Commercial development of an optimisation-based roster engine

David Nielsen & Dr Andrew Mason
Department of Engineering Science
University of Auckland
New Zealand
d.nielsen@auckland.ac.nz & a.mason@auckland.ac.nz

Abstract

This paper presents a general application optimisation-based rostering engine developed in collaboration between the University of Auckland and Mantrack Decision Group (New Zealand). The first of two software products that have been developed as part of this collaborative project is a zero-one IP problem solver (ZIP_R) that employs linear programming and branch and bound methods. The second product is a general rostering software framework (PETRA) that contains ZIP_R as its core solving engine.

1 Introduction

With increasing competition and market pressure for improved performance in the automated production of rosters, the development of automated rostering technology has become a priority area of product development for providers of human-resource management technology. The use of optimisation methods in these products has been made attractive of late with the recent trends in price and performance of personal computers and shell software such as Windows NT.

In this paper we present a roster engine developed in collaboration between the Department of Engineering Science and the New Zealand based company Mantrack Decision Group. The engine, named PETRA (programmable engine targeting rostering applications), is the core of a plug-in rostering module that will be released as a component of the Mantrack HR-Payroll software package. The following diagram illustrates the arrangement of the rostering module within the Mantrack package.

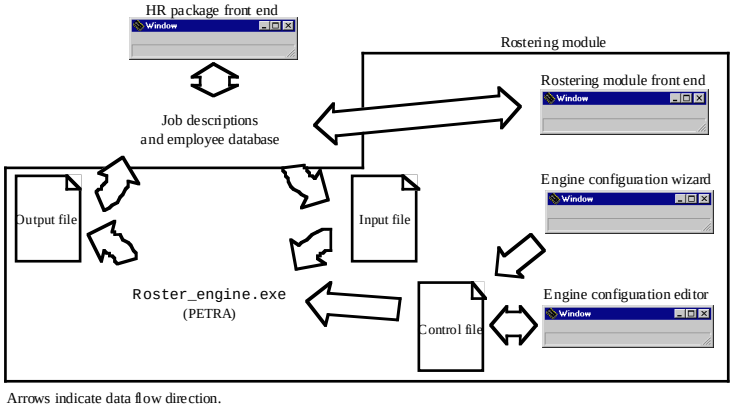

Figure 1. The plug-in rostering module

We also present in this paper the results of PETRA applied to the roster of one of Mantrack's larger rostering clients.

2 Automated rostering and the market

In this section we discuss some issues of commercial priority that have motivated the development of PETRA.

An implicit goal of automated rostering system development is to achieve full roster automation. In particular the ideal automated rostering product for an organisation will be capable of generating rosters that do not require manual completion. The degree to which an automated rostering system achieves this strongly reflects the value of the system for the rostering client. Indeed, over and above the potentially large saving in labour cost that is often required to manually produce rosters there is also the added value to the organisation in the way that the esoteric 'know-how' of manual roster production becomes the organisation's intellectual property and not that of the roster manager.

With the increasing awareness of quality management in industry an automated roster engine provides an ideal framework for the management of roster regulations and quality. By roster quality we imply quantitative quality assessment using quality metric functions in measurable attributes of both the individual's rosters and the overall roster.[1] In our experience with the development of PETRA we have found that the quality control that we can impose with the methods we have implemented equals and in most cases surpasses the requirements of roster quality management for many organisations.

From the commercial priority that has influenced this project we have recognised that in order for the technology we develop to remain competitive in the market it is important that we adopt methods of software engineering to manage the software development. In the past software developed for optimisation-based roster automation has been limited to academic research projects and as such is specialised for particular applications. In general, this has lead to very effective software for the particular applications and although the design is well documented and reused in other applications there is typically little opportunity for reuse of the development. From our collaboration with the software development team at Mantrack we have appreciated that ideally the automated rostering system should require a minimum amount of additional development or customisation of existing code components in order to achieve a high level of automation for a wide range of rostering clients.

Finally we should note that in contrast to the views held by many operations research practitioners, the priorities of the consumers of roster automation technology in industry are generally less concerned with the software speed. Indeed, many roster managers are content with a relatively slow running application provided that it displays its progress with reasonable accuracy such that that they may manage their time performing other tasks between runs. In the case where the roster is fully automated the roster manager can simply set and forget the roster application. The time while the

---

[1] Note that we may have ambiguity using the word *roster* since it can refer to either (1) the work schedule assigned to a single staff member, or (2) the collection of work schedules that are assigned to the overall set of staff. To avoid this we will consistently refer to an "individual's roster" when using definition (1), and to the "roster" or "overall roster" when using definition (2).

roster engine is running is hassle free time that the roster manager can fully dedicate to other tasks.

2.1 The scope of application of the roster engine

The design specification of the roster engine was simply that we seek to achieve the most generic engine possible within the constraints of the optimisation technology available. So far we have achieved a roster engine that can treat the pair of roster situations that we used to motivate the development. Our present view is that the engine can be successfully applied to the majority of Mantrack's rostering clients by setting the configuration of the roster engine (*see* Figure 1 above). For many of those clients for whom we cannot successfully apply the roster engine we expect to be able to customise it to perform full automation of the roster with a comparatively small amount of additional development.

In its present state of development the roster engine has capability that satisfies the requirements for full automation of the roster for NZ Customs staff at Auckland international airport and the large TabCorp (Australia) roster. A common thread between these rosters and indeed others in the service industry is that (1) there is little pattern in the individual's rosters, (2) there are few strict regulations, and (3) the considerations for quality are those that could be regarded as fundamental to all rosters. We have found that the roster automation market for such rosters in both New Zealand and Australia represents significant commercial potential.

To contrast the relatively straightforward rosters discussed above with those rosters for which we expect to have some difficulty, let us briefly consider an example of the rigorous specifications controlling the roster generation for airline cabin crew. An example of such a specification is an "hours of duty" rule (Day [1]). One such rule requires that a flight attendant must not have allocated more than twenty-nine hours of duty time in any seventy-two hour period. Here we define duty time as the period between when an individual begins and ends work on a day. In other words the rule states that for every hour of the roster we must ensure that the sum of duty hours in the previous seventy-two hours is less than twenty-nine. Note that the features associated with the "hours of duty" rule are an example of what we call an accumulation feature, or simply an accumulation (*see* section 3.1 below).

Rules involving accumulations are generally difficult to impose. We find, however, that in many rosters where automation is of significant commercial value there very few rosters that have such rules. We generally find that the accumulations are governed more in terms of guidelines than strict regulations.

3 Optimisation-based roster automation

In essence, the roster engine uses mathematical optimisation techniques to solve a constrained optimisation problem that reflects the physical roster problem. In particular, we use a generalised set partitioning formulation of the roster problem that we solve using linear programming and branch and bound methods. The following zero-one integer programming problem is a typical statement of a roster set partitioning problem.

$$\text{Minimise } z = c^{\mathrm{T}}x$$

subject to $A_{\mathrm{w}}x \geq b$, $A_{\mathrm{s}}x = e$, and $x = (x_1, x_2, \ldots, x_n)^{\mathrm{T}}$, $x_i \in \{0,1\}$ for $i = 1,2,\ldots,n$

where c is a given vector with positive real valued costs,

A_{w} is a given matrix with 0 or 1 entries (work constraint matrix),

b is a given vector with positive integer valued entries,

A_{s} is the staff constraint matrix,

and e is a vector with all 1 valued entries.

We will not discuss the above formulation in any detail, suffice to say the following: -

- The vector inequality constraint of the above problem represents the requirement that we have at least the given number of staff working on each of the associated shifts.
- The vector equality constraint governs the selection of exactly one individual's roster per staff member from the set of individual's rosters proposed for each staff member.

When we execute the roster engine it essentially cycles over two steps. Step (1) generates the roster problem, and step (2) solves the generated roster problem.

The main processes in step (1) are the generation of a set of proposed individual's rosters for each staff member and the evaluation of a real valued cost for each individual's roster reflecting its quality. In the activity of step (1) we are concerned with (a) imposing the regulations that govern the generation of the proposed individual's rosters and (b) making measurements of the quality of each of the proposed individual's rosters. We do not discuss these processes in any detail.

Issues associated with step (2) are discussed in section 3.2 below.

3.1 A view of the management of roster regulations and quality: modelling the roster problem

In this section we present a classification view of the rules and quality issues that influence roster generation. We label these categories as (1) daily shifts, (2) daily shift transitions, (3) work stretches, and (4) accumulations.

A daily shift is a pattern of work over a day; the most simple of which is continuous paid work from the time an employee arrives to their finish time for the day. Another common configuration of a daily shift is the 'split' shift where we have two pieces of work separated by an unpaid break. A work stretch is a set of days defined by a pattern of work. If we consider the work assigned to an individual across a number of days we identify a work stretch as a maximal set of consecutive worked days coupled with the maximal set of subsequent consecutive days off. An accumulation refers to the accumulated value, over the roster period, of a measurable attribute of an individual's roster. An example of an accumulation is the number of paid hours an individual's roster contains.

The four categories are inherent in the design of the roster engine. As such we use these categories as guidelines for how we implement additional capability for the management of new roster regulations and quality. Arguably, these categories are exhaustively inclusive of all features we will require to address in rosters. They are however at least inclusive of all features we have been required to address in the rosters surveyed so far.

The following lists some examples of features in these categories.

Daily shift:	• The time between an individual's start and finish time on a day. • The individuals' preferences for availability during the day.
Daily shift transitions:	• Minimum time off between daily shifts on consecutive days. • Consistency of start times on consecutively worked days.
Work stretches:	• A minimum number of days off after a long set of worked days. • "Single days off".
Accumulations:	• The paid hours an individual should have over the roster. • "Hours of duty" rules.

3.2 Solving the roster problem

Currently, we are successfully treating the roster set partitioning problems we construct using the standard approach that involves relaxing the integer constraints and solving the associated linear programming problem and, where necessary, using a branch and bound search to obtain an acceptable integer solution. We solve the linear programming problems using a specialised version of the revised simplex method that incorporates methods to treat the high levels of degeneracy that we find in roster set partitioning problems.

There are a number of well documented difficulties associated with using the foregoing approach for solving the roster set partitioning problem. In this discussion we focus on (1) the exponential behaviour of the constraint matrix column dimensions with respect to the number of roster days and (2) the average complexity, as a function of basis size, of resolving a fractional LP relaxation solution using a branch and bound search. Of particular interest in this discussion we present two methods that we have implemented in PETRA. These are (1) sub-roster decomposition (Day [1]) and (2) a method of column enumeration with pricing.

Firstly, however, let us briefly discuss the background of other methods that have been successfully implemented to handle these difficulties.

A popular method for handling the exponential column dimension is column generation. A popular approach, which is successfully implemented in many projects, is the use of dynamic programming optimisation in a column generation process.

An approach to the management of complexity in the branch and bound search, which we have implemented in PETRA, is the method of constraint branching (Ryan [2]) combined with a depth first search strategy where we accept the first integer solution encountered. Another method for managing the complexity of the branch and bound search which is often combined with column generation is to assume structure in the physical roster problem to improve limited subsequence (Day [1]) in the constraint matrix and therefore reducing the potential in the constraint matrix for fractional solutions.

Our design specification has required us to provide a generic framework for treating these difficulties. We found that column generation methods currently available were perhaps not flexible and robust enough in their implementation for our purposes. We have found that the two methods we have implemented have afforded us flexibility to treat a wide range of problems yet still providing the high level of performance. Let us briefly discuss these two methods.

In sub-roster decomposition we separate the roster period into several smaller overlapping roster sub-periods (sub-rosters), treating these roster sub-problems

separately and progressively aggregating the full length individual's rosters for each staff member.[2] This helps reduce the number of possibilities of individual's rosters we may have and hence the number of columns. We should also note that with control over the sub-roster size we are also provided with control over the basis size and therefore we may have a degree of control over the potential in the constraint matrix for fractional solutions. Note that using straightforward column generation without a limited subsequence framework does not allow any such control.

Despite the use of the sub-roster decomposition to manage the exponential column dimension, the sub-problems we solve are often large even when we consider sub-rosters of two days in length. An example of this is the TabCorp roster where we have upward of 175,000 columns per two-day sub-roster problem. Motivated by this we have implemented a simple method of column enumeration with pricing to allow us to manage the size of the active column subset[3] by adding new negative reduced columns during solving of the linear relaxation problem.

Our approach using column enumeration with pricing has been facilitated by the efficiency of our problem construction algorithm to enumerate columns. How this comes about is clear if we imagine the situation where we can construct columns and evaluate both their cost and their reduced cost just as fast as we could price them if they were stored in memory. Indeed, we would prefer this approach since we avoid the process of constructing and then storing the columns and we also avoid the slowing effects of rapid memory accesses across large volumes of memory during RSM pricing. Although we cannot enumerate columns as fast as we can price them we do enumerate columns very fast. This has given rise to the possibility of some interesting and potentially advantageous trade-offs between storing and enumerating columns for the active subset.

Rather than enumerating and storing the entire set of columns a priori we have simply enumerated the first 200 columns for each individual and used column enumeration with pricing to add further entering variable columns to the active subset as they are required. Currently we have implemented column enumeration with pricing using a crude full-enumeration approach. This approach sees us performing column enumeration for each staff member, evaluating both costs and reduced costs, and terminating each enumeration when either: (1) we have added 200 columns with negative reduced cost to the active subset, or (2) we have enumerated the full set of columns for the individual. More advanced approaches that focus the column enumeration with pricing using the dual information are being developed.

An interesting point worth discussing here concerns the behaviour of the sub-roster decomposition approach in complying with regulations controlling the roster generation. In particular compliance with rules associated with accumulations is difficult to guarantee using the sub-roster decomposition approach. To treat these and other potential shortcomings of the sub-roster decomposition approach we have configured the engine to perform multiple re-optimisation passes over the roster. In each pass after the initial pass we are re-rostering each sub-period with the intention that the optimisation will adjust what occurs in the roster sub-periods, with respect to the

---

[2] Note that we define the LP costs of the proposed individual's rosters in each sub-problem to be the LP cost of the associated individual's roster for the entire roster period.

[3] We define the active column subset as the set of columns stored in memory that are priced by the revised simplex method.

changes in the adjacent sub-rosters, and therefore improve the state of the overall roster. There is clearly a potential that we may become trapped in a local minimum of the full-length roster problem objective. However, results have so far shown that we can nevertheless achieve high quality results.

4 An overview of the development

The roster engine is essentially composed of two code components (1) the core solving engine and (2) the roster problem manager. These two code components reflect the two steps described in section three: generating the problem and solving the problem. We will discuss some interesting aspects of the code that has been developed for these two components in the following subsections. Firstly, however, due to the unusual amount of commercial priority compared to academic priority that has motivated this project let us briefly discuss and compare the procedural approach to software development with the industry standard object-oriented approach.

Procedural approaches to coding mathematical software typically result in highly tuned code that is generally the fastest code that can perform the given task. Operations research practitioners generally advocate such an approach since we may typically argue that if we develop code to perform optimisation why should it not be optimised itself? However, procedural approaches have by most experience lead to unwieldy software as the system develops and the code files grow in size and number. The object-oriented approach in contrast to the procedural approach has repeatedly demonstrated that it can permit inexpensive development of very complex software applications although it is generally perceived to result in slower code. Advantages of an object-oriented approach include the increase in reuse of design and development of code, and enhanced flexibility of code.

Considerations for the benefits of object-oriented design have motivated recent work in applying a quasi object-oriented approach to the revision of existing procedural code in the roster problem manager.[4]

4.1 A software framework for generating roster set partitioning problems

As we described in section 3.2 the roster problem manager is able to decompose the full-length roster into roster sub-periods. The roster sub-problems are solved sequentially to construct the full-length individual's rosters for each staff member.

The roster problem manager uses an efficient incremental costing enumeration algorithm to pre-construct the sets of proposed individual's rosters and evaluate the cost coefficients. By incremental costing we mean that the cost calculations made for each column we construct are reused in the cost calculation we make for the subsequent column we construct. Using this approach we have exploited the relationship between successively constructed columns to reduce floating-point calculations required evaluating the cost coefficients (and column reduced costs). As such we have made significant reductions in computational effort required during column enumeration.

4.2 The ZIP_R IP solver framework

[4] We use the word *quasi* in the sense that much of the functionality that we can achieve using true object-oriented languages such as C++ and Delphi is not possible using the Fortran 90/95 specification.

A significant part of the development to date has been involved with producing a framework wrapper around the ZIP (Ryan [3]) package. This development combines the ZIP package and framework wrapper into a callable library of routines and defines a set of call-back routines that the user has responsibility to code and maintain. The combined ZIP package and framework wrapper has been titled ZIP_R (zero-one integer programming software for rostering). Without going into detail we will highlight some features of the ZIP_R package.

The general form of the ZIP_R package is in a similar style to the original ZIP package, particularly in the use of user call-back routines. However we should note the fundamental difference in that the ZIP_R package provides storage capability for the constraint matrix and cost coefficients. With the addition of storage capability the ZIP_R package has a style similar to solver packages such as C-Plex while it still provides all of the flexibility of the original ZIP package.

With the addition of storage capability, the ZIP_R package contains functionality that is provided by the user in the original ZIP configuration. This functionality is undertaken by 'private' routines in ZIP_R and hidden from the user. During the solving process ZIP_R calls to the user's call-back routines allowing the user to customise the way that ZIP_R operates on the stored column information. In this way, the new call-back architecture of ZIP_R both contains and extends that of the original ZIP package.

In the following we briefly summarise some features of the ZIP_R package.

Feature 1: Column grouping.

The columns stored in ZIP_R are assigned to user defined column groups. The use of the column grouping facilitates:

(1) Selective pricing of column groups in ZIP_R storage.
(2) Improved pricing performance by allowing the use of specialised pricing procedures for each column group. For example columns that have zero-one entries can be priced efficiently by exploiting the sparse storage representation.
(3) Storage representations customised for each column group for improved compact storage.

Feature 2: User definable encoding of stored columns.

The columns stored in ZIP_R storage may use a user-definable storage encoding. This allows the user to exploit particular considerations for the problem specific structure of the columns to reduce the sizes of the column data representations. Reducing the size of the columns in storage improves the efficiency with which the columns are accessed in memory. To interpret the stored column data, the ZIP_R package calls back to the user to get decoding instructions before executing processes that 'hit' encoded column data such as pricing and constraint branching.

Feature 3: Dynamic and static column storage.

Columns may be stored in the ZIP_R storage using either dynamic or static modes of column storage. Columns stored in the static mode remain fixed in storage until the storage structure is reset. Columns stored using the dynamic mode may be overwritten in storage by other subsequently written dynamic columns.

The dynamic mode of column storage is particularly useful in a column generation approach to solving the rostering problems as a means of managing the size of the constraint matrix column dimension. For example the user can define a criterion for column quality that can be used to increase the density of 'good' columns in the active subset by replacing only those with lowest quality. Improving the density of good columns can potentially have significant impact on the solving performance.

Feature 4: Standard facility for column generation.

The ZIP_R package column generation facility involves a dedicated call-back to the user in the event of a 'no entering variable' condition after the RSM pricing step. The ZIP_R column storage can be called to store columns whenever the execution thread is outside the ZIP_R library so that the user may in fact column generate at any time.

5 An example of the roster engine applied

In this section we present some results of the roster engine for the TabCorp phone-betting call centre roster. We focus particularly on the performance of the sub-roster decomposition approach for treating rules and quality considerations associated with accumulations.

To demonstrate the quality that we are achieving with re-optimisation passes we present and discuss comparative results from the TabCorp roster focussing particularly on the accumulation feature of the roster, which relates to staff requests for the number of jobs they are allocated over the roster. The quality consideration for the accumulation simply says that we should attempt in our allocations of jobs to comply with the numbers of jobs staff have requested per weekly roster.

Let us first briefly outline some interesting features of the optimisation of this roster problem.

With the use of the sub-roster decomposition, the weeklong roster is split into six two-day sub-rosters where adjacent sub-rosters overlap with one day. There were 693 staff and an average of 70 work constraints in each sub-roster problem. Staff could perform either a single or 'split' shift on a day. The addition of the 'split' shift approximately doubled the number of daily shifts on each day compared to the number of daily shifts if the staff had only been able to perform single shifts. Using column enumeration with pricing and dynamic column storage allowed us to cap the column storage for each staff member to 200 columns. This resulted in sub-roster problems with constraint matrices of between 40 and 100 thousand columns. One pass across the 7 day roster took around ten minutes on a Pentium 266 PC.

In the following plot we illustrate the quality of the accumulation feature in three rosters over the same period generated by (1) the existing roster automation engine at TabCorp roster, (2) a single pass using the PETRA roster engine, and (3) a double pass using the PETRA roster engine.

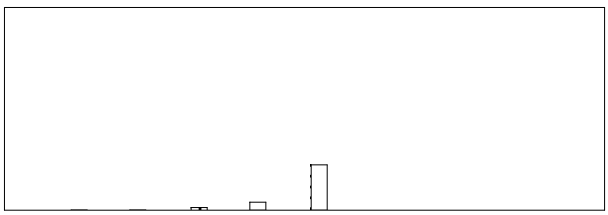

Figure 2. Performance in complying with staff requests for numbers of jobs

In the roster period that we examined there were a total of 2394 jobs that were to be allocated to 693 staff. With each staff member requesting between one and six jobs over the weeklong roster, the sum of the jobs that the staff expected to have allocated was 2610 giving a deficit of 216 jobs. Since the roster engine allocates the exact number of jobs specified to the staff we can see that in both the cases of the series' 'single pass' and 'double pass' the sum of the category values multiplied by the data values is indeed equal to this deficit.

The TabCorp management has not defined a framework for assessing roster quality in terms of this feature and it is unlikely that they ever will. Therefore, we make the fair assumption that a good quality roster with respect to this feature is one where we don't have the situation where a few staff are allocated with more jobs than requested while there is a larger number of staff with less jobs than requested (and vice versa). Also we assume that if a deficit (or surplus) of jobs were unavoidable then we would like to see an even spread of the deficit (or surplus) among staff. With this framework of quality it is clear that the best quality result we can expect is where we have 477 staff getting exactly the number of jobs they requested and 216 staff allocated with one job less than requested.

The plot in Figure 2 above clearly shows a trend of increasing quality of the accumulation starting from the TabCorp results to the single pass results and then to the double pass results. Let us examine the double pass results and compare it to the limit of quality with regards to this feature. Of the 30 staff who had two or less jobs than they requested all had a restrictive availability compared to the number of jobs they had requested. In general, this was the result of staff having leave for a significant part of the week or the individual had requested temporary changes to their availability leaving them much less available to work. It is clear, therefore, that the roster engine has very near achieved the practical limit of quality with regards to this feature in only two passes.

Acknowledgements

We would like to thank the Foundation for Research, Science and Technology for their continuing support with the GRIF funding for this project.

References

[1] P. Day, *Flight attendant rostering for short-haul airline operations*, PhD thesis, Department of Engineering Science, University of Auckland, New Zealand (1996).

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

[3] D.M. Ryan, *ZIP-A zero-one integer programming package for scheduling*, Report C.S.S. 85, A.E.R.E., Harwell (1980).