TOR: Modular Search with Hookable Disjunction

Benoit Desouter[*]and Tom Schrijvers[†]

Ghent University, Belgium

March 8, 2013

Abstract

Horn Clause Programs have a natural exhaustive depth-first procedural semantics. However, for many programs this 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. TOR supports modular composition of search methods and other hooks. The TOR library is already provided and used as an add-on to SWI-Prolog.

1 Introduction

Kowalski's well-known adage [1] crisply captures the essence of programming in the equation:

$$\text{ALGORITHM} = \text{LOGIC} + \text{CONTROL}$$

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. Syntactically, logic and control in Prolog are tightly coupled, and adding a different control means cross-cutting existing code.

Our novel approach to adding, in an orthogonal manner, control, features the following properties:

- It is a light-weight and efficient library-based approach that is easily portable; it is currently an SWI-Prolog library [2] available at `http://www.swi-prolog.org/pack/list?p=tor`.
- Our approach has all the benefits of modularity: search methods can be composed and the library of these heuristics is (user-)extensible.

With TOR, we capture all common search methods in CLP(FD) libraries. This approach is indeed particularly suitable for Constraint Logic Programming, but also useful for general Prolog programs with a large search space.

For a more thorough discussion of TOR, we refer the interested reader to [3]. An earlier version is discussed in [4].

[*]`Benoit.Desouter@UGent.be`

[†]`Tom.Schrijvers@UGent.be`

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

```
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 )
  ).
```

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

2 Problem Statement

We illustrate the heart of the matter on a simple labeling predicate `label/1` written against SWI-Prolog's `clpfd` library [5] (see Fig. 1, left). `label/1` defines a search tree where the branches are created by the disjunction.[1]

Suppose that for a certain call `label([`X_1`,...,`X_n`])` the search tree is too large to fully explore. In order to get some useful answers, certain parts of the tree can be left unexplored. This can for example be achieved by imposing a *depth bound* on Prolog's depth first search (Figure 1 right).

Imposing a depth bound may or may not be a successful approach to getting useful answers. In case it is not, other pruning strategies can be tried, like imposing a node bound or a discrepancy bound. Each of these requires rewriting the `label/1` predicate to incorporate a different pruning technique.

The problems with the above approach should be apparent:

- It follows the well-known copy-paste-modify anti-pattern.
- The same heuristic is implemented over and over in different settings. 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, leading to suboptimal solutions.
- As soon as the labeling code spans several different predicates or multiple invocations of the same predicate, the complexity of adding search heuristics increases drastically.

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

3 Solution Overview

3.1 User Perspective

TOR divides search code into two parts: a) the code that defines the *search tree*, and b) the code that defines the *search method*. The user defines these separately (or reuses library definitions) and combines them into a search goal. This decoupling means that new search methods and new search tree code can be written without awareness of one another and without the modification of any existing code.

Search Tree Code The search tree code sets up the problem specific search tree. To fit in the TOR framework, the code must use TOR's custom disjunction `tor/2` rather than `;/2`. For instance, `tor_label/1` is the TOR-compatible variant of `label/1`:

```
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)
  ).
```

Search Methods A search method is defined as a predicate that captures the essence of that method in a declarative way, as a bare-bones search tree without any useful work (such as labeling variabels). For instance, `dbs_tree/1` captures the depth-bounded search method.

```
dbs_tree(Depth) :-
  Depth > 0,
  Depth1 is Depth - 1,
  ( dbs_tree(Depth1)
  tor
    dbs_tree(Depth1)
  ).
```

Search methods change less frequently. They are usually written by advanced users and library writers. For instance, TOR itself comes with a library of predefined search methods.

Combining Search Tree and Search Method The user imposes a search method on a search tree by calling the TOR predicate `tor_merge(MGoal,TGoal)`, where `MGoal` is a call to the search method predicate and `TGoal` is a call to the search tree predicate. Conceptually, `tor_merge/2` overlays or merges the search trees of the two goals, synchronizing their `tor/2` disjunctions.

To facilitate reuse, we generally recommend to encapsulate the application of `tor_merge/2` to a particular search method in a separate predicate, like `dbs/2` for `dbs_tree/1`.

```
dbs(Depth,Goal) :-
  tor_merge(dbs_tree(Depth),Goal).
```

Wrapping Up In the final step, the TOR predicate `search(Goal)` is used to, conceptually, replace all the occurrences (merged or not) of `tor/2` by proper Prolog disjunctions.

In summary, the behavior of `label/2` of Fig. 1 is recovered as follows:

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

3.2 Modularity Aspects

Modular Composition of Search Tree Code and Search Method Even though their implementations are decoupled, any search tree and any search method code can be combined out of the box with the help of TOR's `search/1` predicate. For instance, a user can define a complex labeling goal as the conjunction of two invocations of `tor_label/1`. It is easy to express a different scenario, either by varying the search tree or the search method code. By `lds`, we denote the limited discrepancy search method; `nbs` stands for node-bounded search:

```
?- search(lds((tor_label([X1,...,Xn]), tor_label([Y1,...,Ym])))).
?- search(nbs(30000,(tor_member([X1,...,Xn]), tor_member([Y1,...,Ym])))).
```

It becomes even more interesting if you label both lists with different variable and value selection strategies.

Modular Composition of Multiple Search Heuristics A further modularity advantage of TOR is that it provides nested invocation as an out of the box way to compose two (or more) search methods. Nesting denotes that both search methods are simultaneously active.

For instance, we can simultaneously apply a depth-limit and perform a limited discrepancy search:

```
?- search(dbs(10,lds(tor_label([X1,...,Xn])))).
```

Contrast this with the non-modular approach were the user would have to write a combined search heuristic `dbs_lds/2` from scratch.

Finally, the compositional nature of the notation can be exploited to its fullest potential to obtain sofisticated search specifications. For instance, the goal

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

applies limited discrepancy search to the whole search tree, and additionally imposes one depth-limit on the search of the Xs and another to that of the Ys. Such a composite heuristic is not readily expressible with any of the existing CLP libraries.

4 Heuristics Library

Following the above approach, it is easy to write various modular search methods yourself. However, Tor already provides a substantial library of which we we cover only two here. Many others can be found in [3].

4.1 Node-Bounded Search

A node-bounded search is much like a depth-bounded search, except that the decrements of the limit are not backtracked. Hence, as an optimization we abort the whole search at once by throwing an exception.

```
nbs(Nodes,Goal) :-
  new_nbvar(Nodes,NodesVar),
  catch(
    tor_merge(nbs_tree(NodesVar),Goal),
    out_of_nodes(NodesVar),
    fail
  ).

nbs_tree(Var) :-
  nb_get(Var,N),
  ( N > 0 ->
    N1 is N - 1,
    nb_put(Var, N1),
    ( nbs_tree(Var)
    tor
      nbs_tree(Var)
    )
  ;
    throw(out_of_nodes(Var))
  ).
```

4.2 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 variables, BestVar and Current. 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 heuristic 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) :-
  fd_inf(Objective,Inf),
  LowerBound is Inf - 1,
  new_nbvar(LowerBound,BestVar),
  Current = LowerBound,
  tor_merge(bab_tree(Objective,BestVar,Current),Goal),
  nb_put(BestVar,Objective).

bab_tree(Objective,BestVar,Current) :-
  nb_get(BestVar,Best),
  ( Best \= inf , (Current == inf ; Best > Current ) ->
      Objective #> Best,
      NCurrent = Best
  ;
      NCurrent = Current
  ),
  ( bab_tree(Objective,BestVar,NCurrent)
  tor
    bab_tree(Objective,BestVar,NCurrent)
  ).
```

5 Search Tree Observation

Tor enables various ways to observe the search tree, so that one can gain insight in the search process itself, e.g., for (performance) debugging purposes. We provide different components that monitor various metrics of the search tree.

We can also visualize the actual search tree. For that purpose, we provide a predicate `log/1` that emits a textual representation of the search tree. A complimentary tool that turns this log into a PDF image is also available in the Tor release.

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. The notion of hookable disjunction enables a surprisingly large number of possibilities for modifying Prolog search.

References

[1] R. Kowalski, *Logic for Problem Solving.* North-Holland, 1979.

[2] J. Wielemaker, T. Schrijvers, M. Triska, and T. Lager, "SWI-Prolog," *Theory and Practice of Logic Programming*, vol. 12, no. 1-2, pp. 67–96, 2012.

[3] T. Schrijvers, B. Demoen, M. Triska, and B. Desouter, "Tor: Modular search with hookable disjunction." To appear in Science of Computer Programming, 2013.

[4] T. Schrijvers, M. Triska, and B. Demoen, "Tor: extensible search with hookable disjunction," in *Proceedings of the 14th symposium on Principles and practice of declarative programming*, PPDP '12, (New York, NY, USA), pp. 103–114, ACM, 2012.

[5] M. Triska, "The finite domain constraint solver of SWI-Prolog," in *Proceedings of the 11th International Symposium on Functional and Logic Programming (FLOPS 2012)*, vol. 7294 of *Lecture Notes in Computer Science*, pp. 307–316, Springer, 2012.