Mixed Integer Linear Programming for the Train Timetabling Problem

Candidate Number: 1026167

University of Oxford

MSc Computer Science

Trinity Term 2021

Acknowledgements

I would like to express my gratitude to my supervisor, Professor Stephen Duncan, for providing me with his competent guidance and help during this project. Furthermore, I would like to thank Professor Duncan for his stunning patience with my lack of organisation and inability to follow a schedule.

Furthermore, I wish to express my gratitude to my departmental co-supervisor Professor Peter Jeavons for helping me structure and organise this thesis.

Lastly, I would like to thank my mother and sisters for their love and support throughout my life and scientific journey.

Abstract

The Train Timetabling Problem is an NP-Hard problem, in which one aims to schedule a number of trains withing a railway network. An exact solution can be obtained using Mixed Integer Linear Programming, however train scheduling is often approached using heuristics due to infeasible computation times that Mixed Integer Linear Programming offers. In this thesis we explore to what extent Mixed Integer Linear Programming can be used on large scale Train Timetabling Problems by developing a parser for the test-cases Swiss Federal Railways has made available, and using it together with a commercial Mixed Integer Linear Programming solver. Furthermore, we adapt the Alternating Direction Method of Multipliers to the Train Timetabling Problem by decomposing it into individual problems involving single trains with coupling constraints in an effort to speed up the computation. We then test out our models, producing an analysis of time complexity and suggesting next steps forward.

Contents

List of Figures

Chapter 1

Introduction

1.1 Motivation

To stay competitive, firms are pressured to reduce costs while increasing the quality of their output. Railway companies are not an exception; while there is a focus on improving train themselves, optimal scheduling of trains still remains unsolved for most large railways. Better schedules can lead to improvements in the quality of the services, while reducing the need of deploying more trains into the system, all at zero physical capital cost. The aim is to construct timetables for a number of trains, such that they all satisfy the business needs of the firm, while reducing the firm's operational costs. In particular, Swiss Federal Railways (SBB) [21] has put forward a challenge which calls for solutions to their particular train scheduling problem, where the participants are to generate schedules for a number of railway networks managed by SBB. In its largest scenario, SBB seeks to schedule 287 trains with over 150 stations. SBB describes its constraints, business needs and costs, along with the railway network in which the trains are to run.

Two trains cannot simultaneously occupy the same track, and therefore when modelling a Train Timetabling Problem (TTP), one usually introduces indicator variables to denote which train occupies a resource, which results in a Mixed Integer Linear Program, the solution to which is a globally optimal timetable [2]. Mixed Integer Linear Programming (MILP) as an NP-Hard problem [14], and due to its exponential time complexity, there has been a shift away from using MILP in its

pure form for large scale problems. There has been developed a number of algorithms that impose additional restrictions on the routing, such as train rerouting possibility [9], [10] and speed modelling [7]. There has also been a drive towards using heuristics and genetic algorithms to speed up the solution [15], [20]. For instance, in the paper *A Genetic Algorithm for Railway Scheduling Problems* [22], the authors implement a sampling algorithm, where a population of problems is allowed to merge and mutate, and each member of the population is assigned a fitness score. For each iteration, members are selected into the next population wave according to their fitness score, and this is iterated until a solution is found. However, the solution only achieves local optimality, something that is a recurring issue in all the above approaches, leading to a potential loss of utility for railway firms.

1.2 Objective

This thesis aims to develop a parser that can convert the Train Timetabling Problems (TTPs) to Mixed Integer Linear Programs that can be applied to real-life large scale railway networks. The solver is to be built for the SBB challenge, but should be easily transferable to other problem formulations by modifying the input processing function. Furthermore, **we will attempt to formalise an algorithm that takes advantage of the structural properties of a TTP by decomposing it into subproblems of individual trains to speed up the computation**. There is no doubt that solving a MILP is NP-hard so worst-case solution times are expected to scale exponentially, but nevertheless, for smaller problems it may still be practical to use MILP, and **we will investigate how the time complexity scales with respect to the size of the problem, and determine a bound on the problem size for which it is computationally tractable to use a MILP solver**.

In order to tackle these challenges, I have drawn on techniques of search algorithms and heuristics from the course *Artificial Intelligence*, and content from *Machine Learning*, which covers convex optimisation and linear programming. As this thesis aims to produce a large scale implementation, a prior analysis of feasibility was conducted using techniques learnt in *Requirements*.

1.3 Structure

In the first chapter, the problem of scheduling trains is introduced, alongside current industry standards for solving it. Further, we discuss the benefits of solving the TTP using MILP, and set an objective for large scale implementation of a MILP solver, and checking how it performs on real-life problems.

The second chapter provides the necessary background required to swiftly follow along this thesis. The Train Timetabling Problem is discussed and formalised to a general mathematical problem. Furthermore, chapter 2 introduces content on Optimisation that is relevant to MILP. This includes various optimisation problems, their properties, definitions and potential solution methods.

The third chapter introduces the SBB challenge, formulates a MILP model and implements a solver. The requirements and business needs presented in the SBB challenge are summarised and formalised, which are then used to construct a mathematical model of the problem in MILP form. Finally, the chapter describes and analyses the implementation of a solver that takes inputs from the SBB challenge, and produces an optimal timetable for all the trains.

To take advantage of the TTP's structural properties, chapter 4 introduces Alternating Direction Method of Multipliers (ADMM), which is an algorithm used to serially solve Mathematical Programs. We adapt the algorithm to our specific problem and produce an implementation.

Chapter 5 is focused on conducting experiments on test cases provided by SBB using the parser, a commercial MILP solver and the ADMM adaptation that is developed. We draw conclusion on the performance, assessing to what extent these may be usable in practice.

Lastly, chapter 6 summarises all the work done in this thesis. Here, we discuss the results, the challenges that we faced, and suggest potential improvements and guidelines for future work.

Chapter 2

Background

2.1 Train Scheduling

2.1.1 Introduction

The Train Timetabling Problem aims to generate timetables for entry and exit of trains into different stations along a railway network [1]. Most formulations of the problem will have specified start stations, end stations, and the respective time constraints on entry and exit. However, the other constraints can vary from problem to problem; some may impose constraints on the paths taken, add penalties to certain sections, as well as time constraints on the intermediary stations. Essentially, the idea is to generate optimal paths from a start to an end station for all trains without collisions. The exact definition for optimality will again vary from problem to problem; in most instances, a cost function can be defined from the constraints imposed, which can be used to compare different routes.

2.1.2 Formal description

In this thesis, we will consider railway networks that can be modelled as Directed Acyclic Graphs (DAGs). A DAG is a graph with each edge directed from one node

to another, and there is no possibility of going in a loop (figure 2.1). Mathematically, a DAG can be represented as a topologically sorted graph $G = (E, V)$, where set of nodes V represents stations, and E is the set of edges connecting the stations.

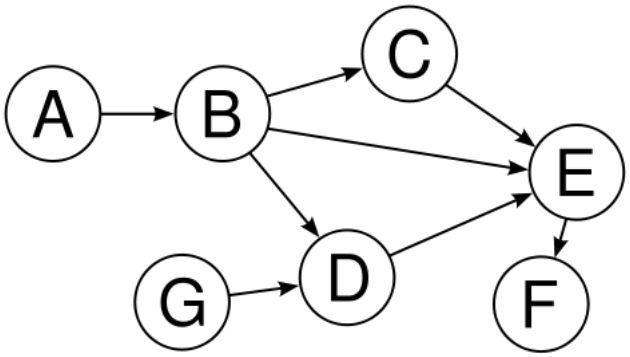


Figure 2.1: Example of a Directed Acyclic Graph

For N trains, we can define the set of start nodes and end nodes for the i-th train as S_i and F_i. Next, we construct a path for each train: $\mathcal{P}_i = v_0^i \to v_1^i \to ... \to v_f^i$, such that $v_0^i \in S_i$ and $v_f^i \in F_i$. We also wish to compute their timetables, which can be represented as a function $T^i : V \mapsto \mathbb{R}^2$, mapping nodes to intervals $[t_{in}, t_{out}]$ of entry and exit times for that node. There may also be constraints on entry and exit of trains into nodes, such as latest entry, earliest exit, etc., which can generally be represented as a function $C^i : V \times \mathbb{R}^2 \mapsto \{True, False\}$, which takes in a node and an interval and returns a boolean, confirming whether time constraints at a node are satisfied or not. These timetable functions will have to be restricted to the following properties:

- For two consecutive nodes u and v, $\max T^i(u) < \min T^i(v)$, meaning that exit time out of a node comes before entry time to the next node.
- $T^i(n) \cap T^j(n) = \emptyset \quad \forall n \in \mathcal{P}_i \cap \mathcal{P}_j \quad \forall i, j \in [1, .., N]$, meaning there is no overlap in entries and exits between different trains sharing common nodes.
- $\wedge_{n \in \mathcal{P}_i} C_i(n, T^i(n)) = True, \quad \forall i \in [1, .., N]$, which makes sure all the problem-specific constraints on entry and exit times of nodes are satisfied along the chosen path. In most train scheduling problems, such constraints will be linear.

2.2 Mathematical Programming

In this section we introduce all concepts and their relevant properties that are referenced throughout the thesis. The section is not intended to give a formal description of the topics, but rather provide the reader with an intuitive understanding of problems in Mathematical Programming and their solution concepts.

2.2.1 Generalised optimisation problem

In general, an optimisation problem can be defined by an objective function $f : \mathbb{R}^n \mapsto \mathbb{R}$, which takes in a set of n variables, and returns a scalar, also referred to as cost. The variables, denoted as x, are then constrained to be in a particular set S. Thus, a general optimisation problem can be written as:

$$\begin{aligned} \min_{x \in dom(x)} \quad & f(x) \\ \text{subject to:} \quad & x \in S \end{aligned} \tag{2.1}$$

2.2.2 Convexity

A convex set is a set where there are no two members x and y, such that a straight line between them crosses the border of the set. Mathematically, a set S is convex iff $\forall x, y \in S$, $\forall t \in [0, 1]$ we have $tx + (1 - t)y \in S$. This is illustrated in figure 2.2.

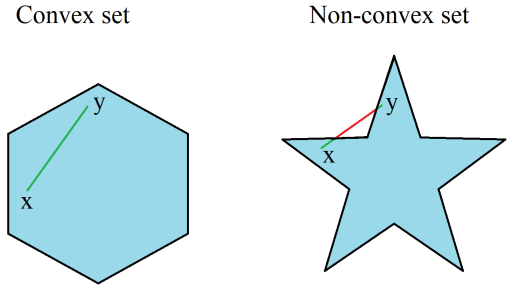


Figure 2.2: Convex vs non-convex sets

The notion of convexity also extends to functions, where a function f is convex if $f(tx + (1 - t)y) \leq tf(x) + (1 - t)f(y)$.

2.2.3 Convex Programming

A program is convex if both the objective function f and the feasible set S are convex. The general form can be written as follows:

$$\begin{aligned} \min_{x \in dom(x)} \quad & f_0(x) \\ \text{subject to:} \quad & f_i(x) \leq 0 \quad \forall i \in [1..m] \\ & h_j(x) = 0 \quad \forall j \in [1..p] \end{aligned} \tag{2.2}$$

Where all functions f_i are convex, and all h_j are affine.

In a convex program, every local optimum is also a global optimum, so an optimiser will never get stuck in a valley. That is a fact that follows from the definition of convexity [13].

2.2.3.1 Interior Point Methods

Interior Point Methods are by far the most popular class of algorithms used to solve constrained convex optimization problems. Assuming that the function is not only convex, but also twice differentiable [19], Interior Point Methods reformulate the problem by adding the constraints as indicator functions into the objective function:

$$V(x) = f_0(x) + \sum_{i=1}^{m} \mathbb{I}[f_i(x)]$$

such that $\mathbb{I}[x] = 0$ if $x \leq 0$ and ∞ otherwise. Further, Interior Point Methods approximate the indicator function $\mathbb{I}$ with logarithmic functions which are differentiable over their domains. The intuitions is for the new objective function to act as an energy potential for a particle, where the logarithmic functions form energy barriers:

$$V(x) = f_0(x) + \gamma \sum_{i=1}^{m} \log[(f_i(x)]$$

The force acting on a particle is the negative gradient of the potential, such that it pushes the particle in space towards its local minimum, at which the net force acting on the particle sums to zero yielding. Applying this principle yields the following differntial equation:

$$-\nabla V(x) = 0$$

which can be solved for x using iterative numerical methods. Usually Interior Point Methods employ the Newton-Raphson method [5], which is a technique to find roots of a function. The logarithmic function is convex, which implies the potential function also is convex, and thus guarantees that the algorithm converges towards optimal x. Figure 2.3 illustrates this method.

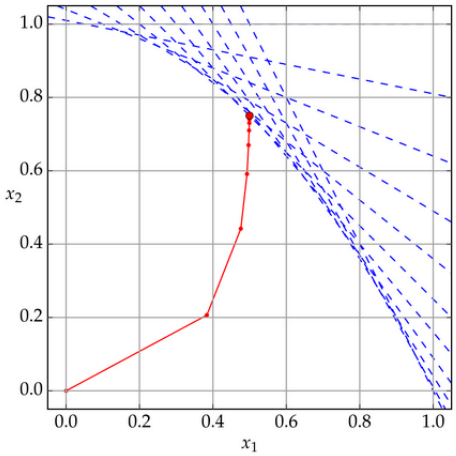


Figure 2.3: Visualisation of the Interior Point Method

2.2.4 Linear Programming

In this thesis, we are focusing on linear programs. A linear program takes the following form:

$$\begin{aligned} \min_{x \in dom(x)} \quad & c^T x \\ \text{subject to:} \quad & a_i^T x \leq b_i(x) \quad \forall i \in [1..m] \\ & h_j^T x = d \quad \forall j \in [1..p] \end{aligned} \tag{2.3}$$

What we see is that the objective function is linear, but also all the inequality constraints are also linear, forming a polyhedral region. Linear Programs are special cases of convex optimisations, and have no closed form solutions. However, Linear Programs can be solved in polynomial time [6].

2.2.4.1 Simplex Algorithm

The linear constraints in a LP form a convex polyhedral region, and since the objective function is linear, it can be shown that the optimum value lays on the surface of the polyhedron. This further implies that the optimum is on one of the extreme points. If our variable of interest is n-dimensional, an extreme point is the point of intersection of n hyperplanes arising from the constraints. In order to avoid searching through all possible vertices, Simplex utilises the fact that a linear function is strictly increasing, and moves between extreme points along the vertices that increase in value. If no such edge is found, the optimal solution is found. If the edge is found, but it connects to no extreme point, then the problem is unbounded.

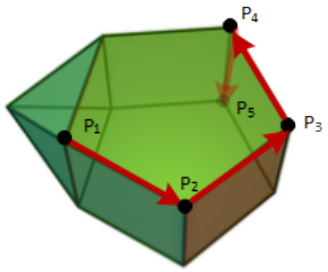


Figure 2.4: Simplex Algorithm

2.2.5 Integer Programming

Integer programming is the class of optimisation problems where some of the variable are required to be integer. Unlike Linear Programming, Integer Programs are NP-Hard [14]. Note that Integer Programs are non-convex, as their feasible region contains certain disjoint points in space, so we cannot use convex programming techniques directly to obtain a solution. Instead, we relax the integrality constraints, and iteratively apply convex optimisation techniques (such as Interior Point Methods, or

Simplex if the problem is linear) combined with the Branch-and-Bound algorithm to find the optimal solution satisfying the integer constraints.

A Mixed Integer Linear Program is a program that contains both continues variables as well as integer variables. It is solved exactly the same way a Linear Integer Program is, because in every iteration Branch-and-Bound relaxes the integrality constraints transforming it into a continuous Linear Program.

2.2.5.1 Branch-and-Bound

The standard way of solving Integer Programs is by relaxing the integrality constraints and using the Branch-And-Bound algorithm until a feasible solution is found [16]. Branch-and-bound is a recursive, divide-and-conquer algorithm, that procedurally relaxes integrality constraints, dividing the problem into sub-problems and adding constraints on the feasible domain.

Consider an integer program where we wish to minimise a function $f(x)$ with respect to the variable x which is to be in a set $\mathcal{S}$. Firstly, we define the current worst objective function and the current best objective as $+\infty$, relax all integer constraints, and solve our first problem, P_0. From there, we start branching into sub-problems P_i. For every P_i, we solve it and get the objective value. If infeasible, we prune the branch. If the objective is greater than the current best objective, we also prune the branch. If all the variables come out to satisfy the integrality constraints[1] and the objective value is less than the current best objective, we update our current best objective and store the solution as the current best. Last case is if the objective to our current problem is less than the best objective, but some variable $x_i = v$ is not an integer, we branch into two subtrees; one having a new constraint $x_i \leq \lfloor v \rfloor$, and the other $x_i \geq \lceil v \rceil$, where $\lfloor . \rfloor$ and $\lceil . \rceil$ are the floor and ceiling operators, respectively. This is done until the algorithm terminates with a feasible integer solution, or all branches are declared infeasible. The Branch-and-Bound algorithm for integer programming is summarised in algorithm 1.

[1]The numbers may not come out as exact integers, so we put a tolerance on how close the number has to be to an integer

Algorithm 1: Branch-and-Bound

Result: Optimal integer solution
$current\_best \leftarrow \inf$;
$current\_best\_solution \leftarrow None$;
$P_0 \leftarrow Problem(f, \mathcal{S})$;
$Q \leftarrow \{P_0\}$;
while $|Q| \neq 0$ **do**
  $P_i \leftarrow Q$;
  $Q \leftarrow Q - P_i$;
  $z, x = solve(P_i)$;
  if $z = infeasible$ **then**
    prune branch ;
  if $z < current\_best$ **then**
    if $x_j \in \mathbb{Z} \quad \forall j$ **then**
      $current\_best \leftarrow z$;
      $current\_best\_solution \leftarrow x$;
    if $\exists x_j \in x$ *such that* $x \notin \mathbb{Z}$ **then**
      $P_{i,1} \leftarrow Problem(f, \mathcal{S} \cup \{x_j \leq \lfloor x_j \rfloor\})$;
      $P_{i,2} \leftarrow Problem(f, \mathcal{S} \cup \{x_j \geq \lceil x_j \rceil\})$;
      $Q \leftarrow Q \cup \{P_{i,1}, P_{i,2}\}$
end
return $current\_best, current\_best\_solution$

In its pure form, the Branch-And-Bound algorithm guarantees to find the optimal solution; however, for large problems, it may become computationally intractable due to the exponential complexity, so heuristics are often used. However, it should be noted that heuristic functions are not always admissible (functions that overestimate the actual cost), thus leading to potential sub-optimal solutions [8].

2.2.6 Lagrangian Duality

Consider a generalised optimisation problem P that can be expressed in the following primal form[2]:

$$\min_{x \in dom(x)} \quad f_0(x)$$

$$\text{subject to:} \quad f_i(x) \leq 0 \quad \forall i \in [1..m]$$

For this problem, we can define a Lagrangian function:

[2]Note that an equality can always be expressed as pair of inequalities:
$f(x) = 0 \implies f(x) \leq 0 \wedge f(x) \geq 0$.

$$L(x, \lambda) = f_0(x) + \sum_{i=1}^{m} \lambda_i f_i(x)$$

For this function, calculating the maximum with respect to lambda recovers the original problem P:

$$\max_{\lambda \geq 0} L(x, \lambda) \equiv P$$

such that the optimum value of P, p^* can be found by minimising the Lagrangian over x:

$$p^* = \min_x \max_{\lambda \geq 0} L(x, \lambda)$$

However, maximising with respect to λ first is usually infeasible, so instead, we swap *min* and *max* operations:

$$q^* = \max_{\lambda \geq 0} \min_x L(x, \lambda)$$

getting the following problem D:

$$\begin{aligned} \max_{\lambda} \quad & \min_x L(x, \lambda) \\ \text{subject to:} \quad & \lambda_i \geq 0 \quad \forall i \in [1..m] \end{aligned}$$

Problem D is what is referred to as the dual problem.

The Minimax Inequality theorem states that for some function $g(x, y)$, $\max_y \min_x g(x, y) \leq \min_x \max_y g(x, y)$. Thus, we can conclude that $q^* \leq p^*$, meaning the solution of the dual serves as a lower bound on the optimum of the primal. $p^* - q^*$ is called the duality gap.

For a Linear Program, it is simple to set up the dual problem as another Linear Program. Consider the following program:

$$\begin{aligned} \min_{x} \quad & c^T x \\ \text{subject to:} \quad & Ax \leq b \end{aligned}$$

The dual function is then given by

$$\min_{x} \max_{\lambda \geq 0} c^T x + \lambda^T (Ax - b)$$

Rearranging the argument to $\max_{\lambda \geq 0} \min_x \{(c^T + \lambda^T A)x\} - \lambda^T b$, we can see that the argument to the min function explodes to negative infinity unless $c^T + \lambda^T A = 0$, in which case our problem is bounded at $-\lambda^T b$. Thus, the dual program can be stated as following:

$$\begin{aligned} \max_{\lambda} \quad & -b^T \lambda \\ \text{subject to:} \quad & c^T + \lambda^T A = 0 \\ & \lambda \geq 0 \end{aligned}$$

Lagrangian duality gives us another way of solving potentially non-convex problems. Under certain conditions, the duality gap is zero, which means solving a dual problem has the same optimum as the primal problem. If not, the dual provides us with a lower bound on the solution, which can be used to optimise search over the primal problem.

2.2.7 Lagrangian Relaxation

Following the section on Lagrangian Duals, we introduce a numerical method to solve optimisation problems, namely Lagrangian Relaxation. Suppose we wish to minimise a convex function $f(x)$ subject to linear inequality constraints:

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{subject to:} \quad & Ax \leq b \\ & Cx \leq d \end{aligned}$$

Suppose the constraint $Ax \leq b$ is a complicating constraint, making the whole problem more difficult to solve. It may be a constraint that couples various entries in x, drastically increasing the time an optimiser spends on the problem. In this scenario, we chose to 'dualise' the difficult constraint by adding it to the objective function, and solving the dual problem (section 2.2.6):

$$\begin{aligned} \max_{\lambda} \quad & \{\min_{x}[J(x,\lambda) = f(x) + \lambda^T(Ax - b)]\} \\ \text{subject to:} \quad & \lambda \geq 0 \\ & Cx \leq d \end{aligned}$$

where λ is a vector of Lagrangian multipliers. In simple terms, we have relaxed the complicating constraints by penalising them in the objective function. However, we are still required to maximise the Lagrange multipliers, keeping them non-negative, as discussed in section 2.2.6.

To solve the problem, we resort to a method called Dual Ascent. Dual ascent is based on the principle of alternating between minimising the relaxed problem with respect to x, and climbing in the direction of steepest ascent (Gradient Ascent) with respect to λ, where the gradient is given by $\nabla_\lambda J(x,\lambda) = Ax - b$ Dual Ascent for a function $J(x,\lambda)$ works as following [4]:

Algorithm 2: Dual Ascent

Result: Solves a Lagrangian Relaxation problem
Formulate a stopping criterion $P(J, x, \lambda) \in \{True, False\}$;
Initialise the arguments x_0 and λ_0;
while $\neg P$ **do**
  Update x by solving the minimisation problem: $x_{n+1} \leftarrow \text{argmin } J(x, \lambda_n)$;
  Update λ using Gradient Ascent: $\lambda_{n+1} \leftarrow \max\{0, \lambda_n + \gamma \nabla_\lambda J(x_n, \lambda_n)\}$;
  Update the stopping criterion $P \leftarrow P(x_{n+1}, \lambda_{n+1}, J(x_{n+1}, \lambda_{n+1}))$;
end
return $x_{n+1}, \lambda_{n+1}, J(x_{n+1}, \lambda_{n+1})$

The presence of the max function in update of λ ensures that the Lagrangian multipliers remain non-negative. As for the stopping criterion, the standard practice is to assign a tolerance on the p-norm of the residual $Ax - b$. Since both the primal and relaxed objective functions are convex in both x and λ, the algorithm guarantees to converge to the global optimum x, given appropriate choice of the step size [3].

2.2.7.1 Augmented Lagrangian Form

Augmented Lagrangian Form is an extension of Lagrangian Relaxation, where further penalisation of the 2-norm of the residual is added to the objective function:

$$J(x, \lambda) = f(x) + \lambda^T(Ax - b) + \frac{\rho}{2}\|Ax - b\|_2^2$$

where ρ is a hyperparameter controlling the magnitude of the penalty. The resulting problem yields the same optima as the primal problem, seeing that optimal x zeros out the penalty terms. The advantage of using an Augmented Lagrangian Form is that it reduces the number of assumptions made on the primal objective function, improving robustness and convergence. The Augmented Lagrangian Form has the same subgradient as the Relaxed Lagrangian problem, and can be solved using Dual Ascent.

Chapter 3

Train Timetabling Problem using MILP

3.1 SBB challenge

In this thesis, the aim of solving the Train Timetable Problem posed by Swiss Federal Railways (SBB) [21]. SBB poses a number of business rules to do with consistency and planning. For this thesis, certain rules were simplified, without altering the core of the problem.

SBB provides the users with .json files that describe the problems, including network graphs modelled as Directed Acyclic Graphs for all trains to be scheduled, and the relevant parameters associated with the problem. The idea is to schedule all trains in a problem, find which paths they are to take, attaching a timetable of entries and exits along the chosen paths, while following all the business rules. It should be noted that this challenge focuses on route sections and not stations (edges of the graph instead of nodes). Below is a summary of the problem we have formulated for this thesis:

1. Every train in the problem is scheduled to a single route.
2. A route has to begin at a starting station (node), and finish in a ending station, as specified in the .json file.

3. A valid route will be a traversal of the DAG for that train.
4. Consistency between entry and exit times; time out of a route section should equal to time in to the next route section along a path.
5. Earliest exits and entries for various route section specified in the problem must be satisfied.
6. Latest exits and entries allow for slack, which is penalised according to the objective function specified by the SBB challenge.
7. Use of certain route sections are penalised by the objective function as specified in the problem.
8. If a minimum running time and/or minimum stopping time for a section are specified, the train must spend at least that duration on the section.
9. Two trains cannot be in the same route at the same time.
10. If two trains are to pass the same route section, the difference between exit of first train and entry of second train must be more than the release time specified in the problem.

3.2 Mathematical model

In this section, we will formalise the setup. This section closely follows the work of Garrisi and Cervelló-Pastor [11], with certain modifications and added formalism. Note that the $|\cdot|$ operator in this thesis is used to indicate length of vectors and sets, not to be confused with norms.

3.2.1 Definitions

- $si \in \mathbb{Z}^+$ - Service intention index, a train to be scheduled.
- $r \in \mathbb{Z}^+$ - Route index, a complete path from a start to an end station.
- $rs \in \mathbb{Z}^+$ - Route Section index, an edge in the graph connecting two stations.

- SI - Set containing unique identifiers for every service intention in the problem. An element is the index of a service intention.
- $t^{in}_{si,r,rs} \in \mathbb{R}^+$ - Time into a route section for a particular train, on a particular route.
- $t^{out}_{si,r,rs} \in \mathbb{R}^+$ - Time out of a route section for a particular train, on a particular route.
- $\alpha_{si,rs} \in \{0, 1\}$ - Boolean variable indicating whether a service intention uses a route section.
- $\delta_{si,r} \in \{0, 1\}$ - Boolean variable indicating whether a service intention uses a route.
- $\beta_{si_1,si_2,rs} \in \{0, 1\}$ - Boolean variable indicating whether two service intentions use the same route section.
- $mrt_{si,rs} \in \mathbb{R}^+$ - Minimum running time a train has to spend on a route section.
- $mst_{si,rs} \in \mathbb{R}^+$ - Minimum stopping time a train has to spend on a route section.
- $EarIn_{si,rs} \in \mathbb{R}^+$ - Earliest time a train is allowed to enter a route section.
- $EarOut_{si,rs} \in \mathbb{R}^+$ - Earliest time a train is allowed to exit a route section.
- $LatIn_{si,rs} \in \mathbb{R}^+$ - Latest time a train is allowed to enter a route section.
- $LatOut_{si,rs} \in \mathbb{R}^+$ - Latest time a train is allowed to exit a route section.
- $win_{rs} \in \mathbb{R}^+$ - Weight associated with entry into a route section. Used for penalty in the objective function.
- $wout_{rs} \in \mathbb{R}^+$ - Weight associated with exit from a route section. Used for penalty in the objective function.
- $p_{si,rs} \in \mathbb{R}^+$ - Penalty of a train using a particular route section.
- $G_i = (V_i, E_i, D_i)$ - Rail network graph for the i-th service intention. V_i is set of vertices, E_i is the set of edges and D_i is a set of data sets associated with every edge.
- D_i - Set of data parameters of service intention i associated with a route section. For an element $d \in D_i$, $d \subseteq \{mrt, mst, LatIn, LatOut, EarIn, EarOut\}$. This is used for book keeping, as certain edges will only have a subset of constraint variables associated with them.

- $S_i \in V_i$ - Set of all starting nodes (stations) of the i-th service intention.
- $F_i \in V_i$ - Set of all final nodes (stations) of the i-th service intention.
- $\mathcal{P}_i$ - Set of admissible paths for the i-th service intention. Each element $P \in \mathcal{P}_i$ must be a subraph traversal $P \subseteq G_i$ with root node contained in S_i and leaf node contained in F_i.

3.2.2 Objective function

In the SBB business rules specification, earliest arrival and departure constraints must be strictly satisfied, however latest arrival and departures are given slack, and thus are placed in the objective function instead. If a late arrival occurs, then it is penalised proportionally to the delay and the weight $win_{si,rs}$. Similarly, same applies to delayed entries. Furthermore, SBB requires a fixed penalty for use of certain route sections. The ratio of delay to route section penalties is 1:60 (expressed in seconds). Thus, the objective function can be written as:

$$\begin{aligned} J(\theta) = \frac{1}{60} \sum_{(si,r,rs)\in SO_1} & win_{si,rs} \max(0, t^{in}_{si,r,rs} - LatIn_{si,rs}) \\ + \frac{1}{60} \sum_{(si,r,rs)\in SO_1} & wout_{si,rs} \max(0, t^{out}_{si,r,rs} - LatOut_{si,rs}) + \sum_{SO_3} p_{si,rs}\alpha_{si,rs} \end{aligned} \tag{3.1}$$

where θ is the grand state vector encompassing $t^{oin}_{si,r,rs}$, $t^{out}_{si,r,rs}$ and $\alpha_{si,rs}$ for all si, r and rs in the problem. The sets SO_1 and SO_2 contain all route sections, routes and service intentions where a requirement on latest entry and latest exists are present. Mathematically, one could simply add such variables to remaining edges with the value infinity, however, that would not be computationally efficient, especially when feeding the problem into a MILP solver. SO_3 contains all route sections and their respective route sections where penalty is non-zero.

It should be noted that the objective function, although convex, does not necessarily have a unique optimum. Consider a single route section problem where the latest entry is one minute, and latest exit is two minutes. Without any additional constraints, we can see that any solution (t^{in}, t^{out}) is optimal as long as $t^{in} \leq 1$ and

$t^{out} \leq 2$, thus proving by example that an optimal timetable does not need to be unique (illustration of this example is shown on figure 3.1).

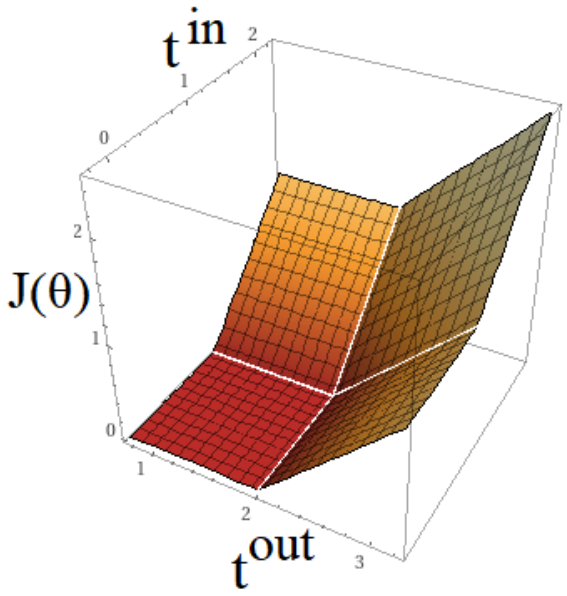


Figure 3.1: Plot of a single route section objective function. $LatIn = 1$, $LatOut = 2$, $p = 0$. The red area is the set of all optimal solutions.

The modelling is done to transform the Train Timetabling Problem into a Mixed Integer Linear Program, however the reader may note that the objective function is non-linear due to the presence of the max operator. However, we can easily rewrite it into linear form. Consider the following programme:

$$\begin{aligned} \min_{x} \quad & \sum_{i \in \mathcal{I}} \max(0, x_i - u_i) \\ \text{subject to:} \quad & x_i \in S_i \quad \forall i \in \mathcal{I} \end{aligned}$$

Introducing slack variables y_i, we can rewrite this program into linear form:

$$\begin{aligned} \min_{y} \quad & \sum_{i \in \mathcal{I}} y_i \\ \text{subject to:} \quad & x_i \in S_i \\ & y_i \geq x_i - u_i \quad \forall i \in \mathcal{I} \\ & y_i \geq 0 \end{aligned}$$

Since the objective is to minimise the expression, slack variables y_i will always either take on the value of $x_i - u_i$ or 0.

3.2.3 Constraints

Having formulated the objective function, we already have our first constraints arising from intoduction of slack variables s:

$$\begin{aligned} s_{si,r,rs} &\geq win_{si,rs}(t^{in}_{si,r,rs} - LatIn_{si,rs}) \\ s_{si,r,rs} &\geq 0 \quad \forall(si,r,rs) \in SO_1 \end{aligned} \tag{3.2}$$

$$\begin{aligned} s_{si,r,rs} &\geq wout_{si,rs}(t^{out}_{si,r,rs} - LatOut_{si,rs}) \\ s_{si,r,rs} &\geq 0 \quad \forall(si,r,rs) \in SO_2 \end{aligned} \tag{3.3}$$

The following constraints will be of two forms; physical constraints related to the mechanics of train movement, as well as constraints specified by the business rules.

Firstly, entry time into a route section must come before the exit time into that route section:

$$t^{in}_{si,r,rs} \leq t^{out}_{si,r,rs} \quad \forall(si,r,rs) \in S_g \tag{3.4}$$

where S_g is the set containing all service intentions, all their paths and corresponding route sections.

So far edges and route sections have been used interchangeably, but this is not entirely the case. It is also important to remember that there are stations (nodes), and these also need to be associated to a route section. We can define a convention that a route section will also contain the entry node to its edge. By this convention, final stations will have no route section association, so we assign them to their edge (figure 3.2). All the appropriate constraints and parameters are formulated in terms of route sections rather than stations, with the only exception being start and end stations, thus making no difference in the computation.

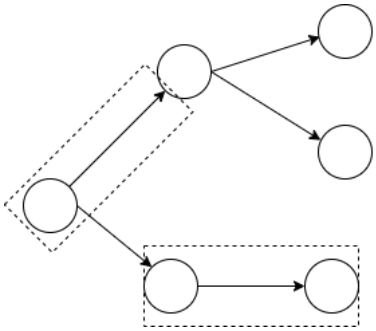


Figure 3.2: Route sections (dotted line) of a railway network

We will also require that exit time of a route section matches the entry time into a consecutive route section, thus:

$$t^{out}_{si,r,rs} = t^{in}_{si,r,rs+1} \quad \forall (si, r, rs) \in S_g \tag{3.5}$$

The indices here are loosely used, but we assume that the set is ordered such that $rs{+}1$ in a path r will always be the consecutive edge. For the last route section, $rs{+}1$ will simply be unidentified, but that is handled during the actual implementation.

In this problem, we have time variables for every possible path that a service intention can take. However, in the end only one path is to be assigned. We know that the path tracking variables $\delta_{si,r}$ can be either 0 or 1, so such a check can be implemented by simply making sure all $\delta_{si,r}$ for a service intention sum to 1:

$$\sum_{r \in \mathcal{P}_{si}} \delta_{si,r} = 1 \quad \forall si \in SI \tag{3.6}$$

Further, we wish to establish a connection between the globally occupied route sections ($\alpha_{si,r}$) and chosen routes ($\delta_{si,r}$). We cannot require that $\alpha_{si,r} = \delta_{si,r}$, as two possible paths may have a common route section but only one path can be assigned, resulting in a contradiction (figure 3.3). Instead, we can simply require selection variable α for the route section to be greater than or equal to all routes it is present in:

$$\alpha_{si,r} \geq \delta_{si,r} \quad \forall (si, r, rs) \in S_g \tag{3.7}$$

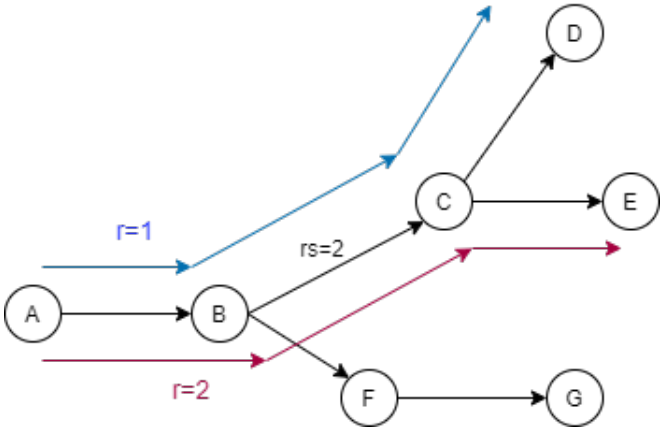


Figure 3.3: Two routes r=1 (blue), r=2 (pink) and a common route section rs=2

To obey the business rules, we must ensure that earliest entry and exits are satisfied. We wish to create a constraint such that if $\delta_{si,r} = 1$, i.e. route r is selected, then $t^{in}_{si,r,rs} \geq EarIn_{si,rs}$ and $t^{out}_{si,r,rs} \geq EarOut_{si,rs}$. Otherwise, we do not care. This procedure can be linearised using a large constant M:

$$t^{in}_{si,r,rs} \geq EarIn_{si,rs} - M(1 - \delta_{si,r}) \quad \forall (si, r, rs) \in S_{ei} \tag{3.8}$$

$$t^{out}_{si,r,rs} \geq EarOut_{si,rs} - M(1 - \delta_{si,r}) \quad \forall (si, r, rs) \in S_{eo} \tag{3.9}$$

where S_{ei} and S_{eo} are sets containing all route sections, routes and service intentions where a constraints on earliest entries and exits are imposed. The big-M is a trick for linearising logical gates, where in our case M should be large enough to deactivate the constraint if the route to which the time belongs is not selected, so $M \geq \max_{si,rs}(EarIn_{si,rs})$. Again, one could simply assign a value of infinity to unconstrained route sections, but that would add numerous unnecessary constraints, making the problem ill-posed and slowing down the solver. Sufficiently large M ensures that if the route is **not** chosen, the constraint will be true regardless of the value of t, making the constraint irrelevant.

SSB also specifies that certain route sections will have a minimum stopping time, and a minimum running time. Minimum stopping time is determined by the business needs, so for instance allowing enough passengers to board and disembark the train, while minimum running time is related to physical speeds the trains can run at. This essentially means that if a route r containing a route section rs is chosen

($\delta_{si,r} = 1$), then the service intention must spend at least a time of $mst_{si,rs} + mrt_{si,rs}$ on that section. The time a service intention spends on a route section is simply $t^{out}_{si,r,rs} - t^{in}_{si,r,rs}$. Thus, using the same reasoning as above, we can formulate the linear constraint:

$$t^{out}_{si,r,rs} - t^{in}_{si,r,rs} \geq mrt_{si,rs} + mst_{si,rs} - M(1 - \delta_{si,r}) \quad \forall (si, r, rs) \in S_{mt} \tag{3.10}$$

Where the set S_{mt} is a set of all route sections, their paths and service intentions that have either a minimum stopping time or a minimum running time criterion.

Lastly, we need to add the constraints that couple the trains together. These relate to the fact that multiple trains cannot simultaneously occupy the same route section. For two service intentions, si_1 and si_2, we find all intersecting edges: $I_{si_1,si_2} = E_{si_1} \cap E_{si_2}$, from $G_{si_1} = (V_{si_1}, E_{si_1})$ and $G_{si_2} = (V_{si_2}, E_{si_2})$. The set of edges I_{si_1,si_2} will contain all route sections that are in common for two railway networks[1]. Thus, the binary variable $\beta_{si_1,si_2,rs}$ is uniquely assigned to every edge in I_{si_1,si_2}. We perform this operation on all **combinations** of the trains; $\{(si_1, si_2), (si_1, si_3), ..., (si_1, si_n), (si_2, si_3), ..., (si_2, si_n), ..., (si_{n-1}, si_n)\}$, and aggregate the indices of service intention pairs, route pairs and route section into a set S_{is}. Fixing the tuples in train combinations to be ordered, we introduce the following convention: if the first service intention enters the section before the second one, $\beta_{si_1,si_2,rs} = 1$. Vice versa, $\beta_{si_1,si_2,rs} = 0$. Above reasoning can be phrased in a linear manner:

$$t^{in}_{si_1,r_1,rs} \leq t^{in}_{si_2,r_2,rs} + M(1 - \beta_{si_1,si_2,rs})$$

if $\beta_{si_1,si_2,rs} = 1$, the constraint $t^{in}_{si_1,r_1,rs} \leq t^{in}_{si_2,r_2,rs}$ has to be satisfied. Otherwise, for a sufficiently large M, LHS will always be less than RHS, and the constraint becomes inactive. To handle the opposite side, where si_2 enters the section first, we write the following linear model:

$$t^{in}_{si_2,r_2,rs} \leq t^{in}_{si_1,r_1,rs} + M\beta_{si_1,si_2,rs}$$

which works in the same way. Lastly, we ensure that the both routes in question

[1]Formally, we should be taking edges that intersect in the unions of all admissible paths, i.e. $(\bigcup_{p_1 \in \mathcal{P}_{si_1}} p_1) \cap (\bigcup_{p_2 \in \mathcal{P}_{si_2}} p_2)$. However, in the SBB challenge, no redundancies on railway networks and possible paths are given, thus if an edge exists, there will be an admissible path including that edge.

are actually in use, otherwise constraints should be inactive:

$$t^{in}_{si_1,r_1,rs} - t^{in}_{si_2,r_2,rs} \leq M(1 - \beta_{si_1,si_2,rs}) + M(2 - \delta_{si_1,r_2} - \delta_{si_2,r_2}) \tag{3.11}$$

$$t^{in}_{si_2,r_2,rs} - t^{in}_{si_1,r_1,rs} \leq M\beta_{si_1,si_2,rs} + M(2 - \delta_{si_1,r_2} - \delta_{si_2,r_2}) \tag{3.12}$$

$$\forall(si_1, r_1, si_2, r_2, rs) \in S_{is}$$

If left this way, service intentions will be able to immediately enter the route section one after the other. We wish to avoid that; if si_1 uses a common route section, we require that the next train si_2 waits R seconds[2] after si_1's exit before entering: $t^{in}_{si_2,r_2,rs} \geq t^{out}_{si_1,r_1,rs} + R$. Linearising with respect to β and δ yields the following system:

$$t^{out}_{si_1,r_1,rs} - t^{in}_{si_2,r_2,rs} + R \leq M(1 - \beta_{si_1,si_2,rs}) + M(2 - \delta_{si_1,r_2} - \delta_{si_2,r_2}) \tag{3.13}$$

$$t^{out}_{si_2,r_2,rs} - t^{in}_{si_1,r_1,rs} + R \leq M\beta_{si_1,si_2,rs} + M(2 - \delta_{si_1,r_2} - \delta_{si_2,r_2}) \tag{3.14}$$

$$\forall(si_1, r_1, si_2, r_2, rs) \in S_{is}$$

The MILP can be summarised as following:

$$\begin{aligned} J(\theta) = \frac{1}{60} &\sum_{(si,r,rs)\in SO_1} win_{si,rs} s_{si,r,rs} \\ + \frac{1}{60} &\sum_{(si,r,rs)\in SO_1} wout_{si,rs} s_{si,r,rs} + \sum_{SO_3} p_{si,rs}\alpha_{si,rs} \end{aligned} \tag{3.15}$$

Subject to:

[2]For the purpose of this challenge, constant R is taken to be fixed throughout the railway network.

$$s_{si,r,rs} \geq win_{si,rs}(t^{in}_{si,r,rs} - LatIn_{si,rs})$$

$$s_{si,r,rs} \geq 0 \quad \forall (si, r, rs) \in SO_1$$

$$s_{si,r,rs} \geq wout_{si,rs}(t^{out}_{si,r,rs} - LatOut_{si,rs})$$

$$s_{si,r,rs} \geq 0 \quad \forall (si, r, rs) \in SO_2$$

$$t^{in}_{si,r,rs} \leq t^{out}_{si,r,rs} \quad \forall (si, r, rs) \in S_g$$

$$t^{out}_{si,r,rs} = t^{in}_{si,r,rs+1} \quad \forall (si, r, rs) \in S_g$$

$$\sum_{r \in \mathcal{P}_{si}} \delta_{si,r} = 1 \quad \forall si \in SI$$

$$\alpha_{si,r} \geq \delta_{si,r} \quad \forall (si, r, rs) \in S_g$$

$$t^{in}_{si,r,rs} \geq EarIn_{si,rs} - M(1 - \delta_{si,r}) \quad \forall (si, r, rs) \in S_{ei}$$

$$t^{out}_{si,r,rs} \geq EarOut_{si,rs} - M(1 - \delta_{si,r}) \quad \forall (si, r, rs) \in S_{eo}$$

$$t^{out}_{si,r,rs} - t^{in}_{si,r,rs} \geq mrt_{si,rs} + mst_{si,rs} - M(1 - \delta_{si,r}) \quad \forall (si, r, rs) \in S_{mt}$$

$$t^{in}_{si_1,r_1,rs} - t^{in}_{si_2,r_2,rs} \leq M(1 - \beta_{si_1,si_2,rs}) + M(2 - \delta_{si_1,r_2} - \delta_{si_2,r_2})$$

$$t^{in}_{si_2,r_2,rs} - t^{in}_{si_1,r_1,rs} \leq M\beta_{si_1,si_2,rs} + M(2 - \delta_{si_1,r_2} - \delta_{si_2,r_2})$$

$$\forall (si_1, r_1, si_2, r_2, rs) \in S_{is}$$

$$t^{in}_{si_1,r_1,rs} - t^{in}_{si_2,r_2,rs} \leq M(1 - \beta_{si_1,si_2,rs}) + M(2 - \delta_{si_1,r_2} - \delta_{si_2,r_2})$$

$$t^{in}_{si_2,r_2,rs} - t^{in}_{si_1,r_1,rs} \leq M\beta_{si_1,si_2,rs} + M(2 - \delta_{si_1,r_2} - \delta_{si_2,r_2})$$

$$\forall (si_1, r_1, si_2, r_2, rs) \in S_{is}$$

3.2.4 Complexity analysis

When solving an MILP, we use the Branch-and-Bound algorithm to consecutively partition the search space, solving the corresponding relaxed Linear Programs.[3] In our case, all our integer variables happen to be binary. Suppose we are at an iteration i, and we decide to branch on the variable x_j. The LP we just solved in iteration $i-1$ is branched into two subprograms, one where $x_j = 0$ and the other $x_j = 1$. We now solve both subprograms in the same way. The search tree will always have a branching factor of 2, as explained in algorithm 1. If no branch is ever pruned, the leafs of the tree will be all possible combinations of the boolean variables, so in fact the worst-case time complexity of our search is $\mathcal{O}(2^v)$, where v is the number of boolean variables. It is impossible to state a closed form number of variables in this problem, due to the unknown structure of the DAG. Assuming a worst-case scenario, all service intentions will all have the same railway graph $G = (E, V, D)$, and the graph starts off with all its root nodes being in S, and all leafs in F. These are then connected through d layers of b stations each (figure 3.4).

[3] To relax in this context means dropping integer constraints.

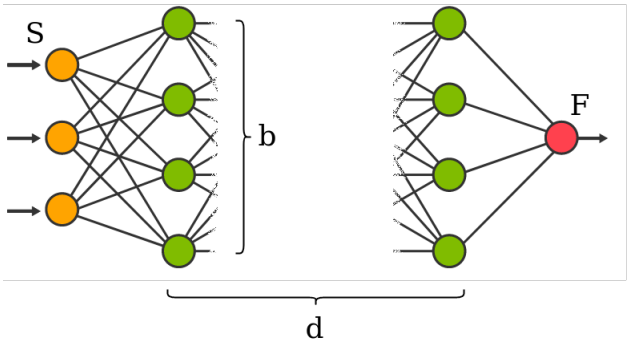


Figure 3.4: Worst case scenario of a railway network

For every service intention si, there will be $|E|$ variables of α, $|S||F|b^d$ paths, corresponding to the number of δ-variables, and a total of $\binom{N}{2}|E|$ coupling variables. Summing over all service intentions, $v = N(|E|+|S||F|b^d)+\frac{1}{2}N(N-1)$. What we see is that the number of variables scale as a square of number of service intentions, but exponentially with respect to the 'depth' of the railway network. Substituting our expression for the number of leafs generated by Branch-and-Bound, we quickly see that the algorithm scales at least exponentially with respect to all our parameters.

3.3 Parser Implementation

3.3.1 Rationale

For the implementation of the mathematical model, Python was chosen as the primary engine due to its versatility and access to vast array of modules. This was done at the expense of computation times, however it should be noted that the numerical MILP solver chosen, MOSEK [18] runs under the hood in compiled C++ code. To formulate the problem into a form that MOSEK can process, we will create vectors that represent the objective function c, and constraint of the form $A\theta \leq b$.

3.3.2 List of dependencies

- Python 3.6
  - Json
  - Networkx
  - Numpy
  - Scipy
  - Matplotlib
  - Seaborn
- MOSEK

3.3.3 Implementation

3.3.3.1 Higher level overview

The flow of the programme is simple: we start by parsing the json, extracting and storing all key information. This is then fed into a function that generates networkx graphs for every service intention, attaching all relevant parameters to edges of the graphs. These graphs are then used to calculate admissible routes, which start in nodes labled 'start' and end in nodes labled 'end'. Having the graphs and the paths, we create book-keeping data structures that help manage indices and sets required to formulate the constraints. Now the variables relating to time (t^{in}, t^{out}), paths (δ), route sections (α) and coupling (β) can be initialised. Having these data structures, we can use them to generate all appropriate constraints to be handled by MOSEK, and define a linear objective function $c^T\theta$. Provided a feasible solution exists, MOSEK returns the values of the variables, which are then parsed into timetables for the chosen paths. A diagram of this process is shown on figure 3.5.

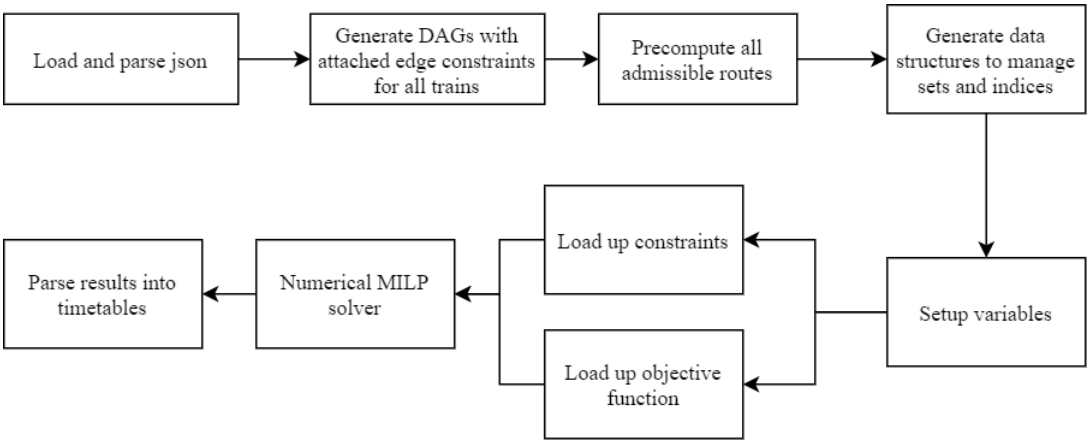


Figure 3.5: Implementation diagram

3.3.3.2 Parsing json

The input comes as a .json file, composed of three main parts: section requirements, route sections and resources. Each route section may specify a penalty, a label and/or a minimum running time specific to itself only. The label is used if a route section is to have specific requirements; this is where section requirement is used. Section requirements is a list of labeled sets of requirements, containing a combination of the following: minimum stopping time, earliest entry and exit, latest entry and exit, entry and exit delay weight $(win, wout)$ and whether the route section is a starting or ending section. These are then stored as dictionaries, so lookup can be done in constant time. In this thesis, resources were reformulated into route sections, so that part is irrelevant.

3.3.3.3 Generation of Direct Acyclic Graphs

To store the graphs, we use a Python module called networkx. For every service intention, we generate a graph in the form of $G_{si} = (V_{si}, E_{si}, D_{si})$, where V_{si} is a set of nodes, E_{si} is a set of edges, and D_{si} is a set of data dictionaries (hash tables) associated with every edge. A script was provided by SBB which can convert their list format to a graph, but essentially works by looping over all route sections gluing them together using the node labels. This script was modified to also append a data dictionary to every edge, thus distributing all the necessary constraints over the appropriate edges. We have now a complete model of the railway network for

the various service intentions, and have associated all constraints are attached to the various route sections.

3.3.3.4 Computing all admissible routes

In our mathematical formulation, we index variables by the route r they take. This means we need to know *all* the admissible routes we can take. Some of our edges have their data field labeled as 'start' or 'end'. For every service intention, we can construct sets of nodes S_{si} and F_{si} for starting and ending nodes, respectively. We can then search for paths. It should be noted that there can be multiple paths between two nodes, so essentially we will have to traverse the entire graph. Since we are dealing with a DAG, we can simply do a breadth-first-traversal (BFT) for every starting node, storing a path every time we hit an ending node. The BFT will only terminate when there are no more nodes to expand, resulting in all paths from a node $n \in S_{si}$ to all nodes in F_{si} (algorithm 3). BFT is an exhaustive search algorithm, but the problem is exponential in the number of nodes; consider the worst case scenario, where a DAG forms a tree, branching by a factor of b, having a depth d. Also, suppose that the root node is in S_i and all the leafs are in F_i. Thus, we are forced to traverse the graph, with the time complexity being $\mathcal{O}(b^{d+1})$. However, our whole problem formulation is NP-hard, which scales exponentially with the number of variables, so the route calculations are unlikely to be the bottleneck of this implementation.

Algorithm 3: Path calculation

```
Result: All possible paths between starting and ending nodes
Initialise 𝒫;
forall si ∈ SI do
    Initialise 𝒫_si;
    Compute F_si, S_si;
    forall n ∈ S_si do
        𝒯 = BFT(n, G_si, F_si);
        𝒫_si ← 𝒫_si ∪ 𝒯;
    end
    𝒫 ← 𝒫 ∪ 𝒫_si;
end
return 𝒫
```

3.3.3.5 Book-keeping for set generation

The variables for computation will be 1-dimensional vectors s (slack), t^{in}, t^{out}, δ, α and β. Thus, we need a way of correctly indexing these variables. In section 3.2, various sets were introduced and loosely defined to manage service intentions, routes and route sections. During implementation, it is instead simpler to create data structures that will help us loop through relevant parts of the code. First we define *si_list*. This list contains all route sections for all routes and service intentions. Suppose the railway network has the DAG shown in figure 3.3 for two identical service intentions, 1 and 2 ($G_1 = G_2$). Suppose also starting nodes for these trains are A and ending nodes are D, E, G. *si_list* for this scenario is shown on figure 3.6:

```
[[[(1, 1, ('A', 'B')), (1, 1, ('B', 'C')), (1, 1, ('C', 'D'))],
  [(1, 2, ('A', 'B')), (1, 2, ('B', 'C')), (1, 2, ('C', 'E'))],
  [(1, 3, ('A', 'B')), (1, 3, ('B', 'F')), (1, 3, ('F', 'G'))]],
 [[(2, 1, ('A', 'B')), (2, 1, ('B', 'C')), (2, 1, ('C', 'D'))],
  [(2, 2, ('A', 'B')), (2, 2, ('B', 'C')), (2, 2, ('C', 'E'))],
  [(2, 3, ('A', 'B')), (2, 3, ('B', 'F')), (2, 3, ('F', 'G'))]]]
```

Figure 3.6: *si_list* example. First element of the tuple is the service intention, second is path number and third is the edge.

Further we set up various maps:

- *t_index_by_edge* maps edges to a set of indices on t
- *delta_index_by_edge* maps edges to a set of indices on δ
- *edges_by_path* maps edge and path to an index of t
- α*_index* maps service intention and edge to an index of α
- TL maps service intention and variable type (s, t^{in}, etc.) to the length of that variable for that service intention

These will help us set up the constraints later by simply looping over *si_list*.

We also wish to define a helping list to manage coupling variables β. We wish to make a list of intersecting edges *beta_index*, as mentioned when setting up constraints (3.11) to (3.14). The list will contain tuples (si_1, si_2, e) for all combinations of service intentions and their intersecting edges. This can be done by algorithm 4

Algorithm 4: Creation of *beta_index*

Result: Setting up an indexing system for coupling variables
Initialise *beta_index*;
forall $(si_1, si_2) \in combinations(SI, 2)$ **do**
  $\mathcal{I} = E_{si_1} \cap E_{si_2}$;
  forall $e \in \mathcal{I}$ **do**
    $beta\_index \leftarrow beta\_index \cap (e, si_1, si_2)$;
  end
end
return *beta_index*

3.3.3.6 Setting up variables

We define the a unified vector containing all variables s, t^{in}, t^{out}, δ, α and β using the MOSEK.Variable() object. To do so, we need to pre-calculate the length of every variable. Also, while t^{in} and t^{out} are continuous, δ, α and β must be integers in $\{0, 1\}$. Therefore, we also wish to generate an index list for all the variables that are boolean, which is done during constraint generation.

To index the time variables, we make use of *si_list*, just as a flat structure, laying out all edges and iterate through its elements. Similarly, δ is indexed by flattening out *si_list*, but on one level higher, such that essentially one element in the list corresponds to one path for some *si*. α comes in the context of edges, so we use $\alpha index$ map for its indexing. The precise indexing of β does not matter, as we will see in the next section. The summary of the variables is found below:

- s: length = $|SO_1| + |SO_2|$
- t^{in}: length = $\sum_{i=1}^{|si\_list|} \sum_{r \in si\_list_i} |r|$
- t^{out}: length = $\sum_{i=1}^{|si\_list|} \sum_{r \in si\_list_i} |r|$)
- δ: length = $\sum_{i=1}^{|si\_list|} |si\_list_i|$, boolean
- α: length = $\sum_{si \in SI} |E_{si}|$, boolean
- δ: length = $|beta\_index|$, boolean

3.3.3.7 Loading constraints and objective function

To store the constraints, we wish to build matrix A and vector b, such that all constraints can be expressed as $A\theta \leq b$. Equality can be rewritten as two inequality

constraints, so for a constraint $d^T\theta = k$ we insert the following: $d^T\theta \leq k$ and $-d^T\theta \leq -k$.

However, it is important to consider memory limitations when working with matrices. For an $\mathbb{R}^{m\times n}$ matrix, the space complexity is $\mathcal{O}(mn)$. The smallest real life problem presented by SBB comes out to have a 27344 variables and 88789 constraints. Due to calculations in seconds, and presence of large linearisation constant M, we must use at least an unsigned int32 datatype for our matrix, which is stored in 4 bytes. Thus, the memory need for such a matrix is $27344 \times 88789 \times 4B = 9.04GB$. Further, we already discussed that the number of variables scales exponentially, thus the space required for storing our problem does too. To work around this issue, we note that most rows in our matrix will be zeros, as most constraints have no more than 5 variables at a time. We will therefore be using a sparse matrix class from Scipy. This is a class that stores indices of non-zero elements, drastically reducing space needed to store a sparse matrix. Theoretically, a $\mathbb{R}^{m\times n}$ matrix with a sparsity density $\rho = \frac{\text{number of non-zero elements}}{m\times n}$ will only need to store the coordinates i, j and data d for non-zero elements. ρ can also be seen as the average number of non-zero elements per row, which in our case is the number of variables per constraint n_c, such that $\rho = \frac{n_c}{n}$. Thus, our space complexity for such a matrix reduces to $\mathcal{O}(n_c m)$. Furthermore, n_c for the TTP does not change with scale, as the only thing growing is number of constraints. Hence, the space complexity of using a sparse matrix for a TTP is $\mathcal{O}(m)$, i.e. scales linearly with the number of constraints. With regard to time-complexity, regular arrays are faster due to constant time indexing. Sparse matrices come in three main formats: list of sparse rows, list of sparse columns or a list of coordinate-data tuples. When looping over service intentions, routes and route sections, we will constantly be indexing the matrices. Thus, the best choice to avoid re-looping over list of sparse rows/columns, it is best to generate a list of coordinate-data tuples, and then convert to a list of sparse rows for faster arithmetic operations. For the vector b, we can use a regular array, as it is not as sparse, and will scale linearly with the number of constraints.

Before loading the constraints, everything in (3.4) - (3.14) is rearranged such that all the variables are on the lesser side of the inequality, and constants on the opposite side. We then set (3.4) - (3.10) by looping over all service intentions si, their routes r and route sections rs in every root, while enumerating the inner index i and the index at the middle level, j. Keeping in mind that our variables are grouped by service intentions, we introduce a map $currlen(si)$ which returns the offset needed for the current service intention. We further define indices s_idx,

tin_idx, $tout\_idx$, δ_idx and x_idx that are updated every loop. We keep track of number of constraints, such that correct, consecutive rows are added. Using i to generate $tin\_idx = currlen(si) + |s_{si}| + i$ and $tin\_idx = currlen(si) + |s_{si}| + |t_s^{in}i|$ we obtain the correct ordering of variables in θ.

Constraints (3.8) - (3.10) can be set in the same loop. To make sure we only load the constraints from the sets S_{ei}, S_{eo} and S_{mt} as required, we implement checks on presence of parameters $EarIn$, $EarOut$ and mrt or mst. From section 3.3.3.3, recall that our DAG $G_{si} = (V_{si}, E_{si}, D_{si})$ has an edge component $e \in E_{si}$, which has a corresponding data component $d \in D_{si}$. The data component for an edge is stored as a hash table, and the relevant parameters in the data component are also hashed, and can thus be retrieved in $\mathcal{O}(1)$ time. If parameter is present, time is indexed using tin/out_idx as discussed, and path variable δ is indexed with $\delta\_idx = currlen(si) + |s_{si}| + |t_s^{in}i| + |t_s^{out}i| + j$. Enumerating slack with its own variable k for the inner loop, we check whether edge rs has parameters $LatOut$ or $LatIn$, in which case (3.2) and (3.3) are loaded with slack index $currlen(si) + k$, and k is incremented.

Further, we load constraint (3.7) by indexing δ and α using δ_idx and α_index respectively, such that $\alpha\_idx = currlen(si)+|s_{si}|+|t_s^{in}i|+|t_s^{out}i|+|\delta_{si}|+\alpha\_index_{si,rs}$.

To load constraint (3.6), we set up a temporary list that stores data and δ_{idx} within the middle loop, appending it to global row and data lists in the outer loop.

Coupling constraints are set in a separate loop going through $beta\_index$. Using $delta\_index\_by\_edge$ to get indices of all routes given a service intention and an edge, we set constraints (3.11) - (3.14) by looping over all possible δ indices for both service intentions, indexing t using $edges\_by\_path$ and β by tracking the index of the outer loop while adding appropriate offsets, then set constraints.

The objective function vector c can be filled using the same indexing system as constraints (3.2) - (3.3) and (3.7).

Summary of the above is represented in algorithm (5).

3.3.3.8 Solver and parsing the output

To solve the problem, the user instantiates a MOSEK environment, which needs to set an objective function and a set of constraints. We define our variable with the total length of all variables, passing it a list of indices that need to be binary. Thereafter, we call for a solution using the MOSEK optimizer. Under the hood, MOSEK uses various simplifications to reduce the problem, and then applies Branch-and-Bound (section 2.2.5.1) together with Simplex Algorithm or Interior Point Methods (sections 2.2.4.1 and 2.2.3.1). If the solver finds a feasible solution, our variables will come out with numerical values. Since only one path per service intention is selected, we find the path where $\delta = 1$, and from there we calculate all the indices in t^{in} and t^{out} that correspond to that path by inverting the mapping functions defined in section (3.3.3.6). Lastly, we list them in a human readable format.

Algorithm 5: Loading all constraints

Result: Matrices A, b and c
Initialise $row, col, data, c$;
forall $si \in si\_list$ **do**
  $i, j, k \leftarrow 0, 0, 0$;
  Initialise $tempcol, temprow$;
  forall $r \in si$ **do**
    forall $rs \in r$ **do**
      $slack\_idx \leftarrow currlen(si) + k$;
      $tin\_idx \leftarrow currlen(si) + |s_{si}| + i$;
      $tout\_idx \leftarrow currlen(si) + |s_{si}| + |t^{in}_{si}| + i$;
      $\delta\_idx \leftarrow currlen(si) + |s_{si}| + |t^{in}_{si}| + |t^{out}_{si}| + j$;
      $\alpha\_idx \leftarrow currlen(si) + |s_{si}| + |t^{in}_{si}| + |t^{out}_{si}| + |\delta_{si}| + \alpha\_index_{si,rs}$;
      $row, col, data \leftarrow (row, col, data) \cup \{3.4(tin\_idx, tout\_idx)\}$;
      if *rs not last in r* **then**
        $row, col, data \leftarrow (row, col, data) \cup \{3.5(tin\_idx + 1, tout\_idx)\}$;
      if $EarIn \in D_{si,rs}$ **then**
        $row, col, data \leftarrow (row, col, data) \cup \{3.8(tin\_idx)\}$;
      if $EarOut \in D_{si,rs}$ **then**
        $row, col, data \leftarrow (row, col, data) \cup \{3.9(tout\_idx)\}$;
      if $mrt \vee mst \in D_{si,rs}$ **then**
        $row, col, data \leftarrow (row, col, data) \cup \{3.10(tin\_idx, tout\_idx)\}$;
      if $LatIn \in D_{si,rs}$ **then**
        $row, col, data \leftarrow (row, col, data) \cup \{3.2(slack\_idx)\}$;
        $k \leftarrow k + 1$, $c_{slack\_idx} \leftarrow 1$;
      if $LatOut \in D_{si,rs}$ **then**
        $row, col, data \leftarrow (row, col, data) \cup \{3.3\}(slack\_idx)$;
        $k \leftarrow k + 1$, $c_{slack\_idx} \leftarrow 1$;
      if $p \in D_{si,rs}$ **then**
        $c_{\alpha\_idx} \leftarrow p$;
      $row, col, data \leftarrow (row, col, data) \cup \{3.7(\delta\_idx, \alpha\_idx)\}$;
      $i \leftarrow i + 1$;
    end
    $tempcol, tempdata \leftarrow (tempcol, tempdata) \cup (\{\delta\_idx, \delta\_idx\}, \{1, -1\})$;
    $j \leftarrow j + 1$;
  end
  $row, col, data \leftarrow (row, col, data) \cup \{3.6(temprow, tempcol)\}$;
end
$i \leftarrow 0$;
forall $(si_1, si_2, rs) \in beta\_index$ **do**
  forall $id_1 \in delta\_index\_by\_edge[si_1][rs]$ **do**
    forall $id_2 \in delta\_index\_by\_edge[si_2][rs]$ **do**
      let $edges\_by\_path[id_n][rs] = ebp_n$;
      $tin_n\_idx \leftarrow epb_n + currlen(si_n) + |s_{si_n}|$;
      $tout_n\_idx \leftarrow epb_n + currlen(si_n) + |s_{si_n}| + |t^{in}_{si_n}|$;
      $\delta_n\_idx \leftarrow id_n + currlen(si_n) + |s_{si_n}| + |t^{in}_{si_n}| + |t^{out}_{si_n}|$;
      $\beta\_idx \leftarrow i + \sum_{si \in SI} currlen(si)$;
      $row, col, data \leftarrow$
        $(row, col, data) \cup \{(3.11 - 3.14)(tin_{si_n}, tin_{si_n}, \delta_n\_idx, \beta\_idx)\}$;
    end
  end
  $i \leftarrow i + 1$;
end
$A, b \leftarrow sparse\_matrix(row, col, data)$;
return A, b

Chapter 4

Solving the MILP using ADMM

4.1 Introduction to Alternating Direction Method of Multipliers

Alternating Direction Method of Multipliers (ADMM) is an algorithm predominantly applied to coupled optimization problems [4]. Such problems consist of local subproblems that require a common consensus, which is usually referred to as coupling or complicating constraints. This is exactly the case in the Train Timetabling Problem; we are scheduling trains that are to conform to their own physical constraints and business rules, but **also** not collide with the other trains in the network.

Suppose we have a problem of the following form:

$$\begin{aligned} \min_{x,z} \quad & f(x) + g(z) \\ \text{subject to:} \quad & A^{loc}x \leq b^{loc} \\ & C^{loc}z \leq d^{loc} \\ & Ax + Bz = c \end{aligned} \tag{4.1}$$

We observe that if the condition $Ax + Bz = c$ did not exist, we could easily have

split our problems into two subproblems in x and z:

$$\begin{aligned} \min_{x} \quad & f(x) \\ \text{subject to:} \quad & A^{loc}x \leq b^{loc} \end{aligned} \tag{4.2}$$

$$\begin{aligned} \min_{z} \quad & g(z) \\ \text{subject to:} \quad & C^{loc}z \leq d^{loc} \end{aligned} \tag{4.3}$$

In this case, $Ax + Bz = c$ is the complicating constraint. We could of course solve this problem by concatenating x and z and obtain a solution using the standard Mathematical Programming techniques discussed in section 2.2, however, that would not utilise the convenient structure of the problem, resulting in longer computation times. Instead we use ADMM, which works by 'dualising' the complicating constraint in Augmented Lagrangian Form as discussed in section 2.2.7, having the objective function

$$J(x, z, \lambda) = f(x) + g(z) + \lambda^T(Ax + Bz - c) + \frac{\rho}{2}\|Ax + Bz - c\|_2^2 \tag{4.4}$$

which is to be solved subject to the local constraints. The problem is iteratively solved using the Dual Ascent (algorithm 2), but instead of simultaneously minimising with respect to both x and z, we do it separately, using known solutions for the other variable from previous iterations:

$$\begin{aligned} x^{n+1} &= \arg\min_{x} J(x, z^n, \lambda^n) \\ z^{n+1} &= \arg\min_{z} J(x^{n+1}, z, \lambda^n) \\ \lambda^{n+1} &= \lambda^n + \rho(Ax^{n+1} + Bz^{n+1} - c) \end{aligned} \tag{4.5}$$

In the paper *Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers* [4], the authors provide a proof that if the functions f and g are proper, closed and convex, then ADMM is guaranteed to converge towards the optimum value of the primal problem. Note that although a MILP is by definition non-convex due to its feasible region (section 2.2.5), the linear objective

function is proper, closed and convex, thus the proof of convergence still holds.

4.2 Mathematical reformulation of the TTP

The TTP formualted as a MILP in section 3.2 is indeed a problem that decomposes into smaller subproblems with added coupling constraints. Inspecting algorithm 5, we see that the constraints for each service intention are loaded sequentially, followed by the constraints relating pairs of service intentions together. As a result, the constraint matrix comes out block-diagonal, where each block is specific to a service intention with all the coupling constraints in a common block on the bottom (see equation (4.6)).

Even if the coupling constraints did not exist and trains were free to phase through one another like ghosts, we would still have an integer program, but one that has a significantly smaller number of constraints. Furthermore, we could solve these problems separately for every service intention, further reducing the computation times. We wish to utilize this structure, and ADMM is an algorithm that is suitable for this procedure. To do so, we need to transform the problem into Augmented Lagrangian form (equation (4.4)). That still results in quite a large problem, so instead we will break it up into subproblems that involve pairs of service intentions, which are to be solved serially for every iteration. This section is inspired by Luan et al. [12], where the authors apply various serial MILP algorithms on TTPs decomposed into geographical, temporal and train based problems.

4.2.1 Reformulation of the TTP in Augmented Lagrangian form

The overall TTP formulated in the section 3.2 Mathematical Model can be written as following:

$$\min[c_{si_1}, c_{si_2}, \cdots, 0] \begin{bmatrix} \theta_{si_1} \\ \theta_{si_2} \\ \vdots \\ \theta_{si_{|SI|}} \\ \beta \end{bmatrix} \tag{4.6}$$

Subject to:

$$\begin{bmatrix} A_{si_1} & 0 & \cdots & 0 & 0 \\ 0 & A_{si_2} & \cdots & 0 & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & \cdots & A_{si_{|SI|}} & 0 \\ \cdots & \cdots & A_\beta & \cdots & \cdots \end{bmatrix} \begin{bmatrix} \theta_{si_1} \\ \theta_{si_2} \\ \vdots \\ \theta_{si_{|SI|}} \\ \beta \end{bmatrix} \leq \begin{bmatrix} b_{si_1} \\ b_{si_2} \\ \vdots \\ b_{si_{|SI|}} \\ b_\beta \end{bmatrix}$$

All scalar elements in δ, α, β are binary

Each θ_{si} is a vector of variables $[s_{si}, t^{in}_{si}, t^{out}_{si}, \delta_{si}, \alpha_{si}]^T$, as defined in section 3.3.3.6, $A_s i$ is the train specific constraint matrix and $\beta_s i$ is the corresponding left hand side. The bottom block, $A_\beta \beta \leq b_\beta$ corresponds to all the coupling constraints from equations (3.11)-(3.14). Similarly, the objective function $c = [c_{si_1}, c_{si_2}, \cdots, 0]^T$ is composed of all the train-specific objectives, with zeros at the position of coupling, as no penalty can occur there.

Neglecting the coupling, every service intention will have its own problem:

$$\begin{aligned} \min_{\theta_{si}} \quad & c^T_{si} \theta_{si} \\ \text{subject to:} \quad & A^T_{si} \theta_{si} \leq b_{si} \quad \forall si \in SI \\ & \text{All scalar elements in } \delta_{si}, \alpha_{si} \text{ are binary} \end{aligned}$$

Of course we cannot simply neglect the coupling, as that would mean trains are free to collide, as they may choose to use the same route sections one another. Instead, noting that no more than two trains can have a coupling per linear constraint (as seen in equations (3.11)-(3.14)), we decompose the coupling constraints $A_\beta \beta \leq b_\beta$ into couplings involving service intention pairs $p, q \in SI$.

For every service intention p we wish to formulate a subproblem that involves constraints on all trains q that p has coupling with, i.e. service intentions that share common route sections. Denote this set of trains $Q_p \subset SI$.

Consider a scenario with only two service intentions. Referring back to the standard form shown in equation (4.1), the service intention state-vectors would correspond to x and z. Thus, for p and q, we need to formulate their coupling constraints in the form $A_\beta^{(p,q)}\theta_p + A_\beta^{(q,p)}\theta_q \leq b_\beta^{(p,q)}$, where $A_\beta^{(p,q)}$ contains all variables belonging to p, $A_\beta^{(q,p)}$ contains all variables belonging to q and $b_\beta^{(p,q)}$ being the RHS of the inequalty. However, a complication arises; the coupling variables $\beta_{p,q,rs}$ do not belong to either train p or q, but rather belong to both, so in order to be able to decompose the coupling constraint into the separated form, we instead assign the common $\beta_{p,q,rs}$ to both trains. This can be done by using a trick; giving half the value of the variable to p, and half to q. We illustrate this with an example on constraint (3.11):

$$t^{in}_{p,r_1,rs} - t^{in}_{q,r_2,rs} \leq M(1 - \beta_{p,q,rs}) + M(2 - \delta_{p,r2} - \delta_{q,r_2})$$

Rearranging the above,

$$(t^{in}_{p,r_1,rs} + M\delta_{p,r_2} + \frac{1}{2}M\beta_{p,q,rs}) + (-t^{in}_{q,r_2,rs} + \delta_{q,r_2} + \frac{1}{2}M\beta_{p,q,rs}) \leq 3M$$

Such that the constraint matrices $A_\beta^{(p,q)}$ and $A_\beta^{(q,p)}$ for this singular constraint are $[0, 1, 0, M, 0, \frac{1}{2}M, 0]$ and $[0, -1, 0, M, 0, \frac{1}{2}M, 0]$, where the zeros are zero-vectors of appropriate length. Denote the coupling matrix with only variables from p, excluding β, $A_{-\beta}^{(p,q)}$, and let $L_{p,q}$ be the matrix that only contains β (note that L is symmetrical in p and q such that $L_{p,q} = L_{q,p}$. Thus,

$$A_\beta^{(p,q)} = [A_{-\beta}^{(p,q)}, \frac{1}{2}L_{p,q,}] \tag{4.7}$$

$$A_\beta^{(p,q)} = [A_{-\beta}^{(q,p)}, \frac{1}{2}L_{p,q}] \tag{4.8}$$

Figure 4.1 shows the sparsity plot (shaded region denotes non-zero elements) of a constraint matrix from a scenario provided by SBB. a) shows the local constraints, while b) is the coupling matrix, and in the full problem is a) stacked on top of

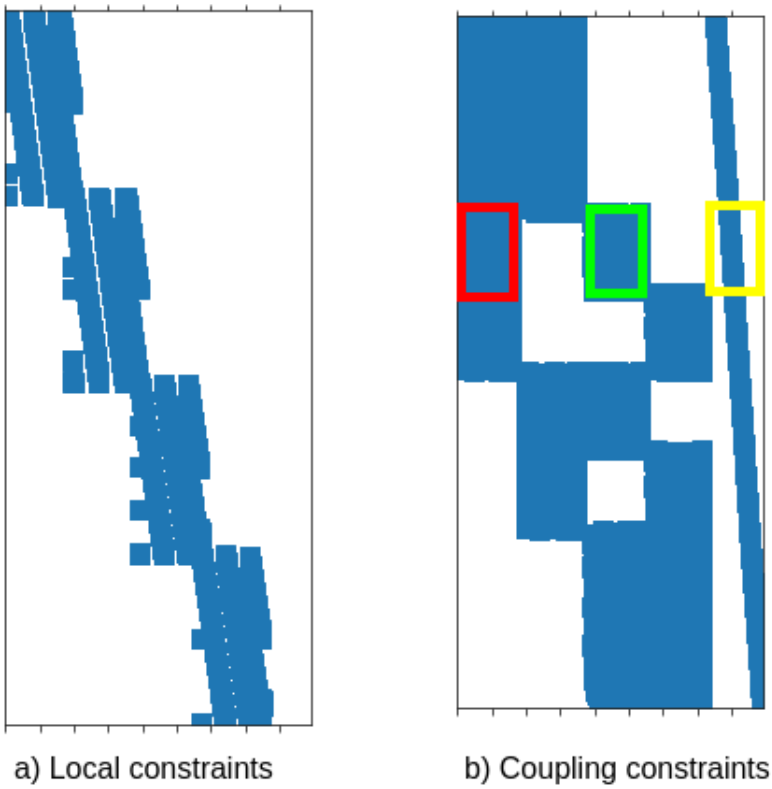


Figure 4.1: Constraint matrix for 4 trains

b). We can see how block pairs of train-specific variables in b) form pairwise train constraints, in sequence of (1,2), (1,3), (1,4), (2,3), (2,4) and (3,4), and the slant line on the far right right corresponds to the coupling variables β. Three blocks were highlighted corresponding to trains 1 and 3; The leftmost block (red) is $A_{-\beta}^{(1,3)}$, the middle (green) is $A_{-\beta}^{(3,1)}$ and the rightmost block (yellow) is $L_{1,3}$. In order to construct $A_{\beta}^{(1,3)}$ and $A_{\beta}^{(3,1)}$, we simply need to cut out the relevant matrices, and concatenate them together as $A_{\beta}^{(1,3)} = [A_{-\beta}^{(1,3)}, \frac{1}{2}L_{1,3,}]$ and $A_{\beta}^{(3,1)} = [A_{-\beta}^{(3,1)}, \frac{1}{2}L_{1,3}]$.

Having a framework for the constraint decomposition, we can rewrite problem (4.6) as an equivalent set of coupled subproblems:

$$
\begin{aligned}
\min_{\theta_p} \quad & c_p^T \theta_p \\
\text{subject to:} \quad & A_\beta^{(p,q)} \theta_p + A_\beta^{(q,p)} \theta_q \leq b_\beta^{(p,q)} \quad \forall q \in Q_p \\
& A_p \theta_p \leq b_p \\
& \text{All scalar elements in } \delta_p, \delta_q, \alpha_p, \alpha_q, \beta_{qp} \text{ are binary } \forall q \in Q_p \\
& \forall p \in SI
\end{aligned}
$$

ADMM in [4] is formulated using equality constraints; however the paper *Distributed Convex Optimization with Many Non-Linear Constraints* [12] shows that the inequality constraints can be dealt with by applying a $\max(0, \cdot)$ function; That is, if a constraint is given by $g(x) \leq 0$, we can equivalently rewrite it as $\max\{0, g(x)\} = 0$, seeing that if $g(x) \leq 0$, constraint reduces to $0 = 0$, satisfying our inequality, otherwise we have $g(x) = 0$ which still satisfies our inequality:

$$
\begin{aligned}
\min_{\theta_p} \quad & c_p^T \theta_p \\
\text{subject to:} \quad & \max\{0, A_\beta^{(p,q)} \theta_p + A_\beta^{(q,p)} \theta_q - b_\beta^{(p,q)}\} = 0 \quad \forall q \in Q_p \\
& A_p \theta_p \leq b_p \\
& \text{All scalar elements in } \delta_p, \delta_q, \alpha_p, \alpha_q, \beta_{qp} \text{ are binary } \forall q \in Q_p \\
& \forall p \in SI
\end{aligned}
\tag{4.9}
$$

There is no need to apply the inequality to equality transformation technique to the local constraints, as they are assumed to be 'easy' to solve, and we have no intention of dualising them.

Rewriting subproblem p from (4.9) in Augmented Lagrangian form gives the following objective function:

$$
\begin{aligned}
J(\theta, \Lambda) = \quad & \sum_{p \in SI} (c_p^T \theta_p + \sum_{q \in Q_p} \lambda_{p,q}^T \max\{0, A_\beta^{(p,q)} \theta_p + A_\beta^{(q,p)} \theta_q - b_\beta^{(p,q)}\} \\
+ \quad & \frac{\rho}{2} \sum_{q \in Q_p} \|\max\{0, A_\beta^{(p,q)} \theta_p + A_\beta^{(q,p)} \theta_q - b_\beta^{(p,q)}\}\|_2^2)
\end{aligned}
\tag{4.10}
$$

where the $\lambda_{p,q}$ are Lagrange multipliers[1] for service intention p coupled to q, and $\Lambda = \{\lambda_{p,q} | p \in SI, q \in Q_p\}$ is the set of all the multipliers.

4.2.2 Applying ADMM to the MILP in Augmented Lagrangian from

The resulting problem in Augmented Lagrangian form (equation 4.10) is still very large, as we are solving it simultaneously for all trains. ADMM solves the problem variable by variable, which means that for train p, only terms involving p will be 'active' when solving the problem with respect to θ_p in an ADMM iteration. Furthermore, the resulting constraint matrix is fully block-diagonal, which means we can rewrite this problem into an equivalent form composed of subproblems (one for every service intention) that are to be solved serially. Thus, every subproblem will have the following objective:

$$
\begin{aligned}
J_p(\theta, \Lambda) = \quad & c_p^T \theta_p + \sum_{q \in Q_p} \lambda_{p,q}^T \max\{0, A_\beta^{(p,q)} \theta_p + A_\beta^{(q,p)} \theta_q - b_\beta^{(p,q)}\} \\
+ \quad & \frac{\rho}{2} \sum_{q \in Q_p} \|\max\{0, A_\beta^{(p,q)} \theta_p + A_\beta^{(q,p)} \theta_q - b_\beta^{(p,q)}\}\|_2^2
\end{aligned}
$$

where Λ and θ is a vector containing all state vectors in SI without loss of generality, as service intentions not coupled to p will simply not be present in the expression. Before we can use the Augmented Lagrangian in a solver, we need to rewrite it in a standard form, adding local constraints, and getting rid of the max operator using slack variables, as previously done in section 3.2.2:

[1] Note that the multipliers are not symmetrical in p and q ($\lambda_{p,q} \neq \lambda_{q,p}$)

$$
\begin{aligned}
\min_{\theta_p, s_{p,q} \forall q \in Q_p} \quad & J_p(\theta, \Lambda) = c_p^T \theta_p + \sum_{q \in Q_p} \left[\lambda_{p,q}^T s_{p,q} + \frac{\rho}{2} s_{p,q}^T I s_{p,q} \right] \\
\text{subject to:} \quad & s_{p,q} \geq A_\beta^{(p,q)} \theta_p + A_\beta^{(q,p)} \theta_q - b_\beta^{(p,q)} \quad \forall q \in Q_p \\
& s_{p,q} \geq 0 \\
& A_p \theta_p \leq b_p \\
& \text{All scalar elements in } \delta_p, \delta_q, \alpha_p, \alpha_q, \beta_{qp} \text{ are binary } \forall q \in Q_p \\
& \forall p \in SI
\end{aligned}
\tag{4.11}
$$

Where $s_{p,q}$ is the slack variable for p coupling to q, and I is the identity matrix. Note, that the resulting program is a Mixed Integer Quadratic Program (MIQP), which we are using to solve a Mixed Integer Linear Program, which is an interesting outcome. Nevertheless, with integer constraints relaxed, the resulting program is convex and Interior Point Methods (section 2.2.3.1) can be used for every iteration of the Branch-and-Bound algorithm (section 2.2.5.1).

During Dual Ascent in conventional ADMM, we iteratively solve the same problem for one variable at a time using previous solutions for every iteration. Here, we will solve a whole set of different subproblems, but we will still use the previously computed variables. For a problem p in the ordered set of problems $\Pi = [P_1, ..., P_p, ..., P_{|SI|}]$, the update rule at iteration $n+1$ will therefore be as following for all p:

$$
\theta_p^{n+1} = \arg\min_{\theta_p} J(\theta_1^{n+1}, \theta_2^{n+1}, ..., \theta_p, \theta_{p+1}^n, ..., \theta_{|SI|}^n, \Lambda^n) \tag{4.12}
$$

which is the solution to the program for a chosen $p \in SI$ (4.11).

We are picking problems from an ordered set, so the indices say that we pick the most recent solution for all non-active θ_j, that is from the current iteration for all problems before p, and from the previous iteration for all the subsequent problems. After solving all the subproblems in iteration $n+1$, we climb in the direction of steepest ascent by updating the multipliers. The gradient is $\nabla_{\lambda_{p,q}} J(\theta, \Lambda) = \max\{0, A_\beta^{(p,q)} \theta_p + A_\beta^{(q,p)} \theta_q - b_\beta^{(p,q)}\}$, so we update $\lambda_{p,q} \in \Lambda$ as following:

$$\lambda_{p,q}^{n+1} = \lambda_{p,q} + \rho \max\{0, A_\beta^{(p,q)}\theta_p^{n+1} + A_\beta^{(q,p)}\theta_q^{n+1} - b_\beta^{(p,q)}\} \tag{4.13}$$

The complexity of a MIQP is similar to MILP as both employ tree search over the feasible domain to deal with the integer valued variables, however, we have reduced the number of strict constraints, and instead increased the number of problems. Time complexity obviously scales linearly with the number of problems, but reduces polynomially with respect to the number of constraints. The reason it does not reduce exponentially, is that we still require all the same variables to be binary, which is the biggest computational expense.

4.3 Implementation

4.3.1 Gathering all the parameters

The parser created in section 3.3.3 generates the full matrix A and vectors b and c, together with a list of indices that need to be binary. In order to set up the problem, we need to extract and store all A_p, b_p, c_p, $A_\beta^{(p,q)}$ and $b_\beta^{(p,q)}$ for all $p \in SI, \forall q \in Q_p$, modifying $A_\beta^{(p,q)}$ according to equation (4.7). To know what slice ranges we need, we can add a tracker to algorithm 5 which will store width and height indices for the various service intentions, as well as their couplings. It should be noted that $b_\beta^{(p,q)} = b_\beta^{(q,p)}$, and that $A_\beta^{(p,q)}$ for $(p,q) = (si_1, si_2)$ is equal to $A_\beta^{(q,p)}$ for $(p,q) = (si_2, si_1)$. This trivial observation can save us a lot of memory; generating these parameters for all $p \in SI, \forall q \in Q_p$ as a permutation loop, we choose to do it in a combination loop of every coupling. We call this set $\mathcal{C}$, which can be generated in algorithm 4. Thus, we create a class for every sub-subproblem, for all combinations of service intentions (listing 1).

To avoid recomputing matrices during ADMM, we initialise a dictionary of sub-subproblems to store all the information. This is a prime example of time vs memory tradeoff, but considering the fact that we have been storing the full matrices for the naive MILP solution approach, this seems reasonable.

Furthermore, we wish to initialise the dual variables, i.e. the Lagrangian multipliers. They are vectors of length equal to the number of coupling constraints there are for

```
import numpy
class SubSubProblem:
    def __init__(self, id1, id2, A, b, c):
        A1, A2, b1, b2, c1, c2 = self.get_local(id1, id2, A, b, c)
        self.A_local = {id1: A1, id2: A2}
        self.b_local = {id1: b1, id2: b2}
        self.c_local = {id1: c1, id2: c2}
        A_12, A_21, b_beta = self.get_coupling(id1, id2, A, b)
        self.A_coupling = {id1: A_12, id2: A_21}
        self.b_coupling = b_coupling
        self.initialise_dual()
    def get_local(self, id1, id2, A, b, c):
    def get_coupling(self, id1, id2, A, b):
    def initialise_dual(self, value):
        self.dual = {id1: numpy.full(self.A_coupling[id1].shape[0], value),
                     id2: numpy.full(self.A_coupling[id2].shape[0], value)}
```

Listing 1: Class to store information for all couplings

a coupling between two service intentions, but note that even though the lengths of the Lagrangian multipliers are equal for service intention p and q, they are not the same variable.

4.3.2 Setting up the solver

The solver underneath the hood will still be MOSEK, but we are required to set up the correct formulation. First, we initialise our solutions θ_p^0 for all $p \in SI$. We could solve every uncoupled problem to do so, but that will make a differnce of one iteration at most, so we choose to simply let all initial solutions be zero vectors. As the termination critereon, we wish to put a tolerance on a p-norm of the constraint violation. Constraint violation for a pair of service intentions is given by $\max\{0, A_\beta^{(p,q)}\theta_p + A_\beta^{(q,p)}\theta_q - b_\beta^{(p,q)}\}$, as we do not wish to penalise negative numbers. The choice of norm is a difficult; there are different meanings to the residuals of the constraints, for instance one may be a violation of time, while the other may be a violation of a binary variable, making standardisation of tolerance difficult. To assign the tightest bound, we can choose the ∞-norm, as it will extract the maximum absolute error from the error vector. Then, we assign a tolerance, at which the algorithm terminates or a maximum number of iterations.

To actually solve the problems, we loop over all service intentions in SI, and for

each we assemble the objective function by doing an inner loop over all the service intentions it couples to. MOSEK API does not operate with arithmetic operators the way Numpy does, and nor does it accept Scipy sparse matrices, but has its own sparse matrix class instead. Nevertheless, Scipy matrices were converted to MOSEK's Matrix.Sparse objects, and the objective function constructed the inbuilt methods that MOSEK features.

One caveat is that MOSEK is based on Conic Programming, which is all about embedding Mathematical Programs into cones, as the creators claim it is more robust and numerically stable. Essentially, Linear and Quadratic Programs are a subset of Conic Programming, as they can be embedded into cones[2]. Intersecting a cone with a hyperplane produces the epigraph[3] of the desired function, and minimising over the epigraph is equivalent to minimising over the function itself [23]. We are interested in embedding the quadratic term, $s_{p,q}^T I s_{p,q}$ into a cone. For that, we choose a rotated quadratic cone $\mathcal{K}_r^n = \{x \in \mathbb{R}^n | 2x_1 x_2 \geq \sum_{i=3}^n x_i^2\}$, and embed the epigraph of $s_{p,q}^T I s_{p,q}$ into the cone using a dummy variable for t for x_1 and a constant of $1/2$ to get it into the desired epigraph $t \geq s_{p,q}^T s_{p,q}$, i.e. $[t, 1/2, s_{p,q}] \in \mathcal{K}_r^{|s_{p,q}|+2}$ [17]. Lastly, we add t to the objective function such that it is minimised. The resulting program MOSEK is solving is of the following form:

$$
\begin{aligned}
\min_{t_{p,q},\theta_p,s_{p,q}\forall q\in Q_p} \quad & J_p(\theta,\Lambda) = c_p^T\theta_p + \sum_{q\in Q_p}\left[\lambda_{p,q}^T s_{p,q} + \frac{\rho}{2}t_{p,q}\right] \\
\text{subject to:} \quad & s_{p,q} \geq A_\beta^{(p,q)}\theta_p + A_\beta^{(q,p)}\theta_q - b_\beta^{(p,q)} \quad \forall q \in Q_p \\
& s_{p,q} \geq 0 \\
& A_p\theta_p \leq b_p \\
& [t, 1/2, s_{p,q}, t_{p,q}] \in \mathcal{K}_r^{|s_{p,q}|+2} \\
& \text{All scalar elements in } \delta_p, \delta_q, \alpha_p, \alpha_q, \beta_{qp} \text{ are binary } \forall q \in Q_p
\end{aligned}
\tag{4.14}
$$

After solving every problem, we update our set of solutions, which are to be used in the subsequent problems. After all the problems are solved in the iteration, we

[2]Formally, a cone $\mathcal{K}$ is a set of points $x \in \mathcal{K}$ such that $\gamma x \in \mathcal{K}$ for all $\gamma \in \mathbb{R}^+$

[3]An epigraph of a function $f(x)$ is $\{(x,t)|t \geq f(x)\forall x\}$, i.e. all the points that lay above the function

can update all the Lagrange multipliers as $\lambda_{p,q} = \rho \max\{0, A_\beta^{(p,q)}\theta_p + A_\beta^{(q,p)}\theta_q - b_\beta^{(p,q)}\}$ for all $p \in SI, \forall q \in Q_p$, and compute the norms. Note, that the ∞-norm is applied to errors in the **whole** state vector θ, which can be achieved through the following operation:

$$\max\{\|\max\{0, A_\beta^{(p,q)}\theta_p + A_\beta^{(q,p)}\theta_q - b_\beta^{(p,q)}\}\|_\infty \mid p \in SI \quad \forall q \in Q_p\}$$

yielding the maximum error in the overall problem. The implementation is briefly summarised in algorithm 6, which omits precise MOSEK definitions and parameters for convenience, as they are implied by the problem.

Algorithm 6: Solving the TTP using ADMM

Result: Solution to the TTP
Get A, b, c and $bool\_index$ from algorithm 5;
Initialise parameters ρ, tolerance ϵ, max iterations $\max\_iter$;
Initialise dictionary of subsubproblems Π;
Set $error = \infty$;
forall $(p, q) \in combinations(SI, 2)$ **do**
  $\Pi \leftarrow \Pi \cup \{SubSubProblem(p, q, A, b, c)\}$;
end
Initialise $\theta_p = 0 \quad \forall p \in SI$;
$i \leftarrow 1$;
while $error \geq \epsilon \vee i \leq \max\_iter$ **do**
  forall $p \in SI$ **do**
    Set new MOSEK environment;
    $\theta_p \leftarrow MOSEK.Variable(bool\_index = bool\_index)$;
    $J_p \leftarrow 0$;
    $c_p \leftarrow 0$;
    forall $q \in Q_p$ **do**
      Using MOSEK arithmetic and matrix operations;
      $s_{q,p} \leftarrow MOSEK.Variable()$;
      $t_{q,p} \leftarrow MOSEK.Variable()$;
      $J_p \leftarrow J_p + \Pi_{p,q}.\lambda_{p,q}^T s_{p,q} + \frac{\rho}{2} t_{p,q}$;
      $MOSEK.Constraint(s_{p,q} \geq 0)$;
      $MOSEK.Constraint(s_{p,q} \geq$
       $\Pi_{p,q}.A_\beta^{(p,q)}\theta_p + \Pi_{p,q}.A_\beta^{(q,p)}\theta_q - \Pi_{p,q}.b_\beta^{(p,q)})$;
      $MOSEK.Constraint([t, 1/2, s_{p,q}] \in \mathcal{K}_r^{|s_{p,q}|+2})$;
      $c_p \leftarrow \Pi_{p,q}.c$;
      $A_p \leftarrow \Pi_{p,q}.A_p$;
      $b_p \leftarrow \Pi_{p,q}.b_p$;
    end
    $J_p \leftarrow J_p + c_p^T \theta_p$;
    $MOSEK.Constraint(A_p\theta_p \leq b_p)$;
    $MOSEK.objective(\min J_p)$;
    $\theta_p \leftarrow MOSEK.solve()$;
  end
  $E \leftarrow \emptyset$;
  forall $p \in SI$ **do**
    forall $q \in Q_p$ **do**
      $\Pi_{p,q}.\lambda_{p,q} \leftarrow \Pi_{p,q}.\lambda_{p,q} + \rho(\Pi_{p,q}.A_\beta^{(p,q)}\theta_p + \Pi_{p,q}.A_\beta^{(q,p)}\theta_q - \Pi_{p,q}.b_\beta^{(p,q)})$;
      $E \leftarrow E \cup \{\|\max\{0, \Pi_{p,q}.A_\beta^{(p,q)}\theta_p + \Pi_{p,q}.A_\beta^{(q,p)}\theta_q - \Pi_{p,q}.b_\beta^{(p,q)}\}\|_\infty\}$;
    end
  end
  $error \leftarrow \max E$;
  $i \leftarrow i + 1$;
end
return $\theta_p \quad \forall p \in SI$

Chapter 5

Experiments

5.1 TTP to MILP parser

In this section, we evaluate the performance of the TTP to MILP parser that was implemented in section 3.3.3. This section will focus on time and space usage, which will aid in evaluating the feasibility of a MILP solver. We test the parser on scenarios provided by SBB.

5.1.1 Time complexity

The scenarios are characterised by the number of service intentions, time constraints and the network structure. It is quite difficult to characterise a DAG, especially with relation to our task. We can consider the number of vertices and edges in the railway graph, average branching factor and how congested the network is. For every scenario, we consider the total railway graph $G = (E, V)$ consisting of merged network graphs for all service intentions ($G = \cup_{si \in SI} G_{si}$).

Number of vertices and edges tells us something about how large a network is, and branching factor will tell us how connected the network is. The average branching factor is defined as the average number of child nodes associated with every vertex.

For a DAG, the average branching factor is given by $b = \frac{|E|}{|V|}$[1].

By congestion, we are referring to bottlenecks, i.e. nodes that have a high number of incoming and outgoing edges. Congested networks can lead to an high number of possible routes a train can take, affecting the computation times; however there is no simple metric of congestion. Consider the networks illustrated on figure 5.1. Let both networks represent railways for a train to start at nodes A or B, and finish at nodes J or H. In network 1, a train can take 6 possible paths, while network 2 allows for 8 admissible paths. Both networks have the same number of nodes and edges, same average branching factor yet one is clearly larger in terms of computation. For this purpose, we define a metric *branching variance* $\sigma = \frac{\sum_{n \in V}(out(n)-b)^2}{|V|}$. A higher variance means that there is a higher degree of congestion, i.e. some nodes will have a higher amount of outgoing edges.

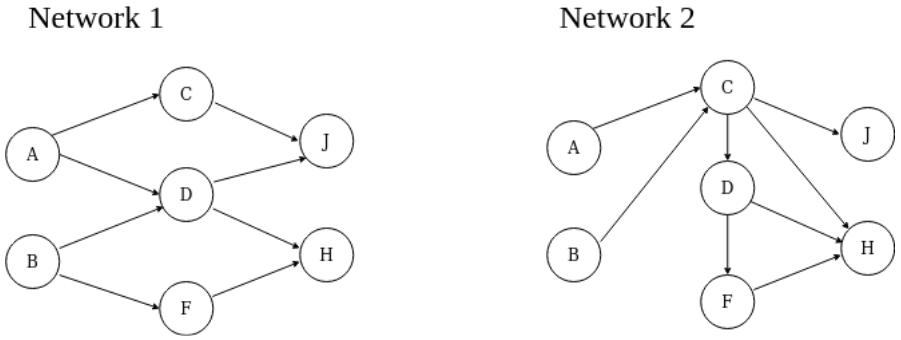


Figure 5.1: Example of a Directed Acyclic Graph

Having a metric for every DAG given as a tuple $(|SI|, |V|, |E|, b, \sigma)$, we can run the test-cases and record the performance of the different sections. The metric for all scenarios is summarised on table 5.1, together with the number of boolean variables, and railway graphs for the first two scenarios are plotted on figure 5.2.

The hardware used in this experiment is a Dell XPS 2018 Laptop with an Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz CPU and 8GB RAM. Even though the CPU has 8 cores, it should be noted that the parser runs sequentially, and the only features that utilises multi-core processing is the MOSEK solver.

During the experiments, the parser failed to fully convert test cases 6,7,8 and 9 due to insufficient memory. The parser failed during matrix assembly, which means that if the parser is to be used for larger problems, the user will need to have high

[1]The proof is simple. Every outgoing edge will have a root node, so summing outgoing edges over all nodes is equivalent to counting the number of edges, thus $b = \frac{\sum_{n \in V} out(n)}{|V|} = \frac{|E|}{|V|}$

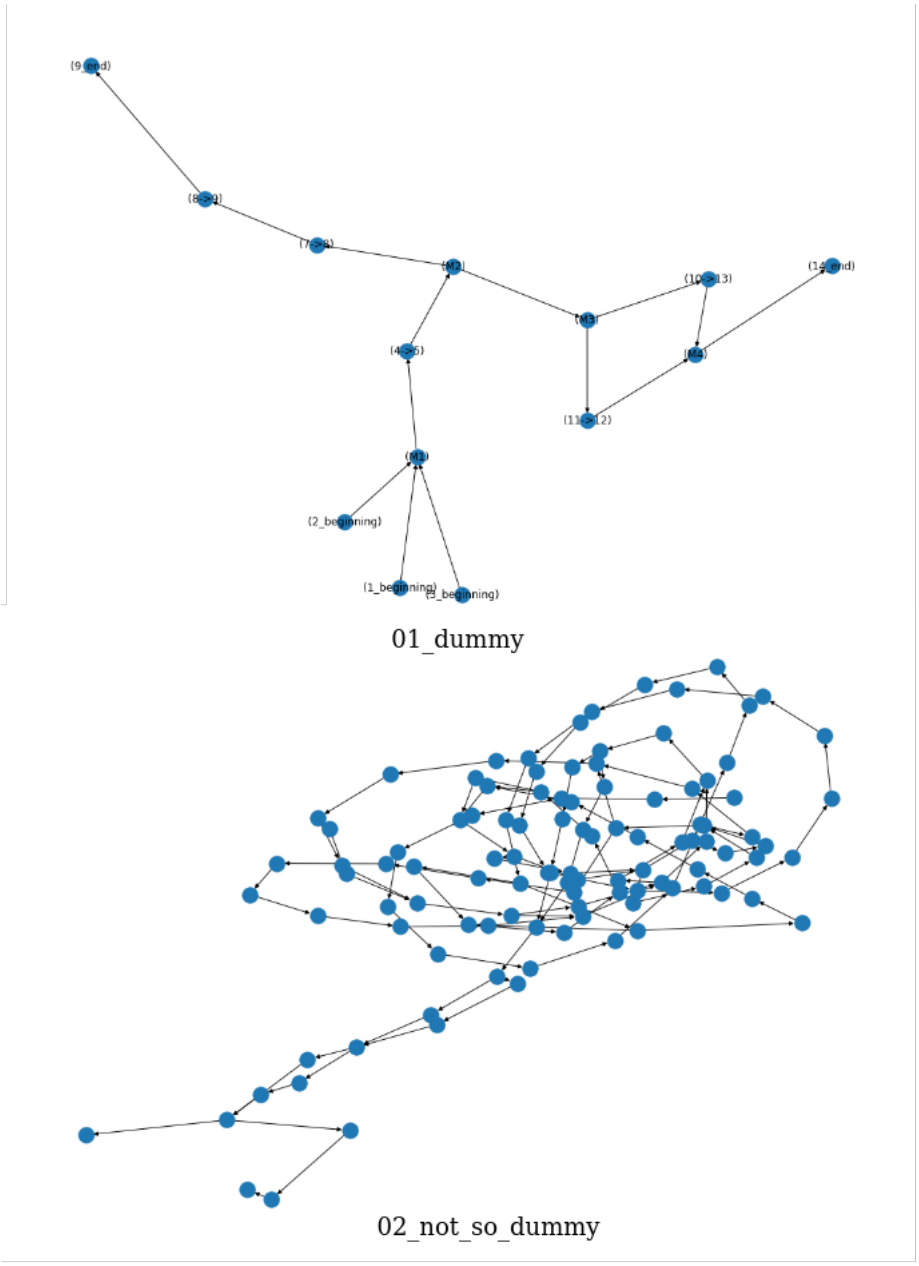


Figure 5.2: Graph plots

amounts of RAM, which is not readily available for commercial computers, or write the matrix to disc, which will significantly impair the performance. Nontheless, cases 0-5 are real life scenarios that the parser was able to process on a Laptop. Table 5.2 shows the computation times for various sections of the program. It is immediately clear that matrix assembly takes up the majority of computing resources, and is the main computational challenge of this parser.

Scenario	$\|SI\|$	$\|V\|$	$\|E\|$	b	σ	constraints	boolean variables
sample_scenario	2	14	14	1.00	0.29	2047	60
01_dummy	4	106	118	1.11	0.13	7654	605
02_a_little_less_dummy	58	313	378	1.20	0.29	88753	17686
03_FWA_0.125	143	326	399	1.22	0.32	300148	67255
04_V1.02_FWA_without_obstruction	148	368	466	1.26	0.40	889003	67943
05_V1.02_FWA_with_obstruction	149	370	469	1.26	0.40	889033	67948
06_V1.20_FWA	365	421	559	1.32	0.80	>16375581	NaN
07_V1.22_FWA	467	421	559	1.32	0.88	NaN	NaN
08_V1.30_FWA	133	591	780	1.32	3.28	NaN	NaN
09_ZUE-ZG-CH_0600-1200	287	908	1267	1.39	2.65	NaN	NaN

Table 5.1: Railway metric for the test-cases

Scenarios	Loading json	Graph construction	Path calculation	Data structure setup	Matrix setup	Total
sample_scenario	$3.2 \times 10^{-4}s$	$1.4 \times 10^{-3}s$	$1.0 \times 10^{-3}s$	$1.1 \times 10^{-3}s$	$0.018s$	$0.022s$
01_dummy	$6.3 \times 10^{-3}s$	$0.04s$	$3.9 \times 10^{-3}s$	$6.4 \times 10^{-3}s$	$0.03s$	$0.09s$
02_a_little_less_dummy	$0.05s$	$0.31s$	$0.02s$	$0.25s$	$1.15s$	$1.78s$
03_FWA_0.125	$0.07s$	$0.63s$	$0.04s$	$0.90s$	$6.43s$	$8.07s$
04_V1.02_FWA_without_obstruction	$0.15s$	$0.79s$	$0.05s$	$1.03s$	$10.27s$	$12.31s$
05_V1.02_FWA_with_obstruction	$0.34s$	$0.89s$	$0.053s$	$1.13s$	$10.70s$	$13.11s$
06_V1.20_FWA	$0.95s$	$4.78s$	$5.36s$	$19.05s$	NaN	$> 30.13s$

Table 5.2: Parser computation times

In the analysis of the data, no major correlations between computing times and metrics except for $|SI|$ and number of constraints. The time seems to scale linearly with the number of constraints, and quadratically with respect to service intentions (figure 5.3).

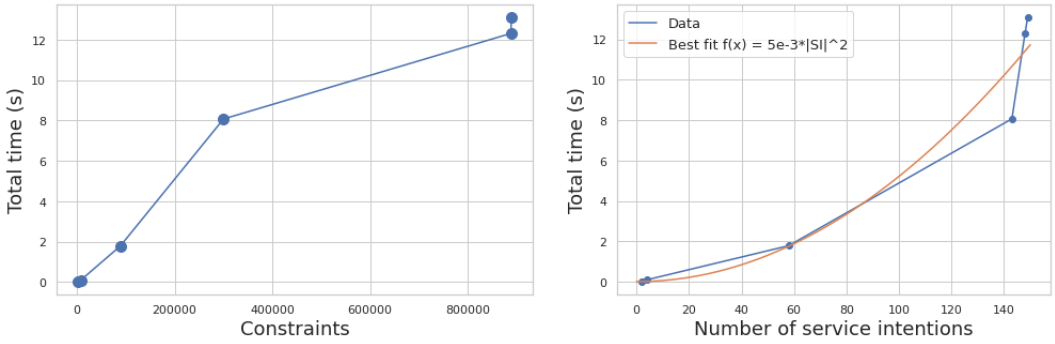


Figure 5.3: Number of constraints and service intentions vs time

The constrains are loaded sequentially, so the time is expected to scale proportionally. There is a very important relationship between SI and number of constraints; in loading the independent constraints, service intentions linearly scale the total number of root sections that the parser iterates through. However, for the coupling,

we create all combinations between two service intentions $\binom{|SI|}{2}$, which simplifies to $\frac{1}{2}(|SI|^2 - |SI|)$, which explains why there are such a large number of coupling constraints compared to the independent problems, and thus explains the time complexity observed.

To conclude, we can consider all metrics unrelated to matrix assembly insignificant. Due to the nature of the test cases, there is no direct evidence of time scaling with the size of the DAG, however, theoretically we know that this cannot be the case, seeing that increased number of edges leads to more routes, and the time complexity with respect to network size should be no better than $\mathcal{O}(|E|)$. Nevertheless, evidence from the data and analysis agree that the complexity with respect to service intentions is of order $\mathcal{O}(|SI|^2)$.

5.1.2 Memory Usage

In section 3.3.3.7 we decided to use a list of sparse rows for the constraint matrices, as they compress the data by only storing non-zero elements per row of constraints, thus offering faster matrix multiplication than a coordinate matrix or a sparse column matrix would. The memory usage for the first 6 test-cases is displayed on table 5.3.

Table 5.3: Constraint memory usage

Scenario	A	b
sample_scenario	103 kB	17 kB
01_dummy	393 kB	68 kB
02_a_little_less_dummy	13.82 MB	1.87 MB
03_FWA_0.125	51.69 MB	6.77 MB
04_V1.02_FWA_without_obstruction	129.61 MB	16.71 MB
05_V1.02_FWA_with_obstruction	129.61 MB	16.71 MB

Figure 5.4 shows a plot of memory vs constraints, which confirms the theory discussed in section 3.3.3.7, which is that matrices A will scale linearly with respect to number of constraints when using Scipy's sparse matrix class. b also scales linearly, as it is an array with a single column, thus every entery corresponds to a constant addition of memory.

As for the magnitude of the size, we see that the parser should be able to handle railways with around 150 trains and 150 stations on a commercial computer.

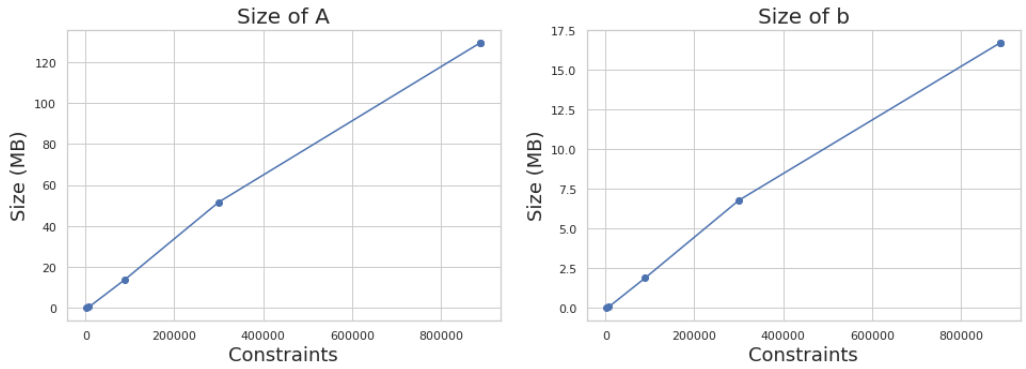


Figure 5.4: Number of constraints vs matrix size

5.2 Naive MILP solver

As seen in the previous section, the parser was able to handle 6 out of 10 scenarios. This means we successfully set up a MILP problem, and can attempt to solve it using MOSEK. We succeeded in the first four test-cases; after *03_FWA_0.125*, Python interpreter killed the processes. The results are shown in table 5.4.

Table 5.4: Solutions using Naive MILP solver

Scenario	Solution time	Objective value
sample_scenario	0.05s	0.0
01_dummy	0.19s	0.1
02_a_little_less_dummy	21.24s	0.1
03_FWA_0.125	464.09s	0.1

Scenario *03_FWA_0.125* is already quite a large problem, with 143 trains and 16 stations, MOSEK managed to find an optimal solution. As discussed in section 3.2.4, the number of integer variables scales quadratically with respect to $|SI|$ and exponentially with respect to the 'depth' of the railway network. In section 3.2.4 we also showed that the Branch-and-Bound algorithm in the worst case scales exponentially with the number of variables, which explains the intractability of the larger examples. Not much data is available (figure 5.5), so it is not possible to confirm the expected trend.

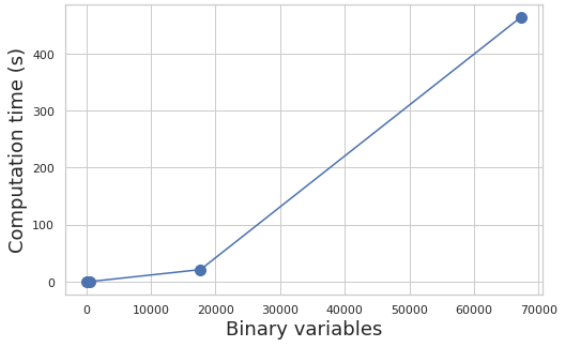


Figure 5.5: Number of binary variables vs computation time

```
si: 111, edge: (1_beginning),(M1), tin: 08:25:03, tout: 08:25:56
si: 111, edge: (M1),(4->5), tin: 08:25:56, tout: 08:26:28
si: 111, edge: (4->5),(M2), tin: 08:26:28, tout: 08:30:00
si: 111, edge: (M2),(7->8), tin: 08:30:00, tout: 08:30:32
si: 111, edge: (7->8),(8->9), tin: 08:30:32, tout: 08:31:04
si: 111, edge: (8->9),(9_end), tin: 08:31:04, tout: 08:50:00
si: 113, edge: (1_beginning),(M1), tin: 07:50:00, tout: 07:50:53
si: 113, edge: (M1),(4->5), tin: 07:50:53, tout: 07:51:25
si: 113, edge: (4->5),(M2), tin: 07:51:25, tout: 07:51:57
si: 113, edge: (M2),(M3), tin: 07:51:57, tout: 07:52:29
si: 113, edge: (M3),(11->12), tin: 07:52:29, tout: 07:53:01
si: 113, edge: (11->12),(M4), tin: 07:53:01, tout: 08:15:28
si: 113, edge: (M4),(14_end), tin: 08:15:28, tout: 08:16:00
```

(a) sample_scenario

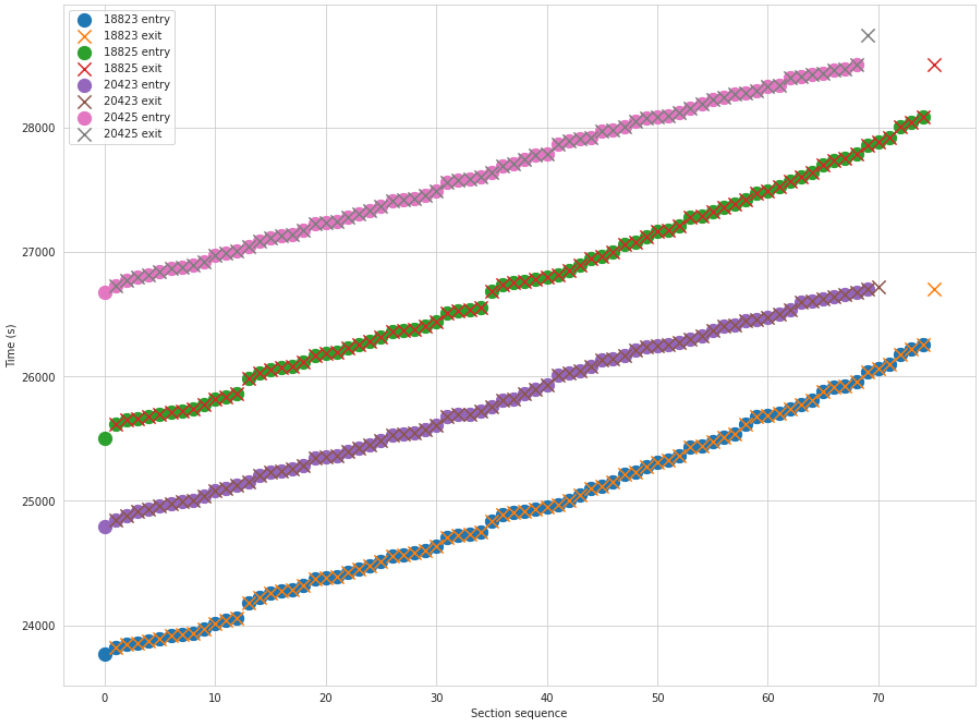
```
si: 18825, edge: (280->285),(285->287), tin: 07:44:16, tout: 07:44:45
si: 18825, edge: (285->287),(287->290), tin: 07:44:45, tout: 07:45:19
si: 18825, edge: (287->290),(290->295), tin: 07:45:19, tout: 07:46:42
si: 18825, edge: (290->295),(295->300), tin: 07:46:42, tout: 07:47:21
si: 18825, edge: (295->300),(300->305), tin: 07:47:21, tout: 07:48:00
si: 18825, edge: (300->305),(305_end), tin: 07:48:00, tout: 07:55:00
si: 20423, edge: (1_beginning),(1->5), tin: 06:53:11, tout: 06:54:05
si: 20423, edge: (1->5),(5->10), tin: 06:54:05, tout: 06:54:45
si: 20423, edge: (5->10),(10->15), tin: 06:54:45, tout: 06:55:17
si: 20423, edge: (10->15),(15->20), tin: 06:55:17, tout: 06:55:34
si: 20423, edge: (15->20),(20->25), tin: 06:55:34, tout: 06:56:00
si: 20423, edge: (20->25),(25->30), tin: 06:56:00, tout: 06:56:20
si: 20423, edge: (25->30),(30->35), tin: 06:56:20, tout: 06:56:31
```

(b) 01_dummy (slice)

Figure 5.6: Generated timetables

The solutions were verified, and timetables were generated as shown on figure 5.6. To visualise the timetables, the sequence of route sections were plotted on a space-time diagram in figure 5.7 for the service intentions involved in the two problems considered. These plots show entry (circles) and exit (crosses) times into each route section, although the chart does not help in determining whether any collisions are scheduled or not, as the chart does not show the actual path the service intention take.

(a) sample_scenario



(b) 01_dummy

Figure 5.7: Visualisation of train timetables

Solving problems without coupling constraints were an easy task for MOSEK. Problem *03_FWA_0.125* with 143 trains was solved in 14.53s (vs 464s in the coupled

case), so if one can manually verify that two trains will not be intersecting, by for instance considering constraints on earliest and latest entry times and exit times, the coupling between those trains can be discarded and solutions can be sped up substantially.

5.3 ADMM solver

The ADMM solver was tested on the two first examples only, as the Python interpreter killed the process during computation of sub-subproblems. This is a design flaw in the implementation that should be further investigated and addressed in the future. The reason was identified to yet again be insufficient memory, where the process used up 100% of both RAM and SWAP. The two test-cases that did work are very promising, but rather inconclusive.

In both instances, optimal solution was found during the first iteration, so the solver terminated as the ∞-norm of the constraint violations was zero. What this means is that every service intention was able to adapt their route to the routes of the service intentions solved in the preceding problems, without violating any constraints nor sacrificing any their primal objective function. This is illustrated using an example on figure 5.8: consider two trains in a network. The first train starts at A and has to end in C, whilst the second train starts at B, and can either end in C or D, such that both paths lead to 0 penalty in its local objective. If solved as uncoupled problems, both trains are routed to C, which can happen as the optimal solution is non-unique, as discussed in section 3.2.2, thus pseudo-randomly converge at **a** optimal solution. If problem for train 1 is solved first in ADMM, then solution to problem 2 will switch to route that ends in C to avoid penalisation of constraints. Conversely, if train 2 is solved for first, when solving for train 1, it will choose to violate the constraints to avoid penalty delay, and in the next iteration train C will adjust its course to avoid the penalty violation. In our case, all the trains were able to adapt to the predecessor without incurring any penalties.

The implementation of the solver procedure was rather inefficient, as the majority of time was spent to assemble the objective function (inner loop of algorithm 6). The current implementation pre-computes all matrices in advance, but redefines the MOSEK environment in every iteration, which MOSEK needs time to compile. However, the actual time finding optima beats the naive solver for *01_dummy*, which

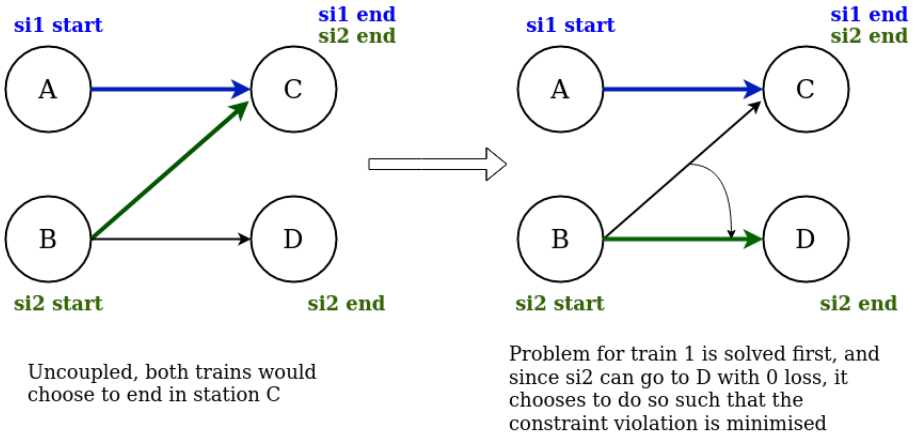


Figure 5.8: Example of rerouting during serial solution of problems in ADMM

Table 5.5: ADMM computation times (averaged over 5 test runs)

Scenario	Time spent searching for optima	Total time compiling objective functions	Initialisation of sub-subproblems	Total time
sample_scenario	0.08s	1.92s	0.033s	2.09s
01_dummy	0.10s	3.18	0.18s	4.12s

for ADMM spends a total of $0.10s$ searching for optima, vs the naive approach which averages $0.18s$. All the times are summarised on table 5.5.

Recall that a definition of a sub-subproblem is every coupled pair, thus the ADMM algorithm scales quadratically with the number of service intentions. However, the time taken to solve every subproblem will fall exponentially due to complexity of Branch-and-Bound. Thus, if the issue with regards to recompilation of subproblems can be fixed, by for instances defining MOSEK environments for every subproblem, declaring Lagrange multipliers and non-active solutions as MOSEK parameters, we can avoid the need to recompile the problem every iteration.

Chapter 6

Conclusion

6.1 Conclusions

In this thesis we developed a parser which converts Train Timetabling Problems into Mixed Integer Linear Programs of the form: $\max\{c^T x | Ax \leq b, Ix \in 0, 1\}$. The parser was then applied to test-cases provided by SBB, where we tried to obtain globally optimal solutions using the MOSEK solver. Furthermore, we formulated and developed an implementation of the Alternating Direction Method of Multipliers for the TTP as an attempt to speed up the computation.

The parser is implemented in Python, and reads input from json files provided by SBB, returning constraint matrices and objective function in standard MILP form. The parser was developed with consideration of time and space complexity, and performed comparatively well on a commercial laptop. It was able to process 6 out of 10 test cases, out of which 5 can be regarded real world problems. The process was terminated by the Python interpreter on the latter cases due to violation system limits imposed by the OS. That being said, a industry grade computer should be able to parse some of the failed test cases, but we had no access to a more powerful machine during development of this thesis. Initially, we started developing the parser to be used with a library called Cvxpy, which is a preprocessing module for Python that can convert problems from symbolic form into standard MILP form and directly pass it to a solver like MOSEK. During the development process, it turned out that loading constraints in symbolic form slowed down the process considerably, and

exceeded our machine's memory capacity after the first two test cases. Furthermore, when implementing ADMM, Cvxpy showed to suffer from numerical instabilities when introducing a large number of slack variables. Therefore, we decided to rewrite the parser to generate sparse matrices and interface with MOSEK directly instead, which solved our memory issues and eliminated numerical instabilities.

Mixed Integer Linear Programming was declared NP-hard a long time ago, so people research quickly shifted towards heuristics and suboptimal solution methods. A similar thing happened in the field of Machine Learning; the Multilayer Percepetron was invented in the 1960s, but abandoned as it was declared computationally intractable, which lead to the an 'AI winter'. However, in late 1990s the concept was picked up again, as hardware had caught up to the theory, which has resulted in massive advancements of technology. Similarly, we wanted to revisit MILP as a solution to the TTP, and see to what extent it can actually be used. We found that it can solve 4/10 scenarios proposed by SBB. That may not sound impressive from a standpoint of solving the challenge, but considering that the largest example we solved had 143 trains spanning over 325 stations, it is safe to say that MILP should not be discarded in the process of developing railway schedules. It should be noted that Garrisi and Cervelló-Pastor [11] who focused on implementation of a genetic algorithm to solve the SBB challange, claim that the second instance *01_dummy* is intractable to solve using a naive MILP solver, which we showed is by far not the case. Although slower, we managed to find the optimal values of 0.0 for all the problems to which we applied the algorithm, while they achieved losses of 0.0, 0.0, 106.32, 1530.42 for the first four test cases. These values predominantly represent delays in minutes, which can be avoided using our programme at the expense of computation times, which is quite a significant result.

The main motivation to undertake this process was advancements in serial optimisation algorithms, which take advantage of structural properties of a Mathematical Programs. In the TTP it is the fact that every train has its own problem that is to be solved in consensus with other trains, forming a partially block-diagonal constraint matrix and fully separable objective functions. ADMM seemed to be a suitable algorithm as it can relax the coupling constraints by embedding them into the cost function of the decomposed problems, which found the solution faster than the naive approach for the largest test case we were able to apply it to. Unfortunately, due to inefficient implementation, it exceeded the OS system limits for the 8 test cases during the setup phase, and could not be tested further.

6.2 Future Work

The parser performs quite well as is, and for all practical purposes is fast enough to convert TTP to a MILP, and we expect it do so if memory requirements can be successfully addressed. If not, the first line of attack would be to parallelise the constraint loading stage, as constraints are all independent and can be loaded concurrently without any exchange of information. In order to actually determine whether it can be used in industry, it would be instructive to run both the parser and the naive solver on an industry grade machine, as we are confident that would parse and solve more scenarios than we were able to during this thesis.

It was very strange to see the ADMM algorithm to fail processing all the cases that the parser was able to handle, as the failure occurred during the initialisation of sub-subproblems, where we specifically designed the classes such that no double information is stored, so that is clearly an issue that needs to be investigated. As mentioned in the experiment phase, MOSEK recompiles every subproblem, which involves heavy computations related to the objective function due to exponential increase in coupling, so the next steps would be to extend the sub-subproblem class to encompass a MOSEK environment, which will store the compiled sub-subproblem, and simply set the new value of Lagrange multipliers and state vectors for every iteration. As a further step towards reducing solver times, it is worthwhile to investigate the possibility of integer constraint relaxation of the coupling variables for every iteration. In section 9.1 of the paper *Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers* [4], the authors suggest that for non-convex feasible sets $\mathcal{S}$, one can relax these constraints during the solution of subproblems, and then project them onto $\mathcal{S}$ during the update. In our case that would be relaxing the binary constraints on β by replacing them with $0 \leq \beta \leq 1$, and rounding β at the end of every iteration. During experimentation, it was discovered that rounding the relaxed variables had a high degree of correlation with the unrelaxed solution, which means there may potentially be a trade-off between time spent solving per subproblem and the number of iterations until convergence.

Lastly, as any industrial grade software, all the code should be re-written in a lower level language to increase robustness and speed.

Bibliography

[1] Diego Arenas et al. "Solving the Train Timetabling Problem, a mathematical model and a genetic algorithm solution approach". In: *6th International Conference on Railway Operations Modelling and Analysis (RailTokyo2015)*. Tokyo, Japan, Mar. 2015. URL: https://hal.archives-ouvertes.fr/hal-01338609.

[2] L. Bai et al. "A mixed-integer linear program for routing and scheduling trains through a railway station". In: *ICORES 2014 - Proceedings of the 3rd International Conference on Operations Research and Enterprise Systems* (Jan. 2014), pp. 445–452.

[3] Stephen Boyd and Almir Mutapcic. *Stochastic Subgradient Methods*. 2007.

[4] Stephen Boyd et al. "Distributed Optimization and Statistical Learning via the Alternating Direction Method of Multipliers". In: *Foundations and Trends in Machine Learning* 3 (Jan. 2011), pp. 1–122. DOI: 10.1561/2200000016.

[5] Mircea Cirnu and Irina Badralexi. "ON NEWTON-RAPHSON METHOD". In: *Romanian Economic Business Review* 5 (Jan. 1995), pp. 91–94.

[6] Michael B. Cohen, Yin Tat Lee, and Zhao Song. *Solving Linear Programs in the Current Matrix Multiplication Time*. 2020. arXiv: 1810.07896 [cs.DS].

[7] Francesco Corman et al. "A tabu search algorithm for rerouting trains during rail operations". In: *Transportation Research Part B: Methodological* 44.1 (2010), pp. 175–192. ISSN: 0191-2615. DOI: https://doi.org/10.1016/

j.trb.2009.05.004. URL: https://www.sciencedirect.com/science/article/pii/S0191261509000708.

[8] Bernardo Cuenca Grau. "Artificial Intelligence". In: *University of Oxford, Ariticial Intelligence Lecture Notes* (2021). URL: https://www.cs.ox.ac.uk/teaching/materials20-21/ai/AI-lecture-1.pdf.

[9] Andrea D'Ariano et al. "Reordering and Local Rerouting Strategies to Manage Train Traffic in Real Time". In: *Transportation Science* 42 (Nov. 2008), pp. 405–419. DOI: 10.1287/trsc.1080.0247.

[10] Andrea D'Ariano, Dario Pacciarelli, and Marco Pranzo. "A branch and bound algorithm for scheduling trains in a railway network". In: *European Journal of Operational Research* 183.2 (2007), pp. 643–657. ISSN: 0377-2217. DOI: https://doi.org/10.1016/j.ejor.2006.10.034. URL: https://www.sciencedirect.com/science/article/pii/S0377221706010678.

[11] Gianmarco Garrisi and Cristina Cervelló-Pastor. "Train-Scheduling Optimization Model for Railway Networks with Multiplatform Stations". In: *Sustainability* 12.1 (2020). ISSN: 2071-1050. DOI: 10.3390/su12010257. URL: https://www.mdpi.com/2071-1050/12/1/257.

[12] Joachim Giesen and Sören Laue. *Distributed Convex Optimization with Many Convex Constraints*. 2018. arXiv: 1610.02967 [math.OC].

[13] Paul Goulart. "C20 - Convex Optimization". In: *University of Oxford* (2020). URL: https://users.ox.ac.uk/~engs1373/local/C20/C20_cvx_opt_lectures.pdf.

[14] Ravindran Kannan and Clyde L. Monma. "On the Computational Complexity of Integer Programming Problems". In: *Optimization and Operations Research*. Ed. by Rudolf Henn, Bernhard Korte, and Werner Oettli. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978, pp. 161–172. ISBN: 978-3-642-95322-4.

[15] Kieran Molloy. *Artificial Intelligence in Train Scheduling Problems*. Feb. 2020. DOI: 10.13140/RG.2.2.36560.12805.

[16] David R. Morrison et al. "Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning". In: *Discrete Optimization* 19 (2016), pp. 79–102. ISSN: 1572-5286. DOI: https://doi.org/10.1016/j.disopt.2016.01.005. URL: https://www.sciencedirect.com/science/article/pii/S1572528616000062.

[17] MOSEK. *3 Conic quadratic optimization¶*. June 2021. URL: https://docs.mosek.com/modeling-cookbook/cqo.html.

[18] MOSEK. *MOSEK optimization software*. URL: https://www.mosek.com/products/mosek/.

[19] Arkadi Nemirovski and Michael Todd. "Interior-point methods for optimization". In: *Acta Numerica* 17 (May 2008), pp. 191–234. DOI: 10.1017/S0962492906370018.

[20] Paola Pellegrini et al. "Real-time train routing and scheduling through mixed integer linear programming: Heuristic approach". In: *Proceedings of 2013 International Conference on Industrial Engineering and Systems Management (IESM)*. 2013, pp. 1–5.

[21] SBB. *Train Schedule Optimisation Challenge*. URL: https://www.crowdai.org/challenges/train-schedule-optimisation-challenge.

[22] P. Tormos et al. "A Genetic Algorithm for Railway Scheduling Problems". In: *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*. Ed. by Fatos Xhafa and Ajith Abraham. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 255–276. ISBN: 978-3-540-78985-7. DOI: 10.1007/978-3-540-78985-7_10. URL: https://doi.org/10.1007/978-3-540-78985-7_10.

[23] Sven Wiese. *A gentle introduction to Continous and Mixed-Integer Conic Programming*. Nov. 2020. URL: https://docs.mosek.com/slides/2020/bologna/talk.pdf.