Tor: Extensible Search with Hookable Disjunction

TOM SCHRIJVERS

*Department of Applied Mathematics and Computer Science, Ghent University
Krijgslaan 281 S9 WE02, 9000 Gent, Belgium
E-mail: tom.schrijvers@ugent.be*

MARKUS TRISKA

*Database and Artificial Intelligence Group, Vienna University of Technology
Favoritenstraße 9, 1040 Wien, Austria
E-mail: triska@dbai.tuwien.ac.at*

BART DEMOEN

*Department of Computer Science, KU Leuven
Celestijnenlaan 200A, 3001 Leuven, Belgium
E-mail: bart.demoen@cs.kuleuven.be*

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

Horn Clause Programs have a natural depth-first procedural semantics. However, for many programs this procedural semantics is ineffective. In order to compute useful solutions, one needs the ability to modify the search method that explores the alternative execution branches.

Tor, a well-defined hook into Prolog disjunction, provides this ability. It is light-weight thanks to its library approach and efficient because it is based on program transformation. *Tor* is general enough to mimic search-modifying predicates like ECLiPSe's `search/6`. Moreover, *Tor* supports modular composition of search methods and other hooks. Our library is already provided and used as an add-on to SWI-Prolog.

KEYWORDS: search, compositionality

1 Introduction

Kowalski's well-known adage crisply captures the essence of programming in the equation "ALGORITHM = LOGIC + CONTROL" (Kowalski 1979). In Prolog, the LOGIC part is captured in the programmer-supplied rules or clauses that have a first-order logic interpretation. The CONTROL component is supplied by the Prolog engine and essentially consists of *search*. In order to answer queries, a Prolog engine performs a backward-chaining depth-first tree search.

Prolog's default search strategy is in practice inadequate to effectively scour large search spaces. As a consequence, the programmer often has to complement Prolog's CONTROL with additional hints or heuristics in the form of extra code. This is particularly prevalent in the context of Constraint Logic Programming where it is

<pre> label([]). label([Var Vars]) :- (var(Var) -> fd_inf(Var,Value), (Var #= Value, label(Vars) ; Var #\= Value, label([Var Vars])) ; label(Vars)). </pre>	<pre> label([],_). label([Var Vars],D) :- (var(Var) -> D > 0, ND is D - 1, fd_inf(Var,Value), (Var #= Value, label(Vars,ND) ; Var #\= Value, label([Var Vars],ND)) ; label(Vars,D)). </pre>
--	---

Fig. 1. Labeling predicate: plain (left) and with depth bound (right).

common practice for the programmer to complement a constraint model with a search specification.

Unfortunately, it is not all that easy to cleanly separate LOGIC and CONTROL when implementing search heuristics in Prolog. When one discovers that Prolog’s CONTROL is ineffective, it is often impossible to orthogonally add one’s own CONTROL without touching the existing LOGIC. The problem is that syntactically logic and control in Prolog are generally tightly coupled, and adding a different control means cross-cutting existing code.

In this paper we present a novel approach to adding, in an orthogonal manner, CONTROL. Our solution features the following properties:

- It is a light-weight library-based approach that is easily portable to different Prolog systems: it is currently an SWI-Prolog library (Wielemaker et al. 2012)
- Our approach has all the benefits of modularity: search methods can be composed and the library of these heuristics is (user-)extensible.
- Its overhead is minimal, as we demonstrate on benchmarks: this is achieved through `term_expansion/2`, a feature present in most Prolog systems.

With `Tor`, we capture all common search methods in CLP(FD) libraries such as ECLiPSe’s `search/6` (Schimpf and Shen 2012). This approach is indeed particularly suitable for Constraint Logic Programming, but also useful for general Prolog programs with a large search space.

2 Problem Statement

We illustrate the heart of the matter on a simple labeling predicate `label/1` written against SWI-Prolog’s `clpfd` library (Triska 2012) (see Fig. 1, left). `label/1` defines a search tree where the branches are created by the disjunction.¹

¹ `fd_inf/2` returns the smallest value in a variable’s finite domain.

Suppose that for a certain call `label([X1, ..., Xn])` the search tree is too large to fully explore. So we decide to reap the low-hanging solutions only, and prune the subtrees that are too deep to explore. Hence, we impose a *depth bound* on Prolog's depth first search. This is achieved by modifying the `label/1` code to that of `label/2` (Fig. 1, right) where the second parameter is the depth bound.

If this heuristic is not successful, and we try another one, and so on.

Problems with this Approach The problems with the above approach should be apparent:

- The approach follows the well-known copy-paste-modify anti-pattern. Variants of the labeling code are copied all over the place, potentially propagating bugs and rendering maintenance into a nightmare. Working code is modified.
- The same heuristic is implemented over and over in different settings (different applications, different labeling predicates, different Prolog systems, ...). This process is error-prone, wastes precious programmer time and is bound to yield non-optimal code quality.
- The effort and expertise required to combine working labeling code with various search heuristics is non-trivial. This means that fewer combinations are explored by programmers under time pressure or unfamiliar with particular heuristics. The end result is that suboptimal solutions are obtained.

Current Solutions In the context of CLP **ECLiPSe** (Schimpf and Shen 2012) copes with this problem by providing a number of search methods in the `search/6` predicate. This predicate lets the user control through its various arguments the selection method, the choice method and the search method: the former two decide on which variable is used during labeling, and which value it is assigned first. They do not concern us here. The search method controls how the search tree is explored, e.g., depth-bounded, node-bounded or limited discrepancy search. Apart from individual search methods, only a fixed number of compositions is supported, such as changing strategy when a depth bound is reached. In this setting users can extend the set of supported heuristics and combinations by reprogramming parts of the `search/6` predicate.

We see the same approach in other Prolog systems' CLP(FD) libraries, albeit to a more limited extent. **SICStus** Prolog (Carlsson and Mildner 2012) allows imposing discrepancy and time limits, and **B-Prolog** (Zhou 2012) provides only a time limit. **GNU Prolog** (Diaz et al. 2012) and **Ciao's** new `clpfd` library provide no limits on top of depth-first search.

We are aware of one other approach to modify Prolog's own search method: the breadth-first and iterative deepening program transformations in **Ciao** (Hermenegildo et al. 2012). These modify annotated predicates in place and are not compositional.

All in all the available library support that Prolog systems provide is very limited indeed. As soon as users face a constraint problem that requires a non-trivial search method, they are forced to write all their search code from scratch, and it can be very daunting to combine different search methods.

Our Solution We propose to solve the above modularity problem concerning search methods by decoupling the code that defines the search tree (e.g., `labeling/1`) from the code that defines the search method (e.g., `depth_bound/2` and `lds/1`²). The two specifications are combined by means of our `search/1` predicate. For instance, we express the three above scenarios as:

```
?- search(label([X1,...,Xn])).
?- search(depth_bound(10,label([X1,...,Xn]))).
?- search(lds(label([X1,...,Xn]))).
```

This approach does not suffer from the many disadvantages of the copy-paste-modify approach discussed above. In particular, the code for `label/1` is not touched, and the code for `depth_bound/2` and `lds/1` is provided in a library or supplied by the user. Existing search tree descriptions and search methods are easily and independently reused and maintained, with all the benefits of increased productivity and code quality.

Moreover, our approach is truly compositional. With the same three components we can express an additional search heuristic that combines a depth bound with `lds`:

```
?- search(depth_bound(10,lds(label([X1,...,Xn])))).
```

Only small syntactic changes are necessary to use our highly modular library based approach: Instead of using Prolog’s disjunction `(;)/2`, the search tree description must use our library’s disjunction predicate `tor/2`.

In the next section we reveal the technical details of our solution.

3 Elaboration

Many variants of Prolog’s default search method are most easily expressed as an action to be taken at the moment the alternative branches are entered, and possibly modifying the branches themselves.

Basically, we achieve this in a general way by providing a hook into disjunction as follows:

- programmers load our `library(tor)` and use the `Tor`-disjunction at appropriate places in their code, i.e., instead of `(;)/2` they use `tor/2`
- the library contains a definition of the `Tor`-disjunction – this definition can be partially evaluated away in many cases
- the library provides a means to specify what action needs to be taken in entering one of the `tor`-disjunctive branches: the definition of the `Tor`-disjunction uses these actions
- the system provides a set of useful and common actions; the user can implement additional actions himself, without the need to modify the library; neither does the user need to know to which program this new action will be applied

² `lds` stands for `limited-discrepancy search`

The next section shows a self-contained example.

3.1 The Wolf, the Goat and the Cabbage

We illustrate the basics of Tor on the well-known problem of the wolf, the goat and the cabbage. The following code, adapted from Sterling and Shapiro (1994), implements this decision problem. Naive depth-first execution of this code loops infinitely.

```
wgc :- initial_state(State),
      wgc(State).

wgc(State) :- final_state(State), !.
wgc(State) :- move(State,Move),
              update(State,Move,State1),
              legal(State1),
              wgc(State1).

initial_state(wgc(left, [wolf, goat, cabbage], [])).
final_state(wgc(right, [], [wolf, goat, cabbage])).

move(wgc(Bank, Left, Right),Move) :-
  ( ( Bank == left -> tor_member(Move, Left)
    ; Bank == right -> tor_member(Move, Right) )
  tor
  Move = alone
  ).

tor_member(X,[V|Vs]) :- ( X = V tor tor_member(X,Vs) ).

... % see Appendix
```

The nondeterminism in this code is situated in the `move/2` and `tor_member/2` predicates. Instead of the regular Prolog disjunction we use Tor's `tor/2` disjunction:

```
G1 tor G2 :-
  ( b_getval(left,Left),
    call(Left,G1)
  ; b_getval(right,Right),
    call(Right,G2)
  ).
```

This definition provides two hooks into the disjunction by means of global variables `left` and `right`. In these hooks the programmer installs handlers for the left and right branches to control the search. For instance, the following handler limits the depth:

```
dbs_handler(Branch) :-
  b_getval(depth_limit,Depth),
  NewDepth is Depth - 1,
```

```
NewDepth > 0,
b_setval(depth_limit, NewDepth),
call(Branch).
```

We can install this depth-bounded search handler as follows:

```
?- DepthLimit = 4, b_setval(depth_limit, DepthLimit),
   Action = dbs_handler, b_setval(left, Action), b_setval(right, Action),
   wgc.
```

The above way to specify and install handlers is clearly too clumsy. The next sections remedy this by introducing the necessary infrastructure.

3.2 Infrastructure

Implicit Disjunctions The `tor/1` declaration implicitly adds Tor-disjunctions between the clauses of a predicate. For instance, `tor_member/2` can be written more conventionally as:

```
:- tor tor_member/2.
tor_member(X, [X|_]).
tor_member(X, [_|Xs]) :- tor_member(X, Xs).
```

Default Handler The convenient predicate `search/1` sets up the default handler for both hooks: `call`.

```
search(Goal) :-
  b_setval(left, call),
  b_setval(right, call),
  call(Goal).
```

With this default handler, `tor/2` corresponds simply to plain disjunction `(;)/2`.³ For instance, with `search/1` we recover the behavior of `label/1` of Fig. 1 from the Tor-variant `tor_label/1` in Fig. 2.

```
search(tor_label(Vars)) ≡ label(Vars)
```

Custom Handlers In order to facilitate installing new handlers, we provide two convenient predicates.

Firstly, `tor_handlers/3` composes the currently installed handlers with the ones provided. Then it runs the provided goal and finally, it resets the installed handlers.

³ Apart from the scope of any cuts in the alternative branches

```

tor_label([]).
tor_label([Var|Vars]) :-
  ( var(Var) ->
    fd_inf(Var,Value),
    (   Var #= Value,
      tor_label(Vars)
    tor Var #\= Value,
      tor_label([Var|Vars])
    )
  ;
  tor_label(Vars)
).

```

Fig. 2. Labeling predicate with hookable disjunction.

```

tor_handlers(Goal,Left,Right) :-
  b_getval(left,LeftHandler),
  b_getval(right,RightHandler),
  b_setval(left,compose(LeftHandler,Left)),
  b_setval(right,compose(RightHandler,Right)),
  call(Goal),
  b_setval(left,LeftHandler),
  b_setval(right,RightHandler).

compose(G1,G2,Goal) :- call(G1,call(G2,Goal)).

```

Section 3.4 shows that this approach enables composing different search methods.

Secondly, in many cases, the handler only needs to precede the actual branch goal by its own goal, much like *before-advice* in Aspect Oriented Programming (Kiczales et al. 1997). For that purpose we introduce the `tor_before_handlers/3` predicate.

```

tor_before_handlers(Goal,Left,Right) :-
  tor_handlers(Goal,before(Left),before(Right)).

before(G1,G2) :- G1, G2.

```

The use of `tor_handlers/3` and `tor_before_handlers/3` is exemplified in Section 3.3, but the idea is already clear from their definition: a search method applied to a goal installs a handler and calls the goal.

3.3 Specifying Search Methods

With the above infrastructure, it is easy to write various search methods in a modular way.

Depth Bounded Search The predicate `dbs/2` implements the depth-bounded search method. While the monolithic code on the right in Figure 1 keeps track of the

remaining depth bound in an auxiliary parameter of the `label/2` predicate, this modular implementation uses a non-intrusive global variable instead. To be truly modular and to permit several simultaneously active depth bounded searches, a new global variable is generated for each instance. The handler code then checks in every branch whether the depth bound is not yet exceeded before decrementing it. These updates to the global variable are undone on backtracking.

```
dbs(Depth,Goal) :-
    gensym(db,Var),
    b_setval(Var,Depth),
    tor_before_handlers(Goal,dbs_handler(Var),dbs_handler(Var)).

dbs_handler(Var) :-
    b_getval(Var,N),
    N > 0,
    N1 is N - 1,
    b_setval(Var,N1).
```

The query in the example of Section 3.1 can now be written as:

```
?- search(dbs(4,wgc(InitialState,FinalState))).
```

Now, we easily obtain the behavior of `label/2` of Fig. 1 as follows:

```
search(dbs(Depth,tor_label(Vars))) ≡ label(Vars,Depth)
```

Discrepancy-Bounded Search The discrepancy-bounded search heuristic is a small variant of depth-bounded search: the bound is only updated in right branches.

```
dibs(Discrepancies,Goal) :-
    gensym(dib,Var),
    b_setval(Var,Discrepancies),
    tor_before_handlers(Goal,true,dbs_handler(Var)).
```

Iterative Deepening Iterative deepening emulates breadth-first search by means of increasing depth-bounds. The implementation below makes use of two variables: `DVar` to keep track of the current depth, and `PVar` to record whether the depth limit has been enforced in the current iteration. The handler `id_handler/2` checks and updates these variables in every node of the search tree. The driver `id_loop/2` initiates each iteration and, if pruning occurred, starts the next one.

```
id(Goal) :-
    gensym(idp,PVar),
    gensym(idd,DVar),
    id_loop(Goal,DVar,0,PVar).

id_loop(Goal,DVar,Depth,PVar) :-
    b_setval(DVar,Depth),
    nb_setval(PVar,not_pruned),
    Handler = id_handler(DVar,PVar),
```



```

( tor_before_handlers(Goal,Handler,Handler)
; nb_getval(PVar,Value),
  Value == pruned,
  NDepth is Depth + 1,
  id_loop(Goal,DVar,NDepth,PVar)
).

id_handler(DVar,PVar) :-
  b_getval(DVar,N),
  ( N > 0 ->
    N1 is N - 1,
    b_setval(DVar,N1)
  ; nb_setval(PVar,pruned),
    false
  ).

```

Limited Discrepancy Search The traditional limited discrepancy search (Harvey and Ginsberg 1995) is a minor variant of iterative deepening. It applies the depth-bound only in right branches.

Branch-and-Bound Optimization This well-known optimization approach posts constraints in the intermediate nodes of the search tree to find increasingly better solutions. Our implementation uses **Tor** to access those intermediate nodes and generate increasingly larger values of the **Objective** variable. It uses two global variables, **BestVar** and **CurrentVar**. The former keeps track of the overall best solution so far, while the latter is the solution that the current node tries to improve upon.

Both the overall and current best solution are initialized to a value smaller than the infimum of the objective variable's domain. Whenever a solution is found, the overall best solution is updated. Whenever we backtrack into a **Tor** choicepoint, the handler synchronizes the current best solution with the overall best solution. If the current best solution was out of sync, the handler also imposes a new lower bound on the objective variable. Note that **inf** denotes negative infinity.

```

bab(Objective,Goal) :-
  gensym(bab,BestVar),
  gensym(bab,CurrentVar),
  fd_inf(Objective,Inf),
  Best is Inf - 1,
  nb_setval(BestVar,Best),
  b_setval(CurrentVar,Best),
  Handler = bab_handler(Objective,BestVar,CurrentVar),
  tor_before_handlers(Goal,true,Handler),
  nb_setval(BestVar,Objective).

bab_handler(Objective,BestVar,CurrentVar) :-
  nb_getval(BestVar,Best),
  b_getval(CurrentVar,Current),
  ( Best \= inf , (Current == inf ; Best > Current ) ->
    Objective #> Best,

```

```

        b_setval(CurrentVar,Best)
    ;   true
    ).

```

More We have implemented many other orthogonal search methods with `tor`, including all those offered by ECLiPSe’s `search/6` predicate.

See our online code supplement: <http://users.ugent.be/~tschrijv/tor>.

3.4 Composing Handlers

The beauty of `Tor` is that handlers can be composed if they avoid interference on global variables. All examples we present in this paper satisfy this condition. Below, we illustrate three kinds of composition.

Firstly, several handlers can be active at the same `Tor`-disjunction. For instance, we can perform branch-and-bound optimization with a depth-bound as follows:

```
?- ..., search(bab(Objective,(dbs(DepthLimit,tor_label(Vars))))).
```

Secondly, different handlers can be used for different parts of the search space. For instance, we can label the `Xs` and `Ys` variables each with their own depth limit:

```
?- ..., search((dbs(XsLimit,tor_label(Xs)), dbs(YsLimit,tor_label(Ys)))).
```

Note that the global variables necessary for the two depth bounds exist simultaneously, but, because we use different ones, they do not interfere.

Finally, one handler can relinquish control to another one. For instance, inspired by ECLiPSe’s `search/6` predicate, we provide a variant `dbs/3` of depth-bounded search that does not fail when the bound is reached, but switches to another strategy. While ECLiPSe only offers two alternatives for the other strategy, `Tor` allows any handler. For instance, once the depth-limit is reached, we visit only a fixed number of nodes with:

```
?- ..., search(dbs(DepthLimit,nbs(NodeLimit),tor_label(Vars))).
```

3.5 Search Tree Observation

`Tor` does not only allow us to manipulate the traversal of the search tree in various search heuristics. It also enables us to observe the search tree in different ways in order to gain insight in the search process for (performance) debugging purposes.

Statistics Similar to SWI-Prolog’s `profile/1`, `time/1` and `statistics/0` predicates, we can provide different components that monitor various metrics of the search tree and provide us with a convenient summary.

```

?- length(Xs,4), Xs ins 1..4,
   search(statistics((tor_label(Xs),writeln(Xs)))), false.
...
[4,4,4,4]

```

```
% Number of solutions: ..... 256
% Number of nodes: ..... 510
% Number of failures: ..... 0
```

The code for `statistics/1` is in the `Tor` library.

To support users who want to check whether they have successfully replaced all regular disjunctions with `Tor`, we also provide a tool that uses SWI-Prolog’s choice point inspection primitives (like `prolog_current_choice/1`) to verify this.

Visualisation In addition to summarized data of the search tree, we can also visualize the actual search tree itself. For that purpose, we provide a predicate that emits a textual representation, a log, of the search tree:

```
log(Goal) :-
    tor_handlers(Goal,log_handler(left),log_handler(right)),
    writeln(solution).

log_handler(Side,Goal) :-
    ( writeln(Side), call(Goal)
    ; writeln(false), false).
```

A complimentary tool that turns this log into a PDF image is also available from our public code repository.

Fig 3 shows two search trees for the 8-queens puzzle: The left one was created with depth limit (search strategy `dbs`) 4, and the right one with depth limit 7, where we stopped the search after finding the first solution. The symbol $\perp$ denotes pruning due to constraint propagation, and $\top$ denotes a solution. The symbol $!$ denotes that a node is not explored because the depth limit is exceeded at this level of the search tree.

Fig. 3. Search trees of 8-queens with depth bound 4 and 7

4 Evaluation

4.1 Automatic Specialization

`Tor` encourages writing fairly abstract and generic code. This style clearly incurs some overhead (notably due to meta-calling) compared to specialized search code.

In order to mitigate that overhead, we exploit Prolog’s homoiconic nature to provide a simple but effective automatic specializer.

Even though there is a large body of work on automatic program specialization for Prolog, notably involving partial evaluation, we decided to write our own program specializer. Its main tasks are 1) to perform *constant propagation* on the global variables `left` and `right`, 2) to replace instantiated meta-calls by direct calls and 3) to inline the handler code into the main search loop. For control we follow a light-weight approach based on declarations of what predicates to inline and specialize.

Example 1 Our specializer yields `label/1` for `search(tor_label(Vars))`. Similarly, we recover SWI-Prolog’s `labeling/2` by specializing its `Tor` variant.

Example 2 The specialized form of the goal `search(dbs(N,tor_label(Vars)))` is `gensym(db, DVar), b_setval(DVar, N), label21(Vars, DVar)`, with:

```
label21([], _).
label21([Var|Vars], DVar) :-
  ( var(Var) ->
    fd_inf(Var, Val),
    ( b_getval(DVar, Depth),
      Depth>0,
      NDepth is Depth+ -1,
      b_setval(DVar, NDepth),
      Var#=Val,
      label21(Vars, DVar)
    ; b_getval(DVar, G),
      G>0,
      NDepth is G+ -1,
      b_setval(DVar, NDepth),
      Var#\=Val,
      label21([Var|Vars], DVar)
    )
  ; label21(Vars, DVar)
).
```

This code is slightly less efficient than that of `label/2`. Firstly, the overhead of global variables is not entirely eliminated here, as `DVar` is still present. Secondly, the two branches have some code in common that could be shared. However, there are no more meta-calls and all code is inlined in the recursive loop of `label21/2`.

In future work, we intend to get rid of the remaining inefficiencies by implementing additional transformations, including Peter Schachte’s approach (Schachte 1997) for eliminating global variables adapted to our setting.

4.2 Benchmarks

To study `Tor`’s overhead, we have performed a number of benchmarks on a MacBook Pro with a 2.4 GHz CPU and 4 GB RAM, running SWI-Prolog 5.11.7 in Mac OS X 10.6.7. Thanks to specialization, plain labeling code with and without `Tor` is identical and their performance too of course. Hence, to get an idea of the specializer’s impact, we disable the specializer for the following benchmarks.

	our label/1		clpfd's label/1		search/6	
	man	Tor	man	Tor	man	Tor
N=6,D= 8	1.80 s	240 %	2.08 s	151 %	2.55 s	132 %
N=6,D= 9	3.63 s	249 %	4.20 s	153 %	5.09 s	135 %
N=6,D=10	6.82 s	269 %	7.87 s	155 %	9.53 s	137 %
N=7,D= 8	14.44 s	244 %	16.63 s	153 %	20.40 s	134 %
N=7,D= 9	32.80 s	269 %	37.80 s	155 %	46.04 s	136 %
N=7,D=10	68.27 s	278 %	78.63 s	157 %	94.30 s	139 %

Fig. 4. Labeling benchmarks without propagation

Pure Search Figure 4 considers the extreme situation where the search is pure enumeration of *unconstrained* constraint variables: `length(N,Vars)`, `Vars ins 1..D`. Hence, no constraint propagators are activated due to choices.

The first column denotes the problem size, expressed in the number of variables `N` and their domain size `D`. The other three pairs of columns denote different implementations of labeling: 1) `label/1` as listed in this paper, 2) `label/1` from SWI-Prolog's `clpfd` library, and 3) `search/6` ported from ECLiPSe to SWI-Prolog with minimal changes. For each of these, we show the absolute runtime of the standard/manual version (`man`) and the relative runtime of the `Tor` version (`tor`).

The impact of `Tor` is pretty consistent across the problem sizes, but depends on the labeling implementation. The overhead is most prominent (140-180 %) in our barebones `label/1`, while it is less so (50-60 %) in `clpfd`'s `label/1`. The latter delegates to `labeling/2`, which involves more generic option processing. Finally, in `search/6` `Tor` compensates its overhead further (to 30-40 %) by not collecting search statistics when these are not demanded. In the original version, these statistics are collected regardless of demand.

In summary, in these propagation-free benchmarks, the overhead of `Tor` is never more than a factor of three for the tight labeling loop and less than 60 % for the conventional option-rich labeling predicates. All in all, we find this is a very reasonable price to pay for the extra flexibility that `Tor` provides. Still, invoking the `specializer` is warranted to get rid of all overhead.

Search vs. Propagation While the overhead of `Tor` is bounded in the previous benchmarks, the performance-wary user may not be willing to accept the modest overhead. However, the previous benchmarks are not representative of realistic CLP problems, that spend a lot of time on constraint propagation in every node of the search tree. All this extra work easily dwarfs the overhead of `Tor`. Figure 5 illustrates this observation on a number of typical CLP benchmarks. For added realism, the benchmarks use the first-fail variable selection strategy, with hand-written labeling code `ff_label/1` and the two library predicates `labeling/2` (SWI-Prolog) and `search/6` (ECLiPSe).

	our ff_label/1		labeling/2		search/6	
	man	Tor	man	Tor	man	Tor
allinterval	4.03 s	101 %	4.02 s	101 %	4.01 s	101 %
golf	3.93 s	99 %	3.92 s	100 %	3.96 s	99 %
mhex	18.59 s	102 %	18.61 s	101 %	18.46 s	101 %
n_queens	2.03 s	103 %	2.05 s	102 %	2.09 s	102 %
sudoku	2.14 s	101 %	2.15 s	101 %	3.40 s	100 %

Fig. 5. Labeling benchmarks with propagation

	plain	lds	dibs-1	dibs-2	credit/bbs
N= 95	2.11 s	0.66 s	0.45 s	0.28 s	0.33 s
N= 96	0.65 s	4.98 s	4.89 s	1.13 s	1.04 s
N= 97	T/O	3.68 s	3.56 s	22.66 s	4.08 s
N= 98	T/O	15.67 s	† 5.71 s	10.16 s	2.50 s
N= 99	T/O	2.42 s	2.22 s	9.85 s	2.57 s

† no solution

Fig. 6. N-Queens benchmarks with various search methods

Search Methods Finally, Figure 6 illustrates again why we want to use different search methods: they can have a significant impact on runtime. The figure shows the runtime of the **n_queens** benchmark for 5 different problem sizes and 5 different search methods: (plain) plain depth-first search, (lds) limited discrepancy search, (dibs-1/-2) discrepancy bounds of 1 and 2, and (credit/bbs) credit-based search with 10,000 credits that switches to a bounded backtracking (1 backtrack) search when the credits are exhausted.

5 Related Work

Earlier Work Tor is related to earlier work on *Monadic Constraint Programming* (MCP) (Schrijvers et al. 2009) in the context of Haskell, and *Search Combinators* (Schrijvers et al. 2011) in the context of C++ and the Gecode library⁴. In contrast to those works, Tor is tailored towards Prolog’s built-in depth-first search and, as a consequence, consists of a much simpler and more elegant design.

Aspect-Oriented Programming The Tor approach is closely related to Aspect-Oriented Programming (AOP) (Kiczales et al. 1997). AOP provides a generic approach for modularly *crosscutting* existing code with new code, so-called *advice*. This advice is injected in arbitrary *join points* (i.e., program points) based on a *pointcut* predicate.

⁴ <http://www.gecode.org>

Obviously Tor is more limited in scope, as only `tor/2` disjunctions are crosscut and only at the positions of the two hooks. However, we believe that these “limitations” are actually Tor’s strength: its simplicity makes it easy to express all common search methods and its discipline favors compositionality.

6 Conclusion

We have presented Tor, a light-weight library-based approach for modifying Prolog’s depth-first search with reusable and compositional search methods.

In future work, we will investigate ways to replace the underlying depth-first queuing strategy. The stack freezing functionality of tabling systems like XSB (Swift and Warren 2012) and YAP (Santos Costa et al. 2012) provides interesting perspectives for this purpose.

References

- CARLSSON, M. AND MILDNER, P. 2012. SICStus Prolog - The first 25 years. *Theory and Practice of Logic Programming* 12, 1-2, 35–66.
- DIAZ, D., ABREU, S., AND CODOGNET, P. 2012. On the implementation of GNU-Prolog. *Theory and Practice of Logic Programming* 12, 1-2, 253–282.
- HARVEY, W. D. AND GINSBERG, M. L. 1995. Limited discrepancy search. In *Proceedings of the 15th International Joint Conferences on Artificial Intelligence (IJCAI 1995)*. 607–613.
- HERMENEGILDO, M. V., BUENO, F., CARRO, M., LÓPEZ-GARCÍA, P., MERA, E., MORALES, J. F., AND PUEBLA, G. 2012. An overview of Ciao and its design philosophy. *Theory and Practice of Logic Programming* 12, 1-2, 219–252.
- KICZALES, G., LAMPING, J., MENHDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J., AND IRWIN, J. 1997. Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP 1997)*.
- KOWALSKI, R. 1979. *Logic for Problem Solving*. North-Holland.
- SANTOS COSTA, V., ROCHA, R., AND DAMAS, L. 2012. The YAP Prolog system. *Theory and Practice of Logic Programming* 12, 1-2, 5–34.
- SCHACHTE, P. 1997. Global variables in logic programming. In *Proceedings of the International Conference on Logic Programming (ICLP 1997)*. 3–17.
- SCHIMPF, J. AND SHEN, K. 2012. ECLiPSe From LP to CLP. *Theory and Practice of Logic Programming* 12, 1-2, 127–156.
- SCHRIJVERS, T., STUCKEY, P. J., AND WADLER, P. 2009. Monadic constraint programming. *Journal of Functional Programming* 19, 6, 663–697.
- SCHRIJVERS, T., TACK, G., WUILLE, P., SAMULOWITZ, H., AND STUCKEY, P. 2011. Search Combinators. In *Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP 2011)*. Springer.
- STERLING, L. AND SHAPIRO, E. 1994. *The Art of Prolog: Advanced Programming Techniques*, 2. ed. MIT Press, Cambridge, MA.
- SWIFT, T. AND WARREN, D. S. 2012. XSB: Extending Prolog with Tabled Logic Programming. *Theory and Practice of Logic Programming* 12, 1-2, 157–187.
- TRISKA, M. 2012. The finite domain constraint solver of SWI-Prolog. In *Proceedings of the 11th International Symposium on Functional and Logic Programming (FLOPS 2012)*. 307–316.

- WIELEMAKER, J., SCHRIJVERS, T., TRISKA, M., AND LAGER, T. 2012. SWI-Prolog. *Theory and Practice of Logic Programming* 12, 1-2, 67–96.
- ZHOU, N.-F. 2012. The language features and architecture of B-Prolog. *Theory and Practice of Logic Programming* 12, 1-2, 189–218.

Appendix A Full Source Code of the Wolf-Goat-Cabbage Problem

```

solve_dfs :-
    initial_state(State),
    solve_dfs(State).

solve_dfs(State) :-
    final_state(State), !.
solve_dfs(State) :-
    move(State, Move),
    update(State, Move, State1),
    legal(State1),
    solve_dfs(State1).

initial_state(wgc(left, [wolf, goat, cabbage], [])).

final_state(wgc(right, [], [wolf, goat, cabbage])).

move(wgc(Bank, Left, Right), Move) :-
    ( Bank == left,
      enum(Move, Left)
    tor
      Bank == right,
      enum(Move, Right )
    tor
      Move = alone
    ).

update(wgc(B,L,R), Cargo, wgc(B1, L1, R1)) :-
    update_boat(B, B1),
    update_banks(Cargo, B, L, R, L1, R1).

update_boat(left, right).
update_boat(right, left).

update_banks(alone, _B, L, R, L, R) :- !.
update_banks(Cargo, left, L, R, L1, R1) :- !,
    select(Cargo, L, L1),
    insert(Cargo, R, R1).
update_banks(Cargo, right, L, R, L1, R1) :-
    select(Cargo, R, R1),
    insert(Cargo, L, L1).

insert(X, [Y|Ys], [X,Y|Ys]) :-
    precedes(X,Y), !.

```

```
insert(X, [Y|Ys], [Y|Zs]) :-  
    precedes(Y,X), !,  
    insert(X,Ys,Zs).  
insert(X, [], [X]).  
  
precedes(wolf, _X).  
precedes(_X, cabbage).  
  
legal(wgc(left, _L, R)) :- \+ illegal(R).  
legal(wgc(right, L, _R)) :- \+ illegal(L).  
  
illegal(Bank) :- memberchk(wolf, Bank), memberchk(goat, Bank).  
illegal(Bank) :- memberchk(goat, Bank), memberchk(cabbage, Bank).
```