Lower Bounds and Quality Guarantees for Online-Dispatching

Diplomarbeit

von
Stephan Westphal

April 2004

Technische Universität Berlin
Fachbereich II: Mathematik und Naturwissenschaften
Institut für Mathematik
Studiengang Wirtschaftsmathematik

Erstgutachter: Prof. Dr. M. Grötschel
Zweitgutachter: Prof. Dr. S. O. Krumke

Acknowledgements

I would like to express my gratitude to everyone who contributed to this thesis.

First of all, I would like to thank Prof. Dr. Sven Oliver Krumke for the cooperation and support. This thesis would not have been possible without him and the mental and technical infrastructure of the Konrad-Zuse-Zentrum für Informationstechnik Berlin.

On the personal side, I am indebted to Katrin G., my family and all the friends and companions who supported me throughout this time.

Berlin, den 12.April 2004

Stephan Westphal

Contents

Chapter 1

Introduction

Each year thousands of motorists get stuck or break down on the road and have to call out for assistance. For this reason the German Automobile Club (ADAC) maintains a fleet of service vehicles, so called "yellow angels", in order to help these people.

Since December 2000 the Konrad-Zuse-Zentrum Berlin (ZIB) has been involved in a project concerning the automization of the online dispatching system of the ADAC. It is the project's goal to develop an algorithm that can be used to guarantee a good quality of service under low operational costs. In the following sections we introduce the ADAC, a precise definition of the problem we have to solve and the main difficulties we are faced by the search of good solutions.

1.1 Some facts about the Yellow Angels

Since its foundation in 1954 the fleet of service vehicles grew from 60 motorcycles in the early days to 1,700 ADAC-owned vehicles and 5,000 extra vehicles run by partners in 2003. Over the last 50 years the service units drove more than 1,400,000,000 kilometers in order to provide assistance in more than 50,000,000 instances [20].

With the permanently increasing volume of traffic the number of breakdowns grew the same way, and so, between 1993 and 2003 there has been a 33.45 % increase in the number of help requests (see Figure 1.1).

The most common causes are problems of the general electrical equipment of the vehicles, as well as engine and ignition damage. In the year 2003 the yellow angels worked with their partners on 3,700,000 assignments and helped with 180,000 traffic accidents. This results in an average of 10,000 assignments per

day. But since cars suffer from extreme weather conditions like very cold winters and very hot summers the load does not arise evenly. Thus, they provided over 660,000 jump starts in the winter 2003. As an example for an extreme day, we should mention the 9th January 2003, on which the yellow angels served 20,549 requests.

The yellow angels have by now a success rate of 83 % which means that 83 % of the cars can be fixed in a way that they can continue their drives.

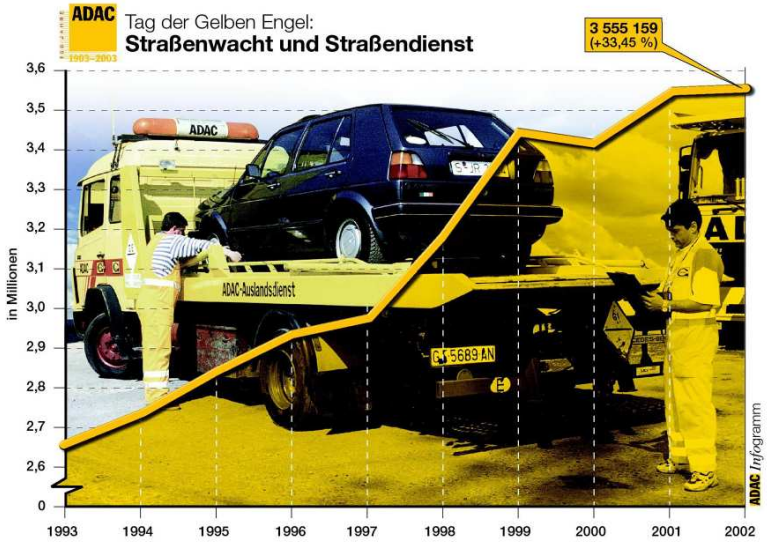

Figure 1.1: Growth of the number of services provided by the ADAC between 1993 and 2002 [1]

The assignments both of the yellow angel as well as of their partners are coordinated in five help centers by human operators (*dispatchers*). People who get stuck on the road and need the help of the yellow angels call these help centers.

1.2 The problem of dispatching

It is the job of the dispatchers to assign every incoming help *request* to a service *unit*. By doing so, the dispatchers have to create tours for the units.

Whenever it is not possible to find a good dispatch, the dispatchers can revert to contractors who can provide assistance to the requests instead of the yellow angels. Contractors are firms, like e.g. garages, which have service cars on their

own and serve the ADAC's customers at a lump sum per service done. Usually, serving a request by a contractor is more expensive than serving it by a yellow angel, but in some cases it is reasonable to do so, especially if there are many requests to be served and the customers are not wanted to wait too long.

We call the problem, which has to be solved by the dispatchers, the vehicle dispatching problem (VDP) and we will use the notation introduced in [15]. On page 80 there is an overview on the most important parameters introduced in the following. An Instance of VDP consists of a set of units U, a set of contractors V and a set of events E. Each unit u has a current position o_u, a home position h_u where u is heading for at the end of its shift, a logon time t_u^{start} which marks the beginning of its shift and a shift end time t_u^{end} which can be exceeded by at most t_u^{max-ot} minutes overtime. Moreover, every unit u has a set of capabilities F_u expressing the abilities of serving the different kinds of damages like jump starting a vehicle or towing a car to a garage.

Like the service units each contractor v has a set of capabilities F_v. Each event e has a position x_e, a release time θ_e^r which marks the time when the motorist has called for assistance, a deadline θ_e^d when the service should be completed at the latest and a set of required capabilities F_e. Moreover, it takes a certain amount of time δ_e to serve a request. For example the ADAC assumes 15 minutes for problems with the muffler, 20 minutes for the gear shift and 20 minutes if the brake does not work properly.

In the following we denote the driving time of unit u from event e to event f as δ_u^{ef} and $\delta_u^{o_u e}$ resp. $\delta_u^{ed_u}$ shall be the driving times of unit u from its current position to event e resp. from event e to its home position h_u.

A feasible solution of VDP is an assignment of events to units and contractors capable of serving them, as well as a tour for each unit such that all events are assigned. Moreover, it is mandatory that the service of events does not start before their release times, and all tours for all units start at their current positions not before their logon times and at their home positions. Such a solution is called a *dispatch*.

The best and therefore desired solution of VDP is the solution that has low operational costs and maintains a good quality of service.

Until quite recently this planning process was completely done by human dispatchers who used their computers just for assistance purposes such as locating the units but not for the optimization. For this reason, the quality of these solutions was highly dependent on the experience and ability of the dispatchers. Furthermore, it was not possible to measure the quality of a dispatch and no quality guarantees could be given apart from some not very meaningful performance figures like the number of requests served per unit and hour.

Hence, the overall goal of the project is to develop an automatic online dispatching system, which was called ZIBDIP, that guarantees low operational costs and short waiting times for the customer.

We recall the two goals of vehicle dispatching in order to get a more precise definition of an optimal dispatch:

1. Minimization of **operational costs** which arise by the assignment of units and contractors. The different cost elements are the following: The usage of a unit costs c_u^{drv} per time unit for driving, c_u^{svc} for service and c_u^{ot} for overtime. The costs of the contractors are specified by a value for costs per service c_v^{svc}.

2. A **good quality of service** is directly dependent on the time the customer has to wait for assistance. Therefore, the maximum and average waiting time has to be our measure for the quality of service achieved by a dispatch. So far the average waiting time is 40 minutes ([20]).

These two objectives are in conflict with each other as a decrease of the operational costs usually causes an increase of waiting times. Since the optimization with respect to objective 1 usually results in a different solution than the optimization with respect to objective 2, both objectives have to be merged in order to make every two different solutions comparable and thus the definition of an optimum becomes possible.

For this reason extra costs related to serving an event e at time t which might be "too late" are specified by the value of a lateness-function $\text{LateCost}(e,t)$ (see Figure 1.2):

$$\begin{aligned}\text{LateTime}(e,t) &:= \max((t-\theta_e^d),0)\\ \text{LateCost}(e,t) &:= 0.1\cdot\text{LateTime}(e,t)\cdot(1+\text{LateTime}(e,t)/2.24)\end{aligned}$$

This lateness penalty is a quadratic function of the time for the reason that it should be more expensive to let a customer wait who is already waiting for a longer time than another customer who did not wait as long. The numeric values used above are the result of discussions with the ADAC.

Finding a good dispatch is difficult for the two following reasons:

1. The complexity

2. The lack of knowledge about the future

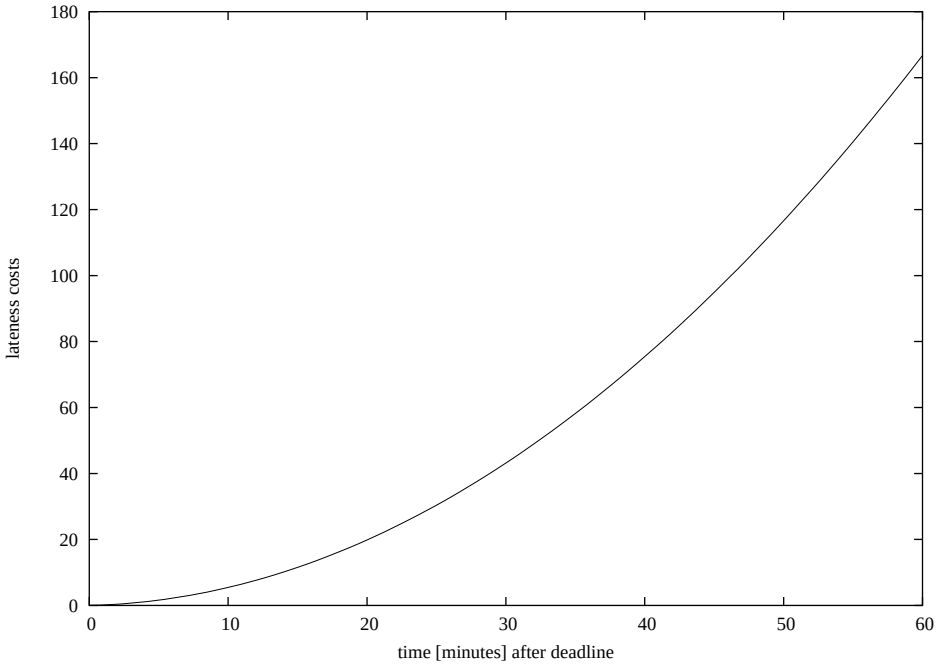

Figure 1.2: The lateness function

In order to illustrate the **complexity** of our problem we make a little calculation. Even on a "normal" day it is not unusual that a dispatcher has to take care of about 100 units and 200 requests at the same time. Assuming an average service time of 15 minutes and a maximum waiting time of 40 minutes (which is desired by the ADAC) the average length by means of served vehicles should not be longer than 4. But even if we only consider tours of length 4 there are $(200 \cdot 199 \cdot 198 \cdot 197=)1{,}552{,}438{,}800$ tours per unit. Having 100 units we have to choose an optimal dispatch from approximately 155 billion tours. It is quite unlikely that a human dispatcher can handle such a huge amount of possibilities.

Even if it was possible to handle all these tours and an optimal dispatch had been found, because of the **lack of knowledge about the future** this dispatch could become suboptimal after the release of an unsuitable request.

Example 1.1 For reasons of simplicity we choose all deadlines properly so that no lateness and overtime costs arise. Consider the following situation: There is only one unit u and one request r_1, so the vehicle is sent to r_1 (Figure 1.3 (i)). While serving r_1 another request r_2 comes up and thus after finishing r_1 u is heading for r_2 in order to provide assistance there (Figure 1.3 (ii)). Since the unit's shift is nearly over, u drives home (Figure 1.3 (iii)). If the dispatcher had known in advance that the driver of the car representing r_2 called, he would have

sent him to r_2 first and then to r_1 (Figure 1.3 (iv)) as this tour is shorter and thus cheaper.

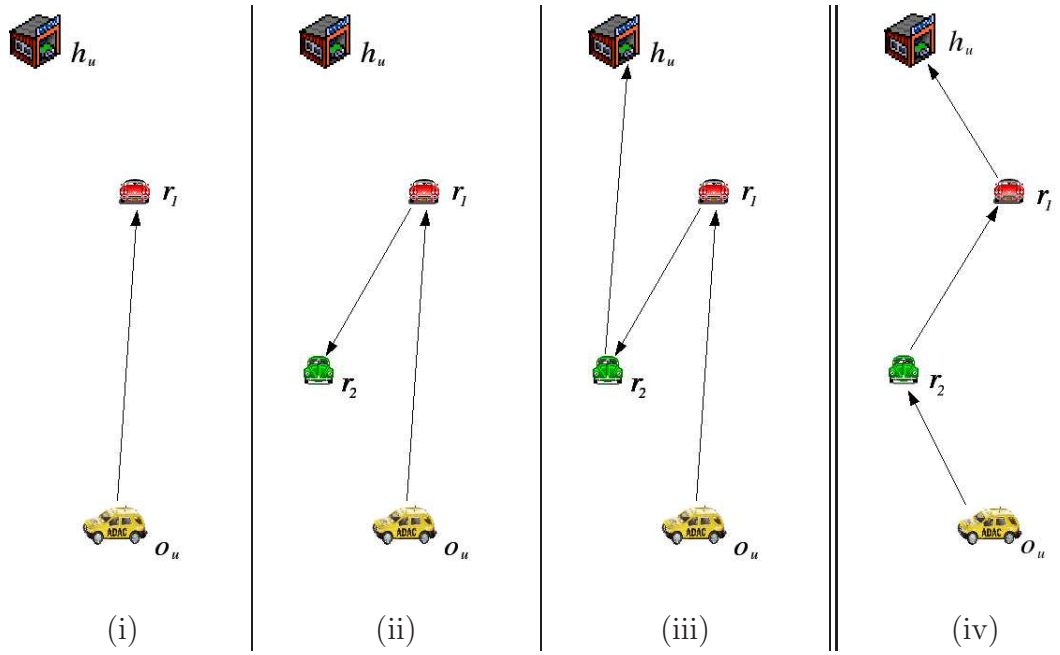

Figure 1.3: Example for problems in online dispatching

Problems of that kind where one has to make decisions before even knowing all the relevant data are called *online-problems*. If all the data was known in advance, we would call the same problem an *offline-problem*. As there are several different strategies to handle online-problems, we present according to [10] some well-established strategies for online-problems in the next section.

1.3 General principles for the design of online algorithms

1. **Greedy**: Every new request is assigned to the currently next best unit. This strategy, called the NEAREST NEIGHBOR policy, is an implementation of the GREEDY-algorithm. The closeness of a request to a unit can be calculated from different values. In our case it would make sense to consider the distance between the vehicle and the request, the distance between the request and the vehicle's home position, the lateness costs, and the unit specific service costs incurred thereby. However this closeness function is chosen, a general problem of every

greedy algorithm is that it always searches for nothing but a local short-term optimum. Although there are plenty of problems which can be solved by greedy algorithms to an optimum (e.g. the MINIMUM SPANNING TREE PROBLEM), our problem cannot be solved this way.

2.**Replan**: The usage of the REPLAN-strategy requires an algorithm for the corresponding offline problem. REPLAN works in a way that we compute the optimal solution for the currently known requests. When something important happens, like the release of a new request or the break down of a yellow angel, we make a new plan which can be totally different from the previous one. Therefore, at every point in time, we have got a plan which is optimal with respect to the known requests and the available service units.

3. **Ignore**: Like REPLAN, IGNORE assumes that we have an algorithm which can solve the corresponding offline problem. But in contrast to REPLAN the once computed solution is not dropped when an inappropriate request is released. The current plan is fulfilled until every request has been served. All requests which are released in the meantime are collected in a buffer and a new plan is computed which optimizes the not yet served requests. Since we have several units, every dispatch is a plan which contains several plans called the tours usually ending at different points in time. So, waiting for the last request to be served means letting all the other units not involved in this service wait until it is done and a new plan can be made. This does not make sense and so the well-known definition of the IGNORE-strategy like mentioned above is not applicable this way. In consequence, we should replan whenever a unit has completed its tour, changing given and not yet fulfilled tours only by appending requests.

In the ADAC project it was decided to solve the VDP by REPLAN, since this strategy had provided promising results in the first evaluations on real-world-data for the ADAC-problem.

Therefore, after each request's release, we have to solve our problem again. For this reason we need an offline-algorithm that computes the best or at least a provably good dispatch in a reasonable time. Such an algorithm is presented in the following chapter.

1.4 Lower Bounds and Quality Guarantees

In the following we recall some basic procedures of benchmarking online algorithms and try to find some quality strategies for our online-algorithm.

1.4.1 Competitive Analysis

Since our algorithm's actions are uniquely determined by the input, it is a *deterministic* algorithm. For this reason we consider nothing but deterministic algorithms in the following. Those algorithms are usually appraised by comparing them with an optimal offline algorithm which knows the whole sequence σ in advance and can, therefore, serve it optimally. This leads to the definition of *c-competitiveness* (taken from [10]).

Notation 1.2 (Competitive Algorithm, Determistic Case) Let $c \geq 1$ be a real number. A deterministic online-algorithm ALG is called *c-competitive* if

$$\mathsf{ALG}(\sigma) \leq c\,\mathsf{OPT}(\sigma)$$

holds for any request sequence σ. The *competitive ratio* of ALG is the infimum over all c such that ALG is c-competitive.

Like Torres showed in [22], it is not possible to find a c-competitive algorithm for VDP . That means that for every algorithm ALG and every $c \in \mathbb{R}$ there is a request sequence σ such that $\mathsf{ALG}(\sigma) > c\,\mathsf{OPT}(\sigma)$. Since most of the problems we are faced with in real life are not as malicious as these worst case instances, it is not likely that our algorithm performs that bad in real life situations.

Thus, we have to find another way to appraise our algorithm's suitability for daily use. Hence, we test our algorithm on real world data delivered by the ADAC and compute the costs incurred, comparing them to the offline solution.

We call this value *experimental competitivity*, since we are not computing c for every σ but for some σ that really took place in reality.

The data shows that our algorithm works much better than the competitivity analysis suggests.

It is evident that the values determined in such a way only represent the considered instances, and, therefore, no generally accepted statement can be obtained this way. But since these values are very close together, we can assume that the values of the other sequences which may take place in practice are similar to them.

This way, a relative error of about 41% could be obtained (see [16]) which means that the solutions computed by our Algorithm have been in the average case about 41 % more expensive than the optimum. In these evaluations of the online performance, the time periods considered have not been longer than two hours.

Because of the high computational demand related to the computation of the offline instances, it was not possible to compute solutions for the offline instances considering more than two hours. For this reason, we introduce a new column generation method which takes a focus on this specific problem in Chapter 4.

Chapter 2

Solving the offline problem

In the following we present two different modeling approaches which can be seen as the most promising ones for the design of exact algorithms. Both of them lead to integer programs. First, we present the *arc-based* model, which is based on 0/1-variables for each arc in the graph consisting of nodes representing the requests, the current positions and the home-positions of the units. This modeling attempt leads to some problems, which will be explained in the ending of the following section.

Because of these problems we will introduce a second model which is based on 0/1-variables for every possible tour which can be driven by contractors or service units. This model is more adaptable since we are able to integrate the quadratic lateness penalties more easily. The high amount of possible tours we have to get along with is handled by a column generation approach which will be introduced in the last section of this chapter.

2.1 Arc-Based Model

In this model, we consider the time discretization $\{0, \ldots, T\}$ with a sufficiently large time horizon T. It makes sense to choose the time units to have the size of one to five minutes.

This model has a variable $x_{u,i,j} \in \{0, 1\}$ for each pair of nodes $i, j \in E$ and each unit $u \in U$. The meaning of $x_{u,i,j} = 1$ is that in the tour of unit u of our solution, node j is directly served after node i and $x_{u,i,j} = 0$ otherwise. A feasible solution is the characteristic vector of a set of paths (one for each unit) through E such that all request nodes are met exactly once.

The consequence is that we have to model

- that the $x_{u,i,j}$ in a solution must form a set of feasible paths from the units' current position o_u to their home position h_u
- that every request is visited exactly by one tour
- that every unit drives exactly one tour
- that the cost depends on the arrival times of the units at the events

Because of the last requirement, we need for every request i additional variables t_i specifying the time when the assigned service unit arrives and variables T_i representing the departure time of this unit.

First, every unit is driving a tour through some request nodes. So we need the following flow constraints in E for each unit u:

$$\sum_{j \in E \cup \{h_u\}} x_{u,o_u,j} = 1 \tag{2.1}$$

$$\sum_{i \in E \cup \{o_u\}} x_{u,i,h_u} = 1 \tag{2.2}$$

$$\sum_{i \in E \cup \{o_u\}} x_{u,i,k} - \sum_{j \in E \cup \{h_u\}} x_{u,k,j} = 0 \qquad \forall k \in E \tag{2.3}$$

Equations 2.1 - 2.3 express that driving a tour, u has to leave o_u (Equation 2.1), pass through some request nodes (Equation 2.3), and finally arrive at h_u (Equation 2.2).

Since the shift of a unit $u \in U$ starts at time t_u^{start}, u cannot depart from its current position o_u earlier than its release time.

$$T_{o_u} \geq t_u^{start} \quad \forall u \in U$$

The difference between the arrival and departure time of a request i has to be at least its service time δ_i since the request has to be served before it can be left.

$$T_i - t_i \geq \delta_i \quad \forall i \in E$$

As the service of an event i cannot start before its release time and it takes at least δ_i time to serve it, i cannot be parted earlier than $\theta_i^r + \delta_i$. This leads to the following equation:

$$T_i \geq \theta_i^r + \delta_i \quad \forall i \in E$$

The deadlines are soft. Therefore, we introduce a variable $y_i \in \mathbb{R}$ counting the lateness at request i. This lateness must additionally be positive since we do not have any benefit of arriving early:

$$\begin{aligned} y_i &\geq t_i - \theta_i^d \quad \forall i \in E \\ y_i &\geq 0 \quad \forall i \in E \end{aligned}$$

Analogously to the lateness we implement the overtime $y_{h_u} \in \mathbb{R}$ for every unit u:

$$\begin{aligned} y_{h_u} &\geq t_{h_u} - t_u^{end} \quad \forall u \in U \\ y_{h_u} &\geq 0 \quad \forall u \in U \\ t_u^{max-ot} &\geq y_{h_u} \quad \forall u \in U \end{aligned}$$

If a unit u drives directly from i to j, it needs $\delta_u^{i,j}$ time to get there and thus the time difference between the departure time at i and the arrival time at j is the driving time δ_u^{ij} from i to j. Therefore we need a constraint which ensures that $T_i + \delta_u^{(i,j)} \leq t_j$ for the case that $x_{u,i,j} = 1$. The following inequality is such a constraint:

$$x_{u,i,j}(t_j - T_i - \delta_u^{(i,j)}) \geq 0 \quad \forall i \in E \cup \{o_u\}, j \in E \cup \{h_u\}, u \in U \tag{2.4}$$

If u does not drive directly from i to j, then $x_{u,i,j} = 0$ and hence the constraint is fulfilled. Otherwise the constraint is only fulfilled if $T_i + \delta_u^{(i,j)} \leq t_j$. However, the constraint is non-linear. Exploiting the fact that we are only interested in binary values for the variables $x_{u,i,j}$, we choose an M which is large enough such that

$$t_j - T_i - \delta_u^{(i,j)} + M(1 - x_{u,i,j}) \geq 0 \quad \forall i \in E \cup \{o_u\}, j \in E \cup \{h_u\}, u \in U$$

has the same effect as equation 2.4.

Since we want every request to be served exactly once, we introduce the following constraint:

$$\sum_{u \in U} \sum_{j \in E \cup \{h_u\}} x_{u,i,j} = 1 \quad \forall i \in E$$

The cost of a solution is then a sum of driving costs, service costs, lateness costs, and overtime costs. Since the $x_{u,i,j}$ specify the paths chosen for the units and the y_i yield the lateness and overtime in time units, we can describe the cost of a solution by

$$\begin{array}{lll} & \sum_{u \in U, i \in E \cup \{o_u\}, j \in E \cup \{h_u\}} & c_u^{drv} \delta_u^{ij} x_{u,i,j} \qquad \text{(driving)} \\ + & \sum_{u \in U, i \in E, j \in E \cup \{h_u\}} & c_u^{svc} \delta_i x_{u,i,j} \qquad \text{(service)} \\ + & \sum_{i \in E} & \mathrm{LateCost}(i, \theta_i^r + y_i) \qquad \text{(lateness)} \\ + & \sum_{u \in U} & c_u^{ot} y_{h_u} \qquad \text{(overtime).} \end{array}$$

This is what we want to minimize.

The complete model reads as follows:

$$
\begin{array}{lllll}
\text{(MIP)} \quad \min & \sum_{u\in U, i\in E\cup\{o_u\}, j\in E\cup\{h_u\}} & c_u^{drv}\delta_u^{ij}x_{u,i,j} & & \text{(driving)} \\
+ & \sum_{u\in U, i\in E, j\in E\cup\{h_u\}} & c_u^{svc}\delta_i x_{u,i,j} & & \text{(service)} \\
+ & \sum_{i\in E} & \text{LateCost}(i,\theta_i^r+y_i) & & \text{(lateness)} \\
+ & \sum_{u\in U} & c_u^{ot}y_{h_u} & & \text{(overtime)}
\end{array}
$$

subject to:

$$
\begin{array}{rcll}
\sum_{j\in E\cup\{h_u\}} x_{u,o_u,j} & = & 1 & \forall u\in U; \\
\sum_{i\in E\cup\{o_u\}} x_{u,i,h_u} & = & 1 & \forall u\in U; \\
\sum_{i\in E\cup\{o_u\}} x_{u,i,k} - \sum_{j\in E\cup\{h_u\}} x_{u,k,j} & = & 0 & \forall k\in E, u\in U; \\
T_{o_u} & \geq & t_u^{start} & \forall u\in U; \\
T_i - t_i & \geq & \delta_i & \forall i\in E; \\
T_i & \geq & \theta_i^r+\delta_i & \forall i\in E; \\
y_i & \geq & t_i-\theta_i^d & \forall i\in E; \\
y_i & \geq & 0 & \forall i\in E; \\
y_{h_u} & \geq & t_{h_u}-t_u^{end} & \forall u\in U; \\
y_{h_u} & \geq & 0 & \forall u\in U; \\
t_u^{max-ot} & \geq & y_{h_u} & \forall u\in U; \\
t_j - T_i - \delta_u^{(i,j)} + M(1-x_{u,i,j}) & \geq & 0 & \forall i\in E\cup\{o_u\}, j\in E\cup\{h_u\}, u\in U \\
\sum_{u\in U}\sum_{j\in E\cup\{h_u\}} x_{u,i,j} & = & 1 & \forall i\in E \\
x_{u,i,j} & \in & \{0,1\} & \forall i\in E\cup\{o_u\}, j\in E\cup\{h_u\}, u\in U \\
y_i & \in & \mathbb{R} & \forall i\in E\cup\{h_u\}
\end{array}
$$

There are several problems connected with the arc based approach just considered:

1. The contractors are not implemented yet. This is a problem which can be solved easily by treating contractors as units with no driving and no overtime costs. Hence, only lateness and service costs are taken into account.

2. The lateness costs considered above are not linear. This non-linearity can not be handled with a simple trick like the one we applied to handle constraint 2.4. If we want to model our problem as an integer linear program by following the arc based approach, we have to replace the quadratic lateness penalty function with one or more linear functions approximating the quadratic lateness penalty with a piecewise linear function. By doing so, we would apply slight changes to the problem.

3. Linear integer programs are usually solved by finding the optimal vertex of the fractional polytope described by the constraints with respect to the objective function. If this vertex is not a feasible solution for MIP, constraints representing cutting planes are added to the linear program and the process is repeated until the best vertex is feasible for MIP and so the best feasible solution for MIP has been found. Thus, we are highly dependent on the quality of the fractional solutions obtained thereby, if we do not want to make too many steps of finding cutting planes and solving the modified linear programs again and again. It is known that inequalitites like 2.4 lead to fractional solutions of a low quality and so the solution of an instance of the MIP described above demands high computational effort.

Since we do not want to set the quadratic lateness penalty aside and we think that the problem characterized in 3. yields a running time which is not real-time compliant, we consider another model, which is more suitable for our purposes:

2.2 Tour-Based Model

This model is based on tour variables. Models of this type are by now well-established in the vehicle routing literature [7].
Let $\mathcal{R}$ be the set of all feasible *tours*. This set splits into the sets $\mathcal{R}_u$ of feasible tours for each unit u. A tour in $\mathcal{R}_u$ can be described by any ordered sequence $(u, e_1, e_2, ..., e_k)$ of k distinct events visited by u in that order. We will use the sequence (u) to denote the *go-home* tour. Feasibility means that the capabilities of the unit are sufficient for all e_i, i.e. $F_{e_i} \subseteq F_u$ for all $i = 1, ..., k$.

For all $R \in \mathcal{R}_u$ we define a binary variable x_R with the following meaning: $x_R = 1$ if and only if the route R is chosen to be in the dispatch, otherwise $x_R = 0$.
The cost of the route R is denoted by c_R. By t_R^e we denote the arrival time at event e in route R. Let the arrival time of u at its home position be $t_R^{h_u}$. Then the cost c_R of route $R = (u, e_1, ..., e_k)$ can be computed as

$$\begin{aligned} c_R = {} & c_u^{drv}\delta_u^{o_u e_1} + \sum_{i=2}^{k} c_u^{drv}\delta_u^{e_{i-1}e_i} + c_u^{drv}\delta_u^{e_k d_u} && \text{(driving)} \\ & + \sum_{i=1}^{k} c_u^{svc}\delta_{e_i} && \text{(service)} \\ & + c_u^{ot}\max\{t_R^{d_u} - t_u^{end}, 0\} && \text{(overtime)} \\ & + \sum_{i=1}^{k} \text{LateCost}(e_i, t_R^{e_i}) && \text{(lateness)} \end{aligned}$$

A feasible route S for a contractor $v \in V$ can be written as a set $\{e_1, e_2, ..., e_k\}$ of events that this contractor may be assigned to serve, i.e. $F_{e_i} \subseteq F_v$ for all $i = 1, ..., k$.
Let t_v^e be the time by which contractor v reaches event e with one of his vehicles if it was assigned to it. Since the services done by contractors are paid in a lump sum, we do not have to care about the overtime and driving costs incurred thereby, and the cost c_S of the services $\{e_1, e_2, ..., e_k\}$ can be computed as:

$$\begin{aligned} c_S = {} & c_v^{svc}|S| && \text{(service)} \\ & + \sum_{i=1}^{k} c_{e_i}^{late}\,\text{LateCost}(e_i, t_v^{e_i}) && \text{(lateness)} \end{aligned}$$

Since the service cost is linear in the number of events served by this contractor, every tour S of a contractor can be combined from elementary contractor tours, each containing only a single event. Let $\mathcal{S}^v$ be the set of elementary feasible tours for v, and let $\mathcal{S}$ be their union over all contractors $v \in V$.
For all $S \in \mathcal{S}^v$ we introduce binary variables $x_S = 1$ if and only if S is chosen to be in the dispatch.

Let a_{R_e}, b_{S_e} be binary coefficients with $a_{R_e} = 1$ (resp. $b_{S_e} = 1$) if and only if event e is served in tour R (resp. in elementary contractor tour S). The VDP can now be formulated as a set partitioning problem as follows:

$$\text{(IP)} \qquad \min \sum_{R \in \mathcal{R}} c_R x_R \quad + \sum_{S \in \mathcal{S}} c_S x_S \tag{2.5}$$

subject to

$$\sum_{S \in \mathcal{S}} b_{S_e} x_S \quad + \sum_{R \in \mathcal{R}} a_{R_e} x_R \quad = 1 \qquad \forall e \in E; \tag{2.6}$$

$$\sum_{R \in \mathcal{R}_u} x_R \quad = 1 \qquad \forall u \in U; \tag{2.7}$$

$$x_R \quad \in \{0,1\} \qquad \forall R \in \mathcal{R}; \tag{2.8}$$

$$x_S \quad \in \{0,1\} \qquad \forall S \in \mathcal{S}. \tag{2.9}$$

This model describes a dispatch which contains exactly one tour for each unit $u \in U$ (2.7) such that every request $e \in E$ is served exactly once (2.6) and such that the sum of the costs which are caused by these tours is minimal (2.5).

2.2.1 The Algorithm ZIBDIP

In the following we describe the algorithm ZIBDIP, which was developed in order to solve (IP).

ZIBDIP solves (IP) by solving its linear relaxation (LP), which is constructed by replacing the integrality constraints (2.8) and (2.9) by the nonnegativity constraints (2.10) and (2.11):

$$\text{(LP)} \qquad \min \sum_{R \in \mathcal{R}} c_R x_R \quad + \sum_{S \in \mathcal{S}} c_S x_S$$

subject to

$$\sum_{S \in \mathcal{S}} b_{S_e} x_S \quad + \sum_{R \in \mathcal{R}} a_{R_e} x_R \quad = 1 \qquad \forall e \in E;$$

$$\sum_{R \in \mathcal{R}_u} x_R \quad = 1 \qquad \forall u \in U;$$

$$x_R \quad \geq 0 \qquad \forall R \in \mathcal{R}; \tag{2.10}$$

$$x_S \quad \geq 0 \qquad \forall S \in \mathcal{S}. \tag{2.11}$$

Since the number of possible tours is exponential in the number of requests and units, it is evident that not all columns of the coefficient matrix of (LP) can be statically enumerated for our problem size. Thus, we work with dynamic column generation, which means that we start with all elementary tours for all contractors $\mathcal{S}$ and a subset $\tilde{\mathcal{R}} \subset \mathcal{R}$ consisting of a tour for each unit from its current position to its home position. This way, both the initial LP and the initial IP are feasible, as we can serve all the requests by contractors and send all the units directly home from their current position. Since this solution is usually very far from optimal, it is reasonable to add the tours of a heuristically designed dispatch, in order to have a fallback solution to meet the real-time requirements, if ZIBDIP would take too much time finding an integer solution.

This restricted problem only considering the tours $\tilde{\mathcal{R}} \subset \mathcal{R}$ reads as follows:

$$\text{(RLP)} \qquad \min \sum_{R \in \tilde{\mathcal{R}}} c_R x_R \quad + \sum_{S \in \mathcal{S}} c_S x_S \tag{2.12}$$

subject to

$$\sum_{S \in \mathcal{S}} b_{S_e} x_S \quad + \sum_{R \in \tilde{\mathcal{R}}} a_{R_e} x_R \quad = 1 \qquad \forall e \in E; \tag{2.13}$$

$$\sum_{R \in \tilde{\mathcal{R}}_u} x_R \quad = 1 \qquad \forall u \in U; \tag{2.14}$$

$$x_R \quad \geq 0 \qquad \forall R \in \tilde{\mathcal{R}}; \tag{2.15}$$

$$x_S \quad \geq 0 \qquad \forall S \in \mathcal{S}. \tag{2.16}$$

If it is not possible to find an optimal solution of (LP) with these columns, which is very likely, we generate new columns which lead to better solutions. This check for a yet disregarded tour $R \in \mathcal{R}_u \backslash \tilde{\mathcal{R}}_u$ which may lead to a better solution of (LP). In order to find such a tour we have to consider the dual (DRLP) of (RLP), which is denoted as follows:

$$\text{(DRLP)} \qquad \max \sum_{e \in E} \pi_e \quad + \sum_{u \in U} \pi_u \tag{2.17}$$

subject to

$$\sum_{e \in E} a_{R_e} \pi_e \quad + \pi_u \quad \leq c_R \qquad \forall R \in \tilde{\mathcal{R}}_u, \forall u \in U; \tag{2.18}$$

$$\sum_{e \in E} b_{S_e} \pi_e \quad \leq c_S \qquad \forall S \in \mathcal{S}. \tag{2.19}$$

$$\sum_{R \in \tilde{\mathcal{R}}_u} x_R \quad = 1 \qquad \forall u \in U; \tag{2.20}$$

For a unit $u \in U$ and a route $R \in \mathcal{R}_u$ let $\bar{c}_R$ be the reduced cost which can be computed by

$$\bar{c}_R = c_R - \sum_{e \in E} a_{R_e} \pi_e - \pi_u.$$

A tour $R \in \mathcal{R}_u \backslash \tilde{\mathcal{R}}_u$ can possibly improve the current solution of (RLP) only if $\bar{c}_R < 0$ [5].

(if they are added to the columns of (RLP). If such a column is found, it is added to the columns of (RLP) and (RLP) is solved again until no better column can be found. After each iteration of the column generation procedure, we have the optimal solution of the *restricted* LP, in which only tours from a subset $\tilde{\mathcal{R}} \subset \mathcal{R}$ have been considered.)

2.2.2 Column Generation

There are several possiblities to find the columns with negative reduced costs. The easiest method is the enumeration of all possible tours in a search tree. Each node in this search tree corresponds with a tour starting at the current position of a unit and ending at the position of the last event served by the unit. A tour associated to a node can be completed to a feasible tour by appending the tour from the position of the last event in the node to the unit's home position. The *pre-cost* of a node in the search tree is the reduced cost of the corresponding tour (without returning to the unit's home position and overtime). The *cost* of a node in the search tree is defined as the reduced cost of the corresponding feasible tour (including the costs for returning to the unit's home position and

overtime). The *dual prices* for the events and units are taken from the previous run of the LP solver. The root node r corresponds to the empty tour. Given a node v in the tree, the children of v are obtained by appending one event to v that is not yet in v. This way all possible tours can be enumerated.

Since a whole enumeration of all possible tours is very time-consuming, we do not scan the whole search tree. In fact, for every node within the search tree we only consider the d most promising children and we do not consider tours serving more than l requests for some intelligently chosen d and l. The d most promising children of a node v and the order in which they are processed is given by sorting the children in the order $<$ of increasing reduced costs of the new node (*greedy*), or increasing primal cost (*primal-greedy*). Since we are especially interested in tours having low reduced costs, we introduce an acceptance threshold a which means that we skip all tours having greater reduced costs than a. By the setting of the parameters for the *maximal search depth* l, the *maximal search degree* d, and the *acceptance threshold* a the processing time and the quality of the output can be customized.

As even the "customized" search tree may still contain a lot of useless nodes, we apply a Branch-and-Bound method in order to explore only the important parts of the search tree (a detailed description of the Branch-and-Bound Method can be found on page 78). For this reason, we skip every subtree having no chance of containing a node with cost smaller than the acceptance threshold. This procedure of skipping a subtree is called *pruning*.

The just described procedure looks like this:

GoodColumns

Input: Restricted linear program RLP, maximal search depth l,
maximal search degree d, acceptance threshold a,
current node v ($v := r$ if not specified),
sorting criterion for partial tours $<$

Output: The linear program RLP with additional columns

① **If** v has length l **then return**
② **while** less than d children of v are visited **do**
③ pick the next best child c of v according to search order $<$
④ **If** cost of c smaller than a **then**
⑤ add column corresponding to c to RLP
⑥ **If** LowerBound(RLP,c) smaller than a **then**

⑦ GoodColumns(RLP,$l, d, c, a, <$)
⑧ **return** RLP

LowerBound(RLP,c) is a function which calculates a lower bound for the tours representing the nodes within the subtree of c. In the next chapter we will show two of such functions.

2.2.3 Implementation of ZIBDIP

The algorithm ZIBDIP reads as follows:

ZIBDIP
Input: Instance of VDP
Output: optimum dispatch

① initialize LP,$l, d, a, t, <$
② **while** true **do**
③ **repeat**
④ // *generate new columns in search tree of degree d and depth l :*
⑤ **For** every $u \in U$ **do**:
⑥ GoodColumns(RLP,$u, l, d, a, <$)
⑦ double [halve] a if less [more] than 1000 columns were generated
⑧ solve RLP
⑨ update dual prices
⑩ **If** LP progress sufficient and elapsed time large enough **then**
⑪ solve RIP corresponding to RLP with time limit t
⑫ increase t
⑬ output corresponding dispatch to a file
⑭ **If** optimality check successful **then**
⑮ mark RLP as optimal
⑯ **break**
⑰ increase l
⑱ **until** RLP progress stalls or $l > |E|$
⑲ **If** LP marked optimal **then break**
⑳ increase d
㉑ set l to initial value

㉒ solve RIP corresponding to RLP to optimality
㉓ **return** corresponding dispatch and gap

We generate columns (⑥) for each unit in loops with increasing values for the maximal search depth (inner loop: ③ - ⑰) and the maximal search degree (outer loop: ② - ㉑). The values for the maximal search depth are increased (⑰)until no progress has been made in the previous step provided the search depth was sufficiently large, at latest when the depth equals the number of events(⑱). The search degree is increased until an optimality criterion is met (⑭) or the search degree has reached the number of events.

While we are adding columns to the LP we fix the upper bound to a negative acceptance threshold: all columns that have reduced costs smaller than the acceptance threshold are added to the LP. This acceptance threshold is updated ⑥ after each iteration depending on the number of columns produced.

2.2.4 Lower Bounds for LP

Since we want to have an estimate about the quality of our current solution, we use a lower bound which is usually attributed to Lasdon coming from the Lagrangean Relaxation of (LP) w.r.t. the constraints (2.6).

Proposition 2.1 *Let (π_e^*, π_u^*) be an optimal solution of (DRLP) and $(x_R^*, x_S^*)^T$ be the corresponding primal solution. Then the cost c_{LP}^{opt} of an optimal solution of (LP) satisfies.*

$$c_{LP}^{opt} \geq \sum_{R \in \tilde{\mathcal{R}}} c_R x_R^* + \sum_{S \in \tilde{\mathcal{S}}} c_s x_S^* + \sum_{u \in \mathcal{U}} \min_{R \in \mathcal{R}_u} (c_R - \sum_{e \in E} a_{R_e} \pi_e^* - \pi_u^*) \tag{2.21}$$

To compute this lower bound, we have to calculate $\min_{R \in \mathcal{R}_u}(c_R - \sum_{e \in E} a_{R_e} \pi_e^* - \pi_u^*)$ for every unit u. This can be done by enumerating all possible tours. Since this is usually very time-consuming, we solve a relaxation of the problem in order to find a lower bound for the minimum reduced cost of u in Chapter 4.

Chapter 3

Lower Bounds for the Branch & Bound pricing Algorithm

In this chapter lower bounds for the tours associated with the nodes within a subtree of a B & B search tree are presented. The quality of this lower bound is important as a good lower bound makes it possible to make a much better pruning and therefore to iterate less nodes. We present two of these lower bounds. The first one has been used until now and delivered good results. The second one works exactly like the first one with the difference that the lateness costs are taken into account in a much more realistic way. As a result only a fraction of the previously examined nodes has to be computed by now. On the other hand, the computation takes more time, since an ASSIGNMENT- problem has to be solved at every node within the search-tree. In fact, we use a two-phase pruning scheme, which only computes the new lower bound if the old lower bound was two weak for pruning. In the last section we will test the new pruning on several different instances.

To every node v of the search tree there is a sequence of requests $S(v)$ which are served by a unit u in the given order. Let e_v be the last event in $S(v)$, t_v be its completion time, $l(v)$ be the length of $S(v)$. Let $c(v)$ be the reduced cost of the tour corresponding to $S(v)$. Let $E_v = \{e | F_e \subseteq F_u \wedge e \notin S(v)\}$ be the set of all requests which can be appended to $S(v)$ so that the resulting tour is feasible. Let l be the maximum depth of the search tree and $d_v := l - l(v)$ be the maximum number of requests still to append to the tour corresponding to v. Let Subtree(v) be the subtree of v within the search tree and let

$$\text{Low}(v) := \min_{w \in \text{Subtree}(v)} c(w)$$

be the value of the cheapest node within the subtree of v. This node corresponds to the minimum reduced costs of a tour beginning with $S(v)$.

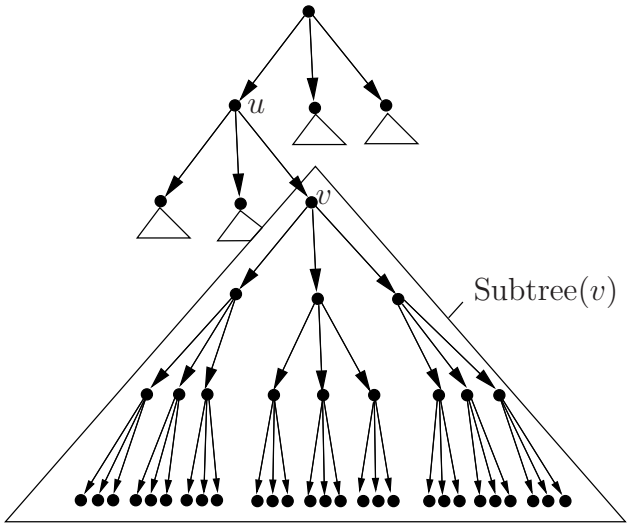

Figure 3.1: Example for a searchtree with $degree = 3, l = 5, l(v) = 2, d_v = 3, S(v) = (e_u, e_v)$

Notation 3.1 (minimum reduced costs of an event) Let

$$\text{MinRedCost}(e_1, e_2, t) := \text{LateCost}(e_2, t + \delta_u^{e_1 e_2}) + c_u^{svc}\delta_{e_2} - \pi_{e_2}$$

be the minimum reduced costs of the event e_2, if e_1 is being left for e_2 at time t. These costs only include minimum lateness and service but no driving and overtime costs.

Therefore, $\text{MinRedCost}(e_v, e, t_v)$ is a lower bound for the reduced costs of e in the subtree below v.

When processing v one has to estimate a lower bound for the reduced costs of a tour beginning with $S(v)$ resp. a node within the subtree of v. If this lower bound is greater than the acceptance threshold, the subtree contains no node which corresponds to a tour with reduced costs less than the acceptance threshold and the subtree can be pruned. By now there used to be the following lower bound scheme:

Notation 3.2 (minimum reduced costs of a node within a subtree)

$$\text{MaximalGain}(v) := c(v) + (\min_{E \subseteq E_v, |E| \leq d_v} \sum_{e \in E} \text{MinRedCost}(e_v, e, t_v))$$

MaximalGain(v) is a lower bound for the reduced costs of a node within the subtree of v, since MaximalGain(v) assumes that all requests can be served directly after e_v.

Proposition 3.3

$$\text{MaximalGain}(v) \leq \text{Low}(v).$$

So far, this lower bound did a good job pruning the search tree. Most of the costs arising in this model are lateness costs implemented to assure customers' short waiting times. Therefore, it seems to be promising to take a closer look on the lateness costs considered by this lower bound and to develop strategies to improve them. The lateness costs in MaximalGain(v) are computed according to the assumption that the service unit starts off for serving each single request in E immediately after serving e_v. But as a tour consists of requests which have to be served in a certain order, only one request can be attended at the completion time of the last request in $S(v)$. The second one has to wait at least for the service-time of the first request, the third request has to wait for the service-time of both of the previous requests extra and so on. Using the lower bound described above the late costs thereby incurred are being disregarded. Therefore, as a general rule, this lower bound is not strong for $d_v > 1$ due to the fact that a service unit can serve only one request at a time.

In the following we introduce a new lower bound taking the late costs of sequencialization into account.

Let $\delta := \min_{e \in E_v} \delta_e$ be the minimum service time of all requests in E_v. It is trivial to see that δ is a lower bound for the servicetime of all requests e in E_v. In practice it takes a service unit at least 15 minutes to serve a customer and, therefore, $\delta \geq 15$.

Since every unit u can serve only one request immediately, the second request has to wait for at least δ time units longer as if it had been served first. The third request has to wait for at least 2δ time units longer as if it had been served first by u etc. Therefore, there are a lot of lateness costs emerging from the sequencialization of the latter requests which have been disregarded yet.

Example 3.4 The service unit u finishes its service at the request e_v at 2:50 pm. There are still three jobs e_1, e_2 and e_3 to do. From e_v it takes u 18 minutes to get to e_1, 10 minutes to get to e_2 and 23 minutes to get to e_3. Every job has a service time of 20 minutes. If the driver served e_2 immediately after serving e_v, he would get there at 3:00 pm. Then he would spend 20 minutes repairing the car. If he drove to e_1 next, it would take him at least (18-10=) 8 minutes to get there, because of the triangle inequality. That's why he cannot start working on

e_1 earlier than 3:28 pm. It will take him another 20 minutes to help the person waiting for him there so that he can leave e_1 no earlier than 3:48. At last he serves e_3. To get there takes him at least 5 minutes, meaning that he can start working there no earlier than 3:53 pm. The following table contains the lower bounds of the arrival times depending on their order:

	Earliest possible arrival time		
	1st	2nd	3rd
e_1	3:08 pm	3:28 pm	3:48 pm
e_2	3:00 pm	3:20 pm	3:40 pm
e_3	3:13 pm	3:33 pm	3:53 pm

We obtain a lower bound for $\text{Low}(v)$ by assigning up to d_v different requests from E_v to the elements of $T_{v,\delta} := \{t_v, t_v + \delta, t_v + 2\delta, t_v + 3\delta, ..., t_v + (d_v - 1)\delta\}$, such that each $t \in T_{v,\delta}$ is covered not more than once and the sum of the reduced costs arising by leaving e_v for $e \in E_v$ at the assigned point in time is minimal. Not every event $e \in E_v$ can be assigned to a $t \in T_{v,\delta}$, since $|E_v| \neq |T_{v,\delta}|$ in the majority of cases. This leads to the following definition of the new lower bound:

Notation 3.5 (lower bound due to improved pruning)

$$\text{ImprovedBound}(v) := c(v) + \min_{\{e_0,e_1,...,e_k\} \subseteq E_v} \sum_{i=0}^{k} \text{MinRedCost}(e_v, e_i, t_v + i\delta),$$

with $k \leq d_v - 1, e_i \neq e_j \ \forall i \neq j$

Note that if we only assign a subset of $T_{v,\delta}$, it suffices to consider the first ones, since for every request e $\text{MinRedCost}(e_v, e, t)$ is monotonic increasing in t. Furthermore, no request e will be assigned to a time t with $\text{MinRedCost}(e_v, e, t) \geq 0$, as the sum can only become smaller by leaving this request out.

Proposition 3.6

$$\text{MaximalGain}(v) \leq \text{ImprovedBound}(v).$$

Proof. Since for every request e $\text{MinRedCost}(e_v, e, t)$ is monotonic increasing in t, we know that for all $e \in E_v$ and $i \in \{0, 1, ..., d_v - 1\}$ $\text{MinRedCost}(e_v, e_i, t_v) \leq$

$\text{MinRedCost}(e_v, e_i, t_v + i\delta)$. Thus,

$$\begin{aligned}
\text{MaximalGain}(v) &= c(v) + \min_{E \subseteq E_v, |E| \leq d_v} \sum_{e \in E} \text{MinRedCost}(e_v, e, t_v) \\
&= c(v) + \min_{\{e_0, e_1, \ldots, e_k\} \subseteq E_v, k \leq d_v} \sum_{i=0}^{k} \text{MinRedCost}(e_v, e_i, t_v) \\
&\leq c(v) + \min_{\{e_0, e_1, \ldots, e_k\} \subseteq E_v, k \leq d_v} \sum_{i=0}^{k} \text{MinRedCost}(e_v, e_i, t_v + i\delta) \\
&= \text{ImprovedBound}(v)
\end{aligned}$$

□

Proposition 3.7

$$\text{ImprovedBound}(v) \leq \text{Low}(v).$$

Proof. For computing $\text{ImprovedBound}(v)$, we only consider service costs and late costs of sequencialization. Since these costs cannot be omitted, $\text{ImprovedBound}(v)$ is a lower bound for $\text{Low}(v)$. □

Remark 3.8 In some cases $\text{ImprovedBound}(v)$ is a strong lower bound for $\text{Low}(v)$. Let E_s be the set of requests to be served after e_v according to the computation of $\text{ImprovedBound}(v)$. If

1. all requests $e \in E_s$ to be served after e_v lie on the service unit's way home from e_v in the order of $\text{ImprovedBound}(v)$'s computation (no extra driving, no extra lateness),
2. $\forall e \in E_s : \delta_e = \delta$ (no extra service cost),
3. $t_u^{end} \geq t_v + (|E_s| - 1)\delta$ (shift end time late enough)
4. and no waiting time (no extra lateness)

then $\text{ImprovedBound}(v) = \text{Low}(v)$

3.1 Modeling

The problem of assigning events to points in time can be modelled as a minimum weight matching problem in a bipartite graph.

Therefore, we define a bipartite graph $G = (V, A)$ with $V = E \dot{\cup} T$, where E contains $|E_v|$ nodes e_i $(i = 0, 1, ..., |E_v| - 1)$ for each request $e_i \in E_v$. The set T contains d_v nodes t_j $(j = 0, 1, ..., d_v - 1$, where t_j represents the time $t_v + j\delta$. For every $i < |E_v|, j < d_v$ with $\text{MinRedCost}(e_v, e_i, t_v + j\delta) < 0$, there is an edge (e_i, t_j) with cost $\text{MinRedCost}(e_v, e_i, t_v + j\delta)$.

Let M_j be the minimum cost matching of cardinality j. Let $c(M_j) := \sum_{e \in M_j} c(e)$. Let M be the optimal matching with $c(M) = \min_{j=1,...,d_v} c(M_j)$. The improved bound can be computed as $\text{ImprovedBound}(v) = c(v) + c(M)$

The minimum weight matching problem in a bipartite graph can be reduced to a network flow problem (see [2]). For basic definitions about network flow problems see page 77.

Therefore, we add a vertex s and connect it to all vertices of E. These edges have zero cost. Orient the edges from s to E and from E to T. Let all edges have capacity 1. This results in the following graph (see Figure 3.1). We will call these special kind of networks *lateness matching networks*.

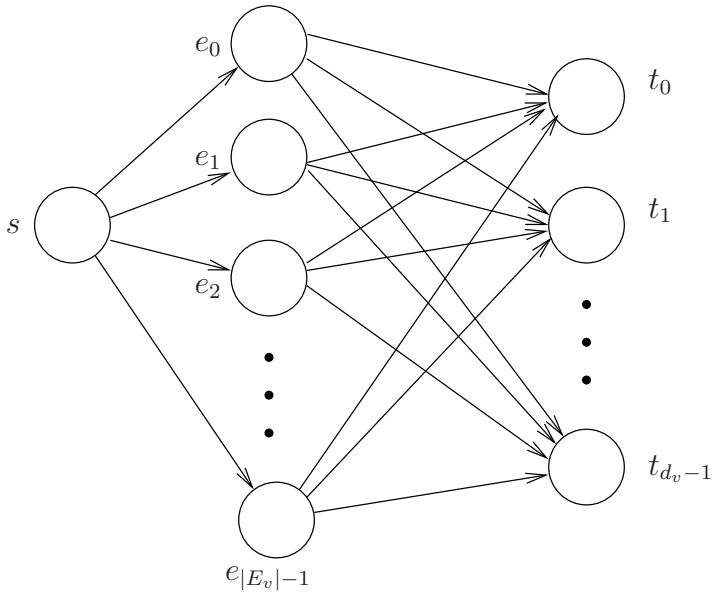

Figure 3.2: The network corresponding to the assignment problem

There is a wide variety of different algorithms which can be used to solve such a minimum cost flow problem. In our case the most promising algorithm is the successive shortest path algorithm (see page 29). It works by adjusting the imbalances of pairs of sources and sinks by augmenting along paths of minimum weight between them. A detailed description of this algorithm reads as follows:

Successive Shortest Path Algorithm

Input: A digraph G, capacities $u\colon E(G) \to \mathbb{R}_+$, numbers $b\colon V(G) \to \mathbb{R}$ with $\sum_{v \in V(G)} b(v) = 0$, and conservative weights $c\colon E(G)\colon \to \mathbb{R}$.
Output: A minimum cost b-flow f.

① Set $b' := b$ and $f(e) := 0$ for each $e \in E(G)$
② **If** $b' = 0$ **then stop**
③ **else** choose a vertex s with $b'(s) > 0$.
④ Choose a vertex t with $b'(t) < 0$ such that t is reachable from s in G_f.
⑤ **If** there exists no such t **then stop**. (There exists no b-flow.)
⑥ Find $s - t$-path P in G_f of minimum weight
⑦ Compute $\gamma := \min\{\min_{e \in E(P)} u_f(e), b'(s), -b'(t)\}$
⑧ Set $b'(s) := b'(s) - \gamma$ and $b'(t) := b'(t) + \gamma$. Augment f along P by γ.
⑨ **Go to** ②

Since we know that there is always an optimal matching of cardinality j, matching exactly the j first points in time, we can find M_j by finding a minimum cost b-flow with $b(s) = j$, $b(t_i) = -1, i = 0, \ldots, j-1$ and $b(v) = 0$ for all other nodes.

The minimum cost flow problem in bipartite graphs is solved by applying the following variant of the successive shortest path algorithm.

Improved Bound

Input: An acceptance threshold *threshold*,
a *lateness matching network* $G = (V, A)$
Output: *true*, if a lower bound *bound*
with *bound* > *threshold* could be found
false, otherwise

① Set $f(e) := 0$ for each $e \in A$
② $j := 1$

③ **while** $j \leq d_v$ **do**
④ Find $s - t_{j-1}$-path P_j in G_f of minimum weight.
⑤ **If** there exists no such path **then return** *true*
⑥ **else**
⑦ **If** $c(P_j) \geq 0$ **then return** *true*
⑧ Augment f along P_j
⑨ Let M_j be the matching corresponding to f
⑩ **If** $c(M_j) \leq threshold$ **then return** $false$
⑪ $bound := c(M_j) + (d_v - j)c(P_j)$
⑫ **If** $bound > threshold$ **then return** *true*
⑬ $j := j + 1$
⑭ **return** *true*

In Comparison to the standard successive shortest path algorithm, we implement some shortcuts in ⑤, ⑦, ⑩ and ⑫. In the following we will explain their meaning.

Therefore, we need the following propositions:

Proposition 3.9 *Let P_j and P_{j+1} be the augmenting paths found by* ImprovedBound(v) *in the jth resp. $j+1$th. iteration. Then*

$$c(P_j) \leq c(P_{j+1})$$

Proof.

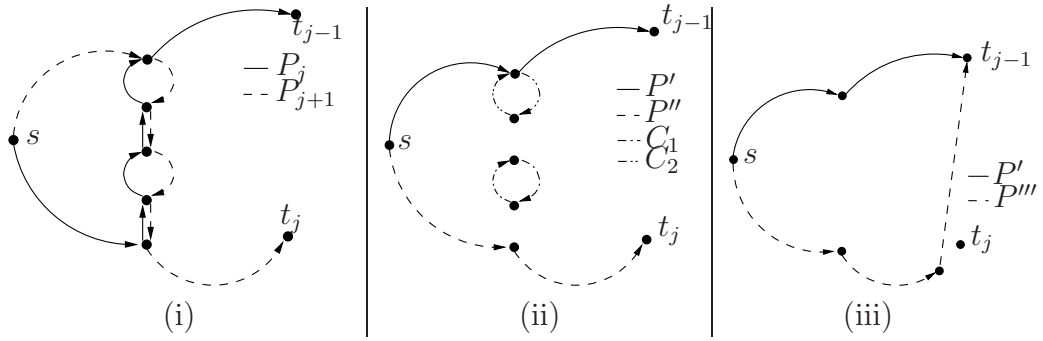

Figure 3.3: Paths and Cycles in $G_{f_{j-1}} \cup G_{f_j}$ for the proof of proposition 3.9

Let f_j be the flow corresponding to M_j and let G_{f_j} be its residual graph. P_j and P_{j+1} are the augmenting paths of iteration j resp. $j+1$ (see Figure 3.3 (i)). By deleting all pairs of reverse edges of P_j and P_{j+1} we obtain two edge-disjoint

cycle-free paths P' and P'' from s to t_{j-1} resp. to t_j and a set $\mathcal{C}$ of cycles (see Figure 3.3 (ii)). Since the reverse edges differ from each other only in their sign, the sum of the costs of a pair of reverse edges is 0 and therefore:

$$c(P_j) + c(P_{j+1}) = c(P') + c(P'') + \sum_{C \in \mathcal{C}} c(C) \tag{3.1}$$

Since the only difference between $G_{f_{j-1}}$ and G_{f_j} is that all edges of P_j have a different orientation (and of course different costs), every edge in P_{j+1} which is not part of $G_{f_{j-1}}$ must have a reverse edge in P_j. As P', P'' and $\mathcal{C}$ do not contain any of these edges, they lie in $G_{f_{j-1}}$. Since f_{j-1} is optimal, $G_{f_{j-1}}$ does not contain any negative cycles and therefore:

$$\begin{aligned} &\forall C \in \mathcal{C} \colon c(C) \geq 0 \\ &\Rightarrow \sum_{C \in \mathcal{C}} c(C) \geq 0 \end{aligned} \tag{3.2}$$

Let P''' be the $s - t_{j-1}$-path which is constructed by changing the last edge's target of P'' from t_j to t_{j-1} and deleting any cycle which might be gained thereby (see Figure 3.3 (iii)). If the last edge of P''' was not part of $G_{f_{j-1}}$, P_j would contain cycles which is impossible. Hence P''' is a s-t_{j-1}-path in $G_{f_{j-1}}$. Because of the monotonic increasing reduced cost function and because of the fact that every cycle in $G_{f_{j-1}}$ is nonnegative:

$$c(P''') \leq c(P'') \tag{3.3}$$

As P_j is the shortest s-t_{j-1}-path in G_{f_j}:

$$c(P_j) \leq c(P') \tag{3.4}$$

$$c(P_j) \leq c(P'''). \tag{3.5}$$

Thus, it follows from above:

$$
\begin{aligned}
c(P_j) + c(P_{j+1}) &\overset{3.1}{=} c(P') + c(P'') + \sum_{C \in \mathcal{C}} c(C) \\
&\overset{3.2}{\geq} c(P') + c(P'') \\
&\overset{3.3}{\geq} c(P') + c(P''') \\
&\overset{3.4,3.5}{\geq} c(P_j) + c(P_j) \\
&\\
&\Rightarrow c(P_j) \leq c(P_{j+1})
\end{aligned}
$$

□

Proposition 3.10 *Let M_j and M_i be minimum cost matchings of cardinality i resp. j with $i < j$, then*

$$c(M_j) \geq c(M_i) + (j - i)c(P_i)$$

Proof.

$$
\begin{aligned}
c(M_j) &= c(M_i) + c(P_{i+1}) + c(P_{i+2}) + \ldots + c(P_j) \\
&\overset{Pro.3.9}{\geq} c(M_i) + (j - i)c(P_i)
\end{aligned}
$$

□

Proposition 3.11 *Let M_k be the minimum cost matching and let h be the highest cardinality of a matching possible among the matchings of G. Then*

$$c(M_1) \geq c(M_2) \geq \ldots \geq c(M_k) \leq c(M_{k+1}) \leq \ldots \leq c(M_h)$$

Proof. Since M_k is optimal we know that

$$c(M_{k-1}) \geq c(M_k) = c(M_{k-1}) + c(P_k)$$

and, therefore, $c(P_k) \leq 0$. From proposition 3.9 it follows that for all j with $1 \leq j < k$ that $c(P_j) \leq c(P_k) \leq 0$ and, thus, $c(M_j) \geq c(M_{j+1})$. Analogously since M_k is optimal we know that

$$c(M_k) \leq c(M_{k+1}) = c(M_k) + c(P_{k+1})$$

and, therefore, $c(P_{k+1}) \geq 0$. From proposition 3.9 it follows that for all j with $k < j \leq d_v$ we have $c(P_j) \geq c(P_k) \geq 0$ and, thus, $c(M_{j-1}) \leq c(M_j)$. □

As a consequence of the propositions mentioned above, we can justify the shortcuts implemented in the Improved Bound algorithm:

The algorithm can be interrupted in ⑩, since we have found a matching M_j with $c(M_j) < threshold$ and, therefore, $c(M) < threshold$.

If $c(P_j) \geq 0$, then due to proposition 3.11 all matchings with a cardinality greater than $j-1$ will have higher costs than M_{j-1}. Since the algorithm has not been interrupted yet by the condition in ⑩, for all $i < j$ we have $c(M_i) > threshold$. This results in $c(M_j) > threshold$ for all $j \leq d_v$ and, therefore, the algorithm can be stopped in ⑦.

The algorithm can be terminated in ⑤, because it is not possible to find a matching with cardinality j or higher. That means, the minimum cost matchings of every cardnality have already been computed and since their costs have all been greater than *threshold* we have found our searched lower bound in $\min_{i \leq j-1} c(M_i)$. Due to proposition 3.11 we know that M_{j-1} is the minimum cost matching.

The algorithm can be interrupted in ⑫, since for all $i \geq j : c(M_i) \geq c(M_j) + (i-j)c(P_j) > threshold$. And since for all $i \leq j : c(M_i) > threshold$, we know that $c(M) > threshold$ and the algorithm can be stopped.

3.2 Computational aspects

Let $l' := \min_{e \in A} c(e)$ be the cost of the cheapest service possible. Since every lower bound computed by our algorithm estimates the value of a matching of cardinality d_v, by adding $-l'$ to each of the arcs from E to T, the matchings do not change, but the values of all estimates change by $d_v l'$. By doing so we ensure that all arcs have nonnegative costs. Instead of working on the network introduced above we use this slightly modified network. Thus, we can use the DIJKSTRA-algorithm for the first shortest path computation. By using node potentials (see [2]), we can use the DIJKSTRA-algorithm for the other iterations, too.

Proposition 3.12 *The running time is* $O(|E_v|^2 d_v)$.

Proof. We have got to find at most d_v shortest paths. Having n nodes, the running time of the DIJKSTRA-algorithm is $O(n^2)$ (see [14]). Since the number

of nodes is $|E_v| + d_v + 1)$ we have got a worst case running time of $O((|E_v| + d_v + 1)^2 * d_v) = O(|E_v|^2 d_v)$. □

3.3 Preprocessing

Since the running time is highly dependent on the number of events considered, we should think about a smaller subset of E_v.

Definition 3.13 (k cheapest requests at t) Let

$$E_{v,t,k} := \{e \in E_v : |\{e' \in E_v : c(e', t) < c(e, t)\}| < k\}$$

be the set of the k cheapest requests at t.

As $E_{v,t,k}$ changes with increasing t (see Figure 3.4), we need another definition:

Definition 3.14 (Set of the k cheapest requests in T) Let $E_{v,T,k} := \cup_{t \in T} E_{v,t,k}$ be the set of requests which belong to the k cheapest requests for at least one $t \in T$.

Proposition 3.15 *There is always an optimal matching M with $M \subseteq \delta(E_{v,T,d_v})$.*

Proof. Suppose there is a minimal matching M with $M \nsubseteq \delta(E_{v,T,d_v})$. Then there is a node $w \in E_v \setminus E_{v,T,d_v}$ with $(w, t_i) \in M, t_i \in T$. This node does not belong to the d_v cheapest nodes at time t_i. As M is of cardinality at most d_v and $|E_{v,t_i,d_v}| \geq d_v$, there is a node $x \in E_{v,t_i,d_v}$ which is not incident to any edge in M. From the definition of E_{v,t_i,d_v} follows that $c((x, t_i)) < c((w, t_i))$ and, therefore, that $c(M \cup (x, t_i) \setminus (w, t_i)) < c(M)$ which leads to a contradiction. □

Usually E_{v,T,d_v} is much smaller than E_v. Therefore, the cost of computation can be lowered by identifying E_{v,T,d_v}. Since it takes a worst case running time of $O(|E_v|^2)$ to determine E_{v,T,d_v}, we used a superset of E_{v,T,d_v} which can be determined in linear time:

Definition 3.16 Let $E'_{v,T,k}$ be the set of requests which belong to the k cheapest requests at the first point in time of T and all the requests which are cheaper than at least one of them at the last point in time of T.

$$E'_{v,T,k} := E_{v,t_0,k} \bigcup \{e \in E_v | \exists e' \in E_{v,t_0,k} : c(e, t_{|T|-1}) < c(e', t_{|T|-1})\}$$

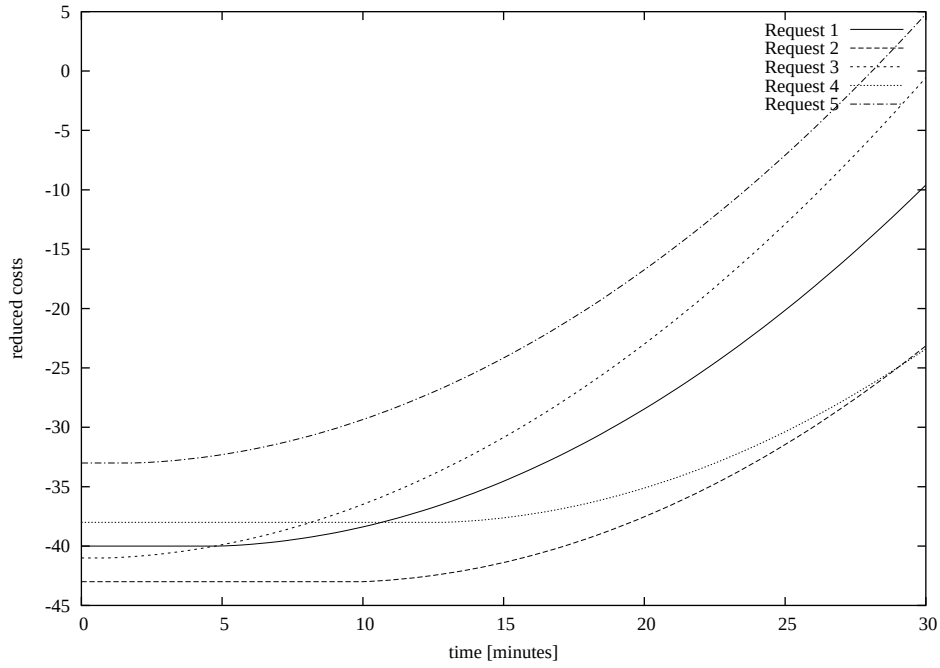

Figure 3.4: Some reduced cost functions

Proposition 3.17 $E_{v,T,d_v} \subseteq E'_{v,T,d_v}$

Proof. Choose $e \in E_{v,T,d_v}$ arbitrarily. There is a $t_j \in T$ with $e \in E_{v,t_j,d_v}$. If $j = 0$, which means that e belongs to the d_v cheapest requests at time t_0, then $e \in E'_{v,T,d_v}$. If $j > 0$ there is an event $e' \in E_{v,t_0,d_v}$ with $c(e, t_0) > c(e', t_0)$ and $c(e, t_j) \leq c(e', t_j)$. As the requests' reduced cost functions are continuous, there has to be a point of intersection in $[t_0, t_j]$ of the cost functions of e and e'. Because of their specific form there can be no other point of intersection in $(t_j, t_{d_v-1}]$ and, therefore, $c(e, t_{d_v-1}) < c(e', t_{d_v-1})$ and $e \in E'_{v,T,d_v}$. □

Since the elements of $E_v \setminus E'_{v,T,d_v}$ won't be assigned to any element of T, we can do all the computations introduced in the previous section with E'_{v,T,d_v} instead of E_v.

3.4 Computational Results

In the following, we are analyzing how good our improved pruning scheme works on real life instances.

Since it is always stronger than the pruning we previously used, the amount of nodes explored is always smaller or at the worst case equal to the number of nodes explored by using the old pruning scheme. Therefore, we can conclude from the theoretical results we proved above that our pruning is always better than the pruning previously used with respect to the number of nodes explored within any search tree. On the other hand, it takes more time to compute this more sophisticated bound. This results in the fact that even if there is only a small fraction of the nodes explored, it might happen that it takes much more time to explore them. For this reason, we focus on the improvement ratio with respect to the time taken for the exploration of the search trees.

The problem instances of ZIBDIP can be divided into two groups. One of them are the snapshots, which are solved for the online-dispatching. The other problems are those, which have to be solved for a-posterio analysis in order to compute lower bounds for the optimal online dispatching. These instances are very different from the snapshots for several reasons. The snapshots are usually smaller by means of the amount of units and events to be handled. Furthermore, the difference between these problems is that at processing time, all events of the snapshots have already been released. Whereas when dispatching instances for the a-posterio-analysis the algorithm has to get along with events which have not already been released. As a matter of fact, the lateness costs of the snapshots are usually higher, and thus we are expecting our new pruning scheme to have a better performance ratio for the snapshots than for the offline instances.

For the test of the codes referenced in this chapter we used two different computers. For the computations made in Subsection 3.4.1, we used a 3.06 Ghz Pentium 4 machine, which was equipped with 2 GB of RAM . For the computation of the results presented in Subsection 3.4.2 we used a 800 Mhz Pentium III machine, which was equipped with 1 GB of RAM. The prototype ZIBDIP ran under Linux system (kernel version 2.4.21) on both of the machines and was compiled using GNU C++-compiler gcc, version 3.3.1 . The compiler flags were $-g-06$. ZIBDIP used the linear programming and integer linear programming solver CPLEX version 8.000

3.4.1 Snapshots

In Table 3.1 we present an overview over the experimental data obtained from the computation of the optimal dispatch. These instances are taken from data the ADAC gave us. They are taken equally from the whole day, such that we have instances with a higher system load, which is defined as $r := \frac{|R|}{|U|}$, like e.g. snapshot.900 having a load of $r = 2.62$ and instances with a lower system load

like snapshot.240 with $r = 1.23$. In fact, all of these snapshots are, due to the definition made in accordance with the ADAC, normal load instances since $1 < r \leq 3$ for all of them.

The first column denotes the name of the instance, the second one the amount of iterations $|I|$ computed, the third one the number of requests $|R|$, and the fourth one the number of units $|U|$. The fifth column contains the amount T_{imp} of time taken for the column generation in seconds. The sixth column contains the number N_{imp} of nodes explored in B & B search trees by the column generation. The seventh and eighth coulmns contain analogously the times T_{usu} and number of nodes N_{usu} for the column generation worked with the old pruning scheme. In the following two columns we calculated the improvement ratios in order to show how much time has been saved and how much nodes had to be explored by our approach.

Name	$\|I\|$	$\|R\|$	$\|U\|$	T_{imp}	N_{imp}	T_{usu}	N_{usu}	$\frac{T_{imp}}{T_{usu}}$	$\frac{N_{imp}}{N_{usu}}$
snapshot.240	9	123	100	0.74	9015	0.8	11682	0.925	0.772
snapshot.270	9	125	100	0.69	8927	0.81	11875	0.852	0.752
snapshot.300	9	142	100	1.11	12671	1.22	16976	0.91	0.746
snapshot.330	9	146	100	1.09	12018	1.25	16997	0.872	0.707
snapshot.360	14	156	100	2.13	19974	2.59	37584	0.822	0.531
snapshot.390	9	175	100	2.04	17939	2.45	26376	0.833	0.68
snapshot.420	9	183	100	1.75	14269	2.04	21774	0.858	0.655
snapshot.450	18	200	100	3.49	27267	4.36	52777	0.8	0.517
snapshot.480	12	203	100	2.86	19191	3.38	29961	0.846	0.641
snapshot.510	18	215	100	4.35	31546	5.78	65186	0.753	0.484
snapshot.540	9	216	113	3.01	26029	3.49	34342	0.862	0.758
snapshot.570	9	218	113	2.23	16530	2.71	25386	0.823	0.651
snapshot.600	63	237	113	44.35	208755	120.12	932008	0.369	0.224
snapshot.630	63	250	113	52.43	222860	137.87	1012705	0.38	0.22
snapshot.660	52	247	113	35.61	163249	82.7	616054	0.431	0.265
snapshot.690	63	261	113	67.96	253480	187.9	1350234	0.362	0.188
snapshot.720	63	269	113	79.18	282461	249.28	1702267	0.318	0.166
snapshot.750	63	278	113	90.23	301210	339.73	2247257	0.266	0.134
snapshot.780	88	291	113	171.94	589261	664	4395655	0.259	0.134
snapshot.810	18	290	113	16.89	68629	23.95	160914	0.705	0.426
snapshot.840	33	296	113	41.64	170170	83.15	665072	0.501	0.256
snapshot.870	33	299	113	40.59	156729	65.94	569405	0.616	0.275
snapshot.900	33	308	118	40.14	162978	78.74	671195	0.51	0.243
snapshot.930	25	307	118	33.43	147367	49.23	349847	0.679	0.421
snapshot.960	33	306	118	47.07	182726	81.76	645013	0.576	0.283
snapshot.1020	33	314	135	36.5	150897	62.97	527625	0.58	0.286
snapshot.1050	75	322	135	170.95	597153	599.28	3847145	0.285	0.155
snapshot.1080	75	311	135	139.08	507560	419.07	2567749	0.332	0.198
snapshot.1110	75	316	143	151.4	565927	471.38	2870062	0.321	0.197
snapshot.1140	33	314	143	35.9	184356	59.13	533537	0.607	0.346
snapshot.1170	88	318	143	250.96	1013204	790.36	4978175	0.318	0.204
snapshot.1200	25	308	143	26.24	144604	32.7	304823	0.802	0.474
snapshot.1230	18	304	143	15.44	85954	18.05	167503	0.855	0.513
snapshot.1260	25	302	143	22.65	148705	29.02	305682	0.78	0.486
snapshot.1290	33	292	143	40.7	226991	85.43	733051	0.476	0.31
snapshot.1320	18	293	143	11.1	80210	13.72	166473	0.809	0.482
snapshot.1350	9	282	143	5.23	41382	5.6	54662	0.934	0.757
snapshot.1380	14	281	143	7.55	63766	8.63	98166	0.875	0.65
snapshot.1410	18	291	143	8.71	65916	9.98	113042	0.873	0.583

Name	$\mid I \mid$	$\mid R \mid$	$\mid U \mid$	T_{imp}	N_{imp}	T_{usu}	N_{usu}	$\frac{T_{imp}}{T_{usu}}$	$\frac{N_{imp}}{N_{usu}}$
snapshot.1440	12	292	143	5.62	34374	6.63	48744	0.848	0.705
snapshot.1470	12	293	143	5.17	35067	5.92	50668	0.873	0.692
snapshot.1500	14	276	144	5.36	47808	6.47	82762	0.828	0.578
snapshot.1530	9	273	144	3.87	30655	4.12	40840	0.939	0.751
snapshot.1560	9	273	144	3.12	31857	3.3	43769	0.945	0.728
snapshot.1590	9	245	144	2.26	24678	2.43	33712	0.93	0.732
snapshot.1620	9	247	144	2.31	24327	2.49	32662	0.928	0.745
snapshot.1650	14	241	143	4.22	47102	4.75	75390	0.888	0.625
snapshot.1680	9	207	143	1.47	19227	1.49	24727	0.987	0.778
snapshot.1710	9	204	141	1.65	22202	1.67	28319	0.988	0.784
snapshot.1740	9	175	139	1.23	20564	1.2	25261	1.025	0.814
snapshot.1770	9	171	137	1.05	16604	0.96	20996	1.094	0.791
snapshot.1800	9	158	136	1.01	17134	0.96	21304	1.052	0.804
snapshot.1830	9	151	132	0.73	12734	0.74	16674	0.986	0.764
snapshot.1860	9	142	127	0.9	15992	0.87	21011	1.034	0.761
snapshot.1890	14	135	122	1.31	23321	1.35	40421	0.97	0.577
snapshot.1920	14	129	119	1.63	33235	1.52	54125	1.072	0.614
snapshot.1950	20	126	111	4.02	75510	3.98	143581	1.01	0.526
snapshot.1980	27	140	108	7.72	138405	8.15	270539	0.947	0.512
snapshot.2010	20	136	96	5.53	81607	6.26	163178	0.883	0.5
snapshot.2040	27	133	92	14.07	161867	17.71	330388	0.794	0.49
snapshot.2070	54	135	90	85.08	617194	157.55	2138728	0.54	0.289
snapshot.2100	44	135	81	58.85	460058	91.59	1311138	0.643	0.351
snapshot.2130	54	135	78	117.82	976590	227.24	3519523	0.518	0.277
$\sum$				2045.36	9969988	5359.92	41474697	0.382	0.24

Table 3.1: The performance of improved pruning solving snapshots

It can be observed that in most of the cases, the computation took less time applying the new pruning scheme. Overall, it took only 38.2 % of the time and only 24.0 % of the nodes have been explored compared to the old pruning.

In the following we have a closer look on the single iterations and try to find out, on which parameters the improvement rate is dependent.

First, we investigated if the improvement rate is dependent on the maximal search depth of an iteration. Therefore, we took all iterations calculated for the computation of the snapshots presented in Table 3.1, computed there specific improvement rate with respect to the time, and the average and standard deviation of this improvement rate for all instances having the same maximal depth. This is illustrated in Figure 3.5.

Because it is possible that the good improvement rates are achieved on instances which did not take a lot of time anyway, one should consider the overall time spent on computations of a specific depth. For this reason, we show in Figure 3.6 the ratios of the sums of the time spent for the calculation of iterations having a specific maximal depth. In contrast to Figure 3.5 the iterations which took longer have a higher weight in Figure 3.6.

As one can see in both of the Figures 3.5 and 3.6, the improvement rate is highly dependent on the maximal depth of the search tree. As a consequence, for

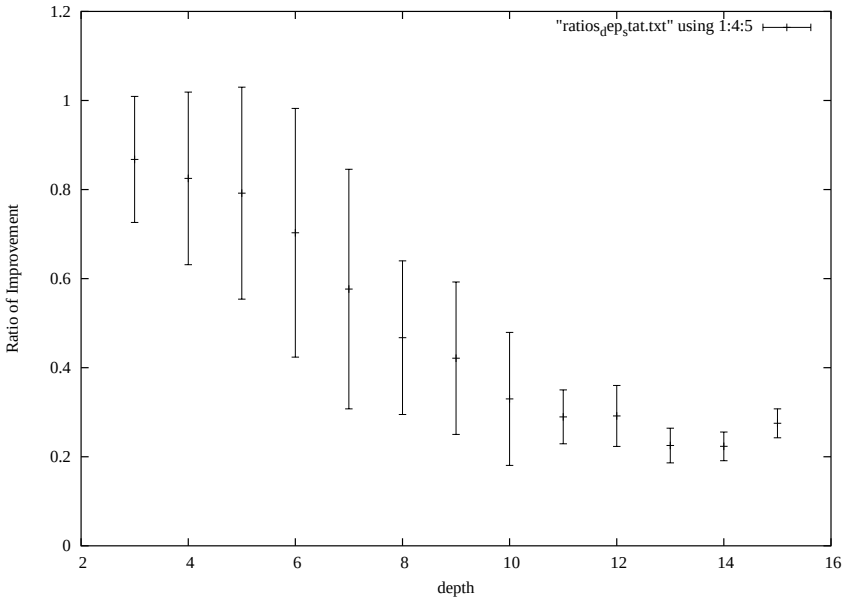

Figure 3.5: The ratio of the improvement resp. to the depth of the subtree

the computation of the snapshots it is reasonable to use the improved pruning for every depth.

Next, we examined the dependency of the improvement rate on the maximal search degree of an iteration. Therefore we took all iterations calculated for the computation of the snapshots presented in Table 3.1, computed their specific improvement rate with respect to the time, and the average and standard deviation of this improvement rate for all instances having the same maximal degree. This is illustrated in Figure 3.7.

Analogously to the maximal search depth of the search trees we calculated the ratio of the total time spent for the exploration of search trees of a specific maximal degree in Figure 3.8.

As a consequence of these two Figures 3.7 and 3.8 we should use the improved pruning scheme, for every search tree independent of its degree.

By testing the snapshots for their maximal depth and degree at the same time, the consequences formulated above are being confirmed. The average of the improvement rates for iterations having the same maximal degree and the same maximal depth are denoted in Table 3.2 and 3.9.

These two figures enforce the consequence denoted above that the improved pruning scheme does always accelerate the column generation when dispatching

	3	4	5	6	7	8	9	10	11	12	13	14	15
3	0.9294	0.9458	0.9111	–	–	–	–	–	–	–	–	–	–
4	0.9432	0.9455	0.9425	0.9194	–	–	–	–	–	–	–	–	–
5	0.8563	0.8718	0.907	0.8495	0.7374	–	–	–	–	–	–	–	–
6	0.8826	0.834	0.8258	0.8246	0.6957	0.5947	–	–	–	–	–	–	–
7	0.8679	0.7481	0.7629	0.66	0.6527	0.5773	0.5566	–	–	–	–	–	–
8	0.8056	0.7142	0.5644	0.4933	0.5366	0.4804	0.4299	0.3595	–	–	–	–	–
9	0.7495	0.5698	0.5244	0.4835	0.3652	0.3754	0.4797	0.3283	0.296	–	–	–	–
10	0.7191	0.5306	0.4675	0.3973	0.3813	0.3141	0.3625	0.3732	0.2983	0.3378	–	–	–
11	0.6991	0.5321	0.3884	0.3114	0.3337	0.3064	0.264	0.3121	0.3105	0.2611	0.2396	–	–
12	0.6532	0.4139	0.3831	0.2761	0.2237	0.2497	0.2584	0.1998	0.2358	0.2734	0.2047	0.2316	–
13	0.6591	0.4079	0.3522	0.2713	0.2198	0.2172	0.2568	0.2026	0.2008	0.2657	0.2045	0.2027	0.2752

Table 3.2: The ratio of the improvement resp. to depth (columns) and degree (rows) of the subtree

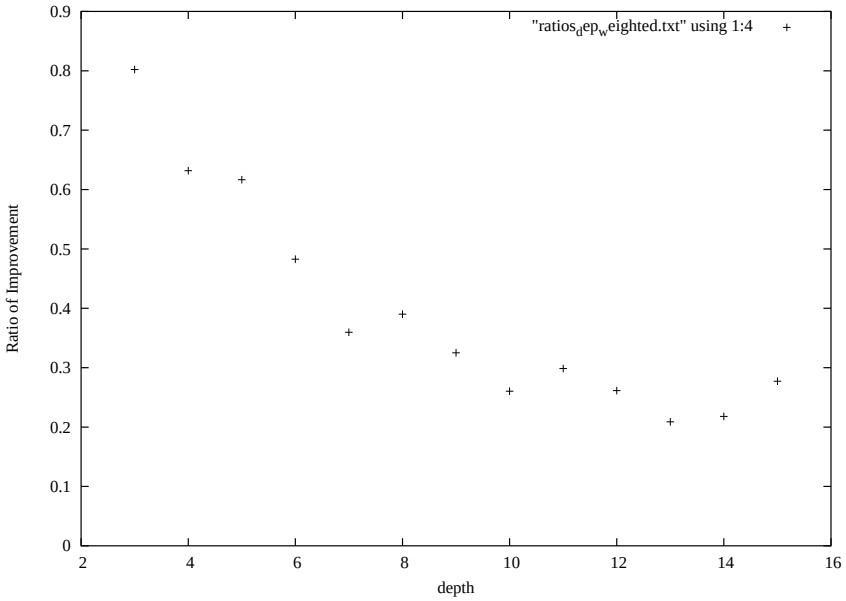

Figure 3.6: Time saved resp. to the depth of the subtree

snapshots. The performance ratio improves with increasing search degree and increasing search depth of the search tree.

3.4.2 Offline-Instances

Since the requests of these instances are not all already released, there will be less lateness costs arising. For this reason, we cannot hope for improvement rates as good as the ones achieved by the computation of the snapshots. We have three groups of offline instances which have to be solved. One of them has got a clairvoyance of one hour, the second one has a clairvoyance of two hours.

One-Hour Instances

Analogously to Table 3.1, Table 3.3 gives an overview of the One-Hour Instances and the performance of our pruning scheme. It can be seen that all in all only 50.1 % of the nodes had to be explored, but due to the different lateness structure this took 96.8 % of the time necessary for this exploration with the usual pruning method.

Like we did for the snapshots, we analyzed, if there are any dependencies of

Name	$\mid I \mid$	$\mid R \mid$	$\mid U \mid$	$\mid C \mid$	T_{imp}	N_{imp}	T_{usu}	N_{usu}	$\frac{T_{imp}}{T_{usu}}$	$\frac{N_{imp}}{N_{usu}}$
ZIBDIP_event_20010102	20	33	21	16	1.49	17048	1.06	28731	1.406	0.593
ZIBDIP_event_20010103	15	18	14	16	0.47	9359	0.31	11865	1.516	0.789
ZIBDIP_event_20010104	21	25	14	16	0.81	13919	0.66	20048	1.227	0.694
ZIBDIP_event_20010105	15	17	18	16	0.23	5448	0.14	6473	1.643	0.842
ZIBDIP_event_20010106	10	14	14	16	0.04	1337	0.02	1379	2	0.97
ZIBDIP_event_20010107	16	18	13	16	0.29	5530	0.2	7954	1.45	0.695
ZIBDIP_event_20010108	21	41	22	16	1.53	23168	1.26	33624	1.214	0.689
ZIBDIP_event_20010109	20	37	17	16	1.05	17534	0.77	22860	1.364	0.767
ZIBDIP_event_20010110	28	29	19	16	3.54	51438	2.9	82797	1.221	0.621
ZIBDIP_event_20010111	15	22	16	16	0.4	8600	0.32	11233	1.25	0.766
ZIBDIP_event_20010112	36	32	18	16	3.16	43794	2.73	75646	1.158	0.579
ZIBDIP_event_20010113	33	39	17	16	3.56	50399	2.96	75665	1.203	0.666
ZIBDIP_event_20010114	18	27	13	16	0.62	10435	0.49	14583	1.265	0.716
ZIBDIP_event_20010115	25	62	23	16	5.4	56677	4.85	110674	1.113	0.512
ZIBDIP_event_20010116	51	56	18	16	35.95	329475	40.26	691274	0.893	0.477
ZIBDIP_event_20010117	51	59	17	16	33.81	187683	40.99	695454	0.825	0.27
ZIBDIP_event_20010118	42	44	16	16	15.07	166801	15.48	318762	0.974	0.523
ZIBDIP_event_20010119	39	34	18	16	5.7	87537	5.05	127600	1.129	0.686
ZIBDIP_event_20010120	21	29	18	16	0.99	23691	0.84	28632	1.179	0.827
ZIBDIP_event_20010121	21	18	12	16	0.35	9212	0.26	10928	1.346	0.843
ZIBDIP_event_20010122	20	32	20	16	2.22	28157	1.72	42073	1.291	0.669
ZIBDIP_event_20010123	15	25	16	16	0.54	10223	0.4	13330	1.35	0.767
ZIBDIP_event_20010124	14	21	13	16	0.36	7122	0.32	9354	1.125	0.761
ZIBDIP_event_20010125	46	29	16	16	6.02	98227	5.16	144763	1.167	0.679
ZIBDIP_event_20010126	27	25	20	16	1.23	28266	0.98	36022	1.255	0.785
ZIBDIP_event_20010127	21	23	16	16	0.83	13744	0.59	19772	1.407	0.695
ZIBDIP_event_20010128	20	12	12	16	0.27	7120	0.17	8384	1.588	0.849
ZIBDIP_event_20010129	35	39	22	16	5.99	83512	5.5	142207	1.089	0.587
ZIBDIP_event_20010130	15	24	15	16	0.7	11784	0.58	16116	1.207	0.731
$\sum$					132.62	1407240	136.97	2808203	0.968	0.501

Table 3.3: One-Hour Instances: Overview

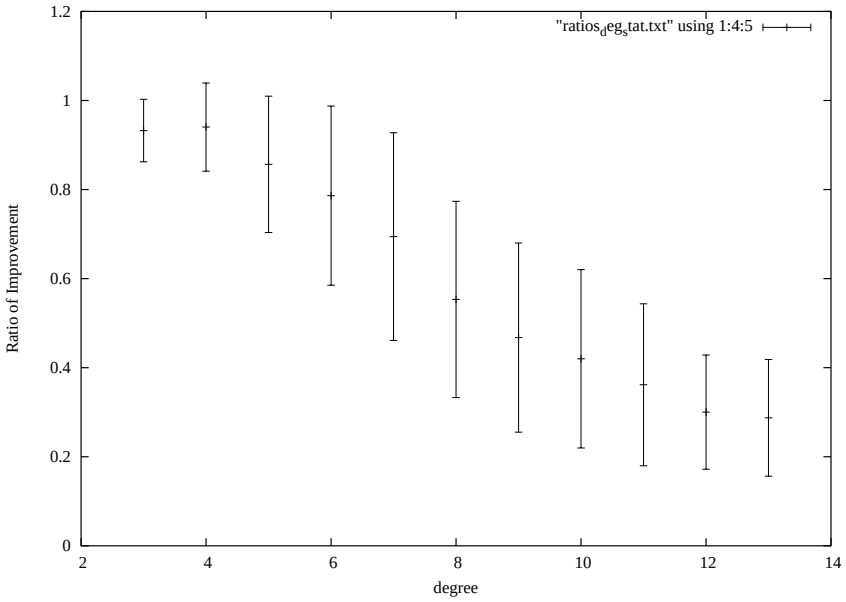

Figure 3.7: The ratio of the improvement resp. to the degree of the subtree

the performance ratio on maximal depth, maximal degree of the iterations and the system load.

In Figures 3.10 and 3.11 we computed the time spent for all iterations with a specific maximum depth resp. degree of a search tree.

According to these Figure, our improved pruning scheme does only make sense for computing One-Hour Instances, when the maximum depth and maximum degree is at least 8.

Figures 3.13 and 3.12 show the improvement rates by means of explored nodes, resp. time, against the system load.

Two-Hour Instances

Analogously to the One-Hour Instances, the Two-Hour Instances have been produced with a clairvoyance of two hours. Table 3.4 gives an overview over these instances and the improvement rates obtained by computing them. Making use of the improved pruning scheme, only 32.2 % of the nodes had to be explored which took 73.6 % of the time compared to the old pruning scheme.

In Figures 3.10 and 3.11 we computed the time spent for all iterations with a specific maximum depth resp. degree of a search tree.

Name	$\mid I \mid$	$\mid R \mid$	$\mid U \mid$	$\mid C \mid$	T_{imp}	N_{imp}	T_{usu}	N_{usu}	$\frac{T_{imp}}{T_{usu}}$	$\frac{N_{imp}}{N_{usu}}$
ZIBDIP_event_20010102	70	84	21	16	1366.58	6126478	1860.04	22050522	0.735	0.278
ZIBDIP_event_20010103	36	39	14	16	42	400297	31.76	696200	1.322	0.575
ZIBDIP_event_20010104	41	51	14	16	76.25	504879	69.79	1266365	1.093	0.399
ZIBDIP_event_20010105	27	33	18	16	8.83	94333	8.8	230183	1.003	0.41
ZIBDIP_event_20010106	26	31	14	16	4.49	59761	2.9	85955	1.548	0.695
ZIBDIP_event_20010107	32	42	13	16	19.72	134489	14.35	259775	1.374	0.518
ZIBDIP_event_20010108	41	71	22	16	53.73	365539	55.08	1086986	0.975	0.336
ZIBDIP_event_20010109	51	67	17	16	183.24	964987	202.09	2886333	0.907	0.334
ZIBDIP_event_20010110	34	54	19	16	42.46	338288	29.85	560005	1.422	0.604
ZIBDIP_event_20010111	49	55	16	16	122.81	640184	126.88	2098925	0.968	0.305
ZIBDIP_event_20010112	60	70	18	16	392.05	2377388	490.61	6771998	0.799	0.351
ZIBDIP_event_20010113	69	84	17	16	1472.47	6425000	2475.07	30369845	0.595	0.212
ZIBDIP_event_20010114	36	56	13	16	48.7	355569	42.45	651282	1.147	0.546
ZIBDIP_event_20010115	63	119	23	16	2170.9	8648216	2280.18	19543933	0.952	0.443
ZIBDIP_event_20010116	76	105	18	16	3801.9	13211887	5589.68	54999845	0.68	0.24
ZIBDIP_event_20010117	57	106	17	16	786.35	1977389	1370.6	13376384	0.574	0.148
ZIBDIP_event_20010118	58	69	16	16	288.66	1566170	265.57	3692891	1.087	0.424
ZIBDIP_event_20010119	130	67	18	16	6506.07	49166973	9212.36	138383990	0.706	0.355
ZIBDIP_event_20010120	41	61	18	16	154.8	1106023	144.52	2097268	1.071	0.527
ZIBDIP_event_20010121	32	44	12	16	17.57	139881	14.99	306876	1.172	0.456
ZIBDIP_event_20010122	41	68	20	16	282.48	1716425	279.23	3570392	1.012	0.481
ZIBDIP_event_20010123	51	59	16	16	252.03	1567465	207.31	3576398	1.216	0.438
ZIBDIP_event_20010124	22	42	13	16	5.92	60330	3.88	75360	1.526	0.801
ZIBDIP_event_20010125	55	47	16	16	160.34	1403528	134.37	2355104	1.193	0.596
ZIBDIP_event_20010126	35	45	20	16	33.97	351920	27.93	591724	1.216	0.595
ZIBDIP_event_20010127	51	58	16	16	161.93	1157805	183.13	3275419	0.884	0.353
ZIBDIP_event_20010128	25	25	12	16	4.02	57656	2.46	76591	1.634	0.753
ZIBDIP_event_20010129	42	63	22	16	79.34	667000	61.78	1081714	1.284	0.617
ZIBDIP_event_20010130	24	48	15	16	11.38	122590	8.09	174493	1.407	0.703
$\sum$					18550.99	101708450	25195.75	316192756	0.736	0.322

Table 3.4: Two-Hour Instances: Overview

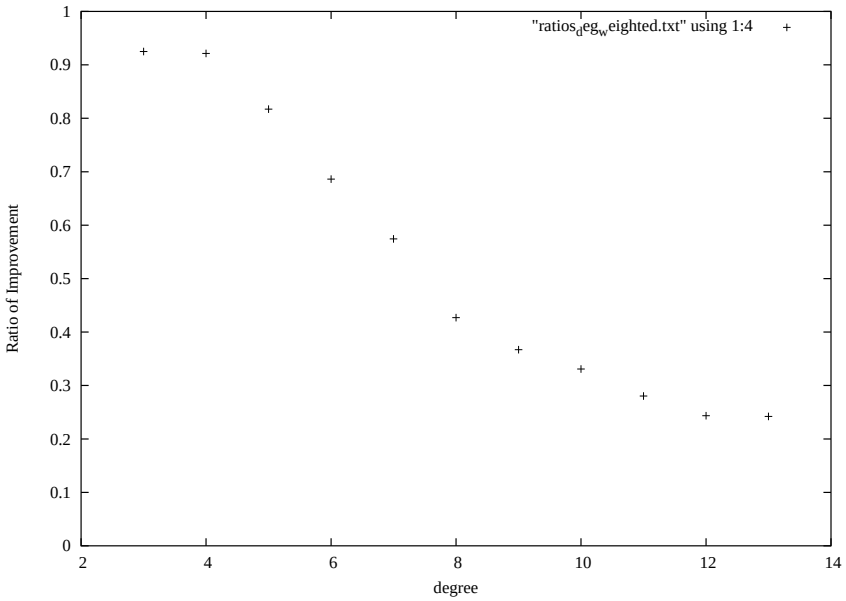

Figure 3.8: Time saved resp. to the degree of the subtree

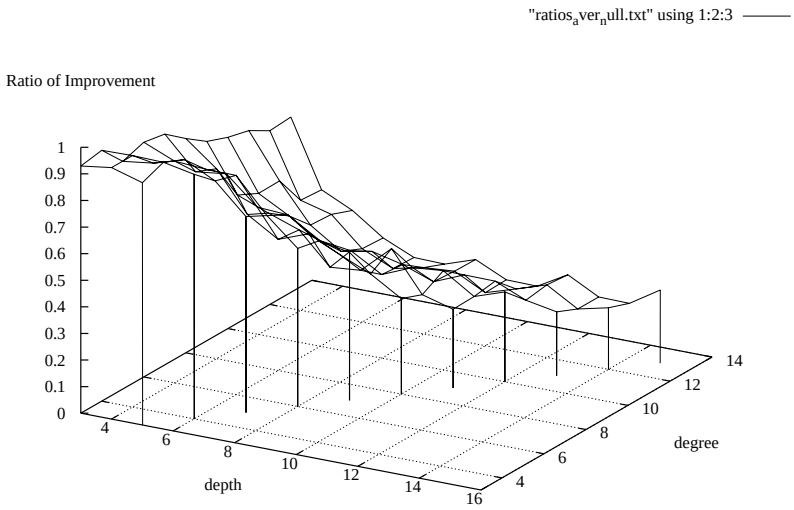

Figure 3.9: The ratio of the improvement resp. to depth (columns) and degree (rows) of the subtree

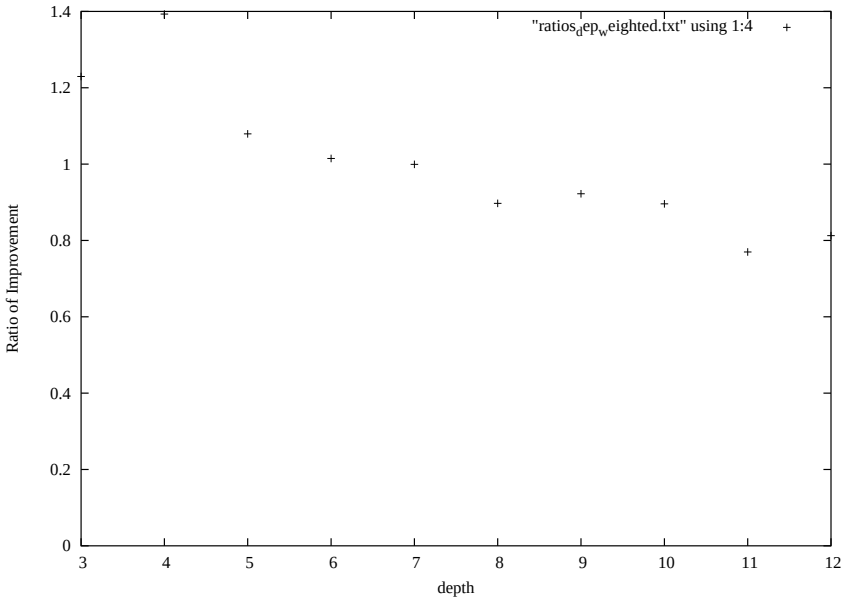

Figure 3.10: One-Hour Instances: Time saved resp. to the depth of the subtree

According to these Figure, our improved pruning scheme does only make sense for computing Two-Hour Instances, when the maximum depth is at least 8 and maximum degree is at least 7. Table 3.5 which is computed the same way as Table 3.2 parallels this observation in a good way.

Figures 3.16 and 3.17 show the improvement rates by means of explored nodes, resp. time, against the system load. In both of these figures can be observed that the improvement rates are getting better with higher system load.

3.4.3 Summary of the Computational Results

For every group of instances observed, the performance ratio due to the improved pruning scheme increased with increasing maximal depth and maximal degree of the search tree. Computing snapshots, it is always reasonable to utilize this advanced pruning scheme. The offline instances for the a-posteriori analysis can be solved in a better time if the maximal depth is at least 8 and if the maximal degree of the search tree is at least 8 for the case of the One-Hour instances and 7 for the case in of the Two-Hour instances.

Since our pruning scheme estimates the lateness costs more precisely at the cost of a higher computational demand, it has a good performance on instances with high lateness costs. For this reason the algorithm performed very well

	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
3	0.9183	1.1172	1.2046	–	–	–	–	–	–	–	–	–	–	–	–	–
4	2.5056	1.9333	1.5375	1.3337	–	–	–	–	–	–	–	–	–	–	–	–
5	2.25	1.4202	1.6148	1.4407	1.4245	–	–	–	–	–	–	–	–	–	–	–
6	1.4917	1.4217	1.5299	1.397	1.3755	1.2261	–	–	–	–	–	–	–	–	–	–
7	1.2827	1.5812	1.394	1.406	1.377	1.1649	1.0518	–	–	–	–	–	–	–	–	–
8	1.2639	1.4519	1.4846	1.3937	1.1576	1.0507	1.0536	0.9121	–	–	–	–	–	–	–	–
9	1.2757	1.4806	1.3913	1.291	1.2141	0.9908	0.9718	0.8572	0.8449	–	–	–	–	–	–	–
10	1.3429	1.3647	1.303	1.2345	1.0705	1.0326	0.799	0.8199	0.8629	0.738	–	–	–	–	–	–
11	1.1635	1.3534	1.1303	1.6029	1.1011	0.9956	1.3698	0.9001	0.9159	0.9892	0.8503	–	–	–	–	–
12	1.0333	1.5366	1.0219	0.9967	0.9493	0.8879	0.8605	0.6875	0.8671	0.8621	0.6492	0.8705	–	–	–	–
13	1.1053	1.0649	1.0549	1.2419	0.9137	0.8557	0.6926	0.8463	0.8408	0.6278	0.8433	0.8489	0.6093	–	–	–
14	1.0541	1.475	1.024	0.9997	0.9203	0.8577	0.8226	0.6358	0.8232	0.8198	0.5889	0.8248	0.8302	0.5752	–	–
15	1.05	1.4451	1.0214	0.9849	0.8817	0.8219	0.7873	0.602	0.7828	0.7816	0.5519	0.7835	0.79	0.538	0.7893	–

Table 3.5: Two-Hour Instances: The ratio of the improvement resp. to depth (columns) and degree (rows) of the subtree

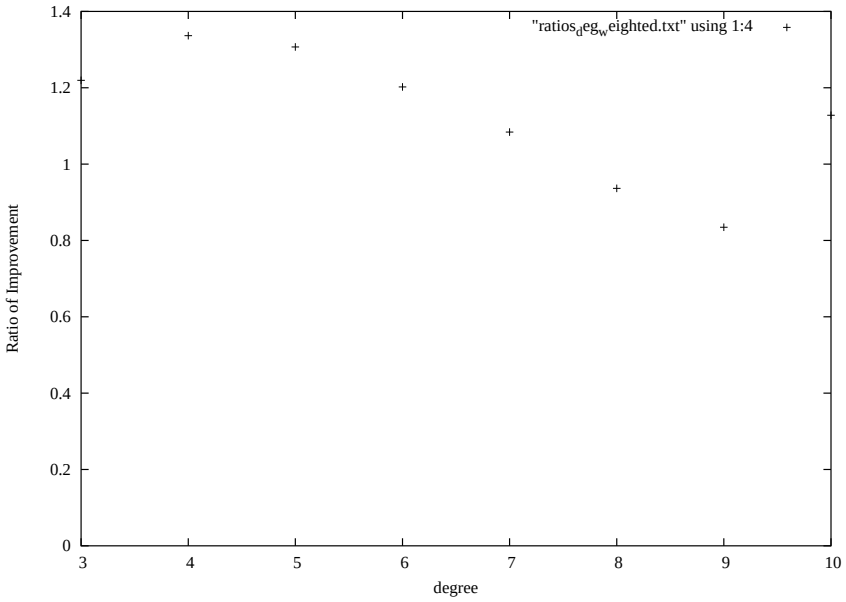

Figure 3.11: One-Hour Instances: Time saved resp. to the degree of the subtree

solving the snapshot problems. When it came to the One-Hour instances, which had a comparable system load like the snapshots, it was not possible, to solve them as efficiently as the snapshots because of their lower lateness costs due to the later release times of the requests. The Two-Hour instances could be solved in a much better way due to the longer tours as a result of a higher system load than the One-Hour instances.

By applying this pruning method to the offline instances, the number of explored nodes has been lowered to 50.1% for the case of the One-Hour instances and 32.2% for the case of the Two-Hour instances. But due to the longer computation time of this method, the performance ratio corresponding to the time reduction was not as good, so that only 3.2% resp. 26.4% of the could be saved by applying improved pruning.

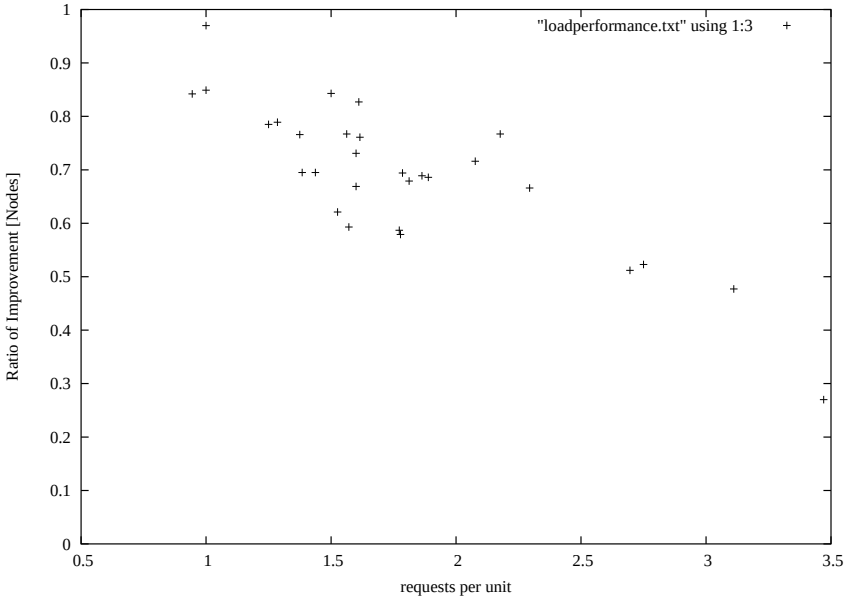

Figure 3.12: One-Hour Instances: performance (nodes) vs. systemload

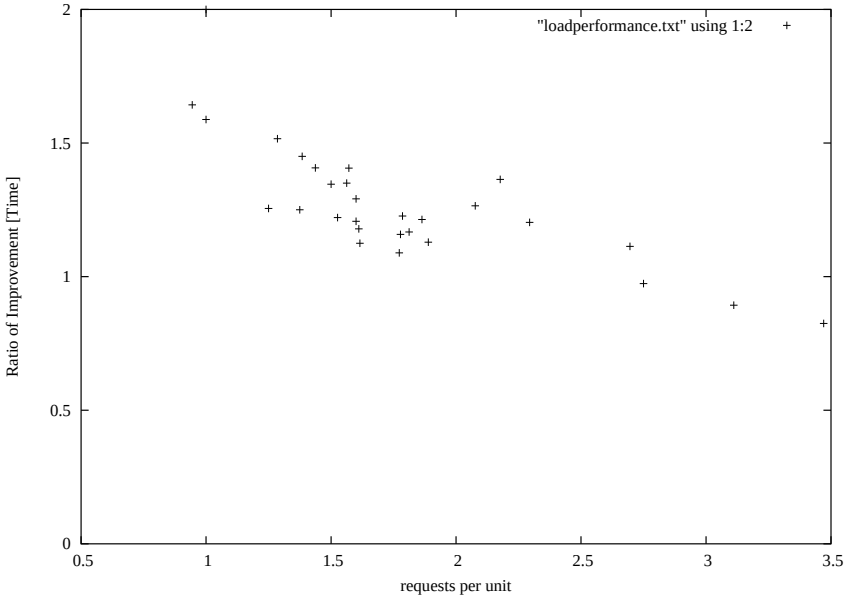

Figure 3.13: One-Hour Instances: performance (time) vs. systemload

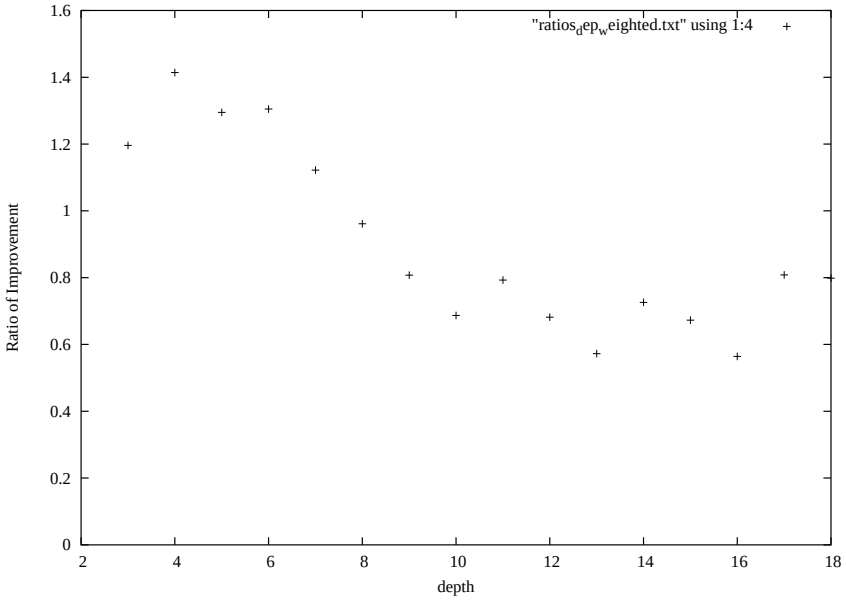

Figure 3.14: Two-Hour Instances: Time saved resp. to the depth of the subtree

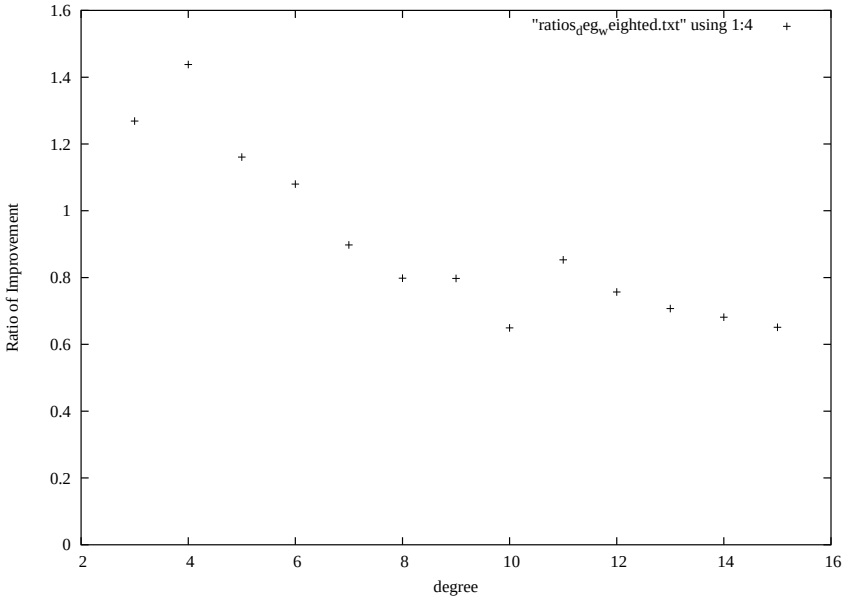

Figure 3.15: Two-Hour Instances: Time saved resp. to the degree of the subtree

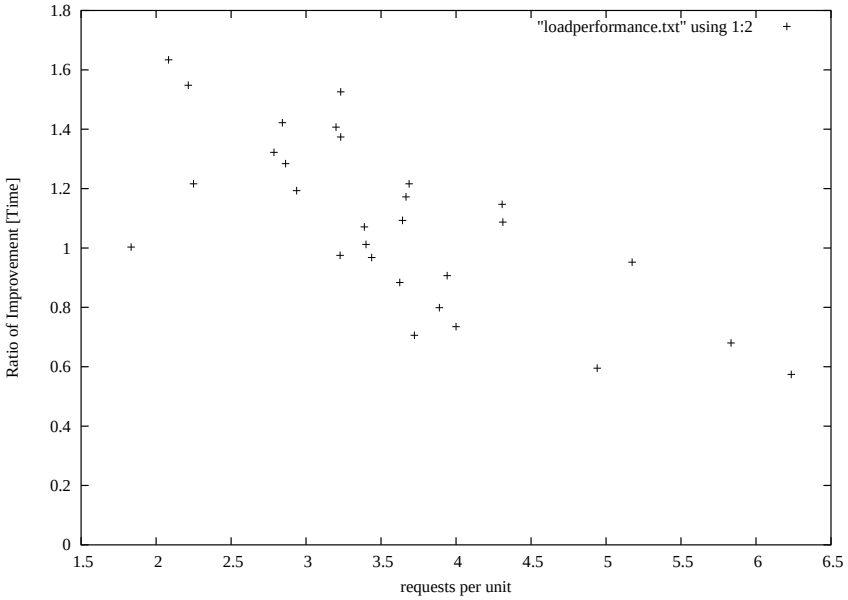

Figure 3.16: Two-Hour Instances: performance vs. systemload

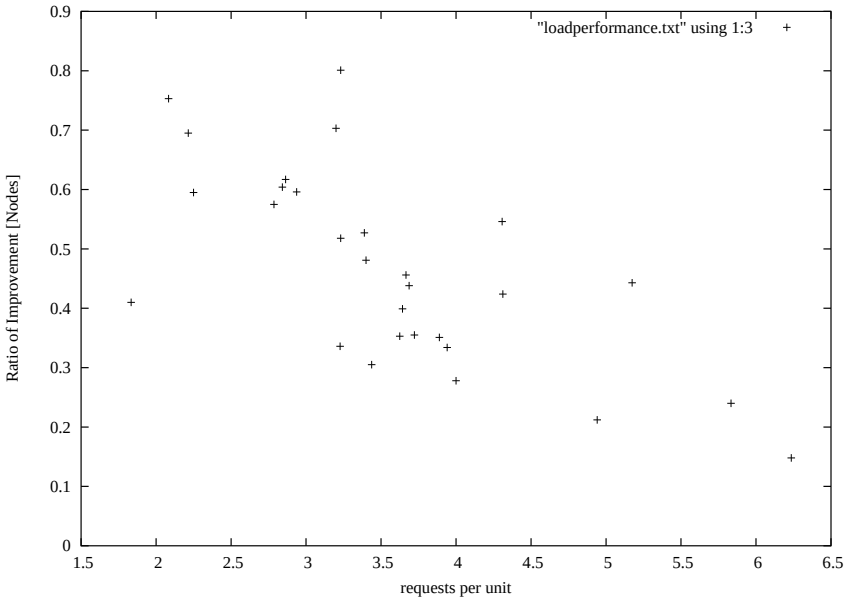

Figure 3.17: Two-Hour Instances: performance vs. systemload

Chapter 4

Column Generation via Resource Constrained Shortest Paths

As we mentioned in Chapter 2, the value of an optimal and feasible solution of RLP is a lower bound for the value of the optimal dispatch. In proposition 2.1 we showed a lower bound for the value of the optimal solution of RLP which can be computed by calculating for every unit u a lower bound for the reduced costs of a tour driven by u. In the remainder of this chapter we concentrate on finding such a lower bound.

4.1 The Problem

The problem of finding such a tour can be formulated as follows: Let $G = (V, A)$ be a graph where V is the set of Nodes $N \cup \{o_u, h_u\}$ with $N := \{e | F_e \subseteq F_u\}$ consisting of the nodes representing the events, which can be served by u on the way from the current position o_u to the home position h_u. A is the set of arcs $A := \{(i, j) \in V \times V | i \neq d_u, j \neq o_u\}$. A path in the graph G is defined as a sequence of nodes $e_0, e_1, \ldots, e_H$, such that each arc (e_{h-1}, e_h) belongs to A. All paths start at the origin ($e_0 = o_u$) and end at the destination ($e_H = h_u$). An increasing time depending cost function $c_{i,j}(t)$ is associated with each arc $(i, j) \in A$ which includes driving, service, lateness and overtime costs at node j and denotes the cost when i is being left for j at time t. This function is increasing in t and looks for all $i, j \in N \cup \{o_u\}$ as follows:

$$
\begin{aligned}
c_{i,j}(t) &:= \text{MinRedCost}(i, j, t) + c_u^{drv} \delta_u^{i,j} \\
c_{i,h_u}(t) &:= c_u^{ot} \max\{t + \delta_u^{i,h_u} - t_u^{end}, 0\} + c_u^{drv} \delta_u^{i,h_u} - \pi_u
\end{aligned}
$$

Let $d_{i,j}(t)$ be the time spent for serving request j directly after serving i, including driving from i to j, waiting and service at j when i is being left at time t. Because of the requests' release times and the resultant waiting times, $d_{i,j}(t)$ is not constant. Hence, for all $i, j \in V \cup \{o_u\}$:

$$d_{i,j}(t) := \max\{\delta_u^{i,j}, \theta_j^r - t\} + \delta_j$$
$$d_{i,h_u}(t) := \delta_u^{i,h_u}$$

The cost of a path is defined as the sum of the costs of its arcs and the duration is defined as the sum of their durations. The problem of finding the column with minimum reduced costs is equivalent to the problem of finding the minimum cost cycle-free path starting no earlier than t_u^{start} and ending no later than $t_u^{end} + t_u^{max-ot}$ in the graph previously described. In the following we will call this problem with regard to its origin MinColumn. Since the costs of the arcs are nondecreasing with respect to time, we can assume that there is always an optimal path starting exactly at t_u^{start}.

Definition 4.1 (Shortest Simple Path Problem with Arbitrary Weights) Let $G = (V', A')$ be a digraph with arc weights $c : A' \to \mathbb{R}$ and two vertices $s, t \in V'$. The task is to find a simple $s-t$-path of minimum weight or to decide that no such path exists.

Proposition 4.2 MinColumn *is a generalization of the* Shortest Simple Path Problem with Arbitrary Weights.

Proof. It suffices to show that every instance of the Shortest Simple Path Problem with Arbitrary Weights can be polynomially transformed into a MinColumn-problem. Consider an arbitrarily chosen problem of finding an $s-t-$path on a graph $G' = (V', A')$ with each arc $(i, j) \in A'$ having weights $c_{(i,j)}$. Since, a simple shortest path cannot contain parallel arcs, it suffices to consider the least cost arc of any two parallel arcs with the same direction.

The corresponding MinColumn-problem can be formulated as follows:

$$N := V' \setminus \{s, t\} o_u := sh_u := t$$

The shift start time t_u^{start} can be chosen arbitrarily. As the Shortest Simple Path Problem with Arbitrary Weights does not have any resource restriction, we choose t_u^{end}, t_u^{max-ot} and c_u^{ot} properly so that no resource restriction will apply.

$$\begin{aligned} t_u^{end} &= M \qquad \text{(big enough)} \\ t_u^{max-ot} &= 0 \\ c_u^{ot} &= 0 \end{aligned}$$

Since we want $c_{i,j}(t)$ to be constant over the time, we choose the requests' release and deadlines as the following so that no lateness costs will arise:

$$\begin{aligned} \theta_e^r &= t_u^{start} \quad \forall e \in N \\ \theta_e^d &= t_u^{end} \quad \forall e \in N \end{aligned}$$

The following parameters are set in a way that

$$\begin{aligned} c_u^{drv} &= 1 \\ \delta_e &= 0 \qquad\qquad\qquad \forall e \in N \\ \pi_u &= -\min_{a \in A'} c_a \\ \pi_e &= -\min_{a \in A'} c_a \quad \forall e \in N \end{aligned}$$

Since the MinColumn-problem is defined on a complete graph and G' does not have to be complete, we set the $\delta_u^{i,j}$ for all $(i,j) \notin A'$ on such a high value that the optimal path cannot include such an arc, if any $s-t-$path exists. We also have to take care that $\delta_u^{i,j} \geq 0$ for all $i,j \in N$, since the durations of the driving are defined to be positive.

$$\begin{aligned} \delta_u^{i,j} &= c_{(i,j)} - \min_{a \in A'} c_a \forall (i,j) \in A' \\ \delta_u^{i,j} &= (|A'|+1) \max_{a \in A'} c_a \forall (i,j) \notin A' \end{aligned}$$

As a result of the definitions just made, for all $i,j \in N$ holds:

$$\begin{aligned} c_{i,j}(t) &= \text{MinRedCost}(i,j,t) + c_u^{drv}\delta_u^{i,j} \\ &= \text{LateCost}(j, t+\delta_u^{ij}) + c_u^{svc}\delta_j - \pi_j + c_u^{drv}\delta_u^{i,j} \\ &= -\pi_j + \delta_u^{i,j} \\ &= \min_{a \in A} c_a + c_{(i,j)} - \min_{a \in A} c_a \\ &= c_{(i,j)} \end{aligned}$$

$$\begin{aligned} c_{i,h_u}(t) &= c_u^{ot} \max\{t + \delta_u^{i,h_u} - t_u^{end}, 0\} + c_u^{drv}\delta_u^{i,h_u} - \pi_u \\ &= \delta_u^{i,h_u} - \pi_u \\ &= c_{(i,h_u)} - \min_{a \in A} c_a + \min_{a \in A'} c_a \\ &= c_{(i,h_u)} \end{aligned}$$

If the optimal solution of this MinColumn problem has weight $\geq (|A'|+1)\max_{a \in A} c_a$, then no $s-t$-path exists. Otherwise, the value of the shortest $s-t-$path is equal to the value of the shortest path in the corresponding MinColumn-problem and the edges corresponding to the optimal solution of the $s-t-$path are the same as for the edges representing the optimal tour of the MinColumn-problem. □

Since the Shortest Simple Path Problem with Arbitrary Weights is $\mathcal{NP}$-hard (see [9]), it can be deduced that MinColumn is also $\mathcal{NP}$-hard and

therefore it is not reasonable that we find a polynomial algorithm for MinColumn.

As the MinColumn problem looks very similar to another problem called RCSP, which is well-investigated we will have a look at its definition, complexity and algorithmic approaches in order to find a way to solve MinColumn.

4.2 The Resource Constrained Shortest Path Problem

The resource constrained shortest path problem asks for the computation of a least cost path obeying a set of resource constraints. More precisely, we are given a graph $G = (V, E)$ with $|V| = n$ and $|E| = m$, a source node s and a target node t, and k resource limits $\lambda^{(1)}$ to $\lambda^{(k)}$. Each edge e has a cost c_e and uses r_e^i units of resource $i, 1 \leq i \leq k$. Costs and resources are assumed to be nonnegative. They are additive along paths. The goal is to find a least cost path from s to t that satisfies the resource constraints. The special case $k = 1$ is called the *single resource case* and the case $k > 1$ is called the *multiple resource case*. Since MinColumn is only restricted in the duration of the paths we will only focus on the single resource case.

4.2.1 Complexity

Handler and Zang [11] showed that every instance of the knapsack problem can be transformed into an instance of RCSP. In the following we show this transformation:

The knapsack problem reads as follows: We are given a set of $n - 1$ items, each having a value v_j and a weight w_j, for $j = 1, \ldots, n - 1$. The goal is to pack items into a knapsack so that the weight limit λ is not exceeded and so that the value of the chosen items is maximized. The knapsack problem is formally given as follows:

$$\text{(KS)} \qquad \max \sum_{j=1}^{n-1} v_j x_j$$
$$\text{s.t.} \sum_{j=1}^{n-1} w_j x_j \leq \lambda$$
$$x_j \in \{0,1\} \qquad j = 1, \ldots, n-1$$

where v_j, w_j, λ are positive integers. Now we set up a $n-$node network with two parallel arcs from every node j to node $j+1$ for $j = 1, \ldots, n-1$. Let $c^{(1)}_{j,j+1} = M - v_j, t^{(1)}_{j,j+1} = w_j$ and $c^{(2)}_{j,j+1} = M, t^{(2)}_{j,j+1} = 0$ be the parameters for the first and the second arcs for $j = 1, \ldots, n-1$, respectively, where $M = \max\{v_j : j = 1, \ldots, n-1\}$ (see Figure 4.1).

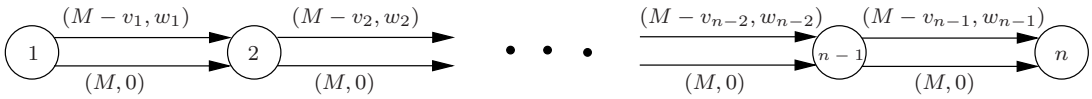

Figure 4.1: The Network corresponding to the knapsack problem

Then it is evident that KS may be solved by finding a shortest path (with respect to parameter c) from node 1 to node n, subject to a resource constraint (with respect to parameter t), with right hand side λ. Since every instance of the knapsack problem, which is $\mathcal{NP}$-hard, can be transformed into a RCSP problem in polynomial time it can be deduced RCSP is also $\mathcal{NP}$-hard.

4.2.2 Algorithmic Approaches

There are several approaches known for solving the RCSP.

Early work dealing with RCSP in the single resource case was done by Joksch [13] who presented a pseudopolynomial algorithm based on **dynamic programming** (see also Lawler [17]). Warburton [23] and Hassin [12] applied the standard technique of **rounding and scaling** to obtain fully polynomial ϵ-approximation scheme for RCSP. Lorenz and Raz [18] improved this algorithm to obtain a better running time.

RCSP can be solved exactly by **ranking paths** in nondecreasing cost order until a path obeying the resource limit has been found. This path constitutes the optimal solution. The asymptotically best algoritm for ranking the k best paths is due to Eppstein [8] and runs in time $O(m + n \log n + kn)$. According to

Handler and Zang [11] the path ranking method is known to perform badly in an experimental setting.

Another exact approach is to formulate the RCSP as a **0/1-integer program** as follows: We define 0-1 variables x_{ij} for edges $(i,j) \in E$:

$$x_{ij} := \begin{cases} 1 & \text{if edge } (i,j) \text{ belongs to an optimal path,} \\ 0 & \text{otherwise} \end{cases}$$

The integer linear program for RCSP is then given by

$$\text{(RCSPIP)} \qquad \min \sum_{i \in V} \sum_{j \in V} c_{ij} x_{ij} \tag{4.1}$$

subject to

$$\sum_{i \in V} \sum_{j \in V} r_{ij} x_{ij} \leq \lambda \tag{4.2}$$

$$\sum_{i \in V} x_{ij} = \sum_{i \in V} x_{ji} \qquad \forall j \in V \setminus \{s,t\} \tag{4.3}$$

$$\sum_{j \in V} x_{sj} = 1 \tag{4.4}$$

$$\sum_{i \in V} x_{it} = 1 \tag{4.5}$$

$$x_{ij} \in \{0,1\} \qquad \forall i,j \in V \tag{4.6}$$

Equation 4.2 ensures that the total resource consumption of the path satisfies the resource limit. Equation 4.3 is the degree constraint for each vertex of the graph (other than source and target) while 4.4 and 4.5 ensure that one edge leaves the source and one edge enters the target vertex.

Hence, RCSPIP is the usual shortest path formulation with an additional resource constraints. RCSPIP has $|E|$ variables and $|V|+1$ constraints.

Since the resource constraint turns the well studied, efficiently solvable shortest path problem into an $\mathcal{NP}$-hard one, there has been a variety of work relaxing this resource constraint by turning it into the objective function and solving scaled cost shortest path problems to obtain bounds for the original problem (see Aneja and Nair[3], Handler and Zang [11], Beasley, Christofides [4] for details)

Since the duration and the cost is not constant over time and we have no discretization of time, it is not possible to formulate an integer program like the

one above. As we are looking for a lower bound, it might be reasonable to replace $c_{i,j}(t)$ and $d_{i,j}(t)$ with constant functions $c'_{i,j}(t) \leq c_{i,j}(t)$ and $d'_{i,j}(t) \leq d_{i,j}(t)$. However, as this would mean neglecting the lateness and overtime costs which form the major part of the overall costs incurred, the lower bounds computed this way would be far too low.

As dynamic programming seems to be the only promising one of the previously described algorithmic approaches, we will have a look at labeling approaches, which can be seen as an improvement of the dynamic programming approaches:

4.2.3 Labeling Approaches

With each path P_i from the origin o_u to the node i is associated a two-dimensional (time, cost) label corresponding to the end of service at node i and the reduced costs of the path P_i. At node i, these labels will be denoted by

$$(T_i^k, C_i^k), i \in V, k \geq 1$$

to indicate the characteristics of the kth path from o_u to i. The indices k and i may be dropped if the context is unambiguous. These labels are calculated iteratively along the path $P_i = (i_0, i_1, i_2, \ldots, i_H)$ as:

$$\begin{aligned}(T_{i_0}, C_{i_0}) &= (t_u^{start}, 0)\\ (T_{i_h}, C_{i_h}) &= (T_{i_{h-1}} + d_{i_{h-1},i_h}(T_{i_{h-1}}), C_{i_{h-1}} + c_{i_{h-1},i_h}(T_{i_h-1}))\end{aligned}$$

where $i_0 = o_u$ and $i_H = i$.

Notation 4.3 Let (T_i^1, C_i^1) and (T_i^2, C_i^2) be two different labels for two paths from o_u to i. The first label *dominates* the second, i.e., $(T_i^1, C_i^1) \prec (T_i^2, C_i^2)$ if and only if $(T_i^2, C_i^2) - (T_i^1, C_i^1) \geq (0, 0)$.

Notation 4.4 A label (T_i, C_i) at a given node i is said to be *efficient* if no other label at i dominates it. A path from o_u to i is said to be efficient if the corresponding label is efficient.

Notation 4.5 For every $i \in V$, let Q_i be the set of labels at node i. Then, $\mathrm{EFF}(Q_i) := \{(T, C) \in Q_i | \nexists (T', C') \in Q_i : (T', C') \prec (T, C)\}$ denotes the *set of efficient labels* among the set of labels Q_i.

Since the reduced costs of the feasible paths, which end at a node $i, i \in V$ no later than t, can be defined as a decreasing step function of t with the nondominated labels as breakpoints we do not have to consider dominated labels. As an example look at Figure 4.2 where $P_1, \ldots, P_5$ are efficient and $P_6, \ldots, P_9$ are dominated labels.

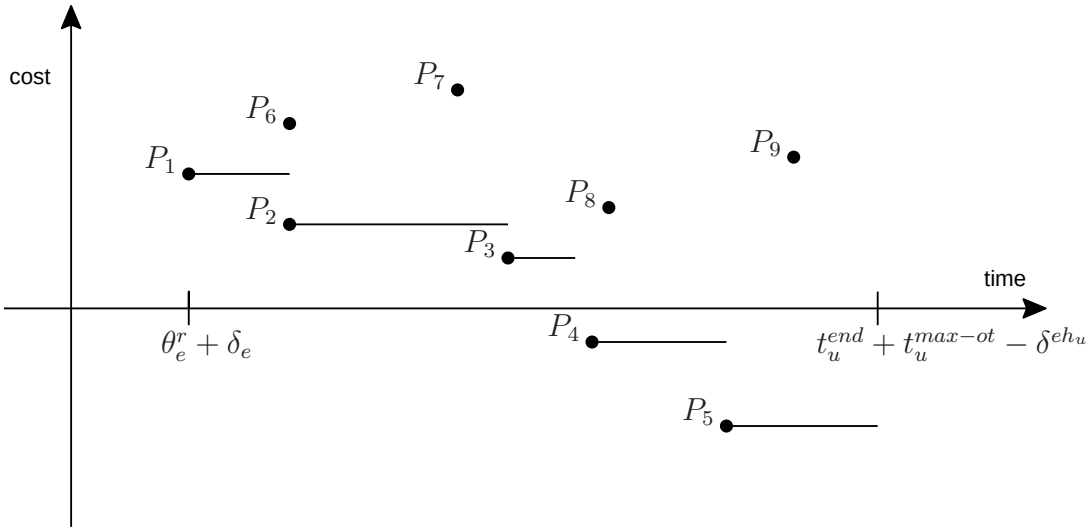

Figure 4.2: Dominance relation between labels associated with different paths

The set of efficient labels at each node can be computed by dynamic programming. The shortest path from o_u to h_u satisfying the time window constraints is obtained directly from the set $\mathrm{EFF}(Q_{h_u})$: it is represented by the least cost label.

A basic operation in shortest path algorithms is the *treatment* of a label (T_i^k, C_i^k). It consists of creating new labels at nodes $j \in N \cup \{h_u\} \setminus \{i\}$ by adding arcs (i, j) to the path from o_u to i associated with label (T_i^k, C_i^k). The new label for a given $j \in N \cup \{h_u\}$ is denoted as $f_{ij}(T_i^k, C_i^k)$ and is constructed as follows:

$$f_{ij}(T_i^k, C_i^k) := \begin{cases} (T_i^k + d_{i,j}(T_i^k), C_i^k + c_{i,j}(T_i^k)), & \text{if } T_i^k + d_{i,j}(T_i^k) \leq t_u^{end} + t_u^{max-ot} \\ \varnothing & otherwise \end{cases}$$

We introduce a permanent labeling algorithm. This algorithm treats the labels sequentially in increasing order of time. The positive duration of the arcs insures that once a label has been treated it is impossible to improve it any further. The algorithm uses sets P_i of permanent labels at node i. Each set P_i contains the labels of node i which have been previously treated. This algorithm can be described as follows:

Label Setting Algorithm for MinColumn

Input: a unit u, a set of requests E,
Output: the set of the efficient labels at h_u

// Initialization:
① $N = \{e \in E | F_e \subseteq F_u\}$;
② $V = N \cup \{o_u, h_u\}$;
③ $Q_{o_u} = \{(T_0^1 = t_u^{start}, C_0^1 = 0)\}; Q_i = \emptyset \quad \forall i \in N \cup \{h_u\}$;
④ $P_i = \emptyset, \forall i \in V$;
⑤ **If** $\bigcup_{i \in V}(Q_i \backslash P_i) = \emptyset$ **return** Q_{h_u}
// Selection of the next label to be treated:
⑥ Choose a label $(T_i^k, C_i^k) \in \bigcup_{i \in V}(Q_i \backslash P_i)$ with T_i^k minimal;
// Treatment of label (T_i^k, C_i^k)*:*
⑦ **For all** $j \in N \cup \{h_u\} \setminus \{i\}$:
⑧ $\quad Q_j := \mathrm{EFF}(f_{ij}(T_i^k, C_i^k) \cup Q_j)$;
⑨ $P_i := P_i \cup \{(T_i^k, C_i^k)\}$;
⑩ **Go to** ⑤

An algorithm like the one just presented has been used so far to calculate a lower bound for the reduced costs of a tour. In the following we explain its steps: ①, ②, ③ and ④ represent the initialization of the algorithm. In the beginning there is only one label at the home-position which is untreated. All other labels are to be constructed in the following. If $\bigcup_{i \in V}(Q_i \backslash P_i) = \emptyset$ in ⑤, all labels have already been treated and, therefore, all paths have been constructed if they were not dominated by other paths resp. labels. In ⑦ and ⑧ the current label is treated resp. the corresponding path is extended.

As all of the arcs have positive durations and every shift has an absolute end, the algorithm terminates.

The problem of the label setting algorithm is that cycles may occur. Since we are searching for the optimal path having no cycles, the algorithm described above solves a relaxation of our problem. Therefore, if the optimal label represents such a tour containing cycles, the cost value of this label represents a lower bound for the minimum reduced costs of a tour.

In fact, in practice cycles of a higher length than 3 rarely take place. If P contained a cycle entered at node j having more than 3 nodes, the vehicle would have to provide assistance in at least 4 cases, before returning to j again. As

the time of an assistance takes in practice at least 15 minutes, this would cause additional lateness costs due to at least 60 minutes.

As we will see in Proposition 4.7 an arc (i, j) is not part of an optimal path if i is left at time t or later and

$$\begin{aligned} 0 &< \text{MinRedCost}(i, j, t) \\ &= \text{LateCost}(j, t + \delta_u^{ij}) + c_u^{svc}\delta_j - \pi_j \\ \Rightarrow \quad \pi_j &< \text{LateCost}(j, t + \delta_u^{ij}) + c_u^{svc}\delta_j \end{aligned}$$

Since $t + \delta_u^{ij}$ is the arrival time t_P^j of P in j, P cannot be optimal if:

$$\pi_j < \text{LateCost}(j, t_P^j) + c_u^{svc}\delta_j$$

As the dual prices are usually not as high that lateness costs due to 60 minutes of lateness can be compensated, cycles containing more than 3 nodes rarely take place in practice. As most of the cycles are 2-cycles which are cycles of the form $(j \to i \to j)$, we describe in the following a method to avoid them. The Label Setting Algorithm for MinColumn can be adapted for 2-cycle elimination by keeping the best and the next best labels having a different predecessor at each node. The modification appears in the treatment of a node i for which $j \in N \cup \{h_u\} \setminus \{i\}$, and the best label in i is obtained by using the arc (j, i). The creation of a path with the cycle $(j \to i \to j)$ is avoided by adding the arc (i, j) to the second best label. It creates the best 2-cycle free path arriving at node j via node i. This leads to a doubling of the stored labels. We applied this method for 2-cycle elimination to our label setting algorithms, since it yields good results, but for reasons of legibility, we will not include it in the pseudocodes denoted on the following pages.

Speeding up the Label Setting Algorithm

There are several ways to improve the running time of the algorithm. We apply time buckets for the labels as they enable us to access the labels in shorter time. Since the speed of the algorithm is highly dependent on the amount of labels generated, we should think about canceling efficient labels in order to accelerate it further. We perceived two kinds of labels which may be dropped:

- Efficient but useless labels
- Efficient labels which are very similar to each other

Buckets This concept has been introduced by Denardo and Fox [6] in order to accelerate ⑥ in the label setting algorithm described on page 60. For classical shortest path problems, a bucket is a set of nodes whose label costs lie within a specified interval. For time constrained shortest path, a bucket contains those labels whose time lies within a specified interval.

Notation 4.6 (Time buckets) For a given bucketwidth w let

$$B_{k,i} := \{(T, C) \in Q_i | T \in [t_u^{start} + kw, t_u^{start} + (k+1)w)\}$$

be the kth bucket of node i.

If the bucketwidth $w \geq \delta_u^{i,j}$ for all $i, j \in V$, then no bucket can contain a label and its successor and the following modified algorithm yields the same results as the Label Setting Algorithm for MinColumn:

Label Setting Algorithm for MinColumn with buckets

Input: a unit u, a set of requests E, a bucketwidth w,
Output: the set of the efficient labels at h_u

// Initialization:
① $N = \{e \in E | F_e \subseteq F_u\}$;
② $V = N \cup \{o_u, h_u\}$;
③ $Q_{o_u} = \{(T_0^1 = t_u^{start}, C_0^1 = 0)\}; Q_i = \emptyset \quad \forall i \in N \cup \{h_u\}$;
④ $P_i = \emptyset, \forall i \in V$;
⑤ **If** $\bigcup_{i \in V}(Q_i \backslash P_i) = \emptyset$ **return** Q_{h_u}
// Selection of the next bucket to be treated:
⑥ Choose h with $\bigcup_{i \in V}(B_{h,i} \backslash P_i) \neq \emptyset$ and h minimal;
// Treatment of all labels in the chosen bucket:
⑦ **For all** $i \in V$ with $B_{h,i} \neq \emptyset$ **do**
⑧ **For all** $(T_i^k, C_i^k) \in B_{h,i}$ **do**
⑨ **For all** $j \in N \cup \{h_u\} \setminus \{i\}$ **do**
⑩ $Q_j := \mathrm{EFF}(f_{ij}(T_i^k, C_i^k) \cup Q_j)$;
⑪ $P_i := P_i \cup \{(T_i^k, C_i^k)\}$;
⑫ **Go to** ⑤

Instead of searching for the label with the smallest time value, they propose searching for the bucket $B_{h,i}$ containing this label in ⑥ and then treat all of the labels contained by this bucket at once (⑦ - ⑪).

Efficient but useless labels Even if notdominated, some labels cannot be part of an optimal path, since all of their successors have to be dominated. The following proposition gives a criteria for some of these labels:

Proposition 4.7 *Let i, j be two different requests. If* $\text{MinRedCost}(i,j,t) > 0$ *then (i,j) is not part of an optimal path if i is left at time t or later.*

Proof. Assume (i,j) was part of an optimal path P_1 with i being left at time t and k was the successor of j on P_1. Then there are three efficient labels (T_i, C_i), $(T_j, C_j) := (T_i + d_{i,j}(T_i), C_i + c_{i,j}(T_i))$,$(T_k^1, C_k^1) := (T_j + d_{j,k}(T_j), C_j + c_{j,k}(T_j))$ on the nodes i, j and k representing this part of P_1 with $T_i = t$. We put another label $(T_k^2, C_k^2) := (T_i + d_{i,k}(T_i), C_i + c_{i,k}(T_i))$ on k and show that it dominates (T_k^1, C_k^1) resp. that $T_k^1 - T_k^2 \geq 0$ and $C_k^1 - C_k^2 \geq 0$. Figure 4.3 illustrates these coherences.

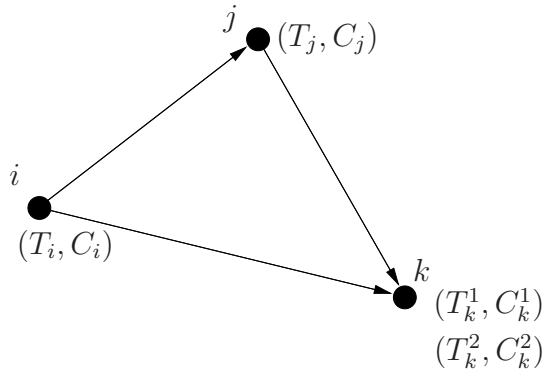

Figure 4.3: Requests and associated Labels for the proof of proposition 4.7

$$
\begin{aligned}
&T_k^1 - T_k^2 \\
&\quad = T_j + d_{j,k}(T_j) - (T_i + d_{i,k}(T_i)) \\
&\quad = T_i + d_{i,j}(T_i) + d_{j,k}(T_i + d_{i,j}(T_i)) - T_i - d_{i,k}(T_i) \\
&\quad = d_{i,j}(T_i) + d_{j,k}(T_i + d_{i,j}(T_i)) - d_{i,k}(T_i) \geq 0 \qquad \text{(triangle inequality)}
\end{aligned}
$$

The final inequation means that it takes at least as much time to drive from i to j and then to k than driving from i to k directly which is intuitively clear.

If $k = h_u$:

$$
\begin{aligned}
&C_k^1 - C_k^2 \\
&= C_j + c_{j,k}(T_j) - (C_i + c_{i,k}(T_i)) \\
&= C_i + c_{i,j}(T_i) + c_{j,k}(T_i + d_{i,j}(T_i)) - C_i - c_{i,k}(T_i) \\
&= c_{i,j}(T_i) + c_{j,k}(T_i + d_{i,j}(T_i)) - c_{i,k}(T_i)) \\
&= \text{MinRedCost}(i, j, T_i) + c_u^{drv}\delta_u^{i,j} \\
&\quad + c_u^{ot} \max\{T_i + d_{i,j}(T_i) + \delta_u^{j,h_u} - t_u^{end}, 0\} + c_u^{drv}\delta_u^{j,h_u} - \pi_u \\
&\quad - (c_u^{ot} \max\{T_i + \delta_u^{i,h_u} - t_u^{end}, 0\} + c_u^{drv}\delta_u^{i,h_u} - \pi_u) \\
&= \text{MinRedCost}(i, j, T_i) + c_u^{drv}(\delta_u^{i,j} + \delta_u^{j,h_u} - \delta_u^{i,h_u}) \\
&\quad + c_u^{ot}(\max\{T_i + \max\{\delta_u^{i,j}, \theta_j^d - T_i\} + \delta_u^{j,h_u} - t_u^{end}, 0\} - \max\{T_i + \delta_u^{i,h_u} - t_u^{end}, 0\}) \\
&> 0 + 0 + c_u^{ot}(\max\{T_i + \delta_u^{i,j} + \delta_u^{j,h_u} - t_u^{end}, 0\} - \max\{T_i + \delta_u^{i,h_u} - t_u^{end}, 0\}) \\
&\geq c_u^{ot}(\max\{T_i + \delta_u^{i,h_u} - t_u^{end}, 0\} - \max\{T_i + \delta_u^{i,h_u} - t_u^{end}, 0\}) \\
&= 0
\end{aligned}
$$

If $k \neq h_u$:

$$
\begin{aligned}
&C_k^1 - C_k^2 \\
&\quad = C_j + c_{j,k}(T_j) - (C_i + c_{i,k}(T_i)) \\
&\quad = C_i + c_{i,j}(T_i) + c_{j,k}(T_i + d_{i,j}(T_i)) - C_i - c_{i,k}(T_i) \\
&\quad = c_{i,j}(T_i) + c_{j,k}(T_i + d_{i,j}(T_i)) - c_{i,k}(T_i)) \\
&\quad = c_u^{drv}(\delta_u^{i,j} + \delta_u^{j,k} - \delta_u^{i,k}) \\
&\qquad + \text{MinRedCost}(i, j, T_i) + \text{MinRedCost}(j, k, T_j) - \text{MinRedCost}(i, k, T_i) \\
&\quad \geq \text{MinRedCost}(i, j, T_i) + \text{MinRedCost}(j, k, T_j) - \text{MinRedCost}(i, k, T_i) \\
&\quad \geq \text{MinRedCost}(j, k, T_j) - \text{MinRedCost}(i, k, T_i) \\
&\quad = \text{LateCost}(k, T_k^1) + c_u^{svc}\delta_k - \pi_k - (\text{LateCost}(k, T_k^2) + c_u^{svc}\delta_k - \pi_k) \\
&\quad = \text{LateCost}(k, T_k^1) - \text{LateCost}(k, T_k^2) \\
&\quad \geq 0 \quad (T_k^1 \geq T_k^2)
\end{aligned}
$$

Therefore, (T_k^1, C_k^1) is not efficient and $(i, j)_t$ cannot be part of an optimal path. □

Efficient labels which are very similar to each other A problem of the label setting algorithms just described is that it is possible that the amount of labels generated is exponential. In the literature it is reported that these algorithms work in pseudopolynomial time, but these running times have been

made under the assumption of a discretization of the time with the number of different time values as input size. As we have got no such discretization, a infinite number of efficient labels is thinkable and it might happen that all tours have to be enumerated.

Thus, we apply a slight change to the treatment of the buckets in order to decrease the number of the labels. In the following we introduce an artificial discretization of the time by the use of rounding which yields a much faster algorithm to the disadvantage of a little accuracy.

Let L be a set of efficient labels, which differ only slightly from another. According to the label setting algorithm previously described, all of these labels had to be treated. This does not seem to be useful since they do not differ very much from each other and, therefore, their successors will not differ from each other very much either. But since all of these labels may be part of an optimal path, we cannot simply drop one of them. This is why we introduce *artificial labels* which dominate efficient labels which are not very distinct from each other in order to decrease the number of generated labels. In order to avoid too many rounding errors, we set the cost and time values of the artificial label as high as possible.

Notation 4.8 Let L be a set of efficient labels. Then let

$$a(L) := (\min\{T \in \mathbb{R} | (T, C) \in L\}, \min\{C \in \mathbb{R} | (T, C) \in L\})$$

be the artificial label with the smallest time and cost values dominating every $l \in L$. Let the predecessor of $a(L) =: (T^1, C^1)$ be the predecessor of a label $(T^2, C^2) \in L$ with $C^1 = C^2$.

The predecessor of $a(L)$ is chosen this way, because the only value in which the labels of L can differ from each other very much is the cost, since their time values are similar to another. Therefore, the time value is the more important value of the both and the predecessor should be chosen according to the chosen cost value.

Example 4.9 For example let $l_1 := (44, -193)$ and $l_2 := (43, -192)$ be two labels on a node i. Since they are both efficient and very similar to each other, we create a third label $l_3 := (43, -193)$ which dominates both. Since we only treat efficient labels and l_1 and l_2 are not efficient anymore, we only have to treat l_3. As l_2 is the non-artificial label having least cost, l_3 has the same predecessor as l_2.

Since the time values of the labels contained by a bucket are very close together we take the buckets as such sets of very similar labels. When treating a

bucket $B_{h,i}$, we construct the artificial label $a(B_{h,i})$ which dominates all labels in the bucket and ,therefore, is the only one to be treated. Since there will only be one label treated per bucket it will have the same effect when we create an artificial label whenever we insert a label into a nonempty bucket. Thus, we can store the buckets as a list of labels and can access them in constant time.

The code for this algorithm which we will only call LSAMBAL(u, E, w) in the following reads as follows.

Label Setting algorithm for MinColumn with buckets and artificial labels

Input: a unit u, a set of requests E, a bucketwidth w,
Output: the set of the efficient labels at h_u

// Initialization:
① $N = \{e \in E | F_e \subseteq F_u\}$;
② $V = N \cup \{o_u, h_u\}$;
③ $maxbuckets := \lfloor \frac{t_u^{end} + t_u^{max-ot} - t_u^{start}}{w} \rfloor + 1$;
④ $Best(i) = \infty \quad \forall i \in V$;
⑤ $B_{h,i} = \emptyset \quad \forall i \in V, h = 0, \ldots, maxbuckets - 1$;
⑥ $B_{0,o_u} = \{(T_0^1 = t_u^{start}, C_0^1 = 0)\}$;
// Iterate all the buckets:
⑦ **For** $h = 0, \ldots, maxbuckets - 1$ **do**
⑧ **For all** $i \in V$ with $B_{h,i} \neq \emptyset$ **do**
⑨ Let (T_i, C_i) be the label contained by $B_{h,i}$;
// If the current label is dominated, continue
⑩ **If** $C_i \geq Best(i)$ **continue**;
⑪ $Best(i) := C_i$;
// Treatment of the current bucket:
⑫ **For all** $j \in N \cup \{h_u\} \setminus \{i\}$ **do**
// According to Proposition 4.7:
⑬ **If** MinRedCost$(i, j, T_i) > 0$ **continue**;
// Create new label:
⑭ $(T_j, C_j) := f_{ij}(T_i, C_i)$
// If the new label violates the time restriction, continue
⑮ **If** $T_j + \delta_u^{jh_u} > t_u^{end} + t_u^{max-ot}$ **continue**;
// Insert the new label into the appropriate bucket:
⑯ $l := \lfloor \frac{T_j - t_u^{start}}{w} \rfloor$;

⑰ **If** $B_{l,j} = \emptyset$ **then** $B_{l,j} := \{(T_j, C_j)\}$
⑱ **else**
⑲ Let (T'_j, C'_j) be the label contained by $B_{l,j}$;
⑳ $B_{l,j} := \{\min\{T_j, T'_j\}, \min\{C_j, C'_j\}\}$;
㉑ **return** $\bigcup_{k=0,\ldots,maxbuckets-1} B_{k,h_u}$;

In ③ the maximum number of buckets is computed. $Best(i)$ stores the smallest cost values of the labels at node i which have already been treated. Since these labels are exactly the labels with a smaller time value, the new label has to have less costs than those previously treated. If $C_i \geq Best(i)$, there would be a label (T''_i, C''_i) with $C''_i = Best(i) \leq C_i$ and $T''_i < T_i$ and accordingly $(T''_i, C''_i) \prec (T_i, C_i)$ and the next bucket can be observed ⑩. In ⑯ we compute the number of the bucket in which (T_j, C_j) has to be inserted. According to Proposition 4.7 (T_j, C_j) will not be part of an optimal path if $\text{MinRedCost}(i, j, T_i) > 0$ (⑬). Since there will never be more than one label in $B_{l,j}$ for every $l < maxbuckets$ and $j \in V$, (T'_j, C'_j) is chosen unambiguously in ⑲. If the condition in ⑮ is violated, it is impossible to extend the tour corresponding to (T_j, C_j) to a tour not violating the maximum overtime restriction. In ⑯ the number l of the bucket $B_{l,j}$ in which the new label (T_j, C_j) has to be inserted is computed. If this bucket is empty (T_j, C_j) is inserted (⑰). Otherwise the dominant label or if necessary an artificial label dominating both is inserted (⑳).

4.3 Column generation

In the previous section we developed a strategy to find a good lower bound for the reduced costs of a tour depending on the width of the used buckets. All the labels produced thereby represent tours. Since we need tours with negative reduced costs, we take a focus on the labels (T, C) with negative costs which will be called *promising labels* in the following.

Notation 4.10 (promising labels) Let

$$L := \{(T, C) \in B_{k,h_u} | k = 0, \ldots, maxbuckets - 1, C < 0\} \tag{4.7}$$

be the set of promising labels which correspond to tours with negative reduced costs if the corresponding paths are simple.

Notation 4.11 (corresponding tours) For every label l let $P(l)$ be the corresponding tour.

Notation 4.12 (promising labels having cycles) Let L be a set of labels, then let

$$L_c := \{l \in L | P(l) \text{ is not simple}\} \tag{4.8}$$

be the set of promising labels which correspond to tours with negative reduced costs if the corresponding paths are simple.

Column Generation via Resource Constrained Shortest Paths (RCCG)

Input: a set of units U, a set of requests E, a bucketwidth w,
a slope s for the bucketwidth function,
the value of the current optimal solution *opt*

Output: a lower bound *lb* for the value of LP,
a set of columns with negative reduced costs

① $lb := opt$
② **For all** $u \in U$ **do**
③ $L :=$ LSAMBAL(u, E, w)
④ $Best := \min_{(T,C)\in L} C$
⑤ $lb := lb + \max\{0, Best\}$
⑥ $P_c := \bigcup_{P\in L_c} \text{decycle}(P)$
⑦ $P_o := \bigcup_{P\in L\setminus L_c} P$
⑧ **return**$(lb, P_c \cup P_o)$

Let the function decycle(P) which is called in ⑥ be a function which constructs the cheapest simple path which can be produced by canceling requests which are served more than once by the path P or $\emptyset$ if the costs of all these simple paths ≥ 0.

4.3.1 How to choose the right bucketwidth

We developed a strategy to find a good lower bound for the reduced costs of a tour depending on the width of the used buckets. While calculating the first iterations of ZIBDIP the improvement of the LP solutions is relatively high, the gap is usually relatively high and the number of columns found is also high. Thus, we do not want to spend much time on the calculation of lower bounds and the creation of tours, since it is not likely that we already reached the desired gap. Later on, when the improvement rates become smaller, one should spend more time on the calculation of the lower bound, since the desired gap may be already reached and more precision is needed to find columns with negative reduced costs. Therefore, it is reasonable to define the bucketwidth as a decreasing function of the number of previously finished iterations. We tested linear functions with different y-intercepts from 15 to 30 and slopes from -0.5 to -3 per iteration. Since it led to the best results, we used for the following somputations a linear bucketwidth function with y-intercept 30 and slope -3. Thus, this function starts with a bucketwidth of 30, followed by an iteration having a bucketwidth of 27 and so on until the bucketwidth drops under 1. At this point it is fixed at 1.

How important the choice of the right function is, can be seen in Figure 4.4 where the different speeds of convergence of a linear function with y-intercept of 30 and slope of -0.5 is compared to a linear function with y-intercept of 15 and slope of -3. As expected, the smaller buckets lead to better lower bounds. In this case the algorithm could not benefit from the shorter running time of the algorithm due to the bigger buckets.

4.4 Computational results

For the test of the codes referenced in this chapter we used the same computers as for the computations done for the last chapter. We used the 3.06 Ghz Pentium 4 machine for the computations of the values in Table 4.1 and Table 4.3. For the computation of the results presented in Subsection 4.2 we used the 800 Mhz Pentium III machine. We used the same version of ZIBDIP as for the last chapter.

In order to test RCCG we observe the convergence speed of the solution of the LP for snapshots in Table 4.1 and for offline instances with 3 hours of clairvoyance in Table 4.2. After all we computed some offline instances with a clairvoyance of 5 hours which are denoted in Table 4.3.

Like the tables in the previous chapter, the first column of Table 4.1 contains the name of the instance, the second one the number of requests $|R|$ and the third

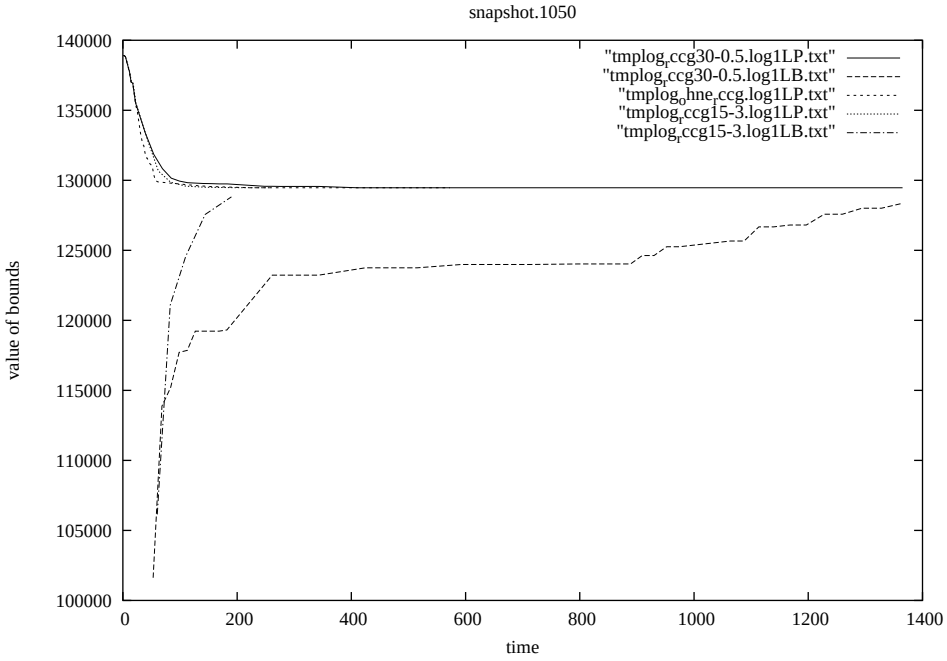

Figure 4.4: dispatching snapshot.1050 with different bucketwidth functions

one the number of units $|U|$. Columns 4-6 show the time in seconds which was needed by RCCG to find an LP-solution with a gap not worse than 10% resp. 5% resp. 1%. Analogously, the columns 7-9 show the times needed to reach this gap by the Branch and Bound column generation applying the improved pruning scheme introduced in Chapter 3.

Name			RCCG			CG via B & B		
	$\mid R \mid$	$\mid U \mid$	10%	5%	1%	10%	5%	1%
snapshot.240	123	100	0.11	0.11	0.11	0.11	0.11	0.11
snapshot.270	125	100	0.11	0.11	0.11	0.10	0.10	0.10
snapshot.300	142	100	0.13	0.13	1.03	0.13	0.13	1.02
snapshot.330	146	100	0.13	0.13	0.13	0.13	0.13	0.13
snapshot.360	156	100	0.16	0.16	0.16	0.15	0.15	0.15
snapshot.390	175	100	0.18	0.18	0.18	0.18	0.18	0.18
snapshot.420	183	100	0.19	0.19	1.53	0.19	0.19	1.52
snapshot.450	200	100	0.22	0.22	0.22	0.22	0.22	0.22
snapshot.480	203	100	0.23	0.23	0.23	0.22	0.22	0.22
snapshot.510	215	100	0.26	0.26	1.32	0.26	0.26	1.33
snapshot.540	216	113	0.26	0.26	9.98	0.25	0.25	9.89
snapshot.570	218	113	0.27	0.27	0.27	0.26	0.26	0.26
snapshot.600	237	113	0.30	0.30	21.56	0.30	0.30	14.78
snapshot.630	250	113	0.32	0.32	44.08	0.32	0.32	33.85
snapshot.660	247	113	0.31	0.31	15.06	0.32	0.32	14.92

Name			RCCG			CG via B & B		
	$\mid R \mid$	$\mid U \mid$	10%	5%	1%	10%	5%	1%
snapshot.690	261	113	0.35	0.35	39.50	0.35	0.35	40.20
snapshot.720	269	113	0.41	0.41	42.86	0.41	0.41	17.81
snapshot.750	278	113	0.44	3.30	100.99	0.43	3.29	98.08
snapshot.780	291	113	0.43	4.15	42.38	0.44	4.16	28.14
snapshot.810	290	113	0.48	0.48	77.09	0.48	0.48	97.70
snapshot.840	296	113	0.44	5.40	48.62	0.46	5.41	20.77
snapshot.870	299	113	0.46	11.13	55.98	0.46	11.07	21.48
snapshot.900	308	118	0.52	8.63	56.22	0.52	8.65	22.34
snapshot.930	307	118	0.50	7.49	46.76	0.51	7.59	25.33
snapshot.960	306	118	0.51	11.44	54.86	0.50	11.47	37.09
snapshot.990	311	118	0.47	0.47	68.51	0.46	0.46	57.97
snapshot.1020	314	135	0.49	3.26	47.90	0.48	3.24	20.54
snapshot.1050	322	135	0.53	19.81	64.09	0.53	12.92	31.27
snapshot.1080	311	135	0.48	12.99	106.19	0.48	12.95	99.54
snapshot.1110	316	143	0.49	2.25	52.81	0.48	2.23	24.21
snapshot.1140	314	143	0.48	4.14	38.72	0.48	4.10	18.05
snapshot.1170	318	143	0.50	4.79	56.34	0.50	4.82	23.43
snapshot.1200	308	143	0.47	5.27	43.81	0.47	5.28	51.65
snapshot.1230	304	143	0.47	0.47	60.20	0.45	0.45	54.10
snapshot.1260	302	143	0.44	2.03	33.25	0.44	2.04	16.47
snapshot.1290	292	143	0.43	0.43	45.44	0.42	0.42	40.67
snapshot.1320	293	143	0.42	0.42	34.10	0.42	0.42	36.01
snapshot.1350	282	143	0.40	0.40	20.71	0.41	0.41	18.79
snapshot.1380	281	143	0.39	0.39	15.61	0.40	0.40	15.54
snapshot.1410	291	143	0.43	0.43	17.81	0.43	0.43	17.54
snapshot.1440	292	143	0.42	0.42	2.52	0.41	0.41	2.50
snapshot.1470	293	143	0.41	0.41	2.68	0.41	0.41	2.68
snapshot.1500	276	144	0.38	0.38	2.67	0.38	0.38	2.68
snapshot.1530	273	144	0.37	0.37	1.20	0.37	0.37	1.21
snapshot.1560	273	144	0.37	0.37	5.76	0.37	0.37	5.74
snapshot.1590	245	144	0.30	0.30	0.30	0.30	0.30	0.30
snapshot.1620	247	144	0.31	0.31	0.31	0.31	0.31	0.31
snapshot.1650	241	143	0.30	0.30	0.30	0.30	0.30	0.30
snapshot.1680	207	143	0.24	0.24	0.24	0.24	0.24	0.24
snapshot.1710	204	141	0.23	0.23	0.23	0.23	0.23	0.23
snapshot.1740	175	139	0.39	0.39	1.58	0.18	0.18	1.59
snapshot.1770	171	137	0.18	0.18	0.18	0.19	0.19	0.19
snapshot.1800	158	136	0.16	0.16	0.87	0.16	0.16	0.85
snapshot.1830	151	132	0.14	0.14	0.14	0.15	0.15	0.15
snapshot.1860	142	127	0.12	0.12	0.75	0.13	0.13	0.75
snapshot.1890	135	122	0.12	0.12	0.12	0.12	0.12	0.12
snapshot.1920	129	119	0.11	0.11	1.36	0.11	0.11	1.30
snapshot.1950	126	111	0.10	0.10	0.95	0.10	0.10	0.93
snapshot.1980	140	108	0.12	0.12	3.37	0.12	0.12	3.61
snapshot.2010	136	96	0.11	0.11	0.35	0.11	0.11	0.33
snapshot.2040	133	92	0.11	0.11	2.88	0.11	0.11	3.26
snapshot.2070	135	90	0.11	0.11	2.35	0.11	0.11	2.33

Name			RCCG			CG via B & B		
	$\mid R \mid$	$\mid U \mid$	10%	5%	1%	10%	5%	1%
snapshot.2100	135	81	0.10	0.10	3.16	0.10	0.10	5.61
snapshot.2130	135	78	0.10	0.10	5.75	0.09	0.09	5.03

Table 4.1: Convergence Speed of RCCG vs. B&B CG on snapshots

As one can see in Table 4.1, applying RCCG instead of the B&B column generation did not lead to an improvement. In fact, it performed worse solving the instances with higher load (snapshot.600 - snapshot.1290).

Opposed to the snapshots presented in Table 4.1, the offline instances shown in Table 4.2 had to be solved by using 16 contractors V . Thus, the Table 4.2 contains the same columns as Table 4.1 with the only difference that the number of contractors is denoted in the fourth column.

Name				RCCG			CG via B & B		
	$\mid R \mid$	$\mid U \mid$	$\mid C \mid$	10%	5%	1%	10%	5%	1%
ZIBDIP_event_20010102.txt	133	25	16	163.96	185.62	267.53	471.24	1588.26	32098.72
ZIBDIP_event_20010103.txt	74	19	16	20.37	24.46	35.94	60.81	875.27	5779.32
ZIBDIP_event_20010104.txt	80	18	16	17.28	25.20	28.94	138.55	155.76	1143.44
ZIBDIP_event_20010105.txt	56	22	16	6.72	11.66	15.96	5.46	10.45	33.49
ZIBDIP_event_20010106.txt	45	16	16	1.85	2.69	5.52	1.08	2.02	7.95
ZIBDIP_event_20010107.txt	68	15	16	12.13	17.01	24.99	22.14	109.67	185.97
ZIBDIP_event_20010109.txt	99	21	16	34.40	49.01	63.26	78.51	245.79	1460.33
ZIBDIP_event_20010110.txt	71	23	16	3.84	9.71	19.34	17.69	55.61	926.38
ZIBDIP_event_20010111.txt	81	19	16	20.26	28.24	36.90	30.13	212.90	1537.50
ZIBDIP_event_20010112.txt	108	21	16	47.81	68.23	88.15	534.75	1121.23	7707.88
ZIBDIP_event_20010113.txt	129	20	16	101.41	118.51	150.35	3316.43	3374.32	34808.34
ZIBDIP_event_20010114.txt	102	16	16	64.77	73.74	91.21	1175.44	4701.45	23132.78

Table 4.2: Convergence Speed of RCCG vs. B&B CG on 3 Hour Instances

RCCG performs much better solving these offline instances than the B&B column generation. Because of the high amount of time necessary to solve these instances to a gap of 1%, we did not compute more of such instances. Moreover, the difference in the convergence speed was so evident that no more tests have been made.

RCCG solved the instances in a small fraction of the time necessary for the B&B column generation scheme.

This is because of the fact that the Branch and Bound column generation is designed for online instances which have an average tour length not greater than 5 in most of the cases. Furthermore, due to the high lateness costs incurring, a lot of pruning can be done so that only a fraction of the nodes of the search trees have to be explored. These specific characteristics cannot be applied to the offline problems, since the search trees are very big because of the greater

average tour length. Furthermore, the portion of the nodes which have to be examined is much higher, as we cannot prune as effective as for the snapshots due to the smaller lateness costs. One could think that for this case, not applying the improved pruning scheme could yield better results, since it is computational more demanding and cannot benefit from high lateness costs by solving offline instances. We tried so but received even much worse results without improved pruning.

Another problem is the greediness of the B&B column generation scheme: Only the most promising chlidren of a node within a search tree are examined which runs the risk of missing important tours.

Because of the achievements made by solving 3-Hour instances, we tried to solve 5-Hour instances (see Table 4.3). This table is not meant to illustrate a comparison of the two column generation approaches like the tables before, but for to show that by now with this new approach instances with a lot higher clairvoyance can be solved in reasonable time.

Name				RCCG		
	$\mid R \mid$	$\mid U \mid$	$\mid C \mid$	10%	5%	1%
ZIBDIP_event_20010102	205	27	16	682.57	791.37	1103.38
ZIBDIP_event_20010103	139	21	16	112.32	154.43	198.44
ZIBDIP_event_20010104	126	21	16	62.41	85.34	132.30
ZIBDIP_event_20010105	95	24	16	31.71	39.96	88.75
ZIBDIP_event_20010106	82	19	16	12.56	18.80	28.59
ZIBDIP_event_20010107	112	19	16	69.90	76.66	109.77
ZIBDIP_event_20010108	180	30	16	218.97	281.20	384.29
ZIBDIP_event_20010109	145	24	16	72.76	108.54	155.21
ZIBDIP_event_20010110	123	25	16	39.97	51.35	68.58
ZIBDIP_event_20010111	133	22	16	70.27	99.45	143.69
ZIBDIP_event_20010112	173	24	16	275.72	366.87	717.27
ZIBDIP_event_20010113	230	23	16	679.18	797.61	3513.56
ZIBDIP_event_20010114	191	20	16	395.29	478.50	3539.63
ZIBDIP_event_20010115	275	30	16	2136.36	2532.63	4354.99
ZIBDIP_event_20010116	247	26	16	826.62	1022.81	6868.20
ZIBDIP_event_20010117	213	23	16	229.42	353.01	647.51
ZIBDIP_event_20010118	149	23	16	109.39	125.01	161.16
ZIBDIP_event_20010119	172	25	16	231.74	274.12	354.73
ZIBDIP_event_20010120	152	23	16	93.00	124.75	175.90
ZIBDIP_event_20010121	114	18	16	26.25	32.56	48.09
ZIBDIP_event_20010122	177	27	16	265.35	348.81	435.58
ZIBDIP_event_20010123	131	22	16	52.80	71.16	108.46
ZIBDIP_event_20010124	102	17	16	16.52	22.78	30.85
ZIBDIP_event_20010125	112	21	16	20.78	30.69	42.85
ZIBDIP_event_20010126	122	25	16	40.19	49.39	71.99
ZIBDIP_event_20010127	141	20	16	83.05	104.37	187.95
ZIBDIP_event_20010128	92	16	16	23.87	28.51	41.36
ZIBDIP_event_20010129	136	28	16	32.99	44.44	76.18

Table 4.3: Convergence Speed of RCCG on 5 Hour Instances

Appendix A

Notation

In this chapter, we outline based on [14] some basic definitions used throughout this thesis. We describe some graph theory, network flows, the Branch-And-Bound Method and some basic definitions considering linear and integer programming. We assume some basic knowledge of linear optimization. For these fundamentals the reader is referred to [5], and especially regarding integer programming [21] and [19].

A.1 Basic notation

We will denote by $\mathbb{R}$ ($\mathbb{Q}$, $\mathbb{Z}$) the real (rational, integer) numbers. The sets $\mathbb{R}_+$ ($\mathbb{Q}_+$, $\mathbb{Z}_+$) stand for the non-negative real (rational, integer) numbers. We denote the set of positive integer numbers without zero by $\mathbb{N} = \mathbb{Z}_+ \backslash \{0\}$. For some $n \in \mathbb{N}$, we define by $\mathbb{K}^n$ the set of vectors with n components from $\mathbb{K}$. The transposition of a vector x is x^T.

A.2 Graph Theory

Formally, an *(undirected) graph* is a triple $G = (V, E, \Psi_1)$ consisting of a nonempty set V, called the *nodes* (or vertices), a set E, called the *edges* (or links), and a relation of *incidence* $\Psi_1 : E \to V \times V$ that associates with each edge two nodes, called its *ends*. Usually we just write $G = (V, E)$ and assume that the incidence relation is given implicitly in E. For each edge $e \in E$ there exist nodes $u, v \in V$ such that $\Psi_1(e) = \{u, v\} = \{v, u\}$. Two nodes that are ends of an edge are *adjacent* to one another (*neighbors*). The *degree* $|\delta(v)|$ of a node

v is the number of incident edges to v. An edge with identical ends is called a *loop*. If two edges join the same pair of ends, they are called *parallel*. A graph is *simple* if it has neither loops nor parallel edges. For a subset $W \subseteq V$ of nodes, $E(W) \subseteq E$ denotes the subset of edges with both ends in W.

A *digraph* (directed graph) is a triple $G = (V, A, \Psi_1)$ consisting of a nonempty set V, called the *nodes* (or vertices), a set A, called the *arcs*, and a relation of *incidence* $\Psi_1 : A \to V \times V$ that associates with each arc an ordered pair of nodes, called its *ends*. Usually, we just write $D = (V, A)$ and assume that the incidence relation is given implicitly in A. For each arc $a = (u, v)$ we call u the *source* and v the *target* of a. Parallel arcs and loops are defined as for undirected graphs. Two arcs (u, v) and (v, u) are called *associated*. For an arc (u, v), the arc (v, u) is called its *associated backward arc*. A digraph where each arc has its associated backward arc is called *bidirectional*. We call the graph $G = (V, E)$ the *underlying graph* of the digraph $D = (V, A)$ if there is a bijection between the arcs of D and the edges of G, such that for each arc $a = (u, v) \in A$ there is an edge $e = \{u, v\} \in E$, and for each edge $e = \{u, v\} \in E$ the arc $a = (u, v)$ and the arc $a' = (v, u)$ are in A. The *overlaying digraph* $D(G)$ of a graph G is the digraph obtained from G by replacing each edge by two associated arcs with the same ends.

In the following, let $G = (V, E)$ denote a graph and $D = (V, A)$ a digraph.

A *path* P in G (or a directed path in D) from v_0 to v_l is a sequence $P = (v_0, e_1, v_1, ..., e_l, v_l)$ of nodes $v_0, ..., v_l \in V$ and edges (arcs) $e_1, ..., e_l \in V (\in A)$ of G (D), such that the nodes v_{i-1} and v_i are the ends of edge e_i (are source and target of e_i) for each $1 \leq i \leq l$. Node v_0 is called the *source* and v_l the *target* of P, while both are denoted as the *endnodes* of P. The nodes $v_1, ..., v_{l-1}$ are called the *inner nodes* of P. The *length* of a path is the number of edges (arcs). We use the notation $e \in P$ ($a \in P$) or $v \in P$, if $e \in E$ ($a \in A$) is an edge (arc) of P or $v \in V$ is a node of P. We denote by $V(P)$ and $E(P)$ ($A(P)$) the set of inner nodes and edges (arcs) of P. That is, for a path $P = (v_0, e_1, v_1, ..., e_l, v_l)$ in G (D) we have $V(P) = \{v_1, v_2, ..., v_{l-1}\}$ and $E(P) = \{v_1, v_2, ..., v_{l-1}\}$ ($= A(P)$). We will use the term *simple path* to denote paths without node repetition. A *(simple) cycle* is a (simple) path where the endnodes are identical. Two paths P_1 and P_2 are *node-disjoint* if $V(P_1) \cap V(P_2) = \emptyset$. Analogously, P_1 and P_2 are *edge-disjoint* (*arc-disjoint*) if $E(P_1) \cap E(P_2) = \emptyset$ ($= A(P_1) \cap A(P_2)$).

A graph $\tilde{G} = (\tilde{V}, \tilde{E})$ is a *subgraph* of $G = (V, E)$ if $\tilde{V} \subseteq V$ and $\tilde{E} \subseteq E$. A graph $G = (V, E)$ is said to be *connected* if there is a path between any two nodes. A *tree* is a connected graph with no cycles. A *spanning tree* is a subgraph of G which has the same set of nodes of G and is a tree.

If $G = (V, E)$ is a graph and $X \subseteq V$, then the set of edges $\delta(X) := \big\{\{u, v\} \in$

$E \mid u \in X, v \notin X\}$ is a *cut*. For a digraph $D = (V, A)$ and a subset $X \subseteq V$ of nodes, let $\delta(X)^+ := \{(u, v) \in A \mid u \in X, v \notin X\}$, $\delta(X)^- := \delta(V \backslash X)^+$ and $\delta(X) := \delta^+(X) \cup \delta)^-(X)$. The arcset $\delta(X)^+$ is called a *directed cut*.

A.3 Network Flows

We have a digraph $G = (V, A)$ with edge capacities $u : E(G) \to \mathbb{R}_+$, two specified vertices s (*the source*) and t (the *sink*). The quadruple (G, u, s, t) is called a *network*. A *flow* is a function $f : E(G) \to \mathbb{R}_+$ with $f(e) \leq u(e)$ for all $e \in E(G)$. We say that f satisfies the *flow conservation rule* at vertex v if

$$\sum_{e \in \delta^-(v)} f(e) = \sum_{e \in \delta^+(v)} f(e)$$

A flow satisfying the flow conservation rule at every vertex is called a *circulation*. For a given network (G, u, s, t), an *s-t-flow* is a flow satisfying the flow conservation rule at all vertices except s and t. We define the *value* of an $s - t$-flow f by

$$\text{value}(f) := \sum_{e \in \delta^+(s)} f(e) - \sum_{e \in \delta^-(s)} f(e)$$

For a digraph G we define $\overleftrightarrow{G} := (V(G), E(G)) \dot{\cup} \{\overleftarrow{e} : e \in E(G)\}$, where for $e = (v, w) \in E(G)$ we define $\overleftarrow{e}$ to be a new edge from w to v. We call $\overleftarrow{e}$ the *reverse edge* of e and vice versa. Given a digraph G with capacities $u : E(G) \to \mathbb{R}_+$ and a flow f, we define *residual capacities* $u_f : E(\overleftrightarrow{G}) \to \mathbb{R}_+$ by $u_f(e) := u(e) - f(e)$ and $u_f(\overleftarrow{e}) := f(e)$ for all $e \in E(G)$. The *residual graph* G_f is the graph $(V(G), \{e \in E(\overleftrightarrow{G}) : u_f(e) > 0\})$.

Given a flow f and a path (or circuit) P in G_f, to *augment* f along P by γ means to do the following for each $e \in E(P)$: if $e \in E(G)$ then increase $f(e)$ by γ, otherwise - if $e = \overleftarrow{e_0}$ for $e_0 \in E(G)$ - decrease $f(e_0)$ by γ.

Given a network (G, u, s, t) and an $s - t$-flow f, an *f-augmenting path* is an $s - t$-path in the residual graph G_f

Given a digraph $G = (V, A)$ with edge capacities $u : E(G) \to \mathbb{R}_+$, and numbers $b : V(G) \to \mathbb{R}$ with $\sum_{v \in V(G)} b(v) = 0$, a *b-flow* in (G, u) is a function $f : E(G) \to \mathbb{R}_+$ with $f(e) \leq u(e)$ for all $e \in E(G)$ and

$\sum_{e\in\delta^+(v)} f(e) - \sum_{e\in\delta^-(v)} f(e) = b(v)$ for all $v \in V(G)$. $b(v)$ is called the balance of vertex v. $|b(v)|$ is called the *supply* if $b(v) > 0$ resp. the *demand* of v otherwise. Vertices v with $b(v) > 0$ are called sources, those with $b(v) < 0$ sinks.

The Minimum Cost Flow Problem is defined as the problem of finding for a given digraph G with edge capacities $u : E(G) \to \mathbb{R}_+$, and numbers $b : V(G) \to \mathbb{R}$ a b-flow with minimum weight respective to a weight function $c : E(G) \to \mathbb{R}$ such that $c(f) := \sum_{e\in E(G)} f(e)c(e)$ is minimum or to decide that no b-flow exists.

A.4 Branch-and-Bound

Branch and Bound is a technique for the complete enumeration of all possible solutions without having to consider them one by one. For many NP-hard combinatorial optimization problems it is the best known framework for obtaining an optimum solution.

To apply the Branch-And-Bound Method to a combinatorial optimization (say minimization) problem, we need two steps:

- "branch": a given subset of the possible solutions can be partitioned into at least two nonempty subsets
- "bound": for a subset obtained by branching iteratively, a lower bound on the cost of any solution within this subset can be computed.

The general procedure then is as follows:

Branch-And-Bound Method
Input: An instance of a problem
Output: An optimum solution S^*.

① Set the initial tree $T := (\mathcal{S}, \emptyset)$, where $\mathcal{S}$ is the set of all feasible solutions .
Mark $\mathcal{S}$ active .
Set the upper bound $U := \infty$ (or apply a heuristic in order to get a better upper bound).
② Choose an active vertex X of the tree T (if there is none *stop*)
Mark $\mathcal{S}$ non-active .

("branch") Find a partition $X = X_1 \dot{\cup} \ldots \dot{\cup} X_t$.

③ **For** each $i = 1, \cdots, t$ **do**:
("bound") Find a lower bound L on the cost of any solution in X_i.
If $|X_i| = 1$ (say $|X_i| = \{S\}$) and $\text{cost}(S) < U$ **then**:
Set $U := \text{cost}(S)$ and $S^* := S$.
If $|X_i| > 1$ and $L < U$ **then**:
Set $T := (V(T) \cup \{X_i\}, E(T) \cup \{\{X, X_i\}\})$ and mark X_i active.

④ **Go to** ②

A.5 Linear Programming

A polyhedron $\mathcal{P} = \{x \in \mathbb{R}^n \mid Ax \leq b\} \subseteq \mathbb{R}^n$, and a linear function $c : \mathbb{R}^n \to \mathbb{R}$ define a *linear program*, for short LP. Minimization and maximization versions are

$$\max\{c^T x \mid x \in \mathcal{P}\} \quad \text{and} \quad \min\{c^T x \mid x \in \mathcal{P}\}. \tag{A.1}$$

Each vector $x \in \mathcal{P}$ is called *feasible*. A vector $x^* \in \mathcal{P}$, which attains the maximum (minimum) in (A.1), is an *optimal solution*.

For any IP we can generate an LP (called the *LP-relaxation*) from the IP by taking the same objective function and same constraints but with the requirement that variables are integer replaced by appropriate continuous constraints.

Unit u	
o_u	current position
h_u	home position
t_u^{start}	logon time
t_u^{end}	shift end time
t_u^{max-ot}	maximum overtime
F_u	capabilities
c_u^{drv}	costs per time unit for driving,
c_u^{wait}	costs for waiting,
c_u^{svc}	costs for service,
c_u^{ot}	costs for overtime.

Contractor v	
F_v	a set of capabilities
c_v^{svc}	costs per service

Event e	
x_e	position
θ_e^r	release time
θ_e^d	deadline
δ_e	service time
F_e	required capabilities
c_e^{late}	lateness coefficient

Tours R	
$\mathcal{R}_u$	feasible tours for u, e.g. $(u, e_1, e_2, ..., e_k)$
c_R	cost of the route R
δ_u^{ef}	the driving time of unit u from event e to event f
$\delta_u^{o_u e}$	driving times of unit u from its current position to event e
$\delta_u^{e d_u}$	driving time of unit u from event e to its home position d_u
t_R^e	arrival time at event e in route R
$t_R^{d_u}$	arrival time of u at its home position

Table A.1: Overview: The most important parameters

Anhang B

Deutsche Zusammenfassung

In dieser Diplomarbeit werden untere Schranken bei der Online-Fahrzeug-Disposition in vielerlei Hinsicht untersucht. Dies basiert auf der Grundlage des am Konrad-Zuse-Zentrum Berlin (ZIB) entwickelten Algorithmus ZIBDIP, der entwickelt wurde, um das Fahrzeugeinsatzplanungssystem des ADAC zu automatisieren. Das Umfeld, die Zielsetzung und die generellen Probleme eines solchen Algorithmus werden in Kapitel 1 beschrieben. Die Modellierung des Problems und die algorithmische Lösung, die ZIBDIP zugrunde liegt, wird im darauffolgenden Kapitel 2 ausführlich erklärt.

Das Problem wird als Set-Partitioning-Problem auf der Menge der möglichen Touren betrachtet. Da diese Menge sehr groß ist, werden nur die Touren betrachtet, die für die Lösungsfindung nötig sind und Touren bei Bedarf durch eine Tiefensuche in einem Branch-and-Bound-Baum gesucht. Um ein schnelles Abarbeiten dieses Suchbaumes zu ermöglichen, müssen *untere Schranken für die reduzierten Kosten aller Touren mit einer gegebenen Anfangssequenz von Einsätzen* berechnet werden, was in Kapitel 3 durch die Lösung eines kostenminimalen Flussproblems gelöst wird. Dieses Verfahren berücksichtigt die Strafkosten, die durch verspätet erbrachte Hilfeleistungen an den Kunden entstehen, besser als das bisher verwendete Verfahren. Obwohl dieses neue Verfahren mehr Rechenzeit zur Berechnung einer unteren Schranke in Anspruch nimmt, konnte die Spaltengenerierung erheblich beschleunigt werden, da in der Regel nur ein Bruchteil der Knoten der jeweiligen Suchbäume durchsucht werden muss.

Der Wert der Offline-Lösung des Online-Problems stellt wiederum eine *untere Schranke für die optimale Lösung des Online-Problems* dar, so dass mit diesem Wert *Aussagen über die Güte einer Online-Lösung* gemacht werden können. Bisher konnten aber nur Offline-Problem bei kompletter Kenntnis der Auftrge der nchsten zwei Stunden in angemessener Zeit gelöst werden. Das Verfahren, das

in Kapitel 3 vorgestellt wird, bringt auf diesen Instanzen zwar eine Verbesserung der Laufzeit, kann das grundlegende Problem dieses Spaltengenerierungsverfahrens beim Lösen großer Offline-Probleme aber nicht beheben: Der bisher gewählte Branch-and-Bound-Ansatz zur Spaltengenerierung wurde speziell für die Schnappschu-Instanzen entwickelt, die bei der Online-Planung auftreten (d.h. bei denen alle zu planenenden Auftrge eine Freigabezeit in der Vergangenheit haben). Diese zeichnen sich durch eine geringe durchschnittliche Tourenlänge und vergleichsweise hohe Strafkosten für Verspätungen aus. Im Gegensatz hierzu weisen offline Probleme eine vergleichsweise hohe Tourenlänge und niedrige Strafkosten auf, was sich auf die Laufzeit der Spaltengenerierung nach dem Branch-and-Bound-Ansatz negativ auswirkt, da auf der einen Seite die Größe der Suchbäume mit der Tourenlänge exponentiell wächst und auf der anderen Seite durch die geringen Verspätungskosten kein effektives Pruning möglich ist.

Aus diesem Grund wird in Kapitel 4 ein neuer Ansatz vorgestellt, der auf dem Lösen der Berechnung einer *unteren Schranke der aktuellen LP-Lösung* mit Resource Constrained Shortest Paths basiert. Dabei wurde hier durch den Einsatz einer künstlichen Diskretisierung der Zeit durch Zeit-Buckets das Problem der möglichen Vollenumeration aller Touren gelöst. Die Granularität der Diskretisierung verändert sich zur Laufzeit des Algorithmus um den verschiedenen Ansprüchen an die Spaltengenerierung bezüglich Genauigkeit der dabei errechneten unteren Schranke, Menge der generierten Spalten und Laufzeit zu jedem Zeitpunkt gerecht zu werden. Dieses Verfahren hat bei der Lösung von Schnappschuss-Problemen zwar keine Verbesserung erbracht (was auch nicht das Ziel war), bei der Lösung von Offline-Problemen konnte aber eine beträchtliche Beschleunigung erreicht werden. Während es bisher nur möglich war, Offline-Probleme bei kompletter Kenntnis der Auftrge der nchsten zwei Stunden zu lösen, konnten mit diesem Ansatz Instanzen gelöst werden, die alle Aufträge der nächsten fünf Stunden bis auf einen Optimalitätsgap von 1% in angemessener Zeit gelöst werden. Somit ist es jetzt möglich, Aussagen über die Güte der von ZIBDIP berechneten Online-Lösungen zu treffen, die bisher nur unter erheblichem Mehraufwand formuliert werden konnten.

List of Tables

List of Figures

Bibliography

[1] ADAC. strassenwacht_strassendienst_15.jpg. `http://presse.adac.de/infogramme/Hilfe_Leistungen/`, February 2004.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice Hall, 1993.

[3] Y. Aneja and K. Nair. The constrained shortest path problem. *Nav. Res. Log. Q.*, 25:549–553, 1978.

[4] J. Beasley and N. Christofides. An algorithm for the resource constrained shortest path problem. *Networks*, 19:379–394, 1989.

[5] V. Chvátal. *Linear programming*. A series of books in the mathematical sciences. New York - San Francisco: W. H. Freeman and Company, 1983.

[6] E. V. Denardo and B. L. Fox. Shortest-route methods: 1. reaching, pruning, and buckets. *Operations Research*, 27:161–186, 1979.

[7] M. Desrochers and J. Desrosiers. A new optimization algorithm for the vehicle routing problem with time windows. *Operations Research*, 40(2):342–354, 1992.

[8] D. Eppstein. Finding the k shortest path. *SIAM Journal of computing*, 28(2):652–673, 1999.

[9] M. Garey and D. Johnson. *Computers and Intractability: A guide to the theory of NP-Completeness*. W.H. Freeman, New York, 1979.

[10] M. Grötschel, S. O. Krumke, J. Rambau, T. Winter, and U. T. Zimmermann. Combiatorial online optimization in real time. In M. Grötschel, S. O. Krumke, and J. Rambau, editors, *Online Optimization of Large Scale systems*, pages 705–730. Springer, 2001.

[11] G. J. Handler and I. Zang. A dual algorithm for the constrained shortest path problem. *Networks*, 10:293–310, 1980.

[12] R. Hassin. Approximation schemes for the restricted shortest path problem. *Mathematics of Operations Research*, 17(1):36–42, 1992.

[13] H. Joksch. The shortest route problem with constraints. *Journal of mathematical Analysis and Application*, 14:191–197, 1966.

[14] B. Korte and J. Vygen. *Combinatorial Optimization*. Springer, Berlin, Heidelberg, New York, 2000.

[15] S. O. Krumke, J. Rambau, and L. M. Torres Carvajal. Real-time dispatching of guided and unguided automobile service units with soft time windows. Technical report, ZIB, 2001.

[16] S. O. Krumke and L. M. Torres Carvajal. Online-studies on munich data. Technical report, ZIB, 2002.

[17] E. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rhinehart and Winston, 1976.

[18] D. H. Lorenz and D. Raz. Approximation schemes for the restricted shortest pah problem. *Operations Research Letters*, 28:213–219, 2001.

[19] G. L. Nemhauser and L. A. Wolsey. *Integer and Combinatorial Optimization*. Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., 1988.

[20] O. Saalmann. 50 jahre strassenwacht.mp3. `http://presse.adac.de/hoerfunk/Technik/`, February 2004.

[21] A. Schrijver. *Theory of linear and integer programming*. Wiley-Interscience Series in Discrete Mathematics. A Wiley-Interscience Publication. Chichester: John Wiley & Sons Ltd., 1986.

[22] L. M. Torres Carvajal. *Online Vehicle Routing*. PhD thesis, ZIB, 2003.

[23] A. Warburton. Approximation of pareto-optima in multiple objective shortest path problems. *Operations Research*, 35(1):70–79, 1987.

Statutory Declaration, Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt die selbständige und eigenhändige Anfertigung dieser Diplomarbeit.

(Stephan Westphal)