CHALMERS
UNIVERSITY OF TECHNOLOGY

Optimization of Train Schedules

Applied to Automated Mining Transports

Master's Thesis in Applied Mathematics

John Christoffer Dahlén
Anton Mårtensson

Department of Mathematical Sciences

Chalmers University of Technology
Gothenburg, Sweden, 2020

www.chalmers.se

Master's Thesis

Optimization of Train Schedules

Applied to Automated Mining Transports

John Christoffer Dahlén

Anton Mårtensson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mathematical Sciences
Chalmers University of Technology
Gothenburg, Sweden 2020

Optimization of Train Schedules
Applied to Automated Mining Transports

Authors: John Christoffer Dahlén, Anton Mårtensson

Contact:
christoffer.dahlen@gmail.com
anton.v.martensson@gmail.com

© John Christoffer Dahlén, Anton Mårtensson, 2020.

Supervisor: Abel Salas, Bombardier Transportation AB, Sweden
Examiner: Ann-Brith Strömberg, Mathematical Sciences

Department of Mathematical Sciences
MVEX03
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone: +46 (0)31-772 10 00

Bombardier Transportation AB, Sweden
Rail Control Systems, Gothenburg
Telephone: +46 (0)10-852 00 00

Cover image by Clay Gilliland, Arizona, USA, 2015.

Typeset in LaTeX
Printed by Chalmers Reproservice
Gothenburg, Sweden 2020

Abstract

Logistics and transportation are extremely important to modern society. In this thesis we investigate how to optimize the pathing and scheduling of trains, particularly targeting automated mining transports, what we call Path Conflict Resolution (PCR). A restriction imposed is that solutions to this problem must be deadlock-free, which implies that trains may not be treated separately but must be scheduled together. We approach this problem using a combination of Alternative Graphs, Mixed Integer Linear Programming (MILP), and Variable Neighbourhood Search (VNS), building upon previous work on the scheduling of passenger trains. Our model extends the previous work by including additional complications in the scheduling, different objective functions, modelling of train lengths and moving blocks, and improvements in the MILP problem generation. We successfully schedule trains on a large realistic rail network within acceptable computation times. The strongest improvement is the development of couplings in the alternative graph, which in our experiments leads to a tenfold reduction in the number of binary variables.

Keywords: Trains, Routing, Scheduling, Mathematical Optimization,
Stochastic Optimization, Mixed Integer Linear Programming,
Variable Neighbourhood Search, Alternative Graphs

Contents

Preamble

In this thesis there are many definitions in use. To help the reader a glossary of operators and symbols can be found in Appendix D.

Chapter 1

Introduction

The matter of logistics has been a subtle but important driver of technology and innovation throughout human history; the question of how to best transport goods becomes increasingly important as society scales up. With the advent of general purpose computing and the information age, much of modern logistics has been turned into (mathematical) optimization problems. Delivery routes, train schedules, work allocation, and more are now problems for computers to solve. However, most of these problems are only partially or approximately solved and there remains much ongoing research on how to construct better models, create faster and more efficient solvers, and solve more problems.

Bombardier is a multinational manufacturer of many types of vehicles and related systems. Its Rail Control Systems department—partly based in Gothenburg, where this project took place—works to create and maintain control systems for many different automated railways all over the world. More specifically, one of the main applications is automated mining transports: transportation of various ores from loading to unloading stations, from mines to harbours. A central part of these systems is the need to compute paths for the trains, from each train's current position to a desired destination (potentially with a requirement to pass through some other points on the way).

This problem is made complex by the presence of many different trains on the same rail network, which will create conflicts, deadlocks, and signal contention. It is also made complex by various additional criteria that may be placed on paths, schedules, switches, and more. Poor solutions to this problem are expensive since trains needing to brake or stand still can involve significant loss of time and money, especially when the trains in question can be a few kilometers long carrying many thousands of tons of ore.

Chapter 2

Problem Description

The problem we attempt to solve, which we call the *Path Conflict Resolution* (PCR) problem, is to find the best paths for a set of trains from some initial locations to their respective destinations such that no deadlocks occur. As the application in this case is logistics and ore transport, the best path should be the one that maximizes profit.

The problem differs from other train or scheduling problems as: a) The trains in our case do not follow regular or predetermined schedules, but are dispatched when ready and must be scheduled in real time; b) The trains have very flexible paths between origin and destination as they generally lack intermediate stops; c) The trains are so long that they may block multiple rail segments, and cannot be simplified to points. Additionally, in normal train traffic a system called *fixed block* is implemented where only a single train can be in a block at each point in time (a block generally being a stretch of rail between two signals). However, in our problem a system called *moving block* is desired. The goal of moving block is to reduce the time trains are waiting by removing the fixed nature of the block system and instead use a safety distance between the different trains, such that each train consititues essentially its own block (hence the name).

What complicates the problem more than anything else is that we do not permit deadlocks. This means that when solving the PCR problem it is almost never possible to consider any train or path separately, meaning there is no obvious way to divide the problem into smaller and simpler parts. Furthermore, to check if a set of paths are deadlock free we also need to evaluate the behaviour of the trains on their respective paths.

2.1 Literature Review

Scheduling in general is a well studied part of mathematical optimization. For train scheduling in particular, published approaches have been found but many deal only with a single track as opposed to a larger rail network [1], use models that are computationally expensive [1], or approach problems that differ largely from our own [2].

The application of job-shop scheduling problems (JSP), as modelled by alternative graphs (see Section 3.2 later) and applied to trains, has been collectively studied by A. Mascis, M. Samà, A. D'Ariano, F. Corman, D. Pacciarelli, and M. Pranzo in various publications [3]–[7].

In [3], Mascis and Pacciarelli first study the JSP and formulate the alternative graph model, establishing key properties of the model.

In [4], D'Ariano, Pacciarelli and Pranzo show how the alternative graph formulation, together with a branch and bound algorithm, can be successfully used to reschedule trains in a small "bottleneck area of the Dutch rail network".

In [5], Corman, D'Ariano, Pacciarelli and Pranzo describe a tabu search algorithm for local rerouting of trains in the ROMA ("Railway traffic Optimization by Means of Alternative graphs") implementation.

In [6], Samà, D'Ariano, Pacciarelli and Corman study fast lower and upper bounds to the scheduling and routing problem by way of constraint relaxation and heuristic solutions.

In [7], Samà, D'Ariano, Corman and Pacciarelli describe a larger system and implementation (named AGLIBRARY) using the established alternative graph formulation together with variable neighbourhood search (VNS) to reschedule and reroute trains in response to perturbations.

2.2 Restrictions and Scope

In order to be able to deal with this problem we also have to include some restrictions, such that it should be possible to model. Here we assume the following restrictions and limitations.

- In reality there is (sometimes large) uncertainty regarding, for example, the exact position, velocity, and length of a train. Our model and implementation will assume that the real world state of a train and switches are as reported, taking into account pre-established safety margins but not potential unknown errors.
- The complete real world system[†] separates scheduling and routing, where the latter refers to sections of track reserved in front of a train during operation. We will only deal with the former, i.e., the planning of train movement and not detailed mechanics related to that movement.
- Physical train control (e.g. engine control and breaking) is managed by on-board systems, either automatic or manual; this project will only deal with scheduling and ignore acceleration and breaking curves.
- For real applications, it is likely desired to minimize the total cost in terms of money rather than time, but we will only study the latter, i.e., time. Other monetary factors are assumed to be expressed as time, delays, or adjustments thereof.
- In real world usage, calculations would be performed while the trains are moving, necessitating (relatively) quick calculations as well as accommodations for the change of system state. For this project we will disregard this complication.
- The schedules require some error margins. These will change for different velocities, however a dynamic description of these margins is out of scope.

[†]In particular, the system currently in use at Bombardier.

Chapter 3

Theory

To deal with the PCR problem we need three mathematical tools: *Alternative Graphs*, which are used for scheduling to deal with the usage of resources; *Mixed Integer Linear Programming* (MILP) which is used to find solutions to the Alternative Graphs and in turn the scheduling subproblem; and the family of optimization algorithms known as *Variable Neighbourhood Search* (VNS) which we will use for the path selection master problem.

In this chapter we will provide a brief introduction to each tool, providing a basis for how the PCR problem is solved.

3.1 Mixed Integer Linear Programming (MILP)

Mixed Integer Linear Programming (MILP) is part of the more general field of Mathematical Optimization (alternatively Mathematical Programming[†]). The goal in mathematical optimization is to find the best variable values given some constraints on the variables.

There are many types of mathematical optimization problems. We will, however, mainly investigate the MILP problems and the related *Linear Programming* (LP) problems. For a mathematical optimization problem to be a MILP problem it can only contain linear constraints and some variables take integer values and the rest being continuous.

A minimization MILP problem can be written in the form

$$\begin{aligned} \underset{\boldsymbol{x},\boldsymbol{t}}{\text{minimize}} \quad & z = \mathbf{c}^{\mathrm{T}}\boldsymbol{t} + \tilde{\mathbf{c}}^{\mathrm{T}}\boldsymbol{x} \\ \text{subject to} \quad & A\boldsymbol{t} + B\boldsymbol{x} \leq \mathbf{a}, \\ & C\boldsymbol{t} + D\boldsymbol{x} = \mathbf{b}, \\ & \boldsymbol{t} \geq \mathbf{0}^n, \\ & x_i \in U_i \subseteq \mathbb{Z}, \qquad i \in \{1, \ldots, m\} \end{aligned}$$

where we have o inequality constraints and p equality constraints. In this form we have that $A \in \mathbb{R}^{o\times n}$, $B \in \mathbb{R}^{o\times m}$, $C \in \mathbb{R}^{p\times n}$, $D \in \mathbb{R}^{p\times m}$, $\mathbf{a} \in \mathbb{R}^o$, $\mathbf{b} \in \mathbb{R}^p$, $\mathbf{c} \in \mathbb{R}^n$, $\tilde{\mathbf{c}} \in \mathbb{R}^m$. A common restriction of a MILP problem is the *Mixed Binary Linear Programming* (MBLP) where all $U_i = \{0, 1\}$, which is the form our problems will take.

[†]It is important not to confuse programming here with programming from computer science. Here it mainly refers to planning. This is due to the fact that the convention of calling the field mathematical programming was established before the widespread use of computers.

The related LP problem similarly only contains linear constraints, but all variables are also continuous. This means that an LP problem can be written in the form

$$\begin{aligned} \underset{\boldsymbol{t}}{\text{minimize}} \quad & z = \mathbf{c}^{\mathrm{T}}\boldsymbol{t} \\ \text{subject to} \quad & A\boldsymbol{t} \leq \mathbf{a}, \\ & C\boldsymbol{t} = \mathbf{b}, \\ & \boldsymbol{t} \geq \mathbf{0}^n, \end{aligned}$$

where $A \in \mathbb{R}^{o \times n}$, $C \in \mathbb{R}^{p \times n}$, $\mathbf{a} \in \mathbb{R}^o$, $\mathbf{b} \in \mathbb{R}^p$, $\mathbf{c} \in \mathbb{R}^n$.

The optimal solution to either problem is an objective value z^* and the associated variable values, such that there exist no other feasible variable values resulting in a lower objective value.

An LP problem is often easier to solve than a MILP problem. This is because in an LP problem it is possible to show that an optimal solution can always be found in an extreme point of the feasible set, i.e., of the set of all points that fulfill the constraints. This can then be used in, for example, the Simplex method, see [8], to solve the problem efficiently.

A MILP problem is harder to solve since the above no longer holds; what would otherwise have been an optimal solution might not adhere to the integrality constraints. However, due to its form there are multiple ways of utilizing the LP problem to solve the MILP problem. One method is to fix the values of all the integer variables which means that the remaining problem is an LP problem. This problem is then either infeasible or provides an upper bound to the MILP problem with an optimal solution $\bar{z}^\star$ such that $\bar{z}^\star \geq z^\star$.

Another method that is commonly used is continuous relaxation, which is done by replacing the integral $x_i \in U_i$ constraints with corresponding continuous limits. This results in that the relaxed MILP can be written as

$$\begin{aligned} \underset{\boldsymbol{x},\boldsymbol{t}}{\text{minimize}} \quad & \underline{z} = \mathbf{c}^{\mathrm{T}}\boldsymbol{t} + \tilde{\mathbf{c}}^{\mathrm{T}}\boldsymbol{x} \\ \text{subject to} \quad & A\boldsymbol{t} + B\boldsymbol{x} \leq \mathbf{a}, \\ & C\boldsymbol{t} + D\boldsymbol{x} = \mathbf{b}, \\ & \boldsymbol{t} \geq \mathbf{0}^n, \\ & x_i \in [\min(U_i), \max(U_i)]. \qquad i \in \{1, \ldots, m\} \end{aligned}$$

Here we know that the optimal MILP value $z^\star$ is greater than or equal to $\underline{z}^\star$, i.e. $z^\star \geq \underline{z}^\star$. This is due to the relaxation theorem, see [8, p.157], but is obvious since the relaxed problem contains all optimal solutions to the original problem.

3.2 Alternative Graphs

The problem of scheduling trains can be related to the so called *job-shop scheduling* problem (JSP). The JSP is one of the archetypical scheduling problems and refers to the optimal allocation of tasks (jobs are sequences of tasks) onto a limited amount of slots in which those tasks may be performed. In our case the jobs are the trains and the tasks are the traversal of rail sections (slots) in order to take each train from its origin to its destination.

For this reason we employ the method of "alternative graphs," developed by Mascis & Pacciarelli and based on preceding methods for JSP modeling [3]. An alternative graph is here defined as a triplet $\mathcal{G}_A = (\mathcal{N}, \mathcal{F}, \mathcal{A})$ of nodes $\mathcal{N}$, "fixed arcs" $\mathcal{F}$, and "alternative arcs" $\mathcal{A}$. Each node $n \in \mathcal{N}$ represents a resource (in our case a section of track), and associated with each node is a time variable t that is the time when the resource starts being utilized (the time a train *enters* the track). Arcs, on the other hand, represent constraints on these time variables. The purpose of alternative graphs is to construct a graph that models the constraints of the problem under study; assigning times that conform to these constraints means creating a valid schedule for the problem.

3.2.1 Fixed Arcs

The arcs in $\mathcal{G}_A$ are directed and weighted. An arc from node n_i to node n_j imposes the constraint that the time difference $t_j - t_i$ must be greater than or equal to that arc's weight. They occur in two forms, the simpler of which are fixed arcs which encode a static ordering constraint. These are elements $f \in \mathcal{F}$ and have weights u. An arc $f_{ij} = (n_i, n_j)$ thus encodes the constraint

$$t_j - t_i \geq u_{ij}. \tag{3.1}$$

Given a graph with only fixed arcs, an optimal solution (i.e., the minimal times t) can be easily calculated using *makespans*. Given two nodes, n_i and n_j, the makespan $\ell(n_i, n_j)$ is defined as the longest path from n_i to n_j, or more rigorously as the maximum total weight out of all paths from n_i to n_j. To assist us, we also introduce the virtual node n_0, which is a special node with a constant time $t_0 = 0$, with fixed arcs from n_0 to the initial node of each job. Using this node, we can define the times t_i for all other nodes n_i as

$$t_i = \ell(n_0, n_i). \tag{3.2}$$

Besides being a simple way to define an optimal solution, this is also useful in that makespans may be computed in polynomial time by, for example, performing a shortest path search on a corresponding graph with all weights negated. In our case a modified Dijkstra's Algorithm starting at n_0 produces all time values t_i in $O(|\mathcal{N}|^2)$ time.

3.2.2 Alternative Arcs

Physical resources may be represented by several different nodes in $\mathcal{G}_A$, in particular when different tasks require the same resources. There may also be cases where different physical resources are in conflict, such that they may not be utilized simultaneously. It is in situations like these that an ordering of the utilizations must be imposed using alternative arcs.

An alternative arc $a \in \mathcal{A}$ has the weight w. Each such arc exists in a pair $a, a^* \in \mathcal{A}$ where exactly one of the arcs must be chosen (hence the name "alternative"). The (a, a^*) pair thus represents a binary choice between the two constraints that those arcs correspond to:

$$a_{jk} = (n_j, n_k)\ ,\ a_{li} = (n_l, n_i) = a^*_{jk} \quad \Rightarrow \quad t_k - t_j \geq w_{j,k} \ \text{ or } \ t_i - t_l \geq w_{l,i} \ \text{ must hold.} \tag{3.3}$$

Note that the two arcs in a single pair do not have any implicit ordering and may be swapped arbitrarily. In particular, $(a^*)^* = a$.

In practice, the choice of arc represents the choice of which node utilization is scheduled first, and the situation is illustrated in Figure 3.1. We will later on refer to the node at which an alternative arc originates as the *release node* of the arc, and the node at which the arc terminates as the *blocked node*; the time of the blocked node is restricted, by the alternative arc, to be some minimum (possibly negative) time after the time associated with the release node.

To solve the alternative graph problem described by $\mathcal{G}_A$ exactly one arc of each pair is chosen (and used as if it was a fixed arc) whilst the other arc is ignored. It is this set of binary choices that turn the JSP from a polynomial problem into an NP-hard problem.

A property of this $\mathcal{G}_A$ formulation is that if any positive cycle occurs then it is infeasible. This can be easily be demonstrated by restating the constraints on the form $t_j \geq t_i + w_{ij}$ and $t_j \geq t_i + u_{ij}$, which means that by combining the constraints of the cycle we find that $t_j \geq t_j + d$, where d is the cycle weight which is defined as the sum of all constants w_{ij} and u_{ij} along the cycle, which is only possible if $d \leq 0$.

This means that if a positive cycle ($d > 0$) is generated when no alternative arcs have been selected then the graph is always infeasible. However if a positive cycle occurs when certain arcs have been selected, then that selection of arcs is infeasible.

3.2.3 Alternative Graphs as an MILP Problem

If we want to convert an alternative graph to a MILP problem then all the fixed arc constraints (3.1) are added to the model and the alternative arcs are added using the *Big M* method to handle the choice of arcs. To use the Big M method starting from the inequalities in (3.3) it is possible to rewrite them on the form

$$t_k - t_j \geq w_{j,k} \qquad \text{or} \qquad t_i - t_l \geq w_{l,i} \tag{3.4a}$$

$$\Rightarrow \qquad t_k - t_j \geq w_{j,k} - M x_{j,k,l,i} \qquad \text{and} \qquad t_i - t_l \geq w_{l,i} - M(1 - x_{j,k,l,i}), \tag{3.4b}$$

where we have added a binary variable $x_{j,k,l,i}$ and a large constant M. This means that when $x_{j,k,l,i} = 0$ the active constraint is $t_j - t_i \geq w_{ij}$ and when $x_{j,k,l,i} = 1$ the active constraint is $t_l - t_k \geq w_{kl}$. It is important for the value of M to be large, however, for computational reasons it is often better to set it as small as possible.

As there is not a fixed way to specify how to define an objective value for the Alternative graph it is just described as linearly dependent on variables represented by the alternative

graph. This means that it is possible to write the AG converted MILP problem as

$$\text{minimize} \quad z = \sum_{i=1}^{n} c_i t_i + \sum_{(j,k),(i,l)\in\mathcal{A}} \tilde{c}_{j,k,l,i} x_{j,k,l,i} + \tilde{c}_{l,i,j,k}(1 - x_{j,k,l,i})$$

subject to

$$t_j - t_i \geq u_{ij} \qquad \forall\, (n_i, n_j) \in \mathcal{F}, \qquad \text{(fixed arcs)}$$

$$\left.\begin{array}{l} t_k - t_j \geq w_{jk} - M x_{j,k,l,i} \\ t_i - t_l \geq w_{li} - M(1 - x_{j,k,l,i}) \\ x_{j,k,l,i} \in \{0,1\} \end{array}\right\} \quad \begin{array}{l} \forall\, a = (n_j, n_k) \in \mathcal{A}, \\ a^* = (n_l, n_i) \in \mathcal{A}, \end{array} \quad \text{(alternative arcs)}$$

$$\begin{array}{ll} t_0 = 0, & \\ t_i \geq 0 & \forall\, n_i \in \mathcal{N}. \end{array}$$

However, if desired it is also possible to add further terms to the objective function by adding objective variables leading to a more general MILP form of the Alternative graph. This is explored in Appendix B.

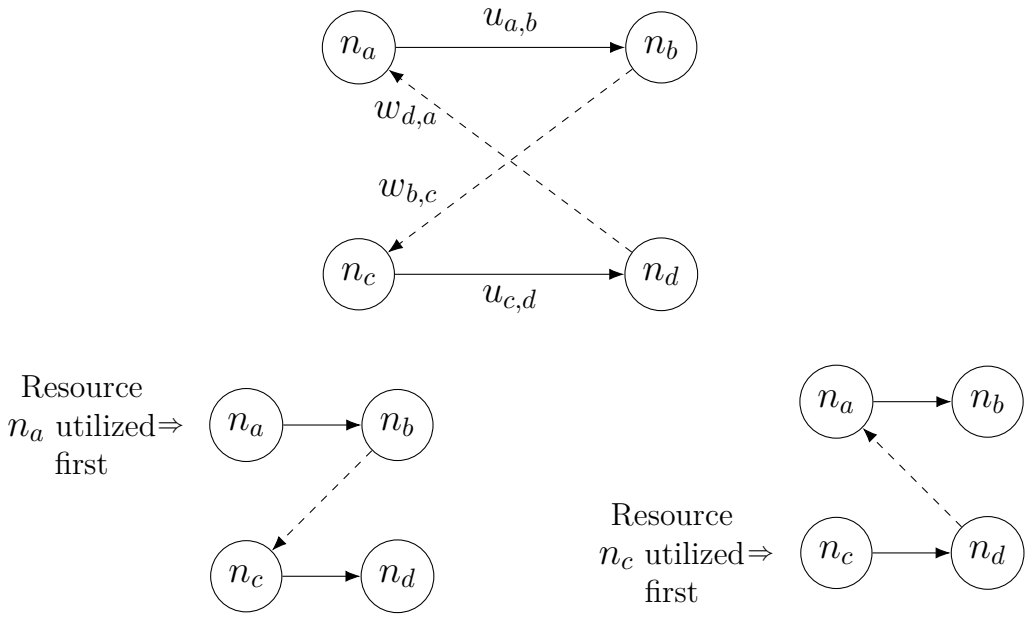


Figure 3.1: The quintessential alternative graph. The node n_a is followed by n_b, the node n_c is followed by n_d, and this is represented by two fixed arcs. Nodes n_a and n_c are in conflict, and this is represented by an alternative arc pair. The choice of arc in the pair represents and ordering decision between n_a and n_c.

3.3 Variable Neighbourhood Search

Variable Neighbourhood search (VNS) is a family of algorithms aimed at solving combinatorial and global optimization problems. These methods are often called metaheuristic since each method is a framework for building heuristics. The main feature of this approach is the construction of various neighbourhoods around points in the search space, and was created by Mladenović in 1995 [9].

In this method we assume that we have a deterministic optimization problem

$$\min \{g(\xi) | \, \xi \in \Xi \subseteq S\}, \tag{3.5}$$

where g is the objective function (i.e. $z = g(\xi)$), ξ is a feasible solution[†], Ξ is the set of all feasible points, and S is the variable space.[‡]

We define an abstract neighbourhood $N = \mathfrak{N}(\xi) \subseteq S$ as a set of solutions that depend on the abstract neighbourhood structure $\mathfrak{N}$ around some solution ξ. For example, the classical neighbourhood for a point $\mathbf{x} \in \mathbb{R}^n$ is all points within some radius r of $\mathbf{x}$, i.e., $\mathfrak{N}\mathbf{x} = \left\{\mathbf{x}' \,\middle|\, |\mathbf{x} - \mathbf{x}'| \leq r\right\}$. However, $\mathfrak{N}$ can involve more abstract structures and sets (in our case the routing of trains, to be expanded on later).

The two central concepts within optimization are global minima ξ_g and local minima ξ_l which may be described by

$$g(\xi_g) \leq g(\xi) \quad \forall \xi \in \Xi, \tag{3.6a}$$
$$g(\xi_l) \leq g(\xi) \quad \forall \xi \in \mathfrak{N}(\xi_l) \cap \Xi. \tag{3.6b}$$

However, in practice it may be impractical to search enough of the solution space to guarantee that the global minimum is found. To construct the VNS algorithms we rely on three axiomatic facts regarding optima, as stated by Hansen et al. [9, p. 321]:

Fact 1 *A local minimum with respect to one neighbourhood structure is not necessarily a local minimum for another neighbourhood structure.*

Fact 2 *A global minimum is a local minimum with respect to all possible neighbourhood structures.*

Fact 3 *For many problems local minima with respect to one or several neighbourhoods are relatively close to each other.*

Using these facts we can construct a collection of neighbourhood structures $N_k, k = 1, \ldots, K$. Each neighbourhood is then written as $N_k = \mathfrak{N}_k(\xi)$. The goal of most VNS algorithms is to find a solution $\xi^\star$ such that

$$g(\xi^\star) \leq g(\xi) \quad , \quad \forall \xi \in \left(\bigcup_{k=1}^{K} \mathfrak{N}_k(\xi^\star) \right) \cap \Xi \tag{3.7}$$

where, due to **Fact 2**, we hope that $\xi^\star$ is a global minimum, or a good approximation thereof. For an efficient search, Mladenović recommends that the size of neighbourhoods N_k increase in size with k, and this index is thus often referred to as the neighbourhood "size". Additionally,

[†]A feasible solution is a point such that fulfills all constraints of the problem.

[‡]We use the symbols "ξ" and "Ξ" instead of the otherwise more usual "x" and "X", since the latter symbols are used for MILP.

it is convenient if different neighbourhoods around the same point are disjoint, but this is not strictly required.

Finding a true global minimum can only be guaranteed when the union of neighbourhoods effectively cover the solution space, defeating the purpose of VNS. In practice, both neighbourhoods and the number K thereof are very small compared to the solution space, with neighbourhoods chosen in some way that the final solution is good even if not globally optimal.

The VNS family of algorithms consists of multiple components, including four common components that are described hereafter. The most common component is "neighbourhood Change" which, as the name implies, handles how the neighbourhood is changed; see Algorithm 1. Most VNS strategies iterate around this function in some fashion, such that when $k = K$ we know that the conditions stated in Equation (3.7) hold true.

The second component is "Shake", which, given a solution ξ, returns a random $\xi' \in N_k$, see Algorithm 2. This is used to search across longer distances and to avoid sticking to local minima, in a manner similar to other stochastic optimization methods.

Algorithm 1 Neighbourhood Change

Also known as the "move or not" function.

Function NeighbourhoodChange(ξ, ξ', k)
  Input: ξ: *The current best feasible solution*
    ξ': *The comparing feasible solution*
    k: *The index of the neighbourhood*

  If $g(\xi') < g(\xi)$ **Then**
    $\xi \leftarrow \xi'$
    $k \leftarrow 1$
  Else
    $k \leftarrow k + 1$
  End If

  Return ξ, k
End Function

Algorithm 2 Shake

A Shake function; a random pick from a neighbourhood.

Function Shake(N)
  Input: N: *A neighbourhood of feasible solutions*
  Return Random $\xi \in N$
End Function

The last two common components are the "Best Improvement" and "First Improvement" searches. These are both local search strategies and each specific VNS algorithm generally uses one or the other, though they can be interchanged. The best improvement strategy, seen in Algorithm 3, picks out the best result out of a given neighbourhood N. The first improvement strategy, seen in Algorithm 4, instead picks the first solution that is better than the given solution, assuming some arbitrary ordering of the points in the neighbourhood N.

Algorithm 3 Best Improvement
Local search of a neighbourhood for the best improvement.

Function BestImprovement(ξ, N)
Input: ξ: *The feasible solution to test against*
N: *A neighbourhood of feasible solutions*

$\xi' \leftarrow \arg\min_{y \in N} g(y)$
If $g(\xi') < g(\xi)$ **Then**
Return ξ' ▷*Return found improvement*
Else
Return ξ ▷*Return ξ if no improvement was found*
End If
End Function

Algorithm 4 First Improvement
Local search of a neighbourhood giving the first found improvement.

Function FirstImprovment(ξ, N)
Input: ξ: *The feasible solution to test against*
N: *A neighbourhood of feasible solutions*

For all $\xi' \in N$ **Do**
If $g(\xi') < g(\xi)$ **Then**
Return ξ' ▷*Return first improvement; leave function*
End If
End For

Return ξ ▷*Return ξ if no improvement was found*
End Function

There are many different VNS algorithms, all sharing the property of using different neighbourhoods to find a solution. We study four of the main variants: Variable neighbourhood Decent (VND), Basic VNS, General VNS, and Fleszar-Hindi extension of basic VNS (FH-VNS).

There are many ways to construct VNS methods—a few key features may be used to differentiate the algorithms. One feature is how often shakes are performed during the search. Another feature is how each algorithm treats its local search. Finally one can roughly guess how diverse the methods are. We roughly take this as how random the algorithm behaves in the search space. See Table 3.1.

Due to the fact that VNS is a family of general algorithms, these usually have to be adapted to the specific problem. In Hansen et al. [9, pp. 335–337], many details of how to construct a VNS method is listed. Our main interest is in that it recommends to restrict the neighbourhood to only investigate "promising" subsets. Furthermore it is recommended that if this is done then the best improvement strategy should always be used.

Algorithm	Shake	Local Search	Description
VND	None	Best	Deterministic search of neighbourhoods.
Basic VNS	Single	Best/First	Shake followed by descent.
General VNS	Single	VND from 1 to K	Shake followed by VND.
FH-VNS	Repeated	Best/First	Repeated Basic VNS, keeping best result.

Table 3.1: Summary of the different VNS algorithms described in this report.

3.3.1 Variable Neighbourhood Decent (VND)

VND is the only method of the algorithms that is deterministic, as it does not use the shake component to diversify the search space. The algorithm finds the best solution in the first neighbourhood and, if it is better than the current, it restarts with the new best solution as the starting point, otherwise it repeats with the next neighbourhood until all neighbourhoods have been searched. A visualization of how a VND algorithm works can be seen in Figure 3.2.

Algorithm 5 Variable Neighbourhood Decent
A search done by selecting the best solution in the neighbourhood around the current solution. If an improvement is found the neighbourhood is reset to $k = 1$ and relocated around that point; otherwise the search continues in the next neighbourhood.

Assume: $\mathfrak{N}$: *The neighbourhood structures*
Function VND(ξ, $k_{\max}$)
  Input: ξ: *The initial feasible solution*
    $k_{\max}$: *The maximum number of visited neighbourhoods*
  Output: ξ: *The best found feasible solution*

  Repeat
    $k \leftarrow 1$
    $\xi'' \leftarrow \xi$
    Repeat
      $N \leftarrow \mathfrak{N}_k(\xi)$ ▷*Build the neighbourhood*
      $\xi' \leftarrow$ BestImprovement(ξ, N)
      $(\xi, k) \leftarrow$ NeighbourhoodChange(ξ, ξ', k)
    Until $k = k_{\max}$
  Until $g(\xi'') = g(\xi)$
  Return ξ
End Function

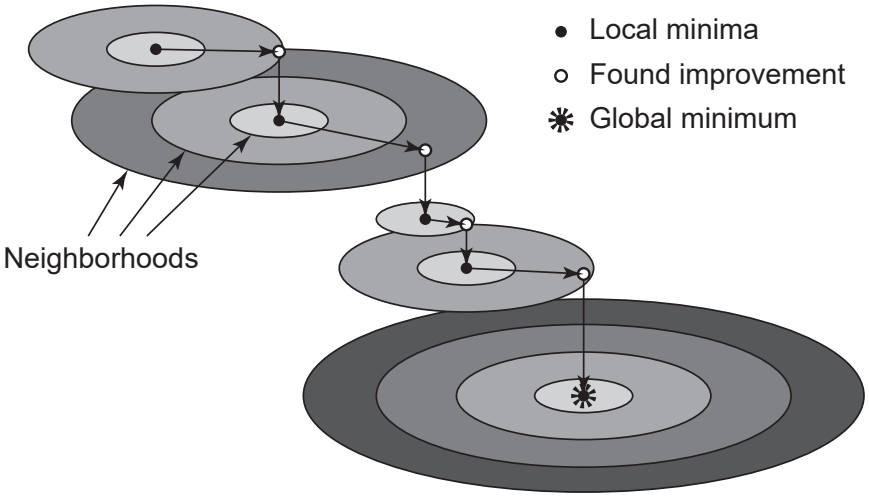


Figure 3.2: Illustration of the VND algorithm. It starts at some initial point and investigates its neighbourhoods, represented by the circles around the points, from the smallest to the largest until it finds a improvement. It then starts over, investigating the neighbourhoods of the new point. This continues until no improvement has been found in any neighbourhood around the current point.

3.3.2 Basic VNS

Basic VNS is a very simple VNS method that includes a local search. It performs a shake in the neighbourhood $\mathfrak{N}_k(\xi)$ to find ξ'. It then builds a new neighbourhood $\mathfrak{N}_k(\xi')$, using the same neighbourhood structure, around that point. It then finds the first (or best) improvement in the new neighbourhood and uses this to find a new candidate.

Algorithm 6 Basic VNS

A search methodology which picks a random solution from the neighbourhood around the incumbent solution, and then selects the first improvement found in a neighbourhood around that solution. This is then compared against the incumbent solution to select how to continue, using the Neighbourhood Change method.

Assume: $k_{\max}$: *The maximum number of investigated neighbourhoods*
$\mathfrak{N}$: *The neighbourhood structures*
$T_{\max}$: *The maximal allowed runtime*
Function BVNS(ξ)
Input: ξ: *The initial feasible solution*
Output: ξ: *The best found feasible solution*

Repeat
$k \leftarrow 1$
Repeat
$\xi' \leftarrow$ SHAKE($\mathfrak{N}_k(\xi)$)
$\xi'' \leftarrow$ FIRSTIMPROVEMENT(ξ', $\mathfrak{N}_k(\xi')$) ▷*Could be BestImprovement*
$(\xi, k) \leftarrow$ NEIGHBOURHOODCHANGE(ξ, ξ'', k)
Until $k = k_{\max}$
$T \leftarrow$ CPUTIME
Until $T > T_{\max}$
Return ξ
End Function

3.3.3 General VNS

General VNS uses both shakes and VND in order to find a solution. The idea is to generate a random point from the current neighbourhood and use it to start a VND search to generate a test candidate. This test candidate is then used to change the neighbourhood.

Algorithm 7 General VNS
A search methodology that finds a random point from which a VND search is performed. The result from the VND search is then used for comparison with the current solution, the best solution kept, and the process repeated.

Assume: $k_{\max}$: *The maximum number of investigated neighbourhoods*
$k'_{\max}$: *The maximum number of investigated neighbourhoods for the VND*
$\mathfrak{N}$: *The neighbourhood structures*
$T_{\max}$: *Maximal allowed runtime*
Function GVNS(ξ)
Input: ξ: *The initial feasible solution*
Output: ξ: *The best found feasible solution*

Repeat
$k \leftarrow 1$
Repeat
$\xi' \leftarrow$ SHAKE($\mathfrak{N}_k(\xi)$)
$\xi'' \leftarrow$ VND(ξ', $\mathfrak{N}$, $k'_{\max}$)
$(\xi, k) \leftarrow$ NEIGHBOURHOODCHANGE(ξ, ξ', k)
Until $k = k_{\max}$
$T \leftarrow$ CPUTIME( )
Until $T > T_{\max}$
Return ξ
End Function

3.3.4 FH-VNS

The Fleszar–Hindi extension of basic VNS is interesting in that the neighbourhood size is held constant while a number of Basic VNS steps is done. The results from these steps are then used to change the neighbourhood.

Algorithm 8 FH-VNS

A search methodology where the current solution is saved, in ξ_s, and the neighbourhood size is held constant for k search attempts. This inner search is performed analogously to Basic VNS, except that the neighbourhood size is here constant. The best result from these k attempts is then compared to the saved solution, and the neighbourhood may then change.

Assume: $k_{\max}$: *The maximum number of investigated neighbourhoods*
$\mathfrak{N}$: *The neighbourhood structures for different* k
$T_{\max}$: *Maximal allowed runtime*
Function FHVNS(ξ)
Input: ξ: *The initial feasible solution*
Output: ξ: *The best found feasible solution*

Repeat
$k \leftarrow 1$
Repeat
$\xi_s \leftarrow \xi$
For $l = 1$ to k **Do**
$\xi' \leftarrow$ Shake($\mathfrak{N}_k(\xi)$)
$\xi'' \leftarrow$ FirstImprovement(ξ', $\mathfrak{N}_k(\xi')$) ▷*Could be BestImprovement*
$\xi \leftarrow$ KeepBest(ξ, ξ'')
End For
$(\xi, k) \leftarrow$ NeighbourhoodChange(ξ_s, ξ, k)
Until $k = k_{\max}$
$T \leftarrow$ CPUTIME
Until $T > T_{\max}$
Return ξ
End Function

Chapter 4

Model Description

In this chapter we present the three stages by which we model PCR problems. Initially, a *Network Model* describes the layout of the tracks and how the trains *might* travel. Given a network and a set of paths, we then construct an alternative graph and *Schedule Model* describing how the trains *may* travel. Finally, the alternative graph is converted into an MILP problem, the solution of which describes how trains *should* travel. Put another way, this is analogous to a reduction from a variable space of arbitrary train movement, to a solution space where movement is subject to various constraints, and finally to a specific (and ideally optimal) solution describing the desired train movement.

It should be noted that the individual problems and solutions discussed in this chapter primarily refer not to the the PCR problem in its entirety, but sub-problems where each train has a single path to take. Repeated exploration of these sub-problems with various different paths is driven by VNS as discussed in the previous chapter.[†]

4.1 Network Model

In order to describe how trains travel across a rail network, we must first properly describe this rail network. For this purpose we introduce a graph model that represents the physical rail.

An instance of the network model is defined as the triplet $\mathcal{G}_N = (\mathcal{M}, \mathcal{E}, \mathcal{C})$ of nodes $\mathcal{M}$, directed edges $\mathcal{E}$, and conflicts $\mathcal{C}$. A node $m \in \mathcal{M}$ represents a section of track, in a specific direction of travel (two-way track sections thus correspond to two nodes), while an edge $e_{ij} \in \mathcal{E}$ represents a valid transition from node m_i to m_j. Conflicts $c \in \mathcal{C}$ describe how nodes interact with regards to occupancy and are described further in Subsection 4.1.1. See Figure 4.1 for a small illustration, and Figure 4.2 for a larger example.

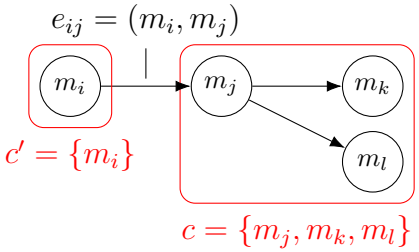


Figure 4.1: An4q illustration of a subset of a network model, with four nodes, three edges, and two conflicts.

Our problem deals with a set of trains $\mathcal{T} = \{\tau_1, \tau_2, \ldots\}$ each of which has an associated origin and destination location(s)[‡], as well as potential waypoints and other requirements imposed on a train and its

[†]It is possible to combine several of these smaller problems into a larger "complete" MILP problem, but such a problem quickly becomes difficult to manage and considered out of scope for this thesis.

[‡]Each train must have at least one destination but may have several, in which case paths may terminate at any one of them.

path. For ease of readability we will use α and β to indicate two different, arbitrary trains instead of τ_i and τ_j or similar.

A path p is a sequence of nodes, where each node follows from its preceding node according to a valid transition in $\mathcal{E}$, described as

$$p = (m_1^p, m_2^p, \ldots, m_e^p), \quad m_i^p \in \mathcal{M}, \quad \left(m_{(i-1)}^p, m_i^p\right) \in \mathcal{E}. \tag{4.1}$$

where we use m_e^p to denote the last node in the path. It is important to note that the value of e is dependent on the path p, and e is therefor sometimes clarified by writing it as e^p.

The paths relevant to us are those associated with each train $\alpha \in \mathcal{T}$, which we denote as $\mathcal{P}^\alpha$. This is the set of all *valid* paths p^α for train α (see Chapter 2 and the introduction of Chapter 5). We are also particularly interested in sets P containing exactly one element from each of the aforementioned sets, i.e., a selection of paths such that each train $\alpha \in \mathcal{T}$ is associated with exactly one element $p^\alpha \in P$.

For a node $m_i^{p^\alpha}$ where both the path p^α and the train α is important the shorthand $m_i^{\alpha p}$ is instead used for ease of reading. Similarly, an end node $m_{e^p}^{p^\alpha}$ may be referred to as $m_e^{\alpha p}$.

4.1.1 Conflicts

While the nodes $\mathcal{M}$ and edges $\mathcal{E}$ are sufficient for a single train, some nodes may overlap in the physical world in non-trivial ways such that we cannot safely model multiple trains (see Figure 4.2). For this reason we need to define the family of conflicts $\mathcal{C}$. Each conflict $c \in \mathcal{C}$ is a set of nodes which overlap in the sense that multiple trains may not safely occupy any subset of c concurrently.

For convenience, all nodes are required to be part of a conflict, i.e., $\forall m_i \; \exists\, c \in \mathcal{C} : m_i \in c$, though for some nodes (sections of one-way track) this may just be a trivial conflict containing only the node itself ($c = \{m_i\}$). This implies that all nodes conflict with themselves, which is desired as we might otherwise construct situations where two trains would occupy the same space.

Given a path p that passes through a conflict c, i.e, $\exists\, i : m_i^p \in c$, we refer to each maximal contiguous subsequence of such nodes in p as a *conflict interval* on p with regard to c. Specifically, it is an interval $[i, j]$ defined on p such that

$$m_k^p \in c \; \forall\, k \in [i, j] \quad \wedge \quad \left(m_{i-1}^p \notin c \vee i = 1\right) \quad \wedge \quad \left(m_{j+1}^p \notin c \vee j = e\right) \tag{4.2}$$

A path may have multiple conflict intervals for a single conflict, if and when the path passes through the same conflict on multiple occasions. This fact is the main reason that this specific definition is important, and is required later on in Subsection 4.2.1.

4.1.2 Associated Properties

There are a number of relevant properties associated with the physical train network, that must also be described in the network model. In particular: the length of the track corresponding to the nodes, the maximum velocity permitted on that track, and the corresponding minimum traversal time for each node. For generic nodes m_j these properties are simply written as l_j, $v_j^{\max}$, and $t_j^{\min}$, respectively. However, we will generally be more interested in what these properties are for specific trains on specific paths, or even for entire paths or parts thereof.

The common case is that some train α travels on some path p, and the time it takes the train to traverse a node $m_i^{\alpha p}$ is defined as

$$t_i^{\alpha p,\min} = \frac{l_i^p}{v_i^{\alpha p,\max}}, \quad \text{where} \quad v_i^{\alpha p,\max} = \min(v^{\alpha,\max}, v_i^{p,\max}), \tag{4.3}$$

i.e., the train travels across a given physical distance at a velocity determined by its own maximum speed and the track's maximum allowed speed, whichever is lowest.

We can easily extend this to intervals on paths by implying a summation across the given interval, giving lengths

$$l^p_{[i,j]} = \sum_{i \le k \le j} l_k^p \quad \text{and} \quad l^p_{]i,j[} = \sum_{i < k < j} l_k^p, \tag{4.4}$$

and minimum times

$$t^{\alpha p,\min}_{[i,j]} = \sum_{i \le k \le j} t_k^{\alpha p,\min}. \tag{4.5}$$

Furthermore we can also add a time penalty t_{ij}^{penalty} for each edge e_{ij} such that for any path that passes by this edge an additional cost is added. This penalty is relevant for choosing paths in both Section 5.1 and Section 5.2. In most cases this is zero. However, for edges to that should be avoided it can be strictly positive.[†]

[†]In our case we have the penalty set to zero for all edges.

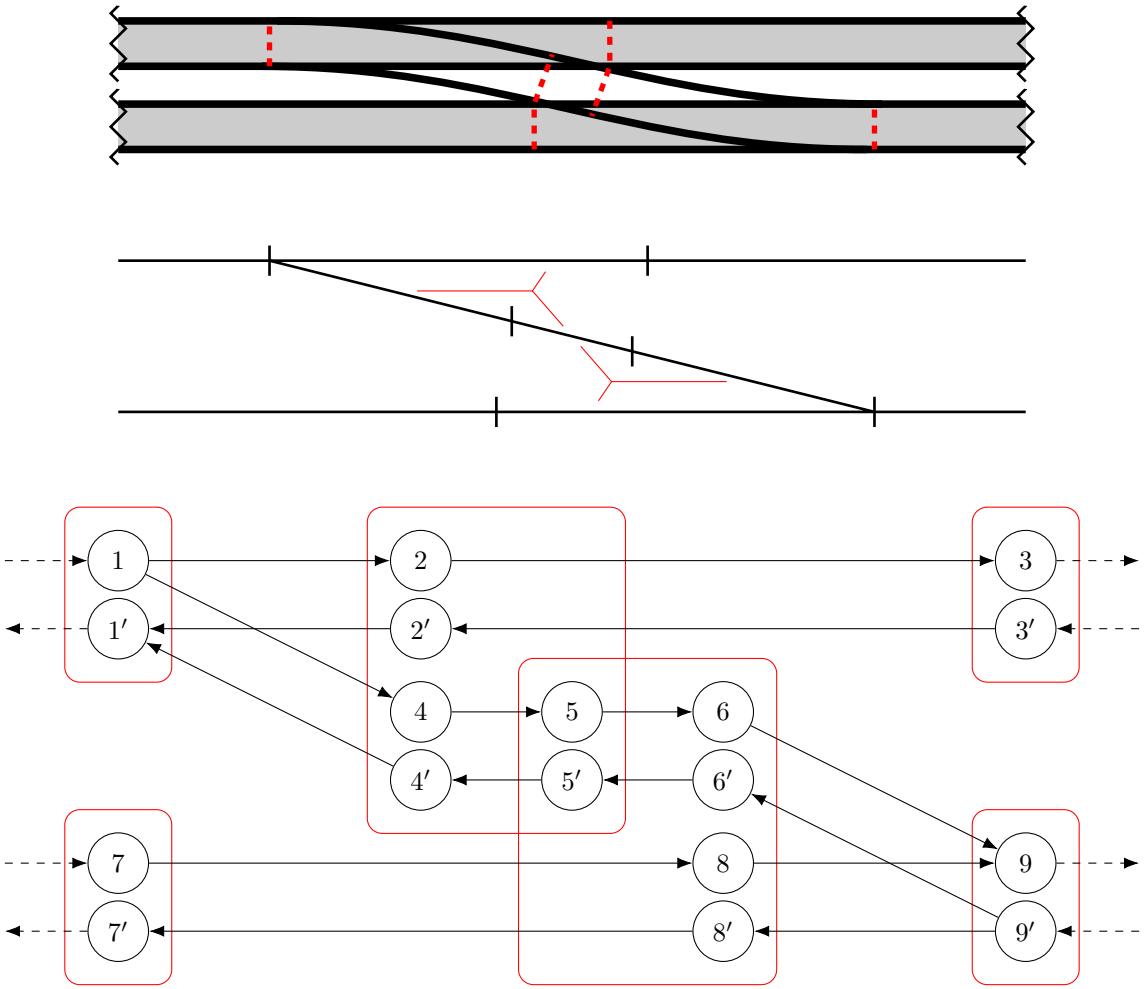


Figure 4.2: A larger network model instance, illustrating correspondence between the physical rail (top), a simplified illustration (middle), and the network graph (bottom).

The physical rail shown in the figure is a pair of parallel tracks (top, shaded), connected by a crossover which connects the upper and lower tracks. The dividing lines between track segments are drawn in dashed red, and their positioning is the result of the physical extents of the two switches that create the crossover. Of particular interest is the fact that the two switches actually overlap in the middle of the crossover, such that any obstacle on that segment will obstruct *both* switches.

The simplified illustration (middle) explicitly shows these non-trivial conflicts by connecting the track segments with two sets of red lines.

In the corresponding network graph (bottom), all conflicts are illustrated by red boxes around the relevant nodes, and the two switches correspond to the two larger such boxes.

4.2 Scheduling Model

Once a specific path has been selected for each train, there remains the problem of scheduling the trains so that they take their respective paths without colliding or causing deadlocks. For this purpose we use the method of alternative graphs (see also Section 3.2). In addition to the graph that is constructed here it is also possible to add additional arcs representing different types of constraints, which can be found in Appendix A.

4.2.1 Alternative Graph Construction

Given a set of selected paths P from the network model $\mathcal{G}_N$, we now wish to construct an alternative graph $\mathcal{G}_A$ that describes the scheduling of these paths. Each path $p \in P$ defined in $\mathcal{G}_N$ will correspond to a distinct path $p' \in P'$ in $\mathcal{G}_A$ (which we write as $p' \sim p$), such that

$$p' = (n_1^{p'}, n_2^{p'}, \ldots, n_e^{p'}, n_{e+1}^{p'}) \tag{4.6a}$$

$$q' \neq p' \;\Rightarrow\; n_j^{q'} \neq n_i^{p'} \;\forall\, i, j \tag{4.6b}$$

where $n_{e+1}^{p'}$ is a *virtual node* for each path, that only exists in $\mathcal{G}_A$ and do not have a corresponding node in $\mathcal{G}_N$. All nodes in all paths p' are unique. Furthermore, the scheduling times t associated with the nodes $\mathcal{N}$ can then also be written as $t_i^{p'}$. The first time $t_1^{p'}$ for each path is a special case in that it is given a constant value $t_1^{\alpha p'} = t^{\alpha,\text{enter}}$ which is the time the train α enters the network, meaning the time it starts being modelled and obstructs other trains.

In addition to these nodes generated by the paths, we also need to include a virtual node n_0 with constant time $t_0 = 0$, as mentioned in Section 3.2. The start of every path will have a fixed arc from n_0 to $n_1^{\alpha p'}$ with weight $u = t^{\alpha,\text{enter}}$. This virtual node, and these arcs from it, do not change the problem being modelled or its solutions. It does, however, allow us to define the solution times using makespans (cf. Equation (3.2)) according to

$$t_i^{p'} = \ell(n_0, n_i^{p'}). \tag{4.7}$$

The relationship between the scheduling nodes $\mathcal{N}$ and the physical nodes $\mathcal{M}$ can be described as

$$\forall n_i^{p'} \in p' \setminus \{n_{e+1}^{p'}\}\ \exists!\, m_j : n_i^{p'} \sim m_j \quad \text{and that} \quad m_j \equiv m_i^p \sim n_i^{p'} \tag{4.8a}$$

$$\nexists m_j \sim n_0 \qquad \nexists m_j \sim n_{e+1}^{p'} \tag{4.8b}$$

meaning that all *real* nodes in $\mathcal{N}$ have corresponding nodes in $\mathcal{M}$, while the virtual nodes do not. An important detail is that this is a many-to-one correspondence, i.e., every real node $n \in \mathcal{N}$ corresponds to exactly one $m \in \mathcal{M} : n \sim m$ but there may be any number of other nodes $n' \neq n : n' \sim m$.

We also want to relate the nodes n in the alternative graph to the conflict zones c. This is done by defining a conflict zone c' in the alternative graph such that

$$n_i^{p'} \in c' \quad \text{iff} \quad \exists!\, m_i^p \sim n_i^{p'} \quad \text{and} \quad m_i^p \in c \tag{4.9}$$

implying that the virtual nodes are not in any conflict zone.

For convenience and readability we may now drop the apostrophe for both p' and c', because of two reasons. First, p and p' are the same abstract path and every real node $n_i^{p'}$ has a

corresponding m_i^p node. The second reason is that c' and c are the same abstract conflict but described in two different ways.

We will now describe how both the "normal" fixed and alternative arcs in both $\mathcal{F}$ and $\mathcal{A}$ are generated. There may be more arcs of either type, but these are regarded as complications and are to some extent addressed separately in Appendix A, Complications.

Fixed arcs due to paths

Each path p describes a chain of nodes n, going from the path's start node n_1^p to the virtual end node n_{e+1}^p, with real nodes in order in-between. This order is imposed by fixed arcs f from each node to its succeeding node:

$$\begin{aligned} f_i^p = f_{i,i+1}^{p,p} &= (n_i^p, n_{i+1}^p) \\ \Rightarrow t_{i+1}^p - t_i^p &\geq u_i^p \end{aligned} \qquad 1 \leq i \leq e^p, \forall p \in P \tag{4.10}$$

The weights u of these nodes generally describe the minimum time it takes the train α on path p to traverse node n_i^p, with the primary exceptions[†] being the first arc f_1^p and the last arc f_e^p, according to

$$u_i^{\alpha p} = t_i^{\alpha p,\min}, \tag{4.11a}$$

$$u_1^{\alpha p} = t_1^{\alpha p,\min} + t^{\alpha,\text{wait}}, \tag{4.11b}$$

$$u_e^{\alpha p} = t_e^{\alpha p,\min} + t^{\alpha,\text{dwell}}, \tag{4.11c}$$

where $t^{\alpha,\text{wait}}$ is an additional delay from when train α enters the network until it may start moving, and $t^{\alpha,\text{dwell}}$ is a delay from the time $t_e^{\alpha p}$ when the train enters its destination node until it leaves the network[‡].

Alternative arcs for train conflicts

While the nodes and fixed arcs described so far are fully sufficient to describe and schedule trains in the absence of conflicts, we generally expect to have a number of conflicts c through which two or more trains pass. These conflicts are modelled using alternative arcs a and a^* for every pair α, β of trains passing through each conflict.

A choice between alternative arcs a and a^*, related to a conflict c, is a choice between which train goes first through that conflict. Specific arcs may be identified using notation like $a_{jk}^{pq} = (n_j^p, n_k^q)$, i.e., an arc from node j of path p to node k of path q. The conjugate can be written as $(a_{jk}^{pq})^* = a_{li}^{qp} = (n_l^q, n_i^p)$. As mentioned in Subsection 3.2.2 the nodes n_j^p and n_l^q are called release nodes while n_i^p and n_k^q are the blocked nodes. The question now is how the pairs a, a^* are generated from the conflicts $\mathcal{C}$.

Given a conflict $c \in \mathcal{C}$, every conflict interval I (see Subsection 4.1.1) for conflict c must interact with every other conflict interval J for c, where I and J correspond to different trains. Each such pair I, J of conflict intervals will result in one alternative arc pair a, a^*, and requires us to find the correct release nodes and blocked nodes that will allow us to impose the desired constraint (that trains must not collide and must obey desired safety margins).

[†]See Initial Nodes in Subsection 4.2.2 for further modifications.

[‡]The opposite of "entering the network," meaning that it is then removed from the model and no longer obstructs other trains.

Defining the arcs using the conflicts require that for each interval we know which node the train has to enter for the train to have left the interval. To do this the release node is specified with the help of the release function $\varphi^{\alpha p}(i)$ which, for a specific train and path, takes an index i of a node in p and returns the index $\varphi^{\alpha p}(i) > i$ identifying the relevant release node[†].

Assuming that $I = [i, j]$ for train α on path p, and $J = [k, l]$ for train β on path q, the desired arcs are given by

$$a \equiv a^{\alpha p, \beta q}_{\varphi(j),k} = \left(n^{\alpha p}_{\varphi(j)}, n^{\beta q}_{k}\right) \tag{4.12a}$$

and

$$a^* \equiv a^{\beta q, \alpha p}_{\varphi(l),i} = \left(n^{\beta q}_{\varphi(l)}, n^{\alpha p}_{i}\right). \tag{4.12b}$$

The choice of constraints imposed by these alternative arcs, by combining implications (3.3) and the relations (4.12), is

$$t^{\beta q}_k - t^{\alpha p}_{\varphi(j)} \geq w^{\alpha p, \beta q}_{\varphi(j),k} \quad \text{or} \quad t^{\alpha p}_i - t^{\beta q}_{\varphi(l)} \geq w^{\beta q, \alpha p}_{\varphi(l),i}. \tag{4.13}$$

This means that the blocked nodes are the beginnings of the intervals, so that the front of the succeeding (blocked) train is blocked from entering the conflict until the back of the preceding train has left the conflict. Exactly when this occurs is determined by the choice of release nodes, using the release function φ, together with the arc weights w and w^*.

Furthermore, it is possible to include additional alternative arcs (or fixed arcs) to represent other complications. They are explored theoretically in Appendix A but not included in the model.

4.2.2 Train Length and Moving Block

The simplest way to define the release node function is as the next node n^p_{i+1} along the path p, i.e., $\varphi(i) = i + 1$. This would mean that each conflict interval is considered empty once the train occupying it has entered the node after said conflict interval. This would work and behave correctly as long as trains are modelled as having no length ("point trains"). However, this no longer works when train the length is accounted for since the back of the train may then overlap with a succeeding train, as illustrated in Figure 4.3a.

In order to resolve this we need to define the release node such that the back of the preceding train has left the conflict when the front of the same train enters the release node, illustrated in Figure 4.3b. This is done by defining the release node as

$$\varphi^{\alpha p}(i) = \min\left\{j \ : \ j > i \ \wedge \ l^p_{]i,j[} \geq l^\alpha\right\} \tag{4.14a}$$

where l^α is the length of train α. Using this gives the result seen in Figure 4.3b. In cases where the above equation has no possible value (implying the release node would be after the end of the path), we instead use the virtual node after the path as the release node:

$$\varphi^p(i) = e^p + 1 \quad \text{if} \quad \nexists\, j > i : l^p_{]i,j[} \geq l^\alpha \tag{4.14b}$$

So far, no explicit values for the weights w have been stated, but the illustrations in (a) and (b) of Figure 4.2 correspond to the simplest definition $w = 0$. While this leads to a safe behaviour,[‡] the back of the preceding train is likely to have left the conflict area some time

[†]Since $\varphi^{\alpha p}(i)$ is often used as an index, such as $n^{\alpha p}_{\varphi^{\alpha p}(l)}$ where α and p are duplicated, we write it as $n^{\alpha p}_{\varphi(l)}$ implying $n^{\alpha p}_{\varphi^{\alpha p}(l)}$.

[‡]In the sense that collisions between trains are prevented.

before the front of the same train reaches the release node, and this time is thus potentially wasted.

To compensate for this we change the weight so that it is defined as

$$w^{\alpha p,\beta q}_{\varphi(j),k} = w_0 - t^{\min}_{\varphi(j)-1} \frac{l^p_{]j,\varphi(j)[} - l^\alpha}{l^p_{\varphi(j)-1}} \tag{4.15}$$

where w_0 is some base value representing a safety margin, which we here assume to be zero for simplicity. This results of this new definition is illustrated in Figure 4.3c.

Moving block

The previous section deal with so called fixed block behaviour, where trains are allowed to enter a block only when all other trains have left. However, as illustrated in Figure 4.3d, this will lead to excessive delays between trains in cases where these blocks are long. We therefore wish to move on to moving blocks, where each train can be said to be a block on its own and other trains may enter the same piece of track provided that some safety margins are met.

Three criteria must be met for moving block to be used; otherwise the fixed block equation above is used instead. Firstly, the physical length of the conflict must be longer than the distance margin L^S for moving block to make a difference. Secondly, the time it takes the first train to travel through the conflict must be longer than the time margin T^S, for the same reason. Finally, the two trains involved must be traveling across the same physical track and in the same direction; otherwise none of this makes sense.

The new arc weight w is computed in a few steps as follows:

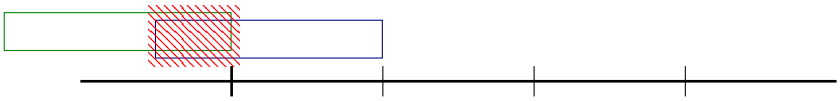
1. Compute the distance d that the first train can travel during the time margin T^S.
2. Let the effective distance margin be $d' = \max\{d, L^S\}$. This ensures that both the time and distance margins are met.
3. Compute the distance the train needs to "back up" from the release node in order for the back of the train to meet the effective margin; $d'' = l^p_{]i,\varphi(j)[} - l^\alpha - d'$.
4. Let $-w$ be the time it takes the train to travel this distance d'' *ending at the release node.*

The result of this new weight calculation is illustrated in Figure 4.3e.

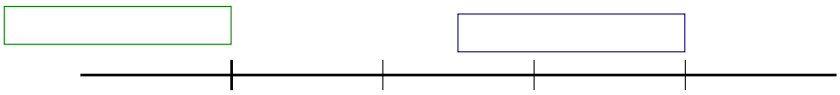
Initial Nodes

An additional detail arises due to train length, where the initial positions of trains require special treatment to make sure that all nodes that each train initially occupies are labeled as such. This is accomplished by having the train path p start from the *back* of the train α, with zero-weight fixed arcs (instead of the regular fixed path arcs) from $n^{\alpha p}_1$ until the node $n^{\alpha p}_i$, which represents the initial front of the train.

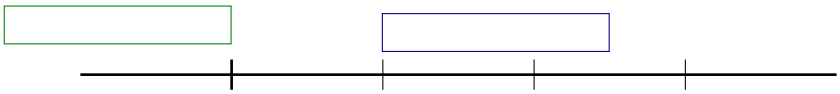
Figure 4.3: Illustrative examples of train length and moving block situations, motivating why Equation (4.15) has been written as it has. Each subfigure below involves two trains (the blue train A to the right, and the green train B to the left) travelling towards the right of the page along a straight rail. The rail is divided into several blocks, separated by short vertical marks in the figures. In all cases, the conflict interval being considered is the single block after the first vertical mark. The point in time being illustrated is when train B is allowed to enter the conflict interval, and the place train A is at when this happens.



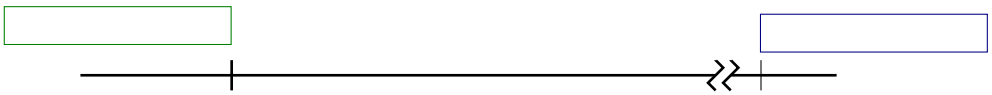
(a) Naive solution from alternative graph theory, where the conflict interval is considered empty when the front of train A has left the interval and entered the subsequent node (the release node, i.e., $\varphi(i) = i + 1$ in this subfigure). This results in a collision.



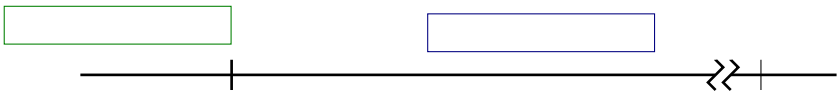
(b) As (a) but with the release node defined using Equation (4.14a), so that train A is just about to enter the release node and has completely cleared the conflict interval.



(c) As (b) but the weight of the alternative arc (which constrains the delay between the trains) has been changed to use the relation (4.15).



(d) As (c) but where the conflict interval is a very long track. This conflict area is handled correctly in that collision is forbidden, but also forces a long delay between trains which is undesired.



(e) As (d) but with the arc weight adjusted according to moving block, such that train B may now enter the same fixed block as train A after some safety margins have been met.

4.2.3 Choice Couplings

It is important to note that any interesting (read computationally difficult) problem will result in many alternative arcs. There is no good way to reduce the number of arcs, since they represent the constraints required for a valid solution. However, it is possible to decrease the number of choices that must be made.

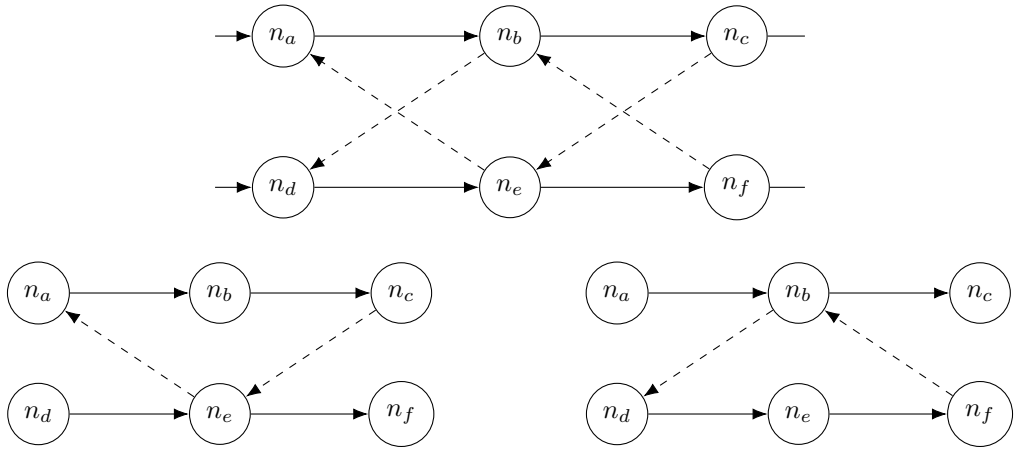
This is possible in cases where we can show that only a subset of alternative arc selections are feasible. Specifically, we look for cases where we can prove that selecting an arc a of one alternative arc pair forces the selection of some arc a' of another pair, and vice versa. We say that the arcs are *coupled*, and notate this as $a \models\!\!\dashv a'$. A example of such a scenario can be seen in Figure 4.4.

The coupling operator is trivially transitive and induces sets K such that for all arcs $a \in K$, $a' \in K \Rightarrow a \models\!\!\dashv a'$. We call these sets *couplings*, and the aforementioned property means that either all arcs in a set are selected or none of them are. Additionally, since alternative arcs come in pairs, so too must the couplings, i.e., $a \in K \Leftrightarrow a^* \in K^*$.

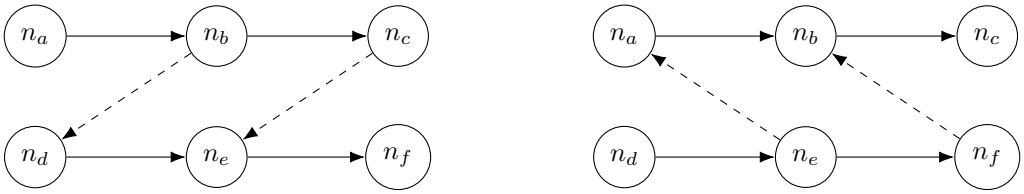
For simplicity we require that all alternative arcs are included in exactly one coupling, or equivalently that the set $\mathcal{K}$ of all couplings is an exact cover of the alternative arcs $\mathcal{A}$. The choices required are thus between pairs of couplings (K, K^*) instead of arcs (a, a^*), which can drastically reduce the size of the search space (which will be proportional to $2^{|\mathcal{K}|/2}$ rather than $2^{|\mathcal{A}|/2}$).

If we construct all $K \in \mathcal{K}$ such that $\forall\, a \in K, a' \in K' \Leftrightarrow a \not\models\!\!\dashv a'$ then we call the set $\mathcal{K}$ "perfectly" coupled. A perfect coupling means that there are no arcs $a \in K$ and $a' \in K'$ such that $a \models\!\!\dashv a'$, also meaning that the couplings represent the minimum amount of binary choices that is still equivalent to the original problem. Note, however, that our definition does not assume a perfect coupling as this may be difficult to achieve in practice.

The introduction of couplings is especially practical when applied to train scheduling. Since trains travel on rails they cannot simply overtake each other and change order. Alternative arc pairs in our case primarily represent the order in which two trains pass some area, and one can thus intuit that two trains travelling one after the other correspond to a coupling of related arcs.



(a) Positive Cycles



(b) Feasible choices of arcs

Figure 4.4: A subset of an alternative graph, showing six nodes on two paths, with two alternative arc pairs. Assume all arcs have positive weights. Two pairs imply four potential choices of arcs: two result in positive cycles and are thus infeasible, shown in (a); and two do not and are thus feasible, shown in (b). The two pairs here are thus *coupled* and there is in fact only a single binary choice instead of two.

4.2.4 Schedule graph

When the Alternative Graph $\mathcal{G}_A$ has been constructed we may attempt to solve it. When solving such a graph we generate a schedule graph $\mathcal{G}_S = (\mathcal{N}, \mathcal{S})$ which consists of nodes $\mathcal{N}$, and a schedule $\mathcal{S}$. A schedule in this case is a set of arcs from $\mathcal{G}_A$, containing all fixed arcs $\mathcal{F}$ and some number (possibly zero) of selected arcs from $\mathcal{A}$. A *complete* schedule or schedule graph is one in which one alternative arc from every pair in $\mathcal{A}$ has been selected, which implies that all imposed constraints are fulfilled as desired as long as no positive cycles exist.

Objective Function

If we want to solve an alternative graph and generate a complete schedule we have to decide how to evaluate the result, which is equivalent to constructing an objective function. To do this we set an objective function according to the problem description, that is that trains should be able to be prioritized and arrive as fast as possible.

As this definition is rather vague we decide to define the objective function as the arrival times $t_e^{\alpha p}$ for each train on its respective p times its priority C^P, i.e.,

$$z = \sum_{\alpha \in \mathcal{T}} C_\alpha^P \, t_e^{\alpha p}. \tag{4.16}$$

meaning that our basic objective function is a weighted sum of the arrival times for the trains α on their respective paths p.

We have previously talked about using makespans ℓ to define and compute the time values t (see (3.2) and (4.7)), where the makespan between two nodes is defined as the maximum total weight of all paths from the first node to the second. However, we have neglected the fact that this assumes a specific selection of alternative arcs, i.e., the makespan is only well defined on some schedule $\mathcal{S}$.[†] The makespan between two arbitrary nodes n_i and n_j is thus properly written as $\ell^{\mathcal{S}}(n_i, n_j)$.

It is also important to note that if we evaluate all the times using $t_i^{p'} = \ell(n_0, n_i^{p'})$ given the schedule $\mathcal{S}$ we get the earliest times that a train can enter an area. However, it is also possible to find the latest time a train can enter an area; this can be done by fixing the arrival times and working backwards.

[†]Conversely, $\mathcal{G}_S$ and $\mathcal{S}$ are referred to as "schedules" *because* they involve specific well defined time values.

4.3 MILP Model

If we reformulate all the requirements that have been stated, without including the coupling reduction, using the method in Subsection 3.2.3 we arrive at the model to

$$\text{minimize} \quad z = \sum_{\alpha \in \mathcal{T}} C_\alpha^P \, t_e^{\alpha p} + C,$$

subject to

$$t_{i+1}^p - t_i^p \geq u_i^p \qquad \forall\, i \in \{1, \ldots, e^p\}, \forall p \in P, \qquad \text{(fixed arcs)}$$

$$\left.\begin{aligned} t_k^{\beta q} - t_j^{\alpha p} &\geq w_{j,k}^{\alpha p, \beta q} - M x_{j,k,l,i}^{\alpha p, \beta q} \\ t_i^{\alpha p} - t_l^{\beta q} &\geq w_{l,i}^{\beta q, \alpha p} - M(1 - x_{j,k,l,i}^{\alpha p, \beta q}) \\ x_{j,k,l,i}^{\alpha p, \beta q} &\in \{0, 1\} \end{aligned}\right\} \quad \begin{aligned} &\forall\, (n_j^{\alpha p}, n_k^{\beta q}), (n_l^{\beta q}, n_i^{\alpha p}) \in A : \\ &\quad (n_j^{\alpha p}, n_k^{\beta q}) = (n_l^{\beta q}, n_i^{\alpha p})^*, \end{aligned} \qquad \text{(alternative arcs)}$$

$$t_1^p = t^{\alpha, \text{enter}} \qquad \forall\, p \in P,$$

$$t_i^p \geq 0 \qquad \forall\, i \in \{1, \ldots, e^p + 1\}, \forall\, p \in P.$$

If we then add in the coupling reduction the model is then

$$\text{minimize} \quad z = \sum_{\alpha \in \mathcal{T}} C_\alpha^P \, t_e^{\alpha p} + C,$$

subject to

$$t_{i+1}^p - t_i^p \geq u_i^p \qquad \forall\, i \in \{1, \ldots, e^p\}, \forall p \in P, \qquad \text{(fixed arcs)}$$

$$\left.\begin{aligned} t_k^{\beta q} - t_j^{\alpha p} &\geq w_{j,k}^{\alpha p, \beta q} - M x_K \\ t_i^{\alpha p} - t_l^{\beta q} &\geq w_{l,i}^{\beta q, \alpha p} - M(1 - x_K) \\ x_K &\in \{0, 1\} \end{aligned}\right\} \quad \begin{aligned} &\forall\, K, K^* \in \mathcal{K} \\ &\forall\, (n_j^{\alpha p}, n_k^{\beta q}) \in K, (n_l^{\beta q}, n_i^{\alpha p}) \in K^* : \\ &\quad (n_j^{\alpha p}, n_k^{\beta q}) = (n_l^{\beta q}, n_i^{\alpha p})^*, \end{aligned} \qquad \text{(alternative arcs)}$$

$$t_1^p = t^{\alpha, \text{enter}} \qquad \forall\, p \in P,$$

$$t_i^p \geq 0 \qquad \forall\, i \in \{1, \ldots, e^p + 1\}, \forall p \in P.$$

Note that we do not include t_0 since it is simply a constant, and only has fixed arcs to $t_1^p \ \forall p \in P$ which we have defined as constants. However, the corresponding node n_0 is still important when dealing with the model as a graph.

Furthermore we have also introduced a constant C which represents all additions to the cost from outside the MILP problem. For example, a contribution to this are the time penalties t_{ij}^{penalty} that may occur when paths P pass the corresponding edges e_{ij}.

Chapter 5

Solution Methodology

To solve the PCR problem, the overall problem is divided into two core problems: one which we will refer to as the *Path Selection* problem where a path must be selected for each train, and a problem referred to as the *Scheduling* problem where, given the paths P, we find a schedule for the trains. A more complete description of the methodology is illustrated in Figure 5.1.

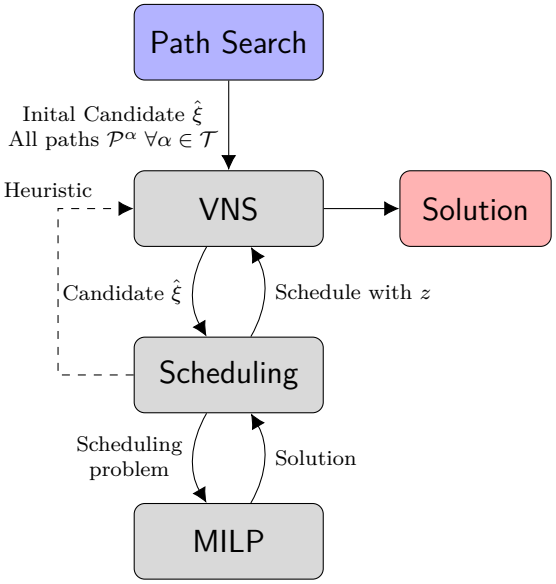


Figure 5.1: Illustration of how the different parts of the solution methodology interact.

The first step is the *Path Search*, where the goal is to find a reasonable set $\mathcal{P}^\alpha$ of possible paths for each train $\alpha \in \mathcal{T}$ using the network model from Section 4.1. Furthermore we must also pick an initial ξ from these paths. Section 3.3 dealt with ξ as a generic, unspecified object but from hereon a candidate ξ refers to a selection of paths such that each train has exactly one path[†]. In our case, the initial candidate used is simple the shortest path for each train individually. If this candidate is infeasible, we randomly select some new paths until we find a feasible solution, or give up entirely if none is found after some number of attempts[‡].

The second step is the VNS, based on the theory in Section 3.3, which is the solution mechanism for the Path Selection problem. It investigates the different combinations of paths by building neighbourhoods around the current solution, and drives the evaluation of candidate solutions where some trains take new paths.

The third step is Scheduling, where a proposed selection of paths ξ from the VNS is used to construct the alternative graph $\mathcal{G}_A$ according to Section 4.2. This includes finding couplings in order to reduce the number of choices, thereby reducing the size of the MILP problem the graph represents.

[†]The notation ξ used mainly as a convenient shorthand and to keep the notation from Section 3.3. It is not defined or used more rigorously than this.

[‡]Anecdotally, we have found this to be very rare and, in cases where no initial solution was found, to indicate that the relevant PCR problem lacks *any* solution.

The last step (per candidate solution) is to solve the MILP problem constructed by the Scheduling layer. This is done by a MILP solver, and results are then passed back up to the Scheduling and VNS systems, guiding the progression of the VNS algorithm.

There are some general details that have to be mentioned here. First, paths selected by the VNS algorithm do not always result in a feasible Scheduling and MILP problem. For that reason we also use the notation $\hat{\xi} \in S$, which we refer to as a candidate and may or may not be a valid solution ξ.

Second, it is often not practical to solve the Scheduling problem optimally. Thus we employ two different objective values for each candidate: a "quick" solve $\hat{g}(\hat{\xi})$, and a "full" solve $g(\hat{\xi})$. The intention is that the quick solve is a "heuristic" solution that is faster to evaluate, and can indicate which candidates are worth sinking more time into with a full solve.

5.1 Path Search

We employ two generalized path searching algorithms.[†] We have not included the easily described but practically intractable method of complete enumeration, i.e., finding *all* paths. As mentioned earlier, the algorithms are applied to each train $\alpha \in \mathcal{T}$ in order to find useful sets of paths $\mathcal{P}^\alpha$.

The first method is what we call a *Penalized Dijkstra* algorithm. It works by finding a path using the classic Dijkstra's algorithm, adding it to the result set, and then increasing the cost of all edges along that path. This is then repeated until the desired number of paths have been found, giving a set of increasingly circuitous paths. The second method is the *Loop Erased Random Walk*, a known algorithm for generating uniformly distributed random paths through a graph. Both algorithms are shown in more detail in Appendix C.

If waypoints are desired, they may be handled by searching from the origin to the first waypoint, then from the first waypoint to the next, etc., and finally from the last waypoint to a destination. The final collection of paths is then the set product of the paths from each individual search.

[†]Generalized in the sense that we desire not a single "best" path but a diverse collection of potential paths.

5.2 VNS

To apply the VNS algorithms to the Pathing problem we have to adapt the algorithms. The main step of applying any VNS algorithm to a problem is to decide how the neighbourhood N should be generated. In our case we also need to decide when we should use $g(\hat{\xi})$ (or $\hat{g}(\hat{\xi})$). Due to these decisions there are minor changes to all the VNS algorithms, which can be found in Section C.1. Of note is that we will always use BestImprovement rather than FirstImprovement, since this expected to perform better together with restricted neighbourhoods (see below).

5.2.1 Neighbourhood Generation

Since evaluating g or $\hat{g}$ takes a lot of computing time we want to limit the amount of candidates that are tested. To achieve this we use restricted neighbourhoods—i.e., smaller neighbourhoods than the complete overarching neighbourhood which has been constructed—which only contain candidates that we are currently interested in. Our approach here is based on the work of M. Samà et al. in [7].

We define our overarching neighbourhood as the r *Trains Rerouted*[†] neighbourhood $\mathfrak{N}_{r\mathrm{TR}}(\xi, r)$, which is defined as the set of all possible candidates when $r = 1, \ldots, r_{\max}$ trains have been rerouted, and where $r_{\max} < |\mathcal{T}|$ is the maximum number of reroutes that are investigated. A single reroute from ξ to ξ' is defined as the replacement of a path $p^\alpha \to q^\alpha$ where $p^\alpha \in \xi$ and $q^\alpha \in \xi'$.

To construct a restricted neighbourhood $\mathfrak{N}_{\mathrm{RN}}(\xi, r, \mathcal{H}_h) \subseteq \mathfrak{N}_{r\mathrm{TR}}(\xi, r)$, with maximum L candidates, we use heuristic measures $\mathcal{H}_h \in \boldsymbol{\mathcal{H}}$ to evaluate whether or not we should reroute a train, $\mathcal{H}_h(\xi, \alpha)$. As we can have multiple heuristics we index them as $h = 1, \ldots, h_{\max} = |\boldsymbol{\mathcal{H}}|$. To relate both h and r to the normal neighbourhood index k we define it so that an increment in k corresponds to an increment in r until $r = r_{\max}$ where it rolls back to $r = 1$ and h is incremented once instead until $r = r_{\max}$ and $h = h_{\max}$. This means that

$$k = 1, \ldots, r_{\max}, \ldots, 2r_{\max}, \ldots\ldots, r_{\max}h_{\max}, \tag{5.1a}$$

$$r = r(k) = 1 + ((k-1) \mod r_{\max}), \qquad h = h(k) = \lceil k/r_{\max} \rceil . \tag{5.1b}$$

Restricted Neighbourhood construction

There are multiple ways to construct a restricted neighbourhood $\mathfrak{N}_{\mathrm{RN}}(\xi, r, \mathcal{H}_h)$. Here we do it by starting from a candidate ξ and generating $\xi' \in \mathfrak{N}_{\mathrm{RN}}$ new candidates using Algorithm 9, adapted from [7].

We start by sorting the paths $\mathcal{P}^\alpha$, storing them in `possiblePaths`, for each train α in ascending order with respect to how much they overlap

$$\|p^\alpha \cap q^\alpha\|^L := \sum_{m_i \in p^\alpha \cap q^\alpha} l_i \tag{5.2}$$

This means that the path that differs the most from the current path q^α is the first in the list and so on.

For each train $\alpha \in \mathcal{T}$ we also evaluate the `TrainScores` using the heuristic $\mathcal{H}_{h(k)}$. With this we construct a table `scoreTable` with L columns and $|\mathcal{T}|$ rows, where the rows are indexed

[†]Reroute here means taking a different path.

Calls	Index of replaced path		
0	0	0	0
1	0	0	1
2	0	1	0
3	0	1	1
4	0	2	0
5	0	2	1
6	1	0	0
⋮		⋮	
11	1	2	1
$\lvert P^\alpha \rvert$	2	3	2
α	0	1	2
Score	0.9	0.6	0.1

Table 5.1: An example over how repeated calls to `NextCandidate` behaves as counting, where we want to reroute 3 trains. Here the indices of the `possiblePaths` are shown against the count that `NextCandidate` has been called. Note how $|\mathcal{P}^\alpha|$ behaves as a mixed base.

using α. Each element in this table is set to the `TrainScores` divided by the row number. Then by using this we find the $r(k) \cdot L$ maximum values in `scoreTable` and count how many occur per train by storing them in `counts`. The `counts` now informs us how many times reroutes of each train can be used when constructing new candidates. This information is used to construct the L new candidates with r reroutes from the initial feasible candidate ξ.

To construct a candidate we select the r highest scored trains from `TrainScores`, with `Count` larger than 0, for rerouting, which we store in `SelectedTrains`. We then decrement `Count` for the selected trains to indicate that it has been used in a candidate.

Now given that we have selected the trains for rerouting we have to form a new unique candidate. Due to how we form `selectedTrains` it is possible to get the same train combination for several consecutive iterations, though not any other combination that has previously occurred. Therefore, to ensure uniqueness, the FirstCandidate function is used to create the initial candidate for a combination of `selectedTrains`, and NextCandidate is used to change the candidate whenever the train selection repeats.

The FirstCandidate simply reroutes each train α in `selectedTrains` and replaces current paths with the first path stored in `possiblePaths`$[\alpha]$. The `NextCandidate` behaves as counting with a mixed base, it takes the last solution ξ' and replaces (increments) the path for the lowest train score (the rightmost digit) with its next path in `possiblePaths`. If we have reached the end of `possiblePaths` for any train we instead roll over, by setting the path to the first in `possiblePaths`, and replacing the path of the train with the next lower train score. An example of this can be seen in Table 5.1.

Summarized in a somewhat more intuitive fashion, the approach described uses a heuristic to quantify how important it is that each train is rerouted. Trains that receive high scores are more likely to be rerouted, will be rerouted in more of the candidates, and are more likely to receive their first choice of path. That the paths are ordered by how different they are from the currently selected means that higher scoring trains are thus more likely to receive a significantly different path.

Algorithm 9 Build Restricted neighbourhood

A method to build a restricted neighbourhood N for $\mathcal{T}$ trains given a number of paths P with L number of elements using the specific neighbourhood $r(k)$ and the heuristic $h(k)$. This algorithm is adapted from [7].

Assume: $\mathcal{T}$: *Trains*
$\mathcal{P}^\alpha$ $\forall \alpha \in \mathcal{T}$:
$\mathcal{H}_h \in \boldsymbol{\mathcal{H}}$: *The set of train Heuristics to be used*
L: *Desired number of solutions*
$r(k), h(k)$: *Mappings from k to specific neighbourhood and heuristic*

Function BUILDRESTNEIGH(ξ, k)
Input: ξ: *The feasible candidate*
k: *Neighbourhood index*
Output: N: *A Restricted Neighbourhood of size L according to* $\mathcal{H}$

$\mathcal{H} \leftarrow\!\shortmid \mathcal{H}_{h(k)}$ ▷*The relevant heuristic for the given value of k*
For α in $\mathcal{T}$ **Do**
`possiblePaths`$[\alpha] \leftarrow\!\shortmid \mathcal{P}^\alpha$
Sort `possiblePaths`$[\alpha]$ in ascending order by $\|p^\alpha \cap q^\alpha\|^L$ where q^α is the path for α in ξ
$s \leftarrow\!\shortmid \mathcal{H}(\xi, \alpha)$ ▷*The score for train α according to the current heuristic*
`trainScores`$[\alpha] \leftarrow\!\shortmid s$
`scoreTable`$[\alpha, :] \leftarrow\!\shortmid (s/1, s/2, \ldots, s/L)$
End For

Identify the $r(k) \cdot L$ maximum entries in `scoreTable`
For α in $\mathcal{T}$ **Do**
`counts`$[\alpha] \leftarrow\!\shortmid$ number of these entries occuring in row `scoreTable`$[\alpha, :]$
End For

For $l = 1, \ldots, L$ **Do**
`selectedTrains` $\leftarrow\!\shortmid$ The set of $r(k)$ trains α with the highest `trainScores`$[\alpha]$ and `counts`$[\alpha] > 0$
Decrement `counts`$[\alpha]$ for all α in `selectedTrains`
If $l = 1$ or `selectedTrains` changed from previous iteration **Then**
$\xi' \leftarrow\!\shortmid$ FIRSTCANDIDATE(ξ, `selectedTrains`, `possiblePaths`)
Else
$\xi' \leftarrow\!\shortmid$ NEXTCANDIDATE(ξ', `selectedTrains`, `trainScores`, `possiblePaths`)
End If
Add ξ' to N unless ξ' is **null**
End For

Return N
End Function

Algorithm 10 First and Next Candidate
Two help function used to build the restircted neighbourood, see Algorithm 9.

Function FirstCandidate(ξ, `selectedTrains`, `possiblePaths`)
Input: ξ: *The feasible candidate*
`selectedTrains`: *The set of trains to be rerouted*
`possiblePaths`: *The collections of possible rerouting paths for each train*
Output: ξ': *The initial candidate based on* ξ

$\xi' \leftarrow \xi$
For α in `selectedTrains` **Do**
Change ξ' so that train α takes the first path in `possiblePaths`[α]
End For
Return ξ'
End Function

Function NextCandidate(ξ, `selectedTrains`, `trainScores`, `possiblePaths`)
Input: ξ: *The candidate to transform into the next candidate*
`selectedTrains`: *The set of trains to be rerouted*
`trainScores`: *The collection of scores for each train*
`possiblePaths`: *The collections of possible rerouting paths for each train*
Output: ξ': *The modified candidate*

If ξ = **null Then**
Return null
End If
$\xi' \leftarrow \xi$
For α in `selectedTrains`, ordered by increasing `trainScore`[α] **Do**
Change ξ' such that train α takes the next path in `possiblePaths`[α] compared to the current one in ξ
If there is no such path **Then**
Change ξ' to use the first path in `possiblePaths`[α]
Else
Return ξ'
End If
End For
Return null
End Function

5.2.2 Neighbourhood Heuristics

In order to find the `TrainScore` in Algorithm 9 we have to evaluate $\mathcal{H}(\xi, \alpha)$ to find the scores $s(\alpha) \forall \alpha \in \mathcal{T}$. An assumption used by the heuristics is that for each ξ a graph $\mathcal{G}_S$ exists. If it does not, $\mathcal{G}_S$ has to be created and solved using either g or $\hat{g}$. The heuristics may use different or modified schedules, with the same nodes $\mathcal{N}$, for comparison to the current $\mathcal{S} \in \mathcal{G}_S$. A common schedule to compare to is the *free net* schedule $\mathcal{S}_0$ where $\nexists a \in \mathcal{S}_0$, i.e. no alternative arcs are selected.

As these are heuristics it means that these can be constructed arbitrarily. Furthermore it is often hard to know which heuristic is good without testing. Here we deal with six different heuristics that we have constructed: Train Cost (TC), Adjusted Train Cost (ATC), Cascading Cost (CC), Waiting Train (WT), Free-Net Waiting Train (FNWT), and Random.

Train Cost (TC)

For TC to calculate the score the "partial objective function" $z^{\alpha}_{\text{partial}}$ is used to denote all contributions that the train α has on z:

$$s(\alpha) = z^{\alpha}_{\text{partial}}. \tag{5.3}$$

Adjusted Train Cost (ATC)

Where TC above computes the total contribution from train α, the Adjusted Train Cost removes from this a reasonable lower bound so as to measure the solution's excess cost for the train. The lower bound is defined as the $z^{\alpha}_{\text{partial}}$ value in the free net:

$$s(\alpha) = z^{\alpha}_{\text{partial}} - \left\{ z^{\alpha}_{\text{partial}} \,\middle|\, \mathcal{S}_0 \right\}. \tag{5.4}$$

Cascading Cost (CC)

The cascading cost is a measure of how much cascading effect the train has on the objective solution. To do this we find the difference between the objective function for the schedule and the schedule with all alternative arcs connecting to that train removed. This means that the train score is defined as

$$s(\alpha) = z - \left\{ z \,\middle|\, \mathcal{S} \text{ with all } a^{\alpha,\beta}, a^{\beta,\alpha} \text{ removed } \forall \beta \in \mathcal{T} \right\}. \tag{5.5}$$

Waiting Train (WT)

Another heuristic is to find the total waiting time for the trains. This is simply done by

$$s(\alpha) = C^P_{\alpha} \sum_{i=1}^{e} t^{\alpha p}_i - t^{\alpha p}_{i-1} - u^{\alpha p}_{i-1}. \tag{5.6}$$

Free-Net Waiting Train (FNWT)

FNWT measures the first order delays of the train. To do this we define a waiting node as a node n^p_i such that $t^p_i > t^p_{i-1} + u^p_{i-1}$. At each waiting node we measure the first order delay i.e.

between $\mathcal{S}_0$ and the schedule with arcs only connecting with the waiting node with the train priority included:

$$s(\alpha) = C_\alpha^P \sum_{\text{waiting nodes: } n_i^{\alpha p}} \ell^{\mathcal{S}_i}(n_0, n_i^{\alpha p}) - \ell^{\mathcal{S}_0}(n_0, n_i^{\alpha p}); \tag{5.7a}$$

$$\mathcal{S}_i = \mathcal{S}_0 \cup \left\{ a_{j,i}^{\beta q, \alpha p} \in \mathcal{S} \;\middle|\; t_i^{\alpha p} = t_j^{\beta q} + w_{j,i}^{\beta q, \alpha p} \right\}. \tag{5.7b}$$

Random

The random heuristic is a neutral heuristic that the others may be compared with. It is simply defined, somewhat arbitrarily, using a random number from a uniform distribution as

$$s(\alpha) = C_\alpha^P \cdot \text{Uniform}(1, 2). \tag{5.8}$$

5.3 Constructing Couplings

As stated in Subsection 4.2.3, to find a coupling between arc pairs it has to be proven that they cannot be chosen independently. Finding all of these relationships naively is time consuming, but we can exploit some of our problem structure to find an interesting subset very quickly.

Specifically, our alternative graphs almost exclusively contain strings of nodes connected by fixed arcs, representing the train paths, and arc pairs that each connect two paths. Given any alternative arc $a = (n_i^p, n_j^q) \in K$ from a node on path p to a node on path q, we can easily search along path q for other arcs $a' = (n_k^q, n_l^p) \in K'$ that connect back to p, and thus might together create a cycle. Due to our problem structure we know that the arc conjugates $a^* = (n_{i'}^q, n_{j'}^p) \in K^\star$ and $a'^\star = (n_{k'}^p, n_{l'}^q) \in K'^\star$ might similarly create a cycle in the opposite direction.

Then, we check if the selection of a and a' or a^* and $a'^\star$ *both* result in positive cycles, i.e.,

$$w_{ij}^{\alpha p,\beta q} + \ell^{\mathcal{S}_0}(n_j^q, n_k^q) + w_{kl}^{\beta q,\alpha p} + \ell^{\mathcal{S}_0}(n_l^p, n_i^p) > 0 \tag{5.9a}$$

and

$$w_{i'j'}^{\beta q,\alpha p} + \ell^{\mathcal{S}_0}(n_{j'}^p, n_{k'}^p) + w_{k'l'}^{\alpha p,\beta q} + \ell^{\mathcal{S}_0}(n_{l'}^q, n_{i'}^q) > 0. \tag{5.9b}$$

Note that the makespans here operate on the so called free net schedule $\mathcal{S}_0$ (see previous subsection). If these inequalities hold[†] then we know that $a \rightleftharpoons a'^\star$ and $a^\star \rightleftharpoons a'$ since we have just demonstrated that the alternative is infeasible (cf. Figure 4.4). We can thus combine the couplings and create $K'' = K \cup K'^\star$ and $K''^\star = K^* \cup K'$, replacing the four "older" couplings.

[†]If any of the makespans ℓ are undefined (e.g., if $j > k$) then there can be no cycle, and the inequality is considered false.

Chapter 6

Implementation and Tests

The theory, model, and methodology described so far were implemented as a proof-of-concept C++ library at Bombardier RCS during 2019. The implementation is capable of reading descriptions of rail networks, trains, complications, etc., and using these to solve the given PCR problem.

MILP problems generated during execution are converted and handed over to external solvers. During our implementation and research work the solver used was primarily Gurobi (version 9.0.0, under an academic license), but we also integrated with Coin-OR Branch and Cut (CBC, an open source solver), as well as two versions of our own solver using Benders' decomposition (with Gurobi solving the intermediate problems).[†]

The implementation aims to never compute the same problem twice, and will cache graphs, problems and results. However, due to memory constraints some results must be evicted and might therefore need to be solve anew. In practice though, this was observed to be very rare as only the worst results were evicted, and VNS is unlikely to visit the same *bad* candidate repeatedly.

Features that were deemed interesting but non-essential were implemented and verified to work as expected, but not extensively researched or included in the results. This includes various extra penalties and costs mentioned in previous chapters, and the contents of Appendix A and Appendix B.

The implementation of train length and moving block was considered extremely important with regards to practical applications, but uninteresting with regards to the model and results. Both features were therefore verified and tested, but not included in the next chapter. See Chapter 8 however.

Tests and experiments were run on a close approximation of a large real world network,[‡] illustrated in Figure 6.1. In the experiments run to produce results for Chapter 7, a set of trains were started on random tracks in the top left section of the illustration, and an equivalent number of randomized trains going the other way (meaning that runs with, for example, 16 trains included eight trains going from loading to unloading, and eight trains going from unloading to loading). The trains received random lengths between 0.5 km and 1.5 km (comparable to the mean track section length), random maximum speeds between 60 km/h and 100 km/h (in practice limited to maximum 80 km/h by the network), and random starting

[†]Also investigated were integrations with the MOSEK and XPRESS solvers, but these were never implemented due to time constaints in the project. The CPLEX solver was also looked at but, unlike the others, documentation, academic licenses, and binaries were not readily available.

[‡]The owner of which has approved of its use and illustration here, but does not wish to be named.

times between $t = 0$ and $t = 10$ hours (compare with the ≈15 hour travel time). For repeated runs the trains were re-randomized.

The baseline, default, scenario was with eight trains, running the General-VNS algorithm, using the Cascading Costs (CC) heuristic, and using Gurobi to solve MILP problems with a five minute time limit. Other scenarios were then created by changing some parameter of this baseline case in order to compare the results.

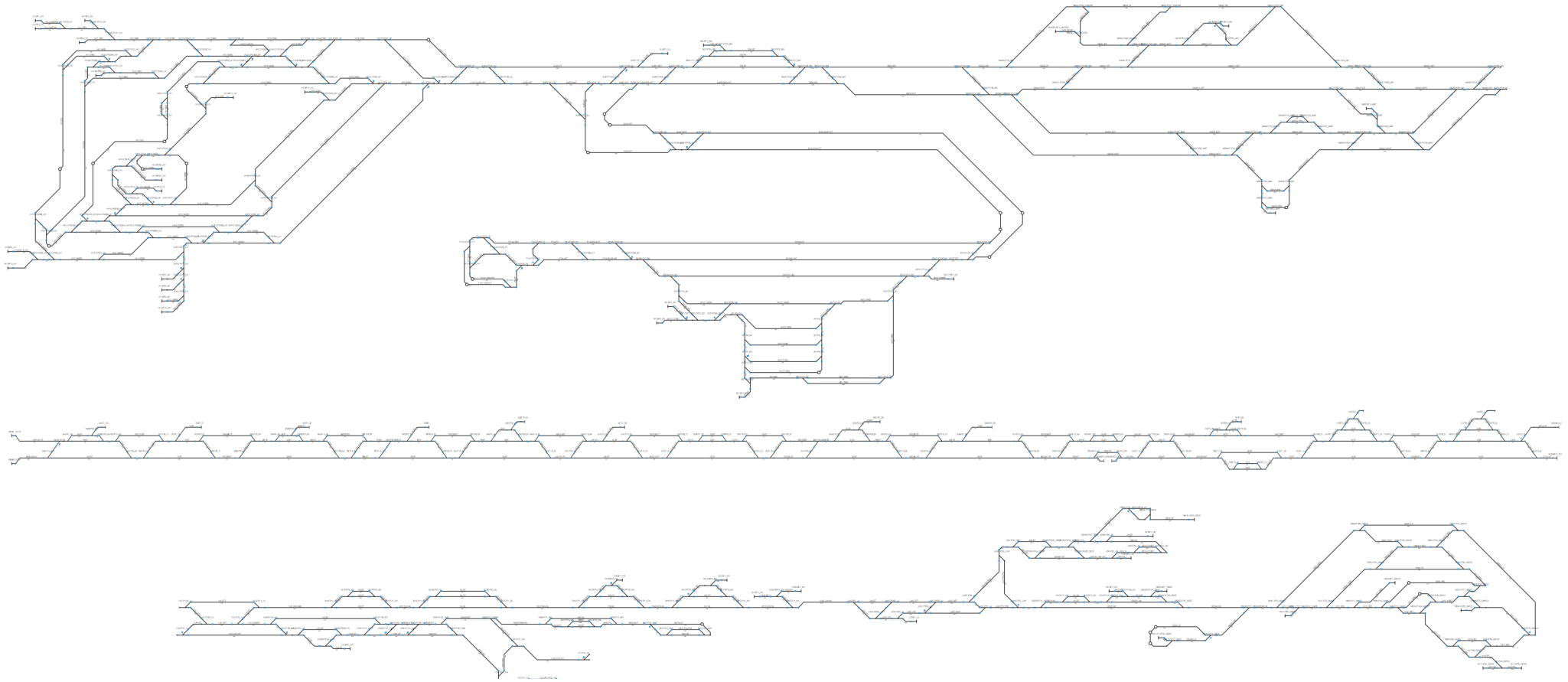


Figure 6.1: Logical (not to scale) schematic of the test network. The top section is an unloading and staging region while the bottom section is a loading region. The middle section is a long "mainline" region that connects the two (top right connects to middle left, middle right connects to bottom left). The network contains 1 164 junctions, 1 320 track sections, and ≈1 180 km of rail in total.

The network is very close to an existing network but some approximations are made since this version is built on somewhat incomplete data. In particular, all speed limits are 80 km/h and fouling distances at junctions are much shorter than they would be in reality.

Chapter 7

Results

The results presented in this chapter where obtained using our implementation, with MILP problems solved by Gurobi 9.0.0. The system used was a Windows 10 laptop with 16 GB memory, a i7-8850H processor at 2.60GHz with six Cores. The code was compiled using MSVC 19.22.27905 for x86 in a Release configuration.

Example timelines of single runs with 8, 10, 16, 20, and 30 trains are shown in Figure 7.1, illustrating how the VNS search progresses. The approximate MILP problem sizes generated in these runs are shown in Table 7.1. The number of continuous variables is approximately proportional to the number of trains, while the number of inequality constraints is approximately quadratic, and the number of binary variables is between quadratic and cubic.

The performance of the MILP solver for different numbers of trains is shown in Figure 7.2, illustrated as the proportion of solve runs that reach a 5% or 0.5% gap over time. The gap for a given objective value z is, in this case, defined compared to the eventually found optimal value z^* per problem, i.e., $(z - z^*)/z^*$. With up to 10 trains, the problems are solved to optimality within seconds. With 16 trains the solver still generates good solutions within seconds, and while finding an optimal solution takes longer time it still does so within the five minute time limit. Even for 20 trains good solutions are found quickly, but reaching optimality takes considerably longer and only $\approx 60\%$ of our solve attempts reach the 0.5% gap threshold (in the remaining cases, the best upper bound is used as a close approximation of z^* when computing the gap). For 30 trains the time to find *any* solution starts to become an issue, and not enough 30 train runs could be completed in time to produce any relevant results.

Our method of generating couplings was extensively tested and Figure 7.3 shows the impact it has on the number of variables. Specifically, we found that couplings reduced the number of binary variables by a factor of ten. Runtimes and results were however not compared as the MILP solvers almost always failed to find *any* feasible solution when couplings were not used (except for instances that were too small to be of interest).

The six different neighbourhood heuristics laid forth in Subsection 5.2.2 were compared, using the final gap at the end of each VND run as a metric for how well each heuristic "guided" the VND to better neighbourhoods. The gap values were compared using pairwise permutation tests, and the results are shown in Table 7.2. Interestingly, only the CC heuristic was significantly better than random values. The TC and ATC heuristics were both comparable with random values, and FNWT and WT were both significantly worse.

The four different VNS algorithms included here were compared by repeatedly running the same scenario using each algorithm in turn. For this comparison ten trains were used instead of eight in order to strike a better balance between resolution and computation time. The mean objective values over time per algorithm is shown in Figure 7.4. The spread of individual runs

is too large to give significant results, though FH-VNS has consistently tended towards being (very) marginally better over time.

Finally, we ran some comparisons between the Gurobi and CBC solvers by giving them identical MILP problems as generated by our implementation. For problem instances where both solvers solved to optimality we found Gurobi to be ≈15–20 times faster. However, with even moderately large problems (>10 trains) CBC often failed to find a solution, and time constraints precluded a more extensive comparison.

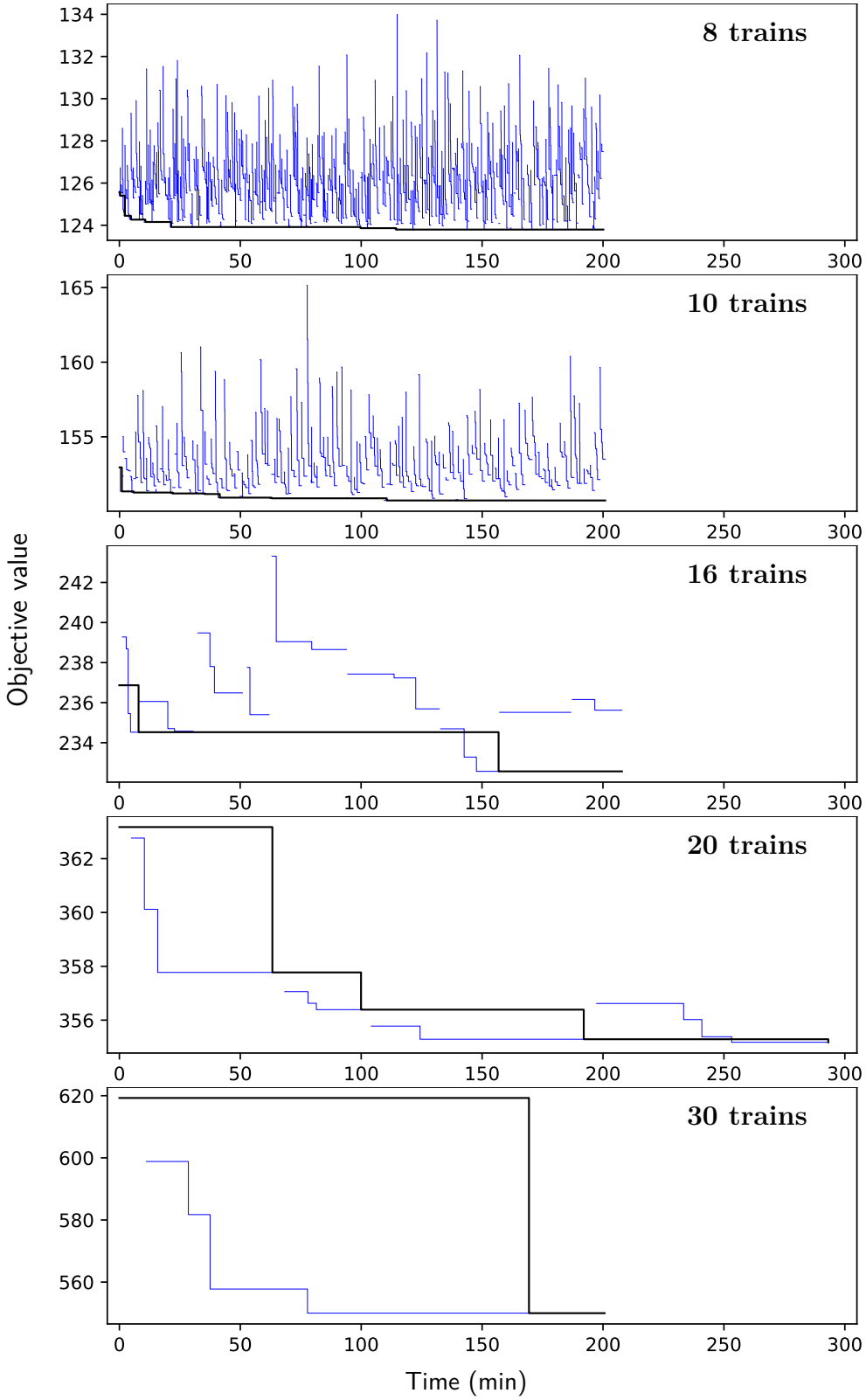


Figure 7.1: Example runs for different numbers of trains. The black lines show the progress of the General-VNS runs, and each blue line represents a single VND run invoked by General-VNS.

Trains	4	8	10	16	20	30
Continuous variables, t	800	1 600	2 100	3 400	4 200	6 500
Binary variables, x	50	230	380	1 200	1 800	4 300
Inequality constraints	1 800	6 900	11 000	29 000	46 000	100 000

Table 7.1: Approximate sizes of the MILP problems generated when scheduling various numbers of trains. The exact counts vary, but the trend is that a doubling of the trains leads to a doubling of the continuous variables (linear as expected), a sixfold increase in binary variables, and a four-fold increase in inequality constraints.

Heuristic		CC	Random	TC	ATC	FNWT	WT
Mean gap (%)		1.03%	1.31%	1.37%	1.39%	1.51%	1.63%
N		1114	1059	1120	1172	1089	867
P-value vs.	CC		<.0001	<.0001	<.0001	<.0001	<.0001
	Random			0.13	0.06	<.0001	<.0001
	TC				0.68	<.01	<.0001
	ATC					<.01	<.0001
	FNWT						0.03

Table 7.2: Comparison of train heuristics for VNS neighbourhood construction. Data was collected from six different runs, each using a different heuristic, on the same 8 trains scenario (cf. the top panel of Figure 7.1). The best found solution was assumed optimal, and the gap at the end of each individual VND run was computed. Pairwise tests between heuristics were estimated by Monte Carlo permutation tests (50 000 samples). Results show that our CC heuristic performs best by a significant margin; the Random, TC, and ATC heuristics are about equally efficent; and the FNWT and WT heuristics perform significantly worse.

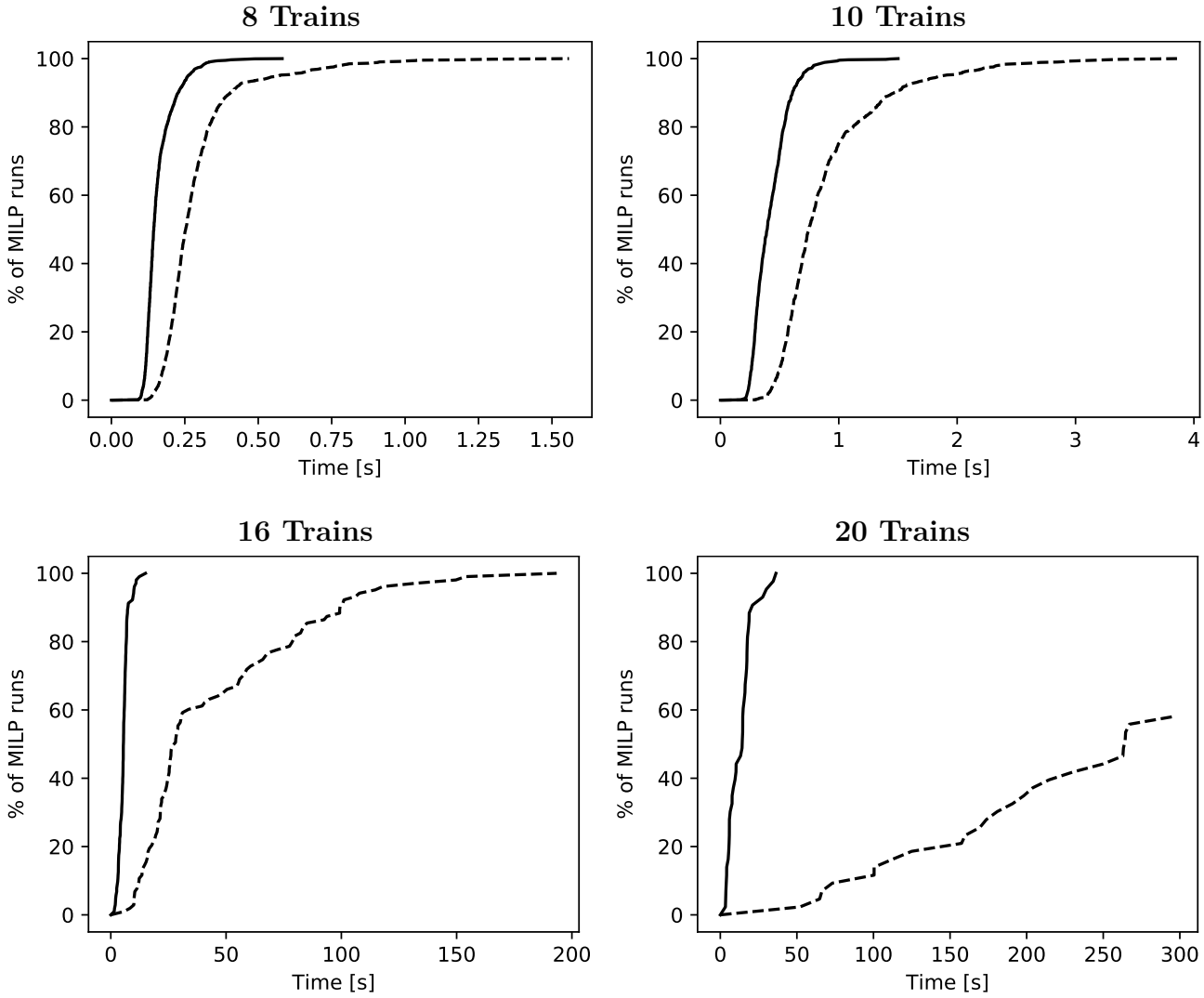


Figure 7.2: MILP progress over time for different numbers of trains. The plots are based on all the "full" solve attempts of MILP problems during four different VNS runs (same runs as in Figure 7.1, excluding the 30 trains run). The solid lines indicate the proportion of solve attempts which have reached a 5% gap at any point in time, while the dashed lines indicate the proportion of attempts which have reached a 0.5% gap (see text for definition).

For problem instances with 8 or 10 trains optimal solutions where quickly found and verfied. For instances with 16 trains "good" solutions are quickly reached, but it takes considerably longer time to reach and prove optimality. For instances with 20 trains good solutions are still found quite quickly but it takes even longer time to lower the gap, and only $\approx 60\%$ reach the 0.5% gap before the runtime limit.

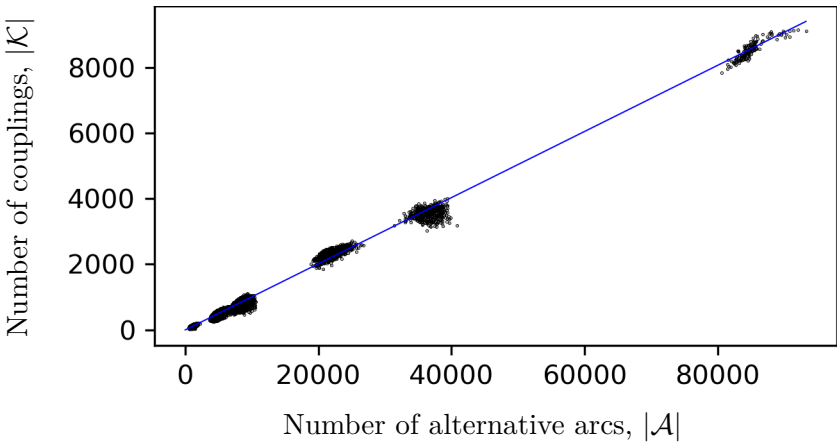


Figure 7.3: Reduction of choices using couplings. Plotted are the number of alternative arcs versus the number of couplings for all MILP problems constructed during the runs from Figure 7.1 (as well as a four train run). The blue line is a linear fit, with slope ≈ 9.9. This means that our MILP instances contain ten times fewer binary variables than if couplings were not implemented.

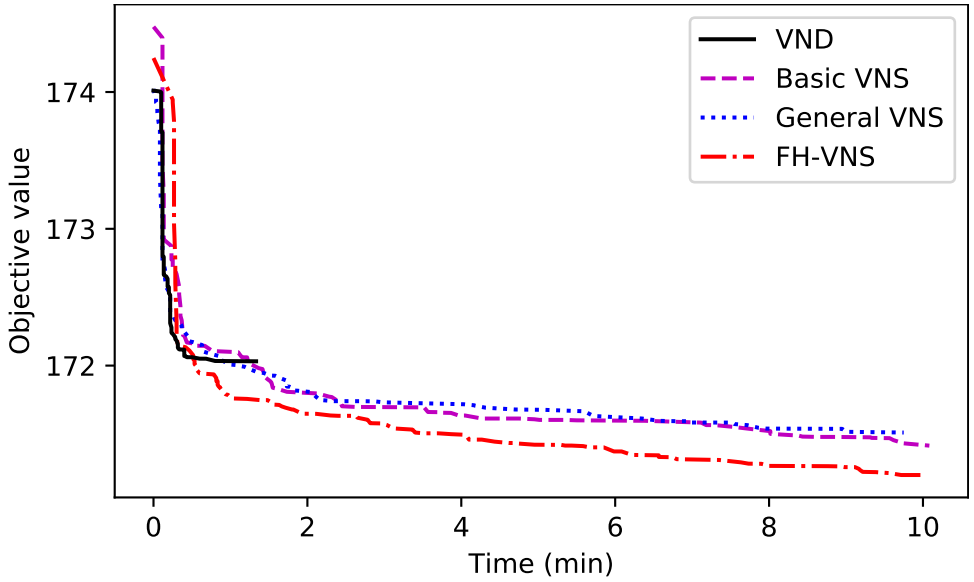


Figure 7.4: Comparison of objective values between the four different VNS algorithms versus computing time. The lines show the mean objective value for 20 runs per algorithm (50 for VND). All runs for all algorithms use the same 10 train scenario and employ the CC heuristic. The spread between individual runs is relatively large so none of the methods can be said to be significantly different, but we have repeatedly observed the same trend of FH-VNS being marginally better over time. It is important to note here that the vertical axis does not begin at zero, and the differences between algorithms is *very* small.

Chapter 8

Discussion and Further Work

Our proof-of-concept implementation and the results produced by it, presented in the previous chapter, show that the model and methodology described in this thesis succeed in scheduling a decent number of trains in a large network within acceptable computing time. What is meant by "a decent number", "large" and "acceptable time" is, however, subject to debate. In our case, the upper limit of ≈ 30 trains is comparable to the expected number of concurrently running trains on the real world equivalent of our test network. The network that our experiments were run on can be objectively stated to be one of the largest industrial rail networks in the world. National passenger and cargo networks may be larger, but the scheduling of such traffic is different in nature due to different and varying goals and constraints. For this work and thesis, "acceptable time" has generally meant 5–15 minutes to solve a single MILP problem instance, and 3–4 hours for a full run.

8.1 Performance

To be useful in practice, the implementation need not only find solutions but do so quickly and efficiently. Our results were achieved running on a general purpose laptop and not a computer designed for these kinds of workloads. Additionally, while the Gurobi solver does some of the work in parallel, our implementation is wholly single threaded. However, the neighbourhood search of the VNS algorithms is close to embarrassingly parallel and an implementation utilizing this should be able to massively reduce the wall clock time taken.

The MILP problems generated here are quite small compared to what may be encountered in, for example, the field of large scale optimization. On the other hand, our problems contain a large proportion of highly interacting binary variables, and are thus inherently difficult to deal with. Our implementation of couplings, in order to reduce the number of binary variables the MILP solver has to consider, has proved critical in enabling the solvers to find solutions at all, let alone quickly. Since our approach to generating the couplings is somewhat simplistic, we expect that further work may improve it and thus further reduce the number of binary variables.

To us, the problem structure appears to lend itself to Benders' decomposition. This was attempted twice. However, in our case we found that Gurobi's own logic far outperformed our own implementations, but further work would be needed to properly develop, test, and evaluate Benders' decomposition methods for these problems.

One method currently used in some train control systems is to divide the network into different smaller regions that function independently, preferably with very limited interaction, and to negotiate transfers between regions. This would mean that every individual scheduling

problem is smaller and involve fewer trains, but trains entering the region cannot be fully taken into account and optimized for. The optimum found in each region thus only applies to that region, unless additional work is put in to coordinate regions in an additional layer of optimization.

Another way to improve performance is to not schedule all trains at the same time. We consider our test setup to be a sort of worst-case scenario, where all trains need to be completely scheduled simultaneously. In practice one might expect some amount of steady state operation, where most trains already have a path and schedule they are currently travelling. In such cases, trains close to their destinations are easier to schedule, and are in fact unlikely to need rescheduling at all. In general, it is possible to consider some trains as "fixed" and only schedule the remainder. These fixed trains would generate constraints to other trains, but no binary choice variables which casue the main performance drain.

8.2 Generality

Of the different neighbourhood heuristics presented and investigated, only the CC heuristic proved to be better than random chance. However, we do not believe this should be assumed to apply to all other scenarios and networks as well. Though, due to time constraints, other networks (and, in particular, other classes of network structures) were not investigated, we expect that the behaviour of the heuristics depend on the specific network structure and scenario involved.

Similarly, while we in our case found very little difference between different VNS algorithms, it is unclear whether or not this is the case in general. There also exists many more variants of VNS algorithms and adjustments that may be made.

8.3 Applicability

For purposes of applying a model to a real world problem, it is important to investigate if and to what extent the model matches the real world. For this thesis this was never practically doable, since we cannot simply borrow a rail network and run tests, and producing a suitably realistic simulation would be a completely different thesis.

As mentioned in Chapter 6, train length and moving block were both considered critical features. The reason they are not included in Chapter 7 is that they, in the end, do not significantly affect the results. Train length does affect how trains interact at junctions, as desired, but the effect on the objective value in our experiments is minimal. Moving block, similarly, changes how trains interact in the desired manner, but again has little effect on the total objective value.

One reason that train length and moving block have little effect on the object value, is that the model does not include acceleration and braking times and distances. There are some ways that this may be improved. For example, for any given train and path, the acceleration at the beginning and braking at the end may be estimated and used to impose upper bounds on the speed for those particular sections, since they are static and do not depend on any choice variable. Additionally, it should be possible to discretize the speed of each train on each track and impose constraints on how trains transition between speeds across tracks. On the other hand, this would involve a lot more work and create many more variables in the corresponding MILP problems, though column generation [10] may be suitable to deal with that aspect.

Even if the model and MILP instances are not adapted to include acceleration and braking, there are various approaches to consider. We expect that acceptable results could be achieved by estimating acceleration and braking curves and adjusting the schedule thereafter. In particular, one of the critical features is that deadlocks are avoided and for that purpose only the arc choices need to be taken into account, i.e., the ordering of trains at any given point. As long as the train orderings are maintained the schedule may be freely[†] adjusted without compromising the no-deadlock constraint.

Finally, real world networks can contain a lot of extra rules regarding scheduling and routing. At least some of these have been explored in Appendix A. These "complications" can be added to specific locations or trains in order to control the solutions and adhere to real world requirements. More work is needed to investigate how these affect feasibility, performance, and results in practice.

[†]Within common sense limits.

Chapter 9

Conclusion

We have developed a working model, solution methodology, and implementation for solving the PCR problem. Furthermore, we have demonstrated that good results may be achieved in reasonable time. Importantly compared to previous work is that our model and implementation still work when the train length is considered and when applying a moving block adjustment. This means that a method has been found such that it is possible to find the good solutions both in cases where some track segments are much shorter or much longer than the trains themselves.

Most significantly we have shown that for an alternative graph it is possible to significantly improve performance by finding couplings, exploiting that the problem structure means that many choices are related. In our work we specifically exploit the fact that two trains cannot change the order in which they are traveling on a continuous track.

Furthermore, the performance of the MILP solver is critical to the overall performance of the optimization, and we find that the commercial solver Gurobi far outperforms the open source CBC solver.

Appendix A

Complications

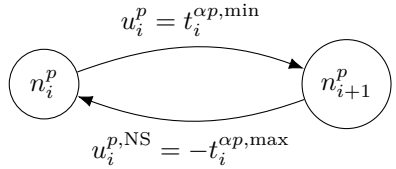
Additional complications, rules, and penalties may be modelled during scheduling as long as they can be represented in the alternative graph. We here present two types of complications that may be useful in the context of train scheduling.

A.1 Stop Rules

Stop rules here refer to additional rules that may be imposed on trains, specifying where and how the trains may or may not stop. We divide these into four types: No Stop and Forced Stop, so called fixed rules; as well as Penalized Stop and Restricted Stop, so called alternative rules.

No-stop

The first fixed rule is No Stop (NS), defined as a section of track in which the train may not stop. For a given node n_i^p followed by n_{i+1}^p, the regular fixed arc imposes the condition $t_{i+1}^p - t_i^p \geq t_i^{\alpha p,\min}$, i.e., a minimum time required to traverse n_i^p. The No Stop rule adds a condition $t_{i+1}^p - t_i^p \leq t_i^{\alpha p,\max}$, which instead enforces a maximum time the train is allowed to spend in the node. If this time is made to correspond to some positive minimum speed this is equivalent to forbidding a stop. In the graph, this condition is represented as a negated and reversed fixed arc between the nodes, as illustrated here:



$$t_{i+1}^p - t_i^p \geq u_i^p \quad \wedge \quad t_{i+1}^p - t_i^p \geq u_i^{p,\mathrm{NS}} \tag{A.1a}$$

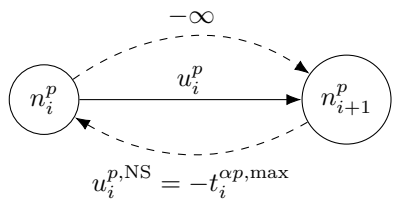
$$\Leftrightarrow \quad t_i^{\alpha p,\min} \leq t_{i+1}^p - t_i^p \leq t_i^{\alpha p,\max} \tag{A.1b}$$

Forced-Stop

Opposite to the No Stop rule is the Forced Stop (FS). This is done simply by increasing the weight of the fixed arc between node n_i^p and its successor. Instead of $u_i^p = t_i^{\alpha p,\min}$ we use a weight $u_i^{p,\mathrm{FS}}$ equal to the minimum traversal time, plus the time required to break, stop, and accelerate the train, plus any desired stop duration.

Penalized Stop

The first alternative rule is the Penalized Stop (PS), where a train may stop at a given node n_i^p but will incur a penalty to the objective function if it does. This is done by adding an alternative arc with weight equal to the No Stop rule, paired with an arc that is trivially fulfilled, an example of which is illustrated here:



$$t_{i+1}^p - t_i^p \geq u_i^p \tag{A.2}$$

$$t_{i+1}^p - t_i^p \leq t_i^{\alpha p,\max} \ \vee \ t_{i+1}^p - t_i^p \geq -\infty \tag{A.3}$$

,

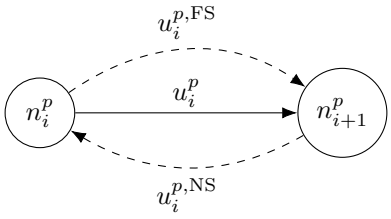
In the MILP formulation the trivial arc may be discarded, leaving

$$t_{i+1}^p - t_i^p \geq u_i^{p,\mathrm{NS}} - M x_i^{p,\mathrm{S}}, \tag{A.4}$$

where $x_i^{p,\mathrm{S}}$ is a binary choice variable which is one if the No Stop option is not followed. A non-negative term including this choice variable is added to the objective function to penalize that case.

Restricted stop

A Restricted Stop (RS) is similar to a Penalized Stop as above but where instead of a trivially fulfilled arc (i.e., the $-\infty$ arc), we use a Forced Stop arc instead:



In this case the MILP formulation must include the constraints of both the alternative arcs

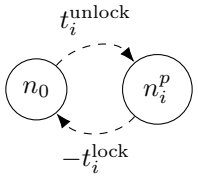
$$t_{i+1}^p - t_i^p \geq u_i^{p,\mathrm{NS}} - M x_i^{p,\mathrm{S}} \tag{A.5a}$$

$$t_{i+1}^p - t_i^p \geq u_i^{p,\mathrm{FS}} - M(1 - x_i^{p,\mathrm{S}}) \tag{A.5b}$$

with the choice variable $x_i^{p,\mathrm{S}}$ specifying which constraint applies, and with a penalty term as with Penalized Stop.

A.2 Timed Lock

A regular occurrence in rail networks are temporary restrictions on certain areas (e.g., for maintenance) such that a train may not enter during a time period $[t_i^{\text{lock}}, t_i^{\text{unlock}}]$. We call this a *Timed Lock* and can enforce this by creating an arc pair between the zero node n_0 and the relevant node n_i^p:



$$t_i^p \leq t_i^{\text{lock}} + M x_i^{\text{lock}} \tag{A.6}$$

$$t_i^p \geq t_i^{\text{unlock}} - M(1 - x_i^{\text{lock}}). \tag{A.7}$$

The arc pair, and the choice variable x_i^{lock}, thus represents the alternatives of trains entering before the lock begins or after the lock ends.

Appendix B

General Objective function

If we start from the MILP Alternative graph problem

$$\text{minimize} \quad z = \sum_{i=1}^{n} c_i t_i + \sum_{(j,k),(i,l)\in\mathcal{A}} \tilde{c}_{j,k,l,i}\ x_{j,k,l,i} + \tilde{c}_{l,i,j,k}\ (1 - x_{j,k,l,i})$$

subject to

$$t_j - t_i \geq u_{ij} \qquad \forall\ (i,j) \in \mathcal{F}, \qquad \text{(fixed arcs)}$$

$$\left.\begin{array}{l} t_k - t_j \geq w_{jk} - M x_{j,k,l,i} \\ t_i - t_l \geq w_{li} - M(1 - x_{j,k,l,i}) \\ x_{j,k,l,i} \in \{0,1\} \end{array}\right\} \quad \forall\ (j,k),(i,l) \in \mathcal{A} : (j,k) = (i,l)^*, \quad \text{(alternative arcs)}$$

$$\begin{array}{ll} t_0 = 0, & \\ t_i \geq 0 & \forall\ i > 0, \end{array}$$

then it is possible to add abstract objective variables $\gamma_r,\ r \in \{1,\ldots,R\}$ to the objective function. It is important when adding these that they don't affect the feasibility of the problem and that it does not lead to the objective value becoming unbounded. To do this we add $\sum_{r=1}^{R} \hat{c}_r \gamma_r$ to the objective function, where $\hat{c}_r > 0$, and introduce a bounding constraint $\gamma_r \geq 0$. As each γ_r has to relate to the rest of the variables in the Alternative graph we also introduce E number of constraints

$$\gamma_r \geq \sum_{i=1}^{n} h_i^{e,r} t_i + \sum_{(j,k),(i,l)\in\mathcal{A}} \tilde{h}_{j,k,l,i}^{e,r}\ x_{j,k,l,i} + \tilde{h}_{l,i,j,k}^{e,r}\ (1 - x_{j,k,l,i}) \quad \forall e \in \{0,\ldots,E_r\} \tag{B.1}$$

where $h^{e,r}, \tilde{h}^{e,r}$ are constants for the eth constraint on γ_r. Here it is important that each γ_r constraint does not have any other γ_r terms. This means that the introduction of γ_r does not affect the feasibility of the problem.

The resulting problem can then be stated in the form

$$\text{minimize} \quad z = \sum_{i=1}^{n} c_i t_i + \sum_{\substack{a,a^* \in \mathcal{A} \\ a=(j,k) \\ a^*=(i,l)}} \tilde{c}_{j,k,l,i}\ x_{j,k,l,i} + \tilde{c}_{l,i,j,k}\ (1 - x_{j,k,l,i}) + \sum_{r=1}^{R} \hat{c}_r \gamma_r$$

subject to

$$t_j - t_i \geq u_{ij} \qquad \forall\ (i,j) \in \mathcal{F},$$

$$\left.\begin{aligned} t_k - t_j &\geq w_{jk} - M x_{j,k,l,i} \\ t_i - t_l &\geq w_{li} - M(1 - x_{j,k,l,i}) \\ x_{j,k,l,i} &\in \{0,1\} \end{aligned}\right\} \qquad \forall\ (j,k),(i,l) \in \mathcal{A} : (j,k) = (i,l)^*,$$

$$\begin{aligned} \gamma_r \geq & \sum_{i=1}^{n} h_i^{e,r} t_i \\ & + \sum_{\substack{(j,k),(i,l) \in \mathcal{A}: \\ (j,k)=(i,l)^*}} \tilde{h}_{j,k,l,i}^{e,r}\ x_{j,k,l,i} + \tilde{h}_{l,i,j,k}^{e,r}\ (1 - x_{j,k,l,i}) \end{aligned} \qquad \begin{aligned} &\forall e \in \{1,\ldots,E_r\} \\ &\forall r \in \{1,\ldots,R\}, \end{aligned}$$

$$\gamma_r \geq 0 \qquad \forall r \in \{1,\ldots R\},$$

$$t_0 = 0$$

$$t_i \geq 0 \qquad \forall\ i > 0.$$

It is important, in regards to the solution methodology, that changes to the objective function probably means that heuristics found in Subsection 5.2.2 are not representative to the problem anymore. Furthermore it is also important to note that if this new variable is not minimized (or stays constant, i.e. $h_i^{e,r} = 0$) then the makespan might not give the correct times in respect to the objective function.

A practical example of what could be added is a delay term γ to the objective function with the constraints $\gamma \geq t_j - t_i - u_{ij}\ \forall (i,j) \in \mathcal{F}$. For this example when taking the makespan it also minimizes the γ and most of the heuristics still have meaning if γ is included in the heuristics.

Appendix C

Algorithms

C.1 Changes to VNS functions

The following algorithms are a modification of the algorithms in 3.3 so that they can be used in 5.2.

Algorithm 11 Best Improvement
Within a neighbourhood N find the best improvement $\xi' \in N$ compared to ξ such that $\hat{g}(\xi') \leq \hat{g}(\xi)$. If none is found return ξ.

Assume: $\hat{g}$: *objective function, where* $z = \hat{g}(\bullet)$
Require: $\hat{g}(\text{unsolvable}) = \infty$:

Function BestImprovement(ξ, N)
    Input: N: *The neighbourhood to be evaluated*
    Output: ξ: *The new solution*

    For ξ' in N **Do**
        If $\hat{g}(\xi') < \hat{g}(\xi)$ **Then**
            $\xi \leftarrow\!\shortmid \xi'$
        End If
    End For

    Return S
End Function

Algorithm 12 Shaking

Given a solution ξ generate a random solution ξ' from the $\mathfrak{N}_{r\mathrm{TR}}(\xi, r)$ neighbourhood with $r(k)$ random trains rerouted.

Assume: $\mathcal{T}$: *Trains*
  $\mathcal{P}^{\alpha}$: *Found by path search*
  $r(k)$:

Function SHAKE(ξ, k)
  Input: ξ: *The current candidate*
    k: *The neighbourhood to be rerouted*
  Output: ξ: *The new candidate*

  $A \leftarrow \varnothing$
  For $i = 0, \ldots, r(k)$ **Do**
    $\alpha \leftarrow$ random train $\alpha \in \mathcal{T} \setminus A$
    Add α to A ▷*So that a train is not rerouted twice*
    $p \leftarrow$ random $p \in \mathcal{P}^{\alpha}$
    Reroute train α with route p in ξ
  End For
  Return ξ
End Function

Algorithm 13 VND (Variable Neighbourhood Decent)

The modified verison of the VND algorithm, see Algorithm 5, for the PCR problem. It has been modified to use the restricted neighbourhood, see Algorithm 9, to investigate the search space.

Assume: $T_{\max}$: *The maximum computation time*
Require: $r_{\max}, h_{\max}$: *Corresponding to* k_{max}

Function VND(ξ, $k_{\max}$)
  Input: ξ: *The initial candidate*
    $k_{\max}$: *The maximum reroutes per heuristic*
  Output: ξ: *The current best solution*

  Repeat
    $k \leftarrow 1$; $\xi'' \leftarrow \xi$
    Repeat
      $N \leftarrow$ BUILDRESTNEIGH(ξ, k) ▷*This is* $\mathfrak{N}_{RN}(\xi, r, \mathcal{H}_h)$
      $\xi' \leftarrow$ BESTIMPROVEMENT(ξ, N)
      $(\xi, k) \leftarrow$ NEIGHBOURHOODCHANGE(ξ, ξ'', k)
    Until $k = k_{\max}$
    $T \leftarrow$ CPUTIME( )
  Until $T > T_{\max}$
  Return ξ
End Function

Algorithm 14 General VNS

The modified version of the General VNS algorithm, see Algorithm 7, for the PCR problem. It has been modified to handle the possibility that ξ' isn't necessarily a feasible candidate. It also changed to use the modified the shake, Algorithm 12, and modified VND, algorithm 13.

Assume: $k_{\max}$: *The maximum amount of investiaged neighbourhoods*
$k'_{\max}$: *The maximum amount of investiaged neighbourhoods for the VND*
$\mathfrak{N}$: *The neighbourhood structures*
$T_{\max}$: *Maximal runtime*
Require: $r_{\max}$, $h_{\max}$: *corresponding to* k_{max}
$r'_{\max}$, $h'_{\max}$: *corresponding to* k'_{max}

Function GVNS(ξ)
Input: ξ: *The initial feasible solution*
Output: ξ: *The best found feasible solution*

Repeat
$k \leftarrow 1$
Repeat
$\xi' \leftarrow$ SHAKE(ξ, k) ▷*This is* SHAKE($\mathfrak{N}_{r\,TR}(k, r)$) *for* $r(k)$
If ξ' is infeasible **Then Continue**
$\xi'' \leftarrow$ VND(ξ', $k'_{\max}$)
$(\xi, k) \leftarrow$ NEIGHBOURHOODCHANGE(ξ, ξ', k)
Until $k = k_{\max}$
$T \leftarrow$ CPUTIME( )
Until $T > T_{\max}$
Return ξ
End Function

Algorithm 15 Basic VNS

The modified version of the Basic VNS algorithm, see Algorithm 6, for the PCR problem. It has been changed to use the restricted neighbourhood, Algorithm 9, and the modified shake, Algorithm 12, to investigate the search space. It also takes in consideration that ξ' isn't necessarily a feasible candidate.

Assume: $k_{\max}$: *The maximum amount of investigate neighbourhoods*
$\mathfrak{N}$: *The neighbourhood structures*
$T_{\max}$: *The maximal runtime*
Require: $r_{\max}, h_{\max}$: *Corresponding to* k_{max}

Function BVNS(ξ)
Input: ξ: *The initial feasible solution*
Output: ξ: *The best found feasible solution*

Repeat
$k \leftarrow 1$
Repeat
$\xi' \leftarrow$ SHAKE(ξ, k) ▷*This is* SHAKE($\mathfrak{N}_{r\,TR}(k, r)$)
If ξ' is infeasible **Then Continue**
$N \leftarrow$ BUILDRESTNEIGH(ξ', k) ▷*This is* $\mathfrak{N}_{RN}(\xi', r, \mathcal{H}_h)$
$\xi'' \leftarrow$ BESTIMPROVEMENT(ξ', N) ▷*Changed from* FIRSTIMPROVEMENT
$(\xi, k) \leftarrow$ NEIGHBOURHOODCHANGE(ξ, ξ', k)
Until $k = k_{\max}$
$T \leftarrow$ CPUTIME
Until $T > T_{\max}$
Return ξ
End Function

Algorithm 16 FH VNS

The modified version of the FH VNS algorithm, see Algorithm 8, for the PCR problem. It has been changed to use the restricted neighbourhood, Algorithm 9, and the modified shake, Algorithm 12, to investigate the search space. It also takes in consideration that ξ' isn't necessarily a feasible candidate.

Assume: $k_{\max}$: *The maximum amount of investiaged neighbourhoods*
$\mathfrak{N}$: *The neighbourhood structures for different* k
$T_{\max}$: *Maximal runtime*
Function FHVNS(ξ)
Input: ξ: *The initial feasible solution*
Output: ξ: *The best found feasible solution*

Repeat
$k \leftarrow 1$
Repeat
$\xi_s \leftarrow \xi$
For $l = 1$ to k **Do**
$\xi' \leftarrow$ Shake(ξ, k) ▷*This is* Shake($\mathfrak{N}_{rTR}(k, r)$)
If ξ' is infeasible **Then Continue**
$N \leftarrow$ BuildRestNeigh(ξ', k) ▷*This is* $\mathfrak{N}_{RN}(\xi, r, \mathcal{H}_h)$
$\xi'' \leftarrow$ BestImprovement(ξ', N) ▷*Changed from* FirstImprovement
$\xi \leftarrow$ KeepBest(ξ, ξ'') ▷*Returns best* ξ, ξ'' *according to* $\hat{g}(\xi), \hat{g}(\xi'')$
End For
$(\xi, k) \leftarrow$ NeighbourhoodChange(ξ_s, ξ, k) ▷*Allways*
Until $k = k_{\max}$
$T \leftarrow$ CPUTIME
Until $T > T_{\max}$
Return ξ
End Function

C.2 Path Searching

Algorithm 17 Dijkstra's Algorithm

Dijkstra's Algorithm on our network graph $\mathcal{G}_N$. Finds the shortest path, accounting for penalties, from m_{from} to some node $m_{\text{to}} \in M_{\text{to}}$.

Assume: $\mathcal{G}_N$: *Network graph, with nodes* $\mathcal{M}$ *and edges* $\mathcal{E}$

Function DIJKSTRA(m_{from}, M_{to})
Input: m_{from}: *Node to search from, also known as the source*
M_{to}: *Set of destination nodes, also known as targets*
Output: p: *Path from* m_{from} *to some node* $m_{to} \in M_{to}$

$Q \leftarrow\!\shortmid \mathcal{M}$
costs$[m] \leftarrow\!\shortmid \infty$, for all $m \in \mathcal{M}$
links$[m] \leftarrow\!\shortmid$ **null** for all $m \in \mathcal{M}$
costs$[m_{\text{from}}] \leftarrow\!\shortmid 0$

While $|Q| > 0$ **Do**
$m_i \leftarrow\!\shortmid$ node in Q with minimum costs$[m_i]$
Remove m_i from Q
If $m_i \in M_{\text{to}}$ **Then**
$m_{\text{to}} \leftarrow\!\shortmid m_i$
Break
End If

For each edge $e_{ij} \in \mathcal{E}$ from m_i to a neighbour m_j **Do**
cost $\leftarrow\!\shortmid$ costs$[m_i] + t_i^{\alpha,\min} + t_{i,j}^{\text{penalty}}$
If cost $<$ costs$[m_j]$ **Then**
costs$[m_j] \leftarrow\!\shortmid$ cost
links$[m_j] \leftarrow\!\shortmid m_i$
End If
End For
End While

$p \leftarrow\!\shortmid$ path containing only m_{to}
$m \leftarrow\!\shortmid m_{\text{to}}$
While $m \neq m_{\text{from}}$ **Do**
$m \leftarrow\!\shortmid$ links$[m]$
Prepend m to p
End While
Return p

End Function

Algorithm 18 Penalizing Dijkstra Path Search

A algorithm to find a number of paths using Dijkstra such that newly paths attempt to avoid traveling similar paths by changing the penalty. This is done by locally increasing the penalty in the Dijkstra algorithm by the time per node for all edges each time a previous path has passes that edge.

Function PenalizingDijkstra(m_{from}, M_{to}, `numDesired`)
Input: m_{from}: *Node to search from, also known as the source*
M_{to}: *Set of destination nodes, also known as targets*
`numDesired`: *Number of paths that are desired*
Output: `paths`: *Collection of possible paths from m_{from} to nodes in M_{to}*

`paths` $\leftarrow\!\shortmid$ empty set of paths
Loop `numDesired` times
`path` $\leftarrow\!\shortmid$ Dijkstra(m_{from}, M_{to})
Add `path` to `paths`
v := `path.time()` / `path.numberOfNodes()`
For each edge e_{ij} taken in `path` **Do**
Increase $t_{i,j}^{\text{penalty}}$ by v for remaining invocations of Dijsktra
End For
End Loop

Return `paths`
End Function

Algorithm 19 Loop Erased Random Walk

Find a number of random paths from start to end.

Function LoopErasedRandomWalk(m_{from}, M_{to}, `numDesired`)
Input: m_{from}: *Node to search from, also known as the source*
M_{to}: *Set of destination nodes, also known as targets*
`numDesired`: *Number of paths that are desired*
Output: `paths`: *Collection of possible paths from m_{from} to nodes in M_{to}*

`paths` $\leftarrow\!\shortmid$ empty set of paths
Loop `numDesired` times

$n \leftarrow\!\shortmid m_{\text{from}}$
Repeat
$n' \leftarrow\!\shortmid$ random neighbour accessible from n
`links`$[n] \leftarrow\!\shortmid n'$
$n \leftarrow\!\shortmid n'$
Until $n \in M_{\text{to}}$

$n \leftarrow\!\shortmid m_{\text{from}}$
`path` $\leftarrow\!\shortmid$ path containing only n
Repeat
$n \leftarrow\!\shortmid$ `links`$[n]$
Append n to `path`
Until $n \in M_{\text{to}}$
Add `path` to `paths`

End Loop
Return `paths`
End Function

Appendix D

Operators and Symbols

This appendix is here to clarify what the operators and symbols used in the thesis mean, and is intended to be a convenient reference.

D.1 Operators

$a \in B$	denotes that a is an element of the set B.
$a \notin B$	denotes that a is not an element of the set B.
$a \leftarrow\!\shortmid b$	assignment, the value b is assigned to a.
$\forall, \exists, \exists!$	are quantifiers meaning "for all", "exists", and "exists exactly one", respectively.
$\lvert A \rvert$	denotes cardinality of A, i.e., the number of elements in the set A.
$\lVert p \rVert^L$	denotes the measure equal to the total length of a path p.
$M^{\mathbf{T}}$	denotes the matrix transpose of the matrix M.
$a \sim b$	denotes that a similar to b. It is mainly used to show that two paths or nodes represent the physical thing, where one of then belongs to the Alternative Model and the other to the Network model.
$a \models\!\!\shortmid a'$	denotes that arcs a and a' have to be chosen the same way; see Subsection 4.2.3.

D.2 Symbols

Section 3.1 – Mixed Integer Linear Programming (MILP)

z is the objective variable of an optimization problem. It is equal to the objective function.

Section 3.2 – Alternative Graphs

$\mathcal{G}_A$ is an alternative graph, where $\mathcal{G}_A = (\mathcal{N}, \mathcal{F}, \mathcal{A})$, representing a pacing or scheduling problem.

$\mathcal{N}$ is the set of nodes in the alternative graph $\mathcal{G}_A$.

$\mathcal{F}$ is a set of fixed arcs in $\mathcal{G}_A$.

$\mathcal{A}$ is a set of alternative arcs in $\mathcal{G}_A$.

n is a node $n \in \mathcal{N}$.

t is a time, associated with a node n in $\mathcal{G}_A$.

Subsection 3.2.1 – Fixed Arcs

f is a fixed arc, where $f \in \mathcal{F}$.

u is the weight of a fixed arc.

$\ell(\cdot, \cdot)$ denotes a makespan in $\mathcal{G}_A$. The makespan is the longest path between two nodes n_i and n_j in $\mathcal{G}_A$.

n_0 is a virtual node in $\mathcal{G}_A$ that always has time $t_0 = 0$.

Subsection 3.2.2 – Alternative Arcs

a is an alternative arc $a \in \mathcal{A}$.

a^* is the alternative arc that is paired with a, and $(a^*)^* = a$. a^* is also known as the "conjugate" of a.

w is the weight of an alternative arc.

Section 3.3 – Variable Neighbourhood Search

$\mathfrak{N}(\cdot)$ is an abstract neighbourhood.

N is a specific neighbourhood.

ξ is a feasible solution.

S is a value in the solution space.

Ξ is the set of feasible points is S.

$g(\cdot)$ is the objective function such that $z = g(\xi)$.

ξ_g is a global minimum to g.

ξ_l is a local minimum to g.

$\xi^\star$ is an approximation of the global minimum ξ_g.

k	is the index of the neighbourhood.
$k_{\max}$	is the maximum number of neighbourhoods.
T	is the current elapsed real time.
$T_{\max}$	is the maximum elapsed real time permitted.

Section 4.1 – Network Model

$\mathcal{G}_N$	is a network graph, where $\mathcal{G}_N = (\mathcal{M}, \mathcal{E}, \mathcal{C})$, which describes the layout of the physical rail.
$\mathcal{M}$	is the set of nodes in the network graph $\mathcal{G}_N$.
$\mathcal{E}$	is the set of (directed) edges in the network graph $\mathcal{G}_N$.
$\mathcal{C}$	is the set of conflicts in the network graph $\mathcal{G}_N$. See Subsection 4.1.1.
m	is a node in $\mathcal{M}$.
m_i	is a specific node in $\mathcal{M}$.
e	is an edge in $\mathcal{E}$.
c	is a conflict $c \in \mathcal{C}$. See Subsection 4.1.1.
$\mathcal{T}$	is the set of trains.
$\tau_1, \tau_2, \ldots$	The train in $\mathcal{T}$.
α, β	refer to two different, arbitary trains in $\mathcal{T}$.
$\mathcal{P}^\alpha$	is the set of all valid paths for train α. As an implementation detail, the $\mathcal{P}^\alpha$ sets that the implementation works with will generally be subsets of the *true* $\mathcal{P}^\alpha$, since computation of *all* valid paths may be expensive.
P	is a set of paths for all trains in $\mathcal{T}$ such that there is a complete bijection between $\mathcal{T}$ and P.
p, q	refer to paths.
p^α	refers to path p taken by train α, in cases where train properties are relevant. By convention, train α will usually take path p and train β will usually take path q.

Subsection 4.2.1 – Alternative Graph Construction

P'	is a set of paths in the alternative graph $\mathcal{G}_A$ and corresponds to P in $\mathcal{G}_N$.
p'	is a path in the alternative graph $\mathcal{G}_A$ and corresponds to path p in $\mathcal{G}_N$, and is thus often written without the apostrophe.
$n_{e+1}^{p'}$	is a *virtual* node after the end of every path in $\mathcal{G}_A$.
$t^{\alpha,\text{enter}}$	is the constant initial time $t_1^{\alpha p'}$ for each train when it enters the network and begins being modelled and obstructs other trains.
f_i^p	is the fixed arc $f \in \mathcal{F}$ from the i-th node of p to its successor.
$t^{\alpha,\text{wait}}$	is a wait time for train α from when the train enters the network until it may start moving.
$t^{\alpha,\text{dwell}}$	is a dwell time for train α from when the train reaches its destination until it leaves the network.

I, J are conflict intervals on some specific paths, given a particular conflict.

φ is an operator/function that gives the "release node" of a given node on a path.

Subsection 4.2.2 – Train Length and Moving Block

l^α is the length of train $\alpha \in \mathcal{T}$

Subsection 4.2.3 – Choice Couplings

K is a coupling, which is a set of arcs $a \in K$ such that for all arcs $a, a' \in K \Rightarrow a \models\!\!\models a'$. See Appendix D.1 and Subsection 4.2.3 for the definition of $\models\!\!\models$.

$\mathcal{K}$ is the set of all couplings K.

Subsection 4.2.4 – Schedule graph

$\mathcal{G}_S$ is a schedule graph, which is an alternative graph where some number of alternative arcs have been selected.

$\mathcal{N}$ is the nodes of $\mathcal{G}_S$, the same nodes as in $\mathcal{G}_A$.

$\mathcal{S}$ is the schedule of a scheduling graph $\mathcal{G}_S$.

C^P, C^P_α is the priority for a path (train α).

$\ell^{\mathcal{S}}(n_i, n_j)$ is the makespan between nodes $\mathcal{N}_i$ and $\mathcal{N}_j$ in the graph $\mathcal{G}_S$ identified by the schedule $\mathcal{S}$.

Chapter 5 – Solution Methodology

$\hat{g}(\xi)$ is a approximate solution of $g(\xi)$.

$\hat{\xi}$ is a candidate solution (or simply candidate) to g, not necessarily feasible.

Section 5.2 – VNS

$\mathfrak{N}_{r\mathrm{TR}}(\hat{\xi}, r)$ is the complete r reroute neighbourhood. It is the neighbourhood where r number of trains have been rerouted from a given candidate $\hat{\xi}$.

r is the number of trains to be rerouted, where $r \leq r_{\max}$.

$r_{\max}$ is the maximum number of trains to be rerouted during the VNS algorithm. It is constrained by $r_{\max} \leq |\mathcal{T}|$

$\mathfrak{N}_{\mathrm{RN}}$ is the restricted neighbourhood to be investigated. It is defined as a subset of $\mathfrak{N}_{r\mathrm{TR}}(\xi, r)$.

L is the number of desired candidates in the restricted neighbourhood, such that $|\mathfrak{N}_{r\mathrm{TR}}(\xi, r)| \leq L$.

$\mathcal{H}$ is an abstract heuristic measure, where $\mathcal{H} \in \boldsymbol{\mathcal{H}}$, used to evaluate the score of a train $s(\alpha)$ during the construction of $\mathfrak{N}_{\mathrm{RN}}$.

$\boldsymbol{\mathcal{H}}$ is the set of abstract heuristic measures.

h is the index of the heuristic measure to be used.

$h_{\max}$ is the number of heuristics to be used, where $h \leq h_{\max} = |\boldsymbol{\mathcal{H}}|$.

$s(\alpha)$ is the score for a train given a heuristics $\mathcal{H}$.

Bibliography

[1] P. Tormos, A. Lova, F. Barber, L. Ingolotti, M. Abril, and M. A. Salido, "A genetic algorithm for railway scheduling problems," in *Metaheuristics for Scheduling in Industrial and Manufacturing Applications*, F. Xhafa and A. Abraham, Eds., Springer, 2008, pp. 255–276. DOI: 10.1007/978-3-540-78985-7_10.

[2] L. Cadarso and Á. Marín, "Improving robustness of rolling stock circulations in rapid transit networks," *Computers & Operations Research*, vol. 51, pp. 146–159, 2014. DOI: 10.1016/j.cor.2014.05.007.

[3] A. Mascis and D. Pacciarelli, "Job-shop scheduling with blocking and no-wait constraints," *European J. of Operational Research*, vol. 143, pp. 489–517, 2002. DOI: 10.1016/S0377-2217(01)00338-1.

[4] A. D'Ariano, D. Pacciarelli, and M. Pranzo, "A branch and bound algorithm for scheduling trains in a railway network," *European J. of Operational Research*, vol. 183, pp. 643–657, 2007. DOI: 10.1016/j.ejor.2006.10.034.

[5] F. Corman, A. D'Ariano, D. Pacciarelli, and M. Pranzo, "A tabu search algorithm for rerouting trains during rail operations," *Transportation Research Part B*, vol. 44, pp. 175–192, 2010. DOI: 10.1016/j.trb.2009.05.004.

[6] M. Samà, A. D'Ariano, D. Pacciarelli, and F. Corman, "Lower and upper bound algorithms for the real-time train scheduling and routing problem in a railway network," Elsevier, 2016. DOI: 10.1016/j.ifacol.2016.07.036.

[7] M. Samà, A. D'Ariano, F. Corman, and D. Pacciarelli, "A variable neighbourhood search for fast train scheduling and routing during disturbed railway traffic situations," *Computers & Operations Research*, vol. 78, pp. 480–499, 2017. DOI: 10.1016/j.cor.2016.02.008.

[8] N. Andréasson, A. Evgrafov, M. Patriksson, Z. Nedělková, K. Sou, and M. Önnheim, *An Introduction to Continuous Optimization — Foundations and Fundamental Algorithms*. Nov. 2016, ISBN: 978-91-44-11529-0.

[9] P. Hansen, N. Mladenović, and J. A. Moreno Pérez, "Variable neighbourhood search: Methods and applications," *4OR*, vol. 6, pp. 319–360, 2008. DOI: 10.1007/s10288-008-0089-1.

[10] J. Desrosiers and M. Lübbecke, "A primer in column generation," in. Mar. 2006, pp. 1–32. DOI: 10.1007/0-387-25486-2_1.

[11] G. Şahin, R. Ahuja, and C. Cunha, "Integer programming based approaches for the train dispatching problem," Sabanci University, Istanbul, Turkey, 2008, Technical Report.

[12] J. Lundgren, M. Rönnqvist, and P. Värbrand, *Optimization*. Lund, Sweden: Studentliteratur, 2010, ISBN: 978-91-44-05308-0.

Department of Mathematical Sciences
Chalmers University of Technology
Gothenburg, Sweden, 2020
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY