The University of York Department of Computer Science

Staff rota timetabling literature review

28/12/2013

1. **Literature Review**

Although this project is to use Jacop java constraint programming technique to solve the nurse roster timetabling problem, we will review other application areas in the staff rota timetabling area and the methods applied to the roster problems. Then we will look at a case study about staff rota timetabling.

* 1. **The Application Areas**

The staff rota automation is used in a lot of areas in the world now. And a lot of papers discussed the application of staff rota automation in many areas. In the following part we will look at a few papers for staff rota automation in different areas.

One important application area is the nurse rostering. Nurse rostering is a multi-objective problem. A systematic approach for nurse allocation is needed to make sure continuous and good level of services and maintains the hard requirements as well as internal policies. The nurse rostering problem becomes complex when factors such as nurse qualifications or license to practice, type of disease and unforeseen accidents are considered.

The personal requirements of the nurses such as leave or work shift preferences add a new dimension to the rostering problem. The requirement to balance all the dimension of the problem makes the scheduling a very complicated and boring manual task. The schedule algorithm should specify the daily shift assignments in a specified time horizon which satisfy the requirements to every nurse. It should be fair for the distribution of work shifts and the schedule should satisfy the requirements of the nurses.

There are various grades of nurses ranging from registered nurse to junior nurse. Some the nurses might be trained to manage certain medical conditions or skilled in certain area such as intensive care[3]. Due to the varied trainings and specializations, certain type of nurses has to be staffed for work shifts requiring those skills[3].

These varied conditions cause manual nurse scheduling to consume a significant amount of time. Even when the schedule has been planned manually, it does not necessarily guarantee the fairness of distribution of work such as the number of night shifts or weekend shifts[3].

While the nurses might have indicated their preferences, the planner might not have taken all these into consideration resulting in poorly designed schedules which has to be modified by the nurses swapping duties or working under undesired conditions. Occasionally, the plans did not attempt to efficiently utilize the manpower properly.[3] So the staff rota timetabling technique is strongly needed in the nurse roster area.

Another application area is railway staff rostering. Staff rostering is a typical problem in the management of large transport companies, especially railway companies and airline companies. Each work schedule is constructed for each staff type(engine men, conductors, catering staff,etc.)[4]. Because the railway net is very large, the staff scheduling problem is very complicated. It will be much more complicated when the staff preference, leave work, staff requirements are considered as described in the above nurse scheduling problem.

Any application area that needs a lot of human resources would need staff roster automation because generating schedule manually is a time-cost, complicated and boring work. And automatic staff rostering is particularly useful for rostering for temporarily employed workers. Sometimes the staffs will be changed every day and the number of staffs is huge so automatic rostering is perfect for this situation.

A paper describes a real world rostering problem for theatre staff at the annual Edinburgh festival. [1] In this festival, nearly eighty members of voluntary staff work over the three weeks. This is not a trivial problem and it currently requires several days worth of manual effort. What makes this project different is the use of voluntary staff, and the festival structure[1]. Most of the rostering solvers focus on paid staff working regular jobs. This regularity, although requiring that rosters be produced all year round, does provide some continuity on a week-by-week (or fortnight-by-fortnight) basis, and does mean that most staff are available each week[1].

In this problem approximately half the staff change each week, and staff availability varies from one week to three. This means that a solution for the first week of the festival can have very little in common with a solution for the second week for example[1]. The other consideration is that all the staff are voluntary, there are two consequences of this. Firstly, there is not the problem of ensuring each member of staff works their contracted hours, however it is important that the rosters are fair, and do not give some staff a lot more work than others. Secondly, because the staff are not being paid, and are using their annual leave to come and help, it is more important to try and keep them happy, than it would be if they were paid employees[1].

The staff rota automation can be applied to a lot of application areas, the common property of the areas is to have a lot of staffs and have a lot of work shifts. In addition, there are a lot of rules about the work shifts and the staffs have their own needs so the dimension of the problems become bigger and bigger with the increase of the requirements. Thus the work to generate a work schedule becomes a time-cost and complicated work. So the automatic technique to generate a work schedule is strongly needed.

* 1. **The methods** 
     1. **Integer Linear Programming**

One of the methods used in the staff rota timetabling is Integer Linear Programming. Linear programming (LP, or linear optimization) is a mathematical method for determining a way to achieve the best outcome (such as maximum profit or lowest cost) in a given [mathematical model](http://en.wikipedia.org/wiki/Mathematical_model) for some list of requirements represented as linear relationships. Linear programming is a specific case of mathematical programming ([mathematical optimization](http://en.wikipedia.org/wiki/Mathematical_optimization)).

More formally, linear programming is a technique for the [optimization](http://en.wikipedia.org/wiki/Mathematical_optimization) of a [linear](http://en.wikipedia.org/wiki/Linear) [objective function](http://en.wikipedia.org/wiki/Objective_function), subject to [linear equality](http://en.wikipedia.org/wiki/Linear_equality)and [linear inequality](http://en.wikipedia.org/wiki/Linear_inequality) [constraints](http://en.wikipedia.org/wiki/Constraint_(mathematics)). Its [feasible region](http://en.wikipedia.org/wiki/Feasible_region) is a [convex polyhedron](http://en.wikipedia.org/wiki/Convex_polyhedron), which is a set defined as the intersection of finitely many [half spaces](http://en.wikipedia.org/wiki/Half-space_(geometry)), each of which is defined by a linear inequality. Its objective function is a [real](http://en.wikipedia.org/wiki/Real_number)-valued [affine function](http://en.wikipedia.org/wiki/Affine_function) defined on this polyhedron. A linear programming [algorithm](http://en.wikipedia.org/wiki/Algorithm) finds a point in the polyhedron where this function has the smallest (or largest) value if such a point exists.

An integer linear program (ILP) consists of a set of integer variables, a set of inequalities involving these variables (which must form a convex region), and a linear function of these variables which we wish to minimise (respectively maximise) whilst maintaining all the inequalities. There exist methods, for example the simplex method, which can be used to help solve problems such as these, however the problem remains NP-complete.

An ILP approach is to solve the problem of rostering staff at call centres. Their approach begins by creating many “lines of work”, where a line of work is a pattern of work that an employee may follow over a given rostering period, (for example a day or a week). They represent a line of work as a column of 0’s and 1’s, with a 1 appearing in row j if the employee is available to serve customers in time period j. These lines of work are created such that they comply with restrictions on employee work patterns (for example union regulations and lunch breaks). A matrix L is then created from the line of work columns. They create a vector **x** where xi represents the number of employees following line i, and a vector **c**, where ci is the cost of the ith line of work. They use vector **s** to denote a vector of staff requirements, where si= n if n employees are required for period i. The algorithm uses an iterative approach, in which they use linear programming and simulation to find accurate values for s, and hence input s as a set of variables, in standard ILP techniques s is predefined before the ILP algorithm begins.

A integer linear program is of the form as follows,

min  + 

 +  = b

x, y >= 0

y  

If all variables need to be integer, it is called a integer linear program(ILP, IP).[5]

For above integer program, its linear relaxation consists in the LP obtained by dropping the integrality constraints:

min  + 

 +  = b

x, y >= 0

Taking Graph 1 as an example, it shows the following problem:

Max y

-x +y <=1

3x+2y<=12

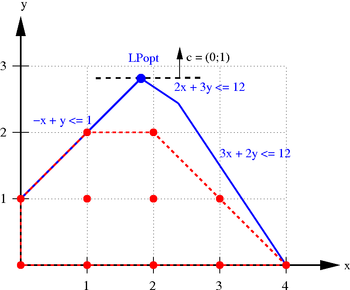
2x+3y<=12

x, y>=0

x, y

The feasible integer points are shown in red, and the red dashed lines indicate their convex hull, which is the smallest polyhedron that contains all of these points. The blue lines together with coordinate axes define the polyhedron of the LP relaxation, which is given by the inequalities without the integrality constraint. The goal of the optimization is to move the black dotted line as far upward while still touching the polyhedron. The optimal solutions of the integer problem are the points (1, 2) and (2, 2) which both have an objective value of 2. The unique optimum of the relaxation is (1.8, 2.8) with objective value of 2.8.

It is a simple example of the integer linear programming problem. Integer linear programming is a well studied area of mathematics, and provided the problem can be easily expressed as a set of linear integer constraints it provides a feasible way for finding optimal rosters. [6]

[](http://en.wikipedia.org/wiki/File:IP_polytope_with_LP_relaxation.png)

Graph 1. IP polytope with LP relaxation

**1.2.2 Constraint Programming**

Another methods often used in staff rota timetabling is Constraint Programming.

**1.2.2.1 Constraint Satisfaction problem**

We use a factored representation for a space of states : a set of variables, each of which has a value. A problem is solved when each variable has a value that satisfies all the constraints on the variable. A problem described this way is called a constraint satisfaction problem, or CSP.[7]

A constraint satisfaction problem consists of three components, X,D, and C:

X is a set of variables, {X1,....,Xn}.

D is a set of domains, {D1,....,Dn}, one for each variable.

C is a set of constraints that specify allowable combinations of values.

Each domain Di consists a set of allowable values, {v1,...,vk} for variable Xi. Each constraint Ci consists of a pair <scope, rel>, where scope is a tuple of variables that participate in the constraint and rel is a relation that defines the values that those variables can take on. A relation can be represented as an explicit list of all tuples of values that satisfy the constraint, or as an abstract relation that supports two operations: testing if a tuple is a member of the relation and enumerating the members of the relation. For example, if X1 and X2 both have the domain {A, B}, then the constraint saying the two variables must have different valus can be written as <(X1, X2),[(A, B), (B, A)]> or as <(X1, X2), X1  X2>.

To solve a CSP, we need to define a state space and the notion of a solution. Each state in a CSP is defined by an assignment of values to some or all of the variables, {Xi=vi, Xj=vj,...}. An assignment thta does not violate any constraints is called a consistent or legal assignment. A complete assignment is one in which every variable is assigned, and a solution to a solution to a CSP is a consistent, complete assignment. A partial assignment i one that assigns values to only some of the variables.

The structure of the CSP is the most important part of it since the same algorithms can be used to search any CSP. Since we know that the structure is standard across all CSPs, we can take a look at heuristics that are able to operate on all different types of problems. However, that is not to say that all algorithms are equally tractable and efficient on all sorts of problems. Currently, the decision of the use of an algorithm for a certain problem is determined empirically.[20]

A constraint is considered n-ary if it involves n variables. So if a constraint affects just a single variable, it is considered unary. Unary constraints can be dealt with as a preprocessing step. Constraints that involve two variables are binary constraints and are of particular interest for two reasons. One is that they can be modeled as a constraint graph, where the nodes of the graph represent the variables and an edge connects two nodes if a constraint exists between the two variables. The second is that a constraint of higher arity (the number of variables involved in a constraint) can always be reduced to a set of binary constraints. (However, that doesn't mean that this is always a good idea--in some cases, the number of binary constraints for a problem can be exponential, thus creating an intractable model.) More complex constraints, with arity > 2, are called global constraints. A simple example of a global constraint is the Alldifferent constraint; this constraint forces all the variables it touches to have different values.[20]

Taking Job-shop scheduling problem as an exmaple for CSP as follows. Factories have the problem of scheduling a day’s worth of jobs, subject to various constraints. In practice, many of these problems are solved with CSP techniques. Consider the problem of scheduling the assembly of a car. The whole job is composed of tasks, and we can model each task as a variable, where the value of each variable is the time that the task starts, expressed as an integer number of minutes. Constraints can also specify that a task takes a certain amount of time to complete. [7]

We consider a small part of the car assembly, consisting of 15 tasks: install axles (front and back), affix all four wheels(right and left, front and back), tighten nuts for each wheel, affix hubcaps, and inspect the final assembly. We can represent the tasks with 15 variables:

X = {Axl, Axle, Whee,, , , ,, , , , , , , Inspect}.

The value of each variable is the time that the task starts. Next we represent precedence constraints between individual tasks. Whenever a task T1 must occur before taskT2, and task T1 takes duration d1 to complete, we add an arithmetic constraint of the form T1 + d1 T2.

In our example, the axles have to be in place before the wheels are put on. And it takes 10 minutes to install an axle, so we write

Axl + 10  Whee;

Axl + 10  ;

Axle + 10  ;

Axle + 10  ;

Next we say that, for each wheel, we must affix the wheel (which takes1 minute), then tighten the nuts(2 minutes), and finally attach the hubcap (1 minute, but not represented yet):

Whee + 1  ;

 + 2   ;

 + 1  ;

 + 2   ;

 + 1  ;

 + 2   ;

 + 1  ;

 + 2   ;

Suppose we have four workers to install wheels, but they have to share one tool that helps put the axle in place. We need a disjuctive constraint to say that Axl and Axle must not overlap in time; either one comes first or the other does:

(Axl + 10  Axle ;) or (Axle + 10  Axl ;) .

This looks like a more complicated constraint, combining arithmetic and logic. But it still reduces to a set of pairs of values that Axl and Axl can take on.

We also need to assert that the inspection comes last and takes 3 minutes. For every variable except Inspect we add a constraint of the form X +   Inspect. Finally, suppose there is a requirement to get the whole assembly done in 30 minutes. We can achieve that by limiting the domain of all variables:

 = {1, 2, 3, ..., 27} .

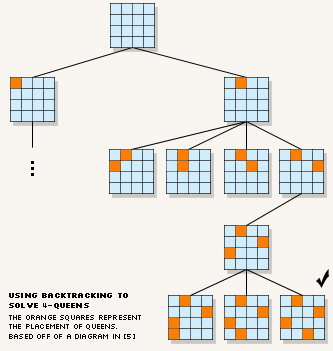
This particular problem is trivial to solve, but CSPs have been applied to job-shop scheduling problems like this with thousands of variables. In some cases, there are complicated constraints that are difficult to specify in the CSP formalism, and more advanced planning techniques.[7]

**1.2.2.2 Backtracking**

If we consider a simple depth-first search on a CSP(Constraint Satisfaction Problem), we realize that because of the constraints we have imposed, at some point during our search we may be unable to instantiate a variable because its domain is empty. In the case that we arrive at a node where the goal test returns false (there are still unassigned variables) and there are no branches leading away from that node, we must go backward. This is called backtracking and it is the most basic of searches for CSPs. A variable is assigned a value and then the consistency of that assignment is checked. If the assignment is not consistent with the state of the problem, another value is assigned. When a consistent value is found, another variable is chosen and this is repeated. If all values in the domain of a variable are inconsistent, the algorithm will backtrack to the previous assignment and assign it a new value.

Backtracking is a general [algorithm](http://en.wikipedia.org/wiki/Algorithm) for finding all (or some) solutions to some [computational problem](http://en.wikipedia.org/wiki/Computational_problem), that incrementally builds candidates to the solutions, and abandons each partial candidate c ("backtracks") as soon as it determines that c cannot possibly be completed to a valid solution.

The figure below illustrates how a simple backtracking search would work. It is the n-queens problem. In chess, a queen can move as far as she pleases, horizontally, vertically, or diagonally. The chess board below has 4 rows and 4 columns. The problem asks how to place 4 queens on an ordinary chess board so that none of them can hit any other in one move.

****

**1.2.2.3 Nurse rostering problem using constriant programming techniques**

A paper uses constraint programming techniques to solve a nurse rostering problem faced by the KlinikumInnenstadt hospital in Munich, Germany. A new nurse roster needs to be generated to each ward of hospital and its 20 or so staff. There are three work shifts that the nurses can take at each day (morning work shift, evening work shift and night work shift). Whe legal regulations, hospital working practices and demands, staff contracted hours and availability, and staff preferences need to be considered. The rostering staff realises that it is more likely that there is not appropriate roster that meets all these constraints. So they weight the constraints. They also realise that in some instances

it is more needed to meet a nurse’s preference than a legal requirement. So they usually make every nurse’s most wanted preference a hard constraint.

The problem variables are modeled as each nurse on each day. The domains of these variables are the work shifts the nurses can take, for example morning, evening and night work shifts. This approach is referred to as the ‘slot model’. There are also a set of ‘free shifts’ which include holidays, bank holidays owed and unpaid leave. These can also be assigned to a nurse if applicable.[1]

They then use a reduction of variable domains 1 based on elimination of interchangeable values 2. That is they reduce the set of ‘free shifts’ to a single value, ‘off-duty’. A nurse-day variable is assigned a value 0 if they are off duty on that day, 1 if they are assigned a morning shift, 2 for an evening shift, and 3 for a night shift.

With the model set out in this form, the basic constraints for this problem are naturally represented using a cardinality constraint of the form cardinality (Min, Max, ConditionList), which is satisfied if and only if at least Min and at most Max conditions in the condition list are satisfied. For example, one constraint is that there should be a minimum number of staff on the ward of hospital at any given time, and at the same time a maximum number should not be exceeded. Thus for each day and each type of shift (excluding ‘off-duty’) we constrain its occurrences each day to lie between its respective minimum and maximum values. Also each nurse has a minimum and maximum number of hours they need to work each month, this corresponds to a minimum and maximum number of shifts they can take each month, which again can be represented by a cardinality constraint.

The paper uses the manual planning process as a guide for creating heuristics that prune the search tree. This involves decomposing the problem into three phases, firstly all ‘free shifts’ are assigned, following this night shifts are assigned, and then finally morning and evening shifts are assigned. This follows from the fact that when manually rostering staff, the hardest shifts to roster are assigned first, and then the easier shifts to roster are fitted around these. During the first phase the decision is between ‘free shift’ and undecided, during the second phase only undecided variables are considered, unless no solution can be found. A result of this is that the search tree is binary and the search space is reduced.

‘Patterns’ are also introduced as another heuristic to direct the search. A pattern describes a preferred sequence of working, for example if most nurses work five days before taking two days off, then a pattern for the first phase of the search would be (\_,\_,\_,\_,\_,0,0). Such patterns encourage the satisfaction of soft constraints as is patterns which are pleasing to staff are considered first. Optimisation is achieved using branch and bound during each phase of the search; the solution to each phase is optimised before the next phase is considered. [1]

Compared to ILP techniques, CSP’s provide a much more natural representationfor rostering problems. They are also more flexible, allowing new constraints to be easily added whenever the problem changes, rather than having to redefine the linear equations of a linear program. Although constraint programming with naive search approaches may be slower than a linear program, the natural representation of the CSP makes it easier to identify heuristics (such as variable and value ordering and symmetry breaking) that can drastically reduce the search effort. There are many methods that can be used to solve the rostering problem. However, constraint programming may be the better one.

**1.2.3 Genetic algorithm**

Genetic algorithm is another method. Genetic algorithm (GA) is a [search](http://en.wikipedia.org/wiki/Search_algorithm) [heuristic](http://en.wikipedia.org/wiki/Heuristic_(computer_science)) method that mimics the process of [natural selection](http://en.wikipedia.org/wiki/Natural_selection). This heuristic (also sometimes called a [metaheuristic](http://en.wikipedia.org/wiki/Metaheuristic)) is routinely used to generate useful solutions to [optimization](http://en.wikipedia.org/wiki/Optimization_(mathematics)) and [search](http://en.wikipedia.org/wiki/Search_algorithm) [problems](http://en.wikipedia.org/wiki/Problem).[[10]](http://en.wikipedia.org/wiki/Genetic_algorithms#cite_note-FOOTNOTEMitchell19962-1) Genetic algorithm belongs to the larger class of [evolutionary algorithms](http://en.wikipedia.org/wiki/Evolutionary_algorithm) (EA), that generates solutions to the optimization problems using techniques inspired by natural evolution, such as [inheritance](http://en.wikipedia.org/wiki/Heredity), [mutation](http://en.wikipedia.org/wiki/Mutation_(genetic_algorithm)), [selection](http://en.wikipedia.org/wiki/Selection_(genetic_algorithm)), and [crossover](http://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)). [8]

In a genetic algorithm, a [population](http://en.wikipedia.org/wiki/Population) of [candidate solutions](http://en.wikipedia.org/wiki/Candidate_solution) (called individuals, creatures, or [phenotypes](http://en.wikipedia.org/wiki/Phenotype)) to an optimization problem is evolved toward better solutions. Each candidate solution has a set of properties (its [chromosomes](http://en.wikipedia.org/wiki/Chromosome) or [genotype](http://en.wikipedia.org/wiki/Genotype)) which can be mutated and altered; traditionally, solutions are represented in binary as strings of 0s and 1s, but other encodings are also possible.[[9]](http://en.wikipedia.org/wiki/Genetic_algorithms#cite_note-FOOTNOTEWhitley199466-2)

The evolution usually starts from a population of randomly generated individuals, and is an [iterative process](http://en.wikipedia.org/wiki/Iteration), with the population in each iteration called a generation. In each generation, the [fitness](http://en.wikipedia.org/wiki/Fitness_(biology)) of every individual in the population is evaluated; the fitness is usually the value of the [objective function](http://en.wikipedia.org/wiki/Objective_function) in the optimization problem being solved. The more fit individuals are [stochastically](http://en.wikipedia.org/wiki/Stochastics) selected from the current population, and each individual's genome is modified ([recombined](http://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)) and possibly randomly mutated) to form a new generation. The new generation of candidate solutions is then used in the next iteration of the [algorithm](http://en.wikipedia.org/wiki/Algorithm). Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

A typical genetic algorithm requires:

1. a [genetic representation](http://en.wikipedia.org/wiki/Genetic_representation) of the solution domain,
2. a [fitness function](http://en.wikipedia.org/wiki/Fitness_function) to evaluate the solution domain.

A standard representation of each candidate solution is as an [array of bits](http://en.wikipedia.org/wiki/Bit_array).[[2]](http://en.wikipedia.org/wiki/Genetic_algorithms#cite_note-FOOTNOTEWhitley199466-2) Arrays of other types and structures can be used in essentially the same way. The main property that makes these genetic representations convenient is that their parts are easily aligned due to their fixed size, which facilitates simple [crossover](http://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)) operations. Variable length representations may also be used, but crossover implementation is more complex in this case. Tree-like representations are explored in [genetic programming](http://en.wikipedia.org/wiki/Genetic_programming) and graph-form representations are explored in [evolutionary programming](http://en.wikipedia.org/wiki/Evolutionary_programming); a mix of both linear chromosomes and trees is explored in [gene expression programming](http://en.wikipedia.org/wiki/Gene_expression_programming).

Once the genetic representation and the fitness function are defined, a GA proceeds to initialize a population of solutions and then to improve it through repetitive application of the mutation, crossover, inversion and selection operators.[8]

Taking genetic algorithms for the bus driver scheduling problem as follows as an example:

Genetic algorithms for the bus driver scheduling problem discusses about the application of genetic algorithms to solve the complex bus driver scheduling problem. The genetic algorithm used for bus drive scheduling has five operators.

* Initialization
* Parent selection scheme
* Cross over
* Mutation
* Population replacement scheme

The bus driver scheduling is basically composed of candidate duties and the set of pieces-of-work.

Gene: 1 2 3 4 5 6 7 8 9 10 11 12 13 14

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 5 | 3 | 2 | 1 | 4 | 1 | 0 | 2 | 3 | 5 | 2 | 4 | 0 |

Figure 2: Solution with fourteen pieces of work

This coding always represents feasible relaxed partitioning solutions. A fitness function is developed to evaluate the fitness of the solution.

Initialization of population is done using random number generator. Encoding of the solutions is done and the pool of solutions is generated. Parent selection is done using roulette-wheel selection.

ps(i)= i=1....n

The roulette wheel contains one slot for each population element. The size of each slot is directly proportional to its respective ps(i), and therefore population memebers with high fitness values are likely to be selected more often than those with lower fitness values. The crossover operator is applied to pairs of selected chromosomes in order to generate one, two or more ‘children’. The procedure is divided into two phases phases. In the first phase, A duty of one of the parents is randomly selected. If it is available, it is added to chromosome under construction. The process is repeated until none of the duties in both parents is available. In the second phase the leftovers that still exist reduced. Two types of mutation operators are developed. [11]

* Basic Mutation
* Improve Mutation

Basic mutation is much like cross over operator. The process of selecting and inserting an available duty into the chromosome is repeated until there are no more available duties for that chromosome. In improve mutation operator, for each free piece-of-work; a duty is filled that also covers the same pieces-of-work of one of the adjacent duties. In population replacement operator, two types of replacement are developed. Generation replacement is the strategy adopted and the whole population is replaced. When the best element of the population is kept for the next generation, we get a generation replacement with elitism.[11]

In recent years genetic algorithms have emerged as a useful tool for the heuristic solution of complex discrete optimisation problems. In particular there has been considerable interest in their use in tackling problems arising in the areas of scheduling and timetabling. However, the classical genetic algorithm paradigm is not well equipped to handle constraints and successful implementations usually require some sort of modification to enable the search to exploit problem specific knowledge in order to overcome this shortcoming.

The basis of the family of genetic algorithms is a classical genetic algorithm consisting of n-point crossover, single-bit mutation and a rank-based selection. The solution space consists of all schedules in which each nurse works the required number of shifts, but the remaining constraints, both hard and soft, are relaxed and penalised in the fitness function.[2]

* 1. **Case study**

In the following chapter we will study a case about University Course Timetabling. A C++ constraint solver is used in this case. [17]

University course timetabling problems are combinatorial problems, which consist in scheduling a set of courses within a given number of rooms and time periods. Solving a real-world timetabling problem manually often requires a significant amount of time, sometimes several days or even weeks. Therefore, a lot of research has been invested in order to provide automated support for human timetablers. Contributions come from the fields of operations research(e.g., graph coloring, network flow techniques) and artificial intelligence (e.g. simulated annealing, tabu search, genetic algorithms, constraint satisfaction).[12]

In practice, there is no timetable that fullfills all the constraints. Thus, we have to distinguish two kinds of constraints. Hard constraints are conditions that must always be satisfied, soft constraints may be violated, but should be satisfied as far as possible.

Most existing constraint-based timetabling either do not support soft constraints [13] or use a branch-and-bound search instead of chronological backtracking[14]. Branch and bound starts out from a solution and requires the next solution to be better. Quality is measured by a suitable cost function that depends on the set of violated soft constraints. With this approach, however, soft constraints play no role in selecting variables and values, i.e., they don’t guide search.

Another approach is to adopt techniques developed to propagate hard constraints; soft constraint propagation is intended to associate values with an estimate of how selecting a value will influence solution quality, i.e., which value is known (or expected to) violate soft constraints, or the other way round, which value is known (or expected to) satisfy soft constraints. By considering estimates in value selection, one hopes that the first solution will satisfy a lot of soft constraints. For example, [15] presents a commercial C++ library providing black-box constraint solvers and search methods for the nurse scheduling problem.

Inspired by this approach, a constraint-based timetabling system for the Ludwig-Maximilians-Universitat of Munich, called IfPlan, has been developed using a CHR solver which performs hard- nad soft-constraint propagation[16].

* + 1. **Approach**

To model the university timetabling problem, only one variable for each course holding the period is needed, i.e., the starting time point, it has been scheduled for. Each domain of the variables consists of the whole week, the periods being numbered from 0 to 167, e.g., 9 donates 9 a.m. on Monday, and so on. Requirements, wishes, and recommendations can be expressed with a small set of specialized constraints.

* No-clash constraints demand that a course must not clash with another one.
* Preassignment constraints and availability constraints are used to express teachers’ preferences and that a course must (not) take place at a certain time.
* Distribution constraints make sure that there is at least on day (hour) between one course and another or that two courses are scheduled for different ways.
* Compactness constraints make sure that one course will be scheduled directly after another.

With Respect to soft constraints, who chose to distinguish three grades of preferences: weakly preferred, preferred and strongly preferred, which are translated to the integer weights 1, 3 and 9.

Since soft constraints may be violated, the values to be constrained must not be removed from the domain of the variable. Moreover, when we have to choose a value for the variable during search, we must be able to decide whether a certain value is a good choice or not. Therefore, each value must be associated with an assessment. We chose to represent a domain as a list of value-assessment pairs. For example, assume the domain of X is [(3, 0), (4, 1), (5, -1)], then X may take one of values 3, 4 and 5, whereas 4 is encouraged with assessment 1 and 5 is discouraged with assessment -1.[17]

* + 1. Solver

The solver is based on three types of constraints.

* Domain(X, D) means that X must be assigned a value occuring in the list of value-assessment pairs D.
* In(X, L, W): Its meaning depends on the weight W. If W = inf, i.e., if the constraint is hard, it means that the assessment for the values occurring in L should be increased by W.
* notin(X, L, W) , if hard, means that X must not be assigned any of the values occurring in the L. If it is soft, it means that the assessment for the values occurring in L should be decreased by W.

Propagating a soft constraint is intended to modify the assessment of the values to be constrained. For example, assume the domain of X is [(3, 0), (4, 1), (5, -1)] and assume the existence of the constraint in (X, [3], 2) stating that 3 should be assigned to X with preference 2. Then we have to increase the assessment for value 3 in the domain of X by adding 2 to the current assessment of 3, obtaining the new domain [(3, 2), (4, 1), (5, -1)] for X. However, applying a hard constraint will still mean to remove values from the variable’s domain. Consequently, an in constraint is processed by either pruning the domain or increasing the assessment for the given values.

fd\_in\_hard @ domain(X, D), in (X, L, W) < = > W = inf|

domain\_intersection(D, L, D1),

domain(X, D1) .

fd\_in\_soft @ domain (X, D), in (X, L, W) < = > W \=inf|

increase\_assessment(W, L, D, D1),

domain(X, D1) .

In case a hard in constraint has arrived, rule fd\_in\_hard looks for the corresponding domain constraint, which contains the current domain D, and replaces both by a new domain constraint, which contains the new domain D1. The domain D1 results from intersecting D with the list of values L. Rule fd\_in\_soft works quite similar, except for D1 results from D by increasing the assessments for the values occurring in L. Note that the guards exclude each other. Therefore, whichever constraint arrives, only one of the rules will be applicable. The rules for notin are similar.

fd\_notin\_hard @ domain(X, D), notin(X, L, W) < = > W = inf|

domain\_subtraction(D, L , D1),

domain(X, D1).

fd\_notin\_soft @ domain(X, D), notin(X, L , W) < = > W \=inf|

decrease\_assessment(W, L ,D, D1),

domain(X, D1).

Subtracting weights, which are always positive, may result in negative assessments.

Whenever a domain of a variable has been reduced to the empty list, the variable cannot be assigned a value without violating hard constraints. This case is dealt with by the following simplification rule.

fd\_empty @ domain(\_, [] ) < = > false.

With only one value left in a domain of a variable, we can assign the remaining value to the variable immediately.

fd\_singleton @ domain (X, [(A, \_ )]) 🡺 X=A.

We use a propagation rule instead of a simplification rule because the domain constraint must not be removed. Without it the processing of in and notin constraints imposed on the domain of a variable would not be guaranteed and thus an inconsistency might be overlooked. [17]

**Treatment of Global Constraints.** Up to now we only dealt with the simple constraints domain, in and notin. Now we exemplify how to express global (n-ary) application-level constraints in terms of in and notin constraints. [17]

no\_clash(W, Xs) means that, depending on the weight W, the variables from Xs must or should be assigned distinct values. It is translated to notin constraints. This translation is data-driven: whenever one of the variables from Xs is assigned a value, this value is discouraged or forbidden for the other variables by the following rule. [17]

fd\_no\_clash @ no\_clash (W, Xs) < = >

Xs \ = [ \_ ],

select\_ground\_var(Xs, X, XsRest)

|

post\_notin\_constraints(W, X, XsRest),

no\_clash(W, XsRest) .

The guard first makes sure that Xs contains at least two elements. Then it selects a ground variable X from Xs, remembering the other variables in XsRest. With no ground variable in Xs, the Prolog predicate select\_ground\_var fails. If the guard holds, no\_clash(W, Xs) is replaced by [17]

* notin constriants produced by the predicate post\_notin\_constraints, one for each member of XsRest, discouraging or forbidding the value X, and
* a no\_clash constraint stating that the variables in XsRest should or must be assigned distinct values.

Note that the predicate post\_notin\_constraints fails in case XsRest contains the value X.

A singleton list of variables means that there is nothing more to do. This case is handled by the following rule.

fd\_no\_clash\_singleton @ no\_clash(\_, [ \_ ]) < = > true.

The translation of the other application-level constraints either follows this scheme or is a one-to-one translation.

**Intersection of the** no\_clash **Rules and the Rules for Primitive Constraints**. In the following, we present two derivations to show how the CHR rules interact with each other. In the first derivation, we deal only with hard constraints.

domain(X, [(1, 0), (2, 0)]), domain(Y, [(1, 0), (2, 0)]),

no\_clash(inf, [X, Y]), in(X, [1], inf) 

domain(X, [(1, 0)]), domain(Y, [(1, 0) , (2, 0)]),

no\_clash(inf, [X, Y]) 

domain(X, [(1, 0)]), domain(Y, [(1, 0 ), (2, 0)]),

no\_clash(inf, [X, Y]), X=1 

domain(X, [(1, 0 )]), domain(Y, [(1, 0), (2, 0)]), X=1

notin(Y, [1], inf), no\_clash(inf, [Y]) 

domain(X, [(1, 0)]), domain(Y, [(1, 0 ), (2, 0 )]), X=1

notin(Y, [1], inf) 

domain(X, [(1, 0)]), X=1, domain(Y, [(2, 0)]) 

domain(X, [(1, 0)]), X=1, domain(Y, [(2, 0)]) , Y=2

In the sec

In the second derivation, we want to show how the rules treat soft no\_clash constraints.

domain(X, [(1, 0 ), (2, 0 )]) , domain(Y, [(1, 0 ), (2, 0)]),

no\_clash(1, [X, Y]) , in(X, [ 1 ], inf ) 

domain(X, [(1, 0 )]) , domain(Y, [( 1 , 0 ), (2 , 0 )]),

no\_clash(1, [X, Y]) 

domain(X, [( 1 , 0 )]) , domain( Y, [( 1 , 0) , ( 2 , 0 )]),

no\_clash(1, [ X , Y ]) , X=1 

domain( X, [( 1 , 0)]) , domain( Y, [( 1, 0 ) , (2 , 0)]) , X=1,

notin( Y , [ 1 ] , 1 ) , no\_clash( 1 , [Y]) 

domain( X, [( 1 , 0 )]), domain(Y , [(1 , 0 )(2 , 0)]) , X=1

notin(Y, [1] , 1) 

domain(X , [(1 , 0)]) , domain( Y , [( 1 , -1) , (2 , 0)]) , X=1

* + 1. **Generation of Timetables**

The generation of a timetable proceeds as follows. Each course is associated with a domain constraint allowing for the whole week, the periods being numbered from 0 to 167. It is important to note that, for each course, the initial assessment for all periods is 0, indicating that no period is given preference initially. Then preassignment constraints and availability constraints will be translated into in and notin constraints. Adding in and notin constraints may narrow the domains of the courses using the rules presented above. Propagation continues until a fixpoint is reached, that is to say, when further rewriting does not change the store. Usually, the solver is not powerful enough to determine that the constraints are satisfiable. In order to guarantee that a valid solution is found a search procedure is called. [17]

* + 1. **Evaluation**

IfIPlan has been in use at the Computer Science Department of the university of Munich since 1996. It brought down the time necessary for creating a timetable from a few days by hand to a few minutes on a computer. The core of the solver takes no more than 20 lines of code. Due to the declarativity of the approach, IfIPlan can be easily adapted to solve timetabling problems for other universities.[17]

1. **JaCoP constraint Solver**

**2.1 JaCoP introduction**

JaCoP is a constraint solver for [constraint satisfaction problems](http://en.wikipedia.org/wiki/Constraint_satisfaction_problem). It is written in [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) and it is provided as a Java library. JaCoP has an interface to the [AMPL](http://en.wikipedia.org/wiki/AMPL) modeling language. Its main focus is on ease of use, modeling power, as well as efficiency. It has a large collection of global constraints implemented to facilitate problem modeling. JaCoP is actively developed since year 2001. Krzysztof Kuchcinski and Radoslaw Szymanek are the core developers of this Java library. There are number of people who have contributed to JaCoP development in addition to core developers. JaCoP development has been influenced by more than 20 research articles from Constraint Programming community. It has been used as a tool in more than 30 research articles. There are many different examples provided so it is easier to learn how to use JaCoP. [18]

The JaCoP project contains a [wrapper](http://en.wikipedia.org/wiki/Wrapper_library) for the [Scala programming language](http://en.wikipedia.org/wiki/Scala_(programming_language)), and a wrapper for [Clojure](http://en.wikipedia.org/wiki/Clojure) is maintained as a separate project [CloCoP](https://github.com/aengelberg/clocop). [18]

JaCoP library provides constraint programming paradigm implemented in Java. It provides primitives to define finite domain (FD) variables, constraints and search methods.

JaCoP library provides most commonly used *primitive constraints*, such as equality, inequality as well as *logical*, *reified*and *conditional constraints*. It contains also number of *global constraints*. Finally, JaCoP defines also *decomposable constraints*, i.e., constraints that are defined using other constraints and possibly auxiliary variables. JaCoP library can be used by providing it as a JAR file or by specifying access to a directory containing all JaCoP classes.

2.2 An example problem using JaCoP

****

**Diagram 2.2**

In [graph theory](http://en.wikipedia.org/wiki/Graph_theory), **graph coloring** is a special case of [graph labeling](http://en.wikipedia.org/wiki/Graph_labeling); it is an assignment of labels traditionally called "colors" to elements of a [graph](http://en.wikipedia.org/wiki/Graph_(mathematics)) subject to certain constraints. In its simplest form, it is a way of coloring the vertices of a graph such that no two adjacent [vertices](http://en.wikipedia.org/wiki/Vertex_(graph_theory)) share the same color; this is called a **vertex coloring**. Similarly, an **edge coloring** assigns a color to each edge so that no two adjacent edges share the same color, and a **face coloring** of a planar graph assigns a color to each face or region so that no two faces that share a boundary have the same color.[19] Here we will consider the vertex coloring.

Consider the problem of Vertex graph coloring as depicted in Dia.2.2 . Below, we provide a simplistic program with hard-coded constraints and specification of the search method to solve this particular graph coloring problem.

import org.jacop.core.\*;   
import org.jacop.constraints.\*;   
import org.jacop.search.\*;   
   
public class Main {   
   
    static Main m = new Main ();   
   
    public static void main (String[] args) {   
        Store store = new Store();  // define FD store   
        int size = 4;   
        // define finite domain variables   
        IntVar[] v = new IntVar[size];   
        for (int i=0; i<size; i++)   
            v[i] = new IntVar(store, "v"+i, 1, size);   
        // define constraints   
        store.impose( new XneqY(v[0], v[1]) );   
        store.impose( new XneqY(v[0], v[2]) );   
        store.impose( new XneqY(v[1], v[2]) );   
        store.impose( new XneqY(v[1], v[3]) );   
        store.impose( new XneqY(v[2], v[3]) );   
   
        // search for a solution and print results   
        Search<IntVar> search = new DepthFirstSearch<IntVar>();   
        SelectChoicePoint<IntVar> select =   
            new InputOrderSelect<IntVar>(store, v,   
                                         new IndomainMin<IntVar>());   
        boolean result = search.labeling(store, select);   
   
        if ( result )   
            System.out.println("Solution: " + v[0]+", "+v[1] +", "+   
                                              v[2] +", "+v[3]);   
        else   
            System.out.println("\*\*\* No");   
    }   
}

This program produces the following output indicating that vertices v0, v1 and v3 get different colors (1, 2 and 3 respectively), while vertex v3 is assigned color number 1.

Solution: v0=1, v1=2, v2=3, v3=1

2.2.1 Finite Domain Variable

The problem is specified with the help of variables (FDVs) and constraints over these variables. JaCoP support both finite domain variables (integer variables) and set variables. Both variables and constraints are stored in the store (Store). Typically, it is created using the following statement.

    Store store = new Store();

Variable *x*:: 1*..*size is specified in JaCoP using the following general statement. Clerly, we required to have created store before we can create variables as any constructor for variable will require providing the reference to store in which the variable is created.

   v[i] = new IntVar(store, "v"+i, 1, size);

2.2.2 Constraints

In JaCoP, there are three major types of constraints:

* primitive constraints,
* global constraints, and
* decomposable constraints.

Primitive constraints and global constraints can be imposed using impose method, while decomposable constraints are imposed using imposeDecomposition method. The statement below imposes a primitive constraint XneqY. Again, in order to impose a constraint a store object must be available.

   store.impose( new XneqY(v[0], v[1]) );

**2.2.3 Search for solutions**

After specification of the model consisting of variables and constraints, a search for a solution can be started. JaCoP offers a number of methods for doing this. It makes it possible to search for a single solution or to try to find a solution which minimizes/maximizes given cost function. This is achieved by using the depth-first-search together with constraint consistency enforcement.

The consistency check of all imposed constrains is achieved calling the following method from class Store.

   boolean result = store.consistency();

When the procedure returns false then the store is in inconsistent state and no solution exists. The result true only indicates that inconsistency cannot be found. In other words, since the finite domain solver is not complete it does not automatically mean that the store is consistent.

To find a single solution the DepthFirstSearch method can be used. Since the search method is used both for finite domain variables and set variables it is recommended to specify the type of variables that are used in search. For finite domain variables, this type is usually. The relative code is shown below.

// search for a solution and print results   
        Search<IntVar> search = new DepthFirstSearch<IntVar>();   
        SelectChoicePoint<IntVar> select =   
            new InputOrderSelect<IntVar>(store, v,   
                                         new IndomainMin<IntVar>());   
        boolean result = search.labeling(store, select);

The depth-first-search method requires the following information:

* how to assign values for each FDV from its domain; this is defined by IndomainMin class that starts assignments from the minimal value in the domain first and later assigns successive values.
* how to select FDV for an assignment from the array of FDVs (v[i]); this is decided explicitly here by InputOrderSelect class that selects FDVs using the specified order present in v[i].
* how to perform labeling; this is specified by DepthFirstSearch class that is an ordinary depth-first-search.

JaCoP offers number of different search heuristics based on depth-first-search. For example, credit search and limited discrepancy search. They are implemented using plug-in listeners that modify the standard depth-first-serch. More defined constraints and search methods can be found in Appendix. A and Appendix. B.

1. **Problem Analysis**

**3.1 introduction to staff scheduling problem**

Staff planning is a typical problem arising in the management of large transport companies ,including railway companies.It is concerned with building the work schedules needed to cover a planned timetable. Each work schedule is constructed for each staff type(engine men, conductors, catering staff,etc).

There are two types of staff planning: Long-term planning and short-term planning. Normally long-term planning task is seperated into two stages:1.staff scheduling and 2 staff rostering. Staff scheduling yields short-term working schedules, called duties, for staff members, such that they satisfy schedule demands. Staff rostering arranges duties into long-term working schedules, called base rosters, and assigns specific staff member to them, such that each staff member performs a roster.

There are a lot of employee scheduling software in the market now. Employee scheduling software automates the process of creating and maintaining a [schedule](http://en.wikipedia.org/wiki/Schedule_(workplace)). Such software will usually track [vacation time](http://en.wikipedia.org/wiki/Vacation_time), [sick time](http://en.wikipedia.org/wiki/Sick_time), compensation time, and alert when there are conflicts. As a database of schedules are accumulated over time, it may analyze past activity and prepare data for [payroll](http://en.wikipedia.org/wiki/Payroll). Although it may not make strategic decisions to lower costs and improve performance, it does manage the tasks.[21]

For smaller businesses it is increasingly important to keep the costs low on this administrative task, which can be quite large keeping the composition of the modern [workforce](http://en.wikipedia.org/wiki/Workforce) in mind. Flexible management of availability of the employees, shift trading, automatic scheduling processes and the such are key in keeping the costs down. Many vendors are based exclusively online to meet the increasingly web savvy workforce of today. Scheduling can be done for a variety of industries, including retail, government, education, healthcare, manufacturing, and distributional services.[21]

**3.2 Nurse rostering problem**

Nurse rostering is an important application area of the staff scheduling. The nurse scheduling problem (NSP), also called the nurse rostering problem (NRP) is the [operations research](http://en.wikipedia.org/wiki/Operations_research) problem of finding an optimal way to assign nurses to shifts, typically with a set of hard constraints which all valid solutions must follow, and a set of soft constraints which define the relative quality of valid solutions.[22] And the nurse scheduling problem is NP-hard.

3.2.1 NP problem and NP-hard problem

A problem is assigned to the NP (nondeterministic polynomial time) class if it is solvable in polynomial time by a [non-deterministic Turing machine](http://mathworld.wolfram.com/NondeterministicTuringMachine.html).

A [P-problem](http://mathworld.wolfram.com/P-Problem.html) (whose solution time is bounded by a polynomial) is always also NP. If a problem is known to be NP, and a solution to the problem is somehow known, then demonstrating the correctness of the solution can always be reduced to a single [P](http://mathworld.wolfram.com/P-Problem.html) (polynomial time) verification. If P and NP are not equivalent, then the solution of NP-problems requires (in the worst case) an [exhaustive search](http://mathworld.wolfram.com/ExhaustiveSearch.html).[23]

A problem is said to be [NP-hard](http://mathworld.wolfram.com/NP-HardProblem.html) if an [algorithm](http://mathworld.wolfram.com/Algorithm.html) for solving it can be translated into one for solving any other NP-problem. It is much easier to show that a problem is NP than to show that it is [NP-hard](http://mathworld.wolfram.com/NP-HardProblem.html). A problem which is both NP and [NP-hard](http://mathworld.wolfram.com/NP-HardProblem.html) is called an [NP-complete problem](http://mathworld.wolfram.com/NP-CompleteProblem.html).[23]

3.2.2 Nurse rostering problem analysis

The **Nurse scheduling problem** (NSP) is all about assignment of shifts and holidays to nurses. Nurses have their wishes/restrictions. The problem is described as finding a schedule that both respects the constraints of the nurses and fulfills the objectives of the hospital. Conventionally a nurse can work 3 shifts like below.

* day shift
* night shift
* late night shift

However, nurses in some hospitals work 2 shifts like below.

* day shift
* night shift

In this problem we must search for a solution satisfying as many wishes as possible while not compromising the needs of the hospital. There are two types of constraints as listed below:

* hard constraints: if the constraint fails then the entire schedule is invalid
* soft constraints: it is desirable that these constraints are met but not meeting them doesn’t make the schedule invalid

Some possible constraints are listed in the following table:

|  |  |  |
| --- | --- | --- |
| Constraints number | Constraints description | Hard/Soft constraints |
| C1 | Working hours must not exceed   * 9 hours per day for day and night shifts * 10 hours per day for late night shifts   This limits a nurse’s work to one shift per 24 hours. | *Hard* |
| C2 | A nurse should not work more than 7 days in a row. If she does work 7 consecutive days, she is required to take at least 2 days off. | *Hard* |
| C3 | A nurse should not work more than 4 late nights in a row. If she does work 4 consecutive late nights, she is required to take at least 2 days off. | *Hard* |
| C4 | If a nurse works a late night shift, she may work neither the day shift, nor the night shift in the following day. | *Hard* |
| C5 | If a nurse works an night shift, she may not work the day shift in the following day | *Hard* |
| C6 | A certain number of nurses is required to provide patient care for each shift of each day | *Hard* |
| C7 | A nurse should not work for 2 consecutive weekends | Soft |
| C8 | A nurse should not work one day out of two on a weekend, either the whole weekend is worked, or the weekend is free | Soft |
| C9 | Chief nurse is exempted from late night shifts | Soft |
| C10 | Don’t combine those nurses, who are not in good relations with each other | Soft |
| C11 | The interval between late night shift must be at least 3 days | *Hard* |
| C12 | The nurse can choose desired work shift | Soft |
| C13 | The nurse can select preferred partner | Soft |
| C14 | The nurse can choose not to work in a specified work shift | Soft |
| C15 | The nurses should have enough rest time between 2 work shifts | Hard |
| C16 | The time between anyone’s 2 consecutive shifts should be at least 2 work shifts(2\*8 hours) | Hard |
| C17 | A nurse’s working hours in a week should be less than 40 hours | Hard |
| C18 | A nurse can choose not to work with another nurse | Soft |

Table 1 The possible constraints

NSP(Nurse scheduling problem) as a scheduling task consists of assignment of shifts and holidays to nurses for each day on the horizon, taking into consideration a variety of conflicting interests or objectives between the hospitals and individual nurses. Given a number of nurses with specific skills and working agreements , a contract may consist of general constraints as there are restrictions on the number of nurses for each shift; the maximum number of shifts in a week, in a day, a month,etc.[24]

Moreover, a number of shifts personal wishes or desires representing nurses’ preferences are allowed. For instance, a demand for the desired day off, demand for doing certain shift on a certain day with a certain nurse,etc. Conventionally, every nurse works on three shifts, day shift, night shift, and late night shift and have some holidays.[24]

A hospital is composed by several departments, and each department is composed of one or more nursing teams called FMU (functionalmedical unit) . The FMU is make up of a team of nurses, with a head nurse who manages the unit’s schedules.

Generally a unit operates 24 hours a day, 7 days a week, with the working day divided into 3 periods: a day shift (D) of 8 working hours, night shift (N) of 8 working hours, and a late night shift (LN) of 10 working hours. The length of on-duty hours may change according to the coun-try’s work laws.

A schedule resolves in advance potential conflicts of needs between two parties, on one hand the department’s needs requiring a certain number of activities per FMU (meaning the number of nurses required per shift), and on the other hand the needs of nursing staff who demand acceptable schedules, with regard to all the rules as well the personal wishes of the nursing staff. Sacrifices must sometimes be imposed; for example, a nurse may be compelled to work on a requested day off. To these difficulties are added both frequent and unforeseen (12 % in France) absences which require adjustments in schedules.[25]

Nurse scheduling automation has been widely recognised, both internationally and within the health service as a way to help utilise staff in the most effective and efficient way.

Within nursing, effective nurse scheduling systems have helped trusts replace paper-based systems and achieve a more efficient and accurate way of managing staff and their time. By stripping back the administrative processes and utilising substantive staff more effectively, trusts have been able to save on average 1.5% of nursing staff costs and up to 1.2 whole time equivalent per ward per annum, while ensuring that the highest possible levels of care are consistently delivered.

An nurse scheduling system specifically for medical staff opens up enormous opportunities to more effectively manage the schedules of junior doctors, and the work patterns of consultants and locums, while operating within the required constraints. It also enables trusts to effectively manage and monitor sickness absence, minimise the use of agency staff through more effective utilisation of salaried staff, quickly adjust staff rotas when consultants' schedules change, and keep accurate audit trails to maintain the highest levels of clinical [governance](http://www.hsj.co.uk/home/board-talk/).

In addition, nurse scheduling system requires less time to administer than traditional paper-based systems and can lead to a saving of one-quarter of a medical staffing coordinator, per specialty, per year. In trusts where consultants are managing junior doctor rotas, nurse scheduling system has led to the freeing up of at least half a programmed activity.

1. **Design and Implementation**

Interview with nurse

* 1. Design
     1. Finite domain variable in the software

In the nurse rostering problem, we assume that there are 3 work shifts in one day which is day shift, night shift and late night shift. Then a day is divided into 3 shifts. So a week is divided into 21(7\*3) work shifts like below.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday |
| Day | 1 | 4 | 7 | 10 | 13 | 16 | 19 |
| Night | 2 | 5 | 8 | 11 | 14 | 17 | 20 |
| Late night | 3 | 6 | 9 | 12 | 15 | 18 | 21 |

We assume that the number of nurses is 10.

Then the variable that indicates a nurse’s work shift is Vij(i indicates nurse number, j indicates work shift number). The meaning of the value of the variable is as follows.

Vij=1 means on duty.

Vij=0 means not on duty.

For example  =1 means that the nurse 1 is on duty in shift 1 (day work shift on Monday). And  =0 means that the nurse 1 is off duty in shift 2 (night work shift on Monday).

So the finite domain variables related to nurse 1 is in the set (,,,,,,,,,,,,,,,,,,,,).

And the finite domain variables related to work shift 1 is in the set (,,,,,,,,,).

So for nurse i, a possible solution is as follows.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Nurse i | Monday | Tuesday | Wednesday | Thursday | Friday | Saturday | Sunday |
| Day | 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| Night | 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| Late night | 0 | 0 | 0 | 0 | 0 | 1 | 0 |

In the table, 1 means nurse i is on duty in that work shift. 0 means nurse i is off duty in that work shift.

* + 1. The nurse rostering Constraints used in this project

We have seen some possible nurse rostering constraints in table 1. Some of them are hard constraints. Some of them are soft constraints. Taking the realistic situation of the small hospitals, I selected some constraints and discussed them with the nurses. The records are shown in the following table.

|  |  |  |
| --- | --- | --- |
| constraint | need it or not | suggestion |
| **The time between anyone’s 2 consecutive shifts should be at least 2 work shifts(2\*8 hours). or whatever** | Yes | The software can allow user to set the number of work shifts between 2 consecutive work shifts |
| **A nurse’s working hours in a week should be less than 40 hours** | Yes | Change it to the A nurse’s working hours in a week should be between A and B.  A is the minimum number of hours the nurse should work.  B is the maximum number of hours the nurse should work. |
| **a nurse can choose not to work with another nurse** | No | The nurse can’t choose not to work with someone in real situation |
| **A nurse can choose the time block they don’t want to work** | No | The nurse is employed by the manager so she can’t choose not to work in some shifts |
| **A nurse can choose the time block they want to work** | Yes | The nurse often has desired work time |
| **The software user can set the number of nurses working in a single work shift** | Yes | It is necessary because the number of nurses required in a work shift is varied in different hospital. |

The working hours of a nurse in a hospital is varied due to the different regulations of the hospitals. And the nurse may work different hours due to their contracts. Some nurses work for 13 hours a week. Some nurses work for 22 hours a week. And some nurses work for 37.5 hours a week. And working for 37.5 hours a week is maximum according to the law. So the minimum work hours and maximum work hours constraints are needed for this software.

For the constraint that the time between anyone’s 2 consecutive work shifts should be at least 2 work shifts. It really depends on how long the nurse works in her last work shifts. For example, if the nurse worked 12 hours in the day, she should have 2 days’ time to rest. Because the nurses also need enough time to have a rest to adjust her body to best situation. So the software should allow the manager to set how long the break between 2 consecutive work shifts is.

A principle should always be considered in the design of the nurse roster software which is that the nurse is a person instead of a machine. So the nurses’ rights and desires should be considered. For example the nurse can choose the time block that they prefer to work.

The nurse roster software is more suitable to big hospitals. Because there are many nurses in big hospitals, generating nurse schedule manually is difficult. The small clinics don’t need the software so much because the nurse schedule can be generated easily by hand in small clinics. And the nurse roster software can save a lot of papers for big hospitals.

The constraint “The nurse can choose the time block they don’t want to work ” is not feasible. Because the nurses are employed by hospitals, working is their duty. They can’t refuse to work in a shift. However, the constraint " the nurse can choose the block that they want to work" is very good because it's easier for nurses to deal with family business such as taking care of children.

* + 1. **Express the constraints in mathematical way**

In order to implement all the constraints, first we should translate the constraints in mathematical way. All the expression of constraints is as follows.

1. The time between anyone’s 2 consecutive shifts (C=the minimum time to rest) should be at least 2 work shifts(2\*8 hours).

       ( i is 0 …9 , j is 0…20)

The expression means that the sum of any nurse’s three consecutive work shifts is 0 or 1. It means that the break between 2 work shifts is at least 2 work shifts’ rest.

1. A nurse’s working hours in a week should be less than 40 hours.

 (i is the number of the nurse, E is the work shifts in a week for nurse i)

8 hours is a work shift, so 40 hours is 5 work shifts. The working shifts in a week for a nurse is less than or equal to 5 work shifts. So the sum of the work shifts variables of a nurse i should be less than or equal to 5.

1. A nurse can choose not to work with another nurse

 (j is the work shift number, C is the set of incompatible nurses)

For a single work shift j, the sum of nurses variables who are in the set of incompatible set in the same work shifts should be less than or equal to 1. That means no incompatible nurses will work together at the same time.

1. A nurse can choose the time block they don’t want to work.

 (i is the nurse number, M is a set of work shifts the nurse i don’t want to work)

The sum of the variables in M for nurse is should be 0. That means the nurse I will not work in those work shifts.

1. A work shifts should be allocated 2 nurses

 (i is the nurse number, I is the set of all the nurses)

The sum of all the nurse variables that works in a single work shift should be 2. That means there is 2 nurses working in that work shift.

1. Nurse’s preference to work in a work shift

 (P is the set of the nurse i ‘s preferred work shifts)

It means at least one preference for nurse i should be satisfied.

* + 1. **Algorithm complexity in the nurse roster problem**
       1. **Big O notation**

In [mathematics](http://en.wikipedia.org/wiki/Mathematics), big O notation describes the [limiting behavior](http://en.wikipedia.org/wiki/Asymptotic_analysis) of a [function](http://en.wikipedia.org/wiki/Function_(mathematics)) when the argument tends towards a particular value or infinity, usually in terms of simpler functions.  In [computer science](http://en.wikipedia.org/wiki/Computer_science), big O notation is used to [classify algorithms](http://en.wikipedia.org/wiki/Computational_complexity_theory) by how they respond (e.g., in their processing time or working space requirements) to changes in input size.

Big O notation characterizes functions according to their growth rates: different functions with the same growth rate may be represented using the same O notation. The letter O is used because the growth rate of a function is also referred to as order of the function. A description of a function in terms of big O notation usually only provides an [upper bound](http://en.wikipedia.org/wiki/Upper_bound) on the growth rate of the function.[26]

Let f(x) and g(x) be two functions defined on some subset of the [real numbers](http://en.wikipedia.org/wiki/Real_number). One writes

https://lh5.googleusercontent.com/vQwVdxCMPMmDN3jxQVLGcSnSAmZfGm4RQlLraQK9UOKtx0R9rqHzfUvlYkYQnQLvcpq-xdvNCdMTGzDrvSmt_0jyxpAAEVebi_LqWKdCdtEXhF5VdkVf7topWPb1zVcVYA

[if and only if](http://en.wikipedia.org/wiki/If_and_only_if) there is a positive constant M such that for all sufficiently large values of x, f(x) is at most M multiplied by g(x) in absolute value. That is, f(x) = O(g(x)) if and only if there exists a positive real number M and a real number x0 such that https://lh6.googleusercontent.com/YeRnmaKe9G6LFSkZTQCiVoTz4LlocA_KAE4cerB3JmHHj8kzLHxhl-d3vNr7omXKj7iA-pcssxb5EwwQ7G7QhatoC2rHUlJxNuitP0eM5gL4H_zpxiClT1gWfKDfDTLUMg

Below are some common orders of growth along with descriptions and examples where possible.[27]

### O(1)

O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.

bool IsFirstElementNull(String[] strings)

{

if(strings[0] == null)

{

return true;

}

return false;

}

### O(N)

O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set. The example below also demonstrates how Big O favours the worst-case performance scenario; a matching string could be found during any iteration of the for loop and the function would return early, but Big O notation will always assume the upper limit where the algorithm will perform the maximum number of iterations.

bool ContainsValue(String[] strings, String value)

{

for(int i = 0; i < strings.Length; i++)

{

if(strings[i] == value)

{

return true;

}

}

return false;

}

### O(N2)

O(N2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set. This is common with algorithms that involve nested iterations over the data set. Deeper nested iterations will result in O(N3), O(N4) etc.

bool ContainsDuplicates(String[] strings)

{

for(int i = 0; i < strings.Length; i++)

{

for(int j = 0; j < strings.Length; j++)

{

if(i == j) // Don't compare with self

{

continue;

}

if(strings[i] == strings[j])

{

return true;

}

}

}

return false;

}

### O(2N)

O(2N) denotes an algorithm whose growth will double with each additional element in the input data set. The execution time of an O(2N) function will quickly become very large.

* + - 1. **Algorithm complexity in this problem**

For the software, the time to find a solution is related to the constraint number and the finite domain variable number. In the diagram 5, x-axis indicates the ratio between constraints and variables. The y-axis indicates the execution time. In other words, y-axis indicates the time to find a solution for the software. The execution time will increase with the increase of the constraint number while the variable number remains invariable. And the execution time will increase with the decrease of the variable number while the constraint number remains invariable.

We usually pay attention to the worst case execution time, the time that the algorithm takes to find a solution in the worst case. Between point “A” and “B” in the diagram 5, the execution time for finding a solution increases quickly. The user of the software should get a warning at the points between A and B points that “the execution time will be very long, would you like to reduce the constraints.” And at a point in the middle of A and B points, the execution time is very long. At that point, the software should remind the user to reduce constraints.

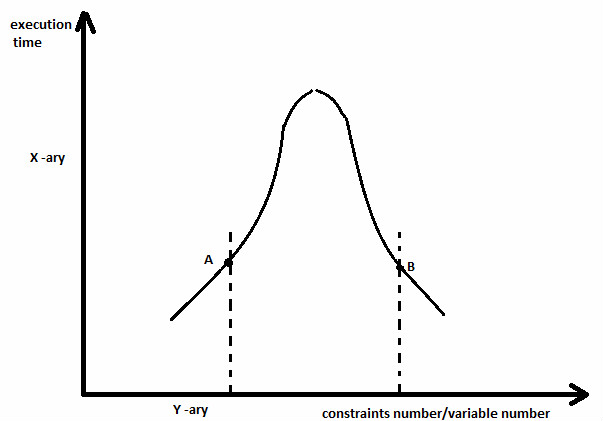


Diagram 5. The relationship between Solution search time and Constraint number/variable number

The diagram 6 shows that the solution number of the roster is related to the constraint number and the variable number. In the diagram 6, the x-axis indicates the ratio between constraints and variables. The y-axis indicates the number of solutions to the current constraints. The number of solutions will decrease with the increase of the constraints while the variables remains invariable. And the number of solutions will decrease with the decrease of the variables while the constraint number remains invariable. So as the ratio increase, the number of solutions will decrease. At the right of “E” point, there will not be any solution. So if the user add too many constraints, the software will not find a solution. So the software should warn the user to reduce constraints when the user add many constraints because there will not be a solution.

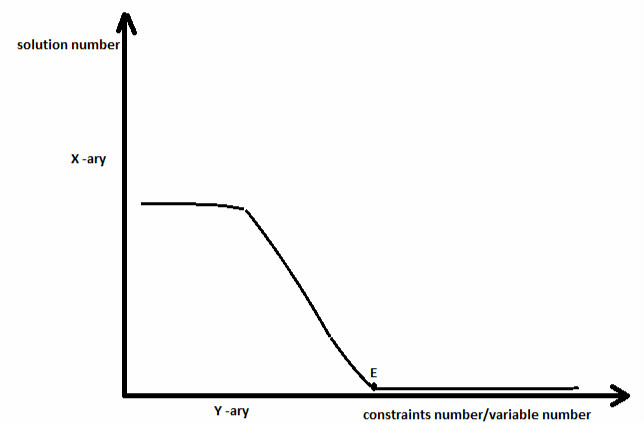


Diagram 6. The relationship between Solution number and Constraint number/variable number

* + 1. The software development paradigms used

Iterative life cycle model will be used in the software development process. The requirement of the project often changes in the development process. The project is not a very large project. So the software development method used in this project needs to be flexible and easy to use. So the iterative model is chosen in this project.

The Iterative model can be thought of as a “multi-waterfall” cycle. As shown in the diagram 7, cycles are divided into smaller and easily managed iterations. Each iteration passes through a series of phases, so after each cycle you will get working software.

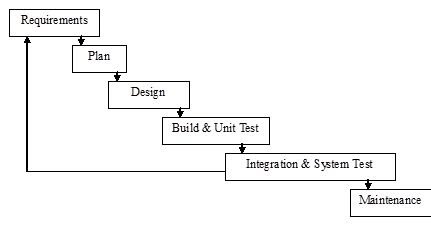


Diagram 7. Iterative model

An iterative life cycle model does not attempt to start with a full specification of requirements. Instead, development begins by specifying and implementing just part of the software, which can be reviewed in order to identify further requirements. This process is then repeated, producing a new version of the software for each cycle of the model.[28]

**Advantages of the iterative model**

1.    Produces working software early during the lifecycle.

2.    More flexible as scope and requirement changes can be implemented at low cost.

3.    Testing and debugging is easier, as the iterations are small.

4.    Low risks factors as the risks can be identified and resolved during each iteration.

5. In iterative model we are building and improving the product step by step. Hence we can track the defects at early stages. This avoids the downward flow of the defects.

6. In iterative model we can get the reliable user feedback. When presenting sketches and blueprints of the product to users for their feedback, we are effectively asking them to imagine how the product will work.

7. In iterative model less time is spent on documenting and more time is given for designing.

**Disadvantages of the iterative model**

1.    This model has phases that are very rigid and do not overlap.

2.    Not all the requirements are gathered before starting the development; this could lead to problems related to system architecture at later iterations.

* + 1. **The software requirement specification**

For ease of reading, the requirements have been laid out in a table. We have split them into functional and non-functional. The first column is the requirement reference number (“FR” stands for “Functional Requirement” and “NFR” for “Non-Functional Requirement”).The second column is a description of the requirement.

|  |  |
| --- | --- |
| No. | Description |
| FR1 | The software should allow user to add nurse information to database and the software should store these data |
| FR2 | The added data can be shown on a list in the software |
| FR3 | The user can log in and see their own roster |
| FR4 | The user can add some nurses’ preference to a work shift |
| FR5 | The user can set the number of the nurses for a single work shift |
| FR6 | The user can set the minimum number of work shifts a nurse should work in a week |
| FR7 | The user can set the maximum number of work shifts a nurse should work in a week |
| FR8 | The user can set the length of the break between the nurse’s 2 consecutive work shifts |
| FR9 | The constraints should show on the screen of the software as soon as the constraints are added |
| FR10 | A nurse roster for a week should be generated and printed on the screen when the user clicks the “Generate Roster” button |
| FR11 | The roster printed should show the date of the work shift |
| FR12 | The software should generate the roster of a week from the current date on |
| NFR1 | The software will run on windows system |
| NFR2 | The solution will be generated within 3 seconds |
| NFR3 | The nurse data that is added will be stored on the computer |
| NFR4 | The software will be used by manager and nurse |
| NFR5 | The software should easy to use and generate roster on behalf of the nuses’ rights |
| NFR6 | The requirement must be well written. The code must be well documented. The code must be modular and extensible |
| NFR7 | There should not be bugs in our software |

* + 1. **Optimize the solution**

When we get a roster solution for the nurses, it may be not good enough. How could we get the best solution? We can assign each solution a rank. When we generate a solution, we rank it. If the solution’s rank is too low, we drop it and search another solution until the rank is high or the limited time is consumed.

When we generate a solution, it may not be suitable to some nurse because some nurse may not be available to work at that time. So we need to change the solution roster. But the change should not affect other nurses’ rosters too much. How could we achieve that? We can design the software to remember the current solutions and change only a part of the solution. In other words, we can only change the part that is related to the affected nurse. Then the other’s nurses rosters will not be affected.

However, how could we change only a part of the solution?

1. Give the software the previous solution and try to find a new solution based on it.
2. Design a function out of the solver. Ask the function to compare the difference between the old solution and the new solution. If there are too many changes, drop the new solution and search another one until that the new solution is very similar to the previous solution.

And how could we compare the solutions with each other? Taking the following roster table into account, one method is to count the number of cell changes of 1->0 and 0->1. And we can give the number of changes a limit such as 5.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Roster 1 | Roster 2 | Roster 3 | Roster 4 | Roster 5 | Roster 6 |
| Nurse A | 0 | 1 | 1 | 0 | 0 | 1 |
| Nurse B | 1 | 1 | 0 | 0 | 1 | 0 |

Another method is to count the number of people affected. In other words, we can count how many nurses need to change their roster.

Optimizing the roster partially is important to the manager. No one wants to change their rosters frequently. So the optimization of solution will be implemented at the end of the project.

* 1. **Implementation**

**4.2.1 Finite domain variables implementation**

In the code, the finite domain variable is defined as follows.

v = new IntVar[i\*j];

Every variable is defined to an integer variable(IntVar) according to the JaCop solver. Because the JaCoP solver’s search method can not accept 2-dimensional array, so I use 1-dimensional array to represent the nurse’s duty for a work shift.

In my previous design, I use the format v[i][j] to represent a nurse’s duty. V[i][j] means the nurse i works in the work shift j if v[i][j] equals to 1. For example, v[1][2] means the nurse 1 works in the work shift 2 if v[1][2] equals to 1. Due to the limit of the JaCoP solver, I have to change the format from v[i][j] to v[n].

In my design, first I assume the number of nurses is 10. And there are three work shifts every day which is day shift, night shift and late night shift. And the software will generate a week’s roster. So the number of the work shifts is 21(3\*7). So the dimension of variable array should be 210(21\*10). That is v[0]…v[209].

For the nurse 0, the variables that represent her duty is taken from the following set.

S0= { v[0], v[1], v[2], v[3], v[4], v[5], v[6], v[7], v[8], v[9], v[10], v[11], v[12], v[13], v[14], v[15], v[16], v[17], v[18], v[19], v[20]}.

For the nurse 1, the variables that represent her work shifts are in the following set.

S1= {v[21], v[22], v[23], v[24], v[25], v[26], v[27], v[28], v[29], v[30], v[31], v[32], v[33], v[34], v[35], v[36], v[37], v[38], v[39], v[40], v[41]}.

.

.

.

The work shifts of the remaining nurse 2 to nurse 9 will be represented using the same formats like above.

The set for nurse 9 is as follows.

S9={v[189], v[190], v[191], v[192], v[193], v[194], v[195], v[196], v[197], v[198], v[199], v[200], v[201], v[202], v[203], v[204], v[205], v[206], v[207], v[208], v[209]}.

In summary , the variable v[i][j] will be changed to the format v[i\*21+j] in the code.

**4.2.2 The initialization of the finite domain variables**

As shown in the following code, v is the integer variable array. It indicates all the work shifts of every nurse. i is the number of the nurse. j is the number of work shifts in a week. So the dimension of the variable array is i\*j. And IntVar() function is called to initialise the variables. Store is the first parameter. The second parameter is an identifier for this variable. The third and fourth parameter indicates the domain of the variable. For example, parameter 0 and 4 indicates the set {0, 1, 2, 3, 4}. The following double loop will initialise all the variables.

v = new IntVar[i\*j];

for(int m=0;m<i;m++){

for(int n=0;n<j;n++){

v[j\*m+n]=new IntVar(store,"v"+m+"-"+n,0,1);

}

}

* + 1. **Adding Constraints**

The JaCoP solver provides a lot of constraint classes for programmer to add appropriate constraints. Then we should use store.impose() method to impose the constraints to the solver. In the Software, we will add constraints one by one. In the beginning, there are no constraints in the solver. We need to add constraints by user.

* + - 1. **adding constraint: set the break between a nurse’s two consecutive work shifts.**

The parameter breakNum is the length of the break between 2 consecutive work shifts. Here we set the limit to {2, 3, 4}. In the following code, if breakNum is 2, then we impose constraint Sum(new IntVar[]{v[21\*m+n],v[21\*m+n+1],v[21\*m+n+2]}, sum1). Sum1 is 0 or 1. It means the sum of any 3 consecutive work shift value is less than or equal to 1. And we impose different constraints according to the value of breakNum. The code for adding this constraint is as follows.

public void setBreak(int breakNum){

if(breakNum==2){

for(int m=0;m<i;m++){

IntVar sum1=new IntVar(store,"s",0,1);

for(int n=0;n<j-2;n++){

store.impose(new Sum(new IntVar[]{v[21\*m+n],v[21\*m+n+1],v[21\*m+n+2]},sum1));

}

}

}else if(breakNum==3){

for(int m=0;m<i;m++){

IntVar sum1=new IntVar(store,"s",0,1);

for(int n=0;n<j-3;n++){

store.impose(new Sum(new IntVar[]{v[21\*m+n],v[21\*m+n+1],v[21\*m+n+2],v[21\*m+n+3]},sum1));

}

}

}else{

for(int m=0;m<i;m++){

IntVar sum1=new IntVar(store,"s",0,1);

for(int n=0;n<j-4;n++){

store.impose(new Sum(new IntVar[]{v[21\*m+n],v[21\*m+n+1],v[21\*m+n+2],v[21\*m+n+3],v[21\*m+n+4]},sum1));

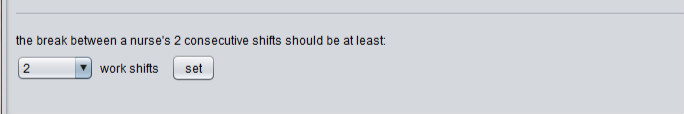
}

}

}

}

In the GUI, we input number in the following input area.



* + - 1. **Adding constraint: set the nurse number of every work shift**

We will set the nurse number of all the work shifts when we add this constraint. The parameter nurseNumber sets the number of the nurses that work in a work shift. The ArrayList nurses add all the variables of the nurses who work in this work shift. Then we impose the sum constraint that the sum of the variables in the nurse\_array is equal to the nurseNumber. The code for adding this constraint is as follows.

public void setNurseNumberPerShift(int nurseNumber){

for(int p=0;p<j;p++){

IntVar n2=new IntVar(store,"n",nurseNumber,nurseNumber);

ArrayList<IntVar> nurses=new ArrayList<IntVar>();

for(int m=0;m<i;m++){

nurses.add(v[j\*m+p]);

}

IntVar[] nurse\_array=new IntVar[nurses.size()];

nurse\_array=nurses.toArray(nurse\_array);

store.impose(new Sum(nurse\_array,n2));

}

}

}

* + - 1. **Adding constraint: set the minimum and maximum number of work shifts in a week.**

In this function, there are 2 parameters, min and max. Min indicates the minimum number of work shifts a nurse should work in one week. In the function, we impose the constraint that the sum of all the work shift variables of the nurse in a week should be in the range from min to max. The code for adding this constraint is as follows.

public void setWorkRangePerWeek(int min,int max){

for(int p=0;p<i;p++){

IntVar workshifts\_per\_nurse=new IntVar(store,"workpernurse",min,max);

store.impose(new Sum(new IntVar[]{v[j\*p], v[j\*p+1], v[j\*p+2],

v[j\*p+3], v[j\*p+4], v[j\*p+5], v[j\*p+6], v[j\*p+7], v[j\*p+8], v[j\*p+9], v[j\*p+10], v[j\*p+11],v[j\*p+12], v[j\*p+13], v[j\*p+14], v[j\*p+15], v[j\*p+16], v[j\*p+17], v[j\*p+18], v[j\*p+19], v[j\*p+20]}, workshifts\_per\_nurse));

}

}

* + - 1. **Adding constraint: set the nurse’s preference to a work shift**

The following method set a nurse’s preference to a work shift. NurseNum is the nurse number. WorkShift is the work shift number that the nurse wants to work. Then we impose the XeqC constraint class. It means that the variable that indicates a nurse’s work shift should be 1.

public void setNurseWorkShiftPreference(int nurseNum,int workShift){

store.impose(new XeqC(v[(nurseNum-1)\*21+workShift],1));

}

* + 1. **Searching the solutions**

After specification of the model consisting of variables and constraints, a search for a solution can be started. JaCoP offers a number of methods for doing this. In the following code to search the solution, the depth first search is used.

public IntVar[] findSolution(){

Search<IntVar> search = new DepthFirstSearch<IntVar>();

SelectChoicePoint<IntVar> select = new InputOrderSelect<IntVar>(store, v, new IndomainMin<IntVar>());

search.setSolutionListener(new PrintOutListener<IntVar>());

search.getSolutionListener().recordSolutions(true);

boolean result = search.labeling(store, select);

if ( result ) {

System.out.println("Solution: " + v[0]+", "+v[1]+", "+v[2]);

}

else

System.out.println("\*\*\* No");

return v;

}

The function findSolution searches for a solution using the depth first search. Then we need to create a Search class object. Here we create a DepthFirstSearch object.

Then we should set the order to choose variables from the FDV array. And we should specify how to assign a value to a variable. The SelectChoicePoint class should be created. And the InputOrderSelect class specify that the FDVs are selected by the order present in the array. And IndomainMin class specify that the value of the variable should be selected from the minimal value in the domain first and later assigns successive values.

The statement boolean result = search.labeling(store, select) will call the search for the solution according to the constraints we have added. If the solver finds a solution, the result is specified true. If it can’t find a solution, the result is false. Finally the findSolution function will return the whole variables array that represents all the nurses’ duties.

* + 1. The GUI

The GUI of the software uses Java Swing toolkit.  Swing is the primary [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) [GUI](http://en.wikipedia.org/wiki/Graphical_user_interface) [widget toolkit](http://en.wikipedia.org/wiki/Widget_toolkit). It is part of [Oracle](http://en.wikipedia.org/wiki/Oracle_Corporation)'s [Java Foundation Classes](http://en.wikipedia.org/wiki/Java_Foundation_Classes) (JFC) — an [API](http://en.wikipedia.org/wiki/Application_programming_interface) for providing a [graphical user interface](http://en.wikipedia.org/wiki/Graphical_user_interface) (GUI) for Java programs.

Swing was developed to provide a more sophisticated set of GUI [components](http://en.wikipedia.org/wiki/Software_component) than the earlier [Abstract Window Toolkit (AWT)](http://en.wikipedia.org/wiki/Abstract_Window_Toolkit). Swing provides a native [look and feel](http://en.wikipedia.org/wiki/Look_and_feel) that emulates the look and feel of several platforms, and also supports a [pluggable look and feel](http://en.wikipedia.org/wiki/Pluggable_look_and_feel) that allows applications to have a look and feel unrelated to the underlying platform. It has more powerful and flexible components than AWT. In addition to familiar components such as buttons, check boxes and labels, Swing provides several advanced components such as tabbed panel, scroll panes, trees, tables, and lists.

* + - 1. Adding nurse data

When we use the software for the first time, we need to add the nurse information to the software database. As shown in the diagram 4.1, we add the nurse ID and nurse number to the database. Once we add the data, the added data will be shown like Diagram 4.2.

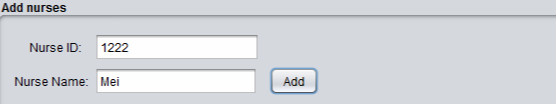


Diagram 4.1

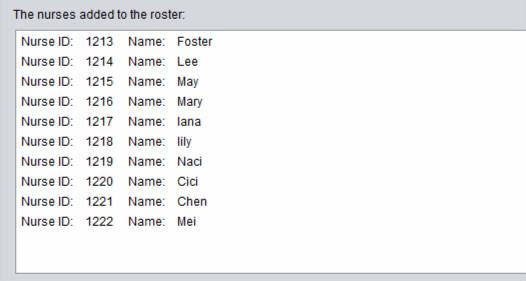


Diagram 4.2

* + - 1. Adding constraints

Set the nurse’s preference: As shown in diagram 4.3, the user can select the nurse and the work work shift preference from the drop down list. Once the constraint is added, the succeed information will be shown on the screen.

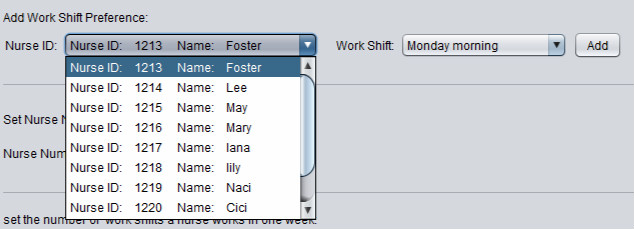


Diagram 4.3 Set the nurse’s preference

* + - 1. Other constraints

Other constraints can be added using the same method as shown in Diagram 4.4. Then the information about the added constraints will be shown like Diagram 4.5.

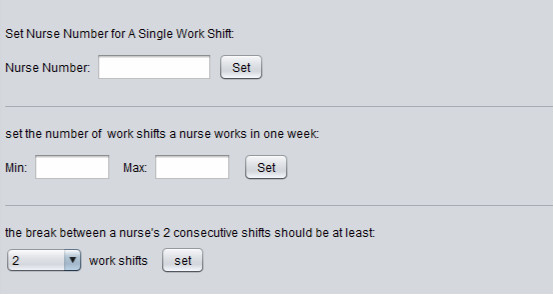


Diagram 4.4 Set the constraints

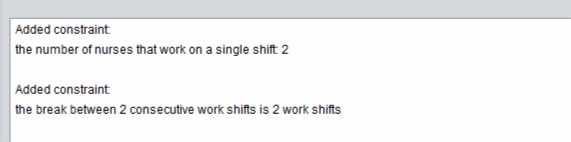


Diagram 4.5 information of the constraints

* + - 1. Generated Roster

Finally when we click the button to generate the roster, a gann-chart that shows the rosters of all the nurses will be generated and printed on the screen like Diagram 4.6.

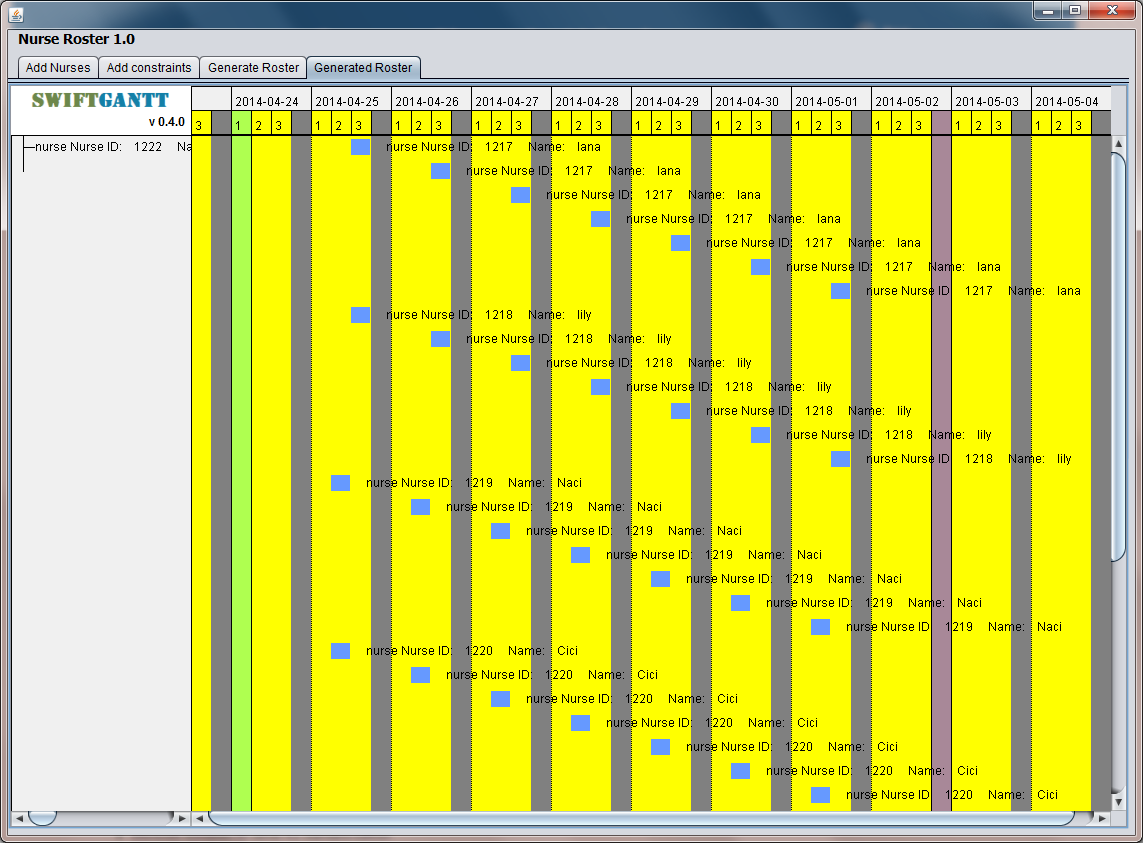


Diagram 4.6 The nurse roster

* + 1. The interaction between the GUI and the solver

In the code, NurseRotaUI is the main GUI class. StaffRota is the main logic class for searching the roster. In theNurseRoataUI class, StaffRota class object is called. And it will search for the solution according to the constraints.

Nurse\_data is an ArrayList object that is used to store all the nurse data. ConstraintDisplay is an ArrayList object that is used to store the information about the added constraints. Once the constraints is added, then the appropriate operation information will be stored in the ArrayList object ConstraintDisplay.

[1] Theatre Staff Rostering for the Edinburgh Festival :A Constraint Programming Approach

[2] Nurse Rostering with Genetic Algorithms

[3] **A Flexible Mixed Integer Programming framework for Nurse Scheduling**

**[4] railway staff rostering**

**[5]** **Mixed Integer Linear Programming** [**http://wenku.baidu.com/link?url=T0jaITdP018TMm0QrNe3E0TaYcDZVXY85Uto4MsXQFC3w-EdJDAEOdA7rvIQtor7u\_H\_CXipBYeWH3zGM1OcSzXKJr12c9oH73WPOuqMzFa**](http://wenku.baidu.com/link?url=T0jaITdP018TMm0QrNe3E0TaYcDZVXY85Uto4MsXQFC3w-EdJDAEOdA7rvIQtor7u_H_CXipBYeWH3zGM1OcSzXKJr12c9oH73WPOuqMzFa)

**[6]Integer Linear Programming**

<http://en.wikipedia.org/wiki/Integer_linear_programming>

[7] Artificial intelligence a modern approach stuart russel peter norvig

[8] Genetic algorithm <http://en.wikipedia.org/wiki/Genetic_algorithms>

[9] Whitley, Darrell (1994). "A genetic algorithm tutorial". *Statistics and Computing* **4** (2): 65–85. [doi](http://en.wikipedia.org/wiki/Digital_object_identifier):[10.1007/BF00175354](http://dx.doi.org/10.1007%2FBF00175354).

[10] Mitchell, Melanie (1996). *An Introduction to Genetic Algorithms*. Cambridge, MA: MIT Press. [ISBN](http://en.wikipedia.org/wiki/International_Standard_Book_Number) [9780585030944](http://en.wikipedia.org/wiki/Special:BookSources/9780585030944).

[11] Crew Rostering problem: a random key genetic algorithm with local search Ravi Kanth Rachakonda, B. Tech

[12] A. Schaerf. A survey of automated timetabling. Technical Report CS-R9567, CWI – Centrum voor Wiskunde en informatica, Amsterdam, The Netherlands, 1995.

[13] F. Azevedo and P. Barahona. Timetabling in constraint logic programming. In 2nd World Congress on Expert Systems, Estoril, Portugal, 1994.

[14] M. Henz and J. Wurtz. Using Oz for college timetabing. In first International Conference on the Practice and Theory of Automated Timetabling, pages 283-296, Edinburgh, Uk, 1995.

[15] H. Meyer auf’m Hofe. ConPlan/SIEDAplan: Personnel assignment as a problem of hierarchical constraint satisfaction. In 3rd International Conference on the Practical Application of Constraint Technology, pages 257-272, London, 1997. Practical Application Company Ltd.

[16] S. Abdennadher and M.Marte. University course timetabling using Constraint Handling Rules. Journal of Applied Artificial Intelligence.

[17] essentials of constraint programming

[18] wiki JaCop <http://en.wikipedia.org/wiki/JaCoP_(solver)>

[19] graph coloring wiki <http://en.wikipedia.org/wiki/Graph_coloring>

## [20] CSP tutorial <http://4c.ucc.ie/web/outreach/tutorial.html>

[21] Employee scheduling software wiki http://en.wikipedia.org/wiki/Employee\_scheduling\_software

[22] Nurse scheduling problem http://en.wikipedia.org/wiki/Nurse\_scheduling\_problem

[23] NP-problem mathworld WolframAlpha <http://mathworld.wolfram.com/NP-Problem.html>

[24] evolutionary algorithm for nurse scheduling problem ahmad jan Hokkaido university

[25] the scheduling problem in essay constraint programming for nurse scheduling georges weil

[26] big o notation http://en.wikipedia.org/wiki/Big\_O\_notation

[27] A beginner’s guide to big o notation http://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/

[28] what is iterative model-advantages, disadvantages and when to use it

<http://istqbexamcertification.com/what-is-iterative-model-advantages-disadvantages-and-when-to-use-it/>

Appendix

## Appendix A JaCoP constraints

### A.1 Primitive constraints

|  |  |
| --- | --- |
| **Constraint** | **JaCoP specification** |
|  |  |
| *X*= *Const* | XeqC(X, Const) |
| *X*= *Y* | XeqY(X, Y) |
| *X≠Const* | XneqC(X, Const) |
| *X≠Y* | XneqY(X, Y) |
| *X > Const* | XgtC(X, Const) |
| *X > Y* | XgtY(X, Y) |
| *X*≥ *Const* | XgteqC(X, Const) |
| *X*≥ *Y* | XgteqY(X, Y) |
| *X < Const* | XltC(X, Const) |
| *X < Y* | XltY(X, Y) |
| *X*≤ *Const* | XlteqC(X, Const) |
| *X*≤ *Y* | XlteqY(X, Y) |
| *X*⋅ *Const*= *Z* | XmulCeqZ(X, Const, Z) |
| *X*⋅ *Y*= *Z* | XmulYeqZ(X, Y, Z) |
| *X*÷ *Y*= *Z* | XdivYeqZ(X, Y, Z) |
| *X* mod*Y*= *Z* | XmodYeqZ(X, Y, Z) |
| *X*+ *Const*= *Z* | XplusCeqZ(X, Const, Z) |
| *X*+ *Y*= *Z* | XplusYeqZ(X, Y, Z) |
| *X*+ *Y*+ *Const*= *Z* | XplusYplusCeqZ(X, Y, Const, Z) |
| *X*+ *Y*+ *Q*= *Z* | XplusYplusQeqZ(X, Y, Q, Z) |
| *X*+ *Const*≤ *Z* | XplusClteqZ(X, Const, Z) |
| *X*+ *Y*≤ *Z* | XplusYlteqZ(X, Y, Z) |
| *X*+ *Y > Const* | XplusYgtC(X, Y, Const) |
| *X*+ *Y*+ *Q > Const* | XplusYplusQgtC(X, Y, Q, Const) |
| |*X*| = *Y* | AbsXeqY(X, Y) |
| *XY*= *Z* | XexpYeqZ(X, Y, Z) |
|  |  |

### A.2 Set constraints

|  |  |
| --- | --- |
| **Constraint** | **JaCoP specification** |
|  |  |
| *e*∈ *A* | EinA(e, A) |
| *S*1 = 2 | AeqB(S1, S2) |
| *S*1 ⊆ *S*2 | AinB(S1, S2) |
| *S*1 ⋃ *S*2 = *S*3 | AunionBeqC(S1, S2, S3) |
| *S*1 ⋂ *S*2 = *S*3 | AintersectBeqC(S1, S2, S3) |
| *S*1 \ *S*2 = *S*3 | AdiffBeqC(S1, S2, S3) |
| *S*1 *<> S*2 | AdisjointB(S1, S2) |
| Match | Match(Set, VarArray) |
| #*S*1 = *X* | CardAeqX(S, X) |
| Weighted sum < S, W > = X | SumWeightedSet(S, W, X) |
| *Set*[*X*] = *Y* | ElementSet(X, Set, Y) |
|  |  |

### A.3 Logical, conditional and reified constraints

|  |  |
| --- | --- |
| **Constraint** | **JaCoP specification** |
|  |  |
| ¬*c* | Not(c); |
| *c*1 ⇔ *c*2 | Eq(c1, c2); |
| *c*1 ∧ *c*2 ∧⋅⋅⋅∧ *cn* | PrimitiveConstraint[] c = {c1, c2, …cn}; |
|  | And(c); |
|  | or |
|  | ArrayList c = |
|  | new ArrayList(); |
|  | c.add(c1); c.add(c2); …c.add(cn); |
|  | And(c); |
| *c*1 ∨ *c*2 ∨⋅⋅⋅∨ *cn* | PrimitiveConstraint[] c = {c1, c2, …cn}; |
|  | Or(c); |
|  | or |
|  | ArrayList c = |
|  | new ArrayList(); |
|  | c.add(c1); c.add(c2); …c.add(cn); |
|  | Or(c); |
| *X*in*Dom* | In(X, Dom); |
| *c*⇔ *B* | Reified(c, B); |
| *c*⇔¬*B* | Xor(c, B); |
| if *c*1then*c*2 | IfThen(c1, c2); |
| if *c*1then*c*2else*c*3 | IfThenElse(c1, c2, c3); |
|  |  |
| **Boolean operations on variables** |  |
|  | BooleanVar[] b = {b1, b2, …, bn}; |
|  | or |
|  | ArrayList b = new ArrayList<="" span=""> |
|  | b.add(b1); b.add(b2); …b.add(bn); |
|  | BoolanVariable result = new BooleanVar(store, "result"); |
| *result*= *b*1 ∧ *b*2 ∧⋅⋅⋅∧ *bn* | AndBool(b, result) |
| *result*= *b*1 ∨ *b*2 ∨⋅⋅⋅∨ *bn* | OrBool(b, result) |
| *result*= *b*1 ⊕ *b*2 | XorBool(b1, b2, result) |
| *result*= *b*1 → *b*2 | IfThenBool(b1, b2, result) |
| *result*= *b*1 == *b*2 == ⋅⋅⋅ == *bn* | EqBool(b, result) |
|  |  |

### A.4 Global constraints

*x*1 + *x*2 + ⋅⋅⋅ + *xn* = *sum*

IntVar[] x = {x1, x2, …, xn};  
IntVar sum = new IntVar(…)  
Sum(x, sum);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
IntVar sum = new IntVar(…)  
Sum(x, sum);

*w*1 ⋅ *x*1 + *w*2 ⋅ *x*2 + ⋅⋅⋅ + *wn* ⋅ *xn* = *sum*

IntVar[] x = {x1, x2, …, xn};  
IntVar sum = new IntVar(…)  
int[] w = {w1, w2, …, wn};  
SumWeight(x, w, sum);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
IntVar sum = new IntVar(…)  
ArrayList w=new ArrayList();  
w.add(w1); w.add(w1); …w.add(wn);  
SumWeight(x, w, sum);

**alldifferent**([*x*1*,x*2*,…,xn*])

IntVar[] x = {x1, x2, …, xn};  
Alldifferent(x);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
Alldifferent(x);

**alldiff**([*x*1*,x*2*,…,xn*])

IntVar[] x = {x1, x2, …, xn};  
Alldiff(x);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
Alldiff(x);

**alldistinct**([*x*1*,x*2*,…,xn*])

IntVar[] x = {x1, x2, …, xn};  
Alldistinct(x);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
Alldistinct(x);

**among**([*x*1*,x*2*,…,xn*]*,val,count*)

IntVar[] x = {x1, x2, …, xn};  
IntervalDomain val = new IntervalDomain(k,l);  
IntVar count = new IntVar(…);  
Among(x, val, count);

**amongVar**([*x*1*,x*2*,…,xn*]*,*[*y*1*,y*2*,…,ym*]*,count*)

IntVar[] x = {x1, x2, …, xn};  
IntVar[] y = {y1, y2, …, ym};  
IntVar count = new IntVar(…);  
Among(x, y, count);

**assignment**([*x*1*,x*2*,…,xn*]*,*[*y*1*,y*2*,…,yn*])

IntVar[] x = {x1, x2, …, xn};  
IntVar[] y = {y1, y2, …, yn};  
Assignment(x, y);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
ArrayList y = new ArrayList();  
y.add(y1); y.add(y2); …y.add(yn);  
Assignment(x, y);

**circuit**([*x*1*,x*2*,…,xn*])

IntVar[] x = {x1, x2, …, xn};  
Circuit(Store, x);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
Circuit(Store, x);

**count**(*value,*[*x*1*,x*2*,…,xn*]*,var*)

int value = …;  
IntVar var = new IntVar(…);  
IntVar[] x = {x1, x2, …, xn};  
Count(x, var, value);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
Count(x, var, value);

**cumulative**([*t*1*,t*2*,…,tn*]*,*[*d*1*,d*2*,…,dn*]*,*[*r*1*,r*2*,…,rn*]*,ResourceLimit*)

IntVar[] t ={t1, t2, …, tn};  
IntVar[] d ={d1, d2, …, dn};  
IntVar[] r = {r1, r2, …, rn};  
IntVar Limit = new IntVar(…);  
Cumulative(t, d, r, Limit);[2](http://jacopguide.osolpro.com/guideJaCoP5.html#fn2x8)  
or using ArrayList

**diff2**([[*x*1*,y*1*,dx*1*,dy*1]*,…,*[*xn,yn,dxn,dyn*]])

IntVar[][] r = {{x1,y1,dx1,dy1}, …,  
{xn,yn,dxn,dyn}};  
Diff(r); or Diff2(Store, r);[1](http://jacopguide.osolpro.com/guideJaCoP4.html#fn1x8)  
or using ArrayList<arraylist></arraylist

**or**

**diff2**([*x*1*,…,xn*]*,*[*y*1*,…,yn*]*,*[*dx*1*,…,dxn*]*,*[*dy*1*,…,dyn*])

IntVar[] x = {x1, …, xn};  
IntVar[] y = {y1, …, yn};  
IntVar[] dx = {dx1, …, dxn};  
IntVar[] dy = {dy1, …, dyn};  
Diff(x, y, dx, dy); or Diff2(Store, x, y, dx, dy);[1](http://jacopguide.osolpro.com/guideJaCoP4.html#fn1x8)  
or using ArrayList

**distance**(*x,y,dist*)

IntVar x, y, dist;  
Distance(x, y, dist);

**element**(*Index,*[*n*1*,n*2*,…,nn*]*,V alue*)

IntVar Index, Value;  
int[] i = {n1, n2, …, nn };  
Element(Index, i, Value);

**element**(*Index,*[*x*1*,x*2*,…,xn*]*,V alue*)

IntVar Index, Value;  
IntVar[] x = {x1, x2, …, xn };  
Element(Index, x, Value);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
Element(Index, x, Value);

**extensionalSupport**([*x*1*,x*2*,…,xn*]*,*{{1*,*2*,…,n*}*,*{*…*}*,…,*{*…*}})

**extensionalConflict**([*x*1*,x*2*,…,xn*]*,*{{1*,*2*,…,n*}*,*{*…*}*,…,*{*…*}})

IntVar[] x = {x1, x2, …, xn};  
int[][] intTuple = {{…}, …});  
ExtensinalSupportVA(x,intTuple);  
or  
ExtensinalConflictVA(x,intTuple);  
or  
ExtensinalSupportSTR(x,intTuple);  
or  
ExtensinalSupportMDD(x,intTuple);

**gcc**([*x*1*,x*2*,…,xn*]*,*[*y*1*,y*2*,…,ym*])

IntVar[] x = {x1, x2, …, xn};  
IntVar[] y = {y1, y2, …, ym};  
GCC(x, y);

**min**([*x*1*,x*2*,…,xn*]*,Xmin*)

IntVar[] x = {x1, x2, …, xn};  
Min(x, Xmin);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
Min(x, Xmin);

**max**([*x*1*,x*2*,…,xn*]*,Xmax*)

IntVar[] x = {x1, x2, …, xn};  
Max(x, Xmin);  
or  
ArrayList x = new ArrayList();  
x.add(x1); x.add(x2); …x.add(xn);  
Max(x, Xmax);

**knapsack**(*profits,weights,quantity,knapsackCapacity,knapsackProfit*)

int[] profits = {p1, p2, …, pn};  
int[] weights = {w1, w2, …, wn};  
IntVar[] quantity = {q1, q2, …, qn};  
IntVar knapsackCapacity = new IntVar(…);  
IntVar knapsackProfit = new IntVar(…);  
Knapsack(profits, weights, quantity, knapsackCapacity, knapsackProfit);

**geost**(*objects,constraints,shapes*)

IntVar xOrigin = new IntVar(store, "x1", 0, 20);  
IntVar yOrigin = new IntVar(store, "y1", 0, 5);  
IntVar shapeNo = new IntVar(store, "s1", 1, 1);  
IntVar startGeost = new IntVar(store, "start"+1, 0, 0);  
IntVar durationGeost = new IntVar(store, "duration"+1, 1, 1);  
IntVar endGeost = new IntVar(store, "end"+1, 1, 1);  
IntVar[] coords = {xOrigin, yOrigin};  
int objectId = 1;  
GeostObject o = new GeostObject(objectId, coords, shapeNo, startGeost, durationGeost, endGeost);  
ArrayList objects = new ArrayList();  
objects.add(o);  
int[] origin = {0, 0};  
int[] length = {10, 2};  
Shape shape = new Shape(j, new DBox(origin, length));  
ArrayList shapes = new ArrayList();  
shapes.add(shape);  
int[] dimensions = {0, 1};  
NonOverlapping constraint = new NonOverlapping(objects, dimensions);  
ArrayList constraints = new ArrayList();  
constraints.add(constraint);  
store.impose(new Geost(objects, constraints, shapes));

**regular**(*fsm,*[*x*1*,x*2*,…,xn*])

FSM fsm = new FSM();  
IntVar[] x = {x1, x2, …, xn};  
Regular(fsm, x);

**sequence**([*x*1*,x*2*,…,xn*]*,set,q,min,max*)

IntVar[] x = {x0, x1 …xn};  
IntervalDomain set = new IntervalDomain(…);  
int q, main, max;  
Sequence(x, set, q, min, max);

**stretch**(*values,min,max,*[*x*1*,x*2*,…,xn*])

int[] values, main, max;  
IntVar[] x = {x0, x1 …, xn};  
Stretch(values, min, max, x);

**values**([*x*1*,x*2*,…,xn*]*,count*)

IntVar[] x = {x0, x1 …, xn};  
IntVar count = new IntVar(…);  
Values(x, count);

**lex**([[*x*11*,x*12*,…,x*1*n*]*,…,*[*xk*1*,xk*2*,…,xkm*]])

IntVar[][] x = {{x0, x1 …, xn}, …};  
Lex(x);  
or  
Lex(x, true);

**soft-alldifferent**([*x*1*,x*2*,…,xn*]*,cost,violation***\_***measure*)

IntVar[] x = {x0, x1 …, xn};  
IntVar count = new IntVar(…);  
SoftAlldifferent(x, cost, ViolationMeasure.DECOMPOSITION\_BASED);  
or  
SoftAlldifferent(x, cost, ViolationMeasure.VARIABLE\_BASED);

**soft-GCC**([*x*1*,x*2*,…,xn*]*,hardCounters,countedV alues,softCounters,cost,violation***\_***measure*)

IntVar[] x = {x0, x1 …, xn};  
IntVar[] hardCounters = {h1, h2, …, hn};  
int[] countedValues = {v1, v2, …, vn};  
IntVar[] softCounters = {s1, s2, …, sn};  
SoftGCC(x, hardCounters, countedValues, softCounters, cost, ViolationMeasure.VALUE\_BASED); or  
other constructors (see API specification).

**Appendix B  
JaCoP search methods**

**B.1 Variable and value selection for FDVs**

* **value selection methods**

|  |  |
| --- | --- |
| **Indomain method** | **Description** |
|  |  |
| IndomainMin | selects a minimal value from the current domain of FDV |
| IndomainMax | selects a maximal value from the current domain of FDV |
| IndomainMiddle | selects a middle value from the current domain of FDV |
|  | and then left and right values |
| IndomainRandom | selects a random value from the current domain of FDV |
| IndomainSimpleRandom | faster than IndomainRandom but does not achieve uniform probability |
| IndomainList | uses values in an order provided by a programmer |
|  | if values not specified uses default indomain method |
| IndomainHierarchical | uses indomain method based provided variable-indomain mapping |
|  |  |

* **variable selection methods**

|  |  |
| --- | --- |
| **Comparator** | **Description** |
|  |  |
| SmallestDomain | selects FDV which has the smallest domain size |
| MostConstrainedStatic | selects FDV which has most constraints assign to it |
| MostConstrainedDynamic | selects FDV which has the most pending constraints assign to it |
| SmallestMin | selects FDV with the smallest value in its domain |
| LargestDomain | selects FDV with the largest domain size |
| LargestMin | selects FDV with the largest value in its domain |
| SmallestMax | selects FDV with the smallest maximal value in its domain |
| MaxRegret | selects FDV with the largest difference between the smallest |
|  |  |

**B.2 Variable and value selection for set variables**

* **value selection methods**

|  |  |
| --- | --- |
| **Indomain method** | **Description** |
|  |  |
| IndomainSetMin | selects a minimal value from not yet assigned values for set variable |
| IndomainSetMax | selects a maximal value from not yet assigned values for set variable |
| IndomainSetRandom | selects a random value from not yet assigned values for set variable |
|  |  |

* **variable selection methods**

|  |  |
| --- | --- |
| **Comparator** | **Description** |
|  |  |
| MinCardDiff | selects set variable which has the smallest difference in cardinality |
|  | between lub and glb. |
| MaxCardDiff | selects set variable which has the greatest difference in cardinality |
|  | between lub and glb. |
| MinGlbCard | selects set variable which has the glb with the smallest cardinality. |
| MaxGlbCard | selects set variable which has the glb with the greatest cardinality. |
| MinLubCard | selects set variable which has the lub with the smallest cardinality. |
| MaxLubCard | selects set variable which has the lub with the greatest cardinality. |
| MostConstrainedStatic | selects set variable which has most constraints assign to it. |
| MostConstrainedDynamic | selects set variable which has the most pending constraints assign to it. |
|  |  |

**B.3 Search methods**

We specify search methods for finite domain variables (IntVar). Similar methods can be defined for set variables (SetVar).

* **Search for a single solution with *list of variables***

   IntVar[] var;   
   ...   
   Search<IntVar> label = new DepthFirstSearch<IntVar>();   
   SelectChoicePoint<IntVar> select = new SimpleSelect<IntVar>(   
                                            *var*,   
                                            *varSelect*,   
                                            *tieBreakerVarSelect*   
                                            *indomain*);   
   boolean result = label.labeling(store, select);

* **Search for a single solution with *list of list of variables***

   IntVar[][] var;   
   ...   
   Search<IntVar> label = new DepthFirstSearch<IntVar>();   
   SelectChoicePoint<IntVar> select =   
                              new SimpleMatrixSelect<IntVar>(   
                                            *var*,   
                                            *varSelect*,   
                                            *tieBreakerVarSelect*   
                                            *indomain*);   
   boolean result = label.labeling(store, select);

* **Search for all solutions**

additional switches for search for all solutions.

   label.getSolutionListener().searchAll(true);   
   // record solutions; if not set false   
   label.getSolutionListener().recordSolutions(true);   
   boolean result = label.labeling(store, select);

To be able to print found solutions during search the following solution listener has to be added to the search.

   label.setSolutionListener(new PrintOutListener<IntVar>());

The found solutions can also be printed after search is completed using the following statement.

   label.printAllSolutions();

* **Search for optimal solution**

   IntVar cost;   
   ...   
   boolean result = label.labeling(store, select, cost);

**B.4 Important methods for search plug-ins**

* **solution listener**– SimpleSolutionListener

important methods

* + printAllSolutions()
  + getSolutions()
  + solutionsNo()
  + recordSolutions(boolean status)
  + searchAll(boolean status)
  + executeAfterSolution(Search search, SelectChoicePoint select)
  + setChildrenListeners(SolutionListener child)
* **time-out listener**– one can set customized time-out listener that implements TimeOutListener interface to perform specific actions at time-out (e.g., print information). Method executedAtTimeOut(int solutionsNo) will be executed at time-out.