Adding Search to Zinc

Reza Rafeh[1], Kim Marriott[1], Maria Garcia de la Banda[1],
Nicholas Nethercote[2], and Mark Wallace[1]

[1] Clayton School of IT, Monash University, Australia
[2] NICTA Victoria Research Laboratory, University of Melbourne, Australia

Abstract. We describe a small, non-intrusive extension to the declarative modelling language Zinc that allows users to define model-specific search. This is achieved by providing a number of generic search patterns that take Zinc user-defined functions as parameters. We show the generality of the approach by using it to implement three very different kinds of search: backtracking search, branch-and-bound search, and local search. Our approach is competitive with hand-coded search strategies.

1 Introduction

Recent approaches to solving combinatorial problems divide the task into two steps: developing a *conceptual model* of the problem that gives a declarative specification without consideration as to how to actually solve it, and *solving* the problem by mapping the conceptual model into an executable program called the *design model*.

The declarative modelling language Zinc [7,5] is a first-order functional language designed to support experimentation with different solving techniques. In its current implementation, conceptual models in Zinc can be automatically mapped into design models that use one of the following three solving approaches: standard constraint programming (CP); a Mixed Integer Programming (MIP) solver; and incomplete search using local search methods.

While the default search used by the automatically mapped design models usually performs well for MIP, this is not the case for CP and local search whose efficiency often depends on the modeller providing an effective, model-specific search strategy. 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.

Here we describe an extension to Zinc to support model-specific search. The extension consists of three high-level search patterns for backtracking search, branch and bound, and local search, respectively, that take complex expressions, functions and predicates as parameters. This combined with user-defined functions give Zinc modellers a degree of flexibility to tailor the search only found previously in procedural search languages. While the actual mechanism of search must still be understood procedurally, the Zinc specification is declarative and requires no additional language features.

P.J. Stuckey (Ed.): CP 2008, LNCS 5202, pp. 624–629, 2008.

2 Using Search in Zinc

We illustrate the use of our search patterns with a simple example, the N-queens problem, which tries to place n queens on an $n \times n$ chess board in such a way that no two queens can take each other. A Zinc model for this problem is:

```
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;
```

The model defines the integer variable `n` to be a parameter, `Domain` to be a new type for the range `1..n`, and `q` to be 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 example, the modeller has defined the `noattack` predicate, which succeeds if queens `qi` and `qj` of rows `i` and `j` respectively, cannot attack each other (`/\` denotes conjunction). The constraint uses the `forall` expression to make sure the `noattack` predicate holds for each pair of queens. The last line declares the model to be a satisfaction problem. Since the `solve` item has no annotation for search, Zinc uses the default search to solve the model.

Backtracking search
Modellers can use Zinc's depth-first search pattern `backtrack(`*init*`,`*expand*`)` 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. Its 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. Note that *expand* has implicit access to the solver state and, thus, can call standard propagation solver reflection functions such as `domain(V)`, which returns the current domain of variable `V`.

As an example, 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.

Branch-and-bound
For optimization problems, Zinc provides a variant of the backtracking pattern extended with branch and bound: `backtrack(`$init$`,`$expand$`,`$bound$`,`$flag$`)`. 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 as before. 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 [1], 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).

Local search
A common class of techniques for solving combinatorial optimization problems are so-called *local search* methods (such as hill-climbing or simulated annealing) which iteratively improve a single valuation by moving to a neighbour. Zinc provides the 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.

These functions can use the following Zinc's local search solver reflection functions similar to those provided in Comet [8]: `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 Val 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 i^{th} 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.

It is worth pointing out that Zinc allows the modeller to override the default violation of constraints and variables by using annotations that can take complex expressions and functions. Also, the Zinc modeller does not have to explicitly set up invariants (or functional constraints). These are inferred automatically from the choice of driver variables based on the initial valuation and the model constraints. The compiler generates an error if some non-driver variable cannot be computed from the driver variables.

Evaluation

To evaluate the expressiveness of our approach, 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 (models can be found at [6]).

Our implementation maps 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 often orders of magnitude faster than the equivalent models using the default search, and that the mapped models are competitive with hand-written models in ECLiPSe that use the same search algorithm (on average, the overhead is less than 10%).

3 Discussion

Our extension to Zinc allows users to run the same conceptual model with different solving methods and, when the default search is too slow, to tailor it with user-defined search. This can be achieved by simply writing functions in Zinc to pass as the required parameters to one of the search templates: backtracking, branch and bound, and local search. The success of this pragmatic solution to an inherently difficult problem is only possible because: (1) the modelling language is powerful enough to allow the user to provide user-defined functions to tailor the search; and (2) a limited number of different generic search schemas covers most of the useful search routines.

Modelling languages for combinatorial problems have traditionally been declarative. Early languages such as AMPL [3] had search built into the solvers and provided only a few simple parameters for controlling it. This approach is too inflexible. The main alternative approach used, for example, in Mosel [2] and OPL [4], is to allow users to specify the search with the model. However, this requires the modelling language to be extended with non-declarative procedural constructs, something that is avoided in our approach.

The closest predecessor to Zinc's search appears to be the ECLiPSe search predicate [1] (which in turn was preceded by that of CHIP). While most search parameters in ECLiPSe are multiple-choice ones, some can be user-defined predicates. The key difference is that ECLiPSe is not fully declarative: modelling and search are both performed by procedural statements.

Acknowledgements. We thank members of the G12 team at National ICT Australia for helpful discussions, in particular Ralph Becket and Peter Stuckey.

References

1. Apt, K.R., Wallace, M.G.: Constraint Logic programming using ECLiPSe. Cambridge University Press, Cambridge (2006)
2. Colombani, Y., Heipcke, S.: Mosel: An overview (2007), `http://www.dashoptimization.com/home/downloads/pdf/mosel.pdf`
3. Fourer, R., Gay, D.M., Kernighan, B.W.: AMPL: A Modeling Language for Mathematical Programming. Duxbury Press (2002)
4. Van Hentenryck, P., Perron, L., Puget, J.F.: Search and strategies in OPL. ACM Transactions on Computational Logic 1(2), 285–320 (2000)

5. Marriott, K., Nethercote, N., Rafeh, R., Stuckey, P.J., de la Banda, M.G., Wallace, M.: The design of the Zinc modelling language. Constraints 13(3) (2008)
6. Rafeh, R.: The Zinc modelling language home page, `http://www.csse.monash.edu.au/~rezar/Zinc`
7. Rafeh, R., Garcia de la Banda, M., Marriott, K., Wallace, M.: From Zinc to design model. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 215–229. Springer, Heidelberg (2006)
8. Van Hentenryck, P., Michel, L.: Constraint-Based Local Search. MIT Press, Cambridge (2005)