Internship report - M1IF
Set Constraints for Local Search in COMET

Loïc Blet

August 29, 2010

Abstract

The aim of this internship is to demonstrate the same effect of set variables for constraint-based local search as was already shown for classical constraint programming in a paper [vHS07], namely that set variables are not only a convenience for faster, higher-level modelling but also sometimes necessary because problem instances with integer variables would not fit into memory or take much more time.

The implementation platform is the constraint programming system COMET,[1] which currently lacks set variables, taking inspiration from a recent PhD thesis [Åg08] in the ASTRA group that introduced set variables to constraint-based local search, as well as from a more recent paper [DDVH09] adding graph variables and tree variables to the constraint-based local search back-end of COMET.

[1] `www.dynadec.com`

Contents

1 Introduction

In this report I will present the research I did on set constraints for local search during two months in the ASTRA group under the direction of Pierre Flener.

Constraint programming [Apt03] is a technique aiming at solving combinatorial problems. It uses constraints to model the problem and then to help pruning an underlying search tree. Local search algorithms are generally used to get a 'good' solution quickly, with no guarantee on its optimality. Constraint-based local search [VHM05] allows to model a problem using constraints and then to guide the local search algorithm with those same constraints.

To our knowledge set constraints have not been used in local search in full scale programming environment yet. We aim to highlight the versatility of set constraints and the possibilities they bring to constraint-based local search.

To begin with I will present the research group where I worked and the activities that took place while I was there. Then I will give some background theory on constraint programming and constraint-based local search so that we can tackle the next parts of this report. After that, I will go through the papers that leaded to this internship subject. Before the main part of this internship, I will briefly explain a piece of work I did to get used to Comet. At this point I will detail the set constraint system I implemented in the local search engine of Comet, and present some results on problems we tried to solve with it. At last, I will discuss possible future work to continue this research.

2 The ASTRA research group

During this internship I was part of the ASTRA research group on constraint programming. It lies inside Uppsala university where I studied the same year. Pierre Flener, my internship advisor, is the founder and leader of ASTRA.

2.1 Presentation

The ASTRA group is studying various aspects of constraint programming in finite domain, such as symmetry detection and breaking, global constraints or modelling. Recently they focused more on local search, as one of the two PhD students in the group, Jun He, is working on grammar constraints for constraint-based local search [HFP09], and my internship tackled issues in local search as well.

Pierre Flener teaches a course on constraint programming at Uppsala university. I attended and enjoyed this course. After passing it I went and asked Pierre for an internship. What you read now is the long term result of this action.

2.2 Activities during the internship

As I knew my internship subject since the end of October, I had the opportunity to join for some great conferences. I went to the ACP (Association for Constraint Programming) summer school 2010, I attended all the talks at SweConsNet (Swedish Constraint programming network) 2010 – which was held at Uppsala university – and I listened to seminars, among which Serdar Kadıoğlu's one enlightened me.

During this internship, I also took the opportunity to write a tutorial on the basic usage of constraint programming for a french website 'le site du zéro'.[2] So far the tutorial has been seen more than 20000 times.

The ACP summer School[3] took place in Aussois, in France, from the 3rd to the 7th of May. I was very lucky to be there. I was the only first year master student and there was only one or two second year master student, the rest of the sixty participants where either PhD students or researchers. I sure learned a lot among this many PhD students, all of them having a subject related to optimization or constraint programming. An other very impressive fact was the list of lecturers. There were many famous researchers in constraint programming. Jean-Charles Régin gave a talk on global constraints, which he hugely contributed to with his papers on 'all different' [Rég94] and 'global cardinality constraint' [Rég96]. Laurent Michel talked about constraint-based local search and he wrote a reference book [VHM05] on this topic together with Pascal van Hentenryck, also present who introduced constraint programming with the Comet language he is developing. Although I didn't knew the other lecturers' name beforehand – except for Pierre Flener –, I was impressed by the level of each presentation and all of them were inspiring. During this summer school I was offered two internships for the next year, which is great news.

Then, the 20th and 21st of May I attended the SweConsNet 2010[4] workshop. The talk that fascinated me most was Roberto Castañeda Lozano's on the automated random testing of a trading system [CLSW10], and he won the master thesis award for this research. During these two days, I talked to many interesting people such as Hakan Kjellerstrand who owns a very active blog on constraint programming[5] and Mikael Zayenz Lagerkvist who develops Gecode, and who told me about the tutorial I wrote (which uses Gecode) without knowing I was its author. That was quite strange to hear about this small piece of work so suddenly!

The seminar I found the most interesting during my work at ASTRA was on 'Simple and Efficient Search Procedures for Combinatorial Optimization' given by Serdar Kadıoğlu. This is where I learned about dialectic search [KS09], which I used at the end of this internship.

[2]`http://www.siteduzero.com/tutoriel-3-193225-decouverte-de-la-programmation-par-contraintes.html`

[3]`http://becool.info.ucl.ac.be/summerschool2010/`

[4]`http://www.sics.se/~agren/SweConsNet2010/`

[5]`http://www.hakank.org/constraint_programming_blog/`

3 Constraint Programming and Local Search

In this section I will introduce the basic ideas of the field I worked on for this internship. To begin with, I will introduce constraint programming, which I studied during Pierre Flener's course, and then I will go deeper into local search and also set variables, which I had to discover for this internship.

3.1 Constraint programming

As said on the ASTRA group web page, constraint programming [Apt03] is a relatively new technology for solving combinatorial (optimisation or decision) problems. It works in an orthogonal but complementary way to mathematical programming (for instance, linear programming and its specialisations), which originated from Operations Research some 50 years ago. Constraint programming also is a powerful alternative to optimisation via (partial) differential equations. For many combinatorial problems, constraint programming has been shown very effective, if not (vastly) superior to mathematical approaches, while the converse or a tie occurs for many other problems. Naturally, all these approaches are bound to converge in the future. Constraint programming applies techniques from algorithms, artificial intelligence, combinatorics, computational logic, concurrent computation, database management, discrete mathematics, operations research, programming languages, symbolic reasoning, user interfaces, visualisation, . . . to application areas as diverse as collaborative decision making, configuration, control, design, diagnostic, finance, logistics, molecular biology, planning, resource allocation, rostering, scheduling, time-tabling, transport, . . .

Now, let us give some formal definitions for this topic. All the definitions of this section are taken from Magnus Ågren's PhD thesis [Åg08].

We use *constraint satisfaction problems* to formally model a given combinatorial problem.

Definition 1 (CSP)**.** *A* constraint satisfaction problem *(CSP) is a three-tuple* $\langle \boldsymbol{V}, \boldsymbol{D}, \boldsymbol{C} \rangle$ *where*

- $\boldsymbol{V}$ *is a finite set of variables;*
- $\boldsymbol{D}$ *is a domain containing the possible values for the variables in* $\boldsymbol{V}$*; and*
- $\boldsymbol{C}$ *is a set of constraints, each constraint in* $\boldsymbol{C}$ *being defined on a sequence of variables taken from* $\boldsymbol{V}$ *and specifying the allowed combinations of values for that sequence.*

Given the definition above: without loss of generality, all variables share the same domain since we can always achieve smaller domains for particular variables by additional membership constraints. Also the set of variables in the variable sequence of a constraint c is referred to as $vars(c)$.

In order to solve a CSP we must find an assignment to all the variables that satisfies all the constraints.

Definition 2 (Configuration and Solution)**.** *Let* $P = \langle \boldsymbol{V}, \boldsymbol{D}, \boldsymbol{C} \rangle$ *be a CSP:*

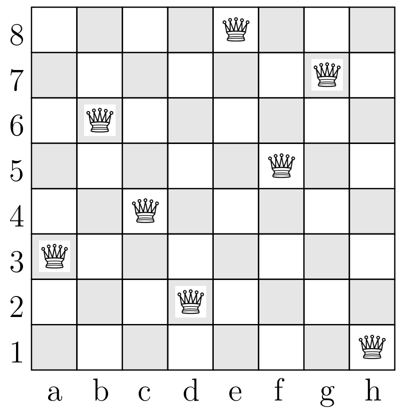

Figure 1: A solution to the n-queens problem for $n = 8$

- *A* configuration *is a function $k : \boldsymbol{V} \rightarrow \boldsymbol{D}$.*
- *The* set of all configurations *for P is denoted K_P.*
- *A configuration k is a* solution *to $c \in \boldsymbol{C}$ (or k satisfies c, or c is satisfied under k) if and only if $\langle x_1, \ldots, x_m \rangle$ is the variable sequence of c and $\langle k(x_1), \ldots, k(x_m) \rangle$ is one of the allowed combinations of values for that sequence as specified by c.*
- *A configuration k is a* solution *to P if and only if k is a solution to all constraints in $\boldsymbol{C}$.*

Example 1 (The n-queens problem)**.** *The n-queens problem is stated as follows:*
Place n queens on an $n \times n$ chess board such that no two queens attack each other following the rules of chess.
Hence, we must place the queens in the chess board such that no two queens share the same row (row constraint), column (column constraint) or diagonal (diagonal constraint). a solution to this problem for $n = 8$ is shown in figure 1.

Here is a classical CSP model of this problem: assuming that we number the queens from 1 to n, the set of variables $\mathbf{V}$ is the set $\{q_1, \ldots, q_n\}$, where each $q_i \in \mathbf{V}$ is the variable for the ith queen. The domain $\mathbf{D}$ is the interval from 1 to n where each $j \in \mathbf{D}$ is the value for the jth row. So, in this model, given a variable q_i representing the ith queen, we let the index i denote the column of the queen i, while the value of q_i under a configuration denotes the row of queen i. For instance, the solution displayed in figure 1 is the configuration:

$$\{q_1 \rightarrow 3, q_2 \rightarrow 6, q_3 \rightarrow 4, q_4 \rightarrow 2, q_5 \rightarrow 8, q_6 \rightarrow 5, q_7 \rightarrow 7, q_8 \rightarrow 1\}$$

Since the column of each queen is predetermined in such a way that the column constraint is satisfied, the only constraints we have to actually care about are the row and diagonal ones. We may state the row constraint as follows:

$$\forall 1 \leq i < j \leq n, q_i \neq q_j$$

By forcing two different queens to be assigned to different values – and thus different rows –, we make sure the row constraint is satisfied. The diagonal constraint is split in two parts according to the top-right to bottom-left (tr-bl) and top-left to bottom-right (tl-br) diagonals respectively. The tr-bl part is stated as follows:

$$\forall 1 \leq i < j \leq n, q_i - i \neq q_j - j$$

For a queen q_k the expression $q_k - k$ give the number of its tr-bl diagonal, so we force this number to be different for each pairs of two queens. With the same reasoning we come to the following tr-bl part of the diagonal constraint:

$$\forall 1 \leq i < j \leq n, q_i + i \neq q_j + j$$

So we now have, all our variables, domains and constraints stated, which conclude this CSP declaration.

3.2 Tree search interleaved with constraint propagation

Constraint programming (called global search when it is opposed to local search) traditionally offers a programming front-end to a global search technique for solving CSPs. This means that the whole search space of a CSP is – maybe implicitly – explored in some systematic way, which gives rise to a search tree, until a solution is found or until it is proved that no solutions exists. This is done by *extending partial configurations* into solutions, and by *backtracking* if an infeasible state is reached. In its most simple version, such a global search technique would enumerate each possible configuration of the given CSP.

The beauty of constraint programming lies in the ability to avoid enumerating each possible configuration of the CSP. this is achieved by reasoning about the semantics of the problem constraints at every node in the search tree, a priori, restricting certain variables from taking certain impossible values. To do this, each constraint is represented by one or more *filtering algorithms* that may reduce the domain for the variables according to some *level of consistency*. The common theme for such filtering algorithm is to exploit necessary conditions of constraints and to recognize values that, if taken, would violate some necessary condition.

The success of constraint programming is due to the concept of *global constraints*. These are primitive constraints that are defined on a non-fixed number of variables and capture some combinatorial substructure.

Example 2 (The AllDifferent global constraint)**.** *For our CSP model of the n-queens problem we used three sets of $\neq$ constraints. But each set of such constraints could actually be explained in this manner (for the row constraint) 'the variables in $\{q_1, \ldots, q_n\}$ must all take different values'. The global constraint* AllDifferent$(\{q_1, \ldots, q_n\})$ *has exactly this meaning. So we can replace the three sets of $\neq$ constraints by just three* AllDifferent(X) *constraints.*

The constraint AllDifferent(X) is the classical example of a global constraint. It captures the combinatorial substructure that all the variable in X must take distinct values: a very common substructure in combinatorial problems. The importance of global constraints comes from the higher levels of consistency they allow.

Example 3 (Consistency allowed by the AllDifferent constraint)**.** *Consider the constraints* $x \neq y$*,* $x \neq z$ *and* $y \neq z$*. Assume that the current set of possible values for* x *and* y *is* $\{1, 2\}$ *and* $\{1, 2, 3\}$ *for* z*. There is clearly no solution* s *to all constraints above where* $s(z) = 1$ *or* $s(z) = 2$*, since then there would not be enough values left for* x *and* y *to make them different. Hence, the values 1 and 2 should be removed from the domain of* z*. But by considering the three* $\neq$ *constraints in isolation we cannot deduce this. However, if we instead consider the global constraint* AllDifferent$(\{x, y, z\})$*, which is equivalent to the previous* $\neq$ *constraints, we can make this deduction. So the level of consistency when using* AllDifferent(X) *can be higher than the level of consistency when using a set of* $\neq$ *constraints.*

We are now going to introduce constraint-based local search.

3.3 Constraint-based local search

In constraint-based local search [VHM05] only parts of the search space of the CSP are explored. So finding a solution to a CSP can never be guaranteed, nor can its non existence be proven.

First, to find solutions a local search algorithm iteratively *moves* between configurations.

Definition 3 (Move)**.** *A* move function *is a function:*

$$m : K \to K$$

Given a configuration k *we call* $m(k)$ *a* move *from* k*, or a* neighbour *of* k*.*

Local search examines the merits of many moves from the current configuration until one of those moves is chosen as the next configuration. The configurations thus examined constitute the *neighbourhood* of the current configuration.

Definition 4 (Neighbourhood)**.** *A* neighbourhood function *is a function:*

$$n : K \to \mathcal{P}(K)$$

Given a configuration k*, the set of configuration* $n(k)$ *is called a* neighbourhood *of* k*, and each element thereof a* neighbour *of* k*.*

In constraint-based local search, *penalties* and *variable conflicts* are used to navigate between configurations in the search space: these are evaluations on how much the constraints are violated respectively how much particular variables contribute to those constraint violations.

Definition 5 (Penalty)**.** *A* penalty function *of a constraint c is a function:*

$$penalty(c) : K \to \mathbb{N}$$

such that penalty$(c)(k) = 0$ *if and only if k satisfies c, given a configuration k. The* penalty *of c under k is* penalty$(c)(k)$*. the penalty of* $\langle \boldsymbol{V}, \boldsymbol{D}, \boldsymbol{C} \rangle$ *under k is the sum:*

$$\sum_{c \in \boldsymbol{C}} \text{penalty}(c)(k)$$

Penalty functions are used to guide the local search algorithm towards promising regions in the search space and to determine whether or not a given configuration is a solution. It is therefore very important that a penalty function reflects the semantics of its constraint in a natural way. In that sense, the definition above is quite vague as many penalty functions will comply for a given constraint – some of which do not guide the search well.

Definition 6 (Variable specific neighbourhood)**.** *The* variable specific neighbourhood *of a variable x under a configuration k is the set of all configurations that differ from k for at most x:*

$$\{l \in K | \forall y \in \boldsymbol{V} : y \neq x \Rightarrow k(y) = l(y)\}$$

Definition 7 (Variable conflict)**.** *Let* penalty(c) *be the penalty function of a constraint c. A* variable conflict function *of c is a function*

$$conflict(c) : \boldsymbol{V} \times K \to \mathbb{N}$$

such that, given a variable x and a configuration k: if conflict$(c)(x, k) = 0$ *then* penalty$(c)(k) \leq$ penalty$(c)(l)$*, for each configuration l in the variable specific neighbourhood of x under k. The* variable conflict *of x with respect to* $\langle \boldsymbol{V}, \boldsymbol{D}, \boldsymbol{C} \rangle$ *and k is the sum:*

$$\sum_{c \in \boldsymbol{C}} conflict(c)(x, k)$$

Variable conflict functions play an important role for the efficiency of a local search algorithm since they allow to discard many uninteresting moves in a given neighbourhood.

We will next give an actual code for the n-queens problem.

3.4 An example with the Comet language: the n-queens problem

The language I used for this internship is COMET, it is an object-oriented language that may look like Java or C++ but it has many additions, including some constraint programming possibilities – global search as well as local search.

Here is a working code for the n-queens problem written in COMET, it is taken from Comet documentation.

Comet code 1.

```
 1. import cotfd;
 2.
 3. Solver<CP> m();
 4. int n = 8;
 5. range S = 1..n;
 6. var<CP>{int} q[i in S](m,S);
 7.
 8. cout << "Initiating search..." << endl;
 9. solveall<m> {
10.   m.post(alldifferent(all(i in S) q[i] + i));
11.   m.post(alldifferent(all(i in S) q[i] - i));
12.   m.post(alldifferent(q));
13. } using {
14.   forall(i in S) by (q[i].getSize()) {
15.     tryall<m>(v in S: q[i].memberOf(v))
16.       label(q[i],v);
17.   }
18.   cout << q << endl;
19. }
```

Line 6 we state the variables, which is an array of n integer variables. Lines 10 to 12 are given the three AllDifferent constraints. And to conclude this code, lines 14 to 16 gives the search procedure, which is to first try to assign a value to the variable with the smallest domain. This procedure is very common, works well in many cases and is also called the fail-first heuristic.

Let us continue with an example of a local search code, also taken from Comet documentation. It is based on the min-conflict heuristic.

Comet code 2.

```
 1. import cotls;
 2.
 3. Solver<LS> m();
 4. int n = 8;
 5. range Size = 1..n;
 6. UniformDistribution distr(Size);
 7.
 8. var{int} q[Size](m,Size) := distr.get();
 9. ConstraintSystem<LS> S(m);
10. S.post(alldifferent(q));
11. S.post(alldifferent(all(i in Size) q[i] + i));
12. S.post(alldifferent(all(i in Size) q[i] - i));
13. m.close();
14.
```

```
15. while (S.violations() > 0) {
16.   selectMax(i in Size)(S.violations(q[i]))
17.     selectMin(v in Size)(S.getAssignDelta(q[i],v))
18.       q[i] := v;
19. }
20.
21. cout << q << endl;
```

Notice how similar it is to the code for global search. The model part is actually the same. An initialisation part is added (line 8), and the search (lines 15 to 18) is now using the *getAssignDelta* move.

We have almost all the needed background for going on with the internship subject. One final thing to introduce is set variables.

3.5 Set variables

We will now consider CSPs with only *set variables* instead of integer variables. Hence, the domain $\mathbf{D}$ of the variables is in fact the power set $\mathcal{P}(U)$ of a set of values U (we will only consider integers), called the *universe*.

The use of set variables allows for more interesting moves than a simple *assign*. The two most basic moves are $\mathit{add}(S, v)$, which add the value v to the set variable S. Similarly, $\mathit{drop}(S, u)$ drops value u from the set variable S. Here are two more interesting moves:

- $\mathit{transfer}(S, u, T)$ transfers value u from S to T.
- $\mathit{swap}(S, u, v, T)$ swaps u of S with v of T.

When we will be dealing with implementing set constraints, we will have to implements these (or some of these) move functions.

4 Related research

We are now going to look into studies that inspired this internship subject and gave it some background. To begin with, the main theoretical study on the subject was done by Magnus Ågren in his PhD thesis [Åg08]. We could say that the main motivation was an article [vHS07] from Willem-Jan van Hoeve and Ashish Sabharwal where they showed the necessity of set constraints for efficiency, our aim being to point out the same need for set constraints in local search. And to conclude this section we will take a look at another trial to implement a new kind of constraints in the Comet language, in a paper [DDVH09] coming from Pham Quang Dung, Yves Deville and Pascal Van Hentenryck.

4.1 Set Constraints for Local Search [Åg08]

This thesis contains all necessary theory to understand constraint-based local search. Its aim is to provide a theoretical framework for set constraints in local search. It provides many 'built-in' set constraints. It gives for each such constraints penalty and conflict measures. It starts with basic constraints such as membership constraints or set equality constraints but then goes into more interesting constraints like inclusion constraint or partition constraint.

The main contribution of this thesis is the creation of a language for generic set constraints and hints as how to implement those. The language is monadic existential second order logic. It allows to express constraints such as $\mathit{Union}(\{S_1, \ldots, S_n\})$ which meaning is that the union of all S_i is the universe U. Here is how it could be written:

$$\exists S_1 \ldots \exists S_n (\forall x (x \in S_1 \vee \ldots \vee x \in S_n))$$

The rest of the thesis is devoted to the computation of penalty function and conflict function for such constraints, as well as their neighbourhoods.

4.2 Two Set-Constraints for Modeling and Efficiency [vHS07]

This article deals with constraint programming using global search. It presents two filtering algorithms for two set constraints: `sum-free` and `atmost1`.

The `sum-free` constraint ensure that a given set S contains no three-tuple (i, j, k) such that $i + j = k$. They use it to model the Schur problem, which decision problem is stated as follows: given two integers k, $n \geq 0$, does there exist a partition of $\{1, \ldots, n\}$ into k sum-free sets? When they compare an integer model equivalent to their set model, they find that even though the same amount of propagation is achieved, the memory usage is lower with the set model.

The `atmost1` constraint ensure that the intersection of two given sets is of cardinal at most one. The authors use it to model the social golfer problem. The problem `golf-`g`-`s`-`w asks for a partition of n golfers into g groups, each of size s, for w weeks, such that no two golfers are in the same group more than once throughout the whole schedule. This time, when comparing an integer model with a set model using their `atmost1` constraint, they notice that the additional time spent pruning values is reducing the time to reach a solution.

Overall this article gives credit to the use of set constraints not only for modeling ease but also for efficiency. It encourages us to search for efficient filtering algorithms for set constraints.

4.3 LS(Graph & Tree): A Local Search Framework for Constraint Optimization on Graphs and Trees [DDVH09]

In this article, the authors present a framework for using graph constraints and tree constraints in Comet. After defining the possible moves in a graph (adding and removing edges or vertices), they give examples of constraints on graphs: the tree constraint or the

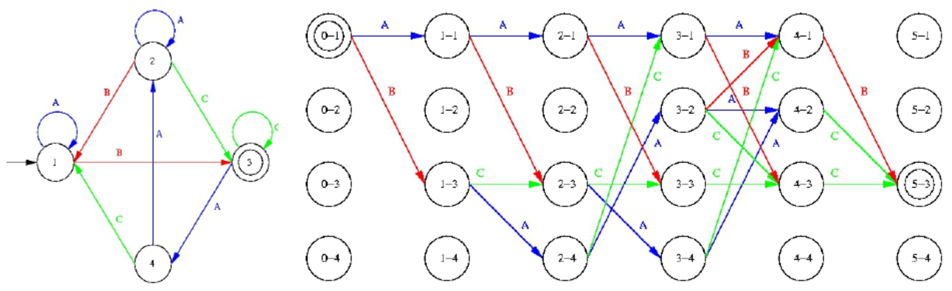
Figure 2: An automaton and its layered graph for a word of length 5

connected constraint. The rest of the article is a detailed explanation on how their implementation works. To conclude they compare an implementation in Comet to one in C++. They acknowledge that the Comet implementation is around 2.5 times slower than the C++ one, but the Comet implementation offers an easier way to code the framework and to use it. They also point out that integrating the C++ framework to the Comet solver (written in C++) would give the best of both worlds: speed and commodity of usage.

5 The regular constraint

Before starting to implement a local search set constraint framework for Comet, Pierre Flener gave me an exercise to 'warm up' with Comet. The goal was to implement a regular constraint in Comet. The constraint implementation idea comes from Benoit Pralong's master thesis [Pra07], which was written in french – thus I was a good candidate for reading and understanding it.

We will now look into the regular constraint (See implementation in Appendix A). Given an automaton and a word, the regular constraint is satisfied if and only if the word is accepted by the automaton. To compute the violations of this constraint, we use the Hamming distance which is equal to the minimal number of letters to change in the given word to get to a word in the given language.

To compute this distance, we use a layered graph of the automaton. this means that we have to unfold the automaton in the following way:

- Let l be the length of the given word w
- Let n be the number of states in the given automaton $A = (\Sigma, Q, q_0, F, \delta)$

We get the layered graph G_A of the automaton A by constructing a graph of $l+1$ layers each of n states corresponding to the set Q. The only transition are those from layer i to layer $i+1$ using the δ function. The last step is to remove transitions leading or coming from non reachable or non co-reachable states. See figure 2 for an example of such a layered graph.

To compute the Hamming distance of a word to the language with the corresponding layered graph, we proceed as follows: in each layer the transition corresponding to the letter in the word is of cost 0, the other transitions are of cost 1. In this new weighted graph, we get the Hamming distance with the cost of a shortest path from the initial state to any final state. With this shortest path we can even get the place of the wrong letters.

We used this constraint to solve an instance of the nurse scheduling problem[6] for six weeks. Our program solves it in an average of 350 milliseconds. On a bigger instance of twelve weeks it takes an average of 1400 milliseconds to solve. A detailed comparison between this version of regular and an other one created by Jun He, Pierre Flener and Justin Pearson from the ASTRA group, can be found in [HFP09].

When discussing with Pierre Flener about this constraint, we noticed that the violations are computed based on one specific shortest path. It directs the search towards some variable more than others, which can be bad in local search where stochastic process are a good thing.

6 A set constraint system for Comet

In this section we will go into the detail of the set constraint system that was implemented in Comet during this internship. The core component is composed of a `SetConstraint` interface and a `SetConstraintSystem` class that will handle all of our set constraints. Then we have to code all the set constraints we want to use. We will show examples of such constraints. Once those foundations are laid we can start solving problems and trying different search heuristic. There, we will present Tabu search and Dialectic search

6.1 The core system

To begin our set constraint system, we need an interface for set constraints (see the code in Appendix B.1) that will be handled by the system. A set constraint will have to fulfill some requirements. First it needs a method telling whether it is satisfied or not. Then it has to have a setup function usually called 'post' in the constraint programming jargon. The rest of the functions are used to guide the search. The constraint should have its penalty function and conflict function, here both are called 'violations'. The functions evaluating moves come at the end of this interface declaration that we give here.

Comet code 3.

```
interface SetConstraint{
  var{boolean} isTrue();
  void post();
  var{int} violations();
  var{int} violations(var{set{int}} s);
  var{set{int}}[] getVariables();
  int getAddDelta(var{set{int}} s, int v);
  int getDropDelta(var{set{int}} s, int u);
  int getFlipDelta(var{set{int}} s, int u, int v);
```

[6]http://en.wikipedia.org/wiki/Nurse_scheduling_problem

```
    int getTransferDelta(var{set{int}} s, int u, var{set{int}} t);
    int getSwapDelta(var{set{int}} s, int u, int v, var{set{int}} t);
}
```

The active part of the core of the system is a class called `SetConstraintSystem` (see the code in Appendix B.1). Its main component is an array of set constraints. The interesting thing about this class is that it is itself implementing the `SetConstraint` interface. It allows us to ask the system for all its violations, or all the violations of one variable. We can use it to judge the effect of moves on the whole problem we try to solve.

6.2 The set constraints

Once we have the core system ready to go, we need to use it with some constraints (see the code for all constraints in Appendix B.2). We started to code simple constraints to test the system while it was still being developed. The first two constraints were membership constraints: $\mathit{InSet}(S, i)$ is satisfied if and only if the integer i is in the set variable S, and $\mathit{NotInSet}(S, i)$ is satisfied if and only if the integer i is not in the set variable S.

The next constraint to be implemented was `Partition` that takes two arguments, a set variable S and an array of set variables p. The constraint is satisfied if and only if the union of all the set variables in p is a partition of the set variable S.

The rest of the constraints we coded were needed for solving the problems we chose to tackle. These are the `SumFree` and `AtMost1` constraints (see section 4.2).

The `Sumfree` constraint has its violations computed easily: it is the number of violating three-tuple (i, j, k) with $i \leq j \leq k$. However we did not find any efficient way of evaluating the cost of moves. We could store the violating tuples and quickly find if dropping a value would pick such a tuple out. However we could not think of a smart way to tell if adding a value would add a violating tuple. That is why we just loop for each two values in the set and check if their sum is in the set as well. Hence a quadratic time in the cardinal of the set to check for the constraint violations.

To compute the violations of the `AtMost1` constraint, we used a feature of Comet called *invariants*. It allows to keep track of an expression of variables, even when these variables change their values. It involves a heavy machinery behind that we were glad not to have to code. Using set invariants the computation of the violations of this constraint becomes easy:

$$\mathit{violations} = \max(1, |S_1 \cap S_2|) - 1$$

where S_1 and S_2 are the set variables concerned by the constraint. As for the moves, they are easily evaluated with a few conditions. Figure 3 is an example for the `getDropDelta` move.

We assume in this algorithm that the parameter S is one of the two set variables concerned in the constraint (S_1 or S_2). Thanks to the Comet invariants, we do not have to compute the first condition over and over. We compute it once when it is declared and then it updates automatically when the sets are modified. Thus this move is very cheap.

```
getDropDelta(set S, int u)
    if $|S_1 \cap S_2| \leq 1$
        return 0
    else if $u \in S_1 \wedge u \in S_2$
        return $-1$
    else
        return 0
```

Figure 3: Computation of the getDrop move for the AtMost1 constraint

If we want the violations of a given set variable, we just have to check whether it is one of the two set variables concerned by the constraint. If it is, then we return the violations of the constraint, if it is not, we return zero.

We will now, at last, use these constraints to solve problems and compare different approaches.

6.3 Experimentations and results

6.3.1 The Schur problem

We already mentioned the Schur problem in section 4.2, so we will only present the model (listed in Appendix C) and the search heuristic used.

To model this problem we only need to start with a random partition of the set from 1 to n and to post a `SumFree` constraint on each of the k sets of this partition. The search will ensure that the k sets always form a partition, thus we have no need for a partition constraint.

We used a tabu search [GL98] to try and solve this problem. This heuristic consist in a greedy search but with a *tabu array* containing the last few variables changed so that we cannot change these before a given number of iterations – which is the length of the tabu array, also called the *tabu length*. This algorithm seems pretty simple but it has many possible improvements coming with many parameters to adjust making it quite complex to fine tune for a specific problem. For this problem we chose a tabu length of 10 and use the best 'transfer' move from any of the k sets to another of these sets. Unfortunately even after tweaking the parameters, it seems that the tabu search is really slow. This is most certainly due to the implementation of the `AtMost1` constraint which, as mentioned before, is not satisfactory. This leads to long iterations and it takes way too long to solve apparently simple instances. For example in [vHS07] it takes less than five seconds to solve the problem for $k = 6$ and $n = 600$ when only one iteration takes more than that in our version.

It is also not impossible that local search is not well suited for solving this problem and global search constraint programming manages to prune very efficiently the search space.

6.3.2 The social golfer problem

We also mentioned this problem in 4.2, we will present our model and an other search heuristic we used for this problem.

The model is simple. Let $\mathit{schedule}(w, g)$ bet the the group g of week w. The constraints are as follow:

$$\forall i \in W, \forall j \in G, \forall k \in W, \forall l \in G, i < k : AtMost1(\mathit{schedule}(i, j), \mathit{schedule}(k, l))$$

where W is the interval from 1 to w and G the interval from 1 to g. We will start with a random initial configuration that has for each week a partition of groups. Then the search will keep the partitions formed inside each week of the schedule. The only moves we will perform here are swaps between two groups of the same week.

We tried two different heuristics on this problem: tabu search that has already been presented, and dialectic search [KS09] (see the code in appendix C). The principle of the dialectic search is to keep the exploration of the search space separated from its exploitation. To do that, at each step of the dialectic search we use three functions consecutively:

- `greedy`, that will greedily improve the current configuration (exploitation part)
- `modify`, that will, step by step, randomly modify the current configuration (exploration part)
- `merge`, that will pick the best configuration along the path of changes performed by `modify` and will then greedily improve it.

This heuristic has nice advantages when compared to tabu search. It is easy to understand and to code, as the separation between exploration and exploitation allows to put our knowledge of the problem in a clear way. It is also nice because each step is guaranteed to have at most as many violations as the previous one, whereas tabu search can oscillate a lot.

We compared the two different heuristics and a model with integer variables given in the Comet documentation (see the code in appendix C) on different instances of the social golfer problem, see table 1 for the runtimes and table 2 for the number of iterations. These numbers are an average over 10 runs for each instance.

It clearly appears that the integer model is faster and takes less iterations to reach a solution in all cases. The code for this model is quite intricate and it involves the definition of a problem specific invariant. The other two heuristics have less of a gap between them. It seems that the dialectic search is better except on three instances where the tabu search performs way better.

We have to keep in mind that when using our set constraint system, we are slowing the whole program since we put a new layer on top of the Comet local search solver instead of the set constraint system being written in C++ and directly put in the Comet kernel. It may slow the program by an order of 2.5 as we encounter the same situation that occurs in [DDVH09].

instance	Tabu	Dialectic	Integer
golf-5-4-4	713	23534	34
golf-6-4-4	703	1652	35
golf-6-4-5	5960	303265	97
golf-9-4-3	454	271	25
golf-9-4-4	2544	1749	70
golf-7-3-2	4	5	1
golf-7-4-4	1050	1015	46
golf-8-4-4	1487	1232	60
golf-6-5-2	39	35	7
golf-6-5-3	424	1131	44
golf-6-5-4	9079	324934	148

Table 1: Runtime on social golfer instances. Instance golf-g-s-w asks to schedule g groups of size s over w weeks. Runtime is in milliseconds.

7 Conclusion

Let us now recapitulate our work and point out possible further work.

7.1 summary

We have built a system for using set constraints in local search, on top of the theoretic foundation laid in [Åg08] and on top of the COMET language. We developed some set constraints to get a proof of concept and went further with the application to well known problems such as the social golfer problem. We built models that where easy to understand and to code with no fine tuned problem specific methods. We have compared two different search heuristics, tabu search and dialectic search. Our preference goes to dialectic search which makes a better use of randomness for exploring the search space before being greedy.

7.2 Further work

We will now discuss ways to continue this work.

Even though we had time to optimize our system a bit, we think there is still room for a two or three times increase in speed for the core system.

We also can easily add other set constraints such as cardinality, union or intersection constraints. Something more rewarding but also more demanding would be to implement the generic set constraints introduced in [Åg08].

There is a lot of options to exploit moves allowed by the set variables that didn't fit the problems we try to solve. For instance we could use big neighbourhood with moves such as the intersection or union of two sets. We would have to look for problems seemingly well suited for such moves.

instance	Tabu	Dialectic	Integer
golf-5-4-4	45	626	19
golf-6-4-4	16	24	10
golf-6-4-5	141	2382	33
golf-9-4-3	5	2	6
golf-9-4-4	11	6	10
golf-7-3-2	1	0	1
golf-7-4-4	11	8	10
golf-8-4-4	10	6	10
golf-6-5-2	3	2	3
golf-6-5-3	12	24	9
golf-6-5-4	229	2808	36

Table 2: Number of iterations on social golfer instances. Instance golf-g-s-w asks to schedule g groups of size s over w weeks.

The goal to find a problem where a local search set constraint model performs better than any other technique is still to be achieved, but with the foundations we laid we could try to convince the company behind COMET to implement set constraints in the kernel of COMET so that it becomes available for everyone to use.

Acknowledgments

First of all, I would like to warmly thank Pierre Flener for making me discover constraint programming. This field was unknown to me a year ago and I was able to conduct an internship in a related field thanks to him, his support and his many pieces of advice. He also gave me the opportunity to meet many great researchers by sending me to the ACP summer school.

Thank you to Farshid Hassani Bijarbooneh as well, a PhD student in the ASTRA group, for the many great moments we spent in the Swedish courses – and also discussing constraint programming.

And my final acknowledgment will go to Sweden as it is a nice country to do research in, for it is calm and peaceful.

References

[Apt03] Krzysztof Apt. *Principles of Constraint Programming.* Cambridge University Press, 2003.

[CLSW10] Roberto Castañeda Lozano, Christian Schulte, and Lars Wahlberg. Testing continuous double auctions with a constraint-based oracle. In David Cohen, editor, *Proceedings of CP'10, the 16th International Conference on Principles and Practice of Constraint Programming*, volume 6308 of *Lecture Notes in Computer Science*, pages 613–627. Springer-Verlag, 2010.

[DDVH09] Pham Quang Dung, Yves Deville, and Pascal Van Hentenryck. LS(graph & tree): A local search framework for constraint optimization on graphs and trees. In Sung Y. Shin and Sascha Ossowski, editors, *Proceedings of SAC'09, the ACM Symposium on Applied Computing*, pages 1402–1407. ACM Press, 2009.

[GL98] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers: Boston, 1998.

[HFP09] Jun He, Pierre Flener, and Justin Pearson. Toward an *automaton* constraint for local search. In Yves Deville and Christine Solnon, editors, *Proceedings of LSCS'09, the 6th International Workshop on Local Search Techniques in Constraint Satisfaction*, volume 5 of *Electronic Proceedings in Theoretical Computer Science*, pages 13–25, 2009.

[KS09] Serdar Kadıoğlu and Meinolf Sellmann. Dialectic search. In Ian P. Gent, editor, *Proceedings of CP'09, the 15th International Conference on Principles and Practice of Constraint Programming*, volume 5732 of *Lecture Notes in Computer Science*, pages 486–500. Springer-Verlag, 2009.

[Pra07] Benoit Pralong. Implementation of the *Regular* constraint in Comet, 2007.

[Rég94] Jean-Charles Régin. A filtering algorithm for constraints of difference in CSPs. In Barbara Hayes-Roth and Richard E. Korf, editors, *Proceedings of AAAI'94*, pages 362–367. AAAI Press, 1994.

[Rég96] Jean-Charles Régin. Generalized arc-consistency for global cardinality constraint. In Dan Weld and Bill Clancey, editors, *Proceedings of AAAI'96*, pages 209–215. AAAI Press, 1996.

[VHM05] Pascal Van Hentenryck and Laurent Michel. *Constraint-Based Local Search*. The MIT Press, 2005.

[vHS07] Willem-Jan van Hoeve and Ashish Sabharwal. Two set constraints for modeling and efficiency. In Jimmy Lee and Peter Stuckey, editors, *Proceedings of ModRef'07, the 6th International Workshop on Constraint Modelling and Reformulation*, 2007. Available at `http://www.cse.cuhk.edu.hk/~jlee/cp07Model/`.

[Åg08] Magnus Ågren. *Set Constraints for Local Search*. PhD thesis, Department of Information Technology, Uppsala University, Sweden, 2008. Available at `http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-8373`.

A Comet code for the regular constraint and a test with the nurse scheduling problem

Comet code 4. *The code for the nurse scheduling problem was written by Jun He.*

```
import cotls;

/* Class for deterministic finite automata */

class Dfa{

    dict{int->int}[] automata;
    set{int} finals;
    set{int} alphabet;
    int currentState;
    bool failed;

    Dfa(int nbStates, set{int} _finals){
        automata = new dict{int->int}[1..nbStates]();
        finals = _finals;
        alphabet = new set{int}();
        currentState = 1;
        failed = false;
    }

    void print(ostream os){
        os << "current state : " << currentState << endl;
        os << "final states : " << finals << endl;
        for(int i = 1 ; i <= automata.getSize() ; i++){
            forall(key in automata[i].getKeys()){
                os << i << " -(" << key << ")-> " << automata[i]{key} << endl;
            }
        }
        os << endl;
    }

    int getNbStates(){
        return automata.getSize();
    }

    set{int} getAlphabet(){
        return alphabet;
```

```
}

set{int} getFinals(){
    return finals;
}

dict{int->int} getTransitions(int state){
    return automata[state];
}

void reset(){
    currentState = 1;
}

void addTransition(int state, int letter, int newState){
    automata[state]{letter} = newState;
    if(!alphabet.contains(letter)) alphabet.insert(letter);
}

void removeTransition(int state, int letter){
    automata[state].remove(letter);
}

int costTransition(int state, int letter, int newState){
    if(automata[state]{letter} == newState) return 0;
    forall(k in automata[state].getKeys()){
        if(automata[state]{k} == newState) return 1;
    }
    return System.getMAXINT();
}
void transition(int letter){
    if(!automata[currentState].hasKey(letter)) failed = true;
    else currentState = automata[currentState]{letter};
}

bool isAccepted(){
    return (!failed) && finals.contains(currentState);
}

Dfa layeredGraph(int length){
    int nbS = this.getNbStates();
    set{int} f = this.getFinals();
    set{int} nf = new set{int}();
    forall(i in f)
        nf.insert(i+length*nbS);

    // State of layer i and state j will be state i*nbS+j
    Dfa lg = new Dfa(nbS*(length+1), nf);

    // Forward pass
    set{int} reachable = {1};
    set{int} nextStep = new set{int}();
    for(int i = 0 ; i < length ; i++){
        forall(j in reachable){
            forall(key in automata[j].getKeys()){
                lg.addTransition(i*nbS+j, key, (i+1)*nbS+automata[j]{key});
                nextStep.insert(automata[j]{key});
            }
        }
        reachable = nextStep;
        nextStep = new set{int}();
    }

    // Moonwalk pass
    set{int} coReachable = nf;
    nextStep = new set{int}();
    dict{int->int} t;
    for(int i = length-1 ; i >= 0 ; i--){
        for(int j = 1 ; j <= nbS ; j++){
            t = lg.getTransitions(i*nbS+j);
            forall(key in t.getKeys()){
                if(coReachable.contains(t{key})) nextStep.insert(i*nbS+j);
                else lg.removeTransition(i*nbS+j, key);
            }
        }
        coReachable = nextStep;
        nextStep = new set{int}();
    }
    return lg;
}

int[,,,] layeredGraphArr(int length){
    int nbS = this.getNbStates();
    set{int} f = this.getFinals();
```

```
range alph = alphabet.getLow()..alphabet.getUp();
//set{int} nf = new set{int}();
//forall(i in f)
//    nf.insert(i+length*nbS);

// Cost of the i-th transition from state a to state b with letter l will be :
// lg[i,a,l,b]
int lg[1..length, 1..nbS, alph, 1..nbS] = System.getMAXINT();

// Forward pass
set{int} reachable = {1};
set{int} nextStep = new set{int}();
forall(i in 1..length){
    forall(j in reachable, key in automata[j].getKeys()){
      lg[i,j,key,automata[j]{key}] = 0;
        nextStep.insert(automata[j]{key});
    }
    reachable = nextStep;
    nextStep = new set{int}();
}

// Moonwalk pass
set{int} coReachable = f;
nextStep = new set{int}();
for(int i = length ; i > 0 ; i--){
    forall(j in 1..nbS, l in alph, k in 1..nbS){
        if(lg[i,j,l,k] == 0){
            if(!coReachable.contains(k))
                lg[i,j,l,k] = System.getMAXINT();
            else
                nextStep.insert(j);
        }
    }
    coReachable = nextStep;
    nextStep = new set{int}();
}
bool flag = false;
forall(i in 1..length){
    forall(j in 1..nbS, k in 1..nbS){
        forall(l in alph)
            if(lg[i,j,l,k] == 0){
            flag= true;
```

```
                        break;
                    }
                if(flag)
                    forall(l in alph)
                        if(lg[i,j,l,k] == System.getMAXINT())
                            lg[i,j,l,k] = 1;
                flag = false;
            }
        }
        return lg;
    }
}

//////////////////////////////////////////////////////////////////////

class Regular extends UserConstraint<LS>{
    Solver<LS> m;
    var{int}[] a;
    bool posted;
    dict{var{int}->int} map;
    var{int}[] variableViolations;
    var{int}[] d;
    var{int} totalViolations;
    var{bool} isConstraintTrue;
    set{int}[] reachable;
    var{int}[,] distSource;
    var{int}[,] distFinal;
    Dfa automata;
    int[,,,] lga;
    int nbS;
    int nbVar;
    int[] shortestPath;
    set{int} alph;
    set{int} finals;

    Regular(var{int}[] _a, Dfa _automata) : UserConstraint<LS>(_a.getLocalSolver()){
        m = _a.getLocalSolver();
        a = _a;
        automata = _automata;
        lga = automata.layeredGraphArr(a.getSize());
        nbS = automata.getNbStates();
        alph = automata.getAlphabet();
        finals = automata.getFinals();
        posted = false;
        post();
    }

    void post(){
        if(!posted){
            map = new dict{var{int} -> int}();
            forall(i in a.getRange()) map{a[i]} = i;
            nbVar = a.getSize();
            reachable = new set{int}[0..nbVar] = {};
            set{int} reached = new set{int}();
            reachable[0] = {1};
            int j = 0;
            int letter = alph.getLow();

            // reachable[j] contains all reachable state of layer j of lg
            while(reachable[j].getSize() > 0 && j < nbVar){
                forall(state in reachable[j], k in 1..nbS)
                    if(lga[j+1, state, letter, k] != System.getMAXINT())
                        reached.insert(k);
                j++;
                reachable[j] = reached;
                reached = new set{int}();
            }

            distSource = new var{int}[i in 0..nbVar, j in 1..nbS](m);
            distFinal = new var{int}[i in 0..nbVar, j in 1..nbS](m);
            variableViolations = new var{int}[i in 1..nbVar](m);
          d = new var{int}[i in 0..nbVar](m);
            shortestPath = new int[0..nbVar] = 0;
            shortestPath[0] = 1;

            updateDists();

      d[0] := 0;
      forall(i in 1..nbVar)
        d[i] <- min(s in reachable[i])(distSource[i,s]);
      forall(i in 1..nbVar)
            variableViolations[i] <- d[i] - d[i-1];
            totalViolations = new var{int}(m);
            totalViolations <- distFinal[0,1];
            isConstraintTrue = new var{bool}(m);
            isConstraintTrue <- (totalViolations == 0);
            posted = true;
        }
    }

    void updateDists(){
        // distSource[i,j] contains the cost of the shortest path
        // from the source to state j in layer i
        distSource[0,1] := 0;
        forall(i in 2..nbS) distSource[0,i] := System.getMAXINT();
        for(int i = 1 ; i <= nbVar ; i++){
            for(int j = 1 ; j <= nbS ; j++){
                if(reachable[i].contains(j)){
                    distSource[i,j] := min(s in reachable[i-1])(((
                      distSource[i-1,(s-1)%nbS+1] + lga[i,s,a[i],j])<0)
                      ?System.getMAXINT():(distSource[i-1,(s-1)%nbS+1]
                       + lga[i,s,a[i],j]));
                } else distSource[i,j] := 0;//:= System.getMAXINT();
            }
        }

        // distFinal[i,j] contains the cost of the shortest path
        // from state j in layer i to any final state
        for(int i = 1 ; i <= nbS ; i++){
            if(finals.contains(i)) distFinal[nbVar,i] := 0;
            else distFinal[nbVar,i] := 0;//:= System.getMAXINT();
        }
        for(int i = nbVar-1 ; i >= 0 ; i--){
            for(int j = 1 ; j <= nbS ; j++){
                if(reachable[i].contains(j)){
                    distFinal[i,j] := min(s in reachable[i+1])(((
                      distFinal[i+1,(s-1)%nbS+1] + lga[i+1,j,a[i+1],s])<0)
                      ?System.getMAXINT():(distFinal[i+1,(s-1)%nbS+1]
                       + lga[i+1,j,a[i+1],s]));
                } else distFinal[i,j] := 0;//:= System.getMAXINT();
            }
        }

        // shortestPath is one shortest path in the layered graph
        set{int} tmp =  argMin(t in reachable[nbVar])(distSource[nbVar,t]);
        int s = tmp.getLow();
        int prev;
        int m = System.getMAXINT();
        for(int i = nbVar ; i >= 1 ; i--){
            prev = s;
            m = System.getMAXINT();
            forall(j in 1..nbS){
                if(m > distSource[i,j]){
                    if(lga[i,j,a[i],prev] < System.getMAXINT()){
                        m = distSource[i,j];
                        s = j;
                    }
                }
            }
            shortestPath[i] = s;
        }
    }

    Solver<LS> getLocalSolver(){
        return m;
    }

    var{int}[] getVariables(){
        return a;
    }

    var{bool} isTrue(){
        return isConstraintTrue;
    }

    var{int} violations(){
        return totalViolations;
    }

    var{int} violations(var{int} x){
        //cout << variableViolations[map{x}] << endl;
        return variableViolations[map{x}];
    }

    int getAssignDelta(var{int} x, int d){
        int i = map{x};
        return min(s in reachable[i-1])(min(t in reachable[i])(
          distSource[i-1,(s-1)%nbS+1] + lga[i,s,d,t]
```

```
          + distFinal[i,(t-1)%nbS+1])) - totalViolations;
    }

    int getAssignDelta(var{int}[] tx, int[] td){
        return sum(i in tx.rng()) getAssignDelta(tx[i], td[i]);
    }

    int getSwapDelta(var{int} x, var{int} y){
        return getAssignDelta(x,y) + getAssignDelta(y,x);
    }

    void swapAssignment(var{int} x, var{int} y){
        int tmp = x;
        x := y;
        y := tmp;
    }
}
/////////////////////////////////////////////////////////////////

int totalRun = 20;
int timeOutMS = 10000;
/*string paraFileName = "expParamters.txt";
ifstream paraFile(paraFileName);
string tempReadLine = paraFile.getLine();
while(!tempReadLine.equals(""))
{
  int index = tempReadLine.find("#");
    if(index<0)
    {
      index = tempReadLine.find("totalRun");
        if(index>=0)
        {
          index = tempReadLine.find("=",index);
      tempReadLine = tempReadLine.suffix(index+1);
            totalRun = tempReadLine.toInt();
        }
    else
    {
      index = tempReadLine.find("timeout");
      if(index>=0)
          {
            index = tempReadLine.find("=",index);
        tempReadLine = tempReadLine.suffix(index+1);
              timeOutMS = tempReadLine.toInt();
          }
    }
    }
    tempReadLine = paraFile.getLine();
}
*/

int[] instanceArray[1..2];
instanceArray[1] = new int[1..4];
instanceArray[1][1] = 1;
instanceArray[1][2] = 2;
instanceArray[1][3] = 3;
instanceArray[1][4] = 4;
instanceArray[2] = new int[1..6];
instanceArray[2][1] = 1;
instanceArray[2][2] = 2;
instanceArray[2][3] = 3;
instanceArray[2][4] = 4;
instanceArray[2][5] = 1;
instanceArray[2][6] = 4;
int instanceSizeArray[1..2] = [4,6];

string instanceNameArray[1..2] = ["\"(1d,1e,1n,1x)\"*","\"(2d,1e,1n,2x)\"*"];
int weekNumArray[1..2,1..8] = [[4,8,12,16,20,24,28,32],[6,12,18,24,30,36,42,48]];
int restartArray[1..2,1..8] = [[80*2,160*2,240*2,320*2,400*2,480*2,560*2,640*2]
                                ,[80,160,240,320,400,480,560,640]];

int runtimeLimit[ii in 1..8] = 60000*ii;

ofstream mFile("loic.csv");
mFile << "Ins,SatInLS,perSAT,minTime,maxTime,avgTime,deltTime,modTime,minIter
           ,maxIter,avgIter,deltIter" << endl;

totalRun = 1;

forall(hhxx in 2..2,hhww in 1..1)
{

int weekNum = weekNumArray[hhxx,hhww];
int restartIter = restartArray[hhxx,hhww];
```

```
string instanceName = instanceNameArray[hhxx];
int instanceSize = instanceSizeArray[hhxx];


Solver<LS> ls();
ConstraintSystem<LS> S(ls);

int t0 = System.getCPUTime();  //start time

int varNum = 7*weekNum;
range valDomain = 1..4;  // 1: day; 2: evening; 3: night; 4: dayoff

var{int} disVariable[1..varNum](ls,valDomain);
int[] assignment = instanceArray[hhxx];
RandomPermutation perm(1..instanceSize);

forall(ww in 1..7)
{
  forall(ee in 1..weekNum)
  {
    int tempt = ee%instanceSize;
    if(tempt==0)  tempt = instanceSize;

    //disVariable[(ee-1)*7+ww] := assignment[perm.get()];
    disVariable[(ee-1)*7+ww] := assignment[tempt];

    if(tempt==instanceSize)  perm.reset();
  }
}

int startState = 1;
set{int} acceptedStates = {3,4,5,6,7,8,10,11,12,13,14,15,17,18,19,20,21,
                                22,24,25,26,27,28,29};
int nbStates = 29;

Dfa dfa = new Dfa(nbStates,acceptedStates);

forall(xx in 1..7)
{
  dfa.addTransition(xx,4,xx+1);

  if(xx==1) dfa.addTransition(xx,3,xx+8);
  else dfa.addTransition(xx+7,3,xx+8);

  if(xx==1) dfa.addTransition(xx,2,xx+15);
  else dfa.addTransition(xx+14,2,xx+15);

  if(xx==1) dfa.addTransition(xx,1,xx+22);
  else dfa.addTransition(xx+21,1,xx+22);
}

forall(xx in 3..8)
{
  dfa.addTransition(xx,3,9);
  dfa.addTransition(xx,2,16);
  dfa.addTransition(xx,1,23);

  forall(yy in 1..3)
    dfa.addTransition(xx+7*yy,4,2);
}

Regular nurseRotation = new Regular(disVariable,dfa);

S.post(nurseRotation);
S.post(disVariable[1]!=disVariable[varNum]);
S.post((disVariable[varNum]==4)||(disVariable[1]==4));

ls.close();

int t1 = System.getCPUTime();  //model building time

Counter it(ls,0);
var{int} violations = S.violations();

Solution bestsol = new Solution(ls);
UniformDistribution d(1..3);

int best = violations;

Counter stable(ls,0);

//////////////////////////---------restart
whenever it@changes(int o, int n)
{
  if(it % restartIter == 0)
  {
    forall(ww in 1..7)
    {
      forall(ee in 1..weekNum)
      {
        int tempt = ee%instanceSize;
        if(tempt==0)  tempt = instanceSize;

        //disVariable[(ee-1)*7+ww] := assignment[perm.get()];
        disVariable[(ee-1)*7+ww] := assignment[tempt];

        if(tempt==instanceSize)  perm.reset();
      }
    }

    nurseRotation.updateDists();

    stable := 0;
  }
}

int tew = 0;
int t2 = t0;

int minIteration = 400000;
int maxIteration = 0;
int detailIteration[1..totalRun];
float deltIteration = 0;
int averageIteration;

int modelStateTime;
int minTime = 400000;
int maxTime = 0;
int detailTime[1..totalRun];
float deltTime = 0;
int averageTime;
int satCount = 0;
bool hasFoundSol = false;

forall(tt in 1..totalRun)
{
```

```
if(tt!=1)
{
  tew = tew + it;
  it := 0;
  forall(ww in 1..7)
  {
    forall(ee in 1..weekNum)
    {
      int tempt = ee%instanceSize;
      if(tempt==0)  tempt = instanceSize;

      //disVariable[(ee-1)*7+ww] := assignment[perm.get()];
      disVariable[(ee-1)*7+ww] := assignment[tempt];

      if(tempt==instanceSize)  perm.reset();
    }
  }

  nurseRotation.updateDists();
  bestsol = new Solution(ls);
  best = violations;
}

int tabu[1..varNum,1..varNum] = -1;
int maxTimeLimit = timeOutMS+t2;

while(violations>0)
{
  cout << violations << endl;
  int tt = System.getCPUTime();
      if(tt>=maxTimeLimit) break;

  select(xx in 1..varNum : S.violations(disVariable[xx])>0)
  {
    int tempt = xx%7;
    if(tempt==0) tempt = 7;
    selectMin(tt in 0..(weekNum-1), yy = tt*7+tempt :
                        disVariable[xx] != disVariable[yy],
          nv = S.getSwapDelta(disVariable[xx],disVariable[yy]) :
          (tabu[xx,yy]<=it || violations+nv < best))(nv)
    {
      disVariable[xx] :=: disVariable[yy];
```

```
        nurseRotation.updateDists();

        tabu[xx,yy] = it + max(violations,6);
        tabu[yy,xx] = tabu[xx,yy];

        if(violations < best)
        {
          bestsol = new Solution(ls);
          best = violations;
          stable := 0;
        }
      }
    }

    it++;
    stable++;
  }

  if(minIteration > it)   minIteration = it;
  if(maxIteration < it)   maxIteration = it;
  detailIteration[tt] = it;

  int tempTime;
  if(tt==1){
    t2 = System.getCPUTime();
    tempTime = t2-t1;
  }
  else{
    int tempRe = t2;
    t2 = System.getCPUTime();
    tempTime = t2-tempRe;
  }
  if(minTime > tempTime)   minTime = tempTime;
  if(maxTime < tempTime)   maxTime = tempTime;
  detailTime[tt] = tempTime;

  if(violations==0){  satCount++; hasFoundSol = true;   }
}

  tew = tew + it;
  averageIteration = tew/totalRun;
  averageTime = (t2-t1)/totalRun;
  forall(tt in 1..totalRun)
  {
    deltIteration = deltIteration + (detailIteration[tt]-averageIteration)^2;
    deltTime = deltTime + (detailTime[tt]-averageTime)^2;
  }

  deltIteration = deltIteration/totalRun;
  deltIteration = sqrt(deltIteration);

  deltTime = deltTime/totalRun;
  deltTime = sqrt(deltTime);

  bestsol.restore();

    cout << totalRun << endl;

cout << "Min Iteration: " << minIteration << endl;
cout << "Max Iteration: " << maxIteration << endl;
cout << "Average Iteration: " << averageIteration << endl;
cout << "Standard Deviation on Iteration: " << deltIteration << endl;
cout << "Violations: " << best << endl;
cout << "time to state model: " << t1-t0 << endl;
cout << "Min Optimazation Time: " << minTime << endl;
cout << "Max Optimazation Time: " << maxTime << endl;
cout << "Average Optimazation Time: " << averageTime << endl;
cout << "Standard Deviation on Optimazation Time: " << deltTime << endl;
cout << "time (total) : " << t1-t0+averageTime << endl;
cout << " The best solution found so far:" << endl;
cout << " Week  Mon  Tue  Wed  Thu  Fri  Sat  Sun " << endl;
forall(xx in 1..weekNum)
{
  cout << "  " << xx << "  ";
  if(xx<10) cout << " ";
  forall(yy in 1..7)
  {
    if(disVariable[(xx-1)*7+yy]==1) cout << "  " << "D" << "  ";
    if(disVariable[(xx-1)*7+yy]==2) cout << "  " << "E" << "  ";
    if(disVariable[(xx-1)*7+yy]==3) cout << "  " << "N" << "  ";
    if(disVariable[(xx-1)*7+yy]==4) cout << "  " << "X" << "  ";
  }
  cout << endl;
}

string satLS;
if(hasFoundSol)  satLS = "SAT";
else  satLS = "UNKNOWN";

float perSAT = (float)satCount*100/totalRun;

mFile << instanceName << hhww << "," << satLS << "," << perSAT << ","
    << (t1-t0+minTime)/1000.0 << "," << (t1-t0+maxTime)/1000.0 << ","
    << (t1-t0+averageTime)/1000.0 << "," << (deltTime)/1000.0 << ","
    << (t1-t0)/1000.0 << "," << minIteration << "," << maxIteration << ","
    << averageIteration << "," << deltIteration << endl;

}
```

B Comet code for the set constraint system

B.1 The core system

Comet code 5. *The SetConstraint interface.*

```
interface SetConstraint{

    var{boolean} isTrue();
    void post();
    var{int} violations();
    var{int} violations(var{set{int}} s);
    var{set{int}}[] getVariables();
    // add v to s
    int getAdd(var{set{int}} s, int v);
    // drops u from s
    int getDrop(var{set{int}} s, int u);
    // replaces u in s by v
    int getFlip(var{set{int}} s, int u, int v);
    // transfers u from s to t
    int getTransfer(var{set{int}} s, int u, var{set{int}} t);
    // swaps u from s with v from t
    int getSwap(var{set{int}} s, int u, int v, var{set{int}} t);

}
```

```
class DummySetConstraint implements SetConstraint{

    DummySetConstraint(){}

    var{boolean} isTrue(){
        Solver<LS> m();
        var{bool} f(m) := true;
        return f;
    }

    void post(){}

    var{int} violations(){
        Solver<LS> m();
        var{int} i(m) := 0;
        return i;
    }

    var{int} violations(var{set{int}} s){
        Solver<LS> m();
        var{int} i(m) := 0;
        return i;
    }

    var{set{int}}[] getVariables(){
        Solver<LS> m();
        var{set{int}} s(m) := new set{int}();
        var{set{int}} t[0..0] = [s];
        return t;
    }

    int getAdd(var{set{int}} s, int v){ return 0; }
    int getDrop(var{set{int}} s, int u){ return 0; }
    int getFlip(var{set{int}} s, int u, int v){ return 0; }
    int getTransfer(var{set{int}} s, int u, var{set{int}} t){ return 0; }
    int getSwap(var{set{int}} s, int u, int v, var{set{int}} t){ return 0; }

}
```

Comet code 6. *The SetConstraintSystem class.*

```
import cotls;
include "SetConstraint";
```

```
class SetConstraintSystem implements SetConstraint{

    Solver<LS> m;
    // this is the main variable for the SetConstraintSystem.
    SetConstraint[] constraints;
    int[] weights;
    int currentIndex;
    int nbConstraints;

    SetConstraintSystem(Solver<LS> _m, int _nbConstraints){
        m = _m;
        nbConstraints = _nbConstraints;
        currentIndex = 0;
        constraints = all(i in 1..nbConstraints)(SetConstraint)new DummySetConstraint();
        weights= all(i in 1..nbConstraints) 1;
    }

    void post(SetConstraint c){
        if(currentIndex > nbConstraints)
            throw ExecutionError("/!\\ Too many constraints /!\\ ");
        currentIndex++;
        constraints[currentIndex] = c;
    }

  // to post a weighted constraint
    void post(SetConstraint c, int w){
        if(currentIndex > nbConstraints)
            throw ExecutionError("/!\\ Too many constraints /!\\ ");
        currentIndex++;
        weights[currentIndex] = w;
        constraints[currentIndex] = c;
    }

    var{boolean} isTrue(){
        var{bool}[] t = all(i in 1..currentIndex) constraints[i].isTrue();
        var{bool} b(m) <- and(j in 1..currentIndex) t[j];
        return b;
    }

    void post(){} //needed to fully implement SetConstraint
    var{int} violations(){
        var{int}[] t = all(i in 1..currentIndex) constraints[i].violations();
        var{int} v(m) <- sum(j in 1..currentIndex) t[j];
        return v;
    }

    var{int} violations(var{set{int}} s){
        var{int}[] t = all(i in 1..currentIndex) constraints[i].violations(s);
        var{int} v(m) <- sum(j in 1..currentIndex) t[j];
        return v;
    }

    var{set{int}}[] getVariables(){ // TODO if needed
        Solver<LS> m();
        var{set{int}} s(m) := new set{int}();
        var{set{int}} t[0..0] = [s];
        return t;
    }

    int getAdd(var{set{int}} s, int v){
        return sum(i in 1..currentIndex) weights[i]*constraints[i].getAdd(s, v);
    }

    int getDrop(var{set{int}} s, int u){
        return sum(i in 1..currentIndex) weights[i]*constraints[i].getDrop(s, u);
    }

    int getFlip(var{set{int}} s, int u, int v){
        return sum(i in 1..currentIndex) weights[i]*constraints[i].getFlip(s, u, v);
    }

    int getTransfer(var{set{int}} s, int u, var{set{int}} t){
        return sum(i in 1..currentIndex) weights[i]*constraints[i].getTransfer(s, u, t);
    }

    int getSwap(var{set{int}} s, int u, int v, var{set{int}} t){
        return sum(i in 1..currentIndex) weights[i]*constraints[i].getSwap(s, u, v, t);
    }

}
```

B.2 The set constraints

Comet code 7. *The* ***InSet*** *constraint.*

```
import cotls;
include "SetConstraint";

class InSet extends UserConstraint<LS> implements SetConstraint{
    Solver<LS> m;
    var{set{int}} s;
    int a;
    bool posted;
    var{int} totalViolations;
    var{bool} isConstraintTrue;

    InSet(var{set{int}} _s, int _a) : UserConstraint<LS>(_s.getLocalSolver()){
        m = _s.getLocalSolver();
        s = _s;
        a = _a;
        posted = false;
        post();
    }

    void post(){
        if(!posted){
            isConstraintTrue = new var{bool}(m);
            isConstraintTrue <- member(a,s);
            totalViolations = new var{int}(m);
            totalViolations <- isConstraintTrue?0:1;
            posted = true;
        }

    }

    Solver<LS> getLocalSolver(){
        return m;
    }

    var{bool} isTrue(){
        return isConstraintTrue;
    }

    var{int} violations(){
```

```
    return totalViolations;
}

var{int} violations(var{set{int}} x){
    if(s === x)
        return totalViolations;
    var{int} zero(m) := 0;
    return zero;
}

var{set{int}}[] getVariables(){
    return all(i in 1..1) s;
}

int getAdd(var{set{int}} S, int v){
    if(!isConstraintTrue && (S === s) && (v == a))
        return -1;
    else
        return 0;
}

int getDrop(var{set{int}} S, int u){
    if(isConstraintTrue && (S === s) && (u == a))
        return 1;
    else
        return 0;
}

int getFlip(var{set{int}} S, int u, int v){
    if(isConstraintTrue && (S === s) && (u == a) && (v != u))
        return 1;
    else if(!isConstraintTrue && (S === s) && (v == a))
        return -1;
    return 0;
}

int getTransfer(var{set{int}} S, int u, var{set{int}} T){
    if((S === s) && (T !== s)){
        if(isConstraintTrue && (u == a))
            return 1;
    } else if((S !== s) && (T === s)){
        if(!isConstraintTrue && (u == a))
```

```
                return -1;
        }
        return 0;
    }

    int getSwap(var{set{int}} S, int u, int v, var{set{int}} T){
        if((S === s) && (T !== s)){
            if(isConstraintTrue && (u == a) && (v != a))
                return 1;
            else if(!isConstraintTrue && (v == a))
                return -1;
        } else if((S !== s) && (T === s)){
            if(isConstraintTrue && (v == a) && (u != a))
                return 1;
            else if(!isConstraintTrue && (u == a))
                return -1;
        }
        return 0;
    }

}
```

Comet code 8. *The* `NotInSet` *constraint.*

```
import cotls;
include "SetConstraint";

class NotInSet extends UserConstraint<LS> implements SetConstraint{
    Solver<LS> m;
    var{set{int}} s;
    int a;
    bool posted;
    var{int} totalViolations;
    var{bool} isConstraintTrue;

    NotInSet(var{set{int}} _s, int _a) : UserConstraint<LS>(_s.getLocalSolver()){
        m = _s.getLocalSolver();
        s = _s;
        a = _a;
        posted = false;
        post();
    }
```

```
void post(){
    if(!posted){
        isConstraintTrue = new var{bool}(m);
        isConstraintTrue <- !member(a,s);
        totalViolations = new var{int}(m);
        totalViolations <- isConstraintTrue?0:1;
        posted = true;
    }

}

Solver<LS> getLocalSolver(){
    return m;
}

var{set{int}} getVariables(){
    return s;
}

var{bool} isTrue(){
    return isConstraintTrue;
}

var{int} violations(){
    return totalViolations;
}

var{int} violations(var{set{int}} x){
    if(s === x)
        return totalViolations;
    var{int} zero(m) := 0;
    return zero;
}

var{set{int}}[] getVariables(){
    return all(i in 1..1) s;
}

int getAdd(var{set{int}} S, int v){
    if(isConstraintTrue && (S === s) && (v == a))
        return 1;
    else
```

```
        return 0;
}

int getDrop(var{set{int}} S, int u){
    if(!isConstraintTrue && (S === s) && (u == a))
        return -1;
    else
        return 0;
}

int getFlip(var{set{int}} S, int u, int v){
    if(!isConstraintTrue && (S === s) && (u == a) && (v != u))
        return -1;
    else if(isConstraintTrue && (S === s) && (v == a))
        return 1;
    return 0;
}

int getTransfer(var{set{int}} S, int u, var{set{int}} T){
    if((S === s) && (T !== s)){
        if(!isConstraintTrue && (u == a))
            return -1;
    } else if((S !== s) && (T === s)){
        if(isConstraintTrue && (u == a))
            return 1;
    }
    return 0;
}

int getSwap(var{set{int}} S, int u, int v, var{set{int}} T){
    if((S === s) && (T !== s)){
        if(!isConstraintTrue && (u == a) && (v != a))
            return -1;
        else if(isConstraintTrue && (v == a))
            return 1;
    } else if((S !== s) && (T === s)){
        if(!isConstraintTrue && (v == a) && (u != a))
            return -1;
        else if(isConstraintTrue && (u == a))
            return 1;
    }
    return 0;
    }

}
```

Comet code 9. *The* ***Partition*** *constraint.*

```
import cotls;
include "SetConstraint";

class Partition extends UserConstraint<LS> implements SetConstraint{
    Solver<LS> m;
    var{set{int}} s;
    var{set{int}}[] parts;
    var{int}[] nbOcc;
    bool posted;
    var{int} totalViolations;
    var{bool} isConstraintTrue;

    Partition(var{set{int}} _s, var{set{int}}[] _parts) :
             UserConstraint<LS>(_s.getLocalSolver()){
        m = _s.getLocalSolver();
        s = _s;
        parts = _parts;
        posted = false;
        post();
    }

    void post(){
        if(!posted){
            totalViolations = new var{int}(m);
            totalViolations := violations();
            isConstraintTrue = new var{bool}(m);
            isConstraintTrue <- (totalViolations == 0)?true:false;
            posted = true;
        }

    }

    Solver<LS> getLocalSolver(){
        return m;
    }

    var{bool} isTrue(){
```

```
    return isConstraintTrue;
}

var{int} violations(){
    set{int} t = s;
    set{int} u;
    int v = 0;
    nbOcc = new var{int}[i in 0..s.getSize()-1](m) <-
           sum(j in parts.getRange()) (member(t.atRank(i), parts[j]));
    totalViolations := sum(i in nbOcc.getRange()) (abs(1-nbOcc[i]));
    return totalViolations;
}

var{int} violations(var{set{int}} x){
    var{int} v(m) := 0;
    if(s === x)
        return v;
    forall(i in parts.getRange()){
        if(parts[i] === x){
            set{int} t = s;
            set{int} u = parts[i];
            v := card(t\u);
            forall(e in u)
                v := v + nbOcc[t.getRank(e)] - 1;
            return v;
        }
    }
    return v;
}

var{set{int}}[] getVariables(){
    return all(i in 0..parts.getSize()) ((i == parts.getSize())?parts[i]:s);
}

int getAdd(var{set{int}} S, int v){
    if(S === s) return 0;
    if(!member(v, s)) return 0;
    forall(i in parts.getRange()){
        if(parts[i] === S){
            set{int} t = s;
            int r = t.getRank(v);
            if(nbOcc[r] == 0) return -1;
```

```
                if(nbOcc[r] > 1) return 1;
            }
        }
        return 0;
    }

    int getDrop(var{set{int}} S, int u){
        if((S === s) || !member(u, s)) return 0;
        forall(i in parts.getRange()){
            if((parts[i] === S) && member(u, parts[i])){
                set{int} t = s;
                int r = t.getRank(u);
                if(nbOcc[r] == 1) return 1;
                if(nbOcc[r] > 1) return -1;
            }
        }
        return 0;
    }

    int getFlip(var{set{int}} S, int u, int v){
        return 0;
    }

    int getTransfer(var{set{int}} S, int u, var{set{int}} T){
        return 0;
    }

    int getSwap(var{set{int}} S, int u, int v, var{set{int}} T){
        return 0;
    }

}
```

Comet code 10. *The* `SumFree` *constraint.*

```
import cotls;
include "SetConstraint";

class SumFree extends UserConstraint<LS> implements SetConstraint{
    Solver<LS> m;
    var{set{int}} s;
    bool posted;
    var{int} totalViolations;
```

```
var{bool} isConstraintTrue;

SumFree(var{set{int}} _s) : UserConstraint<LS>(_s.getLocalSolver()){
    m = _s.getLocalSolver();
    s = _s;
    posted = false;
    post();
}

void post(){
    if(!posted){
        totalViolations = new var{int}(m);
        totalViolations := violations();
        isConstraintTrue = new var{bool}(m);
        isConstraintTrue <- (totalViolations == 0)?true:false;
        posted = true;
    }

}

Solver<LS> getLocalSolver(){
    return m;
}

var{bool} isTrue(){
    return isConstraintTrue;
}

var{int} violations(){
    set{int} t = s;
    int l = t.getSize();
    int v = 0;
    forall(i in 0..l-2){
        forall(j in i..l-1){
            if(t.contains(t.atRank(i) + t.atRank(j)))
                v++;
        }
    }
    totalViolations := v;
    return totalViolations;
}
```

```
  var{int} violations(var{set{int}} x){
      if(s === x)
          return violations();
      var{int} zero(m) := 0;
      return zero;
  }

  var{set{int}}[] getVariables(){
      return all(i in 1..1) s;
  }

// all the following operations are inefficient.
  int getAdd(var{set{int}} S, int v){
      if(!(S === s)) return 0;
      int t = violations();
      s.insert(v);
      t = violations() - t;
      s.delete(v);
      return t;
  }

int getDrop(var{set{int}} S, int u){
      if((S === s) && member(u,s)){
          int t = violations();
          s.delete(u);
          t = violations() - t;
          s.insert(u);
          return t;
      }
      return 0;
  }

  int getFlip(var{set{int}} S, int u, int v){
      return 0;
  }

  int getTransfer(var{set{int}} S, int u, var{set{int}} T){
      if((S === s) && member(u,s)){
          int t = violations();
          s.delete(u);
          t = violations() - t;
          s.insert(u);
```

```
            return t;
        }
        if((S === s) && !member(u,s)) return System.getMAXINT();
        if(T === s){
            int t = violations();
            s.insert(u);
            t = violations() - t;
            s.delete(u);
            return t;
        }
        return 0;
    }

    int getSwap(var{set{int}} S, int u, int v, var{set{int}} T){
        return 0;
    }

}
```

Comet code 11. *The* ***AtMost1*** *constraint.*

```
import cotls;
include "SetConstraint";

class AtMost1 extends UserConstraint<LS> implements SetConstraint{
    Solver<LS> m;
    var{set{int}} s1;
    var{set{int}} s2;
    bool posted;
    var{int} totalViolations;
    var{int} cardInter;
    var{bool} isConstraintTrue;

    AtMost1(var{set{int}} _s1, var{set{int}} _s2) :
            UserConstraint<LS>(_s1.getLocalSolver()){
        m = _s1.getLocalSolver();
        s1 = _s1;
        s2 = _s2;
        posted = false;
        post();
    }

    void post(){
```

```
    if(!posted){
        cardInter = new var{int}(m);
  cardInter <- card(s1 inter s2);
        totalViolations = new var{int}(m);
        totalViolations <- max(1, cardInter)-1 ;
        isConstraintTrue = new var{bool}(m);
        isConstraintTrue <- card(s1 inter s2) <= 1;
        posted = true;
    }

}

Solver<LS> getLocalSolver(){
    return m;
}

var{bool} isTrue(){
    return isConstraintTrue;
}

var{int} violations(){
    return totalViolations;
}

var{int} violations(var{set{int}} x){
    if(s1 === x || s2 === x)
        return totalViolations;
    var{int} zero(m) := 0;
    return zero;
}

var{set{int}}[] getVariables(){
    var{set{int}} ss[0..1] = [s1, s2];
    return ss;
}

int getAdd(var{set{int}} S, int v){
    if(cardInter == 0)
        return 0;
    if((S === s1)  && member(v, s2))
        return 1;
    else if((S === s2) && member(v, s1))
```

```
            return 1;
        return 0;
    }

    int getDrop(var{set{int}} S, int u){
        if(cardInter <= 1)
            return 0;
        if(((S === s1) || (S === s2)) && member(u, s1) && member(u, s2))
            return -1;
        return 0;
    }

    int getFlip(var{set{int}} S, int u, int v){
        if((S === s1) || (S === s2))
            return getDrop(S, u) + getAdd(S, v);
        return 0;
    }

    int getTransfer(var{set{int}} S, int u, var{set{int}} T){
        if(((S === s1) && (T === s2)) || ((S === s2) && (T === s1)))
            return 0;
        if((S === s1) || (S === s2))
            return getDrop(S, u);
        if((T === s1) || (T === s2))
            return getAdd(T, u);
        return 0;
    }

    int getSwap(var{set{int}} S, int u, int v, var{set{int}} T){
        if(((S === s1) && (T === s2)) || ((S === s2) && (T === s1)))
            return 0;
        if((S === s1) || (S === s2))
            return getDrop(S, u) + getAdd(S, v);
        if((T === s1) || (T === s2))
            return getAdd(T, u) + getDrop(T, v);
        return 0;
    }

}
```

C Comet code for the experimentations

Comet code 12. *Code for the integer model of the social golfer problem. This code is directly taken from* Comet *documentation.*

```
import cotls;

class Meet implements Invariant<LS> {
  Solver<LS> m;
  range Weeks;
  range Groups;
  range Slots;
  range Golfers;
  var{int}[,,] golfer;
  var{int}[,] meetings;
  dict{var{int}->Position} position;
  Solver<LS> getLocalSolver() { return m; }
  var{int}[,] getMeetings() { return meetings; }
  dict{var{int}->Position} getPosition() { return position; }
  Meet(Solver<LS> _m, range _Weeks,range _Groups,range _Slots,
         range _Golfers, var{int}[,,] _golfer);
  void post(InvariantPlanner<LS> planner);
  void initPropagation();
  void propagateInt(bool b,var{int} v);
  void propagateInsertIntSet(bool b,var{set{int}} s,int value) {}
  void propagateRemoveIntSet(bool b,var{set{int}} s,int value) {}
  void propagateFloat(bool b, var{float} v) {}
}
Meet::Meet(Solver<LS> _m, range _Weeks,range _Groups,range _Slots, range _Golfers,
    var{int}[,,] _golfer) {
  m = _m;
  Weeks = _Weeks;
  Groups = _Groups;
  Slots = _Slots;
  Golfers = _Golfers;
  golfer = _golfer;
  meetings = new var{int} [Golfers,Golfers](m) := 0;
  position = new dict{var{int}->Position}();
  forall (w in Weeks,g in Groups,s in Slots)
    position{golfer[w,g,s]} = new Position(w,g,s);
}
void Meet::post(InvariantPlanner<LS> planner) {
  forall (w in Weeks,g in Groups,s in Slots)
```

```
    planner.addSource(golfer[w,g,s]);
  forall (p in Golfers,q in Golfers)
    planner.addTarget(meetings[p,q]);
}
void Meet::initPropagation() {
  forall (w in Weeks,g in Groups)
    forall (s in Slots, t in Slots: s < t) {
      meetings[golfer[w,g,s],golfer[w,g,t]]++;
      meetings[golfer[w,g,t],golfer[w,g,s]]++;
    }
}
void Meet::propagateInt(bool b,var{int} v) {
  Position p
    = position{v};
  int oldGolfer = v.getOld();
  int newGolfer = v;
  forall (s in Slots: s != p.slot) {
    int o = golfer[p.week,p.group,s];
    meetings[oldGolfer,o]--;
    meetings[o,oldGolfer]--;
    meetings[newGolfer,o]++;
    meetings[o,newGolfer]++;
  }
}
```

```
class SocialTournament extends UserConstraint<LS> {
  Solver<LS> m;
  range Weeks;
  range Groups;
  range Slots;
  range Golfers;
    var{int}[,,] golfer;
    var{int}[,] meetings;
    dict{var{int}->Position} position;
  var{int}[,,] varViolations;
    var{int} violationDegree;
    SocialTournament(Solver<LS> _m,range _Weeks,range _Groups,range _Slots,
        range _Golfers, var{int}[,,] _golfer);
  void post();
  var{int} violations(var{int} x);
  var{int} violations() { return violationDegree; }
  int getSwapDelta(var{int} x,var{int} y);
}

SocialTournament::SocialTournament(Solver<LS> _m,
    range _Weeks,range _Groups, range _Slots,
    range _Golfers, var{int}[,,] _golfer) : UserConstraint<LS>(_m) {
  m = _m;
  Weeks = _Weeks;
  Groups = _Groups;
  Slots = _Slots;
  Golfers = _Golfers;
  golfer = _golfer;
}

void SocialTournament::post() {
  Meet meetInvariant(m,Weeks,Groups,Slots,Golfers,golfer);
  m.post(meetInvariant);
  meetings = meetInvariant.getMeetings();
  position = meetInvariant.getPosition();
         varViolations   = new var{int}[w in Weeks,g in Groups,s in Slots](m) <-
          sum(t in  Slots: t != s) (meetings[golfer[w,g,s],golfer[w,g,t]] >= 2);
       violationDegree   = new var{int}(m) <-
       sum (g in   Golfers, h in Golfers: g < h) max(0,meetings[g,h]-1);
}
var{int} SocialTournament::violations(var{int} x) {
  Position p = position{x};
    return varViolations[p.week,p.group,p.slot];
   }
  int SocialTournament::getSwapDelta(var{int} x,var{int} y) {
    Position xp = position{x};
    Position yp = position{y};
    assert(xp.week == yp.week);
    assert(xp.group != yp.group);
      int delta = 0;
    forall (s in Slots: s != yp.slot)
      delta += (meetings[x,golfer[yp.week,yp.group,s]]>= 1)
          - (meetings[y,golfer[yp.week,yp.group,s]]  >= 2);
            forall (s in Slots: s != xp.slot)
            delta += (meetings[y,golfer[xp.week,xp.group,s]]    >= 1)
              - (meetings[x,golfer[xp.week,xp.group,s]]    >= 2);
                return delta;
  }


Solver<LS> m();
range Weeks = 1..System.getArgs()[4].toInt();
range Groups = 1..System.getArgs()[2].toInt();
range Slots = 1..System.getArgs()[3].toInt(); // slots per group
range Golfers = 1..(Groups.getUp() * Slots.getUp());
tuple Position { int week; int group; int slot; }
var{int} golfer[Weeks,Groups,Slots](m,Golfers);
init(Weeks,Golfers,Groups,Slots,golfer);
SocialTournament tourn(m,Weeks,Groups,Slots,Golfers,golfer);
m.post(tourn);
var{int} conflict[w in Weeks,g in Groups,s in Slots] = tourn.violations(golfer[w,g,s]);
var{int} violations = tourn.violations();
m.close();
function void init(range Weeks, range Golfers, range Groups,
    range Slots, var{int}[,,] golfer) {
  forall (w in Weeks) {
    RandomPermutation golferPerm(Golfers);
    forall (g in Groups,s in Slots)
      golfer[w,g,s] := golferPerm.get();
  }
}

int tabu[Weeks,Golfers,Golfers] = -1;
UniformDistribution tabuDistr(4..20);
int best = violations;
int it = 0;
int nonImprovingSteps = 0;
int maxNonImproving = 15000;

int t = System.getCPUTime();

while (violations > 0)
  selectMin(w in Weeks,
      g1 in Groups, s1 in Slots: conflict[w,g1,s1] > 0,
      g2 in Groups: g2 != g1, s2 in Slots,
      delta = tourn.getSwapDelta(golfer[w,g1,s1],golfer[w,g2,s2]):
      tabu[w,golfer[w,g1,s1],golfer[w,g2,s2]] < it
      || violations + delta < best)
      (delta) {
        golfer[w,g1,s1] :=: golfer[w,g2,s2];
        int tabuLength = tabuDistr.get();
        tabu[w,golfer[w,g1,s1],golfer[w,g2,s2]] = it + tabuLength;
        tabu[w,golfer[w,g2,s2],golfer[w,g1,s1]] = it + tabuLength;
        if (violations < best) {
          best = violations;
          nonImprovingSteps = 0;
        }
        else {
          nonImprovingSteps++;
          if (nonImprovingSteps > maxNonImproving) {
            init(Weeks,Golfers,Groups,Slots,golfer);
            best = violations;
            nonImprovingSteps = 0;
          }
        }
        cout << violations << endl << it << endl << endl;
        it++;
      }

int t2 = System.getCPUTime() - t;
cout << t2 << endl;
```

Comet code 13. *Code for the set model of the social golfer problem with tabu search. The tabu search is taken from* Comet *documentation.*

```
import cotls;
include "SetConstraintSystem";
include "AtMost1";

Solver<LS> m;
int nbGroups = System.getArgs()[2].toInt();
int sizeGroup = System.getArgs()[3].toInt();
int nbWeeks = System.getArgs()[4].toInt();
int nbGolfers;
SetConstraintSystem cs;

m = new Solver<LS>();
nbGolfers = nbGroups*sizeGroup;
var{set{int}} schedule[i in 1..nbWeeks, j in 1..nbGroups] = new var{set{int}}(m);

cs = SetConstraintSystem(m, nbGroups*nbWeeks*nbWeeks*nbGroups);
forall(i in 1..nbWeeks, j in 1..nbGroups, k in 1..nbWeeks, l in 1..nbGroups : i < k)
```

```
    cs.post((SetConstraint) AtMost1(schedule[i, j], schedule[k, l]));
m.close();

range Weeks = 1..nbWeeks;
range Groups = 1..nbGroups;
range Slots = 1..sizeGroup;
range Golfers = 1..nbGolfers;

function void init(range Weeks, range Golfers, range Groups,
         int sizeGroup, var{set{int}}[,] schedule){
  int drawn = 0;
  UniformDistribution distrG(Groups);
  forall(w in Weeks){
    forall(golfer in Golfers){
      drawn = distrG.get();
      while(card(schedule[w, drawn]) == sizeGroup)
        drawn = distrG.get();
      schedule[w, drawn].insert(golfer);
    }
  }
}

init(Weeks, Golfers, Groups, sizeGroup, schedule);
var{int} violations = cs.violations();
var{int} conflict[w in Weeks, g in Groups] = cs.violations(schedule[w, g]);

cout << endl << "violations : " << violations << endl;

// the tabu search comes from the comet documentation.
int tabu[Weeks,Golfers,Golfers] = -1;
UniformDistribution tabuDistr(4..20);
int best = violations;
int it = 0;
int nonImprovingSteps = 0;
int maxNonImproving = 15000;
int time = System.getCPUTime();
while (violations > 0)
  selectMin(w in Weeks, // this select is very costly.
      g1 in Groups: conflict[w, g1] > 0,
      g2 in Groups: g2 != g1,
      s1 in Golfers: member(s1, schedule[w, g1]),
      s2 in Golfers: member(s2, schedule[w, g2]),
      delta = cs.getSwap(schedule[w,g1], s1, s2, schedule[w,g2]):
      tabu[w,s1,s2] < it
      || violations + delta < best)
      (delta) {
        schedule[w, g1].delete(s1);
        schedule[w, g2].insert(s1);
            schedule[w, g2].delete(s2);
        schedule[w, g1].insert(s2);
        int tabuLength = tabuDistr.get();
        tabu[w,s1,s2] = it + tabuLength;
        tabu[w,s2,s1] = it + tabuLength;
        if (violations < best) {
          best = violations;
          nonImprovingSteps = 0;
        }
        else {
          nonImprovingSteps++;
          if (nonImprovingSteps > maxNonImproving) {
            init(Weeks,Golfers,Groups,sizeGroup,schedule);
            best = violations;
            nonImprovingSteps = 0;
          }
        }
          cout << endl << "violations : " << violations << endl;
        //cout << "schedule : " << schedule << endl;
        cout << "iterations : " << it << endl;
        it++;
      }

int time2 = System.getCPUTime() - time;
cout << "schedule : " << schedule << endl;
cout << endl << "time : " << time2 << endl;
cout << "iterations : " << it << endl;
```

Comet code 14. *Code for the set model of the social golfer problem with dialectic search:*

```
import cotls;
include "SetConstraintSystem";
include "AtMost1";

// copy s to s2
function void copy(range Weeks, range Groups, var{set{int}}[,] s, var{set{int}}[,] s2){
  set{int} t;
  set{int} t2;
  forall(w in Weeks, g in Groups){
    t = s[w, g];
    t2 = s2[w, g];
    if(t.compare(t2) != 0)
      s2[w, g] := t.copy();
  }
}

// initialize the schedule randomly. Partition and group size are enforced.
function void init(range Weeks, range Golfers, range Groups, int sizeGroup,
                  var{set{int}}[,] schedule, var{set{int}}[,] schedule2){
  int drawn = 0;
  UniformDistribution distrG(Groups);
  forall(w in Weeks){
    forall(golfer in Golfers){
      drawn = distrG.get();
      while(card(schedule[w, drawn]) == sizeGroup)
        drawn = distrG.get();
      schedule[w, drawn].insert(golfer);
      schedule2[w, drawn].insert(golfer);
    }
  }
}

// greedily improves the schedule until a local minimum is reached
function void greedy(SetConstraintSystem cs, range Weeks, range Golfers, range Groups,
             var{set{int}}[,] schedule, var{int}[,] conflict, var{int} violations){
  int delta = -1;
  while (delta < 0){
    delta = 1;
    selectMin(w in Weeks, // this select is very costly.
      g1 in Groups: conflict[w, g1] > 0,
      g2 in Groups: g2 != g1,
      s1 in Golfers: member(s1, schedule[w, g1]),
      s2 in Golfers: member(s2, schedule[w, g2]),
      delta = cs.getSwap(schedule[w,g1], s1, s2, schedule[w,g2]))
      (delta) {
        if(delta >= 0) break;
        schedule[w, g1].delete(s1);
        schedule[w, g2].insert(s1);
            schedule[w, g2].delete(s2);
        schedule[w, g1].insert(s2);
      }
  }
}

// the greedy function, but it applies on both schedules
function void greedyForBoth(SetConstraintSystem cs, range Weeks, range Golfers,
         range Groups, var{set{int}}[,] schedule, var{set{int}}[,] schedule2,
          var{int}[,] conflict, var{int} violations){
  int delta = -1;
  while (delta < 0){
    delta = 1;
    selectMin(w in Weeks, // this select is very costly.
      g1 in Groups: conflict[w, g1] > 0,
      g2 in Groups: g2 != g1,
      s1 in Golfers: member(s1, schedule[w, g1]),
      s2 in Golfers: member(s2, schedule[w, g2]),
      delta = cs.getSwap(schedule[w,g1], s1, s2, schedule[w,g2]))
      (delta) {
        if(delta >= 0) break;
        schedule[w, g1].delete(s1);
        schedule[w, g2].insert(s1);
            schedule[w, g2].delete(s2);
        schedule[w, g1].insert(s2);
        schedule2[w, g1].delete(s1);
        schedule2[w, g2].insert(s1);
            schedule2[w, g2].delete(s2);
        schedule2[w, g1].insert(s2);
      }
  }
}

// main function for the dialectic search. Modify some randomly selected
// variables greedily and then keeps the best schedule along the path of
// the changes. Last thing, it improves the current best solution greedily.
function void modifyAndMerge(SetConstraintSystem cs, SetConstraintSystem cs2,
         range Weeks, range Groups, range Golfers, var{set{int}}[,] schedule,
          var{set{int}}[,] schedule2, var{int} violations, var{int} violations2,
           var{int}[,] conflict, var{int}[,] conflict2){
  int nbModif = 0;
  int w = 0;
  int g1 = 0;
  int g2 = 0;
  int s1 = 0;
  int s2 = 0;
  int best = violations;
  int bestStep = 0;
  UniformDistribution distrW(Weeks);
  UniformDistribution d(1..(Weeks.getUp()+Groups.getUp()));
  //UniformDistribution d(Weeks);

  nbModif = d.get();
  // to remember the path
  int W[1..2*nbModif] = 0;
  int iG[1..2*nbModif] = 0;
  int iS[1..2*nbModif] = 0;
  int dG[1..2*nbModif] = 0;
  int dS[1..2*nbModif] = 0;

  forall(i in 1..nbModif){
    // randomly select variables to change.
    w = distrW.get();
    select(h1 in Groups, h2 in Groups : h1 != h2){
      g1 = h1;
      g2 = h2;
    }
    // for these variables, select the best move and remember it.
    selectMin(s1 in schedule2[w, g1], s2 in schedule2[w, g2],
      delta = cs2.getSwap(schedule2[w,g1], s1, s2, schedule2[w,g2])) (delta){
      W[i] = w;
      schedule2[w, g1].delete(s1);
      dG[i*2-1] = g1;
      dS[i*2-1] = s1;
      schedule2[w, g2].insert(s1);
      iG[i*2-1] = g2;
      iS[i*2-1] = s1;
      schedule2[w, g2].delete(s2);
      dG[i*2] = g2;
      dS[i*2] = s2;
      schedule2[w, g1].insert(s2);
      iG[i*2] = g1;
      iS[i*2] = s2;
    }
    if(violations2 < best){
      best = violations2;
      bestStep = i;
    }
    if(best == 0) break;
  }
  // keep the best solution found along the path of the changes.
  greedy(cs2, Weeks, Golfers, Groups, schedule2, conflict2, violations2);
  if(violations2 <= best){
    copy(Weeks, Groups, schedule2, schedule);
  } else {
    forall(step in 1..bestStep){
      schedule[W[step], dG[step*2-1]].delete(dS[step*2-1]);
     schedule[W[step], dG[step*2]].delete(dS[step*2]);
     schedule[W[step], iG[step*2-1]].insert(iS[step*2-1]);
     schedule[W[step], iG[step*2]].insert(iS[step*2]);
   }
   greedy(cs, Weeks, Golfers, Groups, schedule, conflict, violations);
   copy(Weeks, Groups, schedule, schedule2);
  }
}

Solver<LS> m;
int nbGroups = System.getArgs()[2].toInt();
int sizeGroup = System.getArgs()[3].toInt();
int nbWeeks = System.getArgs()[4].toInt();
int nbGolfers;
// we will need two full model to enable the use of the "modify" part of the search.
SetConstraintSystem cs;
SetConstraintSystem cs2;

m = new Solver<LS>();
nbGolfers = nbGroups*sizeGroup;
var{set{int}} schedule[i in 1..nbWeeks, j in 1..nbGroups] = new var{set{int}}(m);
var{set{int}} schedule2[i in 1..nbWeeks, j in 1..nbGroups] = new var{set{int}}(m);

cs = SetConstraintSystem(m, nbGroups*nbWeeks*nbWeeks*nbGroups);
cs2 = SetConstraintSystem(m, nbGroups*nbWeeks*nbWeeks*nbGroups);
forall(i in 1..nbWeeks, j in 1..nbGroups, k in 1..nbWeeks, l in 1..nbGroups : i < k){
    cs.post((SetConstraint) AtMost1(schedule[i, j], schedule[k, l]));
    cs2.post((SetConstraint) AtMost1(schedule2[i, j], schedule2[k, l]));
}
m.close();
```

```
range Weeks = 1..nbWeeks;
range Groups = 1..nbGroups;
range Slots = 1..sizeGroup;
range Golfers = 1..nbGolfers;

init(Weeks, Golfers, Groups, sizeGroup, schedule, schedule2);

// get the invariants computed once and for all.
var{int} violations = cs.violations();
var{int} violations2 = cs2.violations();
var{int} conflict[w in Weeks, g in Groups] = cs.violations(schedule[w, g]);
var{int} conflict2[w in Weeks, g in Groups] = cs2.violations(schedule2[w, g]);

cout << endl << "violations : " << violations << endl;

int it = 0;
int time = System.getCPUTime();

// let the search begin !
greedyForBoth(cs, Weeks, Golfers, Groups, schedule, schedule2, conflict, violations);
while (violations > 0){
  modifyAndMerge(cs, cs2, Weeks, Groups, Golfers, schedule, schedule2,
                   violations, violations2, conflict, conflict2);
  it++;
  cout << "violations : " << violations << ", iterations : " << it << endl;
}

int time2 = System.getCPUTime() - time;
cout << "schedule : " << schedule << endl;
cout << endl << "time : " << time2 << endl;
cout << "iterations : " << it << endl;
```