Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 3 (2011) 1490–1493

Procedia Computer Science

www.elsevier.com/locate/procedia

WCIT 2010

Proposing a New Search Template for Modelling Languages

Reza Rafeh

Department of Computer Engineering
Arak University, Arak, Iran
r- rafeh@araku.ac.ir

Abstract

The major problem in solving combinatorial optimization problems is the huge size of the search space. To explore the search space in a reasonable time, using smart search algorithms is inevitable. One of the main difficulties in implementing search methods is the lack of a uniform, high-level template for all search paradigms. In this paper, we propose a high-level, parametric template suitable for modeling languages which covers both tree search and local search.

Keywords: Combinatorial Optimization Problems; Search Space; Tree Search; Local Search; Zinc Modelling Language

Tackling combinatorial optimization problems consists of two major steps: modeling and solving. Modelling means formulation the problem precisely, while solving involves finding a pro per value for each variable. The most popular techniques for solving constraint decision problems come from three main areas: constraint programming techniques, mathematical methods and local search techniques. For a given problem, different techniques can have quite different efficiency and it is often unclear which one is the best before performing systematic experimentation.

Regarding the modelling step, the most popular tools come from four main areas: constraint programming languages, constraint libraries, (mathematical) modelling languages and specification languages. The most high-level practical modelling is provided by modelling languages. This is because, on the one hand, modelling languages do not require modellers to be skilled programmers (opposed to constraint programming languages and libraries), and on the other hand, they are implementable (as opposed to generic specification languages).

Zinc [1] is a high-level modeling language designed to support experimentation with different solving techniques. Conceptual models in Zinc can be automatically mapped into design models that use one of the following three solving techniques: constraint programming (CP); Mixed Integer Programming (MIP)r; and incomplete search using local search methods.

Using the default search performs well for MIP, but for CP and local search, efficiency often depends on the modeller providing an effective, model-specific search strategy. Therefore, users need to specify search routines for their models. However, allowing users to define their search routines requires the integration of a conceptual model and a search strategy, something that is difficult to achieve cleanly since while the former is best expressed declaratively, the latter is inherently procedural.

In this paper we propose a search template suitable for high-level modeling languages as Zinc. Our first implementation consists of three high-level search patterns for backtracking search, branch and bound, and local search. Zinc search routines take complex expressions, functions and predicates as parameters.

1. Backtracking search in Zinc

doi:10.1016/j.procs.2011.01.037

Before illustrating the use of search patterns, we explain the structure of Zinc models by means of a simple example, the N-queens problem, which tries to place n queens on an n × n chess board in such a way that no two queens can attack each other. Here is a Zinc model for the problem:

```
int: n;
type Domain = 1..n;
array[Domain] of var Domain :q;
predicate noattack(Domain: i,j, var Domain: qi,qj) =
qi != qj /\ qi + i != qj + j /\ qi - i != qj - j;
constraint forall(i,j in Domain where i<j)
noattack(i,j,q[i],q[j]);
solve satisfy;
```

Variable n is defined as an integer parameter. Domain is a new type for the range 1..n, and q is an array of n finite domain decision variables (indicated by the keyword **var**) over that range. For our purposes, the most interesting feature of Zinc is that it allows the user to define new predicates and functions. In the above example, the **noattack** predicate is defined, which succeeds if queens **qi** and **qj** of rows **i** and **j** respectively, cannot take each other (/\ is used for conjunction). By using the **forall** expression, the constraint applies the **noattack** predicate on each pair of queens to ensue they cannot attack each other. The last line declares the model to be a satisfaction problem. If the solve item has no annotation for search (like in this model), Zinc uses the default search to solve the model.

Depth-first search pattern **backtrack(init,expand)** can be used for solving satisfaction problems with backtracking search using a propagation solver. The first argument, init, is the state of the root node in the search tree. This is often the list of variables to label, but can be anything the modeller needs to create choice points, and can include extra information such as a counter to implement iterative deepening. The second argument, expand, is a (possibly user-defined) function that takes the state for the current node and returns its children as a list of pairs of the form **(ns, c)**, where ns is the child's state, and **c** the constraint that should be posted right before this child becomes the current node. It is worth pointing out that expand has implicit access to the solver state and, therefore, can call standard propagation solver reflection functions such as domain(V), which returns the current domain of variable V.

For examle, we can use a standard labeling search for the N-queens model by annotating the solve item as follows:

```
solve satisfy::backtrack(q,std_label);
```

where the initial local state is the list of variables q and function std label is the expand function. It is defined by:

```
function list of tuple(list of $T, var bool): std_label(list of $T:Vs) =
if Vs = [] then []
else [ (tail(Vs), head(Vs) == d) | d in domain(head(Vs))]
endif;
```

which takes a list of variables **Vs** (with polymorphic type list of **$T**) and (by using a list comprehension) for each variable **V** in **Vs** returns a list of tuples in which the first element is the remaining variables (and will be the state of the children nodes) and the second element is an equality constraint between the variable and a value from its domain. The head and tail functions are provided in the Zinc library and return the head and tail of the input list, respectively. Since the output of domain is a set and Zinc's sets are ordered, the domain values are considered from smallest to largest. To instantiate each variable, the backtracking search tries the constraints returned by std label in order.

2. Branch-and-bound

The search pattern **backtrack(init,expand,bound,flag)** is used for optimization problems to implement the backtracking pattern extended with branch and bound. This is used as an annotation to the solve item which is either in the form **solve maximize expr** or **solve minimize expr** for maximization and minimization problems, respectively.

The first two arguments of the pattern are similar to the previous pattern. The two extra arguments are a function, bound, for computing the new bound from the previous and current bounds, and a flag to indicate the kind of branch-and-bound search performed. The flags are similar to those provided in ECLiPSe [2], and include **restart** (to restart the search from the root of the search tree), **continue** (to continue the search from the current node in the search tree), and **dichotomic** (to do dichotomic search).

3. Local search

Local search methods (such as hill-climbing or simulated annealing) try to improve a single valuation by moving to a neighbour. Such methods can be implemented for a model using the search pattern **local_search(init valn, init state,move, finish)**, which takes as arguments the initial valuation (list of variable/value pairs), the initial state information, a function move that takes the current state and returns the new valuation to move to (this needs only to give the values for variables that have been changed in the move) along with the new state, and a function finish that takes the state and indicates whether the search should finish.

Similar to Comet [3], Zinc support these reflection functions: **val(V)** gives the value of variable **V** in the valuation, **var_penalty(V)** the degree of violation associated with variable **V** , **penalty(C)** the violation of constraint **C**, current penalty the total penalty for the current valuation, and **new_penalty(Val)** the total penalty that will result if the changes in valuation V al are applied to the current valuation.

The modeller can then specify, for example, a simple hill-climbing search routine by annotating the solve item as:

```
solve satisfy::local_search([(q[i],i)|i in Domain],1000,move,finish);
```

where initially the ith queen is placed on row i and the initial state is simply the maximum permitted number of moves. The move function is:

```
function valuation: swap($T: v1, $T: v2) = [(v1,val(v2)),(v2,val(v1))];
function tuple(int, valuation): move(int: nmovesleft) =
let {int: i=maximizes(q,var_penalty),
int: j=minimizes([swap(q[i],q[k])|k in Domain], new_penalty)
} in
(nmovesleft-1,swap(q[i],q[j]));
function has_ended: finish(int: nmovesleft) =
if current_penalty == 0 then sol(get_valuation)
elseif nmovesleft =< 0 then end(get_valuation)
else continue
endif;
```

where function swap takes two variables and returns a valuation in which the values of variables have been swapped. The built-in type valuation is defined in Zinc as a list of variable/value pairs. The built-in functions **minimizes** and **maximizes** take a list and a function and return the position of the element in the list that minimizes and maximizes the function, respectively. The move function chooses the most violated queen q1 and determines the queen q2 with which it can be swapped to reduce the overall violation. The number of moves left for the next iteration is decremented. After each move, the function finish is invoked which decides upon the state whether the search should finish. The enumerated type has ended is defined in Zinc's library as:

```
enum has_ended = {sol(valuation),end(valuation),continue};
```

to indicate if the search has found a solution, it has not but it must end, or should continue.

4. Evaluation

By evaluating our approach we intended to show two things: the expressiveness of our approach and the ignorable overhead of the required mappings. To investigate the first aim, we chose a set of 8 well known benchmarks and searched the literature for the best tree and local search strategy for each problem. The three search patterns in Zinc were expressive enough to implement the best search algorithms for all models.

For the first implementation, we map Zinc models into ECLiPSe programs (ECLiPSe was chosen because it supports all target solving techniques). Our results show that the models with user-defined search are faster than the equivalent models with default search, and that using the same search method, the mapped models have

approximately the same performance as hand-written models in ECLiPSe. This shows that our mapping does not lose efficiency.

We are currently working on adding new search patterns on Zinc and for the future we plan to enable the users to combine search paradigms for a given model.

Acknowledgements

We thank Arak University for financial support of this project.

References

1. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P.J., de la Banda, M.G., Wallace, The design of the Zinc modelling language. Constraints 13(3) (2008).

2. Apt, K.R., Wallace, M.G.: Constraint Logic programming using ECLiPSe. Cambridge University Press, Cambridge (2006)

3. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press, Cambridge (2005)