Exact and heuristic approaches to nurse scheduling

Jonas Bæklund
Department of Mathematics
Aarhus University

PhD Thesis, July 2013

Adviser:
Andreas Klose, Aarhus University

Exact and heuristic approaches
to nurse scheduling

PhD thesis by Jonas Bæklund
Department of Mathematics, Aarhus University
Ny Munkegade 118, DK-8000 Aarhus C, Denmark
baeklund@imf.au.dk

Supervised by Andreas Klose

Submitted July 31, 2013

Preface

This dissertation presents the outcome of the research I have performed during the four years of my PhD programme at the Department of Mathematics at Aarhus University. The aim of the research was to develop suitable solution methods for a special Danish version of the nurse rostering problem. As proposed by my supervisor, a hybrid integer programming / constraint programming (IP/CP) method was first developed to handle the particular problem. The reasoning behind this decision was that hybrid IP/CP methods performed well on similar problems in the literature and that the CP is generally a flexible instrument allowing to address different types of (complex) constraints.

An early extended abstract, which presented the basic ideas of this method is published in the Proceedings of the 2nd International Conference on Applied Operational Research (ICAOR'2010) [9]. A paper describing the actual hybrid method and some experimental computational results has been submitted to the Annals of Operations Research; a revised version paper ("minor revision") is currently under revision [8]. Chapter 5 and 7 are based on this paper, but present the different methods in much deeper detail; also more in-depth computational experiments are presented there. The focus is on solving nurse rostering instances to optimality with a two week planning period.

After realising that the pure size of the considered real life rostering problem prevents the use of exact methods, two heuristic methods were developed. A variable neighbourhood search and a scatter search were designed, where both methods use the IP/CP method as a subroutine. A paper on these methods is on its way, and will later this year be submitted to an international journal.

The reader is assumed to be familiar with the basic concepts from linear and integer programming as well as algorithmic computer science; although, Section 2.1 gives a short introduction to integer programming and the standard methods for solving integer programming problems.

Parts of the outcome of the research presented in this dissertation have been presented at the following conferences:

- 23rd European Conference on Operational Research (EURO), Bonn, Germany, July 2009

- International Workshop on Scheduling in Healthcare Systems (SCHEAL10), which was a part of the 2nd International Conference on Applied Operational Research (ICAOR'2010), Turku, Finland, August 2010
- 4th Nordic Optimization Symposium, Aarhus, Denmark, October 2010
- INFORMS Healthcare 2011, Montreal, Canada, June 2011

Acknowledgements

First of all, I would like to thank my supervisor Andreas Klose for the countless meetings and discussions which have helped keeping this project on track. I would also like to thank him for suggestions and corrections for both papers, abstracts, presentations as well as this dissertation with respect to both academic and linguistic matters.

During the spring of 2011, I had the pleasure of visiting Tallys Yunes at the University of Miami for six months. I am thankful for all the discussions and guidance from a person with a comprehensive and superior knowledge of especially constraint programming. I would also like to thank the Department of Management Science at the University of Miami and its chair Edward K. Baker for supporting my attendance to the INFORMS Healthcare conference in Canada.

I thank the Department of Mathematics for sponsoring a number of conferences. These conferences have made it possible for me to share my research and have rewarding discussions with researchers from outside of Aarhus University.

I would also like to thank my friend and coworker Tue Rauff Lind Christensen, with whom I have had countless discussions during all my years of university studies.

A thank should also go to my friends and family, who have supported me during all my studies. Without them this work would never have been possible. A special thank should go to my girlfriend, Milda Plauskaite, for her love and support, especially during the final work on this thesis.

Contents

Preface i

English summary vii

Dansk sammenfatning ix

1 Introduction 1

I Foundations

2 Solution Methods 5
- 2.1 Integer programming 5
  - 2.1.1 Branch-and-Bound 6
  - 2.1.2 Column generation 6
  - 2.1.3 Branch and Price 9
- 2.2 Constraint programming 11
  - 2.2.1 Variables 12
  - 2.2.2 Constraints 12
  - 2.2.3 Search order and branching strategy 14
  - 2.2.4 Adding an objective 15
- 2.3 Combining IP and CP 15
- 2.4 Heuristics 18
  - 2.4.1 Variable Neighbourhood Search 20
  - 2.4.2 Scatter Search 21

3 Nurse rostering 25
- 3.1 Usual models 26
  - 3.1.1 Direct model 26
  - 3.1.2 Set-covering type model 27
- 3.2 Standard constraints 28
- 3.3 "Optimal" schedule 29
- 3.4 Solutions methods from the literature 30

4 The Danish model 33
- 4.1 The confronted problem 33
- 4.2 Distinguishing features 35
- 4.3 Objectives 36

II Exact approaches

5 IP/CP model **41**

5.1 The solution method 41

5.2 The master problem 41

5.2.1 Restricted master problem 43

5.2.2 Lower bound for the master problem 44

5.2.3 Branching in the master problem 44

5.2.4 Exploring the branch-and-bound tree 45

5.3 The pricing sub-problem 49

5.3.1 The CP model 50

5.3.2 Minimal time between shifts (C_1) 52

5.3.3 Day off / free day (C_9) 58

5.3.4 Minimum consecutive workdays (C_2) 59

5.3.5 Maximum consecutive workdays (C_{10}) 61

5.3.6 Shift type limits (C_4) 63

5.3.7 Office days (C_5) 66

5.3.8 Maximum work hours per week (C_3) 68

5.3.9 Recorded work hours (C_7) 71

5.3.10 Complete weekends (C_{11}) 73

5.3.11 Constraints on weekends (C_6, C_{11} and C_{12}) . 77

5.3.12 Combining maximum work hours per week and minimum recorded work hours 78

5.3.13 Pricing the new column 82

5.3.14 Using lower bound to reduce domain 85

5.3.15 Search in the CP model 88

6 IP model **91**

6.1 Pure IP model 91

6.1.1 Connecting constraints 91

6.1.2 Constraints for each nurse 93

6.1.3 The objective 98

6.2 IP/IP model 100

7 Computational results **101**

7.1 IP/CP vs IP vs IP/IP 101

7.2 Comparison of CP and IP 105

7.3 Flexibility of the IP/CP method 107

III Heuristic approaches

8 Variable Neighbourhood Search **111**
8.1 Greedy heuristic . . . 112
8.1.1 Cost calculations . . . 112
8.1.2 Search in the CP model . . . 113
8.2 Infeasibility iterations . . . 113
8.3 Normal iteration . . . 115
8.4 Overview of the method . . . 116

9 Scatter Search **119**
9.1 The diversification method . . . 119
9.2 The improvement method . . . 122
9.3 The reference set . . . 122
9.3.1 Initialising the reference set . . . 123
9.3.2 Updating the reference set . . . 124
9.4 The subset generation method . . . 124
9.5 The solution combination method . . . 124
9.6 Overview of the method . . . 125

10 Computational results **127**

Bibliography **133**

English summary

This thesis addresses the nurse scheduling problem of finding a duty roster for a set of nurses such that the rosters comply with work regulations and meet the management's requests. This problem is usually referred to as nurse rostering. For today's hospitals, personal in general but in particular nurses are becoming more and more a scarce and expensive resource. Accordingly, a careful nurse scheduling that at the same time matches economic needs as well as nurses' preferences is of increasing importance.

The nurse rostering problem is a complex combinatorial optimisation problem and generally very difficult so solve. No general model describes all nurse rostering problems, because there are so many differences between the specific problems encountered at the different wards around the world. The problem confronted in this thesis is from a ward at Aarhus University Hospital Skejby in Denmark. However, the solution methods proposed are general enough, so the methods would probably generalise to other wards in Denmark.

A branch-and-price method for solving the problem exactly is proposed. The master problem is to assign schedules to the nurses, and its linear relaxation is solved by means of column generation. The pricing sub-problem is to generate feasible schedules for the nurses and is solved by constraint programming. A number of specific algorithms for handling the constraints in the sub-problems are proposed. Computational tests show that optimal solutions can be found for instances with a two weeks planning period in a reasonable amount of computing time.

For instances with a longer planning period, two heuristics are proposed. The methods are a variable neighbourhood search and a scatter search. Both methods use the exact branch-and-price method as a sub-routine for performing the search in the heuristics. The experimental results show that the scatter search outperforms the variable neighbourhood search when more than an hour of computation time is allocated. The scatter search seems to find solutions of high quality, and it generally returns a set of high quality solutions. A set of high quality solutions – instead of just a single solution – is from a practical point of view a valuable feature.

Dansk sammenfatning

Denne afhandling omhandler metoder til at udføre vagtplanlægning for sygeplejersker. Vagtplanlægningen går ud på at finde en vagtplan for sygeplejerskerne på en afdeling, således at vagtplanen overholder overenskomster, regulativer og opfylder ledelsens ønsker.

For nutidens hospitaler bliver personale i almindelighed, men især sygeplejersker, en mere og mere begrænset og kostbar ressource. Af denne grund er en omhyggelig vagtplanlægning for sygeplejersker, som samtidig matcher de økonomiske krav samt sygeplejerskernes præferencer, af stigende betydning.

Vagtplanlægning for sygeplejersker er et komplekst kombinatorisk optimeringsproblem, der generelt er meget svært at løse. Grundet de så vidt forskellige betingelser på de forskellige afdelinger rundt omkring i verden, findes der ikke nogen generel matematisk model for vagtplanlægning for sygeplejersker. Det specifikke problem som bliver behandlet i denne afhandling stammer fra en afdeling på Århus Universitetshospital Skejby. Løsningsmetoderne som bliver foreslået er dog så generelle at de sandsynligvis kan bruges på andre afdelinger på sygehuse i Danmark.

Der foreslås en *branch-and-price* metode til at løse problemet eksakt. *Master* problemet er at tildele tidsplaner til sygeplejerskerne, og dens lineære relaksation er løst ved hjælp af søjle generering. *Sub*-problemet, som er at generere mulige tidsplaner for sygeplejerskerne foreslås løst ved hjælp af *constraint programming*. En række specifikke algoritmer til håndtering af betingelserne i *sub*-problemerne er beskrevet. De beregningsmæssige resultater viser at den optimale løsning kan findes, indenfor rimelig tid, for problemer med en planlægningsperiode på to uger.

To heuristikker er foreslået for problemer med en længere planlægningsperiode. Heuristikkerne er en *variable neighbourhood search* og en *scatter search*. Begge metode bruger den foreslåede *branch-and-price* metode som en del-rutine. De beregningsmæssige resultater viser at *scatter search* er bedre end *variable neighbourhood search*, når de får tildelt mere end én times beregningstid. *Scatter search* metoden virker til at finde løsninger af høj kvalitet, og desuden returnerer den typisk en mængde af løsninger med høj kvalitet. I praksis er det en nyttig egenskab at ikke bare én løsning af høj kvalitet findes men at adskillige af sådanne bliver fundet.

Chapter 1

Introduction

Combinatorial optimisation problems have been the subject of an enormous amount of research in the literature of the last 6 decades. The problems considered in the literature are often simplified models of real world applications, and mostly these models include integer variables. The interest of researchers in these types of optimisation problems can be explained by, first, the variety of applications and, second, the fact that they are generally very hard to solve.

Two different communities have in particular conducted research on appropriate models and solutions methods for these problems. The Operations Research community mainly used integer programming techniques, in particular branch-and-bound and branch-and-cut methods. In the Computer Science community, however, the most dominating technique for complex combinatorial optimisation problems has been constraint programming. These two methods have been developed pretty independently for many years, but during the last decade several attempts on combining the two techniques for different applications have been proposed. One of the main pioneers of the field of combining these methods is John Hooker from Carnegie Mellon University.

For today's hospitals, personal in general but in particular nurses are becoming more and more a scarce and expensive resource. Accordingly, a careful nurse scheduling that at the same time matches economic needs as well as nurses' preferences is of increasing importance. Nurse scheduling has received considerable attention in the research community during the last 45 years. It is a highly complex planning problem and as such usually decomposed into the interrelated sub-problems of *nurse staffing*, *nurse rostering*, and *nurse rerostering* [51]. *Nurse staffing* is the long term planning that determines the required number of nurses of different qualification levels. *Nurse rostering* can be described as the task of finding a duty roster for a set of nurses such that the rosters comply with work regulations and meet the management's requests. The day to day planning where nurses that call in sick and other day to day changes have to be considered,

is referred to as *nurse rerostering.* The objective of rerostering is to create a duty roster for all nurses such that all required shifts are still covered and the new rosters are as close to the preplanned rosters as possible. The topic of this thesis is nurse rostering.

In Chapter 2, a short introduction to the standard solution methods of combinatorial optimisation is given. The basic concepts of integer programming (IP), constraint programming (CP) and hybrid IP/CP approaches are presented. The basic ideas of heuristics is also described with a focus on variable neighbourhood and scatter search. The presented approaches are those used in this thesis for solving the nurse rostering problem.

Chapter 3 describes the general nurse rostering problem and the methods which have been proposed in the literature for solving it. A distinction is made between modelling the problem as a “direct” integer model and a set covering type model.

The Danish model and how it differs from the typical nurse rostering problem is presented in Chapter 4. The described problem is the one confronted in this thesis.

Chapter 5 is the main chapter of this thesis and presents a hybrid IP/CP method. The method combines IP and CP in a branch-and-price context where the sub-problems are solved with CP.

In Chapter 6, an integer programming model for the complete nurse rostering problem is proposed. Besides this model, a branch-and-price model where the sub-problems are solved with an IP solver is also presented. Both solution methods are basically used for the purposes of a comparison with the IP/CP method described in chapter 5. Computational results of the methods are given in Chapter 7. The tests were executed on two week instances, as this was the longest planning period for which any of the methods could solve all instances to optimality.

Chapter 8 and 9 describe two heuristics for solving instances with a longer planning period. A variable neighbourhood search is presented in Chapter 8 and in Chapter 9 a scatter search is described. Both methods use the IP/CP as a basis for their heuristic search. The results of a comparison between the two heuristic methods are given in Chapter 10.

Part I

Foundations

Chapter 2

Solution Methods

Four of the basic methods used for solving combinatorial optimisation problems are introduced in this chapter. The introductions focus on what is relevant for the research described in this thesis. The methods introduced are integer programming (IP), constraint programming (CP), a combined IP/CP method and some relevant heuristics.

2.1 Integer programming

In the operational research community the standard way of modelling and solving combinatorial optimisation problems is integer programming. In the literature, IP generally refers to integer linear programming. A given linear objective function has to be optimised, given that the variables obey a set of linear constraints and that the variables need to attain integer values. Let c_j, a_{ij} and b_i be rational constants, then a general IP problem can be stated as:

$$
\begin{array}{lll}
\min & \sum_{j \in X} c_j x_j & \\
s.t. & \sum_{j \in X} a_{ij} x_j = b_i & \forall i = 1, \ldots m \\
 & 0 \leq x_j & \forall j \in X \\
 & x_j \in \mathbb{Z} & \forall j \in X
\end{array}
$$

The last constraint set is referred to as the integer requirement, and when creating a linear relaxation of an IP problem, this is the constraint set which is removed.

In the next section, the branch-and-bound method for solving general integer programming problems is explained. In Section 2.1.2, a method for solving a linear programming problem is described, this method is combined with the branch-and-bound method in Section 2.1.3.

2.1.1 Branch-and-Bound

In the operational research community, the branch-and-bound framework is the standard way of solving integer programming problems. The method was originally proposed by Land and Doig [36] and it is today still one of the most effective methods for solving general IP problems.

The branch-and-bound method for minimisation problems creates a search tree where each node represent an associated IP problem. The root node of the tree is associated with the complete IP problem. In each search tree node, the method solves the linear relaxation of the IP problem associated with the node; if the solution is not integer, the IP problem is divided into at least two sub-problems. The division is made in such a way that the union of the solution spaces of the sub-problems includes all feasible solutions of the parent IP problem, but excludes the parent's optimal non-integer solution. Reapplying this principle again and again creates the IP problems which are associated with the nodes of the search tree. Whenever an integer solution is found, it gives an upper bound on the solution value (when minimising). The optimal objective value of the problem's linear relaxation of an IP problem yields a lower bound on the optimal value of a solution in the sub-tree rooted at the associated node. So, if a node has a lower bound not smaller than the global upper bound, the node can be pruned, as the sub-tree cannot yield an improved solution.

The standard way of dividing the IP problems is to choose a variable x_i attaining a fractional value in the optimal solution of the linear relaxation. Let its value in the optimal solution be $\tilde{x}$. Two child nodes are created; the first node is created by adding the constraint $x_i \leq \lfloor \tilde{x} \rfloor$ to the IP problem of the parent; the other node is obtained by adding: $x_i \geq \lceil \tilde{x} \rceil$.

The order in which the nodes of the tree are solved is controlled through a *node selection strategy*; and how to divide the IP problem is controlled by a *branching strategy*. For more on the topic of node selections strategies see Linderoth and Savelsbergh [39]; for more on branching strategies see Achterberg et al. [1]; and for for more on the general IP approach see Nemhauser and Wolsey [46].

2.1.2 Column generation

Column generation is a method for solving linear programming problems with a huge number of variables or where a reformulation of the original IP problem results in a large number of variables.

The general idea of column generation is to divide the problem into a master problem – consisting of a part of the original constraints – and a sub-problem for generating feasible columns. Column generation is sometimes in the literature denoted as delayed column generation, and it was first described by Dantzig and Wolfe [13] together with the Dantzig-Wolfe decomposition method. The Dantzig-Wolfe decomposition method reformulates a problem into one that can be solved by column generation.

Let a linear programming problem be given in the following form:

$$\begin{aligned} \min \quad & \sum_{j \in X} c_j x_j \\ s.t. \quad & \sum_{j \in X} a_{ij} x_j = b_i && \forall i = 1, \dots m \\ & 0 \leq x_j && \forall j \in X \end{aligned}$$

where X is a very large set. If this is solved with the simplex algorithm in tableau form, the following table would be stored at each iteration:

	x_B	x_N	RHS	
	0	$c_N - c_B B^{-1} N$	$c_B B^{-1} b$	Objective
x_B:	I	$B^{-1} N$	$B^{-1} b$	Constraints

where B is the columns of the basic variables and c_B is the cost of the basic variables with a corresponding set of variables x_B. The matrix N comprises the columns N_j of the non-basic variables with costs c_N and variables x_N. When using the simplex method, a huge amount of information is saved and recalculated at each iteration. Due to the immense size of the set X compared to x_B, the major part of the information belongs to the non-basic columns. In each iteration of the simplex method, a column of negative reduced cost $(c_j - c_B B^{-1} N_j)$ is found and the corresponding variable is made basic.

In each iteration only one column of the matrix $B^{-1}N$ is used, and due to the immense size of N, an enormous amount of time is used to recalculate this matrix.

With the column generation method, the matrix $B^{-1}N$ and the vector $c_N - c_B B^{-1} N$ are hidden in a sub-problem that generates columns of negative reduced costs or proves that no such one exists. Let H be the set of all columns of the problem. The sub-problem

can be stated as the following optimisation problem:

$$\begin{aligned} \min \quad & c_w - c_B B^{-1} w \\ s.t. \quad & w \in H \end{aligned}$$

where c_w is the cost of column w. Remark that $c_B B^{-1}$ is the row vector of dual variables of the current basic solution. For the column generation method to be relevant the set H should not be given explicitly, but implicitly as the set of all solutions to a set of further constraints. Very often, this constraint set obeys a special structure that allows to apply specialised and efficient algorithms for solving the sub-problem.

The master problem is often called the restricted master problem, since it only contains a subset of the complete set of columns. When the sub-problem concludes that no column with negative reduced cost exists, the last solution to the restricted master problem is the optimal solution to the complete problem.

Column generation searches for the optimal solution in almost the same way as the usual primal simplex method; the only change is that the non-basic columns are not kept explicitly but generated when needed.

The set of columns (H) for the problem can often be generated from a set of independent sub-problems. In an iteration where any of the sub-problems finds a negative reduced cost column, at least one such column should be added to the restricted master problem. This is a necessary condition for the column generation to solve the problem to optimality. When a negative reduced cost column (not necessary optimal) for one of sub-problems has been found, it is not necessary to solve any of the sub-problems to optimality. Often it is beneficial to generate more than one column and to generate those with the lowest reduced cost. Any subset or all of the columns generated can be added to the restricted master problem.

If a lower bound for the LP problem is required, it is necessary to calculate a lower bound on the objective of all sub-problems. If a sub-problem is solved to optimality, then the optimal value is the best lower bound for that sub-problem. The lower bound is then the sum of the objective value of the restricted master problem and all lower bounds on the objectives of the sub-problems.

The efficiency of the column generation highly depends on the availability of efficient procedures for solving the sub-problems. Figure 2.1 illustrates the column generation principle when used as a part of a branch-and-price method (The column generation is the part inside the dashed line).

2.1.3 Branch and Price

Branch and price is a generalisation of branch and bound, where the linear programming relaxations are solved using column generation. Problems where the branch and price method could be efficient are problems with a huge number of variables or where a reformulation of the problem results in a model with a huge number of variables.

The branch and price method is illustrated in Figure 2.1. The area inside the dashed line is the column generation; UB is the upper bound for the complete integer problem; it always equals the solution value of the best integer solution found so far. LB always refers to a node and is the lower bound on the solution value of all integer solutions that could be found in the sub-tree rooted at the corresponding node. As it is not necessary to solve the relaxed problems to optimality, an option to do early branching is included in Figure 2.1.

When choosing the branching strategy for a branch an price algorithm, the two main concerns are to obtain a balanced search tree and to keep the special structure of sub-problems. The special structure of the column generation sub-problems is required, as the efficiency of the column generation highly depends on having efficient solvers for the sub-problems available. The standard way of branching in a branch and bound method would usually destroy the special structure of the sub-problem and create an unbalanced search tree as well. The unbalanced search tree arises, because forcing a column not to be in a solution generally shows small impact as there are a huge number of different columns, whereas forcing a column to be in the solution contributes to a substantial problem reduction and thereby usually strongly affects the lower bound.

Instead, branching in a branch and price method is often based on branching on constraints or "indirect" variables. Branching on a constraint means to find a constraint for a sub-problem, whose left-hand side should be be integral in an integer solution to the complete problem, but is fractional in the current fractional solution. The branching is created by enforcing this constraint to be greater or equal to the current value rounded up in the first child, and to be not larger than the rounded down value in the other child. The set of constraints considered should be large enough to ensure integrality of the variables that are required to be integer. The constraints should also be selected such that the special structure of the sub-problems is kept.

An "indirect" variable is a variable that is not visible in the model, but can be calculated from a given solution. When solving a problem resulting from a reformulation with Dantzig-Wolfe decomposition,

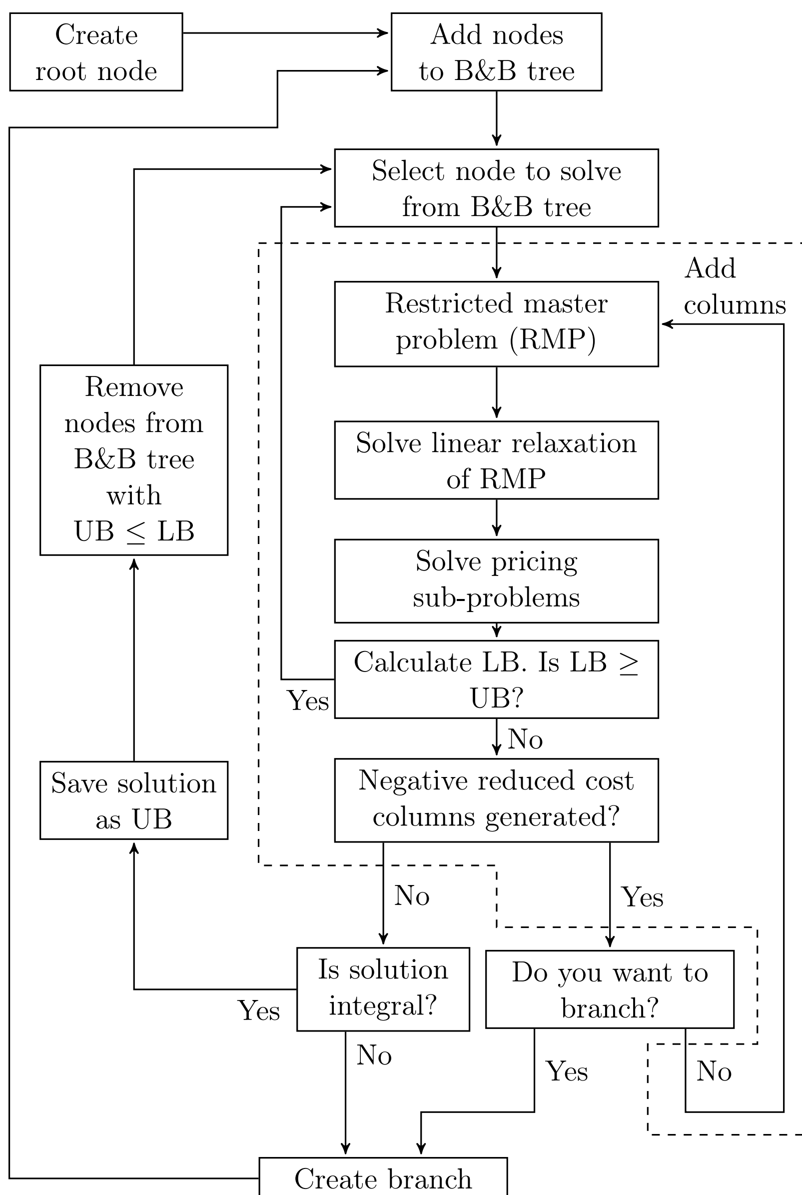

Figure 2.1: The branch and price method. The area inside the dashed line is the column generation part of the method. UB is the global upper bound and LB is the lower bound for the given node.

the “indirect” variables could correspond to the original variables of the model before the reformulation. An “indirect” variable can also be a variable of a sub-problem.

Some branching strategies are enforced through extra constraints in the master problem, others through the sub-problems.

A more soft type of branching can be performed if some left-hand sides of the constraints of the restricted master problem are fractional (if the constraints only include integer constants). By bounding the constraint, a branch can be created. This would not change the sub-problems, but in general it is not enough to ensure integrality.

For more on the general branch-price method see Lübbecke and Desrosiers [40], Barnhart et al. [2] and Vanderbeck [50]. For practical applications see Table 1 in Lübbecke and Desrosiers [40] for a list of papers.

2.2 Constraint programming

Constraint programming (CP) is in its basic form a method for finding a feasible solution for a set of bounded integer variables, given a set of constraints. The constraints can be mathematical equations, logical propositions or implicitly given as an algorithm that concludes if a solution is feasible or not. Even an enumeration of the feasible combinations of values for a subset of the variables can be used as a constraint.

In this thesis the distinction that is sometimes performed between constraint satisfaction, constraint logical programming and constraint programming is not kept. Problem solving originating from any of these areas are placed under the umbrella of constraint programming.

The general idea of CP is to build a search tree, and use logical implications of the constraints to reduce the size of the tree.

In each node of the search tree, the search keeps track of the variables’ domains. Logical implications of the constraints are used to reduce the domains of the variables.

The standard way of creating the search tree is first to reduce the domains of the variables by logical implications. If not all variables’ domains have been reduced to singletons, a branching is performed by creating at least two sub-problems. In each sub-problem a domain reduction of at least one domain is performed. The union of the solution spaces of the sub-problems should be equal to the solution space of the current node. The most common way of branching is to assign a value to a variable in one sub-problem and to remove this value from the variable’s domain in the other sub-problem.

This method for creating the search tree is reapplied in all nodes until either all variables have their domain reduced to singletons or a variable has its domain reduced to the empty set. If all variables have a domain size of one, a feasible solution has been found. If a variable has a domain size of zero, then there does not exist a feasible solution in the current node or in the tree rooted at this node. When this happens, the node is pruned and the search backtracks to another node that has not been solved previously.

For a thorough introduction into CP, see Hentenryck [27] and Marriott and Stuckey [42]. A major contribution to the foundation of CP are due to Jaffar and Lassez [32]. An early application of CP is found in Dincbas et al. [16], and Hentenryck and Saraswat [26] give an introduction to the development of CP and its use in different areas.

2.2.1 Variables

The most common variables used for CP are binary and finite integer variables. In the solver the values that a variable can attain are often stored in an array of possible values. If the variable can attain a huge number of different values, its domain might be stored with just an upper and lower bound. This could reduce the logical implications that can be inferred from the variable, but it reduces the memory required to represent the domain.

Continuous variables can be included in a CP method, but not all constraints can be used for such variables. The most common method is to store the domain of such a variable as an interval. The representation of domains as intervals reduces the number of different constraints and logical implication methods that can be applied for such a variable. The standard branching strategy for such a variable would be to split the domain on the middle.

2.2.2 Constraints

Constraints in a CP-model can be stated in many ways, but to actual use them, it is necessary to create a method for handling them. The method should preferable create some logical implications from the constraint which will reduce the domain of some variables, if this is not possible it should at least be able to conclude if the constraint is fulfilled when all variables have their domains reduced to singletons.

Logical implications, often called domain reduction mechanisms or constraint propagation mechanisms in the CP literature, are the most important aspect when implementing a CP model. If no domain

reductions are performed, a CP search would create a full enumeration of the variables. A small example of how constraint propagation mechanisms can be designed is given below and an example of how different constraint propagation mechanisms can perform with a given constraint and how this influences the solution time is given in Section 5.3.2. When deciding between different domain propagation mechanisms, it is the problem of choosing the right balance between using time for the search or using time for finding domain reductions. But it seems that the effort spend on using more advanced domain reduction mechanisms often pays off, even if the mechanism requires some computation time.

As an example of how domain propagation mechanisms can be designed, lets look at the constraint:

$$B = x_1 + x_2, \tag{2.1}$$

where B, x_1 and x_2 are integer variables. For notation purposes let $D(\cdot)$ define the domain of a given variable, let max and min of a variable be respectively the largest and smallest value in its domain. Let the domain of the variables be:

$$D(B) = \{0, 1, 2, 3, 4\}, \quad D(x_1) = \{0, 2, 4\}, \quad D(x_2) = \{2, 4\}. \tag{2.2}$$

The classical domain propagation mechanisms for a constraint like this is to restrict the domain to obey the following bounds:

$$B \leq \max x_1 + \max x_2, \tag{2.3}$$
$$B \geq \min x_1 + \min x_2, \tag{2.4}$$
$$x_1 \leq \max\{B - x_2\} = \max B - \min x_2, \tag{2.5}$$
$$x_1 \geq \min\{B - x_2\} = \min B - \max x_2, \tag{2.6}$$
$$x_2 \leq \max\{B - x_1\} = \max B - \min x_1, \tag{2.7}$$
$$x_2 \geq \min\{B - x_1\} = \min B - \max x_1. \tag{2.8}$$

The domain propagation mechanism given in equation (2.4) would in the example remove the values 0 and 1 from the domain of B and mechanism (2.5) would remove the value 4 from the domain of x_1. No other domain reductions can be made by any of the given mechanisms. The value of 3 from the domain of B is not removed, even though it is easily seen that it cannot be assigned to B in a feasible solution. More extensive propagation mechanisms could be designed that removed the value, but they would also require more calculations. Depending on the given problem to be solved this could be beneficial or not. The given mechanisms are the ones that are

usually implemented in modern constraint programming solvers, such as IBM ILOG CP solver, Choco, CHIP.

The importance of domain reductions of the variables stems from the fact, that if a value of a variable is removed, the whole sub-tree of branching on this variable would be removed. In the top of the tree this can have an enormous impact on the size of the tree, and thus the time to solve the problem. The following simple example underpins the importance of performing domain reductions as early as possible. Let $x_1, \ldots, x_n$ be a set of variables. Let a node in the search tree be given where all combinations of values in the domains of variable x_1 and x_n are infeasible. Let the domain propagation mechanism of the constraint creating this infeasibility not be able to detect this infeasibility until both variables have been fixed. The search starts by fixing x_1, then x_2 and so on, creating a big search tree. The infeasibility of the whole sub-tree is first recognised when all combinations of all values in the domains of the variables have been tried. If the domain reduction mechanism of the constraint was capable of figuring out this infeasibility in the start node, the whole sub-tree would newer have been created or searched. If several other variables were fixed previously to this sub-tree, similar sub-trees would probably be found at several other places in the search tree, requiring an enormous amount of search time in these infeasible sub-trees.

Every domain propagation mechanism may be checked more than once in each search tree node, as if one mechanism reduces a domain of the variable, then other mechanisms might be able to reduce the domains of some other variables. The general idea is to check all domain propagation mechanisms until no further domain reductions can be found. When applying constraint programming in practise, this introduces the questions of when a propagation mechanisms is necessary to be reapplied and in which order should they be applied.

2.2.3 Search order and branching strategy

The order in which the nodes are solved and the choice of the branching strategy is also very important. The most common way of choosing the order of the nodes is to use a depth first search strategy. The choice of a branching strategy can have a huge impact on the size of the tree. Which branching strategy works the best depends strongly on the specific problem. Examples of branching strategies are:

- Choose the first non-fixed variable and assign the smallest domain value to it.

– Choose the variable with the smallest domain larger than two, and assign one of the values to it.

– Split the domain of a variable into two parts, one for each subproblem.

– Assign values for the most constrained variables first.

2.2.4 Adding an objective

CP can also be used for optimisation problems. This is performed by introducing a variable representing the objective value. The following is a short description of how this variable should be treated in a minimisation problem.

At each node, a lower bound of the objective value of any feasible solution reachable from the current node is calculated. This lower bound is set as the minimal value for the variable representing the objective. If the domain of the variable representing the objective becomes empty, the node is pruned and the search backtracks. When a feasible solution is found, the objective value is calculated and this value is enforced as a global upper bound for the variable representing the objective in all nodes of the tree. This ensures that the next solution found is at least as good as the current best solution. If only one optimal solution is required an epsilon value is subtracted from the global upper bound to ensure that the next solution found is strictly better than the current best solution.

Handling an objective introduces a new important aspect, that is to choose how to calculate a lower bound and how much time to use finding better lower bounds.

Depending on the method used for calculating the lower bound, it might be possible to infer that a domain value of a variable cannot be assigned to it in an optimal solution. Such values should be removed from the domain, as the removal could lead to more domain reductions and thereby to a smaller search tree and perhaps to a better lower bound at the current node.

Standard domain propagation mechanisms only remove domain values that are infeasible in a solution whereas the above also removes values that could exist in a feasible solution but would result in a suboptimal solution.

2.3 Combining IP and CP

Integer programming models and optimisation methods for solving it have been developed in the operational research area, whereas con-

straint programming and the underlying logical inference techniques emerged in the artificial intelligence community. One of the pioneers in combining these two areas is John Hooker [28].

In the operational research community, the focus has been on developing better formulations, in particular by means of cutting planes methods for tightening the linear relaxation. The artificial intelligence community, by the absence of an objective, focused on developing logical inference methods. What Hooker, among others, suggests, is to unify the two methods into a combined improved method.

The standard branch-and-bound framework and the standard framework for constraint programming are quite fixed, so over the years many different generalisations have been proposed. For the branch-and-bound framework, the best known generalisations are the branch-and-cut, branch-and-price and branch-cut-and-price frameworks. For the constraint satisfaction framework, generalisation allowing the inclusion of an objective and continuous variables have been proposed, among others. Both solution methods can be generalised to the same general branch-and-bound framework. A linear integer programming problem is introduced for describing this generalised method. Let b and c be one-dimensional vectors of constants; A is a matrix of constants and x the variable vector of the problem:

$$\begin{array}{ll} \min & f(x) \\ s.t. & x \in \mathcal{X}, \end{array}$$

where

$$\begin{array}{ll} f(x) & = c^T x, \\ \mathcal{X} & = \{x \mid Ax \leq b \wedge x \geq 0 \wedge x \in \mathbb{Z}^n\}. \end{array}$$

In the generalised framework, each node i of the branch-and-bound tree is associated with a problem $(\mathcal{X}_R^i)$:
The framework is initialised in the following way:

- Set the global upper bound equal to infinity (UB $= \infty$).
- Initialise the problem associated with the root node to be equal to a relaxation of the original problem ($\mathcal{X}_R^{\text{root}} = \text{relax}(\mathcal{X})$).

For each node i of the search tree, the following steps are performed:

1. If wanted, calculate $z_R^* = \min\{f(x) \mid x \in \text{relax}(\mathcal{X}_R^i \cap \mathcal{X})\}$, and let x_R^* be the corresponding solution, if it is found.
2. If a feasible solution for the original problem is found this way, update the upper bound. (If $x_R^* \in \mathcal{X} :$ UB $:= \min(z_R^*, \text{UB})$).

3. If desired, calculate a lower bound on the objective of the associated problem. (If z_R^* is calculated, it could be used).
   - If the lower bound (LB) is not lower than the global upper bound, prune the node (LB $\geq$ UB $\Rightarrow$ prune current node).
4. If the node is not pruned: create sub problems $\mathcal{X}_R^1, \ldots, \mathcal{X}_R^k$ associated with the child nodes, such that:
   $\mathcal{X}_R \cap \mathcal{X} \subseteq \bigcup_{i=1}^k \mathcal{X}_R^i$.
5. Repeat from step 1 until all nodes have been searched.

The relaxations used for the initialisation of the root node and in step 1 of the framework are not required to be the same.

This generalised framework incorporates both the branch-cut-and-price framework as well as the constraint programming framework, where an objective is included. The cut-and-price and domain propagation mechanisms are hidden in the selection of how to relax $\mathcal{X}_R \cap \mathcal{X}$ in step one of the framework. So depending on which cutting planes method are applied, it creates different relaxations in step 1. The cuts in the IP method correspond to the domain propagation mechanisms of the CP framework. When domain propagation is performed due to the objective, it corresponds more directly to the logical fixing of variables in an IP setting. The requirements of how to do the branching are not specified in detail. Also more details have to be included to ensure that the framework will terminate.

The relaxation used in step one of the framework is for the standard IP method the linear relaxation, whereas for the CP model the relaxation used is based on the domains of the variables. The domains of the variables in a CP model is the set of all possible values they attain inside the feasible region. The relaxation is all combinations, where each variable attains values from their particular domain. Tighter relaxations can be used for both methods, and are usually a result from some inference methods (cuts or domain propagation).

The lower bound in an IP setting is usually the optimal value of the linear relaxation whereas for a CP it is usually calculated as $\sum_{i=1}^n \min(c_i \min(x_i), c_i \max(x_i))$, where $\min(x_i)$ and $\max(x_i)$ are respectively the minimal and maximum value in the domains of the variable. The differences between the two methods are summarised in Table 2.1.

As the above generalised framework shows, the methods used for IP and CP problems are not that different. The above framework may be usual for finding way of how to combine the two methods. The domain propagation of CP could be used to create some bounds

Table 2.1: Comparing the methods for integer and constraint programming.

	IP	CP
Inference	Linear programming Cutting-planes	Domain filtering Constraint propagation
Search	Branch-and-Bound (Removes x^*_R)¶	Branch-and-Bound (Split a domain)¶
Bounds on the objective	UB by best solution LB by $c^T x^*_R$	UB by best solution LB might be calculated†
Variables	$\mathbb{Z}^+$	Finite domain
Constraints	Linear inequalities and equations	Arithmetic constraints Global constraints

¶ Usual implementation
† If linear: $\sum_{i=1}^{n} \min(c_i \min(x_i), c_i \max(x_i))$

for the linear relaxation, or a linear relaxation of a CP problem could be created to calculate a lower bound.

Another idea for combining the methods is to use the CP method for solving the sub-problems in a branch-and-price framework. This method is used later in this thesis for solving the nurse rostering problem. This idea has earlier been used for the airline crew assignment problem [49, 18]; the employee timetabling problem [14]; and urban transit crew management problems [53].

2.4 Heuristics

For many, in particular NP-hard optimisation problems an exact method is not feasible, as it will require too long computation time or memory usage. When such problems are encountered, a common method is to apply some kind of heuristic. A heuristic is an algorithm that tries to find a solution close to the optimal one.

One of the most basic heuristics for combinatorial optimisation problems is local search. Let X be the set of feasible solutions and $f(\cdot)$ be the objective function to a combinatorial optimisation problem. The neighbourhood of a solution $x \in X$ is a set of solutions from X typically generated by applying small modifications to x.

The local search algorithm is an iterative search procedure moving from a starting solution x to a solution in the neighbourhood of x showing a better objective value. This neighbouring solution is then

used as the starting solution for the next iteration. The algorithm stops, when no solution with a better objective value exists in the neighbourhood of the current solution. The final solution's objective value is thus local optimum with respect to the neighbourhood. Two common methods for selecting the next solution are either to select the first solution improving the current objective value (first-accept) or to select the solution with the best objective value in the neighbourhood (best-accept). Pseudo code for a local best accept search algorithm is presented as Algorithm 1 below.

Local search and a multi start variant hereof have been discussed as early as in Lin [38] and earlier.

Algorithm 1 Local search

```
procedure LOCAL-SEARCH(x , Neighbourhood)
    while True do
        let X^N := Neighbourhood(x)
        let z := x
        for all y ∈ X^N do
            if f(y) < f(z) then
                z := y
            end if
        end for
        if f(z) < f(x) then
            x := z
        else
            Return x
        end if
    end while
end procedure
```

A severe disadvantage of local search is that it often gets caught in a local optimum far away from the optimal solution. Several methods have been proposed in the literature to overcome this weakness, for instance; adjusting the size of the neighbourhood, to repeat the algorithm with a different initial solution each time a local optimum has been found, or to permute the local optimum and then to use the permutation's outcome as a new initial solution. An overview of more advanced heuristics methods, that in different ways trie to avoid getting stuck in a local optimum can be found in the collection Gendreau and Potvin [19].

2.4.1 Variable Neighbourhood Search

Variable neighbourhood search is another method to avoid that a local search gets stuck early in a local optimum. The idea is to select the initial solution for a local search from a set of different neighbourhoods. When the local search finishes in a local optimum, a new initial solution is selected from a set of different neighbourhoods, until some stopping criteria is met. The neighbourhoods can be selected either in sequence, at random or with any other method. To select the initial solution from a set of different neighbourhoods is a kind of a shaking mechanism. The shake is performed to get away from the current local optimum. The local search uses the same neighbourhood for its search, and it could use the same as one of the neighbourhoods used for selecting the initial solution.

A version of the variable neighbourhood search with sequential selection of neighbourhoods is presented in Algorithm 2. This version is denoted as *basic variable neighbourhood search*(BVNS) in Hansen et al. [23].

Algorithm 2 Variable Neighbourhood Search

```
procedure VARIABLE-NEIGHBOURHOOD-SEARCH(x)
    Initialise k := 0
    while k ≤ k_Max do
        Choose y ∈ Neighbourhood_k(x)
        let z := SEARCH-PROCEDURE(y,Neighbourhood)
        if f(z) < f(x) then
            let x := z and k := 0
        else
            k := k + 1
        end if
    end while
    Return x
end procedure
```

See Hansen and Mladenović [24], Hansen and Mladenović [25] or Hansen et al. [23] for a discussion of the variable neighbourhood search technique and how to use the neighbourhoods in the search. For some applications of the VNS technique see Carrabs et al. [10] or Ribeiro and Souza [48].

The Search-procedure in Algorithm 2 could be the local-search from Algorithm 1 or just a single iteration of the local-search or any other search algorithm that in some way searches the given neighbourhood.

Usually the algorithm will be stopped when it reaches a given number of iterations without improvement or a time limit.

2.4.2 Scatter Search

Whereas local and variable neighbourhood search relies on iteratively improving a single solution, scatter search operates on a population of solutions.

Similar to genetic algorithms, scatter search combines solutions into new ones which then replaces other solutions from the population.

The version of scatter search presented here is first described in Glover [20] and more thoroughly in Martí et al. [43], and it can be described with the following methods:

- A diversification method.
- An improvement method.
- A reference set update method.
- A subset generation method.
- A solution combination method.

The *diversification method* has as input a set of solutions for the given problem and should generate at least one solution that is different from the given solutions. The aim of the method is to generate solutions in all areas of the search space. Often the method relies on randomness or some kind of memory to ensure that solutions are different from those previously generated.

The *improvement method* should as the name suggests, improve a solution. Most often, some type of local search is applied to this end. If the given solution is infeasible, the method should be able to search for feasibility besides searching for a better solution value. If no improved solution can be found, the initial solution is returned.

Updating of the population (which in the scatter search is called the reference set) is performed with the *reference set update method*. The input to this method depends on a choice of when to update the reference set. With dynamic update, the method is called each time a new solution is generated, whereas if the static update option is chosen, a set of new solutions is created before this method is called. The method should from the given input select the new reference set. Several criteria can be useful for selecting the new set, and the solution value is always an important criterion. Other criteria used can be that the solutions should be diverse. Typically the reference set is chosen to be of a fixed size.

The *subset generation method* should generate subsets of the reference set, which are used for generating new combined solutions in the *solution combination method*. The subsets generated can be chosen in a various of ways, but it is important to avoid duplicates.

Given a set of solutions, the *solution combination method* should create at least one new solution. The idea of this method is to select good parts from the given solutions and to combine them into a better solution. The method may rely on randomness and it may output more than one solution. Often it is the case that this method will return an infeasible solution; if feasibility cannot be guaranteed the *improvement method* should be able to start from an infeasible solution.

The design of the reference set update method and the subset generation method can be performed quite independent of the problem to be solved, whereas the other methods highly depend on the problem. For a more in depth discussion and rules of thumbs for designing the different methods see Laguna and Armentano [35].

The scatter search can be described by two phases; an initial phase that creates a reference set and a search phase that searches from the initial set.

Initial phase:

1. Create one or more initial solutions, from which all other solutions are created.
2. Generate solutions by the *diversification method*.
3. Improve the solutions with the *improvement method*.
4. Repeat from 2; until a given number of solutions have been generated.
5. Select the reference set from the solutions generated.

Search phase:

6. Generate subsets with the *subset generation method*
7. Generate solutions from one of the subsets with the *solution combination method*.
8. Use the *improvement method* on the solutions.
9. If static update repeat from 7, until no more new subsets.
10. Update reference set with the *reference update method*.
11. If the reference set has changed, repeat from 6. Otherwise from 7.

The algorithm finishes when there are no more new subsets to choose in step 7. Usually a time or iteration limit is also enforced.

It is possible to use the algorithm in an iterative fashion by using the generated reference set or a subset hereof as the initial set of solutions for step 1 in the initial phase. This could be repeated until either a time or iteration limit is reached.

Chapter 3

Nurse rostering

For today's hospitals, personal in general but in particular nurses are becoming more and more a scarce and expensive resource. Accordingly, a careful nurse rostering that at the same time matches economic needs as well as nurses' preferences is of increasing importance. The difficulties in planning these duty rosters arises from the fact that the demand for nurses varies during the day, and that the wards should be staffed around the clock. Nowadays also the nurses' preferences for shifts and special requests for days off should be taken into account when creating the schedules. The quality of the rosters has a great impact on the nurses' job satisfaction and well-being.

In general, nurse rostering is the task of finding a duty roster for a set of nurses such that the rosters fulfil work regulation and the management's requests.

The work regulations are often defined both in general union contracts and in local agreements between each single nurse and the hospital. Besides this, the nurses often have the right to request days off and which shift pattern they would like to work. So each nurse is different and should be treated in that way, when the rosters are created.

A duty roster consists of a set of shifts that the nurses should work. The possible working shift types typically include a *night*, *evening* and *day* shift, besides these several other shift types can be defined. A shift type is a hospital duty with a usually well defined start and end time. Almost all problems considered in the literature include some sort of coverage constraints, which is a requirement of having at least (or exactly) a given number of nurses working a specific shift or during a given time period. Sometimes these constraints are defined for specific qualification levels of the nurses. These constraints may also stipulate both a required as well as a desired number of nurses to work a given shift. A lot of other types of management requests are also often included.

The duty roster that is generated should fulfil these requirements and in some way be optimal. The length of the requested duty roster is normally between 1 and 6 weeks.

Two different types of duty rosters are distinguished, cyclic duty and non-cyclic duty rosters. A cyclic duty roster is a roster where each nurse cycles through all the different schedules of the complete roster. The "fairness" of this type of duty roster is deemed to be high. The disadvantage is that no personal request can be incorporated since all nurses have to work all different rosters. The big advantage is the fairness and that the nurses would know their work schedules many months in advance. The cyclic roster is generated once and is first regenerated when external changes occur. On the other hand, the non-cyclic roster is generated every period and can handle personal preferences and requests. The "fairness" is in general not as good as with a cyclic schedule, but can be handled by introducing some "fairness" constraints or as an optimisation criteria. A non-cyclic duty roster can include planning of official holidays, planned vacation, and other time dependent constraints. In a modern hospital, the cyclic schedule seems to be a leftover from the past, since demands and preferences are quickly changing.

3.1 Usual models

In the literature about nurse rostering there is no general model that describes all of the nurse rostering problems. This is due to fact that there are so many differences between the specific problems encountered at the different wards around the world. The modelling of a problem also has a big impact on which solution procedure can be used for solving the problem.

The literature does not provide a complete mathematical description of all aspects of a real-world problem, but there are two compact models which capture big parts of the real world problem. A lot of the constraints to be taken into account are nonlinear and in general hard to describe mathematically. One of these models describes the feasibility constraints for each nurse indirectly using a set of feasible schedules. The other model is used when column generation is applied to solve the problem.

3.1.1 Direct model

In this model, every day is divided into a set of shifts that a nurse can work. The decision variables v_{nrm} are binary and indicate if nurse n should work shift r on day m or not. Let V_n indicate the matrix

of decisions variables for each nurse, that is $V_n = \{v_{nrm}\}_{r,m}$. The problem may then be written as the integer program:

$$\min \quad \sum_{n=1}^{N} C_n(V_n) \tag{3.1}$$

$$\text{s.t.} \quad \sum_{r=1}^{R} v_{nrm} = 1 \qquad \forall m, n \tag{3.2}$$

$$\sum_{n=1}^{N} \sum_{r=1}^{R} \sum_{m=1}^{M} a_{rmi} e_{ni} v_{nrm} \geq d_i \quad \forall i \tag{3.3}$$

$$V_n \in S_n \qquad \forall n \tag{3.4}$$

$$v_{nrm} \in \{0, 1\} \qquad \forall n, r, m \tag{3.5}$$

In this model $C_n(\cdot)$ is the cost arising when nurse n works the schedule given by the decision variables. Constraint set (3.3) is the coverage constraints. A coverage constraint is a constraint on how many nurses of a given qualification level are required to work some given shifts. The minimal requirement of a constraint is denoted by d_i and is often called the demand of the constraint. The zero-one parameter a_{rmi} is equal to one if shift r of day m is one of the given shifts in coverage constraint i. The zero-one parameter e_{ni} is equal to one if nurse n is of a qualification level required in coverage constraint i. The set S_n is the set of feasible schedules for nurse n, the set can be given either explicitly or by constraints defining the feasible schedules. Constraint (3.2) states that each nurse should only work one shift each day, and the last constraints set (3.4) models that all schedules should be feasible.

This model is a very simple model, ignoring a number of constraints that usually have to be taken into account. But it captures some key aspects of many nurse rostering problems.

3.1.2 Set-covering type model

This model is used when the proposed solution method is a column generation approach, where a column corresponds to a full schedule for one nurse.

For each nurse there is a set of feasible schedules S_n and associated binary decisions variables v_{ns}. The variable is equal to one if nurse n works schedule s.

$$
\begin{aligned}
\min \quad & \sum_{n=1}^{N} \sum_{s \in S_n} c_{ns} v_{ns} && && (3.6)\\
\text{s.t.} \quad & \sum_{s \in S_n} v_{ns} = 1 && \forall n && (3.7)\\
& \sum_{n=1}^{N} \sum_{s \in S_n} a_{nsi} v_{ns} \geq d_i && \forall i && (3.8)\\
& v_{ns} \in \{0,1\} && \forall n \; \forall s \in S_n && (3.9)
\end{aligned}
$$

The constant c_{ns} is the cost of nurse n working schedule number s. The parameter a_{nsi} is a zero-one constant that is equal to one if assigning nurse n to schedule s contributes to meet the demand of coverage constraint i. The first constraint set is the constraint that each nurse should work exactly one schedule. The constraints (3.8) ensures that all coverage constraints should be fulfilled.

This model is also very simple, and in real worlds applications there are always a number of extra constraints that needs to be included.

3.2 Standard constraints

Different hospitals around the world need to take different sets of constraints into account when planning their nurses' duty rosters. The literature accordingly distinguishes a number of different constraint sets when describing these problems. Some constraints even stem from single nurses having local contracts with the ward. Some of the constraints often encountered are the following:

Coverage constraints describe how many nurses of different qualification are supposed to work at a given time period or shift type. These constraints can be expressed as a minimum, maximum or exact requirement, sometimes it is given as a desired and a required level.

Workload constraints restrict the number of hours a nurse should work during the planning period. During shifts on holidays and during weekends the nurse often receives a time bonus. The time bonus is included in the calculations of the nurse's work hours. These constraints can be expressed as a minimum, maximum, range or exact number of hours.

Shift pattern constraints are constraints on how different shifts can be combined. These constraints can be given as strict constraints when a certain pattern is not allowed or as soft constraints when the pattern should be avoided if possible. These constraints include constraints such as: minimum number of hours between two shifts, at least two consecutive night shifts and not more than 4 consecutive night shifts.

Consecutive work / free day constraints are often both given as a maximum and minimum number of consecutive work days a nurse can have. The minimum number of consecutive free days is also often restricted.

Complete weekends constraints state that the nurse should either work the whole weekend or have a no weekend shift in that weekend at all. It is most common that nurses working a complete weekend are working two shifts during that weekend.

Holidays and requested free days should also be taken into account when handling real world problems. Holidays and requested free days are often planned long time before the actual planning takes place, and these can of course not easily be changed.

Shift preferences are in particular, a nurse's personal preferences for having specific days off. Some nurses also like to work night shifts. Such preferences should also be considered when creating the schedule.

Number of shift types constraints limits the number of different shift types a nurse should work during the planning period. A constraint on the number of night shifts is common, but these constraints can be on any shift type and be given as a minimum, maximum, range or exact value.

3.3 "Optimal" schedule

Defining how an optimal solution should look like, or even just comparing two given schedules is not always easy. The big difficulties when defining how an optimal schedule should look like arises due to the fact that all the different nurses and the management have an opinion on which criteria to apply.

From the management point of view, criteria like minimising under coverage of shifts and minimising the cost of overtime payments and payments to auxiliary nurses is often most important. But even when trying to handle only two criteria, an optimal solution is in

general not defined. In some way the criteria should be melted together in a single objective. Doing this is not easy, and there is no general applicable method that always works. The most common way of handling several criteria is to multiply them by a weight and to add them together.

From the nurses point of view other objectives are as important as those of the management. Often used objectives are the following: maximise fulfilment of nurses' preferences and personal wishes, maximise the "fairness" of the generated schedule. These two objectives are hard to define on their own, combining them with others makes it even harder. The maximisation of nurses preferences causes further problems as to figure out which preferences are most important. For instance should the importance of a nurse getting a shift off due to a 50 years birthday be more important than a nurse not being able to go to football practice once, and what about if it were a 50 years birthday compared to five missed football practices.

The fairness of a schedule is also difficult to define, but it often involves criteria like: equal number of night shifts, equal number of evening shifts, equal number of work weekends, and equal amount of preferences fulfilled.

All these different criteria should in some way be combined into an objective and perhaps also some constraints. But for doing this there is no brilliant solution and handling the problem as a multi-criteria problem with so many criteria is not realistic.

3.4 Solutions methods from the literature

Many different solution approaches for solving nurse rostering problems have been proposed in the literature, varying from exact solution methods as integer and constraint programming to a huge variety of heuristics.

Warner [51] describes an early approach that combines manual planning and integer programming. The method is most suited for instances where the nurses work some sort of rotational profile with a manually fixed weekend work pattern. The planning period is divided into two week periods, which are planned almost independently in order to reduce the problem size. The simplifications and the use of rotational profiles make the solution procedure unsuitable for the needs of modern hospitals.

Jaumard et al. [33] propose an integer programming model that is solved exactly by means of column generation/branch-and-price. The pricing sub-problem is to generate feasible duty rosters for each

nurse and the master problem is to combine these rosters into a complete schedule. The pricing sub-problem is solved as a resource constrained shortest path problem with a pseudo-polynomial two phase algorithm. This method is pretty flexible, but the sub-problem is very sensible to the number and types of constraints that are added. Many constraints are also difficult to state as resource constraints.

Also constraint programming has been proposed as a solution method for the nurse rostering problem. Okada [47] proposes a constraint satisfaction problem (CSP). The model is very flexible regarding the constraint set, but no soft constraints or objective is included. The search strategy of the CSP selects the variable and value to branch on according to the nurses' preferences, and the algorithm terminates with the first feasible solution found. Wong and Chun [52] suggest another constraint programming method. They reduce the size of the search tree by adding redundant constraints. The problem instances considered are one week instances. This approach neither includes an objective nor does the method account for preferences for shifts or shift patterns.

Some papers combine a constraint programming model with local search or other non-exhaustive search procedures. Meisels et al. [44] and Meyer auf'm Hofe [45] both describe constraint programming models, which have been implemented in software packages and used in several different hospitals. Meisels et al. [44] use an ordering heuristic to implement the nurses personnel preferences and by this hope to find a solution that obeys most of the preferences. The ordering heuristic is used when branching in the search tree should be performed. The search terminates when a feasible solution is found. Meyer auf'm Hofe [45], on the other hand, combines the usual branch-and-bound search with a local search procedure. For comparing solutions, the local search applies a combination of a lexicographic ordering of the constraints and a weighted sum over the violated constraints. The article also introduces fuzzy constraints which are constraints that can be partly violated.

Li et al. [37] suggest a similar approach. They propose to soften some of the hard constraints in a constraint programming model, where all preferences and costs are ignored. After a solution to the model is found, a tabu search is applied to search for a feasible solution for the unrelaxed problem and to improve the objective of the generated solution.

Tabu search has been proposed in several articles; Burke et al. [6] discuss a hybrid tabu search with human inspired improvement techniques. This approach has been implemented and used in over 40 Belgian hospitals. The system is quite flexible and produces solu-

tions that are almost impossible to improve manually. Dowsland and Thompson [17] also describe a tabu search. A knapsack model is used for pre-processing and an integer flow model is used for post-processing of the solution. Bellanti et al. [3] suggest a pure tabu and an iterative local search procedure for solving a nurse rostering problem from an Italian hospital. Burke et al. [5] propose and compare several genetic, memetic, tabu search and hybrid approaches. All these procedures restrict the search to the feasible region of the search space.

Goodman et al. [22] discuss a greedy randomised adaptive search procedure. The search space of the method includes parts of the infeasible region, but an additional surrogate knapsack constraint is included to ensure that an infeasible solution is easy to repair.

Heuristics for nurse rostering problems are generally designed for a particular objective function and a specific set of constraints. The methods are thus very inflexible towards changes of the problem setting. Moreover, difficult instances of the nurse rostering problem generally show only a few feasible solutions that can be hard to find by heuristics. Moreover, even if additional meta-heuristics search principles are applied, heuristic search procedures often get caught in a local optimum, in particular, if just few solutions are feasible and the heuristic search is restricted to the feasible region.

More references and a comprehensive discussion of nurse rostering and solution approaches for it considered in the literature can be found in Burke et al. [7]. A bibliographic overview without a discussion of the proposed methods can be found in Cheang et al. [11].

Chapter 4

The Danish model

The problem confronted in this thesis is described in a master thesis by Anne Kirstine Andersen [41]. She presents a nurse rostering problem from a ward at Aarhus University Hospital Skejby in Denmark. Most of the problem's constraints stem from union contracts or are constraints commonly imposed from the management. A solution approach for this particular problem would thus probably generalise to other wards in Denmark. In her master thesis, the problem and a solution method for a strict subset of the constraint are described. The solution method is not capable of handling all the constraints and thus not the actual problem to be solved.

The problem is described in the following three sections. Section 4.1 gives information about the size of the problem and the more commonly used constraints. Regulations that distinguish this problem from other problems addressed in the literature are described in Section 4.2, and Section 4.3 discusses how to characterise a good solution for this problem.

4.1 The confronted problem

The confronted problem is to plan the schedules for 28 nurses for a planning period of one month. The nurses are divided into three qualification levels. The levels are ordered such that a nurse with a higher qualification level always can do the work of lower qualified nurses.

There are in general three working shifts each day, except Sunday with five and Saturday and Monday both with four. Besides this, some nurses have to have office days during the planning period, and these should also be included in their schedule. Three of the working shifts take eight hours (night, day, evening). The others are twelve hours shifts and only used during the weekend. The weekend is defined to start Saturday after the night shift between Friday and Saturday and to end after the night shift between Sunday and Monday. The two types of 12 hours shifts are a long day starting the

same time as a normal day shift and a long night shift starting when the long day shift ends.

Some shifts start at one day and end the day thereafter. Such shift are considered to be at the day where most of the working hours is located. For example the night shift which starts at 23:15 is considered to be a working shift of the next day.

The constraints that are very commonly used are listed below. Problem specific constraints are described in Section 4.2:

Minimum time between shifts. In this problem, the minimal time between two working shifts is 11 hours. (C_1)

Minimum consecutive workdays. There must be at least two consecutive workdays, that means a pattern like "free day – work day – free day" is forbidden. (C_2)

Maximum work hours per week. Each nurse should have a maximum of 48 work hours each week. A week starts on a Monday and ends on a Sunday. (C_3)

Shift type limits. There is a limit on how many and how few night and evening shifts a nurse can work during the planning period. Imposing this constraint is a mean of more evenly dividing the "bad" shifts between the nurses. (C_4)

Office days. Some of the nurses have office days. These are not fixed, but given as a demand of a fixed number of shifts in a given period. A nurse working an office shift is not considered to be included in the staffing at the ward. (C_5)

Consecutive work weekends. A nurse is not allowed to have two consecutive work weekends. (C_6)

Recorded work hours. Each nurse has a number of contracted work hours she should work on average in each week. These hours are often called recorded work hours, since weekend shifts and shifts at official holidays earn extra hours. A weekend shift gives an extra of 40 % and shifts during an official holiday gives 50 % extra time. For each nurse, the recorded work hours should hit the average every twelfth week, otherwise the nurse get paid for her overtime or the ward just loses the hours not worked. These twelve week periods starts when the nurse is hired so they are not the same for all nurses and they do in general not fit with the planning period. The management has decided to set a range around the contracted number of hours. The recorded number of hours should then be inside this range at the end of the planning period. (C_7)

Preassigned shifts. Preassigned holidays and non-working days should be respected. (C_8)

4.2 Distinguishing features

Below is a list of the constraints that differ from the ones generally found in the literature:

Day off / free day. In a Danish ward, the days in which a nurse does not work can be divided into three different types: holidays, days off and free days. (C_9)

Holidays are preplanned spare time, and these cannot be changed in reality. The reason is that the union agreements charges a large penalty if a nurse gets her holidays changed.

A day is called a day off if the nurse should not work any shift that day and if the day is in a period of at least 35 hours of spare time. A day off thus depends not only on the fact if the nurse works the corresponding day, but also which shifts the nurse should work the following and preceding day.

A free day is defined as a day with no shifts that is not a day off.

Maximum consecutive workdays. There are two different constraints on the maximum number of consecutive workdays depending on a local agreement between the nurse and the management. (C_{10})

The standard constraint is that a nurse should have a maximum of six work days between two days off. This is the same as having a maximum of six consecutive workdays where a free day is considered as a workday.

If a nurse has accepted, the constraint can be weakened such that a nurse can work a maximum of seven days between two days off, if there is a free day in between. If there is no free day in between, the limit of days between a day off is still six.

Complete weekends. All nurses that have to work during a weekend have to work a complete weekend. A weekend shift does, however, not mean to work the same shift type on a Saturday and the Sunday. Instead the weekend is shifted a bit and covers the first shifts of the Monday, but not the first shifts of the Saturday. However, working a complete weekend still consists of working two shifts of the same type. (C_{11})

12 hours weekend shifts. If a nurse has accepted to work 12 hours shifts, then it should be ensured that the first work weekend after a work weekend with 12 hours shifts is one with 8 hours shifts. (C_{12})

Overlap of nurses. The management has a request for an overlap of at least one nurse from one shift of a day to the same shift of the next day. This is because they want the patients to feel some continuity regarding the nurses which are on duty and also to have a nurse that knows what happened the previous day. This request of an overlap is on both night, day and evening shifts, however the twelve hours shift is in this constraint considered as its corresponding eight hours shift. So, if a nurse works the twelve hours night shift on a Monday and an eight hours night shift on Tuesday, the constraint is fulfilled for the night shift between Monday and Tuesday. (C_{13})

Coverage constraints. Coverage constraints on how many nurses should work each shift are very common. In this problem the constraints require that certain shift combinations are covered by a minimum or best a desired number of nurses having certain qualifications. This differs from the most standard way of stating the constraints, as it restricts shift combinations, accounts for both a minimum and desired number of nurses working and includes only nurses with certain qualifications. (C_{14})

4.3 Objectives

Several different criteria have to be considered in the objective function, in particular the following ones:

Minimise under-staffing. Under-staffing arises when a coverage constraint is not covered by the desired number of nurses. When under-staffing occurs, a penalty is included in the objective. (O_1)

Ensure overlap. Penalties are included if the overlap constraints are not met and treated as soft constraints. (O_2)

Nurses preferences. Fulfilment of the nurses' request and preferences is an important criterion. If not fulfilled a penalty cost is included. Examples of preferences are: A nurse would like to have free a specific date, or a nurse that would like to work only night shifts. (O_3)

Shift change cost. If two consecutive days have been assigned two different work shift types, a penalty cost is added. The value of the cost depends on which shifts are assigned. (O_4)

Recorded work hours. If a nurse is at the end of her 12 weeks period, then the recorded work hours are compared to the figure fixed in the nurse's contractual agreement, and overtime payments are paid if there is an excess. If the recorded hours are below the assigned level, the hours are lost and an opportunity cost is included to account for the unused working hours. (O_5)

The objective of the problem is defined as the weighted sum of the different terms above.

Part II

Exact approaches

Chapter 5

IP/CP model

5.1 The solution method

In the following sections the main solution approach developed in this thesis is described. The solution method is a branch-and-price algorithm which can be used to determine optimal schedules. The master problem is to select individual schedules for the nurses such that the constraints C_{14} and C_{13} are observed while the sum of the objective terms O_1 to O_5 is minimised. We use the "Branch and Cut and Price" project (BCP), version 1.3.3 COIN-OR [12], of COIN-OR to implement the branch-and-price method and CPLEX 12.2 IBM [29] for solving the restricted master problem's linear relaxation.

The pricing sub-problem is to generate feasible schedules with negative reduced cost for each nurse or to show that no such column exist. It includes the constraints C_1 to C_{12}, with the objective consisting of the terms O_3 to O_5 as well as the "shadow prices" associated with the coverage and overlap constraints C_{14} and C_{13}, respectively. Additionally, branching constraints imposed in the branch-and-bound search need to be taken into account. The sub-problem is solved by constraint programming, because of its flexibility and because it allows to implement all the constraints in a straightforward manner. The constraint programming model is implemented in the IBM ILOG solver 6.7 IBM [31], with the use of only a few of the built-in constraints. The major part of the constraints has been implemented using the built-in C++ interface IBM [30].

The master problem is further discussed in Section 5.2, and the sub-problem with descriptions of how the different constraints are implemented are described in Section 5.3.

5.2 The master problem

The master problem is based on the model (3.6) – (3.9) given in Section 3.1.2 on page 28. Besides the coverage constraints, which are already a part of the base model, this model also includes constraints

that handle the overlap requirements. The possibility of having soft covering constraint with a required and a desired level of demand is also included in the model.

Let the set of nurses to be scheduled be N, and let S_n be the set of feasible schedules for nurse n. The constant c_{ns} is the cost of having nurse n work schedule s.This cost includes both direct (O_5), opportunity (O_5) and preference cost (O_4 and O_3).

Let v_{ns} denote a binary decision variable that equals one if and only if nurse n works schedule s.

Let CC be the set of coverage constraints C_{14} and let for each coverage constraint $i \in CC$, a desired demand level d_i be specified. The constant $\bar{d}_i$ is the difference between the desired and required demand level. This constant is zero if the coverage constraint is strict. Otherwise the decision variable y_i, being the difference between the desired and actual staffing level of the current roster, may attain a positive value and a corresponding linear penalty p_i is included in the objective function.

The parameter a_{nsi} is a binary constant that is equal to 1 if and only if nurse n contributes to meet the demand of coverage constraint i by working schedule s.

The overlap constraints C_{13} are collected in the index set OC. If a constraint $j \in OC$ is considered a soft constraint, then the corresponding binary variable z_j indicates if the constraint is fulfilled or not. If the constraint is violated, a penalty of q_j is added to the objective. The 0-1 constant $\bar{z}_j$ is 1 if the corresponding overlap constraint is a soft and 0 if it is a strict constraint. If an overlap constraint is strict, the corresponding variable can be dropped from the model. The parameter o_{nsj} is equal to one if nurse n in schedule s covers the overlap requirement of overlap constraint j. For instance, given an overlap requirement between two shifts of two consecutive days, we have $o_{nsj} = 1$ if nurse n works both shifts in schedule s.

The nurse rostering problem may then be formulated as the following huge integer program:

$$\min \quad \sum_{n\in N}\sum_{s\in S_n} c_{ns}v_{ns} + \sum_{i\in CC} p_i y_i + \sum_{j\in OC} q_j z_j \tag{5.1}$$

$$\text{s.t.} \quad \sum_{s\in S_n} v_{ns} = 1 \qquad \forall n \in N, \tag{5.2}$$

$$\sum_{n\in N}\sum_{s\in S_n} a_{nsi}v_{ns} + y_i \geq d_i \qquad \forall i \in CC, \tag{5.3}$$

$$\sum_{n\in N}\sum_{s\in S_n} o_{nsj}v_{ns} + z_j \geq 1 \qquad \forall j \in OC, \tag{5.4}$$

$$v_{ns} \in \{0,1\} \qquad \forall n \in N\ \forall s \in S_n, \tag{5.5}$$

$$y_i \in \{0,1,\ldots,\bar{d}_i\} \qquad \forall i \in CC, \tag{5.6}$$

$$z_j \in \{0,\bar{z}_j\} \qquad \forall j \in OC. \tag{5.7}$$

The first constraint set (5.2) ensures that each nurse only works one schedule. The set (5.3) is the coverage constraints, and (5.4) is the overlap constraints.

5.2.1 Restricted master problem

The linear relaxation of the restricted master problem that is solved by CPLEX is the following:

$$\min \quad \sum_{n\in N}\sum_{s\in \tilde{S}_n} c_{ns}v_{ns} + \sum_{i\in CC} p_i y_i + \sum_{j\in OC} q_j z_j \tag{5.8}$$

$$\text{s.t.} \quad \sum_{s\in \tilde{S}_n} v_{ns} = 1 \qquad \forall n \in N \tag{5.9}$$

$$\sum_{n\in N}\sum_{s\in \tilde{S}_n} a_{nsi}v_{ns} + y_i \geq d_i \qquad \forall i \in CC \tag{5.10}$$

$$\sum_{n\in N}\sum_{s\in \tilde{S}_n} o_{nsj}v_{ns} + z_j \geq 1 \qquad \forall j \in OC \tag{5.11}$$

$$0 \leq v_{ns} \qquad \forall n \in N\ \forall s \in \tilde{S}_n \tag{5.12}$$

$$0 \leq y_i \leq \bar{d}_i \qquad \forall i \in CC \tag{5.13}$$

$$0 \leq z_j \leq \bar{z}_j \qquad \forall j \in OC \tag{5.14}$$

Where $\tilde{S}_n$ is a subset of all the feasible schedules for nurse n, that is $\tilde{S}_n \subseteq S_n$.

After the restricted master problem has been solved, the dual values of the constraints are passed to the sub-problem. The sub-problem is to generate negative reduced cost columns or to prove that no such column exists for any nurse. If at least one negative reduced cost column is found, it is added to the restricted master problem and the restricted master problem is re-optimised.

5.2.2 Lower bound for the master problem

In each node of the branch-and-bound tree created by branching in the master problem, a lower bound for the linear master problem, that is, the linear relaxation of (5.1) – (5.7), can always be obtained by adding the negative reduced cost for each pricing sub-problem to the linear relaxation of the restricted master problem's objective. The lower bound found is valid for all nodes in the sub-tree rooted at the current node.

As all costs are integer-valued, this lower bound value can be rounded up, to obtain a valid lower bound for the integer master problem in the same node. Of course this rounded lower bound is not a valid lower bound for the linear master problem, and hence the solution value to the restricted linear master problem might fall below this lower bound.

Some initial testing of the algorithm showed that solving all the sub-problems to optimality generally resulted in smaller search tree and lower computation times. Thus all sub-problems are solved to optimality in all iterations of generating schedules for the restricted master problem. The reason for the lower computation time is that the lower bound is very important for pruning nodes in the search tree.

5.2.3 Branching in the master problem

Branching in the master problem is performed after terminating the column generation procedure for solving the master problem's linear relaxation. The column generation stops as soon as the restricted master problem's objective equals or, due to the rounding, drops below the lower bound. In order to escape the well-known tailing off effect, the column generation is also stopped when the restricted master problem's objective does not improve after a given number of subsequent iterations (provided the solution to the current linear master is not integer).

Branching on the v_{ns} variables would give a very unbalanced search tree. Forcing a value of one to such a variable would fix

the nurse's schedule totally, but forcing a value of zero would have an extremely small impact. Instead branching is performed on some auxiliary binary variables defined below.

Let $\bar{S}_n^{\delta\sigma} \subset S_n$ be the set of schedules that assign nurse n to shift σ on day δ. All non-working shifts are in this combined into a single shift type. Branching is then performed on the auxiliary binary variables

$$V_{n\delta\sigma} := \sum_{s \in \bar{S}_n^{\delta\sigma}} v_{ns} ,$$

which attain a value of one if nurse n works shift σ on day δ. When all the variables $V_{n\delta\sigma}$ are integer, all schedules $s \in S_n$ where $v_{ns} > 0$ comprise the same sets of working shifts. Given all working shifts for a nurse n, the free shift types allocated to nurse n in schedules s with $v_{ns} > 0$ are easily deduced. Thus, also all schedules $s \in S_n$ with $v_{ns} > 0$ are identical. But, as all generated schedules are different, only one such variable $v_{ns} > 0$ can exist and the solution v_{ns} is therefore integer.

For any given (nurse, day) pair, we select the variable $V_{n\delta\sigma^*}$ with a value closest to 0.5 in the current LP solution for branching. This leaves only the particular (nurse, day) pair to be selected for deciding on the branching variable. For this purpose, we use a strategy resembling the "strong branching strategy" of LP-based branch-and-cut procedures often used to solve "difficult" integer programs (cf., for instance, Achterberg et al. [1]). First we build a set of candidate (nurse, day) pairs by sweeping through the work days δ, starting with the Sundays, and selecting all nurses n such that $|\{\sigma : V_{n\delta\sigma} > 0\}| > 1$ until a given number of pairs is found. For each candidate pair, each branch ($V_{n\delta\sigma^*} = 0$ vs. $V_{n\delta\sigma^*} = 1$) is investigated and the linear master problem re-optimised, however, without generating additional columns. The branching candidate variable's degradation is then measured as the average objective value on both branches (only when branching was performed because the master problem's objective dropped below the lower bound, the degradation is defined by the smallest objective value observed on the two branches). The candidate variable showing largest "degradation" is then selected for branching.

5.2.4 Exploring the branch-and-bound tree

The order of exploration of the branch-and-bound tree created by the above branching has a large influence on the computation time required for solving a problem instance. Several different orders have been investigated:

BB Best bound search.

BF Breadth first search.

DF Depth first search.

BBR Best bound, with random selection between equal.

DFR Depth first until pruning, then random selection.

SA Simulated annealing inspired selection.

The three first methods for selecting the next node to explore are standard methods from the literature. The other three methods are described below. Some preliminary tests performed with the three standard methods showed that sometimes the search gets stuck in an area of the search tree where no improved integer solution exists or where the lower bound cannot be improved. This is a usual issue with the depth first search, but it was also seen with the best bound search. The reason behind this is that the lower bound for many nodes were the same and that the nodes added last with the best bound were searched first. To overcome this issue the three new methods were implemented; the three standard methods were re-implemented, because some major bugs were found in the BCP-implementation.

The idea behind the three new methods is to do a depth first search, but to select another area of the search tree each time the depth first search does not seem to be able to find a feasible integer solution.

Best bound, with random selection between equal. It is very similar to "best bound search", the only difference is that the node selected for exploration is randomly chosen between the nodes showing the best bound. In the "best bound search" the node that is created first is selected. The rounding used for finding the lower bound for the nodes results in many nodes with the same bound. Hence quite often this order of exploration will differ from the best bound search.

The reason this has an influence for these instances is that the bound is found by rounding, thus resulting in many nodes with the same bound.

Depth first until pruning, then random selection. Sometimes the "depth first search" got caught in sub-tree of the search tree where no feasible integer solutions existed. The algorithm ended up searching in this part of the feasible region until the time limit was exceeded.

The search selection is quite simple: If exploration of a node leads to the need for branching then the first child is explored as the next

node. That is the child where a nurse is forced to work a specific shift of a specific day. If no branching is needed, the next node to explore is selected uniformly at random from the whole search tree.

This method leads to a search that dives down the search tree until it reaches an integer feasible solution or the node is pruned. Then it chooses a node at random and dives from that node. The method seems to find integer solutions quite quickly and it seems to explore different parts of the search tree instead of just looking in the same area.

Simulated annealing inspired selection. This method works similar to the "depth first until pruning, then random selection" method. The difference is that it might stop the diving before an optimal integer solution has been found or the node has been pruned. The criterion that determines if the diving should be stopped early is similar to the acceptance criterion normally used in simulated annealing:

$$P(T) = \alpha^T,$$

where T is the number of dives since a node was randomly selected the last time, and $\alpha \in]0,1[$ is a constant. A dive is accepted with a probability of $P(T)$.

Computational comparison

The different search strategies described above have been tested on 15 different instances with a planning period of 14 days. As parts of the algorithm and some of the search strategies rely on randomness, all instances have been tested with 10 different seed corns. The instances are those also used for the comparison of methods for solving the sub-problems in Section 5.3 and for the comparison of the solution methods in Section 7.2.

The tested search strategies are:

- **BB** Best bound search.
- **BF** Breadth first search.
- **DF** Depth first search.
- **BBR** Best bound, with random selection between equal.
- **DFR** Depth first until pruning, then random selection.
- **SA** Simulated annealing inspired selection.

Table 5.1 summarises the results of the comparison. The first column shows the instance number and the second the number of nurses to be scheduled in the instance. The column below "Min.

Table 5.1: Comparison of search strategies for the branch and bound tree of the master problem. (Averages over 10 runs.)

Ins.	Nurses	Min. time	BB	BF	DF	BBR	DFR	SA
		Seconds	Extra computation time spend in % of the min. time.[a]					
1	20	45.03	3,482.1	2,573.9	0.00	2,296.9	6.79	53.43
2	20	680.58	49.2	84.1	0.00	1.0	34.48	15.48
3	20	41.59	444.4	233.8	4.68	390.6	0.00	5.36
4	28	122.53	464.1	325.6	0.00	303.6	15.52	63.85
5	28	29.19	1,010.4	2,928.1	0.65	2,415.0	0.00	57.11
6	28	33.15	1,868.4	3,205.5	0.00	971.0	3.14	27.45
7	28	19.21	64.3	56.0	0.00	96.2	0.98	14.99
8	28	73.10	38.1	47.2	0.00	22.5	4.26	3.22
9	28	18.73	196.8	480.2	4.52	190.2	0.00	12.51
10	38	34.46	7,024.3	6,586.6	2.53	4,203.5	0.00	45.17
11	38	22.51	751.2	559.9	10.86	377.1	0.00	3.55
12	38	34.62	34,594.7	25,962.0[††]	11.21	12,567.7	0.00	67.67
13	38	36.86	1,826.5	1,230.7	0.00	1,162.3	2.27	31.86
14	38	29.07	586.1	870.5	0.00	405.2	5.39	7.69
15	38	30.32	236.3	591.7	26.11	217.4	0.00	63.01
Average:			3,509.1	3,049.0	4.04	1,708.0	4.85	31.49

[a] The extra computation time is calculated as the extra time used with the given search strategy compared to the one using the minimal; e.g., for the best bound (BB) search strategy, it is calculated as: $(\text{Time}_{\text{BB}} - \min_i \text{Time}_i)/(\min_i \text{Time}_i) \cdot 100\%$

[††] One run did not return any solution before the time limit of 20.000 seconds. The time limit of 20.000 second was used as the solution time of that run.

time" is the smallest computation time obtained with any of the above search strategies for solving the instance. Remark that all computation times are the averages of 10 runs. The rest of the columns are the relative extra time required for the given search strategy compared to the one using minimal computation time. E.g., for the best bound (BB) search strategy, it is calculated as:

$$(\text{Time}_{\text{BB}} - \min_i \text{Time}_i)/(\min_i \text{Time}_i) \cdot 100\%. \tag{5.15}$$

Because the minimum is not over the single run, but over the averages, there is at least one 0.00 in each row.

The results show that the DF and DFR search strategies are the ones that on average use the least computation time. The DF search uses around 4.0 % more computation time than the strategy using the least computational effort, whereas the DFR uses around 4.8 % more time. The DF search did not get caught in an area where no improvement could be found, as it did in the preliminary tests.

Even though the DF search was slightly faster, the DFR search was selected for the final algorithm. This was done because it usually cannot get caught in a small area of the search space.

The good performance of the DF and DFR search strategies is probably due to the fact that the lower bound found in the root node of the restricted master problem is usually very close to, if not equal, to the optimal objective value. Thus, finding a integer feasible solution quickly is more important than improving the lower bound.

The two other standard search strategies showed extremely poor performance. The best bound (BB) search strategy used on average 3,509.1 % more computation time than the strategy using the least, and the breath first (BF) used an average of 3,049.0 % more computation time. The BBR method is the BB method where the node to search is selected randomly between those having the best lower bound. The BBR showed a better performance than the BB by using around half of the extra time than the BB method. Thus it looks like the BB search strategy sometimes got caught in an area of the search space where no improvement could be found.

The SA method did quite well compared to the standard BB and BF method, but it could not match the computation times of the DF and DFR methods. The computation times reported for the SA method refer to the value of the parameter α showing the lowest computation times (5 different settings were tested). On average, the SA method used 31.49 % more time than the one using the least.

5.3 The pricing sub-problem

The pricing sub-problem is the problem of generating feasible schedules with negative reduced costs or proving that no feasible schedule with negative reduced cost exists.

The method used for this problem should be general enough to include all the constraints described in Section 4. It would also be preferable, if the method can handle changes in the constraint set such that the solution method generalises to more than this particular ward.

We have chosen to use constraint programming for solving the sub-problem. One of the main reasons for this choice is that all constraints can this way be implemented in a straightforward manner and it is easy to add new ones.

Besides the constraints given in Section 4, the solution procedure should also handle the reduced cost calculation, which include direct cost, preference cost, and dual values from both the coverage and

the overlap constraints. The sub-problem should also handle the branching constraints imposed in the branch-and-bound tree of the master problem.

The following sections describe how the different constraints are exploited within the constraint programming approach for performing domain reductions, computing lower bounds on the reduced cost of sub-problem and to early prune the search tree.

Section 5.3.1 introduces the general variables which are used to model the constraints. As an introduction to domain propagation mechanisms, Section 5.3.2 includes several different domain propagation mechanisms for the constraint on the minimal time span between shifts, including the one used for solving the instances. Propagation mechanisms for most of the other constraints are described in Section 5.3.3 to Section 5.3.10.

Combining different domain propagation mechanisms into one can often yield a stronger mechanism, by that it is meant that the mechanism can yield some extra domain propagation mechanism that can be used to reduce the domains or prune the nodes earlier in the tree.

Section 5.3.11 is an example of how to combine two domain propagation mechanisms into one, keeping the full domain reduction capability of the individual constraints. Whereas Section 5.3.12 is an example of a domain propagation mechanism that should be used in conjunction with the individual domain propagation mechanisms.

How to calculate a lower bound on the reduced cost is described in Section 5.3.13 and how to use this lower bound to do some domain propagation is shown in Section 5.3.14.

The search strategy used for creating the CP search tree can be found in Section 5.3.15.

5.3.1 The CP model

There are two natural ways of modelling the decisions to be made in the sub-problem. The first way is to use binary decision variables for each shift type, and the second is to use an integer variable for each day indicating which shift the nurse should work.

The set of shift types is expanded from being just actual working shifts to also include a shift type for holiday, a free day and a day off. By doing this it becomes easier to express constraints.

The advantages of the model with binary variables is that some constraints only depend on a small subset of the variables, and that some variables can be merged into a single variable. The first advantage makes it easier to implement the constraints and it reduces

the computational effort to perform the domain propagation mechanisms, as they are called fewer times. Reducing the number of variables by merging them can, for instance, be performed with the binary variables of the weekend shifts. The constraint on working complete weekends enforces that the binary variable for a night shift on a Saturday will always have the same value as the binary value for the night shift of the following Sunday.

When using integer variables to model the problem, there should be one integer variable for each day and it represents which shift type the nurse should work the corresponding day. The advantage of this model is that the constraint that a nurse only can work one shift each day is implicitly fulfilled. An other advantage compared to the other model is that branching or when domain propagation removes several shift types from one day, the domain propagation mechanisms are only called once, whereas the other model calls the algorithm for each value that is removed.

Here, the second approach is applied and the model is built around a single integer variable x_k for each day $k = 1, \ldots, M$ of the planning period. The variable x_k equals the identifier of the shift (work or non-work shift) assigned to the particular day k. To reduce the number of checks necessary in the propagation mechanism of some of the constraints, the shifts are ordered such that the non working shifts are the first shifts, followed by the 8 hours working shifts and ending with the 12 hours working shifts. The ordering makes a difference, when it is necessary to figure out if a day is a working or a non-working day, when the shifts are ordered this way the only information needed is the maximal and minimal value of the variable.

The model is implemented in the ILOG CP solver. The solver includes some different variable types and the only one appropriate for modelling the variables is the standard integer variable implementation. The variable is implemented for the use of describing a size of some measure and not for stating choices. If a constraint depends on a variable, the constraint should be noticed when the variable changes, so that domain propagation will be performed. There are three different types of actions a constraint can "listen" to. The first is that the constraint is noticed when the variable is fixed to a value, the second is when the variable's upper or lower bound changes, and the last is when the domain is changed. When the variable is used as a choice indicator, a fourth option is needed: The constraint is to be noticed if either a variable needs to lie in a specific subset of its domain or when it needs to lie outside this subset.

An example is the constraint on office days. This constraint

should ensure that a given number of office days are planned. For this constraint, only the information about if a variable is fixed to an office day or when the office day is removed from the domain can lead to new domain reductions.

Instead of using the implemented variable type directly, a layer in between has been implemented, which adds the above fourth option of listening to a variable. This extra layer also adds the possibility of deciding in which order the different domain propagation mechanisms are called. This additional way of listening to a constraint makes it easier to implement new constraints, but also the computational effort reduces as less redundant domain propagation methods are called. This implementation enhances the model with the binary model's advantage of only calling relevant domain propagation mechanisms.

This fourth option is divided into three different cases depending on the properties of the subset to be "listened" to. The division is performed due to speed considerations. The general case handles everything which does not fit into the other two cases, but it requires a counter to keep track of when to call the propagation mechanisms. The second case applies when the subset is a single value which does not require a counter. The third case applies when the subset is either the first or the last elements of the domain, the counter can also be avoided here by keeping track of the minimum and maximal value of the domain.

The last feature added is the possibility of having two different propagation mechanism called, when either a variable need to be in a certain subset or when it must lie outside of this subset. This reduces the number of checks required in some propagation mechanisms and it sometimes reduces the complexity of the implementation of the propagation mechanisms.

The branching decisions and the preassigned shifts constraint (C_8) are handled very easily in the constraint programming model, since they correspond directly to reducing some of the domains of the variables.

5.3.2 Minimal time between shifts (C_1)

Domain reductions for a constraint can often be performed in different ways. For most constraints it is an advantage to use the method that yields the most extensive domain reductions, even though it takes some time to calculate this. In this section three different propagation mechanisms for the minimal time between shifts constraint are described.

The minimal time between shifts is the minimal number of hours between the end time of a work shift and the start time of the next work shift. Using a single integer variable x_k to model each day ensures that the minimal time between two shifts of the same day is satisfied, so the constraint should handle the minimal time between shifts of two consecutive days. As the minimum time span between two shifts is 11 hours, there is no restriction regarding shifts of days that are not consecutive.

The first approach does not really perform domain reduction, but prunes the node when the constraint is violated. Two shifts of two consecutive days are said to be incompatible, if a nurse working both shifts would not have the required number of hours between the two shifts. Let r be a value of x_k that is incompatible with value l of x_{k+1}, then add the following domain propagation mechanism:

$$x_k = r \ \wedge \ x_{k+1} = l \quad \Rightarrow \quad \text{"fail"} \tag{5.16}$$

The equation should be understand as follows: If during the search x_k is fixed to shift r and x_{k+1} is fixed to shift l then the mechanism would prune the node, because the assignment is infeasible. The second approach is similar, but this method reduces the domains of the variables instead of pruning the nodes. Let the variables be defined as above, then the following mechanisms should be added:

$$x_k = r \quad \Rightarrow \quad x_{k+1} \neq l, \tag{5.17}$$

$$x_{k+1} = l \quad \Rightarrow \quad x_k \neq r. \tag{5.18}$$

Here the right part should be understood as a domain reduction, where either l or r is removed from the domain of the corresponding variable.

In the first two approaches at least one of the variables should be fixed before any domain reductions can be performed. But it is possible to do domain reductions prior to this and the third method takes this into account. This approach removes values earlier in the search tree and is thus a theoretical stronger constraint than the two other approaches. Let ET(l) denote the end time of shift type l, ST(r) be the start time of shift type r and MTB denote the minimal time between shifts. The constraint can then be stated as:

$$\text{ET}(x_k) + \text{ MTB } \leq \text{ST}(x_{k+1}). \tag{5.19}$$

This constraint can be converted into the following domain propaga-

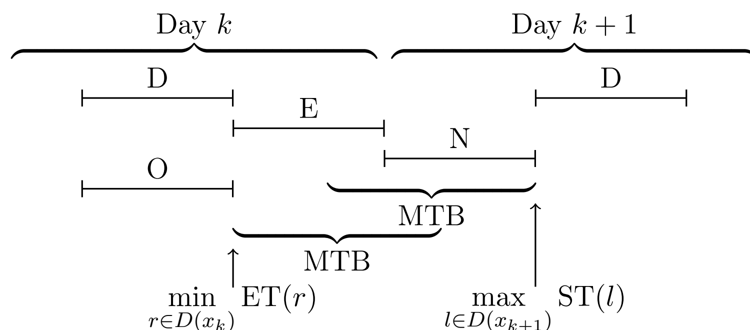

Figure 5.1: Illustration of the implemented domain propagation mechanisms for the minimal time between shifts constraint. D is a day shift, E is an evening shift, N is a night shift and O is an office shift.

tion mechanisms:

$$\min_{l \in D(x_k)} \text{ET}(l) + \text{ MTB } > \text{ST(r)} \quad \Rightarrow \quad x_{k+1} \neq r \; \forall\, r \in D(x_{k+1}), \quad (5.20)$$

$$\text{ET(l)} > \max_{r \in D(x_{k+1})} \text{ST}(r) - \text{ MTB } \quad \Rightarrow \quad x_k \neq l \qquad \forall\, l \in D(x_k). \qquad (5.21)$$

Figure 5.1 illustrates the domain propagation mechanisms. The night shift N of day $k+1$ in this figure can be removed from consideration, because none of the shifts of day k end early enough to ensure a sufficient spare time of MTB hours before the night shift starts. By a similar argument the evening shift E of day k can be removed.

Updating the smallest end time and largest start time and thereby finding which domain values are to be removed, can be done very efficiently by creating two lists: one that sorts the domain values according to the end times and one that sorts them according to the start times. The start times and end times of the non-working shifts are defined such that, when they are in the domain no other domain values can be removed. The number of calls to the propagation mechanisms of this constraint is reduced by collapsing shifts with either the same end time or start time; for each of such collapsed shifts the propagation mechanism is called once, when all shifts of a collapsed shift have been removed from the corresponding day.

Table 5.2: Comparison of domain propagation mechanisms for the minimal time between shifts constraint on CP sub-problems for 14 days instances in 10 runs. Deduced information from the minimal time between constraint is not used for calculating the reduced cost of the new column.

Ins.	Nurses	Sub-prob.	C++3	C++2	C++1	ILOG3	ILOG2	ILOG2B
		Average	In seconds	Deviation in %[a] from C++3				
1	20	310.0	21.55	−4.60	70.04	19.49	3.88	17.01
2	20	9,732.0	740.16	−2.57	53.33	18.15	4.80	18.15
3	20	320.0	33.29	−4.87	225.56	13.60	2.95	15.71
4	28	744.8	128.75	−7.41	89.71	11.08	0.38	10.95
5	28	288.4	16.08	−5.91	45.01	18.25	2.63	15.75
6	28	453.6	25.24	−5.67	56.47	13.14	1.93	14.81
7	28	392.0	11.24	−2.57	137.73	22.90	5.52	20.83
8	28	756.0	66.57	−5.46	152.67	12.45	1.74	13.53
9	28	364.0	10.28	−2.03	91.77	23.01	5.63	21.52
10	38	349.6	15.43	−1.40	107.18	22.67	7.21	21.94
11	38	345.8	13.87	−1.32	99.65	23.44	7.37	22.67
12	38	395.2	15.70	−0.59	103.63	23.76	8.58	24.12
13	38	433.2	17.63	2.26	147.14	22.99	10.71	25.39
14	38	463.6	9.47	0.46	78.99	29.93	8.92	27.20
15	38	456.0	8.87	1.46	95.20	31.62	9.90	28.42
Average:				−2.68	103.61	20.43	5.48	19.87

[a] Positive deviations means longer computation times.

Computational comparison

The following mechanisms were implemented to compare the domain propagation mechanisms for the minimal time between shifts constraint:

- C++1 The first approach from above with the described domain propagation mechanism.
- C++2 The second approach from above with the described domain propagation mechanisms.
- C++3 The third approach from above with the described domain propagation mechanisms.
- ILOG2 The second approach with the use of the built-in constraint “IloIfThen”.
- ILOG3 The third approach with the use of the built-in “IloTable” constraint, which is used as what would in the literature often be called an element constraint.

For the C++1 – C++3 mechanisms the domain propagation mechanisms are implemented in C++ without the use of any of the built-in constraints. The ILOG2 and ILOG3 on the other hand are created only width built-in constraint types.

One mechanism seems to be missing and that is the first idea implemented with the built-in functions. The reason is that the solver in the preprocessing phase will anyway deduce the propagation mechanisms of the second approach from the added constraint of the first approach.

When using the built-in functions, it is important to note that when adding the mechanism from equation (5.17), the mechanism from equation (5.18) is automatically deduced. Adding both results in a lot of redundant calculations which significantly increases the computation time. The inclusion of both mechanisms with the built-in "IloIfThen" function is denoted by "ILOG2B" in Table 5.2.

The CP sub-problems used for testing the different mechanism were those that have to be solved, when solving the two-week instances used in Chapter 7. All instances were solved with 10 different seed corns, because some parts the IP/CP method relies on random selections. The method for calculating the reduced cost for the generated schedule uses the incompatibilities between shifts on consecutive days for speeding up the calculations (see Section 5.3.13). Table 5.2 summarises the results of the computations, where the method for calculating the reduced cost does not make use of the incompatible pairs of shifts on consecutive days enforced by this constraint. This is a more fair direct comparison of the methods. For all other constraints in the sub-problems, the methods chosen for the final algorithm were used. The first column states the instance number, the second is the number of nurses in the instance and the third is the number of sub-problems solved as an average over the 10 runs. The column below C++3 shows the accumulated time spend on solving the sub-problems as an average over the 10 runs with the use of the mechanisms corresponding to C++3. The other columns show the percentage deviation in solution time from the time shown in the C++3 column, when using the corresponding domain reduction mechanisms. The table shows that the C++2 method outperforms all the other methods on the first 12 instances, however on the last three instances the C++3 method was slightly faster. The significant difference between C++2 and ILOG2 is quite surprising, as they rely on the same approach and that the added built-in constraints are used without any extra variable or anything else that should increase the computation time. The big difference between C++3 and ILOG3 is less surprising, because to implement the third approach

with the built-in constraints a couple of extra variables for each day are required. As expected the C++1 showed a very bad performance requiring more than double the time for solving the sub-problems.

Table 5.3: Comparison of domain propagation mechanisms for the minimal time between shifts constraint on CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	C++3	C++2	C++1	ILOG3	ILOG2	None
		Average	In seconds	Deviation in %[a] from C++3				
1	20	310.0	21.28	−5.12	−1.37	19.81	3.53	−4.97
2	20	9,732.0	713.01	−4.69	−1.22	18.43	3.43	−5.34
3	20	320.0	30.27	−5.81	−1.31	13.43	2.57	−6.10
4	28	744.8	116.20	−7.53	−5.25	11.01	0.64	−6.72
5	28	288.4	15.66	−6.48	−4.97	18.41	2.10	−7.92
6	28	453.6	23.84	−6.37	−4.83	12.79	1.67	−7.63
7	28	392.0	10.55	−2.55	0.87	23.30	5.58	−5.99
8	28	756.0	61.03	−5.68	−1.35	12.33	1.77	−5.27
9	28	364.0	9.59	−2.01	1.97	23.59	5.61	−5.62
10	38	349.6	12.04	−2.75	1.34	25.28	5.54	−4.52
11	38	345.8	10.61	−2.43	2.15	26.90	6.21	−4.50
12	38	395.2	12.04	−2.38	2.08	26.87	6.28	−4.46
13	38	433.2	14.82	−3.21	3.00	25.69	5.14	−5.16
14	38	463.6	8.84	−0.65	2.87	30.81	7.91	−5.84
15	38	456.0	8.14	0.35	6.15	32.83	8.80	−6.43
Average:				−3.82	0.01	21.43	4.45	−5.76

[a] Positive deviation means longer computation times.

For testing which method should be used in the final CP algorithm, the complete algorithm was tested with the different implementations of the minimal time between shifts constraint. The ILOG2B method were excluded for this comparison. A method was introduced where the minimal time between shifts constraint was handled indirectly by the method for determining the reduced cost of the schedule, as described in Section 5.3.13 and 5.3.14. This method is denoted by "None". The results of these tests are summarised in Table 5.3.

According to Table 5.3, the method using the least computational time is the "None" method. The reason behind this is that the method for calculating the reduced cost removes the same domain values as the C++3 and ILOG3 methods does. The results of this table does not really show how well they are at removing domain values but on how much computation time is spend on checking the mechanisms. The reason that C++1 does not outperform the others is that the

algorithm for calculating the reduced cost is computational heavy, and the more domain values are removed before the algorithm is called, the less calculations are necessary.

The method corresponding to "None" is used for solving the instance, but on other problem instances where the calculation of the reduced cost could not indirectly handle the minimal time between shifts constraint, either method C++2 or C++3 could be useful. The C++2 method is used in the part on heuristics approaches, where some of the heuristics relies on the same CP model as here, but with the exception that it does not include the calculation of the reduced cost.

5.3.3 Day off / free day (C_9)

The day off / free day constraint should ensure that if a variable is assigned a day off, it is in a period of 35 hours of free time and if it is assigned a free day it should be in a period with less than 35 hours of free time.

Let DO be the day off shift, F be the free day shift and let the constant T be defined as $T = 35$. Let $\mathrm{ST}(x_k)$ and $\mathrm{ET}(x_k)$ be defined as the start and end time of the shift scheduled on day k; then the constraint can be specified as:

$$x_k = \mathrm{DO} \quad \Rightarrow \quad \mathrm{ST}(x_{k+1}) \geq \mathrm{ET}(x_{k-1}) + T \tag{5.22}$$

$$x_k = \mathrm{F} \quad \Rightarrow \quad \mathrm{ST}(x_{k+1}) < \mathrm{ET}(x_{k-1}) + T \tag{5.23}$$

The first constraints can be converted into the following domain propagation mechanisms:

$$x_k = \mathrm{DO} \wedge \mathrm{ST}(r) - T < \min_{l \in D(x_{k-1})} \mathrm{ET}(l) \Rightarrow x_{k+1} \neq r \quad \forall\, r \in D(x_{k+1}), \tag{5.24}$$

$$x_k = \mathrm{DO} \wedge \max_{r \in D(x_{k+1})} \mathrm{ST}(r) - T < \mathrm{ET}(l) \Rightarrow x_{k-1} \neq l \quad \forall\, l \in D(x_{k-1}), \tag{5.25}$$

$$\max_{r \in D(x_{k+1})} \mathrm{ST}(r) - T < \min_{l \in D(x_{k-1})} \mathrm{ET}(l) \Rightarrow x_k \neq \mathrm{DO}. \tag{5.26}$$

The second constraint (5.23) can in a similar way be converted into:

$$x_k = \mathrm{F} \wedge \mathrm{ST}(r) - T \geq \min_{l \in D(x_{k-1})} \mathrm{ET}(l) \quad \Rightarrow \quad x_{k+1} \neq r \quad \forall\, r \in D(x_{k+1}), \tag{5.27}$$

$$x_k = \mathrm{F} \wedge \max_{r \in D(x_{k+1})} \mathrm{ST}(r) - T \geq \mathrm{ET}(l) \quad \Rightarrow \quad x_{k-1} \neq l \quad \forall\, l \in D(x_{k-1}), \tag{5.28}$$

$$\max_{r \in D(x_{k+1})} \mathrm{ST}(r) - T \geq \min_{l \in D(x_{k-1})} \mathrm{ET}(l) \quad \Rightarrow \quad x_k \neq \mathrm{F}. \tag{5.29}$$

Keeping track of the smallest and largest start and end time is performed in a similar fashion as in case of the minimal between shift constraint.

Computational comparison

The domain propagation mechanism for the day off / free day constraint given in equations (5.24) – (5.29) were implemented through the C++ interface to the solver without any use of the built in constraint types (C++). As a comparison the constraint (5.22) and (5.23) were added directly with the use of the built in constraint types "IloTable" and "IloIfThen" (ILOG).

The comparison of the two methods were performed on all the same CP sub-problems as used for comparing the mechanisms of the minimal time between constraint. The propagation mechanisms used were the one chosen for the final algorithm, except that the given mechanism were used for the day off / free day constraint. The results of the comparison are summarised in Table 5.4. The fourth and fifth columns are the average time spent solving sub-problems for the given instance and method. The results shows that the C++ method uses around 42 % less computation time than the ILOG method. The ILOG method is outperformed on all instances and it is using at least 26 % more time on any instance. One reason for the poor performance of the ILOG method is the necessity of including some extra variables when using the built in constraints. The C++ method is of course selected to be used in the final algorithm.

5.3.4 Minimum consecutive workdays (C_2)

This constraint should ensure that there are at least two consecutive workdays, that is patterns like "free day – work day –free day" are forbidden. Here a free day is meant to cover all three types of non-working shifts. Let WS be the set of all shifts that have to be

Table 5.4: Comparison of domain propagation mechanisms for the day off / free day constraint on CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	C++	ILOG	ILOG
		Average	In seconds		Dev. in %[a] from C++
1	20	310.0	20.90	26.71	27.79
2	20	9,732.0	716.85	1,097.49	53.10
3	20	320.0	29.64	44.34	49.58
4	28	744.8	115.83	146.88	26.80
5	28	288.4	15.72	21.44	36.35
6	28	453.6	23.18	29.43	27.00
7	28	392.0	10.17	14.31	40.78
8	28	756.0	61.14	77.36	26.54
9	28	364.0	9.60	13.02	35.65
10	38	349.6	12.20	18.27	49.68
11	38	345.8	10.29	15.55	51.17
12	38	395.2	11.74	18.14	54.47
13	38	433.2	15.42	22.88	48.39
14	38	463.6	9.22	13.19	43.07
15	38	456.0	7.98	12.77	60.05
Average:					42.03

[a] Positive deviation means longer computation times.

considered as working shifts in this constraint, then the constraint can be stated as:

$$x_k \in \mathrm{WS} \quad \Rightarrow \quad x_{k+1} \in \mathrm{WS} \quad \text{or} \quad x_{k-1} \in \mathrm{WS}. \tag{5.30}$$

This constraint is transformed into the following domain propagation mechanisms, which are checked when the domain of a variable is either reduced to be a subset of WS or when it is reduced to contain no shifts from WS:

$$x_k \in \mathrm{WS} \quad \text{and} \quad x_{k+1} \notin \mathrm{WS} \quad \Rightarrow \quad x_{k-1} \in \mathrm{WS}, \tag{5.31}$$
$$x_k \in \mathrm{WS} \quad \text{and} \quad x_{k-1} \notin \mathrm{WS} \quad \Rightarrow \quad x_{k+1} \in \mathrm{WS}, \tag{5.32}$$
$$x_{k+1} \notin \mathrm{WS} \quad \text{and} \quad x_{k-1} \notin \mathrm{WS} \quad \Rightarrow \quad x_k \notin \mathrm{WS}. \tag{5.33}$$

Computational comparison

The mechanisms given in equations (5.31) – (5.33) (C++) are compared to a method that only uses the built-in constraints to implement constraint (5.30) (ILOG). The same sub-problems as used

for comparing the mechanisms for the minimal time between shifts constraint are used for comparing the two methods. The algorithm used for solving the instances is the final algorithm, except that the method used for the minimum consecutive workdays constraint is the ones given.

The results are summarised in Table 5.5, and shows that the C++ method on all instances uses slightly less computation time than the ILOG method. The ILOG method uses on average 0.82 % more computation time than the C++ method. Thus, the C++ is chosen to be used in the final algorithm.

Table 5.5: Comparison of domain propagation mechanisms for the minimum consecutive workdays constraint on CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	C++	ILOG	ILOG
		Average	In seconds		Dev. in %[a] from C++
1	20	310.0	20.90	21.06	0.77
2	20	9,732.0	716.85	723.54	0.93
3	20	320.0	29.64	30.09	1.50
4	28	744.8	115.83	116.17	0.29
5	28	288.4	15.72	15.93	1.29
6	28	453.6	23.18	23.29	0.50
7	28	392.0	10.17	10.24	0.77
8	28	756.0	61.14	61.79	1.07
9	28	364.0	9.60	9.67	0.72
10	38	349.6	12.20	12.33	1.03
11	38	345.8	10.29	10.39	1.03
12	38	395.2	11.74	11.85	0.91
13	38	433.2	15.42	15.52	0.66
14	38	463.6	9.22	9.29	0.70
15	38	456.0	7.98	8.04	0.70
Average:					0.86

[a] Positive deviation means longer computation times.

5.3.5 Maximum consecutive workdays (C_{10})

Depending on the contractual agreement of the nurse, this constraint is defined in two different ways. The standard constraint is that the nurse should have a maximum of 6 days between two days off (holiday shifts are also considered as days off, but free shifts are not). Let H be the holiday shift, DO the day off shift and F the free shift. The

constraint can be stated as:

$$\sum_{k=m}^{m+6} 1_{\left\{x_k \notin \{\mathrm{H,DO}\}\right\}} \leq 6 \quad \forall\, m = -5, \ldots, M-6, \tag{5.34}$$

where $x_{-5}, \ldots, x_0$ are predetermined by the schedule from the previous planning period. The function $1_{\{a\}}$ is the indicator function for the condition a (One if true, zero if false).

If the nurse has accepted, the constraint can be loosened such that the nurse can work up to 7 days between two days off, if at least one of the days in between is a day with the free shift. If there is no free shift in between the maximum is still 6 days. This constraint can be stated as:

$$\sum_{k=m}^{m+7} 1_{\left\{x_k \notin \{\mathrm{H,DO}\}\right\}} \leq 7 \quad \forall\, m = -6, \ldots, M-7, \tag{5.35}$$

$$\sum_{k=m}^{m+6} 1_{\left\{x_k \notin \{\mathrm{H,DO,F}\}\right\}} \leq 6 \quad \forall\, m = -5, \ldots, M-6. \tag{5.36}$$

To handle these different but quite similar constraints, a domain propagation mechanism handling the following constraint has been implemented:

$$\sum_{k=m}^{m+\beta} 1_{\{x_k \leq \alpha\}} \leq \beta \quad \forall\, m = \delta_1, \ldots, \delta_2, \tag{5.37}$$

where α, β, δ_1 and δ_2 are given integer constants. Due to the used ordering of the shifts in the domain of the x_k variables this constraint covers the constraints (5.35), (5.36) and (5.37).

The implemented domain propagation mechanism keeps a counter for the minimal value of the left side of each constraint. This counter is increased each time, one of the indicator functions are proven to be true. If the counter equals β, the variable that has not yet been proven to be less or equal α, has its domain reduced to values larger than or equal to $\alpha + 1$.

Computational comparison

The method that uses the implementation of (5.37) for the maximum consecutive workdays constraint is denoted by C++. A method which relies on only built-in constraint types was created as a comparison, and is denoted as ILOG. The built-in method is based on the "IloTable" constraint to calculate the value of the indicator functions of

the constraints (5.34)–(5.36). When a nurse is to have a maximum of 6 days between two days off, an extra variable for each day is needed when using the built-in constraints types, and when the maximum is 7 days, two variables are needed for each day.

The same CP sub-problems as used for comparing the methods for the minimal time between shifts constraint are used for comparing the C++ and ILOG methods. The algorithm used for the CP sub-problems is the final algorithm, expect that the given method is used for the maximum consecutive workdays constraint. Table 5.6 summarises the results of the comparison, and shows that the C++ on average uses 8.31 % less computation time.

Table 5.6: Comparison of domain propagation mechanisms for the maximum consecutive workdays constraint on CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	C++	ILOG	ILOG
		Average	In seconds		Dev. in %[a] from C++
1	20	310.0	20.90	22.93	9.71
2	20	9,732.0	716.85	781.40	9.01
3	20	320.0	29.64	32.44	9.43
4	28	744.8	115.83	123.91	6.97
5	28	288.4	15.72	17.15	9.07
6	28	453.6	23.18	25.17	8.58
7	28	392.0	10.17	10.90	7.25
8	28	756.0	61.14	65.51	7.16
9	28	364.0	9.60	10.27	6.94
10	38	349.6	12.20	13.22	8.30
11	38	345.8	10.29	11.17	8.58
12	38	395.2	11.74	12.84	9.32
13	38	433.2	15.42	16.85	9.33
14	38	463.6	9.22	9.95	7.93
15	38	456.0	7.98	8.54	7.05
Average:					8.31

[a] Positive deviation means longer computation times.

5.3.6 Shift type limits (C_4)

The number of evening shifts (E), night shifts (L) and long night shifts (LN) are restricted to be in a certain range. The limits on night and long night shifts are on the sum of the number of night and long night shifts during the planning period.

Let $\delta_1 < \delta_2$ be two day indexes, let A be a set of shifts and let B be a variable indicating the number of shifts from the set A scheduled in the planning period. The following constraint generalises the above constraints:

$$\sum_{k=\delta_1}^{\delta_2} 1_{\{x_k \in A\}} = B. \tag{5.38}$$

Domain propagation mechanisms for this constraint have been implemented using two counters, one for the number of days that have been proven to be in A and one for the number of days still containing shifts from A. That is:

$$G_1 = \sum_{k=\delta_1}^{\delta_2} 1_{\{D(x_k) \subseteq A\}}, \tag{5.39}$$

$$G_0 = (\delta_2 - \delta_1 + 1) - \sum_{k=\delta_1}^{\delta_2} 1_{\{D(x_k) \cap A = \emptyset\}}. \tag{5.40}$$

Both counters are updated when the relevant part of the domains of the x_k variables changes. When either G_1 is increased or when the upper bound of B is decreased such that G_1 becomes equal to the upper bound on B, then all variables not already proven to be in A have all shifts in A removed from their domains, that is:

$$G_1 = \max B \ \wedge \ D(x_k) \cap A \neq \emptyset \\ \Rightarrow \ D(x_k) \cap A = \emptyset \quad \forall\, k = \delta_1, \ldots, \delta_2, \tag{5.41}$$

where setting $D(x_k) \cap A$ to the value of $\emptyset$ should be understood as that all shifts in A are removed from the domain of x_k.

Similar when G_0 is decreased or the lower bound of B is increased such that G_0 becomes equal to the lower bound, then all variables not proven not to be in A have all shifts not in A removed from their domains:

$$G_0 = \min B \ \wedge \ D(x_k) \cap A^c \neq \emptyset \\ \Rightarrow \ D(x_k) \subseteq A \quad \forall\, k = \delta_1, \ldots, \delta_2, \tag{5.42}$$

where $D(x_k) \subseteq A$ should be understood such that all shifts in the domain of x_k that are not in A are removed from the domain of x_k.

The limits could have been handled with a simpler constraint, that is constraint (5.38) but where A is restricted to a single shift. The constraint on night and long night shifts could then be handled

with the following constraints:

$$\sum_{k=\delta_1}^{\delta_2} 1_{\{x_k=\mathrm{N}\}} = B_{\mathrm{N}}, \tag{5.43}$$

$$\sum_{k=\delta_1}^{\delta_2} 1_{\{x_k=\mathrm{LN}\}} = B_{\mathrm{LN}}, \tag{5.44}$$

$$B_{\mathrm{N}} + B_{\mathrm{N}} = B. \tag{5.45}$$

This would be valid but could in some cases result in that some domain reductions would be performed later in the search and thus leading to longer computation time.

Computational comparison

The two methods proposed for handling the shift type limits are compared to two methods using the built-in constraints of the ILOG solver. The method using the domain propagation mechanisms of equation (5.41) and (5.42) is denoted by C++ and the method using constraints (5.43) – (5.45) is denoted by C++B. As a comparison a method based on constraint (5.38) using the built-in functions "IloTable" and "IloSum" was created. The method is denoted by ILOG and it has the same strength as the C++ method for reducing domains of the variables. Another method which has the same strength as the C++B method was also implemented; this method uses the "IloDistribute" function and it is denoted by ILOG-B. The "IloDistribute" function cannot handle the propagation mechanisms for constraint (5.38), thus the implementation is based on implementing the constraints (5.43) – (5.45).

The CP sub-problem instances that are used for testing are those already used for comparing the methods for the minimal time between shifts constraint. The propagation mechanisms used for all expect the shift type limits constraints are those of the final algorithm. Table 5.7 summarises the results of the comparison. The C++B is outperformed by the C++ method, as it spends on average 2.83 % more computational time. Yet for three of the instances, the C++B method is very close to C++ using less than 1 % more time. The ILOG method spends on three instances almost the same time as the C++ method, but on the rest it requires at least 4.5 % more computation time; on average it spends 6.16 % more computation time. On average the ILOG-B method is better than the ILOG method, but on four of the instances the ILOG method is significantly faster. One reason for the inferior performance of the ILOG method is that the

implementation requires a large set of extra variables, which are not needed for the ILOG-B method. As the C++ method outperforms the others on all except one instance where the methods spend almost the same computation time, the C++ method is used for the final algorithm.

Table 5.7: Comparison of domain propagation mechanisms for the shift type limits constraint on CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	C++	C++B	ILOG	ILOG-B
		Average	In seconds	Deviation in %[a] from C++		
1	20	310.0	20.90	0.75	4.58	4.07
2	20	9,732.0	716.85	7.68	7.20	10.48
3	20	320.0	29.64	0.93	5.51	3.18
4	28	744.8	115.83	3.05	−0.41	4.83
5	28	288.4	15.72	3.73	0.53	5.33
6	28	453.6	23.18	4.40	0.51	5.66
7	28	392.0	10.17	2.01	7.03	4.84
8	28	756.0	61.14	1.82	6.94	3.58
9	28	364.0	9.60	3.20	7.92	4.35
10	38	349.6	12.20	2.95	9.69	6.67
11	38	345.8	10.29	3.78	9.64	7.29
12	38	395.2	11.74	1.68	9.22	5.16
13	38	433.2	15.42	3.71	8.35	7.45
14	38	463.6	9.22	0.84	6.99	4.05
15	38	456.0	7.98	1.95	8.74	4.66
Average:				2.83	6.16	5.44

[a] Positive deviation means longer computation times.

5.3.7 Office days (C_5)

Some nurses are required to have some office days during the planning period. These are not fixed to specific days, but given as a demand during a given period, for example: two days during the planning period, or one day every second week. Let δ_1 and δ_2 be the two end days for the period in which the office days should be placed and let b be the number of office days to be scheduled in that period, then the constraint can be stated as:

$$\sum_{k=\delta_1}^{\delta_2} 1_{\{x_k = \mathrm{O}\}} = b \tag{5.46}$$

This constraint could be handled with the same propagation mechanisms as used for the constraint (C_4) on shift type limits, which were described in Section 5.3.6. A special version of the propagation mechanisms used for the shift type limits was however implemented. This version only handles constraints, where B is a constant and A is a single shift type. The propagation mechanisms are the same, but the implementation is simpler.

Computational comparison

The mechanisms described above are compared to a method that exclusively based on the built-in constraints. The method using only built-in constraints is denoted by ILOG and is based on the "IloTable" constraint. The mechanism that uses the special version of the implementation is denoted by C++1 and the one that uses the mechanism described in Section 5.3.6 is denoted by C++2.

Table 5.8: Comparison of domain propagation mechanisms for the office days constraint on CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Office days	Sub-prob.	C++1	ILOG	C++2
		Average	Average	In seconds	Dev. in %[a] from C++1	
1	20	1	31.0	1.16	1.07	0.23
4	28	1.25	106.4	58.41	3.93	0.90
5	28	1.86	72.1	8.50	8.09	4.85
6	28	2	113.4	20.51	8.66	4.75
7	28	1	28.0	1.44	7.04	4.07
8	28	1	81.0	24.79	5.16	2.97
9	28	1	26.0	2.48	4.91	1.92
13	38	1	11.4	0.79	4.84	1.92
14	38	1	48.8	3.84	4.71	2.13
15	38	1	48.0	1.72	4.41	2.04
Average:					5.28	2.58

[a] Positive deviation means longer computation times.

The CP sub-problems are the same as the ones used for comparing the methods for the minimal time between shifts constraint, except that only instances where office days had to be planned were included. Table 5.8 summarises the results. The third column is the average number of office days that had to be planned in each CP sub-problem; the fourth column is the number of CP sub-problems solved on average over the 10 runs for the given nurse rostering instance. The results show that the ILOG method uses an average of

5.28 % more time than the C++1 method, whereas the C++2 method only uses an average of 2.58 % more time. Again the poor performance of the ILOG method is probably due to the fact that some extra variables are required to implement the constraint using only built-in constraint types.

5.3.8 Maximum work hours per week (C_3)

The restriction on the maximum working hours can be stated as the following constraint for each week in the planning period. Let δ_1 and δ_2 be the first and last day of the week respectively; let $\mathrm{WH}(x_k)$ be the number of work hours worked if shift x_k is assigned to day k and let MWH be the maximum working hours of that week:

$$\sum_{k=\delta_1}^{\delta_2} \mathrm{WH}(x_k) \leq \mathrm{MWH}. \tag{5.47}$$

Instead of creating propagation mechanisms for this constraint type directly, it was generalised such that it also can be used for the constraint on recorded hours (C_7), as well, The generalised constraint can be expressed as:

$$\sum_{k=\delta_1}^{\delta_2} A_k(x_k) = B, \tag{5.48}$$

where A_k are vectors of numbers and B is a variable.

The propagation mechanisms that removes values from the domain of the x_k variables are as follows:

$$\min B - \sum_{\substack{k=\delta_1 \\ k\neq m}}^{\delta_2} \max_{l\in D(x_k)} A_k(l) > A_m(r) \quad\Rightarrow\quad x_m \neq r, \tag{5.49}$$

$$\max B - \sum_{\substack{k=\delta_1 \\ k\neq m}}^{\delta_2} \min_{l\in D(x_k)} A_k(l) < A_m(r) \quad\Rightarrow\quad x_m \neq r \tag{5.50}$$

$$\forall\, r \in D(x_m)\; \forall\, m = \delta_1, \ldots, \delta_2.$$

It is required to keep track of the inequality's left side. Since a change in a domain would lead to an update of $2(\delta_2 - \delta_1)$ of the left sides (two for each m except for $k = m$), the mechanisms were instead

transformed to the following:

$$\min B - \sum_{k=\delta_1}^{\delta_2} \max_{l \in D(x_k)} A_k(l) + \max_{l \in D(x_m)} A_m(l) > A_m(r) \;\Rightarrow\; x_m \neq r, \quad (5.51)$$

$$\max B - \sum_{k=\delta_1}^{\delta_2} \min_{l \in D(x_k)} A_k(l) + \min_{l \in D(x_m)} A_m(l) < A_m(r) \;\Rightarrow\; x_m \neq r \quad (5.52)$$

$$\forall\, r \in D(x_m)\; \forall\, m = \delta_1, \ldots, \delta_2.$$

It should be noted that the second term of both inequalities is independent of m and hence there are now only two more terms to keep track of than with the other mechanisms. With these mechanisms only up to four terms would have to be updated when a domain of a variable changes.

In order to avoid a lot of unnecessary checks of the inequalities, some extra conditions are checked. As long as these conditions are not met, only updates of the different terms are calculated. The following constant is calculated at the root node of the CP-search:

$$\text{maxDiff} = \max_{k=\delta_1,\ldots,\delta_2} \Big\{ \max_{l \in D(x_k)} A_k(l) - \min_{l \in D(x_k)} A_k(l) \Big\}. \quad (5.53)$$

The condition that has to be fulfilled before any propagation mechanisms (5.51) are checked is:

$$\min B - \sum_{k=\delta_1}^{\delta_2} \max_{l \in D(x_k)} A_k(l) > -\text{maxDiff}. \quad (5.54)$$

If the above condition is true, the following condition is checked for each m and only if it is true, mechanism (5.51) is checked for the corresponding m.

$$\min B - \sum_{k=\delta_1}^{\delta_2} \max_{l \in D(x_k)} A_k(l) > -\Big(\max_{l \in D(x_m)} A_m(l) - \min_{l \in D(x_m)} A_m(l) \Big). \quad (5.55)$$

Similar, the conditions for propagation mechanism (5.52) are:

$$\max B - \sum_{k=\delta_1}^{\delta_2} \min_{l \in D(x_k)} A_k(l) < \text{maxDiff}. \quad (5.56)$$

$$\max B - \sum_{k=\delta_1}^{\delta_2} \min_{l \in D(x_k)} A_k(l) < \max_{l \in D(x_m)} A_m(l) - \min_{l \in D(x_m)} A_m(l). \quad (5.57)$$

The mechanisms to remove values from the domain of the B variable are:

$$B \leq \sum_{k=\delta_1}^{\delta_2} \max_{l \in D(x_k)} A_k(l), \tag{5.58}$$

$$B \geq \sum_{k=\delta_1}^{\delta_2} \min_{l \in D(x_k)} A_k(l). \tag{5.59}$$

Here, this should be understood the way, that all values in the domain of B not obeying the above constraints are removed. The right side of both constraints are the same as the ones used for the mechanisms to remove values from the x_k variables, so no extra terms to keep track of them are required.

Computational comparison

The propagation described above is compared to two different methods using only built-in constraints. The first method is denoted by ILOG1 and is based on the "IloDistribute" and The "IloScalProd" constraints. The second method is based on the "IloTable" and the "IloSum" constraint and is denoted by ILOG2. The ILOG2 method is based on the same idea as the propagation mechanism for C++. The ILOG1 method is on the other hand based on getting bounds on the number of times each type of work shift can be assigned during the week. The numbers are then multiplied by the number of work hours for the corresponding shifts; the sum of these is then restricted to be less than the maximal number of work hours during a week. The strength of the propagation mechanism for ILOG1 is less than ILOG2, but ILOG2 requires more extra variables than ILOG1.

The CP sub-problem instances used for comparison are the same as the ones used for comparing the methods for the minimal time between shifts constraint. The propagation mechanisms used for all other constraints than the maximum work hours per week constraint, are those used in the final algorithm. Table 5.9 summarises the results of the comparison; the ILOG2 method clearly outperforms the ILOG1 method, only on a single instance the ILOG1 method uses slightly less time than the ILOG2 method. On seven of the instances, the ILOG2 and C++ methods spend almost the same computation time; on the other eight instances the ILOG2 method spends between 2.10 % and 4.40 % more computation time. Thus the C++ method is selected for the final algorithm.

Table 5.9: Comparison of domain propagation mechanisms for the maximum work hours per week constraint on CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	C++	ILOG1	ILOG2
		Average	In seconds	Dev. in %[a] from C++	
1	20	310.0	20.90	3.21	0.12
2	20	9,732.0	716.85	3.77	0.10
3	20	320.0	29.64	4.15	−0.68
4	28	744.8	115.83	3.86	−0.28
5	28	288.4	15.72	3.85	0.64
6	28	453.6	23.18	4.41	−0.44
7	28	392.0	10.17	3.24	2.73
8	28	756.0	61.14	2.95	−0.43
9	28	364.0	9.60	3.56	2.34
10	38	349.6	12.20	3.99	2.23
11	38	345.8	10.29	4.17	2.67
12	38	395.2	11.74	4.06	2.37
13	38	433.2	15.42	4.11	2.10
14	38	463.6	9.22	3.84	3.70
15	38	456.0	7.98	3.84	4.40
Average:				3.80	1.44

[a] Positive deviation means longer computation times.

5.3.9 Recorded work hours (C_7)

The constraint on the number of recorded work hours should besides handling the constraint, also make it possible to handle the objective O_5.

If the planning period does not include the end of a nurse's twelve weeks contract period, then the constraint can be handled with a single constraint of the type described in Section 5.3.8 for the constraint on maximal working hours. Let δ_1 and δ_3 respectively be the first and last day in the planning period. Let $\mathrm{RH}_k(x_k)$ be the recorded hours of day k and let B be an integer variable that has been limited to the allowed range around the contracted number of hours for the given nurse. Then the constraint can be stated as:

$$\sum_{k=\delta_1}^{\delta_3} \mathrm{RH}_k(x_k) = B. \tag{5.60}$$

If the planning period includes the end of a nurse's twelve weeks contract period, then the constraint also needs to handle some variables that indicate, how much the recorded hours deviate from the

contracted number of hours. Let CRH be the contracted number of hours for the twelve weeks period minus the number of recorded hours worked before the current planning period and let δ_2 be the last day in the twelve weeks contract period. B_{Actual} is a variable that takes the value of the number of recorded hours from the beginning of planning period up until the last day in the twelve weeks contract period. B_{below} and B_{above} are variables that indicate how many hours the actual number of recorded hours are below or above the contracted number of hours. The rest of the planning period is constrained by the same way as for the case when the planning period does not include the last day of the twelve weeks period:

$$\sum_{k=\delta_1}^{\delta_2} \text{RH}_k(x_k) = B_{\text{Actual}}, \tag{5.61}$$

$$\sum_{k=\delta_2+1}^{\delta_3} \text{RH}_k(x_k) = B, \tag{5.62}$$

$$\text{CRH} = B_{\text{Actual}} + B_{\text{Below}} - B_{\text{Above}}. \tag{5.63}$$

$$B_{\text{Actual}} \geq 0 \tag{5.64}$$

$$B_{\text{Below}} \geq 0 \tag{5.65}$$

The first two constraints are handled by the constraint propagation mechanisms described in Section 5.3.8 and the third is handled with the standard built-in mechanisms which are similar to the mechanisms described in Section 2.2.2.

Computational comparison

A comparison between the above method (C++) and one that is based only on built-in constraints (ILOG) is described in this section. The difference between the methods is that in the C++ method the propagation mechanisms from Section 5.3.8 are used for equations (5.60), (5.61) and (5.62), whereas the ILOG method is based on the "IloTable" and the "IloSum" constraints. The ILOG method is the same as the one denoted by ILOG2 in the computational comparison part of Section 5.3.8.

The CP sub-problem instances used for comparing the methods are the same as for comparison of the methods for the minimal time between shifts constraint. The propagation mechanisms for all other constraints are the ones used in the final algorithm.

The results of the tests are listed in Table 5.10 which and shows that the ILOG method uses an average of 5.21 % more computation time than the C++ method.

Table 5.10: Comparison of domain propagation mechanisms for the recorded work hours constraint on CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	C++	ILOG	ILOG
		Average	In seconds		Dev. in %[a] from C++
1	20	310.0	20.90	21.90	4.78
2	20	9,732.0	716.85	739.98	3.23
3	20	320.0	29.64	30.51	2.93
4	28	744.8	115.83	121.85	5.20
5	28	288.4	15.72	16.48	4.77
6	28	453.6	23.18	23.87	2.98
7	28	392.0	10.17	10.88	7.04
8	28	756.0	61.14	63.33	3.59
9	28	364.0	9.60	10.26	6.83
10	38	349.6	12.20	12.86	5.42
11	38	345.8	10.29	10.89	5.86
12	38	395.2	11.74	12.40	5.56
13	38	433.2	15.42	16.17	4.86
14	38	463.6	9.22	9.80	6.33
15	38	456.0	7.98	8.68	8.73
Average:					5.21

[a] Positive deviation means longer computation times.

5.3.10 Complete weekends (C_{11})

Working a complete weekend at this particular ward means that the nurse works two shifts of the same type during a weekend. The shifts that are defined as weekend shifts consist of exactly two of each type. Let *SatSun* and *SunMon* be the set of shifts that are have to be worked on Saturday and Sunday and on Sunday and Monday, respectively.

The constraint can for all days k that are Sundays be formulated as:

$$1_{\{x_{k-1}=r\}} = 1_{\{x_k=r\}} \quad \forall r \in \mathit{SatSun} \tag{5.66}$$

$$1_{\{x_k=r\}} = 1_{\{x_{k+1}=r\}} \quad \forall r \in \mathit{SunMon}. \tag{5.67}$$

The implemented domain propagation mechanisms are for the generalised constraint:

$$1_{\{x_k=r\}} = 1_{\{x_m=l\}}, \tag{5.68}$$

where k and m are two days and r and l are the two shifts which should either both be worked or none of them. The domain propa-

gation mechanisms of this constraint are then:

$$\begin{aligned} x_k = r &\Rightarrow x_m = l, \\ x_k \neq r &\Rightarrow x_m \neq l, \\ x_m = l &\Rightarrow x_k = r, \\ x_m \neq l &\Rightarrow x_k \neq r. \end{aligned} \tag{5.69}$$

The above propagation mechanisms are sufficient to ensure that the constraints are fulfilled. To create a more extensive propagation mechanism, the following implication can be used for all days k that are Sundays.

$$\begin{aligned} 1_{\{x_{k-1} \in SatSun\}} &\Rightarrow x_k \in SatSun \\ 1_{\{x_k \in SatSun\}} &\Rightarrow x_{k-1} \in SatSun \\ 1_{\{x_{k+1} \in SunMon\}} &\Rightarrow x_k \in SunMon \\ 1_{\{x_k \in SunMon\}} &\Rightarrow x_{k+1} \in SunMon \end{aligned} \tag{5.70}$$

The mechanisms described in equation (5.70) are not used in the final algorithm, as they did not improve the overall computation time. The section below depicts a computational comparison of the performance of the additional propagation mechanisms (5.70).

Computational comparison

The two propagation mechanisms described above are compared to a method, which uses only the built-in constraint types. This method is based on the "IloAbstraction" constraint and denoted by ILOG. The method that implements constraints (5.66) and (5.67) with the use of the propagation mechanisms of equation (5.69) is denoted by C++1. To test the mechanisms of equation (5.70), a further method (C++2) was included. The method is similar to C++1, but additionally uses the mechanisms of equation (5.70).

The comparison of the two methods were performed on the same CP sub-problems used for comparing the mechanisms of the minimal time between shifts constraint.

As with the minimal time between shifts constraint, the set of incompatible pairs of shifts on consecutive days that can be deduced from the complete weekend constraint are used in the method for calculating the reduced cost. For a direct comparison of this method, the reduced cost calculation is restrained from using this set of incompatible pairs. This seems to be a more fair comparison. Later in this section, another comparison for selecting one of these methods for

the final algorithm is presented. The propagation mechanisms used for all other constraints were the ones chosen for the final algorithm.

Table 5.11 summarises the results of the comparison. The C++2 method outperforms both the C++1 and ILOG method. The C++1 and ILOG methods use almost the same computation time, on four instances the ILOG method is slightly faster, whereas otherwise the C++1 method uses less computation time. On average, the ILOG method uses 0.51 % more time than the C++1 method.

Table 5.11: Comparison of domain propagation mechanisms for the complete weekend constraint on CP sub-problems for 14 days instances in 10 runs. Deduced information from the complete weekend constraint is not used for calculating the reduced cost of the new column.

Ins.	Nurses	Sub-prob.	C++1	C++2	ILOG
		Average	In seconds	Dev. in %[a] from C++1	
1	20	310.0	26.45	−0.02	1.72
2	20	9,732.0	958.27	−0.97	0.07
3	20	320.0	49.97	−3.33	−1.27
4	28	744.8	141.79	−0.94	1.33
5	28	288.4	21.61	−1.30	0.54
6	28	453.6	25.33	−0.94	0.16
7	28	392.0	12.80	−2.93	−1.73
8	28	756.0	78.45	−0.96	−0.19
9	28	364.0	17.56	−0.95	2.13
10	38	349.6	14.37	−0.84	1.13
11	38	345.8	12.76	0.09	1.19
12	38	395.2	14.42	−0.28	1.06
13	38	433.2	20.09	−2.13	−0.22
14	38	463.6	13.95	−0.75	1.17
15	38	456.0	12.07	−1.11	0.77
Average:				−1.16	0.52

[a] Positive deviation means longer computation times.

For testing which method should be used in the final CP algorithm, a comparison was made using the full CP algorithm, except that the given method was used for the complete weekend constraint.

The reduced cost calculation of the CP algorithm are described in Section 5.3.13 and 5.3.14. It deduces a set of incompatible shifts on consecutive days from the complete weekend constraint, which is used for speeding up the calculations. As the set of incompatible shifts can fully describe the constraint and the reduced cost calculation ensures that no pair of incompatible shifts is selected, no special propagation

mechanism is actually needed. Let “None” denote the method where no extra domain reduction mechanism is used.

The results of the comparisons are summarised in Table 5.12. They show that the C++1 method uses the least computation time on all except one instance. On this instance the C++1, uses almost the same time as the one that uses the least computation time. As opposed for the minimal time between shift constraint, the “None” method performs worse than the other methods. The performance is probably due to the fact that the computational effort required in the C++1 method is very low and that the inclusion of the C++1 domain reduction mechanisms ensure that the reduced cost algorithm has to be recalculated fewer times. As opposed to the comparison in Table 5.11, the C++2 uses more computation time than the C++1 method. A reason for this is that the extra domain reductions of the C++2 method are indirectly performed by the method for calculating the reduced cost. So the time spend for checking the extra mechanisms for C++2 is larger than the time saved with fewer recalculations of the reduced cost. As before, the ILOG method uses on average more computation time than the C++1 and C++2 method. Thus the C++1 method is chosen for the final CP algorithm.

Table 5.12: Comparison of domain propagation mechanisms for the complete weekend constraint on CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	C++1	C++2	ILOG	None
		Average	In seconds	Dev. in %[a] from C++1		
1	20	310.0	20.67	0.16	0.62	3.36
2	20	9,732.0	757.89	0.62	0.55	2.70
3	20	320.0	30.69	0.57	0.73	3.83
4	28	744.8	124.79	0.44	1.09	5.29
5	28	288.4	15.15	0.81	1.30	1.45
6	28	453.6	25.04	−0.21	−0.39	1.85
7	28	392.0	10.43	0.93	1.52	6.49
8	28	756.0	65.81	1.01	0.82	5.40
9	28	364.0	9.75	1.05	0.64	4.03
10	38	349.6	12.23	0.95	1.31	5.42
11	38	345.8	10.77	0.97	1.25	5.66
12	38	395.2	12.70	0.94	0.89	5.38
13	38	433.2	15.72	0.70	0.99	4.15
14	38	463.6	9.14	0.83	0.68	3.65
15	38	456.0	7.80	0.80	1.34	4.57
Average:				0.71	0.89	4.22

[a] Positive deviation means longer computation times.

5.3.11 Constraints on weekends (C_6, C_{11} and C_{12})

The weekends are more constrained than the weekdays. Finding infeasible combinations early is thus very important. For the given problem, there are three different constraints referring to weekends:

1. Work complete weekends (C_{11}).
2. Consecutive work weekends (C_6).
3. 12 hours weekend shifts (C_{12}).

The second and third constraint sets are not independently addressed but combined into a single, possibly stronger combined constraint, which should allow more domain reductions. The first constraint is treated by it self, and the mechanism has been presented earlier in Section 5.3.10.

The designed mechanism is a graph algorithm, where the feasible combinations of weekend shifts corresponds to a path in the graph.

Because a nurse has to work complete weekends (constraint C_{11}), a nurse may on a weekend either work two eight hours shifts, two twelve hours shifts or no work shift at all. Since all shifts during Sundays are weekend shifts, the variables that correspond to a Sunday are used to represent the weekends. The set of shifts of each weekend is divided into three sets: the first set is free shifts, the second is eight hours work shifts and the third is twelve hours work shifts. The domain propagation mechanism is called each time a domain for a Sunday is proven not to be in one of these sets. For each of these sets there is a set of corresponding arcs in the graph. The nodes of the graph correspond to the different states the nurse's schedule can attain. The state indicates if the nurse worked the previous weekend or not and which type of shift (8 hours or 12 hours) the nurse worked last time she had a work weekend. Figure 5.2 shows the states and the graph, where the node to the left is the state of the nurse in the previous planning period and the node to right and the arcs connected to it are artificial and represent the future.

Feasible combinations of shifts during the weekends correspond to a path connecting the two end nodes. Accordingly, any other node showing an in-degree or out-degree of zero can be removed. This is done using a forward and backward search through the graph removing all nodes of zero in-degree or out-degree together with their incident arcs.

When all arcs corresponding to one of the sets of shifts are removed, the corresponding domain values are removed. During the search, the domains of the weekend variables are reduced. When one

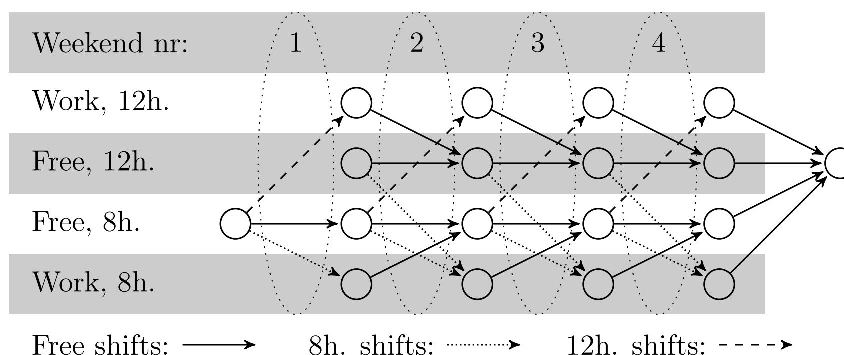

Figure 5.2: Graph of the domain propagation model for constraint (C_6) and (C_{12}).

of the sets is made empty, the corresponding arcs are removed and the forward and backward search is reapplied.

Computational comparison

The above graph algorithm (C++) is compared to a method where the constraints have been implemented using only built-in constraints (ILOG). The ILOG method handles the two constraints individually by means of the "IloTable" and the "IloIfThen" constraints.

The CP sub-problem instances used for testing are again those used for testing the methods for the minimal time between shifts constraint. The propagation mechanisms employed for all the other constraints are the ones used in the final algorithm. Table 5.13 summarises the results of the tests. On average the ILOG method uses 3.50 % more computation time than the C++ method, but on three instances it uses up to 1.39 % less computation time. The reason that the C++ does not show much lower computation time than the ILOG method is probably that the instances are two week instances so that the planning period only covers two weekends. The strength of the C++ method would probably be more visible for instances with a longer planning period.

5.3.12 Combining maximum work hours per week and minimum recorded work hours

Some initial tests of the algorithm led to the observation that time consuming and ineffective searches resulted when a number of the days were assigned as days with no work shift. In order to reduce this part of the search tree, an additional propagation mechanism

Table 5.13: Comparison of domain propagation mechanisms for consecutive work weekends and 12 hours weekend shifts constraints on CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	C++	ILOG	ILOG
		Average	In seconds		Dev. in %[a] from C++
1	20	310.0	20.90	20.75	−0.72
2	20	9,732.0	716.85	732.47	2.17
3	20	320.0	29.64	31.16	5.12
4	28	744.8	115.83	114.21	−1.39
5	28	288.4	15.72	15.81	0.51
6	28	453.6	23.18	23.46	1.23
7	28	392.0	10.17	10.73	5.51
8	28	756.0	61.14	62.89	2.85
9	28	364.0	9.60	9.60	−0.05
10	38	349.6	12.20	12.90	5.67
11	38	345.8	10.29	10.92	6.10
12	38	395.2	11.74	12.82	9.08
13	38	433.2	15.42	16.11	4.49
14	38	463.6	9.22	9.39	1.81
15	38	456.0	7.98	8.79	10.08
Average:					3.50

[a] Positive deviation means longer computation times.

was developed combining the constraint on maximum work hours per week (C_3) and the minimal number of recorded hours (C_7). After testing this extra propagation mechanism with the search strategy that showed the best performance for the final algorithm, this mechanism was however not included for the final algorithm.

The idea underlying the mechanism is to determine the maximal number of recorded hours that can be attained each week subject to the constraint on the maximum actual number of work hours. The sum of these upper bounds for each week is an upper bound for the complete planning period. If this upper bound falls below the minimum requirement, the current node of the CP-search tree is pruned.

Let δ_1 and δ_2 respectively denote the index of the first and last day in the week. Let MWH be the maximum number of work hours allowed per week, and $\mathrm{RH}_k(r)$ and $\mathrm{WH}_k(r)$ respectively denote the number of recorded and worked hours that result if shift r is assigned to day k. An upper bound on the number of recorded hours during

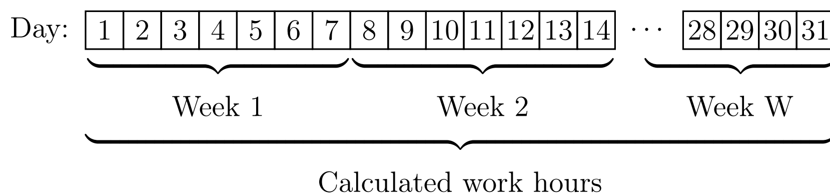

Figure 5.3: Illustration on which days the constraints depend.

a week is then obtained by solving the program:

$$\begin{aligned} \max \sum_{k=\delta_1}^{\delta_2} \mathrm{RH}_k(r_k) \\ \text{s.t.} \sum_{k=\delta_1}^{\delta_2} \mathrm{WH}_k(r_k) \leq \mathrm{MWH}\,, \\ r_k \in D(x_k) \text{ for } k = \delta_1, \ldots, \delta_2\,. \end{aligned} \tag{5.71}$$

The above program is equivalent to a multiple choice knapsack problem [34] and solved by the standard backward dynamic programming algorithm with some improvements. The algorithm starts by calculating the optimal choice for the last variable given an amount of remaining work hours. This optimal choice is calculated for all values between zero and MWH of remaining work hours. The optimal solution for the two last variables can then be calculated for each value of remaining work hours, and so forth. In order to avoid that these optimal solution values are calculated for values of remaining work hours that cannot be attained in any feasible solution; the algorithm starts with a forward search to find the possible values of remaining work hours at each state; the result of a forward search for a standard week is shown in Figure 5.4. This contributes to a substantial reduction in the computational effort required for solving the program (5.71).

A further substantial reduction in the size of the program (5.71) is obtained as follows. In a usual week without official holidays, most of the shifts during a day have both the same actual work and recorded work hours. Shifts that are equal in both parameters $\mathrm{RH}_k(r_k)$ and $\mathrm{WH}_k(r_k)$ can be combined into a single domain value. This does on average more than halve the size of the program (5.71).

When domain reductions occur, the problem for the corresponding week is re-optimised. Instead of starting from scratch, the dynamic programming algorithm starts with the previous optimal solu-

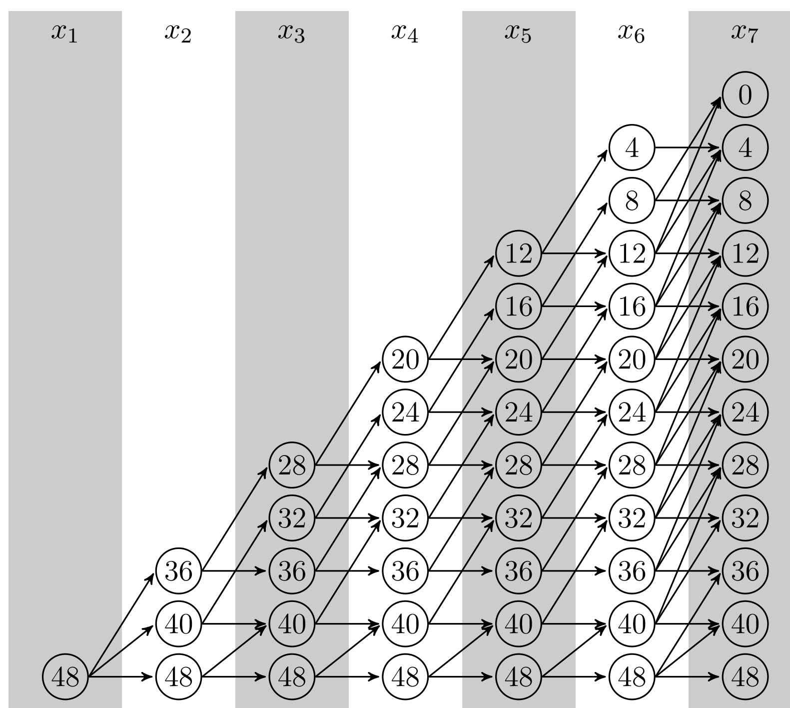

Figure 5.4: Feasible remaining work hours for each variables in a standard week.

tion, re-optimising nodes where either the optimal choice has been removed or where the value of the dependent node of the optimal choice has changed.

Computational comparison

As the above mechanism is an extra propagation mechanism that is not necessary for the model, it is tested against the CP model where it is not included. The CP sub-problems used for the tests are, as usual, the ones used for testing the methods for the minimal time between shifts constraint. The mechanisms employed for the constraints are the ones used in the final algorithm. The results are shown in Table 5.14, which indicates that the model where the mechanism is used spends on average 16.48 % more computation time than when it is not included.

If a different search strategy is used, in particular one based on selecting an unbound variable and fixing it to the smallest domain value first, this mechanism might be useful. Note that this is a standard way of doing branching in a CP model.

Table 5.14: Comparison of domain propagation mechanisms for the recorded work hours constraint on CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	Without	Included	Included
		Average	In seconds		Dev. in %[a] from C++
1	20	310.0	20.90	24.08	15.25
2	20	9,732.0	716.85	835.39	16.45
3	20	320.0	29.64	34.10	14.99
4	28	744.8	115.83	139.62	20.36
5	28	288.4	15.72	18.25	15.87
6	28	453.6	23.18	25.76	11.12
7	28	392.0	10.17	12.28	20.61
8	28	756.0	61.14	73.82	20.59
9	28	364.0	9.60	11.26	17.07
10	38	349.6	12.20	14.34	17.34
11	38	345.8	10.29	11.99	16.45
12	38	395.2	11.74	13.58	15.53
13	38	433.2	15.42	18.02	16.77
14	38	463.6	9.22	10.38	12.53
15	38	456.0	7.98	9.29	16.25
Average:					16.48

[a] Positive deviation means longer computation times.

5.3.13 Pricing the new column

Calculating the reduced cost of a given schedule is easy, but if the reduced cost is first calculated when all variables are fixed, then the constraint programming search would enumerate all feasible solutions to determine the one of least reduced cost. Therefore, instead of first calculating the reduced cost in the leaves of the search tree, a lower bound on the reduced cost is computed at every node of the search tree. This way, the search should be able to early prune a substantial number of branches.

The calculation of the lower bound is based on the domains of the variables x_k. Besides the domains of the variables in the given CP search tree node, only the following constraints are taken into account for computing the lower bound: The constraint C_1 on the minimum time span between shifts, the constraint C_{11} regarding complete work weekends, and some structural constraints on the combination of non-working shifts. These constraints restrict combinations of shifts on consecutive days and can thus be used for improving the lower bound.

The lower bound can be used when searching for any schedule with negative reduced cost, or when searching for an optimal sched-

ule, that is, one of smallest reduced cost. The lower bound calculation should include: the penalty cost of shifts (O_3), the penalty for connections of shifts on consecutive days (O_4), the direct and opportunity cost of diverging from the contracted number of work hours(O_5), the dual values from the coverage (5.3) and the overlap constraints (5.4) and the dual value for the constraint ensuring one schedule per nurse (5.2).

To determine the lower bound, a graph as shown in Figure 5.5 is created. The lower bound results as the length of a shortest path between its two end nodes. The graph shows a node for each shift type of each day, a single node representing the shift type of the last day in the previous planning period and a node representing the future. Each arc represents a combination of two shifts on two consecutive days. In case that such a combination is infeasible, the length of the corresponding arc is set to a sufficiently large value. Otherwise the arc length is obtained as follows.

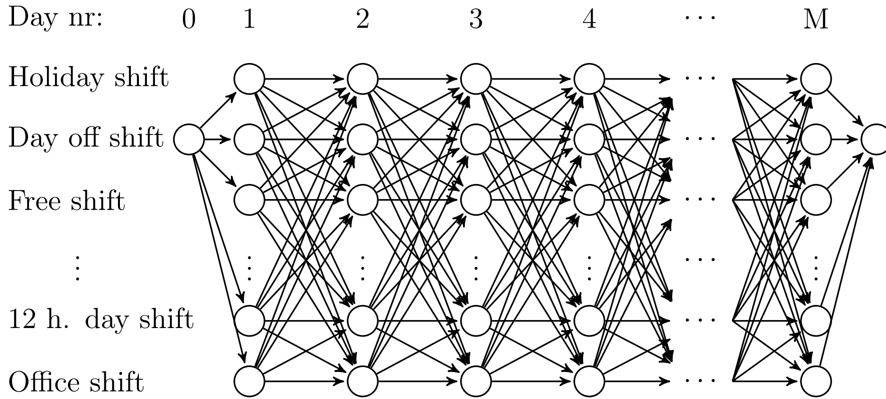

Figure 5.5: Graph for the shortest path algorithm.

Let $\vec{c}_n^{\sigma,\delta}$ be the preference cost (O_3) for nurse n working shift σ on day δ. Let $\vec{c}_n^{\sigma,\delta,\sigma',\delta'}$ be the preference cost (O_4) for nurse n working shift σ on day δ and shift σ' on day δ'.

Let $\overline{CC}_n^{\sigma,\delta}$ be the set of all coverage constraints $i \in CC$ for which nurse n contributes to its adherence if she works shift σ on day δ. Let moreover $\overline{CO}_n^{\sigma,\delta,\sigma',\delta'}$ be the set of those overlap constraints $j \in CO$ where nurse n contributes to their adherence if she works shift σ on day δ and shift σ' on day $\delta' = \delta + 1$. The current dual multipliers of the constraints (5.9), (5.10) and (5.11) are respectively denoted by Π_n, μ_i and λ_j.

The length of an arc emanating from a node that corresponds to shift σ on day δ and ending in a node corresponding to shift σ' of day δ' for nurse n is given as:

$$\bar{c}_n^{\sigma,\delta} + \bar{c}_n^{\sigma,\delta,\sigma',\delta'} - \sum_{i \in \overline{CC}_n^{\sigma,\delta}} \mu_i - \sum_{j \in \overline{CO}_n^{\sigma,\delta,\sigma',\delta'}} \lambda_j. \tag{5.72}$$

For the nurses where the objective (O_5) should be included; let c_n^{Below} be the opportunity cost for each hour of lost work, let c_n^{Above} be the cost of overtime payment, let B_{Below} and B_{Above} be as defined in Section 5.3.9 and let SP be the length of the shortest path in the graph. The lower bound on the reduced cost can then be calculated as:

$$\text{SP} + c_n^{\text{Below}} \min B_{\text{Below}} + c_n^{\text{Above}} \min B_{\text{Above}}. - \Pi_n. \tag{5.73}$$

For the nurses where the objective (O_5) is not to be included, the lower bound on the reduced cost is attained by subtracting the dual value Π_n from the shortest path length.

The shortest path is found by backward calculating the optimal path from the nodes to the start node. As the problem is re-optimised in each node of the search tree, the optimal choice for each node in the graph is stored. When a value in a domain is removed, the corresponding node is removed from the graph including all arcs incident to the node; only paths to nodes left of the domain reductions have to be re-optimised in order to find the (new) shortest path.

When solving the pricing sub-problem, some variables are often fixed to a single value due to domain reductions. When this happens, all paths need to pass a particular node in the graph as illustrated in Figure 5.6, where day no. 3 is fixed to be a day off. As all paths then include the corresponding node, the shortest path problem decomposes into the problem of finding a shortest path from the start node to the fixed day and of finding a shortest path from the fixed day to the end node. When a domain reduction the next time only affects the second half of the graph, only this part of the graph needs to be re-optimised. This decomposition of the problem is performed until all problems consist of a single node.

The performance of this method is compared to other methods in the computational comparison section of the next section(5.3.14). The effect of including the restrictions on combinations of shifts from the other constraints is also shown.

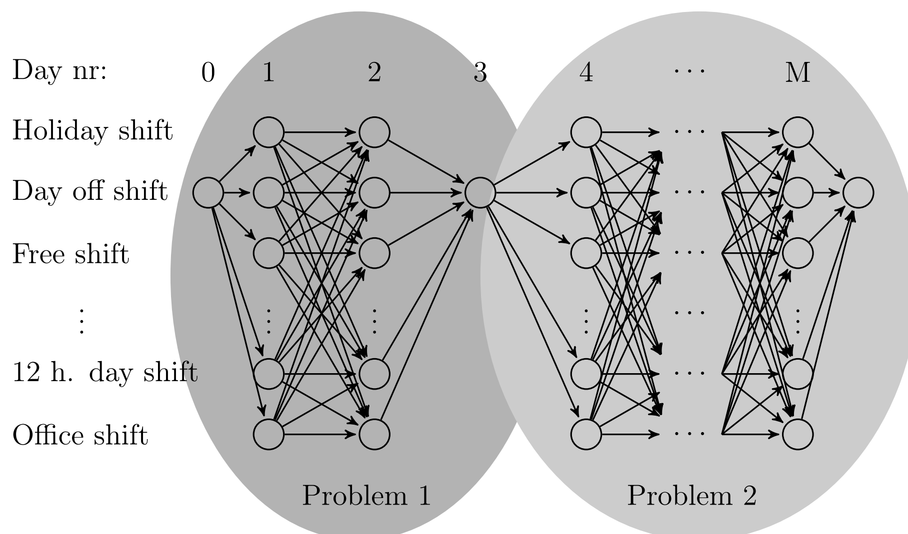

Figure 5.6: Illustration of how to split the graph into two smaller problems.

5.3.14 Using lower bound to reduce domain

The above algorithm to solve the shortest path problem calculates a shortest path tree in the graph. This tree gives the shortest path from the start node to all other nodes. By reversing all arcs and using the same algorithm with the same length on all arcs, another shortest path tree can be computed. This tree gives the shortest path from the end node to all nodes of the tree. The shortest path through a particular node is then given as the union of the shortest paths to the node in the two trees, and its length provides a lower bound on all paths through this node. If this lower bound is non-negative, the node and the corresponding domain value can be removed. When considering this lower bound, costs from the objective O_5 as well as the constant Π_n need to be included.

In Demassey et al. [15] a similar procedure as the one described in this and in the previous section for finding bounds on the objective within the CP-model and using it to perform domain reductions is used. In this thesis, however, a number of problem specific improvements are included, which contribute to a significant reduction in computation time.

Computational comparison

This sub-section analyses the computational effects of the methods mentioned in the two last sections. The methods are also compared to a method that only uses built-in constraints.

Let C++ denote the method that includes all the ideas presented in the previous two sections on calculating a lower bound and using it for domain reductions. The effect of including the restriction on combinations of shifts from the constraint C_1 on the minimum time span between shifts, the constraint C_{11} regarding complete work weekends, and some structural constraints on the combination of non-working shifts has also been tested. The method C++A is the same as C++, except that the restrictions on combinations of shifts are not included in the calculations. As the propagation mechanism for the minimal time between shifts constraint relies on this, the propagation mechanism called C++2 from Section 5.3.2 on the minimal time between shifts constraint is included.

The effect of using the lower bound to reduce domains are demonstrated by including a method, C++B, that only includes the methods described in Section 5.3.13. For this method it is also necessary to include the C++2 method for the minimal time between shifts constraint.

The instances used for testing are again the same as used for comparing the methods for the minimal time between shifts constraint. The propagation mechanisms used for all the other constraints are those employed in the final algorithm.

The results are summarised in Table 5.15, which shows that on average the C++A method uses 62.33 % more time than the C++ method. Not including the domain reductions described in the section above gives an increased computation time of more than 185 %. Thus including both the restrictions on combinations and the domain reductions are very useful for solving all the instances.

Comparing the given mechanisms with one that is based only on built-in constraints is more difficult, because the search strategy – that will be described in Section 5.3.15 – relies on the shortest path through the described graph. Two different comparisons have been made. The first comparison is performed using a different search strategy that does not rely on the shortest path through the graph. The search strategy used branches on the shift and day with the lowest cost($\vec{c}_n^{\sigma,\delta}$). The first child is created by fixing the day to the shift, and the second by removing the shift from the domain of the selected day. The computational results of this search strategy together with the methods described in the previous two sections are denoted by

Table 5.15: Effect of including different ideas for the pricing of the schedule in the CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	C++	C++A	C++B
		Average	In seconds	Deviation in %[a] from C++	
1	20	310.0	21.50	49.34	90.89
2	20	9,732.0	728.80	57.34	105.97
3	20	320.0	29.90	49.60	191.41
4	28	744.8	113.21	36.79	86.91
5	28	288.4	15.28	46.87	194.79
6	28	453.6	23.07	18.73	127.89
7	28	392.0	10.75	52.48	346.46
8	28	756.0	66.50	47.66	178.24
9	28	364.0	10.65	111.56	444.83
10	38	349.6	13.05	73.12	161.67
11	38	345.8	10.84	64.34	141.80
12	38	395.2	12.61	70.42	130.40
13	38	433.2	15.99	72.20	155.50
14	38	463.6	9.79	75.86	192.80
15	38	456.0	8.62	108.70	234.54
Average:				62.33	185.61

[a] Positive deviation means longer computation times.

C++2. The method using only built-in constraints and that uses the same search strategy as C++2 is denoted by ILOG2.

The second comparison is performed with the search strategy described in Section 5.3.15. As the search strategy relies on the shortest path through the graph, an extra mechanism that calculates this shortest path was added to both methods. This extra mechanism does not perform any domain reductions, it only calculated the shortest path. The method described in the previous two sections including the additional mechanism is denoted by C++1; the method based on the built-in constraints with the additional mechanism is denoted by ILOG1. The additional mechanism is redundant for the C++1 method, as it is already calculating the shortest path, but for comparison of the computation times it has to be included.

Table 5.16 summarises the results of the two comparisons for the same instances as for the comparison of C++, C++A and C++B. Both comparisons show that the ILOG method uses on average of at least 500 % more time on solving the instances than the C++ methods.

Table 5.16: Comparison of domain propagation mechanisms for the pricing of the schedule in the CP sub-problems for 14 days instances in 10 runs.

Ins.	Nurses	Sub-prob.	C++1	ILOG1	C++2	ILOG2
		Average	In seconds	Dev. in %[a] from C++1	In seconds	Dev. in %[a] from C++2
1	20	310.0	29.54	473.42	6.99	2,171.07
2	20	9,732.0	993.34	569.21	312.46	618.32
3	20	320.0	41.24	635.43	81.12	207.34
4	28	744.8	155.37	308.29	485.71	121.40
5	28	288.4	21.10	331.88	6.83	2,723.50
6	28	453.6	32.35	307.29	45.33	294.42
7	28	392.0	14.88	1,408.04	14.11	394.27
8	28	756.0	90.69	489.81	214.67	189.31
9	28	364.0	14.51	1,380.61	10.75	334.22
10	38	349.6	18.10	420.59	4.25	171.52
11	38	345.8	15.08	348.41	4.03	183.74
12	38	395.2	17.55	332.68	4.14	187.11
13	38	433.2	21.91	544.96	6.28	313.04
14	38	463.6	13.49	944.09	7.53	259.95
15	38	456.0	11.90	1,253.30	6.30	376.50
Average:				649.87		569.71

[a] Positive deviations means longer computation times.

5.3.15 Search in the CP model

The search of the CP search tree is performed as an depth first search where the first child is always searched first.

The CP search tree is created by branching on the shift of a day δ that is on the shortest path between the end nodes in the graph discussed in Section 5.3.13. In the branch that is selected to be searched first, the day is forced to this shift and in the other the shift is removed from the domain of the day.

Let $\mathrm{SP}(\delta, \sigma)$ be the length of the shortest path through the node that corresponds to day δ with shift σ as given in Section 5.3.13 and let $\mathrm{IMP}(\delta)$ be the importance factor of day δ. This factor is equal to one for weekdays and larger than one for weekend days. The following criteria are then applied in a lexicographic manner to select the day δ for branching:

1 – Smallest domain size:

$$|D(x_\delta)|$$

2 – Largest squared deviation from best choice:

$$\frac{\sum_{\sigma \in D(x_\delta)} \left(\mathrm{SP}(\delta, \sigma) - \min_{\sigma' \in D(x_\delta)} \mathrm{SP}(\delta, \sigma') \right)^2}{|D(x_\delta)|} \mathrm{IMP}(\delta)$$

3 – Largest average path length:

$$\frac{\sum_{\sigma \in D(x_\delta)} \mathrm{SP}(\delta, \sigma)}{|D(x_\delta)|} \mathrm{IMP}(\delta)$$

4 – Longest maximal path length:

$$\max_{\sigma \in D(x_\delta)} \mathrm{SP}(\delta, \sigma) \, \mathrm{IMP}(\delta)$$

The reason for including the importance factor is that weekend days are more restricted and hence should lead to more domain reductions. So branching on weekend days early should often lead to smaller search trees.

The motivation behind the first criterion is that smaller domains require fewer branches until the variable is fixed; smaller domains also lead to more balanced search trees as the impact of fixing and removing a value is more equal. Branching on the variable with the smallest domain is one of the standard ways of performing branching and it often leads to smaller search trees. The other criteria are based on the idea of early pruning of assignments that have a low cost. If a solution is found, it has a low reduced cost, and if not then the lower bound on the reduced cost can often be improved.

The reasoning behind branching towards shifts that are on the shortest path is to early find a solution with a very low reduced cost. The reduced cost of this solution can then be used as an upper bound on the reduced cost, which would often lead to domain reductions and pruning of nodes in the rest of the search tree.

Chapter 6

IP model

6.1 Pure IP model

The nurse rostering problem considered in this thesis may also be formulated as an IP model. The IP model is used for the purpose of a numerical comparison to the suggested IP/CP solution method. It can be solved immediately by any suitable MILP solver. The model was implemented by means of the C++ Concert interface to CPLEX 12.2 IBM [29]. Several different ways to formulate the problem were investigated and the one showing shortest average computation times on a set of smaller instances was selected.

The following notation is used for describing the IP model:

M is the set of days in the planning period,

N is the set of nurses to be scheduled,

R_{nm} is the set of possible shifts for nurse $n \in N$ on day $m \in M$. The shift types include both the fixed holiday shift, the free shift and the day-off shift, besides all the work shifts.

Moreover, the model uses the following binary decision variables:

v_{nmr} equals one, if nurse $n \in N$ works shift $r \in R_{nm}$ on day $m \in M$,

$\bar{v}_{nmrl}$ equals one, if nurse $n \in N$ works shift $r \in R_{nm}$ on day $m \in M$, and shift $l \in R_{n,m+1}$ on day $m+1 \in M$.

The second set of binary variables are used to model the shift change cost (O_4) and the overlap constraints (C_{13}).

Section 6.1.1 describes the constraints, which include several different nurses; whereas Section 6.1.2 describes the constraints that only affect the nurses schedules individually. Section 6.1.3 describes the objective.

6.1.1 Connecting constraints

Coverage constraints (C_{14})

The following constants and sets are used to define the coverage constraints:

CC	is the set of coverage constraints,
N^{CC_i}	is the set of nurses with a qualification level required by coverage constraint i,
M^{CC_i}	is the day for which coverage constraint i has to be observed,
R^{CC_i}	is the set of shifts that contribute to the adherence of coverage constraint i, if worked on day M^{CC_i},
d_i	is the desired demand level for coverage constraint i,
$\bar{d}_i$	is the difference between the desired and required demand level of coverage constraint i.

The sets and constants are defined in such a way that a nurse n contributes to the adherence of coverage constraint $i \in CC$ only if $n \in N^{CC_i}$ and nurse n works a shift from the set R^{CC_i} on day M^{CC_i}.

For each constraint for which the desired demand level is not equal to the required level, the variable y_i represents the undershoot of the demand level from the desired demand level for coverage constraint i. Each coverage constraint can then be stated as the following linear constraint:

$$\sum_{n \in N^{CC_i}} \sum_{r \in R^{CC_i}} v_{nmr} + y_i \geq d_i \qquad \forall i \in CC,\ m = M^{CC_i}, \tag{6.1}$$

$$0 \leq y_i \leq \bar{d}_i \qquad \forall i \in CC. \tag{6.2}$$

Overlap of nurses ($\mathbf{C}_{13}$)

Defining the constraints on overlapping duty periods requires two additional sets of shifts; one set of shifts for the first and another for the following day.

OC	be the set of overlap constraints,
M^{OC_j}	is the first day for which the overlap constraint j has to be observed (overlap constraint j ensures an overlap between day M^{OC_j} and $M^{OC_j} + 1$),
$R_1^{OC_j}$	is the set of shifts that would contribute to the adherence of overlap constraint j if worked on day M^{OC_j} and if a shift of $R_2^{OC_j}$ is worked on day $M^{OC_j} + 1$,
$R_2^{OC_j}$	is the set of shift that would contribute to the adherence of overlap constraint j if worked on day $M^{OC_j} + 1$ provided a shift of $R_1^{OC_j}$ is worked on day M^{OC_j},
$\bar{z}_j$	equals one if overlap constraint j is a soft constraint, and zero if it is a hard constraint.

Only if a nurse works a shift $r \in R_1^{OC_j}$ on day M^{OC_j} and a shift $l \in R_2^{OC_j}$ on day $M^{CC_i} + 1$, the overlap constraint j is fulfilled. The following variable is introduced to handle the case of a soft overlap constraint:

z_j is a continuous variable equal to one if overlap constraint j is not fulfilled, and zero otherwise.

The following constraint can then be added to ensure the required overlap of nurses:

$$\sum_{n \in N} \sum_{r \in R_1^{OC_j}} \sum_{l \in R_2^{OC_j}} \bar{v}_{nmrl} + z_j \geq 1 \qquad \forall j \in OC,\ m = M^{OC_j} \tag{6.3}$$

$$0 \leq z_j \leq \bar{z}_j \qquad \forall j \in OC \tag{6.4}$$

When the overlap constraints are hard constraints the z_j variables will be removed by the solver's preprocessor.

6.1.2 Constraints for each nurse

For all nurses $n \in N$ the following constraints have to be added.

One shift per day

Only one shift per day has to be assigned to a nurse, that is:

$$\sum_{r \in R_{nm}} v_{nmr} = 1 \qquad \forall\, m \in M. \tag{6.5}$$

Connecting the two sets of binary variables

To ensure the connection between the v_{nmr} and the $\bar{v}_{nmrl}$ variables the following constraints are added. The v_{nmr} variables might be substituted by a sum of the $\bar{v}_{nmrl}$ variables as shown in the constraints below:

$$\sum_{l \in R_{n,m+1}} \bar{v}_{nmrl} = v_{nmr} \qquad \forall\, m, m+1 \in M,\ r \in R_{nm}, \tag{6.6}$$

$$\sum_{r \in R_{nm}} \bar{v}_{nmrl} = v_{n,m+1,l} \qquad \forall\, m, m+1 \in M,\ l \in R_{n,m+1}. \tag{6.7}$$

Minimum time between shifts ($\mathbf{C}_1$)

The constraint on the minimum time span between shifts can be expressed as a set of pairs of incompatible shifts, where the first shift of the pair is of day m and the second of the following day $(m+1)$. For all possible pairs $(r\ ,\ l)$ of incompatible shifts, the corresponding connection variable $\bar{v}_{nmrl}$ is then fixed to zero for all days m.

Day off / free day (C_9)

To define the constraint ensuring that a day off shift lies in a period of 35 hours of free time, the following notation is introduced:

ST(r)	is the start time of shift r,
ET(r)	is the end time of shift r,
T	is the minimal amount of free time for a day off ($T = 35$ hours).

Let DO be the day off shift, then the constraint can be formulated as:

$$\sum_{r \in R_{n,m+1}} \mathrm{ST}(r)\, v_{n,m+1,r} - \sum_{r \in R_{n,m-1}} \mathrm{ET}(r)\, v_{n,m-1,r} \geq \mathrm{T}\, v_{nm\mathrm{DO}} \quad \forall m \in M. \tag{6.8}$$

To ensure that a day is not selected as a free day shift if it should be a day off shift, the following constraint is added:

$$\sum_{r \in R_{n,m+1}} \mathrm{ST}(r)\, v_{n,m+1,r} - \sum_{r \in R_{n,m-1}} \mathrm{ET}(r)\, v_{n,m-1,r} < \mathrm{T} + (1 - v_{nm\mathrm{F}})K \quad \forall m, \tag{6.9}$$

where F is the Free shift and K is a big constant selected to be equal to $\max ST(\cdot) - \min ET(\cdot) - \mathrm{T}$.

Maximum consecutive workdays (C_{10})

Two different sets of constraints should be added depending on the contractual agreement of the nurse. The standard constraint set is the following which restricts the number of days between two days off to be less than or equal to 6. Let H be the holiday shift and DO the day off shift, then the constraint can be expressed as:

$$\sum_{k=m}^{m+6} (v_{nk\mathrm{DO}} + v_{nk\mathrm{H}}) \geq 1 \qquad \forall\, m = -6, -5, \ldots, |M| - 6. \tag{6.10}$$

The other case applies if the nurse has accepted that the constraint can be loosened such that the nurse is allowed to have a maximum of 7 days between two days off, if at least one of the days in between is assigned a free shifts. If there is no assigned free shift in between, the maximum is still 6 days. These restrictions can be formulated as

the following constraints:

$$\sum_{k=m}^{m+7}(v_{nk\mathrm{DO}} + v_{nk\mathrm{H}}) \geq 1 \quad \forall\, m = -7, -5, \ldots, |M| - 7, \tag{6.11}$$

$$\sum_{k=m}^{m+6}(v_{nk\mathrm{DO}} + v_{nk\mathrm{H}} + v_{nk\mathrm{F}}) \geq 1 \quad \forall\, m = -6, -5, \ldots, |M| - 6, \tag{6.12}$$

where F is again the free shift.

Minimum consecutive workdays ($\mathbf{C}_2$)

Let WS_{nm} be the set of work shifts for nurse $n \in N$ on day $m \in M$. To ensure a minimum of two consecutive workdays, the following are introduced:

$$\sum_{r \in \mathrm{WS}_{nm}} v_{nmr} \leq \sum_{r \in \mathrm{WS}_{n,m-1}} v_{n,m-1,r} + \sum_{r \in \mathrm{WS}_{n,m+1}} v_{n,m+1,r} \qquad \forall m \in M. \tag{6.13}$$

Shift type limits ($\mathbf{C}_4$)

For each restricted set of shifts the following notation is used:

A	is the set of shifts to be restricted,
$\max A$	is the upper limit on the number of shifts in A during the planning period,
$\min A$	is the lower limit on the number of shifts in A during the planning period,
B	is a continuous variable.

Then the constraint below can be added to the model to enforce the limit:

$$\sum_{m \in M} \sum_{r \in A} v_{nmr} - B = \min A, \tag{6.14}$$

$$0 \leq B \leq \max A - \min A. \tag{6.15}$$

Complete weekends ($\mathbf{C}_{11}$)

As a consequence of the restrictions at this ward that complete weekends has to be worked, the weekend shifts are divided into pairs of shifts, which either both have to be worked or not at all. All the pairs consist of two shifts with the same shift type on two consecutive days.

Let *SatSun* be the shifts, which are restricted between Saturday and Sunday, and let *SunMon* be the shifts, which are restricted between Sunday and Monday.

The constraint can for all days m that are Sundays be formulated as:

$$v_{nmr} = v_{n,m-1,r} \qquad \forall r \in \mathit{SatSun}, \tag{6.16}$$

$$v_{nmr} = v_{n,m+1,r} \qquad \forall r \in \mathit{SunMon}. \tag{6.17}$$

$$\tag{6.18}$$

Because of variable substitutions, these constraint will be removed, when the model is preprocessed.

Consecutive work weekends ($\mathbf{C}_6$)

For all Sundays m in the planning period the following constraint is added to ensure that no nurse has two consecutive work weekends:

$$\sum_{r \in \mathrm{WS}_{nm}} v_{nmr} + \sum_{r \in \mathrm{WS}_{n,m+7}} v_{n,m+7,r} \leq 1. \tag{6.19}$$

Recall that the WS_{nm} is the set of work shifts for nurse $n \in N$ on day $m \in M$.

12 hours weekend shifts ($\mathbf{C}_{12}$)

This constraint should ensure that if a nurse has accepted to work 12 hours shifts, then the first work weekend after a weekend of 12 hours shifts has to be one with 8 hours shifts. The following two sets of shifts are needed to define this constraint:

$\mathrm{WS12}_{nm}$ 12 hours work shifts of nurse n on day m.

$\mathrm{WS8}_{nm}$ 8 hours work shifts of nurse n on day m.

For all sets A of consecutive Sundays of size greater than or equal to three, add the following constraint:

$$\sum_{m \in A} \sum_{r \in \mathrm{WS12}_{nm}} v_{nmr} - \sum_{m \in A} \sum_{r \in \mathrm{WS8}_{nm}} v_{nmr} \leq 1. \tag{6.20}$$

Maximum work hours per week ($\mathbf{C}_3$)

This constraint imposes an upper limit on the number of work hours for each week (a week starts on Mondays). For ease of notation let:

$\mathrm{WH}(r)$ be the work hours of shift r and

MWH denote the maximum work hours per week.

For all m that are Mondays, the following constraint can be added to enforce this upper limit:

$$\sum_{k=m}^{m+6} \sum_{r \in R_{nk}} \mathrm{WH}(r)\, v_{nkr} \leq \mathrm{MWH} \tag{6.21}$$

Office days ($\mathbf{C}_5$)

For the nurses, for which office days should be planned, some constraints to ensure that the right number of office shifts are assigned, have to be included. The demand for office days is given as a number of shifts during a period. Let δ_1 and δ_2 be the first and last day in this period respectively, let b be the given demand and let O be the office day shift, then the constraint

$$\sum_{m=\delta_1}^{\delta_2} v_{nm\mathrm{O}} = b \tag{6.22}$$

is added.

Recorded work hours ($\mathbf{C}_7$)

If the planning period does not include the end of the nurse's twelve weeks contract period, then this constraint can be handled with a single linear inequality. Let maxRH and minRH respectively be the lower and upper bound for the number of recorded work hours during the planning period and let $\mathrm{RH}_m(r)$ be the number of recorded hours for shift r on day m. The constraint added is then the following:

$$\mathrm{minRH} \leq \sum_{m \in M} \sum_{r \in r_{nm}} \mathrm{RH}_m(r) v_{nmr} \leq \mathrm{maxRH} \tag{6.23}$$

If the planning period includes the end of the nurse's twelve weeks contract period, then a couple of constraints are required. The following constants and variables are used to define these constraints:

$\mathrm{RH}_m(r)$	as above; the number of recorded hours for shift r on day m.
δ	Last day in the twelve weeks contract period.
CRH	The number of contracted recorded hours for the twelve weeks period minus the number of contracted hours that has been worked before the current planning period.
minRH	The lower bound on the number of recorded hours for the days after day δ.

maxRH	The upper bound on the number of recorded hours for the days after day δ.
B_{Actual}	A continuous variable representing the actual number of recorded hours.
B_{Below}	A non-negative continuous variable representing the number of recorded hours, the nurse works less than her contracted number.
B_{Above}	A non-negative continuous variable representing the number of recorded hours, the nurse works more than her contracted number.

Then the constraints added can be defined as follows

$$\sum_{\substack{m\in M\\ m\le\delta}}\sum_{r\in R_{nm}}\mathrm{RH}_m(r)v_{nmr} = B_{\text{Actual}}, \tag{6.24}$$

$$\mathrm{minRH} \le \sum_{\substack{m\in M\\ m>\delta}}\sum_{r\in R_{nm}}\mathrm{RH}_m(r)v_{nmr} \le \mathrm{maxRH}, \tag{6.25}$$

$$B_{\text{Actual}} + B_{\text{Below}} - B_{\text{Above}} = \mathrm{CRH}. \tag{6.26}$$

Preassigned shifts ($\mathbf{C}_8$)

Preassigning a nurse to work a specific shift or ensuring that she does not work a specific shift corresponds to fixing some of the v_{nmr} variables to either zero or one.

6.1.3 The objective

The objective function comprises the following cost coefficients:

$c_n^{r,m}$	is the preference cost (O_3) for nurse n of working shift r on day m,
$\bar{c}_n^{r,m,l,m+1}$	is the preference cost (O_4) for nurse n of working shift r of day m and shift l of day $m+1$,
p_i	is a linear penalty (O_1) for not achieving the desired demand level for coverage constraint i.
q_i	The penalty (O_2) for not fulfilling overlap constraint j,
c_n^{Below}	is the opportunity cost for each hour of lost work for nurse $n(\mathrm{O}_5)$,
c_n^{Above}	is the cost of overtime payment for nurse n (O_5).

The objective can then be defined as:

$$\sum_{n\in N}\sum_{m\in M}\sum_{r\in R_{nm}} c_n^{r,m}\, v_{nmr} \tag{6.27}$$

$$+\sum_{n\in N}\sum_{m\in M}\sum_{r\in R_{nm}}\sum_{l\in R_{n,m+1}} \bar{c}_n^{r,m,l,m+1}\, \bar{v}_{nmrl} \tag{6.28}$$

$$+\sum_{i\in CC} p_i y_i + \sum_{j\in OC} q_j z_j \tag{6.29}$$

$$+ c_n^{\text{Below}} B_{\text{Below}} + c_n^{\text{Above}} B_{\text{Above}}. \tag{6.30}$$

The last two terms (6.30) are only included if the planning period includes the last day of the nurse's twelve weeks period of the contracted number of recorded hours.

6.2 IP/IP model

In addition to the IP model of the previous section, the IP/CP approach was also compared to an "IP/IP" approach. This approach uses the same branch-and-price algorithm as the one applied within the IP/CP method, but integer programming instead of the proposed constraint programming method is applied to solve the pricing sub-problems. So this method is still based on a lot of the work performed for developing the IP/CP algorithm described in this thesis. The IP model for the sub-problems consists of all the constraints described in Section 6.1.2, except that there is a separate model for each nurse. Additionally, the model has to handle the dual values from the master problem as well as the branching decisions. The branching decisions correspond exactly to fixing one of the v_{nmr} variables to either zero or one. Let $c_n^{r,m}$ and $\bar{c}_n^{r,m,l,m+1}$ be defined as in Section 6.1.3 and let the dual multipliers μ_i , λ_j, Π_n and the sets $\overline{CC}_n^{r,m}$ and $\overline{CO}^{r,m,l,m+1}$ be defined as in Section 5.3.13. Then the objective for the sub-problem for nurse n can be stated as:

$$\sum_{m\in M}\sum_{r\in R_{nm}} c_n^{r,m}\, v_{nmr} \tag{6.31}$$

$$+\sum_{m\in M}\sum_{r\in R_{nm}}\sum_{l\in R_{n,m+1}} \bar{c}_n^{r,m,l,m+1}\,\bar{v}_{nmrl} \tag{6.32}$$

$$+\, c_n^{\text{Below}} B_{\text{Below}} + c_n^{\text{Above}} B_{\text{Above}}. \tag{6.33}$$

$$-\sum_{m\in M}\sum_{r\in R_{nm}} v_{nmr} \sum_{i\in\overline{CC}_n^{r,m}} \mu_i \tag{6.34}$$

$$-\sum_{m\in M}\sum_{r\in R_{nm}}\sum_{l\in R_{n,m+1}} \bar{v}_{nmrl} \sum_{j\in\overline{CO}_n^{r,m,l,m+1}} \lambda_j \tag{6.35}$$

$$-\,\Pi_n \tag{6.36}$$

The two terms (6.33) are only included if the planning period includes the last day of the nurse's twelve weeks period of the contracted number of recorded hours.

Chapter 7

Computational results

7.1 IP/CP vs IP vs IP/IP

The different solution approaches – IP/CP, IP and IP/IP – have been tested on a number of problem instances with a planning period of two weeks. All instances use the same set of shifts, with the same start and end times and include all the constraints to be addressed for the particular case of the Danish ward underlying this thesis. The instances differ in both the number of nurses available, the demand of the coverage constraints and almost all the details of all other constraints (for instance: number of office days, holidays and shift preferences). The parameters kept fixed correspond to regulations in union agreements and parameters regarding the shift types.

All computations were performed on a Sony laptop with 8 GB of memory and a 2.80 GHz Intel i7 processor on a Linux operating system with a Gnu 4.7.2 C++ compiler. All algorithms were restricted to use only a single core. The computation time was limited to at most 20,000 seconds and the maximal memory usage to be no larger than 3.5 GB. As some parts of the algorithms rely on randomness, every instance was solved with 10 different seed corns.

The computation times presented in this chapter cannot be compared to those of the computational comparisons made in Chapter 5, as those were performed on a Toshiba laptop with 4 GB of memory and a 1.73 GHz Intel i7 processor. All other settings and restrictions were the same.

Table 7.1 summarises the computations results. The columns headed "IP/CP" shows the computation times, the number of times a master problem was solved and the number of schedules generated by the IP/CP algorithm described in Chapter 5; columns "IP/IP" display the results obtained with IP/IP that uses CPLEX's MIP solver for solving the pricing sub-problems. The "Dev. in %" column indicates the relative deviation in computation time from the IP/CP algorithm; a positive number means that the IP/IP method uses more computational time. The last columns, "IP", display the computation

Table 7.1: Average solution times of 14 days instances of 10 runs. (Times in seconds)

		IP/CP			IP/IP				IP	
Ins.	Nurses.	Time	Master it.	Sch. gen.	Time	Dev. in %[a]	Master it.	Sch. gen.	Time	Gap in %
1	20	29.4	15.5	1,126.8	31.6	7.64	23.5	876.7	285.8	
2	20	534.7	486.6	5,786.1	1,368.7	155.98	495.9	11,623.6	10,657.8*	3.25‡
3	20	25.5	16.0	1,238.0	71.2	179.50	27.0	1,012.0	7,344.5*	7.05‡
4	28	86.5	26.6	1,358.3	72.2	−16.52	25.0	1,135.0	14,227.0*	8.05‡
5	28	18.7	10.3	1,119.7	19.8	6.08	15.2	774.2	Timeout**	0.16¶
6	28	21.1	16.2	891.7	24.6	16.57	22.6	937.1	2,804.5*	0.11¶
7	28	12.0	14.0	1,017.0	39.8	231.31	17.0	848.0	6,008.5*	4.02‡
8	28	44.3	27.0	1,034.0	209.5	372.78	66.9	2,261.1	19,304.7*	11.57‡
9	28	11.4	13.0	1,008.0	28.2	147.99	14.0	719.0	8,772.5*	6.60‡
10	38	20.9	9.2	1,084.8	30.0	43.74	11.9	808.4	5,269.9*	0.04†
11	38	14.3	9.1	1,094.1	31.7	121.35	11.3	770.2	13,111.4*	1.36‡
12	38	21.1	10.4	1,083.1	30.1	42.66	11.8	745.9	3,868.5*	0.36‡
13	38	21.3	11.4	1,303.4	28.7	34.63	13.3	870.4	6,471.1*	11.92‡
14	38	17.7	12.2	1,252.0	35.9	103.06	18.0	1,032.5	12,724.1*	0.35†
15	38	17.9	12.0	1,210.0	33.3	86.09	15.0	954.0	6,910.9*	††
	Sum:	896.7	689.5	21,607	2,055.2		788.4	25,368.1		

[a] Deviation in solution time from IP/IP method where positive deviations means longer computation times.
* Out of memory (3.5 GB)
** Timeout (20,000 second)
¶ Solution with optimal solution value found.
† Near optimal solution found.
‡ Integer solution found, but far away from the optimal one.
†† No integer solution found.

times required by CPLEX for solving the integer programming model as well as the relative gap between the best integer solution found and the best lower bound found by CPLEX.

All figures for the IP/CP and IP/IP algorithms are averages over 10 runs. The table shows that on instance 1 and 5 the IP/CP and the IP/IP method uses on average almost the same time, whereas otherwise the IP/CP outperforms the IP/IP method. Taking the average, over all instances, of Time(IP/IP)/Time(IP/CP) shows that the IP/IP method uses 102.19 % more computational time than the IP/CP method. An even worse computational performance showed CPLEX's MIP solver on the integer programming model. Only a single instance could be solved within the limits of 20,0000 seconds of computation time and 3.5 GB of memory usage. In case of instance no. 5 and 6, the IP approach finds solutions with the optimal objective value, but is unable to prove optimality within the time and memory limits. When solving instance no. 10 and 14, the CPLEX solver found a solution quite close to the optimal solution value, and on instance no. 15 no solution was found within these limits. On the other instances, the best solution found was far away from the optimal solution value. The figures in column "Gap in %" are defined as the best solution value minus the lower bound found divided by the optimal solution value. Note that these figures should be used with caution. Generally the coverage constraints (5.3) are met with equality for all near optimal solutions. Certain cost components can be increased in value by adding a certain $\Delta > 0$ to all of them without any changes in the difference of objective value of any near optimal solution. Accordingly, the relative GAP can be made arbitrarily small.

On all except two of the instances, the IP method broke the limit of 3.5 GB memory usage; the maximal memory usage registered with the two other algorithms were however just 47 MB.

The IP/CP method spends on average 3.3 % of the computation time by solving the linear relaxation of the restricted master problem. The time used for selecting which branch to perform are excluded from that measure. An average of 35.5 % of the time is spend for selecting which branch to perform and the last 61.2 % are spend solving the sub-problems. The amount of time spend for selecting the branch and for solving the sub-problems varies a lot. The amount of time spend for solving sub-problems as an average over the 10 runs varies from 32.7 % to 94.1 %; for selecting the branching, it varies between 4.6 % and 63.6 %. The amount of time used for selecting the branching seems quite high, but some preliminary tests on a subsets of instances showed that the computation time was on

average reduced by around 38 % by using a candidate list of size 50 compared to using a size of 10. The decision of using a candidate list with a size of 50 showed the smallest computation time of all tested settings. The size of the candidate list showed a decrease in computation time when increasing the size until 50 was reached.

Some preliminary tests on some three weeks instances showed that all three solution methods described here were incapable of solving the instances, given the time limit of 20,000 seconds. This was both due to the increased number of nodes in the branch-and-bound tree and the increased CPU time required for solving the sub-problems. In the case that the number of feasible schedules for each nurse is smaller and the nurses are more different from each other, it might sometimes be possible to solve some three weeks instances to optimality. However, from a practical point of view it is not interesting to solve three weeks instances as the requested planning period is at least four weeks.

For the IP/CP and IP/IP methods it is very important that the constraints on each nurse's schedule are quite different from each other. If several nurse's have to obey almost the same constraints it would often lead to a huge search tree for the master problem. The advantage of having similar nurses is that they could be grouped together. Schedules generated for a nurse in a group might also be feasible for the other nurses of the same group. The addition of extra schedules could reduce the overall computation time. However, to calculate a lower bound a sub-problem have to be solved for each nurse. Thus if the constraint sets for two nurses are not exactly the same, both sub-problems have to be solved.

An extension to the IP/CP method has also been tested. To this end, the three and four weeks instances were heuristically solved by splitting the planning period into two parts; each part was then solved separately with the IP/CP method. An attempt was also performed by planning the first week, then the second week where the first week was kept fixed and so on. However these attempts all failed as there generally did not exist a feasible schedule when the last part / week had to be planned. Several attempts of dividing the constraints for the whole planning period into constraints for the period which is under scheduling were performed, however none of these trials lead to feasible solutions.

For a longer planning period than two weeks, it seems that it will be very difficult to solve instances with an exact method. However, the two weeks instances can be solved within a reasonable computation time.

7.2 Comparison of CP and IP

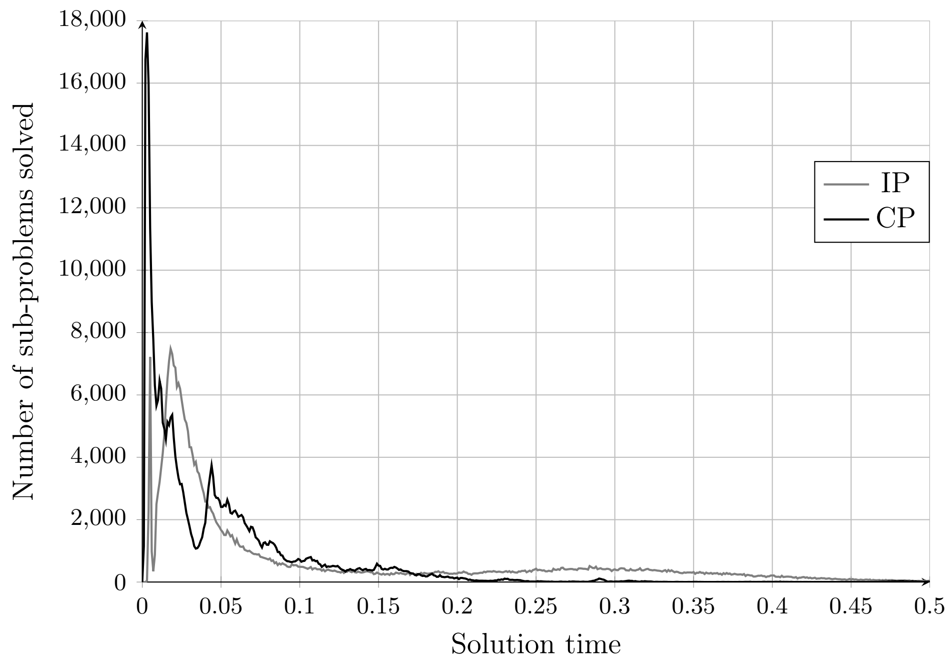

Figure 7.1: The number of sub-problems solved by the two methods in intervals of length 0.001 second. Sub-problems showing longer computation time than 0.5 seconds have not been included; these sub-problems make up less than 1 % of the sub-problems for both methods.

In order to compare the two methods for solving the pricing sub-problems in more detail, all pricing sub-problems were solved by both the CP and the IP algorithm. Each time a sub-problem was encountered, it was solved with both the CP and the IP algorithm. When solving with the IP/CP algorithm only the schedules generated with the CP algorithm were added, likewise when the IP/IP algorithm was used. The solution time and the number of schedules generated for both methods of solving the sub-problem has been collected by solving all 15 instances with 10 different random seed corns.

On average, the IP algorithm uses 114.8 % more computation time than the CP and this result could be observed for both search paths.

Figure 7.2 and 7.1 compares the computation times spent by both methods on solving the sub-problems. Figure 7.1 shows the number of sub-problems solved within a give time interval; the intervals are

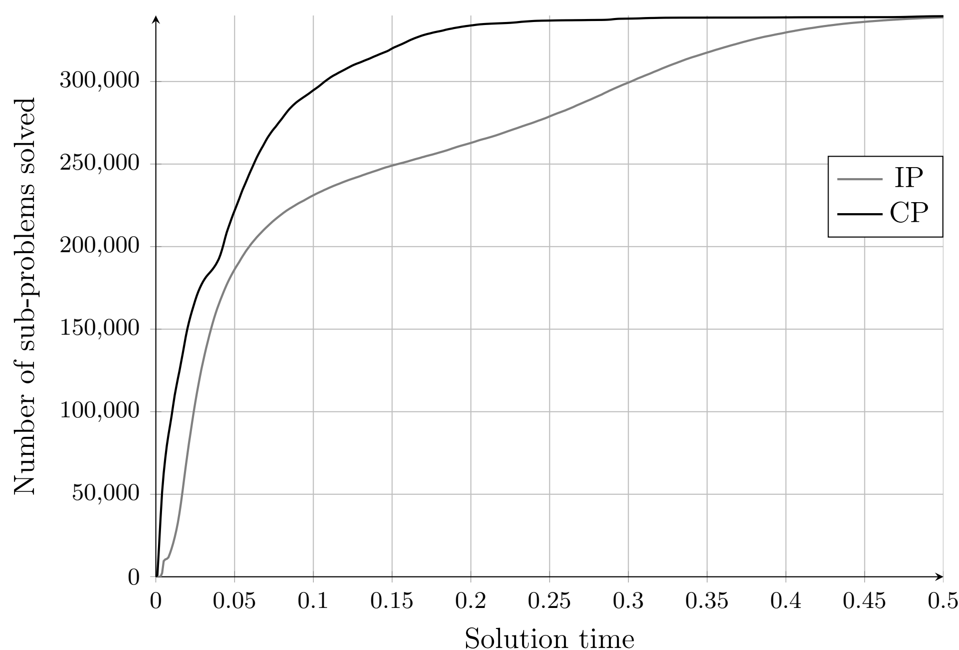

Figure 7.2: The accumulated number of sub-problems solved by the two methods. Sub-problems showing longer computation time than 0.5 seconds have not been included; these sub-problems make up less than 1 % of the sub-problems for both methods.

all of size 0.001 second. Figure 7.2 shows the accumulated number of sub-problems solved with each method within a given amount of time. Thus graphs of Figure 7.2 are approximately the accumulated value of the graphs of Figure 7.1. Sub-problems showing longer computation time than 0.5 seconds have not been included; these sub-problems make up less than 1 % of the sub-problems for both methods. It can be seen that the CP algorithm solves a lot of problems almost instantly, whereas the IP algorithm requires some time before it is able to solve any problem. Even after this initial time, the CP method solves most of the instances faster than the IP method. The CP method solved 98 % of the instances in 0.204 seconds, whereas the IP method required 0.432 seconds.

For every master iteration of the two branch-and-price algorithms, one pricing sub-problem is solved for each nurse. All feasible integer solutions with negative reduced cost found while solving the sub-problem are added to the restricted master problem. In total the sub-problems of the CP method generate around 60 % more schedules with negative reduced cost than the sub-problems of the IP method.

In a sub-problem which is not proven to be infeasible, the CP method generates on average 4.33 schedules, whereas the IP method generates 2.83 schedule. The number of generated schedules for the two methods are quite consistent over the different instances and with both search paths. Creating more schedules will not in general lower the computation time, as a larger restricted master problem also requires more time to be solved. As the same search strategy was used for the IP/CP and IP/IP method, one of the main reasons for the lower number of master iterations in the IP/CP is probably the extra schedules generated. The number of master iterations for the IP/CP algorithm were on average 12.5 % lower than the number of master iterations for the IP/IP algorithm.

7.3 Flexibility of the IP/CP method

In order to investigate the flexibility of the method, we created a few further instances that include some restrictions not present at the Danish ward. These restrictions were: A minimal number of consecutive night shifts, a maximal number of consecutive night shifts, a minimal number of off-days after a set of consecutive night shifts, and a minimal number of non-night shifts between sets of consecutive night shifts. The constraints have been added with some "If Then" statements ensuring that the constraints are not violated. As an example; the constraint on a minimal number of consecutive night shifts is added in the following way for all days i:

$$x_{i-1} = \text{night and } x_i \neq \text{night} \quad \Rightarrow \quad x_{i-d}, \ldots, x_{i-1} = \text{night},$$

where d is the minimal number of consecutive night shifts. The others constraint are implemented in a similar way.

Table 7.2 shows the results obtained when solving the instances with the extra constraints. Instances 16–21 include the first two constraints, whereas instances 22–24 include all four of them. The computation times are still very reasonable, and all instances are solved to optimality.

Table 7.2: Average solution times of 10 runs for the IP/CP method for 14 days instances with extra constraint. (Times in seconds)

		IP/CP		
Instance	Nurses	Time	Master it.	Schedules generated
16	20	25.7	15.7	974.5
17	20	30.1	22.0	1,062.0
18	20	83.3	43.0	1,305.0
19	28	12.9	13.0	955.0
20	28	39.4	24.0	848.0
21	28	18.0	14.3	919.4
22	28	25.8	12.6	1,013.5
23	28	41.0	28.0	595.0
24	28	42.9	40.1	728.7

Part III

Heuristic approaches

Chapter 8

Variable Neighbourhood Search

In this chapter a variable neighbourhood search (VNS) designed for the nurse rostering problem from Chapter 4 is described. The VNS builds on the VNS method described in Algorithm 2 in Section 2.4.1. It is designed to handle infeasibility with respect to the coverage and overlap constraints. All other constraints have to be obeyed in the starting solution and will also be obeyed in the final and all intermediate solutions. The overall VNS algorithm designed is presented in Section 8.4, after all the sub-methods used have been presented.

The initial solution for the VNS is created by a greedy heuristic described in Section 8.1. The VNS designed does not use any shaking of the current solution before the neighbourhood search is applied. When the current solution is infeasible, a sort of indirect shaking is applied. This is achieved by defining the costs of the current infeasible solution in such a way, that the search result in a different solution with either the same or smaller degree of infeasibility. How this is performed is described in Section 8.2.

All of the employed neighbourhoods are based on the same principle. A given set of days is “released”, while all others are kept fixed to the shifts assigned to these days in the current solution. Hence, the neighbourhood corresponds to all solutions where the nurses on the non-released days work the shifts as they were supposed to do in the current solution, whereas the released days can attain any value. The maximum number of released days is given as a parameter Δ. A given neighbourhood can then be described as a set of days to be released. The neighbourhoods are not selected in sequence; how they are selected is described in the sections below together with the descriptions of the neighbourhoods.

The method used for searching the neighbourhoods is the IP/CP solution approach described in Chapter 5. The neighbourhoods used are very large, similar to the ones used by Bent and Van Hentenryck [4] for a pick-up and delivery problem.

In general; the search completely searches the neighbourhood and proves that the best solution found is optimal. However, since it is not necessary to prove optimality in a neighbourhood, the search is terminated if it requires too much time. Besides the time limit for a single iteration, the following limits can also be introduced: A maximum on the number of non-improving iterations, a total time limit and an iteration limit.

Depending on the feasibility of the current solution different neighbourhoods are used. If the current solution is infeasible the neighbourhoods from Section 8.2 are used, and if feasible those from Section 8.3.

8.1 Greedy heuristic

In this section a greedy heuristic is proposed for finding an initial solution to be used as a starting solution for the other heuristics presented.

The heuristic is built up around the constraint programming model that is also used for solving the sub-problem in the IP/CP model. The CP model is the same, except that the coefficients of the objective function are changed. The model and the domain reduction strategies are described in Section 5.3.

The general idea is to use an accelerated constraint programming search that searches for a schedule for the nurses, one at a time. The generated schedules are feasible for the constraints which only depend on the schedule for a single nurse, whereas the coverage constraints (C_{14}) and the overlap constraints (C_{13}) are not necessarily fulfilled.

The search is guided towards schedules that might contribute to the adherence of the coverage and overlap constraints, not fulfilled by the nurses already scheduled. This way, a solution that is close to feasibility is hopefully created. This guidance makes the single schedules dependent on the order in which the nurses are scheduled.

8.1.1 Cost calculations

The costs – which in the CP sub-problem was used for calculating the reduced cost – are in this heuristic used for guiding the search towards a schedules that is close to feasibility.

The costs used within the greedy method are determined in the following way, where the current staffing levels are calculated from the nurses for which schedules already have been generated: All costs

$\bar{c}_n^{\sigma,\delta}$ and $\bar{c}_n^{\sigma,\delta,\sigma',\delta'}$ are initially set to zero. For all the shifts involved in a violated coverage constraint (5.3) and where the nurse has the required qualification level, the difference between the required and the current staffing level is subtracted from the costs $\bar{c}_n^{\sigma,\delta}$ (O_3). This is done in order to let the greedy method prioritise the selection of shifts involved in violated coverage constraints. Similar, for each violated overlap constraint (5.4), a positive constant is subtracted from the corresponding overlap costs $\bar{c}_n^{\sigma,\delta,\sigma',\delta'}$ (O_4).

8.1.2 Search in the CP model

For the purposes of the greedy heuristic, the constraint programming search is based on a static ordering of the days and a greedy dynamic selection of the shift to be assigned to the given day. The days δ are sorted according to non-decreasing order of

$$\sum_{\sigma \in D(\delta)} \left(\bar{c}_n^{\sigma,\delta} + \sum_{\sigma' \in D(\delta-1)} \bar{c}_n^{\sigma',\delta-1,\sigma,\delta} + \sum_{\sigma' \in D(\delta+1)} \bar{c}_n^{\sigma,\delta,\sigma',\delta+1} \right), \qquad (8.1)$$

where $D(\delta)$ is the set of all shifts that can be assigned to day δ. The branching is then performed on the days in this given order.

When a day δ has been chosen, the shift σ to branch on is selected according to a "score value" defined as the sum of the following three terms. The first term is the shift cost $\bar{c}_n^{\sigma,\delta}$ as defined above. The second term relates to the overlap costs with the previous day $\delta - 1$. If this day is fixed to a specific shift σ', this term equals the overlap cost $\bar{c}_n^{\sigma',\delta-1,\sigma,\delta}$. If the day before is not fixed, then this term is set to

$$\min_{\sigma' \in D(\delta-1)} \alpha \bar{c}_n^{\sigma',\delta-1,\sigma,\delta} + \beta \bar{c}_n^{\sigma',\delta-1}, \qquad (8.2)$$

where $\alpha \in [0,1]$ and $\beta \in [0,\alpha]$ are constants. In the similar way, the third term relates to the overlap costs with the following day. If a tie in score values appears, a value is chosen at random.

8.2 Infeasibility iterations

An iteration – where the current solution is infeasible – starts with a random selection of a constraint that is not fulfilled. Because the VNS is designed only to handle infeasibility with respect to the coverage or overlap constraints, the selected constraint is either a coverage or an overlap constraint.

If an overlap constraint is selected, a neighbourhood around the days for which the overlap has to be observed is used. The neighbourhoods to select between are all sets of Δ consecutive days including

the two days for which the overlap has to be observed as well as the day before and after these two days. The neighbourhood is selected at random between those that fulfil those requirements.

If a coverage constraint is selected and the day (M^{CC_i}) for which the coverage constraint should be observed is either a Saturday, Sunday or Monday one of the neighbourhoods listed below are used. If M^{CC_i} is between Tuesday and Friday, only the second type of neighbourhood is used:

Weekend neighbourhoods are a set of consecutive weekends. The specific neighbourhood to be used are chosen in the following way: The set of weekends to be released is created by first adding the weekend in which M^{CC_i} is one of the days, then randomly either the following or previous weekend is added, until $\lfloor \Delta/3 \rfloor$ weekends are selected. The days that are selected to be in the neighbourhood are the Saturday, Sunday and Monday of the weekends released.

One period neighbourhoods are all sets of days consisting of Δ consecutive days, where $M^{CC_i}-1$, M^{CC_i} and $M^{CC_i}+1$ all are members. The specific neighbourhood to use is chosen randomly.

The IP/CP method used for searching the neighbourhoods are not capable of handling infeasible solutions directly. To handle the infeasibility – which stems from the coverage and overlap constraints – some artificial columns are added to the restricted master problem (5.8– 5.14). For each violated coverage or overlap constraint, a artificial column with a single one in the row that corresponds to the violated constraint is added. The variable corresponding to this column is restricted to be between zero and the difference between the current and required level. The cost of such a column is set to a large number. In subsequent iterations the cost of the artificial columns are increased.

For all coverage and overlap constraints that are met with equality in the given solution an artificial column is also added. The column has a value of one in the row which corresponds to the constraint, all others are zero. The corresponding variable is restricted to be between zero and one. The cost of the column is set to value slightly higher than for the ones corresponding to infeasible constraints. This would ensure that if another solution with the same level of infeasibility exist in the neighbourhood, it will have a lower cost than the starting solution. The solution obtained from the improvement method will thus be different from the starting one. The change of solution will lead to other areas of the search space which then hopefully would contain a feasible solution.

8.3 Normal iteration

In an iteration where the current solution is feasible; one of three different types of a neighbourhood is selected with a given probability.

When the optimal solution with respect to a neighbourhood has been found, it would be a waste of computation time to search it again as long as the current solution has not changed. Therefore, a history is kept of neighbourhoods fully explored since a improved solution was found the last time. When a new improved solution is found, the history is reduced to include only the current neighbourhood. If the search is stopped before reaching optimality due to some limit, the neighbourhood is not saved.

Weekend neighbourhoods are all sets of weekends of size $\lfloor \Delta/3 \rfloor$. The weekends are not required to be consecutive; both Saturday, Sunday and Monday of the weekend are released in the neighbourhood.

One period neighbourhoods are all sets of Δ consecutive days where the planning period is thought of as a cyclic order of days, e.g. there is the same number of neighbourhoods as the number of days in the planning period.

Two period neighbourhoods are all sets of days which corresponds to two periods of $\lfloor \Delta/2 \rfloor$ consecutive days where the planning period is thought of as a cyclic order of days. The two periods are allowed to be consecutive but not overlapping, e.g. the number of days released are always $2\lfloor \Delta/2 \rfloor$.

The specific weekend neighbourhood to release is chosen randomly between the weekend neighbourhoods which have not previously been released for the current solution. The choice of the specific one and two period neighbourhoods is more involving. The idea behind the selection is to choose a set of days for resleasing that is quite different from the previous selections. If a neighbourhood have already been tried and completely searched for the current solution it is excluded for consideration. Let "History" denote the sets of days that have previously been released and where the optimal solution has been found. A neighbourhood with a corresponding set of days ϕ is selected with a ratio of:

$$\frac{1}{\max_{\theta \in \text{History}} |\phi \cap \theta|^{\alpha}}, \tag{8.3}$$

where $\alpha \geq 1$ is a constant. Accordingly, the more days it has in common with a previously tested neighbourhood, the smaller the probability of selecting the neighbourhood. The reason using the

days as a cyclic order for generating neighbourhoods is to get a more even probability of releasing the different days.

Some preliminary tests on a small set of instances showed that using all three neighbourhoods above was better than using any pair of the above neighbourhoods. The one period neighbourhoods are a subset of the two period neighbourhoods, thus it could easily be thought that the one period neighbourhoods were redundant. However, the preliminary tests showed that they were not, maybe because the one period neighbourhoods often lead to more improvements than the two periods. Yet, the two periods are still needed to make jumps to other areas of the search space, which are not reachable by the one period neighbourhoods.

8.4 Overview of the method

An stepwise overview of the VNS designed for the nurse rostering problem is given below:

1. Create the initial solution by the greedy heuristic from Section 8.1.
2. While the current solution is infeasible:.
   - 2a. Select a violated constraint at random.
   - 2b. Select a neighbourhood according to Section 8.2 on the infeasibility iteration.
   - 2c. Search the selected neighbourhood with the IP/CP algorithm.
     (An iteration time limit is used to this end.)
   - 2d. If a limit has been reached, stop;
   - 2e. If the obtained solution is still infeasible go to step 2a.
3. Create a history of previously searched neighbourhoods.
   The last searched neighbourhood from the infeasibility iteration is added.
4. While no limits have been reached:
   - 4a. Select a neighbourhood according to Section 8.3 on the normal iteration.
   - 4b. Search the selected neighbourhood with the IP/CP algorithm.
     (An iteration time limit is set to this end.)
   - 4c. If an improved solution has been found reset the history to be the empty set.
   - 4d. If the obtained solution was proven optimal in the neighbourhood, add it to the history.
   - 4e. If a limit have been reached, stop; otherwise go to step 4a.

The limits used are an iteration limit, a time limit and a limit on the maximal number of non-improving iterations.

This VNS differs some from the framework described in Section 2.4.1; first of all the algorithm has been divided into two parts, a search used when the current solution is infeasible and one when it is feasible. Another difference is that instead of using the same neighbourhood for the local search, different neighbourhoods are selected.

Chapter 9

Scatter Search

A scatter search for the nurse rostering problem from Chapter 4 is described in this chapter. The scatter search is designed according to the framework described in Section 2.4.2. The five methods which connect the framework to the problems to be solved are described in separate sections below. A overview of the final scatter algorithm is presented in Section 9.6.

All intermediate and final solutions are feasible with respect to the constraints that relate to the nurses separately; the only constraints that are not necessarily fulfilled are thus the coverage and overlap constraints.

The initial solution is created by the same greedy heuristic as used for finding the initial solution for the VNS. The greedy heuristic used is described in Section 8.1.

The scatter search spends most of its computational effort in the improvement method. Of the three methods that create new solutions, the improvement method is the only one that directly takes the coverage and overlap constraints into account. The starting solution for the improvement method is usually infeasible, because this solution is obtained either by means of the diversification or the solution combination method.

9.1 The diversification method

The diversification method should generate solutions in different parts of the search space. In this implementation, the diversification method takes a single solution as its starting solution.

As suggested in Laguna and Armentano [35], a frequency memory is used to make the created solutions more diverse. A frequency memory of all previous assignments of nurses to the different working shifts is kept. The demand for the different working shifts can be very different, hence the expected number of assignments of these shifts will vary. Thus penalising assignments only according to the number of previous assignments seems not reasonable. A more elab-

orate method thus used to fix assignments according to the difference between the previously generated and expected number of assignments. To use this method, an estimate of the expected number of assignments for the different working shifts is needed. For the normal working shifts, the estimate used is the maximal desired demand for the shift type on the given day, divided by the number of nurses. The reason for this is that the qualification levels of the nurses and the qualification requirements of the coverage constraint are stated such that the maximal desired demand is the number of nurses needed to work the shift. For the office days, a different estimate must be given; the estimate is given as the demand divided by the number of days it can be assigned to. For all days for which an office day can not be assigned, the estimate is zero.

The schedules for the nurses are generated independently. The solution obtained this way does thus usually not meet the coverage and overlap constraints. The CP method of Section 5.3 is used to compute a schedule for each nurse; another search strategy is however used for the purpose of the scatter search.

The CP model is the same as the one employed for solving the pricing sub-problem, except that no domain reductions can be performed in the calculations of the lower bound for the reduced cost as described in Section 5.3.14. Hence the propagation method C++2 from for the minimal time between shifts constraint is added. A part of the search strategy relies on the lower bound on the cost and the assignment of shifts that corresponds to it, so the cost of a schedule is still calculated as in Section 5.3.13, but where all dual values are considered to be zero.
The search strategy in the CP model can be divided into three steps:

1. Branch according to the frequency memory.
2. Branch towards the starting solution.
3. Branch towards the lower bound on the cost of the schedule.

In any path from the root node of the CP search tree to any other node, the branching is performed in sequence. That means that on the first part of the path, the branching method of the first step is used, while the second part is built using the branching method of the second step, and so on. Note that on some paths it might happen that some of the branching methods do not find any branches that they can create. The first feasible solution found is the solution which is returned as the output from the diversification method.

Branch according to the frequency memory. This branching strategy is used until a given number of shifts have been fixed according to the frequency memory. If all days have been fixed to either

working shifts or their domains have been reduced to a subset of the non-working shifts, then the branching strategy is also changed to the next in the sequence. A day and a shift is chosen to be branched on; the first child node is created by fixing the day to the shift and the other child by removing the shift from the domain of the day. The frequency memory does not account for solutions generated by the diversification method; only those obtained after improving them by the improvement method, are used for updating the frequency memory.

The day to branch on is selected according to the frequency memory in the following way. All days which have not have their domains reduced to singletons or a subset of the non-working shifts, receive a weight. The day to be fixed is selected at random according to these weights. For each (day, working shift) pair an estimate of the expected number of assignments is given. Thus an estimate of the expected number of times this day should be a working day is the sum over the working shifts of the estimates for the (day, working shift) pairs; let this estimate for each day be denoted by $\mathrm{EA}_{\mathrm{day}}$. The number of times a working shift for the given day has been previously generated as a working shift, is the number of schedules in the frequency memory where a working shift was assigned to the given day; let this value for each day be denoted by $\mathrm{PA}_{\mathrm{day}}$. Let $|\mathrm{FM}|$ denote the number of solutions registered in the frequency memory. The weight of a day is given as:

$$\max\left(\alpha, 1 + \frac{\mathrm{EA}_{\mathrm{day}} - \mathrm{PA}_{\mathrm{day}}}{|\mathrm{FM}|}\right),$$

where $0 \leq \alpha < 1$ is a constant that defines a minimum weight for a day. The "1+" ensures that the second term of the "max" is between zero and two; the weight becomes one when the expected and scheduled number of working days are the same. The weight becomes smaller when the number of scheduled working days is larger than the expected, and larger if opposite.

The shift to branch on is selected the same way as the day. The weight is not defined on the basis of the difference of the estimate of the expected and the scheduled number of working shifts, but on the difference of the expected and the scheduled number of assignments of the working shift to the selected day.

The motivation behind using this search strategy is to create a solution that is diverse from the previously generated solutions.

Branch toward the starting solution, This is a simple strategy that randomly selects a day that has the shift from the starting solution in its domain and where the day is not already fixed to single

shift. In the first child; the selected day is fixed to the shift from the starting solution and in the other the shift is removed from the domain. When all days have either been fixed or have the value from the starting solution removed from their domains, the search strategy advances to the next in the sequence.

As the starting solution is feasible for all the constraints that only relate to single nurses and is also likely to meet most of the coverage and overlap constraints, branching towards such a solution should generally quickly result in a solution that is feasible or at least close to feasibility.

Branch towards the lower bound on the cost of the schedule. This is the branching strategy used for the CP sub-problems in the IP/CP method; it is described in Section 5.3.15. The idea of the strategy is to greedily search towards the solution with lowest cost. When using this strategy all dual values used in the branching strategy are equal to zero. This search strategy does not stop until all days have been fixed.

As opposed to the two other branching strategies, this search strategy considers the costs of the schedule.

9.2 The improvement method

The variable neighbourhood search described in Chapter 8 is used as the improvement method. The VNS is chosen because it searches some very large neighbourhoods efficiently and is able to handle infeasible solutions; additional repair heuristics are thus not required.

A smaller subsets of days (Δ) is used than if the VNS is employed on its own. Both a time limit on each iteration and a limit on the number of non-improving iterations are applied. The limits are important, because some iterations of the VNS might use a lot of computation time just to prove local optimality of a solution. As the number of neighbourhoods are large, finding a local optimum for all of them is too time consuming.

9.3 The reference set

The reference set is the population of solutions from which new solution are created. The reference set is kept at a constant size after it has been initialised.

9.3.1 Initialising the reference set

The reference set is selected from a pool of initial solutions. This pool is created from the initial solution with the use of the diversification method. The diversification method is given a randomly chosen solution from the pool and it searches for a new solution. If a new solution is found it is forwarded to the improvement method. If the improved solution is not already in the pool it is added.

The reference set is chosen to include besides some of the best solutions from the pool, also some solutions that are very different from the other solutions selected. To define which solution to select for the reference set a measure of diversity has to be defined. Let α_1, α_2 and α_3 be constants such that $0 < \alpha_1 \leq \alpha_2 \leq \alpha_3$. For a given (day, nurse) pair three cases are distinguished:

1. If the nurse is assigned to different non-work shifts on the given day in the two solutions, a diversity of α_1 is assigned.
2. If the nurse is assigned to different work shifts in the two schedules on the given day, a diversity of α_2 is assigned.
3. If the nurse is assigned a work shift in one schedule and a non-work shift in the other on the given day, a diversity of α_3 is assigned.

The diversity between two feasible solutions is the sum over all nurses and days of the constants above. If one or both of the solutions are infeasible the measure is adjusted, because the solutions anyway need to be changed to become feasible. Let the infeasibility of a solution be the sum of the number of overlap constraints that are not fulfilled and the difference between the required and actual staffing level of the coverage constraints that are not fulfilled. Let Infeasibility_1 and Infeasibility_2 be respectively the infeasibility of the first and second solution; the diversity "measure" is then adjusted in the following way:

$$\max\left(0, \text{Measure} - \alpha_2 \max(\text{Infeasibility}_1, \text{Infeasibility}_2)\right). \quad (9.1)$$

The diversity between a solution and a set of solutions is measured as the minimal diversity between the solution and each of the solutions in the set.

The reference set is chosen as suggested in Laguna and Armentano [35], the first half of the reference set is chosen as the solutions with the lowest objective value; the other half is chosen iteratively as the one showing the highest diversity to the current partial reference set.

9.3.2 Updating the reference set

Both a static and dynamic update method were implemented. The static update method keeps a pool of the best solutions generated; the same number of solutions are kept as the size of the reference set. The pool is initialised with the solutions from the reference set, and when the reference set has to be updated it is replaced with the pool.

In the dynamic case, when a new solution has been generated, it is added if it is better than the worst solution in the reference set. If it is added the worst solution from the reference set is removed and the reference set size thus kept constant.

Some initial testing of the scatter method showed quickly that the dynamic method is superior. One of the main reasons was that usually only one or two static updates of the reference set was made. Hence the solutions used for the combination method were only second or third generation of the initial solution.

9.4 The subset generation method

The subset generation method has to generate all subsets of the reference set which should be used for the combination method. The subsets generated are those proposed in Laguna and Armentano [35]:

- All pairs of solutions.
- All pairs with the addition of the solution with best objective value.
- All pairs with the addition of the two solutions with best objective values.
- The set of k solutions with the best objective values, $k = 5, 6, \ldots$

All sets are generated such that in each set all solutions are different, besides this only non-duplicate sets are generated.

9.5 The solution combination method

The solution combination method should combine a given set of solutions into at least one new solution. To do this for the given problem a very simple combination method was implemented. The new solution is created by choosing a random schedule for each nurse from the schedules assigned to the nurse in the given solutions. Choosing the solution this way leads to solutions that are feasible for the constraints that relate to each nurse separately, whereas the coverage and overlap constraints are usually not fulfilled.

It is very difficult to figure out which parts of a solution are "good" and which are not. Hence creating a combination method that selects the good parts of a solution is almost impossible and heavily depends on the given instance. So instead of trying to create an elaborate method that is very dependent on the particular problem instance, a very simple method was chosen. Finding the "good" parts of the solutions is indirectly performed by means of the improvement method; good parts of a solution should also be present in the solution after applying the improvement method, and if the resulting solution is good enough it will be added to the reference set. More and more solutions in the reference set should this show properties of high quality solutions and be selected in turn for the combination method with an even larger probability.

9.6 Overview of the method

The scatter search for the nurse rostering problem is described in two phases. In the initial phase the reference set is created and the second phase is a search phase where the solutions of the reference set are combined into hopefully better solutions.

Initial phase:

1. Create the initial solution by the greedy heuristic from Section 8.1.
2. Initialise a pool of solutions P to be the empty set.
3. Initialise the frequency memory.
4. Add the initial solution to both P and the frequency memory.
5. While the size of P is less than a given size.
   - 5a. Select a random solution from P.
   - 5b. Generate a solution with the *diversification method* from the selected solution according to the frequency memory .
   - 5c. Use the *improvement method* to improve the obtained solution.
     Both a time and a limit on the maximal number of non-improving iterations is enforced.
   - 5d. If the solution attained is not already in P, add it to P and register it in the frequency memory.
   - 5e. Repeat from 5a until P has reached a given size.
6. Select the reference set from the pool P as given in Section 9.3.1.

Search phase

7. Generate a set of subsets SoS with the *subset generation method*.

8. While SoS is not empty.
   - 8a. Select and remove a subset from SoS.
   - 8b. From the selected subset, generate a solution with the *solution combination method*.
   - 8c. Use the *improvement method* to obtain an improved solution.
     Both a time and a limit on the maximal number of non-improving iteration is enforced.
   - 8d. Update the reference set with the *reference update method*.
   - 8e. If an iteration or time limit is reached, stop.
   - 8f. If SoS is not empty go to step 8a.
9. If the reference set has changed since last time step 7 was executed, go to step 7. Stop otherwise.

An iterated version has also been implemented. When the iterated version reaches the stop criterion of step 9, instead of stopping the reference set is re-initialised. The pool P is set equal to the half part of the current reference set showing the best objective function value. The frequency memory is also initialised with this set. The search is then restarted from step 5. In the computational tests, this iterated version came, however, rarely into play, as the overall time limit is often reached before the stop criterion ti be checked in step 9 is reached.

Chapter 10

Computational results

In this chapter, the variable neighbourhood search (VNS) and the scatter search (SS) are compared on a set of problem instances. The instances were created with a planning period of four weeks, because the requested planning period in practise would be around this length. As with the two weeks instances, all instances use the same set of shifts, same start and end times and include all the constraints to be addressed for the particular case of the Danish ward underlying this thesis. The instances differ in both the number of nurses available, the demand of the coverage constraints and almost all the details of all other constraints. The parameters kept fixed correspond to regulations in union agreements and parameters regarding the shift types.

The reason why the heuristics are not tested on the two weeks instances for a comparison to the exact methods is that the parameters had to be changed to work with such a short planning period. Thus the heuristics would perform very differently on the short instances compared to how they perform on the longer ones. The number of days released (Δ) in the VNS is selected (based on some tests) to 14 days, so the VNS would in each iteration solve the complete problem. Changing the parameter Δ would not make a fair comparison as the strength of the VNS stems from the fact that Δ is rather large, smaller values of Δ would not create large enough neighbourhoods to find better solutions. Thus, the exact and heuristics method are not compared, because the exact methods were not able to solve any of the 4 weeks instances. In the IP/CP and IP/IP method the root node was often not even solved before the time limit of 20.000 seconds was reached.

As with the exact method, all computations were performed on a Sony laptop with 8 GB of memory and a 2.80 GHz Intel i7 processor on a Linux operating system with a Gnu 4.7.2 C++ compiler. All algorithms were restricted to use only a single core. The maximal memory usage was limited to be no larger than 3.5 GB.

As the algorithm rely on randomness, every instance, heuristic

pair was tested with 6 different seed corns. The computations were stopped when the computation time reached 10,000 seconds.

As the VNS generally gets caught in a local optimum after some iterations the scatter search was compared to a multistart version of the VNS and not to the simple version. The multistart version of the VNS works as follows: When a given limit on the number of non-improving iterations has been reached, the VNS is restarted. The greedy heuristic used for creating the initial solution relies on randomness, so the starting solution should generally be different from the previously tested ones. Besides this, the neighbourhoods of the VNS are selected randomly; even with the same initial solution, the VNS should thus often search different neighbourhoods and lead to other solutions. In the rest of this chapter the multistart version of VNS is the method denoted by VNS.

A lower bound was found by using the IP model from Chapter 6.1. The model was solved with the use of CPLEX's MIP solver, which was terminated after 20.000 seconds. All instances were tested with both the normal algorithm and when the best known feasible solution was given as input to the solver. The use of the best known solution did usually slightly improve the lower bound, but in general the returned solution was the same as the one given as input. Thus, CPLEX was generally not capable of improving the generated solutions.

For both heuristics several different parameter setting have been tested on a smaller set of instances. For both methods the parameter setting showing the best results is used for the computational tests. One parameter which is included in both methods and where they differ is the number of days to released (Δ). For the VNS, a setting of 14 days showed the best results, whereas for the SS a Δ of 12 days showed the best performance. The reason behind this is probably that the SS uses the combination method to search other interesting areas of the search region, whereas the VNS needs larger neighbourhoods to escape a local optimum.

Table 10.1 summarises the results of the comparison between the heuristics. The deviations are calculated as the best solution value found within the given time limit subtracted by the lower bound found by CPLEX divided by the lower bound. The deviations cannot be compared between the instances, as the lower bounds are often quite weak. Additionally, the same caution should be taken as described for the column "Gap in %" for the experimental results for the exact methods. For instance, the cost of instance 14 is very low and hence a small difference in solution value translates into a huge value in relative difference. It should be noted that the best

Table 10.1: Comparison of scatter search (SS) and multistart variable neighbourhood search (VNS). (Averages over 6 runs.)

Ins.	Nurses	Type	1.000s	3.600s	7.200s	10.000s
			Relative deviation in % from lower bound.			
1	14	SS	0.313	0.275	0.251	0.237
		VNS	0.323	0.279	0.273	0.264
2	28	SS	1.424	1.255	1.146	1.092
		VNS	1.460[2]	1.328[1]	1.247	1.231
3	28	SS	3.924	1.455	1.455	1.455
		VNS	1.841	1.455	1.455	1.455
4	28	SS	21.108	4.803	1.897	1.897
		VNS	10.283	8.416	3.824	3.824
5	28	SS	0.000	0.000	0.000	0.000
		VNS	0.061	0.032	0.019	0.013
6	28	SS	2.809	1.612	1.566	1.517
		VNS	2.589	2.328	2.012	1.996
7	28	SS	3.921	1.341	0.966	0.910
		VNS	2.379	1.842	1.795	1.644
8	28	SS	3.472	1.375	0.945	0.819
		VNS	1.841[2]	1.499	1.328	1.245
9	38	SS	0.000	0.000	0.000	0.000
		VNS	0.056	0.006	0.000	0.000
10	38	SS	0.328	0.004	0.000	0.000
		VNS	0.194	0.012	0.012	0.012
11	38	SS	1.340	0.832	0.780	0.756
		VNS	1.550	0.973	0.938	0.914
12	38	SS	32.795	6.607	5.172	5.112
		VNS	17.130[1]	8.430	8.430	7.085
13	38	SS	35.997	1.415	0.541	0.270
		VNS	7.491	5.202	2.622	2.538
14	38	SS	104.971	78.898	73.355	70.136
		VNS	97.597[1]	89.020	83.727	81.116
Average:		SS	15.172	7.134	6.291	6.014
		VNS	10.342[6]	8.630[1]	7.692	7.381

[1,2,...] Number of runs where no feasible solutions have been found. The runs showing no feasible solution are excluded from the calculations of the deviations.

known solution for instance number 14 has a relative deviation from the lower bound of around 65 %, whereas for all the other instances it is below 2 %. The deviations are calculated after 1.000, 3.600, 7.200 and 10.000 seconds.

The table shows that the scatter search (SS) performs better than the variable neighbourhood search (VNS) for time limits of at least 1 hour (3.600 seconds). After just 1.000 seconds, it is a more open question, as the VNS on most instances found better solutions, however on 6 out of the 84 runs, it did not find any feasible solution.

Two different statistical methods were used to compare the results. The first method is based on the assumption that the difference in solution value between the two heuristics is normally distributed. When the results were analysed for each instance separately, nothing could generally be concluded as there are only six observations for each instance. When all results corresponding to at a given time limit are analysed, it seems reasonable to assume that the difference in solution value follow a normal distribution. Analysing the absolute difference compared to a relative difference was chosen, because of the way the cost of the instances were defined. For the time limits of 3.600, 7.200 and 10.000 seconds, the null hypothesis that the difference between the objective values equals zero could be rejected. The test used was a t-test with a significance level of $\alpha = 0.05$. The values of the test statistic were respectively $\mathrm{t}_{3.600} = 5.17$, $\mathrm{t}_{7.200} = 5.94$ and $\mathrm{t}_{10.000} = 6.12$. For all three limits the scatter method was concluded to be significant better than the VNS method. The one run where the VNS method did not find a feasible solution before the time limit of 3600, was excluded from consideration in the statistical test.

For the time horizon of 1.000 seconds the results were, however, quite different. If the differences where the VNS did not find any feasible solution was excluded, the same hypothesis as before could be rejected, but this time it was the VNS which was significantly better. Depending on which objective value the infeasible solution should be assigned, the statistical test would not be able to conclude that the methods return statistically different results. If a difference between two solutions, where the one is infeasible was defined to have a difference of double the value of the maximal difference obtained for the corresponding instance, then the null hypothesis could barely not be rejected (It would be accepted for $\alpha > 0.069$).

The other statistical method used for comparing the heuristics is the Wilcoxon signed rank test [21]. An advantage of this method, is that the runs without feasible solutions can be included without defining a cost for them, provided that only one of the methods failed to find a solution. The Wilcoxon method ranks the differences

in objective value and uses a test statistic that depends only on this ranking. For the differences where the VNS method did not find a feasible solution, the difference is set to be a value larger than all others. For all four time limits, the null hypothesis of equal solution quality of the two methods could be rejected. Thus, it could be concluded that after 1.000 seconds the solution quality of the VNS method is superior, whereas after 3.600, 7.200 and 10.000 seconds the SS method was superior according to this statistical method.

An illustration of which of the two methods results in a better solution value within a given amount of CPU time is shown in Figure 10.1. For a given time, the figure shows for how many of the 84 runs the SS or the VNS respectively found a better solution than the other. A third line shows on how many instances both methods found a solution of the same objective function value, or failed to find any. The figure shows similar performances as the statistical test. The VNS method is superior up to a computation time of around 1.800 seconds, hereafter the SS is superior. Note that the figure does not say anything about the absolute difference between the objective function values.

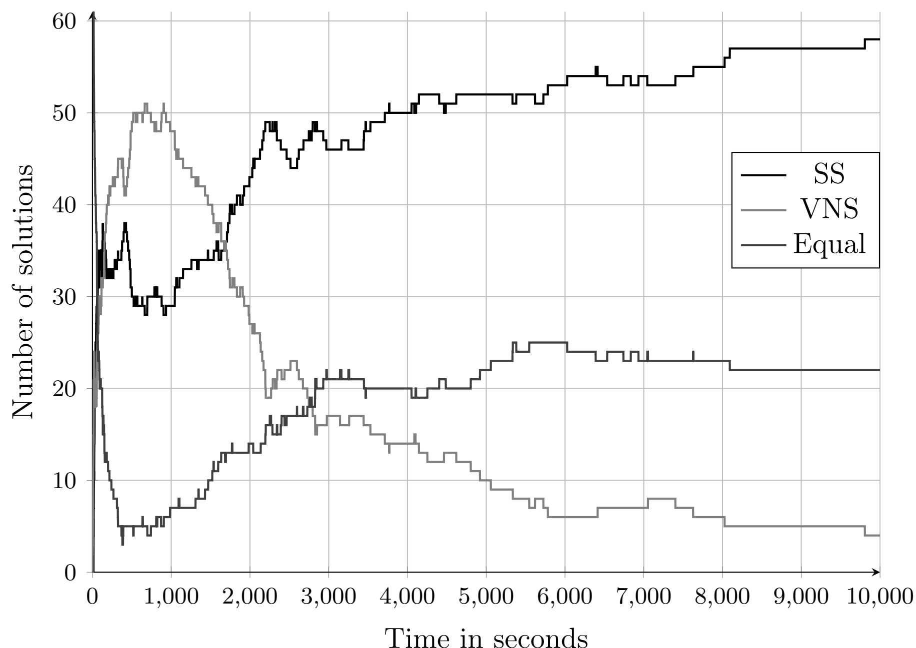

Figure 10.1: The number of instances where either the VNS or Scatter search is the best, or the number of instances where the solution found has the same objective value, as a function of the computation time.

On average, the VNS method was restarted with a new initial solution 33 times before the time limit of 10.000 seconds was reached. The number of times a set of days was released and hence the number of calls to the IP/CP algorithm was an average of 1949 times per run.

The time spend in the initial phase of the scatter search was on average around 1.000 seconds, hence the time spend in the search phase was around 9.000 seconds. Of the time spend in the initial phase, around 4 % was spend in the diversification method and the rest by the improvement method. The time used for the combination method could not really be measured as each call took less than the smallest measurable time. Hence almost all time spend in the search phase was spend in the improvement method. The solution combination method was performed, on average, 330.7 times for each run.

The scatter search generates typically a set of solutions which are different from each other, but where all are very close in solution value to the best solution found. From a practical point of view, this is a very attractive behaviour, as the head nurse could then choose between, e.g., the 10 best solutions. A set of solutions is also generated indirectly by the VNS, by saving all feasible solutions found. However, the objective value span of the first 10 or 100 solutions is much larger than for the SS. Additionally the diversity of the solution set of the VNS was not comparably larger.

The scatter search seems to perform very well on all the instances and significant better than the variable neighbourhood search if at least one hour of computation time is used. In practise, several hours of computation time could easily be assigned, as one instance only have to be solved each 4 weeks. However, using several days is not really an option, as the time between the requests from the nurses have been registered and when the scheduling should be finished is usually not very long.

From an academic point of view it could be very interesting to see how the scatter search compares to other heuristics that do not use the IP/CP as a sub-routine. Designing a suitable neighbourhood for such a heuristic seems to be challenging, if only feasible solutions may be visited by the search. It seems actually to be necessary to include infeasible solutions in the search; but this introduces a lot of difficulties about how to balance solution quality compared to feasibility. Just finding a measure of infeasibility is difficult when so many different types of constraints can be violated. Thus designing such a heuristic would require a big effort, and the outcome will often be dependent on the constraints to be considered for the instances.

Bibliography

[1] Achterberg, T., T. Koch, and A. Martin (2005). Branching rules revisited. *Operations Research Letters 33*, 42–54.

[2] Barnhart, C., E. L. Johnson, G. L. Nemhauser, M. W. Savelsbergh, and P. H. Vance (1998, February). Branch-and-price: Column generation for solving huge integer programs. *Operations Researc 46*(3), 316–329.

[3] Bellanti, F., G. Carello, F. Della Croce, and R. Tadei (2004). A greedy-based neighborhood search approach to a nurse rostering problem. *European Journal of Operational Research 153*, 28–40.

[4] Bent, R. and P. Van Hentenryck (2006). A two-stage hybrid algorithm for pickup and delivery vehicle routing problems with time windows. *Computers & Operations Research 33*, 875–893.

[5] Burke, E. K., P. Cowling, P. De Causmaecker, and G. Vanden Berghe (2001). A memetic approach to the nurse rostering problem. *Applied Intelligence 15*, 199–214.

[6] Burke, E. K., P. De Causmaecker, and G. Vanden Berghe (1999). A hybrid tabu search algorithm for the nurse rostering problem. In *Proceedings of the Second Asia-Pacific Conference on Simulated Evolution and Learning*, pp. 187–194. Springer.

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

[8] Bæklund, J. Nurse rostering at a danish ward. *Annals of Operations Research*. Revised version under revision.

[9] Bæklund, J. (2010, August). Nurse rostering - in a danish hospital. In M. Collan (Ed.), *Proceedings of the 2nd International Conference on Applied Operational Research - ICAOR'2010*, Volume 2 of *Lecture Notes In Management Science*, pp. 145–148.

[10] Carrabs, F., J.-F. Cordeau, and G. Laporte (2007). Variable neighbourhood search for the pickup and delivery traveling salesman problem with lifo loading. *INFORMS Journal on Computing. 19*, 618–632.

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

[12] COIN-OR (2009, Januar). *Branch-cut-price framework.* `https://projects.coin-or.org/Bcp`.

[13] Dantzig, G. B. and P. Wolfe (1960). Decomposition principle for linear programs. *Operations Research 8*(1), pp. 101–111.

[14] Demassey, S., G. Pesant, and L. M. Rousseau (2005a). Constraint programming based column generation for employee timetabling. In R. Barták and M. Milano (Eds.), *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Volume 3524 of *Lecture Notes in Computer Science*, pp. 140–154. Springer Berlin / Heidelberg.

[15] Demassey, S., G. Pesant, and L.-M. Rousseau (2005b). Constraint programming based column generation for employee timetabling. In R. Barták and M. Milano (Eds.), *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Volume 3524 of *Lecture Notes in Computer Science*, pp. 830–833. Springer Berlin / Heidelberg.

[16] Dincbas, M., H. Simonis, and P. V. Hentenryck (1990). Solving large combinatorial problems in logic programming. *The Journal of Logic Programming 8*(1–2), 75 – 93. Special Issue: Logic Programming Applications.

[17] Dowsland, K. A. and J. M. Thompson (2000). Solving a nurse scheduling problem with knapsacks, networks and tabu search. *Journal of the Operational Research Society 51*(7), 825–833.

[18] Fahle, T., U. Junker, S. E. Karisch, N. Kohl, M. Sellmann, and B. Vaaben (2002). Constraint programming based column generation for crew assignment. *Journal of Heuristics 8*, 59–81.

[19] Gendreau, M. and J.-Y. Potvin (Eds.) (2010). *Handbook of Metaheuristics* (2nd ed.), Volume 146 of *International Series in Operations Research & Management Science.* Springer US.

[20] Glover, F. (1998). A template for scatter search and path relinking. In J.-K. Hao, E. Lutton, E. Ronald, M. Schoenauer, and D. Snyers (Eds.), *Artificial Evolution*, Volume 1363 of *Lecture Notes in Computer Science*, pp. 1–51. Springer Berlin Heidelberg.

[21] Golden, B. and W. Stewart (1985). Empirical analysis of heuristics,. In Lawler, Lenstra, Rinooy Kan, and Shmoys (Eds.), *The Traveling Salesman Problem,*, pp. 207–250. John Wiley.

[22] Goodman, M., K. Dowsland, and J. Thompson (2009). A grasp-knapsack hybrid for a nurse-scheduling problem. *Journal of Heuristics 15*(4), 351–379.

[23] Hansen, P., N. Mladenovic, J. Brimberg, and J. A. M. Perez (2010). *Handbook of Metaheuristics* (2nd ed.), Volume 146 of *International Series in Operations Research & Management Science*, Chapter Variable Neighborhood Search, pp. 61–86. Springer US.

[24] Hansen, P. and N. Mladenović (2001). Variable neighborhood search: Principles and applications. *European Journal of Operational Research 130*(3), 449 – 467.

[25] Hansen, P. and N. Mladenović (2003). *Handbook of Metaheuristics* (1st ed.)., Chapter Variable Neighbourhood Search, pp. 145–184. International series in operations research and management science. Kluwer Academic Publishers.

[26] Hentenryck, P. and V. Saraswat (1997). Constraint programming: Strategic directions. *Constraints 2*(1), 7–33.

[27] Hentenryck, P. V. (1989). *Constraint Satisfaction in Logic Programming.* Logic Programming Series. Cambridge, MA: MIT Press.

[28] Hooker, J. (2000). *Logic-based methods for optimization : combining optimization and constraint satisfaction.* Wiley-interscience series in discrete mathematics and optimization. Wiley-Interscience.

[29] IBM (2009a). *IBM ILOG CPLEX V12.1 User's Manual for CPLEX.* IBM.

[30] IBM (2009b, June). *IBM ILOG Solver Reference Manual.* IBM.

[31] IBM (2009c, June). *IBM ILOG Solver V6.7 User's Manual.* IBM.

[32] Jaffar, J. and J.-L. Lassez (1987). Constraint logic programming. In *Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '87, New York, NY, USA, pp. 111–119. ACM.

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

[34] Kellerer, H., U. Pferschy, and D. Pisinger (2004). The multiple-choice knapsack problem. In *Knapsack Problems*, pp. 317–347. Springer Berlin Heidelberg.

[35] Laguna, M. and V. Armentano (2005). Lessons from applying and experimenting with scatter search. In R. Sharda, S. Voß, C. Rego, and B. Alidaee (Eds.), *Metaheuristic Optimization via Memory and Evolution*, Volume 30 of *Operations Research/Computer Science Interfaces Series*, pp. 229–246. Springer US.

[36] Land, A. H. and A. G. Doig (1960). An automatic method of solving discrete programming problems. *Econometrica 28*(3), pp. 497–520.

[37] Li, H., A. Lim, and B. Rodrigues (2003). A hybrid AI approach for nurse rostering problem. In *Proceedings of the 2003 ACM symposium on Applied computing*, SAC '03, New York, NY, USA, pp. 730–735. ACM.

[38] Lin, S. (1965). Computer solutions of the traveling salesman problem. *Bell System Technical Journal 44*, 2245–2269.

[39] Linderoth, J. T. and M. W. P. Savelsbergh (1999, February). A computational study of search strategies for mixed integer programming. *INFORMS J. on Computing 11*(2), 173–187.

[40] Lübbecke, M. E. and J. Desrosiers (2005, November). Selected topics in column generation. *Oper. Res. 53*(6), 1007–1023.

[41] Madsen, A. K. (2007, July). Vagtplanlægning for sygeplejersker. Ms. thesis in oper. res., Department of Mathematical Sciences, Aarhus University, Denmark.

[42] Marriott, K. and P. J. Stuckey (1998). *Programming with constraints : an introduction.* MIT Press Cambridge, Mass.

[43] Martí, R., M. Laguna, and F. Glover (2006). Principles of scatter search. *European Journal of Operational Research 169*(2), 359–372.

[44] Meisels, A., E. Gudes, and G. Solotorevsky (1996). Employee timetabling, constraint networks and knowledge-based rules: A mixed approach. In E. K. Burke and P. Ross (Eds.), *Practice and Theory of Automated Timetabling*, Volume 1153 of *Lecture Notes in Computer Science*, pp. 91–105. Springer Berlin / Heidelberg.

[45] Meyer auf'm Hofe, H. (2001). Solving rostering tasks as constraint optimization. *Practice and Theory of Automated Timetabling III 2079/2001*, 191–212.

[46] Nemhauser, G. L. and L. A. Wolsey (1988). *Integer and combinatorial optimization.* New York, NY, USA: Wiley-Interscience.

[47] Okada, M. (1992). An approach to the generalized nurse scheduling problem-generation of a declarative program to represent institution-specific knowledge. *Computers and Biomedical Research 25*(5), 417 – 434.

[48] Ribeiro, C. C. and M. C. Souza (2002). Variable neighborhood search for the degree-constrained minimum spanning tree problem. *Discrete Applied Mathematics 118*(1–2), 43 – 54. Special Issue devoted to the ALIO-EURO Workshop on Applied Combinatorial Optimization.

[49] Sellmann, M., K. Zervoudakis, P. Stamatopoulos, and T. Fahle (2002). Crew assignment via constraint programming: Integrating column generation and heuristic tree search. *Annals of Operations Research 115*, 207–225.

[50] Vanderbeck, F. (2000, January). On dantzig-wolfe decomposition in integer programming and ways to perform branching in a branch-and-price algorithm. *Oper. Res. 48*(1), 111–128.

[51] Warner, D. M. (1976). Scheduling nursing personnel according to nursing preference: A mathematical programming approach. *Operations Research 24*(5), pp. 842–856.

[52] Wong, G. Y. C. and A. H. W. Chun (2004). Constraint-based rostering using meta-level reasoning and probability-based ordering. *Engineering Applications of Artificial Intelligence 17*(6), 599 – 610.

[53] Yunes, T. H., A. V. Moura, and C. C. de Souza (2005, May). Hybrid column generation approaches for urban transit crew management problems. *Transportation Science 39*(2), 237–288.