New approaches to nurse rostering benchmark instances

Edmund K. Burke[1], Tim Curtois[2]

[1]Department of Computing and Mathematics, University of Stirling, Cottrell Building, Stirling FK9 4LA. UK, e.k.burke@stir.ac.uk
[2]School of Computer Science, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham. NG8 1BB. UK, tim.curtois@nottingham.ac.uk

Abstract

This paper presents the results of developing a branch and price algorithm and an ejection chain method for nurse rostering problems. The approach is general enough to be able to apply it to a wide range of benchmark nurse rostering instances. The majority of the instances are real world applications. They have been collected from a variety of sources including industrial collaborators, other researchers and various publications. The results of entering these algorithms in the 2010 International Nurse Rostering Competition are also presented and discussed. In addition, incorporated within both algorithms is a dynamic programming method which we present. The algorithm contains a number of heuristics and other features which make it very effective on the broad rostering model introduced.

1. Introduction

Rostering problems are found in a wide range of workplaces and industries including healthcare, manufacturing, transportation, emergency services, call centres and many more. Using a computational search algorithm to address these problems results in cost savings and better work schedules. As such, rostering problems in various forms have received a large amount of research attention over the years. This body of research grew steadily throughout the 1960's, 70's and 80's and then accelerated in growth as more powerful desktop personal computers became commonplace in workplaces during the 1990's. As the computational and processing power has grown so has the range and complexity of algorithms that can be applied and the size and complexity of the instances that can be solved. For an overview of rostering problems and solution methodologies see [17]. A very large annotated bibliography of publications relating to staff scheduling is also provided by [16]. For a literature review specifically aimed at the nurse rostering problem, see [11].

As these review papers show, many different approaches have been used to solve nurse rostering problems. These include metaheuristics [5, 8, 9, 20, 28], constraint programming [14, 27, 36], mathematical programming [2, 3], other artificial intelligence techniques (such as case-based reasoning [4]) and hybrid approaches [12, 34]. Each method has strengths and weaknesses. For example, as will be shown in this paper, a mathematical programming approach may be able to solve some instances to optimality extremely quickly but on other instances it may take infeasible amounts of time or use too much memory. A metaheuristic, on the other hand, may be able to find a good solution to difficult instances quite quickly but may not be able to find the optimal solution to another instance which an exact method can solve very quickly. An obvious solution to this well-known phenomenon is to combine and hybridise different techniques. This is one of the principles behind adaptive approaches such as hyperheuristics.

The aim of this paper, however, is to provide new results (upper bounds and lower bounds) for a large collection of diverse rostering benchmark instances. This is the first occasion that a branch and price method has been applied to these instances. We also introduce the dynamic programming algorithm which is at the core of the branch and price method and present a general rostering model which we used for all the instances tested.

Branch and price is a branch and bound method in which each node of the branch and bound tree is a linear programming relaxation which is solved using column generation. The column generation consists of a restricted master problem and a pricing problem. Solving the pricing problem provides new negative reduced cost columns to add to the master problem. The pricing problem can be considered as the problem of finding the optimal work schedule for an individual employee but with the addition of dual costs, that is, additional (possibly negative) costs based on which shift assignments are made or not made. In non-root nodes of the branch and bound tree, there may also be additional branching constraints on certain assignments that must or must not be made.

Although this is the first time that branch and price has been applied to these instances, it has previously been used on the nurse rostering problem [18, 22, 25, 26]. All these earlier applications have similar structure and the same structure is adopted here. The master problem is modelled as a set covering problem and solved using a linear programming method such as the simplex method. The pricing problem is formulated as a resource constrained shortest path problem and solved using a dynamic programming approach. The branch and bound tree is generally too large for a complete search and so heuristic, constraint branching schemes are adopted in which branching is performed on shift assignments in the roster. Although the dynamic programming algorithms all use the same principles (dominance pruning and bound pruning), the actual implementations are dependent on the constraints and objectives present in the pricing problem. For a recent overview of column generation see [24] and for further reading on resource constrained shortest path problems see [21].

In the next section, we discuss the challenge of modelling such a wide variety of instances and how it was solved. In section 3, we introduce the benchmark instances and section 4 presents the branch and price algorithm. Section 5 contains the results of applying the algorithms to the benchmark instances. In section 6, we discuss the International Nurse Rostering Competition and finish with conclusions in section 7.

2. Modelling the Problem

One of the most significant challenges in solving a large diverse collection of instances is developing a model which can be used for all the instances with their varying types of constraints and objectives. In all the instances, there are common types of constraints/objectives which are relatively straightforward to model. These include the cover constraints (ensuring that there is a correct or a preferable number of employees assigned to each shift). However, the types of constraints that can be present in each employee's work schedule can vary significantly from instance to instance. This is due to the reality of each workplace having its own set of rules and requirements defined by different employers, employees, unions and national legislation. Furthermore, each employee often has a different contract to reflect such features as full-time employment, part-time employment and night shift working. To

provide a system which can incorporate these variations, we developed a general constraint based on pattern/string matching or more specifically regular expressions. Regular expressions are a powerful yet compact way of specifying patterns to be found or matched. They are commonly used in Computer Science and so we will not expand upon the subject here. Instead, we refer readers to one of the many textbooks on the subject such as [19]. Using a regular expression constraint in staff scheduling problems appears to be a natural fit and this is not the first example of its application to these type of problems [13, 15, 31]. However, in order to fully include all the variations in the instances we used, our approach is broader than some of this earlier work. First though, we will illustrate by example how this constraint can be applied in staff rostering problems. The basic idea behind the constraint is to consider the employee's work schedule as the 'search text' containing the regular expressions to be matched and the regular expressions to be matched are sequences of shifts. After presenting the examples below, we also provide a figure to illustrate how the constraint works in practice. The figures show a short section of a single employee's schedule. The coloured squares labelled E, D and N represent early, day and night shifts respectively. The highlighted days show where the regular expression in question has been matched.

Example 1: If a night shift (*N*) can only be followed by another night shift or a day off then it could be modelled by the constraint "maximum zero matches of the pattern '*N* followed by any shift other than *N*'". Note that we use the expression "maximum zero" here as another way of saying this pattern must not appear at all. We use this expression instead though because all the matches are expressed as either a maximum or minimum number of matches in order to provide more modelling power.

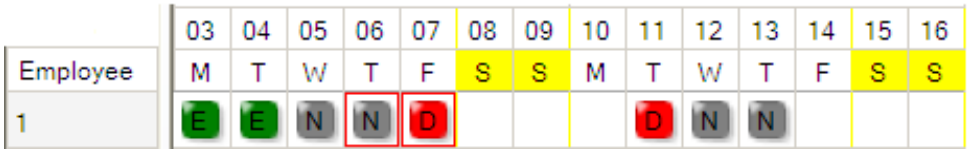

Figure 1 Violation of constraint example 1

Example 2: If an employee must not work more than five consecutive shifts then it could be modelled by the constraint "maximum zero matches of the pattern '*Any*, *Any*, *Any*, *Any*, *Any*, *Any*'" where *Any* is any shift (that is, not a day off).

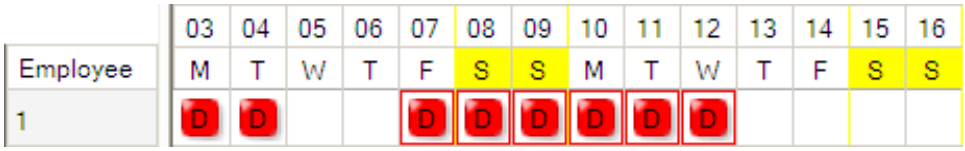

Figure 2 Violation of constraint example 2

Example 3: If an employee must have a minimum of two consecutive night shifts then the constraint would be "maximum zero matches of the pattern 'anything but *N*, followed by *N*, followed by anything but *N*'".

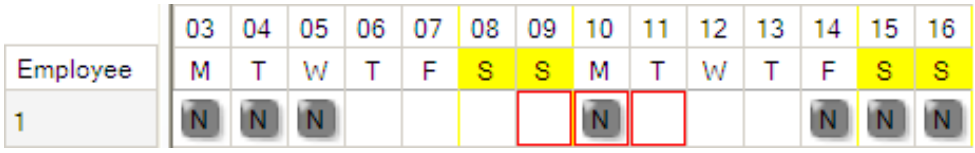

Figure 3 Violation of constraint example 3

As can be seen, the constraint is based on the idea of string/pattern matching. However, it is more like a regular expression and extends some of the previous work because we also allow:

- Grouping: Matching one of a group of shifts at a point in the sequence.
- Negation: Matching anything but a specific shift or group of shifts at a point in the sequence.
- Alternation: Matching multiple patterns.
- Quantifiers: The pattern(s) must appear a minimum or maximum number of times.
- Restricting the search text to a specific region of the work schedule.
- Only matching a pattern if it starts on a particular day in the work schedule.

This enables us to model some of the more complicated constraints such as those relating to weekend work or constraints that only apply between certain dates in the planning period. Using this general, regular expression constraint we can model many of the constraints found in staff scheduling problems. An example list is provided below.

- Minimum/maximum consecutive work days
- Minimum/maximum consecutive non-work days
- Day on/off requests
- Shift on/off requests
- Minimum/maximum number of shifts (optionally within a specific time frame)
- Minimum/maximum number of shifts of a specific type (optionally within a specific time frame)
- Minimum/maximum number of consecutive shifts of a specific type (optionally within a specific time frame)
- Days off after a series of shifts of a specific type
- Shift rotations (which shifts can follow which shifts)
- Minimum/maximum shift rotations
- Minimum/maximum number of weekends worked (or any group of days/dates)
- Minimum/maximum number of consecutive weekends worked

Although all these constraints can be modelled using the regular expression constraint there are though some constraints which cannot. In particular, this includes those relating to the minimum and maximum amount of work time an employee can be assigned. For this type of constraint we developed a general constraint called *Workload* which is simply a minimum or maximum amount of work time which can be assigned to a single employee between any two dates in the planning horizon.

A mathematical model of the problem is now presented.

Sets

E = Employees to be scheduled, $e \in E$.

T = Shift types to be assigned, $t \in T$.

D = Days in the planning horizon, $d \in \{1,...,|D|\}$.

R_e = Regular expressions for employee e, $r \in R_e$

W_e = Workload limits for employee e, $w \in W_e$

Parameters

RU_{er}^{max} = Maximum number of matches of regular expression r in the work schedule of employee e

RL_{er}^{min} = Minimum number of matches of regular expression r in the work schedule of employee e

RW_{er} = Weight associated with regular expression r for employee e

WU_{ew}^{max} = Maximum number of hours to be assigned to employee e within the time period defined by workload limit w

WL_{ew}^{min} = Minimum number of hours to be assigned to employee e within the time period defined by workload limit w

WW_{ew} = Weight associated with workload limit w for employee e

CU_{td}^{max} = Maximum number of shifts of type t required on day d

CL_{td}^{min} = Minimum number of shifts of type t required on day d

CW_{td} = Weight associated with the cover requirements of shift type t on day d

Variables

x_{etd} = 1 if employee e is assigned shift type t on day d, 0 otherwise

RN_{er} = The number of matches of regular expression r in the work schedule of employee e

WN_{ew} = The number of hours assigned to employee e within the time period defined by workload limit w

CN_{td} = The number of shifts of type t assigned on day d

Constraints

Employees can be assigned only one shift per day

$$\sum_{t \in T} x_{etd} \leq 1, \qquad \forall e \in E, d \in D \tag{1a}$$

Objective Function

$$\text{Min } f(x) = \sum_{e \in E} \sum_{i=1}^{4} f_{e,i}(x) + \sum_{t \in T} \sum_{d \in D} \sum_{i=5}^{6} f_{t,d,i}(x) \tag{1b}$$

where

$$f_{e,1}(x) = \sum_{r \in R_e} \max\{0, (RN_{er} - RU_{er}^{\max}) RW_{er}\} \tag{1c}$$

$$f_{e,2}(x) = \sum_{r \in R_e} \max\{0, (RL_{er}^{\min} - RN_{er}) RW_{er}\} \tag{1d}$$

$$f_{e,3}(x) = \sum_{w \in W_e} \max\{0, (WN_{ew} - WU_{ew}^{\max}) WW_{ew}\} \tag{1e}$$

$$f_{e,4}(x) = \sum_{w \in W_e} \max\{0, (WL_{ew}^{\min} - WN_{ew}) WW_{ew}\} \tag{1f}$$

$$f_{t,d,5}(x) = \max\{0, (CL_{td}^{\min} - CN_{td}) CW_{td}\} \tag{1g}$$

$$f_{t,d,6}(x) = \max\{0, (CN_{td} - CU_{td}^{\max}) CW_{td}\} \tag{1h}$$

The objective function (1b) is a weighted sum of the soft constraints (1c-1h). (1c) and (1d) relate to the regular expression constraints, (1e) and (1f) relate to the workload constraints and (1g) and (1h) the cover constraints. When applying this model to an instance in which one of the employee's constraints or a cover constraints is a hard constraint we simply set the weight to a very high number (significantly higher than any of the other weights).

The significant advantage of this model is that it can incorporate the requirements of many different workplaces without needing to be extended. This means that the algorithm does not need to be modified when a new constraint is encountered as long as it can be modelled as a regular expression. For example, the benchmark instances discussed in the next section could be described as different problems because most of them have a different set of constraints and objectives. However, we were able to model them all using this single model.

3. Benchmark Instances

In order to validate our algorithms and encourage more competition and collaboration between researchers working on rostering we have built a collection of diverse and challenging benchmark instances. The collection has grown over several years and has been drawn from various sources such as industrial collaborators (including software companies and hospitals), scientific publications and other researchers. The

collection continues to grow, is currently drawn from thirteen different countries and the majority of the data sets are based on real world rostering scenarios. Table 1 lists the instances. As can be seen, they vary in the length of the planning horizon, the number of employees, the number of shift types and the number of skills. Each instance also varies in the number, priority and type of constraints and objectives present. The objectives were set by the organisation that provided the data. For example, some prefer to minimise overstaffing whereas other prefer to maximise staff satisfaction by setting the weights for those objectives higher instead.

Instance	Staff	Shift types	Length (days)	Skill types	Best known	Ref
Ozkarahan	14	2	7	2	0	[30]
Musa	11	1	14	3	175	[29]
Millar-2Shift-DATA1	8	2	14	1	0	[20]
Millar-2Shift-DATA1.1	8	2	14	1	0	[20]
LLR	27	3	7	1	301	[23]
Azaiez	13	2	28	2	0	[2]
GPost	8	2	28	1	5	
GPost-B	8	2	28	1	3	
QMC-1	19	8	28	1	13	
QMC-2	19	3	28	3	29	
WHPP	30	3	14	1	5	[36]
BCV-3.46.2	46	3	26	1	894	[6]
BCV-4.13.1	13	4	29	1	10	[6]
SINTEF	24	5	21	1	0	
ORTEC01	16	4	31	1	270	[8]
ORTEC02	16	4	31	1	270	[8]
ERMGH	41	4	42	2	779	
CHILD	41	5	42	1	2001	
ERRVH	51	8	42	2	149	
HED01	20	5	31	2	136	[33]
Valouxis-1	16	3	28	1	20	[35]
Ikegami-2Shift-DATA1	28	2	30	9	0	[20]
Ikegami-3Shift-DATA1	25	3	30	8	2	[20]
Ikegami-3Shift-DATA1.1	25	3	30	8	3	[20]
Ikegami-3Shift-DATA1.2	25	3	30	8	3	[20]
BCDT-Sep	20	4	30	1	100	[5]
MER	54	12	42	2	7081	

Table 1 Benchmark Instances

The instances are available for download from http://www.cs.nott.ac.uk/~tec/NRP/, where all the required information on each instance, best solutions, visualisations and other software is also available. The data set files are not included within this paper because of their large size.

Table 2 lists the instances used in the First International Nurse Rostering Competition. The instances were created by the competition organisers and not released before the competition. They are discussed further in Section 6.

Instance	Staff	Shift types	Days	Skills	Instance	Staff	Shift types	Days	Skills
sprint01	10	4	28	1	medium01	31	4	28	1
sprint02	10	4	28	1	medium02	31	4	28	1
sprint03	10	4	28	1	medium03	31	4	28	1
sprint04	10	4	28	1	medium04	31	4	28	1
sprint05	10	4	28	1	medium05	31	4	28	1
sprint06	10	4	28	1	medium_late01	30	4	28	1
sprint07	10	4	28	1	medium_late02	30	4	28	1
sprint08	10	4	28	1	medium_late03	30	4	28	1
sprint09	10	4	28	1	medium_late04	30	4	28	1
sprint10	10	4	28	1	medium_late05	30	5	28	2
sprint_late01	10	4	28	1	long01	49	5	28	2
sprint_late02	10	3	28	1	long02	49	5	28	2
sprint_late03	10	4	28	1	long03	49	5	28	2
sprint_late04	10	4	28	1	long04	49	5	28	2
sprint_late05	10	4	28	1	long05	49	5	28	2
sprint_late06	10	4	28	1	long_late01	50	5	28	2
sprint_late07	10	4	28	1	long_late02	50	5	28	2
sprint_late08	10	4	28	1	long_late03	50	5	28	2
sprint_late09	10	4	28	1	long_late04	50	5	28	2
sprint_late10	10	4	28	1	long_late05	50	5	28	2

Table 2 Competition Instances

4. The Branch and Price Algorithm

The implementation of the branch and price approach follows the previously published algorithms mentioned in section 1. However, quite a lot of time was spent improving the performance of the implementation. For example, profiling the algorithm reveals that during the column generation, typically about 5% of the computation time is spent re-solving the restricted master problem (using the simplex method) whereas the other 95% of the time is used in solving the sub-problems (generating the new columns using the dynamic programming algorithm). This meant that the performance of the algorithm could be most significantly improved through:

1) Reducing the number of calls to the pricing problem solver.
2) Improving the performance of the pricing problem solver.

Stabilisation (reducing the oscillation of the dual values) was particularly important and effective for the first. For the second, additional heuristics and bounding methods were very effective, especially exploiting the fact that it is not necessary to find the most negative reduced cost column each time (that is, the pricing problem does not have to be solved to optimality until it is necessary to show that there are no more negative reduced cost columns). These heuristics are discussed in more detail in section 4.1. For the stabilisation, we used the method presented in [32] which is relatively straightforward to implement and does not depend on instance specific parameters but was also very effective.

To solve the master problem we used the simplex method of the open-source, Coin-OR linear programming solver (clp) [1] which we found to be fast and stable.

Within a time limit, two different heuristic branching strategies are applied in the branch and bound tree to try and find new solutions (upper bounds). For the first strategy, we simply branch on the variables in the master problem by selecting the variable that is closest to one. This strategy often quickly provides an upper bound but this upper bound can usually be improved by the second branching strategy. In the second strategy, we branch on individual employee-shift assignments (constraint branching). At each node in the tree, we select the employee-shift assignment that has the value closest to one when summing all the master problem variables (columns) that contain this assignment. Columns that do not contain this assignment are then removed from the master problem and when the master problem is re-solved the pricing problem solver only generates columns which contain this assignment (and any other forced assignments from ancestor nodes in the tree). We carried out some initial experiments with branching on the most fractional assignments (closest to 0.5) instead. There did not appear to be much difference in solution quality but it was slightly slower on average so chose to branch on assignments closest to one.

The initial solution is provided by applying the variable depth search algorithm for five seconds. If a provable optimal solution (lower bound equal to upper bound) is not found within the time limit, then the best upper bound is returned.

4.1. The Pricing Problem Solver

We use a dynamic programming approach to solve the pricing problem. That is, to generate new columns where the columns are basically new work schedules (also called shift patterns) for individual employees. The problem can be classified as a resource constrained shortest path problem. Figure 4 shows an example graph for an instance with three shift types (**E**arly, **D**ay and **N**ight). A path consists of *n* connected nodes between the source and sink node where *n* is the number of days in the planning horizon. Each shift type and a day off represent the nodes that can be chosen on each day. Resources are collected along the path depending on which nodes are part of that path.

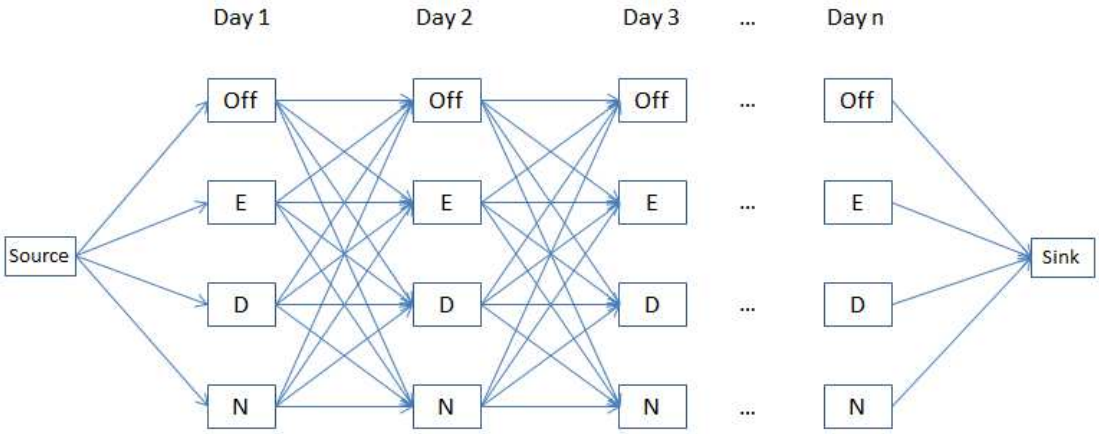

Figure 4 Example graph for the shortest path problem

The idea behind dynamic programming is to use bounding and dominance to prune paths/nodes that can be proven to be unnecessary to expand. Although dynamic

programming can be very effective at solving certain types of problem, in worst cases the number of paths can still grow exponentially.

An interesting feature of our implementation is that we solve the problem over a number of iterations where at each iteration the number of paths that can be expanded is restricted to a maximum and the paths to expand are selected heuristically. If at the end of an iteration the maximum limit was not reached then the problem was solved. Otherwise, we try again with a higher limit but possibly also with a new upper bound (i.e. the best solution found at the previous limit). We also resume the search at the point the limit was reached in order to avoid superfluous repetitions. An outline of the algorithm is provided by Figure 5. and discussed in more detail below.

```
1.  FUNCTION GenerateShiftPattern
    Returns:
     An array of solutions with an objective function value less than
     InitialUpperBound. The objective function value = cost of
     the pattern + dual costs for that pattern.
     If MustBeOptimal is true then it will return the pattern with the
     lowest objective function value.
     If there are no solutions with objective function value less than
     InitialUpperBound then it returns an empty array.
    Input parameters:
     The employee to generate the shift pattern for.
     Dual costs for each possible assignment on each day (may be negative).
     Branching constraints (assignments which must or must not be made).
     A bound (InitialUpperBound). The objective function value of any
     solutions returned must be less than this.
     A Boolean value (MustBeOptimal) to indicate whether it must return the
     optimal solution or the first set of solutions it finds with objective
     function value < InitialUpperBound.
2.  SET BestUpperBound := InitialUpperBound
3.  SET MaxArraySizes := { 32, 128, 512, 2048, 8192, infinity }
4.  Create four empty arrays (CurrentArray, NextArray, Solutions and BestSolutions)
5.  Create an empty shift pattern and make any assignments which must be made
    due to branching constraints (or other constraints) and add it to NextArray
6.  FOR each MaxArraySize in MaxArraySizes
7.    SET MaxArraySizeExceeded := false
8.    Clear the Solutions array
9.    FOR each day (Day) in the planning period
10.     SET CurrentArray as NextArray
11.     SET NextArray as an empty array
12.     FOR each partially completed shift pattern (Pattern) in CurrentArray
13.       FOR each possible shift assignment on Day (including a day off)
14.         Make a copy of Pattern (NewPattern) and add the assignment to NewPattern
15.         Calculate a lower bound for this pattern and check for any constraint
            violations. If there is a violation or the bound is >= BestUpperBound then
            discard NewPattern and GOTO LABEL TryNextAssignment (29.)
16.         IF Day is the last day in the planning period THEN
17.           Add NewPattern to Solutions
18.         ELSE
19.           Do dominance checks to see if NewPattern should be added to NextArray
              and if there are any patterns in NextArray that are dominated and can be
              removed
20.           IF NewPattern needs to be added THEN
21.             IF Adding NewPattern (and removing any dominated patterns) would cause
                  MaxArraySize to be exceeded THEN
22.               SET MaxArraySizeExceeded := true
23.               Test heuristically replacing a pattern in NextArray with NewPattern
24.               GOTO LABEL TryNextAssignment (29.)
25.             END IF
26.             Add NewPattern to NextArray and remove any that are dominated
27.           END IF
28.         END ELSE
29.         LABEL TryNextAssignment
30.       END FOR
31.     END FOR
32.     IF NextArray is empty THEN
33.       GOTO 46.
34.     END IF
35.   END FOR
36.   IF Solutions is not empty THEN
37.     IF MaxArraySizeExceeded = false OR MustBeOptimal = false THEN
38.       RETURN Solutions
39.     END IF
40.     SET BestSolutionCost := the lowest obj. function value in Solutions
41.     IF BestSolutionCost < BestUpperBound THEN
42.       SET BestUpperBound := BestSolutionCost
43.     END IF
44.   Clear the BestSolutions array and add the elements from Solutions
45.   ELSE
46.     IF MaxArraySizeExceeded = false THEN
47.       RETURN BestSolutions
48.     END IF
49.   END ELSE
50. END FOR
51. RETURN BestSolutions
52. END FUNCTION
```

Figure 5 Pseudocode for the shift pattern generating algorithm

The algorithm is able to solve the problem to proven optimality or just return the first set of solutions it finds with an objective function value below a bound. This bound and the flag indicating whether to solve it to optimality are passed as parameters to the algorithm. The other algorithm parameters are variables which may change between calls to the method: The dual costs (from the cover constraints) and any branching constraints. (The branching constraints are assignments which must or must not be made in the shift pattern because they are fixed in the branch and bound tree).

As already discussed, in the column generation algorithm it is not necessary to solve the pricing problem to optimality every time (that is, it does not need to find the most negative reduced cost column) but any negative reduced cost columns are acceptable.

The maximum array sizes to use at each iteration of the algorithm is set at step 3. In the pseudocode, the values shown are: `{ 32, 128, 512, 2048, 8192, infinity }` which is the setting used for the results shown in section 5. Some testing was performed varying the size of this set and the values in it but no clear best setting was found when tested over all instances. However, a general strategy of starting with small values which are solved very quickly before gradually moving to the larger values appears to work best.

At step 15 (i.e. after a shift assignment is made), a lower bound is calculated (a minimum objective function value) for a partially complete pattern by looking at the assignments already made, the objectives for that employee and the dual costs. This lower bound is then used to discard the pattern if it is greater than or equal to the best upper bound found so far. (Although not shown in the pseudocode this exact same procedure is also done at step 5 where other assignments are made due to branching constraints).

After creating a new partial pattern, it is necessary to compare this pattern to the partial patterns in *NextArray* for dominance. The dominance checking simply involves comparing the patterns by examining the values of the variables RN_{er} and WN_{ew} used in the objectives (1c) to (1f) for the employee *e* that the pattern is being generated for. For example, if it is a minimum objective (1f) or (1d) then the pattern with the higher variable value is dominant for that objective. If it is a maximum objective (1c) or (1e) then the pattern with the lower variable value dominates. A pattern dominates another pattern if it dominates for at least one objective and is not dominated on any other objectives.

If the pattern is dominated by an existing pattern then it is discarded (to significantly increase the computation speed the pattern that dominates it is moved to the start of the *NextArray*, this has the effect that the next time the array is iterated through to check for dominance there is a greater chance it finds a dominant pattern more quickly). If the pattern dominates any existing patterns then it is added to *NextArray* and all the patterns that it dominates are removed. If it is identical to an existing pattern (that is, neither are better for any of the objectives) then it is also discarded. If it is incomparable to an existing pattern (that is, they each are better on an objective or cannot yet be compared for an objective) then it must be added. It may still dominate other patterns though which can be removed. It is also useful to note that when comparing two patterns, an objective which can be shown to be already satisfied in both patterns can be ignored in the comparison.

Dominance checking is the most time consuming part of the algorithm, particularly when the number of patterns to compare is large. A number of suggestions have been made in the literature to speed up this process. These include maintaining sorted lists or checking for dominance at different points in the algorithm. However, the feasibility of these suggestions depends upon the type and number of objectives present. We perform the dominance testing at step 19 but it is also possible to compare patterns at steps 30 and 31 instead.

At step 23, the new pattern cannot be added to *NextArray* as it would cause the maximum array size to be exceeded. However, the pattern with the worst objective function value is replaced with the new pattern if it has a better objective function value. This is another heuristic rule which improves the speed of the algorithm.

At step 24 the algorithm moves to step 29 and tries a different shift assignment at the current day (*TryNextAssignment* is simply a label at the end of that loop which in effect immediately moves to the start of the loop and tries the next shift type). During development we did experiment though within going to step 9 (move to the next day) instead of going to the step 29 as it may appear more efficient to not test every shift type if we already have a valid pattern. However, it was found to be much more effective to continue generating new patterns by going to step 29. This is because we are testing all possible shift types which although slower, as a result of the heuristics and rules within lines 16-27 the patterns which remain by the next day (step 9) are the dominant patterns with lower partial objective function values.

It is also worth mentioning that although adding the heuristics described had the most significant impact on the performance of the algorithm, how the algorithm is implemented can also have an effect on the speed of the algorithm. This is particularly the case with respect to memory management and the types of data structures used.

5. Results

In addition to providing results for the branch and price algorithm, we also compare them to an ejection chain based approach called variable depth search (VDS). Although the core of the algorithm is an ejection chain method, it contains a number of other features that have been added since it was originally described in [10]. These include incorporating a dynamic programming method into an iterative constructive method at the start of the algorithm. (This is the same dynamic programming algorithm used to solve the pricing problem in the branch and price method). A solution disruption and repair method (based on [8]) has also been added to extend the algorithm if the time limit is not exceeded and some fast, hill climbing methods which use the search neighbourhoods described in [7] have been incorporated too. The search neighbourhoods are defined by the assignment or de-assignment of shifts to employees or the swapping of two shifts between two employees. Additional neighbourhoods are then defined by extending these search neighbourhoods by considering assignments, de-assignments and swaps of shifts on multiple adjacent days (sometimes called block moves).

The variable depth search operates by taking these individual moves and chaining them into a single much larger move in order to escape from the local minima that they would otherwise be restricted to. Heuristics are used to dynamically select which moves to link together. When a local minimum is reached for this larger chain neighbourhood a re-start heuristic is used. The mechanism involves completely un-

assigning all the shifts assigned to a small number of employees and then re-building their schedules using the dynamic programming method.

For the first set of results and comparisons we applied the branch and price algorithm to each benchmark instance. We then repeated the experiments with the VDS method but setting its maximum time limit parameter to exactly the same amount of time as used by the branch and price algorithm. The results are shown in Table 3.

The instances in Table 3 are ordered on our estimated difficulty in solving them based on all the testing we have done (note that this does not correspond only to their size). The problem is a minimisation problem and results in bold indicate optimal solutions. All experiments were performed on a desktop PC with an Intel Core 2 Duo 2.83GHz processor.

	Branch and Price			VDS	
Instance	LB (root node)	UB	t (s)	UB	t (s)
Ozkarahan	0	**0**	< 0.1	**0**	0.1
Musa	175	**175**	< 0.1	**175**	0.1
Millar-2Shift-DATA1	0	**0**	< 0.1	300	0.1
Millar-2Shift-DATA1.1	0	**0**	< 0.1	**0**	0.02
LLR	301	**301**	0.8	302	0.8
Azaiez	0	**0**	0.3	3	0.3
GPost	5	**5**	2.0	42	2.0
GPost-B	3	**3**	29.3	6	29.3
QMC-1	12.5	**13**	57.6	30	57.6
QMC-2	29	**29**	1.9	39	1.9
WHPP	5	**5**	17.6	2012	17.6
BCV-3.46.2	894	**894**	8.3	895	8.3
BCV-4.13.1	10	**10**	892.7	**10**	892.7
SINTEF	0	**0**	10.5	13	10.5
ORTEC01	270	**270**	69.3	485	69.3
ORTEC02	270	**270**	105.1	540	105.1
ERMGH	779	**779**	19.7	**779**	19.7
CHILD	149	**149**	19.8	161	19.8
ERRVH	2001	**2001**	976.0	2057	976
HED01	136	**136**	396.0	148	396
Valouxis-1	8	80	909.6	60	909.6
Ikegami-2Shift-DATA1	0	**0**	41.7	1	41.7
Ikegami-3Shift-DATA1	2	**2**	597.8	13	597.8
Ikegami-3Shift-DATA1.1	3	4	995.2	14	995.2
Ikegami-3Shift-DATA1.2	3	5	5411.9	9	5411.9
BCDT-Sep	100	**100**	6239.5	200	6239.5
MER	7079	7081	36002.7	7185	36002.7

Table 3 Results for B&P and VDS on Benchmark Instances

For the branch and price we have also included the lower bounds found at the root node of the branch and bound tree (that is, before the integer constraints must be satisfied). The lower bounds are very close (often equal) to the optimal solution objective function value. The branch and price method is able to solve most of the instances to optimality but the computation time varies from less than one tenth of a second up to ten hours on the hardest instance. On the largest (and hardest) instance we set a ten hour time limit as beyond this, in testing, the algorithm had previously encountered a pricing problem in this instance for which the dynamic programming method ran out of memory (using over 2GB). This was because the size of the state space for the dynamic programming method is related to the number of shift types and the length of the planning period and this instance has twelve shift types and a six week horizon. Despite this, a very good upper bound can still be found for this instance within the ten hours.

For some of the easier instances the solutions were actually integer at the root node and so no branching was necessary. Others were slightly fractional but could be made feasible and optimal quite quickly in the branch and bound tree. The harder instances were very fractional though and the algorithm had to go deep in the tree to find an upper bound.

The variable depth search solves some of the easier instances in less than a second but compared to the branch and price some of the other results are worse. Despite this it is still worth noting that all these solutions are far better than could be achieved by hand (and of course in much less time also). One apparently anomalous result is the result for WHPP which appears significantly worse. However, in this instance the weights for the objectives are set such that some of the objectives have a weight of one and all others have a weight of 1000. The higher weighted objective appears to be quite difficult to completely satisfy such that near optimal solutions in terms of the number of objectives satisfied appear quite sub-optimal in terms of the objective function value. Some of the other instances also have similar large steps in weights and therefore also objective functions.

Only on one instance (Valouxis-1) does the variable depth search find a better solution than the branch and price method but there does not appear to be an obvious reason why it outperforms on this instance.

In the results, for both the algorithms we chose a single seed and identical parameter settings for every instance. That is, we did not perform many runs with different settings and chose the best result from each setting. For the instances Valouxis-1, Ikegami-3Shift-DATA1.1, Ikegami-3Shift-DATA1.2 on which the branch and price did not find the optimal solution though, it has also found the optimal solutions too in approximately the same computation times but with different algorithm settings such as different seeds.

For a second set of experiments we restricted the algorithms to shorter computation times. Based on feedback from end users we chose 30 seconds and 10 minutes. The feedback received suggested that if users want good solutions quickly they prefer not to wait more than 30 seconds. If they are willing to wait a bit longer for optimal or near-optimal solutions they generally prefer not to wait more than approximately 10 minutes. The results are shown in

	Branch and price				VDS			
Instance	UB	t (s)	UB	t (s)	UB	t (s)	UB	t (s)

Ozkarahan	**0**	< 0.1	**0**	< 0.1	**0**	0.1	**0**	0.1
Musa	**175**	< 0.1	**175**	< 0.1	**175**	30.0	**175**	600.0
Millar-2Shift-DATA1	**0**	< 0.1	**0**	< 0.1	**0**	0.8	**0**	0.8
Millar-2Shift-DATA1.1	**0**	< 0.1	**0**	< 0.1	**0**	< 0.1	**0**	< 0.1
LLR	**301**	0.8	**301**	0.8	**301**	30.0	**301**	600.0
Azaiez	**0**	0.3	**0**	0.3	**0**	12.1	**0**	12.1
GPost	**5**	2.0	**5**	2.0	16	30.0	8	600.0
GPost-B	**3**	29.3	**3**	29.3	6	30.0	6	600.0
QMC-1	48	30.0	**13**	57.6	36	30.0	16	600.0
QMC-2	**29**	1.9	**29**	1.9	31	30.0	30	600.0
WHPP	**5**	17.6	**5**	17.6	2001	30.0	**5**	600.0
BCV-3.46.2	**894**	8.3	**894**	8.3	895	30.0	895	600.0
BCV-4.13.1	**10**	30.0	**10**	30.0	**10**	30.0	**10**	600.0
SINTEF	**0**	10.5	**0**	10.5	6	30.0	2	600.0
ORTEC01	516	30.0	**270**	69.3	405	30.0	465	600.0
ORTEC02	1550	30.0	**270**	105.0	570	30.0	510	600.0
ERMGH	**779**	19.7	**779**	19.7	**779**	30.0	**779**	600.0
CHILD	**149**	19.8	**149**	19.8	161	30.0	161	600.0
ERRVH	12622	30.0	2189	600.0	2380	30.0	2058	600.0
HED01	201	30.0	**136**	396.0	168	30.0	148	600.0
Valouxis-1	340	30.0	160	600.0	120	30.0	60	600.0
Ikegami-2Shift-DATA1	16	30.0	**0**	41.7	6	30.0	**0**	327.7
Ikegami-3Shift-DATA1	47	30.0	**2**	597.8	32	30.0	13	600.0
Ikegami-3Shift-DATA1.1	38	30.0	23	600.0	32	30.0	14	600.0
Ikegami-3Shift-DATA1.2	35	30.0	26	600.0	37	30.0	9	600.0
BCDT-Sep	500	30.0	330	600.0	440	30.0	210	600.0
MER	14217	30.0	12663	600.0	8759	30.0	7187	600.0

Table 4.

	Branch and price				VDS			
Instance	UB	t (s)	UB	t (s)	UB	t (s)	UB	t (s)
Ozkarahan	**0**	< 0.1	**0**	< 0.1	**0**	0.1	**0**	0.1
Musa	**175**	< 0.1	**175**	< 0.1	**175**	30.0	**175**	600.0
Millar-2Shift-DATA1	**0**	< 0.1	**0**	< 0.1	**0**	0.8	**0**	0.8
Millar-2Shift-DATA1.1	**0**	< 0.1	**0**	< 0.1	**0**	< 0.1	**0**	< 0.1
LLR	**301**	0.8	**301**	0.8	**301**	30.0	**301**	600.0
Azaiez	**0**	0.3	**0**	0.3	**0**	12.1	**0**	12.1
GPost	**5**	2.0	**5**	2.0	16	30.0	8	600.0
GPost-B	**3**	29.3	**3**	29.3	6	30.0	6	600.0
QMC-1	48	30.0	**13**	57.6	36	30.0	16	600.0
QMC-2	**29**	1.9	**29**	1.9	31	30.0	30	600.0
WHPP	**5**	17.6	**5**	17.6	2001	30.0	**5**	600.0
BCV-3.46.2	**894**	8.3	**894**	8.3	895	30.0	895	600.0
BCV-4.13.1	**10**	30.0	**10**	30.0	**10**	30.0	**10**	600.0
SINTEF	**0**	10.5	**0**	10.5	6	30.0	2	600.0

ORTEC01	516	30.0	**270**	69.3	405	30.0	465	600.0
ORTEC02	1550	30.0	**270**	105.0	570	30.0	510	600.0
ERMGH	**779**	19.7	**779**	19.7	**779**	30.0	**779**	600.0
CHILD	**149**	19.8	**149**	19.8	161	30.0	161	600.0
ERRVH	12622	30.0	2189	600.0	2380	30.0	2058	600.0
HED01	201	30.0	**136**	396.0	168	30.0	148	600.0
Valouxis-1	340	30.0	160	600.0	120	30.0	60	600.0
Ikegami-2Shift-DATA1	16	30.0	**0**	41.7	6	30.0	**0**	327.7
Ikegami-3Shift-DATA1	47	30.0	**2**	597.8	32	30.0	13	600.0
Ikegami-3Shift-DATA1.1	38	30.0	23	600.0	32	30.0	14	600.0
Ikegami-3Shift-DATA1.2	35	30.0	26	600.0	37	30.0	9	600.0
BCDT-Sep	500	30.0	330	600.0	440	30.0	210	600.0
MER	14217	30.0	12663	600.0	8759	30.0	7187	600.0

Table 4 Results for B&P and VDS (30s & 10min) on Benchmark Instances

The results in Table 4 show that (as would be expected) the more computation time provided, the better the solutions. When the computation time is increased to ten minutes, the branch and price method is able to further improve twelve instances. Increasing the computation time to ten minutes for the VDS method further improves fifteen instances.

Over 30 seconds the branch and price is better on eight instances, equal on eight and worse on eleven. Over ten minutes it is better on eleven instances, equal on ten and worse on six. On all instances where the algorithms are equal they both found the optimal solutions. It is also evident that under both time restrictions the VDS is generally better on the larger instances and the branch and price outperforms on the smaller instances. It is also interesting to note that if the branch and price does not find the optimal solution within the time limit, the best upper bound it returns is generally worse than the solution found by the VDS in the same time. This suggests the potential benefit of using both algorithms in parallel if possible.

One apparently anomalous result is for the VDS on the ORTEC01 instance where the result is better for 30 seconds versus ten minutes. This is, however, correct. The VDS contains heuristics which automatically adjust based on the pre-defined computation time and the instance size. In this case the heuristic worked very well on this instance.

6. 2010 International Nurse Rostering Competition

In 2010 the First International Nurse Rostering Competition was held. The competition consisted of three ‘tracks’ each with different instances. For the first track (sprint) the algorithms were allowed a maximum of ten seconds computation time to solve each instance. For the second track (medium) the algorithms were allowed ten minutes and for the third track (long) ten hours were permitted. A number of instances for each track were released at the start of the competition and at the end competitors submitted their best solutions found within the time allowed for each instance. The results for the top five algorithms were verified and then tested by the organisers on some hidden instances to produce the final rankings for each track. We entered the competition using both the branch and price and the variable depth search algorithms.

We used the same model as developed for the benchmark instances to model the competition instances and then tested the variable depth search and branch and price algorithms on them. The results[1] are shown in Table 5 and Table 6.

Instance	LB (root node)	UB	t (s)
sprint01	56	**56**	32.3
sprint02	58	**58**	16.8
sprint03	51	**51**	61.6
sprint04	58.5	**59**	29.6
sprint05	57	58	270.4
sprint06	54	**54**	27.4
sprint07	56	**56**	29.6
sprint08	56	**56**	14.0
sprint09	55	**55**	20.2
sprint10	52	**52**	22.9
sprint_late01	37	**37**	25.0
sprint_late02	41.4	**42**	16.1
sprint_late03	47.83	**48**	24.0
sprint_late04	72.5	**73**	131.1
sprint_late05	43.67	**44**	29.2
sprint_late06	41.5	**42**	151.4
sprint_late07	42	**42**	17.9
sprint_late08	17	**17**	10.3
sprint_late09	17	**17**	22.8
sprint_late10	42.86	**43**	27.9
medium01	240	**240**	34.2
medium02	239.25	**240**	41.1
medium03	235.5	**236**	49.6
medium04	236.22	**237**	171.3
medium05	302.1	**303**	150.7
medium_late01	156	157	600.0
medium_late02	18	**18**	17.1
medium_late03	28.25	**29**	94.0
medium_late04	34.33	**35**	152.2
medium_late05	106.67	**107**	189.0
long01	197	**197**	84.9
long02	218.5	**219**	108.2
long03	240	**240**	69.9
long04	303	**303**	120.3
long05	284	**284**	84.4
long_late01	235	**235**	162.8
long_late02	229	**229**	590.9
long_late03	218.5	220	600.2
long_late04	220.6666667	**221**	188.0
long_late05	82.5	**83**	373.7

Table 5. Results of the branch and price applied to the competition instances.

As shown in Table 5, although the branch and price method could solve most of the instances to provable optimality, for the sprint instances the time required was longer than the maximum allowed of ten seconds. Therefore, for those instances we used the

[1] The solutions are also available online at http://www.cs.nott.ac.uk/~tec/ and also on request from the authors.

variable depth search. The results are shown in Table 6 (we have also included the results of testing the variable depth search on the instances within the competition time limits).

Instance	UB	t (s)	Instance	UB	t (s)
sprint01	**56**	10	medium01	244	600
sprint02	**58**	10	medium02	241	600
sprint03	**51**	10	medium03	238	600
sprint04	**59**	10	medium04	240	600
sprint05	58	10	medium05	308	600
sprint06	**54**	10	medium_late01	187	600
sprint07	**56**	10	medium_late02	22	600
sprint08	**56**	10	medium_late03	46	600
sprint09	**55**	10	medium_late04	49	600
sprint10	**52**	10	medium_late05	161	600
sprint_late01	**37**	10	long01	198	36000
sprint_late02	**42**	10	long02	223	36000
sprint_late03	**48**	10	long03	242	36000
sprint_late04	75	10	long04	305	36000
sprint_late05	**44**	10	long05	286	36000
sprint_late06	**42**	10	long_late01	286	36000
sprint_late07	**42**	10	long_late02	290	36000
sprint_late08	**17**	10	long_late03	290	36000
sprint_late09	**17**	10	long_late04	280	36000
sprint_late10	**43**	10	long_late05	110	36000

Table 6. Results of the variable depth search applied to the competition instances.

In Table 7, we show the rankings of the solutions we submitted for the instances released by the competition organisers (fourteen competitors entered the competition).

Instance	Solution	Ranking	Instance	Solution	Ranking
sprint01	**56**	1^{st} =	medium01	**240**	1^{st} =
sprint02	**58**	1^{st} =	medium02	**240**	1^{st} =
sprint03	**51**	1^{st} =	medium03	**236**	1^{st} =
sprint04	**59**	1^{st} =	medium04	**237**	1^{st} =
sprint05	58	1^{st} =	medium05	**303**	1^{st} =
sprint06	**54**	1^{st} =	medium_late01	157	1^{st}
sprint07	**56**	1^{st} =	medium_late02	**18**	1^{st}
sprint08	**56**	1^{st} =	medium_late03	**29**	1^{st}
sprint09	**55**	1^{st} =	medium_late04	**35**	1^{st}
sprint10	**52**	1^{st} =	medium_late05	**107**	1^{st}
sprint_late01	**37**	1^{st} =	long01	**197**	1^{st} =
sprint_late02	**42**	1^{st} =	long02	**219**	1^{st} =
sprint_late03	**48**	1^{st} =	long03	**240**	1^{st} =
sprint_late04	75	1^{st}	long04	**303**	1^{st} =
sprint_late05	**44**	1^{st} =	long05	**284**	1^{st} =
sprint_late06	**42**	1^{st} =	long_late01	**235**	1^{st}
sprint_late07	**42**	1^{st}	long_late02	**229**	1^{st}
sprint_late08	**17**	1^{st} =	long_late03	220	1^{st}
sprint_late09	**17**	1^{st} =	long_late04	**221**	1^{st}
sprint_late10	**43**	1^{st}	long_late05	**83**	1^{st} =

Table 7. Competition Ranking

On every instance our algorithms were first or first equal. However, in the final rankings we did not do so well due to some changes the organisers made to the hidden instances. For the majority of the hidden instances, the start date of the planning horizon was changed (but not the horizon length). We did not foresee this and as a result we incorrectly modelled some of the constraints relating to weekends. As such our solvers' objective function values for nearly all the hidden instances was incorrect, which clearly had a very adverse effect on the final rankings (our final competition rankings were: sprint: 4^{th}, medium: 2^{nd}, long: 2^{nd}).

7. Conclusion

We have presented new results for benchmark nurse rostering problems which will be particularly useful to other researchers. The results also show that a branch and price method can solve some instances very effectively. For other instances the time and resource requirements may be restrictive though. However, with new heuristics and other new ideas it may be possible to improve the performance further. For example, more advanced branching schemes in the branch and bound tree or decomposing the problem by splitting up the planning period may yield improvements. Although the variable depth search is not as successful as the branch and price on some instances, it is still a robust solver and able to find good solutions quickly. Another avenue for future research may be further integration of the two algorithms.

Within both algorithms a dynamic programming method is used which has also been introduced. The algorithm uses a number of novel ideas and heuristics which we believe are general enough to be adapted to other problem domains also.

All the instances tested were modelled using a generic model, at the core of which is a regular expression constraint. Although we cannot claim to be the first to apply this concept to staff scheduling problems we have expanded the idea to make it even more powerful and widely adoptable.

Finally, Figure 6 is a screenshot of a modelling tool for rostering problems (Roster Booster). The software features the variable depth search algorithm and the column generation algorithm (for calculating lower bounds) presented here, and is freely available for download at the website of Staff Roster Solutions Limited (http://www.staffrostersolutions.com). (Staff Roster Solutions is a spin-out company formed by the University of Nottingham to commercially license and develop its research on rostering algorithms such as that presented here).

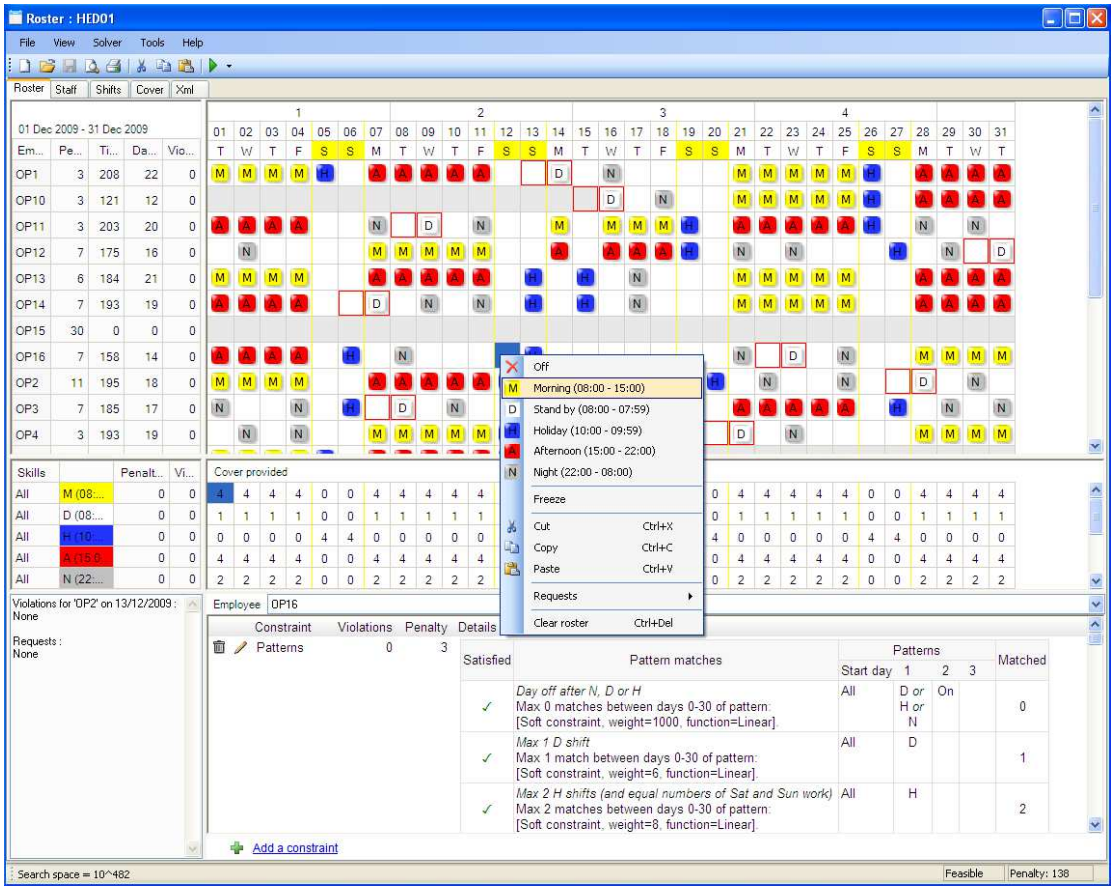

Figure 6 Roster Booster screenshot

References

1. COIN-OR Linear programming solver (https://projects.coin-or.org/Clp). 2010.
2. M.N. Azaiez and S.S. Al Sharif, A 0-1 goal programming model for nurse scheduling, Computers and Operations Research 32(3) (2005) 491 - 507.
3. J.F. Bard and H.W. Purnomo, Preference scheduling for nurses using column generation, European Journal of Operational Research 164(2) (2005) 510-534.
4. G.R. Beddoe and S. Petrovic, Enhancing case-based reasoning for personnel rostering with selected tabu search concepts, Journal of the Operational Research Society 58(12) (2007) 1586-1598.
5. F. Bellanti, G. Carello, F.D. Croce, and R. Tadei, A greedy-based neighborhood search approach to a nurse rostering problem, European Journal of Operational Research 153(1) (2004) 28–40.
6. P. Brucker, E.K. Burke, T. Curtois, R. Qu, and G. Vanden Berghe, A Shift Sequence Based Approach for Nurse Scheduling and a New Benchmark Dataset, Journal of Heuristics 16(4) (2009) 559-573.

7. E.K. Burke, T. Curtois, G. Ochoa, M. Hyde, and J.A. Vazquez-Rodriguez, A HyFlex Module for the Personnel Scheduling Problem. 2010, School of Computer Science, University of Nottingham. Technical Report.

8. E.K. Burke, T. Curtois, G. Post, R. Qu, and B. Veltman, A Hybrid Heuristic Ordering and Variable Neighbourhood Search for the Nurse Rostering Problem, European Journal of Operational Research 188(2) (2008) 330-341.

9. E.K. Burke, T. Curtois, R. Qu, and G.V. Berghe, A Scatter Search Methodology for the Nurse Rostering Problem Journal of Operational Research Society 61 (2010) 1667-1679.

10. E.K. Burke, T. Curtois, R. Qu, and G. Vanden Berghe, A Time Predefined Variable Depth Search for Nurse Rostering, INFORMS Journal on Computing (Accepted for publication, 2012).

11. E.K. Burke, P. De Causmaecker, G. Vanden Berghe, and H. Van Landeghem, The State of the Art of Nurse Rostering, Journal of Scheduling 7(6) (2004) 441 - 499.

12. E.K. Burke, J. Li, and R. Qu, A Hybrid Model of Integer Programming and Variable Neighbourhood Search for Highly-constrained Nurse Rostering Problems, European Journal of Operational Research 203(2) (2010) 484-493.

13. M.-C. Côté, B. Gendron, C.-G. Quimper, and L.-M. Rousseau, Formal languages for integer programming modeling of shift scheduling problems Constraints 16(1) (2011) 54-76.

14. S.J. Darmoni, A. Fajner, N. Mahé, A. Leforestier, M. Vondracek, O. Stelian, and M. Baldenweck, Horoplan: computer-assisted nurse scheduling using constraint-based programming Journal of the Society for Health Systems 5(1) (1995) 41-54.

15. S. Demassey, G. Pesant, and L.-M. Rousseau, A Cost-Regular Based Hybrid Column Generation Approach, Constraints 11(4) (2006) 315–333.

16. A.T. Ernst, H. Jiang, M. Krishnamoorthy, B. Owens, and D. Sier, An Annotated Bibliography of Personnel Scheduling and Rostering, Annals of Operations Research 127(1-4) (2004) 21–144.

17. A.T. Ernst, H. Jiang, M. Krishnamoorthy, and D. Sier, Staff scheduling and rostering: A review of applications, methods and models, European Journal of Operational Research 153(1) (2004) 3-27.

18. P. Eveborn and M. Rönnqvist, Scheduler – A System for Staff Planning, Annals of Operations Research 128 (2004) 21–45.

19. J. Friedl, Mastering Regular Expressions. 2006: O'Reilley Media.

20. A. Ikegami and A. Niwa, A Subproblem-centric Model and Approach to the Nurse Scheduling Problem, Mathematical Programming 97(3) (2003) 517-541.

21. S. Irnich and G. Desaulniers, Shortest Path Problems with Resource Constraints, in *Column Generation*, G. Desaulniers, J. Desrosiers, and M.M. Solomon, Editors. 2005, Springer. pp. 33-65.

22. B. Jaumard, F. Semet, and T. Vovor, A Generalized Linear Programming Model for Nurse Scheduling, European Journal of Operational Research 107(1) (1998) 1-18.

23. H. Li, A. Lim, and B. Rodrigues. A Hybrid AI Approach for Nurse Rostering Problem. in Proceedings of the 2003 ACM symposium on Applied computing. 2003. pp. 730-735.

24. M.E. Lubbecke and J. Desrosiers, Selected Topics in Column Generation, Operations Research 53(6) (2005) 1007-1023.

25. B. Maenhout and M. Vanhoucke, Branching strategies in a branch-and-price approach for a multiple objective nurse scheduling problem, Journal of Scheduling 13(1) (2010) 77-93.

26. A.J. Mason and M.C. Smith. A Nested Column Generator for solving Rostering Problems with Integer Programming. in International Conference on Optimisation: Techniques and Applications. 1998. Perth, Australia. L. Caccetta, et al. (ed). pp. 827-834.

27. H. Meyer auf'm Hofe, Solving Rostering Tasks as Constraint Optimization, in *Selected papers from the Third International Conference on Practice and Theory of Automated Timetabling, Springer Lecture Notes in Computer Science Volume 2079* E.K. Burke and W. Erben, Editors. 2000, Springer-Verlag: Berlin Heidelberg. pp. 191-212.

28. M. Moz and M.V. Pato, A genetic algorithm approach to a nurse rerostering problem, Computers & Operations Research 34 (2007) 667–691.

29. A. Musa and U. Saxena, Scheduling nurses using goal-programming techniques, IIE transactions 16 (1984) 216-221.

30. I. Ozkarahan. The Zero-One Goal Programming Model of a Flexible Nurse Scheduling Support System, in Proceedings of International Industrial Engineering Conference. in Proceedings of International Industrial Engineering Conference. 1989. pp. 436-441.

31. G. Pesant. A Regular Language Membership Constraint for Finite Sequences of Variables. in Principles and Practice of Constraint Programming – CP 2004. Lecture Notes in Computer Science 3258. 2004. pp. 482-495.

32. A. Pigatti, M. Poggi de Aragao, and E. Uchoa. Stabilized branch-and-cut-and-price for the generalized assignment problem. in 2nd Brazilian Symposium on Graphs, Algorithms and Combinatorics, Electronic Notes in Discrete Mathematics vol 19. 2005. Elsevier, Amsterdam. pp. 389–395.

33. J. Puente, A. Gómez, I. Fernández, and P. Priore, Medical doctor rostering problem in a hospital emergency department by means of genetic algorithms, Computers & Industrial Engineering 56 (2009) 1232–1242.

34. R. Qu and F. He. A Hybrid Constraint Programming Approach for Nurse Rostering Problems. in Applications and Innovations in Intelligent Systems XVI. The Twenty-eighth SGAI International Conference on Artificial Intelligence (AI-2008). 2008. Cambridge, England. T. Allen, R. Ellis, and M. Petridis (ed). pp. 211-224.

35. C. Valouxis and E. Housos, Hybrid optimization techniques for the workshift and rest assignment of nursing personnel, Artificial Intelligence in Medicine 20(2) (2000) 155-175.

36. G. Weil, K. Heus, P. Francois, and M. Poujade, Constraint programming for nurse scheduling, IEEE Engineering in Medicine and Biology Magazine 14(4) (1995) 417-422.