhead of

$$\mathcal{O}(\ (\ b_s^2 \times n_s \times n_g + b_s^3 \times n_g^2\ ) \times c_{\preccurlyeq}\ ).$$

Dually, if an element is g-prunable, we have a term of

$$\mathcal{O}(\ (\ b_g^2 \times n_s \times n_g + b_g^3 \times n_g^2\ ) \times c_{\preccurlyeq}\ ).$$

□

Although the worst case time complexity has increased w.r.t. ITVS's, it has only increased with a quadratic term, while, if the size of S or G is exponential, reducing the number of s-bounds, resp. g-bounds, will also reduce search time with an exponential factor.

3.9.6 Related Work

The work presented in this section elaborates the ideas of [Sablon and De Raedt, 1995]. These ideas extend the work of [Sebag, 1994], [Sebag and Rouveirol, 1994] and [Smith and Rosenbloom, 1990]. In [Sebag, 1994], which is restricted to conjunctive tree-structure languages, negative lowerbounds are converted into positive upperbounds, and only those *nearest* to the target concept (i.e., the most specific ones) are stored. In [Sebag and Rouveirol, 1994] this is extended to negative lowerbounds in ILP, which are represented by integrity constraints and ordered by θ-subsumption. In our framework we have generalized the notion of a *nearest miss* (which is introduced in [Sebag, 1994] and defined as a negative lowerbound which is not s-prunable) to all negative information elements not s-prunable nor g-prunable. We also introduce a notion of *nearest match* for all positive information elements not s-prunable nor g-prunable.

Two aspects of the INBF algorithm [Smith and Rosenbloom, 1990] can be compared to ours. In the specific-to-general search INBF drops all positive examples, because no backtracking is involved in searching specific-to-general in a conjunctive tree-structure language. Using Theorem 3.48 our approach would also drop all positive lowerbounds, except one (which would then coincide with s), because any two positive examples will have only one minimal upperbound. In the general-to-specific search INBF processes and then forgets all near-misses w.r.t. s. Its maximally general hypothesis *upper* is only kept consistent with all positive examples and all near-misses, so no backtracking is needed. Our notion of a near-miss generalizes this approach, by converting all negative lowerbounds to positive upperbounds, and considering their maximal lowerbound (i.e., *upper*) as a positive upperbound.

[Hirsh, 1990] informally describes a technique of "skipping data that do not change the versionspace" in the context of the Incremental Versionspace Merging algorithm. Intersecting a versionspace $VS1$ with $VS2$, and then with a subset $VS2'$ of $VS2$ will always yield the latter intersection. Consequently the first intersection operation was not necessary. In our framework, we more formally describe the approach using the notions s-prunable and g-prunable, we relate these notions to the concept of near-misses and to INBF, and provide a framework to automatically generate new information elements.

3.10 Shifting Language Bias with ITVS

[De Raedt, 1992] describes a generic algorithm for shifting the language bias for predicate learning, by using a series of language biases. We briefly discuss the incorporation of shift of bias in the ITVS framework because it is very important in concept learning to have mechanisms for automatically determining a suitable language bias. A suitable language bias is on the one hand expressive enough to represent the target concept, and on the other hand restrictive enough to make the search process feasible. Yet this mechanism has some implications on the management of information elements, which have to be considered. Algorithm 3.10 is an instantiation of Algorithm 4.1 on p. 113 of [De Raedt,

```
procedure biased_ITVS ( $\mathcal{I}nf$: stream of info )
          returns concept,stack,array,index,concept,stack,array,index
   $l := 1$  {3.51}
   $success := false$  {3.52}
   repeat
        $\mathcal{I}nf := reset(\mathcal{I}nf)$  {3.53}
        $s, B_s, I_s, n_s, g, B_g, I_g, n_g := ITVS(\mathcal{L}_C^l, \mathcal{I}nf)$  {3.54}
        if ITVS failed
        then $l := l + 1$  {3.55}
        else  $success := true$
   until success
   return $s, B_s, I_s, n_s, g, B_g, I_g, n_g$
```

Algorithm 3.10 Shift of bias in ITVS

1992]. It shows that shift of bias can be applied in ITVS. The series of language biases is represented with numbers $1, 2, 3, \ldots$. Initially $l = 1$ (Step 3.51), and $success = false$ (Step 3.52). Each time ITVS (i.e., Algorithm 3.3) fails with bias l, the algorithm shifts to the next language bias $l+1$ (Step 3.55), and ITVS is restarted from the beginning in the newly selected bias (Step 3.54), i.e., g is again initialized to $\top$, and s to $\bot$, and all information elements have to be reprocessed. Therefore we introduce an operation reset (Step 3.53) which resets the pointer to $\mathcal{I}nf$'s current element to its first element. The question is whether it is necessary to reprocess all information elements, and whether I_s and I_g could not be used for that purpose. In other words, can this shift of bias algorithm be combined with techniques to discard and automatically generate information elements as discussed in Section 3.9? The answer is that as long as I_s and I_g do not depend on the chosen language bias, they can be reused, instead of resetting $\mathcal{I}nf$. Let us briefly review the possible alternatives:

- if no elements are discarded by ITVS, I_s and I_g contain all information elements of I anyway. Consequently, $I_s \cup I_g$ can be reprocessed, instead of resetting $\mathcal{I}nf$;
- if s-prunable and g-prunable information elements are discarded in ITVS, but no information elements are automatically generated, $I_s \cup I_g$ could contain less elements than I. However, whether or not an information element is discarded does not depend on the chosen language bias, but solely on the relation $\preccurlyeq$ between information elements. Therefore, s-prunable, resp. g-prunable, information elements will remain s-prunable, or g-prunable, when calling ITVS with another language bias. Consequently, also in this case only the elements of $I_s \cup I_g$ should be reprocessed, instead of resetting $\mathcal{I}nf$;
- if information elements are automatically generated, they depend on the used language bias, since they are computed using mub, mlb, msg and mgs. These refinement operators depend on the chosen language bias. Consequently, the information contained in $I_s \cup I_g$ depends on the language bias, and can not necessarily be reused

in combination with another language bias. This does not mean there is no advantage in combining shift of bias with automatically generating information elements: although there will be no advantage in space requirements, the time complexity of ITVS also depends on the size of I_s and I_g (i.e., on n_s and n_g), and can therefore benefit of discarding as many information elements as possible.

3.11 Instance Generation

In this section we will elaborate on extending ITVS towards automatic instance generation. We first derive some general results, then integrate these results in ITVS, and finally describe some related work.

3.11.1 Theory

In an *interactive* concept learning setting convergence can be accelerated by generating new *relevant* instance representations automatically, preferably a minimal sequence of them, and having them classified by an oracle. We will first define relevant lower- and upperbounds, and then discuss how their classification can be used by the concept learning algorithm.

Definition 3.59 (Relevant lower- and upperbounds) Given $s, g \in \mathcal{L}_C$, and $i \in \mathcal{L}_I \cup \mathcal{L}_C$:

- i is a relevant lowerbound w.r.t. s and g iff
$$\exists c_1, c_2 \in \mathcal{L}_C : s \preccurlyeq c_1 \prec c_2 \preccurlyeq g \text{ and } \neg(\ i \preccurlyeq c_1\ ) \text{ and } i \preccurlyeq c_2.$$
- i is a relevant lowerbound iff $\exists s \in \mathcal{S}, g \in \mathcal{G}$ such that i is a relevant lowerbound w.r.t. s and g.
- i is a relevant upperbound w.r.t. s and g iff
$$\exists c_1, c_2 \in \mathcal{L}_C : s \preccurlyeq c_1 \prec c_2 \preccurlyeq g \text{ and } c_1 \preccurlyeq i \text{ and } \neg(\ c_2 \preccurlyeq i\ ).$$
- i is a relevant upperbound iff $\exists s \in \mathcal{S}, g \in \mathcal{G}$ such that i is a relevant upperbound w.r.t. s and g.

Given a relevant upper- or lowerbound, and the fact that it is positive or negative, at least one element of $\mathcal{S}$ or one element of $\mathcal{G}$ is inconsistent and must be adapted.

- Suppose i is a relevant lowerbound w.r.t. s and g. Then there exist c_1 and c_2 such that $s \preccurlyeq c_1 \prec c_2 \preccurlyeq g$, $\neg(\ i \preccurlyeq c_1\ )$, and $i \preccurlyeq c_2$.
  - If i is known to be a *positive* lowerbound, i.e., $i \preccurlyeq t$ (where t is the target concept), then $\neg(\ s \sim i\ )$. In this case c_1 is a negative upperbound (i.e., $\neg(\ t \preccurlyeq c_1\ )$), because $t \preccurlyeq c_1$ would imply $i \preccurlyeq c_1$. In other words, c_1 is a negative upperbound because there is an instance representation covered by t which is not covered by c_1. Indeed, if $i \in \mathcal{L}_I$, then i is an instance representation covered by t but not by c_1. And if $i \in \mathcal{L}_C$, then there must be at least one instance representation covered by i and not covered by c_1, because $\neg(\ i \preccurlyeq c_1\ )$.

  - If i is known to be a *negative* lowerbound, i.e., $\neg(\ i \preccurlyeq t\ )$, then $\neg(\ g \sim i\ )$. In this case c_2 is a negative lowerbound (i.e., $\neg(\ c_2 \preccurlyeq t\ )$), because $c_2 \preccurlyeq t$ would imply $i \preccurlyeq t$. In other words, c_2 is a negative lowerbound because there is an instance representation covered by c_2 which is not covered by t. Note however that c_2 is s-prunable w.r.t. i, and does therefore not contain more information than i.

- Suppose i is a relevant upperbound w.r.t. s and g. Then there exist c_1 and c_2 such that $s \preccurlyeq c_1 \prec c_2 \preccurlyeq g$, $c_1 \preccurlyeq i$, and $\neg(\ c_2 \preccurlyeq i\ )$.

  - If i is known to be a *positive* upperbound, i.e., $t \preccurlyeq i$, then $\neg(\ s \sim i\ )$. In this case c_2 is a negative lowerbound ($\neg(\ c_2 \preccurlyeq t\ )$), because $c_2 \preccurlyeq t$ would imply $c_2 \preccurlyeq i$.
  - If i is known to be a *negative* upperbound, i.e., $\neg(\ t \preccurlyeq i\ )$, then $\neg(\ g \sim i\ )$. In this case c_1 is a negative upperbound ($\neg(\ t \preccurlyeq c_1\ )$), because $t \preccurlyeq c_1$ would imply $t \preccurlyeq i$. Note also that c_1 is g-prunable w.r.t. i.

In all four cases, at least one element in $\mathcal{S}$ or $\mathcal{G}$ is inconsistent with the relevant upperbound, and should be modified. In each case where s must be generalized, c_1 is a negative upperbound. This means that if s is generalized in the direction of g it will have to be more general than c_1. In each case where g must be specialized, c_2 is a negative lowerbound. This means that if g is specialized in the direction of s it will have to be more specific than c_2. Consequently, it is advantageous to choose c_1 as close as possible to g, and c_2 as close as possible to s. However, since $c_1 \prec c_2$ intuitively the ideal is that c_1 and c_2 are *"in the middle"* between s and g. A middle point could be defined based on the number of elements between s and g.

Definition 3.60 (Middle Point) Given $s, g, c_m \in \mathcal{L}_C$, c_m is a middle point between s and g, iff $s \preccurlyeq c_m \preccurlyeq g$ and

$$|\{\ c \mid s \preccurlyeq c \preccurlyeq c_m\ \}| = |\{\ c \mid c_m \preccurlyeq c \preccurlyeq g\ \}|.$$

The closer c_1 and c_2 are to a *middle point* between s and g, the less lower- or upperbounds would be needed on average to converge, since the number of elements between s and g would each time be halved. This strategy halves the part of the search space *between s and g*, i.e., it *locally* searches for a minimal number of instance representations for s and g to converge. This does not necessarily lead to a minimal number of instance representations for the *complete versionspace* between $\mathcal{S}$ and $\mathcal{G}$ to converge, which would be a *global* strategy. A global strategy should find an instance representation covered by half of the complete versionspace, and therefore assumes $\mathcal{S}$ and $\mathcal{G}$ to be available.

In the local strategy as well as in the global strategy, it would be an expensive operation to compute each time the number of elements between one or all couples s and g. Therefore, middle points will only be *approximated*, most often by using domain dependent heuristics.In Section 5.11 we will elaborate on this in the context of Inductive Logic Programming. In Chapter 6 we will use relevant lowerbounds in the context of *experiment generation* for an autonomous agent.

3.11.2 Instance generation in ITVS [T]

SUMMARY: augmenting ITVS with an instance generation method, requires $s \preccurlyeq g$, which is not always the case in ITVS. This section describes an algorithm with linear time complexity to compute an alternative for g more general than s. A dual approach could compute an alternative for s more specific than g.

Since ITVS only computes one maximally specific s and one maximally general g consistent with all information elements, we have to choose for a local strategy for generating instance representations. The problem in ITVS is, however, that $s \preccurlyeq g$ will not always hold. In case it does not, we have to find a consistent alternative s' of s or a consistent alternative g' of g such that $s \preccurlyeq g'$ and $s' \preccurlyeq g$. Requiring that $s' \in \mathcal{S}$ and $g' \in \mathcal{G}$ as well would require backtracking, and storing all maximally specific elements *not* more specific than g and all maximally general elements *not* more general than s (see Chapter 6). This would be in the worst case exponential in time and space. Without those requirements it can be done in linear time and linear space. Note that if i is a relevant upper- or lowerbound *w.r.t. s and g'* or *w.r.t. s' and g*, where g' and s' are consistent with I, it is also a relevant upper- or lowerbound: because of Constraint 3.18 (the Boundedness Constraint) there will still exist $g'' \in \mathcal{G}$ and $s'' \in \mathcal{S}$ such that $g' \preccurlyeq g''$ and $s'' \preccurlyeq s'$.

```
procedure above( $s$: concept; $B_g$: stack ) returns concept
    $B_c := copy( B_g )$  {3.56}
    repeat
        $n_c, s_{ind}, alt_{ind}, B_c := pop( B_c )$
    until $\exists c \in alt_{ind} : s \preccurlyeq c$  {3.57}
    select one $c$ from $alt_{ind}$ with $s \preccurlyeq c$ {3.58}
    while $n_c \neq n_g$
        do $n_c := n_c + 1$  {3.59}
            if $\neg( I_g[n_c] \sim c )$
            then $specs := specializations( c , I_g[n_c] )$
                c := select one c from $specs$ with $s \preccurlyeq c$  {3.60}
    endwhile
    return $c$
endproc
```

Algorithm 3.11 Searching an element more general than s, in case $\neg(s \preccurlyeq g)$

Algorithm 3.11 presents an algorithm to compute g' more general than s, in case $\neg(s \preccurlyeq g)$. First B_g is copied to B_c (Step 3.56). Since s is consistent with all examples, there must be at least one alternative c on B_g such that $s \preccurlyeq c$. Therefore choicepoints are popped from B_c until such c is found (Step 3.57). Because $s \preccurlyeq c$, c is consistent with all s-bounds. Then c must be specialized to be consistent with all g-bounds. This is done in the while-loop. Over the while-loop we have that $s \preccurlyeq c$ and $c \sim \{ I_g[1] , \ldots, I_g[n_c] \}$. Before the while-loop these invariants hold, because of Step 3.58 and Invariant 3.6.4 of B_g (the Soundness invariant for B_g). As long as the index n_c up to where c is consistent with elements from I_g is not n_g (i.e., as long as c is not consistent with *all* g-bounds) n_c is incremented with 1 (Step 3.59), and c is compared to $I_g[n_c]$. If $I_g[n_c]$ is consistent with c, nothing happens. Otherwise, *specializations*(c , $I_g[n_c]$) must contain an element c' such that $s \preccurlyeq c'$, because s is consistent with $I_g[n_c]$, and $s \preccurlyeq c$. This element c' is assigned to c in Step 3.60. We could guarantee that c' is in $\mathcal{S}$ by adding the test *max_general*(c , B_g) at Step 3.60. However, this would introduce backtracking if the test would fail.

This algorithm shows how the depth-first bi-directional approach of ITVS allows the generation of couples (s , g') or (s' , g) by means of which instances can be generated. The instance generation depends on the choice for c_1 and c_2. This choice then depends on the chosen notion of middle point, and on a domain dependent strategy to approximate middle points.

3.11.3 Related work

[Subramanian and Feigenbaum, 1986] has shown that generating a minimal sequence of instances is in general a NP-hard problem. A minimal sequence would each time halve the number of elements in the versionspace. In the worst case this would require to compare each concept representation of the versionspace with each instance representation, to check whether or not the concept representation covers the instance representation. Factorization of the versionspace [Subramanian and Feigenbaum, 1986] can make the problem less complex and domain dependent heuristics can guide this search.

With respect to instance generation bi-directional approaches are much more appropriate than solely specific-to-general strategies and solely general-to-specific strategies, such as, e.g., Marvin [Sammut and Banerji, 1986] or CLINT [De Raedt, 1992]. Uni-directional approaches cannot define a middle point, and therefore also not approximate it. Typ-~~ically, these systems take $c_1 = s$ and c_2 a most specific generalization of s covering at~~ least one instance representation not covered by s, in order not to overgeneralize s. In the bi-directional approach, even a random choice of c_1 and c_2 between s and g could not be worse.

3.12 Conclusion

In this chapter we first introduced the language independent framework of versionspaces to describe concept learning problems and their solutions. Within this framework we introduced an alternative representation for S and G which led to the framework of Iterative Versionspaces. The latter framework is also language independent, and has therefore a wide application potential in the field of Machine Learning. Within this framework we found that the breadth-first strategy of the Description Identification algorithm and the depth-first strategy of the Iterative Versionspaces algorithm can be seen as two extremes that can be described in the same framework. We therefore suggested that several search strategies in between depth-first and breadth-first can be described in and can benefit from the same framework.

In this chapter we have also presented the Iterative Versionspaces algorithm (ITVS), which is a depth-first algorithm in the Iterative Versionspaces framework. The main contribution of ITVS is that its worst case space complexity is *linear* in the number of information elements, which is an exponential gain w.r.t. DI for certain languages. At the same time ITVS is able to determine maximal generality and maximal specificity in linear time. The test for maximality and consistency is coupled to the use of *optimal refinement operators.* Optimal refinement operators avoid searching parts of the search space more than once. Because in the worst case we gain an exponential factor in space while still computing a maximally general and a maximally specific concept representation, we believe our approach contributes to making the use of versionspaces (and concept learning) more practical.

We have also described redundancy in storing information elements in concept learning in a language independent way. We have extended ITVS towards detecting and removing redundant information elements. At the same time we have introduced a method for generating new information elements automatically, which make several other information elements redundant. We also generalized the notion of near-miss, and, as in [Smith and Rosenbloom, 1990], shown that near-misses (and their dual counterpart) play a very important role in converging towards the target concept. Finally, we have also shown that storing both maximally specific and maximally general concept representations is useful in instance generation.

Chapter 4

Disjunctive Iterative Versionspaces

4.1 Introduction

In Chapter 3, we assumed the target concept representation was in the language $\mathcal{L}_C$. In this chapter, we will relax this assumption. If the assumption is false, one could try to approximate the concept representation as well as possible within the language $\mathcal{L}_C$, thereby dropping the requirement of complete consistency with I. Another possibility would be to provide a series of language biases, and shift from one language bias to another as in Chapter 3. The solution of this chapter is to introduce a new concept representation language which is, hopefully, rich enough to represent the target concept. This language is defined by means of $\mathcal{L}_C$, and at the same time a superset of $\mathcal{L}_C$: new concept representations will be introduced by constructing ***disjunctions*** of elements of $\mathcal{L}_C$. Intuitively disjunctions of concept representations can be seen as sets of concept representations. The *cover* of a disjunction of two concepts of $\mathcal{L}_C$ is then the union of the *covers* of both concepts. The idea is then that in a disjunction one of the disjuncts covers one part of the instances to be covered, and the other disjuncts cover the other instances to be covered. As such, the union will cover all instances to be covered. At the same time, none of the disjuncts should cover any of the instances that should not be covered, because then the union would cover these instances as well.

The introduction of disjunctions increases expressiveness of the concept representation language, but at the same time also increases computational complexity. Actually, expressiveness is increased too much to be practically useful. This is shown by describing the set of consistent disjunctions (i.e., *the disjunctive versionspace*) by means of the set of maximally general and the set of maximally specific disjunctions. Therefore we have to impose a *preference criterion* (see Chapter 2), which introduces some notion of minimality, and prefers minimal disjunctions. First, we extend the *Multiple Convergence* approach of [Murray, 1987a] to our language-independent framework: we reduce the possibly huge number of maximally specific disjunctions by approximating them by *almost maximally specific* disjunctions. In this framework we describe the *Disjunctive Description Identification* algorithm. Second we introduce two specific minimality criteria: the minimal length criterion and the minimal set criterion, and how they can be combined with almost maximal specificity. In each step, we gradually restrict the disjunctive versionspace to the remaining elements according to the preference criterion. This finally leads to a depth-first version of the Disjunctive Description Identification algorithm, in which the minimal set criterion is adopted, called the *Disjunctive Iterative Versionspaces* algorithm. Within the framework of Disjunctive Iterative Versionspaces we can also describe how to adopt the minimal length criterion.

As in Chapter 3 we describe *incremental* and *complete* algorithms, and discuss their computational complexity. Although it will be clear from these complexities that a practical application will have to introduce extra pruning, we claim this theoretical study is again very useful and enlightening in the sense that it can serve as a basis for the development of such practical algorithms. Moreover it can give further indications about *where* and *how (much)* to prune.

As in Chapter 3 the results of this chapter are again language independent, although we will have to impose some constraint on the language. This constraint allows to reduce the disjunctive *cover* relation, and the derived $\preccurlyeq_d$ relation, to *cover* and $\preccurlyeq$ on $\mathcal{L}_C$. Otherwise, the disjunctive problem cannot be solved by reducing it to $\mathcal{L}_C$; in that case it has to be solved using ITVS or DI by considering the set of disjunctions as a language independent from $\mathcal{L}_C$.

This chapter is structured as follows: in Section 4.2 we describe the language $\mathcal{DL}_C$ of disjunctive concept representations we want to consider, and we describe the versionspace of consistent elements in $\mathcal{DL}_C$. Because of a possible combinatorial explosion in the number of maximally specific elements, we introduce almost maximally specific concept representations, and we describe the Disjunctive Description Identification algorithm in Section 4.3.1. In Section 4.4 we discuss additional preference criteria for disjunctive concept representations. This leads to Section 4.5, in which we describe the Disjunctive Iterative Versionspaces algorithm. Finally, we conclude in Section 4.6.

4.2 Disjunctive versionspaces

Let us first define a disjunction of concept representations in $\mathcal{L}_C$.

Definition 4.1 (Disjunctions of concept representations in $\mathcal{L}_C$) Given $c_1, \ldots, c_n \in \mathcal{L}_C$, $n \geq 1$, the set $\{ c_1, \ldots, c_n \}$ is the disjunction of $c_1, \ldots, c_n$. We call $c_1, \ldots, c_n$ the *disjuncts* of $\{ c_1, \ldots, c_n \}$.

Notation 4.2 The disjunction $\{ c_1, \ldots, c_n \}$ is denoted as $c_1 \vee \cdots \vee c_n$, or $\bigvee_{j=1}^{n} c_j$. $\bigvee_G c$ denotes $\bigvee_{c \in G} c$, i.e., the disjunction of all elements of G.

By definition a disjunction is a non-empty finite element of $\mathcal{P}(\mathcal{L}_C)$, the *powerset*[1] of $\mathcal{L}_C$. As a special case, each element of $\mathcal{L}_C$ can be regarded as a disjunction with only one disjunct. Not all disjunctions are interesting: further on we identify the interesting ones, and restrict ourselves to these. However, we will first extend *cover* and $\preccurlyeq$ towards disjunctions.

If a disjunction of concept representations is to be interpreted as a new concept representation, we have to define its *cover*.

Definition 4.3 ($cover_d$) For all $c_1, \ldots, c_n \in \mathcal{L}_C$:

$$cover_d(c_1 \vee \cdots \vee c_n) = cover(c_1) \cup \cdots \cup cover(c_n).$$

[1]The powerset of a set S is the set of all subsets of S.

This means that $c_1 \vee \cdots \vee c_n$ covers all instances covered by one of the concept representations $c_1, \ldots, c_n$. The function $cover_d$ is an extension of $cover$, in that for all $c \in \mathcal{L}_C$, we have $cover_d(\, c\,) = cover(\, c\,)$.

We will also immediately extend $\preccurlyeq$ on $\mathcal{L}_C$ to $\preccurlyeq_d$ on disjunctions of elements of $\mathcal{L}_C$.

Definition 4.4 ($\preccurlyeq_d$) For all $d = c_1 \vee \cdots \vee c_n$ and $d' = c'_1 \vee \cdots \vee c'_m$, where $c_1, \ldots, c_n, c'_1, \ldots, c'_m \in \mathcal{L}_C$:

$$d \preccurlyeq_d d' \text{ iff } cover(\, d\,) \subseteq cover(\, d'\,).$$

$d \prec_d d'$ denotes that $d \preccurlyeq_d d'$ and $d \neq d'$. Similarly as for $cover_d$, $\preccurlyeq_d$ is an extension of $\preccurlyeq$, in that $c_1 \preccurlyeq_d c_2$ iff $c_1 \preccurlyeq c_2$ for all $c_1, c_2 \in \mathcal{L}_C$. Nevertheless, we will always write the index d for $cover_d$ and $\preccurlyeq_d$, to avoid confusion with $cover$ and $\preccurlyeq$.

Definition 4.3 gives us immediately the following result.

Proposition 4.5

$$\forall c_1, c_2 \in \mathcal{L}_C : c_1 \preccurlyeq c_2 \text{ iff } cover_d(\, c_1 \vee c_2\,) = cover(\, c_2\,).$$

Proof $cover_d(\, c_1 \vee c_2\,) = cover(\, c_2\,)$ iff
$cover_d(\, c_1\,) \cup cover_d(\, c_2\,) \subseteq cover(\, c_2\,)$ iff
$cover(\, c_1\,) \subseteq cover(\, c_2\,)$.

By definition this is equivalent to $c_1 \preccurlyeq c_2$. □

This result already shows that not all disjunctions are useful: some disjunctions will represent concepts that already have a representation in $\mathcal{L}_C$. As in Chapter 2 and Chapter 3 we argue it is not interesting to have multiple representations for one concept. To exclude multiple representations of this kind, we will work with *reduced* disjunctions only.

Definition 4.6 (Reduced form of a disjunction)

- A disjunct c_j in a disjunction $c_1 \vee \cdots \vee c_n$ is *redundant* iff there exists a k, with $1 \leq k \leq n$ and $k \neq j$, such that $c_j \preccurlyeq c_k$.
- The *reduced* form of a disjunction d is the disjunction of all non redundant disjuncts of d.
- A disjunction d is *reduced* if it does not contain redundant disjuncts.

Notation 4.7 The reduced form of a disjunction d is denoted by $[d]_r$.

Because of Proposition 4.5 $cover(\, [d]_r\,)$ is equal to $cover(\, d\,)$. Reducedness can make the boundary sets of the versionspaces of disjunctive concept representations finite, e.g., in case of grammars [Vanlehn and Ball, 1987]. [Vanlehn and Ball, 1987] also conjecture that a reduced versionspace for disjunctive normal form in first order logic is finite.

We also impose a more fundamental constraint on the disjunctions allowed in this Chapter. The idea of introducing disjunctions of elements of $\mathcal{L}_C$ is to provide a way to solve concept learning problems that cannot be solved in $\mathcal{L}_C$. Taking the set of all (finite) disjunctions of elements of $\mathcal{L}_C$ as a new concept representation language, and $cover_d$ as the corresponding $cover$-function, one can, in general, apply the algorithms DI (see Section 3.5) and ITVS (see Section 3.6). This means one has to search $\mathcal{P}(\mathcal{L}_C)$, by means of $\preccurlyeq_d$. However, under Constraint 4.8 we can express $\preccurlyeq_d$ in terms of $\preccurlyeq$ on $\mathcal{L}_C$. This allows to reduce searching $\mathcal{P}(\mathcal{L}_C)$ using $\preccurlyeq_d$ to searching $\mathcal{L}_C$ using $\preccurlyeq$.

Constraint 4.8 at the same time excludes some concepts for which there might exist multiple representations. E.g., it will exclude disjunctions $c_1 \vee c_2$ which are actually represented by an element c_3 of $\mathcal{L}_C$, i.e., $cover_d(\ c_1 \vee c_2\ ) = cover(\ c_3\ )$.

Constraint 4.8 (The Disjunctions Constraint) For all c_1, c_2 and c_3 in $\mathcal{L}_C$, we restrict ourselves to disjunctions $c_1 \vee c_2$ such that

$$cover(\ c_3\ ) \subseteq cover_d(\ c_1 \vee c_2\ )$$

implies

$$cover(\ c_3\ ) \subseteq cover(\ c_1\ ) \text{ or } cover(\ c_3\ ) \subseteq cover(\ c_2\ ).$$

This constraint restricts Definition 4.3, but does not contradict it. The reverse of the constraint is trivially true. In Chapter 5 we show that this constraint is certainly satisfied in an Inductive Logic Programming context without recursion. A fortiori it is satisfied in the context of attribute-value languages, since these are actually propositional representations.

Corollary 4.9 expresses that under Constraint 4.8 no reduced disjunctive concept representation $c_1 \vee c_2$ can at the same time represent another element $c_3 \in \mathcal{L}_C$.

Corollary 4.9 If $cover_d(\ c_1 \vee c_2\ ) = cover(\ c_3\ )$ then $c_1 = c_3$ or $c_2 = c_3$.

Through Proposition 4.10 and Proposition 4.11 we now express $d_1 \preccurlyeq_d d_2$ by means of $\preccurlyeq_d$ between the disjuncts of d_1 and d_2. In Proposition 4.10 we split up the left-hand side of $d_1 \preccurlyeq_d d_2$; in Proposition 4.11 we split up the right-hand side. For the first operation we do not need Constraint 4.8. For the latter, however, we do.

Proposition 4.10 For all $c_1, c_2 \in \mathcal{L}_C$ and a finite disjunction $d \in \mathcal{P}(\mathcal{L}_C)$:

$$c_1 \vee c_2 \preccurlyeq_d d \text{ iff } (\ c_1 \preccurlyeq_d d \text{ and } c_2 \preccurlyeq_d d\ ).$$

Proof $c_1 \vee c_2 \preccurlyeq_d d$ iff
$cover_d(\ c_1 \vee c_2\ ) \subseteq cover_d(\ d\ )$ iff
$cover(\ c_1\ ) \cup cover(\ c_2\ ) \subseteq cover_d(\ d\ )$ iff
$cover(\ c_1\ ) \subseteq cover_d(\ d\ )$ and $cover(\ c_2\ ) \subseteq cover_d(\ d\ )$ iff
$c_1 \preccurlyeq_d d$ and $c_2 \preccurlyeq_d d$.

□

Note that we did not need to use Constraint 4.8 in this proof.

Proposition 4.11 For all $c_1, c_2, c \in \mathcal{L}_C$:

$$c \preccurlyeq_d c_1 \vee c_2 \text{ iff } (\ c \preccurlyeq c_1 \text{ or } c \preccurlyeq c_2\ ).$$

Proof $c \preccurlyeq_d c_1 \vee c_2$ iff
$cover(\ c\ ) \subseteq cover_d(\ c_1 \vee c_2\ )$ iff
$cover(\ c\ ) \subseteq cover(\ c_1\ )$ or $cover(\ c\ ) \subseteq cover(\ c_2\ )$ (by Constraint 4.8) iff
$c \preccurlyeq c_1$ or $c \preccurlyeq c_2$.

□

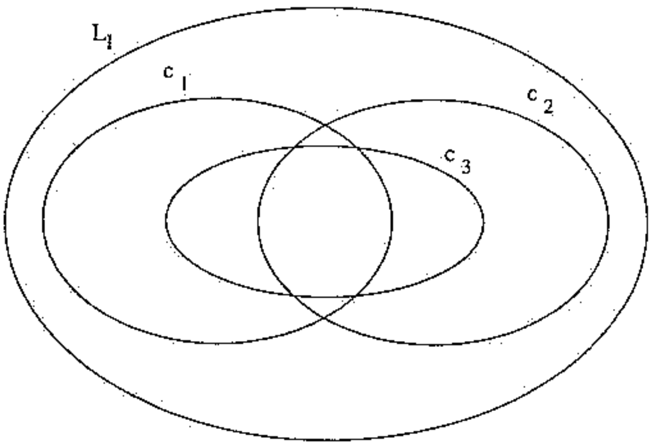


Figure 4.1 c_1, c_2, c_3 do not fulfill the Disjunctions Constraint

By definition of $\preccurlyeq_d$, Proposition 4.11 is equivalent to Constraint 4.8.

Corollary 4.12 For all $c_1, \ldots, c_n, c'_1, \ldots, c'_m \in \mathcal{L}_C$:

$$c_1 \vee \cdots \vee c_n \preccurlyeq_d c'_1 \vee \cdots \vee c'_m \text{ iff } \forall j,\ 1 \leq j \leq n:\ \exists k,\ 1 \leq k \leq m:\ c_j \preccurlyeq c'_k.$$

Corollary 4.12 allows to reduce $\preccurlyeq_d$ on $\mathcal{DL}_C$ to $\preccurlyeq$ on $\mathcal{L}_C$. If Constraint 4.8 does not hold, this reduction is not possible: $c_1 \vee \cdots \vee c_n \preccurlyeq_d c'_1 \vee \cdots \vee c'_m$ would allow each set *cover*(c_i) to be "distributed" over more than one *cover*(c'_j).

In Figure 4.1, for instance, all instances covered by c_3 are covered by $c_1 \vee c_2$, i.e., $c_3 \preccurlyeq_d c_1 \vee c_2$. However, neither $c_3 \preccurlyeq c_1$, nor $c_3 \preccurlyeq c_2$ hold. Because c_3 is related to neither c_1 nor c_2 by means of $\preccurlyeq$, we cannot express $\preccurlyeq_d$ in terms of $\preccurlyeq$. In which respect Constraint 4.8 limits the application of the results in this chapter in the context of Inductive Logic Programming, is discussed in Chapter 5.

From Corollary 4.12 and the fact that $\preccurlyeq$ is a partial order on $\mathcal{L}_C$ follows that the reduced form of a disjunction $c_1 \vee \cdots \vee c_n$ contains only the maximally general elements of $\{ c_1 , \ldots , c_n \}$. Consequently, it is unique.

We can now define the set $\mathcal{DL}_C$ of disjunctive concept representations we want to consider in this chapter.

Definition 4.13 ($\mathcal{DL}_C$) Given a concept representation language $\mathcal{L}_C$, the language of disjunctive concept representations $\mathcal{DL}_C \subseteq \mathcal{P}(\mathcal{L}_C)$ is defined by

$$\mathcal{DL}_C = \{ c \mid c = c_1 \vee \cdots \vee c_n,\ n \geq 1,\ c_1 , \ldots , c_n \in \mathcal{L}_C,\ c \text{ is reduced , and } \\ \forall c' \in \mathcal{L}_C :\ c' \preccurlyeq c \text{ implies } \exists k,\ 1 \leq k \leq n :\ c' \preccurlyeq c_k \} .$$

Note that, although $\mathcal{DL}_C$ is just another representation language, we introduce a new symbol ($\mathcal{DL}_C$) for it, because $\mathcal{DL}_C$ is built upon $\mathcal{L}_C$, and because we will still use properties and operations of the underlying $\mathcal{L}_C$ as well.

Proposition 4.14 The relation $\preccurlyeq_d$ is a partial order on $\mathcal{DL}_C$.

Proof Reflexivity and transitivity follow from the reflexivity and transitivity of $\preccurlyeq$, and of Corollary 4.12. To prove that $\preccurlyeq_d$ is anti-symmetric, suppose $c_1 \vee \cdots \vee c_n$, $c'_1 \vee \cdots \vee c'_m \in \mathcal{DL}_C$, such that $c_1 \vee \cdots \vee c_n \preccurlyeq_d c'_1 \vee \cdots \vee c'_m$ and $c'_1 \vee \cdots \vee c'_m \preccurlyeq_d c_1 \vee \cdots \vee c_n$. We prove that $c_1 \vee \cdots \vee c_n = c'_1 \vee \cdots \vee c'_m$.

For each j_1, $1 \leq j_1 \leq n$, there exists k_1, $1 \leq k_1 \leq m$, such that $c_{j_1} \preccurlyeq c'_{k_1}$. But for this c'_{k_1}, there exists j_2, $1 \leq j_2 \leq n$, such that $c'_{k_1} \preccurlyeq c_{j_2}$. Consequently $c_{j_1} \preccurlyeq c_{j_2}$. Because $c_1 \vee \cdots \vee c_n$ is reduced, $c_{j_1} = c_{j_2}$. Consequently, $c_{j_1} = c_{j_2} = c'_{k_1}$, because $\preccurlyeq$ is a partial order on $\mathcal{L}_C$. This means that each disjunct c_{j_1} of $c_1 \vee \cdots \vee c_n$ also appears in $c'_1 \vee \cdots \vee c'_m$. By reversing the rôles of $c_1 \vee \cdots \vee c_n$ and $c'_1 \vee \cdots \vee c'_m$, each disjunct of $c'_1 \vee \cdots \vee c'_m$ also appears in $c_1 \vee \cdots \vee c_n$. Consequently $c_1 \vee \cdots \vee c_n = c'_1 \vee \cdots \vee c'_m$. □

Now that we have defined disjunctive concept representations, one can wonder whether it is useful to extend the concept learning problem by allowing information elements to contain disjunctions as well. Allowing information elements to be disjunctions of elements of $\mathcal{L}_I \cup \mathcal{L}_C$ could be more informative than (a series of) single elements of $\mathcal{L}_I \cup \mathcal{L}_C$. Suppose $i_1, \ldots, i_m$ are disjunctive information elements, i.e., $i_l = i_{l,1} \vee \cdots \vee i_{l,m_l}$, for all $1 \leq l \leq m$. Also, let $d = c_1 \vee \cdots \vee c_n$, with $c_j \in \mathcal{L}_C$, for all $1 \leq j \leq n$. We will now investigate what is the relation between requiring that $i_1, \ldots, i_m$ are positive lowerbounds (resp. negative lowerbounds, positive upperbounds, negative upperbounds), and requiring that all $i_{l,k}$ (with $1 \leq l \leq m$ and $1 \leq k \leq m_l$) are positive lowerbounds (resp. negative lowerbounds, positive upperbounds, or negative upperbounds). Requiring that $i_1, \ldots, i_m$ are positive lowerbounds (resp. negative lowerbounds, positive upperbounds, or negative upperbounds) amounts to the following respective conditions (by Corollary 4.12):

1. positive lowerbounds: $\forall l\, \forall k\, \exists j : i_{l,k} \preccurlyeq c_j$.
2. negative lowerbounds: $\forall l\, \neg(\forall k\, \exists j : i_{l,k} \preccurlyeq c_j)$,
   or equivalently: $\forall l\, \exists k\, \forall j : \neg(\, i_{l,k} \preccurlyeq c_j\,)$.
3. positive upperbounds: $\forall l\, \forall j\, \exists k : c_j \preccurlyeq i_{l,k}$.
4. negative upperbounds: $\forall l\, \neg(\forall j\, \exists k : c_j \preccurlyeq i_{l,k})$,
   or equivalently: $\forall l\, \exists j\, \forall k : \neg(\, c_j \preccurlyeq i_{l,k}\,)$.

where $1 \leq l \leq m$, $1 \leq k \leq m_l$ and $1 \leq j \leq n$.

This means that for positive lowerbounds (case 1), there is no need to introduce disjunctive information elements: requiring that all $i_{l,k}$ are positive lowerbounds is equivalent to requiring that all i_l are positive lowerbounds. For negative upperbounds (case 4), requiring that all i_l are negative upperbounds is stronger than requiring that all $i_{l,k}$ are negative upperbounds, since $\forall l\, \exists j\, \forall k : \neg(\, c_j \preccurlyeq i_{l,k}\,)$ implies $\forall l\, \forall k\, \exists j : \neg(\, c_j \preccurlyeq i_{l,k}\,)$. The question is, whether there are cases where this difference is useful. If there are no such cases, we can conclude from case 1 and case 4 that there is no need to introduce disjunctive information elements for s-bounds.

For g-bounds (i.e., case 2 and case 3), the index k is only existentially quantified. For positive upperbounds (case 3), this means that of the set $\{\, i_{l,1}, \ldots, i_{l,m_l}\,\}$ only one element must be consistent with all c_j. For negative lowerbounds (case 2), $\forall l\, \exists k\, \forall j : \neg(\, i_{l,k} \preccurlyeq c_j\,)$ implies $\forall l\, \forall j\, \exists k : \neg(\, i_{l,k} \preccurlyeq c_j\,)$. The question again is, whether there are cases where this difference is useful. If there are no such cases, we can conclude that for g-bounds, only one element of the set $\{\, i_{l,1}\,, \ldots, i_{l,m_l}\,\}$ has to be consistent with all c_j in order for i_l to be consistent with d.

We argue that allowing disjunctive information elements makes the disjunctive problem more complex, without fundamentally changing the way it will be handled in this chapter. However, this should be investigated further: studying disjunctive information elements could fit in a global study of more general sorts of information elements (see also Chapter 3 concerning information elements for negations of concept representations). This study is beyond the scope of this thesis. Future work could try to find out how to extend the results of this chapter w.r.t. disjunctive information elements. For the moment we will restrict ourselves to the original problem specification, in that we allow non-disjunctive information elements only.

Constraint 4.15 (Restriction of information elements) Information elements are in $\mathcal{L}_I \cup \mathcal{L}_C$.

Now that we have determined what kind of information elements we deal with, we can extend the definition of consistency to $\mathcal{DL}_C$.

Definition 4.16 (Consistency) $d \in \mathcal{DL}_C$ is *consistent* with

1. a positive lowerbound $i \in \mathcal{L}_I \cup \mathcal{L}_C$, iff $i \preccurlyeq_d d$;
2. a negative lowerbound $i \in \mathcal{L}_I \cup \mathcal{L}_C$, iff $\neg(\, i \preccurlyeq_d d\,)$;
3. a positive upperbound $i \in \mathcal{L}_I \cup \mathcal{L}_C$, iff $d \preccurlyeq_d i$;
4. a negative upperbound $i \in \mathcal{L}_I \cup \mathcal{L}_C$, iff $\neg(\, d \preccurlyeq_d i\,)$.

$d \in \mathcal{DL}_C$ is consistent with $I \subseteq \mathcal{L}_I \cup \mathcal{L}_C$ iff d is consistent with all elements of I. If d is not consistent with i, resp. I, we call d inconsistent with i, or I.

As in Chapter 3 we will denote the set of all s-bounds of a given set I of information elements by I_s. Similarly the set of all g-bounds in I is denoted by I_s. Consequently, $I = I_s \cup I_g$.

Using Corollary 4.12, we can also reduce consistency of a disjunction to consistency of its disjuncts.

Proposition 4.17

- $\forall c_1, \ldots, c_n \in \mathcal{L}_C, \forall i \in I_g : (\, c_1 \vee \cdots \vee c_n\,) \sim i$ iff $(\, c_1 \sim i$ and $\ldots$ and $c_n \sim i\,)$.
- $\forall c_1, \ldots, c_n \in \mathcal{L}_C, \forall i \in I_s : (\, c_1 \vee \cdots \vee c_n\,) \sim i$ iff $(\, c_1 \sim i$ or $\ldots$ or $c_n \sim i\,)$.

Proof The proof consists of four cases:

- For a positive lowerbound i:

$c_1 \vee \cdots \vee c_n \sim i$ iff
$i \preccurlyeq c_1 \vee \cdots \vee c_n$ iff
$\exists j,\ 1 \leq j \leq n : i \preccurlyeq c_j$ iff
$c_1 \sim i$ or ... or $c_n \sim i$.

- For a negative lowerbound i:

  $c_1 \vee \cdots \vee c_n \sim i$ iff
  $\neg(\ i \preccurlyeq c_1 \vee \cdots \vee c_n\ )$ iff
  $\neg\exists j,\ 1 \leq j \leq n : i \preccurlyeq c_j$ iff
  $\forall j,\ 1 \leq j \leq n : \neg(\ i \preccurlyeq c_j\ )$ iff
  $c_1 \sim i$ and ... and $c_n \sim i$.

- For a positive upperbound i:

  $c_1 \vee \cdots \vee c_n \sim i$ iff
  $c_1 \vee \cdots \vee c_n \preccurlyeq i$ iff
  $\forall j,\ 1 \leq j \leq n : c_j \preccurlyeq i$ iff
  $c_1 \sim i$ and ... and $c_n \sim i$.

- For a negative upperbound i:

  $c_1 \vee \cdots \vee c_n \sim i$ iff
  $\neg(\ c_1 \vee \cdots \vee c_n \preccurlyeq i\ )$ iff
  $\neg\forall j,\ 1 \leq j \leq n : c_j \preccurlyeq i$ iff
  $\exists j,\ 1 \leq j \leq n : \neg(\ c_j \preccurlyeq i\ )$ iff
  $c_1 \sim i$ or ... or $c_n \sim i$.

□

It is also straightforward to extend Theorem 3.14 to $\preccurlyeq_d$. This will allow us to prune in $\mathcal{DL}_C$ in a similar way as we did in $\mathcal{L}_C$.

Theorem 4.18
If i is a g-bound, and $x, y \in \mathcal{DL}_C$ such that $x \preccurlyeq_d y$, then $y \sim i$ implies $x \sim i$.
If i is an s-bound, and $x, y \in \mathcal{DL}_C$ such that $x \preccurlyeq_d y$, then $x \sim i$ implies $y \sim i$.

Proof Suppose $x = c_1 \vee \cdots \vee c_n$ and $y = c'_1 \vee \cdots \vee c'_m$. Since $x \preccurlyeq_d y$ there exists for every k, $1 \leq k \leq n$, a j, $1 \leq j \leq m$, such that $c_k \preccurlyeq c'_j$ (because of Corollary 4.12).

For a given g-bound i:

$$y \sim i \text{ iff } \forall j,\ 1 \leq j \leq m : c'_j \sim i.$$

Consequently, $\forall k,\ 1 \leq k \leq n : c_k \sim i$ (by Theorem 3.14), and therefore $x \sim i$.

For a given s-bound i:

$$x \sim i \text{ iff } \exists k,\ 1 \leq k \leq n : c_k \sim i.$$

Consequently, $\exists j,\ 1 \leq j \leq m :\ c'_j \sim i$ (by Theorem 3.14), and therefore $y \sim i$. □

So far we have defined the set $\mathcal{DL}_C$ of disjunctive concept representations, and the relation $\preccurlyeq_d$ on $\mathcal{DL}_C$. We have reduced $\preccurlyeq_d$ to $\preccurlyeq$ on $\mathcal{L}_C$. The next step is to identify the set of all disjunctive concept representations consistent with a given set of information elements, i.e., the *versionspace* of all disjunctive concept representations. As in Chapter 3 we can also represent this set by means of its boundary sets.

Definition 4.19 (Disjunctive versionspace) Let $I = I_s \cup I_g$ be a set of information elements. The set $I_s = \{ i_1 , \ldots , i_n \}$ contains all s-bounds of I; the set I_g contains all g-bounds of I.

- let G be the set of all maximally general elements of $\mathcal{L}_C$ consistent with I_g;
- for all k, $1 \leq k \leq n$, let S_k be the set of all maximally specific elements of $\mathcal{L}_C$ consistent with $\{ i_k \} \cup I_g$.

Then we can define $\mathcal{DG}_I$, $\mathcal{DS}_I$ and $\mathcal{DVS}_I$ as follows:

- If $\forall i \in I_s \, \exists c \in G : c \sim i$, then
  1. $\mathcal{DG}_I = \{ \bigvee_G c \}$;
  2. $\mathcal{DS}_I = Min \{ [s_1 \vee \cdots \vee s_n]_r \mid \forall k, \, 1 \leq k \leq n : s_k \in S_k \}$;
  3. $\mathcal{DVS}_I = \{ d \in \mathcal{DL}_C \mid \exists g \in \mathcal{DG}_I, \exists s \in \mathcal{DS}_I : s \preccurlyeq_d d \preccurlyeq_d g \}$.
- Otherwise: $\mathcal{DG}_I = \mathcal{DS}_I = \mathcal{DVS}_I = \emptyset$.

With the notations of Chapter 3 we can reformulate some of the elements of Definition 4.19:

- $G = \mathcal{G}_{I_g}$;
- For every k, $1 \leq k \leq n$, $S_k = \mathcal{S}_{I_g \cup \{i_k\}}$, i.e.,
  - if i_k is a positive lowerbound: $S_k = \{ c \in mub(\perp , i_k) \mid c \sim I_g \}$, and
  - if i_k is a negative upperbound: $S_k = \{ c \in msg(\perp , i_k) \mid c \sim I_g \}$.

When no confusion is possible, the index I will again be omitted from $\mathcal{DVS}_I$, $\mathcal{DS}_I$ and $\mathcal{DG}_I$.

The idea behind these definitions is similar as in Chapter 3: $\mathcal{DG}$ is supposed to be the set of maximally general elements in $\mathcal{DL}_C$ that are consistent with I, $\mathcal{DS}$ is the set of maximally specific elements in $\mathcal{DL}_C$ consistent with I. All elements more specific than the element in $\mathcal{DG}$ and more general than an element in $\mathcal{DS}$ form the *disjunctive versionspace* of elements consistent with I.

Example 4.20 Figure 4.2 illustrates the idea schematically. A concept representation is depicted by a wedge. All wedges have the same base line. The top of the wedge is labeled with the concept representation it depicts. The information elements are depicted as dots on the baseline of the wedges. A concept representation is consistent with an s-bound if the s-bound is under the corresponding wedge. A concept representation is consistent with a g-bound if the g-bound is *not* under the wedge. If the wedge of c is underneath the wedge of c', then c is more specific than c'. The advantage of this kind of figure over a $\preccurlyeq$-diagram is that it expresses consistency of information elements with concept representations, rather than the relation $\preccurlyeq$.

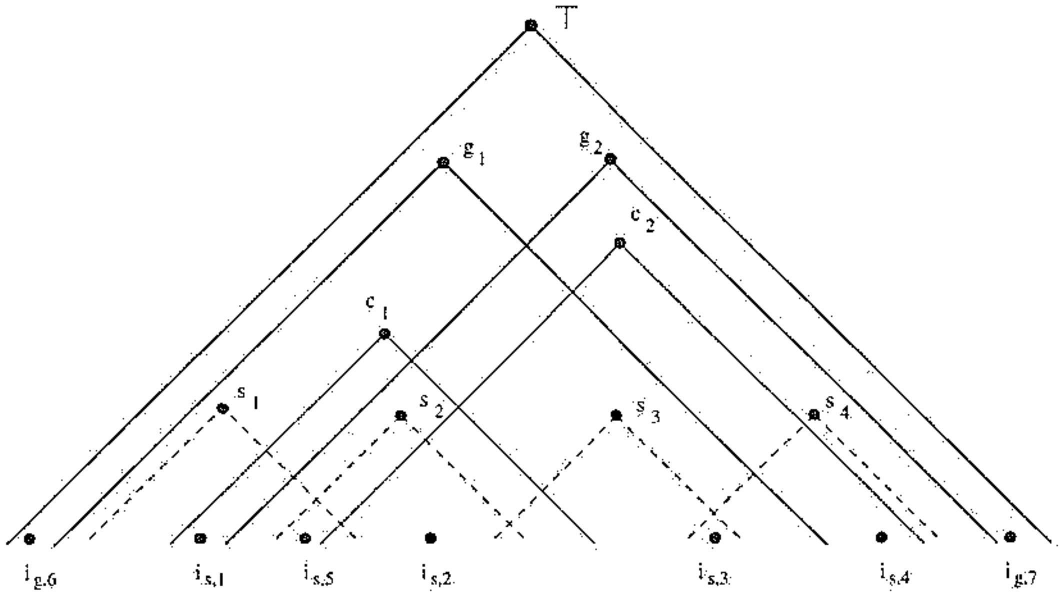


Figure 4.2 The idea behind the Disjunctive versionspaces

In Figure 4.2 $i_{s,1}$ to $i_{s,5}$ are s-bounds, $i_{g,6}$ and $i_{g,7}$ are g-bounds. Suppose g_1 and g_2 are maximally general concept representations in $\mathcal{L}_C$ consistent with I_g. They are not consistent with I, because g_1 is not consistent with $i_{s,4}$ and g_2 is not consistent with $i_{s,1}$. However $g_1 \vee g_2$ is consistent with I, and maximally general. On the other side, each s_j is an element of some S_k. Some s_j are consistent with more than one s-bound. The disjunction $s_1 \vee s_2 \vee s_4$ is maximally specific, reduced and consistent with I. $\diamond$

Now that we have defined $\mathcal{DG}$, $\mathcal{DS}$ and $\mathcal{DVS}$, we prove that they are extensions of $\mathcal{G}$, $\mathcal{S}$ and $\mathcal{VS}$ of Chapter 3, in that $\mathcal{DG}$ is the set of all maximally general consistent disjunctive concept representations, $\mathcal{DS}$ is the set of all maximally specific consistent disjunctive concept representations, and $\mathcal{DVS}$ is the set of *all* consistent concept representations. First we prove the following lemma.

Lemma 4.21 $\forall c_1, c_2, c_3 \in \mathcal{L}_C$: $c_1 \preccurlyeq c_2$ implies $c_1 \preccurlyeq_d c_2 \vee c_3$.

Proof We have that $c_1 \preccurlyeq c_2$ iff $cover(\ c_1\ ) \subseteq cover(\ c_2\ )$. This implies
$cover(\ c_1\ ) \subseteq cover(\ c_2\ ) \cup cover(\ c_3\ )$, which is equivalent to
$cover(\ c_1\ ) \subseteq cover_d(\ c_2 \vee c_3\ )$, and to $c_1 \preccurlyeq_d c_2 \vee c_3$. $\square$

In particular, $c_1 \preccurlyeq_d c_1 \vee c_2$. Therefore this lemma shows that adding a disjunct to a concept representation is a way of *generalizing* the concept representation.

Theorem 4.22

- $\mathcal{DG}$ is the set of maximally general elements in $\mathcal{DL}_C$ consistent with I.

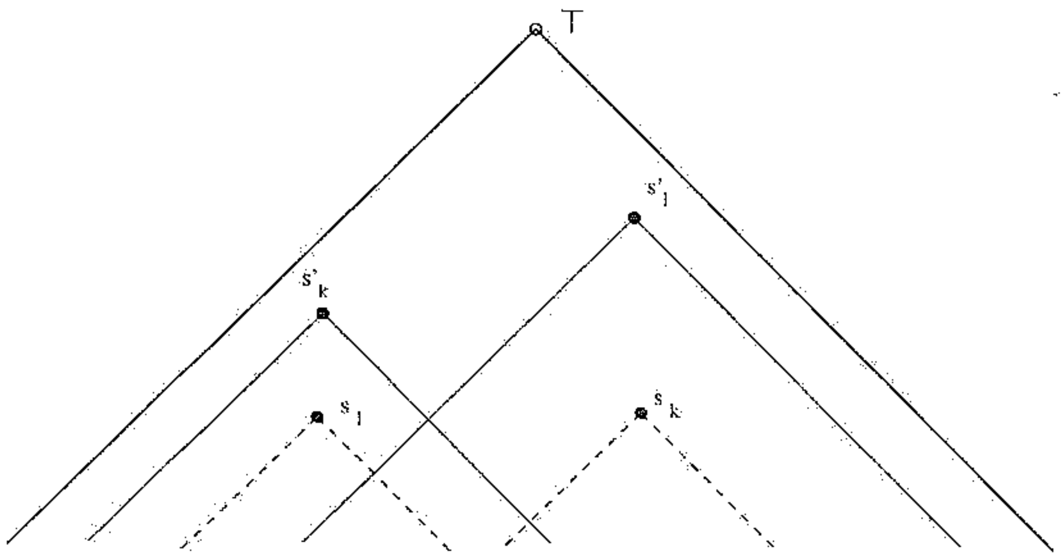


Figure 4.3 Minimality is needed in $\mathcal{DS}$

- $\mathcal{DS}$ is the set of maximally specific elements in $\mathcal{DL}_C$ consistent with I.

Proof First note that, since adding more disjuncts makes a concept representation in $\mathcal{DL}_C$ more general (see Lemma 4.21), the maximally general concept representations should contain as much as possible disjuncts, while the maximally specific concept representations should only contain as few as possible.

Every concept representation consistent with I_g, whether there is an s-bound consistent with it or not, can be included in the maximally general concept representations in $\mathcal{DL}_C$. Consequently, the disjunction of all elements of G is the only maximally general disjunctive concept representation, at least if it is consistent with all s-bounds, i.e., for each s-bound ther is an element of G such that $c \sim i$. If not, then $\mathcal{DG} = \emptyset$.

In a maximally specific concept representation in $\mathcal{DL}_C$ only as much disjuncts as strictly necessary to be consistent with all s-bounds have to be included. First note that for each s-bound i, concept representations consistent with i must contain a disjunct c_k consistent with i. If no such c_k exists for each i, $\mathcal{DG} = \mathcal{DS} = \emptyset$. Otherwise, the set of all disjunctions $c_1 \vee \cdots \vee c_n$, where $c_k \sim i_k$ for all k, $1 \leq k \leq n$, (and where there might exist $l \neq m$ such that $c_l = c_m$), certainly contains all maximally specific concept representations. But if $c_k \sim i_k$ and c_k is not maximally specific in $\mathcal{L}_C$, there is a $s_k \in S_k$ such that $s_k \preccurlyeq c_k$ (because of Constraint 3.18, the Boundedness Constraint) and thus

$$c_1 \vee \cdots \vee c_{k-1} \vee s_k \vee c_{k+1} \vee \cdots \vee c_n \preccurlyeq c_1 \vee \cdots \vee c_n.$$

Therefore we can restrict ourselves to all combinations of elements of S_k: for all others there exists a consistent disjunction which is more specific. Furthermore, some s_l might be more specific than some s_k, i.e., not all combinations are necessarily reduced.

Only the reduced ones are to be included in $\mathcal{DS}$. It could also happen that $s_k, s'_k \in S_k$, that $s_l, s'_l \in S_l$, $s_k \preccurlyeq s'_l$ and $s_l \preccurlyeq s'_k$ (see Figure 4.3). In that case the combinations of these four elements are $s_k \vee s_l$, $s'_k \vee s_l$, $s_k \vee s'_l$ and $s'_k \vee s'_l$. The disjunctions $s'_k \vee s_l$ and $s_k \vee s'_l$ are not reduced. The disjunction $s'_k \vee s'_l$ is reduced but not minimal. Of the four combinations $s_k \vee s_l$ is the only maximally specific disjunction. This shows that reduction only does not necessarily yield minimal elements w.r.t. $\preccurlyeq_d$, and that the extra *Min*-operation is necessary. □

Actually this specification of $\mathcal{DS}$ is not completely constructive: it does not give an algorithm that can generate the elements of $\mathcal{DS}$ one by one. It rather generates *candidates* and then selects the most specific ones. However, it does exclude in a constructive way a lot of candidates which would fail the maximally specificity test anyway, by only combining elements of S_k. Further on we will try to alleviate this non-constructive aspect by *approximating* maximally specific elements by *almost* maximally specific elements.

Theorem 4.23 $\mathcal{DVS}$ is the set of *all* concept representations in $\mathcal{DL}_C$ consistent with I.

Proof ($\subseteq$) First we prove that $\mathcal{DG}$ is a subset of the set of all consistent concept representations. If $\mathcal{DG}$ is empty, this is trivial. Otherwise, we first prove that $\bigvee_G c$ is consistent with I. Since every c in G is consistent with all elements of I_g, $\bigvee_G c$ is consistent with all elements of I_g. For each s-bound i at least one element of G is consistent with i, and therefore $\bigvee_G c$ is consistent with i.

Next we prove that $\mathcal{DS}$ is a subset of all consistent concept representations. This is trivial if $\mathcal{DS}$ is empty. Otherwise, it suffices to prove that $s_1 \vee \cdots \vee s_n$, with $s_k \in S_k$ for all k, $1 \leq k \leq n$, is consistent with I, as reduction and minimization do not affect consistence. For each s-bound i_k we have that s_k is consistent with i_k and with I_g. Consequently, the disjunction $s_1 \vee \cdots \vee s_n$ is consistent with $i_1, \ldots, i_n$ and with I_g, i.e., with I.

Now suppose $s \preccurlyeq_d d \preccurlyeq_d g$, then d is consistent with I_s because it is more general than s, and d is consistent with I_g because it is more specific than g.

($\supseteq$) We will first discuss the case where there exists an s-bound i such that there is no $c \in G$ which is consistent with i. In this case $\mathcal{DVS} = \mathcal{DS} = \mathcal{DG} = \emptyset$. No $d \in \mathcal{DL}_C$ can be consistent with I: every disjunction will be inconsistent either with a g-bound or with the s-bound i. So $\mathcal{DVS}$ and the set of all consistent concept representations are both empty.

Otherwise, suppose $d = c_1 \vee \cdots \vee c_n$ is consistent with I. We first prove $d \preccurlyeq_d \bigvee_G c$. For every j, $1 \leq j \leq n$, take g a maximally general element of

$$C = \{ x \in \mathcal{L}_C \mid c_j \preccurlyeq x \text{ and } x \sim I_g \}.$$

Such g exists because C is not empty ($c_j \in C$) and because of Constraint 3.18 (the Boundedness Constraint). Because g is maximally general in C, it is also an element of G. Consequently $c_j \preccurlyeq_d \bigvee_G c$, and $d \preccurlyeq_d \bigvee_G c$.

Then we have to prove that for $d = c_1 \vee \cdots \vee c_n$ consistent with I, there exists an $s \in \mathcal{DS}$ more specific than d. For every s-bound $i_k \in I_s$, there exists a j,

$1 \leq j \leq n$, with $c_j \sim i_k$. Then the set $C_k = \{ x \in \mathcal{L}_C \mid x \preccurlyeq c_j \text{ and } x \sim i_k \}$ is not empty ($c_j \in C_k$). Let s_k be a minimal element of C_k (which exists because of Constraint 3.18; the Boundedness Constraint). Consequently, s_k is in S_k and is consistent with i_k. Now let s be $[s_1 \vee \cdots \vee s_n]_r$. Then $s \preccurlyeq_d d$. Furthermore, $s \in \mathcal{DS}$, or, if it is not, there is an element $s' \in \mathcal{DS}$ which is more specific than s and minimal by definition of $\mathcal{DS}$ (and again because of Constraint 3.18 – the Boundedness Constraint). This completes the proof. □

We can now present a general solution to the concept learning problem in $\mathcal{DL}_C$ by defining mub_d, mlb_d, mgs_d and msg_d as in Definition 3.29 to Definition 3.32, where $\mathcal{L}_C$ is replaced by $\mathcal{DL}_C$. This allows us in principle to solve the problem by means of DI (see Section 3.5) or ITVS (see Section 3.6).

Definition 4.24 (Disjunctive refinement operators for ITVS) For $d_1, d_2 \in \mathcal{DL}_C$:

1. $mub_d(d_1 , d_2) = Min \{ d \in \mathcal{DL}_C \mid d_1 \preccurlyeq_d d \text{ and } d_2 \preccurlyeq_d d \}$;
2. $mlb_d(d_1 , d_2) = Max \{ d \in \mathcal{DL}_C \mid d \preccurlyeq_d d_1 \text{ and } d \preccurlyeq_d d_2 \}$;
3. $msg_d(d_1 , d_2) = Min \{ d \in \mathcal{DL}_C \mid d_1 \preccurlyeq_d d \text{ and } \neg(d \preccurlyeq_d d_2) \}$;
4. $mgs_d(d_1 , d_2) = Max \{ d \in \mathcal{DL}_C \mid d \preccurlyeq_d d_1 \text{ and } \neg(d_2 \preccurlyeq_d d) \}$.

Theorem 4.25 shows that we can also express mub_d, mlb_d, mgs_d and msg_d in terms of mub, mlb, mgs and msg in $\mathcal{L}_C$.

Theorem 4.25 Given $c_1, c_2 \in \mathcal{L}_C$, and $i \in \mathcal{L}_I$, let $MGS = mgs(c_1 , i) \cup mgs(c_2 , i)$, and $MLB = mlb(c_1 , i) \cup mlb(c_2 , i)$.

1. $mgs_d(c_1 \vee c_2 , i) = \{ [\bigvee_{MGS} g]_r \}$;
2. $mlb_d(c_1 \vee c_2 , i) = \{ [\bigvee_{MLB} g]_r \}$;
3. if $c_1 \vee c_2 \preccurlyeq_d i$, then $msg_d(c_1 \vee c_2 , i) = \{ [c_1 \vee c_2 \vee s]_r \mid s \in msg(\bot , i) \}$ otherwise $msg_d(c_1 \vee c_2 , i) = \{ c_1 \vee c_2 \}$;
4. if $\neg(i \preccurlyeq_d c_1 \vee c_2)$, then $mub_d(c_1 \vee c_2 , i) = \{ [c_1 \vee c_2 \vee s]_r \mid s \in mub(\bot , i) \}$ otherwise $mub_d(c_1 \vee c_2 , i) = \{ c_1 \vee c_2 \}$.

Proof The proof will make extensive use of Corollary 4.12.

1. Let $MGS' = \{ g \in \mathcal{L}_C \mid (g \preccurlyeq c_1 \text{ or } g \preccurlyeq c_2) \text{ and } \neg(i \preccurlyeq g) \}$. We first prove that each element d of $mgs_d(c_1 \vee c_2 , i)$ must be more specific than $[\bigvee_{MGS'} g]_r$. Then we prove that $[\bigvee_{MGS'} g]_r \in mgs_d(c_1 \vee c_2 , i)$. Consequently, we have that $mgs_d(c_1 \vee c_2 , i) = \{ [\bigvee_{MGS'} g]_r \}$. Finally we prove that $[\bigvee_{MGS} g]_r = [\bigvee_{MGS'} g]_r$.
   - The set $mgs_d(c_1 \vee c_2 , i)$ is the set of all maximally general disjunctions d, more specific than $c_1 \vee c_2$ and not more general than i. For each such d, all disjuncts of d are in MGS', because
     - if there were a disjunct c of d that is not more specific than c_1 and not more specific than c_2, then d would not be more specific than $c_1 \vee c_2$, and

  - if there were a disjunct c of d that is more general than i, then d is more general than i.

  MGS' is finite because of Constraint 3.19 (the Finiteness Constraint). Consequently, each $d \in mgs_d(c_1 \vee c_2 , i)$ is more specific than $\bigvee_{MGS'} g$, and therefore also more specific than $[\bigvee_{MGS'} g]_r$.

- Since each element of MGS' is more specific than c_1 or more specific than c_2, $\bigvee_{MGS'} g$ is more specific than $c_1 \vee c_2$. And since none of the elements of MGS' is more general than i, $\bigvee_{MGS'} g$ is not more general than i. Consequently, $[\bigvee_{MGS'} g]_r$ is the only element in $mgs_d(c_1 \vee c_2 , i)$.
- Finally, $mgs(c_1 , i) \subseteq MGS'$ and $mgs(c_2 , i) \subseteq MGS'$. Consequently $MGS \subseteq MGS'$. By definition of mgs, there exists for every $g' \in MGS'$ a $g \in MGS$ such that $g' \preccurlyeq g$. This means that g' will not be included in the reduction of $\bigvee_{MGS'} g$, if $g' \notin mgs(c_1 , i) \cup mgs(c_2 , i)$. Consequently, only the elements of MGS can be included in the reduction of $\bigvee_{MGS'} g$, i.e., the reduction of $\bigvee_{MGS'} g$ is $[\bigvee_{MGS} g]_r$.

2. Let $MLB' = \{ g \in \mathcal{L}_C \mid (g \preccurlyeq c_1 \text{ or } g \preccurlyeq c_2) \text{ and } g \preccurlyeq i \}$. We first prove that each element d of $mlb_d(c_1 \vee c_2 , i)$ must be more specific than $[\bigvee_{MLB'} g]_r$. Then we prove that $[\bigvee_{MLB'} g]_r \in mlb_d(c_1 \vee c_2 , i)$. Consequently, we have that $mlb_d(c_1 \vee c_2 , i) = \{ [\bigvee_{MLB'} g]_r \}$. Finally we prove that $[\bigvee_{MLB'} g]_r = [\bigvee_{MLB} g]_r$.
   - The set $mlb_d(c_1 \vee c_2 , i)$ is the set of all maximally general disjunctions d, more specific than $c_1 \vee c_2$ and i. For each such d, all disjuncts of d are in MLB', because
     - if there were a disjunct c of d that is not more specific than c_1 and not more specific than c_2, then d would not be more specific than $c_1 \vee c_2$, and
     - if there were a disjunct c of d that is not more specific than i, then d is not more specific than i.

     MLB' is again finite because of Constraint 3.19 (the Finiteness Constraint). Consequently, each $d \in mlb_d(c_1 \vee c_2 , i)$ is more specific than $\bigvee_{MLB'} g$, and therefore also more specific than $[\bigvee_{MLB'} g]_r$.
   - Since each element of MLB' is more specific than c_1 or more specific than c_2, $\bigvee_{MLB'} g$ is more specific than $c_1 \vee c_2$. And since each element of MLB' is more specific than i, $\bigvee_{MLB'} g$ is more specific than i. Consequently, $[\bigvee_{MLB'} g]_r$ is the only element in $mlb(c_1 \vee c_2 , i)$.
   - Finally, $mlb(c_1 , i) \subseteq MLB'$ and $mlb(c_2 , i) \subseteq MLB'$. Consequently, $MLB \subseteq MLB'$. By definition of mlb, there exists for every $g' \in MLB'$ a $g \in MLB$ such that $g' \preccurlyeq g$. This means that g' will not be included in the reduction of $\bigvee_{MLB'} g$, if $g' \notin mlb(c_1 , i) \cup mlb(c_2 , i)$. Consequently, only the elements of MLB can be included in the reduction of $\bigvee_{MLB'} g$, i.e., the reduction of $\bigvee_{MLB'} g$ is $[\bigvee_{MLB} g]_r$.

3. We assume $c_1 \vee c_2 \preccurlyeq_d i$. We will label this (3.1). The other case is trivial.

   ($\supseteq$) Firstly, from Lemma 4.21 follows that $c_1 \vee c_2 \vee s$ is more general than $c_1 \vee c_2$. Therefore $[c_1 \vee c_2 \vee s]_r$ is also more general than $c_1 \vee c_2$.

Secondly, assume there is some $[c_1 \vee c_2 \vee s]_r$ such that

- $\neg(c_1 \vee c_2 \vee s \preccurlyeq_d i)$, and
- $[c_1 \vee c_2 \vee s]_r \notin msg_d(c_1 \vee c_2 , i)$.

The latter means that there exists $d \in \mathcal{DL}_C$ such that

- $c_1 \vee c_2 \preccurlyeq_d d$ (3.2), and
- $\neg(d \preccurlyeq_d i)$ (3.3), and
- $d \prec_d [c_1 \vee c_2 \vee s]_r$ (3.4).

Because of (3.4) each disjunct c' of d is more specific than $c_1 \vee c_2$, or more specific than s. Because of (3.1), we then have that each disjunct c' of d is more specific than i or more specific than s. Because of (3.2) and (3.3), there must be at least one disjunct c' which is strictly more specific than s and not more specific than i. This contradicts the fact that $s \in msg(\perp , i)$.
$(\subseteq)$
Suppose $d \in msg_d(c_1 \vee c_2 , i)$ such that d cannot be written as $[c_1 \vee c_2 \vee s]_r$. Then we have:

- $c_1 \vee c_2 \preccurlyeq_d d$ (3.5), and
- $\neg(d \preccurlyeq_d i)$ (3.6).

~~Because of (3.6), there exists a disjunct c' in d which is not more specific than i,~~ i.e., $c' \preccurlyeq d$ and $\neg(c' \preccurlyeq i)$. Then $[c_1 \vee c_2 \vee c']_r$ is more specific than d because of (3.5). Since $d \in msg_d(c_1 \vee c_2 , i)$, we have $[c_1 \vee c_2 \vee c']_r = d$. But since d is not of the form $[c_1 \vee c_2 \vee s]_r$ with $s \in msg(\perp , i)$, $c' \notin msg(\perp , i)$. On the other hand $\neg(c' \preccurlyeq i)$, so there must exist $c'' \in msg(\perp , i)$ such that $c'' \prec c'$. But then we have:

- $c_1 \vee c_2 \vee c'' \prec d$, and thus $[c_1 \vee c_2 \vee c'']_r \prec d$;
- $c_1 \vee c_2 \preccurlyeq c_1 \vee c_2 \vee c''$, and thus $c_1 \vee c_2 \preccurlyeq [c_1 \vee c_2 \vee c'']_r$;
- $\neg(c_1 \vee c_2 \vee c'' \preccurlyeq i)$, and thus $\neg([c_1 \vee c_2 \vee c'']_r \preccurlyeq i)$.

This contradicts the assumption that $d \in msg_d(c_1 \vee c_2 , i)$.

4. We assume $\neg(i \preccurlyeq_d c_1 \vee c_2)$ (4.1), the other case is trivial.
$(\supseteq)$ Firstly, from Lemma 4.21 follows that $c_1 \vee c_2 \vee s$, and therefore also $[c_1 \vee c_2 \vee s]_r$, is more general than $c_1 \vee c_2$.
Secondly, assume there is some $[c_1 \vee c_2 \vee s]_r$ such that

- $i \preccurlyeq_d c_1 \vee c_2 \vee s$, and
- $[c_1 \vee c_2 \vee s]_r \notin mub_d(c_1 \vee c_2 , i)$.

The latter means that there exists $d \in \mathcal{DL}_C$ such that

- $c_1 \vee c_2 \preccurlyeq_d d$ (4.2), and
- $i \preccurlyeq_d d$ (4.3), and
- $d \prec_d [c_1 \vee c_2 \vee s]_r$ (4.4).

Because of (4.4) each disjunct c' of d is more specific than $c_1 \vee c_2$, or more specific than s. For each disjunct c' of d that is more specific than $c_1 \vee c_2$, we have $\neg(i \preccurlyeq c')$, because of (4.1). Because of (4.2) and (4.3) , there must be at least one disjunct c' of d which is strictly more specific than s and more general than i. This contradicts the fact that $s \in mub(\perp , i)$.

$(\subseteq)$

Suppose $d \in mub_d(c_1 \vee c_2 , i)$ such that d cannot be written as $[c_1 \vee c_2 \vee s]_r$. Then we have:

- $c_1 \vee c_2 \preccurlyeq_d d$ (4.5), and
- $i \preccurlyeq_d d$ (4.6).

Because of (4.6), there exists a disjunct c' in d which is more general than i, i.e., $c' \preccurlyeq d$ and $i \preccurlyeq c'$. Then $[c_1 \vee c_2 \vee c']_r$ is more specific than d because of (4.5). Since $d \in mub_d(c_1 \vee c_2 , i)$, we have $[c_1 \vee c_2 \vee c']_r = d$. But since d is not of the form $[c_1 \vee c_2 \vee s]_r$ with $s \in mub(\perp , i)$, we then have $c' \notin mub(\perp , i)$. On the other hand $i \preccurlyeq c'$, so there must exist $c'' \in mub(\perp , i)$ such that $c'' \prec c'$. But then we have:

- $c_1 \vee c_2 \vee c'' \prec d$, and thus $[c_1 \vee c_2 \vee c'']_r \prec d$;
- $c_1 \vee c_2 \preccurlyeq c_1 \vee c_2 \vee c''$, and thus $c_1 \vee c_2 \preccurlyeq [c_1 \vee c_2 \vee c'']_r$;
- $i \preccurlyeq c_1 \vee c_2 \vee c''$, and thus $i \preccurlyeq [c_1 \vee c_2 \vee c'']_r$.

This contradicts the assumption that $d \in mub_d(c_1 \vee c_2 , i)$.

□

This result can be extended to disjuncts of more than two elements of $\mathcal{L}_C$ as well. Consequently, DI or ITVS can be applied with the refinement operators mub_d, mgs_d, mlb_d and msg_d to compute $\mathcal{DS}$ and $\mathcal{DG}$.

The importance of Theorem 4.23 is that it describes the set of all consistent disjunctive concept representations by means of its boundary sets in terms of $\mathcal{L}_C$ and $\preccurlyeq$. The relevance of Theorem 4.25 is that it implements the operations msg_d, mgs_d, mlb_d and mub_d of DI and ITVS, such that these algorithms can be applied to find a solution for the concept learning problem in the language $\mathcal{DL}_C$. However, there are some practical limitations and conceptual objections to this approach:

- as argued in Chapter 3 the size of $G = \mathcal{G}_{I_g}$ might be exponential in the number of elements in I_g. Consequently, the number of disjuncts in $\bigvee_G c$ is exponential in the number of elements in I_g. Therefore it is often impractical to compute $\mathcal{DG}$. On the other hand, a lot of these disjuncts are consistent with information elements that are also consistent with other disjuncts. So from a practical point of view, one can argue that it is not necessary to compute G completely to obtain one consistent solution;

- disjuncts in elements of $\mathcal{DS}$ will never be generalized w.r.t. $\preccurlyeq$. The only way elements in $\mathcal{DS}$ are generalized is by adding more disjuncts (see Theorem 4.25). In this way, it will only be possible to correctly classify very few unseen information elements as s-bounds, because $\mathcal{DS}$ contains the disjunction of maximally specific concept representations in $\mathcal{L}_C$ which are consistent with probably only one s-bound. If the single representation trick holds, for instance, and I_s contains only positive lowerbounds, $mub(\perp , i) = \{ i \}$, for each $i \in I_s$. This means that $\mathcal{DS}$ contains only one element: the reduction of the disjunction of all positive lowerbounds.

- also, if the target concept were non-disjunctive (i.e., consisting of only one disjunct), a representation with one disjunct would never be found because of the introduction of disjuncts.

On the other hand, Theorem 4.23 and Theorem 4.25 will form the basis for more practical solutions. Two methods to come to a more feasible approach will be used. In a first step (Section 4.3) we only *approximate* $\mathcal{DS}$ by relaxing the "maximally specific" requirement to an "almost maximally specific" requirement. In a second step (Section 4.4) we impose additional *preference criteria* (see Chapter 2) on $\mathcal{DL}_C$ to select the preferred concept descriptions first.

4.3 Disjunctive Description Identification algorithm

4.3.1 Almost maximally specific concept representations

Regarding the feasibility of computing $\mathcal{DS}$ and $\mathcal{DG}$, we could argue that the problem at the $\mathcal{DG}$-side is less complex than at the $\mathcal{DS}$-side. The problem on both sides is that there are much more disjuncts in the concept representations than desired. On the one hand, if the disjunction on $\mathcal{DG}$ contains too many disjuncts, it is easy to specialize it, while remaining consistent with I, by removing some of the disjuncts. As long as for each s-bound i one of the disjuncts is consistent with i, consistency is guaranteed. On the other hand, removing disjuncts of the elements of $\mathcal{DS}$ would also specialize them. However, since they are already maximally specific in $\mathcal{DL}_C$, removing disjuncts will therefore make them inconsistent with some s-bound. The only way to remove disjuncts of $\mathcal{DS}$ consistently is to replace two or more by their minimal upperbound. However, the question then arises which of the disjuncts to combine. If a maximum of n disjuncts per concept representation were allowed, one could think of partitioning I_s in n partitions. On the one hand, this would require computation of *all possible partitions* of n elements, since there is no immediate reason to prefer one partition over another. On the other hand, the resulting concept representation would not necessarily be maximally specific (among the disjunctions of n disjuncts), because the minimal upperbound might be consistent with s-bounds that were not assigned to it.

Example 4.26 In Figure 4.4, e.g., if $I_s = \{i_1, i_2, i_3, i_4, i_5\}$ is partitioned into $\{\{i_1, i_2\}, \{i_3, i_4, i_5\}\}$, the minimal upperbound of i_1 and i_2 (i.e., c_1) is also consistent with i_3. Consequently, $c_1 \vee c_2$, where c_2 is the minimal upperbound of i_3 and i_4, is more general than another disjunction with two disjuncts: $c_1 \vee c_3$. ◇

The difference with the proof of Theorem 4.25 (where we also assigned an element of $\mathcal{L}_C$ to each s-bound) is that we now have to *generalize* these concept representations in order to be consistent with *more than one* s-bound, thereby allowing consistency with yet other s-bounds. In order to obtain a maximally specific concept representation we could require the minimal upperbound to be consistent with *only* those s-bounds assigned to it. The problem would then be that *not all* maximally specific disjunctions would be found, in particular those where two minimal upperbounds *both* are consistent with a certain s-bound i (while i would only be assigned to one of them).

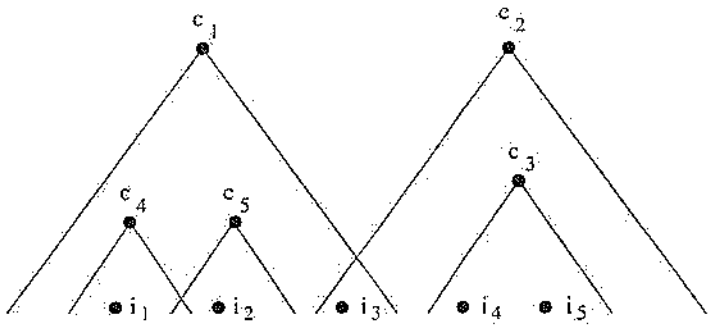


Figure 4.4 Problems with maximally specific concept representations

Example 4.27 Consider again Figure 4.4. If we require that the minimal upperbound of i_1 and i_2 must only be consistent with i_1 and i_2, and must be inconsistent with the s-bounds that are not assigned to it (i.e., i_3), the disjunction $c_1 \vee c_2$ would indeed not be allowed.

However, if there would be no c_3, consistent with i_4 and i_5 only, every minimal upperbound of i_4 and i_5 (i.e., c_2) would be consistent with i_3 as well. Therefore the disjunction $c_1 \vee c_2$ would not be allowed (because i_3 is always consistent with both disjuncts), although it would be maximally specific and consistent with I_s. $\diamond$

In summary, it is impossible to constructively find all maximally specific concept representations with n disjuncts. The problem of assigning s-bounds to disjuncts is typical for specific-to-general approaches. It also makes their result dependent on the order of the presented information elements, because they usually do not consider all possible partitions of I_s. Because of these difficulties we will *approximate* maximally specific concept representations. For each disjunct c_j in the maximally general concept representation, we will require that c'_j is a maximally specific concept representation consistent with *all* s-bounds c_j is consistent with. This liberates us from the problem of having to "distribute" the s-bounds over the available c_j, by computing all possible partitions of I_s. Dropping disjuncts is then possible by removing corresponding disjuncts c_j and c'_j, since both are consistent with the same set of s-bounds. However, whenever c_j turns out to be overly general, c'_j might be overly general as well, and will have to be recomputed (see further).

To formalize the idea of couples $\langle c_j , c'_j \rangle$ of disjuncts, we introduce the following definitions.

Definition 4.28 (Almost maximally specific concept representation) Given a disjunction $d = c_1 \vee \cdots \vee c_n$ and a set I_s of s-bounds, $c'_1 \vee \cdots \vee c'_n$ is called almost maximally specific under d w.r.t. I_s iff $\forall j,\ 1 \leq j \leq n :: c'_j \in \{ s \in S_j \mid s \preccurlyeq c_j \}$ where S_j is the set of all maximally specific elements in $\mathcal{L}_C$ consistent with the set $\{ i \in I_s \mid c_j \sim i \}$, for all $1 \leq j \leq n$.

Because the set I_s will usually be the set of all s-bounds known, we will very often drop "w.r.t. I_s". Note that if s is almost maximally specific under d, then s has the same number of disjuncts as d. Also note that the definition applies in particular to a disjunction with one disjunct: given $c \in \mathcal{L}_C$ and I_s, $c' \in \mathcal{L}_C$ is almost maximally specific under c (w.r.t. I_s), if c' is consistent with all elements of I_s c is consistent with. In this case we have a pair consisting of a maximally general concept representation c and of a maximally specific concept representation c' both consistent with I. This means the restriction to almost maximally specific concept representations is still an extension of the non-disjunctive case; i.e., $c \in \mathcal{G}_I$, and $c' \in \mathcal{S}_I$.

Definition 4.29 (Subdisjunction) $\forall c_1, \ldots, c_n, c'_1, \ldots, c'_m \in \mathcal{L}_C$:

$$c_1 \vee \cdots \vee c_n \text{ is a subdisjunction of } c'_1 \vee \cdots \vee c'_m \text{ iff}$$
$$\{ c_1, \ldots, c_n \} \subseteq \{ c'_1, \ldots, c'_m \}$$

Notation 4.30 $c_1 \vee \cdots \vee c_n$ is a subdisjunction of $c'_1 \vee \cdots \vee c'_m$ is denoted as

$$c_1 \vee \cdots \vee c_n \sqsubseteq c'_1 \vee \cdots \vee c'_m.$$

$d_1 \sqsubset d_2$ denotes $d_1 \sqsubseteq d_2$ and $d_1 \neq d_2$.

Through its definition, $\sqsubseteq$ inherits all its properties from $\subseteq$.

In the above discussion we explained why we will restrict ourselves to almost maximally specific disjunctive concept representations under a subdisjunction of the maximally general disjunctive concept representation. Therefore we will introduce the following terminology.

Definition 4.31 (Almost all consistent disjunctions)

- The set of all subdisjunctions of the maximally general disjunction is denoted by $\mathcal{DG}_{\sqsubseteq}$:

$$\mathcal{DG}_{\sqsubseteq} = \{ d \in \mathcal{DL}_C \mid G \subseteq \mathcal{G}_{I_g} \text{ and } d = \bigvee_G c \text{ and } d \sim I_s \}.$$

- The set of all almost maximally specific concept representations under an element of $\mathcal{DG}_{\sqsubseteq}$ is denoted by $\mathcal{ADS}_{\sqsubseteq}$:

$$\mathcal{ADS}_{\sqsubseteq} = \{ d \in \mathcal{DL}_C \mid \exists g \in \mathcal{DG}_{\sqsubseteq} : d \text{ is almost maximally specific under } g \}.$$

- Finally the set of all disjunctions between an element of $\mathcal{DG}_{\sqsubseteq}$ and an element of $\mathcal{ADS}_{\sqsubseteq}$ is denoted by $\mathcal{ADVS}_{\sqsubseteq}$:

$$\mathcal{ADVS}_{\sqsubseteq} = \{ d \in \mathcal{DL}_C \mid \exists g \in \mathcal{DG}_{\sqsubseteq}, \exists s \in \mathcal{ADS}_{\sqsubseteq} : s \preccurlyeq_d d \preccurlyeq_d g \text{ and}$$
$$s \text{ is almost maximally specific under } d \text{ and } g \}.$$

Figure 4.5 illustrates this situation. The upper half of the figure shows the top of $\mathcal{L}_C$ and the concept representations just below the top. Each leaf of the upper part is a maximally general element in $\mathcal{L}_C$ consistent with I_g. The disjunction of all leaves is the maximally general disjunctive concept representation consistent with I. One particular subdisjunction is illustrated by the large bullet points. For each of the disjuncts c_j in this subdisjunction, the lower part of the figure shows the lower part of $\mathcal{L}_C$ more specific than

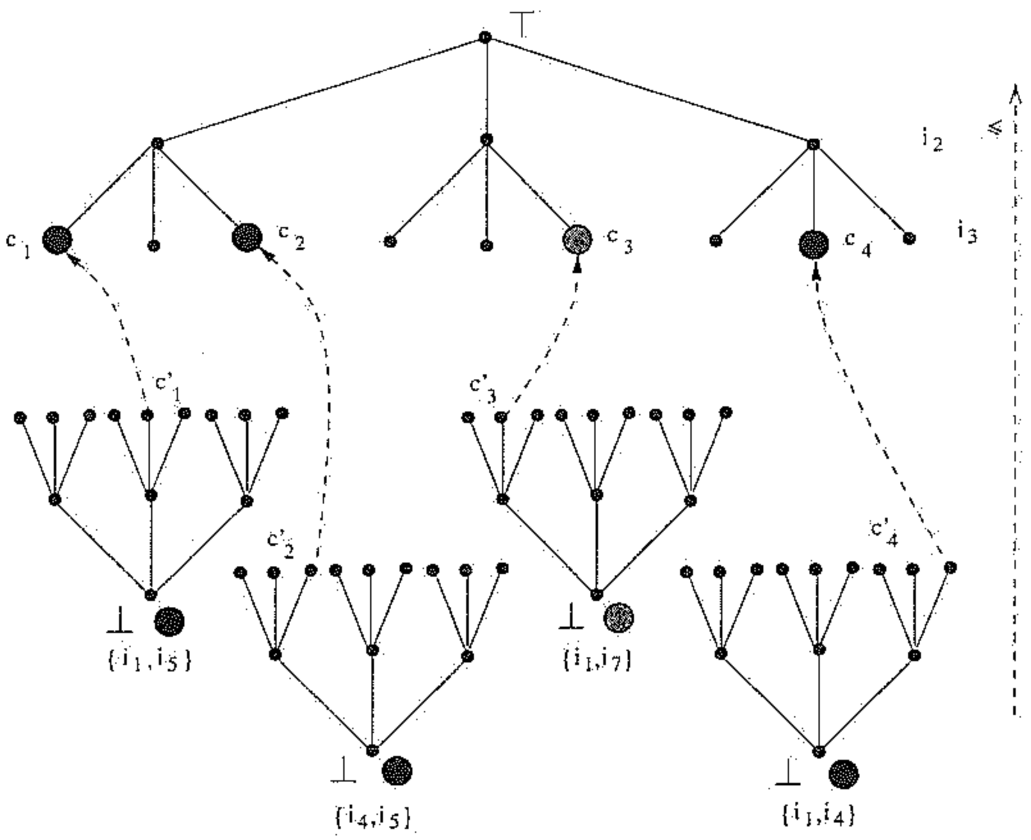


Figure 4.5 The idea behind the Disjunctive DI algorithm

c_j[2]. One particular almost maximally specific disjunction under g is shown: the disjunction $c'_1 \vee c'_2 \vee c'_3 \vee c'_4$. Each of the couples (c_j , c'_j) is consistent with a particular subset of I_s. The set of s-bounds consistent with c_1, for instance, is $\{ i_1 , i_5 \}$. Furthermore, c'_1 is more specific than c_1 and consistent with i_1 and i_5, but not consistent with the other s-bounds, since c_1 is not consistent with other s-bounds. All c_j (and therefore also all c'_j) are consistent with all g-bounds. $c_1 \vee c_2 \vee c_3 \vee c_4$ is an element of $\mathcal{DG}_{\sqsubseteq}$; consequently $c'_1 \vee c'_2 \vee c'_3 \vee c'_4$ is in $\mathcal{ADS}_{\sqsubseteq}$.

The set $\mathcal{ADVS}_{\sqsubseteq}$ contains less elements than $\mathcal{DVS}$, because its lowerbound is the set of all almost maximally specific concept representations under an element of $\mathcal{DG}_{\sqsubseteq}$. In this sense, some solutions of the original problem are lost. On the other hand, this does not mean that these solutions cannot be found when the set I is extended with a newly provided g-bound i. Consider, for instance, Figure 4.6. Suppose $I_g = \{i_2, i_3\}$ and $I_s = \{i_1, i_4, i_5\}$. Also suppose c_1 is maximally general and consistent with I (i.e., c_1 is a consistent subdisjunction containing only one disjunct). The concept representation c'_1

[2]We made four separate drawings to avoid overloading the figure.

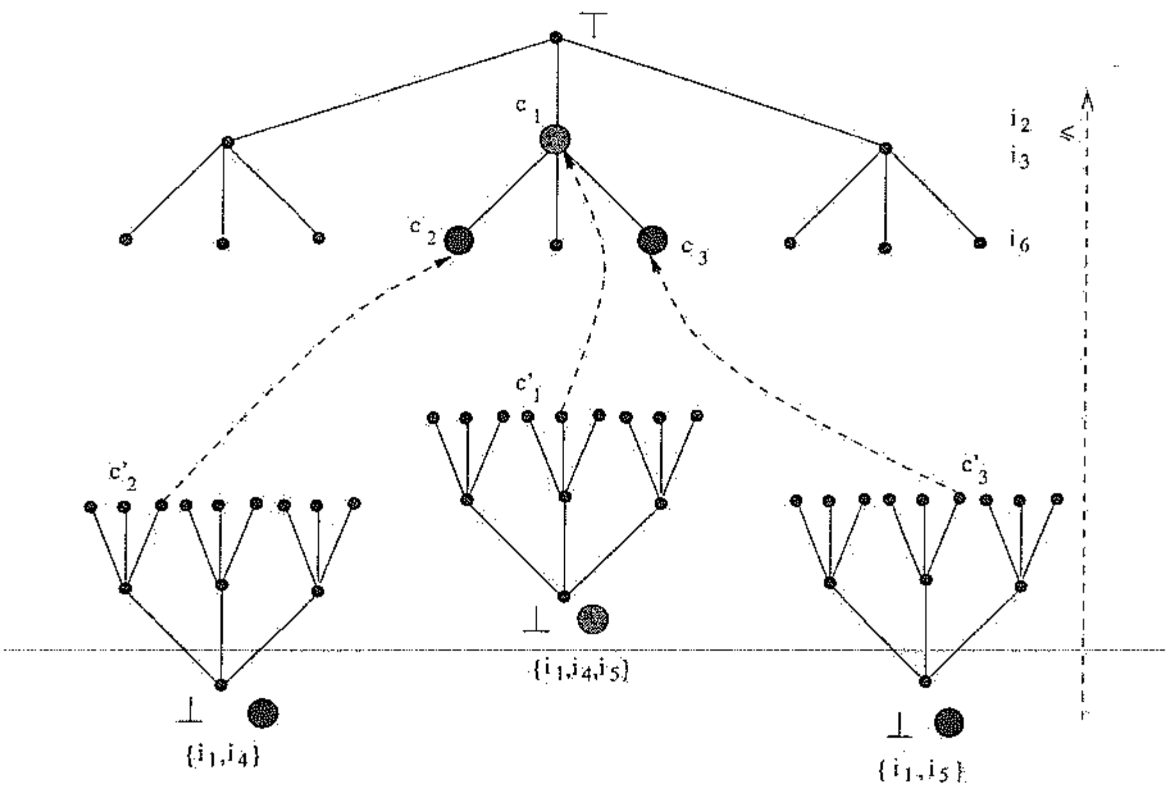


Figure 4.6 $\mathcal{ADVS}_{\sqsubseteq,I}$ is non-monotonic w.r.t. $\subseteq$

is almost maximally specific under c_1. Now suppose a new g-bound i_6 is known, and $\neg(c_1 \sim i_6)$. Therefore $c_1 \notin \mathcal{G}_{I_g \cup \{i_6\}}$. Suppose c_2 and c_3 are specializations of c_1, such that $c_2, c_3 \in \mathcal{G}_{I_g \cup \{i_6\}}$, $c_2 \sim \{ i_1 , i_4 \}$, and $c_3 \sim \{ i_1 , i_5 \}$. The disjunction $c_2' \vee c_3'$ is maximally specific under $c_2 \vee c_3$. Consequently $c_2' \vee c_3' \in \mathcal{ADVS}_{\sqsubseteq, I \cup \{i_6\}}$, but (as drawn in the figure) $c_2' \vee c_3' \notin \mathcal{ADVS}_{\sqsubseteq,I}$. In this way all disjunctions consistent with $I \cup \{ i_6 \}$ can still be found, but they are not necessarily included in $\mathcal{ADVS}_{\sqsubseteq,I}$, i.e., $\mathcal{ADVS}_{\sqsubseteq, I \cup \{i_6\}} \not\subseteq \mathcal{ADVS}_{\sqsubseteq,I}$. In a sense, computing $\mathcal{ADVS}_{\sqsubseteq,I}$ is *non-monotonic* w.r.t. $\subseteq$ when adding information elements to I.

A bidirectional method to compute all possible couples (c_j , c_j') can then be outlined as follows: since maximally general disjunctions are easier to describe than maximally specific ones, we start from maximally general disjunctions. Maximally general disjunctions consist of maximally general disjuncts (see Theorem 4.22). In order not to lose *possible* disjuncts, we are not allowed to prune any maximally general disjuncts as in ITVS. In ITVS maximally general disjuncts not consistent with I_s could be pruned, because none of their specializations would be consistent with I_s (see Section 3.6.3). In the disjunctive case, even if a maximally general concept representation is inconsistent with all *known* s-bounds, it might still be consistent with *future*, and thus *unknown* s-bounds. Almost maximally specific disjunctions consist of maximally specific disjuncts, one for each maximally general disjunct. These maximally specific disjuncts will be consistent with all s-bounds the corresponding maximally general disjunct is consistent with.

The *Disjunctive Description Identification* algorithm (DDI) is a breadth-first implementation of this method, and therefore a disjunctive version of the DI algorithm. In Section 4.5 we present the *Disjunctive Iterative Versionspace* algorithm (DITVS), which is a depth-first implementation of this method (i.e., a disjunctive version of the ITVS algorithm).

Related work

The idea of how to compute maximally specific concept representations is inspired by the *Multiple Convergence* method of the system HYDRA [Murray, 1987a]. Our approach is a generalization of Multiple Convergence in the sense that Multiple Convergence is a disjunctive extension of the Candidate Elimination algorithm, while we will extend DI ([Mellish, 1991]; see Chapter 3). This means our algorithm allows upperbounds as information elements, while Multiple Convergence only allows examples (i.e., lowerbounds). Furthermore Multiple Convergence only works for conjunctive attribute-value languages with k features, while we will extend it to arbitrary languages. Some of the advantages of Multiple Convergence will be preserved: since we assume our algorithm to work incrementally, it is possible to use partially learned knowledge for problem solving. When partially learned knowledge must be updated, we preserve the integrity of the partially learned knowledge as much as possible, by only minimal generalization or specialization steps. Furthermore, storing couples $\langle c_j\ ,\ c'_j\ \rangle$ of disjuncts, allows disjuncts to be handled independently as separate and distinct concepts in $\mathcal{L}_C$. Nevertheless our approach suffers from one big disadvantage: while an overly general c'_j can be specialized in the Multiple Convergence method, because of the restriction on the language $\mathcal{L}_C$, we will have to recompute c'_j, because of the independence of the choice of $\mathcal{L}_C$.

An important aspect of HYDRA, based on the solution disjunction, is the introduction of new concepts represented by one of the resulting disjuncts. The introduction of these new concepts in HYDRA is motivated by the fact that concept representations that maximize inclusiveness while remaining conjunctive, often represent basic concepts in the domain of the target concept [Murray, 1987b]. In an Inductive Logic Programming context, the introduction of new concepts is called *predicate invention*. Related to the approach of [Murray, 1987a] is the *intra-construction* operator of [Muggleton and Buntine, 1988]. It would be intresting for future work to investigate how these approaches are related, and whether they could be described in a language-independent way.

[Hirsh, 1990] informally describes some ideas to learn several disjunctions in the context of the Incremental Versionspace Merging algorithm also. In his approach he considers a disjunctive versionspace as a set of versionspaces (as we will also do further on). However, he requires these versionspaces to be consistent with disjoint sets of s-bounds (as in Example 4.26). Because of the problem described in Example 4.26, this will not always be possible. However, Hirsh does not mention that this could be a problem.

4.3.2 The Disjunctive Description Identification algorithm

We will now present the Disjunctive Description Identification algorithm.

Datastructures

We introduce the following datastructures:

- DVS is a set of (conjunctive) versionspaces vs;
- each (conjunctive) versionspace vs consists of three parts: a maximally general concept representation $g \in \mathcal{L}_C$, the set J_s of indexes in I_s of s-bounds consistent with g, and the set S of almost maximally specific concept representations under g. A versionspace is represented by a term $vs(\ g\ ,\ J_s\ ,\ S\ )$;
- all s-bounds are stored in I_s; the number of s-bounds is n_s.

An important difference with DI is the fact that all s-bounds will have to be stored, because specializing overly general maximally specific concept representations will involve reprocessing s-bounds (see further). As in HYDRA ([Murray, 1987a]; see above) we use an indexing mechanism (the set J_s) to store which s-bounds are consistent with each g. Whenever we specialize g, the specializations can only be consistent with s-bounds whose index is in J_s; the other elements of I_s do not have to be reprocessed. No g-bounds have to be stored because they need not be reprocessed.

We say that a versionspace $vs(\ g\ ,\ J_s\ ,\ S\ )$ is consistent with i, resp. I, iff $g \sim i$, resp. $g \sim I$. Also s is almost maximally specific *in* a versionspace $vs(\ g\ ,\ J_s\ ,\ S\ )$ iff $s \in S$.

Invariants

To express the invariants we introduce some extra notation.

Notation 4.32 $I_s \mid J_s$ is the set of all s-bounds in I_s whose index is in J_s.

Notation 4.33 We denote $(\ \bigcup_{vs(\ g\ ,\ J_s\ ,\ S\ ) \in DVS} I_s|J_s\ )$ by $\bigcup_{DVS} J_s$.

We impose the following invariants on the datastructures:

- **Invariant 4.3.1.** $\{\ g \mid vs(\ g\ ,\ J_s\ ,\ S\ ) \in DVS\ \} = \mathcal{G}_{I_g}$.
- **Invariant 4.3.2.** For all $vs(\ g\ ,\ J_s\ ,\ S\ )$ in DVS, S is the set of all almost maximally specific concept representations under g.
- **Invariant 4.3.3.** For all $vs(\ g\ ,\ J_s\ ,\ S\ )$ in DVS, $J_s = \{\ i \in I_s \mid g \sim i\ \}$.
- **Invariant 4.3.4.** $\bigcup_{DVS} I_s|J_s = I_s$.

For each maximally general concept representations in $\mathcal{L}_C$ consistent with I_g, there exists a versionspace in DVS. In each versionspace $vs(\ g\ ,\ J_s\ ,\ S\ )$ the set S contains all almost maximally specific concept representations under g, and the set J_s contains an index to each s-bounds consistent with g. Finally, all s-bounds are consistent with at least one versionspace of DVS.

After having computed DVS, $\mathcal{DG}_{\sqsubseteq}$ and $\mathcal{ADS}_{\sqsubseteq}$ can be computed from DVS, according to their definition (Definition 4.31).

The main algorithm [T]

SUMMARY: in this section we describe the main algorithm of DDI.

We will first discuss procedure DDI itself (see Algorithm 4.1). The input is, as usual, the stream of information elements $\mathcal{Inf}$. Initially the set DVS contains only one element, namely $vs(\ \top\ ,\ \emptyset\ ,\ \{\ \bot\ \}\ )$. This initialization fulfills all invariants. As DI, DDI is incremental, i.e., it reads unprocessed information elements from $\mathcal{Inf}$ and processes them one by one. Again we will split up the discussion, depending on the information element read (i.e., i) being an s-bound or a g-bound.

If i is an s-bound, it has to be stored in I_s first (see Step 4.1). Then DVS will be split up in a set DVS_{cons} of versionspaces consistent with i, and DVS_{incons} of versionspaces inconsistent with i. Initially DVS_{cons} and DVS_{incons} are both empty (see Step 4.2). Each $vs(\ g\ ,\ J_s\ ,\ S\ )$ consistent with i must be added to DVS_{cons}. Before adding it, however, it will be updated, such that it fulfills the invariants. Therefore the index of i in I_s, which is n_s, is added to J_s (see Step 4.3; cf. Invariant 4.3.3). Then, if i is a positive lowerbound, $generalize\_all(\ S\ ,\ i\ )$ (see Algorithm 3.2) returns the union of $mub(\ s\ ,\ i\ )$ for all $s \in S$ (see Step 4.4). Otherwise, if i is a negative upperbound, $generalize\_all(\ S\ ,\ i\ )$ returns the union of $msg(\ s\ ,\ i\ )$ for all $s \in S$. Not all elements returned by generalize_all are almost maximally specific under g, which is required by Invariant 4.3.2. Therefore only those more specific than g and maximally specific are selected (see Step 4.5). Then $vs(\ g\ ,\ J_s\ ,\ S\ )$ is added to DVS_{cons} (see Step 4.6). Each $vs(\ g\ ,\ J_s\ ,\ S\ )$ inconsistent with i is just added to DVS_{incons} (see Step 4.7). Consequently Invariant 4.3.3 and Invariant 4.3.2 still hold for these versionspaces. If at this point DVS_{cons} is empty, no versionspace of DVS was consistent with i, and because of Invariant 4.3.4 this means no disjunction of $\mathcal{DL}_C$ can be consistent with I. Therefore DDI will fail in this case (see Step 4.8). Otherwise the union of DVS_{cons} and DVS_{incons} is assigned to DVS (see Step 4.9), and Invariant 4.3.4 is fulfilled. Since all elements of the previous value DVS were added to either DVS_{cons} or DVS_{incons}, Invariant 4.3.1 still holds. Therefore all invariants will also hold at the end of the while loop.

If i is a g-bound, all versionspaces not consistent with i must be specialized. This is done in $d\_specialize\_all(\ DVS\ ,\ i\ )$. The procedure d_specialize_all (see Algorithm 4.2) returns, for a given set DVS of versionspaces and an information element i, two sets of versionspaces. The first set is the subset of DVS consistent with i. The second set is derived from all other versionspaces $vs(\ g\ ,\ J_s\ ,\ S\ )$ of DVS (i.e., those not consistent with i). It contains all versionspaces $vs(\ g'\ ,\ J_s\ ,\ \{\ \bot\ \}\ )$ where g' is a maximally general specialization of g consistent with i. The third argument of the versionspace can *in general* only be recomputed from $\{\ \bot\ \}$, because some elements of S might be overgeneral (since g is overgeneral) or consistent with some s-bound g' is not consistent with. Therefore the set S corresponding to g' can in general only be computed by reprocessing all s-bounds consistent with g'. In the d_specialize_all the second and the third argument of the versionspaces $vs(\ g'\ ,\ J_s\ ,\ \{\ \bot\ \}\ )$ are not yet computed (i.e., they do not necessarily satisfy Invariant 4.3.2 and Invariant 4.3.3): at this time g' is not guaranteed to be maximally general, and might therefore not be included in DVS after all.

The set of all consistent versionspaces in DVS, returned by $d\_specialize\_all(\ DVS\ ,\ i\ )$, is assigned to DVS; the set of "newly specialized versionspaces" is assigned to DVS_{new} (see Step 4.10). At this point, all elements in DVS fulfill Invariant 4.3.3 and Invariant 4.3.2, because they were in DVS before and they are unchanged. Furthermore, for all $vs(\ g\ ,\ J_s\ ,\ S\ )$ in DVS, g is maximally general in DVS and also in DVS_{new}. If it were not, Invariant 4.3.1 would contradict the fact that $vs(\ g\ ,\ J_s\ ,\ S\ )$ was in DVS before. Elements $vs(\ g\ ,\ J_s\ ,\ S\ )$ in DVS_{new} are not necessarily fulfilling Invariant 4.3.1 and Invariant 4.3.2. Only those with g not strictly more specific than the "g" of an element in DVS or in DVS_{new} are consistent with all g-bounds and maximally

```
procedure DDI ( $\mathcal{I}nf$: stream of info )
          returns set of disjunctive concept, set of disjunctive concept
   DVS := { $vs( \top , \emptyset , \{ \bot \} )$ }
   $n_s := 0$
   while there are still information elements to be processed
       do $i := read( \mathcal{I}nf )$
          if $i$ is an $s$-bound
          then $n_s := n_s + 1$; $I_s[n_s] := i$  {4.1}
               $DVS_{incons} := \emptyset$; $DVS_{cons} := \emptyset$  {4.2}
               for all $vs( g , J_s , S ) \in DVS$
                   do if $g \sim i$
                      then $J_s := J_s \cup \{ n_s \}$  {4.3}
                           $S := generalize\_all( S , i )$  {4.4}
                           $S :=$ select all $s$ from $S$
                                 with $s \preccurlyeq g$ and $\neg\exists s' \in S : s' \prec s$  {4.5}
                           $DVS_{cons} := DVS_{cons} \cup \{ vs( g , J_s , S ) \}$  {4.6}
                      else $DVS_{incons} := DVS_{incons} \cup \{ vs( g , J_s , S ) \}$  {4.7}
               endfor
               if $DVS_{cons} = \emptyset$
               then failure  {4.8}
               else $DVS := DVS_{incons} \cup DVS_{cons}$  {4.9}
          else {$i$ is a $g$-bound}
               $DVS, DVS_{new} := d\_specialize\_all( DVS , i )$  {4.10}
               for all $vs( g , J_s , S ) \in DVS_{new}$
                   do if $\neg\exists\, vs( g' , J'_s , S' ) \in DVS \cup DVS_{new} : g \prec g'$
                      then $J_s :=$ select all $ind$ from $J_s$ with $g \sim I_s[ind]$  {4.11}
                           $S := d\_generalize\_all( g , J_s )$  {4.12}
                           $DVS := DVS \cup \{ vs( g , J_s , S ) \}$
               endfor
               if $\bigcup_{DVS} J_s \neq I_s$ then failure {4.13}
   endwhile
   $DG := \{ g \mid g = g_1 \vee \cdots \vee g_n$ and $\forall j,\ 1 \leq j \leq n : vs( g_j , J_{s,j} , S_j ) \in DVS$ and $g \sim I_s \}$
   $ADS := \{ s_1 \vee \cdots \vee s_l \mid g_1 \vee \cdots \vee g_l \in DG$ and
          $\forall j,\ 1 \leq j \leq l : vs( g_j , J_{s,j} , S_j ) \in DVS$ and $\forall j,\ 1 \leq j \leq l : s_j \in S_j \}$
   return $DG$, $ADS$
endproc
```

Algorithm 4.1 Disjunctive Description Identification algorithm (DDI)

specific (cf. Invariant 4.3.1). According to the specification of d_specialize_all the set of s-bounds g is consistent with is a subset of J_s, since J_s is the set of s-bounds a generalization of g is consistent with. So the subset of J_s consistent with g is assigned to J_s (cf. Invariant 4.3.3; see Step 4.11). Then *d_generalize_all*(g , J_s) assigns the set of almost maximally specific concept representations under g to S (cf. Invariant 4.3.2; see Step 4.12). In general, the procedure d_generalize_all returns, for a given concept g and a set J_s of indexes to s-bounds, the set S of all almost maximally specific concept representations under g.

Finally the newly computed versionspace vs(g , J_s , S) is added to DVS. After having added a newly computed versionspace for all vs(g , J_s , S) of DVS_{new} with maximally general g, Invariant 4.3.1 will be fulfilled as well. Since the sets J_s of the elements in DVS_{new} have been reduced, Invariant 4.3.4 may be violated. If it is, the disjunction of all elements of $\mathcal{G}_{I_g}$ is inconsistent with some s-bound; hence, no element of $\mathcal{DL}_C$ can be consistent with I (Theorem 4.25). So in that case DDI should fail (Step 4.13). Otherwise, Invariant 4.3.4 is fulfilled. Consequently all invariants will also hold at the end of the while loop.

Then $\mathcal{DG}_{\sqsubseteq}$ (in Algorithm 4.1 denoted by DG) is the set of all disjunctions $\bigvee_{j=1}^{n} g_j$ with $vs(g_j , J_s , S_j) \in DVS$ for all j, $1 \leq j \leq n$, and such that $\bigvee_{j=1}^{n} g_j$ is consistent with I_s. The set $\mathcal{ADS}_{\sqsubseteq}$ (in Algorithm 4.1 denoted by ADS) is the set of all corresponding almost maximally specific elements.

Generalization and Specialization in DDI [T]

SUMMARY: In this section we describe the generalization and specialization operations that are used in the Disjunctive Description Identification algorithm.

To establish the specifications of d_specialize_all is rather straightforward: each versionspace vs(g , J_s , S) from DVS consistent with i is added to DVS_{cons}. For all other versionspaces in DVS, the specified versionspaces based on the maximally general specializations of g (i.e., minimal upperbounds or most general specializations) are added to DVS_{new}. Then DVS_{cons} and DVS_{new} are returned.

The procedure d_generalize_all (see Algorithm 4.2) returns, for a given concept g and a set J_s of indexes to the s-bounds g is consistent with, the set of all almost maximally specific concept representations under g. The implementation of the procedure d_generalize_all is again straightforward, because the s-bounds consistent with g are given by J_s. The implementation uses the procedure generalize_all of Algorithm 3.2. Given an s-bound i, *generalize_all*(S , i) returns all maximally specific generalizations w.r.t. i of all elements in S.

In d_generalize_all S is computed by *recomputing* it from $\perp$. This step could be optimized, because recomputing S is only necessary in case J_s really changed in Step 4.11. If it did not change, the elements of S not consistent with i should just be removed from S.

Properties of DDI

Theorem 4.34 DDI fails iff there exists no disjunction in $\mathcal{DL}_C$ consistent with I.

Proof There are two places where DDI can fail. In both cases, Invariant 4.3.4 is violated, which means that there does not exist an element in $\mathcal{DL}_C$ consistent with I. If DDI does not fail, Invariant 4.3.1 and Invariant 4.3.4 imply that the disjunction of all elements of $\mathcal{G}_{I_g}$ is consistent with I. □

```
procedure d_specialize_all( $DVS$: set of versionspace; $i$: $g$-bound )
            returns set of versionspace, set of versionspace
    {Returns: $DVS_{cons}$: the set of elements in $DVS$ consistent with $i$;
    $DVS_{new}$: for all other versionspaces $vs( g , J_s , S )$ in $DVS$:
    the set of all versionspaces $vs( g' , J_s , \{ \perp \} )$
    where $g'$ is a maximally general specialization of $g$ consistent with $i$ }

    $DVS_{cons} := \emptyset$; $DVS_{new} := \emptyset$
    for all $vs( g , J_s , S ) \in DVS$
        do if $g \sim i$
            then $DVS_{cons} := DVS_{cons} \cup \{ vs( g , J_s , S ) \}$
            else $G := specializations( g , i )$
                 for all $g' \in G$
                     do $DVS_{new} := DVS_{new} \cup \{ vs( g' , J_s , \{ \perp \} ) \}$
                 endfor
    endfor
    return $DVS_{cons}$, $DVS_{new}$
endproc

procedure d_generalize_all( $g$: concept; $J_s$: set of index ) returns set of concept
    {Requires: $J_s = \{ i \in I_s \mid g \sim i \}$
    Returns: the set of all almost maximally specific concept representations under $g$ }

    $S := \{ \perp \}$
    for all $ind \in J_s$
        do $S := generalize\_all( S , I_s[ind] )$
           $S :=$ select all $s$ from $S$
                   with $s \preccurlyeq g$ and $\neg \exists s' \in S : s' \prec s$
    endfor
    return $S$
endproc
```

Algorithm 4.2 Generalization and Specialization in DDI

Theorem 4.35 If DDI does not fail, it returns the set of maximally general concept representations consistent with I, and the set of almost maximally specific concept representations consistent with I.

Proof The first part follows from Invariant 4.3.1, and the postprocessing step. The second part follows from Invariant 4.3.2, and the postprocessing step. □

Complexity analysis [T]

SUMMARY: in this section we describe the computational complexity of DDI.

For this analysis we are using the same notation as in Section 3.8.2. We express the complexity in terms of $\bar{g}$ (the size of the general-to-specific search space in $\mathcal{L}_C$) and $\bar{s}$ (the size of the specific-to-general search space in $\mathcal{L}_C$). The average branching factor of the general-to-specific search space is b_g, and of the specific-to-general search space b_s. Note however that the average branching factor b_g will be different than the one obtained in DI and ITVS (Section 3.8.2), because no g is pruned from DVS for being inconsistent with I_s. The total number of s-bounds is n_s; the total number of g-bounds is n_g. The memory complexity of an information element is c_i, and of a concept representation c_c. The sets J_s contain indices to information elements. We denote the space complexity of an index by c_{ind}. As in Chapter 3, we assume c_i, c_c and c_{ind} are constants. The time complexity of a specialization operation is c_{spec}, of a generalization operation c_{gen}, and of a $\preccurlyeq$-test $c_{\preccurlyeq}$. We also assume c_{spec}, c_{gen} and $c_{\preccurlyeq}$ to be constant.

Theorem 4.36 The worst case space complexity of DDI is

$$\mathcal{O}(\ b_g^{n_g} \times (1 + b_s^{n_s}) \times c_c + b_g^{n_g} \times n_s \times c_{ind} + (n_s + n_g) \times c_i\ ).$$

Proof In the worst case there are $b_g^{n_g}$ elements $vs(\ g\ , J_s\ , S\ )$ in DVS. For each of these DDI stores:

- g itself (this yields the term $\mathcal{O}(\ b_g^{n_g} \times c_c\ )$);
- the set J_s of indexes to the s-bounds consistent with g. In the worst case J_s contains an index to each s-bound (this yields the term $\mathcal{O}(\ b_g^{n_g} \times n_s \times c_{ind}\ )$);
- the set S of maximally specific concept representations more specific than g. In the worst case, S is equal to S_I, which contains $b_s^{n_s}$ elements (this yields the term $\mathcal{O}(\ b_g^{n_g} \times b_s^{n_s} \times c_c\ )$).

Finally there is an extra term $\mathcal{O}(\ (n_s + n_g) \times c_i\ )$, because all s-bounds and all g-bounds have to be stored. □

Theorem 4.37 The worst case time complexity of DDI is:

$$\mathcal{O}(\ \bar{g} \times c_{spec} + \bar{g} \times (b_g^{n_g} + n_s + \bar{s} \times (1 + b_s^{n_s})) \times c_{\preccurlyeq} + \bar{g} \times \bar{s} \times c_{gen}\ ).$$

Proof For each of the $\bar{g}$ elements g in the general-to-specific search space, we have in the worst case all following operations:

- g is specialized once (this yields the term $\mathcal{O}(\ \bar{g} \times c_{spec}\ )$);
- g must be checked to be maximally general. Therefore, it must be compared with the g of all $b_g^{n_g}$ other elements of DVS (this yields the term $\mathcal{O}(\ \bar{g} \times b_g^{n_g} \times c_{\preccurlyeq}\ )$);

- when g is a specialization of g', the set J_s is the set of all elements in J'_s such that the corresponding s-bound is consistent with g. Therefore g must be compared to all these corresponding s-bounds, which could, in the worst case, be all n_s s-bounds (this yields the term $\mathcal{O}(\ \bar{g} \times n_s \times c_{\preccurlyeq}\ )$).
- in the worst case, each specialization of g' to g leads to the recomputation of $S = \mathcal{S}_I$. This means that each of the $\bar{s}$ elements in the specific-to-general search space will be generalized once (yielding the term $\mathcal{O}(\ \bar{g} \times \bar{s} \times c_{gen}\ )$), will be compared to g once (yielding the term $\mathcal{O}(\ \bar{g} \times \bar{s} \times 1 \times c_{\preccurlyeq}\ )$), and will be compared to all $b_s^{n_s}$ other elements of S to check maximally specificity (yielding the term $\mathcal{O}(\ \bar{g} \times \bar{s} \times b_s^{n_s} \times c_{\preccurlyeq}\ )$).

The post-processing step to compute DG and ADS only selects all combinations of g's and the corresponding s's and does not search the search space, neither are there any generalization or specialization operations, nor $\preccurlyeq$-tests involved. □

4.4 Preference criteria for disjunctive languages

In Section 4.3.2 we introduced a constructive and structured way to compute almost maximally specific disjunctive concept representations. The method is less explosive than the way maximally specific disjunctive concept representations can be computed, because it eliminates the partitioning of I_s over all possible disjuncts (see Section 4.3.1). This does not, however, solve the problems of the (unique) maximally general disjunctive concept representation: the number of disjuncts might still be exponential in the number of g-bounds, while some disjuncts could be dropped without losing consistency. To control the number of disjuncts, we introduce an additional preference criterion. We first discuss some preference criteria used in existing systems, and then try to define our own criteria in a language independent way.

4.4.1 Existing systems

Existing systems that learn disjunctive concept representations start from the original problem setting of Chapter 2, i.e., they use positive and negative lowerbounds from $\mathcal{L}_I$ as information elements. Consequently the maximally specific and maximally general disjunctive concept representations s, resp. g, are uniquely determined (as a consequence of Theorem 4.25): $s = [p_1 \vee \cdots \vee p_n]_r$ where $p_1, \ldots, p_n$ are maximally specific concept representations consistent with one positive example, and $g = \bigvee_G g$ where $G = \mathcal{G}_{I_g}$, i.e., the set of all maximally general concept representations consistent with all negative examples. In general these solutions are not desirable, mainly because we want concept representations to be as simple *as possible*. Simplicity cannot be expressed by a language bias, because it only distinguishes the well-formed concept representations from the not-well-formed, on the basis of their own characteristics (often mainly syntactical ones). Therefore an extra *preference criterion* on the set of consistent disjunctive concept representations is needed. A preference criterion partially orders the set of concept representations, such that concept representations with a higher preference are preferred over those with lower preference. Since it is a partial order, it is a relation *between* concept representations, and in particular between *consistent* concept representations. Possible preference criteria are:

- d_1 is preferred over d_2 if d_1 has less disjuncts than d_2. Consequently, the disjunctions with a *minimal number* of disjuncts are the most preferred. We call this the *minimal length* criterion.

- d_1 is preferred over d_2 if d_1 is a subset of d_2. Consequently, the disjunction as a *set* of disjuncts *minimal* for $\subseteq$ is the most preferred. We will call this the *minimal set criterion*.

- Minimality criteria could take the minimality of the disjuncts themselves in account or minimality of the average size of the disjuncts (see [Kodratoff, 1988]). *Complexity based induction* is based on the *minimal description length principle* [Rissanen, 1978] and tries to minimize the number of bits to represent the background knowledge and the examples (see [Conklin and Witten, 1994] and [Muggleton *et al.*, 1992] for applications in ILP).

- Other criteria could be based on reliability and resilience to noise, or easiness to evaluate (see [Lavrač and Džeroski, 1994] and [Kodratoff, 1988]).

These criteria could also be combined.

As already noted in the previous section, it is easier to search $\mathcal{DL}_C$ general-to-specific than specific-to-general. It is also easier to adapt conjunctive general-to-specific methods to learn disjunctions than specific-to-general ones (see also [Dietterich and Michalski, 1983]). A widely used method is *the covering approach*. The basic idea of the covering approach is to introduce new disjuncts only when necessary, thus aiming at having as few disjuncts as possible in the result. In this way the minimal length or minimal set criterion is approximated. All disjuncts must be kept consistent with all g-bounds at all times; with each s-bound at least one disjunct must be consistent.

The covering approach is very well suited for non-incremental general-to-specific algorithms. Roughly speaking it starts with an empty set of disjuncts, then selects an s-bound inconsistent with the set of disjuncts found so far, finds a maximally general disjunct consistent with the s-bound and all g-bounds, and then removes all s-bounds consistent with the new disjunct from further consideration. [Lavrač and Džeroski, 1994] describes a non-incremental generic algorithm using a covering approach in more detail. Many famous systems use the covering approach, for instance AQ [Michalski, 1983], FOIL [Quinlan, 1990], MOBAL [Morik *et al.*, 1993], Progol [Srinivasan *et al.*, 1994], [Muggleton, 1995], SPECTRE [Boström and Idestam-Almquist, 1994]. [Haussler, 1988] shows that this strategy can find a solution with $d \times (ln(n_s) + 1)$ disjuncts, where n_s is the number of s-bounds and d is the number of disjuncts in the minimal length solution. Specific-to-general systems can use a similar approach by using s-bounds to generalize candidate disjuncts as much as possible, i.e., without having inconsistencies with g-bounds. This strategy is used by GOLEM [Muggleton and Feng, 1992].

In incremental approaches not all g-bounds (neither all s-bounds for that matter) are known in advance. Therefore it is necessary to be able to adapt a current set of disjuncts w.r.t. an inconsistent g-bound as well. In the general-to-specific case (e.g., MIS [Shapiro, 1983]) a new disjunct is created each time none of the existing disjuncts covers a new s-bound : specializations of the disjuncts do not cover the s-bound anyway. If any of the disjuncts is inconsistent with a new g-bound, the disjunct is marked and removed from the disjunction. New unmarked disjuncts have to be added for those s-bounds the disjunction became inconsistent with by removing this disjunct. We will further on (Algorithm 4.3) elaborate on this case.

In the specific-to-general case (e.g., CLINT [De Raedt, 1992]), the existing disjuncts could be generalized in order to cover a new s-bound, and only if no existing disjunct can be consistently generalized, a new disjunct is created. This of course at the risk of overgeneralizing existing disjuncts. Specific-to-general methods have the problem of partitioning the positive examples into sets, such that each set of examples generalizes to a disjunct also consistent with all negative examples. The problem then amounts exactly to the problem raised in Section 4.3.1: all possible combinations of partitions of s-bounds may have to be tried. Moreover, in an incremental approach, whenever a chosen partition fails (i.e., a disjunct turns out to be inconsistent with a new g-bound), these methods need a recovery strategy, e.g., through backtracking or by marking and removing the inconsistent disjunct (see also [Bundy *et al.*, 1985]). If the disjunct is again removed, all inconsistent s-bounds are reprocessed. Most systems employ an ad hoc procedure of creating a new disjunct, in particular whenever no existing disjunct can be generalized consistently. Consequently, when used incrementally, the resulting concept representation is highly dependent on the order of the examples, and not necessarily minimal.

Finally, bi-directional methods tend to inherit advantages and disadvantages of both approaches. Examples are Focussing [Bundy *et al.*, 1985], rule shell creation [Mitchell *et al.*, 1983] and the multiple convergence approach of HYDRA [Murray, 1987a].

Covering in an incremental general-to-specific strategy

In this section we elaborate on the covering approach, because it is widely used in incremental and non-incremental disjunctive algorithms. We will discuss a variant that computes disjunctions of maximally general elements consistent with I_g. Algorithm 4.3 describes a simple incremental algorithm implementing the covering approach in the general-to-specific case. D is a set of concept representations $c_1, \ldots, c_n$, representing a maximally general disjunction $c_1 \vee \cdots \vee c_n$ consistent with all known s-bounds and g-bounds. For each new s-bound i that is inconsistent with D (i.e., inconsistent with all of the c_j), a new maximally general concept representation c, consistent with I_g and with i, and not marked is added to D (Step 4.14). Concept representations that are marked, were removed from D before, and should therefore not be included in D again. Which element is actually chosen, depends on the search strategy of the algorithm. The resulting disjunction D is consistent with all s-bounds and all g-bounds.

For each new g-bound i that is inconsistent with D (i.e., inconsistent with at least one of the c_j), all disjuncts inconsistent with i are removed from D (Step 4.15) and marked. Most probably some s-bounds will not be consistent with the resulting D. These are added one by one again as if they were just presented to the algorithm (Step 4.16).

The algorithm only creates new disjuncts when necessary, hoping that the resulting D will contain as few disjuncts as possible. However, in general D will be neither a minimal set nor a minimal length solution. Suppose for instance that $D = \{ c_1 \}$, and that c_1 is consistent with the s-bound i_1. Suppose c_1 is not consistent with a new s-bound i_2. Therefore a new disjunct c_2 will be added to D, which is consistent with i_2. If c_2 is also consistent with i_1, $D = \{ c_1 , c_2 \}$ is not minimal since the disjunct c_1 could be dropped. The main reason for this behavior is that the covering approach never removes a disjunct

```
procedure covering( $\mathcal{I}nf$ : stream of info ) returns set of concept
    $D := \emptyset$
    while $\mathcal{I}nf$ is not empty
        do $i := read(\mathcal{I}nf)$
            if $i$ is an $s$-bound
            then $D := add\_s\_bound( D , i )$
            else {$i$ is a $g$-bound}
                $D := add\_g\_bound( D , i )$
    endwhile
    return $D$
endproc

procedure add_s_bound( $D$: set of concept; $i$: $s$-bound ) returns set of concept
    if $\neg( D \sim i )$
    then $D := D \cup \{ c \}$
        where $c \in \mathcal{G}_{I_g}$ and $c \sim i$ {4.14}
    return $D$
endproc

procedure add_g_bound( $D$: set of concept; $i$: $g$-bound ) returns set of concept
    if $\neg( D \sim i )$
    then $D := D \setminus \{ c \mid \neg( c \sim i ) \}$ {4.15}
        for all $s \in I_s$ such that $\neg( D \sim i )$
            do $D := add\_s\_bound( D , i )$ {4.16}
        endfor
    return $D$
endproc
```

Algorithm 4.3 Covering in an incremental general-to-specific strategy

that is consistent with I_g, even if the disjunct is redundant. On the other hand, detecting and removing redundant disjuncts such as c_1 might lead to a re-introduction of c_1 when c_2 turns out to be inconsistent with I_g.

Since this kind of recovery strategy becomes rather cumbersome and not well-structured, we propose to introduce a backtracking scheme in which the covering approach can be described. Using the same scheme we will also describe a depth-first search which will find minimal set solutions, and an iterative deepening approach to find minimal length solutions (see Section 4.5). Therefore we will first study the minimal set and minimal length criterion as a primary preference criterion in Section 4.4.2. Between equally preferred disjunctions according to these criteria, other preference criteria can be used.

4.4.2 Minimal Length and Minimal Set Preference Criterion

The minimal length criterion

In this section we describe $\mathcal{DG}$ and $\mathcal{DS}$ in combination with the minimal length preference criterion.

Definition 4.38 (Length of a disjunction) $\forall c_1, \ldots, c_n \in \mathcal{L}_C$: the length of the disjunction $c_1 \vee \cdots \vee c_n$ is n.

~~**Notation 4.39** We denote the length of the disjunction $d \in \mathcal{DL}_C$ by $\#d$.~~

We can now formally define the minimal length criterion.

Definition 4.40 (Minimal length criterion) $d \in \mathcal{DL}_C$ is a minimal length solution w.r.t. a set I of information elements iff $\#d$ is minimal w.r.t $\leq$ in $\mathcal{DVS}_I$.

Notation 4.41 We will say that "d is a ML-solution", or "d is ML".

Lemma 4.42 For all ML-solutions d in $\mathcal{DL}_C$ there exists an ML-solution g of $\mathcal{DG}_{\sqsubseteq}$ such that $d \preccurlyeq_d g$.

Proof Let $d = c_1 \vee \cdots \vee c_m$. From Theorem 4.23 follows that d is more specific than the element g' of the singleton $\mathcal{DG}_I$. Consequently, for each c_j of d there exists a disjunct c'_j of g' that is more general. Let $g = c'_1 \vee \cdots \vee c'_m$. Then $g \in \mathcal{DG}_{\sqsubseteq}$. We also have $d \preccurlyeq_d g$. Furthermore, $g \sim I_s$, because $d \sim I_s$, and $g \sim I_g$, because each disjunct of g is also a disjunct of g'. Since $\#d = \#g$, g is ML. □

Lemma 4.43 All ML solutions in $\mathcal{DL}_C$ have the same length as the ML solutions in $\mathcal{DG}_{\sqsubseteq}$.

Proof This is an immediate consequence of the previous lemma. □

Now we can specify the maximally general disjunctive concept representations that are ML.

Theorem 4.44 The set of ML maximally general concept representations in $\mathcal{DL}_C$ consistent with I are the elements of $\mathcal{DG}_{\sqsubseteq}$ of minimal length.

Proof ($\subseteq$) Suppose d is an ML maximally general concept representation consistent with I. Then there exists g in $\mathcal{DG}_{\sqsubseteq}$ of minimal length such that $d \preccurlyeq g$ (Lemma 4.42). Suppose $d \neq g$. Since g is consistent with I, it follows that d is not maximally general. Consequently $g = d$.

($\supseteq$) Suppose g is in $\mathcal{DG}_{\sqsubseteq}$, and of minimal length in $\mathcal{DG}_{\sqsubseteq}$. Then g is consistent with I, and of minimal length in $\mathcal{DL}_C$ (Lemma 4.43). Suppose g is not maximally general. Then there exists $g' \in \mathcal{DG}_{\sqsubseteq}$ strictly more general than g, of minimal length and consistent with I (Lemma 4.42). This means that there is a disjunct c of g and a disjunct c' of g' such that c' is strictly more general than c. This contradicts the fact that g is in $\mathcal{DG}_{\sqsubseteq}$. Consequently, g must be maximally general. □

To search for maximally specific elements that are ML, Theorem 4.44 and Lemma 4.42 suggest to search $\mathcal{DG}_{\sqsubseteq}$ for a minimal element $g = c_1 \vee \cdots \vee c_m$ and consistent with I, first. The disjuncts of g are elements of $\mathcal{G}_{I_g}$, i.e., consistent with I_g and maximally general. For each g, all $s = c'_1 \vee \cdots \vee c'_m$ such that $\forall j,\ 1 \leq j \leq m:\ c'_j \preccurlyeq c_j$ and such that s is maximally specific, have to be computed. For the same reasons as in Section 4.3 it will be impossible to compute all such maximally specific elements constructively, because we cannot just assign each s-bound to one of the c_j. Therefore we will also compute *almost* maximally specific elements under g instead.

Definition 4.45 (Almost maximally specific w.r.t. a preference criterion) Given a set I of information elements and a preference criterion Q, a disjunction s is called almost maximally specific w.r.t. I and fulfilling the preference criterion Q iff there exists a maximally general disjunction g consistent with I and fulfilling the preference criterion Q, such that s is almost maximally specific under g.

The following theorem states that all ML elements of $\mathcal{ADVS}_{\sqsubseteq}$ (see Definition 4.31) are exactly the elements of $\mathcal{DL}_C$ that are between an ML element g of $\mathcal{DG}_{\sqsubseteq}$ and an element s of $\mathcal{ADS}_{\sqsubseteq}$ under g. Consequently, this theorem provides a way to find all elements of $\mathcal{ADVS}_{\sqsubseteq}$ that are ML.

Theorem 4.46

$$\{\, d \in \mathcal{ADVS}_{\sqsubseteq} \mid d \text{ is ML} \,\} = \{\, d \in \mathcal{DL}_C \mid s \preccurlyeq_d d \preccurlyeq_d g,\ g \in \mathcal{DG}_{\sqsubseteq},\ g \text{ is ML, and } s \text{ is almost maximally specific under } d \text{ and } g \,\}.$$

Proof ($\subseteq$) Given $d \in \mathcal{ADVS}_{\sqsubseteq}$ and d is ML. By definition of $\mathcal{ADVS}_{\sqsubseteq}$ there exists $g \in \mathcal{DG}_{\sqsubseteq}$ and s almost maximally specific under d and g such that $s \preccurlyeq_d d \preccurlyeq_d g$. Consequently $\#s = \#d = \#g$, which means that s and g are ML.

($\supseteq$) Given $d \in \mathcal{DL}_C$ such that $s \preccurlyeq_d d \preccurlyeq_d g$, $g \in \mathcal{DG}_{\sqsubseteq}$, g is ML and s is almost maximally specific under d and g. Then $d \in \mathcal{ADVS}_{\sqsubseteq}$, since g in $\mathcal{DG}_{\sqsubseteq}$ and s is almost maximally specific under d and g. Also $\#s = \#d = \#g$, which means that s and d are ML. □

The following theorem proves that an almost maximally specific concept representation under $g \in \mathcal{DL}_C$ is also in $\mathcal{DL}_C$, i.e., it is automatically reduced.

Theorem 4.47 Given $g = c_1 \vee \cdots \vee c_n \in \mathcal{DL}_C$, which is ML and consistent with I. Also given $s = c'_1 \vee \cdots \vee c'_n$, consistent with I, and such that for all j, $1 \leq j \leq n$:

- $c'_j \in \mathcal{L}_C$,
- $c'_j \preccurlyeq c_j$, and
- c'_j is consistent with all s-bounds in I that c_j is consistent with.

Then $s \in \mathcal{DL}_C$.

Proof We have to prove that s is reduced. Suppose it is not, i.e., there exist j and k, $1 \leq j, k \leq n$, such that $c'_j \preccurlyeq c'_k$. This means that $c'_1 \vee \cdots \vee c'_{j-1} \vee c'_{j+1} \vee \cdots \vee c'_n$ is also consistent with I, and has one disjunct less than g. By Lemma 4.42 this contradicts the fact that g is ML. □

Now we can slightly adapt Algorithm 4.1 to compute ML solutions only. The computation of DVS remains the same. In the post-processing step after the while-loop only minimal length solutions are selected for DG:

$$DG = \{\ g \mid g = g_1 \vee \cdots \vee g_n, \forall j, 1 \leq j \leq n : vs(\ g_j\ , J_{s,j}\ , S_j\ ) \in DVS, \\ g \sim I_s \text{ and } g \text{ is ML } \}\ .$$

The construction of ADS remains the same: for each $g_1 \vee \cdots \vee g_n$ in DG the corresponding almost maximally specific concept representations are in ADS. The complexity of the adapted algorithm remains the same, as only the post-processing step changed. This post-processing step again only selects all combinations of g's and the corresponding s's and does not search the search space, neither are there any generalization or specialization operations, nor $\preccurlyeq$-tests involved.

Note that since the Multiple Convergence approach ([Murray, 1987a]; see Section 4.3) is actually computing the same sets DG and ADS in the specific case of conjunctive attribute-value languages with k features, it could be extended with a similar selection step to select only the ML-solutions.

The minimal set criterion

We can now formally define the minimal set criterion as well.

Definition 4.48 (Minimal set criterion) $d \in \mathcal{DL}_C$ is a minimal set solution w.r.t. a set I of information elements iff d is minimal w.r.t $\sqsubseteq$ in $\mathcal{DVS}_I$.

Notation 4.49 We will say that "d is a MS-solution", or "d is MS".

The following proposition shows in a more concrete way what it means for $d \in \mathcal{DL}_C$ to be MS. The proposition proves that for each disjunct in a MS solution, there is at least one s-bound that is consistent with this disjunct only.

Proposition 4.50 Given the disjunction $d = c_1 \vee \cdots \vee c_n \in \mathcal{DL}_C$ consistent with I. Then d is MS iff there exists for each j, $1 \leq j \leq n$, at least one s-bound i such that

- $c_j \sim i$, and

- for each k, $1 \leq k \leq n$, $k \neq j$ implies $\neg(c_k \sim i)$.

Proof ($\Rightarrow$) Consider for each c_j the set

$$\{ i \in I_s \mid \exists k,\ 1 \leq k \leq n :\ k \neq j \text{ and } c_k \sim i \}.$$

Suppose there is a j, $1 \leq j \leq n$, for which this set is equal to I_s. Then $c_1 \vee \cdots \vee c_{j-1} \vee c_{j+1} \vee \cdots \vee c_n$ is a subdisjunction of d and consistent with I, which contradicts the fact that d is MS.

($\Leftarrow$) Consider the disjunct c_j. For c_j there exists an s-bound i such that c_j is consistent with i, and none of the other disjuncts is consistent with i. Consequently, omitting c_j from d would cause d to be inconsistent with i. This is true for all disjuncts c_j of d. Consequently, d is minimal for $\subseteq$ in $\mathcal{DVS}_I$. □

This result also holds for ML solutions. This is an immediate corollary of the following proposition. It says that each ML solution is also a MS solution.

Proposition 4.51 Every ML-solution $d \in \mathcal{DL}_C$ is also a MS-solution.

Proof Suppose $d = c_1 \vee \cdots \vee c_n$ is ML but not MS. Then there exists a subdisjunction d' of d, different from d, in $\mathcal{DVS}_I$. The fact that $\#d' < \#d$ now contradicts that d is ML. □

Similarly as for ML solutions, almost maximally specific disjunctions under an MS solution are automatically reduced.

Theorem 4.52 Given $g = c_1 \vee \cdots \vee c_n \in \mathcal{DL}_C$, which is MS and consistent with I. Also given $s = c'_1 \vee \cdots \vee c'_n$, consistent with I, and such that for all j, $1 \leq j \leq n$:

- $c'_j \in \mathcal{L}_C$,
- $c'_j \preccurlyeq c_j$, and
- c'_j is consistent with all s-bounds in I that c_j is consistent with.

Then $s \in \mathcal{DL}_C$.

Proof We have to prove that s is reduced. Suppose it is not, i.e., there exist j and k, $1 \leq j, k \leq n$, such that $c'_j \preccurlyeq c'_k$. This means that c'_k is consistent with all s-bounds in I that c'_j is consistent with. By construction of s, c_k is consistent with all s-bounds in I that c_j is consistent with. By Proposition 4.50 this contradicts the fact that g is MS. □

The minimal length criterion is stronger than the minimal set criterion, because if there does not exist a consistent disjunction of smaller length than that of a consistent disjunction d, no disjunct of d can be deleted while remaining consistent.

On the other hand, to implement the minimal length criterion directly, all possible disjunctions will have to be searched, because the length of the disjunction does not have a direct relation with $\preccurlyeq_d$, which is the order used to search $\mathcal{DL}_C$. The minimal set criterion on the other hand does, because of Proposition 4.5. Although the minimal set criterion has a more direct relationship to $\preccurlyeq_d$ than the minimal length criterion, and would therefore be more easy to handle, maximally *specific* disjunctive concept representations that are MS are exactly the elements of $\mathcal{DS}$: because the concept representations of $\mathcal{DS}$ are reduced, omitting a disjunct from an element of $\mathcal{DS}$ causes the concept representation to be inconsistent with the s-bound the disjunct originated from. Consequently, also in this case it will be more useful to use almost maximally specific concept representations under the maximally general ones.

Similarly as Theorem 4.46 the following theorem states that all MS elements of $\mathcal{ADVS}_{\sqsubseteq}$ are exactly the elements of $\mathcal{DL}_C$ that are between an MS element g of $\mathcal{DG}_{\sqsubseteq}$ and an element s of $\mathcal{ADS}_{\sqsubseteq}$ under g.

Theorem 4.53

$$\{\, d \in \mathcal{ADVS}_{\sqsubseteq} \mid d \text{ is MS} \,\} = \{\, d \in \mathcal{DL}_C \mid s \preccurlyeq_d d \preccurlyeq_d g, g \in \mathcal{DG}_{\sqsubseteq}\ , \\ g \text{ is MS, and } s \text{ is almost maximally specific under } d \text{ and } g \,\}\ .$$

Proof ($\subseteq$) Given $d \in \mathcal{ADVS}_{\sqsubseteq}$ and d is MS. By definition of $\mathcal{ADVS}_{\sqsubseteq}$ there exists $g \in \mathcal{DG}_{\sqsubseteq}$ and s almost maximally specific under d and g such that $s \preccurlyeq_d d \preccurlyeq_d g$. Suppose $g = c_1 \vee \cdots \vee c_n$, $s = c'_1 \vee \cdots \vee c'_n$ and $d = c''_1 \vee \cdots \vee c''_n$. Then $\forall j,\ 1 \leq j \leq n : \{\, i \in I_s \mid c_j \sim i \,\} = \{\, i \in I_s \mid c'_j \sim i \,\} = \{\, i \in I_s \mid c''_j \sim i \,\}$. Consequently, if d is MS, then g is MS.

($\supseteq$) Given $d \in \mathcal{DL}_C$ such that $s \preccurlyeq_d d \preccurlyeq_d g$, $g \in \mathcal{DG}_{\sqsubseteq}$, g is MS and s is almost maximally specific under d and g. Then $d \in \mathcal{ADVS}_{\sqsubseteq}$, since g in $\mathcal{DG}_{\sqsubseteq}$ and s is almost maximally specific under d and g. Suppose $g = c_1 \vee \cdots \vee c_n$, $s = c'_1 \vee \cdots \vee c'_n$ and $d = c''_1 \vee \cdots \vee c''_n$. Then $\forall j,\ 1 \leq j \leq n : \{\, i \in I_s \mid c_j \sim i \,\} = \{\, i \in I_s \mid c'_j \sim i \,\} = \{\, i \in I_s \mid c''_j \sim i \,\}$. Consequently, if g is MS, then d is MS. □

4.5 The Disjunctive Iterative Versionspaces algorithm

In Chapter 3 we presented the Iterative Versionspaces algorithm, a depth-first version of the Description Identification algorithm, and we constructed a framework in which both algorithms could be described. We will now similarly present a depth-first version of the Disjunctive Description Identification algorithm, and build up a framework to describe disjunctive search algorithms. The underlying motivation for developing a depth-first algorithm to search for disjunctive concept representations, instead of computing all solutions as in DDI, is again that searching all solutions is memory consuming, while concept learning algorithms are usually not asked to find *all* solutions. They should rather find only one, which is maximally preferred w.r.t. the preference criterion. The advantages of finding a maximally general and a maximally specific were discussed in Section 3.4.2: it allows choosing between making more errors of commission rather than errors of omission, and vice versa; it also allows the generation of relevant lower- and upperbounds. As shown in the previous sections, not imposing a preference criterion could lead to very specific solutions, or to solutions with many disjuncts, etc. Therefore we do not have to compute the set DVS of DDI completely, but rather only a subset, until we have found a solution.

As we have illustrated above, this is also the idea behind the covering approach (see Algorithm 4.3): there is, at each moment, only one current set of disjuncts; if this set is overly specific, it is generalized; if it is overly general, it is specialized. However, whereas it is not always clear what the effect is of the covering approach's adding and removing disjuncts, we describe a backtracking scheme in which we can fit several approaches, such as the minimal set criterion by using a depth-first search, the minimal length criterion by using an iterative deepening approach, and the covering approach by yet another kind of backtracking. One could even think of going beyond these search strategies, and of introducing more sophisticated backtracking mechanisms.

In the text we only present the depth-first version, which actually implements the minimal set preference criterion, and explain how the other search strategies can be derived. As such, the development of the Disjunctive Iterative Versionspaces algorithm (DITVS) again establishes a framework in which other disjunctive search strategies can also be described and understood.

This section is structured as follows: first we discuss the datastructures in DITVS (Section 4.5.1), and the invariants on these datastructures (Section 4.5.2). The reader may already want to take a look at the example in Section 4.5.5, before reading the technical section describing the algorithm (Section 4.5.4). Section 4.5.6 describes how the framework of DITVS can be used to adopt the minimal set criterion and the covering approach (Section 4.5.6). Then we discuss the computational complexity of DITVS in Section 4.5.7. Finally, we briefly discuss to what extent the extensions of ITVS are still applicable in Section 4.5.8.

4.5.1 Datastructures

Basically the idea of DITVS is the following: instead of computing the sets $\mathcal{DG}_{\sqsubseteq}$ and $\mathcal{ADS}_{\sqsubseteq}$ (the boundary sets of $\mathcal{ADVS}_{\sqsubseteq}$; see Section 4.3.1) completely, DITVS should only compute one element $g_{vs} = g_1 \vee \cdots \vee g_n$ of $\mathcal{DG}_{\sqsubseteq}$ and one element of $s_{vs} = s_1 \vee \cdots \vee s_n$ of $\mathcal{ADS}_{\sqsubseteq}$, such that g_{vs} and s_{vs} are MS, and such that s_{vs} is almost maximally specific under g_{vs}. The disjuncts g_j of g_{vs} are elements of $\mathcal{G}_{I_g}$ (see Definition 4.19 and Definition 4.31). To find g_{vs} we have to search systematically the set of all subdisjunctions of $\bigvee_G g$ (with $G = \mathcal{G}_{I_g}$) consistent with I_s. In order to find a MS solution, we should search the most preferred elements first. This is one of the main differences of DITVS w.r.t. DDI: whereas DDI only selects MS solutions in a post-processing step, DITVS takes the preference criterion into account during the search. To find s_{vs} we search for an almost maximally specific concept representation under g_{vs}.

In order to compute alternatives for g_{vs} and s_{vs} in case these are inconsistent with I, DITVS should contain *backtrack information*. In ITVS this backtrack information consisted of two stacks B_s and B_g, from which the boundary sets S and G could be recomputed at all times. The stack B_g contained *choicepoints* (ind , s_{ind} , alt_{ind}), such that s_{ind} and alt_{ind} were maximally specific and consistent with the first ind g-bounds; s_{ind} was the current choice in the choicepoint, alt_{ind} contained alternatives to be explored when the branch of s_{ind} turned out to contain no solutions; B_s has a similar structure (for all invariants on B_g and B_s, see Section 3.6.2). Here we will try to find a similar representation for DITVS.

Like DDI, DITVS will be general-to-specific driven: it systematically searches through $\mathcal{DL}_G$ for elements of $\mathcal{DG}_{\sqsubseteq}$. Therefore we will first make a choice for the disjunctive

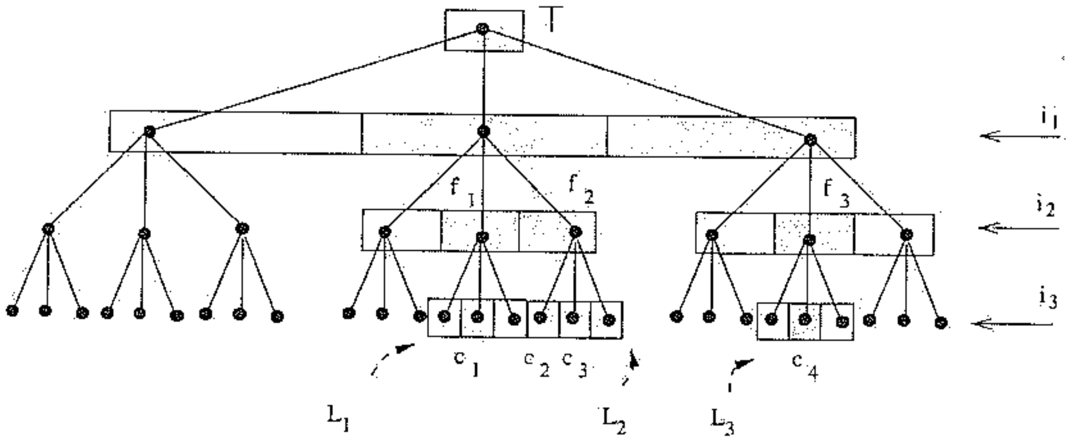


Figure 4.7 Introducing the datastructures of DITVS

counterpart of B_g, the *disjunctive backtrack stack* (further on called d_stack) DB_g. The d_stack DB_g could consist of several backtrack-stacks of the type B_g, one for each of the pairs (g_j , s_j), thus representing a versionspace for each of the disjuncts in g_{vs} and s_{vs}. However, this would give a great amount of overhead, since these versionspaces are not necessarily disjunct. On the one hand, an element that is already rejected in one versionspace, possibly after having searched its generalizations or specializations, will not necessarily be rejected in the other versionspaces, meaning that its generalizations and specializations will be searched again. On the other hand, this could also result in searching all permutations of all disjunctions. Therefore we will integrate these different stacks into one datastructure for DB_g.

If we are to search the set of subdisjunctions of $\bigvee_G g$ in a systematic way, each time we make a certain choice which subdisjunction to explore next, we should be able to represent the chosen subdisjunction, and the remaining subdisjunctions. As in ITVS, this information is contained in a *choicepoint*. We will first explain how we will represent the chosen subdisjunction.

As $\mathcal{G}_{I_g}$ has elements in all parts of $\mathcal{L}_C$, subdisjunctions of $\bigvee_G g$ can have disjuncts of all parts of $\mathcal{L}_C$. Consider the search tree in Figure 4.7. The top layer only contains $\top$, and is consistent with no g-bounds. The layers below consist of the maximally general specializations of $\top$ consistent with i_1, i_2 and i_3 respectively. The current choice of disjuncts is on each layer represented by the shaded boxes. In the figure, for each new g-bound all chosen disjuncts of the previous layer had to be specialized. Suppose the concept representations $f_j \in \mathcal{L}_C$ ($j \in \{ 1 , 2 , \dots \}$) are consistent with the g-bounds i_1 and i_2. L_j is the list of maximally general specializations of f_j, consistent with i_3. The lists L_j are depicted by boxes. We will call the lists L_j *VS-lists*. The list of all VS-lists of one layer is called a *Disjunctlist*. Let us concentrate on the disjunction $c_1 \vee c_2 \vee c_3 \vee c_4$. This disjunction is determined by the VS-lists to which its disjuncts belong, and by the actual choice of disjuncts within each VS-list: $c_1 \vee c_2 \vee c_3 \vee c_4$ is determined by the VS-lists L_1, L_2 and L_3, and, within these, by the choices $\{ c_1 \}$, $\{ c_2 , c_3 \}$ and $\{ c_4 \}$ respectively. $\{ (L_1 , \{ c_1 \}) , (L_2 , \{ c_2 , c_3 \}) , (L_3 , \{ c_4 \}) \}$ then represents the disjunction $c_1 \vee c_2 \vee c_3 \vee c_4$.

In general, we can distinguish two levels of disjunctions in a current choice g_d for g_{vs} to be represented in a choicepoint: g_d could contain one or more elements of a VS-list, and it could contain elements of one or more VS-lists. Consequently, we have to represent g_d on two levels: we have to specify which VS-lists contain disjuncts of g_d, and for each such VS-list, we have to specify which of its elements are disjuncts of g_d. A particular choice of disjuncts within a certain VS-list L is denoted Ch_L.

The second element we have to represent in a choicepoint is the set of remaining choices. We will do this by imposing a *total order* on all possible choices in a choicepoint, and by searching the remaining choices according to this order. This total order can be defined on two levels: we can define an order on the possible choices inside each VS-list (what we will call a *local* order), and then define an order on lists of VS-lists (what we will call a *global* order). Given a list DL of VS-lists, a global order on the possible choices made inside each VS-list will be constrained by the chosen preference criterion.

Constraint 4.54 (The Order Constraint) For the minimal set criterion, all subsets of a certain choice g_d should come before g_d itself.

For the minimal set criterion this means that the chosen local order must be a superset of $\sqsubseteq$. Note that, since we do not use empty disjunctions, the minimal element in the global order will be a singleton. Also note that this constraint does not completely define the total order of alternatives. In Section 4.5.4 we specify a total order for the minimal set criterion which fulfills the constraint. It is based on the positions of the elements of Ch_L within L.

In general a list DL of VS-lists L, together with a particular choice Ch_L in each VS-list, represents a disjunction $\bigvee_{DL} (\bigvee_{Ch_L} c)$, and all remaining choices. A combination of L together with a choice Ch_L is called a Global Disjunct. The list DL is therefore called a Global Disjunctlist.

In the discussion of the DITVS algorithm, we will make abstraction of the actual representation of Ch_L and of the actually chosen local and global order. We will use the following functions:

1. the function `init_choice`, which for a given VS-list L returns the first possible local choice in the chosen enumeration. Because of Constraint 4.54 (the Order Constraint) for the minimal set criterion, `init_choice` always returns a choice representing a singleton containing one element of L.

2. the function `is_last_global_choice`, which for a given Disjunctlist DL returns true if the current global choice for DL is maximal in the global order. Because of Constraint 4.54 (the Order Constraint) for the minimal set criterion, this would be the case when each Ch_L in DL represents the corresponding set L in DL completely.

3. the function `next_global_choice`, which for a given list DL of VS-lists such that $\neg$*is_last_global_choice*(DL), returns the next possible global choice in the chosen order.

After having chosen a particular global order, each choicepoint on DB_g represents a disjunctive concept representation g_d and a corresponding set of alternatives for g_d.

So far, we have described how we can search through the subdisjunctions of $V_G g$ systematically. To this aim we have already partly described the structure of the choicepoints of the d_stack DB_g. In DDI the versionspaces $vs(\ g_j\ ,\ J_{s,j}\ ,\ S_j\ )$ of DVS also contained for each maximally general disjunct g_j the set $J_{s,j}$ of indexes to the s-bounds g_j is consistent with, and the set S_j of maximally specific elements in $\mathcal{L}_C$ more specific than g_j, and consistent with I_g and $I_s|J_{s,j}$. In DITVS we also associate to each disjunct g_j of g_{vs} a set $J_{s,j}$ of indexes to s-bounds g_j is consistent with. When specializing g_j, this allows to check consistency of the s-bounds of $I_s|J_{s,j}$ instead of checking consistency with all elements of I_s. W.r.t. the set S_j, DITVS differs from DDI in that it does not compute S_j completely, but rather only one element $s_j \in S_j$. DITVS should also have the possibility to backtrack on s_j, whenever necessary. Therefore each set S_j is represented by one element s_j, together with a backtrack stack $B_{s,j}$ to compute alternatives for s_j. The stack $B_{s,j}$ has the same form as the stack B_s in ITVS (see above, or Section 3.6.1).

There are several alternative options as to whether the stacks B_s for the elements g_j on DB_g should be stored explicitly or recomputed:

1. One could choose to store a backtrack stack B_s for each element of each VS-list on DB_g. Each time a new s-bound is processed, all backtrackstacks on DB_g are updated. Backtracking to a previous choicepoint on DB_g would then require no recomputation at all. However, as we will see in the presentation of DITVS, these backtrackstacks are not necessarily useful.

2. One could choose to store the backtrack stack only for elements in the top choicepoint of DB_g, and therefore only for the actual disjuncts in s_{vs}. This means that each time the disjunction represented by the top changes (at least some of) the corresponding backtrack stacks have to be recomputed.

3. Another option stores a backtrack stack for each element of each VS-list on DB_g, but never updates them. If possible these stacks are reused when specializing the elements of a VS-list. If they are not reusable, they are recomputed, but only for the disjuncts corresponding to g_{vs}.

The third option is implemented in DITVS (Algorithm 4.4).

Figure 4.8 relates the datastructures of DVS and DITVS (see also Figure 4.5). The large bullet points in the upper part of the figure now depict the disjunction g_{vs}. To each disjunct g_j one concept representation s_j is associated in the lower part of the figure. Whereas the structure DVS used in DDI would contain *all* elements of the lowest layer of the upper part of the figure, DITVS represents only one current disjunction g_{vs} together with the information contained in DB_g: which VS-lists contain disjuncts of g_{vs}, and, for each VS-list, which disjuncts belong to g_{vs}. In the lower part of Figure 4.8 S_j corresponding to g_j would in DDI contain all leaves of the corresponding tree. DITVS stores only one element $s_j \in S_j$, together with the backtrackinformation in $B_{s,j}$. The choicepoints of $B_{s,j}$ are depicted by the dashed boxes.

Another matter related to the structure of DB_g is the way of implementing a maximal generality test for an element g_d represented in a choicepoint of DB_g. We will describe a similar test as in ITVS: it tests whether a given disjunctive concept representation is maximally general and consistent with I_g, *or* whether this concept representation can be obtained by specializing an alternative for g_{vs} on DB_g. Therefore we will need to trace the

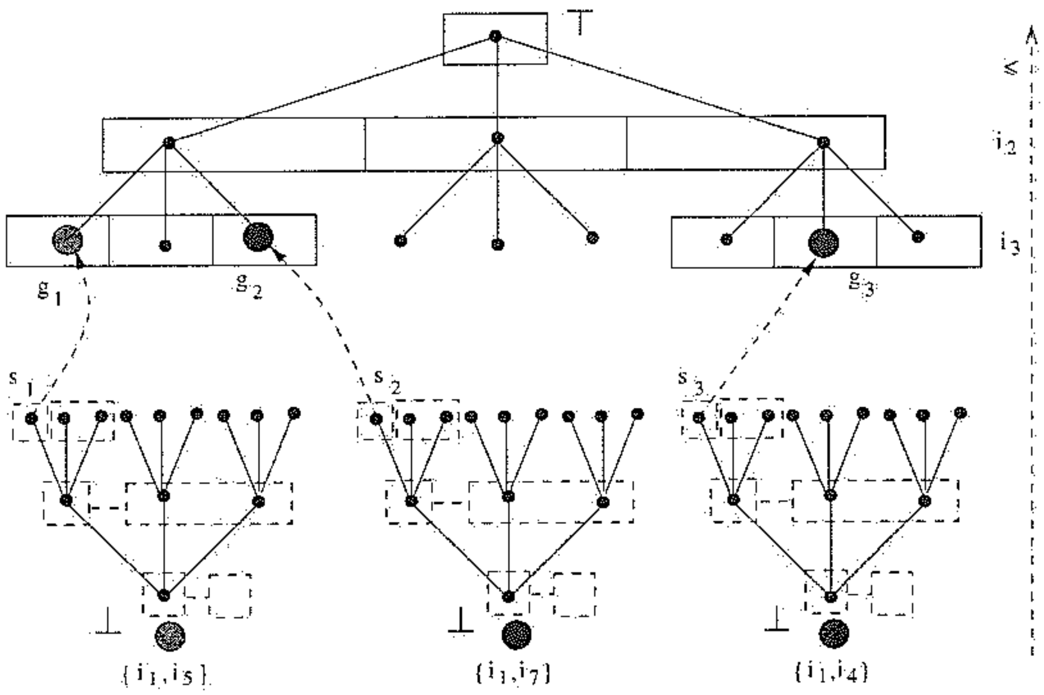


Figure 4.8 Relating the datastructures of DITVS and DDI

path back from each disjunct c_j in g_d towards $\top$, along the concepts c_j it was obtained from by specialization. If c_j was obtained by specializing f_j, we will call f_j the *father-concept* of c_j. The Global Disjunct that contains f_j will be called the *father-disjunct* of c_j. The transitive closure of the father-disjunct relation will be called *ancestor-disjunct*. Similarly the transitive closure of the father-concept relation will be called *ancestor-concept*. For each c_j present in DB_g, DB_g has to contain information about which VS-list is its father-disjunct. However, we will not explicitly represent this information, and assume it is implicitly present. Whenever we need the father-concept, resp. father-disjunct, of c_j, we will use the functions `father_concept` or `father_disjunct`.

In Figure 4.9 the links to the father-concepts are explicitly depicted with full arrows. Each VS-list originates from exactly one father-concept, which belongs to the father-disjunct.

In summary, DITVS uses the following datastructures (see Figure 4.9[3]):

- The d_stack DB_g consists of choicepoints C_B each containing
  - a list DL of *global disjuncts* (i.e., a Global Disjunctlist), and

[3]Note that, as in the previous figures, on Figure 4.9 the bottom choicepoint of DB_g is drawn in the top of the figure, and the choicepoints closer to the top of the stack are drawn below it, to reflect the relation $\preccurlyeq$ in the figure.

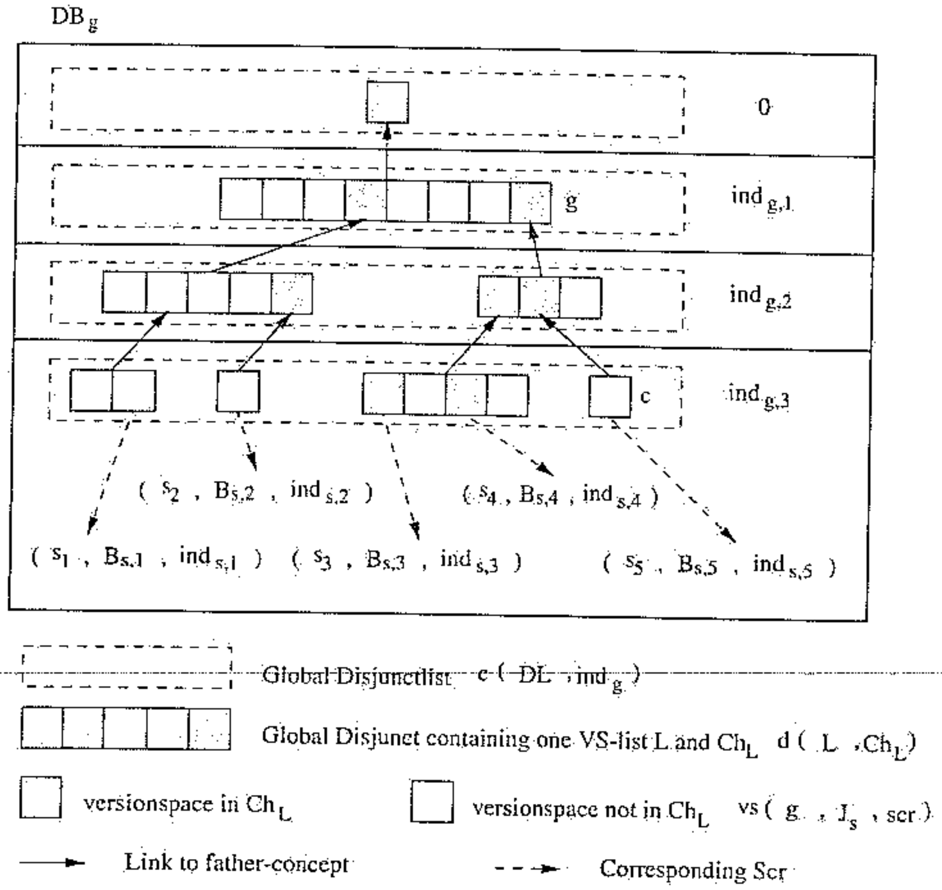

DB_g
0
$ind_{g,1}$
g
$ind_{g,2}$
$ind_{g,3}$
c
$(s_2 , B_{s,2} , ind_{s,2})$
$(s_4 , B_{s,4} , ind_{s,4})$
$(s_1 , B_{s,1} , ind_{s,1})$
$(s_3 , B_{s,3} , ind_{s,3})$
$(s_5 , B_{s,5} , ind_{s,5})$
Global Disjunctlist $c(DL , ind_g)$
Global Disjunct containing one VS-list L and Ch_L $d(L , Ch_L)$
versionspace in Ch_L
versionspace not in Ch_L $vs(g , I_s , scr)$
Link to father-concept
Corresponding Scr

Figure 4.9 The datastructures of DITVS

- an index ind_g in I_g up to where the elements of the VS-lists in the global disjuncts are consistent with I_g.

In Figure 4.9 a choicepoint is depicted as one layer of DB_g. In the algorithms we represent a choicepoint C_B as a term $c(DL , ind_g)$.

- Each global disjunct D on DB_g consists of
  - a list L of Versionspaces (i.e., a VS-list), and
  - the particular choice of a sublist Ch_L of L.

In Figure 4.9 a global disjunct is depicted as a VS-list L where the elements of Ch_L are shaded, and the others are white. In the algorithms we represent a global disjunct by a term $d(L , Ch_L)$.

- Each Versionspace in a VS-list L consists of

- the representation of its maximal element g ($g \in \mathcal{L}_C$),
- the set J_s of all indexes of elements in I_s consistent with g, and
- a representation Scr of the set of almost maximally specific concept representations in $\mathcal{L}_C$, more specific than g and consistent with $I_s \mid J_s$.

In Figure 4.9 a Versionspace is depicted as a square. In the algorithms we represent a Versionspace by a term $vs(\ g\ ,\ J_s\ ,\ scr\ )$.

- Each set of almost maximally specific concept representations Scr is represented by
  - one maximally specific concept representation s consistent with $I_s | J_s$,
  - a stack B_s to backtrack on s, with the same structure as B_s in ITVS, and
  - an index ind_s denoting up to where s is consistent with the elements of $I_s \mid J_s$.

  In the algorithms we represent Scr by a term $scr(\ s\ ,\ B_s\ ,\ ind_s\ )$.

Since each B_s is of the same sort as the stacks used in ITVS, we will use the same operations (**push** and **pop**) on them. For DB_g we have similar **push** and **pop** operations. Additionally we introduce an operation **top** on DB_g: $top(\ DB_g\ )$ returns the list DL and the corresponding index ind_g of the top element $c(\ DL\ ,\ ind_g\ )$ of DB_g. In contrast to **pop**, **top** does not return DB_g with its top element removed.

4.5.2 Invariants

We have the following invariants on the datastructures of DITVS:

- **Invariant 4.5.1.** g_{vs} is a maximally general MS concept representation consistent with I.
- **Invariant 4.5.2.** s_{vs} is an almost maximally specific MS concept representation under g_{vs}.
- **Invariant 4.5.3.** For each choicepoint $C_B = c(\ DL\ ,\ ind_g\ )$ on DB_g, the disjunction $g_d = \bigvee_{DL}(\ \bigvee_{Ch_L} c\ )$ is a maximally general MS concept representation consistent with $I_g[1], \ldots, I_g[ind_g]$ and I_s; g_{vs} is the disjunctive concept representation represented by the top choicepoint of DB_g.
- **Invariant 4.5.4.** For all $d \in \mathcal{DL}_C$: if $d \sim I$ and d is MS, then d is more specific than g_{vs} or than an alternative for g_{vs} on DB_g.
- For each choicepoint $C_B = c(\ DL\ ,\ ind_g\ )$ on DB_g, for each $d(\ L\ ,\ Ch_L\ )$ in DL, and for each $vs(\ g\ ,\ J_s\ ,\ Scr\ )$ in L:
  - **Invariant 4.5.5.** $J_s = \{\ k \mid 1 \leq k \leq n_s \text{ and } g \sim I_s[k]\ \}$.
  - **Invariant 4.5.6.** If C_B is not the bottom choicepoint on DB_g, there exists a father-disjunct and a father-concept for g on DB_g.
  - For $Scr = scr(\ s\ ,\ B_s\ ,\ ind_s\ )$ we have:

* **Invariant 4.5.7.** $s \preccurlyeq g$, and s is maximally specific and consistent with $\{ I_s[k] \mid 1 \leq k \leq ind_s \text{ and } k \in J_s \}$.
* **Invariant 4.5.8.** For all choicepoints (ind_1 , s_1 , alt_1) on B_s: all elements a_1 of alt_1 are more specific than g and are maximally specific in S_J, where $J = I_g[1..n_g] \cup \{ I_s[k] \mid 1 \leq k \leq ind_1 \text{ and } k \in J_s \}$. Furthermore $\neg(a_1 \preccurlyeq s)$.
* **Invariant 4.5.9.** For all choicepoints (ind_2 , s_2 , alt_2) closer to the top of B_s: $ind_1 < ind_2$, $s_1 \prec s_2$, $s_1 \prec a_2$ and $\neg(a_1 \preccurlyeq a_2)$ for all $a_1 \in alt_1$ and $a_2 \in alt_2$.
* **Invariant 4.5.10.** For all $c \in \mathcal{L}_C$, more specific than g and consistent with $I_s \mid J_s$, s or an alternative for s on B_s is more specific than c.
* **Invariant 4.5.11.** If C_B is the top choicepoint of DB_g, and g is a disjunct of g_{vs}, $ind_s = n_s$.

Invariant 4.5.3 expresses that each choicepoint on DB_g represents a disjunctive concept representation g_d consistent with all s-bounds and with the g-bounds up to $I_g[ind_g]$. Similarly as in ITVS, backtracking to this choicepoint will have to be followed by reprocessing all g-bounds g_d is not consistent with, i.e., $I_g[ind_g + 1] , \ldots , I_g[n_g]$.

Invariant 4.5.4 expresses the invariancy of completeness: each MS solution consistent with I is more specific than g_{vs}, or more specific than an alternative for g_{vs} which can be found by backtracking. Note that, like in DDI, no element of a VS-list should be removed from consideration for not being consistent with all s-bounds. Therefore a VS-list should not be removed from the stack until all its elements and all disjunctions of these elements are rejected. This is unlike in ITVS, where all alternatives on B_g which were not consistent with *all* s-bounds could be pruned safely (see Section 3.6.3).

Invariant 4.5.5 to Invariant 4.5.11 express the invariants in each of the versionspaces $vs(g , J_s , Scr)$ on DB_g. Invariant 4.5.5 states that J_s contains all indexes to s-bounds of I_s consistent with g. Invariant 4.5.6 ensures that there is a father-disjunct and father-concept for each g which is not in the bottom choicepoint of DB_g. This will be important when the chain of father-disjuncts from g to $\top$ is followed for testing maximal generality of g.

Invariant 4.5.7 to Invariant 4.5.10 mainly express that for the components s, B_s and ind_s of each $scr(s , B_s , ind_s)$ on DB_g, the same invariants hold as in ITVS for s, B_s and n_s (see Section 3.6.2). Invariant 4.5.7 states that s is more specific than g (*which did not necessarily hold in ITVS*) and consistent with all the s-bounds up to $I_s[ind_s]$ whose index is in J_s. Invariant 4.5.8 and Invariant 4.5.9 express that the stack B_s can be used for testing maximal specificity of s, and for implementing an optimal refinement perator (see Section 3.6.2). Invariant 4.5.10 expresses the invariance of completeness or s_{vs}: each maximally specific disjunct more specific than g and consistent with the nformation elements g is consistent with, can be recomputed from s or an alternative for on B_s. Finally Invariant 4.5.11 expresses that only the stacks B_s of the versionspaces cluded in g_{vs} are updated w.r.t. all s-bounds, as explained above.

4.5.3 Maximal generality and optimal refinement

We will now explain how to search for maximally general MS disjunctions. To find the MS elements of $\mathcal{DG}_{\sqsubseteq}$, the upperbound of all MS elements of $\mathcal{ADVS}_{\sqsubseteq}$, the result of Theorem 4.53 is used. The disjunctions in $\mathcal{DG}_{\sqsubseteq}$ consists of maximally general disjuncts only. Checking whether g_{vs} is maximally general therefore amounts to checking whether each of its disjuncts is maximally general. Checking whether a disjunct is maximally general will be done as in ITVS, by searching for more general elements in the ancestor-disjuncts. Whereas in ITVS the choicepoints on B_g only contained alternatives that are still to be explored, we will have to make a similar distinction between the alternatives already explored and the alternatives still to be explored. The strategy in ITVS amounted to considering a concept representation c for specialization only when no alternative on B_g was more general than c. Consequently, if more than one element in a choicepoint was more general than c, c was only allowed as a specialization of the last one in the list (see Section 3.6.2). We will extend this method to the disjunctive case, by also specializing c only from one alternative. In ITVS the order of choosing alternatives was left-to-right, by each time considering the first element of the list of alternatives as the next candidate. Here the order of traversing a VS-list is determined by the local order on the VS-list. We will therefore assume the following constraint on the local order of each VS-list.

Constraint 4.55 (The consequential order constraint) Given a VS-list L, a (possibly empty) subset S of L and g and g' which are in L, but not in S. Then $\{ g \}$ is before $\{ g' \}$ in the local order on L iff $S \cup \{ g \}$ is before $S \cup \{ g' \}$ in the local order.

If more than one element in a VS-list is more general than c, then we can now require that c is only allowed as a specialization of the largest one in the local order, independently of the other choices made in this VS-list. As such we have a similar situation as in ITVS for testing maximal specificity.

Suppose we have to check whether the disjunct c is maximally general. Also suppose, g_{vs} is a (possibly empty) set of maximally general concept representations, which are not consistent with all s-bounds. The idea is to add c to g_{vs}, if c is maximally general.

Suppose $d(\ L\ ,\ Ch_L\ )$ is an ancestor-disjunct of c, and g is the corresponding ancestor-concept of c (i.e., c is obtained by successive specialization of g). Suppose that there exists a g' which comes after g in the local order on L, and such that $c \preccurlyeq g'$. In that case, it is safe to skip c at this point, because it is either not maximally general, or it will be found as a specialization of g'.

Now suppose that there exists no g' which comes after g in the local order on L, and such that $c \preccurlyeq g'$, and this for all ancestor-disjuncts of g. In that case, there is no other choice left to specialize c from, except the choices with more disjuncts. These choices might result in non-MS solutions, if c is maximally general. Therefore c should be considered as a candidate-disjunct, if it is maximally general. Suppose it is not maximally general. Then there is a maximally general disjunct c'', which is more general than c. Moreover, there exists an ancestor-disjunct $d(\ L\ ,\ Ch_L\ )$ of c, with corresponding ancestor-concept g, and a g'' in L, such that g'' is before g in the local order on L, and such that $c \preccurlyeq c'' \preccurlyeq g''$. Since g'' is before g, the choice $Ch''_L = Ch_L \setminus \{ g \} \cup \{ g'' \}$ was before Ch_L (because of Constraint 4.55). Consequently, all disjunctions of specializations of elements of Ch''_L have been considered, and c'' can only be maximally general if it is in g_{vs}. Consequently, if c is not more specific than a disjunct in g_{vs} it is maximally general. If c is more specific than an element in g_{vs}, it is not maximally general.

Consequently, to check maximal generality of c, c has to be compared to all g' in all ancestor-disjuncts of c, with corresponding father-disjunct g, and such that g' is after g in the local order of the ancestor-disjunct. If none of these g' is more general than c, c is maximally general. Otherwise, it is not maximally general, or it will be generated while specializing g'.

4.5.4 The algorithm [T]

SUMMARY: in this section we describe the Disjunctive Iterative Versionspaces algorithm. This algorithm implements a depth-first search for a maximally general disjunctive concept representation consistent with a given set I of information elements. It incorporates the same ideas as ITVS, i.e., it checks for maximal generality using the disjunctive backtrack stack DB_g, and for maximal specificity using the backtrack stacks B_s on DB_g. At the same time, the method implements an optimal refinement operator.

We will first discuss the procedure DITVS (see Algorithm 4.4), the main loop of the algorithm. First the datastructures are initialized in `initialize`. I_s and I_g are initially empty, n_s and n_g are correspondingly 0. One choicepoint in which there is one global disjunct d, consistent with the $ind_g = 0$ elements of I_g, is pushed onto DB_g. Initially d's VS-list has one element $vs(\top, I_s, InitScr)$, where I_s is initially empty. The only possible global choice of this list will just contain $\top$; consequently the corresponding g_{vs} is $\top$. The corresponding initial Scr is $scr(\bot, \emptyset, 0)$, which represents the singleton $\{\bot\}$. The corresponding maximally specific concept representation s_{vs} is therefore equal to $\bot$. After initialization all invariants are fulfilled. As before (and reflecting the incremental nature of our algorithms), after initialization the main loop reads a new information element i from the infostream $\mathcal{I}nf$ (Step 4.17) and processes it. Again we will split up the discussion, according to i being an s-bound or a g-bound.

Because DITVS's main view is general-to-specific, handling s-bounds and g-bounds is not symmetrical, like it was in ITVS. We will first discuss the case of i being a g-bound. Like in ITVS (but unlike in DDI), i is stored in I_g for reasons of backtracking (Step 4.22). If i is consistent with g_{vs}, nothing has to be adapted, since then each disjunct of g_{vs} is consistent with all g-bounds, since g_{vs} is still consistent with all s-bounds, since DB_g did not change, and since s_{vs} is still almost maximally specific under g_{vs}. In this case all invariants still hold at the end of the while loop. On the other hand, when g_{vs} is not consistent with i, a new g_{vs} will have to be constructed. Given

- DB_g fulfilling Invariant 4.5.3 to Invariant 4.5.10, and
- g_{vs}, a maximally general concept representation consistent with $I_g[1], \ldots, I_g[ind]$, and
- if g_{vs} is consistent with I_s, then it is MS,

d_select_alternative(g_{vs} , DB_g , *ind*) (see Algorithm 4.5) *in general* returns a new g_{vs} and DB_g fulfilling Invariant 4.5.1, and Invariant 4.5.3 to Invariant 4.5.10. Before Step 4.23 the number *ind* of g-bounds DB_g is consistent with is $n_g - 1$, and the preconditions of `d_select_alternative` are fulfilled. Therefore its postconditions will be fulfilled also. Then, given the resulting g_{vs} and DB_g, the structures *scr* in the top of DB_g are updated in `generalize_disjuncts` (see Algorithm 4.8; Step 4.24). In general, given s_{vs} and DB_g fulfilling Invariant 4.5.3 to Invariant 4.5.10, procedure

```
procedure DITVS($\mathcal{I}nf$ : stream of info ) returns disj_concept, disj_concept, d_stack
  $g_{vs}$, $s_{vs}$, $DB_g$ := $initialize$( )
  while there are still information elements to be processed
     do $i := read( \mathcal{I}nf )$ {4.17}
        if $i$ is an $s$-bound
        then $n_s := n_s + 1$; $I_s[n_s] := i$ {4.18}
             $DB_g := update\_Js( DB_g , n_s )$ {4.19}
             if $\neg( g_{vs} \sim i )$
             then $g_{vs}, DB_g := d\_select\_alternative( g_{vs} , DB_g , n_g )$ {4.20}
             $s_{vs}, DB_g := generalize\_disjuncts( DB_g )$ {4.21}
        else {$i$ is a $g$-bound}
             $n_g := n_g + 1$; $I_g[n_g] := i$ {4.22}
             if $\neg( g_{vs} \sim i )$
             then $g_{vs}, DB_g := d\_select\_alternative( g_{vs} ; DB_g , n_g - 1 )$ {4.23}
                  $s_{vs}, DB_g := generalize\_disjuncts( DB_g )$ {4.24}
  endwhile
  return $g_{vs}$, $s_{vs}$, $DB_g$
endproc

procedure initialize( ) returns disj_concept, disj_concept, d_stack
  $I_s := \emptyset$; $n_s := 0$
  $I_g := \emptyset$; $n_g := 0$
  $DB_g := \emptyset$
  $InitScr := scr( \bot , \emptyset , 0 )$
  $L := [ vs( \top , I_s , InitScr ) ]$
  $DL := [ d( L , init\_choice( L ) ) ]$
  $DB_g := push( DL , 0 , DB_g )$
  $g_{vs} := \top$
  $s_{vs} := \bot$
  return $g_{vs}$, $s_{vs}$, $DB_g$
endproc
```

Algorithm 4.4 Disjunctive Iterative Versionspaces (DITVS)

generalize_disjuncts returns s_{vs} and DB_g such that Invariant 4.5.2 to Invariant 4.5.11 are fulfilled. Consequently, in this case all invariants hold at the end of the while loop.

Now we will discuss the case of i being an s-bound. First i is stored in I_s (Step 4.18). Then all J_s on DB_g are updated to fulfill Invariant 4.5.5 (Step 4.19). If g_{vs} is not consistent with i, an alternative DB_g must be searched using *d_select_alternative*(g_{vs} , DB_g , n_g) (Step 4.20). Note that in this case the third argument (i.e., the number of g-bounds DB_g is already consistent with) is n_g. This fulfills Invariant 4.5.1, and Invariant 4.5.3 to Invariant 4.5.10. If g_{vs} is consistent with i, DB_g does not have to be changed. Because of the update of all J_s in DB_g, Invariant 4.5.11 might be violated though. Therefore the top of DB_g must in any case (i.e., whether g_{vs} was consistent with i or not) be updated in *generalize_disjuncts*(DB_g) (Step 4.21). As a consequence, also in this case all invariants hold at the end of the while loop. Consequently Invariant 4.5.1 to Invariant 4.5.11 hold each time the body of the while-loop has been completely executed, and therefore also when the while-loop ends.

Backtracking

In this section we will explain how d_select_alternative (see Algorithm 4.5) works. According to the specification given in the previous section, we have to show that, given

- DB_g fulfilling Invariant 4.5.3 to Invariant 4.5.10,
- g_{vs} is maximally general, and consistent with $I_g[1]$, . . . , $I_g[ind]$, and
- if g_{vs} is consistent with I_s, then it is MS,

d_select_alternative(g_{vs} , DB_g , *ind*) returns a new g_{vs} and DB_g fulfilling Invariant 4.5.1 and Invariant 4.5.3 to Invariant 4.5.10.

For the while-loop of d_select_alternative Invariant 4.5.3 to Invariant 4.5.10 are satisfied. Invariant 4.5.1 is not; instead we have the following:

- **Invariant 4.5.12.** g_{vs} is a maximally general MS concept representation consistent with $I_g[1], \ldots, I_g[ind]$ and with $I_s[1], \ldots, I_s[n_s]$.

At the end of the while-loop, we also have $ind = n_g$. Consequently Invariant 4.5.1 will then be fulfilled.

We will first prove that the invariants hold before the while-loop. Invariant 4.5.3 to Invariant 4.5.10 hold because of the preconditions of d_select_alternative. Furthermore from these preconditions we also know that g_{vs} is consistent with $I_g[1], \ldots, I_g[ind]$, and that if it is consistent with I_s, it is MS. Consequently, if g_{vs} is consistent with I_s, Invariant 4.5.12 is fulfilled. Otherwise, given DB_g fulfilling Invariant 4.5.3 to Invariant 4.5.10, fulfilling Invariant 4.5.3 except for the top choicepoint, and such that g_{vs} is not consistent with I_s, next_disjunction returns g_{vs} and DB_g fulfilling Invariant 4.5.3 to Invariant 4.5.12 (Step 4.25).

Entering the while-loop means that additionally $ind < n_g$ is true. This means that after incrementing *ind* by 1 (Step 4.26), $I_g[ind]$ is still well-defined. Furthermore Step 4.26 only affects Invariant 4.5.12: g_{vs} is now a maximally general MS concept representation consistent with $I_g[1] \ldots I_g[ind-1]$ and with I_s. Consequently, if $g_{vs} \sim I_g[ind]$ is also true, Invariant 4.5.12 is fulfilled. Otherwise, given DB_g fulfilling Invariant 4.5.3 to Invariant 4.5.10 and the index *ind* such that g_{vs} is a maximally general MS concept representation consistent with $I_g[1], \ldots, I_g[ind-1]$ and with I_s, but not with $I_g[ind]$, specialize_disjuncts returns a new g_{vs} and a new d_stack DB_g fulfilling Invariant 4.5.4 to Invariant 4.5.10 (Step 4.27). Furthermore Invariant 4.5.3 is true except for the top choicepoint of DB_g. The disjunction g_{vs} (and thus the top choicepoint of DB_g) is consistent with $I_g[1] \ldots I_g[ind]$, and *if* it is consistent with I_s, *then* g_{vs} is MS. If then

```
procedure d_select_alternative ( $g_{vs}$: disj_concept; $DB_g$: d_stack; $ind$: index )
          returns disj_concept,d_stack
   {Requires:$DB_g$ fulfills Invariant 4.5.3 to Invariant 4.5.10;
   $g_{vs}$ is maximally general, and consistent with $I_g[1]$ , ..., $I_g[ind]$;
   if $g_{vs}$ is consistent with $I_s$, then it is MS;
   Returns:$g_{vs}$ and $DB_g$ fulfilling Invariant 4.5.1 and Invariant 4.5.3 to Invariant 4.5.10. }
   if $\neg( g_{vs} \sim I_s )$
   then $g_{vs}$, $DB_g$, $ind$ := $next\_disjunction( DB_g )$ {4.25}
   while $ind < n_g$
       do $ind_g := ind_g + 1$ {4.26}
          if $\neg( g_{vs} \sim I_g[ind] )$
          then $g_{vs}$, $DB_g$ := $specialize\_disjuncts( DB_g , ind )$ {4.27}
               if $\neg( g_{vs} \sim I_s )$
               then $g_{vs}$, $DB_g$, $ind$ := $next\_disjunction( DB_g )$ {4.28}
   endwhile
   return $g_{vs}$, $DB_g$
endproc

procedure next_disjunction($DB_g$: d_stack ) returns disj_concept,d_stack,index
   {Requires:$DB_g$ fulfills Invariant 4.5.3 to Invariant 4.5.10;
   $DB_g$ fulfills Invariant 4.5.3 except for the top choicepoint;
   $g_{vs}$ is not consistent with $I_s$;
   Returns:$g_{vs}$ and $DB_g$ fulfilling Invariant 4.5.3 to Invariant 4.5.10, and Invariant 4.5.12;
   Fails:iff no such $g_{vs}$ and $DB_g$ exist.}
   $DL$, $ind$, $DB_g$ := $pop( DB_g )$ {4.29}
   repeat
        while $is\_last\_global\_choice( DL )$
            do if $DB_g = \emptyset$
               then failure {4.30}
               else $DL$, $ind$, $DB_g$ := $pop( DB_g )$ {4.31}
        endwhile
        $g_{vs}$, $DL$ := $next\_global\_choice( DL )$ {4.32}
   until $g_{vs} \sim I_s$ {4.33}
   $DB_g$ := $push( DL , ind , DB_g )$ {4.34}
   return $g_{vs}$, $DB_g$, $ind$
endproc
```

Algorithm 4.5 Backtracking in DITVS

$g_{vs} \sim I_s$, Invariant 4.5.12 follows immediately. Otherwise, next_disjunction (Step 4.28) returns g_{vs} and DB_g fulfilling Invariant 4.5.3 to Invariant 4.5.10 and Invariant 4.5.12. Consequently, Invariant 4.5.3 to Invariant 4.5.10 and Invariant 4.5.12 are fulfilled after the while-loop.

We will now prove that the procedure next_disjunction is correct according to its specification. Given DB_g fulfilling Invariant 4.5.3 to Invariant 4.5.10, fulfilling Invariant 4.5.3 except for the top choicepoint, and such that g_{vs} is not consistent with I_s, next_disjunction returns g_{vs} and DB_g fulfilling Invariant 4.5.3 to Invariant 4.5.10 and Invariant 4.5.12. From Invariant 4.5.4 and $\neg(g_{vs} \sim I_s)$ follows that for all $d \in \mathcal{DL}_C$: if $d \sim I$ and d is MS, d is more specific than an alternative for g_{vs} on DB_g. Consequently, if the topmost choicepoint on DB_g that does not contain a last global choice is replaced by its *next global choice*, Invariant 4.5.4 is still fulfilled. If no such next global choice exists, no disjunction consistent with I exists, and DITVS should fail.

First the top choicepoint is popped from DB_g (Step 4.29). While the just popped choicepoint contains a last global choice, Step 4.31 keeps popping choicepoints of DB_g. If at a certain point DB_g becomes empty, DITVS fails (Step 4.30). While popping elements from DB_g none of the other invariants becomes violated. Then the next global choice is selected in the just popped choicepoint (Step 4.32). If this next global choice is consistent with I_s, Invariant 4.5.12 is fulfilled. Otherwise, the same conditions hold as before the repeat-loop. Therefore the repeat-loop can be repeated. After the repeat-loop Step 4.34 pushes the choicepoint $c(DL , ind)$ on DB_g. With respect to the last popped choicepoint the particular local choices Ch_L could have changed by next_global_choice, but not the VS-lists L themselves. Consequently, the only invariant affected is Invariant 4.5.3. However, because of Constraint 4.54 (the Order Constraint), no alternative for g_{vs} on DB_g will be a subset of g_{vs}. Therefore g_{vs} is MS. Consequently, the postconditions of next_disjunction are fulfilled.

If the order fulfills Constraint 4.54 (the Order Constraint) all subdisjunctions of a disjunction g_d will be considered before g_d. However, if g_d is not consistent with I_s, none of its subdisjunctions will be consistent with I_s. Therefore it might be useful to check consistency of g_d with I_s first, before all subdisjunctions of g_d are enumerated and tested in Step 4.32 and Step 4.33: if g_d is not consistent, then its subdisjunctions should not be considered.

Specialization

In this section we show how specialize_disjuncts (see Algorithm 4.6) works.

Given DB_g fulfilling Invariant 4.5.4 to Invariant 4.5.10 and the index *ind* such that g_{vs} (the disjunction corresponding to the top choicepoint of DB_g) is a maximally general MS concept representation consistent with $I_g[1], \ldots, I_g[ind-1]$ and with I_s, but not with $I_g[ind]$, specialize_disjuncts returns g'_{vs} (an updated g_{vs}) and a d_stack DB'_g (an updated DB_g) fulfilling Invariant 4.5.4 to Invariant 4.5.10 (Step 4.27). Furthermore Invariant 4.5.3 is true except for the top choicepoint of DB'_g. The disjunction g'_{vs} (and thus the top choicepoint of DB'_g) is consistent with $I_g[1] \ldots I_g[ind]$, and *if* it is consistent with I_s, *then* g'_{vs} is MS.

Of these postconditions, Invariant 4.5.4 to Invariant 4.5.10 already hold at the start. The other postconditions only concern g'_{vs} and therefore also the top choicepoint of DB'_g. We will add a choicepoint C_B on top of DB_g such that the latter conditions are fulfilled, by specializing the inconsistent disjuncts of g_{vs}. Meanwhile Invariant 4.5.4 to Invariant 4.5.10 should not be violated. Note that g_{vs}, the disjunction represented by the top choicepoint, is MS. This means that if we add C_B, consisting of specializations d_i of the disjuncts of g_{vs}, the disjunction of the father-disjuncts of the d_i is MS. We will need this observation at the end of this argument.

Since g_{vs} is not consistent with $I_g[ind]$, at least one of its disjuncts must be inconsistent with $I_g[ind]$. specialize_disjuncts first selects the list of global disjuncts DL of the top choicepoint of DB_g (Step 4.35). For each global disjunct $d(L , Ch_L)$ in DL, and for each Versionspace

```
procedure specialize_disjuncts ( $DB_g$: d_stack; $ind$: index )
        returns disj_concept,d_stack
  {Requires:$DB_g$ fulfills Invariant 4.5.4 to Invariant 4.5.10; $g_{vs}$ is maximally general,
  MS, and consistent with $I_g[1], \ldots, I_g[ind-1]$, $I_s$, but not with $I_g[ind]$;
  Returns:$g'_{vs}$ and $DB'_g$ fulfilling Invariant 4.5.4 to Invariant 4.5.10;
  Invariant 4.5.3 is true except for the top choicepoint of $DB'_g$;
  $g'_{vs}$ is consistent with $I_g[1] \ldots I_g[ind]$, and if it is consistent with $I_s$, then $g'_{vs}$ is MS.}
  $DL, ind_0 := top( DB_g )$ {$ind_0$ is not relevant} {4.35}
  $DL' := \emptyset$
  $g'_{vs} := \emptyset$
  for all $d( L , Ch_L )$ in $DL$
     do for all $vs( g , J_s , Scr )$ in $Ch_L$
           do if $g \sim I_g[ind]$
              then $L' := [ vs( g , J_s , Scr ) ]$ {4.36}
              else $specs := specializations( g , I_g[ind] )$ {4.37}
                   $specs$ := select all c from $specs$
                          with $d\_max\_general( c , g_{vs} , DB_g )$ {4.38}
                   $L' := \emptyset$
                   for all $g' \in specs$
                      do $J'_s := \{ ind' \in J_s \mid g' \sim I_s[ind'] \}$ {4.39}
                         if $J_s = J'_s$
                         then $Scr' := prune\_and\_reuse( Scr , g' )$ {4.40}
                         else $Scr' := scr( \bot , \emptyset , 0 )$ {4.41}
                         $L' := L' \uplus [ vs( g' , J'_s , Scr' ) ]$ {4.42}
                   endfor
              $DL' := DL' \uplus [ d( L' , init\_choice( L' ) ) ]$ {4.43}
              $g'_{vs} := g'_{vs} \cup \{ g' \mid vs( g' , J'_s , Scr' ) \in init\_choice( L' ) \}$ {4.44}
        endfor
  endfor
  $DB'_g := push( DL' , ind , DB_g )$ {4.45}
  return $g'_{vs}$, $DB'_g$
endproc
```

Algorithm 4.6 Specialization in DITVS

$vs(\ g\ ,\ J_s\ ,\ Scr\ )$ in Ch_L, g is checked for being consistent with $I_g[ind]$ (i.e., each disjunct of g_{vs} is checked). If g is consistent with $I_g[ind]$ it does not have to be specialized: a new VS-list L' containing this one Versionspace is created (Step 4.36). Otherwise, if g is not consistent with $I_g[ind]$, all maximally general specializations of g consistent with $I_g[ind]$ are computed (Step 4.37). Only those that are maximally general are selected, because the maximally general disjunctions consist of maximally general elements of $\mathcal{L}_C$ consistent with I_g (see further). From the resulting list *specs* another VS-list L' is constructed. L' is initialized as the empty list (Step 4.38). For each g' in *specs*, a versionspace $vs(\ g'\ ,\ J'_s\ ,\ Scr'\ )$ is constructed to be added to L'. First the set J'_s is computed (Step 4.39). J'_s is a subset of J_s, because of Theorem 3.14. From Invariant 4.5.5 and because $g' \preccurlyeq g$, we have again Invariant 4.5.5. If the set $J'_s = J_s$, the set represented by Scr' will be a subset of the set represented by Scr. Given Scr and g' prune_and_reuse will return Scr', which represents the subset of Scr of elements more specific than g (Step 4.40). The resulting Scr' fulfills Invariant 4.5.7 to Invariant 4.5.10. If $J'_s \subset J_s$ (i.e., $J'_s \neq J_s$), the elements in Scr are overgeneral for Scr', and Scr' will have to be recomputed. This will only be done later: therefore the representation $scr(\ \bot\ ,\ \emptyset\ ,\ 0\ )$ of an empty set is taken as Scr' (Step 4.41). This also fulfills Invariant 4.5.7 to Invariant 4.5.10. Reusing Scr might save some recomputations of Scr' from $\bot$, but will not always be possible. Each Versionspace $vs(\ g'\ ,\ J'_s\ ,\ Scr'\ )$ is added[4] to L' (Step 4.42). This means only specializations of g are added to L'; each of them has g as father-concept and $vs(\ g\ ,\ J_s\ ,\ Scr\ )$ as father-disjunct. Since the Versionspace $vs(\ g\ ,\ J_s\ ,\ Scr\ )$ is on the top of DB_g, and since the top of DB_g is not changed, Invariant 4.5.6 is fulfilled. Then the combination of L' and the minimal element in the local order for L' (i.e., *init_choice*(L')) is added to DL' (Step 4.43). The disjunct corresponding to Ch_L is added to g'_{vs} (Step 4.44). Finally, DL' is pushed onto DB_g with index *ind* (Step 4.45). After adding a choicepoint to DB_g, Invariant 4.5.3 is only valid for the choicepoints not on top of DB_g. Since the father-concepts of the disjuncts of g'_{vs} are consistent with $I_g[1] \ldots I_g[ind-1]$, g'_{vs} is consistent with $I_g[1] \ldots I_g[ind]$. Moreover, each element added to g'_{vs} is consistent with $I_g[ind]$. Consequently Invariant 4.5.3 is fulfilled. The disjunction g'_{vs} is maximally general because of Invariant 4.5.3 and Theorem 4.53 (see Section 4.5.4). Finally g'_{vs} fulfills the minimal set criterion, because all local choices in DL' are initial choices, i.e., singletons. Suppose one of these disjuncts could be dropped. Then the disjunction of the father-concepts would not be MS because of Proposition 4.50. This contradicts the observation made in the beginning of this section.

Then we still have to prove Invariant 4.5.4. From Invariant 4.5.4 in the preconditions of specialize_disjuncts, each $d \in \mathcal{DL}_C$ consistent with I and MS is more specific than g_{vs} or than some alternative for g_{vs}. Suppose $d \sim i$ and $d \preccurlyeq_d g_{vs}$, then each disjunct d_j of d must be more specific than some disjunct of c_j of g_{vs}. Furthermore $d_j \sim I_g[ind]$. Consequently, because of Constraint 3.18 (the Boundedness Constraint) there is some $x_j \in$ *specializations*($c_j\ ,\ I_g[ind]$) such that $d_j \preccurlyeq x_j \preccurlyeq c_j$. Since the disjunction of these x_j is equal to g'_{vs} or to an alternative, this part is proven. Suppose $d \sim i$ and d is more specific than some alternative of g_{vs}. Then this alternative is still on DB_g. Consequently, Invariant 4.5.4 remains valid.

Maximal Generality and Optimal Refinement

To make sure a disjunction is maximally general, we have to check whether the disjuncts are maximally general. This is what is checked in Step 4.38 of Algorithm 4.6 with the function d_max_general. Given an element $c \in \mathcal{L}_C$, a disjunction g_{vs} and a d_stack DB_g, d_max_general checks whether c is maximally general and consistent with I_g, or whether c will be generated as a disjunct of g_{vs} by specializing an alternative on DB_g. The check is implemented as explained in Section 4.5.2, under Constraint 4.55. First it is checked whether c is more specific than a disjunct

[4]The operation $List_1 \uplus List_2$ appends $List_2$ at the end of $List_1$.

```
procedure d_max_general ( c: concept; $g_{vs}$: disj_concept; $DB_g$: d_stack )
          returns boolean
   {Returns: true iff c is maximally general and consistent with $I_g$, or c will be generated
   as a disjunct by specializing another concept representation which is still to be explored. }
   $c' := c$ {4.46}
   if $\exists g \in g_{vs} : c \preccurlyeq g$
   then return false {4.47}
   else while father_disjunct( $c'$ ) exists
            do Let $d( L , Ch_L ) =$ father_disjunct( $c'$ )
               Let $g =$ father_concept( $c'$ )
               if $\exists g' \in L$ where $g'$ comes after $g$ in the local order on $L$ and $c \preccurlyeq g'$
               then return false {4.48}
               $c' := g$ {4.49}
        endwhile
   return true
endproc
```

Algorithm 4.7 Maximal Generality and Optimal Refinement in DITVS

of g_{vs}. If it is, c is not maximally general. In this case, *false* is returned (Step 4.47). Otherwise, the concept representation c' follows the chain of ancestor-disjuncts of c, i.e., during the while loop c' is always an ancestor-concept of c. Initially c' is set to c (Step 4.46). As long as the bottom of DB_g is not reached (i.e., as long as c' still has a father- disjunct), the father-disjunct $d(L , Ch_L)$ of c' is considered: if there exists a $g' \in L$ which comes after the father disjunct g of c' which is more general than c, then c is not maximally general, or will be considered as a disjunct of g' (see Section 4.5.3). In that case *false* is returned (Step 4.48). Otherwise, if there exists no such g', the father-concept g of c' is assigned to c' (Step 4.49). Hence, c' is still an ancestor-concept of c. When c' reaches the bottom of DB_g, c must be maximally general, and will not be considered when specializing another concept representation.

The function d_max_general does not affect any of the datastructures and therefore any of the invariants.

Generalization

Given a d_stack DB_g fulfilling Invariant 4.5.3 to Invariant 4.5.10, generalize_disjuncts returns s_{vs} fulfilling Invariant 4.5.2 and DB_g fulfilling Invariant 4.5.3 to Invariant 4.5.11. The procedure generalize_disjuncts updates DB_g such that all versionspaces corresponding to disjuncts of g_{vs} are updated, i.e., made consistent with all n_s s-bounds. At the same time all disjuncts of s_{vs} are collected.

First the top of DB_g is popped (Step 4.50). The global Disjunctlist DL is to be updated. All updated global disjuncts will be collected in the list DL'. Initially DL' is empty. Also s_{vs} is empty. For all global disjuncts $d(L , Ch_L)$ in DL, all updated VS-lists will be collected in L'. The list L' is for each global disjunct initialized as empty. All VS-lists $vs(g , J_s , scr(s , B_s , ind_s))$ in Ch_L are to be updated, i.e., s has to be consistent with $I_s|J_s$, and B_s has to contain all information to compute all alternatives for s. From Invariant 4.5.7 follows that $s \preccurlyeq g$. Furthermore $s \preccurlyeq g$ and s is maximally specific and consistent with $I_s[1, \ldots, ind_s] \mid J_s$ and I_g. From the preconditions of generalize_disjuncts follows that B_s fulfills Invariant 4.5.8 to Invariant 4.5.10. Therefore

```
procedure generalize_disjuncts ( $DB_g$: d_stack )
          returns disj_concept,d_stack
   { Requires: $s_{vs}$ and $DB_g$ fulfill Invariant 4.5.3 to Invariant 4.5.10;
   Returns: $s_{vs}$ and $DB_g$ fulfilling Invariant 4.5.2 to Invariant 4.5.11. }
   $DL, ind_g, DB_g := pop( DB_g )$ {4.50}
   $DL' := \emptyset$ {4.51}
   $s_{vs} := \emptyset$ {4.52}
   for all $d( L , Ch_L )$ in $DL$
      do $L' := \emptyset$ {4.53}
         for all $vs( g , J_s , scr( s , B_s , ind_s ) )$ in $Ch_L$
            do $s, B_s := d\_generalize( s , B_s , ind_s , J_s , g )$ {4.54}
               $L' := L' \uplus [ vs( g , J_s , scr( s , B_s , n_s ) ) ]$ {4.55}
               $s_{vs} := s_{vs} \cup \{ s \}$ {4.56}
         endfor
         $DL' := DL' \uplus [ d( L' , init\_choice( L' ) ) ]$ {4.57}
   endfor
   $DB_g := push( DL' , ind_g , DB_g )$ {4.58}
   return $s_{vs}, DB_g$
endproc

procedure d_generalize ( $s$: concept; $B_s$: stack; $ind_s$: index; $J_s$: set of index; $g$: concept )
          returns concept, stack
   {Requires: $s \preccurlyeq g$ and $s$ is maximally specific and consistent with $I_s[1, \ldots, ind_s]$ and $I_g$;
   $B_s$ fulfills Invariant 4.5.8 to Invariant 4.5.10;
   Returns: $s$ maximally specific and consistent with $I_s|J_s$, and
   $B_s$ fulfilling Invariant 4.5.8 to Invariant 4.5.10; }
   Fails: iff no such $s$ and $B_s$ exist. while $ind_s \neq n_s$
      do $ind_s := ind_s + 1$
         if $\boxed{ind_s \in J_s}$ and $\neg( s \sim I_s[ind_s] )$
         then $gens := generalizations( s , I_s[ind_s] )$
              $gens$ := select all $c$ from $gens$
                        with $\boxed{c \preccurlyeq g}$ and $max\_specific( c , B_s )$ {4.59}
              $s, B_s, ind_s := select\_alternative( gens , B_s , ind_s )$ {4.60}
   endwhile
   return $s, B_s$
endproc
```

Algorithm 4.8 Generalization in DITVS

d_generalize(s , B_s , ind_s , J_s , g) returns an s fulfilling Invariant 4.5.7 (i.e., maximally specific and consistent with $I_s|J_s$), and B_s fulfilling Invariant 4.5.8 to Invariant 4.5.10. Such s and B_s exist, since g is consistent with I; consequently, there must be a maximally specific s which is more specific than g and consistent with I. The new versionspace $vs($ g , J_s , $scr($ s , B_s , n_s $)$ $)$ is added to L', and s is added to s_{vs}. In the resulting L' all versionspaces fulfill Invariant 4.5.7 to Invariant 4.5.11. Each of them is added to DL'. Consequently, all versionspaces in DL' fulfill Invariant 4.5.7 to Invariant 4.5.11. Furthermore for none of the versionspaces neither g nor J_s have changed, no versionspaces are added, and none are omitted. After pushing DL' on top of DB_g with the index ind_g of DL, Invariant 4.5.3 to Invariant 4.5.11 are therefore fulfilled. Moreover, s_{vs} is almost maximally specific under g_{vs} (Invariant 4.5.2). Consequently, s_{vs} and DB_g returned by generalize_disjuncts fulfill Invariant 4.5.2 to Invariant 4.5.11.

The procedure d_generalize is almost identical to procedure generalize in Algorithm 3.4, except for the differences which are shown by the boxes. Instead of generalizing w.r.t. all elements in I_s, d_generalize only generalizes w.r.t. elements in $I_s \mid J_s$. Furthermore, instead of testing the resulting generalizations for being consistent with I_g, they are tested for being more specific than g. The result is a concept s maximally specific and consistent with $I_s \mid J_s$, and a stack B_s of alternatives for s.

Some operations on stacks

Algorithm 4.9 contains some of the auxiliary procedures we used in the previous sections. Given a d_stack DB_g and an index n_s in I_s, update_Js updates the sets J_s of all elements $vs($ g , J_s , Scr $)$ in all VS-lists in all choicepoints of DB_g: if $g \sim I_s[n_s]$, n_s is added to the corresponding J_s. If Invariant 4.5.5 was true in all these Versionspaces for $n_s - 1$, it will now be true for n_s.

Given a structure $scr($ s , B_s , ind_s $)$ containing a concept, a stack and an index in I_s, which is the representation of a set S, and given a concept g, prune_and_reuse returns a structure $scr($ s , B_s , ind_s $)$ which is the representation of the *subset* of S of elements more specific than g. It uses therefore the procedure prune_stk which is in structure completely analogous to prune_stack of Algorithm 3.6. The procedure select_alternative is described in Algorithm 3.6: it selects an alternative for s on B_s, and returns this alternative, the rest of B_s and the index ind_s in I_s up to where s is consistent with I_s.

The procedure prune_and_reuse will make Invariant 4.5.7 to Invariant 4.5.10 true, because it removes all elements from B_s not more specific than g, and if necessary also s itself. Consequently, Invariant 4.5.7 to Invariant 4.5.9 remain true for the elements that remain on B_s. To prove that Invariant 4.5.10 holds: suppose that c is more specific than g and consistent with $I_s|J_s$. Then c is more specific than the father-concept of g, for which Invariant 4.5.10 holds. Consequently, s or an alternative s' for s on B_s is more specific than c. But then s or s' is also more specific than g, which means that it is still on B_s after pruning.

The order of searching DB_g

In this section we define a specific local order within VS-lists, and a global order on a Disjunctlist, i.e., a list of VS-lists (see Section 4.5.1). The resulting global order must fulfill Constraint 4.54 for the minimal set criterion. The local order is based on the positions of the elements of Ch_L within L. Similarly, the global order is based on the order of the VS-lists in the Disjunctlist they are in.

Suppose that each subset Ch_L of L is represented as an ordered list[5] of *the positions in L*

[5] We represent a list as an enumeration of its elements separated by commas and surrounded by square

```
procedure update_Js( $DB_g$: d_stack; $n_s$: index ) returns d_stack
    {Returns: a d_stack $DB_g$ obtained by updating the sets $J_s$ of all elements $vs( g , J_s , Scr )$
    in all VS-lists in all choicepoints of $DB_g$: if $g \sim I_s[n_s]$, $n_s$ is added to $J_s$. }
    $DL, ind_g, DB_g := pop( DB_g )$
    $DL' := \emptyset$
    for all $d( L , Ch_L )$ in $DL$
        do $L' := \emptyset$
            for all $vs( g , J_s , Scr )$ in $L$
                do if $g \sim I_s[n_s]$
                    then $J_s := J_s \cup \{ n_s \}$
                    $L' := L' \uplus [ gcr( g , J_s , Scr ) ]$
            endfor
            $DL' := DL' \uplus [ d( L' , Ch_L ) ]$
    endfor
    $DB_g := update\_Js( DB_g , n_s )$
    $DB_g := push( DL' , ind_g , DB_g )$
    return $DB_g$
endproc

procedure prune_and_reuse ( $scr( s , B_s , ind_s )$: scr( concept , stack , index );
            $g$: concept ) returns scr( concept , stack , index )
    {Returns: a structure $scr( s' , B'_s , ind'_s )$ representing all elements of the given
    structure $scr( s , B_s , ind_s )$ more specific than $g$. }
    $B_s := prune\_stk( B_s , g )$
    if $\neg( s \preccurlyeq g )$
    then $s, B_s, ind_s := select\_alternative( \emptyset , B_s , ind_s )$
    return $scr( s , B_s , ind_s )$
endproc

procedure prune_stk( $B_s$: stack; $g$: concept ) returns stack
    {Returns: a stack $B'_s$ obtained by removing all alternatives from $B_s$
    that are not more specific than $g$. }
    if $is\_empty( B_s )$
    then return $\emptyset$
    else $ind, s_{ind}, alt_{ind}, B_s := pop( B_s )$
        $B_s := prune\_stk( B_s , g )$
        $alt_{ind}$ := select all $c$ from $alt_{ind}$ with $c \preccurlyeq g$
        if $alt_{ind} \neq \emptyset$
        then $B_s := push( ind , s_{ind} , alt_{ind} , B_s )$
    return $B_s$
endproc
```

Algorithm 4.9 Operations on stacks in DITVS

of the elements that are in Ch_L (i.e., Ch_L is represented by an ordered set of indexes, instead of by its actual elements). E.g., for $L = [a, b, c, d, e, f]$ the list $[1, 3, 6]$ represents the sublist $[a, c, f]$. Then all sublists of L are ordered according to growing length, i.e., first all sublists of one element, then sublists of two elements, etc. Within a group of lists of the same length, elements are ordered according to the first element in the list where they differ: the list with the smallest element comes first. This results in the following order on sublists of a list L of lenght five:

- $[1], [2], [3], [4], [5]$
- $[1, 2], [1, 3], [1, 4], [1, 5], [2, 3], [2, 4], [2, 5], [3, 4], [3, 5], [4, 5]$
- $[1, 2, 3], [1, 2, 4], [1, 2, 5], [1, 3, 4], [1, 3, 5], [1, 4, 5], [2, 3, 4], \ldots, [3, 4, 5]$
- ...

This defines a local order on each VS-list.

Then we define a *global* order on *lists of VS-lists*, based on the chosen local order on the sublists of each VS-list, as follows. The first element in the global order consists of all minimal elements in each of the VS-lists w.r.t. the local order. For the minimal set criterion, lists of VS-lists can be ordered as follows: elements with less VS-lists come first; if two lists l_1 and l_2 have the same number of VS-lists, the lists are ordered according to the first VS-list that differs: the list with the smallest element comes first. For a disjunction of three VS-lists, each containing two elements, this order would look like this:

- $[[1], [1], [1]]$
- $[[2], [1], [1]]$
- $[[1, 2], [1], [1]]$
- $[[1], [2], [1]]$
- $[[2], [2], [1]]$
- $[[1, 2], [2], [1]]$
- $[[1], [1, 2], [1]]$
- $[[2], [1, 2], [1]]$
- $[[1, 2], [1, 2], [1]]$
- $[[1], [1], [2]]$
- ...

As such, all subdisjunctions of a disjunction will be considered before the disjunction itself is considered, i.e., Constraint 4.54 is satisfied.

brackets, as in PROLOG.

4.5.5 Example

We will now go back to the second example of Section 3.7. In that example $\mathcal{L}_C$ is the direct product of the lattice $\mathcal{M}$ (see Figure 3.11) with itself. Concept representations are couples $c(\ X\ ,\ Y\ )$ with $X, Y \in \mathcal{M}$. In the direct product the relation $\preccurlyeq$ is defined by

$$c(\ X_1\ ,\ X_2\ ) \preccurlyeq c(\ Y_1\ ,\ Y_2\ ) \text{ iff } (X_1 \preccurlyeq Y_1 \text{ and } X_2 \preccurlyeq Y_2).$$

The top element of $\mathcal{L}_C$ is then $c(\ \top\ ,\ \top\ )$ and the bottom element is $c(\ \bot\ ,\ \bot\ )$.

New Information	Stored In
$\neg(\ t \preccurlyeq c(human, \top)\ )$	$I_s[1]$
$\neg(\ c(woman, woman) \preccurlyeq t\ )$	$I_g[1]$
$c(woman, man) \preccurlyeq t$	$I_s[2]$
$c(man, woman) \preccurlyeq t$	$I_s[3]$
$t \preccurlyeq c(animate, animate)$	$I_g[2]$
$\neg(\ t \preccurlyeq c(\top, human)\ )$	$I_s[4]$
$\neg(\ c(man, man) \preccurlyeq t\ )$	$I_g[3]$

Figure 4.10 Information elements of Example 1

On this example ITVS failed because a disjunction is necessary to represent the target concept. The successive information elements are given in Figure 4.10. We have drawn the consecutive stages of DB_g, in the style of Figure 4.9. The top of the figure always represents the bottom of DB_g. Each layer in the figure represents one choicepoint on DB_g. Each layer consists of one or more VS-lists. A VS-list L consists of one or more versionspaces, each represented by a white or shaded rectangle. Shaded rectangles are in Ch_L, white rectangles are not in Ch_L. A rectangle depicting a versionspace $vs(\ g\ ,\ J_s\ ,\ Scr\ )$ shows g and the set J_s. Only for the versionspaces contained in g_{vs}, Scr is shown in a rectangle with rounded corners. A rectangle with rounded corners depicting $scr(\ s\ ,\ B_s\ ,\ ind_s\ )$ only shows s and B_s, because ind_s is always equal to n_s. In the example B_s has never more than one choicepoint $(\ ind\ ,\ s_{ind}\ ,\ alt_{ind}\ )$. If B_s is empty, it is denoted by $\emptyset$. Note that $\top$ is represented by "top", $\bot$ by "bottom".

Figure 4.11 shows DB_g after the first information element has been processed. After the first information element is presented, DB_g has one choicepoint, containing one VS-list, containing one element. The current disjunctive concept representation g_{vs} has only one disjunct: $c(\ top\ ,\ top\ )$. This disjunct is consistent with $I_s[1]$, so the corresponding s must also be consistent with $I_s[1]$. There are several alternatives for s: one of them is chosen as current almost maximally specific disjunction under g (in particular: $s = c(inanimate, bottom)$). The alternatives $c(female, bottom)$ and $c(male, bottom)$ are on the stack B_s.

After the second information element, a g-bound, is processed, DB_g contains a new choicepoint, with one VS-list (see Figure 4.12). Of this VS-list one element (the first one: $c(inanimate, top)$) is included in g_{vs}, since this one disjunct is consistent with all (i.e., one) s-bounds. For this disjunct, $s = c(inanimate, bottom)$ and B_s are reused from its father

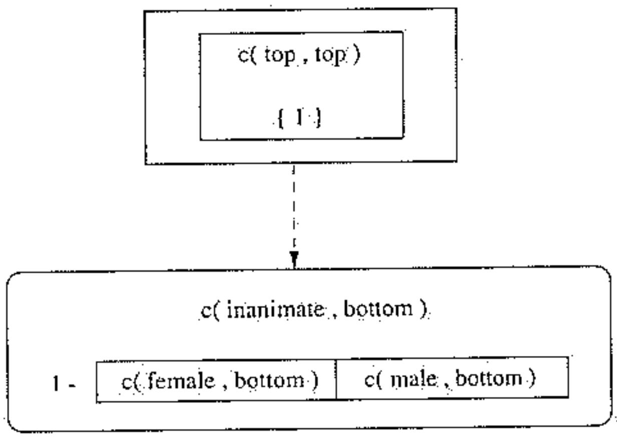


Figure 4.11 DB_g after processing $I_s[1] = \neg(\ t \preccurlyeq c(human, \top)\ )$

disjunct; note however that the corresponding stack B_s has been pruned, because the alternatives $c(female, bottom)$ and $c(male, bottom)$ are not consistent with $c(inanimate, top)$.

After the third information element, again an s-bound, the previous choice in the VS-list of the top choicepoint in DB_g has been changed, because $c(inanimate, top)$ is not consistent with $I_s[2]$ (see Figure 4.13). The concept representation $c(male, top)$ is skipped, because it is also not consistent with $I_s[2]$. The concept representation $c(top, male)$ however is consistent with both $I_s[2]$ and $I_s[1]$. The s under this disjunct is $c(female, man)$, which has one alternative $c(male, bottom)$ on B_s.

After reading the next s-bound, a further global choice must be taken in the top choicepoint of DB_g (see Figure 4.14). The fourth element in the VS-list is only consistent with $I_s[1]$. Consequently more than one disjunct appears in g_{vs}, each with a corresponding s. The first combination that is consistent with all s-bounds is the disjunction $c(male, top) \vee c(top, male)$. Note that none of the disjuncts can be dropped: both of them are consistent with $I_s[1]$; however, only one is also consistent with $I_s[3]$, while the other is consistent with $I_s[2]$. This is ensured by the order in which we search the combinations, because this order satisfies Constraint 4.54 (the Order Constraint). There is only one maximally specific concept representation under $c(male, top)$: $c(male, female)$. Under $c(top, male)$ there is more than one choice: one is chosen as s (i.e., $c(female, male)$), the other one (i.e., $c(male, bottom)$) is pushed onto B_s.

After reading the next g-bound, a new choicepoint is created. Only those disjuncts included in the previous g_{vs} are specialized (see Figure 4.15). For both there is only one choice. In both cases s and B_s are reused.

The next information element is again an s-bound(see Figure 4.16). Note that all sets J_s on DB_g are updated with each new s-bound. In the top choicepoint s and B_s are updated.

Finally we have the last information element $I_g[3]$. Because none of the disjuncts of g_{vs} is consistent with this g-bound, both are specialized. Now $c(female, male)$ is the only specialization of $c(animate, male)$. The concept representations $c(bottom, animate)$ and

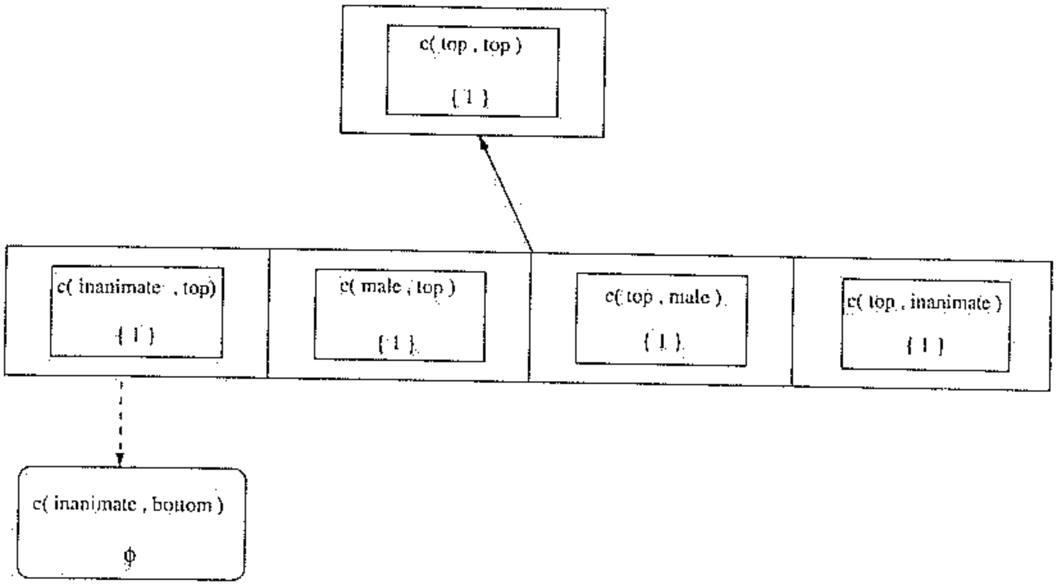


Figure 4.12 DB_g after processing $I_g[1] = \neg(\ c(woman, woman) \preceq t\ )$

$c(male, female)$ are both specializations of $c(male, animate)$. The former, however, in combination with $c(female, male)$ is not consistent with I_s. The latter, in combination with $c(female, male)$ is consistent with I_s. Note that B_s has become empty for both versionspaces. At this point $\mathcal{I}nf$ is empty, and the algorithm halts.

4.5.6 Minimal Length and Covering in DITVS

In this section we will discuss how DITVS can be adapted to obtain ML solutions and how to fit the covering approach (see Section 4.4.1) into the backtracking scheme of Section 4.5.1.

To obtain ML solutions, we can embed Algorithm 4.4 in an iterative deepening loop which places an upperbound n_d on the number of disjuncts that are allowed to be included in g_{vs}. The function `next_disjunction` should therefore only return disjunctions of length maximally equal to n_d. This can be done by changing *last_global_choice*(DL) to return true if DL represents the last global choice of DL *of length maximally* n_d. Similarly, *next_global_choice*(DL), for a DL such that $\neg$*last_global_choice*(DL), should be changed to return only a next global choice of DL *of length maximally* n_d. Whenever $DITVS$ fails to find a solution with maximally n_d disjuncts, it should try to find a solution with maximally $n_d + 1$ disjuncts.

The covering approach is not associated to a particular search strategy as such. Therefore, to obtain the covering approach also, the procedure `select_alternative` will have to be reimplemented. However, an implementation of the covering approach can still use the same framework. We will now briefly show how this can be realized:

- For an s-bound i inconsistent with g_{vs} an extra disjunct will be added (see Figure 4.18). This means that somewhere on DB_g a Versionspace in one of the VS-lists L that is not in the corresponding Ch_L, is selected and added to Ch_L. Then this

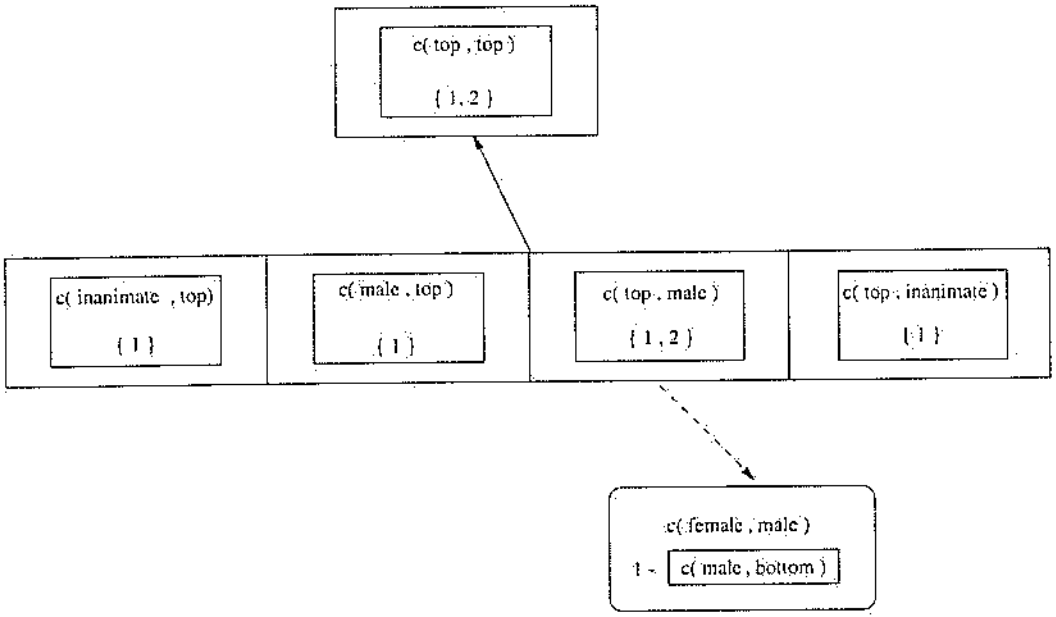


Figure 4.13 DB_g after processing $I_s[2] = c(woman, man) \preccurlyeq t$

choice is to be specialized until it is consistent with all g-bounds. Note that finding a new disjunct implies the search for an element of $\mathcal{L}_C$ consistent with I_g and i, comparable to what happens in `select_alternative`. This means also that all choicepoints closer to the top choicepoint of DB_g than L have to be updated, or have to be created if necessary.

In Figure 4.18 the bold subtree on the right is an update of DB_g. A versionspace is added in the choicepoint with index $ind_{g,1}$. The choicepoints with index $ind_{g,2}$ and index $ind_{g,3}$ have to be updated as well.

- For a g-bound i inconsistent with g_{vs} all inconsistent disjuncts will be removed from the lists Ch_L they are in (see Figure 4.19). If a certain list Ch_L becomes empty, the VS-list can be removed from the corresponding list DL. If DL becomes empty, the choicepoint can be removed from DB_g. In Figure 4.19 the disjunct which has been crossed should be removed. This means that Ch_L in its father-disjunct will become empty. Therefore this father-disjunct can be removed. The father-disjunct above however does not have an empty Ch_L, and should therefore not be removed. This makes g_{vs} consistent with i, but not necessarily with I_s. To be consistent with I_s, disjuncts are to be added again.

4.5.7 Complexity analysis [T]

SUMMARY: in this section we discuss the complexity of the three instantiations of the DITVS algorithm discussed in Section 4.5.6.

We will use the same notation as in Section 4.3.2. Additionally we will denote the number of

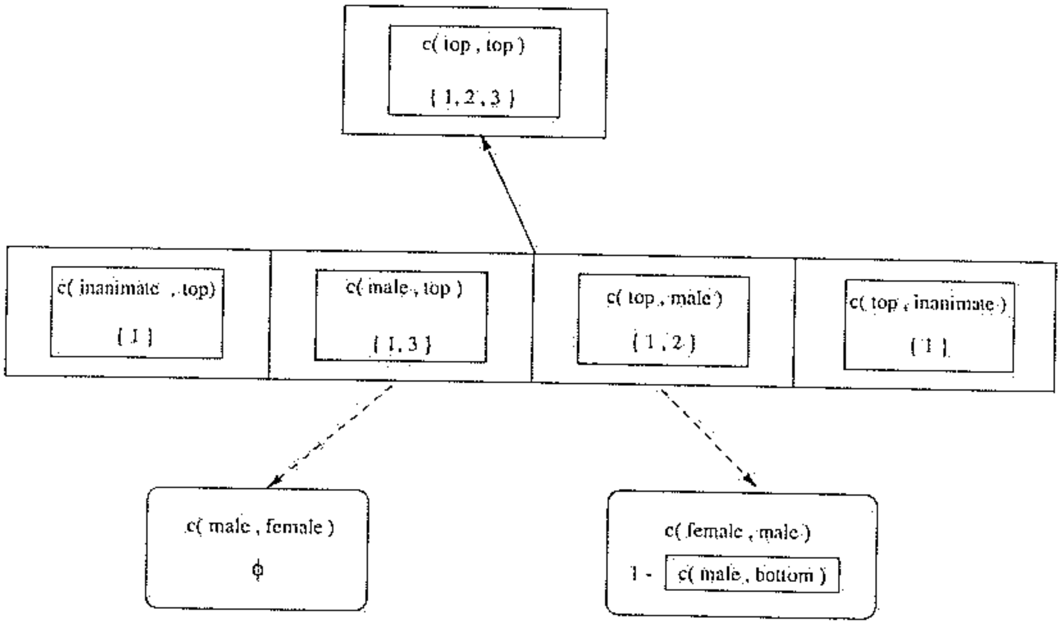


Figure 4.14 DB_g after processing $I_s[3] = c(man, woman) \preccurlyeq t$

disjuncts in g_{vs} as n_d. Note that in the covering approach as well as in the minimal set version of DITVS, n_d is in the worst case equal to n_s (i.e., the number of s-bounds) because of Theorem 4.23.

Theorem 4.56 The worst case space complexity of DITVS is

$$\mathcal{O}(\ n_g \times n_d \times b_g \times (2 + n_s \times b_s) \times c_c + n_g \times n_s \times n_d \times b_g \times c_{ind} + (n_s + n_g) \times c_i\ ).$$

Proof There are in the worst case n_g choicepoints on DB_g, each containing n_d VS-lists of b_g Versionspaces each. Each Versionspace contains g, s, the set J_s and the stack B_s. The stack B_s contains at most $n_s \times b_s$ concept representations. Together with g and s this gives a term $\mathcal{O}(\ n_g \times n_d \times b_g \times (2 + n_s \times b_s) \times c_c\ )$. J_s contains at most (an index to) all n_s elements of I_s. This gives a term $\mathcal{O}(\ n_g \times n_s \times n_d \times b_g \times c_{ind}\ )$. Apart from this I_s and I_g have to be stored completely, which gives another term $\mathcal{O}(\ (n_s + n_g) \times c_i\ )$. □

The major term of the worst case space complexity is of the order $\mathcal{O}(\ n_d \times n_s \times n_g\ )$. The factor n_d arises from the fact that there are basically n_d versionspaces for which information must be stored. The factor $n_g \times n_s$ arises from the fact that we store a backtrack stack B_s in each versionspace on DB_g. If we would only store B_s for the disjunctions of g_{vs} (i.e., only for the versionspaces in the top of DB_g) the factor n_g (the depth of the stack DB_g) would drop. This option would, on the other hand, require that all B_s are always recomputed from $\perp$, i.e., no B_s can be pruned and reused (see prune_and_reuse in Section 4.5.4).

This worst case space complexity is a major gain w.r.t. DDI (see Theorem 4.37), which is exponential in case the size of the general-to-specific search space or the size of the specific-to-general search space is exponential (i.e., when $b_s > 1$ or $b_g > 1$).

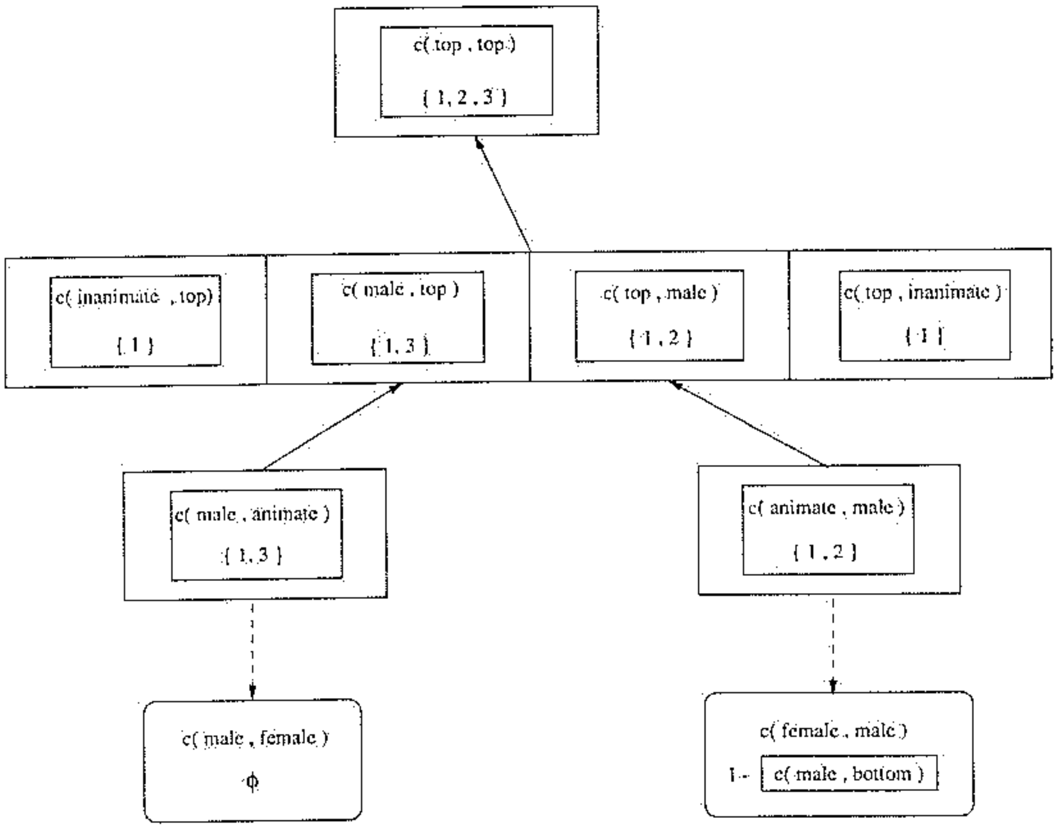


Figure 4.15 DB_g after processing $I_g[2] = t \preccurlyeq c(animate, animate)$

Theorem 4.57 The worst case time complexity for DITVS to compute one maximally general MS disjunction g_{vs} consistent with I and one almost maximally specific s_{vs} under g_{vs} is

$$\begin{gathered} \mathcal{O}(\quad 2^{b_g-1} \times \bar{g} \times c_{spec} + \\ (2^{b_g-1} \times \bar{g} \times (n_g \times b_g + n_s) + n_d \times (n_s + n_g) \times \bar{s} \times (1 + n_s \times b_s)) \times c_{\preccurlyeq} + \\ n_d \times (n_s + n_g) \times \bar{s} \times c_{gen}). \end{gathered}$$

Proof First note that each element g in the general-to-specific search space will, in the worst case, be specialized 2^{b_s-1} times. To see this, consider the VS-list L of which c is a member. There are 2^{b_g-1} possible choices Ch_L of which g is a member and 2^{b_g-1} possible choices Ch_L of which g is not a member. For each of the choices of which c is a member, g needs to be specialized.

For each of these 2^{b_g-1} choices each of the $\bar{g}$ elements of the general-to-specific search space are specialized we have:

- one specialization operation (this yields the term $\mathcal{O}(\ 2^{b_g-1} \times \bar{g} \times c_{spec}\ )$);
- g must be checked for being maximally general. Using the maximally specific test as in ITVS this requires a comparison of g with all concept representations in all ancestor-disjuncts of g. This amounts to $n_g \times b_g$ comparisons in the worst case (this yields the term $\mathcal{O}(\ 2^{b_g-1} \times \bar{g} \times n_g \times b_g \times c_{\preccurlyeq}\ )$);

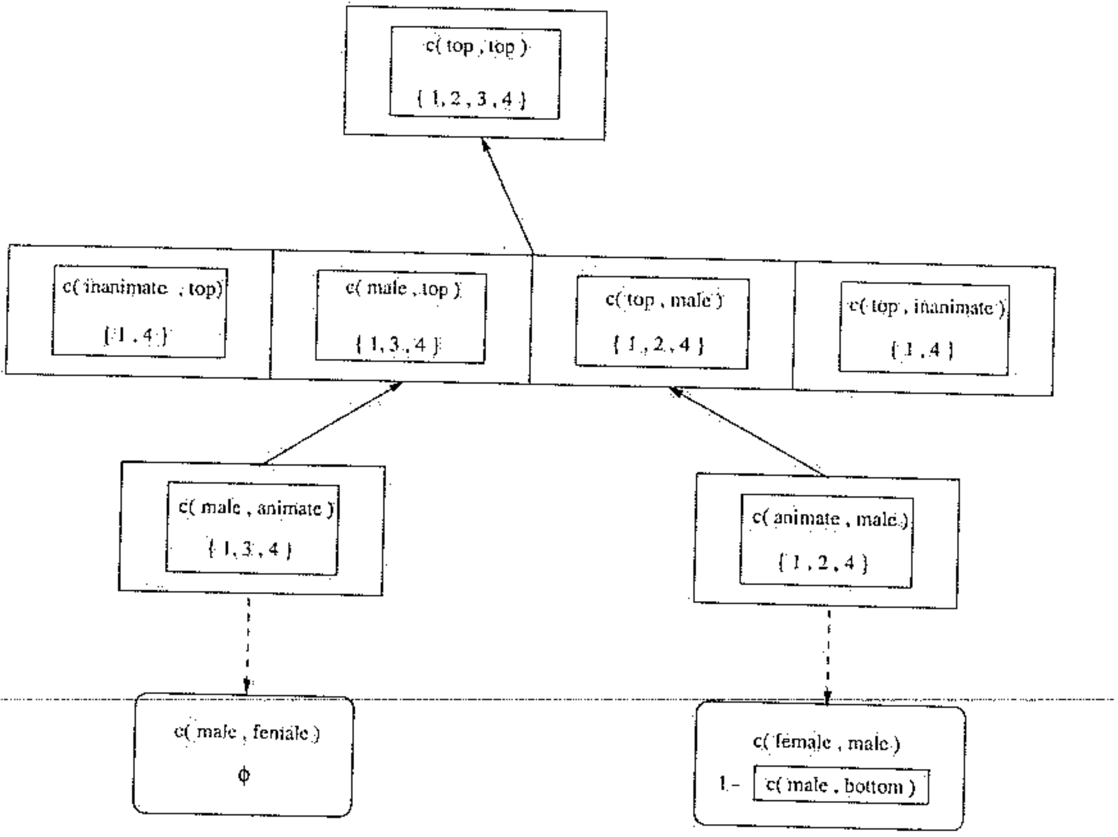


Figure 4.16 DB_g after processing $I_s[4] = \neg(\, t \preccurlyeq c(\top, human)\,)$

- g must be compared to all s-bounds of $I_s|J_s$, with J_s the set of indexes of s-bounds the father-concept of g is consistent with. In the worst case J_s contains all n_s s-bounds (this yields the term $\mathcal{O}(\, 2^{b_g-1} \times \bar{g} \times n_s \times c_{\preccurlyeq}\,)$);

For each of the n_d disjuncts in g_{vs} a corresponding disjunct of s_{vs} must be computed, and this each time a new g_{vs} consistent with I is computed, i.e., in the worst case once for each new information element. For each of these $n_d \times (n_s + n_g)$ updates, we have to search the specific-to-general search space completely, i.e., for each of the $\bar{s}$ elements s of the specific-to-general search space, we have to:

- generalize s once (this yields the term $\mathcal{O}(\, n_d \times (n_s + n_g) \times \bar{s} \times c_{gen}\,)$);
- compare s to g once (this yields the term $\mathcal{O}(\, n_d \times (n_s + n_g) \times \bar{s} \times c_{\preccurlyeq}\,)$);
- compare s to all $n_s \times b_s$ elements of B_s to check whether s is maximally specific (this yields the term $\mathcal{O}(\, n_d \times (n_s + n_g) \times \bar{s} \times n_s \times b_s \times c_{\preccurlyeq}\,)$).

□

W.r.t. ITVS, there is an extra factor 2^{b_g-1} accompanying $\bar{g}$: each operation on a concept representation in the general-to-specific search space is repeated 2^{b_g-1} times in the worst case. Then there is a factor $n_d \times (n_s \times n_g)$ accompanying $\bar{s}$. This factor is caused by the fact that in the

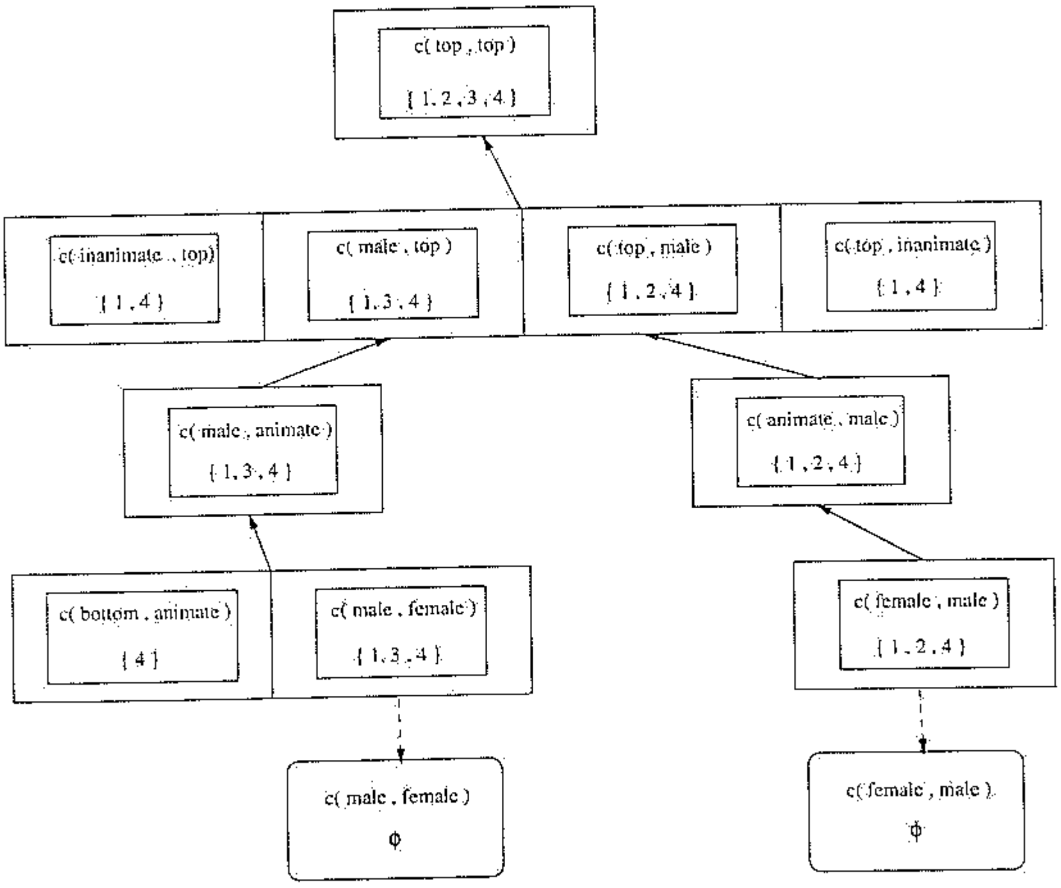


Figure 4.17 DB_g after processing $I_g[3] = \neg(\ c(man, man) \preccurlyeq t\ )$

worst case the specific-to-general search space has to be searched completely n_d times for each of the $n_s + n_g$ information elements (i.e., in the worst case B_s cannot be reused). DDI computes *all* consistent versionspaces, and has therefore a factor $\bar{g}$ accompanying $\bar{s}$. It suffers from the same problem of in the worst case having to recompute the sets S of each versionspace for each new information element. Finally the test for maximal generality or maximal specificity in DITVS is linear in the number of information elements, whereas in DDI it is exponential as soon as $b_s > 1$ or $b_g > 1$. We have the same property as in ITVS: if we want DITVS to generate all maximally general solutions, we have to compare each new candidate solutions to all solutions already found, which would also be exponential in that case.

For the iterative deepening approach to compute ML solutions, the worst case space complexity is the same as in Theorem 4.56. The worst case time complexity is determined by the number n_d of disjuncts in the ML solution found. This is the number of times DITVS must be repeated before this solution is obtained. Consequently, we have to multiply the result of Theorem 4.57 with n_d to obtain the worst case time complexity of the iterative deepening approach to obtain ML solutions.

Theorem 4.58 The worst case time complexity for DITVS to compute one maximally general

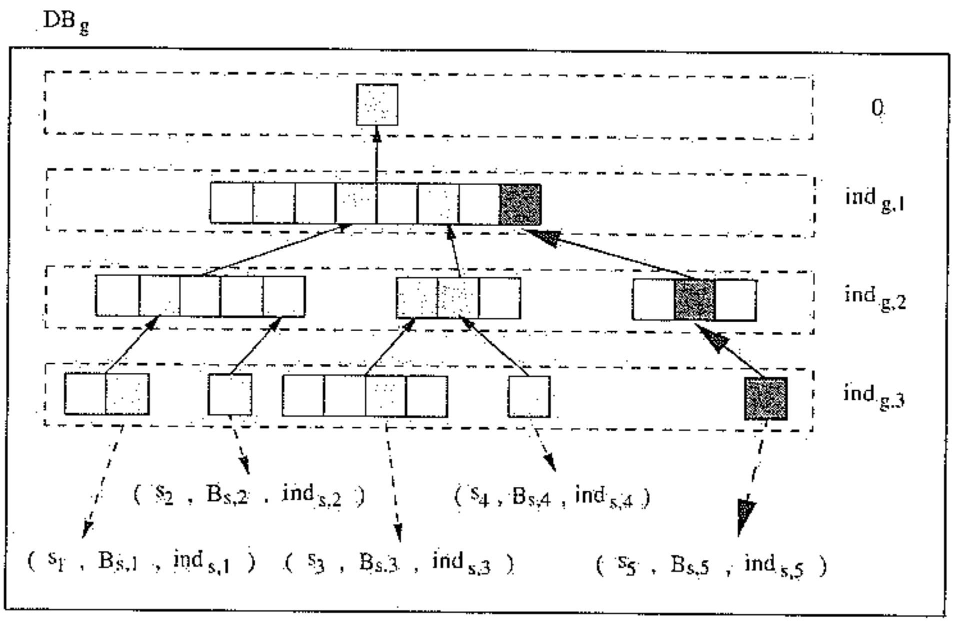


Figure 4.18 Adding a new disjunct in covering approach

ML disjunction g_{vs} consistent with I and one almost maximally specific s_{vs} under g_{vs} is

$$\begin{gathered}\mathcal{O}(\quad n_d \times 2^{b_g-1} \times \bar{g} \times c_{spec} + \\ n_d \times (2^{b_g-1} \times \bar{g} \times (n_g \times b_g + n_s) + n_d \times (n_s + n_g) \times \bar{s} \times (1 + n_s \times b_s)) \times c_{\preceq} + \\ n_d^2 \times (n_s + n_g) \times \bar{s} \times c_{gen}).\end{gathered}$$

□

4.5.8 A short note on the extensions of ITVS in DITVS

At this point we want to elaborate briefly on the extensions of ITVS concerning omitting redundant information elements, shifting the bias, and generating relevant upper- and lowerbounds.

In DITVS omitting redundant information elements (see Section 3.9) is still allowed, because Theorem 3.45 will still hold for each disjunct separately. Theorem 3.45 stated that if an s-bound i_1 is s-prunable w.r.t. an s-bound i_2, or if a g-bound i_1 is g-prunable

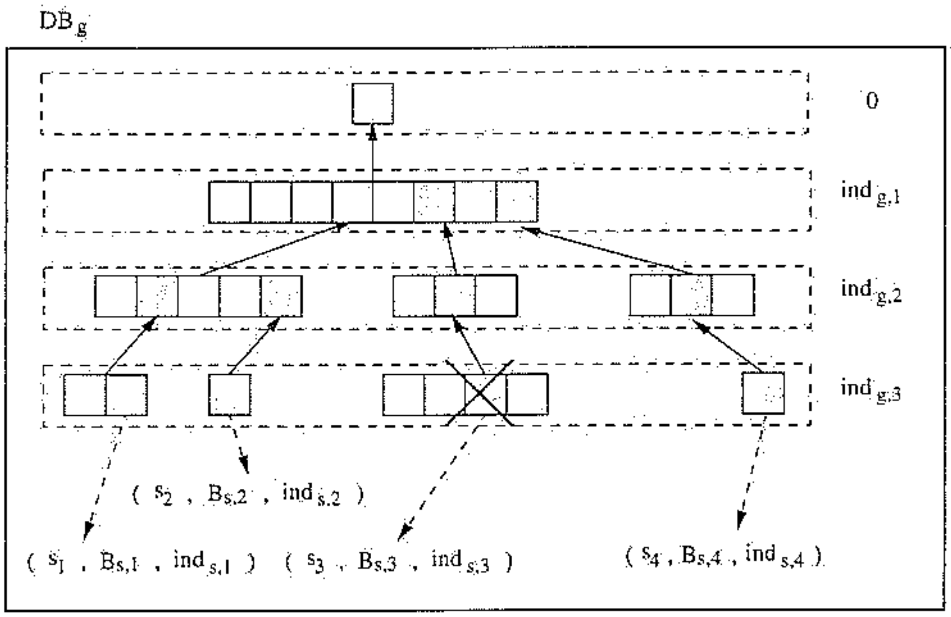


Figure 4.19 Removing a disjunct in covering approach

w.r.t. a g-bound i_2, the fact that a concept representation c is consistent with i_2 always implies that c is consistent with i_1. Section 3.9 also described possibilities of generating new information elements automatically, such that other information elements become s-prunable or g-prunable:

1. replacing two positive lowerbounds by their minimal upperbound, if the latter is unique;
2. replacing two positive upperbounds by their maximal lowerbound, if it is unique;
3. replacing a lower near-miss by a positive upperbound; and
4. replacing an upper near-miss by a negative upperbound.

The first operation is not allowed in DITVS: a disjunction can be consistent with two positive lowerbounds because one disjunct is consistent with one positive lowerbound, and the other disjunct is consistent with the other positive lowerbound. None of the disjuncts is necessarily consistent with both positive lowerbounds, and hence neither with their minimal upperbound. The second operation is still allowed in DITVS: since each disjunct must be consistent with each positive upperbound, it also has to be consistent with the maximal lowerbound, if this is unique. The third operation is also allowed in DITVS: given a lower-near miss i_n w.r.t. a positive lowerbound i_p and with corresponding positive upperbound i_u, one of the disjuncts has to be consistent with i_p and i_n. Consequently, one of the disjuncts has to be consistent with i_p and i_u. Finally, the fourth operation is allowed: given a upper-near miss i_n w.r.t. a positive upperbound i_p and with corresponding positive lowerbound i_l, one of the disjuncts has to be consistent with the i_p and i_n, and consequently with i_p and i_l.

Consequently, only the replacement of positive lowerbounds by minimal upperbounds is not allowed.

The possibility to shift bias by means of a series of languages is still possible, but will be restricted. Instead of shifting to the next language when no concept representation in $\mathcal{L}_C$ is consistent with all information elements, disjunctions will be introduced. Only when $\mathcal{G}_{I_g}$ is empty, and consequently $\mathcal{DG}_I$ is empty (see Definition 4.19), shifting to the next language is necessary (see CLINT [De Raedt, 1992]).

Generating relevant information elements is also possible in DITVS. Relevant lower- and upperbounds can be generated using a disjunct g of g_{vs}, and its corresponding disjunct s in s_{vs}. Because the corresponding disjunct s is always more specific than g, it is not necessary to use Algorithm 3.11 to find an s' more specific than g, or to find a g' more general than s. By classifying the relevant lower- or upperbounds as positive or negative, one of the disjuncts of g_{vs} will be specialized, one of the disjuncts of s_{vs} will be generalized, or a new disjunct will be created.

4.6 Conclusion

In this chapter we have introduced a new concept representation language $\mathcal{DL}_C$ based on $\mathcal{L}_C$ by introducing disjunctions. We have determined under which constraint it is possible to reduce the operations on disjunctions to operations on disjuncts. This constraint determines the applicability of this chapter to learning disjunctive concept representations. In Chapter 5, we will see which restrictions this imposes on the kind of concept representations that can be learned by DITVS applied to ILP.

We have built up a similar framework as in Chapter 3, by describing the versionspace of consistent concept representations in $\mathcal{DL}_C$. An additional preference criterion was needed to restrict the number of disjuncts in the solutions. We have described disjunctive versionspaces in combination with preference criteria in a similar way as in the framework of Versionspaces, namely by means of their maximally general and maximally specific boundaries. We found that the set of all maximally general concept representations fulfilling the minimal length or minimal set criterion are exactly the elements of minimal length of the set of all disjunctions of maximally general disjuncts. We have introduced almost maximally specific concept representations in order to avoid a huge number of maximally specific consistent concept representations. On the basis of these theoretical achievements, we have extended both the Description Identification algorithm and the Iterative Versionspaces algorithm towards disjunctive concept representations. In the same spirit as the

Iterative Versionspaces framework, we could view the resulting datastructures as a framework in which other disjunctive search methods can be described. As an example, we discussed the covering approach.

Disjunctive concept learning has always been considered as a tough problem. The major reason for this is the complexity of the problem. This chapter contributes to the understanding of that complexity, and hopefully also to the further development of algorithms that can handle this complexity.

Chapter 5

(D)ITVS in Inductive Logic Programming

5.1 Introduction

Inductive Logic Programming (ILP) is often said to be in the intersection of Inductive Learning and Logic Programming (e.g., [De Raedt, 1992]). It is the research area in which induction is studied within a logical representation. To this aim it borrows the framework and several techniques from the Logic Programming community. Basically the problems that are studied in ILP are those of Machine Learning in general, formulated in a logical framework. Consequently, the problem of concept learning, one of the major themes in Machine Learning, has also been ported to ILP, and is called the *predicate learning* problem. Currently the representation languages used in ILP are subsets of first order logic. For predicate learning, concepts are represented by (sets of) definite Horn Clauses.

In this chapter we will show that the framework of Iterative Versionspaces (ITVS) and of Disjunctive Iterative Versionspaces (DITVS) also applies to predicate learning. Consequently the properties of our framework will contribute to a better understanding of predicate learning as a search problem. In particular it is important to understand the computational complexities of the search processes involved. Furthermore the ITVS framework allows to search for maximally general and (almost) maximally specific concept representations in an ILP context, and allows the implementation of optimal refinement operators as in ITVS. By implementing predicate learning in the context of ITVS, we also automatically embed it in Mitchell's framework of Versionspaces. Although predicate learning is without doubt considered in this way by many ILP researchers, this formal embedding explicitly relates predicate learning to concept learning.

As we did not touch upon the problem of learning multiple concepts in ITVS, we will not touch upon the *multiple predicate learning* problem, for which we refer to *theory revision* literature (see Chapter 2). As in the framework of (D)ITVS, we assume the background knowledge (see Chapter 2) is correct. Another respect of ILP in which we have to restrict ourselves is the problem of *recursion*. We will use some "standard" ILP solutions to handle recursion, and describe the current limitations of DITVS w.r.t. recursion.

This chapter is organized as follows: in Section 5.2 we introduce basic notions from Logic Programming. Then, step by step, we instantiate the ITVS framework w.r.t. Logic Programming. First we choose $\mathcal{L}_C$ and $\mathcal{L}_I$ in Section 5.3. Then we propose some alternatives for the function *cover* in Section 5.4. In Section 5.5 we introduce θ-subsumption, which could be used as $\preccurlyeq$. However, since θ-subsumption does not take the background knowledge into account, we choose a generalized form of θ-subsumption (generalized subsumption of [Buntine, 1988]), and show how this can be reduced to θ-subsumption (Section 5.6). Then we show how to search the defined concept representation language. Therefore we first briefly discuss refinement operators in general in Section 5.7. Search is determined by bias, and one of the most important forms of bias is language bias. Several kinds of language bias and the closely related notion of starting clauses are discussed in Section 5.8 and Section 5.9. This leads us to an implementation of the four refinement operators of the ITVS framework (*mub*, *mlb*, *msg* and *mgs*) in Section 5.10. In Section 5.11 we describe how the instance generation approach of Chapter 3 can be accomplished in ILP. Finally we demonstrate ITVS on an example when we put it all together in Section 5.12, and conclude in Section 5.13. At the end of the chapter we give an overview of the definitions and constraints of (D)ITVS, together with their inductive logic counterpart in this chapter.

5.2 The Logic of Inductive Logic Programming

Before we can specify the predicate learning problem, and instantiate $\mathcal{L}_C$, $\mathcal{L}_I$ and *cover*, we have to build up the logical framework used in ILP.

We first introduce some notation and terminology of Logic Programming.

Definition 5.1 (Logic Programming Terminology)

- A *variable* is determined by its name. A variable name is a string of characters, starting with an upper-case letter.
- A *functor* is determined by its name and its arity. A functor name is a string of letters, digits and "_", starting with a lowercase letter. The arity of a functor is a non-negative integer. A functor with name f and arity n is denoted by f/n. A functor with arity 0 is called a *constant*.
- A *predicate* is determined by its name and its arity. A predicate name is a string of letters, digits and "_", starting with a lowercase letter. The arity of a predicate is a non-negative integer. A predicate with name p and arity n is denoted by p/n.
- A *term* is a variable, or is of the form $f(\ t_1\ ,\ \ldots\ ,\ t_n\ )$, where f/n is a functor and all t_k, $1 \leq k \leq n$, are terms. $t_1\ ,\ \ldots\ ,\ t_n$ are called the arguments of the term. A term $c(\ \ )$ for the functor $c/0$ is denoted c.
- An *atom* is of the form $p(\ t_1\ ,\ \ldots\ ,\ t_n\ )$, where p/n is a predicate and all t_k, $1 \leq k \leq n$, are terms. $t_1\ ,\ \ldots\ ,\ t_n$ are called the arguments of the atom. An atom $q(\ \ )$ for the functor $q/0$ is denoted q.
- A *literal* is an atom l or the negation $\overline{l}$ of an atom. An atom is also called a *positive* literal; the negation of an atom is also called a *negative* literal.
- A *clause* is a logical disjunction of a finite number of literals. All variables in a clause are implicitly universally quantified. The clause

$$h_1 \vee \cdots \vee h_m \vee \overline{b_1} \vee \cdots \vee \overline{b_n}$$

(with $m \geq 0, n \geq 0$) is usually written as

$$h_1 \vee \cdots \vee h_m \leftarrow b_1 \wedge \cdots \wedge b_n.$$

Sometimes a clause is considered as the *set* of its literals. The set of positive literals in the clause is called the *head* of the clause; the set of negative literals is the *body* of the clause. The empty clause is denoted by $\square$. If a certain predicate p/n occurs both in the head and in the body of a clause, the clause is called *recursive*. Variables appearing in the body of the clause but not in the head are called *existential* variables.

- A *Horn clause* is a clause with *at most* one positive literal ($m \leq 1$). A definite clause is a clause with *exactly* one positive literal ($m = 1$).
- A *definite program* is a conjunction of definite clauses. Since we will only use definite programs, we will often just talk about "programs". Sometimes a program is considered as the *set* of its clauses.
- A *substitution* $\{ X_1/t_1 , \ldots , X_n/t_n \}$, with $X_1 , \ldots , X_n$ ($n \geq 0$) distinct variables and $t_1 , \ldots , t_n$ terms, maps a term, resp. literal, clause or program to another term, resp. literal, clause or program, obtained by simultaneously replacing all occurrences of X_i by t_i, and this for all i, $1 \leq i \leq n$.
- A *variable renaming* is a substitution $\{ X_1/t_1 , \ldots , X_n/t_n \}$ where for all i, $1 \leq i \leq n$, t_i is a variable, and for all i, j, $1 \leq i, j \leq n$, $i \neq j$ implies $t_i \neq t_j$.
- A substitution θ is a *unifier* of two atoms a_1 and a_2, if $a_1\theta = a_2\theta$. In that case a_1 *matches* a_2, and vice versa. A substitution θ is a *most general unifier* of two atoms a_1 and a_2, iff it is a unifier of a_1 and a_2, and for every unifier τ there exists a substitution σ such that $\theta\sigma = \tau$. A most general unifier is unique up to a variable renaming.
- A term, resp. atom, literal, clause or program x is an *instance* of another term, resp. atom, literal, clause or program x', iff there exists a substitution θ such that $x = x'\theta$.
- A term, resp. atom, literal, clause or program, is *ground* iff it does not contain variables. A substitution $\{ X_1/t_1 , \ldots , X_n/t_n \}$ is ground, iff t_i is ground for all i, $1 \leq i \leq n$.
- A clause is *range restricted* iff all variables in the head of the clause also occur in the body of the clause.

Given a set of variables, a set of functors and a set of predicates, the following definitions will be useful in describing the sets of clauses we will use, and in particular $\mathcal{L}_C$ and $\mathcal{L}_I$. In the rest of the thesis we will always assume the set of predicates and the set of functors are finite.

Definition 5.2 (Terms, atoms and clauses)

- Given a finite set of functors $\mathcal{F}$ and a set of variables $\mathcal{V}$. The set $\mathcal{T}_{\mathcal{F},\mathcal{V}}$ of all finite terms that can be constructed with $\mathcal{V}$ and $\mathcal{F}$ is the set containing all variables of $\mathcal{V}$ and all terms $f(t_1 , \ldots , t_n)$ where $f/n \in \mathcal{F}$, and t_i is in $\mathcal{T}_{\mathcal{F},\mathcal{V}}$ for all i, $1 \leq i \leq n$.

- Given a finite set of predicates $\mathcal{P}$, a finite set of functors $\mathcal{F}$, and a set of variables $\mathcal{V}$. The set $\mathcal{A}_{\mathcal{P},\mathcal{F},\mathcal{V}}$ of all atoms that can be constructed with $\mathcal{P}$, $\mathcal{F}$ and $\mathcal{V}$ is the set of all atoms $p(\ t_1\ , \ldots , t_n\ )$ where $p/n \in \mathcal{P}$, and t_i is in $\mathcal{T}_{\mathcal{F},\mathcal{V}}$ for all i, $1 \leq i \leq n$.
- Given a finite set of predicates $\mathcal{P}$, a finite set of functors $\mathcal{F}$ and a set of variables $\mathcal{V}$, the set of definite clauses that can be constructed with $\mathcal{P}$, $\mathcal{F}$ and $\mathcal{V}$ is

$$\mathcal{CL}_{\mathcal{P},\mathcal{F},\mathcal{V}} = \{\ l_0 \leftarrow l_1\ , \ldots , l_m \mid m \geq 0 \text{ and } \forall j,\ 0 \leq j \leq m:\ l_j \in \mathcal{A}_{\mathcal{P},\mathcal{F},\mathcal{V}}\ \}.$$

So far we only described the syntax of logic programs. We will also need to describe their semantics.

Definition 5.3 (Interpretations and models) Let $\mathcal{P}$ be a set of predicates, $\mathcal{F}$ a set of functors, and $\mathcal{V}$ a set of variables.

- An *interpretation* for $\mathcal{CL}_{\mathcal{P},\mathcal{F},\mathcal{V}}$ maps each constant to an element of a certain domain D in which the program is going to be interpreted; it maps each functor $f/n \in \mathcal{F}$ (with $n \neq 0$) to a mapping from D^n to D; and it maps each predicate $p/n \in \mathcal{P}$ to a mapping from D^n to $\{\ true\ ,\ false\ \}$. Intuitively, an interpretation attaches a meaning to all constants, functors and predicates, and determines the range of the variables in the clauses. It describes how these symbols are to be *interpreted*, i.e., mapped to the corresponding item in the domain.
- To each ground clause an interpretation attaches a *truth value*, by interpreting each constant, function and predicate in D, and evaluating the resulting logical formula. If the resulting value is *true*, the clause is called true w.r.t. the interpretation. A non-ground clause is true w.r.t. the interpretation, if each possible ground instance of the clause is true w.r.t. the interpretation. A set of clauses is true w.r.t. the interpretation, if each clause is true w.r.t. the interpretation.
- Similarly, an interpretation attaches a truthvalue to a conjuction of literals. When a conjunction c is checked for its truthvalue w.r.t. an interpretation we call c a query. If a query contains variables, it is an existential query.
- A program P is usually intended to represent a particular set of relations about objects in a certain domain. The interpretation that interprets P according to this intention, is called *the intended interpretation* of P.
- An interpretation is a *Herbrand interpretation* if the domain of discourse D is the *Herbrand base*, i.e., the set of all ground elements of $\mathcal{A}_{\mathcal{P},\mathcal{F},\mathcal{V}}$.
- An interpretation is called a *model* for a set of clauses, iff each clause is true w.r.t. the interpretation.
- A set of clauses P *logically entails* a set of clauses P' (denoted $P \models P'$) iff every model of P is also a model of P'.
- Finally, a set of clauses P is *tautologically true*, if P is true in all interpretations of $\mathcal{CL}_{\mathcal{P},\mathcal{F},\mathcal{V}}$.

For more details on Logic Programming and its current state of the art we refer to [Genesereth and Nilsson, 1987], [Lloyd, 1987] and [Bruynooghe *et al.*, 1994].

5.3 Representing concepts and instances in ILP

How can definite clauses be used to represent concepts? As defined in Chapter 2 a concept is a set of instances. Instances consist of (abstract or concrete) objects in the real world. An instance in Example 2.1 is, e.g., "successfully dragging document d_1 from folder f_1 to folder f_2 at time t_1". The objects involved are d_1, f_1, f_2 and t_1. An abstract object would be, e.g., a particular "successful drag operation", named e_1. Instances that belong to a certain concept p_n are characterized by a particular relation between the n involved objects. In the example, the document d_1 could have been part of folder f_1, folder f_1 could have been open, etc. The relation that characterizes p_n can be represented by a predicate p/n, with the objects as arguments. The problem of learning a concept representation for p_n is to find what relationship in general holds between several objects $t_1, \ldots, t_n$ in order for $(t_1, \ldots, t_n)$ to belong to p_n. The objects are then represented by ground terms. Instances are n-tuples of ground terms. That $(t_1, \ldots, t_n)$ belongs to p_n, is denoted by $p(t_1, \ldots, t_n)$. In this way, the problem of concept learning in a logical context, is to find one or more definite clauses for p/n, i.e., to find a *definition* for p/n. Should $(t_1, \ldots, t_n)$ be an instance of the concept, $p(t_1, \ldots, t_n)$ has to match the head of at least one clause; furthermore, the body of that clause after matching then expresses the conditions for $(t_1, \ldots, t_n)$ to be an instance of the concept. These clauses together represent the concept, and are called *the (concept) definition* of p_n. The problem of concept learning will then be called *predicate learning* [De Raedt, 1992]. One clause expresses a *conjunctive* condition on $(X_1, \ldots, X_n)$; the problem of finding one clause is called the *conjunctive predicate learning* problem (which we will solve by means of ITVS). Disjunctive conditions can be expressed by conjunctions of clauses[1] with the same predicate in the head; the problem of finding more than one definite clause is the *disjunctive predicate problem* (which we will solve by means of DITVS).

This suggests Definition 5.4 as a definition for $\mathcal{L}_C$ and $\mathcal{L}_I$. This definition is commonly used in ILP.

Definition 5.4 *($\mathcal{L}_I$ and $\mathcal{L}_C$)* Given a set of predicates $\mathcal{P}$, a set of functors $\mathcal{F}$, a set of variables $\mathcal{V}$ and a predicate $p/n \in \mathcal{P}$ for which a definition is to be induced:

- $\mathcal{L}_I = \{ (t_1, \ldots, t_n) \mid \forall i,\ 1 \le i \le n:\ t_i \in \mathcal{T}_{\mathcal{F},\mathcal{V}} \text{ and } t_i \text{ is ground} \}$.
- $\mathcal{L}_C = \{ p(t_1, \ldots, t_n) \leftarrow l_1, \ldots, l_m \mid$
  $\forall i,\ 1 \le i \le n:\ t_i \in \mathcal{T}_{\mathcal{F},\mathcal{V}} \text{ and } \forall j,\ 1 \le j \le m:\ l_j \in \mathcal{A}_{\mathcal{P},\mathcal{F},\mathcal{V}} \}$.

$\mathcal{DL}_C$ can then be defined as in Chapter 4. Often instances will also be denoted by $p(t_1, \ldots, t_n)$ instead of $(t_1, \ldots, t_n)$.

Using the intended interpretation we can describe R_I and R_C of Chapter 2.

Definition 5.5 *(R_I and R_C)*

[1]That a disjunction of concept representations is actually a logical conjunction of clauses in the framework of ILP might cause some confusion. Learning sets (i.e., conjunctions) of clauses unfortunately corresponds to "disjunctive concept learning" in the traditional sense. The set $\{ h \leftarrow b_1, h \leftarrow b_2 \}$ can however be interpreted as h if b_1 or b_2, which shows the connection to disjunctive concept learning. To avoid confusion, "disjunction" in this chapter means disjunction in the sense of Chapter 4. If we mean "logical disjunction" we will explicitly say so.

- R_I maps an instance (t_1 , ... , t_n) to the n-tuple consisting of the images of t_j under the *intended interpretation*.
- R_C maps a definition $p(t_1 , \ldots , t_n) \leftarrow b$ to the set of n-tuples ($t'_1 , \ldots , t'_n$) for which there exists a ground substitution θ, such that for all j, $1 \leq j \leq n$, $t_j\theta$ is interpreted as t'_j, and $b\theta$ is true in the intended interpretation.

As defined in Chapter 2, positive and negative lowerbounds are elements of $\mathcal{L}_I$, which are assumed to be consistent with what we called the *target concept representation*. In this context the intended interpretation is meant to be a model of the target concept representation. Therefore the positive lowerbounds that are in $\mathcal{L}_I$ (i.e., the positive examples; see Chapter 2) are assumed to be true w.r.t. the intended interpretation; negative lowerbounds that are in $\mathcal{L}_I$ (i.e., negative examples) are assumed to be false w.r.t. the intended interpretation. After having defined $\preccurlyeq$ further on, we will be able to extend this towards positive and negative upper- and lowerbounds being consistent with the intended interpretation.

Example 5.6 Given

$$\mathcal{P} = \{ \ holds\_at/2, dbl\_click\_succeeds/2, dbl\_click\_initiates/2, dbl\_click\_terminates/2, isa\_folder/1, isa\_document/1 \} ,$$

$$\mathcal{F} = \{ \ is\_closed/1, is\_in/2, is\_open/1, is\_visible/1 \} ,$$

and

$$\mathcal{V} = \{ \ A, B, \ldots, Z \ \} .$$

Then

(d_2 , t_1).
(d_4 , t_4).

are possible instances of the concept of "successfully double clicking document D at time T". This is denoted by

dbl_click_succeeds(d_2 , t_1).
dbl_click_succeeds(d_4 , t_4).

A possible definition of the concept could be the following two clauses:

```
c1: dbl_click_succeeds( D , T ) ←
        isa_document( D ),
        holds_at( is_closed( D ) , T ), holds_at( is_visible( D ) , T ).
c2: dbl_click_succeeds( D , T ) ←
        isa_document( D ), holds_at( is_closed( D ) , T ),
        holds_at( is_in( D , F ) , T ), isa_folder( F ),
        holds_at( is_open( F ) , T ).
```

◇

The question is now, how to induce consistent concept definitions from instances. A concept representation describes conditions on the objects that are part of the instances. These conditions are relations between the objects in the instances, common to all instances. Consequently, these relations have to be known, at least partially, before a concept definition can be induced. In Example 5.6 clause c_1 could be induced from facts such as

holds_at(*is_closed*(d_2) , t_1).
holds_at(*is_closed*(d_4) , t_4).
holds_at(*is_visible*(d_2) , t_1).
holds_at(*is_visible*(d_4) , t_4).
isa_document(d_2).
isa_document(d_4).

In general, the additional knowledge about the relations between the predicates that can be included in the body of the clauses to be learned, is contained in the background knowledge, denoted by $\mathcal{B}$ (see Chapter 2). In ILP $\mathcal{B}$ is usually a logic program, not necessarily definite though. The set of predicates $\mathcal{P}$ and the set of functors $\mathcal{F}$ determine how the induced clauses will be represented, and therefore which definitions $\mathcal{B}$ must contain. In general there are many alternatives. We will for instance formulate the above examples in the *event calculus* representation. It introduces extra predicates *act*/2 and *time*/2, and new constants e_1 and e_2 representing events. We will continue to use this representation in the rest of this chapter, because it will also be used in Chapter 6[2].

Example 5.7 With

$\mathcal{P}$ = { *holds_at*/2, *succeeds*/1,
initiates/2, *terminates*/2, *act*/2, *time*/2,
isa_folder/1, *isa_document*/1 } ,

$\mathcal{F}$ = { *dbl_click*/1, *is_closed*/1, *is_in*/2, *is_open*/1, *is_visible*/1 } ,

and

$\mathcal{V} = \{ A, B, \ldots, Z \}$,

the above instances can be represented as

succeeds(e_1).
succeeds(e_4).

The above concept definition would then be written as:

c_1': *succeeds*(E) ←
act(E , *dbl_click*(D)), *time*(E , T),
isa_document(D),

[2]Technically we then have a concept "an event E is possible", which consists of several subconcepts such as "double clicking a document is possible", "dragging a document from one folder to another folder is possible", etc. However, we will only work with the latter concepts, and consider the event calculus as a way of representing them.

$holds\_at(is\_closed(D) , T), holds\_at(is\_visible(D) , T).$
c'_2: $succeeds(E) \leftarrow$
$act(E , dbl\_click(D)), time(E , T),$
$isa\_document(D),$
$holds\_at(is\_closed(D) , T), holds\_at(is\_in(D , F) , T),$
$isa\_folder(F), holds\_at(is\_open(F) , T).$

The above background knowledge would be written as

$holds\_at(is\_closed(d_2) , t_1).$
$holds\_at(is\_closed(d_4) , t_4).$
$holds\_at(is\_visible(d_2) , t_1).$
$holds\_at(is\_visible(d_4) , t_4).$
$isa\_document(d_2).$
$isa\_document(d_4).$
$act(e_1 , dbl\_click(d_2)).$
$act(e_4 , dbl\_click(d_4)).$
$time(e_1 , t_1).$
$time(e_4 , t_4).$

◇

Now that we have defined $\mathcal{L}_I$ and $\mathcal{L}_C$, we have to define the *cover* function. From *cover*, we then derive *covers* and $\preccurlyeq$ (see Chapter 2). The relation $\preccurlyeq$ determines how $\mathcal{L}_C$ will be searched: it is the partial order on the search space on which the refinement operators are based. Therefore one of the most important forms of *search bias* in ILP is the choice of the $\preccurlyeq$ relation, and therefore, in our framework, the choice of *cover*.

5.4 The function *cover*

The logical choice for the function *cover* is to take logical implication, i.e., a clause c covers an atom a iff $\mathcal{B} \wedge c \models a$. However, most ILP systems only approximate logical implication by subsumption. The most widely used form of subsumption is *θ-subsumption*. However, θ-subsumption is a relation between two clauses which does not take $\mathcal{B}$ into account. Therefore we will use *generalized subsumption* [Buntine, 1988], which does take $\mathcal{B}$ into account, and we will show its relation to logical implication. Unfortunately, generalized subsumption is only semi-decidable. Therefore we will in a further step *approximate* generalized subsumption by means of θ-subsumption.

Definition 5.8 (*$cover_{Ip}$* **and** *$covers_{Ip}$* **in** $\mathcal{L}_C$) A ground atom a is covered by a clause $c = h \leftarrow b$ in an interpretation Ip iff there is a substitution θ such that $h\theta$ is a, and the query $\exists(b\theta)$ is true in the interpretation Ip.

The $\exists$ quantor quantifies all variables in $b\theta$ existentially.

Notation 5.9 The function *cover* w.r.t. interpretation Ip is denoted $cover_{Ip}$.

For disjunctions of clauses we define $cover_{d,Ip}$ as follows:

Definition 5.10 ($cover_{d,Ip}$ **and** $covers_{d,Ip}$ **in** $\mathcal{DL}_C$) A ground atom a is covered by a set of clauses $d = \{ c_1 , \ldots , c_n \}$ in an interpretation Ip iff there is a j, $1 \leq j \leq n$, with $covers_{Ip}(c_j , a)$.

Independently of what interpretation is actually chosen, this definition corresponds to Definition 4.3, i.e.,

$$cover_{d,Ip}(c_1 \vee \cdots \vee c_n) = cover_{Ip}(c_1) \cup \cdots \cup cover_{Ip}(c_n).$$

Definition 5.8 and Definition 5.10 define *cover* w.r.t. an interpretation Ip. The question is now, which interpretation to choose as Ip. An instance $(t_1 , \ldots , t_n)$ is covered by a concept definition $c = h \leftarrow b$ for the predicate p/n if there exists a subsitution θ such that the atom $p(t_1 , \ldots , t_n)$ is equal to $h\theta$, and $\exists b\theta$ is true in the chosen interpretation. The query $b\theta$ is true in the chosen interpretation if there exists a ground instance of $b\theta$ which is true in the chosen interpretation. On the other hand, Constraint 2.8 (the soundness constraint for *cover*) requires that $(t_1 , \ldots , t_n)$ is covered by c iff $R_I((t_1 , \ldots , t_n)) \in R_C(c)$. By Definition 5.5, the definition of R_I and R_C, this means that the soundness constraint for *cover* is fulfilled if and only if the interpretation chosen to evaluate coverage is the intended interpretation.

In general, the problem with the intended interpretation is that it is unknown. This means that queries cannot necessarily be answered. Yet $\mathcal{L}_C$ is to be structured according to *cover* in order to search $\mathcal{L}_C$ for consistent concept definitions. For non-recursive clauses, the body of c only contains predicates defined in the background knowledge. Because of our assumption that the background knowledge is correct w.r.t. the intended interpretation, all queries can be answered by querying the background knowledge. Note that in multiple predicate learning this approach could therefore lead to incorrect results. For recursive clauses, the body of c also contains literals of the predicate which is to be learned. In this case, the background knowledge cannot be used to solve the query either. The only instance representations that are guaranteed to be covered in the intended interpretation are the ones covered by the positive lowerbounds, since the information elements are supposed to be correct w.r.t. the intended interpretation.

Let I_s^+ therefore denote the set of all positive lowerbounds in the set I of all information elements. In the context of MIS, [Shapiro, 1983] uses three different *cover*- functions. MIS is a theory revision system that uses no background knowledge and accepts only positive and negative instances (i.e., examples) as information elements. In that context the three *cover*- functions are:

- Eager: in this case the interpretation Ip is the intended interpretation: for all existential queries and for all other queries that are not in the minimal Herbrand model of the theory I_s^+, Shapiro proposes to consult an oracle, which answers the queries correctly w.r.t. the intended interpretation.

- Lazy: in this case the interpretation Ip is the minimal Herbrand model of the theory I_s^+.

- Adaptive: in this case the interpretation Ip is the minimal Herbrand model of the theory $I_s^+ \wedge P$, where P is the part of the theory already learned.

In our context of single concept learning, using background knowledge B and also accepting elements of $\mathcal{L}_C$ as positive lowerbounds, this can be formulated as follows:

- Eager: in this case the interpretation Ip is the intended interpretation: for all existential queries and for all other queries that are not in the minimal Herbrand model of the theory $B \wedge I_s^+$, an oracle answers the queries correctly w.r.t. the intended interpretation.

  This is the most powerful of the three possibilities because it answers every query correctly. However, this is also its main disadvantage: it relies heavily on the presence of an oracle. Since we suppose the background is correct w.r.t. the intended interpretation, the oracle only has to be consulted for queries about the predicate that is learned itself. But an answer to a query for this predicate can also be used as a new information element with which the current definition has to be consistent. This in turn can give rise to new queries to the oracle, and so on. In general this can also result in problems of termination for recursive clauses (see further).

- Lazy: in this case the interpretation Ip is the minimal Herbrand model of the theory $B \wedge I_s^+$.

  The advantage of this choice is that it does not require an oracle. For non-recursive clauses this is no problem, since we assume the background knowledge to be correct. However, in order to answer the existential queries about the predicate that is learned correctly, I_s^+ must contain enough positive lowerbounds. In general this will not be the case. In this respect, this approach is very much order dependent: clauses could be rejected for not covering a particular instance, because the positive lowerbounds needed to solve recursive queries are not yet known.

  In this context, the automatic generation of new positive lowerbounds as described in Section 3.9 is advantageous, because the resulting positive lowerbounds are more general than the original ones, and might therefore cover more instances. In this way, the information contained in the positive lowerbounds is more optimally used. As an example, one could think of two instances of the base case of a recursive definition, which together generalize to a more general base case. This base case can then be used to answer queries when learning a recursive clause.

- Adaptive: in this case the interpretation Ip is the minimal Herbrand model of the theory $B \wedge I_s^+ \wedge P$. P contains the already found clauses for the predicate that is currently learned.

  Shapiro calls the adaptive approach a compromise between the Eager and the Lazy approach: it does not require an oracle, and is less order dependent than the Lazy approach. For a disjunction $c_1 \vee c_2$, this choice would determine $cover(c_1)$ on the basis of the chosen c_2, and vice versa. This means that the search-space of c_1 would change with another choice of c_2. In general this is problematic, because the search of c_1 will not be sound. Finally, because P could still be incorrect, some mechanism will be needed to avoid infinite loops.

W.r.t. disjunctions, Constraint 4.8 requires that for all c_1, c_2 and c_3 in $\mathcal{L}_C$,

$$cover(c_3) \subseteq cover(c_1 \vee c_2)$$

implies that $cover(\ c_3\ ) \subseteq cover(\ c_1\ )$ or $cover(\ c_3\ ) \subseteq cover(\ c_2\ )$. This constraint requires that the *cover* of c_1 and c_2 can be determined independently. This is no problem if the eager approach is used: each query is solved using the oracle, independently of any other definitions. In the case of the lazy approach, $cover(\ c_1\ )$ and $cover(\ c_2\ )$ are also determined independently, because they are only based on I_s^+. Only in the case of the adaptive approach, the constraint is not valid, because the *cover* of c_1 is based on c_2 and vice versa. This means that DITVS cannot be applied correctly, because (in the terminology of Chapter 4) $cover_d$ on $\mathcal{DL}_C$ cannot be reduced to *cover* on $\mathcal{L}_C$. Then $cover_d$ has to be defined *globally* on sets of clauses, without reducing this to individual clauses. Given this aim DI or ITVS (see Chapter 3) can still be used.

In the remainder, we will assume that we can evaluate *cover* by means of the intended interpretation. For non-recursive clauses, this approach is equivalent to the lazy and the adaptive approach. For recursive clauses, it is in many situations only a theoretical approach, since it relies on the oracle.

Definition 5.11 (*cover* and *covers*)

- A ground atom a is covered by a clause c iff a is covered by c in the intended interpretation.
- A ground atom a is covered by a set of clauses $\{\ c_1\ ,\ \ldots,\ c_n\ \}$ iff a is covered by $\{\ c_1\ ,\ \ldots,\ c_n\ \}$ in the intended interpretation.

The second part of this definition corresponds to Definition 4.3, i.e.,

$$cover_{d,I_P}(\ c_1 \vee \cdots \vee c_n\ ) = cover_{I_P}(\ c_1\ ) \cup \cdots \cup cover_{I_P}(\ c_n\ ).$$

Definition 5.12 ($\top$ **and** $\bot$) For each predicate p/n, we define $\top$ and $\bot$ as follows:

- $\top = p(\ X_1\ ,\ \ldots\ ,\ X_n\ ) \leftarrow .$
- $\bot = \Box.$

Both are assumed to be contained in the definition of $\mathcal{L}_C$ (Definition 5.4). As such, $\top$ covers every element in $\mathcal{L}_I$, and $\bot$ does not cover any element of $\mathcal{L}_I$.

5.5 θ-subsumption

In this section we will define θ-subsumption, and investigate its properties. θ-subsumption will not directly be used as $\preccurlyeq$, because it does not take the background knowledge into account. Therefore we do also not relate θ-subsumption directly to *cover*. However, θ-subsumption will be used for implementing generalized subsumption in Section 5.6. Therefore it will be discussed first.

Definition 5.13 (**θ-subsumption**) A clause c_1 θ-subsumes a clause c_2 iff there exists a substitution θ such that $c_1\theta \subseteq c_2$.

Notation 5.14 "c_1 θ-subsumes c_2" will be denoted as $c_2 \preccurlyeq_\theta c_1$.

Proposition 5.15 The relation $\preccurlyeq_\theta$ is a reflexive and transitive relation on $\mathcal{CL}_{\mathcal{P},\mathcal{F},\mathcal{V}}$.

Proof This follows from the reflexivity and transitivity of $\subseteq$. □

θ-subsumption is decidable [Robinson, 1965], but in general NP-complete [Garey and Johnson, 1979]. [Gottlob, 1987] proves that $c_2 \preccurlyeq_\theta c_1$ is equivalent to "c_1 implies c_2", if c_1 is not self-resolving and c_2 is not tautological, or if c_2 is not ambivalent. A clause is self-resolving iff it resolves with a copy of itself. A clause is ambivalent iff a predicate symbol appears in a positive literal as well as in a negative literal of the clause. As long as we do not use recursive clauses, this condition is fulfilled.

Note that θ-subsumption is neither symmetric, nor anti-symmetric:

p(X) ← q(X, a).

on the one hand θ-subsumes

p(X) ← q(X, a), q(X, Y).

but, on the other hand, it is also θ-subsumed by the latter clause. Both clauses, however, θ-subsume

p(X) ← q(X, a), q(X, b).

and neither of them is θ-subsumed by the latter clause.

[Plotkin, 1970] defines an equivalence relation on the set of all clauses, and identifies exactly one clause per equivalence class as the representant of that class. On the set of all representants of all equivalence classes, $\preccurlyeq_\theta$ will be anti-symmetric. The framework of ITVS requires $\preccurlyeq$ to be a partial order, and thus anti-symmetric. If we want to use θ-subsumption as a basis for generalized subsumption, we have to use these representants as well. This approach is similar as as in Section 2.4.1.

Definition 5.16 (Equivalence of clauses) The clauses c_1 and c_2 are called equivalent w.r.t. θ-subsumption iff $c_1 \preccurlyeq_\theta c_2$ and $c_2 \preccurlyeq_\theta c_1$.

Notation 5.17 "c_1 is equivalent with c_2 w.r.t. θ-subsumption" is denoted as "$c_1 \equiv_\theta c_2$".

Definition 5.18 (Reduced clause) A literal l in a clause c is redundant iff $c \equiv_\theta c \backslash \{ l \}$. A clause is reduced if it does not contain any redundant literals. A reduction of a clause c_1 is a clause c_2 such that $c_2 \subseteq c_1$, $c_1 \equiv_\theta c_2$ and c_2 is reduced.

[Plotkin, 1970] proves that two reduced clauses that are equivalent, are equal up to a variable renaming. This means that there is exactly one reduced clause per equivalence class w.r.t. $\equiv_\theta$, which can be taken as a representant of that equivalence class. Consequently, in the set of all reduced clauses, $\preccurlyeq_\theta$ is anti-symmetric, and thus a partial order.

[Plotkin, 1970] also gives an algorithm to find a reduction of a given clause. [Gottlob and Fermüller, 1993] presents an algorithm for reducing a clause c with in the worst case $| c | \preccurlyeq_\theta$-tests.

[Plotkin, 1970] defines the least general generalization *lgg* of two clauses as follows:

Definition 5.19 (Least general generalization) For all $c_1, c_2, c \in \mathcal{L}_C$: c is a least general generalization (*lgg*) of c_1 and c_2 iff

- $c_1 \preccurlyeq_\theta c$ and $c_2 \preccurlyeq_\theta c$, and
- for all $c' \in \mathcal{L}_C$ such that $c_1 \preccurlyeq_\theta c'$ and $c_2 \preccurlyeq_\theta c'$, we have $c \preccurlyeq_\theta c'$.

By definition a least general generalization is a minimal upperbound (*mub*) w.r.t. θ-subsumption.

Proposition 5.20 For all $c_1, c_2 \in \mathcal{L}_C$: if l_1 and l_2 are least general generalizations of c_1 and c_2, then $l_1 \equiv_\theta l_2$.

Proof This follows from the second condition in the definition of *lgg*. □

[Plotkin, 1970] gives an algorithm for computing the *lgg* of two clauses. We will not present the algorithm here, although we use it to actually implement *mub* in Section 5.10.

We introduce a notion dual to *lgg*: the least specific specialization. The least specific specialization of two definite clauses c_1 and c_2 is the maximal lowerbound of c_1 and c_2 w.r.t. θ-subsumption.

Definition 5.21 (Least specific specialization) For all c_1, c_2 and $c \in \mathcal{L}_C$: c is a least specific specialization (*lss*) of c_1 and c_2 iff

- $c \preccurlyeq_\theta c_1$ and $c \preccurlyeq_\theta c_2$, and
- for all $c' \in \mathcal{L}_C$ such that $c' \preccurlyeq_\theta c_1$ and $c' \preccurlyeq_\theta c_2$, we have $c' \preccurlyeq_\theta c$.

By definition a least specific specialization is a maximal lowerbound (*mlb*) w.r.t. θ-subsumption.

We also have a dual uniqueness result.

Proposition 5.22 For all $c_1, c_2 \in \mathcal{L}_C$: if l_1 and l_2 are least specific specializations of c_1 and c_2, then $l_1 \equiv_\theta l_2$.

Proof This follows from the second condition in the definition of *lss*. □

Proposition 5.23 Given $c_1 = h_1 \leftarrow b_1$ and $c_2 = h_2 \leftarrow b_2$ in $\mathcal{L}_C$, such that c_1 and c_2 have no variables in common. If there exists a most general unifier θ of h_1 and h_2, then
$c = (\ h_1\theta \leftarrow b_1\theta, b_2\theta\ )$ is a least specific specialization of c_1 and c_2, otherwise
$lss(\ c_1\ ,\ c_2\ ) = \bot$.

Proof First suppose there exists no general unifier of c_1 and c_2. Then no clause c exists in $\mathcal{L}_C$ such that $c \preccurlyeq_\theta c_1$ and $c \preccurlyeq_\theta c_2$, except $c = \bot$ (which is defined as the empty clause; see Definition 5.12).

Now suppose there exists a general unifier (and therefore a most general unifier) of c_1 and c_2. Because a most general unifier is unique up to a variable renaming, we can choose θ such that $h_1\theta$ and $h_2\theta$ have no variables in common with either c_1 or c_2. Then c is more specific than c_1 and c_2, because $c_1\theta \subseteq c$ and $c_2\theta \subseteq c$.

For the second part of the definition of lss, suppose there is a $c' = h' \leftarrow b'$ such that $c' \preccurlyeq_\theta c_1$ and $c' \preccurlyeq_\theta c_2$, and the variables of c' are again distinct from those in c_1, c_2 and c. Then there exists a substitution τ_1 and τ_2 such that $h_1\tau_1 = h'$, $h_2\tau_2 = h'$, $b_1\tau_1 \subseteq b'$ and $b_2\tau_2 \subseteq b'$. Furthermore, because all clauses have distinct variables, we have:

- $h_1\tau_1 = h_1\tau_1\tau_2 = h'$,
- $b_1\tau_1 = b_1\tau_1\tau_2 \subseteq b'$,
- $h_2\tau_2 = h_2\tau_2\tau_1 = h_2\tau_1\tau_2 = h'$, and
- $b_2\tau_2 = b_2\tau_2\tau_1 = b_2\tau_1\tau_2 \subseteq b'$.

Consequently, $h' = h_1\tau_1 = h_1\tau_1\tau_2 = h_2\tau_2 = h_2\tau_1\tau_2$, i.e., $\tau_1\tau_2$ is a unifier of h_1 and h_2. Since θ is a most general unifier of h_1 and h_2, there exist a substitution σ such that $\theta\sigma = \tau_1\tau_2$. Consequently $b_1\theta\sigma \subseteq b'$ and $b_2\theta\sigma \subseteq b'$. Thus $c' \preccurlyeq_\theta c$, which concludes the proof. □

[Plotkin, 1971a] describes a similar operation on general (i.e., not necessarily definite) clauses, called the *most general instance*. The most general instance of two clauses having distinct variables is the union of the two clauses. Applied on definite clauses, this gives only the same result as our least specific specialization, if the heads of the two clauses have different variables and are equal up to a variable renaming.

5.6 Generalized subsumption

In Section 2.4 we mentioned the predicates used in the bodies of concept representations are defined in the background knowledge $\mathcal{B}$. $\mathcal{B}$ might also express relationships between these predicates used in the bodies. Therefore $\mathcal{B}$ has to be taken into account when defining $\preccurlyeq$. We will first give an example to show why $\preccurlyeq_\theta$ is in general not suitable.

Example 5.24 Consider clauses c_5, c_6 and the program $P = \{ c_7 \}$.

c_5 : *succeeds(E)* ←
    act(E , dbl_click(D)), time(E , T),
    isa_document(D),
    holds_at(is_in_open_folder(D) , T),
    holds_at(is_closed(D) , T).
c_6 : *succeeds(E)* ←
    act(E , dbl_click(D)), time(E , T),
    isa_document(D), isa_folder(F),
    holds_at(is_open(F) , T),
    holds_at(is_in(D , F) , T),
    holds_at(is_closed(D) , T).
c_7 : *holds_at(is_in_open_folder(D) , T)* ←
    isa_document(D), isa_folder(F),
    holds_at(is_open(F) , T),
    holds_at(is_in(D , F) , T).

The clause c_7 of P expresses a relation between the properties *is_in_open_folder*/2, *is_open*/1 and *is_in*/2. Taking P into account is important in deciding whether or not c_5 is a generalization of c_6. The clause c_5 does not θ-subsume c_6. However, c_5 is intended to be a generalization of c_6, *w.r.t.* P, because the underlined literal in c_5 implies, according to c_7, the underlined literals in c_6. ◇

This example illustrates that $\preccurlyeq_\theta$ is not suitable as $\preccurlyeq$ in the presence of background knowledge. Generalized subsumption [Buntine, 1988] on the other hand allows to make use of background knowledge.

Definition 5.25 (Generalized Subsumption) For all $c_1, c_2 \in \mathcal{L}_C$: c_1 subsumes c_2 w.r.t. program P iff for any Herbrand interpretation I_P of $\mathcal{L}_C$ such that P is true in I_P, we have $cover_{I_P}(\ c_2\ ) \subseteq cover_{I_P}(\ c_1\ )$.

Notation 5.26 "c_1 subsumes c_2 w.r.t. P" is denoted as "$c_2 \preccurlyeq_P c_1$".

Definition 5.25, using $cover_P$ with $P = \mathcal{B}$, is an instantiation of the general Definition 3.1 of $\preccurlyeq$ with the particular choice we made for *cover* in this ILP context (see Definition 5.11). Consequently, $\preccurlyeq_\mathcal{B}$ inherits all properties from $\preccurlyeq$.

[Buntine, 1988] also extends the definition of $\preccurlyeq_P$ to sets of clauses.

Definition 5.27 (Generalized Subsumption for sets of clauses) The disjunction $d_1 \in \mathcal{DL}_C$ subsumes the disjunction $d_2 \in \mathcal{DL}_C$ w.r.t. program P iff for any Herbrand interpretation I_P such that P is true in I_P, we have $cover_{I_P}(\ d_2\ ) \subseteq cover_{I_P}(\ d_1\ )$.

Generalized subsumption is a special case of relative subsumption [Plotkin, 1971b] restricted to definite clauses. Generalized subsumption is a stronger generality relation than θ-subsumption in that $c_1 \preccurlyeq_\theta c_2$ implies $c_1 \preccurlyeq_P c_2$. Actually $\preccurlyeq_\theta$ is just $\preccurlyeq_P$, with $P = \emptyset$.

As for θ-subsumption, an equivalence relation can be defined for generalized subsumption.

Definition 5.28 ($\equiv_P$) For all $c_1, c_2 \in \mathcal{L}_C$:

$$c_1 \equiv_P c_2 \text{ iff } c_1 \preccurlyeq_P c_2 \text{ and } c_2 \preccurlyeq_P c_1.$$

We can also define equivalence classes for generalized subsumption w.r.t. P. Note that each class for θ-subsumption will be a subset of a class for generalized subsumption, and that it is possible that several classes for θ-subsumption are contained in one class for generalized subsumption.

Example 5.29 Consider the clauses c_8 and c_9, and the clauses c_5 and c_7 from Example 5.24.

c_8 : *succeeds*(E) $\leftarrow$
act(E , *dbl_click*(D)), *time*(E , T),
isa_document(D), *isa_folder*(F),
holds_at(*is_open*(F) , T),
holds_at(*is_in*(D , F) , T),
holds_at(*is_in_open_folder*(D) , T),

$\quad\quad holds\_at(\ is\_closed(\ D\ )\ ,\ T\ ).$
$c_9 : succeeds(\ E\ ) \leftarrow$
$\quad act(\ E\ ,\ dbl\_click(\ D\ )\ ),\ time(\ E\ ,\ T\ ),$
$\quad isa\_document(\ D\ ),\ isa\_folder(\ F\ ),\ \underline{isa\_folder(\ F'\ )},$
$\quad holds\_at(\ is\_open(\ F\ )\ ,\ T\ ),$
$\quad \underline{holds\_at(\ is\_open(\ F'\ )\ ,\ T\ )},$
$\quad holds\_at(\ is\_in(\ D\ ,\ F\ )\ ,\ T\ ),$
$\quad holds\_at(\ is\_in\_open\_folder(\ D\ )\ ,\ T\ ),$
$\quad holds\_at(\ is\_closed(\ D\ )\ ,\ T\ ).$

Clause c_8 is equivalent to c_5 under generalized subsumption w.r.t. $P = \{\ c_7\ \}$. However, although c_5 θ-subsumes c_8, c_8 does not θ-subsume c_5. The clause c_8 does θ-subsume c_9, because it is a subset of c_9. But c_9 also θ-subsumes c_8, with $\theta = \{\ F'/F\ \}$. Consequently, $c_5 \equiv_P c_8 \equiv_P c_9$, $c_8 \equiv_\theta c_9$, and $c_5 \not\preccurlyeq_\theta c_8$. ◇

Theorem 5.30, Theorem 5.31 and Theorem 5.33 are taken from [Buntine, 1988].

Theorem 5.30 Given $d = c_1 \vee \cdots \vee c_m \in \mathcal{DL}_C$, $d' = c'_1 \vee \cdots \vee c'_n \in \mathcal{DL}_C$ and P a program. Then $d \preccurlyeq_P d'$ iff for all k, $1 \leq k \leq m$, there exists l, $1 \leq l \leq n$, such that $c_k \preccurlyeq_P c'_l$. □

This means that $\preccurlyeq_{\mathcal{B}}$ as defined in Definition 5.27 for disjunctions of clauses fulfills Constraint 4.8 (the Disjunctions Constraint; see Corollary 4.12).

Theorem 5.31 Given a program P, two definite clauses $c_1 = h_1 \leftarrow b_1$ and $c_2 = h_2 \leftarrow b_2$, and a substitution σ_g replacing all variables in c_2 by distinct skolem constants. The clause c_1 subsumes c_2 w.r.t. P iff there exists a substitution θ such that $h_1\theta = h_2$ and

$$P \wedge b_2\sigma_g \models \exists(\ b_1\theta\sigma_g\ ). \tag{5.1}$$

□

Example 5.32 The clause c_5 subsumes c_6 w.r.t. $P = \{\ c_7\ \}$, because

$c_7 \wedge$
$\quad act(\ sk1\ ,\ dbl\_click(\ sk2\ )\ ),\ time(\ sk1\ ,\ sk3\ ),$
$\quad isa\_document(\ sk2\ ),\ isa\_folder(\ sk4\ ),$
$\quad holds\_at(\ is\_open(\ sk4\ )\ ,\ sk3\ ),$
$\quad holds\_at(\ is\_in(\ sk2\ ,\ sk4\ )\ ,\ sk3\ ),$
$\quad holds\_at(\ is\_closed(\ sk2\ )\ ,\ sk3\ )$
$\quad \models$
$\quad act(\ sk1\ ,\ dbl\_click(\ D\ )\ ),\ time(\ sk1\ ,\ T\ ),$
$\quad isa\_document(\ D\ ),$
$\quad holds\_at(\ is\_in\_open\_folder(\ D\ )\ ,\ T\ ),$
$\quad holds\_at(\ is\_closed(\ D\ )\ ,\ T\ )$

with substitution $\{D/sk2, T/sk3\}$. ◇

Theorem 5.31 can be used to implement generalized subsumption using a theorem prover, e.g., PROLOG. As a consequence, generalized subsumption is in general semi-decidable, i.e., termination of a generalized subsumption test $c_1 \preccurlyeq_P c_2$ can only be guaranteed when $c_1 \preccurlyeq_P c_2$. However, when P does not contain recursive clauses, or when P does not contain any proper functor symbols, it is decidable [Buntine, 1988].

Theorem 5.33 Given $c_1, c_2 \in \mathcal{L}_C$, such that c_2 is not tautologically true, and a program P. Then $P \wedge c_1 \models c_2$ iff there exists $c_1' \in P \wedge c_1$ such that $c_2 \preccurlyeq_P c_1'$. □

This theorem describes the relation of generalized subsumption to implication w.r.t. P. On the one hand, it shows that if $c_2 \preccurlyeq_P c_1$, then c_1 implies c_2 w.r.t. P, i.e., implication is stronger than generalized subsumption. On the other hand, applied on definite clauses and single predicate learning, and if $P = \mathcal{B}$, $P \wedge c_1 \models c_2$ implies that $c_2 \preccurlyeq_P c_1$, because c_1' must be equal to c_1. If P would be allowed to contain clauses for the predicate of c_1 in order to cope with recursion (as in the adaptive strategy of Section 5.4), the fact that c_1 logically implies c_2 w.r.t. P does not necessarily mean that c_1 generally subsumes c_2 w.r.t. P.

Using implication as $\preccurlyeq$ has been studied in [Muggleton, 1994] [Idestam-Almquist, 1993], and [Lapointe and Matwin, 1992]. For several reasons (efficiency and decidability being the most important ones) most ILP systems tend not to choose implication as a basis for *cover* and $\preccurlyeq$, but rather approximate them by subsumption. For an overview of other existing choices for $\preccurlyeq$ we refer to [Muggleton and De Raedt, 1994].

Instead of using Theorem 5.31, generalized subsumption can also be implemented by a combination of θ-subsumption with saturation [Rouveirol, 1994].

Definition 5.34 (Saturation) Given a program P, a definite clause $c = h \leftarrow b$, and a substitution σ_g replacing all variables in c by distinct skolem constants.

- Elementary saturation of c w.r.t. P returns a clause $(\ h\sigma_g \leftarrow b\sigma_g, h'\theta\ )\sigma_g^{-1}$, where $d = h' \leftarrow b'$ is a clause from P and θ a substitution such that $b'\theta \subseteq b\sigma_g$.
- Saturation of c w.r.t. P returns the transitive closure of elementary saturation of c w.r.t. P.

Notation 5.35 The saturation of a clause c is denoted as $Sat(\ c\ )$.

The intuitive idea behind saturation is that, in order to compare two clauses with $\preccurlyeq_\theta$ while taking the background knowledge into account, all relations between the predicates in the body ought to be made explicit, because $\preccurlyeq_\theta$ compares the information basically on a set inclusion basis. Note that for θ-subsumption $Sat(\ c\ )$ is more specific than c, because its body is a superset of the body of c. As with the theorem proving step of Theorem 5.31, computing the transitive closure of this resolution step might not terminate when $\mathcal{B}$ contains recursive clauses (see further).

Example 5.36 $Sat(\ c_6\ )$ w.r.t. $P = \{\ c_7\ \}$ (see Example 5.24) is:

$$\begin{array}{l} Sat(\ c_6\ ) : succeeds(\ E\ ) \leftarrow \\ \qquad act(\ E\ ,\ dbl\_click(\ D\ )\ ),\ time(\ E\ ,\ T\ ), \\ \qquad isa\_document(\ D\ ),\ isa\_folder(\ F\ ), \\ \qquad holds\_at(\ is\_open(\ F\ )\ ,\ T\ ), \\ \qquad holds\_at(\ is\_in(\ D\ ,\ F\ )\ ,\ T\ ), \\ \qquad holds\_at(\ is\_closed(\ D\ )\ ,\ T\ ), \\ \qquad \underline{holds\_at(\ is\_in\_open\_folder(\ D\ )\ ,\ T\ )}. \end{array}$$

Because a skolemized version of the literals *isa_document*(D), *isa_folder*(F), *holds_at*(*is_open*(F) , T) and *holds_at*(*is_in*(D , F) , T) in the body of c_6 resolve with the body of c_7, the literal *holds_at*(*is_in_open_folder*(D) , T) was added to saturate c_6. $\diamond$

Theorem 5.37 from [Jung, 1993] proves that the idea of saturation is equivalent to generalized subsumption.

Theorem 5.37 Given a range-restricted program P, and two definite clauses c_1 and c_2. Then $c_1 \preccurlyeq_P c_2$ iff $Sat(\ c_1\ ) \preccurlyeq_\theta c_2$. $\square$

This theorem will be useful in the implementation of the refinement operators *mub*, *mlb*, *msg* and *mgs* (see Section 5.10). We can already prove the following lemma.

Lemma 5.38 For all $c \in \mathcal{L}_C$: $c \equiv_P Sat(\ c\ )$.

Proof On the one hand $Sat(\ c\ ) \preccurlyeq_\theta c$, and thus $Sat(\ c\ ) \preccurlyeq_P c$. On the other hand $Sat(\ c\ ) \preccurlyeq_\theta Sat(\ c\ )$, so $c \preccurlyeq_P Sat(\ c\ )$ follows from Theorem 5.37. Consequently $c \equiv_P Sat(\ c\ )$. $\square$

Using only range-restricted clauses is no major restriction. W.r.t. the intended interpretation clauses that are not range-restricted are often incorrect w.r.t. the intended interpretation. For instance, the clause

initially(*is_open*(F)).

(which is not range restricted) must be interpreted as *everything* in the domain being initially open, while its intention was actually to express that *all folders* are initially open, as the range restricted clause

initially(*is_open*(X)) $\leftarrow$ *isa_folder*(X).

expresses. This justifies intuitively the restriction to range-restricted clauses in the following sections[3].

Constraint 5.39 (Range-restrictedness) We assume all clauses in $\mathcal{B}$ to be range-restricted.

If we want to restrict ourselves to learning range-restricted clauses only, $\top$ has to be altered accordingly:

$$\top = p(\ X_1\ ,\ \ldots\ ,\ X_n\ ) \leftarrow domain_1(\ X_1\ )\ ,\ \ldots\ ,\ domain_n(\ X_n\ ).$$

The predicate $domain_j/1$ determines the domain D_j of the j-th argument of p/n, e.g., *isa_document*/1. These predicates are used to make every clause in $\mathcal{L}_C$ range-restricted. As such, $\top$ covers every element of $D_1 \times D_2 \times \cdots \times D_n$. The bottom element $\perp$ does not change.

As mentioned before, termination in proving Equation 5.1 of Theorem 5.31 cannot be guaranteed if P contains recursion and functors. In order to ensure termination, we have to

[3][De Raedt, 1992] adopts range-restrictedness as one of the *Practical Language Assumptions*.

set a depth-bound on the chosen proof-procedure in some way. One could use a standard PROLOG mechanism augmented with an upperbound on the depth of inference. To make the computation of the transitive closure of saturation on a clause c terminate, one can put an upperbound on the number of saturation steps. In both cases checking whether $c_1 \preccurlyeq_P c_2$ then results in checking whether (with the notation of Theorem 5.31) $\exists(b_1\theta\sigma_g)$ can be proven from $P \wedge b_2\sigma_g$ with a derivation of depth n. Note that as a consequence the saturated clause might not contain the right literals to prove θ-subsumption. Moreover, bounding the number of saturation steps makes the saturation of c not unique, because the order of application of clauses of $\mathcal{B}$ might result in different clauses after n steps. This means that because of the semi-decidable character of $\preccurlyeq_P$ in general, we cannot guarantee completeness. Therefore, in general, θ-subsumption can only be an approximation of generalized subsumption.

In general, we believe that the termination problems of recursion should be tackled by using extra language bias on $\mathcal{L}_C$ and $\mathcal{B}$. Within this language bias only clauses should be allowed for which particular queries are guaranteed to terminate. One such framework is that of acyclic programs [Apt and Bezem, 1991]. Particular queries (consisting of *bounded goals*) are guaranteed to terminate, given a certain *level mapping*. So, after having chosen a level mapping, allowing only to induce acyclic programs would yield completeness for acyclic programs w.r.t. that particular level mapping. Related (but informal) approaches are described by [Shapiro, 1983] and [Bergadano, 1993], basically by requiring that there is a strict and well-founded ordering on the set of possible queries. [Cameron-Jones and Quinlan, 1993] automatically induces an ordering on recursive queries in the absence of functors. However, in the absence of functors termination problems can also be avoided by applying the *OLDT*[4] proof method [Tamaki and Sato, 1986]. Another (already more theoretically founded) approach that fits in this framework is the use of h-conform theories as described in [De Raedt, 1992]. To our knowledge, little research has been done on the nature of the needed orderings. It is clear that this topic is highly related to work on termination of logic programs [De Schreye and Decorte, 1994]. However, although recursion is considered of major importance in Logic Programming, and therefore also in Inductive Logic Programming, the links with program termination have rarely been studied in ILP.

One can argue the problem then only shifts towards the particular language bias. The advantage however is that this enables to identify classes of programs for which correctness is guaranteed. Although the study of language biases for recursive programs is outside the scope of this chapter and of this thesis, we consider this as an important topic for future research.

5.7 Refinement operators in ILP

In Section 5.10 we implement the refinement operators *mub*, *mlb*, *msg* and *mgs*. These refinement operators refine (i.e., generalize for the generalization operators and specialize for the specialization operators) a given clause w.r.t. another clause. To implement them we use refinement operators that minimally refine an element within $\mathcal{L}_C$, i.e., *not w.r.t. another clause*. We will need one such specialization operator, and one such generalization operator for $\mathcal{L}_C$.

[4]Ordered selection strategy with Linear resolution for Definite clauses with Tabling.

[van der Laag and Nienhuys-Cheng, 1994] presents both a locally finite and locally complete specialization and generalization operator. The specialization operator specializes a clause with three basic steps: unifying two distinct variables, unifying a variable with a term $f(X_1, \ldots, X_n)$ (where $X_1, \ldots, X_n$ are new distinct variables, and f/n is in $\mathcal{F}$), and adding a literal $p(X_1, \ldots, X_n)$ (where $X_1, \ldots, X_n$ are new distinct variables, and p/n is in $\mathcal{P}$) to the body of the clause. The generalization operator basically consists of the inverse of these operations.

The disadvantage of these operators is that they do not return reduced clauses, neither w.r.t. θ-subsumption, nor w.r.t. generalized subsumption. If an operator returns non-reduced clauses, one has to check explicitly whether the refinements are strictly more specific or more general than the original clause. [van der Laag and Nienhuys-Cheng, 1994] calls refinement operators that return only clauses more specific or general than the original clause *proper*, and proves that locally finite and complete proper refinement operators do not exist (neither specialization operators, nor generalization operators) for unrestricted spaces, i.e., first order languages with finitely many predicate symbols and function symbols without any further restriction. Basically problems arise in $\mathcal{L}_C$ because it contains infinite ascending and descending chains. Therefore, if we want generalization operators that are locally finite, complete and proper, we have to restrict the search space with a language bias which avoids the infinite chains. We could put, for instance, an upperbound on the total number of arguments all literals and all occurrences of terms can have in a clause. Alternatively, we could put an upperbound on the number of existential variables in the clause[5]. Actually, these constraints make Constraint 3.19 (the Finiteness Constraint) true.

In the rest of the thesis we will use the refinement operators of [van der Laag and Nienhuys-Cheng, 1994] when we need a specialization or generalization operator that minimally refines elements in $\mathcal{L}_C$, together with the assumption that the space is sufficiently restricted by a *language bias* to allow properness.

5.8 Language Bias

It is clear that the framework sketched so far is highly unpractical because of the huge size of $\mathcal{L}_C$. Therefore the use of a language bias is absolutely necessary. Language bias allows us to consider only that part of the concept representation language that seems worth searching.

Possible choices for language bias in ILP are, for instance,

1. that the body of an allowed clause can only contain a maximum number of terms[6] not occurring in the head of the clause (e.g., CLINT [De Raedt and Bruynooghe, 1990], LINUS [Lavrač *et al.*, 1991], ITOU [Rouveirol, 1992]), or

2. that the literals in the body of an allowed clause must be *linked* to the head through terms appearing in more than one literal, or

[5]Using flattening [Rouveirol, 1994], a representation trick to replace functors by predicates, both restrictions are equivalent.

[6]Using flattening [Rouveirol, 1994] one can actually restrict the number of *variables* not occurring in the head.

3. that the terms in an allowed clause have a maximum degree of indirect relevance to the terms in the head [7] (e.g., CLINT, GOLEM [Muggleton and Feng, 1992]), or

4. that the literals in an allowed clause must be determinate (i.e., the literal's variables not occurring in preceding literals have only one possible value, given the values of the literal's variables occurring in preceding literals) (e.g., GOLEM, DINUS [Lavrač and Džeroski, 1994]), or

5. that the literals in an allowed clause must be relevant to the concept (e.g., PGA [Buntine, 1987], [Subramanian and Genesereth, 1987], [Russell and Grosof, 1990], CLINT), or

6. that an allowed clause must be consistent w.r.t. given modes and types on the arguments of predicates in $\mathcal{P}$ and functors in $\mathcal{F}$ (e.g., MIS [Shapiro, 1983], GOLEM, FOIL , LINUS [Lavrač *et al.*, 1991], [Muggleton, 1995]) .

In the following example we will explain the above bias restrictions on an example clause c_{10}.

Example 5.40 Consider clause c_{10}.

$$
\begin{aligned}
c_{10} : & \ \mathit{succeeds}(\ E\ ) \leftarrow \\
& \quad \mathit{act}(\ E\ ,\ \mathit{dbl\_click}(\ D\ )\ ),\ \mathit{time}(\ E\ ,\ T\ ), \\
& \quad \mathit{isa\_document}(\ D\ ),\ \mathit{isa\_folder}(\ F\ ), \\
& \quad \mathit{holds\_at}(\ \mathit{is\_closed}(\ D\ )\ ,\ T\ ), \\
& \quad \mathit{holds\_at}(\ \mathit{is\_in}(\ D\ ,\ F\ )\ ,\ T\ ), \\
& \quad \mathit{holds\_at}(\ \mathit{is\_open}(\ F\ )\ ,\ T\ ).
\end{aligned}
$$

1. There is only one term appearing in the head of c_{10}: the variable E. All other terms only appear in the body of c_{10}.

2. The clause c_{10} is linked, because every literal in the body can be linked to the head. The time-dependent literals (i.e., the ones containing T) are linked through the literal $\mathit{time}(\ E\ ,\ T\ )$ to the head, because the latter contains the variable E of the head. The literal $\mathit{isa\_document}(\ D\ )$ contains the variable D, which occurs in $\mathit{act}(\ E\ ,\ \mathit{dbl\_click}(\ D\ )\ )$. The latter again contains the variable E. The literal $\mathit{isa\_folder}(\ F\ )$ contains the variable F, which occurs in $\mathit{holds\_at}(\ \mathit{is\_in}(\ D\ ,\ F\ )\ ,\ T\ )$.

3. In c_{10} all literals in the body can be linked to the head through at most two intermediate literals (see the previous item). The number of intermediate literals allowed could be bound: if it were bound by 0 or 1, c_{10} would not be allowed.

4. Whether the clause is determinate, depends on the meaning of the predicates, and the order of the literals. The clause c_{10} is not determinate if there is more than one folder: in that case there is more than one value for F in the literal $\mathit{isa\_folder}(\ F\ )$. If this literal would be put at the end of the body of c_{10}, and supposing that each event corresponds to exactly one action, that each event

[7]This is also called the *depth* of a variable in a functor-free context.

corresponds to exactly one timepoint and that each document is in exactly one folder, the clause is determinate.

5. Suppose $\mathcal{F}$ contains the predicate *was_last_changed*/2, expressing the time a document was last changed. If knowledge is available that literals of the form *holds_at*(*was_last_changed*(D , TC) , T) are irrelevant for defining the predicate *succeeds*/1, clauses for *succeeds*/1 containing a literal of this form are not allowed.
6. Knowledge of types of arguments can exclude clauses. If the second argument of *is_in*/2 is of the type *isa_folder*/1, and if *isa_folder*(D) is incompatible with *isa_document*(D), adding the literal *holds_at*(*is_in*(F , D) , T) to c_{10} is not allowed. Knowledge of modes is also useful: given that T and D are input for the literal *holds_at*(*is_closed*(D) , T), it is not allowed to remove the literal *time*(E , T) from c_{10}, because it outputs T.

◇

Several of these bias restrictions can be declaratively specified, but some of them are less declarative than others, e.g., those restrictions setting a bound on some numeric parameter in the clause. Therefore some recent systems introduce the same kind of restrictions using more declarative language bias specifications. Examples are clause models (introduced in the context of CLAUDIEN; see [Van Laer *et al.*, 1994], but also [Adé *et al.*, 1995]), clause schemata (introduced in the context of MOBAL [Morik *et al.*, 1993]) and predicate sets and clause sets (introduced in the context of TRACY [Bergadano and Gunetti, 1994]). Intuitively these formalisms allow to give a declarative specification of the syntactical constructs allowed in the language bias. This can reduce the search space drastically, because many useless clauses can be excluded syntactically. Procedurally these types of bias specify which (series of) refinement steps of the refinement operators given above are permitted. The main advantages of these declarative specifications of language bias is that differences between several biases can be described declaratively, and that changing to another bias only requires to replace the specification, without having to implement the corresponding refinement operators. The disadvantage is, however, that a lot of these syntactical constructs *hide* semantical restrictions as for instance relevance information, mode information or type information.

Example 5.41 A small example of a clause model is

```
succeeds( E ) <-
   act( E , dbl_click( D ) , time( E , T ) ,
   isa_document( D ) ,
   { < isa_folder( F ) ,
       { holds_at( is_in( D , F ) , T ) ,
         holds_at( [is_open,is_closed]( F ) , T ) ,
         holds_at( is_visible( F ) , T ) ,
         holds_at( is_in_open_dir( F ) , T ) } > ,
     holds_at( [is_open,is_closed]( D ) , T ) ,
     holds_at( is_in_open_dir( D ) , T ) } .
```

The clauses specified by this clause model all contain the same head *succeeds*(E). The literals *act*(E , *dbl_click*(R , S)), *time*(E , T) and *isa_document*(D) also appear in all of them. There are sets of elements delimited by { and }, by [and], or by < and >. The elements of the sets are separated by commas. A set between curly brackets means that any subset of the set can be included in a valid clause. A set between square brackets means that exactly one of the elements of the set should be included. A set between triangular brackets means that the elements in the set form a fixed combination. By nesting this kind of constructs a whole language bias of allowed constructs can be described. In the example any set of literals containing F must be accompanied by the literal *isa_folder*(F). The construct *holds_at*([*is_open*, *is_closed*](D) , T), expresses that only one of the literals *holds_at*(*is_open*(D) , T) *holds_at*(*is_closed*(D) , T) is allowed. $\diamond$

As before we denote the set of clauses denoted by the language bias $*$ on $\mathcal{L}_C$ as $\mathcal{L}_C^*$. In the remainder we will only make a specific choice for $*$ when it is really necessary.

5.9 Starting Clauses

There is a special case of generalization where language bias plays an important role: ~~generalization of $\perp$ with a lowerbound. These generalizations are called *starting clauses*.~~ We will first define starting clauses in our terminology.

Definition 5.42 (Starting clause) Given $i \in \mathcal{L}_I$, a starting clause is an element of $mub(\perp , i)$.

Given a positive lowerbound i, a starting clause is a maximally specific clause in $\mathcal{L}_C$ that is consistent with i, in the presence of background knowledge $\mathcal{B}$, and w.r.t. to the chosen language bias (i.e., starting clauses are in $\mathcal{L}_C^*$). The notion of a starting clause was introduced in the context of specific-to-general ILP systems, in particular in CLINT [De Raedt and Bruynooghe, 1988], working with lowerbounds (examples) only. However, starting clauses can also be used in other settings, because it is a way to transform instances into clauses. Several approaches have been developed to construct starting clauses. In any case, the body of a starting clause is a finite subset of the minimal Herbrand model of $\mathcal{B}$, or a subset of the minimal Herbrand model in which all distinct constants have been replaced by distinct variables. In practice it would not be feasible to add the minimal Herbrand model of any kind of background knowledge theory $\mathcal{B}$ completely to the body of a starting clause, because it may be too large (even when it is finite) or it may be infinite. Therefore the language bias must determine which part of the minimal Herbrand model to include. Language bias restrictions can reduce the size of the starting clause drastically. However, the less literals in the body of the starting clause, the more general the starting clause is w.r.t. θ-subsumption, such that the search space will be reduced. Consequently, the restrictions on the literals to be included in the starting clause mainly determine which part of the search space will be searched. Of course, this is at the risk of the reduced search space not containing the target concept. A general approach to alleviate this problem is the *shift of bias* approach (see Section 3.10). For further discussion and comparison of language biases and starting clauses in ILP we refer to [Adé *et al.*, 1995].

5.10 Implementing the refinement operators of ITVS [T]

SUMMARY: in this section we implement the four refinement operators used in the framework of ITVS, but also in DI, in the context of ILP.

The four operators we have to implement are *mub*, *mlb*, *msg* and *mgs*. Rephrased in terms of $\preccurlyeq_{\mathcal{B}}$ these operators are defined as:

Definition 5.43 (Refinement operators in ILP)

- $mub(\, c_1 \,,\, c_2 \,) = Min\ \{\ c \in \mathcal{L}_C \mid c_1 \preccurlyeq_{\mathcal{B}} c \text{ and } c_2 \preccurlyeq_{\mathcal{B}} c\ \}$;
- $mlb(\, c_1 \,,\, c_2 \,) = Max\ \{\ c \in \mathcal{L}_C \mid c \preccurlyeq_{\mathcal{B}} c_1 \text{ and } c \preccurlyeq_{\mathcal{B}} c_2\ \}$;
- $msg(\, c_1 \,,\, c_2 \,) = Min\ \{\ c \in \mathcal{L}_C \mid c_1 \preccurlyeq_{\mathcal{B}} c \text{ and } \neg(\ c \preccurlyeq_{\mathcal{B}} c_2\ )\ \}$;
- $mgs(\, c_1 \,,\, c_2 \,) = Max\ \{\ c \in \mathcal{L}_C \mid c \preccurlyeq_{\mathcal{B}} c_1 \text{ and } \neg(\ c_2 \preccurlyeq_{\mathcal{B}} c\ )\ \}$.

Max selects maximal elements w.r.t. $\preccurlyeq_{\mathcal{B}}$; *Min* selects minimal elements w.r.t. $\preccurlyeq_{\mathcal{B}}$. To implement these operators, we first introduce a generic algorithm for minimal refinement.

5.10.1 Minimal refinements

```
procedure minimal_refine ( c: concept; $\rho_{\trianglelefteq}$: refinement operator;
            $selection\_criterion, prune\_criterion$: boolean function of concept )
            returns set of concept
  $R := \emptyset$
  $Q := \{\ c\ \}$
  while $Q \neq \emptyset$
      do q := select one $q$ from $Q$ with $true$
         $Q := Q \setminus \{\ q\ \}$
         $Q_{new}$ := select all $q'$ from $\rho_{\trianglelefteq}(\ q\ )$
                  with $\neg\exists q'' \in \rho_{\trianglelefteq}(\ q\ ) \cup Q \cup R : q'' \trianglelefteq q'$ and $q'' \neq q'$
                   and not $prune\_criterion(\ q'\ )$
         $R := R \cup \{\ q' \in Q_{new} \mid selection\_criterion(\ q'\ )\ \}$
         $Q := Q \cup \{\ q' \in Q_{new} \mid \neg selection\_criterion(\ q'\ )\ \}$
  endwhile
  return $R$
endproc
```

Algorithm 5.1 Searching for minimal refinements

The procedure minimal_refine (see Algorithm 5.1) accepts as input a concept c, a proper refinement operator $\rho_{\trianglelefteq}$ (see Section 5.7), a selection criterion and a pruning criterion. The refinement operator $\rho_{\trianglelefteq}$ can be a locally complete specialization operator $\rho_{\preccurlyeq}$ (in which case $\trianglelefteq$ would be $\preccurlyeq$) or a locally complete generalization operator $\rho_{\succcurlyeq}$ (in which case $\trianglelefteq$ would be $\succcurlyeq$). We will use the refinement operators of [van der Laag and Nienhuys-Cheng, 1994] introduced in Section 5.7. The selection criterion and the pruning criterion are both boolean functions of a concept. The following relations are required between the refinement operator and the two criteria:

- if c_1 can be pruned and $c_2 \trianglelefteq c_1$, then c_2 can be pruned;
- if c_1 can be pruned and $c_2 \trianglelefteq c_1$, then c_2 does not satisfy the selection criterion.

This means that subtrees of the search tree whose root fulfills the pruning criterion do not contain concept representations fulfilling the selection criterion. minimal_refine then returns all refinements of c that fulfill the selection criterion, that cannot be pruned, and that are maximal for $\trianglelefteq$.

The algorithm implements a complete search using the ideas of ITVS to obtain maximally specific or maximally general elements only. Q is the set of elements still to be refined, and R is the set of already found refinements of c fulfilling the selection criterion and maximal for $\trianglelefteq$. Initially $Q = \{ c \}$, and $R = \emptyset$. In the while-loop one element q is removed from Q and refined one step. The set of refinements of q

- that are maximal for $\trianglelefteq$ w.r.t. the other elements in $\rho_{\trianglelefteq}(q)$,
- that are maximal for $\trianglelefteq$ w.r.t. the elements in R, and
- that are maximal for $\trianglelefteq$ w.r.t. the elements in Q, and
- that cannot be pruned

is assigned to Q_{new}. These conditions implement an optimal refinement operator as in ITVS. Elements that do not fulfill one of these conditions are not to be refined, either because they are not maximal, or because they can be obtained by refining another element in Q or $\rho_{\trianglelefteq}(q)$, or because they can be pruned. The elements in Q_{new} that satisfy the selection criterion are solutions, and therefore added to R. The others are added to Q. Care should be taken that the algorithm halts. This depends on the chosen selection and pruning criterion, and on the language bias. The pruning criterion is used to prune the search with global search conditions (i.e., conditions not depending on Q or R). These global search conditions can, for instance, take B_s or B_g into account. This is useful because the operations *mub*, *mlb*, *msg* and *mgs* are often followed by an extra selection step to select only maximally general elements w.r.t. B_g, maximally specific elements w.r.t. B_s, or consistent elements w.r.t. I_g or I_s. In DI, the pruning criterion can be used to prune w.r.t. $\mathcal{G}$ or $\mathcal{S}$. The pruning criterion is then used to incorporate these selection steps into the refinement operators.

This generic algorithm for minimal refinement is similar in structure to the refinement operator described in [Shapiro, 1983]. However, the refinement operator of [Shapiro, 1983] is different in the following respects: it searches general-to-specific only; it searches breadth-first; it searches for only one solution; it always starts from $\top$; it does not allow for a generic pruning and selection criterion, and it does not use the optimal pruning of ITVS.

We will now use this generic algorithm to implement the operators of Definition 5.43.

5.10.2 The generalization operator *mub*

Because all elements of $\mathcal{L}_I$ are transformed to starting clauses we can assume we only have to consider $mub(c_1 , c_2)$ where $c_1, c_2 \in \mathcal{L}_C$. If no starting clause can be found for an information element i, $mub(c_1 , i) = \emptyset$.

Proposition 5.44 All elements of $mub(\ c_1\ ,\ c_2\ )$ are equivalent to $lgg(\ Sat(\ c_1\ )\ ,\ Sat(\ c_2\ )\ )$ w.r.t. generalized subsumption.

Proof For all $c \in mub(\ c_1\ ,\ c_2\ )$ we have $c_1 \preccurlyeq_{\mathcal{B}} c$ and $c_2 \preccurlyeq_{\mathcal{B}} c$. Because of Theorem 5.31, we have $Sat(\ c_1\ ) \preccurlyeq_\theta c$ and $Sat(\ c_2\ ) \preccurlyeq_\theta c$. Now consider a least general generalization l of $Sat(\ c_1\ )$ and $Sat(\ c_2\ )$. By definition of least general generalization $l \preccurlyeq_\theta c$. Consequently $l \preccurlyeq_{\mathcal{B}} c$. Since c must be minimal in $mub(\ c_1\ ,\ c_2\ )$, $l \equiv_{\mathcal{B}} c$. Since all least general generalizations are equivalent w.r.t. θ-subsumption, since $\equiv_\theta$ is weaker than $\equiv_{\mathcal{B}}$ and since $\equiv_{\mathcal{B}}$ is transitive, all minimal upperbounds of c_1 and c_2 are equivalent to l w.r.t. generalized subsumption. □

This proposition has also been formulated by [Jung, 1993], and more generally for relative least general generalization by [Muggleton, 1992]. It means that to compute $mub(\ c_1\ ,\ c_2\ )$ we only have to compute the lgg of $Sat(\ c_1\ )$ and $Sat(\ c_2\ )$. All other minimal upperbounds belong to the same class of $\equiv_{\mathcal{B}}$.

Although we now have a way to compute mub for any two clauses in $\mathcal{L}_C$, this does not mean that this minimal upperbound is allowed in the language bias $*$. The elements of mub w.r.t. the bias must be more general than the minimal upperbound l without the bias, though. So to compute them, we have to generalize l minimally such that it is in $\mathcal{L}_C^*$. This can be done using the procedure minimal_refine (see Algorithm 5.1) with

- $c = lgg(\ Sat(\ c_1\ )\ ,\ Sat(\ c_2\ )\ )$;
- $\rho_{\trianglelefteq} = \rho_{\succcurlyeq}$;
- $selection\_criterion(\ q\ ) = (\ q \in \mathcal{L}_C^*\ )$;
- $prune\_criterion(\ q\ ) = (\ false\ )$.

We can further optimize this instantiation when incorporating it in ITVS by choosing a particular pruning criterion. The operation mub is only used in generalizations of Algorithm 3.4. Each call to generalizations in the algorithms of ITVS (Algorithm 3.4 in particular), of the extensions of ITVS (Algorithm 3.8) and of DITVS (Algorithm 4.8) is immediately followed by a selection of particular elements from those returned by generalizations, i.e., from the elements returned by mub. The negation of the conditions of each of these selections could be used as a pruning criterion to optimize the instantiation of minimal_refine. Consider for instance generalize in Algorithm 3.4. Step 3.19, which follows the call to generalizations, selects those generalizations that are consistent with all g-bounds, and that are maximally specific w.r.t. the alternatives on B_s. Then we could take the negation of this condition as a pruning criterion for mub, i.e.,

$$prune\_criterion(\ q\ ) = (\ \neg all\_consistent(\ q\ ,\ I_g\ ,\ n_g\ )\ \text{or}\ \neg max\_specific(\ q\ ,\ B_s\ )\ ).$$

Indeed, if c_1 is not consistent with all g-bounds, then a generalization of c_1 cannot be consistent with all g-bounds either; and if there is an element on B_s which is more specific than c_1, then this element will be more specific than each generalization of c_1 as well.

Whether the algorithm halts, depends on the chosen language bias. In case the language bias restricts the number of existential variables or the total number of arguments of literals and occurrences of terms (as suggested in Section 5.7) it is guaranteed to halt, because minimal subsets of literals will be dropped from $mub(\ c_1\ ,\ c_2\ )$ such that the result is allowed in $\mathcal{L}_C^*$.

Note that this is just a general approach to compute mub. For particular language biases it might not be necessary to apply minimal_refine at all. E.g., if no existential variables are allowed by the language bias, the result can be obtained by removing all literals from $lgg(\ Sat(\ c_1\ )\ ,\ Sat(\ c_2\ )\ )$ that contain existential variables.

5.10.3 The specialization operator *mlb*

The case of *mlb* is dual to the one of *mub*. All maximal lowerbounds of c_1 and c_2 are equivalent w.r.t. generalized subsumption to the least specific specialization of c_1 and c_2. Moreover, the result might again not be allowed by the language bias. In that case it will have to be minimally specialized using an instantiation of `minimal_refine`.

Proposition 5.45 If $lss(\ c_1\ ,\ c_2\ )$ exists, all elements of $mlb(\ c_1\ ,\ c_2\ )$ are equivalent to $lss(\ c_1\ ,\ c_2\ )$ w.r.t. generalized subsumption.

Proof For all $c \in mlb(\ c_1\ ,\ c_2\ )$ we have $c \preccurlyeq_{\mathcal{B}} c_1$ and $c \preccurlyeq_{\mathcal{B}} c_2$. Because of Theorem 5.31, we have $Sat(\ c\ ) \preccurlyeq_\theta c_1$ and $Sat(\ c\ ) \preccurlyeq_\theta c_2$. Now consider a least specific specialization l of c_1 and c_2. By definition of least specific specialization $Sat(\ c\ ) \preccurlyeq_\theta l$. Consequently $Sat(\ c\ ) \preccurlyeq_{\mathcal{B}} l$ and $c \preccurlyeq_{\mathcal{B}} l$ (Lemma 5.38). Since c must be maximal in $mlb(\ c_1\ ,\ c_2\ )$, $c \equiv_{\mathcal{B}} l$. Since all least specific specializations are equivalent w.r.t. θ-subsumption, since $\preccurlyeq_\theta$ is weaker than $\preccurlyeq_{\mathcal{B}}$ and since $\equiv_{\mathcal{B}}$ is transitive, all maximal lowerbounds of c_1 and c_2 are equivalent to l w.r.t. generalized subsumption. □

This theorem means that to compute $mlb(\ c_1\ ,\ c_2\ )$ we only have to compute the lss of c_1 and c_2, and saturate it. All other maximal lowerbounds belong to the same class of $\equiv_{\mathcal{B}}$.

If $lss(\ c_1\ ,\ c_2\ )$ does not exist, $mlb(\ c_1\ ,\ c_2\ ) = \emptyset$.

If $lss(\ c_1\ ,\ c_2\ )$ is not allowed by the language bias $*$, the elements of $mlb(\ c_1\ ,\ c_2\ )$ are the minimal specializations of c_1 and c_2 that are in $\mathcal{L}_C^*$. These can be computed using the procedure `minimal_refine` (see Algorithm 5.1) with

- $c = lss(\ c_1\ ,\ c_2\ )$;
- $\rho_{\trianglelefteq} = \rho_{\preccurlyeq}$;
- $selection\_criterion(\ q\ ) = (\ q \in \mathcal{L}_C^*\ )$;
- $prune\_criterion(\ q\ ) = (\ false\ )$.

Again we can further optimize this instantiation by choosing a particular pruning criterion. The operation *mlb* is only used in `specializations` of Algorithm 3.5. Each call to `specializations` in the algorithms of ITVS (Algorithm 3.5 in particular), of the extensions of ITVS (Algorithm 3.11) and of DITVS (Algorithm 4.6) is immediately followed by a selection of particular elements from those returned by `specializations`, i.e., from the elements returned by *mlb*. As in the implementation of *mub*, the negation of the conditions of each of these selections could be used as a pruning criterion to optimize the instantiation of `minimal_refine`. Consider for instance `specialize_disjuncts` in Algorithm 4.6. Step 4.38, which follows the call to `specializations`, selects those specializations that are maximally general w.r.t. the alternatives on DB_g. Then we could take

$$prune\_criterion(\ q\ ) = (\ \neg d\_max\_general(\ q\ ,\ g_{vs}\ ,\ DB_g\ )\ ),$$

because if c_1 is not maximally general w.r.t. the alternatives on DB_g, then a specialization of c_1 cannot be maximally general w.r.t. the alternatives on DB_g either.

Again the termination of the algorithm depends on the chosen language bias, and language bias specific algorithms might replace the general strategy. In case the language bias restricts the number of existential variables, existential variables can be removed by substituting variables and terms from the head for existential variables. There could be a problem when there are more existential variables than variables in the head of the clause: replacing existential variables by distinct ground terms always gives a maximally general element of $\mathcal{L}_C^*$. If an infinite number of clauses remains after pruning, this means that Constraint 3.19 (the Finiteness Constraint) is not fulfilled. In case the language bias restricts the total number of arguments of literals and occurrences of terms, $mlb(c_1 , c_2)$ is empty when $lss(c_1 , c_2) \notin \mathcal{L}_C^*$. In that case, specialization, i.e., adding extra literals or terms, will not help to obtain elements in $\mathcal{L}_C^*$.

5.10.4 The specialization operator *mgs*

If $\neg(c_2 \preccurlyeq c_1)$, $mgs(c_1 , c_2) = \{ c_1 \}$. To compute $mgs(c_1 , c_2)$ when $c_2 \preccurlyeq c_1$, we instantiate `minimal_refine` with

- $c = c_1$;
- $\rho_{\trianglelefteq} = \rho_{\preccurlyeq}$;
- $selection\_criterion(q) = (q \in \mathcal{L}_C^* \text{ and } \neg(c_2 \preccurlyeq q))$;
- $prune\_criterion(q) = (false)$.

As an optimization the pruning criterion can be instantiated exactly the same way as in *mlb*. Also, for particular language biases, more specific solutions could be implemented.

5.10.5 The generalization operator *msg*

This case is dual to the implementation of *mgs*. If $\neg(c_1 \preccurlyeq c_2)$, $mgs(c_1 , c_2) = \{ c_1 \}$. To compute $mgs(c_1 , c_2)$ when $c_1 \preccurlyeq c_2$, we instantiate `minimal_refine` with

- $c = c_1$;
- $\rho_{\trianglelefteq} = \rho_{\succcurlyeq}$;
- $selection\_criterion(q) = (q \in \mathcal{L}_C^* \text{ and } \neg(q \preccurlyeq c_2))$;
- $prune\_criterion(q) = (false)$.

For optimization the pruning criterion should be instantiated as in *mub*.

5.11 Instance Generation in ILP

In this section we will briefly elaborate on an implementation of the heuristic to find a middle in the context of instance generation (see Section 3.11), because this will be used to generate experiments in the context of an autonomous agent in Chapter 6.

Let us first introduce the following notation.

Notation 5.46 Let $c = (h \leftarrow b_1 , \ldots , b_n)$ be a definite clause. We denote the clause obtained by removing the literal b_j from the body of c, i.e., the clause

$$c = (h \leftarrow b_1 , \ldots , b_{j-1}, b_{j+1} , \ldots , b_n),$$

as $c \ominus \{ b_j \}$. We denote the clause obtained by adding a set of literals $\{ b'_1 , \ldots , b'_k \}$ to the body of c, i.e., the clause

$$c = (h \leftarrow b_1 , \ldots , b_n, b'_1 , \ldots , b'_k),$$

as $c \oplus \{ b'_1 , \ldots , b'_k \}$.

A lowerbound i is relevant (see Definition 3.59) if there is a maximally specific s, a maximally general g, a c_1 and a c_2 in $\mathcal{L}_C$ such that $s \preccurlyeq c_1 \prec c_2 \preccurlyeq g$, and $\neg(i \preccurlyeq c_1)$ and $i \preccurlyeq c_2$. Ideally c_1 and c_2 are chosen close to a middle point (Definition 3.60) between c_1 and c_2: knowing whether i is a positive lowerbound or a negative lowerbound would then exclude half of the candidate hypotheses. In this section we will describe a heuristic to find such a c_1, c_2 and i.

We will base ourselves on a propositional representation of the clauses s, g, c_1 and c_2. Furthermore we assume that for all predicates $p/n \in \mathcal{P}$ we can express that an instance i is *not* covered by p/n. This could e.g., be done by defining a predicate $\overline{p}/n$ for p/n, which covers i if p/n does not cover i.

Let $g = h \leftarrow b_1 , \ldots, b_k$, and let $s = h \leftarrow b_1 , \ldots , b_n$, with $k \leq n$, be two saturated clauses. Assume that neither g nor s contains at the same time a literal b and $\overline{b}$, otherwise they would not cover any instances. There are 2^{n-k} clauses more general than s and more specific than g: the set of all clauses of the form $s \oplus \{ b'_1 , \ldots, b'_j \}$, where $\{ b'_1 , \ldots, b'_j \} \subseteq \{ b_{k+1} , \ldots, b_n \}$. We will try to find an lowerbound that is consistent with 2^{n-k-1} of these clauses, and inconsistent with the other 2^{n-k-1} clauses.

Take a literal $b \in \{ b_{k+1} , \ldots, b_n \}$, such that the saturation of $s \ominus \{ b \}$ is not s. Let

- $c_2 = g$,
- $c_1 = (g \oplus \{ b \})$, and
- $i = (g \oplus \{ \overline{b} \})$.

The literal $\overline{b}$ is not in c_2, because then s would contain b as well as $\overline{b}$. Then we have $i \preccurlyeq c_2$, $\neg(i \preccurlyeq c_1)$, and $c_1 \prec c_2$. Consequently i is a relevant lowerbound.

If i is a positive lowerbound, s should be generalized to be consistent with i, i.e., the literal b should be dropped from s. All clauses consistent with i, more specific than g, and more general than s should not contain the literal b. Consequently, half of the clauses more general than s and more specific than g are excluded as hypothesis.

If i is a negative lowerbound, g should be specialized to be consistent with i, i.e., the literal b should be added to g. All clauses consistent with i, more specific than g and more general than s should contain the literal b. Consequently, half of the clauses more general than s and more specific than g are excluded as hypothesis.

Note that the argument that half of the hypotheses can be excluded is, in general, only correct for propositional clauses. However, it can be used as a heuristic in case of first-order logic clauses.

Another choice would be the following: let b be a literal in $\{ b_{k+1} , \ldots, b_n \}$, such that the saturation of $s \ominus \{ b \}$ is not s. Let

- $c_1 = s$,
- $c_2 = (s \ominus \{ b \})$, and
- $i = (s \ominus \{ b \} \oplus \{ \overline{b} \})$.

The literal $\overline{b}$ is not in c_2, because then s would contain b as well as $\overline{b}$. Then we have $i \preccurlyeq c_2$, $\neg(i \preccurlyeq c_1)$, and $c_1 \prec c_2$. Consequently i is again a relevant lowerbound.

The advantage of a bi-directional approach over a general-to-specific approach or a specific-to-general approach, is that the bidirectional approach has s and g available. Therefore it can determine which literals should be dropped to obtain *relevant* lower-bounds. If, in a specific-to-general approach, only s were available, dropping a literal b from s would not necessarily give a relevant upperbound if there is no g more general than s that does not contain b.

For examples, we refer to Section 5.12.

We will now consider the case of relevant upperbounds. An upperbound i is relevant (see Definition 3.59) if there is a maximally specific s, a maximally general g, a c_1 and a c_2 in $\mathcal{L}_C$ such that $s \preccurlyeq c_1 \prec c_2 \preccurlyeq g$, and $c_1 \preccurlyeq i$ and $\neg(c_2 \preccurlyeq i)$.

Again let $g = (h \leftarrow b_1 , \ldots , b_k)$ and $s = (h \leftarrow b_1 , \ldots , b_n)$, with $k \leq n$, be two saturated clauses, not containing at the same time a literal b_j and a literal $\overline{b_j}$, and this for any j.

Let b be a literal in $\{ b_{k+1} , \ldots , b_n \}$, such that the saturation of $s \ominus \{ b \}$ is not s. Let $c_2 = g$, $c_1 = (g \oplus \{ b \})$, and $i = (g \oplus \{ \overline{b} \})$. Then we have $\neg(c_2 \preccurlyeq i)$, $c_1 \preccurlyeq i$, and $c_1 \prec c_2$. Consequently i is a relevant upperbound.

If i is a negative upperbound, s should be generalized to be consistent with i, i.e., the literal b should be dropped from s. All clauses consistent with i, more specific than g, and more general than s should not contain the literal b. Consequently, half of the clauses more general than s and more specific than g are excluded as hypothesis.

If i is a positive upperbound, g should be specialized to be consistent with i, i.e., the literal b should be added to g. All clauses consistent with i, more specific than g and more general than s should contain the literal b. Consequently, half of the clauses more general than s and more specific than g are excluded as hypothesis.

5.12 Example

The example presented in this section is situated in the context of the autonomous tutor introduced in Section 1.3. In this example the concept "successfully dragging a document D from folder F_1 to another folder F_2" is learned. The information elements in the infostream of this example originate from the example in Section 6.6. Section 6.6 gives an example session of the integration of ILP with an AI planning system, based on the event calculus representation; it merely shows how in the resulting architecture planning, learning, executing actions and observing effects interact. Here we will concentrate on the evolution of the definition of one of the concepts involved in that example.

The predicates of the event calculus are described in detail in Section 6.3. In this section we introduce the clauses we need in the course of the example.

We do have to give some preliminary remarks about the chosen language and language bias, and about the background knowledge, though. The concept to be learned is "successfully dragging a document D from folder F_1 to another folder F_2". This could be represented as in Example 5.6. However, as in Example 5.7 we will use the event calculus notation to represent the learned definitions.

The bottom $\perp$ of the language is the empty clause. (see Program 5.1). The top $\top$ of the language is Clause (5.2). This means that the body of each clause must contain at least the literal *act*(E , *drag_and_drop*(D , $Loc1$, $Loc2$)). Because we will use range-

(5.1) $\perp = \Box$.

(5.2) $\top =$ *succeeds*(E) $\leftarrow$
act(E , *drag_and_drop*(*Doc* , *Loc1* , *Loc2*)),
isa_document(*Doc*), *isa_location*(*Loc1*), *isa_location*(*Loc2*) .

(5.3) *incompatible*(*is_in*(X , Y) , *is_not_in*(X , Y)).
{ For all X and Y, *is_in*(X , Y) is incompatible with *is_not_in*(X , Y). }

Program 5.1 Language and language bias in the example

restricted clauses only, each variable in the clause ranges over a particular domain by including the type of the variable. The possible types are *isa_folder*/1, *isa_document*/1, *isa_location*/1, *isa_desktop*/1, *isa_object*/1 and *isa_event*/1. The chosen language bias also restricts the number of variables in the clause to those occurring in the head of the clause or occurring in the literal *act*(E , *drag_and_drop*(*Doc* , *Loc1* , *Loc2*)). This means that each lowerbound corresponds to a unique starting clause.

Another bias restriction concerns negations of predicates, and negations of properties. On the one hand information like Clause (5.3) is valuable meta-knowledge for the planner used in Chapter 6, in order not to consider subgoals containing incompatible literals. On the other hand, if we introduce negations of properties in order to generate relevant lower- and upperbounds (see Section 5.11), this information is also useful. It allows to remove clauses containing incompatible literals from consideration during learning, i.e., these clauses are not allowed in the language bias. We also do not allow constants in the body of the clauses. This means all distinct constants will be replaced by distinct variables when `minimal_refine` is used to minimally generalize clauses such that they are allowed by the language bias (see Section 5.10.2).

(5.4) *isa_object*(X) $\leftarrow$ *isa_document*(X).
{ Each document is an object. }

(5.5) *isa_object*(X) $\leftarrow$ *isa_folder*(X).
{ Each folder is an object. }

(5.6) *isa_location*(X) $\leftarrow$ *isa_folder*(X).
{ Each folder is a location. }

(5.7) *isa_location*(X) $\leftarrow$ *isa_desktop*(X).
{ Each desktop is a location. }

(5.8) *isa_folder*(X) $\leftarrow$ *isa_object*(X),*isa_location*(X).
{ Each object that is a location is a folder. }

Program 5.2 Background knowledge about types

The background knowledge contains some relations about the types, shown in Program 5.2. The background knowledge also contains clauses as shown in Program 5.3.

(5.9) $holds\_at(\ is\_not\_in(\ Obj\ ,\ Loc1\ )\ ,\ E\ ) \leftarrow$
$isa\_object(\ Obj\ ),\ isa\_location(\ Loc1\ ),\ isa\_location(\ Loc2\ ),$
$different\_loc(\ Loc1\ ,\ Loc2\ ),$
$holds\_at(\ is\_in(\ Obj\ ,\ Loc2\ )\ ,\ E\ )\ .$
{ An object *Obj* in location *Loc2* is not in a location *Loc1* which is different }

(5.10) $holds\_at(\ is\_in\_open\_location(\ Obj\ )\ ,\ E\ ) \leftarrow$
$isa\_object(\ Obj\ ),\ holds\_at(\ is\_in(\ Obj\ ,\ Loc\ )\ ,\ E\ ),$
$holds\_at(\ is\_open(\ Loc\ )\ ,\ E\ ),\ isa\_location(\ Loc\ )\ .$
{ If an object *Obj* is in location *Loc2* which is open, then *Obj* is in an open }

(5.11) $holds\_at(\ is\_not\_in\_open\_location(\ Obj\ )\ ,\ E\ ) \leftarrow$
$isa\_object(\ Obj\ ),\ holds\_at(\ is\_in(\ Obj\ ,\ Loc\ )\ ,\ E\ ),$
$holds\_at(\ not\_is\_open(\ Loc\ )\ ,\ E\ ),\ isa\_location(\ Loc\ )\ .$
{ If object *Obj* is in location *Loc2* which is not open, then *Obj* is not in an }

(5.12) $eternally(\ different\_loc(\ Loc1\ ,\ Loc2\ )\ ) \leftarrow$
$isa\_location(\ Loc1\ ),\ isa\_location(\ Loc2\ ),$
$Loc1 \neq Loc2\ .$
{ If location *Loc1* is not equal to location *Loc2*, }

(5.13) $eternally(\ is\_open(\ DT\ )\ ) \leftarrow\ isa\_desktop(\ DT\ )\ .$
{ A desktop is always open. }

(5.14) $holds\_at(\ P\ ,\ E\ ) \leftarrow\ eternally(\ P\ ),\ isa\_event(\ E\ )\ .$
{ A property *P* holds at the time of event *E* if *P* holds eternally. }

Program 5.3 Background knowledge

Clause (5.14) is a domain-independent clause to link the predicates *holds_at*/2 and *eternally*/1.

Although our instantiation of DITVS works with saturated clauses, we will not show the saturated clauses for reasons of readability: literals that can be re-introduced by saturation with the clauses of Program 5.2 and Program 5.3 are omitted. As an example of a saturated clause, consider Clause (5.15). It is the saturation of Clause (5.17) below.

```
(5.15)  succeeds( E ) ←
            act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
            isa_document( Doc ), isa_object( Doc ),
            isa_folder( Loc1 ), isa_object( Loc1 ), isa_location( Loc1 ),
            isa_folder( Loc2 ), isa_object( Loc2 ), isa_location( Loc2 ),
            different_loc( Loc1 , Loc2 ), different_loc( Loc2 , Loc1 ),
            holds_at( is_in( Obj , Loc1 ) , E ),
            holds_at( is_in( Loc2 , Loc1 ) , E ),
            holds_at( is_in_open_location( Doc ) , E ),
            holds_at( is_in_open_location( Loc1 ) , E ),
            holds_at( is_in_open_location( Loc2 ) , E ),
            holds_at( is_not_in( Doc , Loc2 ) , E ),
            holds_at( is_not_in( Loc1 , Loc2 ) , E ),
            holds_at( is_not_in( Loc2 , Loc2 ) , E ),
            holds_at( is_not_in( Loc1 , Loc1 ) , E ),
            holds_at( is_open( Loc1 ) , E ),
            holds_at( is_open( Loc2 ) , E ),
            holds_at( not_is_open( Obj ) , E ) .
```

Let $g = g_1 \vee \cdots \vee g_n$ be the current maximally general concept representation, and let $s = s_1 \vee \cdots \vee s_n$ be the current almost maximally specific concept representation under g. Initially $n = 1$, $g = g_1 = \top$, and $s = s_1 = \bot$.

Program 5.4 gives the consecutive information elements on the infostream. Clause (5.16) is a positive upperbound. With this information element g_1 is specialized to Clause (5.16) itself.

The other information elements are positive and negative lowerbounds. For each of these, Program 5.4 gives the corresponding starting clause, together with a substitution to obtain the actual positive or negative lowerbound. Applying the substitution to the head of the starting clause gives the actual positive or negative lowerbound; applying the substitution to the body of the clause gives a set of facts that are in the background knowledge.

The second information element, for instance, is a positive lowerbound. Clause (5.17) is the corresponding starting clause. The literal *succeeds*(*e*305)[8] is the actual positive lowerbound. Applying the corresponding substitution to the body of Clause (5.17) gives a set of facts which basically describe the (relevant part of) the situation at the time of event e305. All these facts are true in the background knowledge.

After processing this positive lowerbound, s_1 has been generalized to Clause (5.17) itself (i.e., the minimal upperbound of Clause (5.17) and $\bot$), while g_1 remains unchanged.

[8]The names e1, e2, ... for the events are generated by the planning system. That the numbers are not consecutive is because not all events that are considered are actually executed.

```
(5.16)  succeeds( E ) ←
              act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
              isa_document( Doc ), isa_location( Loc1 ), isa_location( Loc2 ),
              different_loc( Loc1 , Loc2 ), holds_at( is_in( Doc , Loc1 ) , E ) .
Positive upperbound

(5.17)  succeeds( E ) ←
              act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
              isa_document( Doc ), isa_folder( Loc1 ),
              isa_folder( Loc2 ), different_loc( Loc1 , Loc2 ),
              holds_at( is_in( Doc , Loc1 ) , E ), holds_at( is_in( Loc2 , Loc1 ) , E ),
              holds_at( is_in_open_location( Loc1 ) , E ),
              holds_at( is_not_in( Loc1 , Loc2 ) , E ),
              holds_at( is_open( Loc1 ) , E ), holds_at( is_open( Loc2 ) , E ),
              holds_at( not_is_open( Doc ) , E ) .
Positive lowerbound - { E/e305 , Doc/'thesis.mbox' , Loc1/home , Loc2/thesis }

(5.18)  succeeds( E ) ←
              act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
              isa_document( Doc ), isa_location( Loc1 ),
              isa_location( Loc2 ), different_loc( Loc1 , Loc2 ),
              holds_at( is_in( Doc , Loc1 ) , E ),
              holds_at( is_not_in( Loc2 , Loc1 ) , E ),
              [holds_at( is_open( Loc1 ) , E ), holds_at( is_open( Loc2 ) , E ),
              holds_at( not_is_open( Doc ) , E )] .
Positive lowerbound - { E/e317 , Doc/'thesis.tex' , Loc1/thesis , Loc2/home }

(5.19)  succeeds( E ) ←
              act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
              isa_document( Doc ), isa_location( Loc1 ),
              isa_location( Loc2 ), different_loc( Loc1 , Loc2 ),
              holds_at( is_in( Doc , Loc1 ) , E ),
              holds_at( is_not_in_open_location( Doc ) , E ),
              [holds_at( is_open( Loc2 ) , E ),
              holds_at( not_is_open( Doc ) , E ),
              holds_at( not_is_open( Loc1 ) , E )] .
Negative lowerbound - { E/e324 , Doc/'thesis.dvi' , Loc1/texfiles , Loc2/home }
```

Program 5.4 Information elements - Part1

```
(5.20)  succeeds( E ) ←
            act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
            isa_document( Doc ), isa_folder( Loc1 ), isa_desktop( Loc2 ),
            holds_at( is_in( Doc , Loc1 ) , E ),
            holds_at( is_not_in( Loc1 , Loc2 ) , E ),
            holds_at( is_open( Loc1 ) , E ),
            holds_at( not_is_open( Doc ) , E ) .
Positive lowerbound - { E/e337 , Doc/'invitation.tex' , Loc1/thesis , Loc2/desktop }

(5.21)  succeeds( E ) ←
            act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
            isa_document( Doc ), isa_location( Loc1 ),
            isa_folder( Loc2 ), different_loc( Loc1 , Loc2 ),
            holds_at( is_in( Doc , Loc1 ) , E ),
            holds_at( is_in_open_location( Doc ) , E ),
            holds_at( is_not_in_open_location( Loc2 ) , E ),
            [ isa_desktop( Loc1 ), holds_at( is_open( Loc2 ) , E ),
            holds_at( not_is_open( Doc ) , E )] .
Positive lowerbound - { E/e348 , Doc/'invitation.tex' , Loc1/desktop , Loc2/thesis }
...

(5.22)  succeeds( E ) ←
            act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
            isa_document( Doc ), isa_folder( Loc1 ),
            isa_folder( Loc2 ), different_loc( Loc1 , Loc2 ),
            holds_at( is_in( Doc , Loc1 ) , E ),
            holds_at( is_in_open_location( Loc2 ) , E ),
            holds_at( is_open( Loc1 ) , E ), holds_at( not_is_open( Doc ) , E ),
            holds_at( not_is_open( Loc2 ) , E ) .
Negative lowerbound - { E/e405 , Doc/'invitation.ps' , Loc1/home , Loc2/psfiles }
```

Program 5.5 Information elements - Part 2

By means of g_1 and s_1 a relevant instance can be generated, by taking a literal from s_1 such that the saturation of the rest of s_1 does not contain this literal, and adding the negation of that literal to g_1 (see Section 5.11). E.g., Clause (5.18) (without the part in square brackets) is obtained by adding *holds_at(is_not_in(Loc2 , Loc1) , E)* from the body of s_1 to g_1. An instantiation of the resulting clause provides a relevant lowerbound. The starting clause of the instantiation can contain more literals than was specified. These literals are shown between square brackets.

In Section 6.6 more than one concept is learned simultaneously. For instance, also definitions for the concept "successfully dragging document D from location F_1 to another location F_2 has the effect that D is in F_2" are derived. Generating relevant instances for the other concepts might at the same time give instances to update g and s. In this example, the learned concepts are very similar, such that we can explain the generated relevant instances by means of g_1 and s_1.

In Chapter 6 finding an instantiation of Clause (5.18) is done by *making a plan* to obtain a situation in which the body of the clause holds. In order to obtain this desired situation, the plan that is found must be executed. Executing this plan could also give new information elements to update g and s with. In the particular case of Clause (5.18) no intermediate actions have to be executed to find an instantiation of Clause (5.18): Clause (5.18) instantiated with the corresponding substitution { *E/e317* , *Doc/'thesis.tex'* , *Loc1/thesis* , *Loc2/home* } is a positive lowerbound. Therefore s_1 is generalized to Clause (5.23).

```
(5.23)  succeeds( E ) ←
                act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
                isa_document( Doc ), isa_folder( Loc1 ),
                isa_folder( Loc2 ), different_loc( Loc1 , Loc2 ),
                holds_at( is_in( Doc , Loc1 ) , E ),
                holds_at( is_in_open_location( Loc1 ) , E ),
                holds_at( is_in_open_location( Loc2 ) , E ),
                holds_at( is_open( Loc1 ) , E ), holds_at( is_open( Loc2 ) , E ),
                holds_at( not_is_open( Doc ) , E ) .
```

By searching for an instantiation of Clause (5.19) a new relevant lowerbound is generated. Clause (5.19) was obtained by adding *holds_at(is_not_in_open_location(Doc) , E)* to g_1. This time a negative lowerbound *succeeds*(e324) is generated. This means that the instantiation of the body of Clause (5.19) with the corresponding substitution is true, while the head of Clause (5.19) is false.

Processing this negative lowerbound gives rise to a VS-list of 11 specializations of g_1. One of these that is consistent with the negative lowerbound and with both positive lowerbounds is chosen as the new g_1:

```
(5.24)  succeeds( E ) ←
                act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
                isa_document( Doc ), isa_location( Loc1 ),
                isa_location( Loc2 ), different_loc( Loc1 , Loc2 ),
                holds_at( is_in( Doc , Loc1 ) , E ),
                holds_at( is_in_open_location( Doc ) , E ) .
```

Since s_1 is more specific than the new value of g_1, s_1 (and B_s for that matter) can be reused; actually s_1 remains unchanged.

A new relevant lowerbound is generated by finding an instantiation of Clause (5.21). In this case, some intermediate actions are executed during the search for an instantiation of Clause (5.21). The new positive lowerbound represented by starting clause Clause (5.20) and with corresponding instantiation is found. Since this positive lowerbound is consistent with g_1, s_1 must be generalized to be consistent with it as well. This results in Clause (5.25) as new value for s_1.

```
(5.25)  succeeds( E ) ←
            act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
            isa_document( Doc ), isa_folder( Loc1 ),
            isa_location( Loc2 ), different_loc( Loc1 , Loc2 ),
            holds_at( is_in( Doc , Loc1 ) , E ),
            holds_at( is_in_open_location( Loc1 ) , E ),
            holds_at( is_open( Loc1 ) , E ), holds_at( is_open( Loc2 ) , E ),
            holds_at( not_is_open( Doc ) , E ) .
```

Then, the instantiation of Clause (5.21) with the corresponding instantiation is a positive lowerbound. Further generation of relevant lowerbounds gives four more positive lowerbounds. All these positive lowerbounds are consistent with g_1, so s_1 has to be generalized again and becomes equivalent to g_1, i.e., to Clause (5.24).

Then the negative lowerbound Clause (5.22) is in the infostream. It is not consistent with g_1. Therefore g_1 is specialized. There are four maximally general specializations of g_1, shown in Program 5.6. The underlined literals are the ones not in g_1. For each of the disjuncts the corresponding J_s is also shown. Since all clauses together are consistent with all s-bounds, combinations of these clauses are considered. None of these clauses is individually consistent with all eight s-bounds. Therefore disjunctions of these clauses are considered next. The first one according to the global order, i.e., the disjunction of Clause (5.26) and Clause (5.27), is consistent with all s-bounds, and assigned to g. Then the almost maximally specific s under g is computed. It is equivalent to the disjunction of Clause (5.26) and Clause (5.27), i.e., equivalent to g. Therefore no more relevant instances are generated. Furthermore the infostream is empty, so DITVS halts.

5.13 Conclusion

In this chapter we instantiated the framework of Versionspaces and Iterative Versionspaces of Chapter 3 for predicate learning, to obtain the predicate learning setting as it is usually studied in ILP. With this instantiation we have shown in detail how predicate learning amounts to concept learning in an ILP framework, and how general concept learning algorithms as DI and ITVS can be applied to ILP. When the conditions of Constraint 4.8 apply in ILP, we can also use DDI and DITVS to learn sets of clauses. DITVS is not applicable (at least not in a straightforward way) for multiple predicate learning and for recursive predicates. Future work could point out whether the framework of DITVS could be extended towards learning multiple predicates, and whether this can be generalized to the language-independent framework.

(5.26) succeeds(E) ←
act(E , drag_and_drop(Doc , Loc1 , Loc2)),
isa_document(Doc), isa_location(Loc1),
isa_location(Loc2), different_loc(Loc1 , Loc2),
holds_at(is_in(Doc , Loc1) , E),
holds_at(is_in_open_location(Doc) , E),
$\underline{\text{holds\_at(is\_open(Loc2) , E)}}$.
$J_s = \{1,2,3,4,5,6,7\}$

(5.27) succeeds(E) ←
act(E , drag_and_drop(Doc , Loc1 , Loc2)),
isa_document(Doc), isa_location(Loc1),
isa_folder(Loc2), different_loc(Loc1 , Loc2),
holds_at(is_in(Doc , Loc1) , E),
holds_at(is_in_open_location(Doc) , E),
$\underline{\text{holds\_at(is\_in\_open\_location(Loc2) , E)}}$.
$J_s = \{1,3,6,8\}$

(5.28) succeeds(E) ←
act(E , drag_and_drop(Doc , Loc1 , Loc2)),
isa_document(Doc), isa_location(Loc1),
$\underline{\text{isa\_desktop(Loc2)}}$, different_loc(Loc1 , Loc2),
holds_at(is_in(Doc , Loc1) , E),
holds_at(is_in_open_location(Doc) , E) .
$J_s = \{4,8\}$

(5.29) succeeds(E) ←
act(E , drag_and_drop(Doc , Loc1 , Loc2)),
isa_document(Doc), $\underline{\text{isa\_desktop(Loc1)}}$,
isa_location(Loc2), different_loc(Loc1 , Loc2),
holds_at(is_in(Doc , Loc1) , E) .
$J_s = \{3,5,7\}$

Program 5.6 Maximally general specializations of g_1

Overview of Definitions and constraints

- $\mathcal{L}_I$ and $\mathcal{L}_C$ are defined in Definition 5.4. $\mathcal{DL}_C$ is defined in Definition 4.13.
- R_I and R_C are defined by means of the intended interpretation (see Definition 5.5).
- The definition of *cover* for $\mathcal{L}_C$ and $\mathcal{DL}_C$ in Definition 5.11.
- The soundness of *cover* (Constraint 2.8) holds iff *cover* is defined w.r.t. the intended interpretation (see Section 5.4).
- Constraint 4.8 is fulfilled by proving the equivalent condition of Corollary 4.12 in Theorem 5.30.
- The definition of $\preccurlyeq_{\mathcal{B}}$ as an instantiation of Definition 5.27 with $P = \mathcal{B}$ corresponds to the definition of $\preccurlyeq$ (Definition 3.1).
- Concepts are represented by a unique representation by the equivalence relation $\equiv_{\mathcal{B}}$ (an instantiation of Definition 5.28 with $P = \mathcal{B}$).
- The Boundedness Constraint (Constraint 3.18) and Finiteness Constraint (Constraint 3.19) have to be fulfilled by choosing an appropriate language bias (see Section 5.8).

Chapter 6

Integrating Planning and Learning in an autonomous agent

6.1 Introduction

Motivation

This chapter discusses a particular integration of machine learning and AI-planning in a logical framework. We want to demonstrate that the integration using a logical representation is smooth. Because of this, more attention can be devoted to the problems arising specifically in the integrated architecture, such as the problem of controlling when to plan, when to learn, when to observe, when to execute actions, when to do experiments, etc. This chapter is also intended to illustrate the previous chapters: it shows how the framework of Iterative Versionspaces, instantiated in an ILP setting, can be used for machine learning. We have chosen planning to be integrated with learning, because planning is a typical AI-problem. From the point of view of Chapter 1, a planning system can only be considered intelligent, if it has some learning capabilities. Therefore there is a recent growing interest in integrating planning and learning (see, e.g., [Minton, 1993] for a collection of state of the art papers, and further [Veloso *et al.*, 1995], [Shen, 1993], [Benson and Nilsson, 1995], [Sablon and Bruynooghe, 1994]).

Planning Knowledge

Planning systems typically use three, and sometimes four, kinds of domain-dependent knowledge:

- knowledge about actions' preconditions and effects: for each of the available actions, the planner must know under which conditions the action will have particular effects;
- knowledge about the environment (or *external knowledge*): the planner must have knowledge about the objects in its environment and the relations between these objects;
- control knowledge: during the search for a plan the planner sometimes needs to choose among subgoals to explore, or needs to decide which action to use to fulfill a certain subgoal. Control knowledge contains some general rules which are to be

used as heuristics in making these choices. Control knowledge also contains general pruning rules, which are used to stop searching in unpromising directions;

- abstraction knowledge: some planners can work on different levels of abstraction. They make an abstract plan first, and then gradually refine the abstract plan until a working plan is found. Abstraction knowledge specifies how abstract actions are to be replaced by more concrete ones, and which extra constraints then have to be taken into account. The use of abstraction in planning is very important because it reduces the computational complexity of the search when searching on higher levels of abstraction, and mainly reduces planning to scheduling when refining an abstract plan to a workable plan.

Research has studied how to acquire each of these types of knowledge automatically. In this chapter we will focus on the acquisition of the first kind of knowledge. Acquisition of the second kind of knowledge (external knowledge) can be seen as an extension of our approach using a theory revision system, rather than learning concepts independently from each other (see Chapter 2, and see further). The acquisition of control knowledge has been extensively studied as an application of Explanation based learning in the context of PRODIGY [Minton *et al.*, 1989],[Etzioni, 1993a]. Also the acquisition of abstraction knowledge has been studied in PRODIGY [Knoblock, 1994].

Context

The problem of integrating planning and learning arises in the area of *autonomous agents*. We consider a planning agent which has to achieve some initial goals in a certain environment. In order to achieve the goals, the agent is able to *execute* actions in its environment. To achieve the goals the agent has to decide itself which actions to execute and in which order to execute them (i.e., the agent must make up a *plan*). Each of the actions will have a well determined effect in every situation (i.e., the environment is *deterministic*). However, in general the agent itself does neither know the exact effects, nor the conditions under which the effects will take place. Therefore the agent should *observe* the effects of its specific actions. From these observations it *learns* new knowledge, which explains all previously observed effects of actions, in order to avoid making planning mistakes in the future. These two abilities (*planning* and *learning*) are the major components of the agent and have to be coordinated with executing actions and observing effects. Coordination then consists of deciding when to plan, when to learn, when to execute and when to observe. The agent is autonomous in the sense that it makes these decisions itself.

This setup does not exclude the presence of a teacher, who provides the agent knowledge in whatever form. The advantage of having a teacher available is that the teacher can supply solution plans for specific goals for which no plan can be found by the planner. The disadvantage of relying on a teacher is that the agent becomes less autonomous.

This approach very well supports an active learning strategy. The learner does not only update its knowledge (w.r.t. observations of actions), but also proposes relevant experiments. By executing experiments relevant for a certain concept, and observing the effects, the learner obtains new information. If the generated experiment was relevant, this new information allows to exclude at least part of the possible hypotheses for the concept. In this way, the learner does not passively wait for failures of executed actions, but actively provokes successes and failures. As such it anticipates failures which could have occurred later. This strategy is also called *learning by experimentation.*

Representation

We will mainly concentrate on the architecture that *integrates* planning and learning. Less attention will be given to planning and learning methods themselves. The chosen representation for the integrated system is the *event calculus.* The event calculus in its original form was introduced by [Kowalski and Sergot, 1986]. It is a formalism to reason about time and change and can be described in Horn clause logic. Events initiate and terminate periods during which certain properties hold. [Denecker *et al.*, 1992] shows that the event calculus is a powerful tool for *temporal reasoning* in general, by solving some benchmark problems (such as e.g., the Russian Turkey Shooting problem). Temporal reasoning, in general, formalizes the notion of time. It allows to represent and reason with temporal knowledge. Representing and reasoning with temporal knowledge is typical for *planning.* The event calculus has proven to be a well suitable representation for planning. Planners using an event based framework are described in [Lansky, 1988], [Shanahan, 1989], [Denecker *et al.*, 1992], [Missiaen *et al.*, 1995]. In [Shanahan, 1989] and [Missiaen *et al.*, 1995], abduction is presented as a mechanism for planning in the event calculus. The abductive event calculus provides natural representations and planning capabilities that most of the systems based on other representations do not have, such as handling indirect and context dependent effects, and planning for multiple agents. Because the event calculus can be represented in Horn clause logic, learning can be done by *Inductive Logic Programming* (*ILP*) techniques (see Chapter 5). Consequently the learning behavior of the agent can be derived from the learning behavior of the chosen technique. An advantage of the modularity of the logical representation is that each type of knowledge can be acquired independently. The relevant *questions* about the concept to be learned (see Section 3.11) generated by *interactive* ILP systems, can be used as experiments. The observation of the results of an experiment can be interpreted as an answer to the corresponding question. Furthermore, because of the declarative character of the representation, the temporal knowledge as well as the non-temporal knowledge can easily be used by other problem solvers which work with first-order logic representations.

For all those reasons the event calculus turns out to be a natural representation for an architecture integrating *learning by experimentation, planning, and temporal reasoning.* Several other systems that integrate planning and learning use a STRIPS-like representation [Fikes and Nilsson, 1971]. In this representation each action has a list of preconditions, a list of effects that are initiated (the *add-list*), and a list of effects that are terminated (the *delete-list*). This representation is much less suitable for a straightforward application of well known machine learning techniques, and does not allow the integration of more general temporal reasoning. In the context of PRODIGY [Veloso *et al.*, 1995] the STRIPS representation was therefore substantially extended towards first order logic. In this extension inference rules are represented by means of add- and delete-lists as well.

Overview

This chapter is structured as follows: first we specify the problem of integration of planning and learning in the area of an autonomous agent (Section 6.2). Then we present the event calculus as a suitable representation for this integration (Section 6.3). In Section 6.4 we introduce the distinct components of the integrated system. Then the controlling algorithm for the integrated system is given (Section 6.5), and applied on the example of the autonomous tutor (Section 6.6). Finally, we discuss related work (Section 6.7) and conclude.

6.2 Problem specification

We suppose an autonomous agent is situated in a given environment. The agent can execute a given set of *actions* which have an effect on the domain under certain conditions. *Effects* are properties that are *initiated* by the action or properties that are *terminated* by it. *Preconditions* of an action specify the conditions under which the action will have an effect. However, the agent's knowledge of its own actions is incomplete and incorrect. This means that the agent does not know the exact preconditions nor the exact effects of its actions. We suppose the agent's knowledge about the environment itself (i.e., *external* to the agent) is correct and complete.

Given a goal and an initial situation to the agent, the agent can achieve the goal by executing actions. The agent can also observe the environment. We assume the agent *can* observe all aspects in its environment relevant for describing the preconditions and effects of its actions. We assume these observations are correct. We also assume there are no other agents executing actions which could influence the environment or the observations.

Both the assumption that the agent's external knowledge and its observations are correct, are simplifications for the learning task. Further on we argue that an extension towards learning of full domain knowledge amounts to the use of a theory revision system, instead of separate (and independent) learning processes for the distinct concepts. The assumption about correct observations could be relaxed by using a learner that can handle noise.

Whenever, during execution, the agent observes evidence that its knowledge is inconsistent with reality, it will try to adapt its knowledge. The resulting knowledge of the agent should then be consistent with all previously gathered evidence, in order to avoid similar mistakes in the future.

The main cycle of a system realizing this is 'plan, then execute, then observe, then learn' (see Figure 6.1). Starting from a given goal, the agent tries to find a plan for the goal using its current knowledge. Then it starts executing the plan. When the plan is executed (completely or partially), the agent observes the result of executing the actions, and learns from the observed results. An active learner would not only learn from inconsistencies arising during execution of a plan. It would also try to perform relevant experiments, i.e., it would try to execute the action causing the inconsistency in other situations as well, in order to gather more relevant information about the action. Experiments consist of a setting and an action to be executed in that setting. Note that this architecture also fits in the scheme of Figure 1.1.

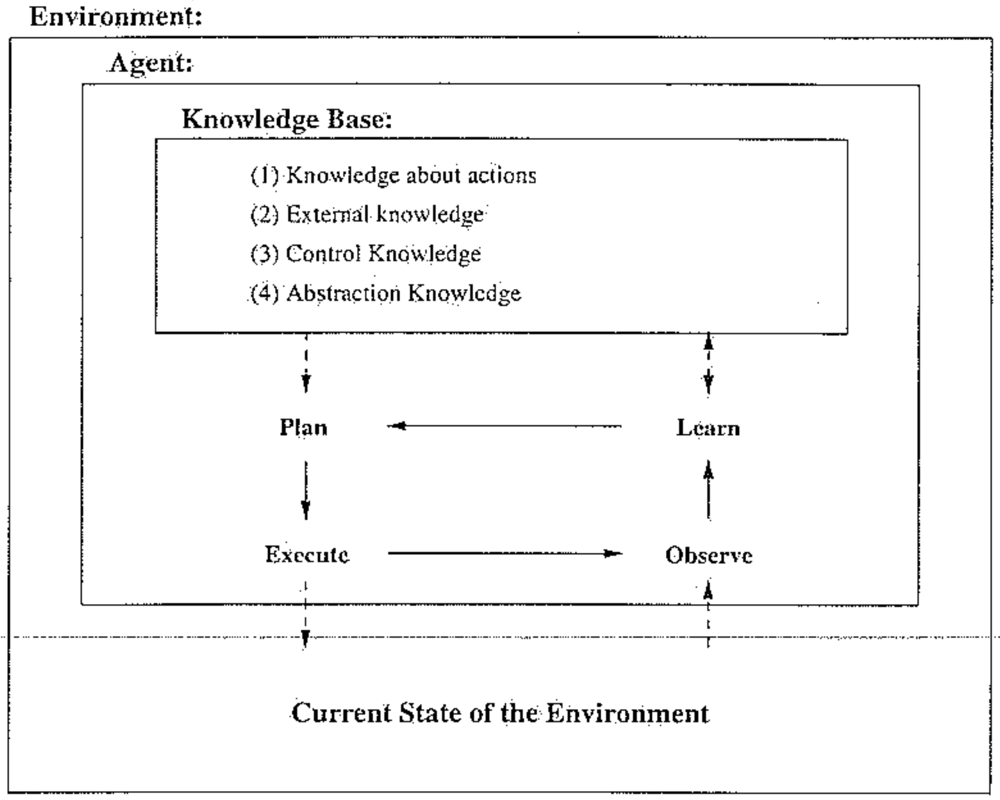


Figure 6.1 The architecture of the agent in its environment

6.3 The event calculus

In this section we present the event calculus in Horn clause logic as a suitable representation formalism for the integrated architecture. Events correspond to *timepoints* (without duration) that initiate and terminate certain (time dependent) properties of the domain. In the framework of the autonomous agent, exactly one action corresponds to each event. Because of their correspondence to time, events must be partially ordered in time.

The predicates that are used in the event calculus are *happens*/1, *time*/2, *act*/2, $\ll$ /2, *holds_at*/2, *initially*/1, *succeeds*/1, *initiates*/2 and *terminates*/2.

- The literal *happens*(E) expresses that event E happens. The literal *time*(E , T) expresses that the event E corresponds to timepoint T. The literal *act*(E , A) expresses that the event E corresponds to the action A. The literal $T_1 \ll T_2$ (to be read as "T_1 is before T_2") expresses that timepoint T_1 is before timepoint T_2. The literal $T_1 \lll T_2$ expresses that timepoint T_1 is before timepoint T_2, or equals T_2. In general, these three predicates are defined by facts expressing which actions happened in the past, which actions will happen in the future, and in which order they happened or will happen.

- *holds_at*(P , T) expresses that property P holds at time T. Clauses for *holds_at*/2 express temporal knowledge of the domain. E.g., Clause (6.1) expresses that an object *Obj* is in an open location if it is in a location *Loc* and *Loc* is open. Clause (6.2) expresses that an object *Obj* is not in a location *Loc*1 if it is in a location *Loc*2 that is different from *Loc*1.

```
(6.1)   holds_at( is_in_open_location( Obj ) , T ) ←
                isa_object( Obj ),
                holds_at( is_in( Obj , Loc ) , T ),
                holds_at( is_open( Loc ) , T ),
                isa_location( Loc ).
(6.2)   holds_at( is_not_in( Obj , Loc1 ) , T ) ←
                isa_object( Obj ), isa_location( Loc1 ), isa_location( Loc2 ),
                different_loc( Loc1 , Loc2 ),
                holds_at( is_in( Obj , Loc2 ) , T ).
```

- Apart from this temporal knowledge, there can be domain dependent predicates expressing non-temporal knowledge. These are defined by definite clauses. E.g., Clause (6.3) and Clause (6.4) express that directories and documents are objects. Fact (6.5) to Fact (6.7) express which objects are documents and which objects are directories.

```
(6.3)   isa_object( Fol ) ←
                isa_folder( Fol ).
(6.4)   isa_object( Doc ) ←
                isa_document( Doc ).
(6.5)   isa_folder( home ).
(6.6)   isa_folder( thesis ).
(6.7)   isa_document( 'thesis.tex' ).
```

- *succeeds*(E) expresses that the event E succeeds. An event succeeds if executing the corresponding action has an effect (i.e., some property not holding before E is initiated by E, or some property holding before E is terminated by E). The clauses for *succeeds*/1 in fact express the *preconditions* for an action at event E to succeed. E.g., Clause (6.8) expresses that an event E succeeds, if the event is a "double-click" action on an object *Obj*, which is closed and which is in an open location. Clause (6.9) expresses that an event E succeeds, if the event is an "open-parent" action on a folder *Fol* which is contained in a closed folder.

```
(6.8)   succeeds( E ) ←
                act( E , dbl_click( Obj ) ), time( E , T ),
                isa_object( Obj ),
                holds_at( is_closed( Obj ) , T ),
                holds_at( is_in_open_location( Obj ) , T ).
(6.9)   succeeds( E ) ←
                act( E , open_parent( Fol ) ), time( E , T ),
                isa_folder( Fol ),
                holds_at( is_in_closed_directory( Fol ) , T ).
```

initiates(E , P) expresses that the event E initiates the property P. E.g., Clause (6.10) expresses that the event E initiates the property *is_open*(Obj), if the action corresponding to E is *dbl_click*(Obj).

- (6.10) *initiates*(E , *is_open*(Obj)) $\leftarrow$
  act(E , *dbl_click*(Obj)),
  isa_object(Obj).

- *terminates*(E , P) expresses that the event E terminates the property P. Properties that are explicitly initiated or terminated by an event are called *primitive properties*. The other properties are called *derived properties*. E.g., the property *is_in_open_location*/1 (see Clause (6.1) and Clause (6.2)) is a derived property. This property is never explicitly initiated or terminated, it will hold only when the conditions of the body of Clause (6.1) or Clause (6.2) are true. Clauses for *initiates*/2 and *terminates*/2 are called *context dependent* if the body of the clause contains literals of the predicate *holds_at*/2. E.g., Clause (6.11) expresses that an event E terminates the property *is_closed*($Fol2$), if the action corresponding to E is *open_parent*($Fol1$), *and if* $Fol2$ is a folder such that $Fol1$ is in $Fol2$.

  (6.11) *terminates*(E , *is_closed*($Fol2$)) $\leftarrow$
  act(E , *open_parent*($Fol1$)), *time*(E , T),
  isa_folder($Fol1$),
  holds_at(*is_in*($Fol1$, $Fol2$) , T),
  isa_folder($Fol2$).

- *initially*(P) expresses that the property P is true in the initial situation.

  (6.12) *initially*(*is_closed*(*'thesis.tex'*)).
  (6.13) *initially*(*is_in*(*'thesis.tex'* , *thesis*)).
  (6.14) *initially*(*is_closed*(*thesis*)).
  (6.15) *initially*(*is_in*(*thesis* , *home*)).
  (6.16) *initially*(*is_open*(*home*)).

  Properties that are initially true are handled as if they were initiated by a special event *start* which happens, succeeds and which is before any other timepoint, i.e.,

  (6.17) *happens*(*start*).
  (6.18) *succeeds*(*start*).
  (6.19) *initiates*(*start* , P) $\leftarrow$ *initially*(P).
  (6.20) $start \ll X \leftarrow X \neq start$.

Together all definitions of each of these predicates form an *event calculus theory*.

The most important part of the event calculus is the specification of the *frame axiom*. The frame axiom allows to reason about how the domain is affected by events. In particular it allows to derive what holds at the time of a certain event from what holds true before that event. The frame axiom can be written in a domain-independent way, and must be part of every event calculus theory.

Clause (6.21) to Clause (6.23) formalize the frame axiom in the event calculus. The first clause expresses that a property P holds at timepoint T, if a successful event E happened before T which initiated P, and if P was not clipped between E and T. Clause (6.22) expresses when a property P is clipped between E and T. This is the case, if a successful event C happened *in between*[1] E and T, which terminated P.

(6.21) $holds\_at(\ P\ ,\ T\ ) \leftarrow$
$\quad happens(\ E\ ),$
$\quad initiates(\ E\ ,\ P\ ),$
$\quad succeeds(\ E\ ),$
$\quad time(\ E\ ,\ T_E\ ),$
$\quad T_E \ll T,$
$\quad \text{not } clipped(\ T_E\ ,\ P\ ,\ T\ ).$

(6.22) $clipped(\ T_E\ ,\ P\ ,\ T\ ) \leftarrow$
$\quad happens(\ C\ ),$
$\quad terminates(\ C\ ,\ P\ ),$
$\quad succeeds(\ C\ ),$
$\quad time(\ C\ ,\ T_C\ ),$
$\quad in(\ T_C\ ,\ T_E\ ,\ T\ ).$

(6.23) $in(\ T_C\ ,\ T_E\ ,\ T\ ) \leftarrow\ T_E \lll T_C,\ T_C \ll T.$

The time constraints are formulated in Clause (6.24) and Clause (6.25). Clause (6.24) expresses the transitivity of time order. Clause (6.25) expresses that the time order is anti-symmetric and anti-reflexive.

(6.24) $T_1 \ll T_3 \leftarrow T_1 \ll T_2\ ,\ T_2 \ll T_3.$
(6.25) $\leftarrow T_1 \ll T_2\ ,\ T_2 \ll T_1.$

Clause (6.26) expresses the constraint that to each event corresponds exactly one action.

(6.26) $\leftarrow act(\ E\ ,\ A_1\ ),\ act(\ E\ ,\ A_2\ ),\ A_1 \neq A_2.$

The event calculus as such allows clauses containing any of the temporal predicates with any timepoint as temporal arguments. This means it can express relations between distinct timepoints. However, in our application we limit ourselves to clauses for *holds_at*/2, *succeeds*/1, *initiates*/2 and *terminates*/2 which refer only to the timepoint occurring in the head. Also, since there is only one event per timepoint, we do not explicitly write the *time*/2-relation, but identify each event with the timepoint associated to it. More specifically the clauses we use have the following form:

- $holds\_at(\ E\ ,\ P\ ) \leftarrow h_1\ ,\ \ldots,\ h_{n_1}, l_1\ ,\ \ldots,\ l_{m_1}.$
- $succeeds(\ E\ ) \leftarrow h_1\ ,\ \ldots,\ h_{n_2}, l_1\ ,\ \ldots,\ l_{m_2}.$
- $initiates(\ E\ ,\ P\ ) \leftarrow act(\ E\ ,\ a\ ), h_1\ ,\ \ldots,\ h_{n_3}, l_1\ ,\ \ldots,\ l_{m_3}.$
- $terminates(\ E\ ,\ P\ ) \leftarrow act(\ E\ ,\ a\ ), h_1\ ,\ \ldots,\ h_{n_4}, l_1\ ,\ \ldots,\ l_{m_4}.$

[1]This representation uses *half open* time intervals, to avoid problems with properties P for which there exists a negated property $\overline{P}$. With open intervals there could be *truth gaps*, where neither P nor $\overline{P}$ hold; with closed intervals there could be inconsistencies, because both P and $\overline{P}$ hold at the same timepoint (see [Missiaen, 1991]).

The literals h_j are temporal literals. They are of the form *holds_at*(P_j , E), i.e., they have the same time argument as the head of the clause. The literals l_k are non-temporal literals.

In this chapter we use the basical event calculus representation, which neither deals with actions with a duration, nor with effects with a duration. These extensions of the event calculus are studied in [Shanahan, 1990], [Van Belleghem *et al.*, 1994], [Missiaen, 1994]. Some related extensions handle time granularity [Evans, 1990] and [Montanari *et al.*, 1992]. For more details about the event calculus as we use it here, we refer to [Missiaen, 1991] and [Missiaen *et al.*, 1995].

6.4 Components of the architecture

In this section we fill in the components of the system by concrete systems.

6.4.1 Planning

In the event calculus a plan is a set P of facts for the predicates *happens*/1, *act*/2 and $\ll$ /2.

Definition 6.1 (A Plan in the event calculus) Given an event calculus theory T, a ground planning goal G, and a proof procedure $\vdash$, a plan P is a solution for G iff

- $T \cup P \vdash G$, and
- $\forall E_i$: *happens*(E_i) $\in P$ implies $T \cup P \vdash$ *succeeds*(E_i).

Given a goal G, the aim of planning is to find a solution plan P for G.

The planning system we will use is CHICA [Missiaen, 1991], [Missiaen *et al.*, 1995]. The proof procedure CHICA uses for $\vdash$ is SLDNF. However, the planning system must not only prove that P is a plan, it must also find P. Therefore SLDNF was extended to SLDNFA [Denecker *et al.*, 1992], [Denecker, 1993]. Basically SLDNFA is SLDNF extended with a mechanism to *abduce* the incompletely defined predicates *happens*/1, *act*/2 and $\ll$ /2 (called *abducibles*). The idea is the following. Suppose the goal G is *holds_at*(p , t_{end}). If the abducibles are completely unknown, it is, in general, impossible to infer the literal *holds_at*(p , t_{end}) with SLDNF. Every clause for *initiates*/2, *terminates*/2 and *succeeds*/1 contains, for instance, a literal of the predicate *act*/2, which can never be proven. However, if these literals are abduced in a way consistent with the constraints of Clause (6.24) to Clause (6.26), then it may be possible to prove *holds_at*(p , t_{end}). Given a goal G, SLDNF on the literals of the defined predicates, and consistent abduction on the literals of the abducibles, results in a set of abduced literals containing the events that have to happen, the associated actions, and the order of the events, in order to prove G. If this set of abduced literals fulfills the conditions of Definition 6.1, it is a solution plan.

CHICA is a planning system using SLDNFA as an inference mechanism. In [Missiaen, 1991] completeness and soundness of CHICA are proven. For more details on the SLDNFA procedure we refer to [Denecker *et al.*, 1992]. [Van Belleghem *et al.*, 1994] illustrates that abduction can potentially be used for planning in the event calculus extended with continuous changes as well.

Example

Figure 6.2 shows an SLDNFA derivation to illustrate the SLDNFA procedure. The set Δp collects the events that happen, and the corresponding actions. The set Δt collects the time relations between these events, and Δc collects constraints that have to be checked each time new facts are abduced. Also the domain-independent constraints Clause (6.24), Clause (6.25) and Clause (6.26) have to be checked when abducing new facts. Suppose the predicates *happens*/1, *act*/2 and $\ll$ /2 do not have a definition (except in the special case of *start*: Fact (6.17) and Clause (6.20)). Goals for these predicates can either be abduced, or resolved with an already abduced fact from Δp or Δt. The goal G is *holds_at*(*is_open*(*'thesis.tex'*) , t_{end}). In the derivation a goal which is resolved in the next step, is underlined. The goal G is resolved with Clause (6.21). In the following node, the goal *happens*(E) can be resolved with Fact (6.17). This would give branch (A), which eventually fails, because *start* does not initiate *is_open*(*'thesis.tex'*). Therefore branch (A) is not included in Figure 6.2. The branch that is shown is the one that abduces *happens*(e_1) (i.e., *happens*(e_1) is added to Δp). The constant e_1 is new, in the sense that it does not occur elsewhere in the event calculus theory. In the next step, $e_1 \ll t_{end}$ is abduced, i.e., added to Δt, and *initiates*(E , *is_open*(*'thesis.tex'*)) is resolved with Clause (6.10). The figure shows another branch (B), to indicate that there might be other clauses to resolve with. These other choices can be considered on backtracking. In the next step, three things happen:

- *act*(e_1 , *dbl_click*(*'thesis.tex'*)) is abduced, i.e., added to Δp;
- *isa_object*(*'thesis.tex'*) is resolved with Clause (6.4); and,
- not *clipped*(e_1 , *is_open*(*'thesis.tex'*) , t_{end}) is checked (this is a branch that is not shown in the figure), and added to Δc.

Each time new facts are abduced, an element in Δc might be violated, so all elements of Δc have to be checked. Then *isa_document*(*'thesis.tex'*) resolves with Fact (6.7), and *succeeds*(e_1) resolves with Clause (6.8). Actually *succeeds*(e_1) can resolve with any clause for *succeeds*/1, but all these other branches fail if they contain a literal for *act*/2 other than *act*(E , *dbl_click*(*Obj*)) (this is because this literal can neither be resolved with a fact in Δp, nor abduced, since Δp already contains an action associated to e_1). Consequently, the literal *act*(e_1 , *dbl_click*(*Obj*)) resolves with the fact *act*(e_1 , *dbl_click*(*'thesis.tex'*)) from Δp, and *isa_object*(*Obj*) resolves with Clause (6.4). Then the goal *isa_document*(*'thesis.tex'*) is resolved with Fact (6.7). The property *is_closed*/2 is primitive, so the literal *holds_at*(*is_closed*(*'thesis.tex'*) , e_1) only resolves with the frame axiom (Clause (6.21)). In the next step, *happens*(E) resolves with Fact (6.17). Another possibility would be to abduce *happens*(e_2). The latter branch is indicated by (C).

The process continues in a similar way, until the empty clause is derived. In that case the goal *holds_at*(*is_open*(*'thesis.tex'*) , t_{end}) can be proven from the event calculus theory, Δp and Δt. The sets

$$\Delta p = \{\ happens(\ e_1\ )\ ,\ act(\ e_1\ ,\ dbl\_click(\ 'thesis.tex'\ )\ )\ \}$$

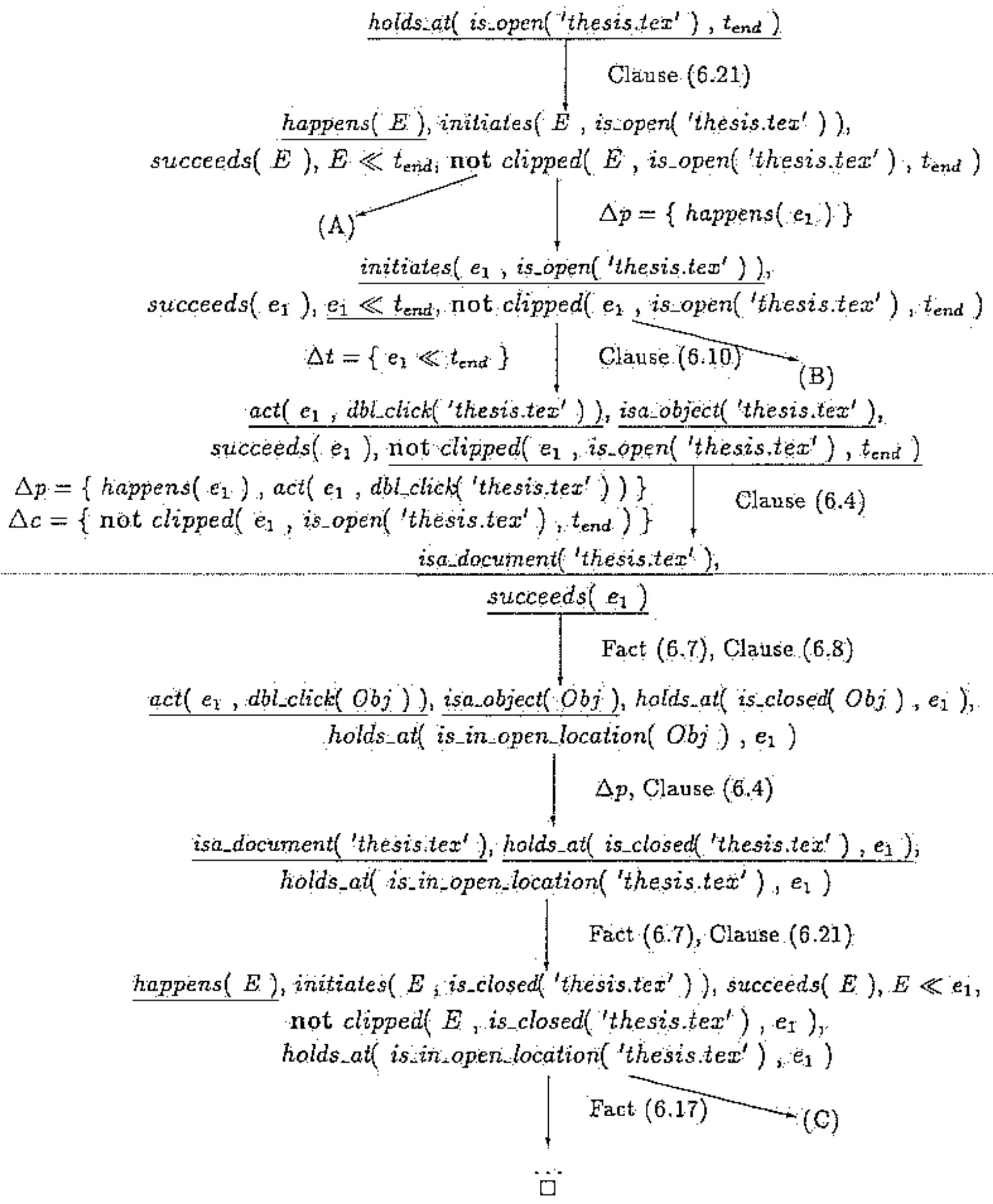

holds_at(is_open('thesis.tex') , t_end)
Clause (6.21)
happens(E), initiates(E , is_open('thesis.tex')),
succeeds(E), E ≪ t_end, not clipped(E , is_open('thesis.tex') , t_end)
(A)
Δp = { happens(e1) }
initiates(e1 , is_open('thesis.tex')),
succeeds(e1), e1 ≪ t_end, not clipped(e1 , is_open('thesis.tex') , t_end)
Δt = { e1 ≪ t_end }
Clause (6.10)
(B)
act(e1 , dbl_click('thesis.tex')), isa_object('thesis.tex'),
succeeds(e1), not clipped(e1 , is_open('thesis.tex') , t_end)
Δp = { happens(e1) , act(e1 , dbl_click('thesis.tex')) }
Δc = { not clipped(e1 , is_open('thesis.tex') , t_end) }
Clause (6.4)
isa_document('thesis.tex'),
succeeds(e1)
Fact (6.7), Clause (6.8)
act(e1 , dbl_click(Obj)), isa_object(Obj), holds_at(is_closed(Obj) , e1),
holds_at(is_in_open_location(Obj) , e1)
Δp, Clause (6.4)
isa_document('thesis.tex'), holds_at(is_closed('thesis.tex') , e1),
holds_at(is_in_open_location('thesis.tex') , e1)
Fact (6.7), Clause (6.21)
happens(E), initiates(E , is_closed('thesis.tex')), succeeds(E), E ≪ e1,
not clipped(E , is_closed('thesis.tex') , e1),
holds_at(is_in_open_location('thesis.tex') , e1)
Fact (6.17)
(C)
...
□

Figure 6.2 Example of a SLDNFA derivation

and

$$\Delta t = \{ e_1 \ll t_{end} \}$$

constitute the plan.

6.4.2 Executing and Observing

After executing a particular action a associated to event e in its environment, the agent can check whether e "really" succeeds, by observing all changes in the environment. Only primitive properties are observed. All observations are added to the background knowledge. Because we assume there are no other agents that influence the effect of e, each observed change must be ascribed to e. If there are no changes, *succeeds*(e) is a negative instance to be provided to the learner. If there are some changes, *succeeds*(e) is provided as a positive instance to the learner. In the latter case, a property p that did not hold before e, but did hold after e, is initiated by e, i.e., *initiates*(e , p) is a positive instance. If a property p, did hold before e, but not after e, it is terminated by e, i.e., *terminates*(e , p) is a positive instance. By assuming that all changes can be observed, all instances *initiates*(e , p) and *terminates*(e , p), such that p was neither initiated, nor terminated by e, are negative instances. This assumption amounts to applying the closed world assumption on the set of all ground atoms *holds_at*(p , e), with p a primitive property. This means that negative instances of *initiates*/2 and *terminates*/2 are only implicitly known. The negative instances that are covered by the current definitions of *initiates*/2 and *terminates*/2, are particularly useful to the learner, because these are inconsistent. This means that these effects were expected, but did not happen. Similarly, positive uncovered instances are also inconsistent with the current definitions of *initiates*/2 and *terminates*/2; these are unexpected effects that did happen. Positive covered instances are consistent with the current definitions.

6.4.3 Learning

After observing the effects of an event e, the learner is provided with positive and negative instances for *succeeds*/1, *initiates*/2 and *terminates*/2. The positive and negative instances are *ground facts*; they can be used as lowerbounds by an ILP learner. Since the definitions for *succeeds*/1, *initiates*/2 and *terminates*/2 should relate the effects of e to the properties that hold at timepoint e, and not to other timepoints, the only part of the the background knowledge that is to be considered when constructing starting clauses (see Section 5.9) consist of

- all primitive properties that hold at timepoint e,
- the clauses defining the derived properties (see Clause (6.1) to Clause (6.2)), and
- the non-temporal clauses(see Clause (6.3) to Fact (6.7)).

Because we assume all these to be correct, the background knowledge is correct. The only reason for a positive instance *succeeds*(e_1) to be uncovered is thus that the definition for *succeeds*/1 is overly specific. Dually, the reason for a covered negative instance is that the corresponding definition is overly general. Suppose, for instance, that the plan found by the SLDNFA derivation of Figure 6.2 is executed. The plan consists of one action *dbl_click*(*'thesis.tex'*), associated to event e_1. Suppose that e_1 does not succeed, although it was expected to (because in the fifth node from the top *succeeds*(e_1) is proven by means of Clause (6.8) from the literals *act*(e_1 , *dbl_click*(*Obj*)), *isa_object*(*Obj*), *holds_at*(*is_closed*(*Obj*) , e_1) and *holds_at*(*is_in_open_location*(*Obj*) , e_1), with the substitution { *Obj*/*'thesis.tex'* }). The only reason why this negative example is covered is that Clause (6.8) is overly general, because the action *dbl_click*(*Obj*) is certainly associated to e_1, and the other literals can be proven using definitions that are assumed to be correct. Moreover, the definitions for *succeeds*/1, *initiates*/2 and *terminates*/2 are not allowed to be recursive. This means that learning a definition for *succeeds*/1, learning a definition for *initiates*/2 and learning a definition for *terminates*/2 are three separate single predicate learning problems without recursion.

Basically any incremental ILP system that can learn the kind of clauses that are allowed in definitions for *succeeds*/1, *initiates*/2 and *terminates*/2, can be used. In [Sablon and Bruynooghe, 1994] we used CLINT [De Raedt, 1992]. CLINT is able to learn the disjunctive concept definitions of the event calculus, provides a nice framework for handling bias in ILP, and has a sound behavior. Furthermore CLINT is interactive: it proposes experiments and learns from the results of their execution. The disadvantages of CLINT (as in some other suitable ILP systems) arise from the fact that it works specific-to-general only. When using maximally specific concept representations, the result is a very cautious agent, which only executes actions in restricted situations in order not to make errors of commission (see Section 3.4). To avoid that the agent would almost never find a plan because its concept definitions are too specific, a general-to-specific approach would be more appropriate in this case: in the beginning the agent applies the actions in several situations in which they do not succeed, or do not have the desired effects. With these observations the agent then specializes its concepts in order to avoid similar mistakes in the future. CLINT has to learn maximally specific definitions for the *negations* of *succeeds*/1, *initiates*/2 and *terminates*/2, and negate these definitions to realize the same behavior as a maximally general concept representation. Furthermore, although CLINT is interactive, the experiments it generates are not always relevant in the sense of Definition 3.59. A bi-directional approach allows to generate relevant experiments (see Section 3.11 and Section 5.11).

We propose to use the framework of Iterative Versionspaces, instantiated to Inductive Logic Programming, as described in Chapter 5. This framework can also learn disjunctions, it is sound, it provides a useful framework for *relevant* instance generation by defining middle points, and offers the choice between using maximally general or maximally specific concept representations.

Under the assumption of having full observations, programs deriving constraints that hold in the background knowledge could also be used to find these definitions. Examples of such systems are CLAUDIEN [De Raedt and Bruynooghe, 1993] and ICL [De Raedt and Van Laer, 1995].

In this context we briefly describe why, in general, the learning module must be a theory revision system, when the external knowledge can be incorrect. Temporal clauses defining derived properties as well as non-temporal knowledge might be incorrect. For a covered negative instance, at least one of the clauses which are used to prove that the instance is covered, is to blame for the inconsistency, and is to be updated. In the above example where the execution of the action *dbl_click*(*'thesis.tex'*) does not succeed, the fact that *succeeds*(e_1) is covered by Clause (6.8) might also be caused by an incorrect definition of *isa_object*/1 or *holds_at*/2. Therefore, the theory revision system should modify the event calculus theory as a whole (except for the domain-independent part and the observations), in order to be globally consistent with the observations. Theory revision theoretically also provides a solution when the problem is extended towards multiple agents, such that more than one event per timepoint is possible.

Similarly as for the external knowledge, the observations of the agent might not be correct. In that case, the learning system, or theory revision system, will have to deal with *noise* (see Chapter 2) in order to find any workable concept definitions.

6.4.4 Experimentation

For each of the learned concept definitions the learner can generate relevant lowerbounds (see Section 5.11). These can be interpreted as experiments in the following way. Suppose c is a relevant lowerbound for the concept "a successful execution of action a", as defined in Section 5.11. The clause c is of the form $succeeds(\ E\ ) \leftarrow act(\ E\ ,\ a\ ), c_1, c_2(\ E\ )$, where c_1 is a conjunction of non-temporal literals, and c_2 is a conjunction of temporal literals. If c covers the representation of an instance i that belongs to the concept, c is a positive lowerbound. If c covers the representation of an instance i that does not belong to the concept, c is a negative lowerbound. This means that in order to evaluate whether c is a positive or negative lowerbound, we have to find an event e, for which

- the associated action is a,
- the temporal and non-temporal literals of c instantiated to e (i.e., c_1, $c_2(\ e\ )$) are true, and
- we know whether the execution of a at timepoint e was successful.

To find such an instance is what we call experimentation. Experimentation consists of two phases:

- first set up the experiment (i.e., find, or "create", a timepoint e such that $c_1, c_2(\ e\ )$ is true;
- then execute action a at timepoint e.

This means that an experiment consists of two components: an experiment setting $S = (\ c_1, c_2(\ E\ )\ )$ and an action a. The experiment setting defines the conditions in which the action a must be executed.

To set up the experiment, the experiment setting S is considered as a goal for the planner. When the goal S is established, the action a can be executed. Observing whether or not the execution of a is successful determines whether the event belongs to the concept, and thus whether or not c is a positive or a negative lowerbound.

Similarly, relevant lowerbounds for concepts of the form "a successful execution of a initiates property p", or of the form "a successful execution of a terminates property p" can be interpreted as experiments.

6.4.5 Simulating the environment

In our implementation of the above architecture, the environment of the agent will be *simulated* for practical reasons. This means that the robot interacts with a software model of the environment, instead of with the environment itself. The software model can also be implemented using the event calculus. However, *the simulator's knowledge is strictly separated from the agent's knowledge.* The agent can only interact with the simulator by

- executing an action: the agent specifies to the simulator which action it wants to execute. The simulator then changes the environment *according to its own model*;
- observing the environment: the agent asks the simulator the current state of the environment. The simulator then returns the set of relations that hold in the current state of the environment.

Actually the use of a simulator implicitly solves the problem of interpreting the observations, i.e., of identifying each object each time the environment is observed, and to find out which relations hold between these observed objects. As we will only concentrate on the learning and planning aspects of the agent and their integration, we assume that in general this problem is solved in an interfacing module (belonging to the agent) between the agent's knowledge base and its execution and observation devices. This interfacing module is in most domains non-trivial, as in robotics it will need techniques to interprete images (*computer vision*), sounds (e.g., *speech recognition*) or even other sensor information. If the agent is able to handle different levels of abstraction, the learning module (which is the one processing observations) should be able to specify a level of abstraction, and the interfacing module should also be able to interpret the observed information on the given abstraction level.

6.5 The resulting architecture

Suppose the agent is in an initial situation S_{init} and has a goal G to achieve. A goal is represented by a conjunction of literals. An experiment $exp(S , A)$ consists of an experiment setting S (represented as a goal) and an action A. Performing an experiment consists of applying A when the goal specified by S holds. In the algorithm an instance is called an inconsistency if it is an uncovered positive instance, or a covered negative instance.

The agent's algorithm to achieve a goal G from an initial current situation S_{curr} can then be described as follows:

1. **Plan:**
   Make a plan P for G from situation S_{curr}.
   If no plan can be found, then fail.

2. **Execute and Observe:**
   Let the set Inf initially be empty.
   As long as P is not empty and there are no inconsistencies between the observations and the current knowledge:

- **Execute:**
  Take the first action from P, and is execute it. If the execution of P has any effects, this gives a change in S_{curr}.
- **Observe:**
  Observe the effects of executing the action: if there *are* any effects, this event is a positive instance for *succeeds*/1, otherwise it is a negative one. Each property that is initiated, resp. terminated, gives a positive instance for *initiates*/2, resp. *terminates*/2. Effects that were expected but are not observed result in negative instances for *initiates*/2 or *terminates*/2. Collect all instances in Inf.

3. **Learn:**
   - For all instances in Inf, update the corresponding definition, such that it is consistent with all known instances.
   - As long as the learner generates relevant experiments $exp(\ S\ ,\ A\ )$:
     - **Goal:**
       Call this algorithm recursively with goal S and initial situation S_{curr}. The aim is to let the planner make a plan to achieve S, and to execute that plan. During the execution new instances are gathered, and processed by the learner. It is also possible that some experiments are performed. However, it is not allowed to do any experiments for the action A during the recursive call, in order to avoid infinite loops.
       Note that executing actions in the recursive call could lead to a new S_{curr}.
     - **Experiment and Observe:**
       If the recursive call does not fail (i.e., the goal S is achieved):
       * **Experiment:**
         execute A (which can again change S_{curr})
       * **Observe:**
         Collect positive and negative instances as above.
       * **Learn from experiment:**
         The collected positive and negative instances are used to update the corresponding definitions.

4. **Stop or Retry:**
   If G is achieved, then announce success.
   Otherwise, call this algorithm recursively with goal G and initial situation S_{curr}. If the recursive call succeeds, then announce success; otherwise fail.

If the algorithm fails, the planner is unable to make a plan for the given goal G. The planner will not be able to make a plan, if its definitions for the preconditions and effects are overly specific, i.e., they restrict the set of situations in which the action is expected to succeed, or to have a certain effect, too much. In that case, the only way to get out of this deadlock autonomously, is to execute some heuristically generated actions (or, if no better solution is available, execute some randomly generated actions), and observe the effects.

A semi-autonomous agent would in this case call the help of a teacher. The teacher could give the agent a full solution plan, or only some actions to get out of the impasse. By executing this plan or these actions, the learner is provided with new instances, which then lead to an update of the agent's knowledge. Typically this kind of systems heavily rely on the teacher in the beginning, but as they obtain more good example actions to execute, gradually get better in achieving their goals autonomously (see e.g., [Benson, 1995]).

Each time the algorithm is called recursively in Step 4, a new plan for G is made. Some planning systems allow the possibility for *plan repair*, when something goes wrong in the execution of a plan (see e.g., [Wilkins, 1988]). Suppose that after a plan has been made and partially executed, the preconditions of the next action in the plan are not fulfilled. Basically, plan repair would then adapt the remaining part of the plan such that the preconditions of all remaining actions are again fulfilled. In this way repairing the rest of the plan avoids recomputing a completely new plan. [Missiaen, 1991] informally describes conditions and a method to extend CHICA for plan repair.

6.6 Example

In this section we present an example session with the algorithm of Section 6.5, in which the autonomous tutor of Section 1.3 learns concepts about a graphical user interface. The primitive properties in the example are *is_in*/2, *is_open*/1 and *not_is_open*/1. The derived properties are *is_not_in*/2, *is_in_open_location*/1, *is_not_in_open_location*/1. The definitions of the derived properties belong to the background knowledge and are presented in Program 5.3. Apart from Program 5.3 the background knowledge also consists of the definitions of Program 5.2.

The actions the tutor can perform in the example are *dbl_click*(*Obj*) to open an object *Obj*, *menu_close*(*Obj*) to close an object *Obj*, and *drag_and_drop*(*Doc* , *Loc*1 , *Loc*2) to drag a document *Obj* from location *Loc*1 to another location *Loc*2.

We suppose the clauses for *succeeds*/1, *initiates*/2 and *terminates*/2 of the actions *dbl_click*(*Obj*) and *menu_close*(*Obj*) are known. These are provided initially as *positive upperbounds* to the learner. This means the tutor will always correctly apply these actions during planning. During execution of these plans, the correct applications of these definitions gives positive lowerbounds of *succeeds*/1, *initiates*/2 and *terminates*/2, from which an almost maximally specific definition for each of the predicates and for each of the actions is derived. We concentrate on the action of dragging a document from one folder to another folder. We show how the algorithm of Section 6.5 applies to this case. The initial definitions for *succeeds*/1, *initiates*/2 and *terminates*/2 of a drag-action are given in Program 6.1. They are provided to the learner as positive upperbounds. For planning we will always use the maximally general definitions returned by the learner (see Section 6.4.3). In Section 5.12 we have illustrated how the concept "successfully dragging a document D from folder F_1 to another folder F_2" was learned by DITVS, instantiated to ILP. The information elements used in Section 6.4.3 were taken from the example in this session. This allows to view both the ILP aspect and the integration aspect of the same example session. As in Section 5.12 we do not present the saturated clauses.

The initial situation is shown in Figure 6.3. Apart from the folders and documents shown, the folder *mail* contains a document *defence.mbox*, and the folder *texfiles* contains the file *'thesis.dvi'*. All folders and documents, except *home* and *thesis*, are not open. There is only one desktop, called 'desktop'.

```
(6.27) succeeds( E ) ←
            act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
            isa_document( Doc ), isa_location( Loc1 ), isa_location( Loc2 ),
            different_loc( Loc1 , Loc2 ), holds_at( is_in( Doc , Loc1 ) , E ) .
(6.28) initiates( E , is_in( Doc , Loc2 ) ) ←
            act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
            isa_document( Doc ), isa_location( Loc1 ), isa_location( Loc2 ),
            different_loc( Loc1 , Loc2 ), holds_at( is_in( Doc , Loc1 ) , E ) .
(6.29) terminates( E , is_in( Doc , Loc1 ) ) ←
            act( E , drag_and_drop( Doc , Loc1 , Loc2 ) ),
            isa_document( Doc ), isa_location( Loc1 ), isa_location( Loc2 ),
            different_loc( Loc1 , Loc2 ), holds_at( is_in( Doc , Loc1 ) , E ) .
```

Program 6.1 Initial definitions

- **a.1. Plan:** suppose a first goal for the tutor to achieve is to put the mailbox *'thesis.mbox'* in the folder *thesis*, i.e., G is

  $$holds\_at(\ is\_in(\ 'thesis.mbox'\ ,\ thesis\ )\ ,\ t_{end}\ ).$$

  The planner returns a plan for this goal:

  $$act(\ e305\ ,\ drag\_and\_drop(\ 'thesis.mbox'\ ,\ home\ ,\ thesis\ )\ ).$$

- **a.2. Execute and observe:** This plan is executed, and the effects are observed. The real effects are exactly the ones expected, i.e., there are only positive lowerbounds.

- **a.3. Learn:**

  – Each of these positive lowerbounds is consistent with the current maximally general definitions, but not with the current almost maximally specific definitions. Therefore the maximally specific definitions are generalized using these positive lowerbounds. For *succeeds*/1, clause Clause (5.17) is induced.

  – A first experiment is generated: the setting of the experiment consists of the body of Clause (5.18), i.e., G_1 is

  $$\begin{array}{l} isa\_document(\ Doc\ ),\ isa\_location(\ Loc1\ ),\\ isa\_location(\ Loc2\ ),\ different\_loc(\ Loc1\ ,\ Loc2\ ),\\ holds\_at(\ is\_in(\ Doc\ ,\ Loc1\ )\ ,\ E\ ),\\ holds\_at(\ is\_not\_in(\ Loc2\ ,\ Loc1\ )\ ,\ E\ )\ . \end{array}$$

  The action to be executed at timepoint E is

  $$A_1 = drag\_and\_drop(\ Doc\ ,\ Loc1\ ,\ Loc2\ ).$$

  * **Goal:** the planner makes a plan for G_1 by calling the algorithm recursively. Actually, this goal is already fulfilled with the substitution { $E/e317$, *Doc/'thesis.tex'* , *Loc1/thesis* , *Loc2/home* }. Since there are no actions to be executed, no inconsistencies are found, and no experiments will be done in the recursive call.

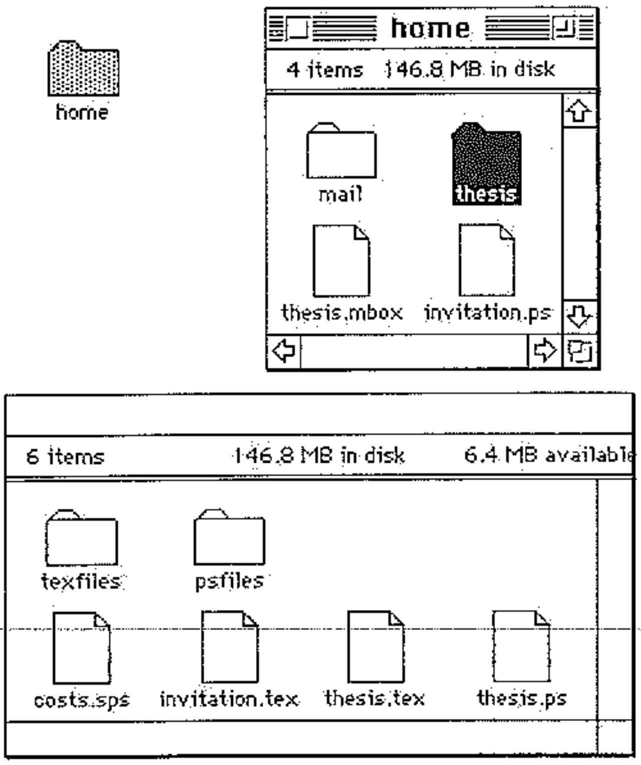


Figure 6.3 Initial situation

* **Experiment and Observe:** Consequently, the action A_1, instantiated to

$$drag\_and\_drop(\ 'thesis.tex'\ ,\ thesis\ ,\ home\ ),$$

can immediately be executed.

- **Experiment:** A_1, corresponding to event e317, is executed.
- **Observe:** Again the real effects are exactly the ones expected, i.e., there are only positive lowerbounds.
- **Learn from experiment:** Consequently the maximally specific definitions are updated. For *succeeds*/1, clause Clause (5.23) is induced.

– A second experiment is generated: the setting of the experiment consists of the body of Clause (5.19), i.e., G_2 is

$$\begin{array}{l} isa\_document(\ Doc\ ),\ isa\_location(\ Loc1\ ),\\ isa\_location(\ Loc2\ ),\ different\_loc(\ Loc1\ ,\ Loc2\ ),\\ holds\_at(\ is\_in(\ Doc\ ,\ Loc1\ )\ ,\ E\ ),\\ holds\_at(\ is\_not\_in\_open\_location(\ Doc\ )\ ,\ E\ ). \end{array}$$

The action to be executed at timepoint E is

$$A_2 = drag\_and\_drop(\ Doc\ ,\ Loc1\ ,\ Loc2\ ).$$

- * **Goal:** the planner makes a plan for G_2 by calling the algorithm recursively. Again, this goal is already fulfilled, in this case with the substitution { $E/e324$, $Doc/$'*thesis.dvi*' , $Loc1/texfiles$, $Loc2/home$ }. Again there are no actions to be executed, no inconsistencies found, and no experiments to be done in the recursive call.
- * **Experiment and Observe:** Consequently, the action of the experiment can immediately be executed.
  - · **Experiment:** the action

    drag_and_drop('*thesis.dvi*' , *texfiles* , *home*),

    corresponding to event $e324$, is executed.
  - · **Observe:** In this case the action does not succeed. Since it was intended to succeed,
    succeeds(e324) is a negative lowerbound. Furthermore, this action was expected to initiate the property *is_in*('*thesis.dvi*' , *home*), and to terminate the property *is_in*('*thesis.dvi*' , *texfiles*). Consequently,

    initiates(e324 , *is_in*('*thesis.dvi*' , *home*)) and

    terminates(e324 , *is_in*('*thesis.dvi*' , *home*))

    are also negative lowerbounds.
  - · **Learn from experiment:** Consequently the maximally general definitions of Program 6.1 are updated; for *succeeds*/1 this resulted in Clause (5.24). Similar clauses are derived for the predicates *initiates*/2 and *terminates*/2.

– A third experiment is generated: the setting of the experiment consists of the body of Clause (5.21), i.e., G_3 is

isa_document(*Doc*), *isa_location*(*Loc*1),
isa_folder(*Loc*2), *different_loc*(*Loc*1 , *Loc*2),
holds_at(*is_in*(*Doc* , *Loc*1) , *E*),
holds_at(*is_in_open_location*(*Doc*) , *E*),
holds_at(*is_not_in_open_location*(*Loc*2) , *E*) .

The action to be executed at timepoint E is

$$A_3 = drag\_and\_drop(\ Doc\ ,\ Loc1\ ,\ Loc2\ ).$$

- * **Goal:** the planner makes a plan for G_3 by calling the algorithm recursively. This time, the goal is not fulfilled immediately.
  - · **b.1 Plan:** The plan for achieving goal G_3 is

    act(e328 , *menu_close*(*home*)),
    act(e337 , *drag_and_drop*('*invitation.tex*' , *thesis* , *desktop*)).
  - · **b.2 Execute and Observe:** Executing the action *menu_close*(*home*) only results in expected effects. Therefore

    drag_and_drop('*invitation.tex*' , *thesis* , *desktop*)

    is also executed.

· **b.3 Learn:** All collected positive lowerbounds are used to update the definitions of *succeeds*/1, *initiates*/2 and *terminates*/2 for *menu_close* and *drag_and_drop*. At this point experiments could be done for the action *menu_close* or for *drag_and_drop*. However, for *drag_and_drop* this is not allowed, since we are in the middle of an experiment for *drag_and_drop* on level a; for *menu_close* no relevant lowerbounds can be generated, because s already converged to g.

· **b.4 Stop or Retry:** G_3 is checked. It is fulfilled, so level b succeeds.

$*$ **Experiment and Observe:** Action A_3 is executed, instantiated with $\{$ *E/e348* , *Doc/'invitation.tex'* , *Loc1/desktop* , *Loc2/thesis* $\}$.

· **Experiment:** the action

$$drag\_and\_drop(\ 'invitation.tex'\ ,\ desktop\ ,\ thesis\ ),$$

corresponding to event *e348*, is executed.

· **Observe:** Again the real effects are exactly the ones expected, i.e., there are only positive lowerbounds.

· **Learn from experiment:** Consequently the maximally specific definitions are updated to be consistent with these positive lowerbounds. ~~The maximally general concept representations remain unchanged.~~

– Some more experiments are generated, which yield more positive lowerbounds. By means of these, the maximally specific definitions are generalized further, in this case until the maximally specific concept representation is equivalent to the maximally general one.

- **a.4 Stop or Retry:** The original goal G is checked. It is fulfilled, so the algorithm halts.

Then the tutor could still have other goals, for which it would like to check whether the plans it makes execute correctly. One of the following goals is

$$holds\_at(\ is\_in(\ 'invitation.ps'\ ,\ psfiles\ )\ ,\ t'_{end}\ ).$$

One of the actions in the plan for this goal is

$$act(\ E\ ,\ drag\_and\_drop(\ 'invitation.ps'\ ,\ home\ ,\ psfiles\ )\ ).$$

Executing this action does not succeed, and therefore gives the negative lowerbound of Clause (5.22). By means of this negative lowerbound, *succeeds*/1 is further updated as described in Section 5.12. The definitions for *initiates*/2 and *terminates*/2 are updated in a similar way.

6.7 Related Work

Most symbolic systems integrating planning and learning work with a STRIPS-like representation: with each action an add-list and a delete-list is associated. The add-list contains the properties that become true when executing the action, and a delete-list contains the properties that become false. The frame axiom is expressed by the STRIPS assumption: all properties are carried over from one situation to the next unless they are explicitly deleted by the executed action. This assumption makes it difficult to handle context dependent effects and derived properties. Moreover, it is more difficult to incorporate general temporal knowledge in the STRIPS framework. Furthermore STRIPS-like representations make it more difficult to apply *existing* machine learning algorithms in a straightforward way and to inherit their characteristics. The event calculus, on the other hand, has a clear semantics, and is obviously more expressive than the STRIPS representation. This is one of the main advantages of our approach.

The best known related work in the area of learning by experimentation in a planning environment is part of the PRODIGY system. The global PRODIGY system is a general problem solving system with learning capabilities. One part of the research is concerned with learning by experimentation. PRODIGY uses a STRIPS-like representation for the actions (called operators) as well as for the domain knowledge (called inference rules). This means they take an operator-oriented viewpoint, opposed to the time-oriented viewpoint of the event calculus. On the other hand, PRODIGY has the advantage of having a complete architecture to be fitted in, so that it can make use of other already implemented learning techniques, such as techniques to learn search-control rules or to learn abstractions for hierarchical planning. Possible future research could point out that the event calculus is powerful enough to represent similar methods. Research on learning by experimentation in PRODIGY [Gil, 1991] and [Carbonell and Gil, 1990] also concentrates on acquisition of pre- and postconditions of operators, and does not learn any domain knowledge. PRODIGY has another difference with our system: it does also take into account effects from other actions than the last one executed [Gil and Carbonell, 1987]. On the one hand, this makes it difficult to decide which action is responsible for an unexpected effect in the system; on the other hand this assumption might be realistic in systems where not all effects are immediately detectable. Rephrased in event calculus terms, it means that the clauses for *succeeds*/1, *initiates*/2 and *terminates*/2 must also be allowed to contain temporal literals with a timepoint other than the one occurring in the head. In order to learn such clauses, the background knowledge should not only contain the observations at the time the action was executed, but also all other relevant observations. Consequently, this is just a matter of bias for the starting clause, although it will make the problem much more complex.

LIVE [Shen, 1993] is designed to learn to solve goals in an unknown environment, and therefore has to learn conditions and consequences of its actions in that environment. LIVE also uses a STRIPS based representation, and has a similar plan-execute-observe-learn cycle as our system. The learning component works general to specific and only uses the information provided by *failing* plan executions. LIVE does only take the opportunity to do experimentation when faulty knowledge is detected *during planning*. In our active approach, experimentation is done during the learning phase. Searching for inconsistencies in a logically represented theory can in general be done by theorem proving (e.g., SATCHMO [Manthey and Bry, 1988]).

LIVE also does not rely on the presence of a teacher. It has an exploration phase in case no plan can be found for a given goal. It then selects some actions, using heuristics that try to maximize the information gain, and executes them. LIVE does therefore not have an active learning strategy, because it will only start exploring or experimenting when necessary to solve its goals.

There are also some autonomous systems that learn preconditions and effects of actions through observation of an expert, guided by the heuristic "imitate activity that you see in the environment" [Hume and Sammut, 1992], or by analyzing execution traces [Wang, 1994].

Finally, as noted in Chapter 1 and Section 6.2, the basic cycle of the architecture is very much related to the one of LEX ([Mitchell *et al.*, 1983]).

6.8 Conclusion

We have presented an architecture integrating planning and learning in the context of an autonomous agent. The underlying representation is the event calculus. The event calculus is a logical representation for temporal reasoning in general, and for planning in particular. Because it is a logical representation, clauses of the event calculus can be learned using ILP techniques. Furthermore, interactive ILP systems provide a framework for learning by experimentation. Concerning the integration using a logical representation, we have illustrated that the logical representation offers the possibility to concentrate on the integration aspects themselves, e.g., on controlling the distinct components of the agent. As this chapter's aim was only to show the ease of integrating planning and learning using logic, more questions have risen than there have been solved.

From this point of view there are several possibilities for future work. By relaxing the many constraints, other ILP learning techniques can be used for learning. We have argued that in general theory revision systems have to be used to learn knowledge about actions and domain knowledge at the same time. Also learning algorithms that can cope with noise can be used when the observations are not guaranteed to be correct. We also mentioned ILP systems learning constraints could be used to find regularities in the set of observed facts. From this point of view our setup can also be used as a benchmark for other ILP systems. Because of the modularity of the logical representation, systems learning distinct types of knowledge can easily be combined.

In this chapter we have merely concentrated on the integration of Planning and Learning, but the line of reasoning can be used to integrate other AI problem solvers with ILP as well. As an example, simulating the environment can also be done in the event calculus (Section 6.4.5). More generally, [Missiaen, 1995] describes a domain independent simulator in the event calculus. The event calculus also allows to represent knowledge about multiple agents. In that context agents could learn more about each other's behavior. One such agent is studied in an ongoing masters' thesis at our department, in the context of computer assisted exercises[2] [De Wolf and Huys, 1995]. In this application several players play a multi-player simulation game. This kind of simulation games is used to train people for decision making, e.g., to get practice in economical or military strategies. In the setup of [De Wolf and Huys, 1995] one of the players is observed by an agent. This agent derives a set of rules about the behavior of that player. The aim is to reuse these rules to simulate the player. As a learning component [De Wolf and Huys, 1995] uses CLAUDIEN. This application is another example of an integration of machine learning with problem solving in a logical context.

[2]In cooperation with SHAPE Technical Centre, The Hague.

Chapter 7

Conclusions

To conclude, we will briefly summarize the achievements of the thesis and present some possible directions for future research.

Achievements

In Chapter 2 to Chapter 4 we have studied concept learning in a language independent framework. This framework is built up based on the framework of Versionspaces of [Mitchell, 1982], and the Description Identification algorithm of [Mellish, 1991]. In this framework we have developed the Iterative Versionspaces algorithm, which efficiently computes a maximally general and a maximally specific consistent concept representation. The bi-directional approach allows a dynamic choice which strategy to use for problem solving (making errors of omission, rather than errors of commission, or vice versa), and allows to generate relevant lower- and upperbounds automatically. We have also developed a theory of redundant information elements, which allows to reduce the amount of memory needed to store information elements. A further reduction can be obtained by safely replacing information elements by automatically generated ones. The implementation of the efficient maximal generality and maximal specificity tests, of the optimal refinement operators, and of the removal of redundant information elements, are described language independently, and are also applicable in other concept learning systems. We have also described how the concept representation language can be made more expressive by introducing disjunctions. We have identified a condition under which the concept learning problem in the language of the disjunctions can be reduced to a search problem in the underlying language of disjuncts. We have illustrated that the increase in expressiveness is too large to be practically useful. Therefore we introduced almost maximally specific concept representations, and combined this with preference criteria, which minimize disjunctions on the basis of set inclusion, or on the basis of their length. This resulted in disjunctive extensions of the Description Identification algorithm and of the Iterative Versionspaces algorithm.

In Chapter 5 and Chapter 6 the frameworks of ITVS and DITVS are instantiated by means of concept representation languages and instance representation languages consisting of definite Horn clauses. We have described in detail how predicate learning is an instantiation of concept learning in a first order logic framework. In particular we have regarded predicate learning as a search problem: we have described how to define languages of definite clauses for predicate learning, and how to structure them. By doing so, the predicate learning problem can be solved by language independent concept learning techniques. The application of the language independent framework to ILP also shows that the framework is applicable on non-trivial concept representation languages.

Future Research

We believe it is important to keep up both a language independent point of view, and at the same time language specific point of view. The former point of view allows to abstract language specific aspects, such that generally applicable techniques can be developed and applied in distinct languages. Several ideas in this thesis are based on language specific techniques in attribute value languages. By generalizing them into the language independent framework, these ideas become applicable to, for instance, ILP. The latter point of view allows to *apply* the general ideas, to evaluate them, to find new interesting problems to be solved, and to find possible solutions for these problems. Following this line of reasoning, there are several topics which look interesting for future research, both in the direction from ILP towards a language independent framework, and vice versa.

Although we have instantiated our framework in the context of ILP, not all consequences of this instantiation are fully understood. For instance, the part on redundant information elements has not yet been studied in an ILP context. We think this technique is useful in the context of ILP, as it reduces the number of information elements that have to be stored. It might also be interesting to study the relation with existing work on redundancy in the context of ILP.

An important topic that is related to learning disjunctions is the automatic introduction of new concepts in the background knowledge. This is called predicate invention in the context of ILP. As observed in Chapter 4 this is related to learning disjunctions by considering the separate disjuncts as the definitions of new concepts. Here also we think that there are relations that go beyond the particular languages in which existing approaches are described, and that it would be interesting to identify these relationships.

Throughout the thesis we have not worried about noisy data. As described in Chapter 2 handling noise implies a relaxation of the *success criterion*. In this sense this is orthogonal to the approach we have taken: we have fixed the success criterion, and then investigated preference criteria (at least for the disjunctive case). [Mitchell, 1978] and [Hirsh, 1990] present approaches that embed noise-handling in the context of Versionspaces. It would be interesting to investigate to what extent their approach is applicable in the framework of Iterative Versionspaces, to what extent it can be combined with our preference criteria, and what the consequences are for the resulting Versionspaces.

On two occasions we have restricted the kind of information elements that the algorithms in this thesis accept: they did not accept information elements for the negation of a concept, and they did not accept information elements containing disjunctions. In ILP this kind of information might be acceptable and informative, though. As an example, the integrity constraints used by CLINT [De Raedt, 1992] provide a more general framework of information elements in which this kind of information would be allowed. We think it is interesting to investigate what corresponds to the use of integrity constraints in the language independent framework, and what are the consequences for the resulting versionspaces and the resulting algorithms.

Allowing information elements for the negation of a concept would imply one can learn negative concepts. Some ILP systems adopt multivalued logics in order to handle concepts together with their negation (e.g., CLINT [De Raedt, 1992] and MOBAL [Morik *et al.*, 1993]). We think this must be possible in a more abstract (and language independent) framework. That the negative information elements can be described in such a framework (see Section 3.2), is a first indication that this is possible. We also wonder whether the frameworks we developed can be extended in the direction of multiple concept learning, and under which conditions.

We have only instantiated the framework of ITVS in a classical ILP setting, in which definite clauses are learned, and in the so-called *normal semantics* of ILP (i.e., the usual one). Recently some systems have been developed that also derive clauses which are not definite (called constraints), and this in the so-called *non-monotonic semantics* of ILP (see [Muggleton and De Raedt, 1994]). It could be useful to investigate to which extent these approaches fits in extensions or in similar frameworks as the Iterative Versionspaces framework. Can the non-monotonic setting of ILP also be described in a language independent context? Points of departure for further research in this direction might be the ideas of [De Raedt and Džeroski, 1994] and [De Raedt and Van Laer, 1995], which relate the non-monotonic semantics to classical concept learning methods.

Bibliography

[Adé *et al.*, 1994] H. Adé, B. Malfait, and L. De Raedt. RUTH : an ILP Theory Revision System. In *Proceedings of the 8th International Symposium on Methodologies for Intelligent Systems (ISMIS94)*, 1994.

[Adé *et al.*, 1995] H. Adé, L. De Raedt, and M. Bruynooghe. Declarative Bias for Specific-To-General ILP Systems. *Machine Learning*, 1995. To appear.

[Apt and Bezem, 1991] K.R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9:335–363, 1991.

[Benson and Nilsson, 1995] S. Benson and N. Nilsson. Reacting, planning, and learning in an autonomous agent. In K. Furukawa, D. Michie, and S. Muggleton, editors, *Machine Intelligence*, volume 14. Oxford University Press, 1995. To appear.

[Benson, 1995] S. Benson. Action model learning and action execution in a reactive agent. In *Proceedings of the 12th International Conference on Machine Learning*, 1995. Accepted.

[Bergadano and Gunetti, 1994] F. Bergadano and D. Gunetti. Learning clauses by tracing derivations. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 11–29. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[Bergadano *et al.*, 1993] F. Bergadano, S. Brusotti, D. Gunetti, and U. Trinchero. Inductive test case generation. In *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 11–24, 1993.

[Bergadano, 1993] F. Bergadano. Inductive database relations. *IEEE Transactions on Data and Knowledge Engineering*, 5(6):969–972, 1993.

[Birkhoff, 1979] G. Birkhoff. *Lattice Theory*. American Mathematical Society Colloquium Publications. American Mathematical Society, 1979. 3rd Edition, 3rd Print.

[Boström and Idestam-Almquist, 1994] H. Boström and P. Idestam-Almquist. Specialization of Logic Programs by Pruning SLD-Trees. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 31–48. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[Bratko and Grobelnik, 1993] I. Bratko and M. Grobelnik. Inductive learning applied to program construction and verification. In *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 279–292, 1993.

[Bratko *et al.*, 1989] I. Bratko, I. Mozetic, and N. Lavrac. *Kardio : a study in deep and qualitative knowledge for expert systems.* The MIT Press, 1989.

[Bruynooghe *et al.*, 1994] M. Bruynooghe, S. Debraya, M. Hermenegildo, and M. Maher, editors. *The Journal of Logic Programming - Special Issue: 10 years Journal of Logic Programming*, volume 19 & 20. May 1994.

[Bundy *et al.*, 1985] A. Bundy, B. Silver, and D. Plummer. An analytical comparison of some rule-learning programs. *Artificial Intelligence*, 27:137–181, 1985.

[Buntine, 1987] W. Buntine. Induction of Horn-Clauses: methods and the plausible generalization algorithm. *International Journal of Man-Machine Studies*, 26:499–520, 1987.

[Buntine, 1988] Wray Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36:375–399, 1988.

[Cameron-Jones and Quinlan, 1993] R.M. Cameron-Jones and J.R. Quinlan. Avoiding pitfalls when learning recursive theories. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1050–1055. Morgan Kaufmann, 1993.

[Carbonell and Gil, 1990] J.G. Carbonell and Y. Gil. Learning by experimentation : The operator refinement method. In Y. Kodratoff and R.S. Michalski, editors, *Machine Learning: an artificial intelligence approach*, volume 3. Morgan Kaufmann, 1990.

[Clancey, 1987] W.J. Clancey. *Knowledge-Based Tutoring. The GUIDON Program.* The MIT Press Series in AI, 1987.

[Cohen and Feigenbaum, 1981] P.R. Cohen and E.A. Feigenbaum, editors. *The handbook of artificial intelligence*, volume 3. Morgan Kaufmann, 1981.

[Conklin and Witten, 1994] D. Conklin and I.H. Witten. Complexity-Based Induction. *Machine Learning*, 16:203–225, 1994.

[De Raedt and Bruynooghe, 1988] L. De Raedt and M. Bruynooghe. On interactive concept-learning and assimilation. In D. Sleeman, editor, *Proceedings of the 3rd European Working Session on Learning*. Pitman, 1988.

[De Raedt and Bruynooghe, 1990] L. De Raedt and M. Bruynooghe. Indirect relevance and bias in inductive concept-learning. *Knowledge Acquisition*, 2:365–390, 1990.

[De Raedt and Bruynooghe, 1992a] L. De Raedt and M. Bruynooghe. Belief updating from integrity constraints and queries. *Artificial Intelligence*, 53:291–307, 1992.

[De Raedt and Bruynooghe, 1992b] L. De Raedt and M. Bruynooghe. A unifying framework for concept-learning algorithms. *The Knowledge Engineering Review*, 7(3):251–269, 1992.

[De Raedt and Bruynooghe, 1993] L. De Raedt and M. Bruynooghe. A theory of clausal discovery. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1058–1063. Morgan Kaufmann, 1993.

[De Raedt and Džeroski, 1994] L. De Raedt and S. Džeroski. First order jk-clausal theories are pac-learnable. *Artificial Intelligence*, 70:375–392, 1994.

[De Raedt and Van Laer, 1995] L. De Raedt and W. Van Laer. Inductive constraint logic. Unpublished, 1995.

[De Raedt *et al.*, 1991] L. De Raedt, G. Sablon, and M. Bruynooghe. Using interactive concept-learning for knowledge base validation and verification. In M. Ayel and J.P. Laurent, editors, *Validation, Verification and Testing of Knowledge Based Systems*, pages 177–190. John Wiley & Sons, 1991.

[De Raedt *et al.*, 1993] L. De Raedt, N. Lavrač, and S. Džeroski. Multiple predicate learning. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 1037–1042. Morgan Kaufmann, 1993.

[De Raedt, 1992] L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach.* Academic Press, 1992.

[De Schreye and Decorte, 1994] D. De Schreye and S. Decorte. Termination of logic programs: the never-ending story. *Journal of Logic Programming*, 19 & 20:199–260, May 1994.

[De Wolf and Huys, 1995] B. De Wolf and M. Huys. Inductief logisch programmeren toegepast op computerondersteunde oefeningen. Master's thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1995. in Dutch.

[Denecker *et al.*, 1992] M. Denecker, L. Missiaen, and M. Bruynooghe. Temporal reasoning with abductive event calculus. In B. Neumann, editor, *Proceedings of the 10th European Conference on Artificial Intelligence*, pages 384–388. John Wiley & Sons, 1992.

[Denecker, 1993] M. Denecker. *Knowledge Representation and reasoning in Incomplete Logic Programming.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1993.

[Dietterich and Michalski, 1983] T.G. Dietterich and R.S. Michalski. A comparative review of selected methods for learning from examples. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: an artificial intelligence approach*, volume 1. Tioga Publishing Company, 1983.

[Etzioni, 1993a] O. Etzioni. Acquiring search-control knowledge via static analysis. *Artificial Intelligence*, 62:255–301, 1993.

[Etzioni, 1993b] O. Etzioni. Intelligence without robots: A reply to brooks. *AI Magazine*, Winter:7–13, 1993.

[Evans, 1990] C. Evans. The macro-event calculus: Representing temporal granularity. In *Proc. of PRICAI, Tokyo*, 1990.

[Fikes and Nilsson, 1971] R.E. Fikes and N.J. Nilsson. Strips: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189 – 208, 1971.

[Garey and Johnson, 1979] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-completeness.* Freeman, San Francisco, California, 1979.

[Genesereth and Nilsson, 1987] M. Genesereth and N. Nilsson. *Logical foundations of artificial intelligence.* Morgan Kaufmann, 1987.

[Gil and Carbonell, 1987] Y. Gil and J.G. Carbonell. Learning by experimentation. In *Proceedings of the 4th Machine Learning Workshop.* Morgan Kaufmann, 1987.

[Gil, 1991] Y. Gil. A domain-independent framework for effective experimentation in planning. In L.A. Birnbaum and G.C. Collins, editors, *Proceedings of the 8th International Workshop on Machine Learning.* Morgan Kaufmann, 1991.

[Ginsberg and Harvey, 1992] M.L. Ginsberg and W.D. Harvey. Iterative broadening. *Artificial Intelligence*, 55:367–383, 1992.

[Gottlob and Fermüller, 1993] G. Gottlob and C.G. Fermüller. Removing redundancy from a clause. *Artificial Intelligence*, 61:263–289, 1993.

[Gottlob, 1987] G. Gottlob. Subsumption and implication. *Information Processing Letters*, 24:109–111, 1987.

[Haussler, 1988] D. Haussler. Quantifying inductive bias : AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36:177 – 221, 1988.

[Hirsh, 1990] H. Hirsh. *Incremental Version-Space Merging: A General Framework for Concept Learning.* Kluwer Academic Publishers, 1990.

[Hirsh, 1992a] H. Hirsh. The computational complexity of the candidate-elimination algorithm. Technical Report 36, Computer Science Department, Rutgers University, 1992.

[Hirsh, 1992b] H. Hirsh. Polynomial-time learning with version spaces. In *Proceedings of the 9th National Conference on Artificial Intelligence (AAAI-92)*, pages 117–122. AAAI Press, 1992.

[Hume and Sammut, 1992] D. Hume and C. Sammut. Applying inductive logic programming in reactive environments. In S. Muggleton, editor, *Inductive logic programming*, pages 539–549. Academic Press, London, 1992.

[Idestam-Almquist, 1993] P. Idestam-Almquist. *Generalization of clauses.* PhD thesis, Stockholm University, Department of Computer and Systems Sciences, 1993.

[Indermaur, 1995] K. Indermaur. Baby steps. *BYTE*, 20(3):97–104, March 1995.

[Jung, 1993] B. Jung. On inverting generality relations. In *Proceedings of the 3rd International Workshop on Inductive Logic Programming*, pages 87–101, 1993.

[Knoblock, 1994] C.A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68:243–302, 1994.

[Kodratoff, 1988] Y. Kodratoff. *Introduction to Machine Learning.* Pitman, 1988.

[Korf, 1985] R. Korf. Depth-first iterative deepening : an optimal admissable search. *Artificial Intelligence*, 27:97–109, 1985.

[Kowalski and Sergot, 1986] Robert A. Kowalski and Marek Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.

[Lansky, 1988] A. L. Lansky. Localized event-based reasoning for multiagent domains. *Computational Intelligence*, 4(4):319 – 340, nov 1988.

[Lapointe and Matwin, 1992] S. Lapointe and S. Matwin. Sub-unification: a tool for efficient induction of recursive programs. In *Proceedings of the 9th International Workshop on Machine Learning*. Morgan Kaufmann, 1992.

[Lavrač and Džeroski, 1994] N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.

[Lavrač *et al.*, 1991] N. Lavrač, S. Džeroski, and M. Grobelnik. Learning non-recursive definitions of relations with LINUS. In Yves Kodratoff, editor, *Proceedings of the 5th European Working Session on Learning*, volume 482 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1991.

[Lloyd, 1987] J.W. Lloyd. *Foundations of logic programming*. Springer-Verlag, 2nd edition, 1987.

[Manthey and Bry, 1988] R. Manthey and F. Bry. SATCHMO: a theorem prover implemented in prolog. In *Proceedings of the 9ht International Conference on Automated Deduction (CADE88)*, pages 415–434. Springer-Verlag, 1988.

[Mellish, 1991] C. Mellish. The description identification problem. *Artificial Intelligence*, 52:151 – 167, 1991.

[Michalski and Tecuci, 1994] R.S. Michalski and G. Tecuci, editors. *Machine Learning: a multistrategy approach*, volume 4. Morgan Kaufmann, 1994.

[Michalski, 1983] R.S. Michalski. A theory and methodology of inductive learning. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: an artificial intelligence approach*, volume 1. Morgan Kaufmann, 1983.

[Michalski, 1986] R.S. Michalski. Understanding the nature of learning: Issues and research directions. In R.S. Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: an artificial intelligence approach*, volume 2, pages 3–25. Morgan Kaufmann, 1986.

[Minton *et al.*, 1989] S. Minton, J.G. Carbonell, C.A. Knoblock, D.R. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: A problem solving perspective. *Artificial Intelligence*, 40:63–118, 1989.

[Minton, 1993] S. Minton, editor. *Machine Learning methods for Planning*. Morgan Kaufmann, 1993.

[Missiaen *et al.*, 1995] L.R. Missiaen, M. Denecker, and M. Bruynooghe. CHICA, an abductive planning system based on event calculus. *Journal of Logic and Computation*, 5(5), September 1995. Forthcoming.

[Missiaen, 1991] L. Missiaen. *Localized abductive planning with the event calculus.* PhD thesis, Department of Computer Science, Katholieke Universiteit Leuven, 1991.

[Missiaen, 1994] L.R. Missiaen. Discrete event simulation using event calculus. In P.T. Metaxas, editor, *Proceedings of the 6th International Conference on Tools with Artificial Intelligence*, pages 506–512. IEEE Computer Society Press, Los Alamitos, CA, 1994.

[Missiaen, 1995] L.R. Missiaen. ECSIM: Discrete event simulation using event calculus. *International Journal on Artificial Intelligence Tools*, 1995. Accepted.

[Mitchell *et al.*, 1983] T.M. Mitchell, P.E. Utgoff, and R. Banerji. Learning by experimentation: acquiring and refining problem-solving heuristics. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: an artificial intelligence approach*, pages 163–190. Tioga publishing company, 1983.

[Mitchell *et al.*, 1985] T.M. Mitchell, S. Mahadevan, and L.I. Steinberg. LEAP: a learning apprentice for vlsi design. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*, pages 573–580. Morgan Kaufmann, 1985.

[Mitchell, 1978] T.M. Mitchell. *Version Spaces: An Approach to Concept Learning.* PhD thesis, Stanford University, 1978.

[Mitchell, 1982] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[Montanari *et al.*, 1992] A. Montanari, E. Maim, E. Ciapessoni, and E. Ratto. Dealing with time granularity in the event calculus. In *Proceedings of FGCS, Tokyo*, pages 702–712, 1992.

[Morik *et al.*, 1993] K. Morik, S. Wrobel, J.-U. Kietz, and W. Emde. *Knowledge Acquisition and Machine Learning: Theory, Methods and Applications.* Academic Press, 1993.

[Muggleton and Buntine, 1988] S. Muggleton and W. Buntine. Machine invention of first order predicates by inverting resolution. In *Proceedings of the 5th International Workshop on Machine Learning*, pages 339–351. Morgan Kaufmann, 1988.

[Muggleton and De Raedt, 1994] S. Muggleton and L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[Muggleton and Feng, 1992] S. Muggleton and C. Feng. Efficient induction of logic programs. In S. Muggleton, editor, *Inductive logic programming*, pages 281–298. Academic Press, 1992.

[Muggleton *et al.*, 1992] S. Muggleton, A. Srinivasan, and M. Bain. Compression, significance and accuracy. In *Proceedings of the 9th International Workshop on Machine Learning*, pages 338–347. Morgan Kaufmann, 1992.

[Muggleton, 1992] S. Muggleton. Inductive logic programming. In S. Muggleton, editor, *Inductive logic programming*, pages 3–28. Academic Press, 1992.

[Muggleton, 1994] S. Muggleton. Inverting implication. *Artificial Intelligence*, 1994. To appear.

[Muggleton, 1995] S. Muggleton. Mode-directed inverse resolution. In K. Furukawa, D. Michie, and S. Muggleton, editors, *Machine Intelligence*, volume 14. Oxford University Press, 1995. To appear.

[Murray, 1987a] K.S. Murray. Multiple convergence: an approach to disjunctive concept acquisition. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1987.

[Murray, 1987b] K.S. Murray. Multiple convergence: An experiment in disjunctive concept acquisition. Technical Report AI TR87-56, Artificial Intelligence Laboratory, The University of Texas at Austin, 1987.

[Omar, 1994] R. Omar. Artificial intelligence through logic? *AI Communications*, 7(3 & 4), 1994.

[Plotkin, 1970] G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, pages 153–163. Edinburgh University Press, 1970.

[Plotkin, 1971a] G. Plotkin. *Automatic Methods of Inductive Inference*. PhD thesis, Edinburgh University, 1971.

[Plotkin, 1971b] G. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6, pages 101–124. Edinburgh University Press, 1971.

[Quinlan, 1990] J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[Rich and Knight, 1991] E. Rich and K. Knight. *Artificial Intelligence*. McGraw-Hill, 1991. Second Edition.

[Rissanen, 1978] J. Rissanen. Modeling by Shortest Data Description. *Automatica*, 14:465–471, 1978.

[Robinson, 1965] J.A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12:23–41, 1965.

[Rouveirol, 1992] C. Rouveirol. Extensions of inversion of resolution applied to theory completion. In S. Muggleton, editor, *Inductive logic programming*, pages 63–92. Academic Press, 1992.

[Rouveirol, 1994] C. Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14:219–232, 1994.

[Russell and Grosof, 1990] S. Russell and B. Grosof. A sketch of autonomous learning using declarative bias. In P.B. Brazdil and K. Konolige, editors, *Machine Learning, Meta-Reasoning and Logics*, pages 19–54. Kluwer Academic Publishers, 1990.

[Sablon and Bruynooghe, 1994] G. Sablon and M. Bruynooghe. Using the event calculus to integrate planning and learning in an intelligent autonomous agent. In C. Bäckström and E. Sandewall, editors, *Current Trends in AI Planning*, pages 254–265. IOS Press, 1994.

[Sablon and De Raedt, 1995] G. Sablon and L. De Raedt. Forgetting and compacting data in concept learning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 1995. To appear.

[Sablon *et al.*, 1994] G. Sablon, L. De Raedt, and M. Bruynooghe. Iterative versionspaces. *Artificial Intelligence*, 69:393–409, 1994.

[Sammut and Banerji, 1986] C. Sammut and R. Banerji. Learning concepts by asking questions. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: an artificial intelligence approach*, volume 2, pages 167–192. Morgan Kaufmann, 1986.

[Sebag and Rouveirol, 1994] M. Sebag and C. Rouveirol. Induction of maximally general clauses consistent with integrity constraints. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 195–215. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[Sebag, 1994] M. Sebag. Using constraints to building version spaces. In F. Bergadano and L. De Raedt, editors, *Proceedings of the 7th European Conference on Machine Learning*, volume 784 of *Lecture Notes in Artificial Intelligence*, pages 257–286. Springer-Verlag, 1994.

[Shanahan, 1989] M.P. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1989.

[Shanahan, 1990] M. Shanahan. Representing continuous change in the event calculus. In *Proceedings of the 9th European Conference on Artificial Intelligence*, page 598, 1990.

[Shapiro, 1983] E.Y. Shapiro. *Algorithmic Program Debugging*. The MIT Press, 1983.

[Shen, 1993] W.M Shen. Discovery as autonomous learning from the environment. *Machine Learning*, 12:143–165, 1993.

[Sleeman and Brown, 1982] D. Sleeman and J.S. Brown, editors. *Intelligent Tutoring Systems*. Academic Press, 1982.

[Smith and Rosenbloom, 1990] B.D. Smith and P.S. Rosenbloom. Incremental non-backtracking focusing: A polynomially bounded generalization algorithm for version spaces. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI-90)*, pages 848–853. AAAI Press, 1990.

[Smith *et al.*, 1985] R.G. Smith, T.M. Mitchell, H.A. Winston, and B.G. Buchanan. Representation and use of explicit justifications for knowledge base refinement. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1985.

[Srinivasan *et al.*, 1994] A. Srinivasan, S.H. Muggleton, R.D. King, and M.J.E. Sternberg. Mutagenesis: Ilp experiments in a non-determinate biological domain. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237 of *GMD-Studien*, pages 217–232. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

[Subramanian and Feigenbaum, 1986] D. Subramanian and J. Feigenbaum. Factorization in experiment generation. In *Proceedings of the 5th National Conference on Artificial Intelligence (AAAI-86)*, pages 518–522. Morgan Kaufmann, 1986.

[Subramanian and Genesereth, 1987] D. Subramanian and M. Genesereth. The relevance of irrelevance. In *Proceedings of the 10th International Joint Conference on Artificial Intelligence*. Morgan Kaufmann, 1987.

[Tamaki and Sato, 1986] H. Tamaki and T. Sato. OLD resolution with tabulation. In E.Y. Shapiro, editor, *Proceedings of the 3rd International Conference on Logic Programming*, volume 225 of *Lecture Notes in Computer Science*, pages 84–98. Springer-Verlag, 1986.

[Tecuci and Kodratoff, 1990] G. Tecuci and Y. Kodratoff. Apprenticeship learning in imperfect domain theories. In Y. Kodratoff and R.S. Michalski, editors, *Machine Learning: an artificial intelligence approach*, volume 3, pages 514–551. Morgan Kaufmann, 1990.

[Utgoff, 1986] P.E. Utgoff. Shift of bias for inductive concept-learning. In R.S Michalski, J.G. Carbonell, and T.M. Mitchell, editors, *Machine Learning: an artificial intelligence approach*, pages 107–148. Morgan Kaufmann, 1986.

[Van Belleghem *et al.*, 1994] K. Van Belleghem, M. Denecker, and D. De Schreye. Representing continuous change in the abductive event calculus. In P. Van Hentenrijck, editor, *Proc. of the International Conference on Logic Programming*, pages 225–240, 1994.

[van der Laag and Nienhuys-Cheng, 1994] P.R.J. van der Laag and S.-H. Nienhuys-Cheng. Existence and nonexistence of complete refinement operators. In F. Bergadano and L. De Raedt, editors, *Proceedings of the 7th European Conference on Machine Learning*, volume 784 of *Lecture Notes in Artificial Intelligence*, pages 307–322. Springer-Verlag, 1994.

[Van Laer *et al.*, 1994] W. Van Laer, L. Dehaspe, and L. De Raedt. Applications of a logical discovery engine. In *Proceedings of the AAAI Workshop on Knowledge Discovery in Databases*, pages 263–274, 1994.

[Vanlehn and Ball, 1987] K. Vanlehn and W. Ball. A version space for grammars. *Machine Learning*, 2:39–74, 1987.

[Veloso *et al.*, 1995] M. Veloso, J. Carbonell, A. Pérez, D. Borrajo, E. Fink, and J. Blythe. Integrating planning and learning: the PRODIGY architecture. *Journal for Experimental and Theoretical Artificial Intelligence*, 7:81–120, 1995.

[Wang, 1994] X. Wang. Learning planning operators by observation and practice. In *Proceedings of the Second Conference on AI Planning Systems*, 1994.

[Wilkins, 1988] D. Wilkins. *Practical Planning: Extending the Classical AI Paradigm.* Morgan Kaufmann, 1988.

[Winston, 1975] P.H. Winston. Learning structural descriptions from examples. In P.H. Winston, editor, *Psychology of Computer Vision.* The MIT Press, 1975.

[Wrobel, 1993] S. Wrobel. On the proper definition of minimality in specialisation and theory revision. In *Proceedings of the 6th European Conference on Machine Learning,* volume 667, pages 65–82. Lecture Notes in Artificial Intelligence, 1993.

Inhoudsopgave

Samenvatting

1 Inleiding

Automatisch leren is het deelgebied van het onderzoeksgebied *Kunstmatige Intelligentie* waar algoritmen worden ontwikkeld om intelligente programma's (of *kennis-systemen*) zichzelf te laten aanpassen. Deze intelligente programma's hebben als hoofdtaak bepaalde problemen op te lossen aan de hand van hun kennis. Door middel van de leer-algoritmen zijn ze in staat hun kennis om problemen op te lossen te wijzigen, om op die manier meer problemen te kunnen oplossen, of problemen beter of sneller te kunnen oplossen. De vorm van leren waar we ons op zullen concentreren is het leren van concepten.
Deze kennis-systemen moeten dan hun hoofdtaak (het oplossen van problemen) afwisselen met het aanpassen van hun kennis (het leren). Op die manier ontstaat er een wisselwerking van enerzijds het oplossen van problemen aan de hand van de huidige kennis, en van anderzijds het aanpassen van die huidige kennis. De informatie die nodig is om kennis aan te passen kan uit verscheidene bronnen voortkomen: enerzijds kan er een menselijke gebruiker zijn die informatie aan het systeem doorgeeft. Anderzijds kan het systeem eventueel beschikken over een evaluatie-mechanisme, dat gevonden oplossingen kan toetsen. Indien de gevonden oplossing niet voldoet, of kan verbeterd worden, beschikt het systeem over informatie om zijn kennis aan te passen. Ook wanneer een oplossing wel voldoet, kan het systeem daaruit besluiten trekken: deze informatie bevestigt dat de gebruikte kennis tot goede oplossingen leidt. Indien het systeem ook beschikt over een mechanisme om zelf nieuwe problemen te genereren, deze problemen te laten oplossen door middel van de huidige kennis, en de oplossingen te laten toetsen (waaruit dan opnieuw belangrijke informatie kan gehaald worden om te leren), spreken we van een *actief* leer-systeem. De zelf gegenereerde problemen kunnen beschouwd worden als *experimenten*. Deze cyclus werd beschreven in het systeem LEX [Mitchell *et al.*, 1983].
Opdat het systeem afwisselend de huidige kennis zou kunnen gebruiken en ze dan weer aanpassen, moet het gebruikte leer-algoritme *incrementeel* zijn: het moet de huidige kennis kunnen aanpassen aan de hand van de nieuwe informatie, zonder daarbij noodzakelijkerwijze alle vroeger vergaarde informatie opnieuw te moeten verwerken.
Een ander belangrijk aspect is de *integratie* van het algoritme om de problemen op te lossen en het leer-algoritme. Kennis voorstellen door middel van logische formalismen laat toe op een eenvoudige en eenduidige manier het kennis-systeem en het leer-systeem te integreren. Bovendien zijn kennisvoorstellingen gebaseerd op logica zeer expressief; ze zijn wiskundig grondig onderbouwd, en hun semantiek is goed gekend. Tegenstanders voeren aan dat het gebruik van logische voorstellingen in intelligente systemen praktisch onbruikbaar zou zijn, en dat kennis van nature niet logisch en declaratief zou zijn, maar wel proceduraal. Zelfs indien deze argumenten gegrond zouden blijken te zijn, voeren wij aan dat het gebruik van logische voorstellingen dan toch nog kan gebruikt worden om deze intelligente systemen en de problemen die ze kunnen oplossen formeel te beschrijven en ten gronde te bestuderen. Vanuit dit oogpunt spitsen wij onze aandacht dan ook toe op het leren van kennis voorgesteld door middel van logische formalismen, in het bijzonder door middel van predicatenlogica.

Deze thesis bestaat hoofdzakelijk uit twee delen. In een eerste deel (hoofdstukken 2, 3 en 4) wordt het incrementeel leer-probleem voorgesteld als een zoek-probleem. Hierbij gaat speciale aandacht naar het beschrijven van de oplossingsruimte en naar de zoekstrategie, waarbij de tijdscomplexiteit van de algoritmen wordt afgewogen tegenover de geheugencomplexiteit. Speciale aandacht wordt besteed aan het vermijden van het opslaan van overtollige informatie. Ook wordt aandacht besteed aan het uitbreiden van de zoekruimte, indien ze geen oplossing blijkt te bevatten. Deze uitbreiding gebeurt op een voorstellings-onafhankelijke manier, in het bijzonder door het invoeren van disjuncties.
De belangrijkste bijdragen van dit deel zijn:

- het *Iterative Versionspaces* algoritme, dat in het slechtste geval een geheugenverbruik heeft dat lineair is in functie van het aantal informatie-elementen. Dit algoritme wordt beschreven in een ruimer kader: het kader van de Iteratieve Versieruimten;

- de voorstellings-onafhankelijke beschrijving van overtollige informatie, en van technieken om informatie te vervangen door meer gecompacteerde informatie met dezelfde informatie-inhoud;

- het beschrijven van de met disjuncties uitgebreide oplossingsruimte, al dan niet in combinatie met bijkomende, voorstellings-onafhankelijke beperkingen die het disjunctieve probleem praktisch bruikbaar en interessant moeten houden;

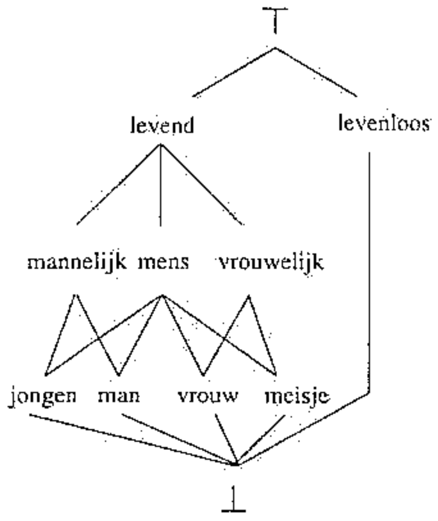
- het *Disjunctive Description Identification* algoritme, en het *Disjunctive Iterative Versionspaces* algoritme, die de zoekstrategieën die eerst niet-disjunctief werden bestudeerd, nu uitbreiden naar het disjunctieve geval.

In het tweede deel worden deze ideeën toegepast in het onderzoeksdomein van *Inductief Logisch Programmeren (Eng. Inductive Logic Programming; ILP)* [Muggleton en De Raedt, 1994]. In het bijzonder specifiëren we de onderscheiden voorstellings-afhankelijke parameters van het in het voorheen geschetste kader van Iteratieve Versieruimten.
De belangrijkste bijdragen van dit tweede deel zijn:

- het volledig formalizeren van het leren van concepten in een ILP context, zodat de resultaten uit het eerste deel onmiddellijk toepasbaar zijn in een ILP context;

- dit deel illustreert bovendien dat het kader van Iteratieve Versieruimten, zowel disjunctief als niet-disjunctief, inderdaad toepasbaar zijn in een voorstellings-specifieke context.

Deze samenvatting is gestructureerd als volgt. Sectie 2 geeft aan wat moet verstaan worden onder het leren van concepten. Sectie 3 bespreekt de verschillende elementen van het eerste deel: (niet-disjunctieve) iteratieve versieruimten, het compacteren van informatie en disjunctieve iteratieve versieruimten. Sectie 4 bespreekt dan het tweede deel: de toepassing van Iteratieve Versieruimten in Inducief Logisch Programmeren. Tenslotte besluiten we in Sectie 5.



Figuur 1: De tralie $\mathcal{M}$

2 Inleiding tot het leren van concepten

We introduceren het probleem van leren van concepten aan de hand van een voorbeeld.

Voorbeeld Beschouw de tralie $\mathcal{M}$ voorgesteld in Figuur 1 (een uitbreiding van een tralie uit [Mellish, 1991]). Elk van de elementen in deze tralie stelt een verzameling "wezens" voor: *levend* stelt bijvoorbeeld alle levende wezens voor, *mens* alle menselijke wezens, *man* alle mannelijke mensen van 18 jaar oud of ouder, *jongen* alle mannelijke mensen jonger dan 18, enz. Voorbeelden van voorstellingen van menselijke wezens zijn, bijvoorbeeld, *hilde*, *wim*, *luc*, *hendrik*. Andere levende wezens zijn bijvoorbeeld *tweety* (een vrouwelijke vogel), en *oliver* (een mannelijke vogel). De wezens in kwestie zijn niet voorgesteld in de figuur. Onze manier van spreken geeft aan dat we, om te redeneren over bepaalde wezens of over verzamelingen van wezens, elk wezen en elke verzameling van wezens op een of andere manier moeten *voorstellen*. Redeneren over wezens en verzamelingen van wezens wordt dan teruggebracht tot redeneren met de overeenkomstige voorstellingen. In onze terminologie noemen we een wezen een *instantie*; een verzameling van wezens noemen we een *concept*. ◇

We komen dan in het algemeen tot de volgende noties:

- Objecten uit het domein waarin we werken noemen we *instanties*. In het voorbeeld zijn de objecten wezens. Elk object wordt voorgesteld door een *instantie-beschrijving*. Zo laat de beschrijving *hilde* toe om over het *echte* wezen "hilde" te praten. De verzameling van alle instantie-beschrijvingen noemen we de instantie-taal; de instantie-taal noteren we met $\mathcal{L}_I$.

- Een verzameling van objecten noemen we een *concept*. Elk concept wordt voorgesteld door een *concept-beschrijving*. Zo laat de beschrijving *mens* toe om over de verzameling van alle menselijke wezens te praten. De verzameling van alle concept-beschrijvingen noemen we de concept-taal; de concept-taal noteren we met $\mathcal{L}_C$.

Tussen instantie-beschrijvingen en concept-beschrijvingen moet er een relatie bestaan die overeenkomt met de relatie "is een element van" tussen instanties en concepten. Dit is de relatie "dekt". Deze relatie is afhankelijk van de gekozen concept- en instantie-taal. Opdat ze zou overeenkomen met "is een element van" moet ze voldoen aan de volgende voorwaarde: een concept-beschrijving dekt een instantie-beschrijving enkel en alleen indien het overeenkomstig concept de overeenkomstige instantie bevat.
Indien niet alle deelverzamelingen van instanties een voorstelling hebben in $\mathcal{L}_C$, is het soms mogelijk een concept t te identificeren zonder dat alle elementen ervan gegeven zijn. Bovendien kunnen we ook *negatieve* informatie gebruiken om een concept t te identificeren. Indien men weet dat een bepaalde instantie *niet* tot het concept behoort, kan men immers alle concepten uitsluiten die deze instantie toch bevatten. In de grond is het identificeren van een concept, gegeven een verzameling instanties die er wel toe behoren (positieve instanties), en een verzameling instanties die er zeker niet toe behoren (negatieve instanties), een *leer-probleem*. Het voordeel van een concept te identificeren, is dat we dan ook van andere instanties kunnen bepalen of ze tot dat concept behoren of niet, zonder dat expliciet gegeven is of ze positieve of negatieve instanties zijn. Bijgevolg is dit een vorm van *inductie*: uit de eigenschappen van *specifieke instanties* die tot een concept behoren, leiden we *algemene voorwaarden* af om tot dat concept te behoren. Daardoor kan ook voor nog niet geziene instanties worden afgeleid dat ze tot het concept behoren. Op deze manier geleerde concepten zullen in het algemeen in kennis-systemen worden beschouwd als nieuwe kennis; met deze nieuwe kennis kunnen een aantal voorheen onoplosbare problemen worden opgelost.
In het vervolg van deze samenvatting zullen we alleen nog werken met instantie-*beschrijvingen* en concept-*beschrijvingen*. Omwille van de leesbaarheid zullen we ze echter altijd afkorten tot *instanties*, resp. *concepten*. Het leer-probleem wordt dan herleid tot het vinden van een concept dat alle positieve instanties dekt, en geen enkele negatieve instantie dekt.

Voorbeeld Stel dat men moet bepalen welke wezens kunnen praten, d.w.z. dat men een concept $c \in \mathcal{L}_C$ moet vinden dat alle wezens bevat die kunnen praten, en geen andere. Stel dat gegeven is dat *hilde* and *wim* kunnen praten, d.w.z. dat ze positieve instanties zijn van c, en dat *tweety* niet kan praten, en dus een negatieve instantie is van c. Dan moet c gelijk zijn aan *mens*. Immers: *mannelijk*, *man* en *jongen* bevatten *hilde* niet; *vrouwelijk*, *vrouw* en *meisje* bevatten *wim* niet; *levenloos* en $\perp$ bevatten *hilde* noch *wim*; *levend* en $\top$ bevatten *tweety*; alleen *mens* bevat *hilde* en *wim*, terwijl het toch *tweety* niet bevat. Omdat we nu afgeleid hebben dat alle menselijke wezens kunnen praten, betekent dit dat *luc* en *hendrik* eveneens kunnen praten, hoewel dat niet expliciet gegeven was. Dit resultaat is afhankelijk van de gekozen concept-taal. Indien bijvoorbeeld ook het concept "mensen die aan de K.U.Leuven werken" tot $\mathcal{L}_C$ behoorde, zou ook dat een mogelijke oplossing zijn. $\diamond$

In het algemeen kan een leeralgoritme niet zelf bepalen of het het te zoeken concept heeft gevonden of niet. Enerzijds vereist dit dat er voldoende positieve en negatieve instanties gekend zijn om alle andere kandidaat oplossingen uit te sluiten. Anderzijds vereist het dat $\mathcal{L}_C$ het te zoeken concept inderdaad bevat. Geen van beide voorwaarden is noodzakelijkerwijze voldaan. Daarom beperken de meeste leer-algoritmen zich tot het vinden van één *kandidaat oplossing* die alle positieve instanties dekt, en geen enkel negatieve instantie dekt.

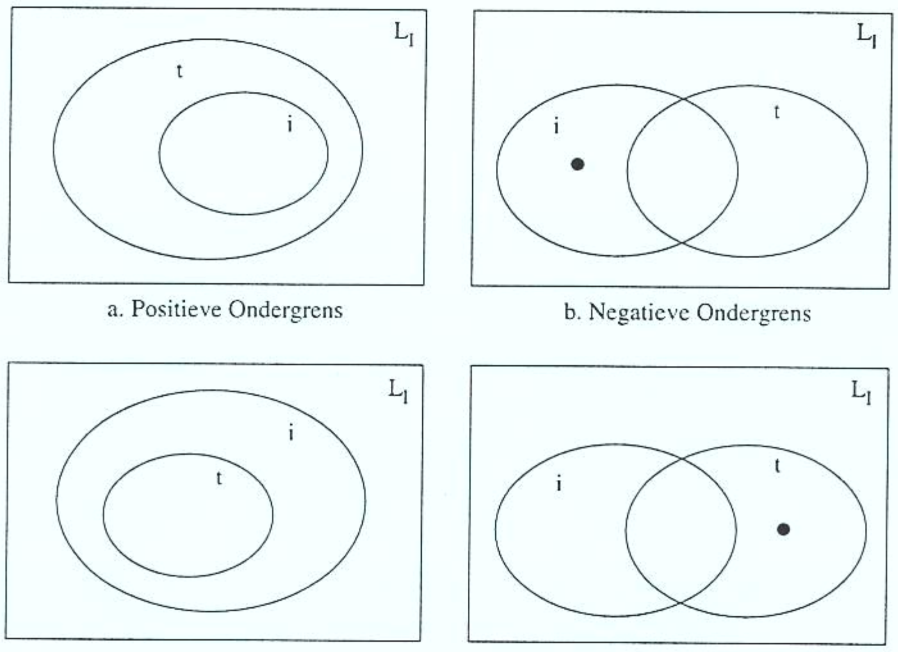
3 Iteratieve Versieruimten

Het oplossen van het probleem van Sectie 2 gebeurt meestal door middel van een zoekproces in de concept-taal $\mathcal{L}_C$. Om op een systematische manier $\mathcal{L}_C$ te doorzoeken, wordt $\mathcal{L}_C$ over het algemeen gestructureerd door middel van de *orde-relatie* "algemener dan". In het algemeen is (de voorstelling X_1 van) een verzameling instanties V_1 algemener dan (de voorstelling X_2 van) een verzameling V_2, indien V_2 een deelverzameling is van V_1. Dat X_1 algemener is dan X_2 wordt genoteerd door $X_2 \preccurlyeq X_1$. Als X_1 algemener is dan X_2, zeggen we ook dat X_2 *specifieker* is dan X_1, dat X_1 een *veralgemening* is van X_2, en dat X_2 een *specializatie* is van X_1.

Voorbeeld De tralie in Figuur 1 stelt niet alleen de concepten van $\mathcal{L}_C$ voor, maar ook hoe deze concepten zijn geordend t.o.v. "algemener dan". Het meest algemene element $\top$ in $\mathcal{L}_C$, dat de verzameling van *alle* wezens voorstelt, staat bovenaan. Het meest specifieke element $\bot$ staat onderaan. Een element X_1 van de tralie is algemener dan een element X_2 van de tralie, indien $X_1 = X_2$, of indien X_1 rechtstreeks verbonden is met X_3 en boven X_3 staat, en X_3 algemener is dan X_2. Zo is, bijvoorbeeld, *levend* zowel algemener dan *mannelijk* als algemener dan *vrouwelijk*. Anderzijds is *mens* niet algemener dan *mannelijk*, omdat *mens* niet gelijk is aan *mannelijk*, en omdat *man* noch *vrouw* algemener zijn dan *mannelijk*. ◇

Informatie over hoe concepten uit $\mathcal{L}_C$ zich onderling verhouden met betrekking tot de relatie "algemener dan", noemen we *achtergrondkennis*. De achtergrondkennis noteren we door $\mathcal{B}$. In deze thesis bestuderen we alleen het leren van één concept, en veronderstellen we daarbij dat de achtergrondkennis correct is.
De orde-relatie "algemener dan" laat toe concepten te identificeren op basis van een meer algemeen soort van informatie: positieve en negatieve onder- en bovengrenzen. Het gebruik van deze vier soorten *informatie-elementen* voor het leren van concepten werd ingevoerd door [Mellish, 1991]. We zullen ze illustreren aan de hand van een voorbeeld. Figuur 2 illustreert de relatie tussen een informatie-element i en het te zoeken concept t op een visuele manier.

Voorbeeld Stel dat we opnieuw zoeken naar het concept "kan praten". Indien we weten dat alle mannen kunnen praten, betekent dit dat het te zoeken concept *algemener* moet zijn dan *man*. Indien we weten dat planten niet kunnen praten, mag het te zoeken concept *niet algemener* zijn dan *levenloos*. Van bepaalde concepten kan dus gegeven zijn dat ze al dan niet een *ondergrens* vormen voor het te zoeken concept, met andere woorden of ze een *positieve ondergrens* dan wel een *negatieve ondergrens* zijn voor het te zoeken concept. Voor een positieve bovengrens moet het te zoeken concept dan alle instanties dekken die de positieve ondergrens dekt (zie Figuur 2.a). Voor een negatieve ondergrens moet het minstens één instantie *niet* dekken, die de negatieve ondergrens wel dekt (zie Figuur 2.b).



a. Positieve Ondergrens

b. Negatieve Ondergrens

c. Positieve Bovengrens

d. Negatieve Bovengrens

Figuur 2: Soorten informatie-elementen

Duaal kan van bepaalde concepten gegeven zijn dat ze een *positieve of negatieve bovengrens* zijn voor het te zoeken concept. Bijvoorbeeld, er kan gegeven zijn dat alle pratende wezens levende wezens zijn (m.a.w. alleen levende wezens kunnen praten); dit wil zeggen dat *levend* een positieve bovengrens is: *levend* is algemener dan het te zoeken concept. Alleen instanties die gedekt worden door de positieve bovengrens mogen gedekt worden door het te zoeken concept (zie Figuur 2.c). Ook kan gegeven zijn dat niet alle pratende wezens vrouwen zijn (m.a.w. het zijn niet alleen vrouwen die praten); dit wil zeggen dat *vrouw* een negatieve bovengrens is: *vrouw* mag dan niet algemener zijn dan het te zoeken concept. Of met andere woorden: er moet minstens één instantie zijn die gedekt wordt door het te zoeken concept die *niet* gedekt wordt door de negatieve bovengrens (zie Figuur 2.d). ◇

Men kan de relatie "algemener dan" uitbreiden naar instanties, door te definieren dat een concept algemener is dan een instantie, indien het concept de instantie dekt. Indien $\mathcal{L}_I$ een deelverzameling zou zijn van $\mathcal{L}_C$ (zodat "algemener dan" al gedefinieerd was), levert dit geen inconsistenties op indien de *single-representation trick* [Cohen en Feigenbaum, 1981] geldt. Op die manier kunnen we ook instanties als ondergrenzen beschouwen.
Van elk gegeven informatie-element weten we of het een positieve of negatieve onder- of bovengrens is voor het te zoeken concept. Bijgevolg zullen kandidaat-oplossingen ook aan deze voorwaarde moeten voldoen. We noemen een concept c daarom *consistent* met een positieve ondergrens i, indien c algemener is dan i; consistent met een negatieve ondergrens,

indien c niet algemener is dan i; consistent met een positieve bovergrens, indien i algemener is dan c; en consistent met een negatieve bovergrens, indien i niet algemener is dan c (zie opnieuw Figuur 2).
De uitgebreide probleemstelling wordt gegeven in Probleem 1. We veronderstellen hier

Gegeven:

- een taal $\mathcal{L}_C$ van concept-beschrijvingen ;
- een taal $\mathcal{L}_I$ van voorbeeld-beschrijvingen ;
- de achtergrondkennis B;
- een relatie $\preccurlyeq : (\mathcal{L}_I \cup \mathcal{L}_C \times \mathcal{L}_I \cup \mathcal{L}_C) \rightarrow \{ \mathit{true}, \mathit{false} \}$;
- een verzameling I van *informatie elementen*: I bevat positieve ondergrenzen, negatieve ondergrenzen, positieve bovengrenzen en negatieve bovengrenzen van een doel-concept-beschrijving t;

Vind: een element $h \in \mathcal{L}_C$, indien er een bestaat, zodat h consistent is met alle elementen van I. h wordt een *hypothese* genoemd.

Probleem 1: Het leren van concepten

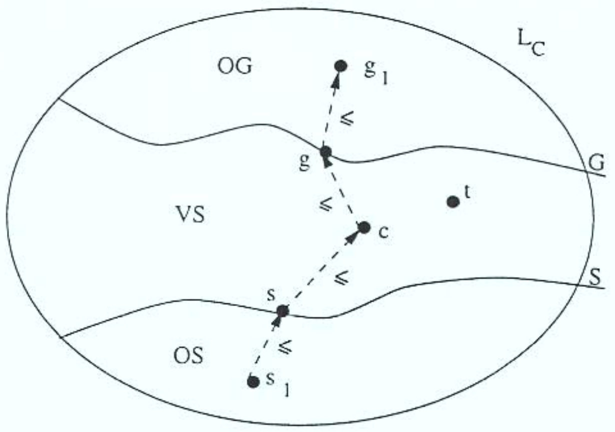
dat de gegeven informatie-elementen correct zijn, d.w.z. dat ze geen *ruis (Eng. noise)* bevatten.
De orde "algemener dan" laat vooral ook toe $\mathcal{L}_C$ systematisch te doorzoeken. Dit betekent dat alle elementen van $\mathcal{L}_C$ liefst maar één maal worden beschouwd als kandidaat-oplossing (dit noemen we een *optimaal* zoekproces), en dat bepaalde delen van $\mathcal{L}_C$ kunnen worden gesnoeid tijdens het zoekproces. Om te snoeien kunnen we de boven- en ondergrenzen in twee groepen onderverdelen: positieve ondergrenzen en negatieve bovengrenzen noemen we s-grenzen; negatieve ondergrenzen en positieve bovengrenzen noemen we g-grenzen. Snoeien gebeurt dan met de volgende twee regels:

- indien een concept x niet consistent is met een s-grens i, dan zal elk concept x' *specifieker* dan x evenmin consistent zijn met i; bijgevolg kan x' gesnoeid worden.
- indien een concept x niet consistent is met een g-grens i, dan zal elk concept x' *algemener* dan x evenmin consistent zijn met i; bijgevolg kan x' gesnoeid worden.

Voorbeeld Indien *mannelijk* wordt verworpen als oplossing, omdat het niet consistent is met de positieve ondergrens *hilde*, kunnen ook onmiddellijk *man* en $\perp$ worden verworpen omdat ze specifieker zijn dan *mannelijk*. Zo kan ook $\top$ worden verworpen, van zodra is gevonden dat *levend* de negatieve ondergrens *tweety* dekt. $\diamond$

3.1 Iteratieve Versieruimten

Naargelang het probleem kunnen verschillende aspecten moeten worden ingevuld.



Figuur 3: Over-algemene, over-specifieke en consistente elementen in $\mathcal{L}_C$

- het probleem kan gesteld zijn als een incrementeel probleem: hoewel de verzameling I nadien nog kan worden uitgebreid, moeten er toch al bepaalde problemen opgelost kunnen worden met de tot nu toe geleerde kennis. In dit geval moet een hypothese gezocht worden die consistent is met alle tot nu gekende informatie-elementen. Deze hypothese kan dan al gebruikt worden om nieuwe problemen op te lossen. Het afwisselen van leer-fasen met probleem-oplossings-fasen vereist dat een hypothese die consistent was met alle voorheen gekende informatie-elementen in I, maar niet consistent is met een pas gekend informatie-element i, moet kunnen aangepast worden tot een hypothese consistent met alle elementen van I en met i, zonder het zoekproces van vooraf aan te moeten beginnen. In deze thesis hebben we ons beperkt tot incrementele leer-algoritmen.

- men kan $\mathcal{L}_C$ *specifiek naar algemeen*, *algemeen naar specifiek* of in beide richtingen tegelijkertijd doorzoeken. In deze thesis bespreken we alleen bi-directionele algoritmen. Dit heeft als voordeel dat bij het oplossen van problemen zowel voor een maximaal algemene oplossing als voor een maximaal specifieke oplossing kan gekozen worden, en het laat toe *relevante* informatie-elementen automatisch te genereren.

Gegeven een probleemstelling als Probleem 1, dan is de oplossingsruimte de *versieruimte (Eng. Versionspace)* $\mathcal{VS}$ van alle consistente concepten in $\mathcal{L}_C$. Op voorwaarde dat $\mathcal{L}_C$ geen oneindige dalende of stijgende ketens bevat voor de relatie $\preccurlyeq$, en dat $\mathcal{L}_C$ convex is, kan een versieruimte worden voorgesteld door middel van de verzameling $\mathcal{G}$ van alle meest algemene elementen in $\mathcal{VS}$, en de verzameling $\mathcal{S}$ van alle meest specifieke elementen in $\mathcal{VS}$: er bestaat dan immers voor elke consistente $c \in \mathcal{L}_C$ een $s \in \mathcal{S}$ en een $g \in \mathcal{G}$ zodat $s \preccurlyeq c \preccurlyeq g$ [Mitchell, 1982]. Figuur 3 stelt deze situatie voor: de taal $\mathcal{L}_C$ wordt verdeeld in drie delen: de verzameling OG van over-algemene concepten (d.w.z. concepten die inconsistent zijn met minstens één g-grens), de verzameling OS van over-specifieke concepten (d.w.z. concepten die inconsistent zijn met minstens één s-grens), en de verzameling $\mathcal{VS}$ van alle concepten consistent met alle g-grenzen en alle s-grenzen. Aan de bovenste rand van $\mathcal{VS}$ bevindt zich

$\mathcal{G}$; aan de onderste rand van $\mathcal{VS}$ bevindt zich $\mathcal{S}$.

[Mellish, 1991] stelt het "Description Identification" algoritme (DI) voor: een bi-directioneel breedte-eerst programma dat $\mathcal{G}$ en $\mathcal{S}$ berekent. Een eerste voordeel van het volledig berekenen van $\mathcal{G}$ en $\mathcal{S}$ is dat alle informatie over de concepten consistent met I vervat zit in $\mathcal{G}$ en $\mathcal{S}$, en de informatie-elementen dus niet hoeven opgeslagen te worden. Een tweede voordeel is dat, op voorwaarde dat het te zoeken concept tot $\mathcal{L}_C$ behoort, sommige niet-gegeven instanties reeds *met zekerheid* als positief of negatief kunnen geclassificeerd worden: een instantie i die gedekt wordt door alle elementen in $\mathcal{S}$ zal zeker ook gedekt worden door het te zoeken concept; een instantie i dat door geen enkel element van $\mathcal{G}$ wordt gedekt, kan in geen geval door het te zoeken concept gedekt worden. Het nadeel van deze aanpak ligt in het feit dat de grootte van $\mathcal{G}$ en $\mathcal{S}$ in het slechtste geval een exponentiële functie van het aantal elementen in I kan zijn [Haussler, 1988]. Een tweede nadeel is dat ook de tijd om $\mathcal{G}$ en $\mathcal{S}$ voor ieder nieuw informatie-element te berekenen in het slechtste geval een exponentiële functie is van het aantal informatie-elementen.

Het *Iterative Versionspace* algoritme (ITVS) [Sablon *et al.*, 1994] is een bi-directioneel diepte-eerst algoritme om Probleem 1 op te lossen. Het voordeel van ITVS is dat het geheugen-verbruik in het slechtste geval een *lineaire* functie kan zijn van het aantal elementen in I. ITVS moet daartoe wel alle informatie-elementen bijhouden. ITVS berekent ook niet telkens heel $\mathcal{G}$ en $\mathcal{S}$ (wat ook niet is gevraagd in Probleem 1) maar wel slechts één maximaal algemeen element $g \in \mathcal{G}$, en één maximaal specifiek element $s \in \mathcal{S}$. Bovendien houdt ITVS in de stapels B_s en B_g *terugkeer*-informatie *(Eng. backtrack information)* bij. Deze terugkeer-informatie bestaat uit *keuzepunten* waar het diepte-eerst algoritme een keuze heeft moeten maken welk alternatief eerst te onderzoeken, en welke alternatieven later te onderzoeken. Aan de hand van de terugkeer-informatie kunnen dan ook alle alternatieven berekend worden, indien nodig.

ITVS kan, gegeven een nieuw informatie-element, het huidig maximaal algemeen element g aanpassen in een tijd lineair in het aantal informatie-elementen, afgezien van het terugkeren *(Eng. backtracken)* doorheen een ruimte met grootte exponentieel in het aantal informatie-elementen. Om echter ongeziene informatie-elementen *met zekerheid* te classificeren is nog steeds de volledige verzameling $\mathcal{G}$ of $\mathcal{S}$ nodig. Beide verzamelingen kunnen, indien nodig, in het kader van ITVS berekend worden. De tijd die daarvoor nodig is, is in het slechtste geval dan wel een lineaire factor slechter dan de tijd van DI.

3.2 Compactie van I

Vermits ITVS als dusdanig alle informatie-elementen expliciet moet bijhouden, en vermits ook de rekentijd van ITVS een functie is van het aantal informatie-elementen, is het belangrijk zo weinig mogelijk informatie elementen te *moeten* bijhouden. Dit wil zeggen, dat we, ten eerste, geen overbodige informatie-elementen wensen bij te houden. Hiermee bedoelen we informatie-elementen met een kleinere informatie-inhoud dan een ander informatie-element. Ten tweede zullen we ook trachten paren informatie-elementen te vervangen door één enkel informatie-element met dezelfde informatie-inhoud [Sablon en De Raedt, 1995]. We zullen deze operaties illusteren aan de hand van voorbeelden.

3.2.1 Overbodige informatie-elementen

Voorbeeld Stel dat *man* en *hendrik* twee positieve ondergrenzen zijn voor het te leren concept. Het concept *man* dekt de instantie *hendrik*; eisen dat *man* een ondergrens is voor een hypothese c, komt neer op eisen dat alle instanties gedekt door *man* ook gedekt worden door c, dus in het bijzonder dat *hendrik* gedekt wordt door c. Van zodra *man* een positieve ondergrens is, is *hendrik* daarom ook een positieve ondergrens. Dit betekent dat *hendrik* niet hoeft bijgehouden te worden. ◇

In het algemeen hoeft men van twee positieve ondergrenzen waarvan de ene algemener is dan de andere, steeds slechts de meest algemene bij te houden. Hetzelfde geldt voor negatieve bovengrenzen.
Negatieve ondergrenzen hebben een duaal karakter: van twee negatieve ondergrenzen waarvan de ene algemener is dan de andere, moet men slechts de meest specifieke bijhouden. Hetzelfde geldt voor positieve bovengrenzen.

Voorbeeld Stel dat het concept *vrouwelijk* en de instantie *tweety* (een vrouwelijke vogel), beide negatieve ondergrenzen zijn voor het te leren concept. Eisen dat *vrouwelijk* een negatieve ondergrens is voor een hypothese, komt neer op eisen dat er een negatieve instantie bestaat die gedekt wordt door *vrouwelijk*. Bijgevolg zal eisen dat *tweety* een negatieve ondergrens is, impliceren dat *vrouwelijk* een negatieve ondergrens is. Dit betekent dat *vrouwelijk* niet hoeft bijgehouden te worden. ◇

Er is nog een derde geval van overbodige informatie-elementen.

Voorbeeld Stel dat *levend* een positieve bovengrens is, en dat "de appel van Newton" een negatieve ondergrens is. De instantie "de appel van Newton" wordt niet gedekt door *levend*, maar wel door *levenloos*. Eisen dat *levend* een positieve bovengrens is voor een hypothese c, heeft dan tot gevolg dat c onmogelijk "de appel van Newton" kan dekken. Bijgevolg hoeven we "de appel van Newton" niet bij te houden. ◇

Dit voorbeeld toont aan dat negatieve ondergrenzen die *niet* specifieker zijn dan elke positieve bovengrens, overbodig zijn. Duaal zijn negatieve bovengrenzen die *niet* algemener zijn dan elke positieve ondergrens overbodig.

3.2.2 Vervangen van informatie-elementen

In een volgende stap kunnen we ook twee gegeven positieve informatie-elementen vervangen door één informatie-element met dezelfde informatie-inhoud. We bespreken twee gevallen: ten eerste, hoe en wanneer twee positieve informatie-elementen kunnen vervangen worden door één ander positief informatie-element, en ten tweede, hoe en wanneer één negatief informatie-element kan vervangen worden door een positief informatie-element.

Voorbeeld Stel dat *vrouw* en *man* beide positieve ondergrenzen zijn. Merk op dat *mens* de enige veralgemening is van *vrouw* en *man* in $\mathcal{L}_C$, die maximaal specifiek is. Daarom is eisen dat *vrouw* en *man* positieve ondergrenzen zijn equivalent met eisen dat *mens* een positieve ondergrens is. Immers, een hypothese c consistent met *vrouw* en consistent met *man*, zal algemener moeten zijn dan *vrouw* en *man*. De maximaal specifieke van zulke veralgemeningen is *mens*, en deze is uniek. Dus zal c algemener moeten zijn dan *mens*. Dit betekent dat *mens* een positieve ondergrens is. ◇

In het algemeen kunnen we twee positieve ondergrenzen die slechts één maximaal specifieke veralgemening hebben, weglaten door te eisen dat deze maximaal specifieke veralgemening een positieve ondergrens is.
Duaal, kunnen we ook twee positieve bovengrenzen die slechts één maximaal algemene specializatie hebben, weglaten door te eisen dat deze maximaal algemene specializatie een positieve bovengrens is.
Ten tweede bespreken we hoe we één negatieve ondergrens kunnen vervangen door één positieve bovengrens. Op zich levert dit geen winst op, maar de positieve bovengrens kan eventueel tesamen met een andere positieve bovengrens vervangen worden door hun meest algemene specializatie.

Voorbeeld Stel dat we moeten zoeken welke wezens rokjes dragen, dat *vrouw* een positieve ondergrens is, en dat *mens* een positieve bovengrens is. Stel nu dat de negatieve ondergrens *jongen* gegeven is. Dan is er maar één meest algemeen concept dat algemener is dan *vrouw* en toch niet algemener is dan *jongen*, namelijk *vrouwelijk*. Dit betekent dat de negatieve ondergrens *jongen* door de positieve bovengrens *vrouwelijk* kan vervangen worden. In dit specifieke geval zijn er twee maximaal algemene specializaties van de positieve bovengrenzen *mens* en *vrouwelijk*, zodat ze niet kunnen vervangen worden door één andere. ◇

Net zoals het kader dat geschetst wordt voor ITVS, is deze aanpak onafhankelijk van de gekozen taal. Bovendien zijn de eigenschappen van redundante informatie-elementen ook onafhankelijk van ITVS geformuleerd. Dat maakt ze toepasbaar bij elk algoritme dat informatie-elementen expliciet moet bijhouden. Bij nader inzien zijn verschillende van deze aspecten in de context van attribuut-waarde talen ook reeds impliciet toegepast. Zo kunnen, bijvoorbeeld, de resultaten van het *Incremental Non-Backtracking Focussing* algoritme van [Smith en Rosenbloom, 1990] ook in deze context worden geïnterpreteerd. De negatieve ondergrenzen die kunnen omgezet worden naar positieve bovengrenzen, worden in een attribuut-waarde context *near-miss* genoemd [Winston, 1975].

3.3 Disjunctieve Iteratieve Versieruimten

De keuze van de concept-taal is zeer belangrijk bij het leren van concepten. Indien de taal niet toelaat het te zoeken concept voor te stellen, kan ITVS (of een leersysteem in het algemeen) geen oplossing vinden voor Probleem 1. Het bepalen van een taal die voldoende concepten bevat, is dus een belangrijk probleem. Dit probleem kan gedeeltelijk opgelost worden door *automatisch* de te gebruiken taal te veranderen, bijvoorbeeld door ze uit te breiden (dit noemt men *shift of bias* [Utgoff, 1986], [De Raedt, 1992]).

Voorbeeld De taal $\mathcal{L}_C$ uit de vorige voorbeelden zou kunnen uitgebreid worden met nieuwe concepten als *volwassene*, *kind*, enz. De relatie "algemener dan" moet ook uitgebreid worden met betrekking tot deze concepten: *man* en *vrouw* zijn specifieker dan *volwassene*, *volwassene* is specifieker dan *mens*, enz. Met de uitgebreide taal kunnen dan meer concepten worden uitgedrukt dan met de oorspronkelijke taal. Merk op dat de gekozen uitbreiding *domein-afhankelijk* is (het concept *volwassene* is niet zomaar in elke taal nuttig als uitbreiding), en dus is ze ook *taal-afhankelijk*. ◇

Een mogelijke *taal-onafhankelijke* uitbreiding, die in vele leer-programma's wordt gebruikt, is het "combineren" van concepten uit $\mathcal{L}_C$ tot nieuwe concepten. Zulk een "combinatie" van concepten dekt een instantie enkel en alleen indien *één van de concepten* de instantie dekt. Uit het volgende voorbeeld blijkt waarom deze combinaties klassiek *disjuncties* worden genoemd. De elementen van een disjunctie noemen we *disjuncten*.

Voorbeeld Veronderstel dat we opnieuw moeten bepalen welke wezens rokjes dragen. Gegeven zijn de positieve instanties *hilde* en *liesje*. De instantie *liesje* wordt gedekt door het concept *meisje*. Ook gegeven zijn de negatieve instanties *wim* en *tweety*. Een leerprogramma (bijvoorbeeld ITVS) dat een oplossing zoekt in $\mathcal{L}_C$, faalt: na de eerste twee informatie-elementen zijn er twee mogelijke maximaal specifieke concepten: *mens* en *vrouwelijk*. De eerste mogelijkheid dekt echter ook de negatieve instantie *wim*; dus dekken ook alle veralgemeningen van *mens* deze negatieve instantie. De tweede mogelijkheid dekt ook de negatieve instantie *tweety*; dus dekken ook alle veralgemeningen van *vrouwelijk* deze negatieve instantie. Bijgevolg is er geen oplossing.

Beschouw nu de disjunctie van *meisje* en *vrouw* (genoteerd door *meisje* $\vee$ *vrouw*) als een nieuw concept. Het concept *meisje* $\vee$ *vrouw* dekt alle instanties die door *meisje of* door *vrouw* worden gedekt. Dan dekt *meisje* $\vee$ *vrouw* beide positieve instanties *hilde* en *liesje*, maar geen van de negatieve instanties *wim* en *tweety*. ◇

Net als in Sectie 3.1 volgt dan ook hoe disjuncties zich verhouden volgens de relatie "algemener dan". Deze relatie laat opnieuw toe de verzameling van de disjuncties op een gestructureerde manier te doorzoeken. Nochtans beperkt men zich dikwijls tot het zoeken in $\mathcal{L}_C$ naar elke disjunct afzonderlijk. Dit kan niet zonder meer taal-onafhankelijk gebeuren. Het gaat alleen als voor alle c_1, c_2 en c_3 in $\mathcal{L}_C$ geldt dat $c_1 \preccurlyeq c_2 \vee c_3$ impliceert dat c_1 meer specifiek is dan c_2 of meer specifiek is dan c_3.
Deze beperking geldt in het gebied van Inductief Logisch Programmeren (zoals het wordt besproken in Sectie 4) voor niet-recursieve concepten. A fortiori geldt ze dan ook voor attribuut-waarde voorstellingen, omdat deze ook als propositionele, niet-recursieve voorstellingen kunnen beschouwd worden.

Voorbeeld Het concept *man* is specifieker dan de disjunctie *mannelijk* $\vee$ *vrouwelijk*, omdat *man* specifieker is dan *mannelijk*. Het concept *meisje* is eveneens specifieker dan *mannelijk* $\vee$ *vrouwelijk*.

Indien een concept *dier* niet specfieker zou zijn dan *mannelijk*, noch specifieker dan *vrouwelijk*, toch specifieker zou zijn dan *mannelijk* $\vee$ *vrouwelijk*, en *tot $\mathcal{L}_C$ zou kunnen behoren*, dan zou de beperking niet opgaan. ◇

Door gebruik te maken van deze beperking kan men de definitie van consistentie uitbreiden naar disjuncties: een disjunctie van concepten is consistent met een s-grens enkel en alleen indien *één van de concepten* consistent is met de s-grens; een disjunctie van concepten is consistent met een g-grens enkel en alleen indien *elk van de concepten* consistent is met de g-grens. Merk op dat het toevoegen van disjuncten een concept *algemener* maakt, en dat het weglaten van disjuncten een concept *specifieker* maakt.

Voorbeeld De disjunctie *meisje* $\vee$ *vrouw* is consistent met de positieve ondergrens *liesje* omdat *meisje* consistent is met *liesje*; de disjunctie is consistent met de positieve ondergrens *hilde* omdat *vrouw* consistent is met *hilde*; de disjunctie is consistent met de negatieve ondergrenzen *tweety* en *wim*, omdat *elke disjunct* consistent is met *tweety* en met *wim*. ◇

Omdat $c_1 \vee c_2$ niet meer instanties dekt dan c_1, indien c_1 algemener is dan c_2, is het niet nuttig de disjunctie $c_1 \vee c_2$ toe te laten. In het algemeen noemen we een disjunctie gereduceerd, indien geen enkele van zijn disjuncten algemener is dan een andere van zijn disjuncten. In het vervolg werken we alleen met gereduceerde disjuncties.
Hoewel het invoeren van disjuncties toelaat meer concepten te beschrijven, is het op zich niet praktisch bruikbaar om automatisch concepten te leren, omdat de grootte van de versieruimte van alle disjuncties consistent met een gegeven aantal informatie-elementen in het algemeen combinatorisch explosief is. In deze versieruimte is er één maximaal algemeen element, namelijk de disjunctie van alle elementen van $\mathcal{L}_C$ consistent met alle g-grenzen, en zijn er relatief veel maximaal specifieke elementen.

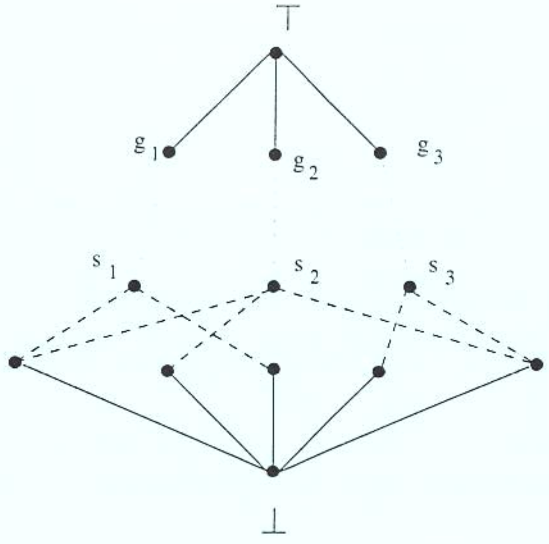
Voorbeeld Nemen we opnieuw het concept "kan praten". Gegeven de positieve ondergrenzen *hilde*, *wim* en *liesje*. De instantie *tweety* is een negatieve ondergrens. Dan is de disjunctie *mannelijk* ∨ *mens* ∨ *levenloos* de meest algemene consistente disjunctie consistent met alle gegeven informatie-elementen. Dat deze disjunctie consistent is met alle informatie-elementen is eenvoudig te controleren; dat ze ook maximaal algemeen is, komt door het feit dat geen enkele van de disjuncten kan veralgemeend worden zonder *tweety* te dekken, en geen enkele andere disjunct kan toegevoegd worden, en toch een gereduceerde disjunctie te bekomen.

De enige meest specifieke disjunctie is *man* ∨ *vrouw* ∨ *meisje*. Deze disjunctie is consistent met alle informatie-elementen. Geen enkele disjunct kan echter weggelaten worden, of meer specifiek gekozen worden. ◇

Zoals dit voorbeeld aantoont, bevat de resulterende versieruimte tal van disjuncties die ongewenst zijn: het te zoeken concept was net zoals hoger *mens*. Door het toelaten van disjuncties bevatten de meeste oplossingen echter disjuncties, en wordt er onvoldoende veralgemeend en gespecializeerd. Om al de ongewenste disjuncties uit te sluiten zou men haast voor elke mogelijke disjunctie een boven- of ondergrens moeten vinden die er niet consistent mee is. Daarom moeten we op één of andere manier aangeven dat we niet alle disjuncties willen beschouwen, en op één of andere manier het aantal disjuncten beperken. In een eerste stap beschrijven we hoe we, vertrekkend van de maximaal algemene disjunctie meer interessante consistente deel-disjuncties kunnen construeren. In een tweede stap zullen we door middel van een extra *voorkeur-criterium* de verzameling van de meest gewenste deel-disjuncties selecteren.

3.3.1 Deel-disjuncties en bijna maximaal specifieke disjuncties

In een eerste stap beschouwen we van de maximaal algemene disjunctie g_{max} alleen *deel-disjuncties* die consistent zijn met alle s-grenzen; een deel-disjunctie is een disjunctie waarvan de disjuncten een deelverzameling vormen van de disjuncten van g_{max}. De gekozen deel-disjuncten zullen dan consistent zijn met alle informatie-elementen. Van een maximaal specifieke disjunctie kunnen we geen disjuncten weglaten, en toch consistent blijven met alle s-grenzen. De enige manier om daar het aantal disjuncten te beperken is het veralgemenen van twee of meer disjuncties tot een nieuwe disjunct. De keuze welke disjuncties



Figuur 4: Bijna maximaal specifieke disjuncties

te veralgemenen, is niet triviaal: in principe komen alle partities van de disjuncten in aanmerking. Daarom voeren we een nieuw begrip in: de *bijna maximaal specifieke* disjuncties onder een disjunctie g. Een *bijna maximaal specifieke disjunctie onder een disjunctie* $g_1 \vee \cdots \vee g_n$ is een disjunctie $s_1 \vee \cdots \vee s_n$ zodat voor elke j, $1 \leq j \leq n$, g_j algemener is dan s_j, s_j maximaal specifiek is en consistent met *alle* s-grenzen waarmee g_j consistent is (zie Figuur 4). Op die manier wordt het probleem welke disjuncties van de maximaal specifieke disjunctie moeten gecombineerd worden opgelost: er zullen nog precies n disjuncten overblijven.

Het nadeel van deze methode is dat de resulterende disjuncties *te specifiek* kunnen zijn. Dit zal blijken wanneer een nieuwe s-grens bekend wordt die met geen enkele disjunct van g consistent is. In dit geval is de gekozen deel-disjunctie, noch zijn specializaties een oplossing. Ook kunnen de resulterende disjuncties *te algemeen* zijn. Dit zal blijken wanneer een nieuwe g-grens bekend wordt, die met minstens één van de disjuncten van s inconsistent is. In dat geval zullen ook de overeenkomstige disjuncten in g inconsistent zijn, en dus ook de disjuncten van g_{max}. Daarom moeten de over-algemene disjuncten van g_{max} eerst worden gespecializeerd. Daarna kunnen deel-disjuncties van deze nieuwe maximaal algemene disjunctie worden beschouwd, tesamen met de bijhorende bijna maximaal specifieke disjuncties.

Voorbeeld In het vorige voorbeeld kunnen we de volgende deel-disjuncties van $g = mannelijk \vee mens \vee levenloos$ worden beschouwd: *mens* en $mens \vee levenloos$. De andere (echte) deel-disjuncties (d.w.z. *mannelijk*, *levenloos*, $mannelijk \vee mens$, en $mannelijk \vee levenloos$) zijn niet consistent met alle s-grenzen. Men kan opmerken dat de tweede disjunct van $mens \vee levenloos$ met geen enkele s-grens consistent is, en daarom eveneens zou kunnen weggelaten worden (zie verder). De disjunct *mens* is consistent met alle s-grenzen. Een bijna algemeen specifieke disjunctie onder g moet dan eveneens een disjunct bevatten die met alle s-grenzen consistent is. De enige mogelijkheid voor s is dan eveneens *mens*. Het resultaat is dan $g = mens$ en $s = mens$.

Wanneer zou blijken dat de instantie *hansje*, die gedekt is door het concept *jongen*, een negatieve ondergrens is, volgt dat *mens* te algemeen is. Een mogelijke specializatie van *mens* die wel consistent is met alle informatie-elementen is de disjunctie $man \vee vrouw \vee meisje$. De enige bijna maximaal specifieke disjunctie onder $man \vee vrouw \vee meisje$ is dan $man \vee vrouw \vee meisje$ zelf. ◇

3.3.2 Bijkomende voorkeur-criteria

Een andere manier om toch de gewenste disjuncties als oplossing te bekomen is een bijkomend *voorkeur-criterium* te specifiëren. In het algemeen specificeert een voorkeur-criterium welke concepten verkozen worden boven andere concepten. Meestal bevat het voorkeur-criterium een notie van *minimaliteit*. Twee mogelijke criteria die in deze thesis worden onderzocht zijn het criterium van *minimale lengte* (ML) en het criterium van de *minimale verzameling* (MS). Een consistente disjunctie noemen we ML, indien er geen consistente disjunctie bestaat met minder disjuncten. Een consistente disjunctie noemen we MS, indien er geen enkele deel-disjunctie bestaat die ook nog consistent is.

Voorbeeld In het vorige voorbeeld kon de disjunct *levenloos* uit de disjunctie $mens \vee levenloos$ worden weggelaten omwille van beide criteria. Enerzijds bestaat er een consistente disjunctie met één disjunct (namelijk *mens*); anderzijds is *mens* ook een deel-disjunctie, die toch nog consistent is met alle informatie-elementen.

◇

We hebben dan het volgende resultaat: de verzameling van alle consistente disjuncties die voldoen aan het MS (resp. ML) voorkeur-criterium is de verzameling van alle disjuncties d waarvoor geldt dat:

- d is specifieker dan een deel-disjunctie g van g_{max}, die voldoet aan het MS (resp. ML) criterium;
- d is algemener dan een bijna maximaal specifieke disjunctie s onder g.

DDI is een disjunctieve uitbreiding van DI: het berekent in een eerste stap alle disjuncten van de maximaal algemene disjunctie, en alle overeenkomstige maximaal specifieke disjuncten. In een tweede stap berekent DDI alle combinaties van deze disjuncten die consistent zijn met alle s-grenzen, en de overeenkomende bijna maximaal specifieke disjuncties onder deze deel-disjuncties.

DITVS is een disjunctieve uitbreiding van ITVS: het berekent *één* maximaal algemene disjunctie, die voldoet aan het MS-criterium, en een overeenkomende bijna maximaal specifieke disjunctie. DITVS kan aangepast worden zodat de oplossing ML is.

De voor- en nadelen van DITVS t.o.v. DDI zijn analoog aan de voor- en nadelen van ITVS t.o.v. DI: enerzijds is er een besparing van geheugen door het niet expliciet bijhouden van *alle* deel-disjuncties van de maximaal algemene disjunctie; anderzijds kunnen disjuncten die werden verworpen als disjunct voor de ene disjunctie, later eventueel opnieuw berekend moeten worden als deel van een andere disjunctie.

4 Iteratieve Versieruimten en Inductief Logisch Programmeren

4.1 Inleiding

In Inductief Logisch Programmeren (ILP) tracht men inductieve leer-problemen op te lossen door gebruik te maken van logische voorstellingen. In die zin is ILP te situeren in de doorsnede van de onderzoeksgebieden automatisch leren en *Logisch Programmeren (Eng. Logic Programming)*. Op dit ogenblik stelt men in ILP concepten hoofdzakelijk voor door middel van predicatenlogica. Predicatenlogica is expressiever dan attribuut-waarde voorstellingen, omdat deze laatste in wezen propositioneel zijn.
In het bijzonder is het leren van concepten, voorgesteld door middel van een logisch formalisme, bestudeerd in ILP. In die context wordt dit dan *leren van predicaten* genoemd.
In deze sectie tonen we hoe het algemene kader voor het leren van concepten uit Sectie 3 kan worden toegepast in ILP. Daartoe zullen we de verschillende elementen van dit kader verder specifiëren: de instantie-taal, de concept-taal en de relatie "dekt".
Het belang van deze sectie is dan tweeërlei. Enerzijds tonen we aan dat we op die manier het leren van predicaten kunnen beschouwen als een specifieke vorm van het leren van concepten, zodat alle algemene eigenschappen van Sectie 3 onmiddellijk toepasbaar zijn. Anderzijds tonen we hiermee ook aan dat het kader van Sectie 3 ook effectief praktisch kan toegepast worden.

4.2 De instantie-taal en de concept-taal

Concepten komen in ILP overeen met *predicaten*. De instanties van een concept worden meestal voorgesteld door feiten van het overeenkomstige predicaat. De concepten zelf worden voorgesteld door middel van *definiete clausules (Eng. definite clauses)* met het overeenkomstige predicaat in het hoofd. Een definiete clausule is van de vorm $h \leftarrow b_1 , \ldots , b_n$. Hierin zijn h en $b_1 , \ldots , b_n$ atomen *(Eng. atom*, of ook *positive literal)*. Het atoom h is het hoofd van de clausule; de atomen $b_1 , \ldots , b_n$ vormen het lichaam. Deze atomen kunnen ook veranderlijken bevatten; alle veranderlijken in een clausule zijn impliciet *universeel gekwantifieerd*. Een clausule kan ook beschouwd worden als een verzameling van de er in voorkomende atomen, waarbij een strikt onderscheid wordt gemaakt tussen de atomen in het hoofd en die in het lichaam. In het lichaam van deze clausules worden concepten uit de achtergrondkennis gebruikt. In een ILP context betekent dit dat deze predicaten reeds zijn gedefinieerd, en dat hun definities tot de achtergrondkennis behoren.

Voorbeeld In het voorbeeld dat we in de vorige secties hebben ontwikkeld kunnen we nu ook concepten uitdrukken die relaties voorstellen tussen verschillende wezens. Zo zijn er de relaties *ouder*, *vader*, *moeder*, enz. We gaan er steeds vanuit dat we een van de concepten moeten leren, d.w.z. uitdrukken in termen van de andere. Die andere worden op dat ogenblik als volledig gekend beschouwd (zie Sectie 2), d.w.z. zij behoren tot de achtergrondkennis bij het leren.

Stel bijvoorbeeld dat het concept *ouder* is gekend, en dat we (de definitie van) het concept *vader* moeten uitdrukken in termen van de predicaten uit de tralie $\mathcal{M}$, en het predicaat *ouder*. Mogelijke concept-definities zijn bijvoorbeeld:

$$vader(\ X\ ,\ Y\ ) \leftarrow levend(\ X\ ), mens(\ Y\ ),$$

wat uitdrukt dat elk levend wezen X de vader is van elke mens Y, of

$$vader(\ X\ ,\ Y\ ) \leftarrow mannelijk(\ X\ ), ouder(\ X\ ,\ Y\ ), animate(\ Y\ ),$$

wat uitdrukt dat elke mannelijk levend wezen X dat de ouder is van een levend wezen Y, de vader is van Y. Talrijke andere clausules zijn natuurlijk mogelijk. ◇

4.3 De relatie "dekt"

In ILP is een instantie gedekt door een concept indien de instantie een logisch gevolg is van het concept, tesamen met de achtergrondkennis B. Een concept $h \leftarrow l$ dekt een instantie i, indien het hoofd h *unificeert* met de instantie i, en indien het lichaam l een logisch gevolg is van de achtergrondkennis. Om te testen of een instantie gedekt wordt door een concept zal men in het algemeen een stellingbewijzer (bijvoorbeeld PROLOG) gebruiken. Indien men zich beperkt tot logische voorstellingen zonder functie-symbolen of indien men geen recursieve definities gebruikt, kan men garanderen dat deze test steeds eindigt. Anders kan men dat niet, en is de test eigenlijk maar een benadering van het ideaal.

Voorbeeld Stel dat de volgende feiten tot de achtergrondkennis behoren:

man(wim).	*ouder(wim , liesje).*
meisje(liesje).	*ouder(oliver , tweety).*
mannelijk(oliver).	*ouder(hilde , hansje).*
vrouwelijk(tweety).	

Door middel van de relatie "algemener dan" voorgesteld in de tralie $\mathcal{M}$, kunnen we uit deze feiten een aantal andere logische gevolgen afleiden (zoals *mannelijk(wim)*, *levend(tweety)*, *human(liesje)*, enz.) We veronderstellen dat de atomen die niet op deze manier afleidbaar zijn en die we niet hebben opgesomd (zoals *man(oliver)*, *vrouw(tweety)*, *vrouw(liesje)*, enz.). *geen* logisch gevolg zijn van de achtergrondkennis.

Dan dekt de clausule

$$vader(\ X\ ,\ Y\ ) \leftarrow \quad mannelijk(\ X\ ), ouder(\ X\ ,\ Y\ ), levend(\ Y\ ),$$

de instanties *vader(oliver , tweety)* en *vader(wim , liesje)*, en geen andere, omdat bij andere waarden voor X en Y het lichaam van deze clausule geen logisch gevolg is van de gegeven achtergrondkennis. ◇

Niet alle veranderlijken uit het lichaam moeten ook in het hoofd voorkomen. Omgekeerd beperken we ons wel tot clausules waarvan het hoofd geen veranderlijken bevat die niet in het lichaam voorkomen. Clausules die aan deze laatste voorwaarde voldoen noemen we *beperkt in bereik (Eng. range-restricted).*

Voorbeeld Het concept dat uitdrukt dat iemand een kind heeft (laten we het concept *is_een_ouder* noemen) kan bijvoorbeeld gedefinieerd zijn door de clausule

$$is\_een\_ouder(\ X\ ) \leftarrow ouder(\ X\ ,\ Y\ ).$$

Deze clausule drukt uit dat voor elke X en Y geldt dat *als* X de ouder is van Y, dat *dan* X een ouder is; of met andere woorden: voor elke X geldt dat *als* er een Y bestaat waarvan X de ouder is, dat *dan* X een ouder is.

De clausule

$$bemint(\ X\ ,\ Y\ ) \leftarrow mens(\ X\ )$$

laten we niet toe, omdat ze uitdrukt dat X *elke* Y bemint. Om (minstens) aan te geven wat het *domein* is van deze Y eisen we dat Y ook in het lichaam voorkomt. Een versie van deze clausule die beperkt in bereik is, is bijvoorbeeld:

$$bemint(\ X\ ,\ Y\ ) \leftarrow mens(\ X\ ), mens(\ Y\ )$$

◇

Volgens de definitie van Sectie 3.3 is een disjunctie van concepten in de context van ILP een verzameling van clausules met hetzelfde predicaat in het hoofd. Een disjunctie van clausules dekt een instantie, indien het hoofd van minstens één van de clausules unificeert met de instantie, en indien het lichaam van die clausule een logisch gevolg is van de achtergrondkennis. Net zoals in het taal-onafhankelijke geval, is het gebruik van disjuncties nuttig indien er onvoldoende concepten (dus predicaten) gedefinieerd zijn in de achtergrondkennis.

4.4 De relatie "algemener dan"

De relatie "algemener dan" komt in het ideale geval overeen met de relatie "impliceert". Ook dit zal men weer moeten benaderen door middel van testen die voor logische voorstellingen zonder functie-symbolen of voor niet-recursieve definities correct zijn. Een mogelijke benadering is het gebruik van *veralgemeende subsumptie* [Buntine, 1988] om te testen of de ene clausule algemener is dan de andere. Veralgemeende subsumptie is een veralgemening van θ-subsumptie [Plotkin, 1970], in die zin dat veralgemeende subsumptie rekening houdt met de achtergrondkennis, daar waar θ-subsumptie dit niet doet: θ-subsumptie is een louter syntactische operatie. We zullen zowel θ-subsumptie als veralgemeende subsumptie illustreren aan de hand van een voorbeeld.

Voorbeeld De clausule c_1

$$c_1 : vader(\ X_1\ ,\ Y_1\ ) \leftarrow \ ouder(\ X_1\ ,\ Y_1\ ), mannelijk(\ X_1\ ), levend(\ Y_1\ )$$

is algemener dan clausule c_2

$$c_2 : vader(\ X_2\ ,\ Y_2\ ) \leftarrow \ ouder(\ X_2\ ,\ Y_2\ ), mannelijk(\ X_2\ ), mens(\ X_2\ ), levend(\ Y_2\ )$$

met betrekking tot θ-subsumptie, omdat men de veranderlijken in c_1 zodanig kan instantiëren dat de atomen van c_1 ook in c_2 voorkomen (namelijk $X_1 = X_2$ en $Y_1 = Y_2$). Zo is c_1 ook algemener dan c_2' met betrekking tot θ-subsumptie:

$c_2' : vader(\ wim\ , Y_2\ ) \leftarrow\ ouder(\ wim\ , Y_2\ ), mannelijk(\ wim\ ), mens(\ wim\ ),$
$\qquad levend(\ Y_2\ ).$

Maar c_1 is niet algemener dan c_3 met betrekking tot θ-subsumptie.

$c_3 : vader(\ X_3\ , Y_3\ ) \leftarrow\ ouder(\ X_3\ , Y_3\ ), man(\ X_3\ ), mens(\ X_3\ ),$
$\qquad levend(\ Y_3\ ).$

Immers, het atoom *mannelijk*(X_1) kan voor geen enkele instantiatie van X_1 een element zijn van het lichaam van c_3. Nochtans dekt c_1 alle instanties die door c_3 zijn gedekt. Immers, stel dat voor een bepaalde instantie *vader*(a , b) de atomen *ouder*(a , b), *man*(a), *mens*(a) en *levend*(b) uit het lichaam van c_3 een logisch gevolg zijn van de achtergrondkennis. Dan unificeert *vader*(a , b) ook met het hoofd van c_1. Bovendien was verondersteld dat de atomen *ouder*(a , b) en *levend*(b) uit het lichaam van c_1 een logisch gevolg zijn van de achtergrondkennis. Tenslotte zal ook *mannelijk*(b) een logisch gevolg zijn van de achtergrondkennis, vermits het concept *mannelijk* algemener is dan het concept *man*. Dus is c_1 toch algemener dan c_3.

Om tot dit resultaat te komen, kunnen we veralgemeende subsumptie t.o.v. de achtergrondkennis gebruiken. De clausule c_1 is algemener dan c_3 t.o.v. veralgemeende subsumptie, omdat ten eerste de hoofden van c_1 en c_3 unificeren, en omdat ten tweede alle atomen van het lichaam van c_1 een logisch gevolg zijn van de achtergrondkennis *tesamen met het lichaam van* c_3*, waarin alle veranderlijken zijn geïnstantieerd met nieuwe constanten.* We zullen deze bewerking wat meer in detail bespreken. Het unificeren van de hoofden van c_1 en c_3 heeft tot gevolg dat X_1 met X_3 wordt geünificeerd, en Y_1 met Y_3 wordt geünificeerd. Het instantiëren van alle veranderlijken van het lichaam van c_3 met nieuwe constanten sk_x en sk_y levert de volgende atomen:

$$A_1 = \{ouder(\ sk_x\ , sk_y\ ), man(\ sk_x\ ), mens(\ sk_x\ ), levend(\ sk_y\ )\}.$$

Door deze instantiatie is het lichaam van c_1 eveneens geïnstantieerd tot

$$A_3 = \{ouder(\ sk_x\ , sk_y\ ), mannelijk(\ sk_x\ ), levend(\ sk_y\ )\}.$$

Welnu, de atomen uit A_3 zijn een logisch gevolg van de achtergrondkennis tesamen met de atomen uit A_1. Immers, *ouder*(sk_x , sk_y) en *levend*(sk_y) zijn elementen van A_1. Uit het feit dat het concept *mannelijk* algemener is dan het concept *man* (dit zit in de achtergrondkennis) volgt dat *mannelijk*(sk_x) een logisch gevolg is van de achtergrondkennis tesamen met A_1, indien *man*(sk_x) een logisch gevolg is van de achtergrondkennis tesamen met A_1. Welnu, *man*(sk_x) behoort tot A_1, en is er dus een logisch gevolg van. ◇

In het algemeen is een clausule c_1 een veralgemening van een clausule c_2 met betrekking tot θ-subsumptie, indien c_1 (beschouwd als verzameling van atomen) zodanig geïnstantieerd kan worden dat het een deelverzameling is van c_2. Merk ook op dat veralgemeende subsumptie steeds is gedefinieerd *ten opzichte van* een logisch programma P (in het voorbeeld t.o.v. de achtergrondkennis). In die zin is θ-subsumptie een speciaal geval van veralgemeende subsumptie, namelijk het geval dat $P = \emptyset$.

We zullen nu aantonen dat veralgemeende subsumptie toch kan geïmplementeerd worden aan de hand van θ-subsumptie. θ-subsumptie vergelijkt in wezen of de ene clausule, beschouwd als een verzameling atomen, een deelverzameling is van de andere clausule. Indien de achtergrondkennis in rekening moet gebracht worden, moeten we er voor zorgen dat de literals die nodig zijn om θ-subsumptie te testen, aanwezig zijn in de specifiekere clausule. De operatie die men daarvoor nodig heeft is *saturatie* [Rouveirol, 1994]. Saturatie van een clausule levert een nieuwe clausule die men bekomt door het lichaam van de oorspronkelijke clausule uit te breiden met alle atomen die een logisch gevolg zijn van de andere atomen in het lichaam. Men kan dan bewijzen dat een clausule c_1 een veralgemening is van een clausule c_2 met betrekking tot veralgemeende subsumptie t.o.v. een programma P, enkel en alleen indien de clausule c_1 een veralgemening is van de saturatie van c_2 met betrekking tot θ-subsumptie.

Voorbeeld De saturatie van c_3 is de clausule c_3':

$$\begin{aligned} c_3' : vader(\ X_3\ ,\ Y_3\ ) \leftarrow\ & ouder(\ X_3\ ,\ Y_3\ ), man(\ X_3\ ), mannelijk(\ X_3\ ), \\ & mens(\ X_3\ ), levend(\ X_3\ ), levend(\ Y_3\ ). \end{aligned}$$

De atomen $mannelijk(\ X_3\ )$ en $levend(\ X_3\ )$ zijn een logisch gevolg van het atoom $man(\ X_3\ )$ uit c_3, en moeten dus aan c_3 worden toegevoegd om de saturatie van c_3 te bekomen.

Om te controleren of c_1 algemener is dan c_3' kunnen we nu θ-subsumptie gebruiken: het atoom $mannelijk(\ X_3\ )$ uit c_1 dat geen element was van c_3 is door saturatie wel een element van c_3'. $\diamond$

4.5 Enkele andere belangrijke aspecten van ILP

Op verscheidene belangrijke aspekten van ILP wordt in deze thesis niet dieper ingegaan. We denken hier in de eerste plaats aan zoekoperatoren voor ILP en aan vooraf bepaalde beperkingen *(Eng. bias)*, in het bijzonder vooraf bepaalde beperkingen qua taal *(Eng. language bias)*.

Zoekoperatoren worden in het algemeen in een leer-context *verfijningsoperatoren* genoemd. Er zijn zowel *veralgemeningsoperatoren* als *specializeringsoperatoren*. Voor onze doeleinden is het voldoende de operatoren te gebruiken die worden besproken in [van der Laag en Nienhuys-Cheng, 1994].

Om van het leer-probleem een praktisch realizeerbaar zoek-probleem te maken, moet het probleem van vooraf aan in verschillende opzichten beperkt worden. Zo kunnen er vooraf bepaalde beperkingen zijn wat betreft het deel van de taal dat zal doorzocht worden, met welke zoekstrategie dit zal gedaan worden, welke heuristieken daarbij zullen gebruikt worden, enz. Vooral de vooraf bepaalde beperkingen op de taal maken dat het probleem praktisch realizeerbaar. Een belangrijke deelgebied binnen ILP (en ook in heel het onderzoeksgebied van automatisch leren) bestudeert en vergelijkt verschillende mogelijkheden van taal-beperkingen; er zijn ook verscheidene mogelijkheden bestudeerd om deze beperkingen in min of meerdere mate *declaratief* te specifiëren. Hiervoor verwijzen we naar [Adé *et al.*, 1995].

5 Besluit

In deze thesis hebben we het kader van Iteratieve Versieruimten uitgebouwd waarin we zoekstrategieën en oplossingsruimten voor het automatisch leren van concepten hebben bestudeerd. Dit kader hebben we dan toegepast op Inductief Logisch Programmeren. Een van de belangrijkste aspecten van het werk is de taal-onafhankelijke aanpak. De bekomen resultaten zijn dan ook toepasbaar voor een ruim pakket van leer-problemen. Verscheidene van de bijdragen in deze thesis veralgemenen taal-specifieke noties en technieken naar een taal-onafhankelijk niveau. Hiermee hebben we dan ook aangetoond dat verscheidene aspecten van taal-specifieke aanpakken in wezen taal-onafhankelijk zijn. Door deze aspecten te isoleren en taal-onafhankelijk te bestuderen worden zij dan ook toepasbaar in andere leer-systemen.

Referenties

[Adé *et al.*, 1995] H. Adé, L. De Raedt, en M. Bruynooghe. Declarative Bias for Specific-To-General ILP Systems. *Machine Learning*, 1995. Te verschijnen.

[Buntine, 1988] Wray Buntine. Generalized subsumption and its application to induction and redundancy. *Artificial Intelligence*, 36:375–399, 1988.

[Cohen en Feigenbaum, 1981] P.R. Cohen en E.A. Feigenbaum, redacteurs. *The handbook of artificial intelligence*, volume 3. Morgan Kaufmann, 1981.

[De Raedt, 1992] L. De Raedt. *Interactive Theory Revision: an Inductive Logic Programming Approach*. Academic Press, 1992.

[Haussler, 1988] D. Haussler. Quantifying inductive bias : AI learning algorithms and Valiant's learning framework. *Artificial Intelligence*, 36:177 – 221, 1988.

[Mellish, 1991] C. Mellish. The description identification problem. *Artificial Intelligence*, 52:151 – 167, 1991.

[Mitchell *et al.*, 1983] T.M. Mitchell, P.E. Utgoff, en R. Banerji. Learning by experimentation: acquiring and refining problem-solving heuristics. In R.S Michalski, J.G. Carbonell, en T.M. Mitchell, redacteurs, *Machine Learning: an artificial intelligence approach*, blz. 163–190. Tioga publishing company, 1983.

[Mitchell, 1982] T.M. Mitchell. Generalization as search. *Artificial Intelligence*, 18:203–226, 1982.

[Muggleton en De Raedt, 1994] S. Muggleton en L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.

[Plotkin, 1970] G. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, blz. 153–163. Edinburgh University Press, 1970.

[Rouveirol, 1994] C. Rouveirol. Flattening and saturation: Two representation changes for generalization. *Machine Learning*, 14:219–232, 1994.

[Sablon en De Raedt, 1995] G. Sablon en L. De Raedt. Forgetting and compacting data in concept learning. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, 1995. Te verschijnen.

[Sablon *et al.*, 1994] G. Sablon, L. De Raedt, en M. Bruynooghe. Iterative versionspaces. *Artificial Intelligence*, 69:393–409, 1994.

[Smith en Rosenbloom, 1990] B.D. Smith en P.S. Rosenbloom. Incremental non-backtracking focusing: A polynomially bounded generalization algorithm for version spaces. In *Proceedings of the 8th National Conference on Artificial Intelligence (AAAI-90)*, blz. 848–853. AAAI Press, 1990.

[Utgoff, 1986] P.E. Utgoff. Shift of bias for inductive concept-learning. In R.S Michalski, J.G. Carbonell, en T.M. Mitchell, redacteurs, *Machine Learning: an artificial intelligence approach*, blz. 107–148. Morgan Kaufmann, 1986.

[van der Laag en Nienhuys-Cheng, 1994] P.R.J. van der Laag en S.-H. Nienhuys-Cheng. Existence and nonexistence of complete refinement operators. In F. Bergadano en L. De Raedt, redacteurs, *Proceedings of the 7th European Conference on Machine Learning*, volume 784 van *Lecture Notes in Artificial Intelligence*, blz. 307–322. Springer-Verlag, 1994.

[Winston, 1975] P.H. Winston. Learning structural descriptions from examples. In P.H. Winston, redacteur, *Psychology of Computer Vision*. The MIT Press, 1975.