A Generic Nurse Rostering Algorithm for the INRC2010 Instances

E. Bulog
Department of Engineering Science
University of Auckland
New Zealand
ebul014@aucklanduni.ac.nz

Abstract

The issue of generating automated staff rosters has received a great deal of academic interest of late, and has also led to several timetabling competitions, most notably the First International Nurse Rostering Competition (INRC2010) which was held earlier this year (Haspeslagh et al. 2010).

Building on the work of several staff members and students in the Engineering Science Department (Dohn, Mason, and Ryan 2010), (Engineer 2003), (Mason and Smith 1998), this paper will present a column-generation approach to solving nurse rostering problems. The capabilities of this rostering engine have been expanded to allow it to be tested and benchmarked using the instances made available in the INRC2010.

One of the main issues which has traditionally plagued rostering problems is the lack of generality. Consequentially, customising a rostering engine to suit a specific instance can take a significant amount of time and effort. In our column generation approach, however, the use of pre-processor macro expansions allows us to easily customise the framework to suit a particular problem instance. Significant effort has been made to further automate this process for problem instances of the format given by the INRC2010.

Due to the inherently complicated nature of these problems, large instances can take a very long time to solve. In column-generation approaches, the bulk of this time is spent in thesub-problem, while the master-problem is comparatively quick. Consequentially, it will be the focus of future work to embed heuristic methods into the column-generation sub-problem in order to reduce the overall runtimes.

Key words: Rostering, column generation, pre-processing.

1 Introduction

1.1 Staff Rostering

In many large organisations, staff costs account for a very significant portion of all expenses, and so generating efficient staff rosters is of extreme importance from a

financial perspective. However, rosters cannot be generated purely based on a financial basis. Staff preference is also a very important factor in need of consideration, as a roster which is financially efficient may lead to poor staff moral if, for example, it contains a succession of shifts which may be undesirable from a staff perspective.

This results in two conflicting objectives; minimising staff costs and minimising staff unhappiness. Traditionally, this has been approached by either representing staff preferences as hard-constraints, referred to as *rules*, or as soft-constraints which can have varying weights or *costs* assigned to them which reflect their relative importance (Burke et al. 2005), (Burke et al. 2001). If a roster violates such a soft-constraint, the total cost of the roster will be increased to reflect the violation.

Instances of such rostering problems can differ vastly between any two organisations due to specific regulations and other requirements. Traditionally, this has meant that a significant amount of time and effort is required to tailor the solution approach to each instance.

1.2 Column Generation

The Column Generation approach (Barnhart et al. 1994) provides a useful tool for solving Integer Problems with a large number of columns. The problem is broken down into a sub-problem and a master-problem. The sub-problem, or *pricing problem* generates a limited number of columns, with attractive reduced costs, and adds them to a *column pool*, containing a relatively small subset of all possible columns. The restricted master-problem then performs the actual optimisation using only columns from this column pool, saving us the time and computation effort involved with generating all remaining columns.

In the context of the nurse rostering framework, a column of the constraint matrix corresponds to a potential *roster-line* which a nurse could work. A roster-line consists of a sequence of shifts occurring at intervals over the rostering horizon which a particular staff member must work. If complete enumeration is used to produce these roster-lines, the number of columns will increase dramatically with the size of the problem, making the nurse rostering problem a perfect candidate for the column generation framework.

Because we elect to construct columns through an iterative process, as detailed in section 3, we can check for dominance at all stages, meaning we can eliminate some columns at early stages during the generation process, further increasing the efficiency of this algorithm.

In order to implement this column generation, code has been built around the Coin-OR Branch-Cut-Price, or *BCP* framework, forming what will henceforth be referred to as *BCP-nurse*. BCP is an open source framework for solving mixed integer programs using the branch, cut and price algorithms. The bulk of the additional code in BCP-nurse has been added to the pricing problem, or column generation phase.

1.3 Boost C++ Pre-Processor Library

The Boost C++ Libraries are a set of free libraries which contain many useful extensions to the existing C++ functionality. The existing modelling framework makes use of the Boost pre-processor library to express entities and their attributes using macro, or `#define`, expansions provided in the *user.hpp* C++ header file.

This means that the problem specific customisation can be performed as the code is compiled, rather than incurring overheads during runtime. For more detail, please refer to Mason and Ryan (2009).

2 Model Formulation

Indices

i = employee: $\mathbb{S} = \{0, 1, 2, ..., s\}$; j = demand: $\mathbb{D} = \{0, 1, 2, ..., d\}$;
k = roster-line: $\mathbb{R}'_i = \{0, 1, 2, ..., r\}$;

Parameters

c_{ik} = cost associated with assigning employee i to roster-line k;
b_j = staffing level required by demand j

$$a_{ijk} = \begin{cases} 1, & \text{if employee } i\text{'s roster-line } k \text{ contributes to demand } j \\ 0, & \text{otherwise} \end{cases}$$

Decision variables

$$x_{ik} = \begin{cases} 1, & \text{if employee } i \text{ works roster-line } k \\ 0, & \text{otherwise} \end{cases}$$

Model

Minimize $\displaystyle\sum_{i\in\mathbb{S}} \sum_{k\in\mathbb{R}'_i} c_{ik}x_{ik}$

$$\sum_{k\in\mathbb{R}'_i} x_{ik} = 1 \qquad \text{for } i \in \mathbb{S} \tag{1}$$

$$\sum_{i\in\mathbb{S}} \sum_{k\in\mathbb{R}'_i} a_{ijk}x_{ik} \geq b_j \qquad \text{for } j \in \mathbb{D} \tag{2}$$

$$x_{ik} \in \{0, 1\} \qquad \text{for } i \in \mathbb{S},\ k \in \mathbb{R}'_i \tag{3}$$

Explanation

We are provided with a set $\mathbb{S}$ of s employees each with different skills, and a set $\mathbb{D}$ of d demands to be met. The objective is to minimise total cost of all roster-line assignments. Constraint (1) requires each staff member to be assigned to exactly one roster-line. Constraint (2) ensures minimum staffing level is satisfied for each demand specified. Note that $\mathbb{R}'_i$ is the set of roster-lines returned by the column generator for employee i, such that $\mathbb{R}'_i \subset \mathbb{R}_i$, where $\mathbb{R}_i$ is the set of all roster-lines for employee i.

3 The Column Generation Framework

In our formulation, a column of the constraint matrix corresponds to a potential roster-line which an employee could work. The aim of the column generation phase is to produce a small subset of all possible columns which may to be useful to the master problem. In order to achieve this, we construct a number of roster-lines for each staff member and then return a few of these with the most negative reduced costs, if any exist.

We do not simply enumerate all roster-lines during this stage. Instead, our column generation approach is based around the concepts of *entities* and *attributes*. Entities can be thought of as the building blocks for the roster-lines. The most fundamental entity is called a *shift*, which is simply a period of time in the rostering period during which a nurse may work in order to satisfy some demand. There are four other entity types; an *off-stretch* is a rest period between shifts, an *on-stretch* is a sequence of shifts falling on consecutive days, a *work-stretch* is a pairing of an on-stretch and an off-stretch, and the fifth entity is the roster-line itself. The process of constructing a roster-line using the other entities is outlined in section 3.1.

Attributes summarise the key characteristics of each entity. An attribute may be anything from the number of hours worked during a shift entity, to the number of consecutive weekends worked during a roster-line entity. Attributes may be used to ensure the feasibility of certain entities during the construction phases outlined in the rest of this section. For example, if we have a problem instance where the maximum number of consecutive working weekends is one, a roster-line generated with three consecutive weekends worked will be discarded as it is infeasible. Attributes can also be used to apply costs to the generated roster-lines, which is particularly useful when dealing with instances from the INRC2010, where everything must be modelled as a soft constraint. In the example given, our roster-line with three consecutive working weekends would incur a cost of two, one for each unit of violation.

3.1 Entity Construction

All shift and off-stretch entities are enumerated in the input files, as outlined in section 4.2, so are treated as inputs to this phase. The remaining entities are constructed using the process of either *initialisation* or *accumulation*. Initialisation is when a particular entity is first created, whereas accumulation creates a new entity by chronologically building off an existing entity of the same type.

For example, an on-stretch entity is initialised from a single shift entity. The resulting on-stretch, containing only a single shift, can then be combined with other shift entities using accumulation to produce a number of new on-stretch entities containing two shifts. In this example of accumulation, the first on-stretch containing only one shift is termed the *parent* entity, while the new on-stretches containing two shifts are referred to as *child* entities. The accumulation process can then continue by treating the on-stretches containing two shifts as parent entities, thus producing a number of child on-stretch entities which each contain three shifts.

Table 1 outlines the rules behind these processes; note that work-stretches cannot be accumulated, as they can only ever contain exactly one on-stretch and exactly one off-stretch. The combination of initialisation and accumulation allows for efficient generation of a large number of potentially useful roster-lines, which are the end product of the column generation process.

Table 1: Entity Initialisation and Accumulation

Entity	Initialisation	Accumulation
on-stretch	shift	on-stretch + shift
work-stretch	on-stretch + off-stretch	N/A
roster-line	work-stretch	roster-line + work-stretch

Table 1 also illustrates the three main phases of column generation; *on-stretch generation*, *work-stretch generation* and finally *roster-line generation*, in which on-stretches are generated from shifts, work-stretches are generated from on-stretches and off-stretches, and roster-lines are generated from work-stretches respectively. This relationships between entities are illustrated in figure 1.

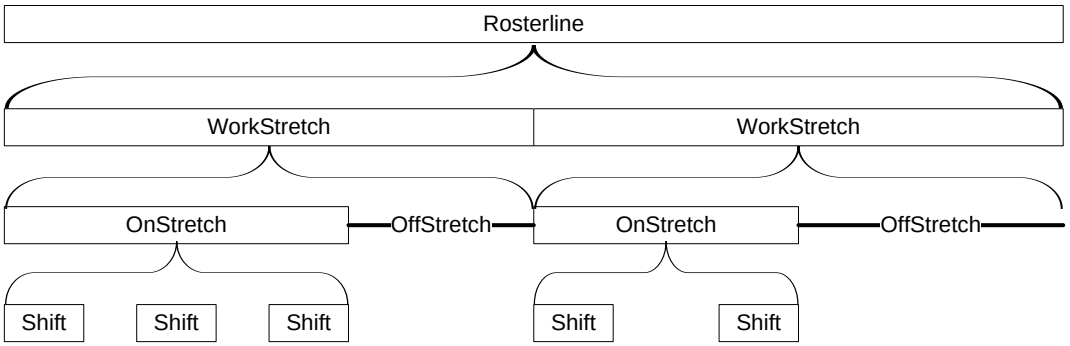

Figure 1: The Relationships between the Five Rostering Entities

Attributes for each entity are also calculated during all three phases following specific rules which are defined for the initialisation and accumulation phases. Some attributes follow simple additive rules, whereas others are more complicated. Examples of such calculation rules are provided in section 4.3. These attributes are also used to check for entity dominance as outlined in section 3.1.4.

3.1.1 On-Stretch Generation

An on-stretch can start with any shift and end with any later shift which satisfies some feasibility criteria; we may, for example, choose to limit our generation to on-stretches spanning a maximum of eight days. As such, we can construct a network whereby shifts represent nodes, and arcs represent feasible transitions between shifts. The nodes can be ordered chronologically, and there will be no arcs between shifts occurring on the same day, as we limit nurses to working one shift per day. Also, an arc will not exist between shifts occurring with a gap of a day or more between them, as an on-stretch, by definition, must consist of a sequence of shifts occurring on consecutive days. A small example of such a network is provided in figure 2.

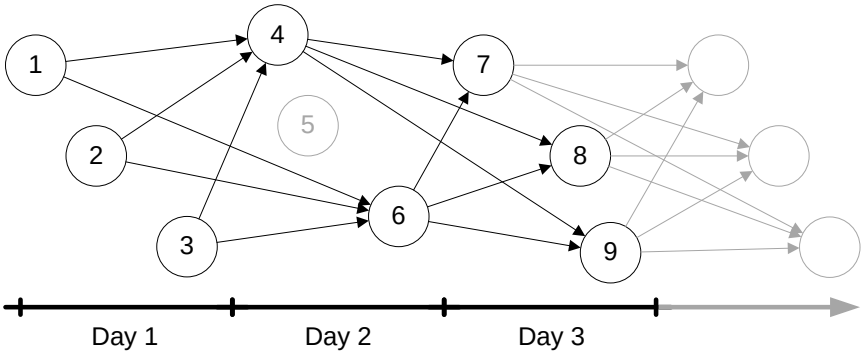

Figure 2: A Simplified Example of an On-Stretch Graph

This network also has resources which represent the attribute values. Generating an on-stretch beginning with shift i and ending with shift j corresponds to a path from node i to node j through the network. Such an on-stretch will only be feasible if the associated attributes are within the feasible range, so we can generate all feasible on-stretches by solving a one-to-all shortest path problem starting from each node in our network.

We can also use this network to generate only on-stretches which do not include a certain shift simply excluding the corresponding node. For example, node five has been excluded from the example in figure 2 meaning that we will only generate on-stretches (and therefore roster-lines) which do not include shift five. We may do this if an employee cannot work the given shift, or in order to enforce a branching decision made by the master problem, as the master problem branches on employee-shift assignments.

3.1.2 Work-Stretch Generation

As illustrated in table 1, work-stretches are generated directly from the pairing an on-stretch with an off-stretch, with no accumulation mechanism. Consequentially, the work-stretch generation phase is relatively simple. We examine each pair and construct the corresponding work-stretch if the time which elapses between the end of the on-stretch and the start of the off-stretch is within certain bounds.

3.1.3 Roster-Line Generation

In order to generate roster-lines, we need to combine a number of work-stretches together to span the entire rostering horizon. To this end, we can create another network where the work-stretches correspond to arcs forming transitions between days, or nodes. The first and last days of the rostering horizon correspond respectively to source and sink nodes in the network. Again, we must consider attributes as resources in the network, and so we can solve a shortest path problem with resource constraints. A feasible path through the network corresponds to a feasible roster-line.

3.1.4 Entity Dominance

The concept of entity dominance is used in all three stages of our column generation algorithm. At each stage, we compare the entities to see if any are dominated, in which case they are discarded, preventing further computational effort in generating extensions from them. An entity dominates another entity of the same type if it's cost is less than or equal to that of the dominated entity, and the cost of any future extensions will remain no more than the cost of the extensions of the dominated entity. All extensions of the dominated entity must remain feasible for the dominating entity.

This can be quite a complicated process, as each entity may have a number of attributes which behave in differing ways. Attributes of the entities are compared and dominance is established based on the methods specified by the instance-specific user.hpp C++ header file, which is generated by the Input File Generator outlined in section 4.2. There are several methods available, the most simple of which is only allowing dominance if all attribute values are equal, and the dominating entity

has a cost which is less than or equal to the cost of the dominated entity. This particular case will always hold, but there are some situations where we can exploit the structure of the attribute to relax the equality requirement, resulting in improved efficiency.

4 Adapting the Framework to the INRC 2010

4.1 Code Synthesis XSD

The instances for the INRC2010 were provided in xml format, along with an xsd schema file which outlines the data format. The Code Synthesis XSD libraries were employed to read in the data. These libraries firstly create header files which implement the elements given in the schema file as C++ classes which have all appropriate members and functions to retrieve data. Once this framework has been set up based on the schema file, any xml instance file from the competition can be read in and the appropriate data stored in these classes. This approach allows for all instance data to be read in efficiently and easily accessed through code, since the format provided by the schema file is used by all instance files provided by the INRC2010.

4.2 The Input File Generator

C++ code was written around the XSD libraries to create a project referred to as the *Input File Generator*. This code is passed an xml instance file as an input and generates all required files for the nurse rostering software itself, or *BCP Nurse*. These include four text files which provide information about the available staff, all possible shifts, demands to be met and all possible off-stretches. A parameter file is also generated which is customised by the Input File Generator. The most complicated file generated here is the user.hpp C++ header file, which includes the attributes, rules and costs (among other things) which are specific to each problem instance, as mentioned in section 1.3. This file, containing Boost macro expansions, is required to be pre-processed and compiled into the BCP Nurse framework before the other five files are read in and the optimisation is solved. Figure 3 illustrates this process.

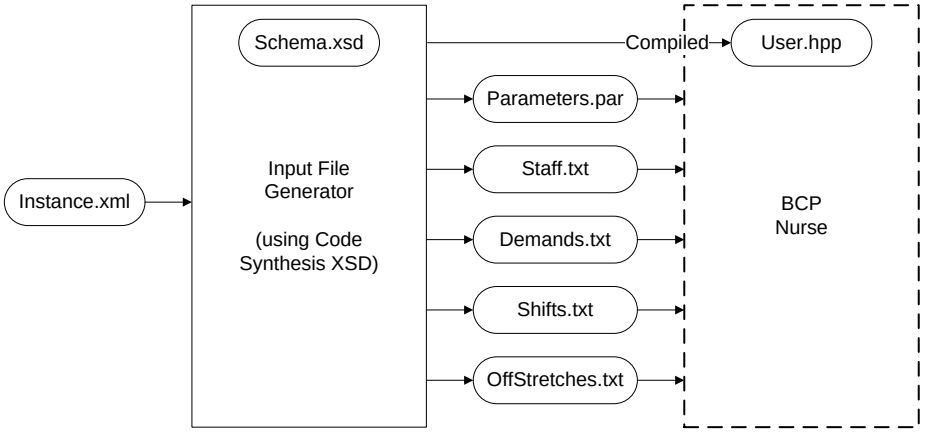

Figure 3: The Modified Setup Process

4.3 Additional Attributes

In order to improve the flexibility of the nurse rostering engine, several attributes were added to the various entities. These attributes allowed the model to penalise a number of additional unwanted features, such as the number of consecutive working weekends or patterns of shifts which staff may not want to work, as detailed in section 5. The attributes for all entities were all implemented in the user.hpp C++ header file, which is customised for the particular instance by the Input File Generator described in section 4.2. This customisation includes specifying the pricing scheme for each attribute, and any bounds on feasibility.

5 Applying Costs to Unwanted Shift Patterns

As mentioned, staffing preferences are a very significant aspect of nurse rostering. For a roster to be effective, it must not only be efficient in terms of cost and other concerns of the hospital, but it should also cater to the needs of the staff. This is a much harder aspect to quantify than monetary costs. As such, a key change which has been made to the rostering engine allows the model to handle sequences or patterns of shifts which nurses may not want to work.

These unwanted shift patterns may take several forms, but are characterised by a sequence of on-stretches and/or off-stretches over a number of consecutive days. Most of these patterns can be described by a single on-stretch; a common example of a two-day unwanted pattern of shifts might be a night shift on one day followed by an early shift on the second day, as illustrated in Figure 4. Nurses would not want to work a roster-line containing one (or many) of these patterns because this sequence of shifts does not allow for much rest in between. This example also illustrates why it may also be in the best interests of the hospital to avoid such patterns occurring, as patterns such as this can cause fatigue and/or demoralise staff.

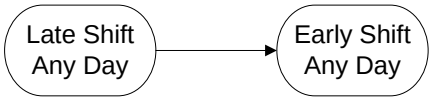

Figure 4: A Two-Entry Shift-Based Pattern

Unwanted shift patterns might also be defined not in terms of specific shifts worked, but simply by a sequence of days which are said to either be *on* or *off*. A day is termed an on-day if the employee of interest works a shift during that day, otherwise it is termed an off-day. An example of this type of unwanted pattern could be an off-day on a Friday followed by two on-days (Saturday and Sunday) as illustrated in Figure 5. This pattern essentially reflects that nurses do not want to work a full weekend if they are not working on the preceding Friday. Unwanted shift patterns such as this require both an on-stretch and an off-stretch in order to be fully described.

We require a general definition of an unwanted shift pattern which can include both on-stretches and off-stretches. To this end, each pattern is broken up into a number of *parts*. Each part contains a sequence of one or more consecutive shifts or days which can be expressed by either a single on-stretch or a single off-stretch. As such, a part can be referred to as either an on-stretch part or an off-stretch part.

Figure 5: A Three-Entry Day-Based Pattern

The individual pattern entries (shifts or days on/off) which make up each part are referred to as the *entries* of that part.

For example, if a particular rostering scenario included the two unwanted patters provided in Figures 4 and 5, then these two patterns would be composed of the parts and entities as outlined in Figure 6. Pattern 0 is consists of a single on-stretch part containing two entries. Pattern 1 consists of an off-stretch part comprised of a single entry and an on-stretch part containing two entries. For convenience, all numbering schemes start from zero.

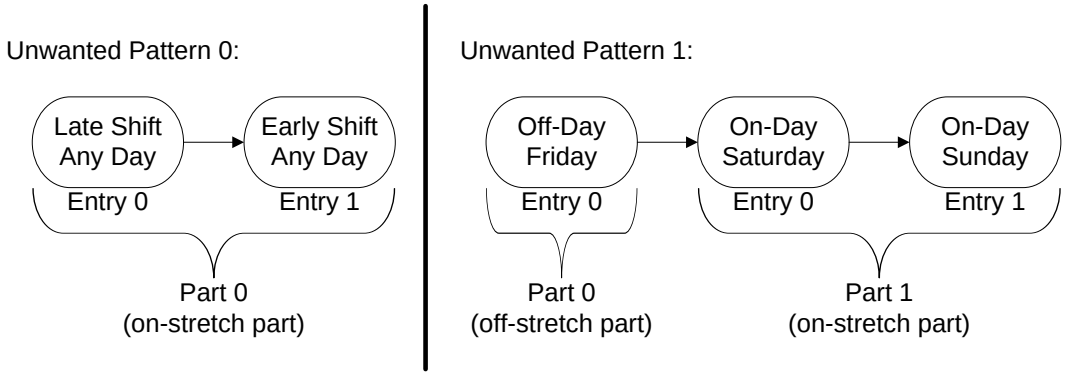

Figure 6: The Parts and Entries of Two Patterns

Attributes can be used to track the appearance of these unwanted shift patterns within each roster-line. Each part must be monitored separately within its respective entity, be it an off-stretch or on-stretch. This is further complicated by the fact that off-stretch attributes cannot be calculated (they can only be read as input), so the calculations for all attributes monitoring an off-stretch part must actually be performed in the work-stretch containing the off-stretch.

Each part of a pattern requires two attributes; a *tracker* and a *counter*. For convenience, each tracker and counter attribute are labelled with two numbers. The first of these numbers references the pattern which they correspond to, and the second number refers to the part within that pattern. For example, `pTracker1_2` denotes the pattern tracker attribute for part 2 of pattern 1 and `pCounter0_1` is used to denote the pattern counter attribute for part 1 of pattern 0.

The tracker attribute is used to monitor the construction of each part within the pattern. The value of this attribute is calculated based on how many consecutive entries have been added to the on-stretch or off-stretch which potentially belong to the part. Once the value of a pattern counter attribute has reached a maximum value (determined by the total number of entries in the part) the corresponding pattern tracker attribute is incremented. The pattern counter attribute simply counts how many complete parts are present in the given on-stretch or off-stretch.

Unwanted patterns can have different weights or costs assigned to them. These costs can be applied directly to the corresponding counter attributes, meaning that

each occurrence of a pattern within a potential roster-line is penalised accordingly, usually using a simple linear relationship.

6 Future Work

At the time of submitting this conference paper, I remain unable to generate meaningful results for most of the INRC2010 instances due to issues with my code. I am hoping to have results to present at the conference, which can be compared with the results generated by the winners of the INRC2010.

The primary aim of my work remains to embed heuristic methods into the column generation sub-problem in order to reduce the time spent in this phase. I am hoping to begin trialling a number of approaches and comparing their effects on both the runtime and solution quality. I will be able to use both the numerous instances provided by the INRC2010, as well as other data sets available to me in order to test these methods on a wide range of problems.

Acknowledgments

Although this is still very much a work in progress, I would like to acknowledge my supervisor, Dr. Andrew Mason, for his untiring patience and support. I would also like to extend my thanks to the others who have provided the body of work on which the BCP-nurse framework is based, most notably Faram Engineer and Anders Dohn, the latter of whom has helped me on a number of occasions.

References

Barnhart, C., E.L. Johnson, G.L. Post, M.W.P. Savelsbergh, and P.H. Vance. 1994. "Branch-and-Price: Column Generation for Solving Huge Integer Programs."

Burke, E.K., P. De Causmaecker, S. Petrovic, and G. Vanden Berghe. 2001. "Fitness Evaluation for Nurse Scheduling Problems."

Burke, E.K., T. Curtois, G. Post, R. Qu, and B. Veltman. 2005. "A Hybrid Heuristic Ordering and Variable Neighbourhood Search for the Nurse Rostering Problem."

Dohn, A., A.J. Mason, and D. Ryan. 2010. "A Generic Solution Approach to Nurse Rostering."

Engineer, F.G. 2003. "A Solution Approach to Optimally Solve the Generalised Rostering Problem."

Haspeslagh, S., P. De Causmaecker, M. Stølevik, and A. Schaerf. 2010. "First International Nurse Rostering Competition 2010."

Mason, A.J., and D. Ryan. 2009. "Customised Column Generation for Rostering Problems: Using Complie-time Customisation to create a Flexible C++ Engine for Staff Rostering."

Mason, A.J., and M.C. Smith. 1998. "A Nested Column Generator for Solving Rostering Problems with Integer Programming."