Duty selection: different relaxations for different views

F.L. Niks

October 2, 2009

Summary

In these modern times it is important for a company to plan its processes very well, to reduce its costs as much as possible. One of the possibilities to accomplish this is to optimize the set of duties needed to carry out all the work. This research aimed at providing a solution for this problem with the following goal:
Develop a method to create and select duties quick and in an optimal way, which can be easily implemented in existing planning software.

In this case, a duty is a set of tasks that can be executed by an employee. The main goal is to divide all the needed tasks among different duties, in such a way that the execution of the duties will generate the least amount of costs, while meeting all constraints. This is also called the Crew Scheduling Problem (CSP). There are several different solution methods available to solve this problem. The most common one is to formulate the problem as a so-called Set Covering Problem.

Solving the Crew Scheduling Problem is often divided in two subproblems. First all (or as much as possible) feasible duties are generated. Then the duties with the lowest costs are selected from this set. For the generation process, two methods are proposed: a relatively new one using topological sort, and the more well-known column generation. For the selection process, it is proposed to implement a relaxation method, to implement an algorithm called the volume algorithm, or use an external solver.

It would be perfect if all of these methods could be implemented and tested. However, due to limited resources the research had to be narrowed down. The generator that was developed by Woutersen (2005) is used, so the focus shifts more to the selection proces. For this process, it would be interesting to see which relaxation method (method for simplifying the problem) is better when solving the problem as a Set Covering Problem. So the new goal of the research becomes:

Implement an algorithm which solves the Set Covering Problem within the existing duty generator, using the best heuristic based on a relaxation method (either Lagrangian relaxation or surrogate relaxation).

In simple terms, solving the Set Covering Problem comes to finding the set of columns (duties) which generate the lowest amount of costs while covering all the rows (tasks). It is a very versatile model, and also easy to solve. But the Crew Scheduling Problem can also be written as a Multidimensional Knapsack Problem. A closer look on both the (Generalised) Set Covering Problem and the Multidimensional Knapsack Problem reveals some interesting points. In fact, both models are almost the same. But for each view point, different relaxation methods have been developed.

A relaxation of a problem is a more simplified version of a problem, with an optimal solu-

tion value at least as small as the original problem (in case of an minimization problem). There are several methods to construct a relaxation. With Lagrangian relaxation, some or all the constraints are put in the objective function with some weight values. It is a popular method to use for relaxing the Set Covering Problem: when almost all constraints are put in the objective function, then solving the Lagrangian relaxation becomes trivial. Surrogate relaxation rather compresses some or all the constraints into a single one. This is where it is useful to compare the Set Covering Problem with the Multidimensional Knapsack Problem. The surrogate relaxation of a Multidimensional Knapsack Problem reduces to a Singledimensional Knapsack Problem, for which a number of good solution methods are already available.

An algorithm which solves the Crew Scheduling Problem is presented, which is based upon the algorithm described in Caprara et al. (1999). Some modifications were made, such as using different parameter values and stopping criteria. During one step of the algorithm, a relaxation is solved. Solution methods for solving a Lagrangian relaxation as well as a surrogate relaxation are implemented to see which relaxation method generates better results.

This algorithm was tested on the same test cases as used by Woutersen (2005). The setup of these tests were mainly focused on finding a good answer in a relatively short amount of time. The results of the Lagrangian relaxation tests, the surrogate relaxation tests as well as the optimal solutions (found by Woutersen (2005)) were compared to each other.

When it purely comes to the final results, then Lagrangian relaxation proves to be the best one. However, the starting solution used in the algorithm performed unexpectedly well, sometimes being better than both relaxation methods. The algorithm itself needs further research: the quality of the solution varies with each case, and it has problems with handling larger and more complex problems. But this research did show the first signs that the algorithm has potential to be a good alternative.

Contents

Chapter 1

Introduction

In these modern times it is important for a company to plan its processes very well. When the planning has been done well it will reduce the costs as much as possible. And that is crucial, considering the fierce competition that is present in a lot of sectors. Less costs means more profit, and/or the ability to offer a lower price than the competition does. A lot of companies are therefore searching for software which is able to optimize there business processes as well as possible, in different areas.

An important part of these costs are caused by the personnel. As a matter of fact, for many companies personnel costs are the largest costs in their administration. Assuming that a company is not able to outsource their activities to countries which have lower salary rates than the country they are currently operating in, it is then favorable to reduce the amount of employees as much as possible. One of the possibilities to accomplish this is to reduce the amount of duties needed.

1.1 The importance of planning personnel quickly

Planning of personnel can be done in several ways. One extreme situation is that a team of planners is trying to solve the entire puzzle manually. The opposite extreme situation is that a computer system is optimizing everything, and preferably in real time as new information becomes available. This is often referred to as the 'black box' solution. The first situation is clearly not efficient. A computer is able to process all data much quicker. However, in many cases it is not desirable to let the computer do everything. Aside from political reasons (planners will get fired for example), the persons responsible in the company must have complete faith in the system. And that is something most people lack, even in this modern IT age.

A mixture of these situations is therefore often applied. For example, a black box solution is still implemented, but the users are able to add so-called overrules. With these, the user is able to change some parts of the final solution or even the input of the method. This is a good solution if the system is pretty trustworthy, but sometimes cannot be accurate because of irregular data (for example extra demand for personnel due to some special event).

Another example of a mixture is to let the system solve the problem in a short amount of time, and let the planner judge the quality of the solution. Then the planner can change the problem formulation, for example by adding or removing a task. The problem will then be

solved again. After solving several of these scenarios, the planner can choose which one is best suited.

For this approach it is important that the system is proposing a solution within a small amount of time. If a planner is optimizing the schedule for the next day, the system should not require several hours to solve the problem once. This will even not be necessary in most of the cases. For a lot of problems much time is needed to find an optimal solution, or to verify that the solution found is indeed an optimal one. But just as often a 'good' solution is found very quickly. In this case, a 'good' solution is a solution with costs acceptable close to the optimal solution value. With accepting such solutions, a lot of time can be saved.

1.2 The search for a solution to plan the amount of personnel

To be able to offer an instrument for this last mixture, this research was started. A similar research (but without the focus on planning quickly) has been done by Woutersen (2005). She has done some research on the Crew Scheduling Problem: generating duties in an optimal way. This generation process consists of two main steps: generating a set of feasible duties, and then finding the best subset. This subset must still comply to all constraints and demands, but with minimum costs. Since Woutersen (2005) already did research on workstation demands, this research focuses more on task based demand.

A second aspect which shall be investigated is the design of the algorithm itself. Woutersen (2005) used CPLEX, an external program, for selecting the subset of duties. The main advantage of CPLEX is that it is able to find the optimal solution, guaranteed (as long as the user does not have set constraints such as maximum computation time). A disadvantage however is that it always will be an external system, outside of the system the planner is really using to optimize the business processes. Not only interfaces need to be created for the communication between both systems, but additional licenses must also be acquired. If the method can easily be implemented within the planning software, the maintenance to the entire system will be less cumbersome, and owners of the software do not have to buy licenses from more than one party.

This gives us the initial goal of the research:

Develop a method to create and select duties quick and in an optimal way, which can be easily implemented in existing planning software.

And so the research was started. However, at one point it turned out that the research would become too extensive. Not only the method would become a combination of two separate sub methods (one for the generation of the duties, one for the selection part), but there were also several alternatives available for each of these sub methods. It would simply take to much time to implement and compare all these alternatives. So it was decided to reduce the research to implementing and testing only one alternative for only one sub method.

This brings us back to the inspiration of this research, the work of Woutersen (2005). This research was no longer just inspired by that thesis, it has become a continuation for several reasons:

- A basis has already been implemented. Duties can be generated by the method of Woutersen (2005). This duty set will now be the input for the method to be developed.
- The same test cases can be used to test the new method. The main advantage is that optimal solutions are already available. These would serve as a good reference for the performance of the new method.

The new elements of the research are:

- During the first part of this research two relaxation methods have been found: Lagrangian relaxation and surrogate relaxation. Because only one algorithm will be implemented, in which both relaxation methods can be integrated, the research can now concentrate more on comparing both relaxation methods. Ideally, we would like to be able to conclude which one will be more useful for these kind of planning problems.
- The research retains its focus on developing a method that can be easily implemented within existing planning systems. To be more precise: the research will conclude if it is wise to replace CPLEX by another method within the currently existing duty generator.

The final goal of this research has now become:

Implement an algorithm which solves the Set Covering Problem within the existing duty generator, using the best heuristic based on a relaxation method (either Lagrangian relaxation or surrogate relaxation).

1.3 Outline of the thesis

Chapters 2 and 3 are about the initial research goal. In chapter 2, some explanation and examples are given about task based demand. In chapter 3 the first steps are taken towards developing an algorithm to generate such duties.
Chapter 4 is all about the change in the research. Considerations are explained, as well as the assumptions that are made before continuing.
All research considering the final research goal is described in chapters 5, 6 and 7. Chapter 5 shows that the Set Covering Problem can be viewed in more than one way. Chapter 6 then shows that for each of these views there is a proper relaxation method: the Lagrangian relaxation and the surrogate relaxation. A method to solve the Set Covering Problem where both relaxation methods are used is described in chapter 7.
The research will be wrapped up in the final chapters. All test results are being discussed in chapter 8, and then conclusions are made in chapter 9. Finally, chapter 10 shows some recommendations for continuation of this research.

Chapter 2

Task based demand

As indicated in the previous chapter, this research has set its focus on task based demand. Therefore it is of importance to first explain the concept of task based demand. What exactly is task based demand? How are minimization problems concerning task based duties solved? And in what areas can task based demand be found?
In an article written by Ernst et al. (2004) answers are given to these questions among others. This chapter therefore highlights some relevant aspects from this article, which discusses task based demand. Section 2.1 starts by giving the definition of task based demand. In section 2.2 solution methods are proposed, which can be used to find the most efficient set with duties based on tasks. Finally section 2.3 shows some application areas where task based demand plays an important role.

2.1 Modeling task based demand

First of all, it is important to know what task based demand is. Ernst et al. (2004) describe it very well. Task based demand is the demand of a company for workforces. In this case the demand is derived from the list of tasks or jobs the company must carry out.
The most basic properties of a task are

- the time window: tasks may only start after and must end before the given time
- the duration: the amount of time needed to complete the task.

A note on the duration: the duration does not need to be equal to the length of the time window. If the duration is smaller, then the execution of the task may start later than the begin time of the time window. If the duration is longer, then it is not possible to finish the job within the time window. One has to either accept the fact that the task cannot be completed in time, or adjust the time window (which is recommended in order to get a correct planning of all tasks).
Next to these basic properties, a task may also have several others. For example specific skills are needed for the employee who is going to carry out the task. Or the fact that the task can only be carried out on a certain location. These properties vary from task to task.

So how are all these tasks divided among the employees? That can be done by assigning duties to employees (also found in literature as shifts or pairings). A duty is a set of tasks that can be executed by an employee (or several employees). Of course, there are a lot of constraints to take into account when forming these duties. Most of these constraints come from labor

agreements with the government. Examples of these constraints are the minimum and maximum length of a duty, constraints considering breaks and the fact that there must be enough employees to carry out all the duties.

The main goal is to divide all the tasks among different duties, in such a way that the execution of the duties will generate the least amount of costs, while meeting all constraints. Most of the times, this will be the same as minimizing the number of duties. The reason for this assumption is that for most companies personnel costs are the highest cost (especially when one only takes costs into account that are associated with the tasks directly). Usually, one employee is needed for each duty. So the number of employees increases when the number of duties increases. This means that the costs for hiring one extra employee must be paid when one extra duty is added. The conclusion is that it is profitable to reduce the number of duties as much as possible. This problem (generating the set of duties with the least amount of costs while meeting all constraints) is called the Crew Scheduling Problem (CSP) or the Crew Pairing Problem.

The Crew Scheduling Problem is often solved in two steps.
In step one, a set of feasible duties is generated. In the most ideal situation, this would be the set of all feasible duties (a feasible duty is a duty where all constraints regarding this duty are satisfied). During the explanation of step two, it becomes clear why this is the most ideal situation. Unfortunately, this set will often be too large to handle. So a trade off must be made between the size and quality of the set and the effort needed for creating and investigating this set.
In the second step, one selects the duties from the generated set such that all tasks are being carried out, and the costs for executing this combination of duties are minimal. As mentioned before, it is unlikely that after the first step all feasible duties are generated. This implies that the optimal set of duties might not be available in the second step at all. That is why one must make the trade off in the first step. When the set of duties is larger and has better quality, the chances of a better solution of the Crew Scheduling Problem will be higher. One can overcome this problem by accepting the solution as it is now, if it is good enough. If not, then step one and two must be carried out again, this time generating more duties that haven't been generated yet.

Next to task based demand, Ernst et al. (2004) also explain flexible demand and shift based demand or workstation occupation demand. To give a short explanation of the latter: workstation occupation demand is basically the same as task based demand. The difference is that the company now has a number of workstations, which need to be occupied within a time window. Woutersen (2005) has based her thesis on this workstation occupation demand.

2.2 Solution methods

To solve a Crew Scheduling Problem one can choose from a large variety of different solution methods. To give an impression of the available options, several of them are introduced in this section.

Mathematical programming approaches Mathematical programming approaches are generally able to achieve the solutions that have the lowest costs. However, according to Ernst et al. (2004) there are difficulties that don't make it easy to apply such an approach to an arbitrary

problem:

- In many cases a method called column generation is applied. The danger of this method is that the complexity of the problem will be hidden in the definition of the columns. The result is that a part of the method (the so-called pricing problem) becomes hard to solve. Often, a heuristic must be used to solve the problem. Otherwise solving the pricing problem will take to much resources (such as time or computer memory). This means that the advantage of solving the master problem with an exact method will be lost.

- The number of constraints and objective functions that can be formulated using a mathematical approach is somewhat more limited compared to other methods. The result is that these approaches are used to solve problems with only a few complications, or simplified versions of real-life problems.

- Using and implementing a good mathematical programming approach is relatively hard and costs a lot of time. That is why it is only useful to use such a method when the profits of using the method is worth the extra effort.

The Set Covering Problem The most commonly used mathematical model is the Set Covering Problem. It is also one of the formulations used by Woutersen (2005), and (most importantly) it is the formulation that is used during this research. For this reason a more detailed description including mathematical representations will be given in chapter 5, but a brief introduction will be given here.
The Set Covering Problem is described for the first time by Dantzig (1954). He discussed the problem on how to make schedules for the personnel at some toll booths. The Set Covering Problem can roughly be described as follows: the duties and the tasks that need to be carried out are stored in a matrix. The rows represent the tasks, and the columns represent the duties. It is said that a column (duty) 'covers' a row (task) when the task is part of that duty. The goal of the problem is finding the subset of columns, such that the costs of using those duties is as low as possible while all tasks are being carried out. It appeared that this formulation was so generally, that many problems in crew rostering and crew scheduling could be dealt with this approach. Another advantage is that the formulation can be 'customized' for real life situations by adding some side constraints (which has been done by Woutersen (2005)).
Unfortunately, problems tend to become too large to handle, even with modern day technology. A nice example of this is given by Teeuw and Woutersen (2003) where the duties used as input for the algorithm were created randomly. There are two possibilities when one needs to handle large problems:

- Only a limited set of duties is created, either randomly, using intuitive criteria or iteratively (form a small set of duties, solve the problem, add more duties, solve the problem and so on). This results in a formulation with a reasonable size to handle. This is a heuristical approach.

- Partially creating all possible duties with the use of the previously column generation approach. The idea is to generate 'good' columns to form a more restricted version of the problem to be solved. Column generation is generally an exact method.

Other mathematical programming approaches Besides formulating a Set Covering Problem, there are some other mathematical programming methods as shown by Ernst et al. (2004). When one needs to deal with more than one objective function, *goal programming* is a

good approach to use. *Stochastic programming* can be used if one has to deal with stochastic demand (demand that cannot be forecasted exactly). For airline crew rostering, *decomposition techniques* have been developed. These techniques decompose the main problem in subproblems in various ways. Then, each subproblem can be solved using different mathematical programming approaches if necessary. Some other approaches mentioned by Ernst et al. (2004) are *network models*, *dynamic programming* and *matching models*.
The approach that has been used by Woutersen (2005), next to the Set Covering Formulation, is the Implicit Formulation. Here, the decision variables are not defined as the number of times a certain duty is selected, but as the number of times a certain starting of finishing time is selected. Through the constraints and objective function, a starting time will be linked to a finishing time, thus creating a duty. This formulation requires less decision variables, but more constraints are needed.

Other solution methods Mathematical programming approaches are popular, but they are not the only possibilities. Ernst et al. (2004) discuss several other approaches that one can use for solving the Crew Scheduling Problem.
Methods that belong to the *artificial intelligence* approaches are fuzzy set theory and decision support systems. The latter do not completely solves the problem, but rather assist the planner. They are useful when there are a lot of human factors that cannot be formulated into the problem. It also provides a first step toward automatic planning if the planners have no trust in a 'black box'.
Constraint programming is particular useful when the problem contains a high amount of constraints, or when any feasible solution suffices and doesn't need to be optimal. Constraint programming is however not a good method to use when the solution does need to be (near) optimal. To overcome this, there has been done some research about combining the flexibility of constraint programming with other optimisation techniques.
Metaheuristics are a popular class of solution methods. They can be used to solve problems that are already difficult by themselves (for example combinatorial/discrete optimisation problems), or for solving real-life problems which became too complicated to solve with an exact solution approach. Metaheuristics are derived from a variety of fields: classical heuristics, artificial intelligence, biological evolution, neural engineering and statistical mechanics. A few examples of metaheuristics are simulated annealing, tabu search, genetic algorithms, neural networks and machine learning. These methods gained their popularity for a number of reasons including:

- These methods are relatively robust. It can not be guaranteed that they will find an optimal solution, but usually they produce a good solution in a relatively small amount of time. To show why this is an advantage: with many integer programming approaches a feasible solution may not be returned for a long time.

- Most metaheuristics are simple to implement. On top of that, they are able to deal with and exploit problem specific information.

- (Meta)heuristics are able to deal with complex objective functions.

That is why these metaheuristics are generally chosen when mathematical problems are not able to solve real-life rostering problems. However, if the problems are highly constrained, then constraint programming is a better choice.

2.3 Application areas

Ernst et al. (2004) show a lot of different application areas where crew scheduling is playing an important role. A selection of these areas (where planners have to deal with task based demand) will be given in this section.

Transportation systems If one considers transportation systems, all of them have two things in common:

1. Temporal as well as spatial features are involved. This means that each task has a starting and finishing location next to the time window.

2. All tasks that need to be performed by the employees are derived from a given timetable.

For airlines, crew scheduling is very important. There, personnel costs are the second biggest costs. Because these costs have such an huge economical impact in this area, a lot of research has been done for scheduling the crew of an airline. An example is the research done by Arabeyre et al. (1969) about the solution methods used at different airline companies. Klabjan et al. (2001) have developed a random method. They also used a network of computers to utilize parallel programming (which should reduce the computation time).
Of course, research is also done for employees of a train or bus company, or any company who delivers another method of transportation. Freling et al. (2001) did some research for a train company. They have developed a branch-and-price algorithm, which (with a few modifications) can also be used in other fields of transportation systems like airlines or bus companies.

Venue management For venue management, locations do not play much of a role, since all tasks need to be carried out on the same location. The difficulty here lies in the various skills that are needed for the tasks. Each task needs a specific skill, and each employee has certain skills. The problem now extends with the question of which employee is able to carry out which task. Examples can be found in the airline industry again (customs staff, aircraft refueling and ground staff including baggage handlers), but also in the world of sports (scheduling umpires in the American Baseball League and for cricket games in England).

Hospitality and tourism Crew scheduling in hotels, tourist resorts and fast food restaurants could be seen as the combination of crew scheduling for the application areas mentioned above. Considering a hotel, staff costs (including costs for training) is often the highest costs it has for a single expense (sometimes it makes for more than 30% of the total costs). Employees need to have certain skills, since there are so many tasks in a hotel that need a specific skill (such as catering, housekeeping, reception duties and maintenance). Hence the costs for training if someone doesn't have the right skill yet. Next to these difficulties, a hotel also has to deal with uncertain demand. Since it is not known in advance how many guests will stay at the hotel, it is also unknown how many employees are needed. All this makes crew scheduling a crucial part of the management planning.

Postal services Another good example of an application area (not mentioned by Ernst et al. (2004)) is the postal services. Lansdorp (2002) and Teeuw and Woutersen (2003) have done a research about crew scheduling for a postal service company. Teeuw and Woutersen (2003) explain well what the problem for this company looks like. There are four types of tasks that need to be done: unloading a truck with mail, presorting the mail on district level, sorting the

mail according to the order in which it will be delivered and the delivery itself. An interesting aspect of this problem is that these tasks need to be done in this order: it is of course impossible to deliver the mail if it is still in the truck, and the delivery is a lot harder when the mail is not in the right order yet. Skills are also an issue here. Postmen without a drivers license are not allowed to deliver parcels that are too large to fit on a bike. Even the location is a factor that may not be overlooked. It is far from efficient to let a postman first deliver mail on one side on the city, then let him deliver mail on the other side of the city. All these constraints make the problem for postal services an interesting one, and therefore it was kept in mind while the algorithm described in the next chapter was developed.

Chapter 3

An algorithm for generating task based duties

To offer a new tool for solving the Crew Scheduling Problem with task based demand, the search for an algorithm was started. Since many successes are made with the Set Covering Problem formulation, the algorithm would be designed to solve such a problem. Another important property the algorithm must have is that it should be easy to implement in existing planning software (as stated in the goal described in section 1.2).
The algorithm will consist of two main steps, which is explained in section 3.1. Then, sections 3.2 and 3.3 will each show different methods that can be used for each of these steps.

3.1 Outline of the algorithm

As described in 2.1, the Crew Scheduling Problem is often divided into two subproblems. The Set Covering Problem is a kind of formulation that enables this. First, all columns and rows of the problem are generated. Second, the most profitable rows (which represent duties) are selected from the generated problem. This has been done by for example Teeuw and Woutersen (2003).
Therefore, this algorithm consists of two main steps. In the first step, the columns of the Set Covering Problem are generated. When this is translated to the Crew Scheduling Problem, it means that in this step the duties will be generated that are needed to carry out the tasks. Probably, not all of the columns generated are necessary to cover all the rows. This is where step two comes in. In step two of the algorithm, the optimal set of columns will be selected. The vital criterion this set must meet is that every row must still be covered by at least one column. The goal here is to select that set of columns, where the value of the objective function (the function that determines the costs of a solution) is as low as possible while fulfilling this criterion.
For both steps, a separate algorithm can be used that is designed to solve the relevant (sub)problem in an efficient way. So, several candidate methods for each step have been researched. These methods are described in the sections below.

3.2 Step one: generating the columns

For this step of the algorithm, several methods that are specifically designed to generate the columns efficiently are investigated. The most ideal situation is that the method used is able to

generate all feasible columns (as indicated earlier in chapter 2). This guarantees that if in step 2 the best solution is found, then this solution is indeed optimal. After all, there is no feasible column left that might be able to improve the solution.
Unfortunately, in reality it is impractical or even impossible to compute all these columns. The constraints of a Crew Scheduling Problem may allow for a large amount of feasible duties. For example, when labor agreements are not very strict, or when there is a great variety in times at which a duty may start. The impact on the algorithm will be that it will take a large amount of time, in most cases more than is desired. Another bottleneck is formed by the computer system resources. If both steps of the algorithm are executed separately and step two is executed right after step one, then all the duties must be stored in the computer system's memory. If this memory is not large enough for holding the columns, out of memory exceptions are the result. These will seriously slow the system down or even end the program.

Due to the practical limitations, it becomes a challenge to find a method that generates only (the best) part of the feasible duties. Generating duties randomly is a possibility, but if the algorithm is able to find duties which have a high chance of being selected in a good solution, the method in step two will be able to find better solutions with less computation time. Another approach one can take is to iterate between both steps. First, a small set of columns is generated, and then the best solution within this set is computed. The algorithm then returns to step one to add some more columns to the already generated set. With the larger set, step two is entered to see if a better solution can be found. This way, step one and two are repeated until no better solution can be found or the algorithm runs out of resources.

In the following subsections short descriptions are given about each method that was investigated.

3.2.1 A method based on topological sort

In an early stage of this research, the development for a new method for generating columns was started. The new method would be based on the concept of topological sort. This concept has been used by Kisteman (2004) to solve a Vehicle Routing Problem. The main constraints that were considered (time windows and other types of dependency relations) made it possible to create a directed acyclic graph of the places that need to be visited. From here, a tree with all possible trips could easily be derived. A follow-up of this research was done by Niks (2005). The same problem was solved, but a statistical clustering method was used as a kind of preprocessor. Each cluster created contains the places which should be visited within a trip. For each cluster, the algorithm of Kisteman (2004) was used to set the places in the right order to visit. To incorporate the dependency relations, a new concept called *conditions* was developed. A condition between two stops defines either of the following. Both stops must be together in the same trip. A trivial example is the origin and destination relation between two stops. Another possibility is that two stops may never be visited during the same trip. This happens when it is impossible to travel between two places within the given time windows.

The ideas used in both researches might also be used for solving the Crew Scheduling Problem, since both problems show some similarities. A task can be seen as a stop. Both have a time window in which the task must be executed or the stop must be visited. Also, both entities have a duration (the time it takes to execute the task or how long one needs to stay at the stop for loading and unloading). Something else both problems share are travel times. If the tasks need

to be done at different locations, traveling time needs to be taken into account. The traveling time between two tasks performed at the same location is simply defined as zero. Section 2.3 shows that tasks may also have other dependency relations between each other.

Although the development of this method is far from complete, a first concept was created. The method is a modified version of the topological dependencies algorithm from Kisteman (2004).
The duties are formed using breadth-first search. Contrary to depth-first search (where all duties are generated simultaneously), duties are formed one-by-one. This has the advantage that a time limit can be used. If a certain amount of time has passed, the algorithm can stop. The tree now contains a set of completed duties, which the main algorithm in step two can use. If the main algorithm re-enters step 1, the algorithm can continue where it left, trying to finish the tree.
For extra control, conditions are used in a new idea called conditional subsets. For some tasks, it is not allowed to execute them before certain requirements are met. A good example of these type of tasks are breaks. Since labor agreements often state that an employee may take a break after several hours, it is best to plan them within the duties. Because of the rules stating when a break may be 'executed' those breaks are put in a conditional subset. At the start of forming the duty tree, the algorithm is not allowed yet to access the conditional subset with breaks. Only when a branch reaches a certain point (where the duty is already several hours long, and the employee may take his break), the conditional subset becomes accessible. From there on, the algorithm may add a break to the duty, up to the point where other regulations say that no more breaks may be added.

The method is not free of problems. The algorithm was initially developed for generating all feasible duties at the same time. Unfortunately, as mentioned earlier, it is not wise to use such a method. Adjustments must be made for the method to be able to deal with generating subsets of duties. One of the ideas was to use a combination of breadth-first search and depth-first search. This way, more diverse duties will be generated from the beginning, so hopefully the more useful duties will be found in less time. As mentioned before, iterating between step one and two of the main algorithm could also be a good idea. To reduce the computation time of the main algorithm, it is convenient to generate a 'good' subset. In step two, this subset should result in a good solution. The problem now is how to define a 'good' subset. It is difficult to tell beforehand if a subset is 'good' or not. This question could take an entire research by itself.

3.2.2 Methods involving a master problem

Step one can be summarized simply in the words "column generation". However, in literature there is a group of methods which also is referred to as "column generation". The idea of these "column generation" techniques is that it might not be necessary to generate the entire constraint matrix. For this, it is necessary that the problem can be rewritten into a main problem (the master problem) and one or more subproblems that are relatively easy to solve. The master problem contains the decision variables that have been generated and the constraints that are related to them. In each step, the subproblems are solved to determine which decision variable will improve the current best solution of the master problem. This one will be added to the master problem. These methods are very useful when the amount of columns becomes very large compared to the amount of rows. "Column generation" methods are described in more detail by Wolsey (1998) and Wagelmans (2005).

3.3 Step two: selecting the right duties

The second step of the main algorithm takes the subset created in the first step, and tries to find the optimal solution of the problem given this subset. As stated in section 3.1, a solution is optimal if the objective or cost function is as low as possible, while all rows are covered by the selected columns.

3.3.1 Relaxation algorithms

A class of algorithms that has been used a lot for this problem is the class of relaxation algorithms. These algorithms typically follow a simple scheme. The trick is to look at a so-called relaxation. Relaxations are simplified versions of the main problem that is going to be solved, in the sense that relaxations have less constraints that must be taken into account. A relaxation takes less effort to solve compared to the original problem (otherwise, there is no use in formulating a relaxation). In general, the solution of the relaxation will be infeasible. So it is necessary to use a method which transforms this solution into a feasible one. The relaxations often have multipliers: parameters that indicate how important a certain part of the model is (for example a multiplier may indicate the importance of a constraint). At the end of an iteration, these multipliers are updated. The new multipliers will be used to solve the relaxation again, resulting in a different 'relaxed' solution as in the previous iteration. From this new infeasible solution, a new feasible one is derived, which is followed in updating the multipliers again. The point of updating the multipliers after each step is to find solutions of the relaxations which lead to better feasible solutions for the main problem. The algorithm stops when the stopping criteria defined beforehand are met.

Relaxation algorithms belong to the class of heuristics. They have all the benefits from (meta)heuristics (recall paragraph 2.2), such as good solutions that are generated in a reasonable amount of time. They are also easy to implement. This is why a relaxation algorithm would be a good candidate method.

3.3.1.1 Algorithms using Lagrangian relaxation

Most of the relaxation algorithms for solving the Set Covering Problem are based on Lagrangian relaxation. Lagrangian relaxation is a method to simplify the problem by taking some or all of the constraints and putting them in the objective function (preceded by a multiplier). The costs of these constraints are such that the objective value will be worse when the constraints are violated, and better if this does not happen. For updating the multipliers, so-called subgradient optimization is often used.

More information about Lagrangian relaxations and subgradient optimization can be found in chapter 6. An example of an algorithm using Lagrangian relaxation is developed by Caprara et al. (1999). This algorithm promises good results when solving a Set Covering Problem, as is also shown by Teeuw and Woutersen (2003).

3.3.1.2 Algorithms using surrogate relaxation

Another method of relaxing a problem (next to Lagrangian relaxation) is surrogate relaxation. Surrogate relaxation is still relatively unknown. While Lagrangian relaxation 'absorbs' the constraints into the objective function, surrogate relaxation rather 'compresses' the constraints. The

constraints are all added up (with a certain weight), so the result will be a problem with only one constraint left. In general, a surrogate relaxation problem takes more effort to solve than a Lagrangian relaxation problem. The advantage however is that algorithms using surrogate relaxation tend to converge faster. This means that less iterations are needed. If the number of iterations is small enough, savings will be made on the total amount of computation time.

An application of surrogate relaxation within a heuristic for Set Covering Problems is described by Lorena and Lopes (1994). They used the same test cases used by Beasley (1990), according to Lorena and Lopes (1994) one of the best known heuristics for solving the Set Covering Problem. Their results were comparable in terms of bounds, but they found the solutions within about half the computation time. This is due to the fact that their method converges much faster, so less iterations were needed. These promising results make it a method that could be used in step two of the main algorithm.

3.3.2 The volume algorithm

The volume algorithm is a relatively new algorithm developed by Barahona and Anbil (2000). It is used by Barahona and Anbil (2002) to solve a Set Partitioning Problem. A Set Partitioning Problem is a special case of a Set Covering Problem. Rows must now be covered by exactly one column. Therefore, the Crew Scheduling Problem is actually a Set Partitioning Problem. In real life, tasks can only be part of one and only one duty. It doesn't make sense that a task (that must be done exactly once) is executed by multiple duties. The reason why the Crew Scheduling Problem is modeled as a Set Covering Problem most of the times, is that the Set Covering Problem guarantees the existence of at least one feasible solution. Set Partitioning Problems may become too strict for a feasible solution to exist (Teeuw and Woutersen (2003)).

Barahona and Anbil (2000) have based their algorithm on what they call the subgradient algorithm. A note must be made that this algorithm is not *the* subgradient algorithm. In fact, it is a Lagrangian relaxation algorithm which uses subgradients to update the multipliers. According to them, finding a feasible solution for the main problem is not part of the algorithm, since another method (for example a combinatorial method or solving a smaller linear programming problem) must be used in order to find feasible solutions for the main problem. Another drawback is that the subgradient algorithm lacks a well defined stopping criterion. To counter these drawbacks, they have developed an extension to this subgradient algorithm called the volume algorithm. Barahona and Anbil (2002) showed good results for this algorithm when it is used to solve the standard Set Covering Problem, and it seems easy to implement (just as with regular subgradient algorithms).

3.3.3 The use of an external solver

The fourth alternative was to use a solver, as was done by Woutersen (2005). However, this contradicts with the main goal of the research, which is developing and implementing an algorithm which makes the use of such a solver unnecessary. Also, it is expected that the exact method used by this solver will take extra computation time. Just as with other methods as heuristics, this problem can be compensated by restricting the resources of the solver (for example by limiting the allowed maximum computation time). The direct consequence is that the solution will not be guaranteed optimal anymore, since it is unknown if a better solution could be found if the solver was allowed more time. With this, the exact method loses its strength. An advantage

is that implementations together with column generation were already available. The problem then becomes how to adjust these implementations, so that the Crew Scheduling Problem can be solved with it.

It was not the intention of this research to investigate all the methods described in this chapter. There simply was not enough time to do this. Therefore, in the following chapter, choices will be made to narrow down the research.

Chapter 4

Considerations and assumptions

In the previous chapter the first steps were taken in developing an algorithm and its implementation for solving the Crew Scheduling Problem. Several alternatives for both main steps have been presented. Now, a decision must be made on which combination is going to be implemented. The main problem becomes what can be done in the time available.
How the research is narrowed down will be explained in section 4.1. Then in section 4.2 assumptions are made for the remainder of the research. Then for readability, some definitions for basic terms are given in section 4.3.

4.1 Narrowing the research

Beforehand, it is unknown which combination of methods is best to use in the algorithm. Naturally, it is hard to make predictions for the new algorithm. Other methods (column generation and the methods based on relaxations) have been use successfully for similar problems, but this does not imply that they will also be useful for this particular problem. And while using an external solver is not preferred, it is possible that it will still be the best alternative: the solver generates a solution of such a quality compared to other methods, that it is worth the extra costs and computation time.

It would be perfect for the research if all of these methods could be implemented and tested. Then, based on the results, it can be concluded which methods must be used in the final implementation. But implementing new methods or adjusting existing ones consumes time, and time was limited. It was even concluded that there was not enough time available to implement one method for each of the main steps! So choices were made, and the outcome was that the research had to be narrowed down.

First of all, the idea of developing an entire new duty generator has been dropped. Instead, the starting point will be the generator that was developed by Woutersen (2005). This has the advantage that step one can be neglected. The generation of the duties will be done entirely by the method used for the existing generator. This means that all feasible duties will be generated. The tests showed that for the test cases used, the time and memory needed are not significantly large. Another advantage is that the same test cases can be used. Because all the results from Woutersen (2005) are optimal since an exact method was used, all the results from the current research can be compared with the optimal solutions.

What remains is finding a good set of duties (step 2). It is this part that will be redesigned by

using a heuristic instead of the solver. For this, a method will be implemented which uses either Lagrangian relaxation or surrogate relaxation. Which one is better is hard to tell. Lagrangian relaxation is widely used in such situation. However, surrogate relaxation converges faster, so the best solution will be reached faster (and perhaps will be better too).

Therefore, the new goal of the research becomes:

Implement an algorithm which solves the Set Covering Problem within the existing duty generator, using the best heuristic based on a relaxation method (either Lagrangian relaxation or surrogate relaxation).

4.2 Some assumptions

Because this thesis is a continuation of the work of Woutersen (2005), almost all assumptions made there are also valid here. The most important of these assumptions are:

1. The type of Crew Scheduling Problem that will be solved is the one based on workstation occupation demand. The demand is now based on how many employees are necessary at a workstation at a moment. It is possible to convert task based demand to workstation occupation demand, however some information about the tasks will be lost.

2. Problems which do not have workstation occupation demand 24 hours a day, will be solved per day. In situations where workstations need to be occupied all the time, the problem will be split (if necessary) in several subproblems. Each subproblem has a time interval of no more than a week.

3. Also, it is assumed that each problem is related to a single workstation. If there is demand for several workstations, the problem will be split for each workstation.

4. There are a lot of possible constraints for the Crew Scheduling Problem. The ones that can be handled by the duty generator are:

    - Basic constraints regarding the length of the duties (minimum and maximum length, the length must be a multiple of a certain time interval and some lengths are forbidden).
    - Duties may not start or end within a specific time interval on a day or during an hour.
    - Each duty will have a maximum of one break. There must be a certain amount of labor time before and after the break.
    - There is a minimum and maximum number of duties with a specific length that start during an interval.

There are some extra assumptions that must be made. Woutersen (2005) uses two different models to describe the problem. Here, only one of them, the Set Covering Formulation, is considered. The techniques used are designed for the Set Covering Formulation, and cannot be applied to the Implicit Formulation.

One of the constraints stated above is not used. It was not possible to model the maximum number of duties that start within a given time interval, with a specific length. The reason is that

instead of a $\geq$ sign, a $\leq$ sign must be used here, while everything else must remain the same (see also chapter 5).

4.3 Basic terms defined

From here on, some terms and variable names will return on a regular basis. Those will be defined and explained here.

Sets and indices

M = the set of planning periods

i = a single planning period from set M

N = the set of feasible duties

j = a single duty from set N

The planning periods are defined by Woutersen (2005) as the unit which need to be carried out (similar to tasks). The entire interval where the set of duties must be created for, is divided into planning periods. The length, starting and finishing times are determined by the constraints.

Decision variables and constants

x_j = the number of times duty j has been selected

c_j = the costs associated with the single execution of duty j

$a_{ij} = \begin{cases} 1 & \text{if planning period } i \text{ lies within the interval in which duty } j \text{ is carried out} \\ 0 & \text{otherwise} \end{cases}$

r_i = the number of employees required on the workstation during planning period i

Special sets For notational efficiency, two special sets of duties and planning periods are defined.

$I_j = \{i \in M : a_{ij} = 1$ for each duty $j \in N$

$J_i = \{j \in N : a_{ij} = 1$ for each planning period $i \in M$

Set I_j contains all planning periods that are within the interval of duty j. So, duty j is able to fulfill the demand of these planning periods. Set J_i contains all duties which are able to fulfill the demand of planning period i.

Chapter 5

Different views on the same problem

As indicated before, Woutersen (2005) used two different kind of formulations, of which one of them will be used in this research: the Set Covering Problem. That is why a brief introduction of the Set Covering Problem will be given in section 5.1. However, it appeared that the Crew Scheduling Problem can also surprisingly be viewed as a so-called Multidimensional Knapsack Problem. Section 5.2 gives a brief introduction to the Multidimensional Knapsack Problem. Then section 5.3 shows why the Crew Scheduling Problem can also be viewed as a Multidimensional Knapsack Problem, and why the research still not deviated from the choice of only using the Set Covering Problem formulation.

5.1 Set Covering Problem

A model that would eventually be called the Set Covering Problem was first applied by Dantzig (1954). This article describes a proposal to solve the problem on how to schedule some toll booths, explained by Leslie (1954). The days were divided in a number of periods, and for each period it was determined how many booths were needed. Also, there were several different working patterns (the duties). A toll collector works according to one of these patterns. Because each pattern has one or more breaks which take an entire period, he is available for a booth or not during a given period. An illustration of this idea is given in table 5.1.

Table 5.1: An illustration of the toll booth problem

Pattern	Period t						Total
	(1)	(2)	(3)	(4)	(5)	(6)	
1.1	x_{11}	-	x_{13}	x_{14}	-	x_{16}	w_1
1.2	x_{21}	-	x_{23}	x_{24}	-	x_{26}	w_2
2.1	x_{31}	x_{32}	-	x_{34}	x_{35}	-	w_3
2.2	x_{41}	x_{42}	-	x_{44}	x_{45}	-	w_4
2.3	x_{51}	x_{52}	-	x_{54}	x_{55}	-	w_5
3.1	-	x_{62}	-	x_{64}	x_{65}	-	w_6
-	-	-	-	-	-	-	w_0
Total	b_1	b_1	b_1	b_1	b_1	b_1	N

The periods can be seen as the ”tasks” that need to be carried out. For each period t, at least b_t toll collectors are needed. The patterns show which duties can be assigned to the collectors.

For example pattern 1.1 shows that the collector gets a break in at least period 2 and 5. It is possible however that a collector following a pattern is not needed in a period, even if he should be working. This is reflected by x_{it}, where $x_{it} = 1$ if the collector following pattern i is actually needed in period t and $x_{it} = 0$ otherwise. The constant w_i assures that the collector following pattern i does work at least w_i periods. As can be seen, each pattern is repeated a number of times, since it is possible that more than one collector follows the same pattern. However, each row represents one duty, so it is needed to repeat the patterns several times.

Over time, this proposal has been developed into what is now called the Set Covering Problem. In time, it has become one of the most popular models used. It is easy to formulate and understand, and it always guarantees that a feasible solution exists. In terms of duties and workstation occupation demands, if it seems that a workstation is not properly occupied yet, one can solve this very easily. The most straight forward method is to simply take a duty, where the employee needs to work at the workstation at the time there is a shortage.
The Set Covering Problem is also a very versatile model, which can be adapted by adding restrictions, to make the model more problem specific. Its area of application is not limited to just duties and demands. The model can also be used to determine for example which locations to choose for a service center or factory to serve all the regions. Because of its versatility, the model has been widely researched. Basic information about the Set Covering Problem can be found in Wolsey (1998), references to articles about several applications are mentioned in Ernst et al. (2004).

Standard Set Covering Problem The standard Set Covering Problem is formulated as:

$$\min \sum_{j \in N} c_j x_j \tag{5.1}$$

$$\text{s.t.} \sum_{j \in N} a_{ij} x_j \geq 1 \text{ for all } i \in M \tag{5.2}$$

$$x_j \in \{0, 1\} \quad \text{for all } j \in N \tag{5.3}$$

Equation 5.1 shows the objective function. Clearly, the objective is to minimize the total costs of using the selected duties. Equation 5.2 shows the constraints regarding the planning periods. It makes sure that at least one duty will be selected which is able to fulfill the demand of a planning period. One can see that r_i (the number of employees required on the workstation during planning period i) is not used here, because for the standard Set Covering Problem it is assumed that it is sufficient to fulfill the planning periods only once (hence $r_i = 1$ for all $i \in I$). It is also assumed that duties can only be selected once or not at all.
Often, in literature the matrix representation is used. In this case, there are rows and columns. Rows represent the units that need to be served or carried out (in this case the planning units) and the columns represent the units that are able to serve or carry out those units (the duties). So often, when considering Set Covering Problems, one speaks of ”finding the set of columns which generate the lowest amount of costs while covering all the rows”.

Generalised Set Covering Problem The problem solved by Woutersen (2005) is however not a standard Set Covering Problem. As stated before, it is easy to adapt a Set Covering Problem, so it would fit a specific problem better. This has been done here. The main differences are

that in the generalised problem, the binary variable x_j and the binary constant r_i have become (positive) integers. The generalised Set Covering Problem is therefore formulated as:

$$\min \sum_{j \in N} c_j x_j \tag{5.4}$$

$$\text{s.t.} \sum_{j \in N} a_{ij} x_j \geq r_i \qquad \text{for all } i \in M \tag{5.5}$$

$$x_j \geq 0 \text{ and integer for all } j \in N \tag{5.6}$$

It is extremely important to take this change into account when developing an algorithm. That is because now bounds on the decision variables are gone. One of the consequences is that the number of possible solutions has increased tremendously. These seemingly small changes have a huge impact on how to solve the problem.

5.2 Multidimensional Knapsack Problem

There is another formulation one can use for solving the Crew Scheduling Problem. It will be shown that the problem can be written as a Multidimensional Knapsack Problem.

Detailed information on knapsack problems and how to solve them is given by Kellerer et al. (2004). There, it is shown that knapsack problems are useful in situations where choices have to be made. The following example illustrates the main idea of the formulation, and how it gets its name. Suppose there is a traveler, who is packing his suitcase (or knapsack). It is obvious that the traveler can only carry a certain amount of weight, and that the suitcase has a limited volume. Because of these limitations, the traveler can not take all of his possessions with him, so he has to choose which items to put in the suitcase. He now faces the problem of which combination of items has the highest value for the traveler during the voyage, while fitting in the suitcase.

The mathematical formulation of the standard knapsack problem (also known as the Single dimensional Knapsack Problem) is:

$$\max \sum_{l \in L} p_l y_l \tag{5.7}$$

$$\text{s.t.} \sum_{l \in L} w_l y_l \leq c \tag{5.8}$$

$$y_l \in \{0, 1\} \quad \text{for all } l \in L \tag{5.9}$$

with

L = the set of all available items that can be put into the knapsack

l = a single item

$$y_l = \begin{cases} 1 & \text{if item } l \text{ is put into the knapsack} \\ 0 & \text{otherwise} \end{cases}$$

p_l = the profit of item l for the traveler

w_l = the weight of item l

c^{knap} = the capacity of the knapsack

Equation 5.7 shows the objective: make the profit of the items in the knapsack as high as possible. The constraint in equation 5.8 makes sure that the selection does not exceed the available capacity. For the standard problem, it is assumed that there is a single capacity which need to be taken into account, such as weight or volume (hence the problem is called single dimensional). One assumption that must be made is that

$$\sum_{l \in L} w_l \geq c^{knap}$$

otherwise there is no problem at all. If the total weight of all available items together is smaller than the capacity of the knapsack, then one is able to put all of the items in the knapsack, thus acquiring the highest profit possible.

Just as with the standard Set Covering Problem, the Single dimensional Knapsack Problem can be altered to fit problem specific properties. Kellerer et al. (2004) discuss some of those variations, but the two variants that are relevant here are the the (Un)bounded Knapsack problem and the Multidimensional Knapsack Problem. The Unbounded Knapsack Problem lets go of the binary property of the decision variables y_l. Instead of supposing that an item can be taken or not, the choice can be made on how many times the item is put in the knapsack. So, equation 5.9 becomes

$$y_l \geq 0 \text{ and integer for all } l \in L \, . \tag{5.10}$$

In reality, there isn't an infinite supply of objects. Putting a bound b_l on y_l gives the Bounded Knapsack Problem with equation 5.9 replaced by

$$0 \leq y_l \leq b_l \text{ and integer for all } l \in L \, . \tag{5.11}$$

The Multidimensional Knapsack Problem is a variation, where there is more than one restriction on the capacity. For example, the capacity is limited in maximum weight, volume as well as the number of items, which makes the number of capacity restrictions a total of three. The difference with the single dimensional problem lies in equation 5.8:

$$\sum_{l \in L} w_{kl} y_l \leq c_k^{knap} \text{ for all } k \in K \tag{5.12}$$

where

K = the set of different capacity constraints of the knapsack

k = a capacity constraint

c_k^{knap} = the maximum value of capacity constraint k.

Again, the assumption is made that

$$\sum_{l \in L} w_{kl} \geq c_k^{knap} \text{ for all } k \in K$$

for the same trivial argument as with the single dimensional problem. It now is only a small step further towards combining the Bounded and the Multidimensional Knapsack Problems:

$$\max \sum_{l \in L} p_l y_l \tag{5.13}$$

$$\text{s.t.} \sum_{l \in L} w_{kl} y_l \leq c_k^{knap} \qquad \text{for all } k \in K \tag{5.14}$$

$$0 \leq y_l \leq b_l \text{ and integer for all } l \in L \tag{5.15}$$

The next step is to reverse the principle, and reformulate the problem towards a minimization problem. In other words, the objective becomes to take as less as possible. What if the traveler went to climb a mountain? Then he needs certain objects, like warm clothing, tools and food. But carrying all those objects makes it harder to climb. So the traveler wants to travel as light as possible, while having all necessary items with him. Suppose

$$z_l = \begin{cases} 1 & \text{if item } l \text{ is } \textit{not} \text{ put into the knapsack} \\ 0 & \text{otherwise} \end{cases}$$

It will be shown that for the Bounded Multidimensional Knapsack Problem maximizing $\sum_{l \in L} p_l z_l$ is equivalent to minimizing $\sum_{l \in L} p_l y_l$. This makes sense, since every item that can be left behind, is an item that does not need to be put in the knapsack. The proof given below is that the Bounded Multidimensional Knapsack Problem can be rewritten to the minimization problem. The reverse is trivial: the steps in the rewriting process can be reversed. It is easy to see that for all $l \in L$

$$z_l = b_l - y_l \ .$$

Then the problem

$$\begin{aligned} \max & \sum_{l \in L} p_l z_l \\ \text{s.t.} & \sum_{l \in L} w_{kl} z_l \leq c_k^{knap} \qquad \text{for all } k \in K \\ & 0 \leq z_l \leq b_l \text{ and integer for all } l \in L \end{aligned}$$

becomes

$$\begin{aligned} \max & \sum_{l \in L} p_l (b_l - y_l) \\ \text{s.t.} & \sum_{l \in L} w_{kl} (b_l - y_l) \leq c_k^{knap} \qquad \text{for all } k \in K \\ & 0 \leq b_l - y_l \leq b_l \text{ and integer for all } l \in L \end{aligned}$$

or

$$\begin{aligned} \max & \sum_{l \in L} p_l b_l - \sum_{l \in L} p_l y_l \\ \text{s.t.} & \sum_{l \in L} w_{kl} b_l - \sum_{l \in L} w_{kl} y_l \leq c_k^{knap} \text{ for all } k \in K \\ & 0 \leq b_l - y_l \leq b_l \text{ and integer} \quad \text{for all } l \in L \end{aligned}$$

The next step is to rewrite each of these equations. When this is done for the objective function the result is

$$\max \sum_{l \in L} p_l b_l - \sum_{l \in L} p_l y_l \ .$$

Since it is of no use to optimize a constant value, this equals to

$$\max - \sum_{l \in L} p_l y_l = \min \sum_{l \in L} p_l y_l \ . \tag{5.16}$$

The constraints

$$\sum_{l \in L} w_{kl} b_l - \sum_{l \in L} w_{kl} y_l \leq c_k^{knap}$$

can be rewritten to

$$\sum_{l \in L} w_{kl} y_l \geq \sum_{l \in L} w_{kl} b_l - c_k^{knap} \ . \tag{5.17}$$

Also,

$$0 \leq b_l - y_l \leq b_l$$

equals

$$0 \leq y_l \leq b_l \ . \tag{5.18}$$

Summarizing equations 5.16, 5.17 and 5.18, the model is now formulated as

$$\min \sum_{l \in L} p_l y_l \tag{5.19}$$

$$\text{s.t.} \sum_{l \in L} w_{kl} y_l \geq \sum_{l \in L} w_{kl} b_l - c_k^{knap} \text{ for all } k \in K \tag{5.20}$$

$$0 \leq y_l \leq b_l \text{ and integer} \qquad \text{for all } l \in L \tag{5.21}$$

which is indeed a model where $\sum_{l \in L} p_l y_l$ is minimized.

For the remainder of the thesis, Multidimensional Knapsack Problem refers to the unbounded variant of this minimization problem.

5.3 Comparing SCP and MKP

A closer look on both the Generalised Set Covering Problem (equations 5.4-5.6) and the Multidimensional Knapsack Problem (equations 5.19-5.21) reveals some interesting points. In fact, both models are almost the same:

$$\begin{aligned} x_j &= y_l \\ c_j &= p_l \\ r_i &= \sum_{l \in L} w_{kl} b_l - c_k^{knap} \ . \end{aligned}$$

The only difference between the two models is the value of the weights. For the Generalised Set Covering Problem, $a_{ij} \in \{0, 1\}$, while for the Multidimensional Knapsack Problem $0 \leq w_{kl} \leq 1$. So the Multidimensional Knapsack Problem is a even more general problem than

the Generalised Set Covering Problem.

So why going to all the trouble of explaining the principles of knapsack problems, if it appears that both problems are almost the same? Is there a reason that one view on the problem might be better than the other? Yes, there is. For each view point, different methods have been developed. Each of these methods try to exploit the specific characteristics of the model. The Generalised Set Covering Problem has the extra property that the value of the weights are more restricted. Apparently, to make optimal use of this property, entire different models are necessary.

The quality of the results depends on the model and the solution method used. It seems natural that the Generalised Set Covering Problem is easier to solve, and may have results of higher quality (if the model is not solved with an exact method). But it is difficult to tell beforehand which one is better. The set covering approach is mostly used to solve these kind of problems, due to the fact that for duties, the ”weights” are either 0 or 1. But as has been explained in this chapter, most of the methods are developed to solve the standard problem, instead of the more generalised one. Knapsack problems are hardly used for these kind of problems, which sounds logical since knapsack problems actually miss some of the information available. But since a knapsack problem is only slightly more general, it might be useful use methods for solving knapsack problems as well. So again it is almost impossible to tell how the performance of an algorithm based on the knapsack view will be. Therefore, in the following chapters, a method that can be used to solve both model will be explained. It will be shown that for each view, this method has a different approach towards solving the model.

Chapter 6

Relaxations

As describer earlier in section 3.3.1, difficult problems are often solved by (iteratively) solving a relaxation of the problem. What this means is explained in section 6.1 in more detail. That is why the (Generalized) Set Covering Problem will also be solved by means of relaxations. In this case two means: the Lagrangian relaxation method (explained in section 6.2) and the surrogate relaxation (explained in section 6.3). Section 6.4 lies both relaxation methods side by side and shows a comparison between the two.

6.1 Relaxing a problem

The purpose of finding a mathematical model for a certain problem is to search for an optimal solution for that model. If the model was correct for the described real-life problem, then an optimal solution reflects the actions that one should take in order to achieve the most profit or reduce the costs as much as possible. So the most ideal situation is that the solution method used to solve the model actually finds an optimal solution.

Exact methods (such as enumerating all the possible solutions and calculate which ones has the optimal value, or a more advanced method) guarantee that an optimal solution will be found. However, this has its price. For many problems, such methods need a lot of time and system resources. So, other methods as heuristics are used (see also chapter 2.2).

Sometimes, optimal solutions are found and their optimality proven, but in most cases one can only try to find solutions as good as possible. For finding good solutions, one has to find primal bounds and dual bounds. These terms will be explained below for a minimization problem, since the main problem treated in this thesis is a minimization problem. The explanation for a maximization problem is analogue to that of the minimization problem.

Solution values of feasible solutions are called primal bounds. In the case of a minimization problem, primal bounds are upper bounds. Note that the optimal solution value is defined as the lowest possible solution value associated with a feasible solution. So every feasible solution has a solution value which is equal or larger than the optimal solution value, and therefore is an upper bound on the optimal solution value. Suppose that there is solution 1 and solution 2, with solution values z_1 and z_2 respectively. Both solutions are feasible, and solution value z_1 is the optimal solution value. Now suppose that $z_2 < z_1$. Because solution z_2 is feasible, solution value z_2 is better than solution value z_1. The consequence is that solution value z_1 cannot be the optimal solution value.

The opposites of primal bounds are dual bounds. In this case dual bounds are lower bounds, since those are opposites of upper bounds. Lower bounds (values lower than the optimal solution value) can never belong to a feasible solution for the same reason stated above. Suppose that solution value z_1 is indeed optimal, but still $z_2 < z_1$. Then solution 2 can not be feasible. If it was, then solution value z_2 would be the optimal solution value according to the proof above. But it was just made clear that z_1 is the optimal value, so solution b must be infeasible. A warning must be made on reversing the statement ("Solutions with a value equal or larger than the optimal solution value is feasible"). This is not true. It is easy to create an infeasible solution, while the solution value rises above the optimal value (one way is to take a random infeasible solution, and to keep increasing the value of a decision variable that does not make the solution feasible).

The trick in finding the best solution possible is to find lower upper bounds, and higher lower bounds. An illustration is given in figure 6.1.

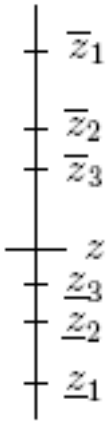

Figure 6.1: Primal and dual bounds

In this figure, z is the optimal value, $\overline{z}_s$ are upper bounds and $\underline{z}_t$ are lower bounds. As can be seen, the lower $\overline{z}_s$ becomes, the more it closes in on z. The same goes for increasing values of $\underline{z}_t$. Because $\underline{z}_t \leq z \leq \overline{z}_s$, the optimal solution value has been found when $\underline{z}_t = \overline{z}_s = z$. Unfortunately, some algorithms are not able to find an upper and a lower bound which are equal, or simply take too much time to do this. So often the algorithm is designed in such a way that it stops when $\overline{z}_s - \underline{z}_t \leq \epsilon$, where ϵ is a suitable chosen small nonnegative value. The solution is not guaranteed to be optimal, but since the solution value obtained is at most $z + \epsilon$, one is satisfied with the current solution found.

So how can these bounds be determined? Finding any primal bound should not be a problem most of the times. As has been shown before, every feasible solution is a primal bound. The real challenge here is to find good primal bounds. The traveling salesman problem is a good example of this. A feasible solution can be found quickly by selecting (random) cities one after another to find a tour. However, enumerating all feasible tours to find the best one isn't a smart thing to do, because the number of tours grows exponentially with the number of cities. Even a small number of cities may result in an enormous amount of tours.

A different challenge is presented by finding good dual bounds. One of the most important methods to find them is by means of ”relaxations”. A relaxation of a problem is a more simplified version of this problem, with an optimal solution value at least as small as z. Since the relaxation is easier to solve, this optimal solution value should be found and hopefully within a reasonable time.

Definition 1 *A problem* $(RP)z^R = \min\{f(x) : x \in T \subseteq R^n\}$ *is a relaxation of* $(IP)z = \min\{c(x) : x \in X \subseteq R^n\}$ *if:*

1. $X \subseteq T$, *and*
2. $f(x) \leq c(x)$ *for all* $x \in X$.

Proposition 1 *If* RP *is a relaxation of* IP, $z^R \leq z$.

To prove proposition 1 suppose that x^* is an optimal solution of IP. Then $x^* \in X \subseteq T$ and $z = c(x^*) \geq f(x^*$. Because $x^* \in T$, $f(x^*)$ must be a upper bound on z^R, according to definition 1. The result is that $z^R \leq f(x^*) \leq z$.

There are several methods to construct a relaxation. A simple method is to make a linear programming model out of an integer programming model. In other words, let go of the constraints that the decision variables must be integers. The solution of the relaxation is then acquired by solving this linear programming model. To get a feasible solution from the solution of the relaxation, the simplest method is to to round the values of the decision variables to the nearest integer. This approach must be used with care, especially with the rounding part. This will be explained with figure 6.2.

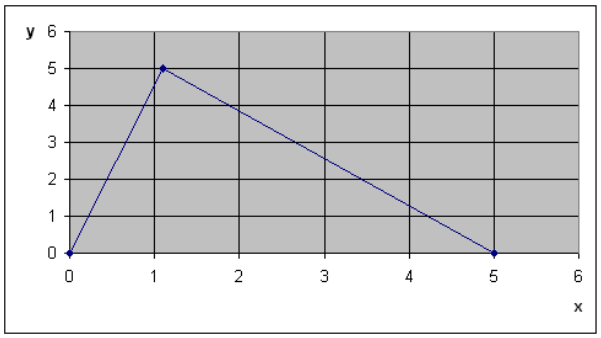

Figure 6.2: Example of a solution space

In this figure, an example of a solution space is given. All solutions (x, y) lying within the area (indicated by the triangle with corner points $(0, 0)$, $(1.1, 5)$ and (5.0)) are feasible solutions for a certain problem. Suppose that $(x, y) = (1.1, 5)$ is the solution with the optimal solution value. If an integer model is solved, the solution must be rounded. Looking purely at the values of x and y, this means it must be rounded towards $(x, y) = (1, 5)$, which is an infeasible solution. The nearest feasible integer solution is $(x, y) = (1, 4)$. However, it is still possible that other integer solutions have a better solution values than this one. In the worst case, this solution might have the worst solution value of them all.

This and more (detailed) information can be found in Wolsey (1998).

6.2 Lagrangian relaxation

One of the possible methods that are mentioned in Wolsey (1998) is Lagrangian relaxation. In short: with Lagrangian relaxation, some or all the constraints are put in the objective function with some weight values.

Consider the following integer programming model:

$$z = \min cx$$

$$Dx \leq d$$

$$x \in X$$

where $Dx \leq d$ are m complicated constraints, which make the problem hard to solve. Without these constraints, the problem would be easy to solve, so they must be "removed". To still take them into account, they are put in the objective function with the vector $u = (u_1, \ldots, u_m)$ as a vector of weights. For any value of $u \geq 0$ the Lagrangian relaxation (subproblem) $IP(u)$ is defined as

$$z(u) = \min cx - u(d - Dx)$$

$$x \in X$$

Proposition 2 *Problem $IP(u)$ is a relaxation of problem IP for all $u \geq 0$.*

According to definition 1 two properties need to be proven.

1. The feasible region of the Lagrangian relaxation $IP(u)$ is at least as large as the feasible region of the original problem IP. This is true because $\{x : Dx \leq d, x \in X\} \subseteq X$. This is a logical conclusion. All solutions that are feasible with constraints, remain feasible when these constraints are lifted. Next to these, there are solutions which violate one or more constraints. These solutions become feasible when the constraints are lifted.

2. The objective value of $IP(u)$ is at least as small as the objective value of IP for all solutions $x \in X$. Because $u \geq 0$ and $Dx \leq d$ for all $x \in X$, then it is also true that $cx - u(d - Dx) \leq cx$ for all $x \in X$.

It can be seen that the constraints $Dx \leq d$ are handled by adding them to the objective function at the cost (or profit) of $u(d - Dx)$. Therefore, u is sometimes called the *price* or *dual variable* associated with constraints $Dx \leq d$. But generally, it is known as the *Lagrange multiplier*.

The Lagrangian relaxation method is a popular method to use for relaxing the (standard) Set Covering Problem. Usually, all the constraints have been relaxed when a Lagrangian relaxation is derived from the Set Covering Problem. When almost all constraints are put in the objective function (leaving a linear program with only some boundary constraints on the decision variables), then solving the Lagrangian relaxation becomes trivial. The Lagrangian relaxation of the standard Set Covering Problem is

$$\min \sum_{j \in N} c_j x_j - \sum_{i \in M} u_i (\sum_{j \in N} a_{ij} x_j - 1)$$

$$\text{s.t.} x_j \in \{0, 1\} \text{ for all } j \in N$$

It is not difficult to rewrite the objective function to make it easier to read:

$$\begin{aligned}
\min \sum_{j \in N} c_j x_j - \sum_{i \in M} u_i (\sum_{j \in N} a_{ij} x_j - 1) &= \min \sum_{j \in N} c_j x_j - \sum_{i \in M} u_i \sum_{j \in N} a_{ij} x_j + \sum_{i \in M} u_i \\
&= \min \sum_{j \in N} c_j x_j - \sum_{i \in I_j} u_i \sum_{j \in N} x_j + \sum_{i \in M} u_i \\
&= \min \sum_{j \in N} (c_j x_j - x_j \sum_{i \in I_j} u_i) + \sum_{i \in M} u_i \\
&= \min \sum_{j \in N} (c_j - \sum_{i \in I_j} u_i) x_j + \sum_{i \in M} u_i \\
&= \min \sum_{j \in N} c_j(u) x_j + \sum_{i \in M} u_i
\end{aligned}$$

where

$$c_j(u) = c_j - \sum_{i \in I_j} u_i$$

is defined as the *Lagrangian cost*.

Solving the problem now becomes very easy. If the Lagrangian cost $c_j(u) < 0$, then $x_j = 1$, and when $c_j(u) > 0$, $x_j = 0$. For the case that $c_j(u) = 0$, x_j can be either 0 or 1, one must be careful when there are many values for j where $c_j(u) = 0$. Suppose that for a given u, $n = 1$ with n the number of decision variables where there is only one value of j where $c_j(u) = 0$. For the Lagrangian relaxation, the optimal solution value will remain the same if x_j is chosen differently. So either solution can be taken as the base solution in the method for finding a feasible solution of the main problem. This is where the problem lies. Both solutions of the relaxation will result in a different feasible solution, with probably different solution values. Most methods do not look ahead or look back, so the best option at the current iteration is to take the solution with the lowest solution value. The number of optimal solutions is exponential with n: there are 2^n optimal solutions for the Lagrangian relaxation. So this problem can be dealt with easily when n is small, but it becomes harder when n takes higher values.

Usually, Lagrangian relaxation is used within an iterative algorithm. In each iteration, the multipliers u are updated, hoping that in the next iteration, the solution of the relaxation is a more useful one for finding better feasible solutions. Often, the concept of *subgradients* is used for updating the multipliers.

Definition 2 *A* subgradient *at u of a convex function $f : R^m \rightarrow R^1$ is a vector $\gamma(u) \in R^m$ such that $f(v) \geq f(u) + \gamma(u)^T(v - u)$ for all $v \in R^m$. For a continuously differentiable convex function f, $\gamma(u) = \nabla f(u) = \left(\frac{\partial f}{\partial u_1}, \ldots, \frac{\partial f}{\partial u_m}\right)$ is the gradient of f at u.*

A basic subgradient algorithm is given in algorithm 1.

The algorithm is designed to solve problem $IP(u)$. In every iteration, a solution and its value is acquired. Then, new values for u as computed. With the new u, the relaxation is solved again. The new solution value should be better than the solution value from the previous iteration, or at least not too much worse. Eventually, the solution values will converge. When that happens, the solution value cannot be improved significantly anymore. The algorithm will stop and the best solution found so far will be returned. Details about this algorithm will not be

Algorithm 1 Basic subgradient algorithm

1: Initialization: $u = u^0$
2: **repeat**
3: Solve the Lagrangian problem $IP(u^k)$ with optimal solution $x(u^k)$.
4: $u^{k+1} = \max\{u^k - \mu_k(d - Dx(u^k)), 0\}$
5: $k = k + 1$
6: **until** Convergence has been achieved

explained here; most of them will also be discussed with the algorithm described in chapter 7. More information on algorithm 1 is given by Wolsey (1998).

For the Generalized Set Covering Problem, all the constraints (except the boundary constraints for the decision variables x) are relaxed also. This results in the following formulation for the Lagrangian relaxation:

$$\min L(u) = \sum_{j \in N} c_j(u) x_j + \sum_{i \in M} u_i r_i \tag{6.1}$$

$$\text{s.t.} x_j \geq 0 \text{ and integer for all } j \in N \tag{6.2}$$

with Lagrangian cost

$$c_j(u) = c_j - \sum_{i \in I_j} u_i \ .$$

There are two differences compared to the Lagrangian relaxation of the standard Set Covering Problem. Negligible is the difference in the objective function. Within the sum of multipliers, those multipliers now need to be multiplied with the demanded workload. For a given vector u, this remains a constant. Therefore, this change does not have any consequences. More important is the difference in the boundaries of the decision variables x_j. For the standard Set Covering Problem, x_j is limited to the values 0 and 1. For the Generalized Set Covering Problem, x_j may take any positive integer value (including the value 0).

The consequence is that the Lagrangian relaxation cannot be solved trivial anymore. If the procedure described earlier would be used, then in the strict sense for all j where $c_j(u) < 0$, $x_j \rightarrow \infty$. After all, the problem is to minimize $L(u)$, and with these decision variables approximating infinity, $L(u) \rightarrow -\infty$. This is not a practical situation. For example for the Set Covering Problem, it makes no sense to choose a duty an infinite number of times.

This will be illustrated with a simple numerical example. Suppose that $r_i = \{3, 2, 5\}$. A duty has been chosen that covers all three planning periods, and that has negative Lagrangian cost. For the Lagrangian relaxation, the best thing to do is to select the duty an infinite number of times. But it is only necessary a maximum of five times. This way, the demand for all three planning periods are fulfilled. Selecting the duty more than 5 times makes it redundant. It would be good for the solution value of the Lagrangian relaxation, but for the original problem, the solution value will become infinity. This is a contradiction with the objective, which is minimizing the costs.

There are several methods to deal with this problem. One of them is to really select the duties an infinite number of times for the solution of the relaxation. This will be corrected

when the feasible solution for the original problem is formed. A method that is used in the algorithm described in chapter 7 takes this problem into account when selecting the duties in the Lagrangian relaxation. Upper bounds on the decision variables are derived, so the relaxation can be solved trivial again. Note that the standard Set Covering Problem is also solved by choosing the duties the number of times that is equal to the upper bound (the upper bound on x_j equals 1 for the standard Set Covering Problem). At first, the upper bound on x_j is defined as $\max_{i \in I_j} r_i$. During the computation of the solution for the relaxation, the upper bounds may be adjusted, in order to have tighter bounds on the decision variables.

6.3 Surrogate relaxation

Although Lagrangian relaxation is by far the most popular method when a relaxation of a problem is derived, there are other methods that can be used. One of these methods is surrogate relaxation. Since surrogate relaxation is relatively unknown, not much literature can be found on this subject. Two of the most fundamental articles about this subject are written by Greenberg and Pierskalla (1970) and Glover (1975). Greenberg and Pierskalla (1970) describe the (highly detailed) basic theory about the surrogate approach. Glover (1975) introduces a surrogate duality theory. The method has been applied for the standard Set Covering Problem by Lorena and Lopes (1994). They developed an algorithm somewhat similar to the algorithm of Caprara et al. (1999). In each iteration, after solving the surrogate relaxation problem, the new set of multipliers is computed with the use of subgradients.

While Lagrangian relaxation "absorbs" the constraints in the objective function, surrogate relaxation rather "compresses" some or all the constraints into a single one which is called the *surrogate constraint*. So for the standard Set Covering Problem the surrogate relaxation is defined as

$$\begin{aligned} \min & \sum_{j \in N} c_j x_j \\ \text{s.t.} & \sum_{i \in M} u_i \sum_{j \in N} a_{ij} x_j \geq \sum_{i \in M} u_i \\ & x_j \in \{0,1\} \qquad\qquad \text{for all } j \in N\ . \end{aligned}$$

The surrogate relaxation problem of the Generalized Set Covering Problem does not differ much:

$$\begin{aligned} \min & \sum_{j \in N} c_j x_j \\ \text{s.t.} & \sum_{i \in M} u_i \sum_{j \in N} a_{ij} x_j \geq \sum_{i \in M} u_i r_i \\ & x_j \geq 0 \text{ and integer} \qquad \text{for all } j \in N \end{aligned}$$

Contrary to the Lagrangian relaxation problem, it is not possible anymore to solve the problem trivially. As can be seen in the examples of the Set Covering Problem above, a small linear

programming model remains, which needs to be solved. However, since the number of constraints has been reduced, one should be able to get a good solution with less effort. Also, this is where it is useful to compare the Set Covering Problem with the Multidimensional Knapsack Problem. The surrogate relaxation of a Multidimensional Knapsack Problem is still a knapsack problem. What it does is actually reduce the number of dimensions on the knapsack. When all constraints are relaxed by means of surrogate relaxation, the Multidimensional Knapsack Problem is actually reduced to a Single dimensional Knapsack Problem. For this type of problems, there are already a number of good methods. Some of them are discussed by Kellerer et al. (2004).

6.4 Comparing both relaxations

With these two different solution methods, it now becomes clear why different views on the problem are discussed in chapter 5.

The standard Set Covering Problem has been solved successfully quite a few times using Lagrangian relaxation. For the Generalized Set Covering Problem, it is not clear immediately if Lagrangian relaxation will be succesfull as well. There is an extra problem that needs to be dealt with. Remember that for the Generalized Set Covering Problem, the values of the decision variables do not have upper bounds in the mathematical model. As shown before, they are fortunately bounded in practice. To successfully use Lagrangian relaxation, it needs to be determined what those bounds will be. Lagrangian relaxation is expected to give better results if the difference between the lower bounds and the upper bounds becomes smaller (with a difference equal to 1 being the most ideal situation, since solving the problem then becomes similar to solving the standard Set Covering Problem).

When viewing the problem as a Multidimensional Knapsack Problem, surrogate relaxation promises to be a good method for making the problem less difficult to solve. The reduction of dimensions to a single one makes it possible to use an efficient method for solving the resulting problem. A downside is that some information will be lost. In general, the values of a_{ij} in a Set Covering Problem are binary, while these values could be any positive value in a Multidimensional Knapsack Problem.

Beforehand, it is difficult to say which method of relaxing the problem will give better results. Solving a Lagrangian relaxation won't take much time in most of the situations. The relaxation can be solved in a trivial way, even with the changes needed for making the standard Set Covering Problem a general one. However, when used within a subgradient algorithm, the solution of this algorithm will converge less faster towards a stable solution. In other words, the subgradient algorithm will take a large number of iterations before finding a solution. Singledimensional Knapsack Problems are not as easy to solve as the Lagrangian relaxation of a Set Covering Problem. Therefore, a single iteration of the subgradient algorithm will take much more time when using surrogate relaxation. The main advantage of using surrogate relaxation however is that the solution of the subgradient algorithm might converge faster. An example of such an algorithm is shown by Lorena and Lopes (1994). So surrogate relaxation will be the better choice if the number of iterations of the subgradient algorithm is reduced by a larger factor than the factor by which the computation time per iteration has increased.

This research determines empirically which one of the relaxation methods is better. In the

following chapter, a subgradient algorithm will be explained, where both relaxation methods are implemented.

Chapter 7

An algorithm using different relaxations and subgradients

In this chapter, the algorithm that has been implemented to solve the Set Covering Problem will be described extensively. In section 7.1, a short introduction will be given about the choice for this algorithm and how it has been developed. Section 7.2 explains what the main algorithm, used by both methods, looks like. Every step of this algorithm will be described in detail in the several subsections. Finally, sections 7.3 and 7.4 explain how the Lagrangian relaxation and the surrogate relaxation of the problem is solved within the main algorithm.

7.1 Introduction

The algorithm presented here is based upon the algorithm described in Caprara et al. (1999), including some modifications recommended by Freling (1997). The choice made for this algorithm is primarily based on the promising results from earlier research. One of those researches was done by Teeuw and Woutersen (2003). There it was used to solve the problem of scheduling the tasks done at a Dutch mail delivery company (see also section 2.3).

Next to this algorithm, there were some other approaches that could be used. The methods mentioned in section 3.3 can also be applied on problems concerning shift based demand instead of task based demand. In both cases, a Set Covering Problem must be solved, so the only difference between the two types of demand lies in the procedure used to generate the demands.

The final choice for this algorithm was made after some conversation about which algorithm to use (Wagelmans and Gootjes (2005)). The conclusion was to use this algorithm because of the good results mentioned before, and the addition of investigating the replacement of Lagrangian relaxation with surrogate relaxation. However, this algorithm was designed and tested for the standard Set Covering Problem. It is not difficult to adapt the procedures for the generalised Set Covering Problem, but it is unknown if the algorithm will still give satisfying results. This must be concluded from the results of the computational experiments. Since the number of modifications becomes quite significant, the entire algorithm will be described in the sections below.

7.2 The main algorithm

In this section, the main framework of the algorithm will be described. This framework will be used by both relaxation methods. The algorithm belongs to the class of subgradient algorithms, which means that it uses subgradients where the values of the multipliers need to be updated. The problem that this specific implementation will solve has been generated according to the method described and implemented by Woutersen (2005). There, the Crew Scheduling Problem was solved using either one of two formulations: the set covering formulation (traditional formulation), and the implicit formulation. Trivially, the problems must be formulated using the set covering formulation (this algorithm is not able to deal with the implicit formulation). A single problem might be divided into several subproblems: this happens when during the selected period there are regular moments where the workload is equal to 0 (for example for companies who do not need employees during the night). At these moments, the problem can be split, with subproblems as the result. In this case, the algorithm will be applied on each subproblem, since each subproblem is a Set Covering Problem by itself.

The main framework has the following simple schedule:

Algorithm 2 Relaxation solver

1: Set up the algorithm
2: **repeat**
3: Execute a single iteration
4: **until** (Some) stop criteria are met

7.2.1 The set-up procedure

First of all, the problem needs to be reformulated. The most important reason was to make the representation more suitable for the algorithm to deal with. Instead of using an object oriented programming code, the problem was reformulated towards a matrix representation. This way, calculations were more easy to perform. Second, during reformulating the problem, a specific problem could be detected. In Woutersen (2005), one constraint is used that actually does not belong to a Set Covering Problem. In that representation, it was possible to set a minimum and maximum number of times a duty with certain parameters can be chosen. A pure Set Covering Problem representation, as it is used here, is able to deal with the constraint stating that some duties must be chosen a minimum number of times. These constraints look exactly the same as the other constraint, stating that the total number of duties covering a row must be equal or greater than the number of times the row must be covered. However, constraints about the maximum number of times duties can be chosen look as follows: $\sum_{i \in \tilde{I}} x_i \leq n$ (with $\tilde{I}$ a certain set of duties and n the maximum total number of selected duties from this set). As one can see, these constraints have a $\leq$ sign, while a pure Set Covering Problem representation may only contain constraints with a $\geq$ sign (see also section 5.1).

The next step is to set the starting values of some variables. The starting values used here are the same as used in Caprara et al. (1999). The starting vector of multipliers u_i^0 is computed in a similar greedy way:

$$u_i^0 = \min_{j \in J_i} \frac{c_j}{|I_j|}, i \in M \ .$$

For updating these multipliers, the step size λ will be defined, and will be given the starting value of 0.1. Finally, the initial value of the upper bound will be computed. For this, algorithm 3 is used (see section 7.2.2), with $u = 0$ so the original costs are used instead of the costs of the relaxation.

7.2.2 A single iteration

The course of a single iteration follows these steps:

1. Solve the Lagrangian relaxation of the problem.
2. Solve the main problem by using the GREEDY algorithm and the results of solving the Lagrangian relaxation.
3. Solve the surrogate relaxation of the problem.
4. Solve the main problem by using the GREEDY algorithm and the results of solving the surrogate relaxation.
5. Update the multipliers.
6. Update the step size.

Solving the relaxation of the problem For the main problem, both relaxations (Lagrangian and surrogate) are derived and solved. Also the main problem is solved with the results of solving both relaxations. The better of the two solutions will be stored. The method used for solving the relaxed problem is specific for the relaxation used. The method for solving the Lagrangian relaxation is described in section 7.3. Details about the method for solving the surrogate relaxation can be found in section 7.4.

Solving main problem with GREEDY heuristic To obtain a "good" solution of the main problem, and so hopefully a better upper bound, a greedy heuristic has been applied:

Algorithm 3 GREEDY (u^k, x)

1: Initialize x and m
2: **repeat**
3: Compute the score σ_j for every $(j \in N | x_j > 0)$, and let $(j^* \in N | x_j > 0)$ be the column where σ_j is minimal
4: $\Delta m = \max_{i \in I_{j^*}} m_i$
5: $x_{j^*} = x_{j^*} + \Delta m$
6: For every $i \in I_{j^*}$ let $m_i = \max(0, m_i - \Delta m)$
7: **until** $m = 0$
8: Remove redundant columns in x
9: **return** x

For some statements in the algorithm more details will be described below.

Line 1 The vectors x (solution vector with the number of times the columns have been selected) and m (vector with for each row the number of times it has not been covered yet by x) need to be initialized in either one of two ways. When GREEDY has been called to solve the problem in the set-up procedure, x^0 will be set to 0, and m will be equal to r. Since multipliers are not determined yet, these will also be set to 0. In iteration k, $x^k = x(u^k)$. Then for all $i \in M$, $m_i = \max(0, r_i - \sum_{j \in J_i} x_j)$. Since $x(u^k)$ is the solution of the relaxed problem (which is not feasible for the main problem), there are still some rows not covered (completely). So, there is at least one $m_i > 0$.

Line 3 In Caprara et al. (1999), several rules have been tried to define σ_j. The rule which obtained the best results is also used here, which is

$$\begin{aligned}\sigma_j &= \gamma_j/\mu_j \text{ if } \gamma_j > 0\\ &= \gamma_j\mu_j \text{ if } \gamma_j < 0\\ &= \infty \text{ if } \mu_j = 0\end{aligned}$$

where

$$\mu_j = \sum_{j \in J_i} m_i$$

and

$$\gamma_j = c_j - \sum_{i \in (I_j | x_j > 0)} u_i^k \ .$$

Line 8 After all rows have been covered, it is possible that columns have become redundant a number of times. This happens when in a later iteration, columns selected will possibly also cover some rows that are covered by already selected rows. If several of these selections cover the rows that are covered by a column that has been selected earlier, then the latter becomes redundant. For example, consider the following matrix A, where every row must be covered at least once:

$$A := \left\{ \begin{array}{ccc} 1 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{array} \right\} \ .$$

Suppose that the values of σ_j are computed in such a way that in the first iteration, column 1 will be selected once, and the same goes for column 2 and 3 in the second and third iteration. Then $x = (1, 1, 1)$. But clearly the optimal solution is $x = (0, 1, 1)$. So column 1 has become redundant and must be removed from the solution.

Caprara et al. (1999) have used a mixed approach to remove these redundant columns. They removed the redundant columns partially by a heuristic and partially with a complete enumeration method. Preferably, only the enumeration method would have been used. This method looks at every possible solution of the problem. Therefore, it guarantees that the optimal solution will be found. However, it is also a costly method in terms of resources mainly time. To keep the loss in time at a minimum, it is mixed with a heuristic approach.

Almost the same approach has been used here. Define x_j^R as the number of times column j has been selected redundantly. Or, when column j is selected $x_j - x_j^R - 1$ times, then $m > 0$. While $\sum_{i \in M} > 10$, then for column $j^* \in M$ with $x_{j^*}^R > 0$ and maximum cost c_j, x_{j^*} will be lowered by $x_{j^*}^R$. This implies that x_{j^*} will become 0. When $\sum_{i \in M} \leq 10$, an enumeration tree

will be used to find the best solution. The building of the tree starts with the solution known at that time. Then, nodes will be made for each column j where $x_j^R > 0$. The solution passed over to the node equals the starting solution, where x_j and x_j^R are lowered by 1. This procedure will be repeated for each node, until a node contains a solution with $\sum_{i \in M} = 0$. This will be the final solution of a branch, since there are no more redundant columns to remove. The best solution from the tree will be the final branch solution with the best (or lowest) objective function value.

Update the multipliers The multiplier vector u^k is updated by using the following formula (for every $i \in M$):

$$u_i^{k+1} = \max \left\{ u_i^k + \lambda \frac{UB - L(u^k)}{\|s(u^k)\|^2} s_i(u^k), 0 \right\} \tag{7.1}$$

As one can see, this is where the step size λ is used. UB is an upper bound of the problem, which here is the best upper bound found up to iteration k. It must be stated that only the solutions of the main problem derived by Lagrangian relaxation are used. The vector $s(u^k)$ is the subgradient vector, defined by:

$$s_i = r_i - \sum_{j \in J_i} x_j(u)$$

for all $i \in M$.

Update the step size The step size λ will be updated every $p(= 20)$ iterations. When updating, the best and worst lower bounds computed in the last p iterations are compared. Again, only the lower bounds resulting from the Lagrangian relaxation are used. If the difference between these two values is larger than 1% of the best lower bound, then the current value of λ will be multiplied by 0.5. When the difference is smaller than 0.1% compared to the best lower bound, then λ is multiplied by 1.5. All these values have been taken directly from Caprara et al. (1999), but it is possible that with other values better results are obtained. Experimenting with other values has been left for further research.

7.2.3 Stopping criteria

There are several criteria where the algorithm stops if one of them is met.

The most important one is the maximum computation time. All other criteria can be ignored if one wants to, but the maximum computation time ensures that the algorithm stops after a certain amount of time. By ignoring the other criteria, and setting the maximum computation time at a large amount (for example several hours), the algorithm tries to find the best possible solution within the time allowed, and is not hindered by the other criteria.

If one wants to know a solution in a short amount of time, it is useful to use some other criteria as well. These criteria then stop the algorithm prematurely, to save computation time in which probably no better solutions are going to be found. One of them is closely related to the maximum computation time. Instead, there is a maximum number of iterations. This criterion is useful in cases where the computation time for each iteration is very small. In Caprara et al. (1999), the maximum number of iterations is set at 10 times the number of rows. However, during the implementation phase it was discovered that this number would be too small for some

cases. Therefore, this value is set at 20 times the number of rows.

The third criterion that is used is based on the convergence of the algorithm. If within the last 300 iterations, the lower bound has not improved with an absolute value greater or equal than 1.0 and a relative value greater or equal to 0.1%, then the algorithm will be stopped. In other words: the algorithm will stop if the lower bound has not been improved significantly for 300 iterations, since it is expected that not much improvement can be obtained in further iterations. Just as with the values for updating the multiplier vector, these values have been directly taken from Caprara et al. (1999).

7.3 Solving the Lagrangian relaxation

Usually, a Lagrangian relaxation of a problem can be solved in a trivial way (see also section 6.2). Here, it is done with a somewhat different decision rule for different reasons.

If a Lagrangian relaxation is solved in the usual way, all columns j with $c_j(u) \leq 0$ are chosen. This is also the case in this research. Caprara et al. (1999) used a somewhat different rule. They chose all columns j with $c_j(u) \leq 0.001$ instead of 0. The reason for not taking the regular value 0 was a large number of columns with Lagragian cost close to 0. In their computational experience, when solving large scale data sets one may have to deal with over 1000 Lagragian costs with $|c_j(u)| < 0.001$. The consequence will be that there are a lot of near-optimal solutions possible when choosing a subset of columns with these costs. Because of this, there will also be a lot of different values for the subgradient $s(u^k)$ which can be used in eqn. 7.1. The datasets used for this research were not as large as the ones Caprara et al. (1999) used. Therefore, the same problem was not expected here, and the regular value of 0 was used.

As mentioned before in section 6.2, bounds need to be determined for the decision variables. A straightforward method is to take for each $j \in N$, $\max(r_i | i \in I_j)$ where $c_j(u) \leq 0$. This way, the Lagrangian cost function will be as low as possible, while it remains practical. Of course, selecting the columns more times than these bounds will decrease the Lagrangian cost function, but since then the columns will become redundant, it is not reasonable to do so.

Tests during the implementation of the algorithm showed that these bounds would not give good results. That is why the following approach has been taken. From the set of columns where $c_j(u) \leq 0$, the column j^* with minimum Lagrangian cost $c_j(u)$ will be selected. The number of times column j^* will be selected is equal to the number of times that the rows, covered by this row, still needs to be covered, considering the selection that has already been made. After column j^* has been removed from the set, the process will repeat until this set is empty.

The following example illustrates this procedure. Consider matrix A:

$$A := \begin{Bmatrix} 1 & 1 & 0 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{Bmatrix}$$

and the array with workloads r:

$$r := \begin{Bmatrix} 1 \\ 2 \\ 1 \end{Bmatrix} .$$

Suppose column 2 has the lowest Lagrangian costs. Column 2 covers rows 1 and 2, and the maximum value of their workloads is 2. Hence, column 2 will be selected 2 times. If column 3 is selected next, it will be selected only once instead of twice. Although it covers both rows 2 and 3, row 2 is already fully covered by column 2. So, it only needs to cover row 3, which has a workload of 1. Column 1 will not be selected at all. The rows it covers are now already fully covered by the other two columns, so there is no need to select column 1 anymore.

Even with the last method, there is a probability that redundant columns may occur, so these have to be removed. This is done a little bit different from what happens at the end of algorithm 3, and follows the method from Caprara et al. (1999). After determining which columns have (partially) been selected redundantly, the selection of the column with the highest Lagrangian cost will be reduced to the number where it will not be redundant. This process repeats until no more columns are redundant.

7.4 Solving the surrogate relaxation

For solving the surrogate relaxation problem, a single-dimensional knapsack problem must be solved (see also section 6.3). An algorithm called Bouknap has been used to do this. The Bouknap algorithm is a dynamic programming approach. The basic idea behind the approach is that there are two possibilities in each step: should an item be taken in the knapsack (the solution), or not. Since Bouknap is just used as a tool to solve the relaxation, it will not be further explained here. Detailed information on the algorithm can be found in Kellerer et al. (2004) and Pisinger (2000). The source code used for Bouknap is identical to the code used for the research in Pisinger (2000).

Note that the problem solved here is a minimization problem. However, Bouknap is developed to solve the standard single-dimensional bounded knapsack problem, which is a maximization problem. So before using this tool, the problem needs to be reformulated to a maximization problem. This is done in the same way as has been shown in section 5.2.

One of the limitations of Bouknap is that it is only able to deal with integer values. However, due to the multiplication with the multipliers, it will be a coincidence when one of the weights, the profits or the capacity will be an integer. So it is necessary to round them to integer values. The downside is that the problem will become less accurate, and with it its solution. The inaccuracy has been reduced to a minimum by multiplying these values with a constant, before rounding them. This way, more digits become significant, thus minimizing the information lost. On the other side, the constant cannot be too large. Otherwise, the values for the weights, the profits and the capacity will become too large for Bouknap to handle, especially when the problem contains many rows and columns. The value used for the constant was 100. After some quick tests, it appeared that using this value gives better solutions, while Bouknap was still able to handle the input, even with larger problems. In a future research, it is interesting to investigate the influence of this constant more extensively. For example, the constant can be made problem-size specific: a smaller size of the problem allows the constant to take a larger value.

The bounds used for the surrogate relaxation problem are not the same as with the Lagrangian relaxation problem. For the surrogate relaxation problem, the bounds taken are the maximum workloads of the rows each column is covering. These bounds would not give good

results for the Lagrangian problem and a somewhat different method of finding these bounds was used (section 7.3). But there it was possible to construct the bounds while the problem was being solved. The entire surrogate relaxation problem will be used as input for Bouknap, so the bounds need to be determined in advance. A method to reduce these bounds beforehand was not found, but it probably will be worthwhile to keep searching for one. The time complexities of some recursions within Bouknap are proportional to the size of the bounds (Kellerer et al. (2004)), so reducing the bounds will result in less computation time needed, and probably a better solution. If one allows the algorithm the same amount of time to run, more iterations can be made, thus increasing the chance of finding a better solution.

Chapter 8

Results

The algorithm that was described in the previous chapter has been tested on several cases. Section 8.1 explains how these tests were performed and under which circumstances. The test cases originate from both literature as well as real life practice. But they all have in common that they are also used by Woutersen (2005). Section 8.2 gives a small summary of these cases, and highlights the special properties of a problem which has been tested. For each test case, section 8.3 shows the results, including a summary of the most remarkable ones.

8.1 Description of the tests

The method of Caprara et al. (1999) (and so the method that has been implemented) makes use of several parameters. Initially, the original values of these parameters have been used. The reason is that this research is mainly focused on the difference between the two relaxation methods, and not on fine tuning the method for solving Crew Scheduling Problems. However, if during small tests better values were found, these were used in the remainder of the research. The parameters which were used with a different value are:

- Maximum computation time. In contrast with the research of Caprara et al. (1999), a limit on the computation time was set. The algorithm was not allowed to start with a new iteration after 15 minutes from the start. With some exceptions, this means that the total computation time will not be longer than 15 minutes and a few seconds. This research was focusing on a fast algorithm, which could be used for real time planning. That is why the total computation time was limited, with the assumption that 15 minutes is a reasonable time to wait for planners.

- Maximum number of iterations. The original value was set to 10 times the number of rows, in this research this value was doubled to 20 times the number of rows. The reason for this change is that during the implementation phase it became clear that the maximum number of iterations would be reached for most of the problems, while only a small part of the maximum computation time was used. To give the algorithm the opportunity to find better solutions, it was allowed more iterations to perform.

- Lagrangian cost threshold. This parameter is used when solving a Lagrangian relaxation. For a minimization problem, a column will be selected when its Lagrangian cost lies below the threshold. The usual value for this threshold is 0. However, Caprara et al. (1999) decided to use the value 0.001. In their problems, they encountered a lot of columns with

Lagrangian costs close to 0. This means that there are a lot of near optimal solutions. They showed that if they also selected all or most of these ”near optimal columns”, the lower bound converges faster. In the cases described in section 8.2, the relative amount of ”near optimal solutions” is not as large as in the problems of Caprara et al. (1999), so their approach would not be beneficial here (in fact, only more columns would be selected that are removed directly in the next step).

For these tests, it is assumed that the solution must be acquired in a short amount of time. This reflects the situation that a person needs to get the solution fast. If this person is not satisfied with the solution, measures can be taken (changing the conditions if possible, hire extra people etc.). The other extreme is that there is enough time for the calculations. This is for example the case when the program is allowed to keep running all night long. This increases the chance of finding the best solution possible. However, this much running time is not always necessary to find the best solution, as will be shown with some results later on. In fact, it will be shown that in most of the cases, the solution will converge rather fast, and that even 15 minutes is more than enough.

During the tests the upper and lower bounds found in each iteration are stored. This enables the research on the behavior of the solutions found (constant versus variable). The second property that was looked at is the best solution found so far at each iteration. This gives some insight in how fast the solutions will converge to the best possible ones.

The computation time was not researched. The main reason is the method of implementation of both methods. Woutersen (2005) used the program CPLEX for solving all the problems. This is a commercial program, which has been optimized by professional programmers for fast performance. The subgradient algorithm has been implemented using the programming language Delphi, with average programming skills. Delphi is not a preferred programming language for calculations such as these (C++ is a better choice when it comes to performance). Delphi was still chosen because of the compatibility with the program written by Woutersen (2005). This way, her duty sets could directly be used for testing the subgradient algorithm.

8.2 The test cases

The subgradient algorithm was tested on the same test cases as used by Woutersen (2005). One of the advantages of using the same cases is that the optimal solution is known for all cases. With optimal solutions, the results can be judged on their quality. Another advantage is the time saved. Since the focus of the research was set on selecting an optimal set of duties, it would be convenient if the feasible duties to select from were already generated.

There is one test instance that has not been used in this research. The case about Falck Airport Security contains the constraint that conflicts with the definition of a pure Set Covering Problem (see section 7.2.1). Of course, it would be possible to remove these constraints. But with this modification, the optimal solution found by Woutersen (2005) could not be used anymore for reference. After the test phase, there was doubt if using the Falck test case would have been useful anyway. There are two other test cases with similar properties, mainly the property that workstations need to be occupied 24 hours a day. These cases (Company X and Adelaide casino) had several problems, for example extreme long running times. The expectation was

that the Falck case would suffer from similar problems.

In the sections below, the remaining test cases will be explained briefly. A short summary will be given, where the most interesting properties will be highlighted. More information about the different cases, as well as graphs and tables about the workstation demand and constraint, can be found in the work of Woutersen (2005).

8.2.1 Company X

The Company X case is a real life one. It clients rely on its services 24 hours a day. Therefore, employees must be at work around the clock. This property makes it difficult to split the case into smaller subproblems. This was addressed and shown with computational results by Woutersen (2005). The result was that the overstaff became larger, due to overlap in accepted duties.
It is interesting to see if the same problem arises when the subgradient algorithm is used. For this, the cases were executed twice. One time the duty set for an entire week was determined, the other time with the same week divided into days. The split was made at midnight. Not only is this the natural border between days, it was also a time of day were the demand was at its lowest.
A second property which could be examined with this case was the number of starting times allowed for each duty. In the most restrictive set of constraints, duties may only start within certain intervals. At one starting time, duties where even compulsory to have a certain length. In the least restrictive set, duties were allowed to start when needed.
All other constraints were kept the same across each test case.

8.2.2 Call center

This test case has its origin in the research of Musliu et al. (2004). Again, this is real life data, and concerns a call center.
The difference between the sets of constraints lies in the basic units of the duty lengths. The length of a duty is a multiple of this basic unit. For example, when the basic unit equals 30 minutes, duties with a length of 7 hours and 30 minutes are allowed, but a length of 7 hours and 15 minutes is forbidden.
The tests with these sets of constrains are executed twice; with or without the constraint concerning breaks. In the first case, a break must be incorporated within the duties. In the second case, the duties do not contain any breaks.

8.2.3 Adelaide casino

Adelaide casino is located in Australia. The demand for security is such, that 24 hours a day security officers need to be present in the casino. The cases are taken from Mills and Panton (1992).
This case looks a bit like the case of Company X. Here, there is a 24 hour demand as well. Also, the cases are tested for an entire week and are split in several subproblems with the size of a day. The different sets of constraints however are more diverse compared to the sets of the

Company X case.

8.2.4 Cash desk employees

Woutersen (2005) has taken the implicit formulation proposed by Thompson (1995). He tested his proposal with a few examples. One of them is a case about the cash desk employees of a supermarket.
The special property that can be investigated with this case is again the length of the duty. This time, not the basic unit has been researched, but the length itself. Or more precisely, the diversity of the duty lengths. With each set of constraints, more or less duty lengths are allowed.

8.2.5 Traffic exchange operators

The final test case comes from an article written by Henderson and Berry (1976). They discussed an example about traffic exchange operators. Although the operators are needed on a 24 hour basis, a duty set will not be created for all these hours. Instead, the duties are only generated for the workstation demand between 6:00 a.m. and midnight each day. During the night, the demand is relatively stable but the hours are unattractive for the employees. Therefore, it has been chosen to look at these hours separately.
This time, special attention is directed to the workstation demand itself. Each set of constraints has been tested with three different workstation demands. One time with the original demand size, the other times with the demand doubled or tripled.

8.3 Results

In the subsections below, the results for each of the test cases will be discussed. The layout will roughly be the same for each case. The main subjects are the course of the values for the upper and lower bounds, the best lower bound found and finally the best upper bound found (which belongs to the best solution found).

8.3.1 Company X

First, the results of the tests where the problem has been split will be discussed. A sample of these results are shown in figure 8.1. Only the results for the Monday subproblem of set D are shown, other subproblems and sets show similar results. One of the plots shows the bounds found in each iteration, the other one shows the best bound values found up to each iteration.

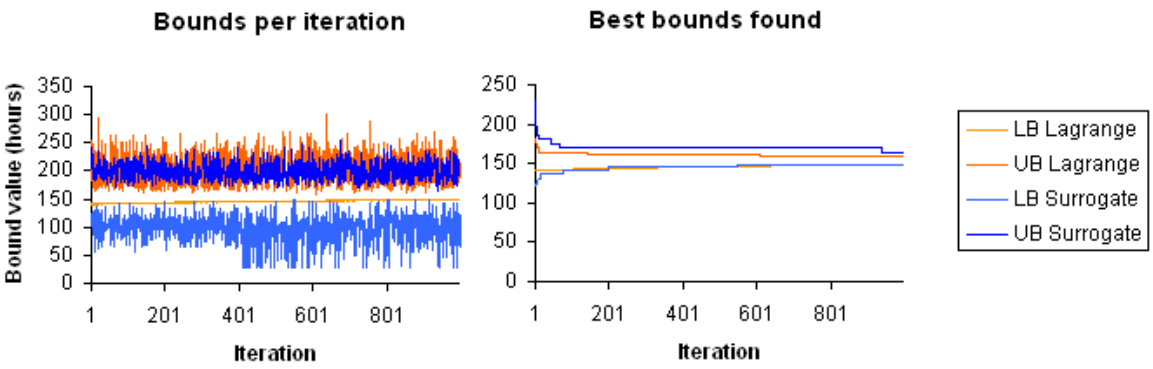

Figure 8.1: Results for the ”Company X” case, set D (split per day), Monday

Two things can be noticed from the plots. One thing is that the lower bounds computed with Lagrangian relaxation are relatively stable. There is not much variation in these lower bounds. However, for the lower bounds computed with the surrogate relaxation as well as the upper bounds computed with both methods, something different can be seen. The variation in these values is very large. Also, large differences may occur from one iteration to the next. However, there still exists some consistency. Even though relatively large or small values occur regularly, there seems to be a maximum or minimum value to these values. This phenomenon appeared not to be exclusive to the Company X cases. This can be seen in the results of all test cases.

The plots with the best bounds found so far also show some interesting results. The largest improvements are made rather quickly. The upper bound does not improve in most of the iterations. If there is an improvement, then it is most likely that this improvement takes place within the first half of the executed iterations. The lower bound does keep improving, but the improvements become smaller when more iterations have been executed.

These observations do not change when the number of starting times are more restricted or not. Restrictions on starting times only influence the number of iterations which can be executed within the same amount of time. With more starting times allowed (with more feasible duties as a direct consequence), the problem will be more difficult to solve, thus needing more time to be solved.

If there was a competition between the Lagrangian relaxation and the surrogate relaxation when it comes to the lower bound, then the surrogate relaxation would have won that competition. In most of the cases, the surrogate relaxation gave the best lower bound. However, this does not guarantee that the surrogate relaxation also gives the best upper bound. In fact, the upper bound of the Lagrangian relaxation was the best upper bound in almost all of the test runs.

But it seems that for both relaxation methods the lower bound improves when more starting times are allowed. In fact, for the cases with the least restrictive set of constraints, the lower bounds are just 0.98 percent (Lagrangian relaxation) or 0.17 percent (surrogate relaxation) away from the optimal solution. This seems to be a logical result. More feasible duties available means that there are more possibilities to find a solution that lies close to the optimal one.

However, this logic apparently does not apply to the upper bound. The gap between the

best lower bound and the best upper bound roughly stays the same for all sets of constraints. This means that with more starting times allowed, the upper bound or solution values tend to get further away from the optimal solution. This must not be confused with the solution value itself. As expected, the values themselves do become smaller with less restrictive sets.

The most important value to look at is of course the upper bound or solution value. For the Company X case, the subgradient algorithm combined with Lagrangian relaxation showed lower upper bounds compared to the surrogate relaxation. Another observation is that the solution values do not go down as fast as the optimal solution with more allowed starting times. This has to do with the gap between the lower and upper bound, as was described above.
There was one little surprise for the least restrictive case. For three of the seven subproblems (or days), the starting solution appeared to be the best solution. The best solution of the subgradient algorithm (with either relaxation method) was still worse than the solution following from the method described in section 7.2.1.

Unfortunately, these results are the only valid results of this case. As explained before, tests were also performed on cases where the week was not split into days. But none of the tests were finished in the allowed running time of 15 minutes. One of the cases was finished after 45 minutes, the other two cases were stopped because they took even more time. It appeared that the algorithm was still busy with its first iteration. This indicates that the iterative algorithm to remove redundant columns might not be suitable for larger or more complicated problems. As expected (since only one iteration was completed), the results of this case are rather poor. The best solution found for this problem was 23.68 percent overstaffing with Lagrangian relaxation or 37.04 percent with surrogate relaxation while the overstaffing for all the other cases (with problem sizes of a day) lies around 10%.

Summarizing the descriptions above, the general results for these cases are as follows.
The best solution will be found when someone uses the Lagrangian relaxation. Also, the lower bounds seem to be constant. If somewhat more stability of the upper bound as well as a higher lower bound is preferred, then surrogate relaxation is the better relaxation method.
The number of starting times allowed seems to have some influence on the lower bound. With less limitations (so with more feasible duties), the difference between the optimal solution and the lower bound will become smaller. However, the difference between the lower and upper bounds seems to remain the same.

8.3.2 Call center

The same observations were made as with the Company X case: a stable lower bound with Lagrangian relaxation, the other bounds are fluctuating, but seemingly within boundaries. If a closer look is taken at this fluctuation, a few differences can be seen if breaks need to be incorporated within the duties. Without breaks, the fluctuation of the upper bound computed with surrogate relaxation looks similar to that of the upper bound computed with Lagrangian relaxation. But if the break constraint is applied, the Lagrangian upper bound varies more. The second difference is the absolute value of the fluctuation. With breaks, the 'maximum' upper bound becomes larger. Figure 8.2 shows these observations for the Saturday subproblem of two constraint sets.

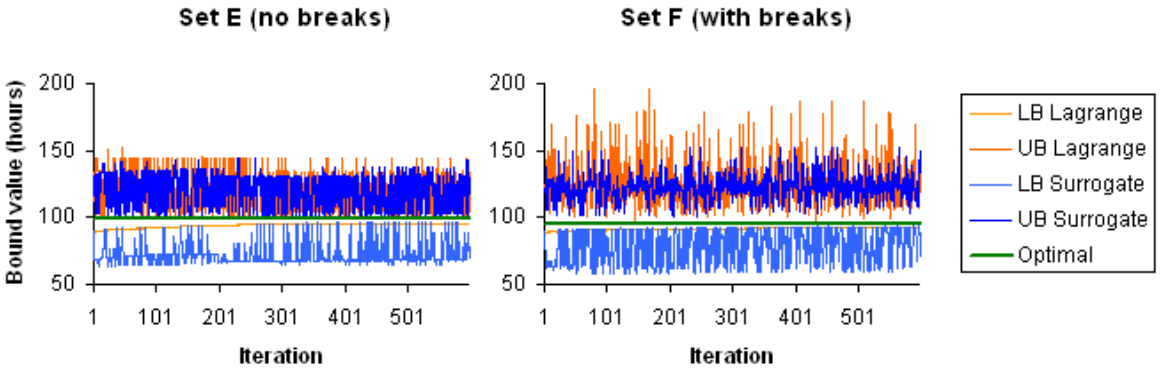

Figure 8.2: Bounds found in each iteration for the ”Call center” case, Saturday subproblem

When the best results found so far (a sample of these are shown in figure 8.3) are studied instead of the results for each iteration, the following can be seen. The upper bounds rapidly improve in the beginning of the process. A lot of improvements happen in a short amount of iterations. After relatively few iterations, there seems to be no more improvements. If another (small) improvement does happen, it usually happens within the first half of the process. The lower bounds on the other hand do keep improving during the whole process. But the improvements become smaller when more iterations are executed.

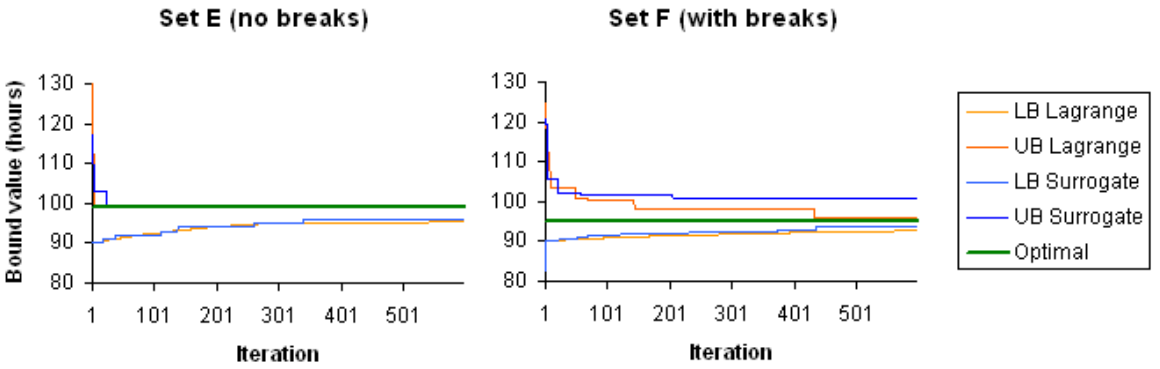

Figure 8.3: Best bounds found for the ”Call center” case, Saturday subproblem

There seems to be no difference in the fluctuation or the speed of improvements when the basic unit is changed.

In all cases, the lower bounds computed with surrogate relaxation were better, in the sense that the difference between these bounds and the optimal solution are smaller. This is remarkable, since also in all cases, the upper bounds computed with Lagrangian relaxation are the better ones. So better lower bounds do not guarantee better upper bounds.
Breaks do not have an influence on the difference between the lower bound and the optimal solution. However, when breaks are applied, the difference between the lower and upper bounds becomes roughly twice as big.
Again, different sizes of the basic unit seems to have no influence on the lower bounds.

In all cases, the algorithm with Lagrangian relaxation generates the better solution value. Without breaks, the overstaffing remains below 1 percent, which does not count for the surrogate relaxation. With breaks and the Lagrangian relaxation, the overstaffing remains below 5 percent, while the overstaffing acquired with surrogate relaxation rises above 10 percent. The optimal solution will be better approximated when no breaks need to be added to the duties. An optimal solution was even found for the Saturday and the Sunday subproblems, if no breaks had to be added (see also figure 8.3 for the Saturday subproblem). Just as with the values found in each iteration, as well as the best lower bound, the size of the basic unit seems to have no influence on the solution value. A characteristic shown by Woutersen (2005) was that the overstaff increases when breaks need to be applied and the length of the break is shorter than the length of the basic unit. However, with the relaxation algorithm, this cannot be seen.

All this gives the following general results for this test case. Again, the method combined with Lagrangian relaxation generates the best solution value, and must be chosen if one is only interested in this result. The upper bound of the surrogate relaxation on the other hand does not fluctuate as much as the one of the Lagrangian relaxation. And again, the lower bound of the surrogate relaxation does approach the optimal solution more closely.
No influence of the length of the basic unit on the results was found. However, the algorithm seems to have some trouble when dealing with breaks. The differences in results between iterations become larger, the solution value found lies further away from the optimal solution value and the difference between lower and upper bound also increases.

8.3.3 Adelaide casino

The test case for the Adelaide casino is the second one where there is continuously demand, so it is the second one where the demand has been tested as a whole and split into separate days. First, the tests for the entire week will be discussed.

The first two constraint sets, A and B, were unusual in the following sense. To find a feasible solution, it was necessary to add some duties, which do not entirely lie in the interval that is optimized. This causes extra overstaff to occur due to duties which are selected twice, while they might have been selected only once if both periods are optimized at the same time. Therefore, Woutersen (2005) added penalty costs to ensure that duties will be selected which lie within the optimized interval as much as possible (thus trying to reduce the forced extra overstaff to a minimum). When solving these problems with the relaxation algorithm, incorrect behavior of the lower and upper bounds was observed. To see what the effect on the algorithm and the final solution value would be, the tests were executed with and without penalty costs.

For set A a relatively large amount of iterations has been executed (3066 iterations). For the other sets, this number was surprisingly small, with 11 at the maximum. The algorithm could only execute 1 iteration for set B and C (which were allowed to take respectively 46 and 87 minutes). For sets G and H, the first iteration took even longer, so the algorithm was aborted (just as with the week tests of Company X, recall section 8.3.1). One note, finding the optimal solution was also troublesome for these sets, but there an intermediate result was returned. Since not even the first iteration was finished, the algorithm could not do this. So (not taking set A into consideration), it seems that the algorithm has some trouble when dealing with continuous workstation demand. Just as with the Company X case, the algorithm needs a lot of

time to reduce the redundant columns in its temporary solutions. Still, for sets A, D, E and F a familiar sight can be detected. Again, the lower bound of the algorithm with Lagrangian relaxation looks very steady, while the other bounds are fluctuating.

More surprises can be found when the lower bounds are studied. For sets D (shown in figure 8.4), E and F, the lower bound of the surrogate relaxation remained below the same bound of the Lagrangian relaxation. For most of the other test cases, the Lagrangian lower bound was usually the lower one. Of course, with the low amount of iterations, it cannot be predicted if the surrogate lower bound would have eventually become larger. But in this case, the Lagrangian lower bound was already very close to the optimal solution right from the start. This reduces the chances for the surrogate lower bound to become better than the Lagrangian one in the long run. Another observation on the lower bound is that the difference between the lower and the upper bound is relatively large. When the difference is expressed as a percentage of the upper bound, the difference is around 5 percent for sets E and F. For sets C and D, the difference becomes much larger, ranging from 17 to 79 percent.

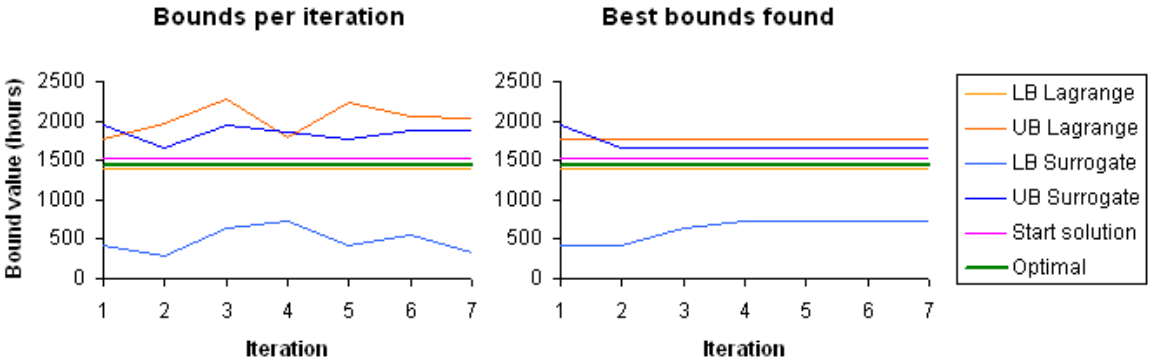

Figure 8.4: Results for the ”Adelaide” case, set D (entire week)

If this upper bound is studied more closely, it can be seen that, except for set A, the solution of neither relaxation method was the best. For these test cases, the starting solution appeared to give the best solution. The solution often was relatively good too: For set E and F (sets with a lot of variation in the feasible duties) the solution value was only 1.28 percent larger than the optimal solution value, for set B the difference was even smaller (1.07 percent).
Adding penalty costs would not be wise to do here. Not only incorrect bound values are returned when using surrogate relaxation, but for set A (see figure 8.5[1]) the Lagrangian relaxation even found an optimal solution (with the surrogate relaxation only 0.61 percent away) when not adding penalty costs. Set B gave performance problems, but even with only one iteration executed, the Lagrangian relaxation found a better solution without penalty costs, and the start solution was even better (with a solution value only 1.07 percent away from the optimal one).

[1]The best lower bounds of the surrogate relaxation for the first four iterations are 680, 780, 1100 and 1100. For a better overview of figure 8.5, the vertical axis starts at value 1200, and so these values are not shown.

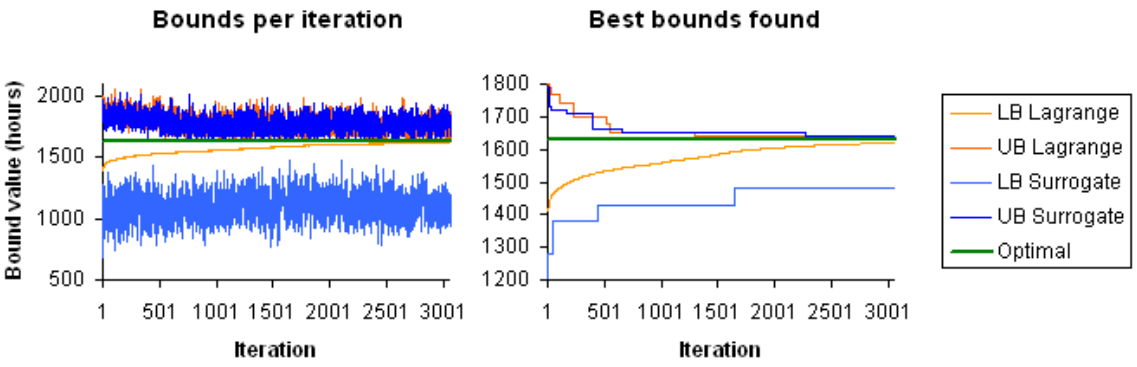

Figure 8.5: Results for the ”Adelaide” case, set A (entire week)

In this case, it can be seen that a lot of variation in the feasible duty set leads to better solutions. The problems are such that only a few iterations can be executed within the 15 minutes allowed. This means that the advantage of executing more iterations (thus increasing the chances of finding a better solution) is not valid here.
Because of the strange behavior (such as the small amount of iterations), it is hard to tell which of the relaxation methods is the better one. However, the starting solution showed to be better than both methods in most of the cases.

These were the results for when the entire week was calculated at once. But how are the results when this problem was split into problems with the size of a day?

For most of the subproblems, the lower bound of the surrogate method is very unstable. There seems to be more variation compared to the other test cases. For many subproblems, the surrogate lower bound looks just as unstable as the Lagrangian one, while in most of the cases, the variation should be less. Also, in most of the times, the lower bound of the Lagrangian method needed more time than usual to converge to a value near the optimal solution. However, a strange exception can be seen. For set F, it starts normally (the usual variation). But from a certain point, the values do not change. Think of a heart that stops beating. The same can be seen with set A, but there the variation starts appearing again after a number of iterations. An example can be seen in figure 8.6.

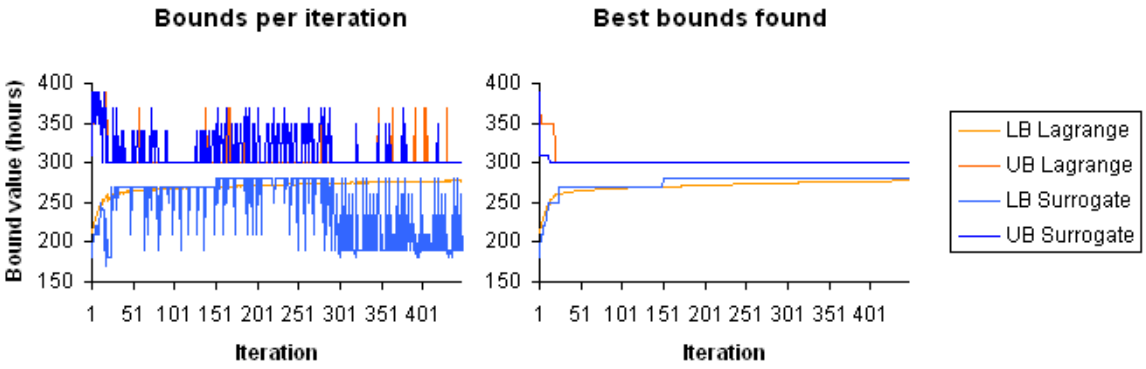

Figure 8.6: Results for the ”Adelaide” case, set A (split per day), Friday

The difference between the lower and upper bounds differs very much, ranging from less than 1 percent, to more than 15 percent of the upper bound.
One exception is set F, which showed a difference of -0.59 percent for the Lagrangian method. This should not be possible. This means that the (best) lower bound is higher than the (best) upper bound. However, the lower bound should always be less than or equal to the upper bound. The program was searched through thoroughly, but the error which caused this could unfortunately not be found. This means that caution must be taken when making conclusions considering the lower bounds. The upper bounds are all fine: the solutions forming these bounds are all feasible.

It appeared that it doesn't really matter what the distance is between the best lower bound and the optimal solution. For constraint set E where the distance was 0.61 percent as well as constraint set C where the distance was 7.97 percent, an optimal solution was found. For set E (the Monday subproblem is shown in figure 8.7), it appeared that for at least one iteration, the lower and upper bound were very close to each other. Considering the fact that the Lagrangian lower bound converges rather fast in most of the cases, one might consider the use of a so-called epsilon parameter. When the difference between both bounds becomes smaller than this parameter, the algorithm could be stopped early.

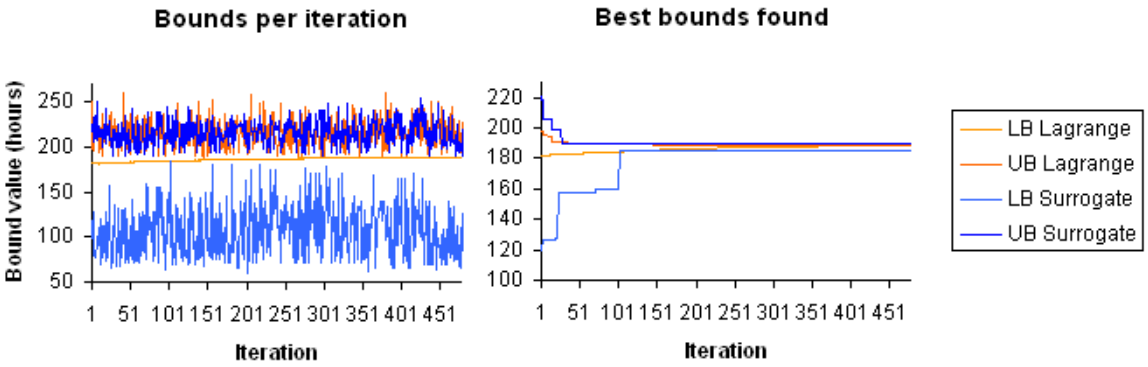

Figure 8.7: Results for the "Adelaide" case, set E (split per day), Monday

When it comes to approaching the optimal solution with the upper bound, the algorithm performs much better when the problem has been split. For the worst case (constraint set B), the difference between the upper bound and the optimal solution was 10.26 percent of the optimal value. The difference remained below 1 percent in most of the cases. Even better: for four cases, an optimal solution was found. When comparing these duty sets with the duty sets generated by Woutersen (2005), it can be seen that both solutions are not identical. This shows that for a certain problem, the optimal solution doesn't need to be unique.

With regard to the overstaffing, the same is shown as by Woutersen (2005). The overstaffing itself is lower when computed for the entire week at once, even while the difference between the solution value of the method and the optimal value is larger.

Again, the starting solution was eventually the best solution for a lot of the problems, six to be exact. For only 1 of these problems, the starting solution really was the best. For the other five, the best solution of the relaxation method was equal to the starting solution.

Considering the penalty costs: this time it is indeed better to add them under the condition that Lagrangian relaxation is used (for the surrogate relaxation, bound values again appear to be incorrect). Although without penalty costs, the Lagrangian relaxation found a solution for set A with a value only 1.01 percent away from the optimal solution value. But with penalty costs added, the Lagrangian relaxation was able to find an optimal solution. For set B, the differences become even larger (2.14 percent with penalties versus 6.41 percent without).

This all leads to the following observations for the tests with the problem split up. Because the subproblems are now much smaller, more iterations could be executed. The result is that the optimal solution was better approximated. For most of the cases however, the execution of these iterations wasn't a useful exercise, since the starting solution (which was already good by itself) turned out to be the best one. It doesn't matter how close the lower bound approaches the optimal solution. Even when the gap to the optimal value was relatively large, an optimal solution could still be found. But still, it would be better to solve the entire week at once. The solution value will be lower in this case, since overlap in duties will be reduced.

8.3.4 Cash desk employees

The lower bounds of the Lagrangian relaxation for the cash desk are again relatively constant. The lower bounds produced by the surrogate relaxation are unstable, but now and then higher than the Lagrangian lower bounds. However, this doesn't imply a better upper bound: in some of the iterations where the lower bound of the surrogate relaxation was better, the upper bound remained above the upper bound of the Lagrangian relaxation. The upper bounds themselves are again stable and unstable at the same time. The values changes from iteration to iteration, sometimes with a rather large difference, but the bounds always seem to stay within certain boundaries.

For most of the different subproblems of set B and C, the lower bound found with the Lagrangian relaxation lies very close to the optimal value. For most of the subproblems, the lower bound is not more than 1 percent less than the optimal solution value. With the surrogate lower bounds, something else can be seen. Sometimes, the lower bound rises above the optimal value. This is probably due to the undetected error in the program (see also section 8.3.3). In general, the differences between the bounds are rather large, ranging from 4.61 percent to over 27 percent of the upper bound.

The algorithm makes the largest improvements of the lower bound in the first few iterations. In later iterations, the improvements made will be rather small most of the times. Within the first half of the iterations executed, the lower bound has converged (as in become relatively stable). Sometimes a (large) improvement has been made in the second half, but this didn't occur often.

The Cash desk employees case is another case where it was better to stop before the algorithm was even started. For both set B and C, the starting solution appeared to give the best solution. If only the relaxation methods are considered, then the Lagrangian relaxation is the better one for all test cases. In this case, the addition of the surrogate relaxation served no purpose.

The influence of the size of the feasible duty set can be seen in the overstaff. It appeared that the overstaff will be smaller when the size of this set increases. This is a logical result: the larger the set, the more options there are.

Figure 8.8 shows the results for one subproblem as an example to show some of the conclusions stated before.

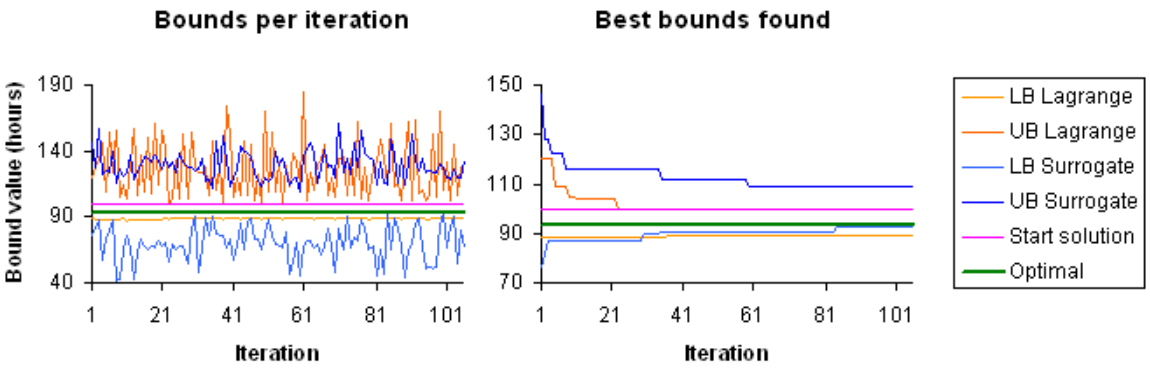

Figure 8.8: Results for the ”Cash desk employees” case, set B, Wednesday

So it appeared that with more diversity in the feasible duties, the lower bounds will better approach the optimal value, and the overstaff will be lower. However, the distance between the lower and the upper bound increases. This has its impact on the overstaff. If another method is used for transforming a solution of the relaxation into a feasible solution, the difference between the bound might become lower. This makes the improvement a diverse duty set makes on the overstaff even larger.

This test case showed that the execution of more iterations is debatable. Sometimes a better solution was found in one of the last iterations. However, the solution was often of a good quality after only a quarter of the iterations executed. It depends on the demanded quality of the solution as well as the resources and time available if the execution of more iterations is useful or not.

8.3.5 Traffic exchange operators

The final test case also shows a constant or increasing Lagrangian lower bound, and indicates the presence of boundaries for the lower and upper bounds. For this particular test case, the variance of the upper and lower bounds seems to be less than the variance for the other test cases. Excluding set A, the Lagrangian lower bounds seem to be very constant. When plotted, these lower bounds do not seem to change during the course of the algorithm, and lie very close to or are equal to the optimal solution value. Remarkable is that the Lagrangian upper bounds still vary. In most of the cases the best solution value is found within the first half of the iterations executed. So again, good solutions are found relatively fast. When the best possible solution is needed, more iterations might be useful: now and then an even better solution was found in the second half.

The course of the values per iteration is not influenced by the size of the workstation demand. In figure 8.9, somewhat similar plots (of course with different values) can be seen for all demand sizes. Not even a relation between the size and the number of iterations used can be detected.

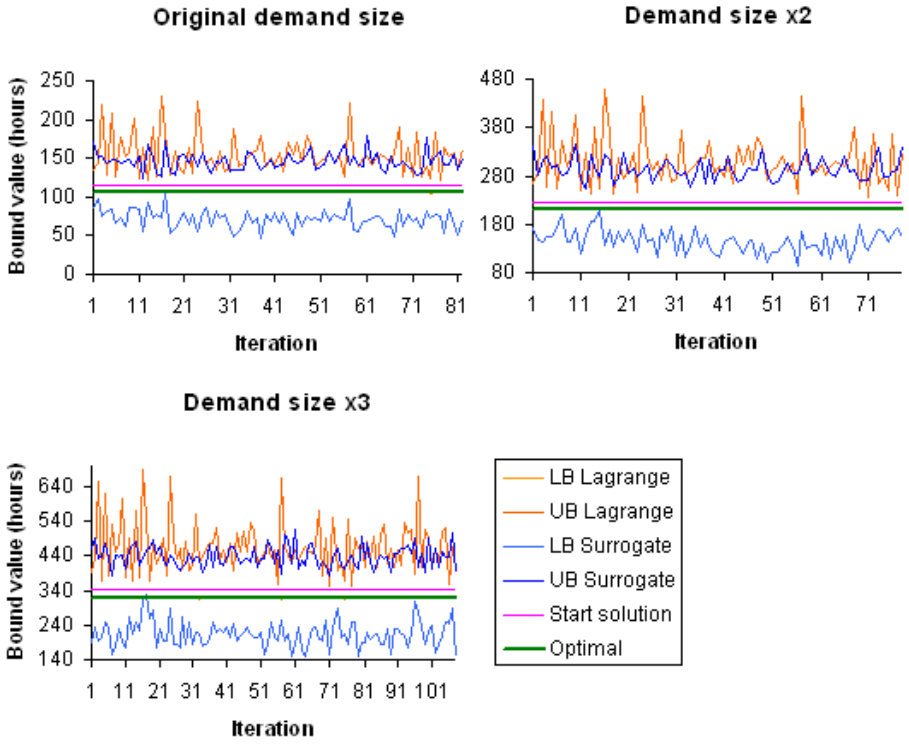

Figure 8.9: Bounds found in each iteration for the ”Traffic exchange operators” case, set D

The best lower bounds usually lie very close to the optimal solution value, or are even equal to this value. This is mainly true for the Lagrangian relaxation, the lower bounds of the surrogate relaxation seems to have a little bit more trouble in approximating the optimal solution value. Figure 8.9 also gives an illustration of this: the plot of the lower bound of the Lagrangian relaxation can hardly be seen since it is overlapped by the plot of the optimal solution value. The surrogate lower bound sometimes becomes larger than the optimal solution value. This should not be possible, but the cause is probably the error as mentioned in section 8.3.3.

There are some exceptions: set A and F. Set A will be explained later on in this section. The results of set F with original demand size are shown in figure 8.10, but the same can be seen when the demand size is multiplied. The lower bound of the surrogate relaxation lies relatively far from the optimal solution value (while the Lagrangian lower bounds equals the optimal value). This doesn’t need to be a problem, but comparing with the other sets, the difference between the optimal value and the lower bound is rather large.

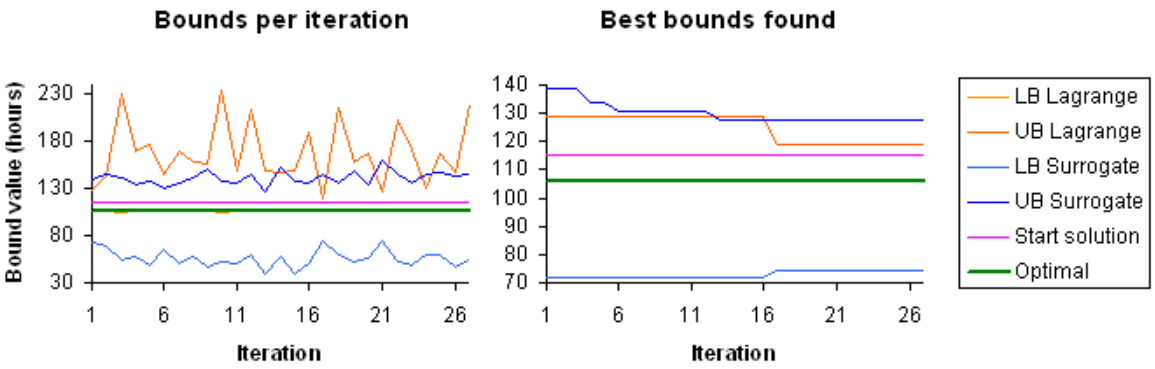

Figure 8.10: Results for the ”Traffic exchange operators” case, set F, original demand

The difference between the lower and upper bounds is on the other hand relatively large. Combined with the observation that the lower bounds are generally very close to the optimal solution, it shows why the solutions generated for this test case are not very good (not close to the optimal solution). For set F, where the surrogate lower bound was far from the optimal solution, this might have generated a better upper bound. Unfortunately, the difference between the lower and upper bound was extra large, so the solution value is not better than the solution value for other sets or relaxation methods.

The size of the workstation demand seems to have no influence on the lower bounds.

This test case proves once more that the starting solution must not be underestimated. For all subproblems except one, the starting solutions appeared to give the best solution. This is also the reason why the line for the starting solution value is omitted in figure 8.11, the line is identical to the one of the optimal solution value. The best relaxation method was the Lagrangian one: Lagrangian relaxation generated better results than the surrogate relaxation for all subproblems.

There was one particular set (A), which did not behave as other sets did. The results for the original demand size are shown in figure 8.11. First, even the optimal solution lies already far from the requested workstation demand (overstaff is 20.75 percent). This amount of overstaff was also found by the algorithm: both the starting solution heuristic as the Lagrangian relaxation found an solution with this amount (while the surrogate relaxation did not). This is probably caused by the fact that only duties of one duty length were allowed. This creates a very limited set of feasible duties, so it should not be difficult to find an optimal solution value for this set.

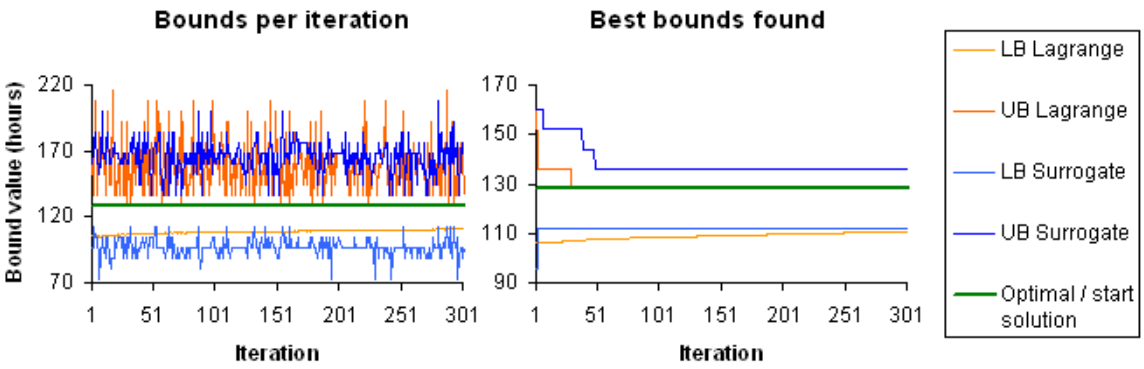

Figure 8.11: Results for the ”Traffic exchange operators” case, set A, original demand

For the other sets, the algorithm did not perform that well. Since the optimal solution had (almost) no overstaff, the algorithm resulted in solutions with a decent amount of overstaff (note that the lower bounds were close to the optimal solution value, but the difference in lower and upper bounds were large).

As expected, the optimal value multiplies with the same factor when the workstation demand is multiplied. For the algorithm the same can be seen. However, the solution value is not exactly multiplied with the same factor, it deviates slightly. For example, for set F, the solution values found by the surrogate algorithm are 127.5, $249,75$ and 433 for the single, double and triple workstation demand cases.

These results can be summarized as follows. The Lagrangian relaxation is again the better relaxation method considering the final solution value. Also the lower bounds seem to be more stable. However, the starting solution must not be forgotten, since this was often the best solution value. The size of the workstation demand does not appear to show some influence on the bounds and the course of the algorithm. And finally, this test case also confirms the results found by the Thompson case: set A shows that the solution will be better when less different duty lengths are allowed.

Chapter 9

Conclusions

Using the results from chapter 8, some conclusions can be drawn. This has been done for different aspects that were encountered. Section 9.1 shows the difference between both relaxation methods. In section 9.2, some thoughts are mentioned about the algorithm itself. The conclusions in section 9.3 are related to the nature of the problem, and how it affects the performance of the algorithm. And finally, section 9.4 states the final conclusions and advice.

9.1 Differences between the different methods

When it purely comes to the final results, then Lagrangian relaxation proves to be the best one. For nearly all the test cases, this relaxation method generated the best final solution. However, the starting solution should not be underestimated. Once in a while this solution already appeared to be the best: Lagrangian relaxation as well as surrogate relaxation were not able to find a better solution for these cases.

Since the starting solution performed unexpectedly well, it was given a closer look. Taking a closer look at how the starting solution was formed, we see that it basically comes down to executing the the GREEDY heuristic (3), with multipliers set to 0. So, for cases where the best solution of the relaxation methods were equal to the starting solution, it would have been interesting to see if the multipliers have converged to 0. Unfortunately, the multipliers were not logged for research: the main focus of the research was the (final) solution values of the relaxation methods, results like these were not anticipated.

The upper bounds that were found during the course of the algorithm showed a lot of variation. On the other hand, nearly every upper bound seemed to be 'semi-constant': the bounds themselves show a lower and an upper bound. The upper bound of the surrogate relaxation varied less than the upper bounds of the Lagrangian relaxation. Unfortunately, this does not translate in better results.

Even though the program contained an error regarding the lower bounds, some conclusions can still be made. The lower bounds of the Lagrangian relaxation did not show irregularities like with the upper bounds. It converges steadily, or even remains constant. The lower bounds of the surrogate relaxation were less stable, but sometimes did come closer to the optimal solution value. Note that this cannot be taken as a measurement of the quality of the lower bound. Some tests (for example the Adelaide case) showed that the optimal solution value can be found, both

with ’good’ and ’worse’ lower bounds.

9.2 The algorithm itself

The algorithm is not very consistent in its performance. Often the solution value found lies close to the optimal solution value. This holds especially for the Lagrangian relaxation. As much as often the final solution value still lies relatively far from the optimal solution value. This is partially due to the formulation of the problem, which will be described later on in section 9.3.

The maximum allowed amount of computation time as well as the maximum number of iterations appeared to be more than enough for most test cases. Good solutions were found early on. During the second half of the iterations executed only slightly better solutions were found, and only once in a while. It is however up to the planner if the best solution value must be found, which takes a lot of computation time. This because a solution is not guaranteed optimal while the lower and upper bounds are not equal.

The way the algorithm is currently implemented does have trouble with the larger problems. Especially problems with workstation demand for 24 hours a day, which are solved for an entire week, give some trouble. It doesn’t matter if the number of variables becomes large, or the number of constraints, both properties make the algorithm a lot slower. This can be solved by designing a new algorithm for removing redundantly selected duties, or even accept that they exist.

9.3 The nature of the problem

The properties which seem to have no influence on the algorithm or its results are the size of the workstation demands and the multiples of the allowed duty lengths.

One of the properties that did show its influence is the amount of starting times. If more starting times were allowed, the lower bound gave a better approximation of the optimal solution value. However, the difference between the lower and the upper bound did not change. In other words: with more starting times, the final solution value became worse (as in further away from the optimal solution value). The number of different duty lengths showed a similar influence: more allowed duty lengths gave a better lower bound while the upper bound became worse.

This leads to the following conclusions. A change in the problem formulation will only influence the results and the course of the algorithm when this change has an impact on the size of the set with feasible duties. In general, the lower bound will approach the optimal solution value more closely when there are more feasible duties available to select from. But at the same time the difference between the upper bound and the optimal solution value will increase, since the difference between both bounds does not change. There is one exception to this rule however. If the number of iterations executed is relatively small, a larger set of feasible duties does improve the solution.

Another property of a duty which influences the algorithm is the presence of a break. The algorithm seems to have trouble when dealing with breaks. The algorithm needs more time for each iteration, both bounds have more trouble with approximating the optimal solution value and so on.

Problems with a 24-hour workstation demand can best be solved for intervals as large as possible. This indeed contradicts with what the algorithm can handle. Smaller problems can be dealt with more easily (as already mentioned in section 9.2), with the result that the optimal solution value is approximated more closely. But in this case, looking at the approximation alone is not enough. As shown by Woutersen (2005), the total overstaff increases when the calculated interval is split into smaller ones. This conclusions remains valid for the algorithm. So although the problems become easier to solve, the total overstaff increases with smaller intervals. Therefore, final solution wise, it is still better to solve the problem for an interval as large as possible.

9.4 Advice about the usefulness

It is hard to conclude from this research if the designed algorithm is useful as a replacement for an external solver like CPLEX. There is a lot to improve as it is now, mainly how it is designed and implemented. These improvements are essential in order to solve larger scale problems in a reasonable amount of time.

But this research did show the first signs that the algorithm itself has potential to be a good option. Already solutions are found which are close to the optimal solution value, in a reasonable amount of time. The expectation is that, if the settings of the different parameters are fine-tuned, even better solutions can be found and/or the computation time will decrease.

In the current form it is advised to keep using the Lagrangian relaxation for the following two reasons. First, solving a surrogate relaxation problem (which is a Singledimensional Knapsack Problem) needs more time than solving an Lagrangian relaxation one (which can be solved straightforward due to the absence of constraints). And secondly, in the current algorithm the surrogate relaxation can only be used as some sort of supplement. All updates are carried out using information of only the Lagrangian relaxation. The surrogate relaxation is only used to give a shot at finding a better solution than the solution found by the Lagrangian relaxation. The 'unfortunate' part is that the final solution of the Lagrangian relaxation is the better one of the two. So eventually, not using the surrogate relaxation will only save computation time.

Chapter 10

Recommendations for future research

As was concluded in the previous chapter, a lot of extra research and improvements are needed to make the algorithm work. This chapter therefore will give some directions in which the research may be continued.

Continuation with task based demand The search for a method to generate duties based on task based demand is far from being complete. In chapter 3, a few possibilities are proposed for such a method. To finish the original research intentions, it is necessary to implement each alternative, and run some tests to compare them with each other (similar to what has been done now with the Lagrangian relaxation and the surrogate relaxation). This way, it can be determined which combination of alternatives works best to solve this duty generation problem. Of course, these are not the only existing alternatives, one can search if there are even better ones available.

One of the most intriguing propositions is to use topological sort to generate all feasible duties. For Vehicle Routing Problems, this method has been researched before. But for solving a Crew Scheduling Problem, this is a fairly new approach. In section 3.2.1 some directions are given to start with, but there are still a number of issues that need to be solved. Whether this approach will work good is something a new research should find out.

Rework on the current implementation The method does not work optimally the way it is currently implemented. There are still lots of opportunities to make it work better.

One of them is of a technical nature. As mentioned in section 8.1, the algorithm was developed in Delphi for compatibility reasons. However, Delphi should not be the programming language of choice when it comes to calculations like these. Other languages such as C++ are far more efficient. So implementing the exact same algorithm in another programming language will already result in shorter computation times. The program was also written by someone with decent knowledge and experience when it comes to programming, but not a true expert. Surely a true expert will be able to find some more improvements within the code.

Another opportunity lies within the use of the different parameters. For this research most parameters had gotten the values as used by Caprara et al. (1999). A new research can be dedicated to find other values for these parameters which work best for this algorithm. For example, if the parameters used for the stop criteria are correctly tuned, either computation time

can be saved or the opportunity is created to find an even better solution (when the algorithm is not stopped before finding this solution). The same will apply when the step size is updated correctly. There is a downside however. To truly optimize the duty selection process, a new research is probably needed every time the algorithm is applied in a certain situation (like a new company that decides to use it).

Additions to the algorithm Next to improving the current implementation, the algorithm can also be enhanced by adding new features.
The algorithm's stopping criteria can be expanded with some sort of ϵ parameter. Such a parameter is most commonly used to indicate the quality of a solution. If the (absolute or relative) difference between the solution value of the solution found and the optimal value is less than or equal to ϵ, then this solution is considered to be 'good enough' and will be returned as the result of the algorithm.

The only problem is, the optimal solution value is usually unknown. After all, the algorithm is executed to search for it (or something close to it). But in this case measuring the difference between the lower and upper bound against ϵ is sufficient. After all, since the optimal solution should be within those two bounds, this difference is always greater than or equal to the difference between the solution value found (the upper bound) and the optimal value.

The use of ϵ must be examined thoroughly. It is up to the planner to decide which deviation from the optimal solution value is still acceptable. It is also important to take the performance of the algorithm in consideration. In some of the test cases, it appeared that a relatively good solution was found in a short amount of time, and that the algorithm needed a lot more time to improve the solution only slightly. So accepting a solution which is a 'little less good' may reduce the computation time significantly.

Finally, one of the pleasant surprises was that there was actually a 'third' method, which was fairly good: the starting solution. In quite a number of test cases it appeared that the method used to generate an initial solution immediately provided the best solution. If it is possible to come to the conclusion that this solution is already good enough, than it can be taken directly as the result of the algorithm. This saves all the computation time that was needed otherwise to perform all the iterations. This leads to some interesting questions: "Is it possible to use the start solution generation heuristic as a new method, next to Lagrangian relaxation? And if so, how should it be implemented best?"

Bibliography

J.P. Arabeyre, F. Fearnley, F.C. Steiger, and W. Teather. The ariline crew scheduling problem: a survey. *Transportation Science*, 3:140–163, 1969.

F. Barahona and R. Anbil. The volume algorithm: producing primal solutions with a subgradient method. *Mathematical programming*, 87:385–399, 2000.

F. Barahona and R. Anbil. On some difficult linear programs coming from set partitioning. *Discrete Applied Mathematics*, 118:3–11, 2002.

J.E. Beasley. A langragian heuristic for the set covering problem. *Naval Research Logistics*, 37:151–164, 1990.

A. Caprara, M. Fischetti, and P. Toth. A heuristic method for the set covering problem. *Operations research*, 47(5):730–743, Sep.-Oct. 1999.

G.B. Dantzig. A comment on edie's "traffic delays at toll booths". *Operations research*, 2: 339–341, 1954.

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

R. Freling. *Models and techniques for integrating vehicle and crew scheduling*. PhD thesis, Tinbergen Institute, Erasmus University Rotterdam, 1997.

R. Freling, R. Lentink, and M. Odijk. Scheduling train crews: a case study for the dutch railways. In S. Voss anJ. Daduna, editor, *Computer-aided scheduling of public transport, lecture notes in economics and mathematical systems*, volume 505, pages 153–166. Springer Publishers, 2001.

F. Glover. Surrogate constraint duality in mathematical programming. *Operations research*, 23 (3):434–451, May-June 1975.

H.J. Greenberg and W.P. Pierskalla. Surrogate mathematical programming. *Operations research*, 18(5):924–939, September-October 1970.

W.B. Henderson and W.L. Berry. Heuristic methods for telephone operator shift scheduling. *Management Science*, 22:1372–1380, 1976.

H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack problems*. Springer-Verlag, 2004.

J. Kisteman. Pickup and delivery problem and customer satisfaction. Master's thesis, University of Groningen, January 2004.

D. Klabjan, E.L. Johnson, G.L. Nemhauser, E. Gelman, and S. Ramaswamy. Solving large airline crew scheduling problems: random pairing generation and strong branching. *Computational optimization and applications*, 20:73–91, 2001.

K. Lansdorp. Van taak tot dienst. Master's thesis, Erasmus University Rotterdam, December 2002. In Dutch.

E.C. Leslie. Traffic delays at toll booths. *Journal of the Operations Research Society of America*, 2(2):107–138, May 1954.

L.A.N. Lorena and F.B. Lopes. A surrogate heuristic for set covering problems. *European Journal of Operational Research*, 79:138–150, 1994.

R.G.J. Mills and D.M. Panton. Scheduling of casino security officers. *Omega: International Journal Of Management Science*, 20:183–191, 1992.

N. Musliu, A. Schaerf, and W. Slany. Local search for shift design. *European Journal of Operation Research*, 153:51–64, 2004.

F. Niks. Solving the capacitated pickup and delivery problem with time windows and precedence relations using a statistical clustering method and conditions. Bachelor's thesis, Erasmus University Rotterdam, 2005.

D. Pisinger. A minimal algorithm for the bounded knapsack problem. *INFORMS Journal on Computing*, 12:75–84, 2000.

M. Teeuw and A. Woutersen. Het samenstellen van dienstroosters bij tpg post. Bachelor's thesis, Erasmus University Rotterdam, December 2003. In Dutch.

G.M. Thompson. Improved implicit optimal modeling of the labor shift scheduling problem. *Management Science*, 41:595–607, 1995.

A. Wagelmans. Advanced mathematical programming, college year 2004-2005. Colleges notes, 2005.

A. Wagelmans and M. Gootjes. Personal conversation, September 2005.

L.A. Wolsey. *Integer programming*. Wiley-Interscience Series in Discrete Mathematics and Optimization. Wiley, John & Sons, Incorporated, 1998.

A.M.E. Woutersen. Diensten genereren in de praktijk. Master's thesis, Erasmus University Rotterdam, May 2005. In Dutch.