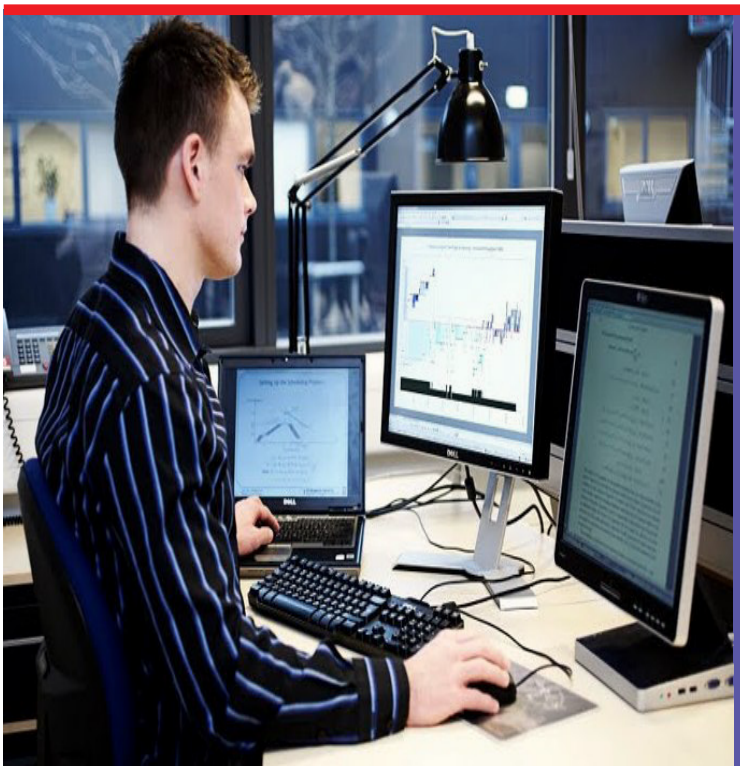
A Generic Solution Approach to Nurse Rostering

Dohn, Anders Høeg; Mason, Andrew; Ryan, David

Publication date:
2010

Document Version
Publisher's PDF, also known as Version of record

A Generic Solution Approach to Nurse Rostering



Report 5.2010

DTU Management Engineering

Anders Dohn
Andrew Mason
David Ryan
February 2010

DTU Management Engineering
Department of Management Engineering

A Generic Solution Approach to Nurse Rostering

Anders Dohn
adohn@man.dtu.dk
Department of Management Engineering, Technical University of Denmark

Andrew Mason, David Ryan
Department of Engineering Science, University of Auckland

February 26, 2010

Abstract

In this report, we present a solution approach to the nurse rostering problem. The problem is defined by a generic model that is able to capture close to all of the problem characteristics that we have seen in the literature and in the realistic problems at hand. The model is used directly in the solution algorithm which gives a very versatile solution method. The method at the same time is constructed to exploit a number of problem specific features and thereby we have a both versatile and efficient solution method. The approach presented uses a set partitioning model of the rostering problem, which is solved in a branch-and-price framework. Columns of the set partitioning problem are generated dynamically and branch-and-bound is used to enforce integrality. The column generating subproblem is modeled in three stages that utilize the inherent structure of roster-lines. Some important features of the implementation are described. The implementation builds on the generic model and hence the program can be setup for any problem that fits the model. The adaption to a new problem is simple, as it requires only the input of a new problem definition. The solution method is internally adjusted according to the new definition. In this report, we present two different practical problems along with corresponding solutions. The approach captures all features of each problem and is efficient enough to provide optimal solutions. The solution time is still too large for the method to be immediately applicable in practice, but we suggest a number of ways to improve the method further.

Keywords: generalized rostering problem, nurse rostering, nurse scheduling, column generation, branch-and-price, set partitioning problem, set covering problem, integer programming, linear programming, shortest path problem with resource constraints, dynamic programming, label setting.

1 Introduction

1.1 General introduction

This paper presents a solution approach to the generalized rostering problem with special emphasis on nurse rostering. In the generalized rostering problem the aim is to generate a feasible roster of high quality for a specified unit, section, department, or similar. Rosters typically have a large number of shift coverage constraints that have to be met. Coverage constraints may express an overall minimum staffing requirement for individual shifts or may more specifically represent needs for staff with certain skills or with particular contract conditions. Furthermore, the individual roster-lines are typically strictly governed by laws, union regulations and internal agreements. Altogether, this usually makes it hard to create feasible rosters, let alone high quality ones.

Historically, rosters were created manually by the head of the section or by an experienced member of the staff. Usually, the rosters were made by modifying former rosters or by putting

together roster-lines and parts of roster-lines which were known to be good. It takes a lot of experience to build good rosters and even with experience, the process of building the rosters is very time consuming. Therefore, there has been and still is a large demand for automated rostering tools. Within the last decade the supply of software products has increased significantly to meet this demand.

The airline industry is an area with many rules and regulations on rosters and with an inherent geographical dimension, which make rostering even harder. Staff expenses at the same time constitute a significant part of the airline budgets. For these reasons the airline industry was also the first industry to truly adapt computer aided rostering in the daily operation and are now fully dependent on scheduling and rostering software. Other industries with complicated rostering rules and requirements are following the example of the airline industry, and a lot of time and money can evidently be saved by restructuring the rostering process and by using automated tools where possible. Some desired properties of rosters are very hard to express explicitly and hence a human planner will always be a part of the rostering process. There are, however, many hard rules and easily assessable objectives, for which a computer based tool is perfectly suited. In such a tool it is possible to examine more solutions than what would ever be possible by hand. It is also possible to switch to mathematically based techniques, altogether. This requires a large number of calculations and is only achievable because of the calculation power of modern computers. In this paper, we present one such technique based on the idea of column generation.

One of the major obstacles in developing rostering software has been the varying requirements from one application to another. Most systems have been custom made to match the exact requirements of a particular company or institution as this is this only way to produce truly applicable rosters. The main problem with this approach is the time and money needed to develop the system. For the same reason, automated scheduling systems have in the past been reserved for institutions with a large and very apparent need for automation. The underlying basis of the algorithm presented here is very general and can hence be applied to a large variety of rostering problems. The setup still has to be adapted to a given application, but this adaption is easy and fast and hence a tailor made product can be made at significantly lower costs than before. This opens up a market of smaller customers with the same need for automated scheduling software, but with smaller budgets. These customers have previously been inaccessible because of the usual cost of customized software.

1.2 Problem specification

The objective in this project is to create a rostering tool, which is practically applicable and which complies with the requirements of realistic settings in a company. Several literature reviews have addressed the lack of broadly applicable approaches:

- *However, one key point that can be drawn from an analysis of the literature over the years is that very few of the developed approaches are suitable for directly solving difficult real world problems.* - Burke et al. (2004)
- *It is hard to avoid the conclusion that, in the United States at least, practitioners do not accept academically produced management and computer science solutions to the nurse-scheduling problem.* - Kellogg and Walczak (2007)

This project is an attempt to deal with the gap between the research and the actual requirements of hospitals. This means that the approach has to accommodate for the requirements of real-world problems. We list below, some of the important statements from the recent literature. These quote are from broad literature surveys, as we expect these not to be biased towards certain methods or application.

- *Another important area requiring further work is generalisation of models and methods. Currently, models and algorithms often require significant modification when they are*

to be transferred to a different application area, or to accommodate changes within an organisation. - Ernst et al. (2004b)

- *In other words, roster algorithms and rostering applications will need to involve individual-centric work constraints, preferences and business rules.* - Ernst et al. (2004b)
- *Models should be rich enough to capture the caregiving environment. Nurse X is not the same as Nurse Y; a scheduling model that considers them as interchangeable may not be solving the correct problem.* - Kellogg and Walczak (2007)
- *If this is our goal [to solve real nurse scheduling problems in real hospitals], then we must address the full range of requirements and demands that are presented by modern hospital workplaces.* - Burke et al. (2004)
- *Optimal solutions derived from techniques with high computing times are usually less valuable than one that is based on an flexible algorithm or user intuitive application.* - Cheang et al. (2003)

The method presented in the report, has been designed to comply with the described needs of the real-world. The last point has not yet been accommodated, but is one of the main concerns in the subsequent development of this project.

The work presented here is based mainly on academic projects from The University of Auckland, New Zealand. At the university, work on the nurse rostering problem was initiated by Smith (1995). Smith presents a column generation setup to solve a nurse rostering problem from Middlemore Hospital in Auckland. Following the promising results on a single application, two projects followed, with the aim of building a generalized rostering framework, which would be able to solve various rostering problems. In his PhD Thesis, Nielsen (2003) describes a general modeling framework for rostering problems. Roster-lines are generated by solving a subproblem using constraint programming techniques. Nielsen uses experimental data from Auckland Healthcare. The most recent of the projects is described in a Master's Thesis by Engineer (2003). Column generation is used to generate rosters for various problems. Engineer formulates the pricing problem as a three stage shortest path problem with resource constraints and solves it using label setting. The algorithm developed proves to be efficient and at the same time general enough to allow solution of applications with very different characteristics.

In our work, we also use two student projects from Danish universities. These describe other solution approaches, but we use the projects mainly for their specifications of nurse rostering problems in Danish hospitals. Poulsen (2004) describes a hybrid solution method, where a standard IP-solver is used to solve a set partitioning problem to generate anonymous weekly schedules. These are combined in a simulated annealing based meta-heuristic. Poulsen uses three different wards for experimental testing: Two from the National Hospital of Denmark and one from Odense Universitets Hospital. In the other project, Bliddal and Tranberg (2002) describe a constraint programming approach. Interviews are conducted at five different hospital wards and the individual properties of each are reported.

Together, the former projects describe 10 different realistic settings. From these we are able to list a number of general roster characteristics:

- Fixed planning period.
- Fixed number of shifts.
- Time norm for each employee.
- Maximum number of days on in a week / on-stretch.
- Some combinations of on/off days prohibited.
- A minimum rest period after a shift is required.

- Specific shift transitions are not allowed.
- Single days-on / days-off are undesirable.
- Each nurse has individual preferences.
- May have shift assignments which are fixed in advance.

We, at the same time, find a number of individual rules and agreements which are very specific and only apply to few of the problems. We must be able to cater for these individual rules, as well as we cater for the general ones listed above.

- On all days: at least one of the nurses was also there the day before (Bliddal and Tranberg, 2002, A2 / Holbæk sygehus).
- A nurse cannot work two consecutive weekends (Poulsen, 2004, Anæstesi- og operationsklinik / Rigshospital).
- Minimize the number of different shifts in a stretch (Poulsen, 2004, Obstetrisk klinik / Rigshospital).
- One week with 60 hours allows only 16 hours the following week (Poulsen, 2004, Gynækologisk og obstetrisk operationsafdeling / Odense Universitets Hospital).
- If working night shifts, at least to consecutive night shifts must be scheduled (Nielsen, 2003).
- If a set of days on ends with a night shift, then the following on-stretch must not begin early, unless there is a 'long' off-stretch in-between (Nielsen, 2003).
- Some nurses have a weekly off day called a zero-day. For each nurse, it is preferred that zero-days are always on the same day of the week (Poulsen, 2004, Anæstesi- og operationsklinik / Rigshospital).
- A special shift type must be covered by the same employee for a whole week (Poulsen, 2004, Gynækologisk og obstetrisk operationsafdeling / Odense Universitets Hospital).

In a bibliographic survey by Cheang et al. (2003) a list of commonly occurring constraints in the literature is presented, where the constraints are grouped into 16 different categories. Burke et al. (2009) in a similar way lists 26 sets of constraints that occur in practical nurse rostering problems. The list is a slight revision of a list originally formulated by Berghe (2002). Our algorithm should also be able to deal with all constraints of these types.

1.3 Literature

Staff rostering has already received a lot of attention in the literature. We refer to the extensive literate reviews of Burke et al. (2004), Ernst et al. (2004b), and Cheang et al. (2003). Ernst et al. (2004a) present a massive collection of references to papers on rostering. For an updated overview of available literature, the EURO Working Group on Automated Timetabling (Curtois, 2009) is a good resource.

The solution method presented here builds on the idea of column generation. For an introduction to column generation, see Desrosiers and Lübbecke (2005). The literature on column generation for rostering problems are, naturally, of special interest to us.

Jaumard et al. (1998) solve the nurse rostering problem using column generation. The subproblem is formulated as a shortest path problem with resource constraints, where each possible shift is represented by a node. It is solved by a two-stage algorithm proposed by the authors. Bard and Purnomo (2005b) solve a nurse rostering problem with individual preferences for the nurses. Columns are generated by a, so called, double swapping heuristic. High-quality solutions are found within minutes. In Bard and Purnomo (2005a) the model

is extended to allow downgrading of workers with higher level skills. Beliën and Demeulemeester (2006) schedule trainees in a hospital using branch-and-price. Interestingly, columns are generated for activities instead of the conventional employee roster-line columns. The problem of scheduling trainees is somewhat simpler than the rostering problem we consider, and only for this reason, is it possible to use the alternative column generation model. In a succeeding paper (Beliën and Demeulemeester, 2007) it is also concluded that the activity-decomposed approach does not have the same modeling power. However, if the problem allows this model to be used, the performance may be enhanced by doing so.

A branch-and-price approach to the nurse rostering problem is also described in Maenhout and Vanhoucke (2008). The approach has a number of features in common with our approach. The authors describe different pruning and branching strategies, e.g. lagrangian dual pruning and branching on the residual problem. Beliën and Demeulemeester (2008) extend the usual nurse rostering model to also include scheduling of the operating room and show that considerable savings can be made by integrating the two scheduling problems.

Eitzen et al. (2004) present a set covering model for crew scheduling problem at a power station. Three column generation based solution methods are proposed to solve the set covering model: the column expansion method, the reduced column subset method, and branch and price. Al-Yakoob and Sherali (2008) solve a crew rostering problem for a large number of gas stations using a column generation approach. The model takes into account the individual preferences of the employees. A heuristic founded on the column generation algorithm is able to solve realistic problems.

In the column generating subproblem presented in this report, two shortest path problems with resource constraints are solved. Much literature has been published on shortest path problems with resource constraints. See Irnich and Desaulniers (2005) for a literature review. Desrosiers et al. (1984) present an early version of the algorithm with time as the only resource, which is generalized by Desrochers (1988). Lübbecke (2005) suggests discarding all labels that cannot lead to a column with negative reduced cost. Righini and Salani (2006) present a significant improvement in performance by using bidirectional search. Chabrier (2006) utilize an idea on the potential value of each node to improve performance further.

1.4 Problem definition

The generation of nurse rosters is typically a complex process which requires a deep insight into the particular problem at hand. Even though most nurse rostering problems may appear to be similar, they usually differ on a few crucial points, so that a setup for one application cannot be directly adapted to another application. In the following, we describe a generalized rostering problem that captures the settings of most realistic nurse rostering problems.

A typical rostering problem is characterized by having a number of employees for which individual roster-lines are generated in order to satisfy certain coverage requests. Coverage request can be on segments, shifts, days or even weeks and are typically combined with demands for particular skills. Roster-lines have to comply with a number of laws, rules, and internal negotiations in order to be valid. The generalized rostering problem consists of putting together feasible roster-lines for all employees in order to satisfy all demands. Usually, the difficulty when modeling rostering problems lies in the validation and assessment of roster-lines. The rules usually vary a lot from one problem to another.

In the following, we introduce the main concepts in rostering that will be used throughout this report. These concepts at the same time define the rostering problems that we are trying to solve. The goal is to be able to solve any practical problem that fits this definition. Below we introduce the basic entities of a roster. The association between shifts, on-stretches, off-stretches, work-stretches and roster-lines is also illustrated in Figure 1.

Shift Defines a period of time where an employee is working. A predefined set of the shifts exists and a solution specifies which shifts an employee is working.

On-stretch A series of shifts is put together to form an on-stretch. The definition of the on-stretch is very loose. Usually, the shifts in an on-stretch are on consecutive days,

but it is not required to be so. The length of on-stretches may be constrained in terms of minimum and maximum number of hours, shifts, days or similar.

Off-stretch A period of time where the employee is not working.

Work-stretch An on-stretch and an off-stretch are joined together in a work-stretch. There is often a restriction on how on-stretches and off-stretches can be combined.

Roster-line A roster-line consists of a sequence of work-stretches. A roster-line spans the full scheduling horizon. We say that the entities introduced above are components of roster-lines. A roster-line typically has to respect constraints on total amount of hours worked, number of weekends worked, etc.

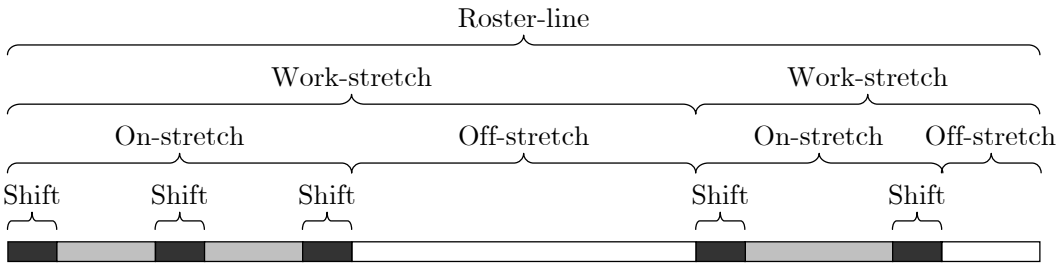


Figure 1: The entities of a roster and their internal association.

We refer to shifts and off-stretches as simple entities and to on-stretches, work-stretches and roster-lines as composite entities. On top of the entities introduced, we define a few more concepts.

Attribute An attribute belongs to one or more of the entities and is tracked as these are constructed and extended. Attributes of shifts and off-stretches are typically part of the input data. Attributes of composite entities are typically calculated from attribute values of their respective components.

Rule Rules define the validity of roster-lines, and their components. Rules are stated as constraints on attribute values. Attributes are introduced when needed to describe all applicable rules.

Cost Rules may be defined as soft constraints. Instead of declaring the entity infeasible, when a violation occurs, a certain cost is applied. Again, attributes are used to describe costs for a particular problem.

Demand A rostering problem consists of a number of demands that have to be met. A predefined set of shifts may contribute to each demand. Each demand may have a number of skill requirements, as well. All demands together define the validity of the roster. If a roster consists of only feasible roster-lines and all demands are met, the roster itself is feasible. Demands may be defined as soft and a penalty applies, if the demand is not met.

If a rostering problem fits these definitions, it is possible to solve it using the approach presented in this report. Engineer (2003) shows that this does indeed include a great variety of problems.

Example. Throughout this report, an example will be used to illustrate ideas and concepts. The example used origins from Middlemore Hospital and is the same as was used by Smith (1995) and Engineer (2003). A roster must be made for 86 nurses spanning a four week period. The problem has five different shift types:

M	Morning shift (7:00 a.m. - 3:30 p.m.).
8	Short morning shift (8:00 a.m. - 1:00 p.m.).
A	Afternoon shift (2:30 p.m. - 11:00 p.m.).
6	Short afternoon shift (6:00 p.m. - 11:00 p.m.).
N	Night shift (10:45 p.m. - 7:15 a.m.).

Below is an example of a simple roster-line with four work-stretches covering four weeks in total. The roster-line contains two types of shifts, namely regular morning shifts and short morning shifts.

Days																												
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	
M	M	M	M	M	-	-	8	8	8	8	-	-	-	-	8	8	8	8	-	-	M	M	M	M	M	-	-	

Roster-lines, like the one shown here, are combined to meet all demands. Demands may be defined for any combination of shifts, but in this example, demands involve only one or two shifts and only for shifts on the same day. This makes it possible to visualize the demands nicely in a table as seen below.

			Days																											
Shifts	Skills	Type	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
6A	0	$\geq$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
6A	2	$\geq$	9	9	9	9	9			9	9	9	9	9			9	9	9	9	9			9	9	9	9	9			
6A	3	$=$	13	13	13	13	13	8	8	13	13	13	13	13	8	8	13	13	13	13	13	8	8	13	13	13	13	13	8	8
6A	5	$\geq$	2	2	2	2	2			2	2	2	2	2			2	2	2	2	2			2	2	2	2	2			
6A	6	$\geq$	3	3	3	3	3			3	3	3	3	3			3	3	3	3	3			3	3	3	3	3			
8M	1	$\geq$	1	2	1	2	1	2	1	2			1	2	1	2	1	2	1	2	1	2	1	2			1	2	1	1
8M	3	$\geq$	4	2	4	2	4	2	4	2			4	2	4	2	4	2	4	2	4	2	4	2			4	2	4	1
8M	6	$\geq$	5	5	5	5	5			5	5	5	5	5			5	5	5	5	5			5	5	5	5	5			
A	3	$\geq$	9	9	9	9	9	6	6	9	9	9	9	9	6	6	9	9	9	9	9	6	6	9	9	9	9	9	6	6
M	0	$\geq$						1	1						1	1						1	1						1	1
M	2	$\geq$						7	7						7	7						7	7						7	7
M	3	$=$						10	10						10	10						10	10						10	10
M	6	$\geq$						1	1						1	1						1	1						1	1
N	0	$\geq$	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
N	2	$\geq$	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
N	3	$=$	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

An optimal roster for this instance is found in Appendix A. A nurse rostering problem from the National Hospital of Denmark (Poulsen, 2004) is modeled and solved in Appendix B.

2 Model

The generalized rostering problem is modeled as a generalized set partitioning problem and columns are generated dynamically as a part of the solution process. The problem is split in a master problem and a column generating subproblem. The master problem combines the roster-lines in order to meet all demands, while the subproblem provides feasible roster-lines.

2.1 Master problem

Given is a set of all employees, $\mathcal{E}$, and a set of demands $\mathcal{D}$. The goal is to find a combination of roster-lines, one for each employee, such that all demands are met at the lowest possible cost. The set $\mathcal{R}_e$ holds all feasible roster-lines for employee e. Because of the vast number of feasible roster-lines, it is not possible to generate all of these a priori. Instead, columns are generated iteratively. The set $\mathcal{R}'_e$ contains the roster-lines, of employee e, that have been generated so far. Three sets of decision variables are used:

$$\begin{aligned} \lambda_e^r &= \begin{cases} 1, & \text{if roster-line } r \text{ is chosen for employee } e. \\ 0, & \text{otherwise.} \end{cases} \\ s_d^- &= \text{Amount of under-coverage (slack) for demand } d. \\ s_d^+ &= \text{Amount of over-coverage (surplus) for demand } d. \end{aligned}$$

λ_e^r is a binary variable, while s_d^- and s_d^+ are continuous variables. The amount of permitted under- and over-coverage is regulated by imposing bounds on s_d^- and s_d^+, respectively.

Three sets of costs apply to the master problem. c_e^r gives the cost of roster-line r for employee e. c_d^- and c_d^+ specify the cost of under- and over-coverage for demand d, respectively. The parameter a_{ed}^r describes the roster-lines. $a_{ed}^r = 1$, if roster-line r for employee e contributes to demand d, and $a_{ed}^r = 0$ otherwise. The master problem can now be formulated as:

$$\min \sum_{e \in \mathcal{E}} \sum_{r \in \mathcal{R}'_e} c_e^r \lambda_e^r + \sum_{d \in \mathcal{D}} \left(c_d^- s_d^- + c_d^+ s_d^+ \right) \tag{1}$$

$$\sum_{r \in \mathcal{R}'_e} \lambda_e^r = 1 \qquad \forall e \in \mathcal{E} \tag{2}$$

$$\sum_{e \in \mathcal{E}} \sum_{r \in \mathcal{R}'_e} a_{ed}^r \lambda_e^r + s_d^- - s_d^+ = b_d \qquad \forall d \in \mathcal{D} \tag{3}$$

$$\lambda_e^r \in \{0, 1\} \qquad \forall e \in \mathcal{E}, \forall r \in \mathcal{R}'_e \tag{4}$$

$$s_d^- \geq 0, s_d^+ \geq 0 \qquad \forall d \in \mathcal{D} \tag{5}$$

The objective (1) is to minimize the total cost of all roster-lines while also minimizing penalties from under- and over-coverage. A feasible solution contains one roster-line for each employee (2). All demands must be met or the appropriate slack and surplus variables are adjusted accordingly (3). (4) and (5) set the domains of the decision variables. The LP-relaxation of (1)-(5) is denoted the *Restricted Master Problem*.

For any solution of the master problem given by λ_e^r, s_d^-, and s_d^+, we also have a dual solution. Let τ_e be the dual variables of constraints (2) and similarly, let π_d be the dual variables of constraints (3). If a primal solution does not exist, the dual problem is unbounded and no meaningful dual values can be found. Instead a dual ray is send to the subproblem to restore primal feasibility.

2.2 Subproblem

The subproblem, also referred to as the pricing problem, is responsible for generating new columns to the master problem. Given a vector of dual values, the subproblem returns a column with negative reduced cost, if one exists. Such a column will enter the basis in the master problem and dual values are updated. If no additional columns exist, the master problem solution is optimal.

The subproblem is formulated as a three stage model and follows the setup described in Mason and Smith (1998). Different rules and preferences may apply to each employee,

therefore a separate subproblem exists for each of the employees. Given is a set of legal shifts for employee e, $\mathcal{S}_e$, and a set of legal off-stretches, $\mathcal{F}_e$. From $\mathcal{S}_e$ a set of feasible on-stretches can be found $\mathcal{O}_e$. The set of legal work-stretches, $\mathcal{W}_e$, is created by checking legality for all feasible combinations of on-stretches in $\mathcal{O}_e$ and off-stretches in $\mathcal{F}_e$. By sequencing legal work-stretches, all roster-lines are generated and together defines the set $\bar{\mathcal{R}}_e$. All entities have a reduced cost and the objective of the subproblem is to find the feasible roster-line with the most negative reduced cost or to prove that no roster-lines with negative reduced cost exist. The reduced cost of a shift is calculated as the sum of all dual values of demands to which the shift contributes, plus any fixed cost of the shift. The fixed cost may be employee dependent, e.g. to represent individual shift preferences. Off-stretches have fixed costs only. The reduced cost of an on-stretch or a work-stretch is the sum of the costs of its components plus the cost imposed by attributes. The cost of a roster-line is equal to the dual value of the corresponding employee constraint, τ_e, plus the sum of work-stretch reduced costs plus additional attributes costs.

In the case where the subproblem is used to restore primal feasibility in the master problem, all entity costs are based solely on the values of the dual ray. All fixed costs and attribute costs are disregarded.

In order to have a fully descriptive subproblem model, we need to specify which entities of each type are feasible. As the model deals with each entity separately, we define feasibility for each entity separately. This means, e.g. that we specify what an on-stretch must respect to be feasible. When defining work-stretches, we may then assume that they are built from feasible on-stretches only. In the same way, roster-lines are always a sequence of feasible work-stretches. Shifts and off-stretches are part of the input and hence assumed to be legal. Skills and other employee dependent properties may, however, still disallow shifts and off-stretches.

For a compact description of on-stretches and roster-lines, we introduce a recursive definition of these two entities. Instead of describing an on-stretch by all the shifts it contains, we describe it by its latest shifts and a parent on-stretch, which contains exactly the remaining shifts. Note that the parent on-stretch may be an infeasible on-stretch. Also, an on-stretch may not have a parent on-stretch, if it contains only one shift. Figure 2 visualizes the recursive definition. Roster-lines are analogously defined recursively from work-stretches and other roster-lines. In the recursive formulation, we differentiate between accumulated cost and total reduced cost. The accumulated cost is solely calculated from the cost of components. The attribute costs are added to get the total reduced cost. This division is necessary in a recursive definition, to avoid adding attribute costs multiple times.

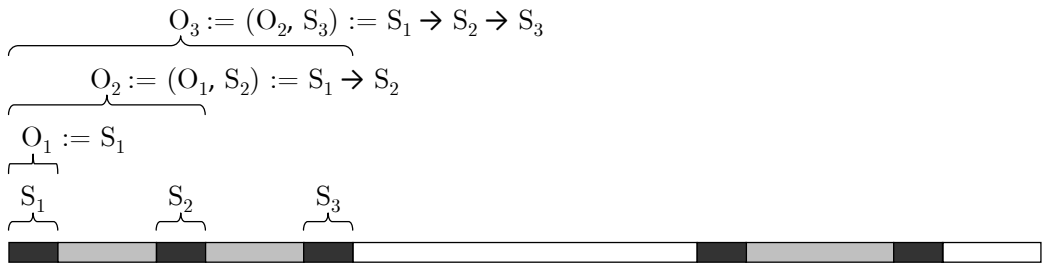


Figure 2: Recursive formulation of on-stretch.

Attributes are used to quantitatively ensure feasibility and to specify costs. All entities hold the two attributes: start time and end time. Each entity also has a number of individual attributes. These attributes are problem specific and are hence not specified in the generic model here. Some properties of a feasible entity are ensured already in the construction of the entity and a few designated parameters are used to construct only entities which are feasible in the problem considered. These parameters are described below.

Max days in an on-stretch There is always an upper limit on the number of days in an

on-stretch. In the extreme case, this bound is equal to the length of the horizon, but in most cases it is much lower and this limits the number of feasible on-stretches significantly.

Min/max time between shifts On-stretches are chronological sequences of shifts, and hence one shift will inherently stop before the next one in the on-stretch starts and the minimum time between shifts is 0. In many practical problems a rest period is required on top of that and can be enforced by setting this parameter larger than 0. Such rest periods are then enforced in the construction of the entities and no additional attributes are required for this purpose.

There is a significant difference between the attributes of simple entities versus attributes of composite entities. For simple entities, attributes merely hold a value which is given as a part of the input to the subproblem. Attributes of composite entities are calculated as a part of the solution procedure. Hence, a part of their definition is a description of how to calculate their value. For on-stretches and roster-lines, there are two scenarios: The entity may have only one entity component and in this case the attribute value is inferred directly from the attribute values of the component. We refer to this as value *initialization*. If, on the other hand, the entity has both a component and a parent of its own type, the new attribute values are calculated from attributes of both the parent and the component. We say that the attribute values are *accumulated*. Work-stretch attributes are always initialized. As opposed to simple entities, the attributes of composite entities may be restricted by bounds and costs may be imposed for certain values of the attribute. Figure 3 depicts the definition of the entities.

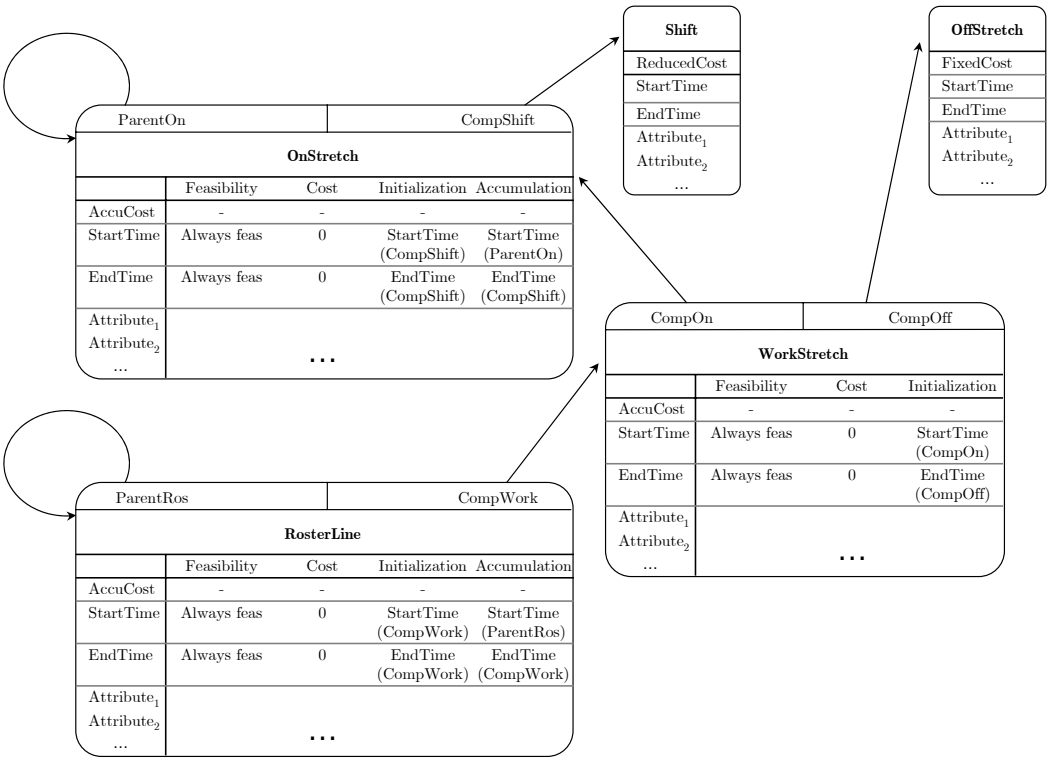


Figure 3: Recursive definition of entities.

For a specific problem, attributes are added to the problem definition as needed and the initialization and accumulation functions are given as part of their definition. By leaving the definition of attributes to the user, the model is kept generic enough to capture the settings

of most practical problems. It is challenging to design the subproblem solution algorithm, so that it can solve any problem that falls under this broad definition without compromising efficiency. The approach that we have taken to achieve this, is described in more detail in Section 4. First, to make it more clear how the attributes are defined in an individual problem, we go back to the example from Middlemore Hospital.

Example (continued). The problem from Middlemore Hospital is modelled as follows:

Shift
ReducedCost
StartTime
EndTime
PaidHours
DaysOn

OffStretch
FixedCost
StartTime
EndTime
DaysOff
WeekendsOff
SingleDaysOff

OnStretch				
	Feasibility	Cost	Initialization	Accumulation
AccuCost	-	-	-	-
StartTime	Always feas	None	StartTime(ParentOn)	StartTime(CompShift)
EndTime	Always feas	None	EndTime(CompShift)	EndTime(CompShift)
PaidH1-2	Always feas	None	PaidHours(CompShift)[1]	PaidH1-2(ParentOn) + PaidHours(CompShift)[1]
PaidH3-4	Always feas	None	PaidHours(CompShift)[2]	PaidH3-4(ParentOn) + PaidHours(CompShift)[2]
DaysOn	LB	Table	DaysOn(CompShift)	DaysOn(ParentOn) + DaysOn(CompShift)
PShiftSTime	Always feas	None	StartTime(CompShift)	StartTime(CompShift)
MornToAft	Always feas	Linear	0	MornToAft(ParentOn) + 1[3]

[1] If CompShift is in week 1-2, =0 otherwise.
[2] If CompShift is in week 3-4, =0 otherwise.
[3] If the difference between StartTime(CompShift) and PShiftSTime(ParentOn) shows that this is a Morning-To-Afternoon transition, =0 otherwise.

WorkStretch			
	Feasibility	Cost	Initialization
AccuCost	-	-	-
StartTime	Always feas	None	StartTime(CompOn)
EndTime	Always feas	None	EndTime(CompOff)
PaidH1-2	Always feas	None	PaidH1-2(CompOn)
PaidH3-4	Always feas	None	PaidH3-4(CompOn)
DaysOn	Always feas	None	DaysOn(CompOn)
DaysOff	Always feas	Table	DaysOff(CompOff)
SingleDaysOff	Always feas	None	SingleDaysOff(CompOff)
FeasibleOnOff	Must be 1	None	1[1]

[1] If the values of DaysOn(CompOn) and SingleDaysOff(CompOff) are compatible, =0 otherwise.

RosterLine				
	Feasibility	Cost	Initialization	Accumulation
AccuCost	-	-	-	-
StartTime	Always feas	None	StartTime(CompWork)	StartTime(ParentRos)
EndTime	Always feas	None	EndTime(CompWork)	EndTime(CompWork)
PaidH1-2	LB and UB	None	PaidH1-2(CompWork)	PaidH1-2(ParentRos) + PaidH1-2(CompWork)
PaidH3-4	LB and UB	None	PaidH3-4(CompWork)	PaidH3-4(ParentRos) + PaidH3-4(CompWork)
DaysOn	UB	None	DaysOn(CompWork)	DaysOn(ParentRos) + DaysOn(CompWork)
DaysOff	UB	None	DaysOff(CompWork)	DaysOff(ParentRos) + DaysOff(CompWork)
SingleDaysOff	UB	None	SingleDaysOff(CompWork)	SingleDaysOff(ParentRos) + SingleDaysOff(CompWork)

The actual bounds and costs are read as a part of the data input and can be varied for each employee. The on-stretch attribute DaysOn is only restricted by a lower bound as the upper bound is enforced by a parameter, as described earlier.

3 Solution method: Master problem

The problem is solved in a branch-and-price framework. The master problem is solved with a standard LP-solver and columns are generated iteratively in the subproblem. If the optimal master problem solution is not integer, branching is used to cut off the fractional solution.

3.1 Master problem

The master problem as defined in Section 2.1 is an LP-problem and we apply a standard solution tool (like CPLEX) to solve it. Preliminary tests show that the main part of the solution time is used in the subproblem solver, so the LP-solver is instructed to always find the optimal solution, as this does not affect the overall solution time notably.

3.2 Branching

Branching is used to remove fractional solution from the solution space of the LP-relaxed master problem. In regular Branch-and-Bound algorithms, variable branching is the method of choice. It is, however, very complex and in most cases highly inefficient to apply variable branching in a Branch-and-Price algorithm. Instead, we use constraint branching where certain constraints are (implicitly) introduced in the current restricted master problem.

Here, we use a specialization of the constraint branching method proposed by Ryan and Foster (1981). If the solution of the restricted master problem is fractional, the columns/roster-lines of one or more employees are in the solution with a fractional value. As two columns of an employee are never identical, two fractionally selected columns will differ in at least one of the included shifts. This in turn means that the employee is not assigned to that shift with a value of 1. In a feasible integer solution, employees are always assigned to shifts (with a value of 1) or not assigned to them at all. We may therefore branch on the employee/shift assignment.

When branching on an employee/shift assignment the set of feasible roster-lines for the employee is split into two subsets: A subset of all roster-lines containing the shift and a subset with the remaining roster-lines. The two branches are created by removing all roster-lines from the first or the second subset, respectively.

In a branch-and-price setup, the columns that have already been added to the restricted master problem are removed as described. The remaining columns are removed implicitly by forcing the subproblem of the particular employee to include or exclude the particular shift.

4 Solution method: Subproblem

The subproblem solution methodology follows directly from the problem definition. The solution process consists of three stages:

1. Shifts are combined into on-stretches.
2. On-stretches and off-stretches are paired to form work-stretches.
3. Roster-lines are generated by sequencing work-stretches.

Feasibility is verified in all three stages. Only feasible on-stretches advance to the following stage, where they are combined with off-stretches. Again, the generated work-stretches are checked and the feasible work-stretches are sequenced in the third stage. A feasible roster-line starting on the first day and spanning the whole horizon is a subproblem solution. If such a roster-line has a negative reduced cost, it may be returned to the master problem.

The three stages are solved separately and in the following, the three solution algorithms are described in detail. On-stretch generation and roster-line generation can be formulated as shortest path problems with resource constraints. Such shortest path problems can be solved with a label setting algorithm, which builds on the concepts of the well-known Dijkstra's algorithm for shortest path problems without resource constraints. An initial *label* representing a (partial) entity is set in the start node. Following, the label is extended to all succeeding nodes. Before extending labels from a new node, the labels of that node are compared against each other to see if any of them are *dominated* by others. If a label is dominated, it means that there is another label with less cost, where any extension of the dominating label is as good as or better than the same extension of the dominated label. The attribute values are compared to ensure this.

4.1 On-stretch generation

In the first stage of the algorithm, on-stretches are generated. An on-stretch can start at any given shift and any later shift may be the last shift of the on-stretch. The problem is modeled as a shortest path problem with resource constraints in a graph where nodes represent shifts. Shortest paths are defined with respect to cost, i.e. the "shortest" path in the graph is the least cost path. The attributes make up the resources of the graph, which may restrict feasibility and introduce additional costs. An example of such a graph is shown in Figure 4.

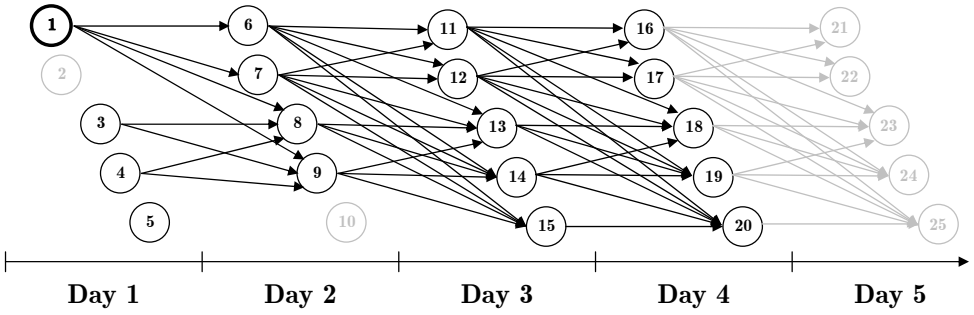


Figure 4: Graph representation of the on-stretch generation problem.

The graph consists of a node for each shift. Arcs between nodes exist when the two shifts are allowed to be consecutive in an on-stretch, i.e. they can be neither too close nor to far in time.

Generating all on-stretches corresponds to finding the all-to-all shortest paths of the graph. We do this by solving a one-to-all shortest path problem from each of the nodes in

the graph. For each of these problems, the start node is selected and the remaining graph is reduced to only allow on-stretches up to the maximum length. Figure 4 shows the shortest path problem with Node 1 as start node. If the maximum on-stretch length is 4 days, all arcs out of nodes 16-20 are removed. Node 2-5 are unreachable from Node 1 and will therefore never hold any labels.

To generate only roster-lines without certain shifts, some nodes may be excluded in the graph of a particular employee. In Figure 4, Shift 2 and Shift 10 have been excluded. Shifts may be disallowed if the employee does not have the appropriate skills and employees may simply be restricted from certain shifts as part of the input data for the problem, e.g. it is common to have employees that can never work night shifts or that have days off on particular days.

The shortest path problems are solved with a label setting algorithm. The graph is acyclic and the nodes can hence be sorted topologically and be treated in that order. All on-stretches are validated by checking the attribute values against the feasibility criterion given in the problem definition. All feasible on-stretches are sent to stage two.

4.2 Work-stretch generation

The second stage is the simplest of the three stages. On-stretches from stage one are combined with off-stretches to form work-stretches. All compatible pairs are examined. Work-stretch attribute values are checked for feasibility and the principle of domination described in Section 4.5 is used to remove unpromising work-stretches before proceeding to the third stage. The second stage is visualized in Figure 5.

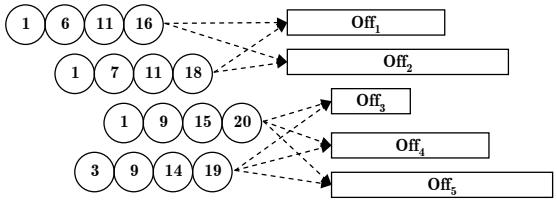


Figure 5: Visualization of the work-stretch generation problem.

4.3 Roster-line generation

The roster-line generation problem is another shortest path problem with resource constraints. The problem has a node for each day in the horizon. The work-stretches generated in stage two are the transitions between days and therefore become the arcs of the graph.

In the generalized rostering problem, we have a predetermined start and end day, and this transfers to a source node and a sink node in the graph. The problem becomes a one-to-one shortest path problem.

The problem is solved using a label setting algorithm. Labels are set for all work-stretches starting on day 0 in the respective end nodes. Work-stretches starting on day 0 may be continuations of work-stretches that started in the previous scheduling horizon. See Engineer (2003) for details.

The labels of the end node represent complete roster-lines. Again, roster-line attributes of these roster-lines have to respect the feasibility criterion given in the problem definition. The best feasible roster-line is also the optimal solution to the subproblem.

The major part of the subproblem solution time is used in the third stage and it is therefore worthwhile to improve the algorithm for this stage as much as possible.

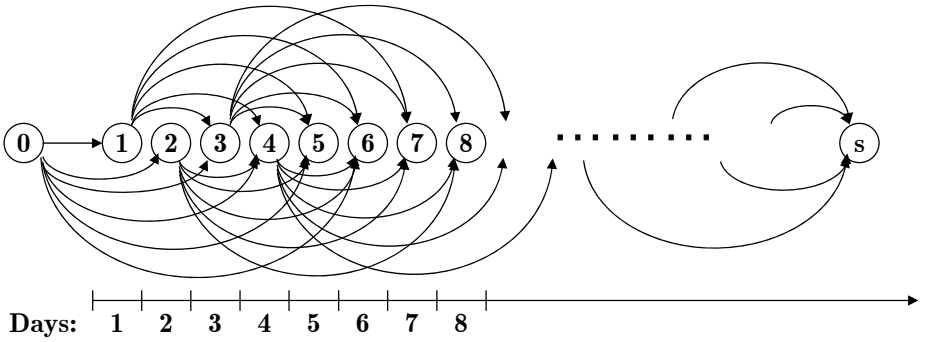


Figure 6: Graph representation of the roster-line generation problem.

4.3.1 Propagating attribute bounds from the end node

For a special type of attributes, it is possible to predict infeasibility previously to reaching the end node. Labels that can never lead to a feasible roster-line should be removed, as this results in fever labels and thereby a more efficient algorithm.

In the general case, the feasibility of an attribute can be described by an arbitrary set of values. To move the feasibility criterion from the end node to previous nodes, an inverse accumulation function is required. All arcs of the graph are given, so we know which transitions can be taken between nodes. Initially, the feasible attribute values of the end node are given. To propagate the feasible attribute values to a preceding node, we need a function that given a set of feasible values outputs a set of all values that can possibly accumulate to those values.

This is easy, if all attributes are additive. The inverse function is subtraction. For a particular attribute, each outgoing arc of a node will add a certain amount to the value of this attribute. The arc leads to a succeeding node, and feasible values of the current node are found by subtracting the accumulation value from each of the feasible values of the succeeding node. The complete set of feasible values of the current node is the union of the feasible value sets of all the outgoing arcs. As the graph is acyclic, the feasible values of all nodes are found by running through the nodes in a reversed topological order.

If the feasibility of an attribute is defined by bounds on the attribute value, the calculations are similar. We are still considering a set of feasible values, now defined by bounds. The lower bound of a node is set to the minimum of all successor lower bounds minus the respective accumulation value of the arc. Analogously, the upper bound is the maximum of succeeding upper bounds minus accumulation value. An example is shown in Figure 7.

Instead of requiring additivity for all attributes, we divide the attributes into two categories: attributes with bounds that can be propagated and attributes for which we will not propagate the feasibility criterion. For simplicity, we have chosen to limit propagation to bounded attributes. The feasibility criterion of all attributes that we have seen in practice can be expressed by bounds. To propagate bounds in the recursive formulation, we use the initialization function of the roster-line entity and assume that the initialization function for the attributes gives exactly the accumulation factor. In this case, bounds are propagated as described above. If this is not the case, the bounds are not propagated.

The immediate advantage of using the initialization function to propagate bounds is that no problem specific knowledge is required. The attribute values are not required to be monotone (e.g. non-decreasing), and the user does not have to specify additional properties of the attribute values. In the former projects, knowledge of e.g. maximum number of paid hours per day has been used to propagate bounds. These bounds are weaker and require more information from the user.

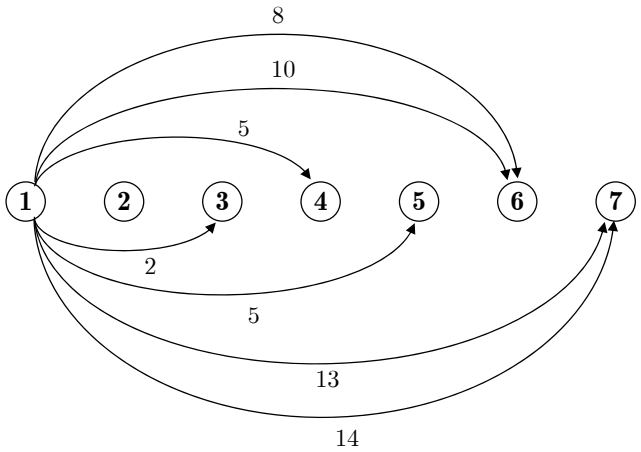


Figure 7: Example of attribute bound propagation. lb_1 and ub_1 are calculated as:
$lb_1 = \min\{13-2, 12-5, 14-5, 16-8, 16-10, 20-13, 20-14\} = 6$,
$ub_1 = \max\{16-2, 14-5, 14-5, 18-8, 18-10, 24-13, 24-14\} = 14$.

4.3.2 Propagating attribute value domains from the start node

In the same way as bounds can be propagated backward from the end node, we can also propagate the value domain of attributes forward from the start node. In this way, for each node, we get a set of possible attribute values. Any value in the domain which is not in the set of feasible values can be removed.

The value domains can be used to eliminate arcs from the graph. We check all arcs against the value domain of their start node combined with the feasibility set of the end node. If none of the values of the domain are accumulate to values of the feasibility set using that arc, it can be eliminated from the graph. This check is only valid for additive attributes, but the arc is eliminated if the condition is violated for one or more of the attributes. Figure 8 visualizes the idea.

See Appendix D for some general thoughts on bound propagation and for more ideas on improving the shortest path algorithm.

4.4 Applying branching decisions

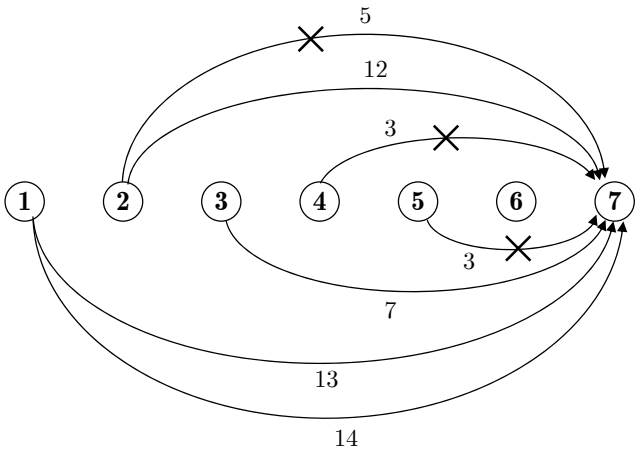
There are two ways that branching decisions can change the subproblem as described in Section 3.2. A branching decision may enforce or prohibit a shift.

If a shift is prohibited, the appropriate node is excluded from the graph of the on-stretch generation stage. This means that no on-stretches will contain the shift and therefore no roster-lines contain the shift.

Enforcing a shift requires modifications in both the on-stretch generation and in the work-stretch generation. In the graph of stage one, all shifts on the same day as the enforced shift, are excluded from the graph (See Figure 9). In stage two, all off-stretches that overlap in time with the enforced shift are excluded.

4.5 Domination

Domination is an important concept in all three stages of the subproblem solution algorithm. So far we have not discussed how labels are actually compared in order to find the dominated ones. The work-stretch generating stage described is not truly a label-setting algorithm, but



Bounds:	[6;14]	[8;9]	[13;16]	[12;14]	[14;14]	[16;18]	[20;24]
Domains:	[7;10]	[8;9]	[13;15]	[12;14]	[14;14]	[17;18]	[dl_7;du_7]

Figure 8: Example of attribute domain propagation. dl_7 and du_7 are calculated as:
$dl_7 = \max(\min\{7+13, 7+14, 7+5, 7+12, 10+7, 10+3, 11+3\}, 20) = \max(12, 20) = 20$,
$du_7 = \min(\max\{10+13, 10+14, 11+5, 11+12, 17+7, 17+3, 18+3\}, 24) = \min(24, 24) = 24$.
To eliminate arcs, the lower bound of the start node plus the accumulation value is compared with the upper bound of the end node:
$7+13 \leq 24$,$7+14 \leq 24$, $7+5 \leq 24$, $7+12 \leq 24$, $10+7 \leq 24$, $10+3 \leq 24$, $11+3 \leq 24$.
Also, the upper bound of the start node plus the accumulation value is compared with the lower bound of the end node:
$10+13 \geq 20$, $10+14 \geq 20$, $9+5 \geq 20$, $9+12 \geq 20$, $15+7 \geq 20$, $14+3 \geq 20$, $14+3 \geq 20$.
This check eliminates the three arcs which have been crossed out in the figure.

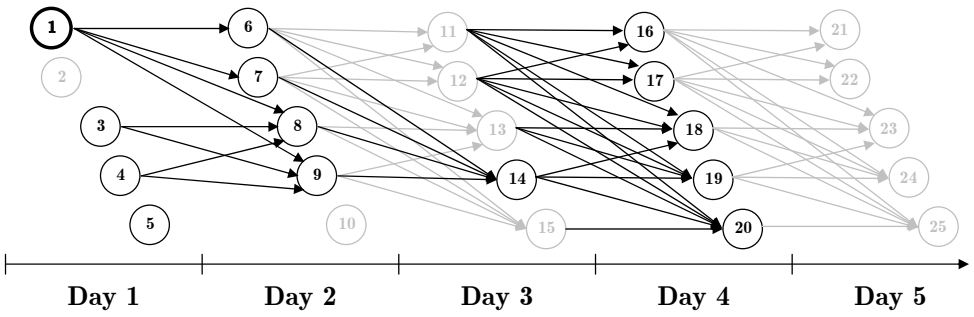


Figure 9: The graph of Figure 4 when shift 14 has been enforced by a branching decision.

the work-stretches generated can still be considered as labels and domination is applied as in the shortest path problems.

A label l_a may be dominated if another label l_b exists, where the cost of l_b is less than or equal to the cost of l_a. Any feasible extension of l_a must also be a feasible extension of l_b, and with the extension, l_b must still cost no more than l_a. The values of the attributes are compared to ensure that the extension of l_b is always better than the extension of l_a.

It depends on the attribute, what a better value is and this must therefore be specified as a part of the problem definition. A very strict requirement that is always valid is to allow domination only when the values of the attribute are equal. If this is required for all attributes, a feasible extension of l_a is always a feasible extension of l_b and the cost

of the labels are affected equally. Problem insight may relax the requirement for some attributes. E.g., for some attributes a lower value is always to prefer. This happens if there is no (effective) lower bound on the attribute value and if costs related to that attribute are non-decreasing. There may even be attributes where the value can be ignored when checking dominance. This would typically be attributes that only have an immediate effect on the cost or the feasibility of the label and where the attribute value is not used in further accumulations.

An important aspect of dominance is that we must always remember to account for indirect effects. An attribute may be unbounded and with no related costs, but if it contributes to the calculation of another attribute value, it may indirectly affect cost and feasibility. Indirect effects may go across the stages of the subproblem solution algorithm. For this reason, it is very hard, to infer dominance requirements of attributes automatically. We therefore leave it to the user to specify those requirements. A safe setting is to only allow dominance for equal attribute values. However, the algorithm is more efficient, the more relaxed the domination requirements are.

5 Implementation

A major focus of this project has been on the implementation. From the beginning the goal was to make an implementation, which can capture the generic problem definition given in Section 1.4. At the same time it must be as efficient as a tailored implementation. This rather ambitious goal was reached by creating an adjustable implementation, where the problem definition is part of the input to the compiler, and the code implicitly includes the problem definition. For a given problem, the definition is integrated in the rest of the code. The generated compilation will behave as if the code was purpose built for that problem. That said, there may naturally be problem specific properties that are not exploited, as we are still building on a very general model. This setup requires the code to be recompiled when it has been adapted to a new problem. It is thereafter possible to solve multiple instances of the same problem with the executable program.

The implementation has been coded in C++ using the branch-and-cut-and-price framework of COIN-OR (Lougee-Heimer, 2003) and the Preprocessor Library of Boost (Karvonen and Mensonides, 2001). The code was continuously checked for memory leaks using the memory checking tool of Valgrind (Seward and Nethercote, 2005). The implementation has been optimized using the program profiler GProf (Graham et al., 1982). The subproblem solver is implemented as a stand alone module and can therefore be compiled and tested independently. This was very useful for debugging.

5.1 The user-file

All the user has to change to solve a new problem is the so called user-file. The user-file contains the problem definition and the definition is parsed into the code where needed. A design decision had to be made on the format to be used for formulating the problem definition. In the current version, this format is build to fit the Boost Preprocessor Library. This has the immediate advantage of not requiring an external parser. The Boost commands are build on regular C++ syntax and hence any standard C++ compiler will parse the problem definition in this format. This format may, however, not be completely intuitive. In a future version it may be an idea to change to a more readable format, like e.g. xml, and develop a dedicated model parser.

The user-file follows the problem definition presented in Section 1.4 closely. To solve a given instance of a problem, it is, obviously, necessary to read the instance data from somewhere. Therefore, in practice, the problem definition contains string names for things that have to be read from an external source.

We illustrate the usage of the user-file by the example from before.

Example (continued). First, a few parameters are defined as described in Section 2.2.

```
#define MAX_DAYS_IN_ONSTRETCH 6
#define MIN_MINUTES_BETWEEN_SHIFTS_END_TO_START   480
#define MIN_MINUTES_BETWEEN_SHIFTS_START_TO_START 1435
#define MAX_MINUTES_BETWEEN_SHIFTS_START_TO_START 2050
#define MAX_MINUTES_BETWEEN_SHIFT_OFFSTRETCH_END_TO_START 720
```

The definitions of shifts and off-stretches are straight forward. The keyword `ATT` is short for the Boost command `BOOST_PP_LIST_CONS` and similarly `END` is short for `BOOST_PP_LIST_NIL`. It is not essential to know these Boost commands to understand the definitions. The attributes are defined as lists of single attribute definitions. In the definition, components/-parents are referred to with a single letter identifying their type: shift (`s`), off-stretch(`f`), on-stretch(`o`), work-stretch(`w`), roster-line(`r`). Below, we show the definition of shifts and off-stretches. The attribute definitions have three arguments: internal attribute name, attribute value type, and string name.

```
# define SHIFT_ATTRIBUTES \
 ATT( (starttime     , int, "Starttime"), \
 ATT( (endtime       , int, "Endtime")  , \
 ATT( (paidhours     , int, "PaidHours"), \
 ATT( (dayson        , int, "DaysOn")   , \
 END )))))
#define SHIFT_NAME_TITLE "ShiftName"
#define SHIFT_COST_TITLE "Cost"
#define SHIFT_CONSTRAINTS_TITLE "Constrain Indices"

# define OFFSTRETCH_ATTRIBUTES \
 ATT( (starttime     , int, "Start Time"), \
 ATT( (endtime       , int, "End Time")  , \
 ATT( (daysoff       , int, "DaysOff"), \
 ATT( (weekendsoff   , int, "WeekendsOff"), \
 ATT( (singlesoff    , int, "Single Days off"), \
 END )))))
#define OFFSTRETCH_COST_TITLE "Cost"
```

The definitions of composite entities are not as simple, but they still follow the definition presented in Section 1.4 strictly. As described in Section 4.5 it has been necessary to add the domination criterion to the definition. The on-stretch is defined with 8 arguments: internal attribute name, attribute value type, string name, feasibility type, domination type, cost type, accumulation function, and initialization function. Additional parameter values needed for the feasibility type or the cost type are read from the data input using the string name of the attribute.

```
# define ONSTRETCH_ATTRIBUTES \
 ATT( (paidhours12   , int, ""          , feas_all , domi_exact    , cost_none  , \
        o.paidhours12 + ((s.starttime < 20160)? s.paidhours : 0)                , \
        (s.starttime < 20160)? s.paidhours : 0                              ) , \
 ATT( (paidhours34   , int, ""          , feas_all , domi_exact    , cost_none  , \
        o.paidhours34 + ((s.starttime >= 20160)? s.paidhours : 0)               , \
        (s.starttime >= 20160)? s.paidhours : 0                             ) , \
 ATT( (starttime     , int, ""          , feas_all , domi_exact    , cost_none  , \
        o.starttime                                                             , \
        s.starttime                                                         ) , \
 ATT( (endtime       , int, ""          , feas_all , domi_exact    , cost_none  , \
        s.endtime                                                               , \
        s.endtime                                                           ) , \
 ATT( (dayson        , int, "odayson"   , feas_lbub, domi_exact    , cost_lookup, \
        o.dayson + s.dayson                                                     , \
        s.dayson                                                            ) , \
 ATT( (morntoaft     , int, "morntoaft" , feas_all , domi_preferlow, cost_linear, \
        o.morntoaft + ((s.starttime - o.prevsstarttime > 1445                   \
                    && s.starttime - o.prevsstarttime <= 1890) ? 1 : 0) , \
        0                                                                   ) , \
 ATT( (prevsstarttime, int, ""          , feas_all , domi_none     , cost_none  , \
        s.starttime                                                             , \
        s.starttime                                                         ) , \
 END )))))))
```

The work-stretch is defined similarly to the on-stretch, but has no accumulation function, and hence takes the arguments: internal attribute name, attribute value type, string name, feasibility type, domination type, cost type, and initialization function.

```
# define WORKSTRETCH_ATTRIBUTES \
 ATT( (paidhours12   , int, ""          , feas_all , domi_exact    , cost_none  , \
        o.paidhours12                                                       ), \
```

```
 ATT( (paidhours34   , int, ""          , feas_all , domi_exact    , cost_none  , \
        o.paidhours34                                                          ), \
 ATT( (starttime     , int, ""          , feas_all , domi_exact    , cost_none  , \
        o.starttime                                                            ), \
 ATT( (endtime       , int, ""          , feas_all , domi_exact    , cost_none  , \
        f.endtime                                                              ), \
 ATT( (dayson        , int, ""          , feas_all , domi_preferlow, cost_none  , \
        o.dayson                                                               ), \
 ATT( (daysoff       , int, "wdaysoff"  , feas_all , domi_preferlow, cost_lookup, \
        f.daysoff                                                              ), \
 ATT( (singledaysoff , int, ""          , feas_all , domi_preferlow, cost_none  , \
        f.singlesoff                                                           ), \
 ATT( (feasonoff     , int, "fe"        , feas_lbub, domi_none     , cost_none  , \
        (int)(o.dayson < 5 || !f.singlesoff)                                   ), \
 END ))))))))
```

Finally, the definition of roster-lines takes the same arguments as the definition of on-stretches: internal attribute name, attribute value type, string name, feasibility type, domination type, cost type, accumulation function, and initialization function.

```
# define ROSTERLINE_ATTRIBUTES \
 ATT( (paidhours12   , int, "2a"        , feas_lbub, domi_exact    , cost_none  , \
        r.paidhours12 + w.paidhours12                 , w.paidhours12          ), \
 ATT( (paidhours34   , int, "2b"        , feas_lbub, domi_exact    , cost_none  , \
        r.paidhours34 + w.paidhours34                 , w.paidhours34          ), \
 ATT( (dayson        , int, "4"         , feas_lbub, domi_preferlow, cost_none  , \
        r.dayson + w.dayson                           , w.dayson               ), \
 ATT( (daysoff       , int, "5"         , feas_lbub, domi_preferlow, cost_none  , \
        r.daysoff + w.daysoff                         , w.daysoff              ), \
 ATT( (singledaysoff , int, "16"        , feas_lbub, domi_preferlow, cost_none  , \
        r.singledaysoff + w.singledaysoff             , w.singledaysoff        ), \
 ATT( (endtime       , int, ""          , feas_all , domi_preferlow, cost_none  , \
        w.endtime                                     , w.endtime              ), \
 END ))))))
```

In this way, the problem definition is formulated in a syntax that can be parsed by any C++ compiler. It would be stretching it very far, to say that this is C++ code, but in the following section, we show how the definition is converted to real code in the appropriate parts of the code.

5.2 Code preprocessing

We are not going to present all the code for the entities, but in the following, we will present the most important ideas that have enabled us to integrate the specific problem definition in a generic solution algorithm. An essential part of the code is the definition of the attribute object. The code for this is found in Appendix C.

We illustrate the idea by example.

Example (**continued**)**.** The shift class contains a number of variables and functions, but most importantly it has a field for each of the attributes, defined by:

```
#define STYPE(elem, n)   Attribute<BOOST_PP_CAT(1, n), BOOST_PP_TUPLE_ELEM(3,1,elem), \
                               accu_none, init_none, feas_all, domi_exact, cost_none>

#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);
```

In this example, the code above unrolls to the following:

```
Attribute<10, int, accu_none, init_none, feas_all, domi_exact, cost_none> starttime;
Attribute<11, int, accu_none, init_none, feas_all, domi_exact, cost_none> endtime;
Attribute<12, int, accu_none, init_none, feas_all, domi_exact, cost_none> shifttype;
Attribute<13, int, accu_none, init_none, feas_all, domi_exact, cost_none> paidhours;
Attribute<14, int, accu_none, init_none, feas_all, domi_exact, cost_none> dayson;
```

The remaining part of the class is mainly used to accommodate data input and output. The off-stretch class is designed similarly to the shift class.

For the composite entities, more functionality is needed in the classes. We must be able to initialize and accumulated entities correctly. Also, these entities must be checked for feasibility and compared in domination checks. This is implemented by letting the attributes themselves know how to initialize, accumulate, and check feasibility and domination. E.g. an

on-stretch attribute knows, by calling its initialize-function, how to initialize its own value. All it needs is a shift-object to initialize from.

The on-stretch class has the following fields and sub-classes related to its attributes:

```
#define OTYPE(elem, n)   Attribute<BOOST_PP_CAT(2, n), BOOST_PP_TUPLE_ELEM(8,1,elem), \
  BOOST_PP_CAT(accu_, BOOST_PP_CAT(2, n)), BOOST_PP_CAT(init_, BOOST_PP_CAT(2, n)),   \
  BOOST_PP_TUPLE_ELEM(8,3,elem), BOOST_PP_TUPLE_ELEM(8,4,elem),                       \
  BOOST_PP_TUPLE_ELEM(8,5,elem)>

#define OACCU(_1, _2, i, elem)                                                        \
  template<class T_value> class BOOST_PP_CAT(accu_, BOOST_PP_CAT(2, i)) {             \
    public: T_value accumulate (const OnStretch& o, const Shift& s) const {          \
      return (BOOST_PP_TUPLE_ELEM(8,6,elem));                                         \
    }                                                                                 \
  };;
BOOST_PP_LIST_FOR_EACH_I(OACCU, _, ONSTRETCH_ATTRIBUTES);

#define OINIT(_1, _2, i, elem)                                                        \
  template<class T_value> class BOOST_PP_CAT(init_, BOOST_PP_CAT(2, i)) {             \
    public: T_value initialize (const Shift& s) const {                              \
      return (BOOST_PP_TUPLE_ELEM(8,7,elem));                                         \
    }                                                                                 \
  };;
BOOST_PP_LIST_FOR_EACH_I(OINIT, _, ONSTRETCH_ATTRIBUTES);

#define OATTR(_1, _2, i, elem)  OTYPE(elem, i) BOOST_PP_TUPLE_ELEM(8,0,elem);;
BOOST_PP_LIST_FOR_EACH_I(OATTR, _, ONSTRETCH_ATTRIBUTES);
```

This unroll to the following for `paidhours12`:

```
template<class T_value> class accu_20 {
public:
    T_value accumulate (const OnStretch& o, const Shift& s) const {
        return (o.paidhours12 + ((s.starttime < 20160)? s.paidhours : 0));
    }
};

template<class T_value> class init_20 {
public:
    T_value initialize (const Shift& s) const {
        return ((s.starttime < 20160)? s.paidhours : 0);
    }
};

Attribute<20, int, accu_20, init_20, feas_all, domi_exact, cost_none> paidhours12;
```

The attribute values are initialized in the constructor. The constructor calls the constructors of each of the attributes and these will in turn call the appropriate initialization or accumulation functions. The on-stretch constructor makes the following initializations (among other things that are not shown here):

```
OnStretch(const Shift& s) :
    // ... lines omitted ...
#   define OCONS1(_1, _2, elem)  , BOOST_PP_TUPLE_ELEM(8,0,elem)(s)
    BOOST_PP_LIST_FOR_EACH(OCONS1, _, ONSTRETCH_ATTRIBUTES)
{}
```

This unrolls to:

```
OnStretch(const Shift& s) :
    // ... lines omitted ...
    paidhours12(s),
    paidhours34(s),
    starttime(s),
    endtime(s),
    dayson(s),
    morntoaft(s),
    prevsstarttime(s)
{}
```

As shown above, the on-stretch constructor passes the shift to the attribute constructors. They will in turn pass it to the initialization function, which knows what information to extract from the shift, as it was illustrated for `paidhours12`.

In the same way, the on-stretch also has a function to check feasibility. An on-stretch is feasible only if all attributes have feasible values. Therefore, the feasibility check is implemented as:

```
inline bool OnStretch::checkFeasibility() const {
#   define OCHKF(_1, _2, elem)                                                    \
              if(!(BOOST_PP_TUPLE_ELEM(8,0,elem).checkFeasibility())) return false;;
    BOOST_PP_LIST_FOR_EACH(OCHKF, _, ONSTRETCH_ATTRIBUTES);

    return true;
}
```

This unrolls to:

```
inline bool OnStretch::checkFeasibility() const {
    if(!(paidhours12.checkFeasibility())) return false;
    if(!(paidhours34.checkFeasibility())) return false;
    if(!(starttime.checkFeasibility())) return false;
    if(!(endtime.checkFeasibility())) return false;
    if(!(dayson.checkFeasibility())) return false;
    if(!(morntoaft.checkFeasibility())) return false;
    if(!(prevsstarttime.checkFeasibility())) return false;

    return true;
}
```

The `checkFeasibility()`-function of the attributes is identified by the feasibility type in the attribute definition. For example, an attribute with the `feas_all` feasibility type has the following function:

```
bool checkFeasibility() const {
    return true;
}
```

The function is obviously redundant, but this format makes the setup very versatile. We make sure that functions are declared `inline` and that the compiler is set to optimize the program. Compiler optimization will remove the overhead introduced by the redundant functions.

Feasibility type `feas_lbub` introduces the following function, where the bound values have been initialized appropriately:

```
bool checkFeasibility() const {
    return (this->value >= lb && this->value <= ub);
}
```

The implementation of domination is similar to the implementation presented for feasibility check.

6 Conclusions and future work

In this project, we have successfully implemented a Branch-and-Price algorithm to solve the generalized rostering problem. The solution approach builds on a generic model and hence allows solution of problems with varying characteristics. A literature study indicated that there is a gap between the current rostering research and the requirements from the end users. From the literature and by looking at the rostering problems at hand, it was clear that a solution approach to the generalized rostering problem must be very flexible. At the same time, rostering problems are typically highly constrained and it is often a demanding task to even find feasible solutions. Therefore the solutions approach must not only be flexible, but also very efficient.

To meet these requirements, the problem was modeled as a generalized set partitioning problem and a branch-and-price algorithm was built to solve the problem. The column generating subproblem is modeled in three stages which is essential to make the problem tractable in realistic settings. A number of improvements of the subproblem solution algorithm were also discussed.

To give maximum flexibility and maximum performance, the problem definition was included implicitly in the implementation. This allows the user to specify rules and constraints which are special for a particular problem. The code is recompiled when a new problem definition is given, and this ensures a high efficiency throughout the algorithm. Using the approach presented here, it is possible to model all constraints seen in the 10 application

found in former projects as well as all commonly occurring constraints listed by Cheang et al. (2003) and all but one of the constraints in Burke et al. (2009). The last constraint models tutorship, and it is not straight forward to introduce a dependency between employees in the model, as this cannot be dealt with in the subproblem. It would, naturally, be very interesting to extend the model to also capture this last constraint type.

The value of the algorithm was illustrated for two different nurse rostering applications. It was possible to find optimal solutions for both applications. The solution time is, however, still too high for the algorithm to be directly applicable in practice.

Future work should, in our opinion, primarily be focus on algorithmic improvements and on extending the number of applications.

Algorithmic improvements could be in the master problem, where e.g. dual stabilization has been employed with success in similar problems. It would also be interesting to run experiments, where parts of the master problem are locked while the remaining problem is solved.

The algorithm has so far been able to prove optimality of found solutions. In practice, this is very seldom a desired feature, and future work on the algorithm should try to incorporate heuristic choices, wherever possible, to make the algorithm faster. It would be very nice, if such improvements could be turned on and off as desired. This would mean that optimal solutions could still be found for testing and benchmarking purposes. Even when proving optimality, most columns can still be generated with heuristics. In the best case, an exact pricing problem is only used to verify optimality.

Including the setup for new problems is hopefully relatively uncomplicated. On the way, it may uncover required features that are not currently supported in the algorithm. Hopefully, it will be easier and easier to adapt to new problems as new modeling features are made available. It is certainly our belief that it is possible to capture all important features in this framework. This would be verified by modeling new rostering problems from the real-world as well as problems from the literature. A good place to start may be with the datasets collected by the EURO Working Group on Automated Timetabling (Curtois, 2009).

When building decision support software, it is important to involve the industrial partners early in the process. The algorithm developed here is founded in work which has been carried out in close cooperation with the industry. However, the model presented has not been evaluated by industrial partners yet, and hence this should be given high priority in the work that follows.

To complete the work on this project there are a few other things that have to be done as well. The whole setup should be tested algorithmically. It would be very nice to get a detailed overview of how time is spent and how the proposed improvements affect solution time. As different settings are tested for new problems, it should also be assessed how the new settings affect solution time. This would eventually put our approach into a greater context. A number of questions should be answered in the work that follows: Can we solve all problems? Which problems can we not solve, and why? How long time does it take to change the setup to a new problem? How long time would we expect it to take, to solve the new problem? What are the limitations with respect to the size of the problems?

Burke et al. state that:

> *The current state of the art is represented by interactive approaches which incorporate problem specific methods and heuristics (that are derived from specific knowledge of the problem and the required constraints) with powerful modern metaheuristics, constraint based approaches and other search methods.* - Burke et al. (2004)

We believe that we have provided a viable alternative to those methods.

References

Al-Yakoob, S. M., H. D. Sherali. 2008. A column generation approach for an employee scheduling problem with multiple shifts and work locations. *Journal of the Operational Research Society* **59**(1) 34–43.

Bard, J. F., H. W. Purnomo. 2005a. A column generation-based approach to solve the preference scheduling problem for nurses with downgrading. *Socio-Economic Planning Sciences* **39**(3) 193–213.

Bard, J. F., H. W. Purnomo. 2005b. Preference scheduling for nurses using column generation. *European Journal of Operational Research* **164**(2) 510–534.

Beliën, J., E. Demeulemeester. 2006. Scheduling trainees at a hospital department using a branch-and-price approach. *European Journal of Operational Research* **175**(1) 258–278.

Beliën, J., E. Demeulemeester. 2008. A branch-and-price approach for integrating nurse and surgery scheduling. *European Journal of Operational Research* **189**(3) 652–668.

Beliën, Jeroen, Erik Demeulemeester. 2007. On the trade-off between staff-decomposed and activity-decomposed column generation for a staff scheduling problem. *Annals of Operations Research* **155**(1) 143–166.

Berghe, G. Vanden. 2002. An advanced model and novel meta-heuristic solution methods to personnel scheduling in healthcare. Ph.D. thesis, School of Computer Science and Information Technology, University of Nottingham.

Bliddal, C., O. Tranberg. 2002. Vagtplanlægning med constraint programming. Master's thesis, Informatics and Mathematical Modelling, Technical University of Denmark, Denmark.

Burke, E. K., T. Curtois, R. Qu, G. Vanden-Berghe. 2009. Problem model for nurse rostering benchmark instances. Tech. rep., ASAP, School of Computer Science, University of Nottingham, Jubilee Campus, Nottingham, UK. Http://www.cs.nott.ac.uk/∼tec/NRP/papers/ANROM.pdf.

Burke, E. K., P. De Causmaecker, G. V. Berghe, H. Van Landeghem. 2004. The state of the art of nurse rostering. *Journal of Scheduling* **7**(6) 441–499.

Chabrier, A. 2006. Vehicle routing problem with elementary shortest path based column generation. *Computers and Operations Research* **33**(10) 2972–2990.

Cheang, B., H. Li, A. Lim, B. Rodrigues. 2003. Nurse rostering problems–a bibliographic survey. *European Journal of Operational Research* **151**(3) 447–460.

Curtois, T. 2009. EURO working group on automated timetabling. Homepage. Http://www.asap.cs.nott.ac.uk/watt/resources/employee.html.

Desrochers, M. 1988. An algorithm for the shortest path problem with resource constraints. Tech. rep., Technical Report Les Cahiers du GERAD G-88-27, University of Montreal, Montreal.

Desrosiers, J., M. E. Lübbecke. 2005. A primer in column generation. G. Desaulniers, J. Desrosiers, M.M. Solomon, eds., *Column Generation*, chap. 1. Springer, New York, 1–32.

Desrosiers, J., F. Soumis, M. Desrochers. 1984. Routing with time windows by column generation. *Networks* **14**(4) 545–565.

Eitzen, G., D. Panton, G. Mills. 2004. Multi-skilled workforce optimisation. *Annals of Operations Research* **127**(1-4) 359–372.

Engineer, F. G. 2003. A solution approach to optimally solve the generalized rostering problem. Master's thesis, Department of Engineering Science, University of Auckland, New Zealand.

Engineer, Faramroze G., George L. Nemhauser, Martin W.P. Savelsbergh. 2008. Shortest path based column generation on large networks with many resource constraints. Technical report, Georgia Tech, College of Engineering, School of Industrial and Systems Engineering.

Ernst, A. T., H. Jiang, M. Krishnamoorthy, B. Owens, D. Sier. 2004a. An annotated bibliography of personnel scheduling and rostering. *Annals of Operations Research* **127**(1-4) 21–144.

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

Graham, S. L., P. B. Kessler, M. K. McKusick. 1982. Gprof: A call graph execution profiler. *Proceeedings of the SIGPLAN '82 Symposium on Compiler Construction* **17**(6) 120–126.

Irnich, S., G. Desaulniers. 2005. Shortest path problems with resource constraints. G. Desaulniers, Jacques Desrosiers, M.M. Solomon, eds., *Column Generation*, chap. 2. GERAD 25th Anniversary Series, Springer, 33–65.

Jaumard, B., F. Semet, T. Vovor. 1998. A generalized linear programming model for nurse scheduling. *European Journal of Operational Research* **107**(1) 1–18.

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

Kellogg, D. L., S. Walczak. 2007. Nurse scheduling: From academia to implementation or not? *Interfaces - Baltimore* **37**(4) 355.

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).

Lübbecke, M. E. 2005. Dual variable based fathoming in dynamic programs for column generation. *European Journal of Operational Research* **162**(1) 122–125.

Maenhout, B., M. Vanhoucke. 2008. Branching strategies in a branch-and-price approach for a multiple objective nurse scheduling problem. Tech. rep., Faculty of Economics and Business Administration, Ghent University, Belgium.

Mason, A. J., M. C. Smith. 1998. A nested column generator for solving rostering problems with integer programming. L. Caccetta, K. L. Teo, P. F. Siew, Y. H. Leung, L. S. Jennings, V. Rehbock, eds., *International Conference on Optimisation: Techniques and Applications*. 827–834.

Nielsen, D. 2003. A broad application optimisation-based rostering model. Ph.D. thesis, Department of Engineering Science, University of Auckland, New Zealand.

Poulsen, H. T. 2004. Vagtplanlægning for sygeplejersker - et kombinatorisk optimeringsproblem. Master's thesis, Datalogisk Institut, University of Copenhagen, Denmark.

Righini, G., M. Salani. 2006. Symmetry helps: Bounded bi-directional dynamic programming for the elementary shortest path problem with resource constraints. *Discrete Optimization* **3**(3) 255–273.

Ryan, D. M., B. Foster. 1981. An integer programming approach to scheduling. *Computer Scheduling of Public Transport. Urban Passenger Vehicle and Crew Scheduling. Proceedings of an International Workshop* 269–280.

Seward, J., N. Nethercote. 2005. *Proceedings of the General Track. 2005 USENIX Annual Technical Conference* 17–30.

Smith, M. C. 1995. Optimal nurse scheduling using column generation. Master's thesis, Department of Engineering Science, University of Auckland, New Zealand.

A Optimal solution of the Middlemore test instance

Total obj = 23

Nurse	Cost	Days 0 1 2 3 4 5 6	7 8 9 10 11 12 13	14 15 16 17 18 19 20	21 22 23 24 25 26 27
0	0	M M M M M - -	M M M M M - -	M M M M M - -	M M M M M - -
1	0	M M M - - - -	M M M M M - -	M M M M M - -	- - M M M - -
2	0	M M M M M - -	M M M M M - -	A A A A A - -	M M M M M - -
3	0	M M M M M - -	M M M M M - -	M M M M M - -	M M M M M - -
4	0	- - M M M M M	- - M M M M M	- - M M M M M	- - M M M M M
5	0	N - - N N N N	- - N N N N N	- - N N N N N	- - N N N N N
6	0	M M M M M - -	M M M M M - -	M M M M M - -	M M M M M - -
7	0	M M M M M - -	M M M M M - -	M M M M M - -	M M M M - - -
8	0	A A A - - M M	M - - - M M M	M M - - - A A	A A - - M M M
9	0	N N - M M - -	M M M M M - -	- M M M M - -	M M M M M - -
10	0	M M M - - - -	M M M M M - -	M M M M M - -	M M M - - - -
11	0	M M M M M - -	M M - - - - -	M M M M M - -	- - - M M - -
12	0	6 6 6 - - - -	6 6 6 6 6 - -	6 6 6 6 6 - -	- - 6 6 6 - -
13	0	8 6 6 6 - - -	8 6 - - - - -	8 6 6 6 - - -	8 6 - - - - -
14	0	- 6 6 6 6 - -	- - - 6 6 - -	6 6 6 - - 6 6	6 - - - - - -
15	0	6 6 - - - - -	6 6 - - - M M	- - - - - - -	6 6 6 6 - - -
16	0	M M - - - - -	- M M - - - -	- - - - - - -	M M M M - - -
17	0	M M M M M - -	M M M M - - -	M M M M M - -	M M M M - - -
18	0	A - - - M M M	M M - - A A A	A A - - - A A	A A A - - A A
19	0	M M M M M - -	M M M M M - -	M M M M M - -	M M M M M - -
20	0	M M - - M M M	M M - - M M M	M M - - M M M	M M - - M M M
21	10	- - A A A A A	- - A A A A A	- - A A A A A	- - A A A A A
22	0	M M - - M M M	- - M M M - -	M M M M M - -	- M M M - - -
23	0	M M M M M - -	- - - M M M -	- M M M M - -	- - M M M - -
24	0	- - - - - - -	6 6 6 - - - -	- - - 6 6 6 6	- - - 6 6 - -
25	0	- - M M M M M	- - M M M M M	- - M M M M M	- - M M M M M
26	0	M M M M M - -	M M M M M - -	M M - - M M M	M M - - A A A
27	0	M M - - A A A	A A - - A A A	A A - - A A A	- - M M M M M
28	0	A - - M M M M	M - - M M M M	M - - M M M M	M - - A A A A
29	0	M M M M M - -	M M M M M - -	M M M M M - -	M M M M M - -
30	0	M M M - - A A	A - - - A A A	- - - M M M M	- - A A A A A
31	0	- - A A A A A	- - - M M M M	M - - - A A A	A - - A A A A
32	0	M M M M M - -	- A A A A - -	M M M M - - -	M M M M M - -
33	0	A A A A - - -	M M M M M - -	M M M M M - -	M M M M - - -
34	0	M M M M M - -	- M M M M - -	- M M M M - -	M M M M M - -
35	0	- - - - - - -	A A A A - - -	M M M M M - -	- - M M M - -
36	0	M M M M M - -	- 8 8 8 8 - -	- 8 8 8 8 - -	N N N N N - -
37	0	- - - - - N N	- - - M M - -	N - M - - - -	N N - - - - -
38	0	- - 6 6 6 - -	- - 6 6 6 - -	- 6 6 6 6 - -	6 6 - - - - -
39	0	- M M M M - -	M M M M M - -	M M M M - - -	M M M M M - -
40	0	A A - - A A A	- - - - - - -	- - - - - - -	M M M M M - -
41	0	M M M M - - -	M M M M M - -	- M M M M - -	M M M M M - -
42	0	M M M M M - -	M M M M - - -	M M M M M - -	- M M M M - -
43	0	M M M M M - -	M M M M - - -	M M M M - - -	M M M M M - -
44	2	- - 8 8 - - -	M 8 8 - - - -	M 8 8 - - - -	- 8 8 - - - -
45	0	A A A A A - -	A A A A A - -	A A A A - - -	A A - - A A A
46	0	M M M M M - -	M M M M M - -	A A - - M M M	M - - A A A A
47	0	M M M M M - -	M M M M M - -	- N N N - M M	- - M M M M M
48	0	A A A A - - -	A A A A - - -	M M M M M - -	M M M M M - -
49	0	M M M M M - -	- M M M M - -	M M M M M - -	M M M M - - -
50	0	M M M M M - -	M M M M - - -	M M M M - - -	M M M M M - -
51	0	M M M M M - -	M M M M - - -	- M M M M - -	M M M M M - -
52	0	- - - - - - -	- - - - - - -	A A A A A - -	M M M M - - -

53	0	- - - M M M M M	M - - A A A A	A - - - A A A	A A - - M M M
54	0	N N N - - N N	N N - - - N N	N N - - N N N	- M M - - N N
55	0	M - - A A - -	M M M M M M - -	M M M M M M - -	- - A A A - -
56	0	- - - - A A A	A - - M M M M	M - - A A - -	M M M M M - -
57	0	A A - - - - -	- - - - - - - -	- M M M - - -	A A A - - - -
58	1	- - - - - - -	A A A - - A 6	6 - - M M - -	- - - - - M M
59	0	- - - - - - -	- A A A - - -	- - - - - - -	- - A A A - -
60	0	- - A A A - -	- - - - - - -	- - A A A - -	- - - - - - -
61	0	- M M M M - -	M M M M M M - -	M M M M - - -	M M M M M - -
62	0	M M M M M - -	M M M M - - -	- M M M M - -	M M M M M - -
63	0	M M M M M - -	M M M M M M - -	M M M M M - -	M M M M M - -
64	0	- M M M M - -	M M M M M M - -	M M M M - - -	M M M M M - -
65	0	M - - M M M M	M - - - M M M	M - - M M M M	M - - - M M M
66	0	A A A A A - -	M M M M M M - -	M M M M M - -	M M M M M - -
67	0	- - N N N N N	- - N N N N N	- - N N N N N	- - N N N N N
68	0	M M M M M - -	M M M M M M - -	M M M M M - -	M M M M M - -
69	0	- M M M M - -	M M M M M M - -	M M M M - - -	M M M M M - -
70	0	- - - - - 6 6	6 - - - 6 6 6	6 - - - - - -	6 6 6 6 6 - -
71	0	M M M M M - -	M M M M M M - -	M M M M M - -	M M M M M - -
72	0	- N N N N - -	N N N N N - -	N N N N - - -	M M M M M - -
73	0	8 8 8 - - - -	8 8 8 - - - -	8 8 - - - - -	8 8 8 8 - - -
74	10	- 8 8 8 8 - -	N N N N - - -	- 8 8 8 8 - -	N N N N - - -
75	0	M M M M M - -	- M M M M - -	M M M M M - -	A A A A - - -
76	0	A A A A A - -	M M M M - - -	A A A A A - -	M M M M - - -
77	0	A A - - - - -	- M M M M - -	- A A - - - -	M M M M - - -
78	0	M M M M M - -	- - A A A - -	M M M M M - -	A A A - - - -
79	0	N N N N N - -	N N - - N N N	N N - - N N N	N N - - N N N
80	0	- - - A A - -	A A - - - - -	A A A A - - -	- - - - - - -
81	0	M M - - - M M	- - A A - - -	M M M M - - -	- A A - - - -
82	0	M M M - - A A	A - - A A A A	- - M M M M M	- - M M M M M
83	0	M - - M M - -	M M M M M M - -	- - M M M - -	M M M M M - -
84	0	M M M M - - -	- A A - - M M	- - A A A - -	A A A A A - -

Overview of demands with (number of employees assigned) / (demand):

Shifts	Skills	Type	Cost	Days 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
6A	0	≥	0	3/1	3/1	2/1	2/1	3/1	1/1	1/1	4/1	4/1	2/1	2/1	2/1	1/1	1/1	2/1	2/1	1/1	1/1	1/1	1/1	1/1	2/1	2/1	1/1	1/1	2/1	2/1	2/1
6A	2	≥	0	12/9	12/9	12/9	11/9	10/9			10/9	10/9	9/9	10/9	11/9			12/9	12/9	10/9	9/9	10/9			12/9	11/9	10/9	10/9	10/9		
6A	3	=	0	13/13	13/13	13/13	13/13	13/13	8/8	8/8	13/13	13/13	13/13	13/13	13/13	8/8	8/8	13/13	13/13	13/13	13/13	13/13	8/8	8/8	13/13	13/13	13/13	13/13	13/13	8/8	8/8
6A	5	≥	0	2/2	2/2	2/2	3/2	5/2			4/2	4/2	3/2	2/2	2/2			5/2	4/2	3/2	4/2	4/2			2/2	2/2	3/2	3/2	4/2		
6A	6	≥	0	3/3	3/3	3/3	4/3	5/3			5/3	5/3	4/3	4/3	3/3			6/3	4/3	3/3	4/3	5/3			3/3	3/3	3/3	3/3	4/3		
8M	1	≥	0	20/12	17/12	16/12	16/12	17/12			18/12	20/12	19/12	20/12	21/11			18/12	18/12	19/12	20/12	21/12			18/12	16/12	20/12	19/12	16/11		
8M	3	≥	0	43/42	44/42	42/42	42/42	42/42			42/42	43/42	44/42	45/42	41/41			42/42	45/42	46/42	48/42	42/42			43/42	43/42	49/42	47/42	41/41		
8M	6	≥	0	10/5	11/5	10/5	11/5	11/5			11/5	11/5	10/5	11/5	8/5			9/5	11/5	11/5	12/5	9/5			11/5	11/5	13/5	13/5	12/5		
A	3	≥	0	11/9	9/9	9/9	10/9	11/9	7/6	7/6	9/9	9/9	10/9	10/9	9/9	7/6	6/6	9/9	9/9	9/9	9/9	10/9	6/6	6/6	9/9	9/9	10/9	9/9	10/9	8/6	8/6
M	0	≥	0						1/1	1/1						2/1	2/1						3/1	3/1						3/1	3/1
M	2	≥	0						7/7	7/7						8/7	8/7						8/7	8/7						8/7	8/7
M	3	=	0						10/10	10/10						10/10	10/10						10/10	10/10						10/10	10/10
M	6	≥	0						1/1	1/1						1/1	1/1						2/1	2/1						4/1	4/1
N	0	≥	0	2/1	2/1	2/1	1/1	1/1	2/1	2/1	1/1	1/1	1/1	1/1	1/1	2/1	2/1	1/1	2/1	2/1	2/1	2/1	2/1	2/1	1/1	1/1	2/1	2/1	2/1	2/1	2/1
N	2	≥	0	3/3	3/3	3/3	3/3	3/3	4/3	4/3	3/3	3/3	4/3	4/3	3/3	3/3	3/3	3/3	3/3	4/3	4/3	3/3	3/3	3/3	3/3	3/3	4/3	4/3	3/3	3/3	3/3
N	3	=	0	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4	4/4

Overview of demands showing surplus for each demand:

Shifts	Skills	Type	Cost	Days 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27
6A	0	≥	0	2	2	1	1	2	-	-	3	3	1	1	1	-	-	1	1	-	-	-	-	-	1	1	-	-	1	1	1
6A	2	≥	0	3	3	3	2	1			1	1	-	1	2			3	3	1	-	1			3	2	1	1	1		
6A	3	=	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
6A	5	≥	0	-	-	-	1	3			2	2	1	-	-			3	2	1	2	2			-	-	1	1	2		
6A	6	≥	0	-	-	-	1	2			2	2	1	1	-			3	1	-	1	2			-	-	-	-	1		
8M	1	≥	0	8	5	4	4	5			6	8	7	8	10			6	6	7	8	9			6	4	8	7	5		
8M	3	≥	0	1	2	-	-	-			-	1	2	3	-			-	3	4	6	-			1	1	7	5	-		
8M	6	≥	0	5	6	5	6	6			6	6	5	6	3			4	6	6	7	4			6	6	8	8	7		
A	3	≥	0	2	-	-	1	2	1	1	-	-	1	1	-	1	-	-	-	-	-	1	-	-	-	-	1	-	1	2	2
M	0	≥	0						-	-						1	1						2	2						2	2
M	2	≥	0						-	-						1	1						1	1						1	1
M	3	=	0						-	-						-	-						-	-						-	-
M	6	≥	0						-	-						-	-						1	1						3	3
N	0	≥	0	1	1	1	-	-	1	1	-	-	-	-	-	1	1	-	1	1	1	1	1	1	-	-	1	1	1	1	1
N	2	≥	0	-	-	-	-	-	1	1	-	-	1	1	-	-	-	-	-	1	1	-	-	-	-	-	1	1	-	-	-
N	3	=	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-

B Rigshospitalet - Anæstesi- og operationsklinik

B.1 Problem definition

Shift
ReducedCost
StartTime
EndTime
PaidHours
WeekendsOn
WeekendPenalty

OffStretch
FixedCost
StartTime
EndTime
BFEven
BFOdd
DaysOff

OnStretch				
	Feasibility	Cost	Initialization	Accumulation
AccuCost	-	-	-	-
StartTime	Always feas	None	StartTime(ParentOn)	StartTime(CompShift)
EndTime	Always feas	None	EndTime(CompShift)	EndTime(CompShift)
PaidHours	Always feas	None	PaidHours(CompShift)	PaidHours(ParentOn) + PaidHours(CompShift)
WeekendsOn	UB	None	WeekendsOn(CompShift)	WeekendsOn(ParentOn) + WeekendsOn(CompShift)
WeekendPenalty	Always feas	None	WeekendPenalty(CompShift)	WeekendPenalty(ParentOn) + WeekendPenalty(CompShift)

WorkStretch			
	Feasibility	Cost	Initialization
AccuCost	-	-	-
StartTime	Always feas	None	StartTime(CompOn)
EndTime	Always feas	None	EndTime(CompOff)
PaidHours	Always feas	None	PaidHours(CompOn)
BFEven	Always feas	None	BFEven(CompOff)
BFOdd	Always feas	None	BFOdd(CompOff)
WeekendsOn	Always feas	None	WeekendsOn(CompOn)
WeekendPenalty	Always feas	None	WeekendPenalty(CompOn)

RosterLine				
	Feasibility	Cost	Initialization	Accumulation
AccuCost	-	-	-	-
StartTime	Always feas	None	StartTime(CompWork)	StartTime(ParentRos)
EndTime	Always feas	None	EndTime(CompWork)	EndTime(CompWork)
PaidHours	LB and UB	None	PaidHours(CompWork)	PaidHours(ParentRos) + PaidHours(CompWork)
InfeasWeekend	UB	None	0	1[1]
LastWeekendOn	Always feas	None	WeekendsOn(CompWork)	WeekendsOn(CompWork)[2]
BFEven	Always feas	None	BFEven(CompWork)	BFEven(ParentRos)[3] + BFEven(CompWork)
BFOdd	Always feas	None	BFOdd(CompWork)	BFOdd(ParentRos)[4] + BFOdd(CompWork)
BFCost	Always feas	Linear	0	(2 - BFEven(ParentRos))[3] + (2 - BFOdd(ParentRos))[4] + BFCost(ParentRos)
BFCostPartTime	Always feas	Linear	0	(3 - BFEven(ParentRos))[3] + (3 - BFOdd(ParentRos))[4] + BFCost(ParentRos)
WkdPenalty	Always feas	LinearCap	WkdPenalty(CompWork)	WkdPenalty(ParentRos) + WkdPenalty(CompWork)

[1] If LastWeekendOn(ParentRos) = 1 ∧ WeekendsOn(CompWork) = 1, =0 otherwise.
[2] If CompWork starts in one week and ends in another, =LastWeekendOn(ParentRos) otherwise.
[3] If CompWork does not start in an odd week and end in an even week, =0 otherwise.
[4] If CompWork does not start in an even week and end in an odd week, =0 otherwise.

B.2 Optimal solution

Total obj = 286

Nurse	Skills	Cost	Days																										
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	0, 1	0	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	-
1	0, 2	13	N	-	D	N	-	-	-	D	N	-	N	-	-	-	D	D	D	D	L	-	-	N	-	D	D	D	-	N	-
2	0, 3	2	-	D	D	L	N	-	-	D	D	D	D	L	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	-
3	0, 4	2	D	D	D	D	D	-	-	D	D	N	-	D	-	-	L	A	L	D	D	-	-	D	D	-	L	N	-	-	-
4	0, 1	0	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	-
5	0, 2	0	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	-
6	0, 3	2	D	D	D	D	D	-	-	D	L	D	D	L	-	-	D	-	D	D	N	-	-	D	D	D	D	D	-	-	-
7	0, 4	0	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	-
8	0, 1	0	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	-
9	0, 2	2	-	D	D	D	N	-	-	D	D	D	D	D	-	-	D	D	L	D	D	-	-	D	D	D	D	L	-	-	-
10	0, 3	15	D	D	D	D	D	-	-	D	D	D	A	D	-	N	-	D	-	D	D	-	-	D	D	D	D	D	-	-	-
11	0, 4	2	D	D	D	A	L	-	-	D	D	D	D	D	-	-	D	D	D	-	N	-	-	L	D	D	D	D	-	-	-
12	0, 1	2	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	L	-	-	-	L	D	D	N	-	-	-
13	0, 2	2	D	D	A	D	D	-	-	L	D	D	-	N	-	-	D	D	D	D	D	-	-	D	D	D	D	L	-	-	-
14	0, 3	13	D	N	-	L	D	-	-	N	-	D	N	-	-	-	N	-	D	D	D	-	-	N	-	L	N	-	-	N	-
15	0, 4	12	D	L	N	-	D	-	-	A	D	D	D	D	-	-	D	D	N	-	D	-	-	D	N	-	-	D	N	-	-
16	0, 1	2	A	D	D	D	L	-	-	L	D	-	D	N	-	-	D	D	D	D	A	-	-	D	D	D	D	D	-	-	-
17	0, 2	12	D	N	-	D	D	-	-	D	N	-	D	D	-	-	A	D	D	N	-	-	-	D	L	D	D	-	N	-	-
18	0, 3	28	L	D	N	-	-	-	-	N	-	L	L	-	N	-	D	-	N	-	D	-	-	D	D	N	-	-	-	-	-
19	0, 4	25	D	A	D	-	-	-	N	-	D	L	-	D	-	-	D	L	-	L	-	-	N	-	D	-	L	D	-	-	-
20	0, 1	25	-	D	D	D	-	-	N	-	D	-	L	D	-	-	L	L	D	-	-	-	N	-	-	L	D	D	-	-	-
21	0, 2	39	D	D	-	-	A	N	-	D	-	D	D	D	-	-	-	N	-	A	-	N	-	A	N	-	-	D	-	-	-
22	0, 3	36	N	-	D	N	-	-	-	D	D	-	-	A	N	-	N	-	-	D	D	-	-	D	D	D	N	-	-	-	-
23	0, 4	31	-	L	L	-	-	N	-	-	L	N	-	D	-	-	-	N	-	L	-	N	-	L	-	N	-	D	-	-	-
24	0, 1	21	L	-	L	D	D	-	-	D	A	D	D	-	-	N	-	-	A	N	-	-	-	D	A	-	A	A	-	-	-
25	2	0	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	-
26	3	0	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	D	D	A	D	D	-	-	-
27	4	0	D	D	D	D	D	-	-	D	D	A	D	D	-	-	D	D	D	D	D	-	-	D	D	D	D	D	-	-	-

Shifts	Skills	Cost	Days																												
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
A		0	-	-	-	-	-			-	-	-	-	-			-	-	-	-	-			-	-	-	-	-			
D		0	4	5	5	3	5			4	4	2	2	6			4	3	3	4	5			4	3	2	3	6			
D	1	0	3	5	5	6	4			4	5	4	5	4			4	4	5	4	2			4	3	4	5	4			
D	2	0	4	4	3	4	3			5	3	4	4	4			4	5	4	4	3			4	3	5	5	3			
D	3	0	3	4	4	2	3			4	3	4	2	1			3	2	3	5	5			5	5	3	3	3			
D	4	0	5	3	4	2	3			3	5	2	3	6			4	3	2	2	3			3	4	2	2	5			
L		0			-	-	-					-	-	-					-	-	-					-	-	-			
L	0	0	-	-	1	1	1			-	-	1	1	1			-	-	1	1	1			-	-	1	1	1			
N	0	0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	

Shifts	Skills	Cost	Days																												
			0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
A		0	$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$			$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$			$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$			$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$	$\frac{1}{1}$			
D		0	$\frac{19}{15}$	$\frac{20}{15}$	$\frac{20}{15}$	$\frac{18}{15}$	$\frac{17}{12}$			$\frac{20}{16}$	$\frac{20}{16}$	$\frac{18}{16}$	$\frac{18}{16}$	$\frac{19}{13}$			$\frac{19}{15}$	$\frac{18}{15}$	$\frac{18}{15}$	$\frac{19}{15}$	$\frac{17}{12}$			$\frac{20}{16}$	$\frac{19}{16}$	$\frac{18}{16}$	$\frac{19}{16}$	$\frac{19}{13}$			
D	1	0	$\frac{4}{1}$	$\frac{6}{1}$	$\frac{6}{1}$	$\frac{7}{1}$	$\frac{5}{1}$			$\frac{5}{1}$	$\frac{6}{1}$	$\frac{5}{1}$	$\frac{6}{1}$	$\frac{5}{1}$			$\frac{5}{1}$	$\frac{5}{1}$	$\frac{6}{1}$	$\frac{5}{1}$	$\frac{3}{1}$			$\frac{5}{1}$	$\frac{4}{1}$	$\frac{5}{1}$	$\frac{6}{1}$	$\frac{5}{1}$			
D	2	0	$\frac{5}{1}$	$\frac{5}{1}$	$\frac{4}{1}$	$\frac{5}{1}$	$\frac{4}{1}$			$\frac{6}{1}$	$\frac{4}{1}$	$\frac{5}{1}$	$\frac{5}{1}$	$\frac{5}{1}$			$\frac{5}{1}$	$\frac{6}{1}$	$\frac{5}{1}$	$\frac{5}{1}$	$\frac{4}{1}$			$\frac{5}{1}$	$\frac{4}{1}$	$\frac{6}{1}$	$\frac{6}{1}$	$\frac{4}{1}$			
D	3	0	$\frac{4}{1}$	$\frac{5}{1}$	$\frac{5}{1}$	$\frac{3}{1}$	$\frac{4}{1}$			$\frac{5}{1}$	$\frac{4}{1}$	$\frac{5}{1}$	$\frac{3}{1}$	$\frac{2}{1}$			$\frac{4}{1}$	$\frac{3}{1}$	$\frac{4}{1}$	$\frac{6}{1}$	$\frac{6}{1}$			$\frac{6}{1}$	$\frac{6}{1}$	$\frac{4}{1}$	$\frac{4}{1}$	$\frac{4}{1}$			
D	4	0	$\frac{6}{1}$	$\frac{4}{1}$	$\frac{5}{1}$	$\frac{3}{1}$	$\frac{4}{1}$			$\frac{4}{1}$	$\frac{6}{1}$	$\frac{3}{1}$	$\frac{4}{1}$	$\frac{7}{1}$			$\frac{5}{1}$	$\frac{4}{1}$	$\frac{3}{1}$	$\frac{3}{1}$	$\frac{4}{1}$			$\frac{4}{1}$	$\frac{5}{1}$	$\frac{3}{1}$	$\frac{3}{1}$	$\frac{6}{1}$			
L		0			$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$					$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$					$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$					$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$			
L	0	0	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{1}$	$\frac{2}{1}$	$\frac{2}{1}$			$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{1}$	$\frac{2}{1}$	$\frac{2}{1}$			$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{1}$	$\frac{2}{1}$	$\frac{2}{1}$			$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{1}$	$\frac{2}{1}$	$\frac{2}{1}$			
N	0	0	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	$\frac{2}{2}$	

C Attribute.hpp

```
template<int id, class T_value,
template<class>class T_accum,
template<class>class T_init,
template<int,class> class T_feas,
template<class> class T_domi,
template<int,class> class T_cost >
class Attribute :
    public T_accum<T_value>,
    public T_init<T_value>,
    public T_domi<T_cost<id,T_value> >,
    public T_feas<id,T_value>
{
public:
    Attribute() {}
    Attribute(T_value value_) {
        this->value = value_;
    }
    template<class Component2> Attribute(const Component2& c2) {
        this->value = this->initialize(c2);
    }
    template<class Component1, class Component2>
    Attribute(const Component1& c1, const Component2& c2) {
        this->value = this->accumulate(c1, c2);
    }

    bool non_dominating(
        const Attribute<id, T_value, T_accum, T_init, T_feas, T_domi, T_cost> &a,
        double& rcdiff) const {
            return this->non_dominating_base(this->value, a.value, rcdiff);
    }

    operator T_value() const {return this->value;}
    Attribute<id, T_value, T_accum, T_init, T_feas, T_domi, T_cost>&
        operator= (const T_value value_) {
            this->value = value_;
            return *this;
    }
};
```

D Improving the shortest path algorithm

In the following, we describe some ideas to further improve the shortest path algorithm of stage three in the subproblem solver. Most of the improvements suggested here are still of the exact algorithm, and hence give an enhanced performance without compromising optimality. Optimality is, however, not required and much more can be gained by switching to a heuristic approach. In the last subsection we sketch a few ideas on how to do this.

D.1 Bounding the reduced cost

As we are only creating columns with negative reduced cost, we implicitly have a upper bound on the reduced cost of the end node labels (equal to 0). As the reduced costs are added together, this bound may be propagated just as the attributes are. This will remove labels which can never expand to columns with negative reduced cost.

D.2 Safe bounds

As well as propagating bounds and value domains in the graph, we may also calculate *safe bounds*. The intention of a safe bound is to identify attribute values which are so far from their bounds that they will never be able to introduce additional cost and will never lead to infeasible labels. This can be used to dominate more labels. Let us illustrate the idea by a small example. We have two labels l_a and l_b, which only differ in attribute values on attribute a_1. l_b has a lower reduced cost than l_a. If the domination criterion of a_1 requires the values to be equal to allow domination, l_a cannot be dominated. However, we may be able to deduce that the value of a_1 for e.g. l_b is so small that it will never be able to reach its upper bound, but at the same time the value is so large that it is always above the lower bound. In that case, l_a may be removed, as any feasible extension of l_a is also a feasible extension of l_b. We must of course remember to account for indirect effects as well.

It may not seem like a very useful improvement of the algorithm. There are however some cases, where many attribute values fall within the safe bounds. In some cases, the safe bounds even coincide with the propagated feasibility bounds. In such a case, the attribute can be disregarded, altogether, when checking domination. This special case occurs, when an attribute is not changed in the last part of the graph. If, for example, we have an attribute that counts the number of hours worked in the first week, then after the first week, we should be able to evaluate the value of the attribute and thereafter disregard it. The safe bound will capture this without any problem specific knowledge, i.e. it is not necessary to specify a point in time after which the attribute is not changed.

D.3 Bidirectional search

If all attributes are additive, we may solve the shortest path problem with a bidirectional search as described by Righini and Salani (2006) for the vehicle routing problem with time windows. As we are solving a one-to-one shortest path problem, instead of creating all labels from the start node, we can also expand labels backwards from the end node. This requires the definition of an inverse accumulation function for all attributes, which is of course easy, if they are all additive. Requiring additivity for all attributes limits the model significantly, which is against the basic idea of our setup. It may, however, give a significant speed up, so it may be worth considering as a functionality that can be switched on and off depending on the problem definition.

D.4 Faster dominance check for attributes with "equal" requirement

In the domination check, instead of checking labels for attribute values which have to be equal, we may store the labels in a way, so the check is unnecessary. If we store each label in

a set where all other labels of the set hold exactly the same values of the "must-be-equal"-attributes, then we only need to check domination between the labels of this set and the attributes used for storing do not have to be checked. In the implementation, a `Map` or a `HashTable` could be used to store the sets.

D.5 Generalization of bound propagation

So far, we have assumed that attributes are additive when bounds are propagated. This assumption may be relaxed slightly. If an attribute is not additive bounds can still be propagated if:

Non-decreasing value If the value of the attribute is guaranteed to be non-decreasing the upper bound of the end node may be copied to all other nodes. If the value of the attribute ever exceeds the upper bound, it will not be able to decrease below that bound.

Non-increasing value As for a non-decreasing value, a non-increasing value allows us to copy the lower bound of the end node to all other nodes.

Initialization has least effect If the initial attribute value is equal to or less than any effect that the accumulation function may have on the attribute value, then the upper bound may be propagated back from the end node, using the initialization function. As the accumulation function will always increase the value of the attribute with at least the same amount, the bound may be lowered with the value of the initialization function.

Initialization has most effect Similarly, the lower bound may be propagated back from the end node, if the accumulation function always increases the value of the attribute with at most the initialization value.

A special case is where the attribute is additive and the initialization value is equal to the accumulation value. In this case, the initialization has both least effect and most effect and hence both bounds are propagated.

D.6 Dominance by attribute valuation

For some attributes it is possible to put a price on the value difference for a certain attribute. E.g. if we have two labels l_a and l_b, which only differ in attribute values on attribute a_1. If l_a has $a_1 = 0$ and l_b has $a_1 = 1$ and there is a linear cost on a_1, then l_b may still be able to dominate l_a if the difference in reduced cost is already larger than the linear cost on a_1. This is a generalization of an idea described by Chabrier (2006) for node costs.

D.7 Collapse graph for roster-line generation

The graph for roster-line generation contains a very large number of arcs between few nodes. We may be able to use arc aggregation to decrease the size of the network. See Engineer et al. (2008) for more on this.

D.8 Solving the subproblem heuristically

The main time of the shortest path problem solution algorithm is spent creating and comparing labels. Hence, significant speedups can be achieved by reducing the number of labels.

One way of reducing the number of labels is to make the intermediate feasibility criterions stricter, i.e. by imposing stricter bounds. It may sometimes be "almost impossible" for labels to extend to feasible roster-lines in the end node and we may chose to remove these labels. Similarly, as we above described an upper bound on the reduced cost, this bound may be lowered to remove unpromising labels.

The domination criterion can also be heuristic. For attributes that require equal values, we may introduce a tolerance, as describe by Engineer (2003). This could also be extended to attribute values required to be "less than" or "greater than", so they are instead required to be "almost less than" or "almost greater than". If we are using dominance by attribute valuation as described above, the price of the attribute difference may be estimated to be lower than what is theoretically possible. More labels are dominated if the price is lowered. This was also proposed by Chabrier (2006) for node costs.

We may put a hard limit on the number of labels in each node. The smaller this limit is, the faster the algorithm is. It takes careful considerations to choose the remaining labels carefully. However, if chosen wisely, we may not sacrifice much on quality of the generated columns, while the speedup may be substantial. This idea was also employed by Engineer (2003).

In this report, we present a solution approach to the nurse rostering problem. The problem is defined by a generic model that is able to capture close to all of the problem characteristics that we have seen in the literature and in the realistic problems at hand. The model is used directly in the solution algorithm which gives a very versatile solution method. The method at the same time is constructed to exploit a number of problem specific features and thereby we have a both versatile and efficient solution method. The approach presented uses a set partitioning model of the rostering problem, which is solved in a branch-and-price framework. Columns of the set partitioning problem are generated dynamically and branch-and-bound is used to enforce integrality. The column generating subproblem is modeled in three stages that utilize the inherent structure of roster-lines. Some important features of the implementation are described. The implementation builds on the generic model and hence the program can be setup for any problem that fits the model. The adaption to a new problem is simple, as it requires only the input of a new problem definition. The solution method is internally adjusted according to the new definition. In this report, we present two different practical problems along with corresponding solutions. The approach captures all features of each problem and is efficient enough to provide optimal solutions. The solution time is still too large for the method to be immediately applicable in practice, but we suggest a number of ways to improve the method further.

ISBN 978-87-90855-72-7

DTU Management Engineering
Department of Management Engineering
Technical University of Denmark

Produktionstorvet
Building 424
DK-2800 Kongens Lyngby
Denmark
Tel. +45 45 25 48 00
Fax +45 45 93 34 35

www.man.dtu.dk