New computational results for nurse rostering benchmark instances

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

[1]School of Computer Science, University of Nottingham, Jubilee Campus, Wollaton Road, Nottingham. NG8 1BB. UK

Abstract

This paper presents the results of applying a branch and price algorithm and an ejection chain method to a wide range of benchmark nurse rostering instances. The majority of the instances are real world and 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.

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 an optimisation/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 through 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 computation 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, 19, 27], constraint programming [14, 26, 35], mathematical programming [2, 3], other artificial intelligence techniques (such as case-based reasoning [4]) and hybrid approaches [12, 33]. 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 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 a 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 and 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 a branch and price method has been applied to these instances.

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 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 branch and price has been applied to these instances, it has previously been used on the nurse rostering problem [18, 21, 24, 25]. All these previous applications have similar structure and is the same one 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 [23] and for further reading on resource constrained shortest path problems see [20].

In the next section we discuss the challenge of modelling such a wide variety of instance and how we solved it. 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 largest 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 constraint/objective which are relatively straightforward to model. These include the cover constraints (ensuring there are correct or preferable number of employees assigned to each shift). However, the types of constraint that can be present in each employee's work schedule can vary significantly from instance to instance. To provide a system which can incorporate these variations we developed a constraint based on pattern/string matching or more specifically regular expressions. Using a pattern matching 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, 30]. 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 patterns to be matched and the patterns to be matched are sequences of shifts.

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*".

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).
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*".

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 min or max 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 allows 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.

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 and the number of shift types. Each instance also varies in the number, priority and type of constraints and objectives present.

Origin	Instance	Staff	Shift types	Length (days)	Ref
	Ozkarahan	14	2	7	[29]
	Musa	11	1	14	[28]
	Millar-2Shift-DATA1	8	2	14	[19]
	Millar-2Shift-DATA1.1	8	2	14	[19]
	LLR	27	3	7	[22]
	Azaiez	13	2	28	[2]
	GPost	8	2	28	
	GPost-B	8	2	28	
	QMC-1	19	3	28	
	QMC-2	19	3	28	
	WHPP	30	3	14	[35]

	BCV-3.46.2	46	3	26	[6]
	BCV-4.13.1	13	4	29	[6]
	SINTEF	24	5	21	
	ORTEC01	16	4	31	[8]
	ORTEC02	16	4	31	[8]
	ERMGH	41	4	48	
	CHILD	41	5	42	
	ERRVH	51	8	48	
	HED01	20	5	31	[32]
	Valouxis-1	16	3	28	[34]
	Ikegami-2Shift-DATA1	28	2	30	[19]
	Ikegami-3Shift-DATA1	25	3	30	[19]
	Ikegami-3Shift-DATA1.1	25	3	30	[19]
	Ikegami-3Shift-DATA1.2	25	3	30	[19]
	BCDT-Sep	20	4	30	[5]
	MER	54	12	48	

Table 1 Benchmark Instances

The instances are available for download from http://www.cs.nott.ac.uk/~tec/NRP/, where best solutions, visualisations and other software are also available.

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 [31] 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). The initial solution is provided by applying the variable depth search algorithm for five seconds.

If a provable optimal solution (lower bound equals upper bound) is not found within the time limit then the best upper bound is returned.

4.1. The Pricing Problem Solver

As already mentioned, 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 basic 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 an 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 1. 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 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 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 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 1 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 steps 5. and 19. 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 can then be used to discard the pattern if it is greater than or equal to the best upper bound found so far.

After creating a new partial pattern it is necessary to compare this pattern to the partial patterns in *NextArray* for dominance. If the pattern is dominated by an existing pattern then it is discarded (and as a heuristic the pattern that dominates it is moved up the *NextArray* so it may dominate more quickly next time). 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, as a heuristic the pattern with the worst objective function value is replaced with the new pattern if it has a better objective function value.

At step 24. it is possible to go to step 9 (move to the next day) instead of going to step 29. However, it was found to be much more effective to continue generating new patterns by going to step 29. This is because the patterns which remain by step 9 are more dominant and have lower 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. Particularly with regards to memory management and the types of data structures used.

5. Results

The first algorithm tested is 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 its 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.
We set the maximum run time for the VDS at 30 seconds. This time limit has been selected based on feedback from end users. Although they may be willing to wait longer time periods for optimal solutions, if they want a near optimal solution quickly, they prefer not to have to wait more than 30 seconds. Also, the VDS results are provided here mainly for comparative purposes. For more experiments in which the run time for this algorithm is varied see [10].

The second algorithm shown in Table 2 is the branch and price method.

Table 2 shows the results of applying the two different algorithms to the test instances. The instances in Table 2 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.

	VDS		Branch and Price		
Instance	UB	t (s)	LB (root node)	UB	t (s)
Ozkarahan	**0**	0.1	0	**0**	< 0.1
Musa	**175**	30.0	175	**175**	< 0.1
Millar-2Shift-DATA1	**0**	0.8	0	**0**	< 0.1
Millar-2Shift-DATA1.1	**0**	< 0.1	0	**0**	< 0.1
LLR	**301**	30.0	301	**301**	0.8
Azaiez	**0**	12.1	0	**0**	0.3
GPost	16	30.0	5	**5**	2.0
GPost-B	6	30.0	3	**3**	29.3
QMC-1	36	30.0	12.5	**13**	57.6

QMC-2	31	30.0	29	**29**	1.9
WHPP	2001	30.0	5	**5**	17.6
BCV-3.46.2	895	30.0	894	**894**	8.3
BCV-4.13.1	**10**	30.0	10	**10**	892.7
SINTEF	6	30.0	0	**0**	10.5
ORTEC01	405	30.0	270	**270**	69.3
ORTEC02	570	30.0	270	**270**	105.1
ERMGH	**779**	30.0	779	**779**	19.7
CHILD	161	30.0	149	**149**	19.8
ERRVH	2380	30.0	2001	**2001**	976.0
HED01	168	30.0	136	**136**	396.0
Valouxis-1	120	30.0	8	80	909.6
Ikegami-2Shift-DATA1	6	30.0	0	**0**	41.7
Ikegami-3Shift-DATA1	32	30.0	2	**2**	597.8
Ikegami-3Shift-DATA1.1	32	30.0	3	4	995.2
Ikegami-3Shift-DATA1.2	37	30.0	3	5	5411.9
BCDT-Sep	440	30.0	100	**100**	6239.5
MER	8759	30.0	7079	7081	36002.7

Table 2 Results for Benchmark Instances

As can be seen, the variable depth search solves some of the easier instances in less than a second. For the instances Musa and LLR, the optimal solution was also found quickly but as the algorithm has no lower bound (other than zero) it carries on trying to improve the solution further and uses the full thirty seconds. Compared to the branch and price some of the results may seem disappointing but it is important to note 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.

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 actually 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). However, a very good upper bound can still be found 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 quite deep in the tree to find an upper bound.

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.

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 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 are shown in Table 3 and Table 4.

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	47.83333333	**48**	24.0
sprint_late04	72.5	**73**	131.1
sprint_late05	43.66666667	**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	42.85714286	**43**	27.9
medium01	240	**240**	34.2
medium02	239.25	**240**	41.1
medium03	235.5	**236**	49.6
medium04	236.21875	**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	34.33333333	**35**	152.2
medium_late05	106.6666667	**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 3. Results of the branch and price applied to the competition instances.

As shown in Table 3, 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 variable depth search. The results are shown in Table 4.

Instance	UB	t (s)
sprint01	**56**	10.0
sprint02	**58**	10.0
sprint03	**51**	10.0
sprint04	**59**	10.0
sprint05	58	10.0
sprint06	**54**	10.0
sprint07	**56**	10.0
sprint08	**56**	10.0
sprint09	**55**	10.0
sprint10	**52**	10.0
sprint_late01	**37**	10.0
sprint_late02	**42**	10.0
sprint_late03	**48**	10.0
sprint_late04	75	10.0
sprint_late05	**44**	10.0
sprint_late06	**42**	10.0
sprint_late07	**42**	10.0
sprint_late08	**17**	10.0
sprint_late09	**17**	10.0
sprint_late10	**43**	10.0

Table 4. Results of the variable depth search applied to the *sprint* competition instances.

In Table 5 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
sprint01	**56**	1st =
sprint02	**58**	1st =
sprint03	**51**	1st =

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

Table 5. Competition Ranking

On every instance our algorithms were first or first equal. Unfortunately though in the final rankings we did not do so well due to some changes the organisers made to the hidden instances. In all the instances released by the organisers (including some "hint" instances they released) the start date of the planning period was the 1st January 2010 (which is a Friday). In 17 out of the 20 hidden instances the organisers kept the planning horizon as four weeks but changed the start date of the planning period to 1st June 2010 (which is a Tuesday). As changing the start date does not effect the difficulty of the problem but just makes the modelling process slightly more complicated we were not expecting the start date to change in the hidden instances. However, the change meant that in our objective function all the indices for weekend related constraints were out by exactly four (Tuesday index minus the Friday index). 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: 4th, medium: 2nd, long: 2nd).

7. Conclusion

We have presented some 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.

Finally, Figure 2 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).

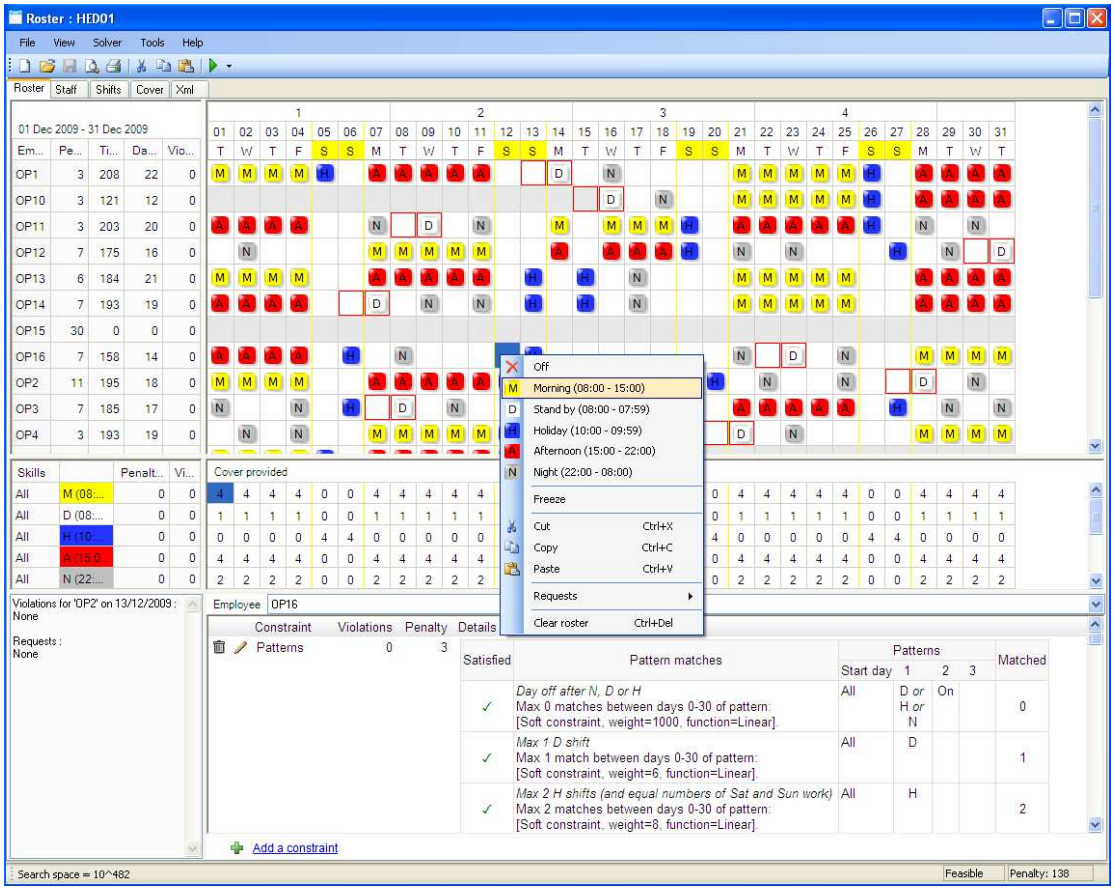

Figure 2 Roster Booster screenshot

References

1. COIN-OR Linear programming solver (https://projects.coin-or.org/Clp). 2010.
2. Azaiez, M.N. and S.S. Al Sharif, *A 0-1 goal programming model for nurse scheduling.* Computers and Operations Research, 2005. **32**(3): pp. 491 - 507.
3. Bard, J.F. and H.W. Purnomo, *Preference scheduling for nurses using column generation.* European Journal of Operational Research, 2005. **164**(2): pp. 510-534.
4. Beddoe, G.R. and S. Petrovic, *Enhancing case-based reasoning for personnel rostering with selected tabu search concepts.* Journal of the Operational Research Society 2007. **58**(12): pp. 1586-1598.
5. Bellanti, F., G. Carello, F.D. Croce, and R. Tadei, *A greedy-based neighborhood search approach to a nurse rostering problem.* European Journal of Operational Research, 2004. **153**: pp. 28□40.
6. Brucker, P., 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, 2009. **16**(4): pp. 559-573.
7. Burke, E.K., 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. Burke, E.K., 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, 2008. **188**(2): pp. 330-341.
9. Burke, E.K., T. Curtois, R. Qu, and G.V. Berghe, *A Scatter Search Methodology for the Nurse Rostering Problem* Journal of Operational Research Society, 2010. **61**: pp. 1667-1679.
10. Burke, E.K., T. Curtois, R. Qu, and G. Vanden Berghe, A Time Predefined Variable Depth Search for Nurse Rostering. 2007, School of Computer Science and IT, University of Nottingham. Technical Report. Available from: http://www.cs.nott.ac.uk/TR/2007/2007-6.pdf
11. Burke, E.K., P. De Causmaecker, G. Vanden Berghe, and H. Van Landeghem, *The State of the Art of Nurse Rostering.* Journal of Scheduling, 2004. **7**(6): pp. 441 - 499.
12. Burke, E.K., 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, 2010. **203**(2): pp. 484-493.
13. Côté, M.-C., B. Gendron, C.-G. Quimper, and L.-M. Rousseau, *Formal languages for integer programming modeling of shift scheduling problems* Constraints, 2011. **16**(1): pp. 54-76.
14. Darmoni, S.J., 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, 1995. **5**: pp. 41-54.
15. Demassey, S., G. Pesant, and L.-M. Rousseau, *A Cost-Regular Based Hybrid Column Generation Approach.* Constraints, 2006. **11**(4): pp. 315□333.
16. Ernst, A.T., H. Jiang, M. Krishnamoorthy, B. Owens, and D. Sier, *An Annotated Bibliography of Personnel Scheduling and Rostering.* Annals of Operations Research, 2004. **127**: pp. 21□144.

17. Ernst, A.T., H. Jiang, M. Krishnamoorthy, and D. Sier, *Staff scheduling and rostering: A review of applications, methods and models.* European Journal of Operational Research, 2004. **153**(1): pp. 3-27.
18. Eveborn, P. and M. Rönnqvist, *Scheduler – A System for Staff Planning.* Annals of Operations Research, 2004. **128**: pp. 21□45.
19. Ikegami, A. and A. Niwa, *A Subproblem-centric Model and Approach to the Nurse Scheduling Problem.* Mathematical Programming, 2003. **97**(3): pp. 517-541.
20. Irnich, S. 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.
21. Jaumard, B., F. Semet, and T. Vovor, *A Generalized Linear Programming Model for Nurse Scheduling.* European Journal of Operational Research 1998. **107**(1): pp. 1-18.
22. Li, H., 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.
23. Lubbecke, M.E. and J. Desrosiers, *Selected Topics in Column Generation.* Operations Research, 2005. **53**(6): pp. 1007-1023.
24. Maenhout, B. and M. Vanhoucke, *Branching strategies in a branch-and-price approach for a multiple objective nurse scheduling problem.* Journal of Scheduling, 2010. **13**(1): pp. 77-93.
25. Mason, A.J. 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.
26. Meyer auf'm Hofe, H., *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.
27. Moz, M. and M.V. Pato, *A genetic algorithm approach to a nurse rerostering problem.* Computers & Operations Research, 2007. **34**: pp. 667□691.
28. Musa, A. and U. Saxena, *Scheduling nurses using goal-programming techniques.* IIE transactions, 1984. **16**: pp. 216-221.
29. Ozkarahan, I. *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.
30. Pesant, G. *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.
31. Pigatti, A., 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.
32. Puente, J., A. Gómez, I. Fernández, and P. Priore, *Medical doctor rostering problem in a hospital emergency department by means of genetic algorithms.* Computers & Industrial Engineering, 2009. **56**: pp. 1232□1242.

33. Qu, R. 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.
34. Valouxis, C. and E. Housos, *Hybrid optimization techniques for the workshift and rest assignment of nursing personnel.* Artificial Intelligence in Medicine, 2000. **20**: pp. 155-175.
35. Weil, G., K. Heus, P. Francois, and M. Poujade, *Constraint programming for nurse scheduling.* IEEE Engineering in Medicine and Biology Magazine, 1995. **14**(4): pp. 417-422.