A Java library for Constraint-Based Local Search: Application to the master thesis defense timetabling problem

PHAM Quang Dung
Hanoi University of Science and technology
1, Dai Co Viet Road, Hanoi, Vietnam
dungpq@soict.hust.edu.vn

HUYNH Thanh trung
Hanoi University of Science and technology
1, Dai Co Viet Road, Hanoi, Vietnam
thanhtrunghuynh93@gmail.com

TA Duy Hoang
Hanoi University of Science and technology
1, Dai Co Viet Road, Hanoi, Vietnam
hoangduybk56@gmail.com

NGUYEN Thanh Hoang
Hanoi University of Science and technology
1, Dai Co Viet Road, Hanoi, Vietnam
thnbk56@gmail.com

ABSTRACT

Constraint-Based Local Search (CBLS) is an architecture for local search that uses constraints and objectives to control the local search. CBLS has many advantages in designing and implementing local search programs such as compositionality, modularity and reusability. We implement in this paper a Java library for CBLS. The implemented library will then be applied to the resolution of a combinatorial optimization problem arising in the education management in most of Vietnamese universities: the master thesis defense timetabling problem. Experimental results show the modelling flexibility and the efficiency of the constructed library. Our constructed library will be released as open source.

CCS Concepts

•**Computing methodologies** → **Artificial intelligence;** *Modeling and Simulation;*

Keywords

Combinatorial Optimization; Constraint-Based Local Search; Constraint Programming; Master Thesis Defense Timetabling

1. INTRODUCTION

Combinatorial optimization problems arises in many real-life applications such as bin packing [**?**], rostering [**?**], timetabling [**?**], sequencing [**?**], etc. In these problems, the goal is to find a solution satisfying a given set of constraints while optimizing an objective function. Constraint Progrmaming (CP) [**?**] is a framework for solving combinatorial optimization problem. It is a combination of a propagation that uses constraints for pruning the solution space and a backtrack search for exploring the solution space. CP is a complete approach that ensures to find an optimal solution and prove its optimality if exists, but the time complexity is exponential for complex problems. For optimization problems, as CP is using a branch-and-bound approach, one can stop the execution of the CP solver if the specified time limit is expired.

Local Search (LS) is an alternative approach for solving combinatorial optimization problems that explores partial solution space. It starts from an initial solution and iteratively moves from a current solution to one of its neighbors. LS is an incomplete approach. It cannot ensure to find optimal solutions. However, in many cases, it can find high-quality solutions in reasonable time.

Constraint-based Local Search (CBLS), proposed by Van Hentenryck and Michel in 2005 [**?**], is an architecture for local search that uses constraints and objectives to control the search. CBLS has many advantages in designing and implementing of local search programs: compositionality, modularity and reusability. Comet, also invented by Van Hentenryck and Michel [**?**], is a rich programming language that features the CBLS architecture. Comet has been widely used for a while. Unfortunately, Comet is no longer supported. In this paper, we construct a library framework for CBLS. The library is built in Java [**?**] programming language which is one of the most popular programming language for industrial applications. The constructed library will then be applied to the resolution of a real-world problem in education management in most of vietnamese universities: the master thesis defense timetabling (MTDT) problem. We show the flexibility and extensibility of our library. The experiments show the efficiency of the library in solving MTDT problem. Our constructed library will be released as open source.

The paper is organized as follows. Section **??** presents the API of the constructed library including interfaces, fun-

SoICT 2015, December 03-04, 2015, Hue City, Viet Nam

DOI: http://dx.doi.org/10.1145/2833258.2833300

damental classes and illustrating examples. Section **??** describes the formulation of the MTDT problem, and presents the model and search for solving the problem using the constructed library. The experimental results will be presented in Section **??**. Section **??** concludes the paper and gives further works.

2. API OF THE LIBRARY

A CBLS program consists of two main components: the model and the search. The model includes decision variables, constraints, functions describing the problem to be solved. The search uses the constraints, functions of the model to drive the search toward high-quality solutions.

2.1 Modelling

Decision variables.

Decision variables model the problem. An assignment of values to decision variables gives a solution. A neighboring solution is generated from a current solution by changing the value of one decision variable, or swapping the values of two variables. Over decision variables, functions and constraints are defined representing the problem structure. In our framework, a variable is initialized as follows:

```
VarIntLS x = new VarIntLS(ls, min, max)
```

in which `ls` is a `LocalSearchManager` object which manages all objects (variables, functions, constraints) of the CBLS program and maintains relating information among them.

Invariants and differentiable objects.

Invariants and differentiable objects are key concepts of CBLS [**?**]. Invariants maintain a number of property of the problem. Invariants are defined over decision variables. A change of a decision variable[1] induces a propagation that updates the invariants defined on it thanks to a dependency graph. Differentiable objects, i.e., functions, constraints, are invariants and feature differentiation in the sense that they allow users to query the variation of properties maintained under the change of variables. Functions maintain some properties describing the problem structure, for instance, the sum of values of all decision variables. Functions are objectives to be optimized or are used to state constraints of the problem. A constraint maintains a number of violations of the constraint, and allows users to query the variation of the number of violations when a decision variable is changed. In a CBLS program, users only need to specify invariants, functions, and constraints. The engine of the framework will maintain and update these objects during the search. Users do not have to explicitly implement procedures to update these invariants, functions, constraints. Figure **??** illustrates the dependency graph. When the decision variable X_3 is changed, the propagation engine will update F_2, and then C_1, C_2, C_3, and then C_4. The remaining functions F_1, F_3, and constraint C_5 which are not defined over X_3 are kept unchanged. In our constructed library, all invariants, functions, and constraints implement the common interface `Invariant`, `IFunction`, and `IConstraint`. These predefined interfaces enables the extensibility of the library framework. Users can build their own invariants, functions, or constraints implementing the corresponding interfaces, and integrate them into the system.

[1]During the local search, one or some decision variables will be changed in order to move from a current solution to one of its neighbors.

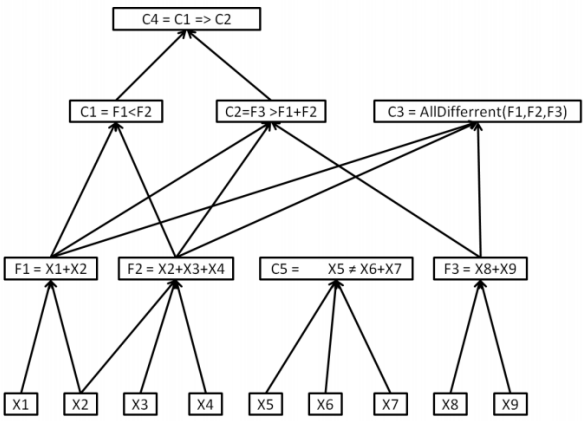

Figure 2.1: Example of dependency graph.

2.2 Search

The search is a procedure that iteratively moves from a current solution to one of its neighbors. In a CBLS program, the search queries the quality of the neighbors of considered neighborhood before deciding to select one by using the differentiation interface of constraints and objective functions. The definition of neighborhood and the selection of a neighbor to move to depends on the given problem and the search strategies (tabu search, simulated annealing,...) of the users.

We will demonstrate the API on two following examples. These are two constraint satisfaction problems: n-queens and Balanced Academic Curriculum Problem. The objective is to show how to use API for modelling the problem at hand. To assess the performance of the library in solving these problems, they are compared with constraint programming models in CHOCO library [**?**], which is also implemented in Java. All the experiments were conducted on machine Intel Core i5, 2.5GHz, 4GB Ram.

Example: n-queens.

Place n queens on a chess board of size $n \times n$ such that no two queens are on the same row, the same column, or the same diagonal. For modelling the n-queens problem, we use an array of variables $x[1, \ldots, n]$, in which $x[i]$ represents the column of the queen placed on row $i, \forall i = 1, \ldots, n$. The constraints of the problem are:

$$x[i] \neq x[j], \forall 1 \leq i < j \leq n \quad (1)$$

$$x[i] + i \neq x[j] + j, \forall 1 \leq i < j \leq n \quad (2)$$

$$x[i] - i \neq x[j] - j, \forall 1 \leq i < j \leq n \quad (3)$$

Listing **??** gives a complete example of the CBLS program for solving n-queens problem using multi-stage greedy search strategy proposed in [**?**]. Line 1 is the number of queens experimented. Line 2 initializes a `LocalSearchManager` object `ls`. Lines 3–5 initialize the array of decision variables in which the domain of each variable `x[i]` is $\{0, \ldots, n-1\}$ (see line 5). Line 6 creates a constraint system `S`. Line 7

Table 2.1: Comparison between Constraint Programming in CHOCO and CBLS for n-queens

n	CHOCO	CBLS
8	0.06	0.05
100	0.36	0.3
500	-	0.58
1000	-	1.47
2000	-	5.71
5000	-	53.2
10000	-	307.97

states the constraint (**??**). Lines 8–11 states the constraint (**??**). Lines 12–15 states constraint (**??**). Line 16 closes the model which generates the dependency graph relating variables and constraints of the model. Lines 17–24 is the search applying multi-stage greedy strategy. At each iteration, the search selects the most violating variable `sel_x` (line 20), and them it chooses one value `sel_v` for `sel_x` such that the number of violations of the constraint system `S` reduce most when `sel_x` is reassigned to `sel_v`. Line 22 performs the local move which assigns `sel_v` to `sel_x` and induces a propagation for automatically updating all the constraints of the model (this is done by the engine of the framework thanks to the dependency graph maintained).

Listing 1: A CBLS program for n-queens problem

```
1.  int n = 10000;
2.  LocalSearchManager ls = new LocalSearchManager();
3.  VarIntLS[] x = new VarIntLS[n];
4.  for(int i = 0; i < n; i++)
5.    x[i] = new VarIntLS(ls,0,n-1);

6.  ConstraintSystem S = new ConstraintSystem(ls);

7.  S.post(new AllDifferent(x));

8.  IFunction[] f = new IFunction[n];
9.  for(int i = 0; i < n; i++)
10.   f[i] = new FuncPlus(x[i],i);
11. S.post(new AllDifferent(f));

12. f = new IFunction[n];
13. for(int i = 0; i < n; i++)
14.   f[i] = new FuncMinus(x[i],i);
15. S.post(new AllDifferent(f));

16. ls.close();

17. int k = 0;
18. MinMaxSelector mms = new MinMaxSelector(S);
19. while(k < 1000000 && S.violations() > 0){
20.   VarIntLS sel_x =
          mms.selectMostViolatingVariable();
21.   int sel_v =
          mms.selectMostPromissingValue(sel_x);
22.   sel_x.setValuePropagate(sel_v);// local move
23.   k++;
24. }
```

Table 2.1 shows experimental results of the above CBLS program for n-queens problem on different value of the number of queens. In this table, we compare with a constraint programming in CHOCO using default search. Columns 2–3 present the execution times (in seconds) of CHOCO and CBLS (for CBLS, we report the average time in 10 executions) for finding a feasible solution. The table shows that CBLS program implemented in our library perform very well.

Example: Balanced Academic Curriculum Problem (BACP).

BACP is the problem number 30 of CSPLib [**?**] which consists of assigning a set of courses to a given number of periods satisfying constraints on prerequisites, on the load and the number of courses of periods. The input of BACP problem are:

- Set of courses $C = \{0, 1, \ldots, n-1\}$, for each course $i \in C, c[i]$ is the number of credits of i
- Set of periods $\{1, 2, \ldots, P\}$
- $minCourses, maxCourses$: minimum and maximum number of courses of each period
- $minCredits, maxCredits$: minimum and maximum number of credits of each period
- $L = \{\langle i, j\rangle\}$: course i must be assigned to the period before course j

For modelling the BACP, we use an array of variable $x[0, \ldots, n-1]$ in which $x[i]$ represents the period to which the course i is assigned. The constraints of the BACP problem are:

- $x[i] < x[j], \forall \langle i, j\rangle \in L$
- $minCourses \leq \sum_{i=0}^{n-1}(x[i] = p) \leq maxCourses, \forall p = 1 \ldots, P$
- $minCredits \leq \sum_{i=0}^{n-1}(x[i] = p) * c[i] \leq maxCredits, \forall p = 1 \ldots, P$

Listing **??** depicts the model for BACP problem in which lines 3–5 initialized an array of decision variables `x[0,...,n-1]`, the domain of `x[i]` is $\{1, \ldots, P\}$. Line 6–8 states the constraints of the prerequisites. Lines 9–13 states the constraints of the minimum and the maximum number of courses assigned to each period in which line 10 creates a function `Y` representing the number of elements of `x` having value `p` (`p` is the period considered). Lines 14–18 states the constraints on the minimum and the maximum number of credits assigned to each period in which line 15 creates a function `Y` representing the sum of credits of course `i` where `x[i]` is equal to `p`.

Listing 2: Model for BACP

```
1.  ls = new LocalSearchManager();
2.  S = new ConstraintSystem(ls);
3.  x = new VarIntLS[n];
4.  for(int i = 0; i < x.length; i++)
5.    x[i] = new VarIntLS(ls,1,P);

6.  for(int k = 0; k < I.length; k++){
7.    S.post(new LessThan(x[I[k]],x[J[k]]));
8.  }

9.  for(int p = 1; p <= P; p++){
10.   ConditionalSum Y = new ConditionalSum(x, p);
11.   S.post(new LessOrEqual(minCourses, Y));
12.   S.post(new LessOrEqual(Y, maxCourses));
13. }

14. for(int p = 1; p <= P; p++){
15.   ConditionalSum Y = new ConditionalSum(x, c, p);
16.   S.post(new LessOrEqual(minCredits, Y));
17.   S.post(new LessOrEqual(Y, maxCredits));
18. }
19. ls.close();
```

Table 2.2: Comparison between a constraint programming model in CHOCO and local search for BACP

Instance	CHOCO	CBLS
bacp-1.inp	0.8	1.89
bacp-2.inp	0.75	3.92
bacp-3.inp	0.75	2.86
bacp-4.inp	0.76	2.42
bacp-5.inp	0.77	5.1
bacp-6.inp	0.8	0.41
bacp-7.inp	0.75	2.15
bacp-8.inp	0.76	1
bacp-9.inp	0.76	2.44
bacp-10.inp	0.76	6.14

Table 2.2 presents the experimental results of CBLS program using built-in tabu search. We compare with a constraint programming model in CHOCO using default search. Columns 2–3 presents the execution times (in seconds) of CHOCO model and CBLS (for CBLS, we report the average time in 10 executions). The table shows that CHOCO model performs very well for BCAP problem. CBLS program using default tabu search can find feasible solutions in all executions but slower than CHOCO.

Table 2.3 presents partial list of the API of the constructed library including common interfaces of invariants, constraints, functions, and basic constraints $<=, !=, ==, \Rightarrow, ...$, functions $+, -, *$,div, mod,... We also design and implement fundamental global functions (e.g.,`Sum, ConditionalSum, Element, Occurrence`,...) and constraints (`AllDifferent, MultiKnapsack`,...) allowing users to easily model different problems.

For example, the function `ConditionalSum(VarIntLS[] x, int[] w, int val)` represents the sum

$$\sum_{i=0}^{x.length-1} (x[i] = val) * w[i]$$

. The library is designed to be extensible. Users can implement their own constraints, functions, or invariants and integrate them into the system.

Listing **??** shows the generic tabu search for constraint satisfaction. The input parameters consists of

- `c`: the constraint to be satisfied
- `tbl`: the tabu tenure
- `maxStable`: the search will be restarted if the best solution is not improved after `maxStable` from the last improvement.
- `maxIter`: maximum number of iterations of the search

Line 3 retrieves the list of all variables `x` participating in the constraint `c`. Lines 4-6 initialize a 2-dimension array `tabu` representing the tabu structure: `tabu[i][v]` indicates the iteration from which the move `x[i]`←`v` is allow. In other words, if `it <= tabu[i][v]`, then the move `x[i]`←`v` is forbidden (see line 19). At each iteration, the search collects a list `moves` of best legal neighbors of the neighborhood of current solution (lines 15–31). A legal neighbor is the neighbor that is not forbidden or is better than the best solution found so far (aspiration criterion). This is done by scanning all pairs of one variable `x[i]` and a value `v` of its domain (see lines 16–17). The legality of a neighbor is checked in lines 20–21). If no such `moves` contains no neighbor, the search is restarted (line 33, see the detail of `restart` in lines 54–60). Otherwise, a random neighbor is selected from `moves` (lines 35–36). Line 37 performs the move and line 38 update `tabu` structure. The counter `nic` will be augmented by 1 if new selected solution does not improve the best solution found so far `best` and the search is restarted in case `nic` exceeds `maxStable` (see lines 42–48). Lines 40–41 update the best solution and reset the counter `nic`.

Listing 3: generic tabu search

```
1  public void search(IConstraint c, int tbl,
2                     int maxStable, int maxIter){
3    VarIntLS[] x = c.getVariables();
4    int maxD = JCBLS.getMaxValueDomain(x);
5    int[][] tabu = new int[x.length][maxD+1];
6    JCBLS.fill(tabu, -1);
7    Random R = new Random();
8    int it = 0;
9    int nic = 1;
10   int best = c.violations();
11   ArrayList<PairInt> moves =
12                       new ArrayList<PairInt>();
13   while(it < maxIter && c.violations() > 0){
14     int minD = JCBLS.MAX_INT;
15     moves.clear();
16     for(int i = 0; i < x.length; i++){
17       for(int v : x[i].getDomain())
18         if(v != x[i].getValue()){
19         int d = c.getAssignDelta(x[i], v);
20         if(tabu[i][v] < it ||
21                       d + c.violations() < best){
22           if(minD > d){
23             moves.clear();
24             moves.add(new PairInt(i, v));
25             minD = d;
26           }else if(minD == d){
27             moves.add(new PairInt(i, v));
28           }
29         }
30       }
31     }
32     if(moves.size() == 0){
33       restart(x,tabu);
34     }else{
35       PairInt m =
36               moves.get(R.nextInt(moves.size()));
37       x[m.i].setValuePropagate(m.j);
38       tabu[m.i][m.j] = it + tbl;
39       if(c.violations() < best){
40         best = c.violations();
41         nic = 1;
42       }else{
43         nic++;
44         if(nic > maxStable){
45           restart(x,tabu);
46           nic = 1;
47         }
48       }
49     }
50     it++;
51   }
52 }
53
54 public void restart(VarIntLS[] x, int[][] tabu){
55   for(int i = 0; i < x.length; i++){
56     int v = JCBLS.randomSelect(x[i].getDomain());
57     x[i].setValuePropagate(v);
58   }
59   JCBLS.fill(tabu, -1);
60 }
```

Table 2.3: Partial list of the API of the library

Name	Description
`LocalSearchManager`	a local search manager
`VarIntLS`	represents the decision variables
`Invariant`	interface for invariants
`IFunction`	interface for functions
`IConstraint`	interface for constraints
`FuncPlus, FuncMinus, FuncMult FuncMod, FuncDiv`	+,-,*,div,mod
`IsEqual, LessOrEqual LessThan, NotEqual Implicate, AND, OR`	==, <= <, != =>, ∧, ∨
`Sum, ConditionalSum`	sum of decision variables, functions
`Element`	represent the element of an array whose index is a variable or function
`Occurrence`	represent the number of occurrences of a variable in an array
`Min, Max`	minimum and maximum element of an array
`AllDifferent`	constraint stating that the elements of an array are pairwise distinct
`Multiknapsack`	constraint for assignment with capacity constraint
`AtMost`	bound on the number of occurrences of a value in an array must be bounded
`MinMaxSelector`	selection operator, used for searching solutions
`TabuSearch`	built-in TabuSearch
`MultiStageGreedySearch`	built-in multi-stage greedy search

3. THE MTDT PROBLEM

Schedule the timetable for master thesis defense is a struggling mission that staffs must carry out in most of vietnamese universities due to some policies. In each defense session (two or three defense sessions are open each academic year), there are a set of master students who will defense their thesis. Each student has one master thesis being scheduled in the defense session, henceforth, we use thesis-student instead of student or his thesis. The jury of each thesis-student consists of five members: two examiners, a president, a secretary, and an additional member and this jury must be scheduled in one room and a slot of the session satisfying a given set of constraints. Among five members of the jury, there must be two members who are not professors/lecturers of the university and who are invited to participate in the jury: one is an examiner and the other is additional member. The supervisor of a thesis-student cannot be a member of the jury of that thesis-student. Two juries sharing a member must be scheduled in two different slots. The assignments of the professors/lecturers to juries should optimize some objectives, for instance, the occurrences of professors/lecturers in juries should be balanced, the theme of a thesis-student should match with the expertise of two examiners participating in the jury of that thesis-student, etc. We describe in the following section the mathematical formulation of the problem.

3.1 Problem formulation

Input.

- $IP\{0,\ldots,m_1-1\}$: set of professors of the university
- $EP=\{m_1,\ldots,m_1+m_2-1\}$: set of professors outside the university
- $P=IP\cup EP$: set of professors participating the defense schedule
- $S=\{0,\ldots,n-1\}$: the set of thesis-students, for each thesis-student s, $sup(s)$ is the supervisor of s
- $m(s,p)$: the score that measures the matching of expertise of professor p and the thesis-student s

Variables.

- $xp(i,s)$ represents the professor assigned in the ith position of the jury of thesis-student s
  - $xp(1,s)$ represents the examiner 1, $xp(1,s)\in EP$
  - $xp(2,s)$ represents the examiner 2, $xp(2,s)\in IP$
  - $xp(3,s)$ represents the president, $xp(3,s)\in IP$
  - $xp(4,s)$ represents the secretary, $xp(4,s)\in IP$
  - $xp(5,s)$ represents the additional member, $xp(5,s)\in EP$
- $xs(s)$: slot of the jury of the thesis-student s
- $xr(s)$: room of the jury of the thesis-student s

Invariants.

- $o(p)=\sharp\{(i,s)\mid i\in\{1,\ldots,5\}\wedge s\in S\wedge xp(i,s)=p\}, \forall p\in P$: number of juries that the professor p participates in
- $e(p)=\sharp\{(i,s)\mid i\in\{1,2\}\wedge s\in S\wedge sp(i,s)=p\}$: number of times professor p is scheduled as examiner
- $minP=\min_{p\in P}\{o(p)\}$
- $maxP=\max_{p\in P}\{o(p)\}$

Constraints.

$$xp(i,s)\neq xp(j,s), \forall 1\leq i<j\leq 5, s\in S \tag{4}$$

$$xs(s_1)=xs(s_2)\Rightarrow xp(i,s_1)\neq xp(j,s_2), \forall s_1\neq s_2\in S, \quad i\neq j\in\{1,\ldots,5\} \tag{5}$$

$$xs(s_1)=xs(s_2)\Rightarrow xr(s_1)\neq xr(s_2), \forall s_1\neq s_2\in S \tag{6}$$

$$e(p)\leq\lambda \tag{7}$$

Constraint (??) states that 5 members of a jury of each thesis-student must be all different. Constraint (??) states that if the juries of thesis-students s_1 and s_2 share a professor, then the slots assigned to these juries must be different. Constraint (??) states that if the juries of s_1 and s_2 are assigned to the same slot, then they must be assigned to different rooms. Finally, constraint (??) states that the number of times each professor is assigned as an examiner must be less than or equal to a given bound λ.

Objectives.

$$obj1 = maxP - minP \tag{8}$$

$$obj2 = \sum_{s \in S} m(s, xp(1,s)) + m(s, xp(2,s)) \tag{9}$$

There are two objective functions: $obj1$ in (??) to be minimized reflects the balance of load between professors participating in the defense session; $obj2$ in (??) to be maximized reflects the match between the expertise of the examiner 1 and the examiner 2 with the corresponding thesis-student of the jury. These objective functions are in lexicographic order. We would like to optimize $obj1$ first, and then to optimize $obj2$.

BUI et al. have considered the MTDT problem in [?]. In that paper, the objective of matching between the expertise of two examiners and the corresponding thesis-student of the jury was not taken into account. Instead, the paper considers the objective of minimizing waiting-time of professors. The author then proposed a multi-objective genetic algorithm for solving it that computes a set of non-dominant solutions. In this paper, we model the problem and use black box tabu search of the constructed library for solving the MTDT problem formulated in Section ??.

Listing ?? depicts the model of MTDT problem in our library. Lines 3–12 declare decision variables in which `xp[i][0], xp[i][1],...,xp[i][4]` represent 5 members of the jury of thesis-student `i`, `xr[i]` and `xs[i]` respectively represents the room and slot of the jury of thesis-student `i`. Lines 17–23 state the constraint (??). Lines 24–27 state the constraint (??). Lines 28–31 state the constraint (??), and the constraint (??) is stated in lines 35–44. The first objective function (??) is stated in lines 45–51 and lines 52–58 state the second objective function (??).

Listing 4: Model of MTDT

```
1.  ls = new LocalSearchManager();
2.  S = new ConstraintSystem(ls);
3.  xp = new VarIntLS[n][5];
4.  xs = new VarIntLS[n];
5.  xr = new VarIntLS[n];

6.  for (int i = 0; i < n; i++) {
7.    xp[i][0] = new VarIntLS(ls, m1, m1 + m2 - 1);
8.    xp[i][1] = new VarIntLS(ls, 0, m1 - 1);
9.    xp[i][2] = new VarIntLS(ls, 0, m1 - 1);
10.   xp[i][3] = new VarIntLS(ls, 0, m1 - 1);
11.   xp[i][4] = new VarIntLS(ls, m1, m1 + m2 - 1);
12. }

13. for (int i = 0; i < n; i++) {
14.   xs[i] = new VarIntLS(ls, 0, nbSlots - 1);
15.   xr[i] = new VarIntLS(ls, 0, nbRooms - 1);
16. }
17. for (int i = 0; i < n - 1; i++)
18.   for (int j = i + 1; j < n; j++)
19.     for (int p1 = 0; p1 < 5; p1++)
20.       for (int p2 = 0; p2 < 5; p2++)
21.         S.post(new Implicate(
22.                new IsEqual(xp[i][p1], xp[j][p2]),
23.                new NotEqual(xs[i], xs[j])));

24. for (int i = 0; i < n; i++) {
25.   VarIntLS[] y = copyRow(xp,i);
26.   S.post(new AllDifferent(y));
27. }

28. for (int i = 0; i < n - 1; i++)
29.   for (int j = i + 1; j < n; j++)
30.     S.post(new Implicate(new IsEqual(xr[i], xr[j]),
31.            new NotEqual(xs[i], xs[j])));

32. for (int i = 0; i < n; i++)
33.   for (int j = 0; j < 5; j++)
34.     S.post(new NotEqual(xp[i][j], sup[i]));

35. VarIntLS[] y = copyColumn(xp, 1);
36. for(int i = 0; i < m1; i++){
37.   IFunction o = new Occurrence(y, i);
38.   S.post(new LessOrEqual(o,lambda));
39. }

40. y = copyColumn(xp,0);
41. for(int i = m1; i < m1+m2; i++){
42.   IFunction o = new Occurrence(y, i);
43.   S.post(new LessOrEqual(o,lambda));
44. }

45. y = copyTo1DArray(xp);
46. occ = new IFunction[m1+m2];
47. for(int i = 0; i < m1+m2; i++)
48.   occ[i] = new Occurrence(y, i);
49. IFunction oMax = new Max(occ);
50. IFunction oMin = new Min(occ);
51. obj1 = new FuncMinus(oMax,oMin);

52. match1 = new IFunction[n];
53. match2 = new IFunction[n];

54. for(int i = 0; i < n; i++){
55.   match1[i] = new Element(m,i,xp[i][0]);
56.   match2[i] = new Element(m,i,xp[i][1]);
57. }
58. obj2 = new FuncPlus(new Sum(match1),
                        new Sum(match2));

59. ls.close();
```

Listing ?? is the search for MTDT problem. We use a built-in TabuSearch component of the constructed library as a black box. In this search procedure, the neighborhood of a solution is defined to be the set of solutions generated from the that solution by reassigning a new value to one selected variable. The search is divided into 3 phases. In Phase 1, we search a feasible solution (see line 2). In the second phase, we search a solution minimizing the objective function `obj1` while keep the constraint system `S` satisfied. In the third phase, we search a solution maximizing the second objective function `obj2` while keeping the satisfaction of the constraint system `S` and the value of the first objective function `obj1`. The time limit for each phase is 30 seconds.

Listing 5: Search of MTDT

```
1.  TabuSearch ts = new TabuSearch();

2.  ts.search(S, 20, 30, 1000000, 200);

3.  ts.searchMaintainConstraintsMinimize(
            obj1, S, 20, 30, 100000, 200);

4.  ts.searchMaintainConstraintsFunctionMaximize(
            obj2, obj1, S, 20, 30, 100000, 200);
```

Table 3.1: Instance description

Ins.	Number Thesis-students	Number Rooms	Number Slots	Int. Prof.	Ext. Prof.
1	9	2	5	9	4
2	17	4	5	12	11

Table 3.2: Experimental results for MTDT problem

Instance	CHOCO		CBLS		
	$obj1$	$obj2$	$obj1$	$obj2$	t_best (s)
1	3	9	2	15	228.26
2	-	-	2	37	710.39

3.2 Experiments

We conduct an experiment in two real instances of the defense session in April 2015 from School of Information and Communication Technology, Hanoi University of Science and Technology. The description of the instances is given in Table 3.1 We compare the performance of CBLS program above and the performance of a CP program in CHOCO [**?**] with the default search.

The CHOCO program consists of two steps:

- Step 1: Find a solution minimizing the first objective function $obj1$
- Step 2: Find a solution maximizing the second objective function $obj2$ while maintaining the quality of $obj1$[2]

The time limit for each step of CHOCO program is 60 minutes. The run the CBLS 10 times. Each time, the TabuSearch in Listing **??** is repeated with different random initial solution with the total time limit of 15 minutes. Table 3.2 presents the experimental results of CHOCO and CBLS programs in 2 instances. The table shows that for the first instance (small instance), CHOCO can find a feasible solution but the CBLS program found a feasible solution with better objective values than the CHOCO program. For the second instance (larger instance), the CHOCO program cannot find any feasible solution after 60 minutes while the CBLS program can find a feasible solution. It is difficult to compare a CP and a CBLS algorithms because CP and CBLS are different paradigms in the resolution of combinatorial optimization problems and have different objectives. CP aims at finding the optimal solution and proving its optimality. CP is suitable for problem instances which are reasonable small. Whereas, CBLS aims at finding good solutions for large or very large problem instances. Choosing a good model, search strategies, or implement dedicated constraint propagation to improve the efficiency of a CP algorithm for large problem instances require significant research efforts. The objective of this comparison is to show that using a simple CP model and a default search of a CP solver (i.e., CHOCO solver) for such complex problem is not useful. In this case, a CBLS model and a built-in Tabu Search implemented in our library brings good results and it is simple to do.

Using the CBLS approach has also another advantage. The schedule computed automatically by algorithms might need to be modified (by reassigning, exchanging professors to thesis-students) due to other requirements that were not taken into account at the modelling step. In these cases, staffs need to be query the effect of the modifications on different criteria before taking decision. The available modelling API of the constructed library will be helpful for this purpose.

[2]This is done by posting the constraint saying that that $obj1$ is not greater than the best value of $obj1$ found in step 1.

4. CONCLUSION

In this paper, we implement a Java library[3] for constraint-based local search (CBLS) which is based on Comet programming language invented by Van Hentenryck and Michel. The library features many advantages of CBLS architecture and is implemented in Java, one of the most popular programming language for building industrial applications, which facilitates the development of industrial applications. Users can use the library as a black box by modelling the problem as call built-in search components (e.g., tabu search) or users can extend the library by designing and implementing user-define functions, constraints and integrate them into the system. The constructed library is then applied to the resolution of a real-world combinatorial optimization problem arising in most of vietnamese education institutions, the master thesis defense timetabling problem. The experimental results show the flexibility in modelling different problem and good performance. The process of finding solution to the given problem similar to the way human make the schedule manually. In this scenario, people need to query the effect of different actions (reassigning, exchanging professors between places of the schedule) on a number of criteria before taking decision. Hence, the available of the modelling API of our constructed library is very useful for this purpose. Our constructed library will be released as open source.

For future works, on the one hand, we continue to develop efficient algorithms for solving MTDT problem by exploiting the combination between CP and CBLS in Large Neighborhood Search approach. On the other hand, we focus on developing domain-specific CBLS such as scheduling, vehicle routing, as well as applying the library in other real-world industrial combinatorial optimization problems.

5. ACKNOWLEDGMENTS

This paper was partially sponsored by Vietnamese National Foundation for Science and Technology Development (project FWO.102.2013.04). This paper was also partially sponsored by Hanoi University of Science and Technology (project T2015-040).

[3]The library and the source code of examples, of the MTDT are available at http://mso.soict.hust.edu.vn/index.php/resources/155-source-code-for-master-thesis-defense-timetabling-problem