I.J. Intelligent Systems and Applications, **2013, 10, 11-17**
Published Online September 2013 in MECS (http://www.mecs-press.org/)
DOI: 10.5815/ijisa.2013.10.02

Using Heuristic-based Search for Zinc Models

Reza Rafeh
Department of Computer Engineering, Faculty of Engineering, Arak University, Arak, Iran
E-mail:r-rafeh@araku.ac.ir

Roya Rashidi
Department of Computer Engineering, Islamic Azad University, Arak Branch, Arak, Iran
E-mail: e.r.rashidi@googlemail.com

Abstract— The Zinc modelling language provides a rich set of constraints, data structures and expressions to support high-level modelling. Zinc is the only modelling language that supports all solving techniques: constraint programming, mathematical methods, and local search. By providing search patterns, it allows users to implement their search methods in a declarative way. There are currently three search patterns implemented in Zinc: backtracking search, branch and bound search, and local search. In this paper we explain how Zinc efficiently implements user-defined local search algorithms.

Index Terms— Zinc, ZLoc, Modelling, Local Search

I. Introduction

Zinc [1] is an extensible mathematical modelling language which not only provides a wide set of constraints, data types and expressions, but also supports user-defined types, predicates and functions. A conceptual Zinc model can be automatically mapped into different design models (executable models) suitable for different solving techniques: Constraint Programming (CP), Mixed Integer Programming (MIP), and Local Search (LS). Zinc uses default search techniques for solving models; for CP it uses the first-fail heuristic , for MIP standard MIP branch and bound search [2], and for LS it uses a hill-climber with a tabu facility to prevent cycling on a plateau.

However, to solve Zinc models more efficiently, users need to implement their own search methods. Therefore, Zinc has been extended to support model-specific search algorithms by providing parametric search patterns. To date, three search patterns have been implemented in Zinc: backtracking search, branch and bound and local search [3]. Users must provide these search patterns the required parameters which are high-level data structures and user-defined functions. Our results show that using model-specific search algorithms gains an outstanding improvement on the execution time [1]. However, the local search solver used for Zinc was inefficient. It was due to implementing the solver in ECLiPSe [4,5] a logic programming language in which changing the value of a variable is unnatural and is inefficient. This is while local search algorithms need changing the value of variables repeatedly.

Here we address this issue by implementing a new local search solver for Zinc: ZLoc [6] a C++ library. Our experimental results show that ZLoc performs as well as Comet [7] in implementing local search routines [6]. We also show that ZLoc has greatly improved the quality of mapping user-defined local search techniques in Zinc into equivalent design models.

The rest of the paper is organized as follows. In Section 2, we introduce Zinc and its modeling capabilities. In Section 3, we briefly explain the search mechanism in Zinc. Sections 4, 5, and 6 detail the existing search patterns in Zinc. In Section 7, we introduce ZLoc as a local search solver for Zinc. In Section 8, we evaluate the performance of local search routines in Zinc. Finally, we conclude the paper in Section 9.

II. Zinc

A Zinc model includes the following components, all of which may be iteratively used except for the solve item which must be unique in the model.

- Variable Declaration and Assignment Items

Variables must be declared and possibly initialized. Declaration of each variable must specify its type and its instantiation. Zinc supports a variety of types: int, float, strings, Booleans arrays, sets, lists, tuples, records, discriminated union (i.e. variant records) and enumerated types [1]. Zinc also supports array with any arbitrary index sets over arbitrary data types and sets over arbitrary data types.

There are two kinds of instantiations for variables which divide them into two categories: parameters and decision variables. The default instantiation for variables assume them to be parameters. The var keyword is used to distinguish decision variables from

parameters and is added before decision variables definition. While decision variables are instantiated after the model being solved, parameters must be initialized either in the model or in a data file,

- Constraint Items

Constraints are Boolean expressions which may be satisfied or violated based on the value of their variables. Zinc also supports global constraints, such as alldifferent and sorted which are applied to lists of any type.

- Assertion Items

An assertion is a type of constraint imposed on parameters to check their values and prohibit modelling errors.

- Constrained Type Items

Zinc enables modellers to define a type associated with a constraint to make the model more concise and more readable.

- Function and Predicate Items

One of the most powerful features of Zinc is allowing users to define their own predicates and functions. Predicates are actually Boolean functions. Predicates and functions arguments are allowed to be polymorphic.

- Include Items

Libraries are allowed in Zinc models using include items. This item also allows a model to be split across multiple files.

- Solve Item

Every model must have exactly one solve item. The item is used for constraint satisfaction problems as solve satisfy and for optimization problems as solve minimize/maximize obj.

- Output Item

After solving the model, the value of variables can be printed out using the output item. If no output item is specified, the default output routine prints the name and value of each decision variable.

- Annotation Item

To specify the non-functional information in the model modellers can use annotations. Annotations are used to classify decision variables, constraints and objective function and define soft constraints in the model.

As an example, Figure 1 depicts a Zinc model for the n-queen problem which uses the min-conflict search. The first line defines a parameter n. In line 2, an integer range is declared. In line 3 an array of variables is declared in which a variable shows the column of a queen. The problem's constraints are defined in lines 4-6, which ensure no two queens can take each other.

```
1.  int: n;
2.  type Domain = 1..n;
3.  array[Domain] of var Domain :q;
4.  constraint alldifferent([Q[i]| i in 1..n]);
5.  constraint alldifferent([Q[i]+i| i in 1..n]);
6.  constraint alldifferent([Q[i]-i| i in 1..n]);
7.  solve satisfy;
```

Fig. 1: n-queen model in Zinc

III. Search in Zinc

If no search algorithm specified, the default search is applied for the model. However, Zinc modellers may specify their own search methods using search patterns and Zinc structures. Our extension to Zinc currently provides three generic search patterns: backtracking search, branch and bound, and local search.

IV. Backtracking Search

Zinc supports depth-first search using a propagation solver with backtracking for solving satisfaction problems. To do so, the search pattern backtrack (init, expand) has been added to Zinc. The first argument is the initial state or the root of the search tree which is usually the set of variables to label. The second argument is a user-defined function that takes the local state for the current node and returns its children as a list of pairs of the form (ns, c), where the ith pair gives the state ns for its ith child, and the constraint c that should be posted right before this child becomes the current node. Note that expand has implicit access to the solver state and can, therefore, make use of standard propagation solver reflection functions such as domain(V) which returns the current domain of variable V. For example, the function defined in Figure 2 implements the standard labeling.

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

Fig. 2: Standard labeling

The std_label labels a list of variables Vs in order and,

for each V in Vs, tries the domain values from smallest to largest. The head and tail functions are provided in the Zinc library to return the head and tail of the input list, respectively. The N-queens model can now be mapped to a design model that uses this labeling by simply adding an annotation to the solve statement in the model:

solve satisfy:backtrack(queens, std_label);

To call the search pattern with queens as the initial local state and std label as the expand function. Note that in Zinc, lists are syntactic sugar for arrays.

To improve the performance of the model, we can use a first-fail search that labels from the middle of each queen's domain. To do this, the search routing shown in Figure 3 must be added to the model. The search routine obtains in middle out a list of domain values ordered from the middle out, and uses it to return as children a list of tuples assigning those values to the variable V in Vs with the minimum domain size (thanks to the reflection function domain_size and Zinc's library function minimizes).

```
list of Domain: middle_out =
[if (i mod 2==0) then (n-i+1)
div 2 else (n+i) div 2 + 1 endif | i in 0..n-1]
function list of tuple(list of var Domain, var bool):
first_fail_middle_out(list of var Domain:Vs) =
if Vs=[] then []
else let { int: min = minimizes(Vs,domain_size),
var Domain: V = Vs[min],
list of var Domain:
VRest = [ Vs[j] | j in Domain where j!= min]) }
in
[(VRest, V=d) | d in middle_out]
endif;
solve
satisfy::backtrack(queens,first_fail_middle_out);
```

Fig. 3: A backtracking search for n-queen

The backtracking search pattern is surprisingly powerful thanks to the modeller being able to choose the init state and the kind of constraints to be posted by each child node. Thus, for instance, modellers can use a counter to implement iterative deepening.

V. Branch-and-Bound

For optimization problems, Zinc provides a variant of the backtracking search pattern extended with branch and bound: backtrack(init,expand,bound,flags).

The first two parameters are the same as the previous pattern. The two extra parameters are a function bound for computing the new bound from the old 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 [4], 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).

To show the usage of this pattern we write a model-specific labeling routine for 0-1 knapsack, which tries to find a subset of an initial set of items with maximum total profit such that the sum of their weights does not exceed the capacity of the knapsack. The proposed search routine tries to place items in order of their expected utility profit/weight. The model and the search routine are depicted in Figure 4.

```
1.  type item = record(int:id,profit,weight);
2.  set of item: All_Items;
3.  int: Max_Capacity;
4.  var set of All_Items: Selected_Items;
5.  constraint
6.    sum(X in Selected_Items) (X.weight) =<
Max_Capacity;
7.  function list of tuple(list of tuple(var set of
$T,$T),var bool):
8.    set_labeling(list of tuple(var set of
$T,$T):L)=
9.    if L=[] then []
10. else let{$T:P=head(L), list of $T:Rest=tail(L)}
in
11. [(Rest, P.2 in P.1),(Rest, not (P.2 in P.1))]
12. endif;
13. function int:bound_func(int:old, int:curr) =
curr+1;
14. list of item: Sorted_Items=
15. [X.2 | X in sort([(-X.profit/X.weight,X)|X in
All_Items])];
16. solve maximize sum(S in Selected_Items)
(S.profit)::
17. backtrack([(Selected_Items,j)|j in
18. Sorted_Items],set_labeling,bound_func,restart
);
```

Fig. 4: A search method for knapsack

In line 1 item is defined as a new type to store the item's information: three integers that indicate its identifier, profit and weight. The model then defines two parameters: All_items as the initial set of items, and Max_Capacity as the maximum capacity of the knapsack. Then it defines Selected_Items as a set variable which will provide the solution of the problem. Lines 5 and 6 define a constraint indicating that the sum of the item weights cannot exceed the knapsack capacity. And finally, line 16 declares the model to be an optimization problem with an associated objective function to be maximized.

The set_labeling function plays the role of the expand function by taking a list of pairs (each with a set variable and a possible item) and creating two child nodes, the first stating that the item is a member of the list, and the second stating it is not. The remainder of

the list is returned as the state of the created nodes. In this example, bound function forces the search to find a better solution than the most recent one (at least one unit more). Every time a better solution is found, we restart the search from the root of the search tree.

Alternatively, we could use binary search by simply using the dichotomic flag and the following bound function:

function float:

bound_func2(float:old,curr)=

*min(old,curr)+abs(old-curr)*0.5;*

VI. Local Search

To support local search, Zinc provides the following pattern: local_search(init_valn, init_state, move, finish) where init_valn is the initial valuation (a list of variable/value pairs), init_state is the initial state, move is a function which takes a state and returns a new valuation to move to and finish is a function which determines when the search should finish. In each iteration of the algorithm, the move function is called to take the current state and return a new state, then the finish function is called to decide whether the search should continue or not [3].

We show the usage of the pattern by means of an example. Figure 5 depicts a local search algorithm for the n-queen model. The swap function is a polymorphic function which takes two variables and returns a valuation in which the value of variables is exchanged (line 1). In the move function the variable qi with maximum conflict is swapped with variable qj to minimize the total conflict (line 4). Then, the number of remaining steps is decreased.

```
1.  function valuation: swap($T: v1, $T: v2) =
[(v1,val(v2)),(v2,val(v1))];
2.  function tuple(int, valuation): move(int:
nmovesleft) =
3.  let {int: i=maximizes(q,var_penalty),
4.  int: j=minimizes([swap(q[i],q[k])|k in
Domain], new_penalty)
5.      } in
6.  (nmovesleft-1,swap(q[i],q[j]));
7.  function has_ended: finish(int: nmovesleft) =
8.  if current_penalty == 0 then sol(get_valuation)
9.  elseif nmovesleft =< 0 then end(get_valuation)
10. else continue
11. endif;
12. solve satisfy::local_search( [(q[i],i)|i in
Domain], 1000,move,finish);
```

Fig. 5: A local search method for n-queen

The finish function takes a state and checks the violation degree. A zero value means that the problem is solved and search terminates (line14). If the number of remaining steps is zero the search stops (line 9), otherwise, search continues (line 10).

The first argument of local_ search initializes the array q to place the i'th queen in column i. The second argument defines the number of allowed steps as 1000. The two next arguments are move and finish function. The move function is repeatedly called until the search terminates (line 18).

VII. Zloc

ZLoc is a C++ library for modelling constraint optimization problems and solving them using local search methods. ZLoc supports mathematical structures and expressions, multi dimension arrays with arbitrary index set over variety of data types, lists, sets, and constraints over variety of data types and global constraints such as alldifferent [6].

ZLoc utilizes Zinc with a fast local search solver. The first local search solver of Zinc was implemented in ECLiPSe and was inefficient. It was because ECLiPSe is a logic language in which changing the value of variables is unnatural, something that is essential for implementing local search methods.

ZLoc supports Zinc data types, user-defined functions, predicates, expressions and constraints. In addition, it provides necessary operations for guiding local search.

ZLoc is similar to Comet in many aspects. Nonetheless, Comet has no support for some features of Zinc models like user-defined functions and predicates which are the key features in implementing user-defined search methods [3].

Each ZLoc model consists of two sections: declaration and search. In the declaration section, the problem is modelled by defining expressions and constraints. In the search section the problem is solved using a local search method. ZLoc users can use C++ structures [8] in addition those existed in ZLoc to implement the search algorithm. Each ZLoc model includes the following components:

- Data types: ZLoc supports lists, arrays and sets over arbitrary data types. While in Zinc array index set is not necessarily integer, in ZLoc index set of an array can be an integer range or a set of integers.
- Variables: Based on instantiation, Zinc variables are classified as parameters and decision variables. A parameter is initialized before solving the model while the value of a decision is determined after solving. ZLoc decision variables may be integers, floats and sets.
- Expressions and operations: ZLoc supports variety of mathematic operations, logic expressions, operations over sets such as union, intersection and membership.

comprehened acts as a loop to initialize lists, arrays and sets. Other iteration means include forall, sum, max, prod and min.

- Constraints: ZLoc supports variety of constraints over integers, floats, sets. Global constraints like alldifferent are supported as well. Similar to Comet, ZLoc provides necessary functions to guide local search such as get_violation (v) to calculate the violation degree associated with variable v, get_assign_delta(x, v) to calculate the changing violation degree by assigning value v to variable x, get_assign_delta ([x1, x2,..], [v1, v2,..]) to calculate the changing violation degree by assigning value vi to variable xi, get_swap_delta (a, b) to calculate the new violation degree after swapping variables a and b.

For each constraint the violation degree is computed accordingly. For instance, the violation degree of arithmetic constraint l>=r is max (0, r-l) [7]. In ZLoc, when the value of a variable is changed, the violation degree of all constraints in which the variable appears, is automatically updated.

ZLoc uses the pattern local_search<T> (T1 init, T2 move, T3 finish) for implementing local search algorithms. local_search is a template function whose type is determined by the output of init function which equals the input type of both move and finish functions. The variables of the problem are initialized in the init function. Similar to Zinc, the move function is repeatedly called until the finish function terminates the search.

For example, consider the model of n-queen problem in Figure 6. The array q is declared in the first line. The constraints of the problem are declared in lines 3-5 that check that queens do not threat each other vertically or diagonally.

```
1.  Array<varInt> q(1,n);
2.  elemParameter<int> i;
3.  add_cons(all_different(i,1,n,q[i]));
4.  add_cons(all_different(i,1,n,q[i]+i));
5.  add_cons(all_different(i,1,n,q[i]-i));
6.  template <class type1>
7.  List<tuple<type1*,type1> > swap1(type1& var1,type1& var2)
8.  {
9.     List<tuple<type1*,type1> > l;
10.    l=make_list<tuple<type1*,type1> (make_tuple(&var1,var2))
11.  (make_tuple(&var2,var1));
12.     return l;
13.  }
14.  int move(int counter)
15.  {
16.     int i=maximize(q,get_violation<varInt>);
17.  List<List<tuple<varInt*,varInt>>> l;
18.  for(int k=1;k<=domain(q).length(); k++)
19.     l.insert(swap1(q[i],q[k]));
20.     int j=minimize( l,get_assign_delta<varInt,varInt>);
21.     swap(q[i],q[j]);
22.     return counter-1;
23.  }
24.  has_ended finish(int counter)
25.  {
26.     if(counter<=0)
27.        return _end;
28.     if(get_violations()==0)
29.           return _sol;
30.     return _continue;
31.  }
32.  local_search<int>(init1,move,finish);
```

Fig. 6: A ZLoc model for n-queens

Function swap1 takes two variables var1 and var2 and returns the list [(*var1, var2), (*var2, var1)] in which the value of variables is swapped (lines 7-13). The make_list function is implemented in ZLoc to make a list. It is a template function and its type is determined by the elements of the list. The make_tuple function from boost Library [6] is used to make a tuple.

Function init initializes the array q and also the maximum number of allowed moves (lines 14-23). Its output is the input of the move function. Similar to Figure 1, queen qi with maximum conflict is selected in the move function (line 16). The maximize function like in Zinc takes a list (or an array) and a function and for every element in the list, the function is called and position of the element that maximize the function is returned. The get_violation function is a template function that gets a variable and returns its violation degree. In this example, the input type of the function is varInt because its input is an element of array q. In lines 18-20 the index of variable which its swap with the variable with maximum violation minimizes the total violation is determined. The get_assign_delta function takes a list of pairs variable/value and returns the difference of violation degree after assigning new values to variables. In line 21 the queens qi and qj are swapped with each other. In line 22, the number of remaining steps is decreased.

The finish function takes the number remaining steps and checks whether the search should continue or not. Similar to Zinc, the returned value of this function is of type has-ended which is an enumerated type. If _continue is returned then the function move is called again otherwise the search is terminated.

VIII. Evaluation

To evaluate the efficiency of ZLoc, we used 4 well-known problems as benchmark and compare the execution time and the quality of final solution with the previous local search solver of Zinc and with Comet.

We used the same search technique for each problem. For the knapsack problem, we select the most beneficial items that meet the capacity requirement. At each step, if there is a violation of the capacity constraint, we select an item to remove with a probability inversely proportional to its benefit. If there is no violation, we compare the total profit of the items currently in the knapsack with the best profit found so far and, if necessary, we update the best profit. Also, if there is some space left, we select an item to put in the knapsack with a probability proportional to its benefit value. For n-queen, we used the min-conflict technique. For open stacks, the search routine uses a tabu facility and starts with a random permutation of products. Then, it swaps two products in this sequence to decrease the maximum number of open stacks. For perfect squares, simulated annealing is used.

The experimental results are depicted in Table 1. All experiments were performed on a 3.0 GHz Pentium 4 with 1Gb memory running Windows XP. The timings are the average of 20 executions.

Table 1: Comparing ZLoc with the current solver of Zinc and Comet

	Zinc			Comet			ZLoc		
Problem	**Execution time(s)**	**Percentage of possible answers(%)**	**Best solution**	**Execution time(s)**	**percentage of possible answers(%)**	**Best solution**	**Execution time(s)**	**percentage of possible answers(%)**	**Best solution**
Knapsack (34Items)	1.4905	100	139	0.8586	100	143	0.2938	100	139
n-queen (128 queens)	159.871	100	-	63.9523	100	-	0.4088	100	-
OpenStack (15customer, 15product)	3.4045	100	9	7.8249	100	10	2.176	100	9
Perfect Square (7*7)	52.2885	74	-	0.1967	8	-	0.1782	77	-

As can be seen from Table 1, ZLoc outperforms both Comet and the current local search solver of Zinc. This is true for both execution times and the quality of solution. The only exception is Knapsack for which Comet finds a bit better solution (with 3% rise in quality).

IX. Conclusion

We introduced ZLoc, a new local search solver for the Zinc modelling language implemented as a C++ library. ZLoc supports modeling features of Zinc as well as user-defined local search algorithms. Our results showed that ZLoc is more efficient than Comet and the previous local search solver in Zinc. The main goal of Zinc has been allowing the modellers to employ all solving techniques for their models automatically and see which technique gives them the best result. Deficiency of the previous local search solver of Zinc jeopardized this goal since the models mapped to local search techniques were not competitive with tree search techniques.

Now this problem has been fully addressed and we are ready to complete mapping Zinc models to existing solving techniques. In addition, we are currently working on new search patterns for Zinc.

References

[1] Rafeh R., "The Modelling Language Zinc," in Clayton School of IT. vol. Ph.d.Thesis: Monash University, 2008.

[2] N. Jaberi, and R. Rafeh, "A Survey of Linearization Techniques for Nonlinear Models," International Journal of Computational Intelligence and Information Security, vol. 3, no. 2, 2012.

[3] Rafeh R., Marriott K., and de la Banda M., "Adding Search to Zinc," CP 2008, vol. LNCS 5202, pp. 624-629, 2008.

[4] Apt K., and Wallace M., Constraint Logic programming Using ECLiPSe: Cambridge University Press, 2007.

[5] R. Rafeh,. "Proposing a new search template for modelling languages", Procedia CS **3**: 1490-1493, 2011.

[6] Rashidi R., Rafeh R., Rahmani M., and Khadem E. A., "ZLoc: A C++ library for local search," *International Journal of the Physical Sciences,* vol. 6(31), pp. 7095 - 7099, 2011.

[7] Hentenryck P. V., and Michel L., *Constraint-Based Local Search*: MIT Press, 2005.

[8] "Boost C++ libraries home page."

Authors' Profiles

Reza Rafeh is a faculty member at Arak University (Iran). He got his PhD from Monash University (Australia). His interesting areas are compiler design, constraint programming and theorem proving. He is the chief editor of Software Engineering Journal as well as an editorial board of Soft Computing Journal.

Roya Rashidi got her master from Islamic Azad University, Arak branch (Iran). She is interested in local search algorithms and constraint programming.

