Customised Column Generation for Rostering Problems:
Using Compile-time Customisation to create a Flexible C++ Engine for Staff Rostering

Andrew J Mason and David Ryan
Department of Engineering Science
School of Engineering
University of Auckland
New Zealand
a.mason@auckland.ac.nz

Anders Dohn
Department of Management Engineering
Technical University of Denmark

Abstract

This paper describes a new approach for easily creating customised staff rostering column generation programs. In previous work, we have built a large very flexible software system which is tailored at run time to meet the particular needs of a client. This system has proven to be very capable, but is difficult to maintain, and incurs the time penalties of run-time customisation. Our new approach is to customise the software at compile time, allowing compiler optimisations to be fully exploited to give faster code. The code has also proven to be easier to read and debug.

Keywords: Rostering, Column Generation

1 Introduction

For many years, staff in the Engineering Science department have been involved in developing rostering software for organisations such as NZ Customs, TabCorp and Air New Zealand (Mason, 1995; Mason, 2001). Each of these rostering problems has its own characteristics and requirements, and so has required customised software development. A long term goal has been to develop a flexible rostering engine that can be applied to rostering problems from a wide range of problem domains.

In 1995, Andrew Mason and Mark Smith, a masters student at the University of Auckland, developed a very fast column generation system in the Fortran programming language to solve a particular nurse rostering problem (Smith 1995; Mason and Smith, 1998). This system used a nested column generation approach (described below) to quickly construct entering columns for the underlying set partitioning problem. The speed of this system is perhaps best illustrated by the experimental finding that generating just one column at a time gave the best performance; we are unaware of any

other system for which is the case. Although very fast, this system was difficult to customise for different problems.

This project was followed by the work of PhD student David Nielsen, who in 2003 developed a software system whose capabilities were sufficiently general to solve a range of staffing problems (Nielsen, 2003). This system was successfully implemented by Mantrack for one of their clients, TabCorp. However, this system did not use column generation which made it most suitable for problems such as those with flexible part time staff where the rosters have little long term structure.

In 2002, Andrew Mason and masters student Faram Engineering developed a new C++ GENIE software system which generalised the key ideas in the earlier work (Engineer, 2003). This system was much more flexible, but this flexibility came at the cost of increased run times arising from a flexible, fully object oriented design. For example, the GENIE system must track 'attributes' which are stored in C++ classes. These classes, and their associated parameters, are generated when the software starts. This means we must incur all the overheads associated with calling arbitrary class types, and also that the compiler is very limited in the range of optimisations it can perform.

In 2009, Andrew Mason and Anders Dohn developed Genie^{++}, a new software framework in which the customer specific customisations are performed not at run time, but instead at compile time. This has the advantage that the problem is fully specified at compile time, and so the code optimisation techniques available in modern compilers can be fully utilised. As we will show, it also produces code that is easier to read and debug because it is written using the language of the specific rostering problem being solved.

2 Modelling Framework

We formulate the staff rostering problem (SRP) as a generalised set partitioning problem as follows. We assume there are *s* staff to be rostered, with these staff having different skills. Conceptually, we assume that each of these staff has n_i alternative rosters, termed *roster lines*, that they may work during the next rostering period. (A rostering period can range from one or two weeks up to five or six.) Each roster line consists of sequences of shifts separated by time off. (It is often convenient to assume a staff member works no more than one shift per day, but this is not required.) Because we formulate this problem with a minimisation objective, we assume each of these roster lines has some associated quality measure that measures the staff member's dislike of that roster line.

To address the business needs, we assume that the requirement for staff can be specified by lower and upper bounds on the numbers of staff present across the day with different skill levels. For example, we may specify that we require "at least 5 staff of skill level 3 or higher between 10am and 2pm on Monday," and "no more than 2 staff at skill level 6, between 10am and 2pm on Monday." There is a set of *p* such requirements that define the work requirement for the roster. The *j*'th roster line for person *i* is modelled as a column where the coefficient $a_j^i[k]$ defines how this roster line contributes to the *k*'th work requirement. Thus, we can formulate our problem as a large generalised set partitioning problem, as follows:

$$\text{SRP:} \quad \text{minimise} \sum_{i=1}^{s} \sum_{j=1}^{n_i} c_j^i \, x_j^i$$

$$\text{s.t.} \quad \sum_{j=1}^{n_i} x_j^i = 1 \qquad \forall \; i=1, 2, \ldots, s \qquad (1)$$

$$\text{s.t.} \quad \sum_{i=1}^{s} \sum_{j=1}^{n_i} \sum_{k=1}^{p} a_j^i[k] \, x_j^i \begin{Bmatrix} \geq \\ \leq \\ = \end{Bmatrix} b_k \qquad \forall \; k=1, 2, \ldots, p \qquad (2)$$

$$x_j^i \in \{0,1\} \qquad \forall \; i=1, 2, \ldots, s \; , j=1, 2, \ldots, n_i \qquad (3)$$

To complete the above formulation, we need to consider the construction of the columns $\boldsymbol{a}_j^i$ for each staff member. These columns are generated during the solve process. The code to perform this generation forms the bulk of the Genie^{++} system.

3. Column Generation

The goal of the column generator is to determine the best (or a set of good) entering columns during the linear programming and branch and bound steps of the solution process. This requires careful modelling of the quality (objective function coefficient) of the roster line, as well as any rules that define legal and illegal roster lines. For example, if we have morning "M" and night "N" shifts, then staff might prefer a sequence of four day shifts "MMMM" over a more disruptive sequence such as "DDNN"; we need to be able to reflect this in the objective function. Furthermore, union rules might make the sequence "DNNN" illegal because it contains 3 (or more) consecutive night shifts, and so such sequences need to be detected and banned during the column generation process. Examples of other quality and legality issues might include:

- Maximum number of days on in a row / week.
- Some combinations of x-on followed by y-off days prohibited.
- A minimum rest period after a shift is required.
- Specific shift transitions are not allowed.
- Split weekends (one day worked and one day off) are undesirable.
- Single days-on / days-off are undesirable.
- Staff cannot work two consecutive weekends
- Night shifts must occur in sequences of two or more consecutive night shifts.

The key concepts underpinning the column generator that enable us to efficiently model these quality and legality requirements are the ideas of *entities* and *attributes*. Figure 1 shows the relationship between these entities, while Table 1 describes the rules by which our construction scheme builds up entities by combining and extending other entities. The generator starts with *shifts*, each of which may have attributes such as "number of hours worked", "is a weekend shift" and so on. The generator combines shifts together to give *on-stretches*, being sequences of days worked. For example, if we have morning "M" and night "N" shifts, then we might form an on-stretch "MMNN". This on-stretch will also have attributes associated with it that are formed from simple operations on the underlying shift attributes, such as "number of hours worked" and "number of weekend shifts". More complicated attributes can also be determined such as "number of days on" (in this case 4), and "number of day-to-night transitions" (1 in this case). The on-stretches are then combined with days off (*off-stretches*) to form *work-stretches*, which in turn are combined to form *roster-lines* that specify the sequence of activities for a staff member during the roster period.

on-stretch + shift → on-stretch

off-stretch + day-off → off-stretch

on-stretch + off-stretch → work-stretch

roster-line + work-stretch → roster-line

Table 1: Construction rules for the rostering entities

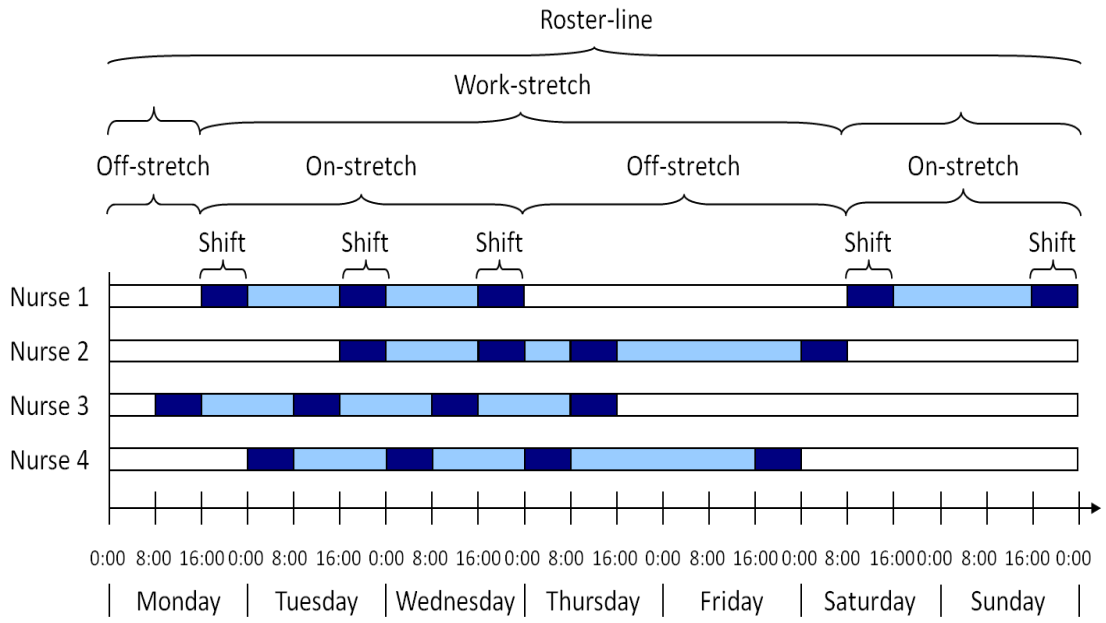

Figure 1: The entities that we use to describe a roster for a staff member.

The on-stretch, off-stretch, work-stretch and roster line entities are constructed during the column generation phase in a nested fashion using a dynamic programming approach. As in standard dynamic programming, only the best of any equivalent entities are kept. This gives us a nested column generation system that best exploits the special structure of our problem to efficiently solve the resource constrained shortest path column generation sub-problem.

Whenever a new entity is constructed using this process, the *attribute values* associated with that entity need to be calculated. These attribute values are then used to check legality and calculate quality values. The rules for calculating an attribute depend both on the entities being combined and the particular rules associated with that attribute. In the original C++ code, this multiplicity of possibilities was handled with objects and case statements, producing slow run times.

In our new design, the goal was to create customer-specific C++ code that would then be compiled to form that customer's solver engine. It was originally expected that a second program would be written to create this C++ code by interpreting some roster problem description language that we would have to create. However, after some initial experimentation, and thanks to recent advances in the C++ Boost Pre-processor Library (Karvonen and Mensonides, 2001), we realised that we could describe our rostering problems directly in the C++ pre-processor macro language. (This is the language that programmers use when writing statements such as

```
#define CURSOR(top, bottom) (((top) << 8) | (bottom)).
```

Although not a commonly used feature, C++ compilers allow the user to stop the compilation process after the pre-processor directives have been expanded, but before the code has been compiled. The output of this step is easy-to-read C++ code with variable and field names that match those from the real problem, making the code very easy to follow. The following code examples give an overview of the power of this system.

The code shown in Code Listing 1 demonstrates how the attributes are defined for a shift object. Notice that each line specifies the actual variable name (in lowercase), the variable type, and display name. The lowercase names will appear directly in the resulting C++ code.

```
// SHIFT_ATTRIBUTES must contain: starttime and endtime
# define SHIFT_ATTRIBUTES \
      ATT( (starttime    , int, "Starttime"), \
      ATT( (endtime      , int, "Endtime")  , \
      ATT( (shifttype    , int, "ShiftType"), \
      ATT( (paidhours    , int, "PaidHours"), \
      ATT( (dayson       , int, "DaysOn")   , \
      END )))))
```

Code Listing 1: Defining the attributes for a shift

The #define statement in Code Listing 1 is using a Boost pre-processor array structure to store the attribute definitions. These then get expanded by the pre-processor into a set of fields for the shift object. The code to perform this expansion, and the resulting C++ code that is generated, are shown below in Code Listing 2 and Code Listing 3. (Some code has been deleted to improve the clarity of this.)

```
class Shift {
public:
#          define    SATTR(_1,    _2,    i,    elem)       STYPE(elem,    i)
BOOST_PP_TUPLE_ELEM(3,0,elem);;
      BOOST_PP_LIST_FOR_EACH_I(SATTR, _, SHIFT_ATTRIBUTES);
```

Code Listing 2: An example of Boost pre-processor code used to expand a pre-processor array

```
class Shift {
public:
      Attribute<10, int, ... > starttime;
      Attribute<11, int, ... > endtime;
      Attribute<12, int, ... > shifttype;
      Attribute<13, int, ... > paidhours;
      Attribute<14, int, ... > dayson;
}
```

Code Listing 3: The code produced when the rostering definition in Code Listing 1 is expanded using the Boost pre-processor code in Code Listing 2.

As detailed earlier, an on-stretch is created by appending a shift (with the attributes given above) to another (possibly empty) on-stretch. The following code (Code Listing 4) illustrates how the attribute values are defined for the resulting on-stretch.

```
ATT( (paidhours, int, "paidhours", feas_all, domi_exact, cost_none,
o.paidhours + s.paidhours, s.paidhours) \
```

Code Listing 4: Definition of the 'paidhours' attribute in an on-stretch, including code for calculating the paidhours attribute value

Code Listing 4 defines a new on-stretch attribute with the name "paidhours." This parameter is tracked during the column generation so that it can be checked in the final roster line against the target of 80 paid hours per fortnight for this staff member. The most important entries in `paidhours` definition are the last two which are actual C++ statements that will be compiled. The first of these, `o.paidhours + s.paidhours`, specifies that when an on-stretch 'o' and a shift 's' are combined, the value for this attribute is calculated as the sum of the on-stretch's paid hours (o.paidhours) and the shift's paid hours (s.paidhours). The last entry handles the case when a blank on-stretch has a shift added to it.

The `o.paidhours + s.paidhours` definition gets expanded to give code such as the following (Code Listing 5):

```
T_value initialize (const Shift& s) const {
      return (o.paidhours + s.paidhours);
}
```

Code Listing 5: The code produced when the rostering definition in Code Listing 4 is expanded using the Boost pre-processor. This code is executed when a new on-stretch is formed by adding another shift to an existing on-stretch.

The other parameters in Code Listing 4 determine feasibility, dominance and cost contribution rules for `paidhours` as follows. The feasibility rule is used to discard an entity whenever it breaks some rostering rule; in this case 'feas_all' mean that all values are feasible. (This 'rule' is in fact the name of a C++ class.) Most rules, when required, can be expressed in terms of lower and upper bounds on the attribute value, and so another `feas_lbub` class is provided to handle this.

The `domi_exact` parameter details the rule for dominance. During the column generation, we can often determine that one entity dominates the other in the sense that any roster line containing the dominating entity will be better than one containing the dominated entity. (For example, an on-stretch "DND" might be dominated by "DDD" as the latter is perhaps equivalent from a rules point of view but better from a quality perspective as it avoids the day/night transitions.) The `domi_exact` term in this example says that two on-stretches must have the same attribute value if one is to be tested for dominance against the other.

Finally, the term `cost_none` describes how the value of this attribute is used to determine an associated cost value that contributes to the quality of the roster line. In this case, the paid hours of an on-stretch has no impact on the cost of a roster line. However, a commonly used option for this is to look up a table that translates the attribute value into a contribution to the objective. For example, an attribute might track the number of shift changes in an on-stretch, and penalise these in the objective (perhaps in some non-linear fashion) once each on-stretch is embedded within a work-stretch.

4. Results and Conclusions

The new GENIE^{++} framework has been implemented in C++ using the branch-and-cut-and-price framework of COIN-OR (Lougee-Heimer, 2003). We use constraint branching (Ryan and Foster 1981) where branches assign a shift to a staff member; this branch is then enforced both within the masters and within the column generator for that person.

The original system developed by Faram Engineer was successfully tested on four problems including a nursing example from Middlemore Hospital, scheduling for the United States postal service, scheduling shift work at an Australian energy power plant, and rostering helicopter pilots. These tests demonstrated the ability of the system to accurately model the variety of rules found in these different examples. The new system follows the design approach of this original system, and so shares its capabilities to solve problems of this diverse nature.

The new code has proven to be much faster to both develop and debug. The new system uses the COIN-OR framework unlike the old which used the ZIP software developed by Professor David Ryan (Ryan 1980). Combined with the rapid advancement of hardware, these multiple changes make comparisons of run times difficult. However, one indication of the improvements achieved is the finding of a better solution for the Middlemore nurse rostering problem using the new system. Experiments with this Middlemore nursing instance show that on a typical desktop PC, we can prove optimality in 224 seconds (but note that this required a careful choice of branching order). A near optimal solution is found after 97 sec. The system spends 23 seconds in the root node. (The old code took 357 seconds (15.5 times longer) to solve the root node, and 1617 seconds (16.6 times longer) to find the first integer solution. Optimality was never proven.) The run times from another problem, the Rigshospitalet in Copenhagen, are not as promising, due to more flexibility in each subproblem. This flexibility means that each subproblem takes much longer to solve as there are many more possible roster lines to consider. We cannot prove optimality for this instance within 10 hours. However, we can find a solution within 1.4% of the lower bound after 15777 seconds (4 hours, 23 minutes). A better solution (gap 0.4%) is found after 23350 seconds (6 hours, 29 minutes). The root node takes 6169 seconds (1 hour, 43 minutes) to solve.

These rostering problems often have very large integrality gaps, and so, as shown in the previous results, the branch and bound process can be very time consuming. To reduce these potentially long run times, we are working on the development of heuristic techniques that can be embedded within both the column generation and the branch and bound processes. For problems such as the Rigshospitalet instance, the sub-problem is very flexible in the sense that there are many feasible columns, which should mean that heuristics can easily find good (but not necessarily optimal) entering columns, thereby significantly reducing the times to find near optimal solutions. We are confident that these improvements will, eventually, produce a system that is sufficiently flexible and reliable to meet the rigorous requirements of commercial application. We look forward to reporting results on this in the future.

Acknowledgements

The authors Andrew Mason and Anders Dohn wish to acknowledge the contribution of Professor David Ryan to this project. Not only did Professor Ryan inspire the two authors to look at complex rostering problems, but he also made it possible for Anders Dohn to visit Auckland in 2009.

References

Engineer, Faramroze G., 2003. *A solution approach to optimally solve the generalized rostering problem*, Masters thesis, University of Auckland, 2003

Karvonen, V., P. Mensonides. 2001. *Preprocessor metaprogramming. C++ library.* Http://www.boost.org/ (Boost 1.36.0: 14/08/2008).

Lougee-Heimer, R., 2003. The Common Optimization INterface for operations research: Promoting open-source software in the operations research community. *IBM Journal of Research and Development* 47(1) 57-66. Http://www.coin-or.org/ (23/01/2009).

Mason, Andrew J., David M. Ryan, David M. Panton, 1998. Integrated Simulation, Heuristic and Optimisation Approaches to Staff Scheduling, *Operations Research*, Vol 46, Number 2, pp161-175

Mason, Andrew J., Mark C Smith, 1998. A Nested Column Generator for solving Rostering Problems with Integer Programming in *International Conference on Optimisation : Techniques and Applications*, L. Caccetta; K. L. Teo; P. F. Siew; Y. H. Leung; L. S. Jennings, and V. Rehbock (eds.), Curtin University of Technology, Perth, Australia, p827-834, April 1998

Mason, Andrew J., David Nielsen, 1999. PETRA : A Programmable Optimisation Engine and Toolbox for Personnel Rostering Applications presented at the *15th Triennial International Federation of Operational Research Societies (IFORS) Conference IFORS 99*, August 16-20, 1999, Beijing, China; available as School of Engineering Technical Report 593, University of Auckland

Mason, Andrew J., 2001. Elastic Constraint Branching, the Wedelin/Carmen Lagrangian Heuristic and Integer Programming for Personnel Scheduling" in *Annals of Operations Research*, 108(1), pp239-276

Nielsen, D., 2003. *A broad application optimisation-based rostering model*, PhD thesis, University of Auckland, 2003

Nielsen, D., Andrew Mason, 1998. Commercial development of a general application optimisation-based rostering engine, *Proceedings of the 33rd Annual Conference of the Operational Research Society of NZ*, pp10

Ryan, D. M. (1980) ZIP - A Zero-One Integer Programming Package for Scheduling, *Report C.S.S. 85, A.E.R.E.*, Harwell, Oxfordshire.

Ryan, D.M. & Foster, B.A. (1981). An integer programming approach to scheduling. In A. Wren (ed.), *Computer scheduling of public transport urban passenger vehicle and crew scheduling*, North Holland, Amsterdam, 1981, pp. 268-280.

Smith, Mark C., 1995. *Optimal nurse scheduling using column generation*, Masters thesis, University of Auckland