DEGREE PROJECT IN MATHEMATICS 120 CREDITS, SECOND CYCLE
STOCKHOLM, SWEDEN 2016

Solving the Train Timetabling Problem by using Rapid Branching

JERKER ANDERSSON

KTH ROYAL INSTITUTE OF TECHNOLOGY

SCHOOL OF ENGINEERING SCIENCES

Solving the Train Timetabling Problem by using Rapid Branching

J E R K E R A N D E R S S O N

Master's Thesis in Optimization and Systems Theory (30 ECTS credits)
Master Programme in Mathematics (120 credits)
Royal Institute of Technology year 2016
Supervisor at VTI: Per-Olov Lindberg
Superviro at KTH: was Krister Svanberg
Examiner: Krister Svanberg

TRITA-MAT-E 2016:02
ISRN-KTH/MAT/E--16/02--SE

Royal Institute of Technology
School of Engineering Sciences

KTH SCI
SE-100 44 Stockholm, Sweden

URL: www.kth.se/sci

Solving the Train Timetabling Problem by using Rapid Branching

by Jerker Andersson

Abstract

The topic of this thesis is the implementation of rapid branching to find an integer solution for the train timetabling problem. The techniques that rapid branching are based on are presented. The important aspects of rapid branching are discussed and then the algorithm is applied to some artificial problems. It is shown that rapid branching can be both faster and slower than a standard integer solver depending on the problem instance. For the most realistic set of the examined instances, rapid branching turned out to be faster than the standard integer solver and produce satisficingly high quality solutions.

Sammanfattning

Den här uppsatsen handlar om en implementering av rapid branching för att hitta en heltalslösning till optimeringsproblemet vid tidtabelläggning för järnvägar. Rapid branching är en algoritm skapad för att fungera bra på storskaliga heltals-optimeringsproblem. I uppsatsen beskrivs några sätt att skapa egna tidtabelläggnings problem och sedan jämförs rapid branching med en vanlig heltalslösare för de problemen. Det visar sig att algoritmen rapid branching kan vara både snabbare och långsammare än att använda en konventionell heltalslösare. För den mest realistiska instansen av tidtabelläggning problemet visade det sig att rapid branching var snabbare än heltalslösaren samt att den funna lösningen var av tillfredsställande hög kvalitet.

Acknowledgements

First I would like to thank my advisor from VTI Per Olov Lindberg for many exciting and interesting discussions on different mathematical subjects discussed in this thesis. I would also like to thank my supervisor and examiner from KTH Krister Svanberg for guidence and suggestions on topics to examine. At last I would like to give a big thanks to my family for their love and support.

Contents

List of Figures

List of Tables

Abbreviations

(TTP)	Train Timetabling Problem
(P)	Arbitrary relaxation of Train Timetabling Problem
$(\overline{TTP})$	Linearly Relaxed Train Timetabling Problem
$(TTP)_{\boldsymbol{\mu}}$	Lagrangian Relaxed Train Timetabling Problem
$(\overline{BP})_k$	Bundle Subproblem in iteration k
(RTTP)	Restricted Train Timetabling Problem
$(\overline{RTTP})$	Linearly relaxed Restricted Train Timetabling Problem

Symbols

$\mathcal{B}$	Set of block intervals	b
$\mathcal{T}$	Set of time intervals	t
$\mathcal{V}$	Set of block-times	(b, t)
$\mathcal{R}$	Set of requests	r
$\mathcal{N}_r$	Graph for routes of r	$(\mathcal{A}_r, \mathcal{V}_r)$
$\mathcal{A}_r$	Arcs for graph $\mathcal{N}_r$	a
$\mathcal{V}_r$	Vertices for graph $\mathcal{N}_r$	ν
$\mathcal{P}_r$	Set of possible paths for r	p
$\mathcal{P}$	Set of combinations of all possible paths	$\{p_1, p_2 \ldots, p_R\}$
$\mathbf{c}$	Capacities of all blocks b	c_b
Δ^a	Matrix of capacity consumptions for arc a	δ^a_{bt}
$\mathbf{d}^p$	Matrix of capacity consumptions for path p	d^p_{bt}
$\mathbf{x}$	Vector of decision variables of paths p	$x_p \in \{0, 1\}$
$\mathbf{v}$	Vector of revenues for each path p	v_p
$\mathbf{l}$	Vector of lower bounds for decision variables x_p	l_p
$\mathbf{u}$	Vector of upper bounds for decision variables x_p	u_p
$\boldsymbol{\mu}$	Vector of lagrangian multipliers of block-times	μ_{bt}

Chapter 1

Introduction

The main topic of this thesis concerns finding an integer solution for the Train Timetabling Problem(TTP) by using rapid branching. The (TTP) is a large scale integer mathematical optimization problem and rapid branching is a novel heuristic algorithm for solving large scale integer mathematical optimization problems. The rapid branching algorithm has been developed mainly by Steffen Weider in "Integration of Vehicle and Duty Scheduling in Public Transport"[1] by combining ideas from several mathematical optimization topics. Some examples of these topics are Branch and Bound algorithms, Shortest Path algorithms, Lagrangian Relaxations and Bundle Methods. The goal of rapid branching is to find a satisfyingly good solution faster by trading exactness for speed. For some material on recent implementations of rapid branching, see Borndörfer et al. "Rapid Branching"[2].

1.1 Context

Today train timetables are made by scheduling trains individually and then manipulating the paths such that they don't interfere with each other. It was not long ago that the planners planned timetables with pen and paper. Up until recently it was not possible to solve (TTP) with computers but due to recent advances in algorithms and processing power it should now be possible to find at least an approximately optimal solution to (TTP). The question is if a schedule generated by rapid branching is more profitable than the schedules produced by methods used today.

The rapid branching algorithm should be applied to real world data and the schedule found should be compared to the schedule found by the methods used today. In order to verify that rapid branching produces feasible solutions to (TTP) it has also been applied to some artificial problems.

1.2 Project

The project about finding the solution for (TTP) was initiated by "Statens väg- och transportforskningsinstitut"(VTI) which has an office located on the campus of The Royal Institute of Technology(KTH). The team that has been working on the project are Jan-Eric Nilsson, Per Olov Lindberg, Abderrahman Ait Ali, Tobias Gurdan, Alain Kaeslin and myself Jerker Andersson. Jan-Eric is the project leader, Per Olov has served as advisor, Abderrahman has handled the bundle method, Alain and Tobias the shortest path algorithm and I have implemented the integer programming part of the problem.

This thesis is about finding an integer solution for the (TTP) by using rapid branching. This is a subpart of the larger project that consists of solving the (TTP) for a certain stretch of track along the Swedish railroad "Malmbanan". Finding a solution for (TTP), which is briefly introduced in section 1.3 and is described more in detail in section 2.2, can be divided into three distinct main parts or phases. The part done by me consists of finding a integer solution for (TTP) and another part is a shortest path implementation by Alain and Tobias. There is also a bundle phase implementation made by Abderrahman. Abderrahman has also been responsible for the interfacing between the different parts and has also worked on the shortest path implementation. PO has been the project advisor and is responsible for the modelling and theory.

1.3 Introduction to the Train Timetabling Problem

The Train Timetebling Problem(TTP) consists of finding a conflict-free schedule for a set of trains/requests such that the revenue from using the timetable is maximized. When we talk of conflict-free schedule we mean that all scheduled paths are mutually conflict-free. Each path describe the movement of a train trough "space time". To model paths and the movement of the train along the track time and space are discretized. The track and time are discretized into intervals and form a block-time-space. Each block-time has a capacity of a maximum number of trains that can occupy it, i.e for a single-track railroad the capacity is one and for double track the capacity would be two. As a train moves along the track it will occupy block-times and consume the capacities of block-times. If two paths overlap such that the combined capacity consumption for some block-time is larger than the maximum capacity of that block-time then the two paths are in conflict. If no capacity is violated then the two paths are mutually conflict-free. If further all scheduled paths are jointly conflict-free then the schedule is said to be conflict-free.

Finding a conflict-free schedule may sound like an easy problem but the issue is the huge number of feasible paths for each train. In practice the number of possible paths are so large such that they are too many to explicitly construct and examine. This is why the shortest path algorithms are used. Each shortest path calculation will provide a path that is optimal to a specific sub-problem of the (TTP). The bundle method is used to construct and identify appropriate sub-problems to solve . Once the shortest path algorithm has found sufficiently many different paths it is possible to construct an integer solution using the found paths as a solution space. By solution space it is meant that the paths generated by the shortest path algorithm are the paths that can be scheduled for requests.

1.4 Aim and scope

The goal of this thesis is to implement rapid branching to find a sufficiently good integer solution for (TTP). One aim of the main project is to show that it is possible to speed up solving (TTP) by parallelizing the shortest path algorithm calculations and by using

the disaggregate bundle method instead of the bundle method. The bundle method is likely to find non-integer solutions so it is important to have an efficient algorithm to find an integer solution. It is worth mentioning that rapid branching is a heuristic method and as such it does not necessarily find an optimal solution and is more likely to find an approximately optimal solution.

When mathematically modelling (TTP) the rules pertaining traffic on the track may complicate the model. In Sweden there is a "3 minute rule" that says moving trains must have free track at least 3 minutes ahead if continuing at constant speed. There are more rules for signalling and stopping at special sidetracks beside the normal track. Some of these rules are relaxed for the (TTP) solved in this thesis. How to accurately take such rules into account is certainly one possible future question to address. Future questions are discussed in later chapters but the main questions that are to be addressed in this thesis are:

- Is rapid branching applicable on (TTP)?
- How fast is rapid branching?
- How good is the solutions found by rapid branching?
- How does changing parameters for rapid branching change speed and quality of solutions?
- How does rapid branching compare to conventional solvers?

Chapter 2

Mathematical Model

In this chapter, the mathematical notations and models will first be explained. Then some sections about different relaxations will follow. The last part is about the bundle method and the restricted problems.

2.1 Modelling and basic notation

What follows is some basic notation regarding the movement of trains along a track. To describe the position of a train the track is discretized into a set of *blocks* $b \in \mathcal{B} = \{1, \dots B\}$. Associated with each block b is a *capacity* c_b of trains per time unit. For most of the blocks b, the corresponding capacity will be 1, but there will be certain blocks with larger capacity where trains may pass each other, e.g stations. The time is also discretized into *intervals* $t \in \mathcal{T} = \{1, \dots T\}$. The block and time intervals together form a *block-time-space* $(b, t) \in \mathcal{B} \times \mathcal{T} = \mathcal{V}$.

Different companies can make requests for using the track. The *train requests* r belong to the set $\mathcal{R} = \{1, \dots, R\}$. For each request r there are different possible paths in the block-time-space to take it from the starting station S_r^s to the ending station S_r^e. The paths for request r can be described by the *train movement graph* $\mathcal{N}_r = (\mathcal{A}_r, \mathcal{V}_r)$. The vertices $\mathcal{V}_r \subseteq \mathcal{V}$ are the different positions in space-time while the arcs $a_r \in \mathcal{A}_r$ describe train movement. An example of an arc $a \in \mathcal{A}_r$ would be: at time t_1 train r accelerates to full speed from standing still at station A and then continues at full speed until it

reaches station B at time t_2. This is the definition of arcs as in "Railway Timetabling using Lagrangian Relaxation"[3].

For a request r, let $\mathcal{P}_r$ denote all possible paths p. Then the set of all possible paths of all requests is $\mathcal{P} = \bigcup_{r \in \mathcal{R}} \mathcal{P}_r$. Associated with each path p is a revenue v_p. All arcs a result in a capacity consumption of the block-times $(b,t) \in \mathcal{V}$. Let the capacity consumption of a be denoted by the matrix $\Delta^a = (\delta^a_{bt})$ where the rows represent the blocks and the columns represent the time intervals. The element $(\delta^a_{bt}) \in \Delta^a$ is equal to one if arc a occupies block b at time t and equal to zero otherwise. An arc a of a request r can occupy blocks the train has not yet physically entered or it has already physically passed. This is necessary to model certain safety rules pertaining railway timetabling, e.g. a moving train must have a free track 3 minutes ahead if moving at full speed. The capacity consumption for a certain path p is defined in the same way as $\mathbf{d}^p = (d^p_{bt})$. For all block-times $(b,t) \in \mathcal{V}$ that are occupied by a path p the corresponding element $d^p_{bt} \in \mathbf{d}^p$ is equal to one. If a block-time (b,t) is not occupied by the path p then the value of d^p_{bt} is zero.

2.2 Train Timetabling Problem

The train timetabling problem (TTP) corresponds to finding a set of paths $\mathcal{P}_I \subset \mathcal{P}$ such that the profit is maximized, no capacity constraints c_b are violated and there is one and only one path chosen for each request r. In order to avoid ending up in a state where there is no feasible solution the null path is allowed for a request r. Choosing the null-path is equivalent to not scheduling the train at all. The binary variable x_p is introduced as a decision variable to show if a path p is chosen. If the path p is chosen then the value of x_p is one and if p is not chosen then x_p is zero. Let $\mathbf{x} = (x_p)_{p \in \mathcal{P}}$ be a vector with elements x_p for all possible paths p and let $\mathbf{v}$ be a vector of revenues v_p of all possible paths.

A path p could be feasible for several requests r. If it is the case that a path p is feasible for more than one request then they are considered different paths with different binary variables and likely different associated revenues v_p. For example say that a path p is feasible for request r_1 and request r_2. Then there are two binary variables x_{p_1} and x_{p_2} which respectively correspond to revenues v_{p_1} and v_{p_2} but the two binary

variables correspond to the same capacity consumption matrix $\mathbf{d}^p = \mathbf{d}^{p_1} = \mathbf{d}^{p_2}$. With all previous definitions in this chapter (TTP) can be stated as IP-problem:

$$
\begin{array}{lllllll}
 & \max\limits_{x_p} & \sum\limits_{p\in\mathcal{P}} v_p x_p & & & & \\
(TTP) & \text{s.t.} & \sum\limits_{p\in\mathcal{P}} d^p_{bt} x_p & \leq & c_b, & \forall (b,t) \in \mathcal{B}\times\mathcal{T} & (i) \\
 & & \sum\limits_{p\in\mathcal{P}_r} x_p & = & 1 & \forall r \in \mathcal{R} & (ii) \\
 & & x_p & \in & \{0,1\} & \forall p \in \mathcal{P} & (iii)
\end{array}
\tag{2.1}
$$

Constraint (i) imposes that no capacity is violated. Observe that as mentioned earlier a train may occupy more than the block it occupies physically. The second constraint (ii) says that no request should have more than one scheduled path. If two requests have departure windows that overlap then the two requests are likely to have feasible paths that are identical. In those cases such paths are considered as different paths. They are also likely to have different revenues associated to identical paths since the revenue usually is dependent on train type etc. The last constraint (iii) says that a path is either chosen or it is not, i.e x_p is a binary variable.

Now the notion of fixing a variable will be explained. If a variable x_p is fixed to zero or one then the original (TTP) would be slightly perturbed but the new problem would again be on the form of a (TTP). To keep track of fixed variables two vectors $\mathbf{l}$ and $\mathbf{u}$ with the same size as $\mathbf{x}$ are introduced. The vectors model fixings by providing a lower and upper bound for elements of $\mathbf{x}$. If the value of x_p for a certain path is fixed to one then it would correspond to $l_p = u_p = 1$, i.e that particular path is used by a particular request. Similarly x_p may be fixed to zero which would indicate that the path p is not scheduled. Then it would hold that $l_p = u_p = 0$.

2.3 Relaxations

The constraints of (2.1) can be relaxed in different ways. For a problem (P) to be a relaxation of (TTP) two criteria must be satisfied. First it must hold that the solution-space of (TTP) called X is a subspace of the solution-space of (P) called X_P, i.e $X \subseteq X_P$. Let $(TTP)(\mathbf{x})$ and $(P)(\mathbf{x})$ denote the objective value of (TTP) and (P) respectively

given a feasible solution $\mathbf{x} \in X$. Then since (TTP) is a maximization problem the second criterion for a relaxation is stated as:

$$(TTP)(\mathbf{x}) \leq (P)(\mathbf{x}) \quad \forall \mathbf{x} \in X \tag{2.2}$$

This says that for each feasible solution $\mathbf{x} \in X$ it must hold that the objective value of (TTP), is less than or equal to the objective value of (P).

2.3.1 Linear Relaxation

If (TTP) is linearly relaxed then constraint (iii) of 2.1 is replaced by $x_p \in [0,1]$, $\forall p \in \mathcal{P}$. This problem is called $(\overline{TTP})$ and is stated as the following LP-problem:

$$\begin{aligned}
(\overline{TTP}) \quad & \max_{x_p} \quad \sum_{p \in \mathcal{P}} v_p x_p & & & & \\
& \text{s.t.} \quad \sum_{p \in \mathcal{P}} d_{bt}^p x_p & \leq & c_b, & \forall (b,t) \in \mathcal{B} \times \mathcal{T} & (i) \\
& \qquad \sum_{p \in \mathcal{P}_r} x_p & = & 1 & \forall r \in \mathcal{R} & (ii) \\
& \qquad x_p & \in & [0,1] & \forall p \in \mathcal{P} & (iii)
\end{aligned} \tag{2.3}$$

It is obvious that every feasible solution of (TTP) is also feasible for $(\overline{TTP})$. It also holds that the objective value of (TTP) is not larger than the objective value of $(\overline{TTP})$ for any feasible solution $\mathbf{x}$. Thus $(\overline{TTP})$ is a relaxation of (TTP). The difference between (TTP) and $(\overline{TTP})$ is that for the unrelaxed problem it is only allowed to choose a single path for each request, whereas in the linearly relaxed case one can choose to schedule a fractional combination of paths such that the sum of all fractions add up to one for each request r. This corresponds to a flow in the graph $\mathcal{N}_r$ where each request r has a source node $\mathcal{S}_r^s$ with capacity one and a sink node $\mathcal{S}_r^e$ with demand one.

2.3.2 Lagrangian Relaxation

Lagrangian relaxation is done by moving a constraint to the objective. The moved constraints are then multiplied with lagrangian multipliers, which are introduced to penalize violations of the moved constraints. A lagrangian relaxation of (TTP) is found in equation 2.4. This lagrangian problem is from now on referenced to as $(TTP)_{\boldsymbol{\mu}}$. The

difference between (TTP) and $(TTP)_{\boldsymbol{\mu}}$ is that the capacity inequality constraint (i) in 2.1 has been moved to the objective function in 2.4. Since it is imposed that $\mu_{bt} \in \boldsymbol{\mu}$ is non-negative then violating capacities is penalized while not violating capacities is rewarded in the objective of 2.4.

$$\begin{aligned} & \max_{x_p} \quad \sum_{p \in \mathcal{P}} v_p x_p + \sum_{(b,t) \in \mathcal{V}} \mu_{bt}(c_b - \sum_{p \in \mathcal{P}} d_{bt}^p x_p) \\ (TTP)_{\boldsymbol{\mu}} \quad & \text{s.t.} \quad \sum_{p \in \mathcal{P}_r} x_p = 1 \quad \forall r \in \mathcal{R}(ii) \\ & \qquad x_p \in \{0,1\} \quad \forall p \in \mathcal{P}(iii) \end{aligned} \tag{2.4}$$

It is possible to split 2.4 into one sub-problem for each request r. Each sub-problem is a shortest path problem(SP) which makes 2.4 relatively easy to solve. Any solution $\mathbf{x}$ that is feasible to (TTP) must also be feasible for $(TTP)_{\boldsymbol{\mu}}$. If the solution-space of (TTP) is X and the solution-space of $(TTP)_{\boldsymbol{\mu}}$ is X_P it holds that $X \subseteq X_P$. Observe that a solution $\mathbf{x}$ that is feasible for $(TTP)_{\boldsymbol{\mu}}$ may be infeasible for (TTP) since the relaxed capacity inequality constraint may be violated.

The second criteria for $(TTP)_{\boldsymbol{\mu}}$ to be a relaxation is that for every solution $\mathbf{x}$ that is feasible to (TTP) equation 2.2 must hold. This will hold if it is imposed that $\boldsymbol{\mu}$ is non-negative. If $\boldsymbol{\mu}$ is non-negative then the objective value of 2.4 will be larger or equal to the objective value of 2.1. This is true since, for any feasible solution to 2.2, the term

$$\mu_{bt}(c_b - \sum_{p \in \mathcal{P}} d_{bt}^p x_p) \tag{2.5}$$

is non-negative for all (b,t).

Since the objective value of $(TTP)_{\boldsymbol{\mu}}$ is an upper bound for (TTP) it is a good idea to make the gap between objective values as small as possible. If $\varphi(\boldsymbol{\mu})$ is defined as the optimal value of 2.4 given a set of non-negative multipliers $\boldsymbol{\mu} = (\mu_{bt}) \geq 0$ then the gap is minimized by solving the following problem:

$$\begin{aligned} & \min \quad \varphi(\boldsymbol{\mu}) \\ (TTP)_{\varphi} \quad & \text{s.t.} \quad \boldsymbol{\mu} \geq 0 \quad (i) \end{aligned} \tag{2.6}$$

By looking at 2.4 one can realize that $\varphi(\boldsymbol{\mu})$ has some nice properties. It is a piecewise affine convex function. This is true since there is a finite number of possible path configurations and $\varphi(\boldsymbol{\mu})$ is the maximum of a set of affine functions. The solution of 2.6 may be found by using the bundle method described in section 2.4.

What now follows are some concepts needed for the section on the bundle method. For an arbitrary set of lagrange multipliers $\overline{\boldsymbol{\mu}} \geq 0$ there is an optimal solution $\overline{\mathbf{x}}$ to $(TTP)_{\overline{\boldsymbol{\mu}}}$. Let the optimal value $\overline{\mathbf{x}}$ be a function of $\overline{\boldsymbol{\mu}}$ according to $\mathbf{x}(\overline{\boldsymbol{\mu}}) = \overline{\mathbf{x}}$. By plugging $\overline{\mathbf{x}}$ into the objective function of 2.4 the following function can be defined:

$$\overline{\varphi}(\boldsymbol{\mu}) = \sum_{p \in \mathcal{P}} v_p \overline{x}_p + \sum_{(b,t) \in \mathcal{V}} \mu_{bt} (c_b - \sum_{p \in \mathcal{P}} d_{bt}^p \overline{x}_p) \tag{2.7}$$

The function $\overline{\varphi}(\boldsymbol{\mu})$ is a linear function of $\boldsymbol{\mu}$ that is equal to $\varphi(\boldsymbol{\mu})$ at $\overline{\boldsymbol{\mu}}$. It can also be called a supporting plane of φ. The slope of the supporting plane is a *subgradient* for φ at $\overline{\boldsymbol{\mu}}$. The subgradient is defined as

$$\overline{\mathbf{g}} = \mathbf{g}(\overline{\boldsymbol{\mu}}) = (\overline{g}_{bt}) \quad \text{where} \quad \overline{g}_{bt} = c_b - \sum_{p \in \mathcal{P}} d_{bt}^p \overline{x}_p \quad \forall (b,t) \tag{2.8}$$

By using the definition of subgradient in equation 2.8 the supporting linear function of φ at $\overline{\boldsymbol{\mu}}$ is defined as

$$\widetilde{\varphi}(\boldsymbol{\mu}) = \varphi(\overline{\boldsymbol{\mu}}) + \mathbf{g}^T(\overline{\boldsymbol{\mu}})(\boldsymbol{\mu} - \overline{\boldsymbol{\mu}}) \tag{2.9}$$

2.4 Bundle method

Only the most important aspects of the bundle method will be described here. For a more detailed description of the bundle method see for example [4]. The bundle method is an iterative method. Assume that at iteration k the lagrange multipliers $\boldsymbol{\mu} = \boldsymbol{\mu}_k$ are used. The function φ is approximated by supporting planes calculated in iterations $\{1, \ldots, k\}$. For each such supporting plane there is a corresponding subgradient $(\mathbf{g}_l)$. The first step is then to solve the relaxed bundle subproblem $(\overline{BP})_k$ found in 2.10.

$$\begin{array}{lll} & \min\limits_{\boldsymbol{\mu}} & \overline{\varphi}^k(\boldsymbol{\mu}) + \dfrac{u_k}{2}|\boldsymbol{\mu} - \boldsymbol{\mu}_k|^2 \\ (\overline{BP})_k & \text{s.t.} & \boldsymbol{\mu} \geq 0 \quad (i) \end{array} \tag{2.10}$$

The function $\overline{\varphi}^k(\boldsymbol{\mu})$ is defined as

$$\overline{\varphi}^k(\boldsymbol{\mu}) = max_{l \in \mathcal{L}_k}\{\varphi(\boldsymbol{\mu}_l) + \mathbf{g}_l^T(\boldsymbol{\mu} - \boldsymbol{\mu}_l)\} \tag{2.11}$$

which in words translates to being the maximum of the supporting linear functions at $\boldsymbol{\mu}_l$ where l belongs to a *bundle* $\mathcal{L}_k$ of subgradients. In 2.10 $|.|$ denotes the euclidean norm. The reason for including a quadratic term is too avoid taking to long steps and the parameter u_k included as a way of controlling the step-size.

Let the optimal solution of 2.10 be $\mathbf{y}_{k+1}$. For this set of multipliers the problem $(TTP)_{\mathbf{y}_{k+1}}$ found in 2.4 is solved. When $(TTP)_{\mathbf{y}_{k+1}}$ is solved it is possible but not necessary that a new subgradient and supporting plane is found. Now either a serious step or a null step is taken. What kind of step is made depends on the achieved descent compared to the predicted descent. The achieved descent is $\varphi(\boldsymbol{\mu}_k) - \varphi(\mathbf{y}_{k+1})$ and the predicted descent is $\varphi(\boldsymbol{\mu}_k) - \overline{\varphi}(\mathbf{y}_{k+1})$. If the achieved descent is at least some fraction $m_l \in (0, 0.5)$ of the predicted descent then a serious step is made and otherwise a null step is taken.

If a serious step is made then the following is done. The lagrangian multipliers is updated according to $\boldsymbol{\mu}_{k+1} = \mathbf{y}_{k+1}$. The bundle is also updated such that only the active supporting planes at $\boldsymbol{\mu}_{k+1}$ are included in the bundle. The parameter for controlling step-sizes u_{k+1} is updated. It is updated such that the curvature of the objective function in 2.10 between $\boldsymbol{\mu}_k$ and $\mathbf{y}_{k+1}$ fits the curvature of φ. Some other constraints on u_{k+1} is that it may never be smaller than some minimum value u_{min} and it may never be less than ten times smaller than u_k, i.e $u_{k+1} \geq max\{u_{min}, \frac{u_k}{10}\}$.

When a nullstep is taken then the lagrange multipliers are updated according to $\boldsymbol{\mu}_{k+1} = \boldsymbol{\mu}_k$. The new supporting hyperplane is added to the bundle and the step-size parameter u_{k+1} is updated the same way as for a serious step.

Once the step-sizes have been updated the algorithm repeats by solving a new sub-problem. This is repeated until a certain maximal number of iterations is exceeded or a tolerance criterion is fulfilled.

2.5 Restricted Problems

In this section, the notion of restricted problem used in this report is defined. The original problem is (TTP) and all paths $p \in \mathcal{P}$ are possible paths for some train to choose. The set of all paths $\mathcal{P}$ is in general a huge set. The set is usually so huge such that it is not possible to store all paths explicitly.

For the implementation discussed in this thesis for solving (TTP) there are multiple phases. There is an initial phase where the bundle method is used to solve ($\overline{TTP}$) to satisfactory precision. During this phase paths will continuously be generated by a shortest path algorithm. A subset of all paths generated during this phase is stored in $\mathcal{P}_I$. The set of paths $\mathcal{P}_I$ is the solution space for the restricted problem.

The restricted problem is a modified version of (TTP). In (TTP) each request r may choose any feasible path. But in the restricted problem (RTTP) each request r may only choose paths among a subset of all feasible paths. These paths have been explicitly stored and therefore it is not necessary to generate new paths by using the shortest path algorithm. Remember that the shortest path algorithm is used to solve 2.4 when it is decomposed to one problem for each request r. If the subset of explicitly stored paths is denoted $\mathcal{P}_I$ and the set of paths feasible for request r is denoted $\mathcal{P}_{I,r}$ then it is possible to state the restricted problem as:

$$
\begin{aligned}
& \max_{x_p} && \sum_{p \in \mathcal{P}_I} v_p x_p && && \\
(RTTP) \quad & \text{s.t.} && \sum_{p \in \mathcal{P}_I} d_{bt}^p x_p \leq c_b, && \forall (b,t) \in \mathcal{B} \times \mathcal{T} && (i) \\
& && \sum_{p \in \mathcal{P}_{I,r}} x_p = 1 && \forall r \in \mathcal{R} && (ii) \\
& && x_p \in \{0,1\} && \forall p \in \mathcal{P}_I && (iii)
\end{aligned}
\tag{2.12}
$$

This has the same structure as the formulation of (TTP) found in 2.1 hence (RTTP) is a modified version of the original (TTP). Just as it is possible to linearly relax the (TTP) one can relax the binary constraint of (RTTP) to get the linearly relaxed restricted problem. One can further restrict variable x_p by imposing that it is less than some upper bound u_p and greater than some lower bound l_p. Then a linearly relaxed restricted problem ($\overline{RTTP}$) is defined as:

$$
\begin{aligned}
(\overline{RTTP}) \quad & \max_{x_p} \quad \sum_{p \in \mathcal{P}_I} v_p x_p \\
& \text{s.t.} \quad \sum_{p \in \mathcal{P}_I} d^p_{bt} x_p \leq c_b, \quad \forall (b,t) \in \mathcal{B} \times \mathcal{T} \quad (i) \\
& \qquad \sum_{p \in \mathcal{P}_{I,r}} x_p = 1 \qquad \forall r \in \mathcal{R} \qquad (ii) \\
& \qquad x_p \leq u_p \qquad \forall p \in \mathcal{P}_I \qquad (iii) \\
& \qquad x_p \geq l_p \qquad \forall p \in \mathcal{P}_I \qquad (iv)
\end{aligned}
\tag{2.13}
$$

It is ($\overline{RTTP}$) in 2.13 that is the subproblem solved during the phase of rapid branching where an integer solution is found. The lower bounds l_p are initially all zeros and the upper bounds u_p are initially all ones. The bounds are then used to model fixed variables, fixing of variables are explained in section 2.2.

Chapter 3

Rapid Branching

3.1 Introduction

Rapid branching is a heuristic algorithm that works well on IP's with a large number of columns. The goal of the algorithm is to produce a sufficiently high-quality integer solution. Since this project is focused on solving the (TTP), found in equation 2.1, henceforth it is assumed that the algorithm is applied to the (TTP).

3.2 Algorithm

The solving of (TTP) can be split into several phases. One initial bundle phase during which the linearly relaxed version of (TTP), found in 2.3, is solved to satisficing precision. This will produce a fractional initial solution $\mathbf{x}^{(0)} \in [0,1]^n$, where $n = |\mathcal{P}|$ is the total number of possible paths. In practice n is a huge number so the used bundle method is a column generating method. Due to this only a subset of all paths will be explicitly generated during the bundle phase. Then as explained in section 2.5 a subset of all generated paths is saved to use as solution-space during the search for a feasible integer solution $\mathbf{x}^{(k)} \in \{0,1\}$. If $\mathcal{P}$ is the set of all feasible paths, $\mathcal{P}_{bp}$ is the set of all paths generated during the bundle phase and $\mathcal{P}_I$ is the set of paths stored to be used during the search for a integer solution. Then the following relations hold;

$$\mathcal{P}_I \subseteq \mathcal{P}_{bp} \subseteq \mathcal{P} \tag{3.1}$$

The bundle phase will not be explained further here. For more information on the bundle method see section 2.4 or the text "Integration of Vehicle and Duty Scheduling in Public Transport"[1].

In the first phase of the rapid branching algorithm, a fractional solution is found while the second phase consists of finding a feasible integer solution that is satisfactory. The second phase is a Branch-and-bound phase that can be divided into two parts. The first part consists of finding variables that are "potentially fixable" and "potentially beneficial" to fix. Of course all variables are potential fixes but not all would be beneficial to fix. With some slight abuse of notation we will denote variables that are potentially beneficial to fix as potentially fixable, in this chapter and following chapters.

Since the problem solved is assumed to be large the algorithm tries to find several variables to fix at the same time. This is in contrast to standard Branch & Bound where variables are usually fixed one by one. It is also assumed that fixing a single variable results in a small change of objective value.

If a branch and bound tree is defined as in figure 3.1 then an example of what can be found during the first part of the second phase is seen in figure 3.2. In this case the values to fix three of the variables to has been found. During the first part a series of linearly relaxed restricted problems ($\overline{RTTP}$), defined in 2.13, is solved. The problems are LP's restricted to a explicitly generated set of paths. For more information on ($\overline{RTTP}$) see section 2.5. The series of problems will produce a set of branches that are potentially fixable. Instead of exploring all of these branches one is chosen. The choice is made by calculating a potential function for each branch and then the branch with largest potential is explored.

The second part of the second phase consists of finding the variables to fix among the set of potential fixes. When applying fixes it is desirable that the objective value of the restricted problem does not decrease too much. During this part all fixes are applied at first and if this results in a restricted problem where the objective value is below some target value then we reject the fixes. Instead it is analysed if applying half of the fixes gives a restricted problem where the optimal objective value is above some new target value. This is repeated until either one variable to fix remains or the target

value is reached. In both of those two cases the fixes are applied and the second phase is restarted with finding variables to fix.

To summarise, the generation of variables to fix corresponds to finding a branch of the tree and how far down it should be explored. The application of fixes corresponds to finding how far down the branch it is profitable to move.

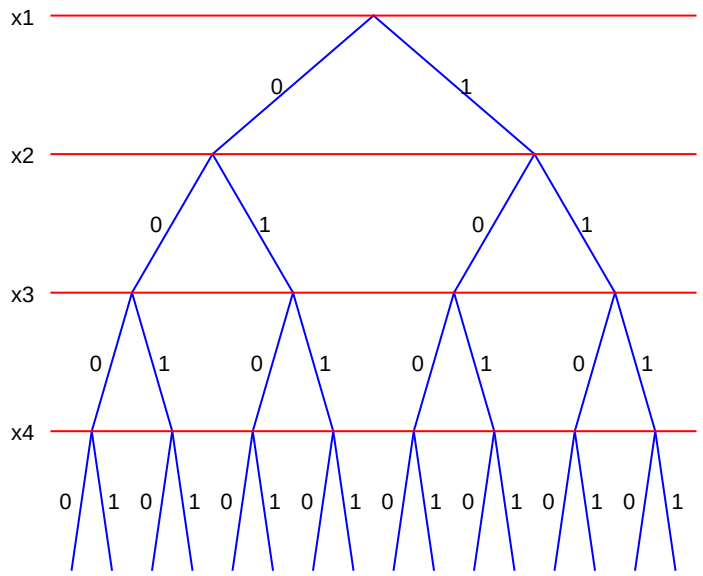

FIGURE 3.1: Branch and bound tree, the choice for each variable is zero or one.

3.2.1 Identifying variables to fix

Algorithm 1 produces a set of indices $B^\star$ for variables that are potentially fixable to one, i.e for a subset $\mathcal{R}_{B^\star} \subseteq \mathcal{R}$ the indices in $B^\star$ corresponds to paths that are candidates to be scheduled in the timetable. The following part will go through the details of algorithm 1.

The various parts of the pseudocode will now be explained. First a fast run-through of the algorithm will be made. Then a more detailed version will follow where the ideas behind different parts of the algorithm are discussed. In the first part the algorithm will be run-through line by line while in the second part there will be some jumping between lines.

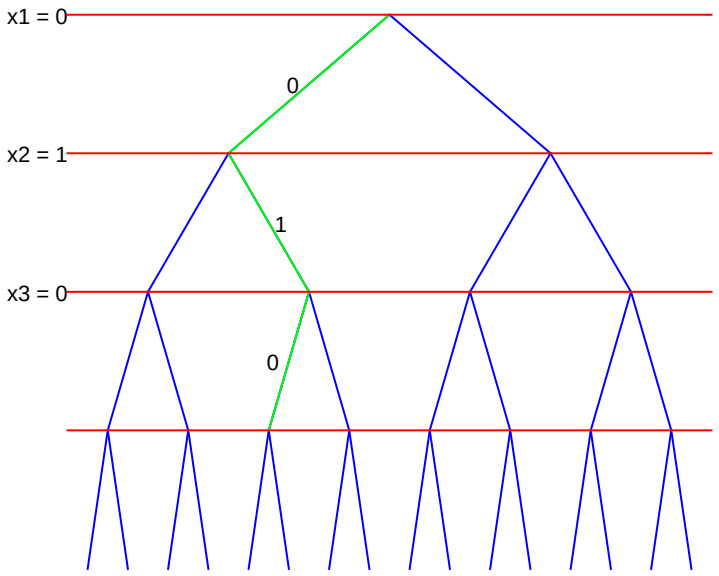

FIGURE 3.2: Branch and bound tree, where some variables to fix has been found.

Short summary of generating potential fixes

The first part of the algorithm just states the output, input, step counters, constants and initial values of variables. The output is the set of potential fixes $B^\star$ and the input is $\mathbf{l}, \mathbf{u}, \mathbf{v}, \mathbf{p}$. The inputs $\mathbf{l}, \mathbf{u}, \mathbf{v}, \mathbf{p}$ are respectively, lower bound of variables in $\mathbf{x}^{(i)}$, upper bound of variables in $\mathbf{x}^{(i)}$, cost vector of variables in $\mathbf{x}^{(i)}$ and stored paths for variables in $\mathbf{x}^{(i)}$. The step counters i and k count different things, i is a normal step counter for the number of iterations while k counts the number of iterations without progress. The constants are $\epsilon, \delta, \alpha, q, k_{max}, k_s$. The ϵ is the maximum deviation from one for a variables value to be considered near integral, δ is a weight for the number of near integer variables when calculating the potential function $f(\mathbf{x}^{(i)})$, α is used as a weight when updating the perturbed cost vector $c_p = c_p + \alpha \cdot (x_p^{(i)})^2$ in iteration i for path p, q is added as a bonus revenue to the cost vector of the largest fraction during the spacer step of the algorithm, k_{max} is the maximum number of times the main loop of the algorithm may loop without *progress* and k_s is the number of steps without *progress* between each spacer step. The initial value of variables that change throughout the algorithm are the currently largest potential function $f^\star$ and the perturbed cost vector $\mathbf{c}$.

After the main loop has started the first thing to do is to solve the linearly relaxed

Algorithm 1 Generating a set of indices of variables to fix, $B^\star$

output $B^\star$
input $\mathbf{l}, \mathbf{u}, \mathbf{v}, \mathbf{p}$
Step counters: $i, k, \leftarrow 0$
Constants: $\epsilon \in (0, 0.5)$ $\delta, \alpha \in (0, \infty)$ $q, k_{max} \in \mathbb{Z}$ $k_s < k_{max}$
Initial value of variables: $f^\star \leftarrow 0$ $\mathbf{c} = \mathbf{v}$
while $k \leq k_{max}$ **do**
  $\mathbf{x}^{(i)} = \overline{RTTP}(\mathbf{c}, \mathbf{l}, \mathbf{u}, \mathbf{p})$
  $obj^{(i)} = \mathbf{v}^T \mathbf{x}^{(i)}$
  $B^{(i)} = \{b : x_b^{(i)} \in [1-\epsilon, 1] \quad \& \quad l_b = 0 \quad \& \quad u_b = 1\}$
  $f(\mathbf{x}^{(i)}) = obj^{(i)} + \delta * |B^{(i)}|$
  if $\mathbf{x}^{(i)} \in \{0,1\}^n$ **then**
    return $B^\star = B^{(i)}$
  end if
  if k is a positive multiple of k_s **then**
    $j^\star = \underset{j}{\text{argmax}} \{x_j^{(i)} \mid j \notin B^\star \bigcup B^{(i)}\}$
    $B^\star = B^{(i)} \bigcup \{j^\star\}$
    $\mathbf{c}(j^\star) = \mathbf{c}(j^\star) + q$
  else
    $c_p = c_p + \alpha * (x_p^{(i)})^2 \quad \forall p \in \mathbf{p}$
    if $f(\mathbf{x}^{(i)}) \geq f^\star$ **then**
      $B^\star \leftarrow B^{(i)}, \quad f^\star \leftarrow f(\mathbf{x}^{(i)}), \quad k \leftarrow k-1$
    end if
  end if
  $i \leftarrow i+1, \quad k \leftarrow k+1$
end while
if $|B^\star| = 0$ **then**
  $B^\star \leftarrow Strongbranching(\mathbf{l}, \mathbf{u}, \mathbf{v}, \mathbf{x}^{(0)}, \mathbf{p})$
end if

restricted problem ($\overline{RTTP}$) which gives the current iteration solution $\mathbf{x}^{(i)}$. Then the objective value $obj^{(i)}$ is calculated using the un-perturbed cost vector $\mathbf{v}$. After that the index of all near integer variables that are not yet fixed is found and stored in $B^{(i)}$. The potential function is then calculated using the objective value $obj^{(i)}$ and the number of near integer variables. A check is then made if the current solution is integer, if it is integer then $B^\star$ is set to the current $B^{(i)}$ and the algorithm is terminated.

If the current solution $\mathbf{x}^{(i)}$ is not integer then a new check is made whether a *spacer step* should be made. The spacer step consist of finding the index, denoted $j^\star$, of the largest fraction not yet in $B^\star$ or $B^{(i)}$. $B^\star$ is then set to the union of $j^\star$ and $B^{(i)}$. A bonus q is added to the element with index $j^\star$ in the perturbed cost vector $\mathbf{c}$.

If the check for spacer step failed then the perturbed cost vector is updated using the current solution $\mathbf{x}^{(i)}$. Then a new check is made if the current potential function $f(\mathbf{x}^{(i)})$ is larger than or equal to the previously largest potential function $f^\star$. If this is true then $B^\star$ is set to $B^{(i)}$, the value of the best potential $f^\star$ is set to be equal to the value of the current potential $f(\mathbf{x}^{(i)})$ and the step counter k is decreased by one. The last part of the loop consists of increasing both step counter i and k by one.

Once the main while loop has finished running a check is made if any potential fixes has been found. If no fixes has been found then one variable to fix is chosen by *strong branching*. The input to the strong branching algorithm is $\mathbf{l}, \mathbf{u}, \mathbf{v}, \mathbf{x}^{(0)}$ and $\mathbf{p}$.

Important aspects of generating potential fixes

This section will describe some of the more interesting parts of the algorithm. The most important things about algorithm 1 are *perturbation branching*, *spacer steps*, *potential function* and *strong branching*.

Perturbation branching is one of the main ideas behind rapid branching. The idea is that by perturbing the cost vector $\mathbf{c}$ it is possible to drive a solution towards integrality. The equation for the perturbed $\mathbf{c} = (c_p)$ is

$$c_p = c_p + \alpha \cdot (x_p^{(i)})^2 \tag{3.2}$$

where it is assumed that α is positive. Because of this, variables that are close to one get a larger relative bonus cost in comparison to variables that are closer to zero. Since the bonus costs are cumulative this will hopefully lead to a situation where some paths become more profitable than the rest. Then the solutions will probably become more integral as the number of iterations increase. The value of α controls the rate of change for $\mathbf{c}$. It may be possible to change the equation for updating the cost vector and still drive solutions towards integrality but the way it is defined in equation 3.2 seems sufficient for all intents and purposes.

The spacer step is done after the algorithm hasn't found a solution with a higher potential function for a number of iterations. The number of iterations without progress between spacer steps are k_s. The spacer step is introduced to get a change such that the algorithm may find new solutions with higher potential function values. During the spacer step the index of the largest fraction is found and denoted as $j^\star$. Then the revenue of the path with index $j^\star$ gets a large bonus cost q. This is likely to lead to the variable with index $j^\star$ being integer in subsequent iterations. There are some more rules that may be imposed on $j^\star$. It may be required that no variable corresponding to the same request is near integer, i.e $j^\star$ does not correspond to any request which has an index already in $B^{(i)}$.

The potential function f is calculated as

$$f(\mathbf{x}^{(i)}) = \mathbf{v}^T\mathbf{x}^{(i)} + \delta \cdot |B^{(i)}| \tag{3.3}$$

and here $\mathbf{v}$ is the unperturbed cost vector. The number of near integer variables is $|B^{(i)}|$ and the scalar δ is the weight for that. The near integer variables are the variables that are unfixed and within some interval around one. The indices are found by checking the fractional solution $\mathbf{x}^{(i)}$.

$$B^{(i)} = \{b : x_b^{(i)} \in [1-\epsilon, 1] \quad \& \quad l_b = 0 \quad \& \quad u_b = 1\} \tag{3.4}$$

For each iteration i a new solution $\mathbf{x}^{(i)}$ is found, which will result in a new potential function f and a new set of indices $B^{(i)}$. After the maximum number of iterations has been reached it is the set of indices $B^{(i)}$ that correspond to the iteration with the largest potential function $f^\star$ that is the output of the algorithm. The potential f is used to decide which branch to analyse further if multiple possible branches are found.

Strong branching is done if no set of variables to fix was found through perturbation branching. If the main loop in algorithm 1 is completed and $B^\star$ is still empty then it is necessary to find at least one possible variable to fix and that is done by strong branching. The algorithm for strong branching is found in algorithm 2 but the main idea is to choose a locally optimal strategy, i.e find the variable that if fixed gives the largest

objective value. It would be possible to test all unfixed variables but in the current implementation only a subset of all variables is tested.

The input for the strong branching algorithm is $\mathbf{l}, \mathbf{u}, \mathbf{v}, \mathbf{x}^{(0)}$ and $\mathbf{p}$. When the restricted problem was solved using the unperturbed cost vector $\mathbf{v}$ the solution found was $\mathbf{x}^{(0)}$. This solution is used in the strong branching algorithm to find a set of potential fixes. The parameter n is the maximum number of variables to analyse. It is the n largest fractions in $\mathbf{x}^{(0)}$ that are analysed. For each of those fractions the variables are fixed to one and then the restricted problem is solved. After all variables have been tested the index of the variable that gives the best objective value is chosen as a potentially fixable variable. $B^\star$ is set to this index and is the output of the algorithm 2. The same index $B^\star$ is then also the output of algorithm 1.

Algorithm 2 Generating an index of a variable to fix by strong branching, $B^\star$

output $B^\star$
input $\mathbf{l}, \mathbf{u}, \mathbf{v}, \mathbf{x}^{(0)}, \mathbf{p}$
parameters n
$S^\star = \{\text{index of n largest fractions not yet fixed in } \mathbf{x}^{(0)}\}$
for $k = 1 : n$ **do**
  $\mathbf{h} = \mathbf{l}$
  $i = S^\star(k)$
  $\mathbf{h}_i = 1$
  $\mathbf{x}^{(k)} = \overline{RTTP}(\mathbf{v}, \mathbf{h}, \mathbf{u}, \mathbf{p})$
  $obj^{(k)} = \mathbf{v}^T \mathbf{x}^{(k)}$
end for
$k^\star = \underset{k}{\text{argmax}}(obj^{(k)})$
$B^\star = S^\star(k^\star)$

3.2.2 Applying fixes

Once a set of variables $B^\star$ that are potentially fixable is found they are analysed to determine what variables are fixable without reducing the objective value by to much. In the following section algorithm 3 will be presented. The format is the same as in section 3.2.1 with a short summary first and then an exposition on the more important aspects of the algorithm.

Algorithm 3 Examine the set $B^\star$ to determine what variables to fix

output $\mathbf{l}, \mathbf{u}, obj^{(0)}$
input $B^\star, \mathbf{l}, \mathbf{u}, \mathbf{v}, \mathbf{p}, \boldsymbol{\mu}, obj^{(0)}, R$
parameters $\kappa \in (0, 1)$
variables $indicator = false \quad B^c = \{\}$
$\mathcal{L}_p = v_p - \sum_{\forall (b,t) \in \mathcal{V}} \mu_{bt} * \delta^p_{bt} \quad \forall p \in \mathbf{p}(B^\star)$
Sort $B^\star$ by decreasing $\mathcal{L}$ values;
while $indicator = false$ **do**
  $\mathbf{l}_{temp} = \mathbf{l}, \quad \mathbf{u}_{temp} = \mathbf{u}$
  $\mathbf{l}_{temp}(B^\star) = 1, \quad \mathbf{u}_{temp}(B^c) = 0$
  $t = obj^{(0)}(1 - \kappa * |B^\star|/R)$
  $\mathbf{x}_{temp} = \overline{RTTP}(\mathbf{v}, \mathbf{l}_{temp}, \mathbf{u}_{temp}, \mathbf{p})$
  $obj_{temp} = \mathbf{v}^T \mathbf{x}_{temp}$
  if $obj_{temp} \geq t$ or $|B^\star| = 1$ **then**
    $indicator = true$
    $obj^{(0)} = obj_{temp}$
  else
    Move half of indices in $B^\star$ to B^c;
    $n = |B^\star|$
    $B^c = [B^c \quad B^\star(\lfloor n/2 \rfloor + 1 : n)]$
    $B^\star = B^\star(1 : \lfloor n/2 \rfloor)$
  end if
end while
$\mathbf{l} = \mathbf{l}_{temp}, \quad \mathbf{u} = \mathbf{u}_{temp}$

Short summary of applying potential fixes

The input for this algorithm is the set of potential fixes $B^\star$, current upper bounds $\mathbf{u}$, current lower bounds $\mathbf{l}$, stored paths $\mathbf{p}$, lagrange multiplier vector $\boldsymbol{\mu}$ from bundle phase, objective value $obj^{(0)}$ from earlier iterations and the number of requests R. The output of the algorithm is the objective value associated with the fixes applied and a new set of bounds $\mathbf{l}$ and $\mathbf{u}$ for all variables. The only parameter of the algorithm is κ which is used when calculating the *target value* t. The variables are an indicator for when a set of fixes to apply has been found and the set B^c which is the indices that has been removed from $B^\star$ during the algorithm.

The first step of the algorithm is to calculate the *lagrangian revenue* $\mathcal{L}_p$ for all indices in $B^\star$. Then $B^\star$ is sorted by decreasing lagrangian revenue. After this the main loop of the algorithm starts and it is continued until the indicator is true. The first thing in the loop is that temporary bounds are created. The variables with indices in $B^\star$ are fixed to one and the variables with indices in B^c is fixed to zero.

Then the target value is calculated as a fraction of the objective value found during earlier iterations. The next step is to find a solution $\mathbf{x}_{temp}$ for the restricted problem using the temporary bounds. The objective value obj_{temp} is calculated using $\mathbf{x}_{temp}$ and then a check is made if the target value t has been reached or if there is only one potentially fixable variable left in $B^\star$. If this is true then the indicator is changed to true, the objective value $obj^{(0)}$ is updated and the currently temporary bounds become the current bounds. Otherwise half of the indices in $B^\star$ are moved to B^c. The indices moved from $B^\star$ are those associated with lowest lagrangian revenue. After this the while loop is restarted and it continues until there is either only one variable left to fix or the target value is reached for some set of fixes.

Important aspects of applying potential fixes

The most important aspects are the sorting of $B^\star$ by decreasing lagrangian revenue $\mathcal{L}$, the target value t and how indices in $B^\star$ are handled if the target value is not reached.

The lagrangian revenue $\mathcal{L}$ is a sort of reduced cost. It is calculated for a path p as:

$$\mathcal{L}_p = v_p - \sum_{\forall (b,t) \in \mathcal{V}} \mu_{bt} \cdot \delta_{bt}^p \tag{3.5}$$

which can be written on vector form as:

$$\mathcal{L}_p = v_p - \boldsymbol{\mu}^T \mathbf{d}^p \tag{3.6}$$

This should be calculated for all paths p with an index in $B^\star$. The constant v_p is the revenue of the path and can be found in $\mathbf{v}$. The vector $\mathbf{d}^p$ is the capacity consumption in the block-time space. The lagrange multipliers $\boldsymbol{\mu}$ are the last multipliers found during the bundle phase. The sorting of $B^\star$ by lagrangian revenue is a heuristic way of ordering them. For some of the cases considered in chapter 4 where no lagrangian multipliers are provided the indices are ordered by decreasing costs v_p.

The target value t is calculated as:

$$t(B^\star) = obj^{(0)}(1 - \kappa \cdot |B^\star|/R) \tag{3.7}$$

The target value is a fraction of the original objective value found during the bundle phase. The final target value t_{end} once each request r has a chosen path is

$$t_{end} = obj^{(0)}(1 - \kappa \cdot |R|/R) = obj^{(0)}(1 - \kappa) \tag{3.8}$$

From this it is clear that κ is the acceptable deviation from the original objective value when finding an integer solution. Here a choice may be made for what is considered the original objective value. Either $obj^{(0)}$ is the objective value from the solution found from the original linear problem where no variables have been fixed yet. Another choice would be to use the objective value associated with the last set of fixes that was applied. The pseudo code for this choice is found in algorithm 3.2.2.

During algorithm 3 if the target value is reached then the variables in $B^\star$ are fixed to one and the variables in B^c are fixed to zero. If the target value is not reached then a backtracking in the branch and bound tree is performed. This is called a binary backtracking mechanism in [5]. What happens is that half of the indices in $B^\star$ are moved to B^c. This is why the indices in $B^\star$ are sorted by lagrangian revenue. The result of moving the indices to B^c is that the part of the tree that was examined in the previous iteration is pruned. This is visualized in figure 3.3 and 3.4.

In figure 3.3 it is assumed that initially $B^\star = \{1, 2, 3, 4\}$ where the variables have been ordered by decreasing lagrangian revenue. At first it is examined if fixing all four variables to one gives a restricted problem such that when solved the objective value reaches the target value. Let's assume that the target value is not reached. Then the worst half of the indices in $B^\star$ are moved to B^c. Since those moved variables will remain in B^c until a set of fixes to apply has been found many branches of the branch and bound tree can be pruned.

In our example $B^\star = \{1, 2\}$ and $B^c = \{3, 4\}$. Branches that conflict with this partitioning are pruned and the possible branches left to explore are seen in figure 3.4. If

the target value once again is not reached then there will be only one index left in $B^\star$ which would then be fixed. This small example shows how the algorithm quickly prunes many branches of the branch and bound tree while producing a solution. The downside is that many branches go unexplored and the true optimal solution may lie in a pruned part of the tree. It would be possible to prune the whole left side of the branch and bound tree since at least one fix is always applied when algorithm 3 is called. Each time algorithm 3.2.2 is called at least half of the remaining unfixed branch and bound tree will be pruned.

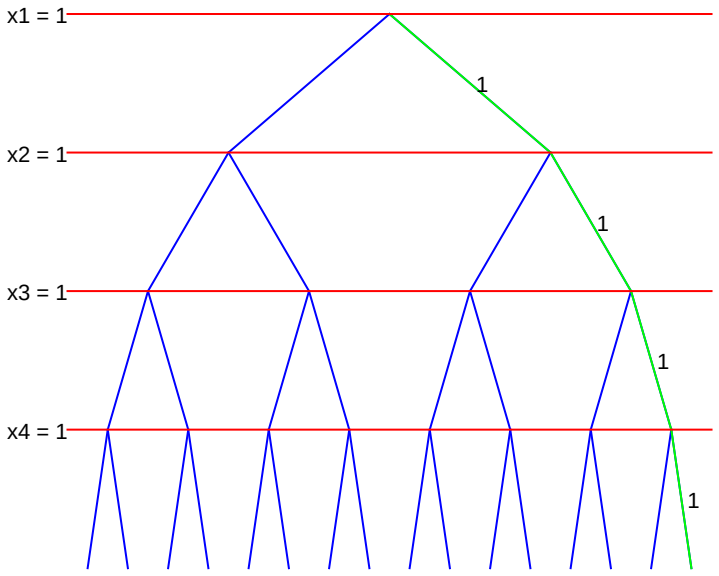

FIGURE 3.3: Branch and bound tree, $B^\star$ contain 4 indices.

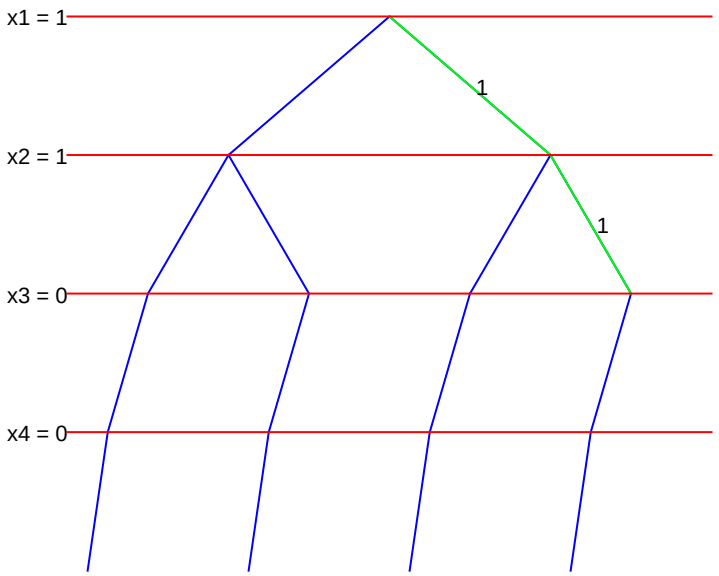

FIGURE 3.4: Branch and bound tree, $B^\star$ contain 2 indices.

Chapter 4

Application of algorithm

4.1 Introduction

The algorithm described in earlier chapters is supposed to be applied on data from the real world. In order to test if it works and also test how well it performs in comparison to standard solvers some made up problems have also been created. This chapter will describe how problems are generated, how they are solved, how they differ from each other and the solutions found.

4.2 Randomly generated set of paths without desired departure times

In this section a problem where paths are generated randomly will be described. The numbers are not truly random numbers but pseudo randomized using Matlab's inbuilt function "randi(n)". By using the same *seed n* before starting to randomize it is possible to repeatedly solve the same problem. The main objective in this thesis is to fast find a "good" integer solution to (RTTP) defined in 2.12. When trying to find an integer solution to (RTTP) a series of of linearly relaxed restricted problems ($\overline{RTTP}$), defined in 2.13, are solved. The solutions of these LP problems are found using Matlab's inbuilt mixed-integer linear programming solver "intlinprog()". For this solver one specifies what variables are integer and if no integer variables are specified then simply an LP problem is solved. An input for the solver is also the bounds of variables which makes

applying rapid branching easy. While finding a solution for rapid branching no variables are declared as integer, the bounds **l** and **u** are used to show what variables are fixed.

To compare the solution from rapid branching with conventional techniques one solves the same initial problem using the same solver "intlinprog()" but declares that all variables need to be integer.

Since we want the comparison to be fair between conventional techniques and rapid branching some original options of "intlinprog()" is changed. For rapid branching a tolerance κ is used in 3.7 for calculating the target value. But due to how rapid branching works, where if only one potential fix remains, it is imperatively applied it is possible to end up with a solution that does not reach it's target value. So what the tolerance is for rapid branching is a bit fuzzy but it is close to κ. The tolerance for finding an integer solution using "intlinprog()" is therefore set to κ. While the solver tries to find a integer solution it keeps track of the upper and lower bound on the objective function value. If the relative difference between the upper and lower bound is less than κ then a satisficingly good solution has been found. The revenues of the paths have been scaled such that the objective value takes a value between zero and one.

The restricted problems are described in section 2.5. Since the paths are already explicitly generated during the integer solution phase, applying the algorithm to a set of randomly generated paths is interesting. When talking about paths in this section it is actually the matrix of capacity consumption of a path that is meant. The capacity consumption of a randomly generated path can be seen in figure 4.1.

4.2.1 Description of generated paths

The paths used in the restricted problem 2.13 are generated randomly by a set of rules. For each request r there is a null-path that corresponds to not scheduling that request. In the plots that would correspond to a horizontal line, i.e the request remains in the same block for every time interval. The revenue v_p associated to the null-path is set to zero. For every other path the departure time t_d is a random integer $t_d \in \{1, \ldots, \omega T\}$ where $\omega \in (0, 1)$ is a fraction such that the maximum starting time is an integer. An

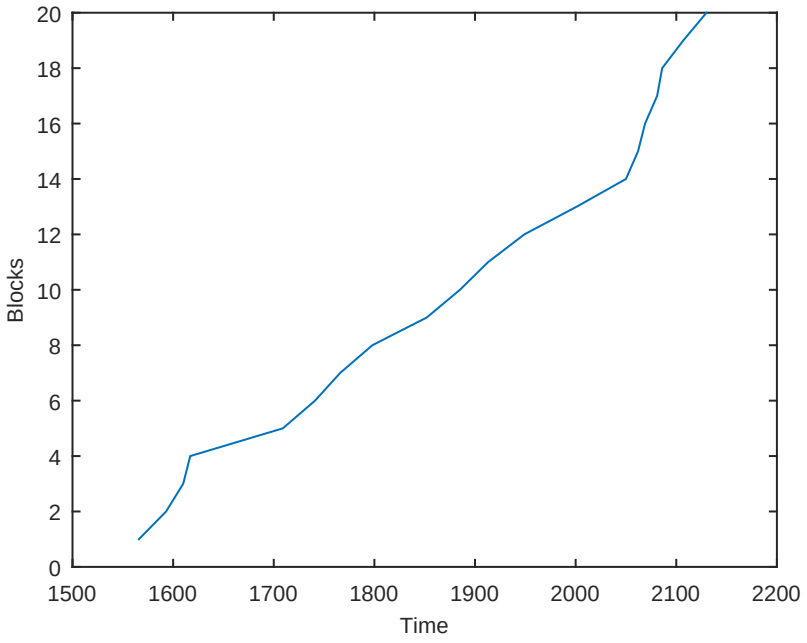

FIGURE 4.1: Capacity consumption of one request

example for ω used is $\frac{5}{8}$. The departure time t_d corresponds to the first capacity consumption element d^p_{bt} that is equal to one.

Between the starting block and ending block there are two types of blocks. The normal type of block has a capacity c_b of one request per time interval. Every fourth block is considered as a station where the capacity is larger than one. In plot 4.5 the capacity c_b at stations are 5.

There is a maximum number of time-intervals that a train may remain in each block. The maximum time a train may stay at a station-block is longer than the time it may remain in a normal block. The number of time intervals t_b that a train remains in a block b is a random integer such that $t_b \in \{1, \dots t_{max}\}$ where t_{max} depends on the type of block. If T is the total number of time intervals then an example is to use $t_{max} = \frac{T}{60}$ for normal blocks and $t_{max} = \frac{T}{30}$ for stations.

Once the final block has been reached the train is assumed to consume zero capacity. This could be interpreted as the final block having infinite capacity or trains reaching

the final station being redirected to individual sidetracks. There is a revenue v_p associated with each path. When solving a real life (TTP) the revenue of a path most likely depend on the departure and arrival time but for the randomly generated paths the revenue v_p is just a random number $v_p \in (\frac{1}{R^2}, \frac{1}{R}]$. This is to scale the objective such that the optimal value of the linear solution most likely is 1 and the optimal value of the integer solution hopefully is within some tolerance of 1 set by the user. If the optimal value of the linear solution is one then it is also easy to see what the gap percentage is by just looking at the integer solution optimal value.

If the total number of blocks is B then half of the paths have the first block as starting station and the rest have block B as starting station. This is to make the problem more difficult to solve and impose that trains need to meet at stations.

As for the size of a typical problem. If the total number of requests is R the number of paths generated for each request is R^2, e.g $R = 20$ then there are 400 generated paths for each request. This means there are a total number of $R^3 = 8000$ paths generated. For the solution seen in figure 4.5 the number of blocks are $B = 20$ and the number of time intervals are $T = 2880$. This corresponds to dividing a day into $30s$ time-intervals.

Let I denote the identity matrix. The linearly relaxed ($\overline{RTTP}$) found in 2.13 can then be expressed in matrix notation as:

$$\begin{aligned} \max_{\mathbf{x}} \quad & \mathbf{v}^T\mathbf{x} & & & \\ \text{s.t.} \quad & A\mathbf{x} & \leq & \mathbf{b}, & (i) \\ & I\mathbf{x} & \leq & \mathbf{u}, & (ii) \\ & -I\mathbf{x} & \leq & \mathbf{l}, & (iii) \\ & A_{eq}\mathbf{x} & = & \mathbf{b}_{eq} & (iv) \end{aligned} \tag{4.1}$$

The objective here is to find a fractional solution $\mathbf{x}$ such that the profit is maximized. The columns in A_1 are the capacity consumption matrices reshaped into vectors. The block-times (b, t) have been labeled in some way , e.g number the block-times (b, t) $\{(1,1),(1,2),\dots(1,T),(2,1)\dots,(B,T)\}$ as $\{1,2,\dots T,T+1,\dots,BT\}$. The elements in $\mathbf{b}$ are the corresponding capacities of block-times. Constraint (ii) handles the constraints

that $x_p \leq u_p$ and constraint (iii) that $x_p \geq l_p$. The equality constraint (iv) is so that for each request r the variables in $\mathbf{x}$ that correspond to paths for r sum up to one. If there was two request and two possible paths for each request then constraint (iv) would look like this:

$$\mathbf{A}_{eq}\mathbf{x} = \mathbf{b}_{eq} \quad \Leftrightarrow \quad \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} x_{11} \\ x_{12} \\ x_{21} \\ x_{22} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Where for x_{ij} the i represent the request and the j is the index of the path among all paths feasible for request i. It is possible to combine constraints $(i), (ii)$ and (iii) into one large inequality matrix:

$$\mathbf{M}\mathbf{x} \leq \mathbf{n} \quad \Leftrightarrow \quad \begin{bmatrix} \mathbf{A} \\ \mathbf{I} \\ -\mathbf{I} \end{bmatrix} \mathbf{x} \leq \begin{bmatrix} \mathbf{b} \\ \mathbf{1} \\ \mathbf{0} \end{bmatrix}$$

4.2.2 Solutions found and results

Once the matrices in 4.1 have been properly defined the algorithms found in section 3.2.1 and 3.2.2 are applied to the problem. Since random numbers are pseudo-random, different seeds create different paths which means different problems. For each number of requests 5 different seeds are used. The seeds are $n \in \{1, R, 5R, 10R, 20R\}$ The number of requests examined are $R \in \{5, 10, 15, 20\}$. When trying to solve for 25 requests the problem becomes too large and the integer solver fails to find a solution. This is since the Matlab inbuilt solver is used for the sub-problems. For larger problems a solver that is more specialized for large-scale problems are needed, e.g a bundle solver.

Plots of the solutions found for different number of requests are found in figure 4.2, 4.3, 4.4 and 4.5. For $R \in \{5, 10, 15\}$ the solver manages to find and schedule a path for each request. For $R = 20$ the solver only finds paths for 19 requests and for one train the null path is chosen. It is also clear from the plots that trains indeed meet at

stations along the track. It is important to mention that the plots consist of linear segments between block-times for when a train first enters a new block. The lines may thus not represent the true position of the train along the track. From the figures it seems like the solver mainly schedules trains from one direction first and schedule trains later in the other direction. This must be due to difficulty finding paths that meet at stations.

It is interesting to study the total solution time as a function of the size of the

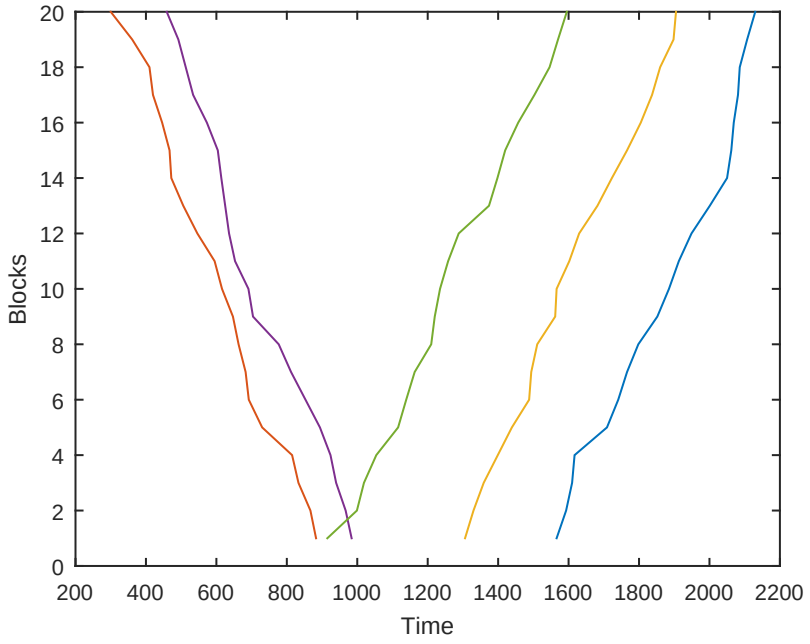

FIGURE 4.2: Capacity consumptions for 5 requests, no desired departure time

problem. This is plotted in figure 4.6. The size used here is the total number of generated paths which is R^3. The total number of generated paths are R^3 since for each request r there are R^2 number of paths generated and there is a total number of R requests.

Another interesting thing is to compare an integer solver with rapid branching. The comparison for a gap tolerance of 20% can be seen in table 4.1. From the table one can see that for the number of requests tested here the dedicated integer solver is faster than using rapid branching. Some comments on the values.

For all of the above solutions the tolerance gap for rapid branching has been 0.2, i.e 20%. In table 4.2 the objective value of the linear solution, the objective value of the

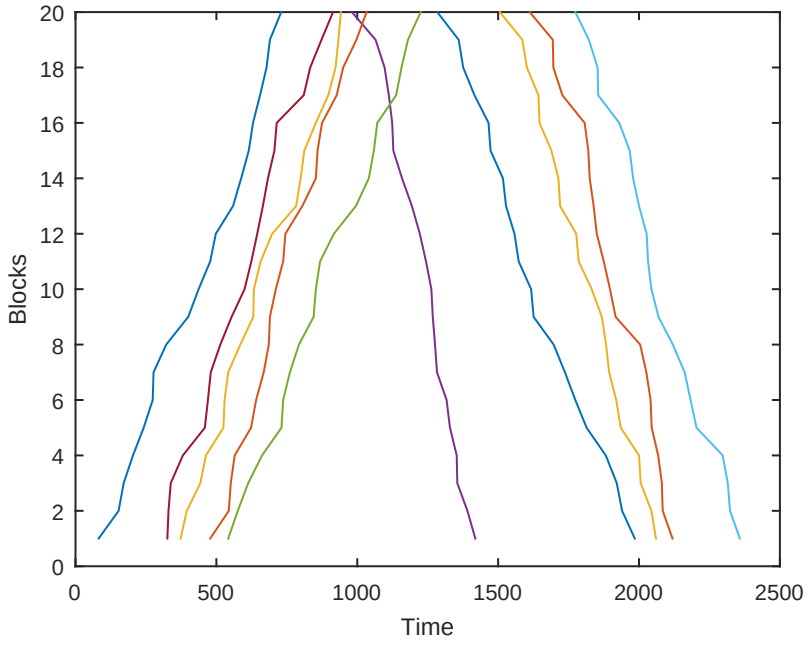

FIGURE 4.3: Capacity consumptions for 10 requests, no desired departure time

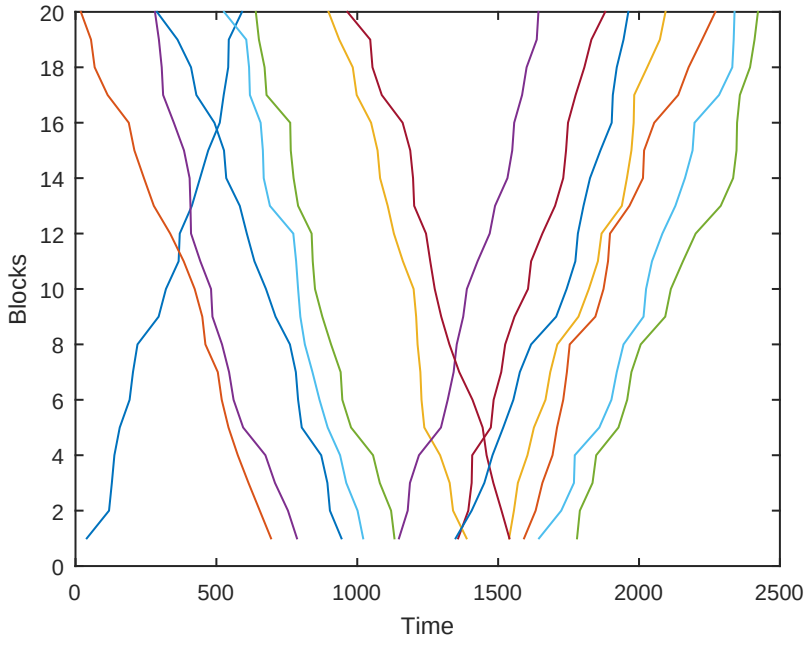

FIGURE 4.4: Capacity consumptions for 15 requests, no desired departure time

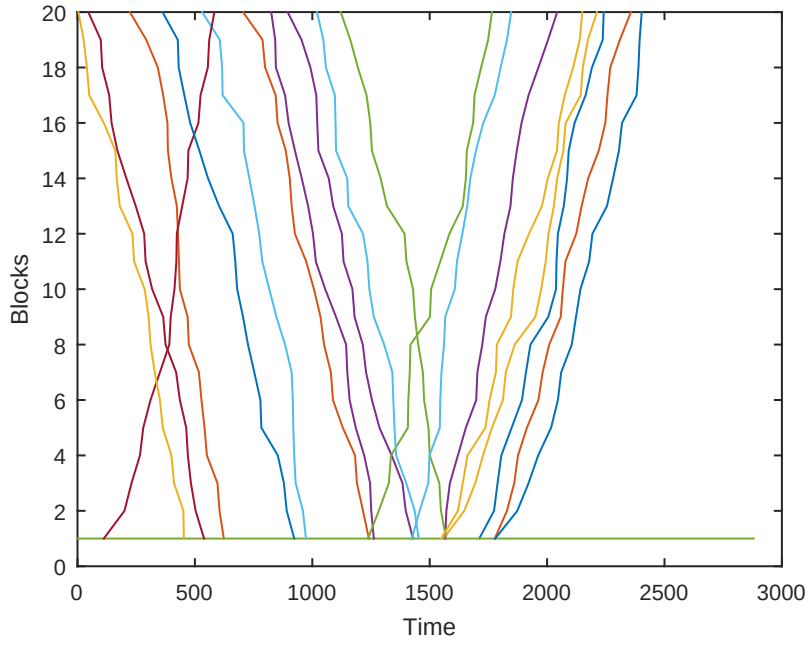

FIGURE 4.5: Capacity consumptions for 20 requests, no desired departure time

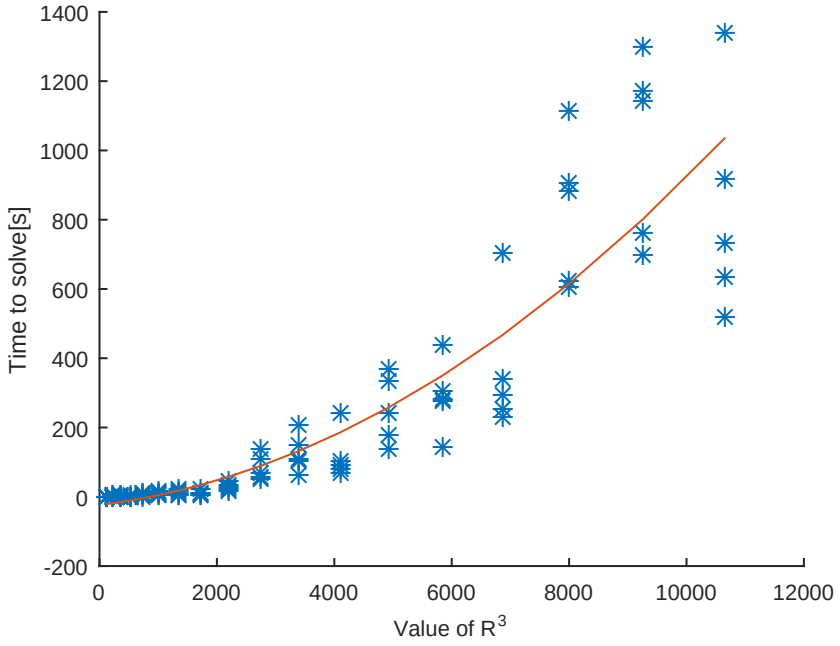

FIGURE 4.6: Fitted plot for rapid branching time vs total number of generated paths

Requests	Average time Integer Solver [s]	Average time Rapid Branching [s]
5	0	0
10	2	8
15	24	129
20	168	829

TABLE 4.1: Comparison of average time for integer solver and average time for rapid branching when requests have no desired departure time.

κ	Linear Obj	RB Obj	RB Time [s]	IS Objective	IS Time[s]
0.05	1	0.893	109	0.982	31
0.05	1	0.831	168	0.942	44
0.05	1	0.907	107	0.907	24
0.05	1	0.920	209	0.947	26
0.05	1	0.853	83	0.987	24
0.1	1	0.893	113	0.982	32
0.1	1	0.853	171	0.849	41
0.1	1	0.907	110	0.907	24
0.1	1	0.920	215	0.947	26
0.1	1	0.902	84	0.987	24
0.2	1	0.893	109	0.796	18
0.2	1	0.911	145	0.849	40
0.2	1	0.929	101	0.778	18
0.2	1	0.916	206	0.764	19
0.2	1	0.840	60	0.987	24

TABLE 4.2: Average objective value of linear solver, integer solver and rapid branching algorithm for 15 requests given κ and 5 different seeds. Requests have no desired departure time.

integer solver and the objective value of the rapid branching algorithm are presented for different gap tolerances κ. This has been investigated for the case of 15 requests.

4.3 Randomly generated set of paths with desired departure times

For the problems in this section the paths are generated almost the same way as in section 4.2 except each request has a desired departure time. The solvers used are the same as in section 4.2 and the tolerances are defined in the same way.

4.3.1 Description of paths generated

In this section the departure time t_d of each request belongs to some set of time intervals $t_d \in T_{I,r}$. If all discrete time intervals in $\mathcal{T}$ are indexed $\{1, 2, \dots T\}$ then I have chosen the set of time intervals $T_{I,r}$ such that the departure time of request one $t_{d,1} \in \{1, 2, \dots N\}$ and departure time of request two $t_{d,2} \in \{N+1, N+2, \dots 2N\}$ and so on. The set of time intervals for each request is therefore disjoint in this section. The value on N is unless otherwise stated 100, i.e each request has a departure window of 50 minutes.

The preferred departure time t_{pref} is in the middle of each set of time intervals. For request one $t_{pref,1} = N/2$ and for request two $t_{pref,2} = N + N/2$ and so on. The preference of certain times are modelled by using a triangular function for calculating the revenue v_p of a path p. The function used is

$$v_p = \frac{N - 2 \cdot |t_d - t_{pref}|}{N \cdot R} \tag{4.2}$$

The denominator is there to scale the revenues such that the upper bound on the objective function value is one. Ideally one would like to include arrival time when calculating the revenue of a path but that has not been implemented in this section.

4.3.2 Solutions found and results

The number of requests examined are $R \in \{5, 10, 15, 20\}$. For each number of requests several different seeds are used. The seeds are $n \in \{1, R, 5R, 10R, 20R\}$. The plots for solutions when the seed $n = 1$ are found in figure 4.7, 4.8, 4.9 and 4.10. All requests are scheduled for $R \in \{5, 10, 15\}$ but for $R = 20$ the null-path are chosen for two requests.

In the figures one can see that the requests start occupying block one or block 20 at times that are a multiple of hundred. This the earliest possible departure time and should not be confused with the actual departure time. The definition of departure time I use is the time when the request starts to occupy the next block, i.e block 2 or block 19. It is easiest to observe in figure 4.7 that departure times are close to $\{50, 150, 250, 350, 450\}$, which is what we expect from how we defined the revenues of paths found in 4.2, since a departure times close to those numbers maximize the revenue.

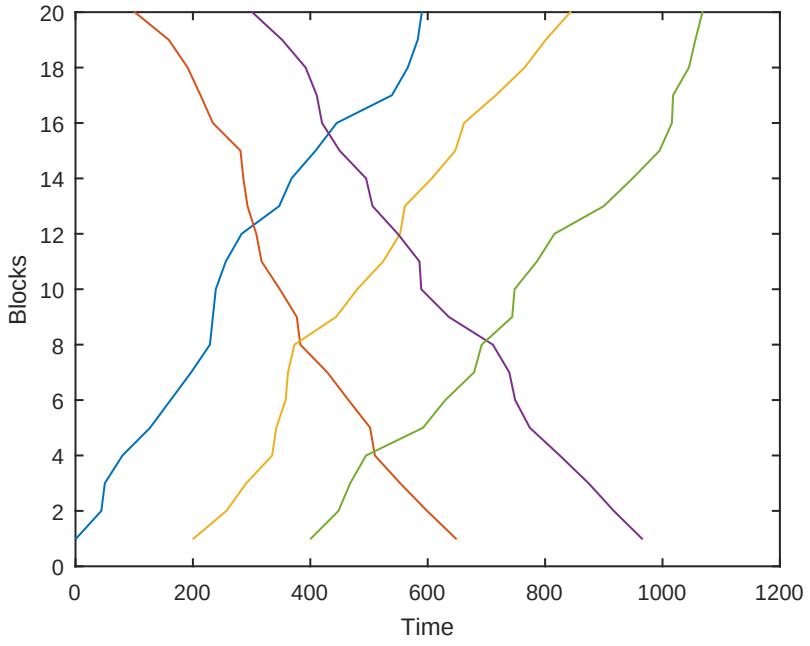

FIGURE 4.7: Capacity consumptions for 5 requests, each request has a desired departure time

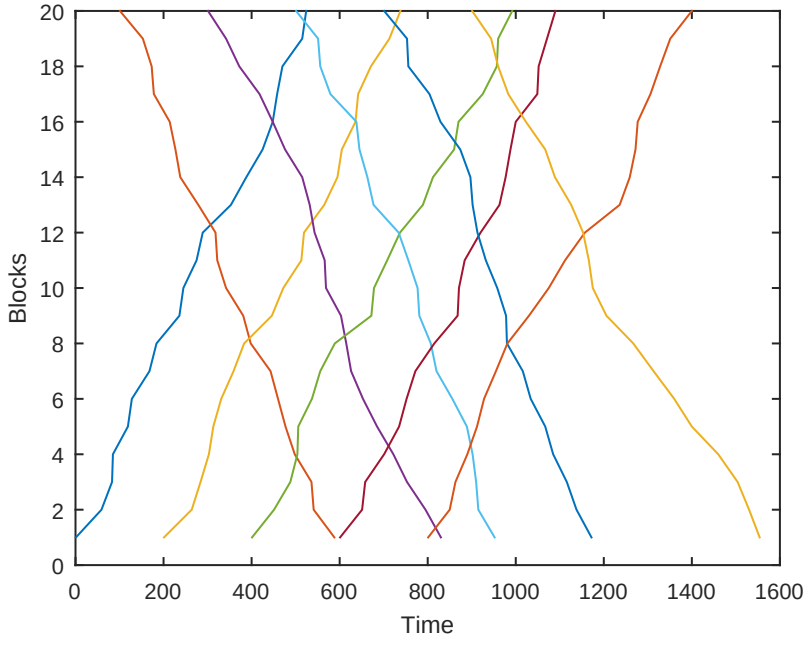

FIGURE 4.8: Capacity consumptions for 10 requests, each request has a desired departure time

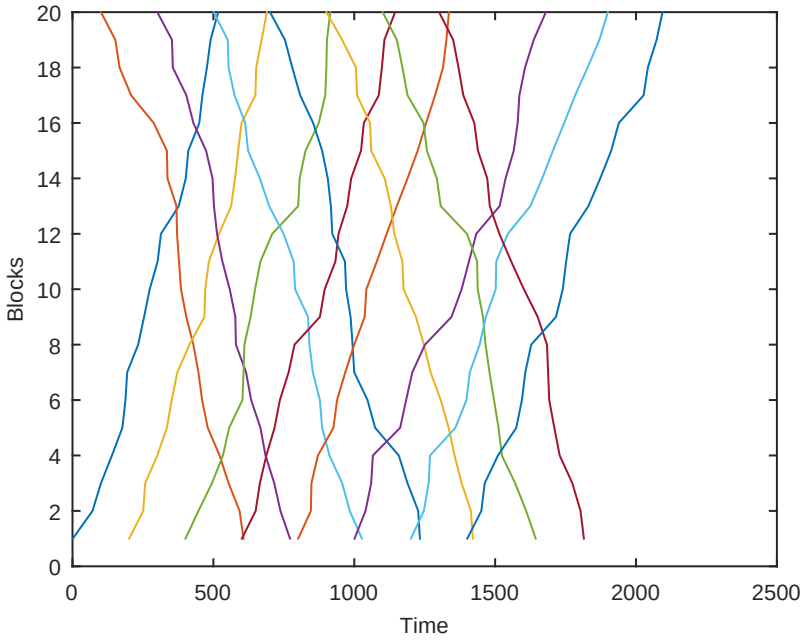

FIGURE 4.9: Capacity consumptions for 15 requests, each request has a desired departure time

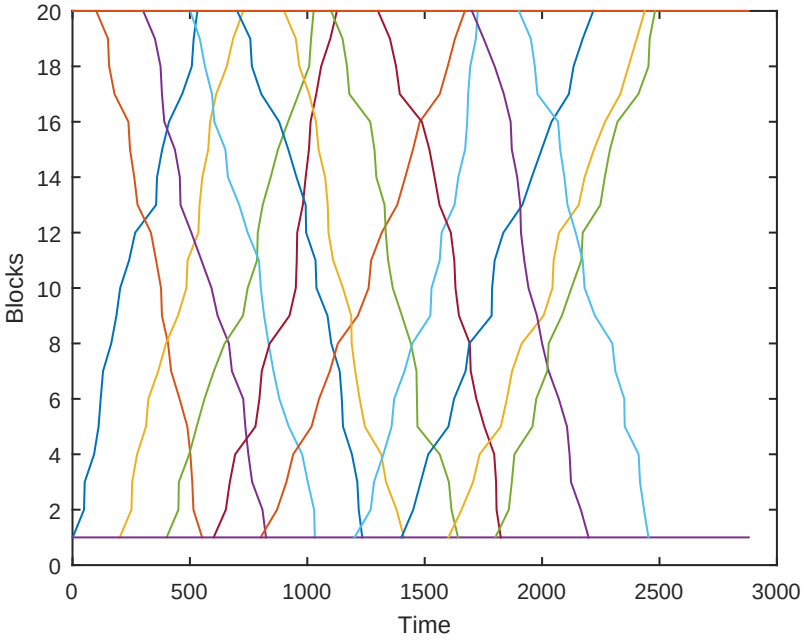

FIGURE 4.10: Capacity consumptions for 20 requests, each request has a desired departure time

The time to solve as a function of the total number of generated paths can be found in figure 4.11.

In table 4.3 a comparison is made of average time to solve (TTP) using the stan-

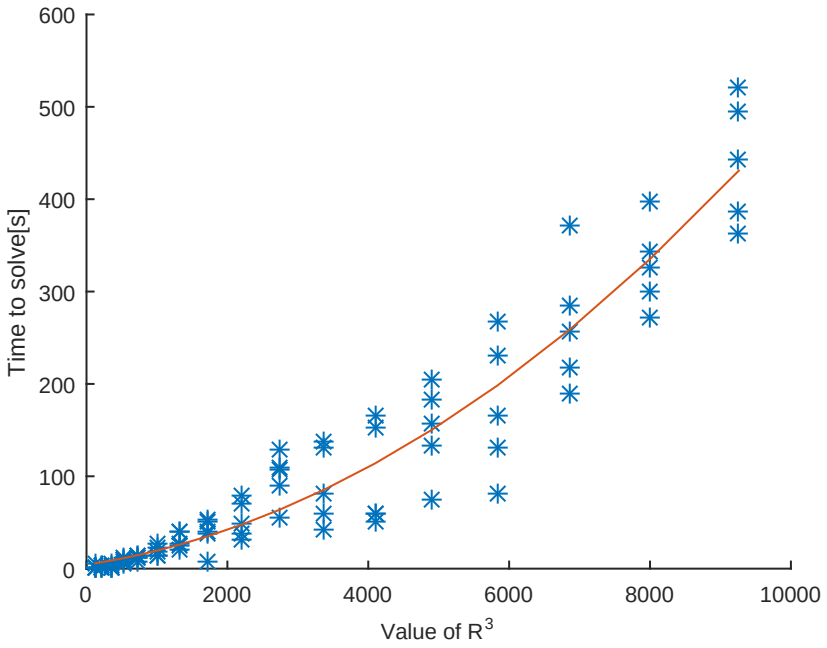

FIGURE 4.11: Fitted plot for rapid branching time vs total number of generated paths, each request has a desired departure time

dard integer solver and rapid branching for $\kappa = 0.2$. From this table we see that rapid branching is much faster than the inbuilt integer solver. This result is opposite to the one in the previous section where the integer solver was faster than rapid branching. Taking a desired departure time into account makes generated paths in this section more representative of real world cases. That rapid branching is faster than the standard integer solver in this section is a really important result.

If the tolerance κ is changed when the number of requests are kept constant to 10 a comparison of average time to find a solution using rapid branching and the inbuilt Matlab integer solver can be found in table 4.4. In this table one can observe that rapid branching is faster than the integer solver independent on the value of κ. It also shows that the tolerance is more important for the integer solver solution than the rapid branching solution. The average objective values for different values on κ can be found

Requests	Average time Integer Solver [s]	Average time Rapid Branching [s]
5	0	1
6	0	2
7	1	2
8	11	8
9	118	12
10	290	19
15	3027	59

TABLE 4.3: Comparison of average time for integer solver and average time for rapid branching when requests have a desired departure time.

κ	Linear Obj	RB Obj	RB Time [s]	IS Objective	IS Time [s]
0.05	0.969	0.720	25	0.902	239
0.05	0.967	0.872	28	0.858	602
0.05	0.971	0.936	22	0.914	13
0.05	0.967	0.788	21	0.890	290
0.05	0.982	0.830	24	0.888	340
0.1	0.969	0.860	15	0.802	238
0.1	0.967	0.872	28	0.776	610
0.1	0.971	0.936	22	0.804	13
0.1	0.967	0.916	15	0.890	289
0.1	0.982	0.798	17	0.808	338
0.2	0.969	0.850	15	0.664	232
0.2	0.967	0.872	28	0.702	578
0.2	0.971	0.936	22	0.804	13
0.2	0.967	0.916	15	0.656	291
0.2	0.982	0.798	17	0.808	338

TABLE 4.4: Objective value of linear solver, integer solver and rapid branching algorithm for 10 requests given κ and 5 different seeds. Requests have a desired departure time.

κ	Average obj Linear Solver	Average obj Rapid Branching	Average obj Integer Solver
0.05	0.971	0.829	0.890
0.1	0.971	0.876	0.816
0.2	0.971	0.874	0.727

TABLE 4.5: Comparison of average objective value for integer solver and rapid branching for 10 requests and different κ values. Requests have a desired departure time.

in table 4.5. In this table one can observe that the lowest tolerance gives the worst average objective value for rapid branching. This is kind of peculiar behaviour but it seems like setting κ to a moderate value is best.

Chapter 5

Conclusion

In this thesis the algorithm rapid branching applied to the Train Timetable Problem(TTP) has been described. It has been shown that using rapid branching can be both slower than using conventional algorithms, see section 4.2.2, and it can be faster than conventional algorithms, se section 4.3.2. The case that is most similar to the real life is described in section 4.3.2 where rapid branching is shown to yield satisfactory high quality solutions much faster than Matlab's inbuilt integer solver.

The algorithm and the pseudo code for rapid branching is discussed and presented in chapter 3. In this thesis rapid branching was implemented and tested on problems meant to be similar to problems in real life. Rapid branching which combines several techniques to solve large-scale integer mathematical optimization problems was mainly developed by Steffen Weider, see [1].

If we return to the questions posed at the end of chapter 1 we can now answer them. Rapid branching is certainly applicable on (TTP). This is shown by the results in chapter 4. Rapid branching is fast for the problem instances examined in this thesis, which also can be seen in the results in previous sections. The solutions from rapid branching is at least comparable to the solutions found with Matlab's inbuilt integer solver. In many cases the objective value using rapid branching is higher. Changing the tolerance κ does not seem to change the final objective value by much. Setting the tolerance κ too low is even disadvantageous in the problems examined in this thesis. How rapid branching compares to conventional solvers depend on the problem one wants to solve.

For small problems there is not much difference in objective values and solving time. When the problem size increase rapid branching is faster and the objective value are in many cases better than the objective value from conventional solvers.

What remains to do is to test rapid branching on real world data where paths are generated using the bundle method and shortest path algorithms. This is an ongoing process that will not be presented in this thesis. There is a section in chapter 2 where bundle method is discussed but it has not been implemented to solve the sub-problems yet so there is no results to present for it. So in the future I hope that the algorithm for rapid branching that has been presented in this thesis has been implemented to work in conjunction with the bundle method and shortest path algorithms to produce feasible and approximately optimal solutions to (TTP) for real world data. If there is a solver that can solve larger sub-problems than the ones found in section 4.2.1 and 4.3.1 rapid branching should be able to be applied on even larger problems.

Another thing that would be interesting to test is how a desired arrival time for each request impacts the solutions. One could also see what happens if the desired departure and arrival times are not evenly spaced and if there are faster or slower trains moving along the track, i.e a train may want to pass another train moving in the same direction. All of these suggestions will be included when the algorithm is implemented for real life data but that will not be presented in this thesis and is left for the future.

Bibliography

[1] Steffen Weider. *Integration of Vehicle and Duty Scheduling in Public Transport*. PhD thesis, 2007. URL http://dx.doi.org/10.14279/depositonce-1672.

[2] Ralf Borndörfer, Andreas Löbel, Markus Reuther, Thomas Schlechte, and Steffen Weider. Rapid branching. *Public Transport*, 5(1):3 – 23, 2013.

[3] U. Brännlund, P. O. Lindberg, A. Nou, and J. E. Nilsson. Railway timetabling using lagrangian relaxation. *Transportation Science*, 32(4):358–369, November 1998.

[4] Krzysztof C. Kiwiel. A cholesky dual method for proximal piecewise linear programming. *Numerische Mathematik*, 68(3):325–340, 1994. ISSN 0029-599X. doi: 10.1007/s002110050065. URL http://dx.doi.org/10.1007/s002110050065.

[5] Ralf Borndörfer, Thomas Schlechte, and Steffen Weider. Railway Track Allocation by Rapid Branching. In Thomas Erlebach and Marco Lübbecke, editors, *10th Workshop on Algorithmic Approaches for Transportation Modelling, Optimization, and Systems (ATMOS'10)*, volume 14 of *OpenAccess Series in Informatics (OASIcs)*, pages 13–23, Dagstuhl, Germany, 2010. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-939897-20-0. doi: http://dx.doi.org/10.4230/OASIcs.ATMOS.2010.13. URL http://drops.dagstuhl.de/opus/volltexte/2010/2746.

TRITA -MAT-E 2016:02
ISRN -KTH/MAT/E--16/02--SE