ELSEVIER

Artificial Intelligence 69 (1994) 393–409

Artificial Intelligence

Research Note

Iterative versionspaces

Gunther Sablon *, Luc De Raedt, Maurice Bruynooghe

Department of Computer Science, Katholieke Universiteit Leuven, Celestijnenlaan 200A, B-3001 Heverlee, Belgium

Received April 1994; revised June 1994

Abstract

An incremental depth-first algorithm for computing the $\mathcal{S}$- and $\mathcal{G}$-set of Mitchell's Candidate Elimination and Mellish's Description Identification algorithm is presented. As in Mellish's approach, lowerbounds (examples) as well as upperbounds can be handled. Instead of storing the complete $\mathcal{S}$- and $\mathcal{G}$-sets, only one element $s \in \mathcal{S}$ and $g \in \mathcal{G}$ is stored, together with backtrack information. The worst-case space complexity of our algorithm is *linear* in the number of lower- and upperbounds. For the Candidate Elimination algorithm this can be exponential. We introduce a test for membership of $\mathcal{S}$ and $\mathcal{G}$ with a number of coverage tests *linear* in the number of examples. Consequently the worst-case time complexity to compute $\mathcal{S}$ and $\mathcal{G}$ for each example is only a linear factor worse than the Candidate Elimination algorithm's.

Keywords: Concept learning; Versionspaces; Example generation

1. Introduction

Most concept learning methods use either a specific to general or general to specific strategy. If one of these approaches is taken, either a maximally general *or* a maximally specific hypothesis is learned. For some applications it is more interesting to use a bi-directional strategy, and to learn maximally general *and* maximally specific hypotheses. On the one hand this allows to use the current hypothesis from a maximally specific viewpoint (allowing only to make errors of omission) as well as from a maximally general viewpoint (allowing only to make errors of commission) [1]. On the other hand, this approach allows to

* Corresponding author. E-mail: gunther@cs.kuleuven.ac.be. Fax: ++ 32 16 20 53 08. Telephone: ++ 32 16 20 10 15.

SSDI 0004-3702(94)00039-4

generate *relevant* examples automatically. Relevant examples are covered by a maximally general but not by all maximally specific hypotheses.

In the Description Identification algorithm of [9] (which is an extension of the Candidate Elimination algorithm of [10]) the set $\mathcal{S}$ of *all* maximally specific concept descriptions and the set $\mathcal{G}$ of *all* maximally general concept descriptions are computed using bi-directional breadth-first search, and *all* members of $\mathcal{S}$ and $\mathcal{G}$ are stored. This is often very expensive, because the size of these sets can grow exponential in the number of examples (see also [5]). Korf [7] argues that exponential breadth-first search often exhausts the available memory long before an appreciable amount of time is used. In recent concept learning approaches, such as Inductive Logic Programming (*ILP*) [8, 11, 12], memory becomes increasingly important because of the use of very expressive description languages.

This paper presents an incremental algorithm to compute $\mathcal{S}$ and $\mathcal{G}$ using bi-directional depth-first search. It takes a different position in the trade-off between space complexity and time complexity while producing the same results. Our algorithm stores only *one* maximally general and *one* maximally specific consistent concept description, together with backtrack information. The underlying idea is similar to that of (iterative deepening) depth-first search versus breadth-first search in general: depth-first search avoids the memory problems of breadth-first search at the expense of recomputing certain elements [7].

The main contribution of the Iterative Versionspace algorithm is that its worst-case space complexity is *linear* in the number of information elements, which is an exponential gain with respect to the Candidate Elimination algorithm. We introduce a test for maximality (i.e., maximal specificness or maximal generality) and consistency with worst-case time complexity *linear* in the number of information elements. This test is based on the use of an *optimal generalization operator and an optimal refinement operator* [4]. Optimal operators avoid searching parts of the search space more than once. They are not only useful in our algorithm, but also in concept learning algorithms, and search algorithms in AI in general (see also [19]). Incremental computation of $\mathcal{S}$ and $\mathcal{G}$ from the backtrack information remains possible. The time penalty is linear with respect to the Candidate Elimination algorithm. Many properties of bi-directional search with the Candidate Elimination algorithm are retained. Because we gain an exponential factor in space and lose only a linear factor in time, we believe our approach contributes to making the use of versionspaces (and concept learning) more practical.

The paper is organized as follows. Terminology and notation are introduced in Section 2. Section 3 discusses related work and further motivates the presented approach. The Iterative Versionspace algorithm is presented in Section 4, followed by an example in Section 5, and the complexity analysis in Section 6. Finally, Section 7 discusses the generation of examples.

2. Terminology

We denote the language of concept descriptions by $\mathcal{L}$. $c_1 \prec c_2$ denotes that c_1 is more specific than c_2; $c_1 \preccurlyeq c_2$ denotes that c_1 is equal to or more specific than c_2. We assume the existence of a bottom element $\perp$ (minimal with respect to $\preccurlyeq$) and a top element $\top$ (maximal with respect to $\preccurlyeq$) in $\mathcal{L}$.

The Iterative Versionspace algorithm is provided with four types of information with respect to an unknown target concept $\mathcal{T}$: positive and negative lowerbounds, and positive and negative upperbounds (as in [9]). We employ the single-representation trick [2], i.e., we assume lower- and upperbounds belong to $\mathcal{L}$.

Definition 1. For $i \in \mathcal{L}$ and a target concept $\mathcal{T} \in \mathcal{L}$:
(1) if $i \preccurlyeq \mathcal{T}$, then i is a positive lowerbound (generalization) (i.e., i is a positive example);
(2) if $\neg(i \preccurlyeq \mathcal{T})$, then i is a negative lowerbound (discrimination) (i.e., i is a negative example);
(3) if $\mathcal{T} \preccurlyeq i$, then i is a positive upperbound (specialization);
(4) if $\neg(\mathcal{T} \preccurlyeq i)$, then i is a negative upperbound (differentiation).

We will also refer to positive lowerbounds and negative upperbounds as *s-bounds*, because they are used to generalize $\mathcal{S}$ (see further). Similarly, negative lowerbounds and positive upperbounds will be referred to as *g-bounds*.

The goal of the concept learning algorithm is then, given a set $I \subset \mathcal{L}$ of positive and negative lower- and upperbounds, to find the target concept $\mathcal{T}$.

Searching the target concept amounts to searching for a *consistent* element c of $\mathcal{L}$. Consistency is defined by:

Definition 2. $c \in \mathcal{L}$ is *consistent* with
(1) a positive lowerbound $i \in \mathcal{L}$, if $i \preccurlyeq c$;
(2) a negative lowerbound $i \in \mathcal{L}$, if $\neg(i \preccurlyeq c)$;
(3) a positive upperbound $i \in \mathcal{L}$, if $c \preccurlyeq i$;
(4) a negative upperbound $i \in \mathcal{L}$, if $\neg(c \preccurlyeq i)$.
$c \in \mathcal{L}$ is consistent with $I \subset \mathcal{L}$ if c is consistent with all elements of I.

We define the notions of subsets of minimal and maximal elements of a set of concept descriptions with respect to the relation $\preccurlyeq$:

Definition 3. $Min(Set) = \{c \in Set \mid \neg(\exists c' \in Set: c' \prec c)\}$.

Definition 4. $Max(Set) = \{c \in Set \mid \neg(\exists c' \in Set: c \prec c')\}$.

Using these notions we define $\mathcal{S}$ and $\mathcal{G}$ of [10]:

Definition 5. For a given set I of lower- and upperbounds,

- $\mathcal{S}_I = Min(\{s \in \mathcal{L} \mid s \text{ is consistent with } I\})$, and
- $\mathcal{G}_I = Max(\{g \in \mathcal{L} \mid g \text{ is consistent with } I\})$.

$\mathcal{S}_I$ determines a "lowerbound" on the set of all concept descriptions consistent with I. Analogously $\mathcal{G}_I$ is an "upperbound". Together, $\mathcal{S}_I$ and $\mathcal{G}_I$ determine a versionspace $\mathcal{VS}_I$, which is exactly the set of *all* concept descriptions consistent with I (see [10]):

Definition 6. $\mathcal{VS}_I = \{c \in \mathcal{L} \mid \exists g \in \mathcal{G}_I, \exists s \in \mathcal{S}_I : s \preccurlyeq c \preccurlyeq g\}$.

The idea is that if c is more general than $s \in \mathcal{S}_I$, it will be consistent with all s-bounds; and if it is more specific than $g \in \mathcal{G}_I$, it will be consistent with all g-bounds.

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

If $\mathcal{T} \in \mathcal{VS}$, $\mathcal{S}$ and $\mathcal{G}$ can also be used to classify some unclassified elements of $\mathcal{L}$:

Definition 7. $c \in \mathcal{L}$ is a
(1) positive lowerbound if $\forall s \in \mathcal{S} : c \preccurlyeq s$;
(2) negative lowerbound if $\forall g \in \mathcal{G} : \neg(c \preccurlyeq g)$;
(3) positive upperbound if $\forall g \in \mathcal{G} : g \preccurlyeq c$;
(4) negative upperbound if $\forall s \in \mathcal{S} : \neg(s \preccurlyeq c)$.

The four basic operations needed in our algorithms are:

Definition 8. For $c_1, c_2 \in \mathcal{L}$, the set of least upperbounds of c_1 and c_2 is $\mathtt{lub}(c_1, c_2) = Min(\{c \in \mathcal{L} \mid c_1 \preccurlyeq c \text{ and } c_2 \preccurlyeq c\})$.

Definition 9. For $c_1, c_2 \in \mathcal{L}$, the set of greatest lowerbounds of c_1 and c_2 is $\mathtt{glb}(c_1, c_2) = Max(\{c \in \mathcal{L} \mid c \preccurlyeq c_1 \text{ and } c \preccurlyeq c_2\})$.

Definition 10. For $c_1, c_2 \in \mathcal{L}$, the set of most specific generalizations of c_1 not covered by c_2 is $\mathtt{msg}(c_1, c_2) = Min(\{c \in \mathcal{L} \mid c_1 \preccurlyeq c \text{ and } \neg(c \preccurlyeq c_2)\})$.

Definition 11. For $c_1, c_2 \in \mathcal{L}$, the set of most general specializations of c_1 not covering c_2 is $\mathtt{mgs}(c_1, c_2) = Max(\{c \in \mathcal{L} \mid c \preccurlyeq c_1 \text{ and } \neg(c_2 \preccurlyeq c)\})$.

3. Motivation and related work

The Candidate Elimination algorithm (CE) [10] was the first algorithm to formalize the concept learning problem as a search problem. It computes $\mathcal{S}$ and $\mathcal{G}$ breadth-first from lowerbounds only. At the same time $\mathcal{S}$ and $\mathcal{G}$ are used to check maximal specificness, maximal generality and consistency. Therefore

CE does not need to store the lowerbounds. CE has been criticized however for its possible exponential behavior in storing and computing $\mathcal{S}$ and $\mathcal{G}$ [5]. The Description Identification algorithm (DI) of [9] extends CE, in that it cannot only handle lowerbounds, but also upperbounds. However, it has basically the same worst-case time and space complexity as CE.

Mitchell [10] also discusses basic depth-first specific to general and general to specific algorithms for concept learning, and their relation to CE. The main advantage of depth-first algorithms is their linear space complexity. However, without searching the complete search space they are unable to check maximal specificness or maximal generality, to detect convergence, and to classify unclassified bounds.

Depth-first search versus breadth-first search in general is discussed in [7]. Korf also presents depth-first iterative deepening as a search strategy with linear space complexity, and, in an exponential search space, the same time complexity as breadth-first search.

Motivated by this general result, we propose the Iterative Versionspace algorithm (ITVS). As Mellish's algorithm, ITVS can handle upperbounds as well as lowerbounds. ITVS performs bi-directional (i.e., on $\mathcal{S}$ and $\mathcal{G}$) depth-first search, and guarantees maximal specificness and maximal generality in linear time. Still, convergence cannot be detected and unclassified bounds cannot be classified without searching the complete search space.

Under strong restrictions (in particular, in conjunctive languages over tree-structured attribute hierarchies) earlier incremental approaches already presented better complexity results than CE. Incremental Non-Backtracking Focusing (INBF) of [17] achieves this by avoiding backtracking (as [1] did non-incrementally). INBF employs only one maximally general concept description (*upper*). It specializes *upper* only when there exists a single consistent specialization, i.e., in case of specialization with respect to a near-miss. 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 [6] only represents $\mathcal{S}$ 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, $\mathcal{S}$ could be exponential as well. Hirsh also notes that in certain applications general descriptions are preferred over specific ones. Indeed, we could think of an agent learning preconditions of actions (see [15]): taking an element of $\mathcal{S}$ 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 $\mathcal{S}$ and $\mathcal{G}$.

In ITVS bi-directionality will also provide the ability to generate relevant examples. Generating relevant examples from an overly general and an overly specific description was also employed in DISCIPLE [20] and APT [13]. Nedellec [13] formalizes some of the ideas of DISCIPLE by introducing the *Smallest Generalization Steps Strategy*, which is user-guided to accelerate convergence with respect to blind depth-first search, and which implements `lub` and `mgs` in a subset of first-order logic to avoid overgeneralization. One way to accelerate convergence with respect to CE is the use of *focus sets* and *seed sets* [14], which in fact contain *approximations* of lower- and upperbounds. However, in both cases the worst-case time and space complexities remain the same.

We believe our algorithm is similar in spirit as the approaches of [6, 17, 20], by avoiding to store $\mathcal{S}$ and $\mathcal{G}$. However, it is more general than those approaches, because it has the same power and generality as DI.

4. Iterative versionspaces

The Iterative Versionspace algorithm combines general to specific and specific to general depth-first search. The search is pruned with the pruning principles of [10] (without storing $\mathcal{S}$ or $\mathcal{G}$ explicitly), and the implicit use of an optimal refinement operator [4] and an optimal generalization operator.

We use the following data structures:

- s is the current most specific concept description, g is the current most general concept description.
- The array I_s contains all s-bounds (i.e., positive lowerbounds and negative upperbounds), and I_g contains all g-bounds (i.e., negative lowerbounds and positive upperbounds). 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[1] B_s contains pairs (*ind* , *alternatives*), called choicepoints, where *ind* is an index in I_s, and *alternatives* is a non-empty list of concept descriptions used to backtrack on s and to test maximal specificness of s. Similarly, the stack B_g contains choicepoints (*ind*, *alternatives*), where *ind* is an index in I_g, and *alternatives* is a non-empty list of concept descriptions used to backtrack on g and to test maximal generality of g.

We have the following invariants on our data structures:

(1) (a) $s \in \mathcal{S}_I$, and (b) $g \in \mathcal{G}_I$, with $I = I_g[1..n_g] \cup I_s[1..n_s]$.

(2) For all choicepoints (*ind*, *alternatives*) on B_s: all elements of *alternatives* are consistent with (a) $I_g[1..n_g]$, and (b) $I_s[1..ind]$, i.e., with *all* g-bounds and the first *ind* s-bounds.

[1] We use the operation `push`(*ind*, *alternatives*, B_c) to put the choicepoint (*ind*, *alternatives*) 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.

Algorithm 1: Iterative Versionspaces (ITVS)

```
procedure ITVS()
  returns concept, stack, array, index, concept, stack, array, index
    s := ⊥; g := ⊤; B_s := ∅; B_g := ∅; n_s := 0; n_g := 0;
    while there are still bounds to be processed do
      read(i) with i an unprocessed bound
      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 ¬consistent(g,i) then
          g, B_g, ind := select_alternative(∅,B_g,n_g)
ⓑ         g,B_g := specialize(g,B_g,ind)
ⓒ       s, B_s := generalize(s,B_s,n_s − 1)
      else
        n_g := n_g + 1; I_g[n_g] := i
        B_s := prune_stack(B_s,i)
        if ¬ consistent(s,i) then
          s, B_s, ind := select_alternative(∅,B_s,n_s)
          s,B_s := generalize(s,B_s,ind)
        g, B_g := specialize(g,B_g,n_g − 1)
    return s, B_s, I_s, n_s, g, B_g, I_g, n_g
endproc
```

(3) For all choicepoints (*ind*, *alternatives*) on B_g: all elements of *alternatives* are consistent with (a) $I_s[1..n_s]$, and (b) $I_g[1..ind]$, i.e., with *all* s-bounds and the first *ind* g-bounds.

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.

In the following algorithms, the procedure `consistent` is straightforward, and not described in detail: `consistent`(c,i) returns true when concept description c is consistent with bound i, and false otherwise.

Consider Algorithm 1. s is initialized to $\bot$, g to $\top$, B_s and B_g to the empty stack, and n_s and n_g to 0. The main loop of the algorithm processes the bounds one by one in the given order. Below, we only explain the actions to be taken for a positive lowerbound i; the cases of negative lowerbounds, and negative and positive upperbounds are analogous.

First i is stored in I_s. Then all alternatives for g on B_g not consistent with i are pruned with the procedure `prune_stack` (see Algorithm 4). Indeed, if an alternative c on B_g does not cover i, then certainly none of its specializations will cover i, so c can be deleted from B_g. This pruning step ensures

Algorithm 2: Generalization in ITVS

```
procedure generalize(s: concept; B_s: stack; n_c: index)
  returns concept, stack
    while n_c ≠ n_s do
      n_c := n_c + 1
      if ¬ consistent(s, I_s[n_c]) then
        if I_s[n_c] is positive lowerbound then
ⓓ         generalizations := lub(s,I_s[n_c])
        else
ⓔ         generalizations := msg(s,I_s[n_c])
ⓕ       generalizations := select all c from generalizations
          with all_consistent(c,I_g,n_g) ∧ max_specific(c,B_s)
ⓖ       s,B_s,n_c := select_alternative(generalizations,B_s,n_c)
    return s, B_s
endproc

procedure max_specific(s: concept; B_s: stack)
  returns boolean
    B_c := copy(B_s)
    max_specific := true
    while ¬ is_empty(B_c) ∧ max_specific do
      ind, alternatives, B_c := pop(B_c)
      max_specific := (¬∃c ∈ alternatives : c ≼ s)
    return max_specific
endproc

procedure all_consistent(c: concept; I_c: array; n_c: index)
  returns boolean
    return (∀ ind, 1 ≤ ind ≤ n_c: consistent(c,I_c[ind]))
endproc
```

invariant 3(a).

If i is not consistent with g (see ⓐ), an alternative for g is popped from B_g using the procedure `select_alternative`[2] (see Algorithm 4). In general `select_alternative`(*alternatives*, B_c, n_c) (with *alternatives* a list of elements in $\mathcal{L}$, B_c a stack of more alternatives, and n_c an index in I_c) works as follows: it selects an element c of *alternatives* (see ⓙ); if *alternatives* is initially empty, it first pops a choicepoint from B_c (see ⓘ). If *alternatives* and B_c are empty, no c consistent with I can be found, so `select_alternative` announces failure and halts ITVS (see ⓗ). Otherwise, it returns c, B_c (containing the rest of the list *alternatives* and the rest of B_c; see ⓚ) and n_c, such that c is consistent with

[2] 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*.

Algorithm 3: Specialization in ITVS

```
procedure specialize(g: concept; B_g: stack, n_c: integer)
  returns concept, stack
    while n_c ≠ n_g do
      n_c := n_c + 1
      if ¬ consistent(g, I_g[n_c]) then
        if I_g[n_c] is negative lowerbound then
          specializations := mgs(g,I_g[n_c])
        else
          specializations := glb(g,I_g[n_c])
        specializations := select all c from specializations
          with all_consistent(c,I_s,n_s) ∧ max_general(c,B_g)
        g,B_g,n_c := select_alternative(specializations,B_g,n_c)
    return g, B_g
endproc

procedure max_general(g: concept; B_g: stack)
  returns boolean
    B_c := copy(B_g)
    max_general := true
    while ¬ is_empty(B_c) ∧ max_general do
      ind, alternatives, B_c := pop(B_c)
      max_general := (¬∃c ∈ alternatives: g ≼ c)
    return max_general
endproc
```

the elements of $I_c[1..n_c]$. Therefore, initially n_c must be such that all elements of *alternatives* are consistent with the elements of $I_c[1..n_c]$.

After the call to `select_alternative` all bounds on I_g from *ind* up to n_g are reprocessed (see ⓑ). Given that g is already consistent with the first *ind* bounds in I_g, `specialize`(g,B_g,ind) returns a maximally general g consistent with I, and a stack B_g fulfilling conditions 3(a) and 3(b). Then, given that s is consistent with the first $n_s - 1$ bounds of I_s, `generalize`$(s,B_s,n_s - 1)$ returns a maximally specific s consistent with I, and a stack B_s fulfilling conditions 2(a) and 2(b). Therefore all invariants will hold at ⓒ, and hence at the end of the while loop.

We will now explain how `generalize` works (see Algorithm 2); `specialize` (see Algorithm 3) is again dual.

When $n_c = n_s$, s is 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 wasn't consistent already). If $I_s[n_c]$ is a positive lowerbound, consistent generalizations are the least upperbounds of s and $I_s[n_c]$ (see ⓓ). Otherwise $I_s[n_c]$ is a negative upperbound, and the consistent

Algorithm 4: Stack operations in ITVS

```
procedure prune_stack(B_c: stack; i: bound)
  returns stack
    if is_empty(B_c) then
      return ( ∅ )
    else
      ind, alternatives, B_c := pop(B_c)
      B_c := prune_stack(B_c,i)
      alternatives := select all c from alternatives
        with consistent(c,i)
      if alternatives ≠ ∅ then
        B_c := push(ind, alternatives, B_c)
    return B_c
endproc

procedure select_alternative(alternatives: list; B_c: stack; n_c: index )
  returns concept, stack, index
    if alternatives = ∅ then
      if is_empty(B_c) then
(h)     failure
      else
(i)     n_c, alternatives, B_c := pop(B_c)
(j)   c := head(alternatives)
    if tail(alternatives) ≠ ∅ then
(k)   B_c := push(n_c, tail(alternatives), B_c)
    return c, B_c, n_c
endproc
```

generalizations are the most specific generalizations of s not covered by $I_s[n_c]$ (see ⓔ). Note that all consistent generalizations of s are also consistent with $I_s[1..n_c-1]$. In ⓕ only those generalizations also consistent with all g-bounds and maximally specific are selected. Then `select_alternative` (see ⓖ) finds a next candidate s, the corresponding B_s, and the index up to where s is consistent with I_s.

`max_specific`(s,B_s) checks whether s is maximally specific, and guarantees that each $c \in \mathcal{L}$ will not be generalized more than once. This restriction in fact implements an *optimal*, but still complete, generalization operator. By definition, a generalization operator is optimal if each $c \in \mathcal{L}$ will not be generalized more than once. This is dual to an optimal refinement operator, introduced in CLAUDIEN [4]. The procedure `max_general`, dual to `max_specific`, implements an optimal refinement operator.

First `max_specific` (see Algorithm 2) copies the parameter B_s to B_c in order not to change B_s. `max_specific` compares s to the alternatives c on B_s:

- If there exists a c on B_s such that $c \preccurlyeq s$, then there exists a consistent

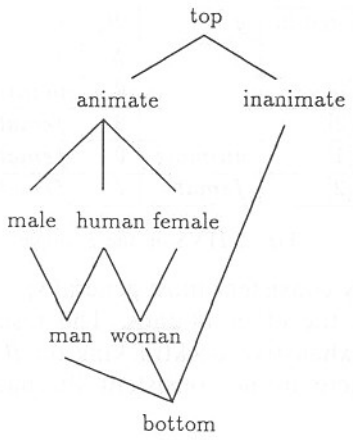

Fig. 1. Taxonomy of $\mathcal{L}$.

most specific generalization c' of c such that $c' \preccurlyeq s$ (possibly s itself). Completeness of `msg` and `lub` guarantees that on backtracking c' will be generated. Thus, if there exists a consistent most specific generalization c' of c such that $c' \prec s$, then s does not belong to $\mathcal{S}$, and can be skipped. Else, c' must be s. So s will be considered when generalizing c, and can therefore also be skipped at this point, without losing completeness of the generalization operator.

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

`all_consistent`(c,I_g,n_g) checks whether c is consistent with $I_g[1..n_g]$ and is straightforward. If B_g is empty, this test can be replaced by $c \preccurlyeq g$. This is more efficient for languages in which $|\mathcal{G}|$ is always equal to 1.

5. Example

We will use an example and taxonomy (see Fig. 1) of [9].
Fig. 2 shows the consecutive stages of an example session with ITVS. g is initialized to $\top$, and s is initialized to $\bot$. The first bound is a negative upperbound and is stored in $I_s[1]$. s is generalized to *inanimate*, and the alternatives *female* and *male* are put on B_s. The second bound is not consistent with *inanimate*, so *inanimate* should be generalized. *inanimate* can only be generalized to *top*. *top* is more general than *female*, so it can be skipped at this point. The most recent alternative on B_s (i.e., *female*) is assigned to s. *female* is consistent with $I_s[2]$. The third bound forces g to be specialized to *animate*, which is maximally general since there are no alternatives on B_g. According to the fourth bound, *male* should be pruned from B_s and g should

New Information	Stored in	g	B_g	s	B_s
		$\top$	$\emptyset$	$\bot$	$\emptyset$
$\neg(\mathcal{T} \preccurlyeq human)$	$I_s[1]$	$\top$	$\emptyset$	$inanimate$	$[(1, [female, male])]$
$woman \preccurlyeq \mathcal{T}$	$I_s[2]$	$\top$	$\emptyset$	$female$	$[(1, [male])]$
$\neg(inanimate \preccurlyeq \mathcal{T})$	$I_g[1]$	$animate$	$\emptyset$	$female$	$[(1, [male])]$
$\mathcal{T} \preccurlyeq female$	$I_g[2]$	$female$	$\emptyset$	$female$	$\emptyset$

Fig. 2. ITVS on the example.

be specialized. The only consistent most general specialization is *female*, which is also consistent with the other bounds. The result is *female*. Convergence could 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.

6. Complexity analysis

We analyze the worst-case time and space complexity as in [10]. For the time complexity analysis we count the number of coverage tests, whereas for the space complexity we count the number of elements of $\mathcal{L}$ stored.

First we introduce some notation. b_s is the average upward branching factor in $\mathcal{S}_I$, and b_g the average downward branching factor in $\mathcal{G}_I$.[3] The average upward branching factor is the average number of generalizations (i.e., most specific generalizations and least upperbounds) that pass test Ⓕ in Algorithm 2.

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

is the size of the specific to general search space, and

$$\bar{g} = \frac{b_g^{n_g+1} - 1}{b_g - 1} \approx b_g^{n_g}$$

is the size of the general to specific search space. 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 refinement operator (see Section 4), we may use the same branching factor as in CE, in which searching parts of the search space more than once is avoided by removing doubles from $\mathcal{S}$ and $\mathcal{G}$. The resulting complexity analysis shows again that optimal operators are useful, and can be implemented efficiently (with respect to time and space) in the context of versionspaces. Moreover, we believe this result is also applicable to concept learning algorithms in general (see also [4]).

Theorem 12. *ITVS has a worst-case space complexity of* $O(n_s \times b_s + n_g \times b_g)$.

[3] We assume upward and downward branching factors are bounded.

ITVS stores all n_s s-bounds in I_s. Furthermore B_s contains in the worst-case a list of $b_s - 1$ alternatives per s-bound. Analogously I_g contains all n_g g-bounds, and B_g contains in the worst-case a list of $b_g - 1$ alternatives per g-bound. Consequently, ITVS stores $(n_s \times b_s + n_g \times b_g)$ elements of $\mathcal{L}$. This worst-case space complexity is *linear* in the number of bounds.

Theorem 13. *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*

$$O(n_g \times (b_s + b_g) + n_s \times b_s^2).$$

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

$$O(n_s \times (b_s + b_g) + n_g \times b_g^2).$$

We will discuss the case of an s-bound i. In the pruning step all alternatives on B_g have to be compared with i. This gives $n_g \times (b_g - 1)$ comparisons in the worst case. Then, for each of the b_s generalizations of s we have n_g comparisons to check consistency with I_g, and $n_s \times b_s$ comparisons to check maximal specificness.

Theorem 14. *To compute a most specific concept description s and a most general concept description g, ITVS has a worst-case time complexity of*

$$O((n_g + n_s \times b_s) \times \bar{s} + (n_s + n_g \times b_g) \times \bar{g} + (n_s \times n_g \times (b_s + b_g))).$$

In the worst case the specific to general and general to specific search spaces have to be searched completely. 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. Guaranteeing maximal specificness 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 total worst-case complexity of $O((n_g + n_s \times b_s) \times \bar{s})$. Guaranteeing consistency and maximal generality of g gives $O((n_s + n_g \times b_g) \times \bar{g})$.

The pruning steps are done only once for each bound. For B_g this results in $n_g \times (b_g - 1)$ comparisons for each of the n_s s-bounds. Similarly pruning of B_s will give an extra term of $O(n_g \times n_s \times b_s)$.

Corollary 15. *To compute $\mathcal{S}$ and $\mathcal{G}$ for each new bound, and to detect convergence, ITVS has a worst-case time complexity of*

$$O((n_g + n_s \times b_s) \times \bar{s}^2 + (n_s + n_g \times b_g) \times \bar{g}^2 + (n_s \times n_g \times (b_s + b_g))).$$

To compute $\mathcal{S}$ from B_s and $\mathcal{G}$ from 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, in the specific to general search space and in the worst case, for each of the $\bar{s}$ elements of the search space the complete search space has

to be searched for consistent and maximally specific elements. After having computed $\mathcal{S}$ and $\mathcal{G}$, detecting convergence is just checking whether $\mathcal{S}$ and $\mathcal{G}$ are equal and singletons. Since classifying unclassified elements also requires recomputation of $\mathcal{S}$ and $\mathcal{G}$, it has the same worst-case time complexity.

CE has a worst-case time complexity of (see [10])

$$O((\bar{g} + \bar{s}) \times (b_s^{n_s} + b_g^{n_g})),$$

because each element s in the specific to general search space, has to be compared to all $b_g^{n_g}$ elements in $\mathcal{G}$ to check consistency, and to all $b_s^{n_s}$ elements in $\mathcal{S}$ to check maximal specificness. 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.

Consequently, CE has to store the sets $\mathcal{S}$ and $\mathcal{G}$ completely, and therefore has a worst-case space complexity of

$$O(b_s^{n_s} + b_g^{n_g}).$$

ITVS is a linear factor worse than CE in time to compute $\mathcal{S}$ and $\mathcal{G}$ completely, because $\bar{s} \approx b_s^{n_s}$ and $\bar{g} \approx b_g^{n_g}$. Nevertheless, ITVS is linear in space, whereas CE is exponential as soon as $b_s > 1$ or $b_g > 1$. As in [7] 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, and this certainly for Inductive Logic Programming.

In the special case of conjunctive languages with k features (as discussed in [17] and [6]), $b_s = 1$ and $b_g = k$. In this case, ITVS is still exponential in time when backtracking is needed to update s or g. [17] and [6] are not, basically because they do not compute a consistent element of $\mathcal{G}$. If a maximally general consistent concept description is needed, their algorithms will have to be extended, and will show an exponential behavior as well.

7. Example generation

In an *interactive* concept learning setting convergence can be accelerated by generating new relevant examples automatically, preferably a minimal sequence of them. Unfortunately generating a minimal sequence of examples is in general a NP-hard problem [18]. Factorization [18] or domain-dependent heuristics might guide this search.

Storing both maximally specific *and* maximally general elements is useful in an *interactive* concept learning setting. Ideal would be then to have $s \in \mathcal{S}$ and $g \in \mathcal{G}$, and to find $c_1, c_2, i \in \mathcal{L}$ such that $s \preccurlyeq c_1 \prec c_2 \preccurlyeq g$, together with $\neg(i \preccurlyeq c_1) \wedge (i \preccurlyeq c_2)$ or $(c_1 \preccurlyeq i) \wedge \neg(c_2 \preccurlyeq i)$. In the former case i would be a *relevant* lowerbound, in the latter a relevant upperbound. The closer c_1 and c_2 are to a *middle point* between s and g, the less lower- or upperbounds would be needed to converge. By definition $c_m \in \mathcal{L}$ is a middle point between s and g, if $s \preccurlyeq c_m \preccurlyeq g$ and

Algorithm 5: Searching an element more general than s, in case $\neg(s \preccurlyeq g)$

```
procedure above(s: concept; B_g: stack)
  returns concept
    B_c := copy(B_g)
    repeat
      n_c, alternatives, B_c := pop(B_c)
    until ∃c ∈ alternatives : s ≼ c
    select one c from alternatives with s ≼ c
    while n_c ≠ n_g do
      n_c := n_c + 1
      if ¬ consistent(I_g[n_c], c) then
        if I_g[n_c] is negative lowerbound then
          specializations := mgs(c,I_g[n_c])
        else
          specializations := glb(c,I_g[n_c])
①       c := select one c from specializations with s ≼ c
    return c
endproc
```

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

However, working in a depth-first way we cannot guarantee that $s \preccurlyeq g$ will always hold. In case it doesn't, we can find a consistent alternative s' of s and a consistent alternative g' of g such that $s \preccurlyeq g'$ and $s' \preccurlyeq g$. Requiring that $s' \in \mathcal{S}$ and $g' \in \mathcal{G}$ would require backtracking and is therefore in the worst case exponential in time. Without those requirements it can be done in linear time. This is shown in Algorithm 5.

Since s is consistent with all examples, there must be at least one alternative c on B_g such that $s \preccurlyeq c$. This means c is consistent with all s-bounds. For each negative lowerbound $I_g[n_c]$, $\texttt{mgs}(c,I_g[n_c])$ must contain an element c' such that $s \preccurlyeq c'$, because s is consistent with $I_g[n_c]$, and $s \preccurlyeq c$. We could guarantee that c' is in $\mathcal{S}$, by adding the test $\texttt{max\_general}(c,B_g)$ at ①, but this would introduce backtracking. The analogue holds for each positive upperbound.

With respect to example generation bi-directional approaches are much more appropriate than specific to general or general to specific only (i.e., systems such as Marvin [16] or CLINT [3]), because the latter cannot even define a middle point. Typically, these systems take $c_1 = s$ and c_2 a most specific generalization of s covering at least one example 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. This shows that bi-directional approaches are better suited for generating minimal sequences of examples.

8. Conclusion

We have shown how depth-first specific to general and general to specific approaches can be combined to a depth-first version of the Candidate Elimination algorithm. The resulting algorithm returns a maximally general and a maximally specific consistent concept description. Space requirements of the algorithm are linear in the number of bounds. Testing for maximal generality and maximal specificness requires a number of coverage tests linear in the number of bounds. The approach also provides a framework for example generation in an interactive setting.

Acknowledgements

We would like to thank Hilde Adé, Céline Rouveirol and Wim Van Laer for comments on earlier versions of this paper. We would also like to thank Benjamin Smith for his additional comments on the INBF algorithm. Maurice Bruynooghe and Luc De Raedt are supported by the Belgian National Fund for Scientific Research and by the ESPRIT Project No. 6020 on Inductive Logic Programming.

References

[1] A. Bundy, B. Silver and D. Plummer, An analytical comparison of some rule-learning programs, *Artif. Intell.* **27** (1985) 137–181.
[2] P.R. Cohen and E.A. Feigenbaum, eds., *The Handbook of Artificial Intelligence*, Vol. 3 (Morgan Kaufmann, Los Altos, CA, 1981).
[3] L. De Raedt, *Interactive Theory Revision: An Inductive Logic Programming Approach* (Academic Press, New York, 1992).
[4] L. De Raedt and M. Bruynooghe, A theory of clausal discovery, in: *Proceedings IJCAI-93*, Chambery, France (1993) 1058–1063.
[5] D. Haussler, Quantifying inductive bias: AI learning algorithms and Valiant's learning framework, *Artif. Intell.* **36** (1988) 177–221.
[6] H. Hirsh, Polynomial-time learning with version spaces, in: *Proceedings AAAI-92*, San Jose, CA (1992) 117–122.
[7] R.E. Korf, Depth-first iterative deepening: an optimal admissible tree search, *Artif. Intell.* **27** (1985) 97–109.
[8] N. Lavrač and S. Džeroski, *Inductive Logic Programming: Techniques and Applications* (Ellis Horwood, Chicester, England, 1993).
[9] C. Mellish, The description identification problem, *Artif. Intell.* **52** (1991) 151–167.
[10] T.M. Mitchell, Generalization as search, *Artif. Intell.* **18** (1982) 203–226.
[11] S. Muggleton, ed., *Inductive Logic Programming* (Academic Press, New York, 1992).
[12] S. Muggleton and L. De Raedt, Inductive logic programming: theory and methods, *J. Logic Program.* (to appear).
[13] C. Nedellec, How to specialize by theory refinement, in: *Proceedings ECAI-92*, Vienna, Austria (1992) 474–478.
[14] J. Nicolas, Empirical bias for version space, in: *Proceedings IJCAI-91*, Sydney, Australia (1991) 671–676.

[15] 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, eds., *Current Trends in AI Planning* (IOS Press, Amsterdam, Netherlands, 1994) 254–265.

[16] C. Sammut and R. Banerji, Learning concepts by asking questions, in: R.S Michalski, J.G. Carbonell and T.M. Mitchell, eds., *Machine Learning: An Artificial Intelligence Approach* **2** (Morgan Kaufmann, Los Altos, CA, 1986) 167–192.

[17] B.D. Smith and P.S. Rosenbloom, Incremental non-backtracking focusing: a polynomially bounded generalization algorithm for version spaces, in: *Proceedings AAAI-90*, Boston, MA (1990) 848–853.

[18] D. Subramanian and J. Feigenbaum, Factorization in experiment generation, in: *Proceedings AAAI-86*, Philadelphia, PA (1986) 518–522.

[19] L.A. Taylor and R.E. Korf, Pruning duplicate nodes in depth-first search, in: *Proceedings AAAI-93*, Washington, DC (1993) 756–761.

[20] G. Tecuci and Y. Kodratoff, Apprenticeship learning in imperfect domain theories, in: Y. Kodratoff and R.S. Michalski, eds., *Machine Learning: An Artificial Intelligence Approach* **3** (Morgan Kaufmann, Los Altos, CA, 1990) 514–551.