A comparison of several heuristics for crew rescheduling at NS

Ronald van der Velden (298970)

Bachelor thesis

Erasmus Universiteit Rotterdam

Supervisor: Dr. D. Huisman

July 5, 2009

Abstract

In this thesis, we present two methods for solving the Crew Rescheduling Problem at Netherlands Railways (NS). The first method is based on a Resource-Constrained Shortest Path Problem (RCSPP). The second one is a conflict-based approach, based on the methods used in practice by human planners. Both methods are tested and fine-tuned extensively. We show that especially the RCSPP-based approach gives fast and good results. We also compare both methods with the method of Column Generation with Dynamic Duty Selection which is described in [11]. We show that this last method takes significantly more computation time, but that it usually gives better results than the greedy methods described in this thesis.

Contents

Chapter 1

Introduction

A lot of effort is put in the yearly process of designing rolling stock and crew schedules at Netherlands Railways (NS), the largest railway company in The Netherlands. Schedules contain about 10000 tasks (as defined in chapter 2), 2800 rolling stock units, 3000 drivers and 3500 conductors (according to [4]). So developing these schedules is a serious challenge.

In this context, Operations Research has proven to be very useful. In 2001, a new method for creating crew schedules was introduced which considered both schedule efficiency and employee satisfaction. Five years later, a completely new timetable was created that resulted in an estimated profit increase of 40 million euro. This research received the Franz Edelman Award for Achievement in Operations Research and the Management Sciences.

The schedules are used day after day and it would be ideal if no deviation from these schedules ever occurred. However, reality is different. Each day, both employees and passengers suffer from multiple disruptions. Technical failure of rolling stock, obstructed tracks and accidents cause several trips to be changed or cancelled. Whenever a single trip is changed, this has consequences for other trains and employees. Since employees and rolling stock both perform multiple tasks on a single day, a single trip change may have many consequences throughout the country.

Therefore, at several locations throughout the country, dispatchers of Network Operations Control (NOC, see [6]) carefully watch all deviations from the schedule and solve the problems that arise. This has to be done very quickly to prevent problems from accumulating. Usually, first the rolling stock schedules are revised and then the crew schedules are updated. A detailed problem description is given in chapter 2.

In this paper, we focus on crew rescheduling. In the past few years, a lot of research has been done on advanced methods for rescheduling train and airline crew, yielding good results. Huisman and Pothoff ([11]) have shown that column generation can be used succesfully for crew scheduling at NS, especially when combined with methods to select subsets of duties for rescheduling. This method, along with others, will be discussed in chapter 3.

In this paper we will discuss two easier, greedy methods for solving the same problem. The first method is discussed in chapter 4. This method is based on the Resource Constrained Shortest Path Problem (RCSPP), which is also used in [11] as the pricing problem for the column generation algorithm. However, we do not completely update the graphs iteratively but use a greedy method to quickly solve each RCSPP once. The second method is based on conflict-solving, which is also done by human planners at NOC. We will try to solve conflicts or shift them to a later point in time. This method is presented in chapter 5.

We will first fine-tune both methods, testing an extensive set of parameter settings for each method. We will see how each of them influences solution quality, where quality is seen as a combination of several factors. Finally, we will compare both methods with the method described in [11]. The experiments and results are described in chapter 6. Finally we will come to a conclusion in chapter 7.

Chapter 2

Problem description

2.1 Terminology

One of the most important concepts in the crew rescheduling problem is that of a ***task***. Each task consists of a departure time and an arrival time, as well as a start and an end station. Between these two stations, the train might stop at several additional locations. An example is a task from Amersfoort (14:12) to Zwolle (15:07). This local train also stops at 10 intermediate stations, but no crew changes can be made during these stops. Amersfoort and Zwolle are examples of so-called relief ***locations***. In the crew scheduling phase, we also know what rolling stock is used to perform a task.

Multiple consecutive tasks can have the same ***train number***. These tasks are then performed one after another, usually with the same rolling stock (although train units may be coupled or decoupled at several stations). The task mentioned above is performed after the task from Utrecht (13:49) to Amersfoort (14:11). These tasks form train 5647 from Utrecht to Zwolle. All train numbers of the form 56xx together form the ***line*** Utrecht-Zwolle, where odd numbers run from Utrecht to Zwolle and even numbers from Zwolle to Utrecht.

For each task, a driver is needed. The set of tasks of one specific driver on a single day constitutes a ***duty***. The duties are usually created whenever a new timetable is introduced. Each duty contains driving tasks, transfer time between these tasks and (optionally) a meal break and/or some tasks during which the driver does not really drive a train, but only uses it as a passenger to get from one station to another. This is called ***deadheading***. An example of a duty is shown in Table 2.1.

Start	End	From	To	Train	Role
05:46	06:32	Amsterdam	Alkmaar	4008	Drive
06:46	07:10	Alkmaar	Hoorn	3408	Drive
07:20	07:44	Hoorn	Alkmaar	3431	Drive
07:54	08:34	Alkmaar	Amsterdam	829	Pass
08:55	09:45	Amsterdam	Gouda	4031	Drive
09:46	10:08	Gouda	Rotterdam	4031	Drive
10:18	10:31	Rotterdam	Dordrecht	1935	Pass
11:44	12:10	Dordrecht	Breda	2241	Drive
12:20	12:46	Breda	Dordrecht	2238	Drive
12:48	13:07	Dordrecht	Rotterdam	2238	Drive
13:11	13:32	Rotterdam	Den Haag HS	2238	Drive
13:32	13:44	Den Haag HS	Leiden	2238	Drive
13:45	14:04	Leiden	Haarlem	2238	Drive
14:06	14:22	Haarlem	Amsterdam	2238	Drive

Table 2.1: Duty Asd:17 in dataset Lls_A. The horizontal line at 10:31 indicates a meal break.

2.2 Rescheduling

Because of smaller and larger disruptions, some of the tasks are modified or cancelled during the day of operations. This may lead to time or space conflicts: transfer times may become too short, or a task that ends at a certain station is followed by another task that starts at another station, with not enough time between them to transfer from the first station to the second. It might also occur that the last task does not end at the depot where the driver is based, or that the meal break is no longer possible due to the extension of one or more tasks.

By reassigning tasks to other duties, feasibility has to be restored for all duties. According to [11], we have to take into account the objectives of feasibility, operational cost and robustness. First of all, a proposed schedule has to satisfy the constraints before it can be used in reality. In the case of NS, the operational costs of rescheduling consist of time (extending duties) and using taxis. Finally, since all changes have to be communicated to the crew, schedules are more reliable when they deviate less from the original schedule.

2.3 Restrictions

As mentioned before, not every assignment of tasks to duties is feasible. There are several constraints that each duty has to satisfy:

- When task v is planned after task u, then the destination of u should equal the origin of v.
- Between two tasks u and v which are performed using different rolling stock, enough transfer time should be planned. This transfer time depends on the role of the driver during tasks u

and v: When the driver has to drive task v, more transfer time is needed than when he is deadheading on v.

- A duty should preferably be no longer than 8.5 hours, but may be extended with at most one hour if necessary.
- Duties longer than 5.5 hours have to contain a meal break of at least 30 minutes at one of the relief locations with a canteen. This break must be planned somewhere in the middle of the duty, so that the time before and after the break does not exceed 5.5 hours.
- Each duty should contain a sign-on and a sign-off period.
- The destination of the last task in a duty should be the depot of the corresponding driver. In case of a severe disruption, a taxi may be necessary to transfer drivers back to their depot.
- Not every driver may drive on every part of the network. For each task, we have a list of depots where the drivers are based that may perform the task.

2.4 Actual duty status

An important aspect of processing the input data, is checking the actual status of each duty. Since rescheduling is done during operations, each duty may have one of the following statuses at the so-called ***time of rescheduling***:

- Not started: Some of the duties have not started yet, so the complete duty can be modified
- On train: A lot of drivers are deadheading or driving a train when rescheduling takes place. These tasks have to be completed by the originally assigned crew since the train has probably already left the station. It might be the case that the task has been replaced by another one, which has another end time and/or destination.
- On transfer: Some drivers have already completed a task and are waiting at the station for another train. If they are assigned to another task instead, we have to make sure that there is again enough time left to warn the driver that he has to go to another platform.
- Finished: Some duties are already finished when we start rescheduling. We do not have to take these duties into account.

2.5 Data

When we have to reschedule, we need a lot of data such as:

- The set of tasks for the whole day. Some of these tasks may be cancelled or replaced by a modified task, in which the times or end station are changed.
- The set of employees with their original duty (as it was at the start of the day) and their current duty (which might already be changed due to earlier disruptions). The latter one may no longer be feasible. Due to the cancellation or replacement of tasks, the duty might for example contain place or time conflicts.
- The set of relief locations (stations and depots). For each location, we know whether it is possible to have a meal break there.
- The set of reserve duties available at each large railway station.
- Information about train numbers and rolling stock. This can be used to check whether two tasks are performed using the same rolling stock. If that is the case, no transfer time is needed between the two tasks. First of all, we know for each task which task is performed immediately after the first one, using the same rolling stock. Secondly, we have information about coupling and decoupling.

Chapter 3

Literature

In this chapter, we will briefly discuss relevant literature about crew scheduling and rescheduling, both in the railways field and in other areas. We will discuss which methods are currently in use and which are the most important differences between railway crew rescheduling and some other areas.

3.1 Timetabling at Netherlands Railways

3.1.1 Timetabling process

Railway firms face a lot of complex problems when developing new timetables. Sometimes it is possible to modify an existing timetable, but this is not always the case. As described in the award-winning article [8], Netherlands Railways introduced a completely new timetable in 2006. The article describes the different stages of the scheduling process. Below we will shortly summarize these stages:

- Line planning: determining the routes and frequencies of trains. For example, the decision was made to run one InterCity train from The Hague to Groningen every hour, and two Sprinter trains from Utrecht to Rhenen per hour.
- Event scheduling: determining arrival and departure times of each train at each station, bridge etc. A cyclic timetable is generated so that the departure and arrival times are the same for every hour of the day.
- Routing trains through stations: developing route plans for each train at each station. Especially at larger stations with a lot of switching tracks, this is a complex problem. It might turn out that no feasible routing exists for a given event schedule.

- Rolling stock scheduling: assigning rolling stock units to each train so as to create enough seat capacity on each route. There are several types of rolling stock. Furthermore, it is possible to combine and split units of rolling stock which increases the complexity of this problem.

- Crew scheduling: assigning drivers and conductors to each train. Given the set of tasks that have to be performed on a given day, duties have to be created for each driver and each conductor so that each task is assigned to one duty. Labour rules play an important role in this process, as will be discussed later.

3.1.2 Crew scheduling phase

Over the years, Netherlands Railways have used diffent methods to create the crew schedules. Before turning to computer-aided scheduling, crew scheduling took 6 full-time months of work by 24 planners, according to [9]. To support the planners, the CREWS system was developed by SISCOG.

The system consisted of three parts: data management, prescheduling and scheduling. In the prescheduling phase, tasks were coupled to create possible sequences. Tasks and sequences were partitioned in several sets, for which the schedules could then be created quasi-independently. Scheduling could be done manually, semi-automatic and automatic and it was also possible to use a combination of these modes in the so-called mixed-mode. All modes were based on a search tree and duties were created by adding (sequences of) tasks step-by-step.

The next system was TURNI, developed by Double-Click. It was introduced in 2000 and used during the development of a new set of labour rules called "Sharing Sweet and Sour" as an alternative to the "Circling-the-Church" principle that was proposed by the board of the company.

TURNI is described in more detail in [7]. TURNI consists of a duty-generation and a duty-selection module. The problem is modeled as a set covering problem: A subset of all possible duties has to be chosen, which covers all tasks, while minimizing the total cost and satisfying several constraints. Dual information from the solution is used to generate new duties, which are then added to the set covering problem.

3.2 Papers on crew rescheduling

In the last few years, attention has shifted from scheduling problems towards rescheduling problems. Since planners often face complex planning problems when one or more large disruptions take place, and these problems have to be solved in as little time as possible, computational support would be more than welcome. Since little time is available and it is important to minimize the number of changes to the existing schedules, the software used for normal crew scheduling cannot be used for disruption management.

In [4], crew rescheduling during planned track maintenance is considered. The authors combine column generation with Lagrangian relaxation, as is discussed in more detail in [5]. The developed methods are capable of solving large rescheduling problems, but solving the test cases took between 9 and 16 hours of computation time. That is acceptable for the given maintenance problems, since maintenance is planned several weeks ahead.

The techniques have been adapted in [11] to speed up the rescheduling process for unpredicted disruptions. Extensions include smart methods to generate core problems which only contain a small subset of all duties and tasks and several speed improvements such as column fixing and partial pricing. When not all tasks can be covered, neighbourhoods of the initial core problem are explored to solve for the remaining tasks. Experiments show that rescheduling now takes several seconds to five minutes, and in most cases all tasks can still be assigned.

3.3 Airline crew rescheduling

Crew rescheduling also takes place in the airline industry and is among others described in [10]. First the flight schedules are recovered and then the crew schedules are revised. The working hours can be modified, trips can be switched to different employees or reserve employees may be used to solve the problem. Different approaches have been tested and implemented in real-life: column generation using RCSPP as a pricing problem and a heuristic approach using depth-first brand and bound.

However, there are important differences between (disruptions at) airline and railway companies. Usually the tasks in railway companies are shorter and therefore a railway duty contains more tasks than an airline duty. Furthermore, railway firms like the Netherlands Railways do not use overnight-duties. In other words, each day ends at 04:00 at night and each day can be seen seperately. This also implies that rescheduling is only necessary for a single day.

Chapter 4

RCSPP-based approach

The first approach that we will include in our experiments, is the graph-based greedy approach described in [3]. The authors have two objectives in mind. The primary goal is to minimize the number of additional task cancellations, while the secondary goal is to minimize deviation from the initial schedule. In this paper, we will construct an adapted version of this method and compare it to other methods.

We will first describe the original approach that is used in [3]. We then turn to the definition of a ***Resource-Constrained Shortest Path Problem*** and show how we can use a RCSPP labeling algorithm to improve the greedy approach. Finally, we describe in detail, how the RCSPP is constructed. Experiments to fine-tune the parameters and compare the method to others, are described in Chapter 6.

4.1 Original algorithm

To solve the rescheduling problem, all affected duties are selected. All tasks in these duties, that start after the time of rescheduling, are aggregated in one list. The duties are sorted according to one of several possible criteria. For each duty, a directed acyclic graph is built where each task is represented by a node and possible transfers between tasks are represented by arcs. For the current position of the driver and the duty ending, a source and sink node are created.

With each node, a value is associated which indicates how attractive it is to include the associated task in the duty under consideration. Tasks which were already included in the original duty get a higher value than the tasks which were not. Tasks which have already been assigned to another duty in an earlier stage of the algorithm, are given a slightly negative value so as to minimize the amount of deadheading.

When a duty graph is built, a modified shortest path algorithm is used to find the path from source to sink having the largest value, where the value of a path is defined as the sum of the values of all visited nodes. The algorithm exploits the fact that the graph is acyclic and directed.

When the optimal path is found, a check is performed to make sure that a meal break is possible (if necessary) in the modified duty. If not, the authors say they use the second-best path, although they do not describe how this path is found using their algorithm.

When the sink node cannot be reached, the end time of the duty is increased in steps of 30 minutes until a feasible path can be found. When all affected duties have been rescheduled but there are uncovered tasks left, reserve duties are considered.

4.2 Resource-Constrained Shortest Path Problems

4.2.1 Definition

The authors of [3] do not go into detail when describing the checks for meal breaks. There are two possibilities for finding paths that satisfy the break criteria:

- Find the k best paths through the graph, ignoring the break rules. Afterwards, check if the optimal path satisfies the rules. If not, select the second-best path and so on.
- Model the problem as a ***Resource-Constrained Shortest Path Problem*** (RCSPP). One can, for example, take as a resource the time that is left before a meal is needed. The solution to this RCSPP satisfies the break criteria.

According to [1], the RCSPP first appeared in [2]. The problem is to find an optimal (usually shortest) path through a network, while satisfying a set of constraints, defined over one or more resources such as time, distance or vehicle load. At each node, there are ***resource windows*** which indicate what the feasible values for the resources are at that specific node. A ***resource extension function***, associated with each arc, defines how each resource changes as it is passed on from one node to each other.

An example of a RCSPP instance is given in Figure 4.1. The goal in this example is to find the path from node S to node T that gives the largest revenue. In the graph, each node has a revenue value attached to it. The revenue of the path is defined as the sum of the revenue associated with all visited nodes. As resources, we take the fuel consumption and the path revenue. For each arc, the fuel consumption is shown. Note that in this example, paths with greater revenue and less fuel consumption are preferred over paths with a smaller revenue or a larger fuel consumption.

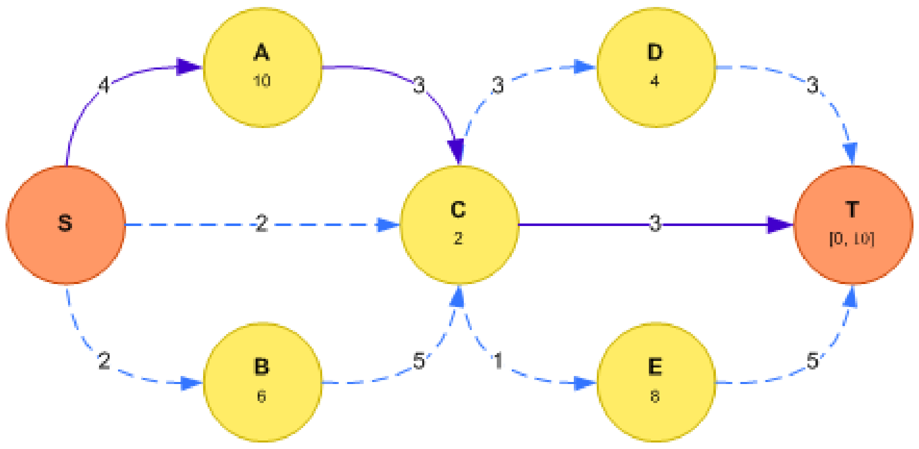

Figure 4.1: Example of a RCSPP. The thick arcs represent the optimal path. The numbers on the arc indicate the fuel consumption. The numbers on nodes A-E indicate the revenue for visiting that node.

The resource extension function is quite simple: When one travels along an arc, the fuel-resource is increased by the consumption associated with that arc. The revenue is increased by the amount associated with the node one visits. There is only one resource window, associated with the sink node: The total fuel consumption at the end of the trip should be no more than 10.

4.2.2 Labeling algorithm

A labeling algorithm can be used to solve the problem, as suggested in [1]. With each partial path from the source to some arc, a ***label*** is associated which describes the state of the resources at the end of the partial path. A label at node i can be extended to an adjacent node j, by applying the resource extension function associated with the arc (i, j) and checking if the new label will be feasible according to the resource window at node j.

In this paper, we will only consider directed acyclic graphs and therefore the above algorithm can be applied efficiently without cycling. We use a topological node ordering, so we order the nodes in such a way that a certain node comes before all other nodes to which it has an outbound degree.

At each step during the execution of the algorithm, we select the next node from this ordered set of unvisited nodes. All labels at this node are forwarded to neighbour nodes and the current node is marked as visited. When a label a is forwarded creating a new label b, a link from b to a is maintained to be able to track back the optimal path from sink to source. When all labels have been forwarded, the best label at the sink node is selected.

To speed up the labeling process, one can ignore several labels by applying dominance rules. A label a with resource vector $(a_1, ..., a_r)$ is (in case of minimization) said to dominate another label

Label	Iteration	Previous	Node	Revenue	Fuel	Notes
1	0	0	S	0	0	
2	1	1	A	10	4	
3	1	1	C	2	2	
4	1	1	B	6	2	
5	2	2	C	12	7	
6	3	4	C	8	7	Dominated by 5
7	4	3	D	6	5	
8	4	5	D	16	10	
9	4	3	S	2	5	
10	4	5	S	12	10	Optimal
11	4	3	E	10	3	
12	4	5	E	14	8	
13	5	7	S	6	8	
14	5	8	S	16	13	Infeasible
15	6	11	S	10	8	Dominates 13
16	6	12	S	14	13	Infeasible

Table 4.1: Iterations of labeling algorithm for RCSPP

b with vector $(b_1, ..., b_r)$ when all $a_i \leq b_i$ and at least one $a_i < b_i$. Labels which are dominated by other labels at the same node, can be excluded from the forwarding process. This reduces the number of labels.

4.2.3 Example algorithm execution

Table 4.1 shows the labels which are created during each step of the algorithm. The initial amount of fuel consumption is set to 0. We can see that label 6 with resources $(8, 7)$ is dominated at node C by label 5 with resources $(12, 7)$: When there is 7 fuel used, the best revenue you can get up till node C is 12, so there is no need to keep a path with revenue 8. The optimal path is $S-A-C-T$ with revenue 12 and 10 fuel used.

4.3 Modified algorithm using a RCSPP

To find the optimal duty completion for each driver, the authors of [3] construct a graph for each driver. In this paper we will use a combination of their approach and the one that is being used for the pricing problem in [11]. The construction of our graphs will be described below. For each driver, we search an optimal path through the graph, which corresponds to the best duty completion.

4.3.1 Nodes

We actually create one large RCSPP-graph for all drivers together. For each driver, we create a start node and an end node. Furthermore, for each task that has to be assigned to a driver, we create two nodes: A deadheading node and a driving node. When a single driving node is assigned to multiple duties, only one driver will do the actual driving while the others will be ***passenger-on-train***. This means, that all those drivers have had enough transfer time, to be able to drive that train, but only one does. When the driving employee leaves the train at a certain station, the others are able to take over the driving without any additional transfer time.

4.3.2 Arcs

Between two task nodes we create an arc if it is possible to perform the second task after the first one. Since we have two versions of each node, we can have up to four arcs between them. If the two tasks share the same rolling stock, if a meal break is possible or if there is at least *minTransferDriving* transfer time, then an arc between the two drive nodes is created. If a meal break is possible or there is *minTransferDriving* transfer time, then we connect the first passenger node with the second drive node. If neither arc has been created, but the tasks share the same rolling stock or a meal break is possible or the transfer time is at least *minTransferPassenger*, then both nodes of the first task are connected with the passenger node of the second task. If any of the first two arcs is created, we do not connect to the passenger nodes since driving or passenger-on-train is more attractive than deadheading and we do not destroy any paths by using this limitation.

An example of several task nodes and the arcs between them, is shown in Figure 4.2. In this example, we assume that *minTransferDriving* $= 15$ and *minTransferPassenger* $= 10$. The nodes that correspond to the earliest task (Rotterdam-Gouda) has a thick border. Of course it is possible to continue on the same train, which goes to Utrecht. However since the transfer time is only one minute, an employee who is deadheading to Gouda cannot drive the train to Utrecht. Since a driving employee can always continue driving, there is no need to create an arc to the passenger node of the task to Utrecht. A transfer to the train to Den Haag is also possible, but since the transfer time is less than 15 minutes and it is not the same rolling stock, the employee cannot drive the train. Finally, since the next train to Utrecht leaves after 16 minutes, the employee can always drive this train, even if he was deadheading on the previous train to Gouda.

Furthermore, we try to connect each task with each duty ending. If the task ends at the correct depot, the cost of the arc is only based on the end time of the duty, which is calculated from the

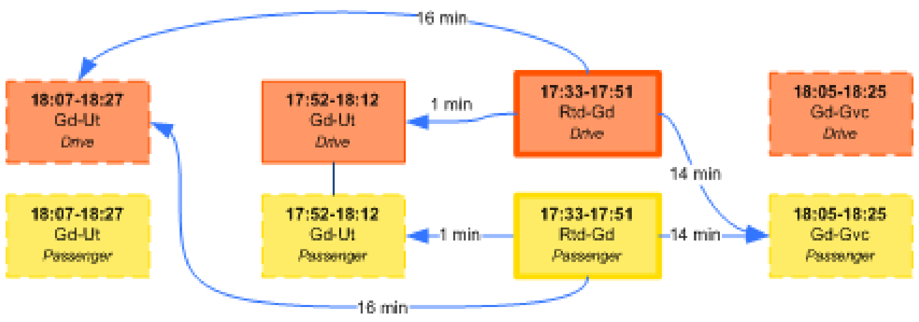

Figure 4.2: Illustration of the arcs that are created between tasks. For the thick-bordered task boxes, we show the outgoing arcs that are used.

end time of the task and the official sign-off time. If the duty ends no later than originally planned, the cost equals zero but otherwise we add a penalty, which is described below. When the task does not end at the depot, we assign a penalty of *costArtTaxi* with the arc for using a taxi. We also have to estimate the end time of the duty by calculating the estimated travel time of the taxi.

To do this, we calculate the travel time between each two stations before we run the algorithm. We do this by solving a shortest path problem for each end station, using Dijkstra's algorithm because we cannot have negative cycles. The arc length $l(a, b)$ between two stations a and b is here defined as the duration of the shortest task from a to b. Of course this gives a rough estimate, since transfer times are not taken into account. Taxi times may also be negatively influenced by traffic congestion, or positively because of the fact that there are far more roads than railway tracks. Because of this last fact, we estimate the taxi travel time to the depot by taking half the shortest distance found by Dijkstra's algorithm.

Finally, we add arcs from the source node of each duty to every possible task that starts after the artifical start time of the duty and that has the same start location as the duty. It depends on the actual status of each duty, which arcs we create. If the driver is currently on a train, we only create an arc to the corresponding task node. If the driver is on a transfer between two tasks, we create an arc if the task was the next planned task in the duty, or if there is enough transfer time left. If the duty has not started yet, an arc to the task is possible if there is at least *signOn* minutes between the start of the duty and the start of the task.

4.3.3 Resources

As resources in the RCSPP, we use the total score, the time left until the end of the duty and the time left without a break. With both the arcs and the nodes, we associate a certain score. For the arcs, this score is based on whether the transfer (to which the arc belongs) does already exist in the current solution or not. By using this score, it is more likely that sequences of tasks are created that also occur in the original planning, although maybe not in the same duty.

Furthermore, with each arc we associate the time and time-without-break consumption, plus a flag that indicates whether a break is possible between the two tasks. If no break is possible, both resource consumptions are equal. When a break is possible, the break consumption is equal to the duration of the second task, while the time consumption is the difference between the end of the second task and the end of the first task.

For the nodes, the score is adjusted every time a new duty is scheduled. We use the same scores as those in [3]. Driving nodes that still have to be assigned are given a value of 50 when the task was originally assigned to the current duty, and 10 otherwise. As soon as the task is assigned, a small negative score of -1 is used. This prevents the system from assigning a task twice. Passenger nodes get value -2. It is more attractive to have someone as a passenger-on-train than as a normal passenger. If the driver is too late, another driver is available to drive the train. Also, if value -1 is used for both modes, we get a lot of equal labels and that reduces the algorithm's speed. By using slightly different scores the number of labels is reduced.

4.3.4 Settings

For each node, we assign a list of depots from which drivers are able to perform the task that is associated with the node. When we run the algorithm for a certain employee, we only forward labels to nodes that can be performed by that specific employee. In [3] it is suggested, to use the same rule for normal deadheading and passenger-on-train nodes, even though in real life these modes of transport can be used by every employee for every task. The authors have shown that the quality of the solutions does not change significantly, while the calculation times are significantly reduced. We will check how this adjustment, which will be controlled by a new setting *usePassengerTrick*, influences the quality of the results and the computational speed.

In [3], the authors increase the maximum end time of the duty each time when the algorithm could not find a feasible path from source to sink. We will use a different approach: The maximum end time is set to one hour after the original end time. With each arc from a task to the sink, we associate a cost which is based on the difference in end time. Arcs that correspond to shortening the duty do not incur any cost. For the other arcs, the cost consists of a fixed part and a variable part that depends on the actual change in end time. We use paramaters *costEndLater* for the fixed cost and a cost per quarter of an hour of *costQuarterLater*. The influence of these settings will also be investigated.

Finally, we do not reschedule all duties, but only a subset which we call a ***core problem***, as suggested by [11]. Of course a smaller core problem means lower calculation time, but excluding too much tasks may also lead to infeasibility for some duties. We will compare several algorithms to generate these core problems. These will be described in Chapter 6.

Chapter 5

Conflict-based approach

5.1 Introduction

In this chapter, we will follow another approach to reschedule tasks in case of a disruption. This method more closely resembles the manual methods that are currently used by the dispatchers at NS. They mainly focus on ***conflict*** solving. Given the current duties in which some conflicts exist, they try to solve these conflicts by moving some tasks to other duties.

Several types of conflict may occur. Often we encounter a place conflict, as is the case in Figure 5.1. Due to an imaginary disruption between Amersfoort and Zwolle, the second train in duty A, which originally had to go to Zwolle, now turns and goes back to Amersfoort. However, the next task of this driver starts at Zwolle. Place conflicts may also occur at the end of a duty, when the last task does not end at the correct depot.

Also, we can have a time conflicts. Sometimes tasks become longer than planned, for example when a train has to turn near the end location of the task because of a disruption and drive back to

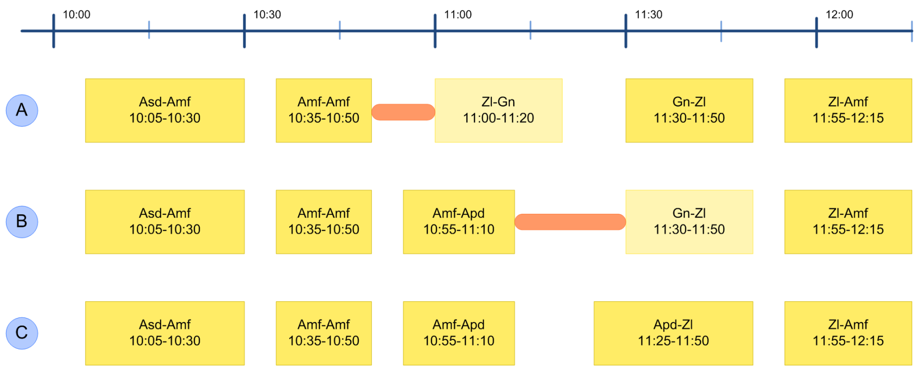

Figure 5.1: Example of a duty containing a conflict. The thick line indicates a conflict. By changing some tasks, the conflict is moved to the end of the duty until it is completely resolved.

the start location or when the turning takes some time because of heavy traffic near the disruption. If the end time is later than the start time of the next task, then we have a time conflict. When a duty becomes too long due to the replanning of the last task, we also have a conflict.

Finally, a planned break may become infeasible. We then have a break conflict: The driver has to work continuously without a break although he has to have a break after a certain amount of time.

5.2 Method

5.2.1 Solving conflicts

We assign to each conflict c the duty δ_c in which it occurs, the conflict type $type_c$ the preceding task $pred_c$ and the next task $succ_c$ with start time $succ_c.start$ (if any). Conflicts can be sorted by looking at the end time of the preceding task. The idea is now to solve one conflict at a time by removal and (optional) insertion of tasks in the corresponding duty, eventually creating a new conflict. By only changing the part of the duty *after* c, we may shift conflicts to a later point in time. This is something also seen at human planning: The earliest conflicts are solved first, since they are the most urgent ones. At the end, we will be left only with conflicts at the end of a duty (both time and space).

An example of this method can be see in Figure 5.1. A place conflict occurs at 10:50. By replacing the lighter-shaded task Zl-Gn by Amf-Apd, the conflict is solved. However, we find a new place conflict at 11:10 now. This can be solved by replacing Gn-Zl by Apd-Zl. Now the duty is feasible again. Note however, that we may now have two unassigned tasks (unless another driver was scheduled to deadhead on the same trips).

The basic method for solving conflicts is shown as Algorithm 5.1. We use the following symbols: C is the sorted set of all conflicts, C_δ contains only conflicts for duty δ. RST is the time of rescheduling. We ignore time conflicts at the end of a duty in line 5. For a given conflict, we first remove the successor (if any) in line 17 or, in case of a break conflict, all successors in lines 10-14; sometimes this solves the conflict. If the conflict still exists, we pick a task to insert after the conflict in line 32. If anything has changed, we update the set of conflicts in line 37 and start again.

Algorithm 5.1 Solving conflicts by task removal and insertion in `solveConflicts()`

```
Determine $C$
repeat
  for all $c \in C$ do
    if $(succ_c = \emptyset) \wedge (type_c =$'TIME'$)$ then
      Ignore $c$
    end if
    $changed = false$
    if $succ_c \neq \emptyset$ then
      if $type_c =$'BREAK' then
        for all Tasks $t$ in $\delta_c$ do
          if $t.start > RST \wedge t.start >= succ_c.start$ then
            Remove $t$ from $\delta$
          end if
        end for
      else
        if $succ_c.start > RST$ then
          Remove $succ_c$ from $\delta$
        end if
      end if
      $changed = true$
    end if
    $solved = true$
    Determine $C'_{\delta_c}$
    for all $c' \in C'_{\delta_c}$ do
      if $c = c'$ then
        $solved = false$
      end if
    end for
    if $solved = false$ then
      Let $i = pickInsertionTask(\delta_c, pred_c)$
      if $i \neq \emptyset$ then
        Insert $i$ into $\delta_c$
        $changed = true$
      end if
    end if
    if changed=true then
      Update $C$
      Break inner loop
    end if
  end for
until No conflict could be solved
```

5.2.2 Picking tasks

Note that the algorithm uses a function `pickInsertionTask`, which determines which task to insert after the conflict. We determine for each possible task a score (based on the current duty) using another function `getInsertionScore`. Then we insert the task with the highest score.

There are some constraints that determine whether a given task will be considered for insertion:

- The driver needs to be qualified to drive the given task, unless it is a passenger task
- The task may not end more than one hour after the end of the original duty
- Enough transfer time should be inserted, when a transfer to another rolling stock unit is made
- The task cannot be placed inside a break
- When the task has been inserted, a break should still be possible (if necessary)

Then, we judge the task using the following criteria. The score is the sum of a selected subset of values, assigned to these criteria.

- If the task is not yet assigned to a driver, add *valueUnassigned*. We encourage assignment of unassigned tasks, so as to end with as little unassigned tasks at possible.
- If the task starts at the correct location (so no taxi is needed), add *valueNoTaxi*. Using a taxi is costly and takes more time, so we use them as little as possible.
- If the task ends closer to the depot than its predecessor, and the task ends in the second half of the duty, add *valueCloseToHome*. We hope to reduce the use of taxi trips at the end of the duty using this setting.
- If the task ends at the depot, add *valueVisitDepot*. The previous argument also holds here.
- If the task has the same train number as its predecessor, add *valueSameRS*. Human planners often use this rule-of-thumb since it makes the crew schedules similar to the rolling stock schedules (which already have been rescheduled) and reduces the number of transfers to other rolling stock.
- If it has another train number than its predecessor, but the train series is the same, add *valueSameSeries*. Of course this is less preferable, but it might make the duties 'easier', in the sense that they use less different routes. We hope that in this way, it will be easier to make the rest of the duty feasible.

- If the end location of the task is the same as the start of its predecessor, add *valueRetour*. The previous argument also applies here.
- If the transfer time T before the task is less than *shortTransferLimit*, add *valueShortTransfer* times (*shortTransferLimit* $-T$). This prevents the solver from creating large gaps between tasks. More gaps means less real driving hours and thus more unassigned tasks.

Most of these criteria can be checked very fast. We will test different settings for the parameters to see if their effect is as expected, and what gives the best results. Statistics and an analysis can be found in chapter 6.

5.2.3 Filling gaps

As mentioned earlier, inserting new tasks after a conflict may cause large transfer times and an increase in the number of unassigned tasks. Experiments showed, that some duties contained gaps of more than two hours, both between tasks and at the end of the duty. It would be nice if these gaps were filled with other tasks. When this leads to new conflicts, we can use our conflict-solving algorithm. The algorithm for gap filling is shown as Algorithm 5.2. Here G denotes the set of gaps, where δ_g is the duty in which gap $g \in G$ occurs, and $pred_g$ is the last task before gap g. Filling gaps may cause cycling due to break conflicts that occur before the inserted task. To prevent us from cycling, we maintain a list F of forbidden gap-task-combinations that have already been tested without success.

Algorithm 5.2 Filling gaps with extra tasks in `solveConflicts()`

```
1: repeat
2:    $changed = false$ and $F = \emptyset$
3:    Determine $G$
4:    if $G = \emptyset$ then
5:       Stop the algorithm
6:    end if
7:    for all $g \in G$ do
8:       Let $t =$ pickGapTask($g$)
9:       if $t \neq \emptyset$ then
10:         $changed = true$
11:         Insert $t$ into $\delta_g$
12:         Insert $(g, t)$ into $F$
13:         solveConflicts()
14:         Return to outer loop
15:      end if
16:   end for
17: until $changed = false$
```

Selecting a task to insert into the gap is done in `pickGapTask()`, which is quite similar to picking a task for insertion after a conflict. However, now we impose some additional restrictions on the tasks that are considered:

- The duty must not contain the task t
- The task must start in the gap, not after it
- The combination (gap, t) has not yet been tried, so $(gap, t) \notin F$
- If the gap has no predecessor (which is the case after a break), the task must start at the current location

Of course we also need to be able to determine all gaps. We consider two types of gaps: Gaps between tasks and gaps at the end of the duty. For the first type, we check if the gap is not used for a taxi transfer. Also, when the time between two tasks is used for a break, we calculate the earliest time at which the break may be ended. The gap is then considered to start at that time. We could also have chosen to plan the gap before the break, but that would have caused difficulties when the time of rescheduling is during the gap. Finally, we only consider gaps that have a duration of at least *minGapSize* minutes.

Gaps at the end of a duty are only considered when they have a size of at least *minDutyEndGapSize*, which may differ from *minGapSize*. Filling the end of a duty is more risky since it is more difficult to end on time and at the depot, so we may set *minDutyEndGapSize* to be larger than *minGapSize* so that we only consider very large gaps at duty ends. We may also consider reserve duties as gaps; this is controlled by parameter *useReserveGaps*.

The list of gaps is finally sorted by decreasing gap size, so that the largest gap is filled first.

Chapter 6

Computational experiments

6.1 Duty and task selection

In the literature, we have found several possibilities for selecting duties and tasks for rescheduling. We will investigate the following three selection methods:

- *ALL*: Select all tasks and duties in the dataset
- *SIM*: Select duties that are affected, or that are similar to affected duties. The method is described in [11] in more detail, where the duty selection is called a core problem.
- *AFF*: Select the duties that are affected by the disruption, and all of the tasks in these duties

We first use one of these methods to determine the duties that will be rescheduled. Then we optionally add tasks from other duties, which might improve the scheduling process. To select these duties, we also use one of the mechanisms in the list above. Finally, we have the parameter *usePassengerTrick* which indicates if drivers are allowed to deadhead on a train for which they do not have a driving license.

6.2 Datasets

To test our algorithms, we have data for five major disruptions in 2007. For each of them, two datasets are available. The first one contains the real data from the disruption as it actually took place, while the second one is a manually modified version of this disruption. In this second set, the time of the day is changed. Based on this modified disruption time and official NS rescheduling rules, the schedule for the second set is updated. This means that in total, we have ten datasets available.

Code	Disruption	Time	Drivers	Tasks	Locations
Ac_A	Abcoude,	11:07	67	355	32
Ac_B	Amsterdam-Utrecht	16:37	62	437	32
Bl_A	Beilen, Zwolle-Groningen	7:10	15	109	19
Bl_B		16:10	15	95	16
Ht_A	Den Bosch, ri.	08:00	59	439	30
Ht_B	Breda/Eindhoven	15:30	57	461	34
Lls_A	Lelystad,	03:52	25	175	27
Lls_B	Almere-Amsterdam	12:52	24	93	16
Ztm_A	Zoetermeer, Den Haag	07:59	22	150	26
Ztm_B	CS-Gouda	11:29	26	136	21

Table 6.1: Characteristics of the available datasets.

In Table 6.1, the properties of these datasets are summarized. For each dataset, we first show the code and the tracks that are affected by the distruption. The third column gives the time at which rescheduling takes place. The following columns show the number of drivers involved, the number of tasks in their duties after the rescheduling moment and the number of relief locations in these tasks. We define a duty to be affected when one or more of the tasks in the duty are cancelled or replaced by another task.

6.3 Results

6.3.1 RCSPP-based approach

Duty and task selection

We have used several settings to test each of the instances listed in Table 6.1. To each set of selected duties, we have added all 84 reserve duties. These are only considered after the normal duties have been rescheduled. We will compare the performance of the different settings based on the computational time needed, the number of duties for which no feasible completion could be found and the number of unassigned tasks. The results for case Lls_B are given in Table 6.2. A summary of the results for all datasets can be found in Table A.1 the appendix.

The first three columns define the combination of parameters. The following three columns denote the number of duties that was included for rescheduling (reserve duties not included), the number of tasks that were included as driving or passenger tasks and the number of tasks that had to be assigned. The last columns show the computation time, the total number of duties for which no feasible completion could be found and the number of tasks that were selected for rescheduling, but could not be re-assigned to any duty. Since not all original duties satisfy the constraints (for

	Duties	Tasks	P.Trick	Duties	Tasks total	Tasks must	Time (s)	Infeasible	Unassigned
1	AFF	ALL	Yes	24	5885	69	88	0	4
2	AFF	ALL	No	24	5885	69	102	0	4
3	AFF	AFF	Yes	24	102	69	3	2	11
4	AFF	AFF	No	24	102	69	1	2	11
5	AFF	SIM	Yes	24	887	69	1	0	5
6	AFF	SIM	No	24	887	69	3	0	5
7	SIM	ALL	Yes	78	6133	402	89	1	11
8	SIM	ALL	No	78	6133	402	126	1	11
9	SIM	SIM	Yes	78	753	402	6	1	13
10	SIM	SIM	No	78	753	402	6	1	13

Table 6.2: Influence of duty and task selection on solution quality and computation time of the RCSPP-based approach for dataset Lls_B

example, a duty might only contain a break of 28 minutes), and some of these duties have already started, it is not always possible to find a feasible completion.

We see a few interesting things here. First of all, there are huge differences in computation time between the different settings. Including only the tasks of affected duties gives results in a few seconds, while including more duties and tasks leads to times between 1 and 3 minutes for this instance and up to 30 minutes for larger cases.

Furthermore, including more duties leads to more unassigned tasks, while we would have liked to see less. We hoped that when we included more duties and thus more tasks, more options became available to create feasible duty completions which covered more tasks. But including more tasks also makes the problem more complex and we see that a greedy algorithm is less capable of handling a more complex case. Including more passenger tasks leads to better solutions with fewer unassigned tasks (compare row 3 with row 5 and row row 9 with row 7). Here the number of tasks also increases, but since these tasks do not have to be covered, the problem is equally solvable by the RCSPP-based algorithm. Finally, there is not much difference in solution quality between the cases where we used *usePassengerTrick* parameter and the cases where we did not, although the computation times do differ, especially when we include a lot of passenger tasks in larger instances.

Objective value

In the second experiment we test the influence of the objective function on the solution quality. This objective function can be modified by changing the values that are assigned to the nodes in the graph. We measure the number of unassigned tasks but we also look at the percentage of driving tasks that is done by the driver who was originally assigned to that task and finally we check the distribution of the task types (driving, passenger-on-train and passenger). We do this

	Own	Other	Pot	Pass	Unassigned	% Own	% Driving
1	50	10	-1	-2	5	54.7	64.6
2	50	10	-2	-1	5	54.7	62.1
3	20	10	-1	-2	5	34.4	68.1
4	10	10	-1	-2	4	26.2	69.9
5	50	10	0	-1	5	57.8	62.7
6	50	10	0	0	5	57.8	62.7
7	50	10	1	1	5	65.6	18.1

Table 6.3: Influence of node values on solution quality for the RCSPP-based approach for case Lls_B

for several settings of the node values. The settings as well as the results are shown in Table 6.3, again for case Lls_B. Summarized results for all test instances can be found in Table A.2 in the appendix.

We see that we can easily influence how many tasks are assigned to another driver and how many to the planned driver, by changing the ratio of the `valueDriveOwn` to the `valueDriveOther` parameters. We also see that by balancing this ratio, we can decrease then number of unassigned tasks. It seems like there is a trade-off between less unassigned tasks and a lower percentage of tasks that are driven by their planned driver.

Furthermore, by making deadheading more attractive, we slightly decrease the percentage of driving tasks without really influencing the number of unassigned tasks. It only increases the number of tasks per duty. However, more tasks per duty makes the schedule less robust since the number of transfers also increases. Only when we give positive weight to extra passenger tasks, we see that the important tasks do not get enough weight and more tasks become unassigned (case 7).

Duties ending later

Finally, we have tested the influence of parameters `costEndLater` and `costQuarterLater` on the solution quality and on the end times of the generated duty completions. As explained earlier, the first parameter gives the fixed cost of a duty ending later, while the second one gives a variable cost per quarter of an hour. The results for nine different combinations of values for these parameters are given in Table 6.4, again for instance Lls_B.

We see that the number of unassigned tasks does not differ much, when we increase the values of one or both parameters. The average number of minutes each duty ends earlier or later (where the minutes are first averaged over all duties with tasks in a single instance and then over all instances), is somewhat influenced. We see that the number of duties ending later can be cut down a bit. The effects can better be seen in Table A.3 in the appendix, where the summarized results

	Fixed	Quarter	Unassigned	# Earlier	# Later	# Equal	Avg. Earlier	Avg. Later
1	0	3	5	15	9	7	118	28
2	5	3	5	15	8	8	113	22
3	25	3	6	19	3	10	109	11
4	0	10	5	17	4	10	110	13
5	5	10	5	17	4	10	110	13
6	25	10	6	19	3	10	108	11
7	0	50	5	18	4	10	110	9
8	5	50	5	18	4	10	110	9
9	25	50	6	19	3	10	108	11

Table 6.4: Influence of *costEndLater* and *costQuarterLater* on the end times of the duties for the RCSP-based approach and dataset Lls_B

for all test instances are shown. We then also see, that influencing the duty end times comes at the cost of more unassigned tasks.

6.3.2 Conflict-based approach

To compare the influence of the large set of parameters for the conflict-based approach, we have chosen to use one method for selecting duties and tasks: All affected duties are selected and then the tasks from a selection of similar duties are added as passenger-only tasks. We initially set all parameters to 0 and then incrementally add functionality to see the seperate effect of several subsets of parameters. Including reserve duties gave very bad results in our first tests. It might be useful to consider adding multiple tasks to them at once, for example by using the RCSPP approach. But adding individual tasks to empty (reserve) duties almost always leads to conflicts, so we left this out of consideration.

Value of unassigned tasks and no-taxi transfers

We start by setting two fundamental parameters: *valueUnassigned* gives the value that is assigned to a candidate task that has not been assigned yet to any driver, while *valueTaxi* gives the value of a taxi that does not require a taxi trip when added to the duty. The results for case Ac_A are given in Table 6.5. Results for all cases can be found in Table A.4 in the appendix.

We see that both parameters have an effect on the number of tasks that is left unassigned and the number of duties that end with a taxi trip to the depot. The larger *valueUnassigned*, the lower the number of unassigned tasks. The influence of *valueTaxi* is most visible when we compare cases (1) with (2): Increasing the value of a transfer at the same station decreases the number of unassigned tasks. No difference is visible when we compare (4), (5) and (7) in this table, neither in the table in the appendix. The difference mentioned can be explained as follows. When taxi

	valueTaxi	valueUnassigned	Time (s)	Unassigned tasks	Duties with taxi end
1	0	0	14.85	180	52
2	100	0	13.57	110	29
3	0	100	15.58	54	48
4	100	100	13.16	62	33
5	150	100	13.48	62	33
6	100	150	13.29	62	33
7	200	100	15.33	62	33
8	100	200	14.16	62	33

Table 6.5: Influence of *valueUnassigned* and *valueTaxi* on the duty ends, task assignment and computational

	valueShortTransfer	shortTransferLimit	Time (s)	Unassigned	Taxi ends
1	3	15	13.84	62	33
2	6	15	13.21	62	33
3	3	20	13.26	62	33
4	5	20	14.41	62	33
5	3	30	13.38	62	33
6	5	30	15.84	62	33
7	2	45	13.37	62	33
8	4	45	12.69	67	34
9	6	30	12.44	63	31

Table 6.6: Influence of *valueShortTransfer* and *shortTransferLimit* on the duty ends, task assignment and computational time for case Ac_A.

trips are used, a driver is moved to another station than the one where he was originally located. Taxi trips are usually used when no similar train trip is available and it is therefore possible that, when the duty ends, no trip back exists since that trip would use the same track.

Rewarding short transfers

We now turn to a set of parameters that can be used to reward short transfer times, in the hope that they lead to tighter crew schedules with less unassigned tasks. This can be achieved by setting *valueShortTransfer* to a positive value and by using *shortTransferLimit* to indicate what should actually be considered as a short transfer. We have set *valueTaxi* to 100 and *valueUnassigned* to 150 in all test runs. The combinations that we have used, along with the results for case Ac_A, can be found in Table 6.6. Summarized outcomes for all cases can be found in Table A.5 in the appendix.

We see that the use of these two parameters leads to equal or worse results than the situation in the previous section, where we did not reward short transfers. The results can be seen more clearly when comparing Table A.4 and Table A.5 in the appendix. This is not really what we expected to see and we do not have an explanation for these results. We will investigate the combined effect

	visitDepot	sameRS	sameSeries	closerToHome	Time (s)	Unassigned	Taxi ends
1	10	0	0	0	13.72	62	33
2	20	0	0	0	12.91	60	32
3	30	0	0	0	12.24	60	31
4	20	0	0	2	9.49	59	28
5	20	0	0	5	9.27	72	25
6	20	0	0	10	6.36	100	16
7	20	2	1	2	9.33	62	27
8	20	4	2	2	9.39	61	27
9	20	10	4	2	8.97	62	26
10	20	20	6	2	8.82	61	27

Table 6.7: Influence of *valueVisitDepot*, *valueSameRS*, *valueSameSeries* and *valueCloserToHome* on the duty ends, task assignment and computational time for case Ac_A.

of these parameters with four others in the next section.

Rewarding both trips close to the depot and multiple trips on the same train

The next step is rewarding trips that end at the depot, or end closer to the depot than their starting point. This can be achieved by using *visitDepot* and *valueCloserToHome*, respectively. We also reward combinations of consecutive trips on the same rolling stock or on trains on the same route, as explained earlier. Settings and results are given in Table 6.7. Summarized results over all datasets are reported in Table A.6 in the appendix.

We first observe that we can actually limit the number of taxi trips to depots by rewarding trips that end there (compare 1-3). The effect of rewarding trips that bring an employee closer to his depot are even stronger (compare 2, 4-6) although this leads to an increased number of unassigned tasks. Of course, when we prefer trips close to depots of affected duties, the few trips that are further from their depot will not be covered. When we also prefer trips on the same rolling stock, we see that the results only get worse in terms of unassigned tasks and taxi trips at duty ends. However, such transfers might still be prefered since robustness of schedules is also an important factor at NS. We will therefore use the settings of (7) in our next test runs.

Filling gaps in duties

We now turn to our second major strategy, namely filling gaps in the duties. We will first ignore gaps at the end of each duty and only consider gaps between two tasks. We investigate what number of iterations is necessary and what minimum length a gap should have. The results for case Ac_A are reported in Table 6.8. Summarized results can be found in Table A.7 in the appendix.

We see that the more iterations we use for gap filling, the worse the solution gets. Also note

	maxGapIterations	minGapSize	Time (s)	Unassigned	Taxi ends
1	25	20	18.06	70	29
2	100	20	50.50	95	31
3	500	20	100.96	95	32
4	25	40	17.88	69	28
5	100	40	27.23	73	31
6	500	40	26.69	73	31
7	25	60	17.37	69	28
8	100	60	17.25	69	28
9	500	60	17.13	69	28

Table 6.8: Influence of *maxGapIterations* and *minGapSize* on the duty ends, task assignment and computational time for case Ac_A

	maxGapIterations	minDutyEndGapSize	Time (s)	Unassigned	Taxi ends
1	25	60	15.98	63	32
2	50	60	22.35	69	32
3	100	60	23.87	69	32
4	150	60	22.26	69	32
5	200	60	22.69	69	32
6	25	90	16.83	64	32
7	50	90	20.64	69	30
8	100	90	21.04	69	30
9	150	90	21.80	69	30
10	200	90	22.36	69	30

Table 6.9: Influence of *maxGapIterations* and *minDutyEndGapSize* on the duty ends, task assignment and computational time for case Ac_A.

that raising the minimum gap size (and thus reducing the number of gaps that are filled), the better the results. When we compare the results in this run with result (7) in the previous run, we can conclude that gap filling is not really a smart idea.

Filling gaps after duties

Finally, we look what happens when we also consider the period between a new duty's end and the original end as a gap and try to fill it as well. We set *minGapSize* to 60 and try different combinations of *maxGapIterations* and *minGapSize*, as can be see in Table 6.9. Summarized results are presented in Table A.8 in the appendix.

When considering the number of unassigned tasks, we see that gap filling at the end of a duty strongly improves the solutions. However, this comes at the cost of an increased number of taxi trips at the end of the duties (compare to results 7-9 in the previous run). This can also be seen when one compares Table A.7 with Table A.8 in the appendix. Again, increasing the number of iterations makes the solutions worse.

6.3.3 Comparison

As we have already shown, the two greedy algorithms presented in this paper are able to generate solutions to the crew rescheduling problem very fast, although the solutions are not always satisfactory. In this section, we will compare the two methods with the method described in [11], both with and without local neighbourhood search.

Settings and data used

To compare the four methods, we have selected four test case: Ac_A, Bl_A, Ztm_B and Ht_B.

For the column generation algorithm, we have used the penalty values and settings described in [11]. We have created a new set of reserve duties by random selection, where each reserve duty had a probability of 0.5 to be included in the set. This resulted in a random selection of 35 reserve duties. In the local neighbourhood algorithm, we have used $r = 3, s = 3$ since the paper reported these values to give good results.

For our two greedy methods, we have used the same set of reserve duties. We have used the most promising set of parameter settings for each method. For both algorithms, the tasks of all affected duties were considered. Furthermore additional passenger tasks were added by using the SIM-method described earlier. For the greedy SPPRC-algorithm, we used *valueDriveOwn* = 50, *valueDriveOther* = 10, *valuePot* = −1, *valuePassenger* = −2, *costEndLater* = 0 and *costQuarterLater* = 3. For the conflict algorithm, we've used *maxGapIterations* = 25, *minDutyEndGapSize* = 60, *minGapSize* = 60, *valueVisitDepot* = 20, *valueSameRS* = 2, *valueSameSeries* = 1, *valueCloserToHome* = 2, *valueNoTaxi* = 100, *valueUnassigned* = 150, *valueShortTransfer* = 3 and *shortTransferLimit* = 30.

Results

In Table 6.10 through 6.13, we present the results for each of the four cases. Here ColGen denotes the normal column generation approach described in [11], ColGenDynamic adds local neighbourhood search, RCSPP stands for the greedy RCSPP approach from chapter 4 and Conflict for the conflict-based approach from chapter 5.

We see that overall, the methods with column generation perform best and the conflict approach scores worst at all criteria. More specific, both greedy methods are usually faster than the column generation but they are not always able to solve the problem without leaving some tasks unassigned. The column generation algorithm was able to assign all tasks to drivers, except for (just) two tasks in Ht_B case.

Method	Unassigned tasks	Time (s)	New taxi tasks
ColGen	0	224	1
ColGenDyn	0	271	1
RCSPP	33	2	11
Conflict	63	15	32

Table 6.10: Comparison for case Ac_A

Method	Unassigned tasks	Time (s)	New taxi tasks
ColGen	0	14	2
ColGenDyn	0	9	2
RCSPP	5	0	1
Conflict	25	1	4

Table 6.11: Comparison for case Bl_A

Method	Unassigned tasks	Time (s)	New taxi tasks
ColGen	2	156	4
ColGenDyn	2	307	4
RCSPP	17	3	6
Conflict	81	5	36

Table 6.12: Comparison for case Ht_B

Method	Unassigned tasks	Time (s)	New taxi tasks
ColGen	0	54	0
ColGenDyn	0	53	0
RCSPP	14	1	2
Conflict	22	2	9

Table 6.13: Comparison for case Ztm_B

Chapter 7

Conclusions

In this paper we have discussed the crew rescheduling problem at Netherlands Railways in detail. We have described the problem and shown what methods do already exist in this and other rescheduling areas. We have also developed two new, greedy approaches, based on rescheduling practice as it is currently done manually by NS dispatchers.

Through experiments, we have first fine-tuned both algorithms. We have demonstrated the influence of the parameters on the various quality aspects of the produced solutions. The RCSPP-based approach, when applied to small- or medium-size cases, leaves only a few tasks unassigned and gives results within seconds. The conflict-based approach, which most closely resembles the methods used in practice, gives worse results and takes more time. In practice it is possible to ignore several constraints (usually after discussing the issue with the drivers) to improve the solution, which might give better results for conflict-solving than the ones observed here.

Afterwards, both methods have been compared with the column generation approach as described in [11]. The greedy algorithms clearly outperform the column generation algorithm in speed. However, column generation is able to solve most problems, leaving only some or no tasks to be canceled. Running times for the column generation have been shown to lie between one and five minutes, which is still acceptable in practice. Since canceling additional tasks due to crew schedules is very complicated and undesired, we can state that this algorithm is the best one currently available.

We therefore advice to put efforts in testing and extending the column generation algorithm. One can think of extra penalties and cost components, such as a flexible cost for changing duty end times, depending on the real end time difference. Redesigning the column generation, optimizing for both flexibility and speed, could also improve usability and running times. This will be done in the near future and we're looking forward to the results.

Appendix A

Additional experimental results

A.1 RCSPP-based approach

	Duties	Tasks	P.Trick	Duties	Tasks total	Tasks must	Time (s)	Infeasible	Unassigned
1	AFF	ALL	Yes	370	68214	2212	1988	4	59
2	AFF	ALL	No	370	68214	2212	2837	2	63
3	AFF	AFF	Yes	370	2623	2212	44	7	101
4	AFF	AFF	No	370	2623	2212	30	5	99
5	AFF	SIM	Yes	370	6499	2212	23	5	72
6	AFF	SIM	No	370	6499	2212	55	3	75
7	SIM	ALL	Yes	704	68599	4592	1377	5	107
8	SIM	ALL	No	704	68599	4592	2047	3	109
9	SIM	SIM	Yes	704	5691	4592	79	6	126
10	SIM	SIM	No	704	5691	4592	76	4	125

Table A.1: Influence of duty and task selection on solution quality and computation time of the RCSPP-based approach for all datasets together. All numbers are totals over the ten datasets.

	Own	Other	Pot	Pass	Unassigned	% Own	% Driving
1	50	10	-1	-2	72	62.9	76.0
2	50	10	-2	-1	70	59.6	71.7
3	20	10	-1	-2	69	46.7	76.8
4	10	10	-1	-2	75	32.2	76.8
5	50	10	0	-1	68	66.2	74.3
6	50	10	0	0	71	64.2	73.0
7	50	10	1	1	82	85.2	42.5

Table A.2: Influence of node values on solution quality for the RCSPP-based approach. Percentages are averaged over all test instances, the number of unassigned tasks is summed over the instances (where a total of 2212 tasks had to be assigned).

	Fixed	Quarter	Unassigned	# Earlier	# Later	# Equal	Avg. Earlier	Avg. Later
1	0	3	76	139	105	248	117	33
2	5	3	79	150	86	255	110	33
3	25	3	98	203	30	268	97	28
4	0	10	88	176	60	263	102	20
5	5	10	88	177	59	263	101	19
6	25	10	98	206	27	268	97	26
7	0	50	92	183	56	263	100	17
8	5	50	90	192	49	263	99	17
9	25	50	101	213	24	265	95	23

Table A.3: Influence of *costEndLater* and *costQuarterLater* on the end times of the duties. Numbers of unassigned tasks and duties ending earlier/later are summed over all instances. Average minutes earlier and later are averaged over all duties per dataset, then averaged over all datasets.

A.2 Conflict-based approach

	valueTaxi	valueUnassigned	Time (s)	Unassigned	Taxi ends
1	0	0	49.57	1052	197
2	100	0	43.39	724	130
3	0	100	45.98	458	185
4	100	100	38.03	467	129
5	150	100	38.51	467	129
6	100	150	38.56	453	131
7	200	100	43.69	467	129
8	100	200	39.68	453	131

Table A.4: Influence of *valueUnassigned* and *valueTaxi* on the duty ends, task assignment and computational time. All numbers are totals over the ten datasets.

	valueShortTransfer	shortTransferLimit	Time (s)	Unassigned	Taxi ends
1	3	15	43.78	453	131
2	6	15	38.42	453	131
3	3	20	38.37	453	131
4	5	20	39.66	453	131
5	3	30	40.51	453	131
6	5	30	47.26	470	132
7	2	45	38.06	470	131
8	4	45	38.18	495	127
9	6	30	36.88	504	125

Table A.5: Influence of *valueShortTransfer* and *shortTransferLimit* on the duty ends, task assignment and computational time. Note that all results are totals of the ten datasets.

	visitDepot	sameRS	sameSeries	closerToHome	Time (s)	Unassigned	Taxi ends
1	10	0	0	0	37.7	456	121
2	20	0	0	0	36.16	454	118
3	30	0	0	0	34.8	459	114
4	20	0	0	2	26.57	460	92
5	20	0	0	5	23.34	509	75
6	20	0	0	10	17.69	581	57
7	20	2	1	2	25.62	467	88
8	20	4	2	2	27.6	473	90
9	20	10	4	2	25.42	473	89
10	20	20	6	2	27.3	479	92

Table A.6: Influence of `valueVisitDepot`, `valueSameRS`, `valueSameSeries` and `valueCloserToHome` on the duty ends, task assignment and computational time for all datasets together. All numbers are again totals of the ten datasets.

	maxGapIterations	minGapSize	Time (s)	Unassigned	Taxi ends
1	25	20	54.86	516	109
2	100	20	164.51	628	135
3	500	20	466.54	676	147
4	25	40	53.37	509	107
5	100	40	105.27	532	122
6	500	40	102.74	535	122
7	25	60	48.2	480	102
8	100	60	132.05	498	104
9	500	60	62.12	498	104

Table A.7: Influence of `maxGapIterations` and `minGapSize` on the duty ends, task assignment and computational time for all cases together. All results are totals for the ten datasets together.

	maxGapIterations	minDutyEndGapSize	Time (s)	Unassigned	Taxi ends
1	25	60	47.51	425	123
2	50	60	73.77	430	127
3	100	60	93.33	436	127
4	150	60	88.81	436	127
5	200	60	91.45	436	127
6	25	90	47.76	429	120
7	50	90	71.63	449	119
8	100	90	82.15	454	118
9	150	90	83.73	454	118
10	200	90	134.28	454	118

Table A.8: Influence of `maxGapIterations` and `minDutyEndGapSize` on the duty ends, task assignment and computational time. Again the results are totals of the ten datasets.

Bibliography

[1] G. Desaulniers, J. Desrosiers, and M. Solomon. Resource-constrained shortest path problems. In *Column Generation*, GERAD 25th Anniversary Series, pages 33–65. Springer, 2005.

[2] M. Desrochers. *La fabrication d'horaires de travail pour les conducteurs d'autobus par une methode de generation de colonnes.* PhD thesis, Universite de Montreal, Centre de recherche sur les transport, 1986.

[3] J.H.A. Gottgens, M.A. Koppers, E.A.J. Vendrik, and F.P. Westgeest. Crew rescheduling during the day of operation at NS. Logistic case studies 2, Erasmus Universiteit Rotterdam, 2008.

[4] D. Huisman. A column generation approach to solve the crew re-scheduling problem. *European Journal of Operational Research*, 180(1):163–173, 2007.

[5] D. Huisman, R. Jans, M. Peeters, and A.P.M. Wagelmans. Combining column generation and lagrangian relaxation. In *Column generation*, GERAD 25th Anniversary Series, pages 247–270. Springer, 2005.

[6] J. Jespersen-Groth, D. Pothoff, J. Clausen, D. Huisman, L. Kroon, G. Marti, and M. N. Nielsen. Disruption management in passenger railway transportation. Technical Report EI2007-05, Econometric Institute, Erasmus Universiteit Rotterdam, 2007.

[7] L. Kroon and M. Fischetti. Scheduling train drivers and guards: the dutch noord-oost case. Technical report, Erasmus Universiteit Rotterdam, Hawaii International Conference on System Sciences, 2000.

[8] L. Kroon, D. Huisman, E. Abbink, P. Fioole, M. Fischetti, G. Marti, A. Schrijver, A. Steenbeek, and R. Ybema. The new dutch timetable: The OR revolution. *Interfaces*, 39(1):6–17, 2009.

[9] E.M. Morgado and J.P. Martins. CREWS_NS: Scheduling train crews in the netherlands. *AI Magazine*, 19(1):25–38, 1998.

[10] R. Nissen and K. Haase. Duty-period-based network model for crew rescheduling in european airlines. *Journal of Scheduling*, 9(3):255–278, 2006.

[11] D. Potthoff, D. Huisman, and G. Desaulniers. Column generation with dynamic duty selection for railway crew rescheduling. Technical Report EI2008-28, Econometric Institute, Erasmus Universiteit Rotterdam, 2008.