International Journal of the Physical Sciences Vol. 6(31), pp. 7095 - 7099, 30 November, 2011
Available online at http://www.academicjournals.org/IJPS
DOI: 10.5897/IJPS11.1499
ISSN 1992 - 1950 ©2011 Academic Journals

Review

ZLoc: A C++ library for local search

Roya Rashidi[1], Reza Rafeh[2]*, Mohsen Rahmani[2] and Ehsan Azizi Khadem[3]

[1]Department of Computer Engineering, Arak Branch, Islamic Azad University, Arak, Iran.
[2]Department of Computer Engineering, Malayer Branch, Islamic Azad University, Malayer, Iran.
[3]Department of Computer Engineering, Lorestan University, Iran.

Accepted 20 October, 2011

Local search is known as an effective technique for solving combinatorial optimization problems. However, there are few tools that provide high level facilities for users to implement their own local search algorithms. In this paper, we introduce ZLoc, a new C++ library for local search. ZLoc supports many high-level features usually found in modeling languages, such as Zinc. It allows users to define their models in terms of variables and constraints, then it specifies their favorite local search method for solving the model.

Key words: ZLoc, C++, search, algorithm.

INTRODUCTION

Combinatorial optimization problems appear in many academic and practical applications, such as planning, timetabling, routing and DNA sequencing. Often such problems are NP-Hard, that is, there is no general and efficient algorithm for solving them. This is because the search space grows exponentially with the size of the problem (Marriott and Stuckey, 1998).

The main steps to tackle combinatorial optimization problems are modeling and solving. The solving step uses three main techniques: mathematical methods, local search and constraint programming (Banda et al., 2006).

Mathematical methods are rooted in operation research and are believed to be efficient and effective. Examples include methods such as simplex (Vanderbei, 2008) and interior point (Roos et al., 2006). However, these techniques usually require the modelers to provide a linear formulation for problem which may be difficult for some problems. Constraint programming techniques are more flexible than mathematical methods. These techniques prune the search space to reduce the execution time. However, for some real problems, the execution time may be considerably high (Rafeh, 2008a). Local search techniques are very efficient and effective for solving those problems for which a good heuristic is known. These techniques avoid exploring the search space completely, as a result they do not guarantee to find the solution and even if the solution is achieved, it is unclear how far that solution diverges from the optimal one. But if local search is guided properly, often high-quality solution is achieved faster than the aforementioned algorithms (Beck et al., 2011).

Modeling tools that are most relevant to our work include constraint programming languages, constraint programming libraries and (mathematical) modeling languages. Constraint programming languages, such as Comet (Hentenryck and Michel, 2005) and Eclipse (Aggoun et al., 2008) are generic programming languages and support all solving techniques. However, they lack support of high level modeling. As a result, the user must have sophisticated programming skills (Aggoun et al., 2008; Hentenryck et al., 2005; Rafeh, 2008b). Constraint programming libraries, such as Localizer++ (Michel and Hentenryck, 2001) are built in other programming languages (often object-oriented). Similar to constraint programming languages, they do not support high-level modeling and furthermore, some restrictions may be imposed by the host language. On the positive side, for using such libraries, there is no need to learn a new language. Mathematical modeling languages have simple syntax and are close to mathematic expressions. They support high level modeling which makes these languages accessible for non-programmers. The main issue with current modeling languages is their

*Corresponding author. E-mail: r-rafeh@araku.ac.ir. Tel: +98 861 2625005. Fax: +98 861 2625003.

support of just one solving technique. For example, AMPL (Fourer, 2002) supports MIP, OPL (Hentenryck et al., 1999) supports CP techniques and Localizer (Michel and Hentenryck, 1997) supports local search algorithms. Zinc is the only high-level modeling language that supports all the solving techniques (Rafeh, 2008a).

In this paper, we present ZLoc which is a new C++ library to implement local search algorithms. ZLoc supports many high level structures that exist in mathematical modeling languages, especially Zinc.

The rest of the paper is organized as follows: brief reviews of the most relevant tools to ZLoc, introduction to ZLoc and its features, an example in ZLoc, comparison of ZLoc with Comet and finally conclusion.

RELATED WORK

Recently, modeling tools for local search have received significant attention. Local search algorithms for optimization problems were first developed in operations research. One barrier in developing a modeling tool for local search algorithms is the strong connection between models and search strategies, which makes it difficult to find a framework that provides a separate formulation for modeling and search.

In 1997, Localizer was developed as the first modeling language for local search. Localizer introduced the concept of invariants to specify incremental algorithms declaratively. An invariant (which is also called a one-way constraint) is an equation of the form Y = f (X1; X2; :::; Xn) which relates variable Y to variables X1; X2; :::; Xn (we say variable Y is functionally dependent on variables X1; X2; :::; Xn). Any changes of the value of any variable Xi (which is also called driver variable) causes the value of Y (which is also called dependent variable) to be updated (but not vice versa).

Later in 2001, Localizer++ was developed as a C++ library for local search. Localizer++ followed Localizer's idea of invariants, but used this concept for a larger class of constraints. In 2001, the Comet project was started. Comet also provides invariants. It provides a set of constraints with default definition of violation degree, but allows users to define new classes of constraints with their own meaning of violation.

Comet is an object-oriented constraint programming language which has been written in Java and supports local search (it has recently extended to support propagation techniques as well). It allows the specification of both conceptual and design models.

ZLOC

ZLoc is a new C++ library which allows users to model their problems and solve them using local search algorithms. ZLoc supports mathematic structures and expressions, lists, sets, multi dimension arrays with arbitrary index set over variety of data types and constraints over variety of data types and global constraints, such as *alldifferent*.

Our motivation to design ZLoc was to utilize Zinc with a fast local search solver. The current local search solver of Zinc is implemented in eclipse which is a logic language.

In logic languages, changing the value of variables is unnatural, something that is essential for implementing local search algorithms. This limitation precludes solving Zinc models by local search algorithms efficiently. 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 features. Nonetheless, Comet lacks some features found in Zinc models like user-defined functions and predicates which are the key features in implementing user-defined search algorithms (Rafeh, 2011).

Each ZLoc model consists of two sections: declaration and search. In declaration section, the problem is modeled by defining expressions and constraints. In search section the problem is solved using a local search technique. To implement the search algorithm in addition to built-in structures in ZLoc, the user can also use C++ structures.

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 (Rafeh, 2008a). ZLoc decision variables may be integers, floats and sets.
- Expressions and operations: ZLoc supports variety of mathematic operations, logic expressions and 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 and 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 ([x_1, x2,..], [v_1, v2,..])* to calculate the changing violation degree by assigning value v_i to variable x_i and *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

```
1.  const int n=6;
2.  Range Size(1,n);
3.  int T=(n*(pow(n,2)+1))/2;
4.  Array<Array<varInt> > magic(1,pow(n,2));
5.  Array<Array<int> > tabu;
6.  elemParameter<int> i,k;
7.  add_cons(forall(k,Size,sum(i,Size,magic[k][i])==T));
8.  add_cons(forall(k,Size,sum(i,Size,magic[i][k])==T));
9.  add_cons(sum(i,Size,magic[i][i])==T);
10. add_cons(sum(i,Size,magic[i][n-i+1])==T);
11. add_cons(sum(i,Size,magic[i][n-i+1])==T);
```

Figure 1. The magic square model.

arithmetic constraint l>=r is max (0, r-l) (Hentenryck et al., 2005). In ZLoc, when the value of a variable is changed, the violation degree of all constraints in which the variable appears, is automatically updated.

Example

Here, we explain ZLoc by means of an example, the magic square problem. The problem consists of arranging numbers 1..n^2 in an n × n matrix such that the sum of every row, column and main diagonal are equals, that is, n × (n^2 + 1)/2. The model is depicted in Figure 1.

In the first line we declare the size of square. A range of numbers 1 to n is defined in line 2. In line 3, we initialize *T* as the sum of rows, columns and diagonals. In line 4, a two-dimensional array is declared. For every i,j, *magic[i][j]* shows the value of magic square in row i and column j. A two dimensional array *tabu* in line 5 is defined to keep 'tabu' positions. In line 6, variables i,j are defined to be used in loops. Constraints in lines 7 to 11 ensure that the sum of rows, columns and diagonals is n × (n^2+1)/2.

The search part of the model is depicted in Figure 2. Function *move* selects a neighbor using 'tabu' search technique. The neighbor is obtained by swapping *magic[i][j]* and *magic[k][l]* so that the violation degree will be the most reduced. In lines 2 to 4, necessary lists are defined. Lines 5 to 9 store pair(i,j) in list *pairIndex_with_violation* if it is not 'tabu' and the violation degree of *magic[i][j]* is greater than zero. By using function *get_swap_delta* in lines 10 to 18, after swapping *magic[i][j]* and *maigic[k][l]* the new violation degree is calculated. Note that neither (i,j) nor (k,l) must be 'tabu'. Then, pairs (i,j) and (k,l) with the new violation are stored in list. Line 19 sorts *viol_swap_2cel* ascending and selects its first element. This element involves pairs (i,j) and (k,l) as well as the minimum violation degree. We swap *magic[i][j]* and *magic[k][l]* by using function *swap* in line 23. In lines 24 and 25 both pairs are stored in 'tabu' list and remain there until *it+tabuLength* iteration. In line 26, *it* is incremented as the iteration controller.

EXPERIMENTAL RESULTS

Since ZLoc shares many features with Comet, we compared them using a bunch of well-known benchmarks. Table 1 shows the execution time and possibility of finding feasible solutions. The execution time shown in the table is the average of 20 run for each model. The local search technique used for solving the model is mentioned in the table. Models have been run on an Intel Pentium IV CPU 3.0 GHz, 1.00 GB of RAM and Microsoft Windows XP Operation System.

As shown in Table 1, ZLoc works better for queens (dis-equality constraints), knapsack and open stacks. Although, for perfect squares Comet is better with respect to the execution time, Zloc finds feasible solution by a better chance. The main reason for Comet to be faster in solving queens (alldifferent), magic squares and perfect squares is that aggregated functions like *sum* and global constraints like *alldifferent* have been implemented efficiently. We are currently making such functions more efficient in ZLoc. Since Zinc is a high level modeling language, there is an overhead in mapping Zinc models to design models. The execution times shown in the table includes the mapping overhead. The mapping process

```
1.  int move(int it){
2.    List<tuple<int,int> > pairIndex_with_violation;
3.    List<tuple<int,int,int,int,int>>
4.          viol_swap_2cel,viol_swap_2cel2;
5.    for(int i=1;i<=n;i++)
6.      for(int j=1;j<=n;j++)
7.        if(tabu[i][j]<=it)
8.          if(get_violation(magic[i][j])>0)
9.            pairIndex_with_violation.insert(make_tuple(I,j));}
10.  for(int z=1;z<=pairIndex_with_violation.length();z++){
11.    tuple<int,int> cel1=pairIndex_with_violation[z];
12.      int i=get<0>(cel1);
13.      int j=get<1>(cel1);
14.    for(k=1;k<=n;k++){
15.      for(int l=1;l<=n;l++){
16.        if((I != k || j != l) && tabu[k][l]<=it){
17.          int viol=get_swap_delta(magic[i][j],magic[k][l]);
18.  viol_swap_2cel.insert(make_tuple(viol,I,j,k,l));}}}
19.  viol_swap_2cel2=sort2(viol_swap_2cel);
20.  tuple<int,int,int,int,int> selected_pair=viol_swap_2cel2[1];
21.  i=get<1>(selected_pair);   j=get<2>(selected_pair);
22.  k=get<3>(selected_pair);   l=get<4>(selected_pair);
23.  swap(magic[i][j],magic[k][l]);
24.  tabu[i][j]=it+tabuLength;
25.  tabu[k][l]=it+tabuLength;
26.  it++;
27.  return it;}
```

Figure 2. Selecting a neighbor by using tabu search technique in magic square problem.

Table 1. Experimental results.

Problem	Technique	Comet		ZLoc	
		Execution time (s)	Percentage of possible answers (%)	Execution time (s)	Percentage of possible answers (%)
n-queen (512 queens with alldifferent constraint)	Min conflict	0.3695	100	8.971	100
n-queen (128 queens with dis-equality constraints)	Min conflict	63.9523	100	35.5225	100
Knapsack (34 items)	Greedy	0.8586	100	0.075	77
Magic square (6 × 6)	Tabu search	0.381	100	2.7126	100
OpenStack (15 customers,15 products)	Greedy	7.8249	100	4.7086	100
Perfect square (7 × 7)	Simulated annealing	0.1967	8	0.3538	28

with and its overhead has been discussed in details (Rafeh, 2008a).

CONCLUSIONS AND FUTURE WORK

We introduced ZLoc, a C++ library for local search. ZLoc supports high-level structures usually found in modeling languages like Zinc. Our experimental results show that ZLoc is competitive with Comet which is known as an efficient constraint programming language for local search.

In future, we plan to make this library as efficient as possible and link it to Zinc to allow Zinc modelers try local search techniques for solving their models in addition to other solving techniques supported with Zinc.

REFERENCES

Aggoun A, Lada, Chan D (2008). Eclipse User Manual" Book Eclipse User Manual Series Eclipse User Manual ed., Editor ed.^eds. 245 p.

Banda DLMG, Marriott K, Rafeh R, Wallace M (2006). The Modelling Language Zinc," CP 2006, vol. LNCS 4204, pp. 700-705.

Beck JC, Feng TK, Watson JP (2011). Combining Constraint Programming and Local Search for Job-Shop Scheduling," INFORMS J. Comput., 23(1): 1-14.

Fourer R, Gay DM, Kernighan BW (2002). AMPL: A Modeling Language for Mathematical Programming, Duxbury Press, 540 p.

Hentenryck PV, Lustig I, Michel LA, Puget JF (1999). The OPL Optimization Programming Language, MIT Press, 255pp.

Hentenryck PV, Michel L, LIU L (2005). Constraint-Based Combinators for Local Search," Springer, 10: 363-384.

Marriott K, Stuckey PJ (1998). Programming with Constraints: An Introduction, the MIT Press, 483pp.

Michel L, Hentenryck PV (1997). Localizer: A Modeling Language for Local Search," Principles and Practice of Constraint Programming, CP97, pp. 237-251.

Michel L, Hentenryck PV (2001). Localizer++: An Open Library for Local Search," Technical Report CS-01-02, Brown University, 35 p.

Rafeh R (2008a). The Modelling Language Zinc. Clayton School of Information Technology. Melbourne, Monash. PhD: 201.

Rafeh R (2008b). The Modelling Language Zinc. Monash University. PhD. Thesis.

Rafeh R (2011). Proposing a new search template for modelling languages," Procedia CS, 3: 1490-1493.

Roos C. Terlaky T, Vial JP (2006). Interior point methods for linear optimization, Birkhäuser, 497p.

Vanderbei RJ (2008). Linear programming: foundations and extensions, Springer, 467p.