NTNU – Trondheim
Norwegian University of
Science and Technology

Fare inspection optimization in train networks

Lars Bakke Krogvig

Master of Science in Physics and Mathematics
Submission date: June 2014
Supervisor: Helge Holden, MATH

Norwegian University of Science and Technology
Department of Mathematical Sciences

Abstract

In this thesis we present the *inspector scheduling problem* set in the local train network around Oslo serviced by Norges Statsbaner (NSB). We review current literature on the subject and present our own solution method adapted to NSB's inspection policy. By using mathematical optimization and specifically column generation we construct an optimal set of patrols plans and a corresponding probability distribution. Inspections are randomized by sampling patrols each work day. Solving the optimization problem presents computational challenges and we present a heuristic method for finding approximate solutions. The solution method is validated through several numerical experiments using example train networks inspired by the actual NSB local train network. The results suggest that our method is viable for practical applications although some work remains before this can be realized.

Sammendrag

I denne masteravhandlingen presenterer vi problemet som er å planlegge optimale reiseplaner for billettkontrollører på NSBs lokaltog rundt Oslo. Vi gjennomgår eksisterende litteratur om temaet og presenterer vår egen løsningsmetode tilpasset NSBs retningslinjer for billettkontroller. Ved hjelp av matematisk optimeringsteori og nærmere bestemt kolonnegenerering kan vi konstruere et optimalt sett av reiseplaner og en tilhørende sannsynlighetsfordeling. Tilfeldige reiseplaner kan trekkes ut fra settet hver dag ved hjelp av sannsynlighetsfordelingen. Det tilhørende optimeringsproblemet har en stor grad av kompleksitet som kan gjøre det nødvendig å bruke heuristiske fremgangsmåter for å finne gode løsninger. Vi presenterer en slik metode og utfører en serie av numeriske eksperimenter for å validere metoden. I eksperimentene benytter vi oss av konstruerte eksempelnettverk inspirert av lokaltognettet rundt Oslo. Resultatene indikerer at metoden vår har potensiale til å kunne brukes i praksis men at det fremdeles gjenstår noe arbeid før dette kan bli gjennomførbart.

Preface

This master's thesis concludes my study at the Master's Degree Programme in Applied Physics and Mathematics with specialization in Industrial Mathematics at The Norwegian University of Science and Technology.

I would like to thank Truls Flatberg at SINTEF Technology and Society for providing me with the opportunity to write this thesis and for helping me through the course of the semester. I would also like to thank my supervisor Helge Holden from the Department of Mathematical Sciences for guiding me along the way. Thanks also to Trond Inge Berg at Norges Statsbaner (NSB) for answering my questions and giving me helpful insight into the work of fare inspectors.

Lars Bakke Krogvig
Trondheim, June 19, 2014.

Contents

Abstract . . . i
Sammendrag . . . iii
Preface . . . v

1 Introduction **1**

2 TRUSTS **3**
2.1 Fare Inspection Optimization . . . 3
2.2 The LA Metro . . . 4
2.3 TRUSTS Problem setting . . . 4
2.4 Linear optimization problem (LP) formulation . . . 9
2.5 Constraining patrol duration . . . 11
2.6 Discussion . . . 13

3 Explicit formulation **15**
3.1 NSB local train network . . . 15
3.1.1 Differences from the LA Metro . . . 16
3.2 Train Networks . . . 16
3.3 Inspection model . . . 21
3.4 Optimization modelling . . . 23
3.4.1 Master problem . . . 23
3.4.2 Dual problem . . . 25
3.4.3 Column Generation . . . 26
3.4.4 Subproblem . . . 27
3.5 Solution methods . . . 28

4 Heuristics and acceleration strategies **31**
4.1 Interpreting the subproblem . . . 31
4.2 Acceleration strategies . . . 33
4.3 Heuristic methods . . . 36

5 Numerical results **40**
5.1 Basic example network . . . 40
5.2 Acceleration strategies . . . 42
5.3 Scalability and behaviour . . . 43
5.4 Heuristic methods . . . 47

6 Concluding remarks and further work 53
6.1 Model improvements 53
6.2 Heuristic improvements 54
6.3 Other further work 55

Bibliography 56

A Primal and dual problem 57

B Algorithms 60

C Python code 64

Chapter 1

Introduction

In most public transport systems passengers are required to purchase tickets to travel. As physically restricting ticketless passengers from entering the system often requires costly infrastructure and personnel, many transportation companies opt to use the honour based proof-of-payment system to collect passenger fares. In a proof-of-payment system passengers are free to enter the system without being checked for tickets, but risk being inspected and fined along the way by fare inspectors.

The main disadvantage of the proof-of-payment approach is that revenue may be lost when passengers choose to not purchase tickets and avoid being inspected during their journeys. If on the other hand fare evading passengers are inspected they must pay a fine that is usually much more expensive than tickets. Transportation companies must assign fare inspectors to discourage passengers from fare evading and to recover lost revenue. When successfully applied, the revenue lost due to fare evasion is less than the costs of implementing and maintaining systems that would restrict passengers from travelling without tickets.

The degree of success of a proof-of-payment system relies heavily on the effectiveness of the fare inspections. In realistic situations the capacity of the fare inspection staff is limited, and only a fraction of the total passenger volume can be inspected each day. Ticket sales often constitute a large share of the total revenue for transportation companies, and thus designing and executing well-planned inspections is of great importance. Unfortunately designing inspection plans can be a challenging task. It is desirable to discourage as many passengers as possible from fare evading, but any regularity in the execution of fare inspections is likely to be noticed and exploited by passengers that travel regularly. Finding the balance between effectiveness and unpredictability is key.

There are many different ways of attacking the problem of scheduling fare inspections. The problem itself is hard to universally formulate, as different transportation systems have different features that present different challenges to service operators that in turn may have different interests. It is also difficult to uncover exactly how passengers behave and respond to inspections, and this uncertainty makes it hard to declare any method of scheduling inspections superior to another.

In this master thesis we will consider the problem of planning inspections in general *train networks*. Our main goal will be to develop a procedure to schedule fare inspections for train company *Norges Statsbaner* (NSB) for use in the local train net-

work surrounding Oslo, shown in Figure 1.1. Even so we will make our approach in general terms to broaden the applicability and adaptability of the resulting method as much as possible.

We will start by giving a brief overview of the relevant scientific literature that currently exists on the topic and present a recent solution approach to similar problem, namely a method of scheduling fare inspections for the Los Angeles Metro Rail system. Then we will move on to presenting the NSB setting and build a framework for describing the problem in detail. We subsequently present an alternate solution approach and discuss some of the practical aspects of the solution method. Finally we present the results of a few numerical experiments designed to validate and explore the potential of our approach.

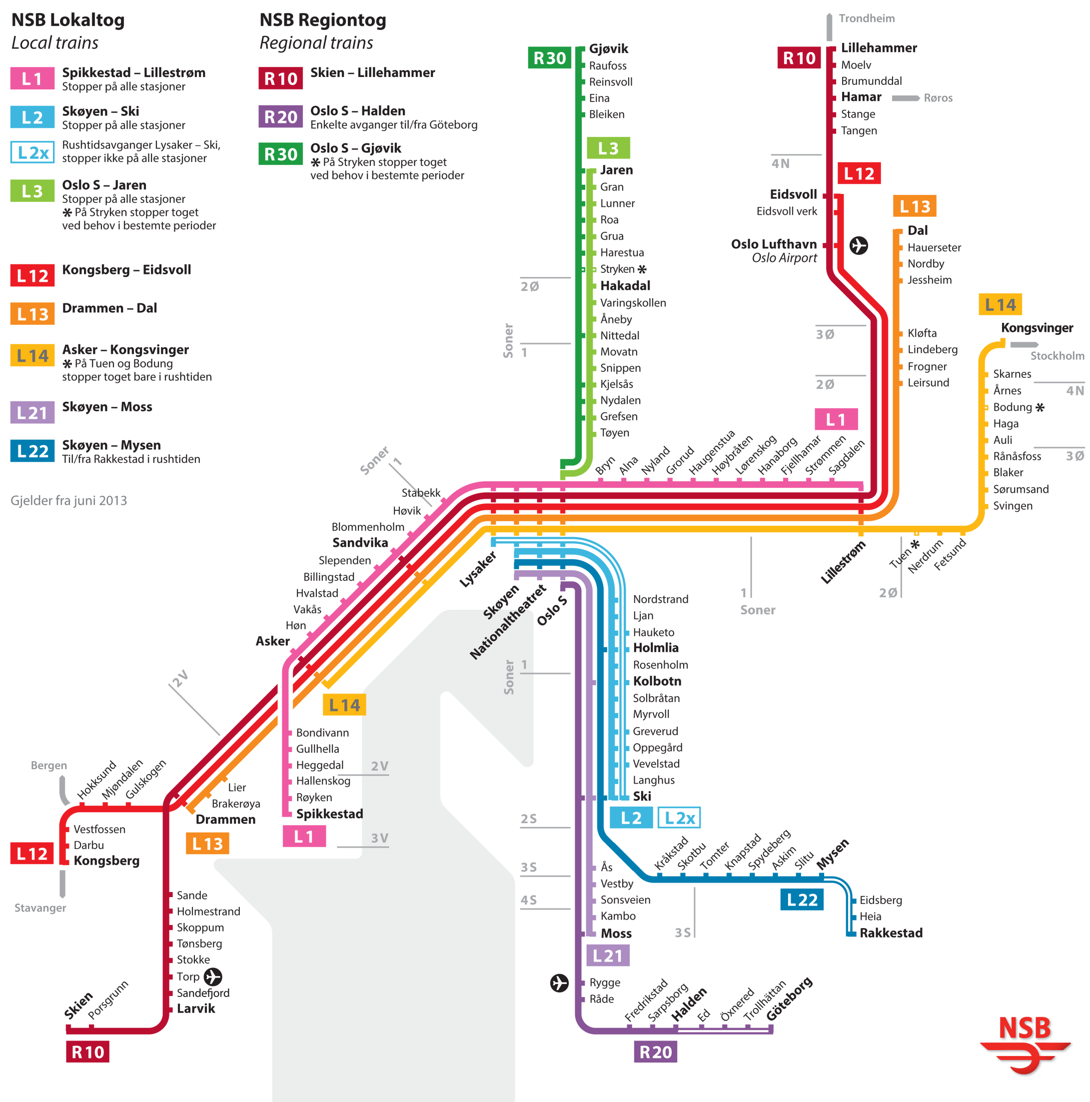


Figure 1.1: Local trains around Oslo, Norway, operated by NSB.

Chapter 2

TRUSTS

2.1 Fare Inspection Optimization

The problem of scheduling fare inspections in trains is wide and ambiguous, and only one of many complex train-related problems that warrant the use of mathematical analysis. Mathematical optimization is currently a central tool in this field which has been applied successfully to a number of problems in the past. Several examples that underline this can be found in [1]. The *inspector scheduling problem* is not found here as it is currently not a well-established train optimization problem. While issues related to fare evasion in proof-of-payment transportation systems certainly have been studied before, measures to alleviate the problem can be found using different schools of thought. The optimization angle to scheduling inspectors is one of these and the one that we will be pursuing in the following.

Currently there is little available scientific literature that deal specifically with the inspector scheduling problem, let alone from a mathematical optimization perspective. To prepare for our treatment of the problem we will present in detail a paper by Yin et al. [9] that presents an inspector scheduling application for the Los Angeles Metro rail system called *Tactical Randomization for Urban Security in Transit Systems (TRUSTS)*. The application applies mathematical optimization to generate daily travel plans for inspectors with restricted working hours. A refinement of TRUSTS applied to the same setting is presented by Jiang et al. [5]. In this paper the possibility of unforeseen events and delays are taken into consideration, and means to adapt travel plans dynamically are introduced. So-called execution uncertainty will be beyond the scope of this master thesis and we will focus on [9] in the following.

TRUSTS will be our starting point for handling the inspector scheduling problem set in the NSB local train network, and a presentation of its solution approach will be given shortly. We will assume throughout that readers have a basic understanding of linear optimization problems (linear programming) and refer to Chvátal [2] for background material. More technical details will be explained along the way whenever necessary. To start off we present the LA Metro setting that TRUSTS is developed for. We then go on to explain the problem formulation in more mathematical terms. Key features and assumptions made along the way will be explained and clarified as much as possible. To round off the chapter we include a short dis-

cussion of the solution technique and its viability for different settings like the NSB local train network we consider in subsequent chapters.

2.2 The LA Metro

The LA Metro system is a proof-of-payment urban rail system consisting of six different lines. Passengers can purchase tickets from machines outside any station but are free to enter the system as they please. The price for a single trip on any one line is fixed and independent of its origin and destination. Day passes, weekly passes and other special tickets are offered as well, but only single-trip tickets are considered in the following.

Fare inspectors work in teams called *patrol units*, and each day several units operate within the system. Ticket inspections are carried out either on board the trains or in the stations. During *on-train* inspections patrol units move through trains inspecting passengers sequentially along the way. The patrol units may inspect only parts of the train before departing. When performing *in-station* inspections patrol units stand by station exits inspecting passengers for tickets as they leave. Passengers that are caught by any form of inspection without tickets receive a fine that is much greater than the ticket price.

2.3 TRUSTS Problem setting

The lines in the LA metro only a handful of transfer points between then, and single-trip tickets are valid for travel with a single line only[1]. Because of this TRUSTS treats each line in the LA metro as independent, and only one is considered at a time. Each line is occupied by trains travelling back and fourth according to a daily schedule.

The Transition Graph

To represent the train schedule the directed *transition graph* $G = (V, E)$ is introduced. A vertex $v = (s, t)$ in the transition graph is a pair of a station s and a time t that corresponds to a train stop. In G there are two types of edges $e = (vv')$, namely *train edges* and *waiting edges*. There is a train edge between vertices $v = (s, t)$ and $v' = (s', t')$ if and only if s and s' are consecutive stops for a train at times t and t' respectively. Train edges correspond to displacements of trains and thus constitute the basis for passenger journeys. Waiting edges are edges that correspond to waiting in train stations between stops of different trains. There is a waiting edge between vertices $v = (s, t)$ and $v' = (s, t')$ if two trains stop at station s at times t and t' and no trains stop at the same station in between. A basic example of a transition graph is given in Figure 2.1. Trips that passengers or patrol units take in the train system can be represented by paths in the transition graph.

[1] See `http://www.metro.net/riding/fares/` for terms.

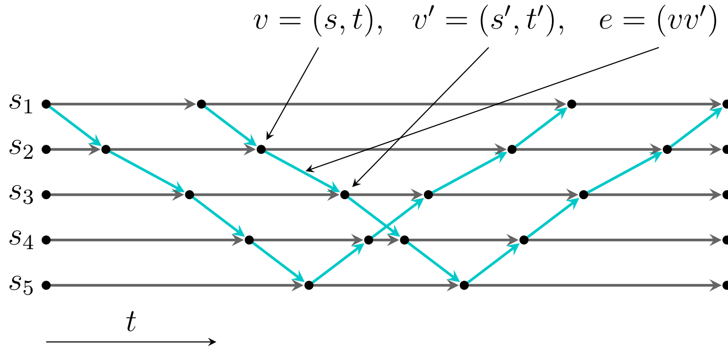


Figure 2.1: A transition graph for a train line with five stations, s_1 to s_5, with two trains travelling back and fourth. Time increases towards the right. Vertices are shown as black dots, train edges as diagonal blue arrows and waiting edges as grey arrows.

The Passengers

The ticket price for a single journey is fixed and denoted by ρ. Fare evaders that are caught during their journeys must pay a fine $\beta > \rho$. Passengers can travel between any two stations on the metro line at many different times per day whenever trains are available. To differentiate between similar journeys at different times, *passenger types* are introduced. A passenger type $\lambda \subset E$ is a path in G corresponding to a distinct journey with a particular train. Passengers that travel between the same two stations at different times are thus considered to be of different types. Paths that represent passenger types consist of one or more consecutive train edges and ends with a single waiting edge. The final waiting edge corresponds to the passenger exiting its destination train station where an in-station inspection may be in progress. The set of passenger types is denoted by Λ.

The Patrols

Fare inspections are scheduled by assigning transition graph paths or *patrols* $P_i \subset E$ to each patrol unit. The total number of patrol units is denoted by γ. A patrol is a complete description of the patrol unit's planned movements for the entire work day. Each patrol starts and ends in one of several designated times and places. The sets of vertices where patrols may start and end are denoted by V^+ and V^- respectively, both subsets of V. For algorithmic convenience source and sink vertices v^+ and v^- are added to V, along with dummy edges from v^+ to each starting vertex $v \in V^+$ and from each ending vertex $v \in V^-$ to source vertex v^-. In this way, any transition graph path that starts in v^+ and ends in v^- is a valid patrol. A complete daily patrol schedule is a collection of patrols $\mathbf{P} = (P_1, \ldots, P_\gamma)$, one for each patrol unit.

It is assumed that patrol units are able to inspect a constant number of passengers per minute. We denote this number by μ. The *inspection effectiveness* f_e is the number of passengers inspected during an inspection on edge e divided by the total number of present passengers, or in other words, the relative fraction of present passengers that is inspected. To compute the effectiveness we use the edge *duration*

h_e, which is the time difference between the two vertices belonging to the edge:

$$h_e := t' - t, \qquad \text{where } e = ((s,t),(s',t')) \,.$$

We also need to know the number of present passengers on each edge, which we refer to as the *passenger volume* and denote by u_e. The inspection effectiveness can be computed in different ways depending on which model is used, the simplest of which would be

$$f_e := \min\left\{\mu \cdot \frac{h_e}{u_e},\ 1\right\}. \tag{2.1}$$

In the LA metro trains consist of two cars that inspectors can not switch between while the train is moving and in [9] the inspection effectiveness is capped at 0.5 to reflect this. It is also assumed here that a patrol unit can inspect $\mu = 10$ passengers per minute.

Inspection probability

Given a daily patrol schedule $\mathbf{P}$ passengers are inspected during their journeys with different probabilities depending on their passenger type, i.e. when or where they travel. We denote the *inspection probability* given $\mathbf{P}$ for a passenger of type λ by $q_{\lambda|\mathbf{P}}$.

In addition to the patrols in use the inspection probability also depends on the operating procedure of the patrol units. We work under the assumption that passengers are inspected sequentially as inspectors move through trains as opposed to being selected at random during inspections. When the entire train has been covered, the inspectors can continue the inspection by re-inspecting the train and new passengers that may have boarded the train during the inspection. In TRUSTS the following model for the inspection probability is used:

$$q_{\lambda|\mathbf{P}} := \min\left\{\sum_{i=1}^{\gamma} \sum_{e \in P_i \cap \lambda} f_e,\ 1\right\}. \tag{2.2}$$

For passengers of type λ the inspection probability is given by the total inspection effectiveness over edges occupied by both the passengers and patrol units, upper bounded by 1. The model is based on the assumption that passengers of each type are distributed evenly in the trains and that inspection contributions from multiple patrol units can be added. The latter assumption is realistic when the number of overlapping teams is small and the units can cooperate.

We illustrate the model with an example: Suppose a patrol unit performs an inspection on two consecutive train edges e_1 and e_2. There are three types of passengers: λ_1 uses both e_1 and e_2, λ_2 uses only e_1 and λ_3 uses only e_2. During the inspection on e_1 the patrol unit inspects $\mu \cdot h_{e_1}$ passengers which is equivalent to a fraction f_{e_1} of all u_{e_1} passengers on board the train. As passengers of different types are distributed evenly, the same fraction f_{e_1} of passengers of type λ_1 and λ_2 are inspected. On the next edge e_2, an additional fraction f_{e_2} of passenger of type λ_1 are inspected along with the same fraction of passengers type λ_3. These two steps are shown in Figure 2.2 for the situation when $f_{e_1} + f_{e_2} < 1$.

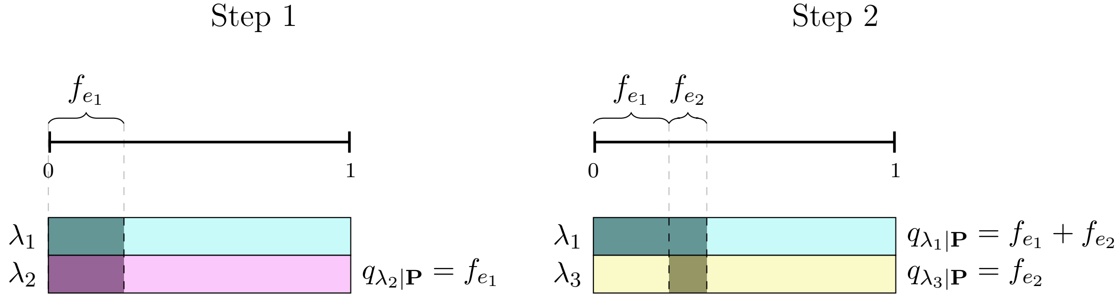


Figure 2.2: Inspection effectiveness is added to find the inspection probability.

The inspection probability is upper bounded by 1 to ensure its validity as a probability. The sum in (2.2) can exceed 1 when patrol units stay on trains over multiple edges or several teams inspect the same edge. In Figure 2.3 we see an illustration of the previous example, this time with $f_{e_1} + f_{e_2} > 1$. Inspections continue at the same rate even when all passengers of some type are inspected.

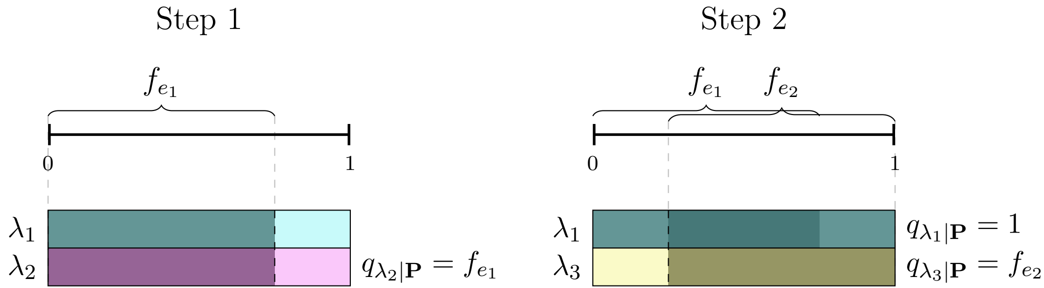


Figure 2.3: The inspection probability can not exceed 1.

We have now seen a justification of the probability model for train edges. In [9] the same reasoning is also applied to in-station inspections. Each passenger type include only one waiting edge at the end of its journey, and only one in-station inspection is encountered.

Income

A central assumption made in the TRUSTS model is all passengers have fixed travel plans regardless of their inspection probability. Passengers can be thought of as commuters that travel between pre-determined stations at pre-determined times. Being regular users of the Metro, passengers are further assumed to be able to estimate their probability $q_{\lambda|\mathbf{P}}$ of being inspected. It is also assumed that passengers are economically rational, by which we mean that they always minimize their *expected cost* of travelling. This means that passengers will not purchase tickets when the long-run cost of doing so is higher than the cost evading fares and paying fines occasionally. The expected cost of travelling without a ticket for a single journey is equal

to the fine size β times the inspection probability $q_{\lambda|\mathbf{P}}$ for that journey. Only when this expected cost is higher than the ticket price will a rational passenger purchase a ticket. Thus, the income per passenger of type λ given a fixed patrol schedule $\mathbf{P}$, $u_{\lambda|\mathbf{P}}$, is given by

$$u_{\lambda|\mathbf{P}} := \min\{\rho,\ \beta \cdot q_{\lambda|\mathbf{P}}\}. \tag{2.3}$$

Randomization

To keep passengers from noticing and exploiting regular patterns in the execution of fare inspections, the use of patrols paths must somehow be randomized. The set of all possible daily patrol schedules $\mathcal{P}$ is denoted by $\mathcal{P}$, and the probability of selecting daily schedule $\mathbf{P}$ on a given day is denoted by $\pi_{\mathbf{P}}$.

Determining the optimal probabilities $\pi_{\mathbf{P}}$ for all possible patrol plans $\mathbf{P} \in \mathcal{P}$ would be impractical as the number of possible paths in the transition graph is vast. This can be avoided by instead searching for the optimal *marginal coverage* x_e on for each edge $e \in E$. The marginal coverage for an edge can be interpreted as the expected number of patrol units using that edge per day given some probability distribution $\boldsymbol{\pi}$ over a restricted set $\mathcal{P}_r \subset \mathcal{P}$ of possible patrol schedules. It is defined as follows:

$$x_e := \sum_{\mathbf{P} \in \mathcal{P}_r} \pi_{\mathbf{P}} \sum_{i=1}^{\gamma} \theta(P_i, e), \tag{2.4}$$

where

$$\theta(P_i, e) := \begin{cases} 1 & \text{if } e \in P_i \\ 0 & \text{otherwise.} \end{cases}$$

Finding optimal values for x_e is feasible as the number of variables is equal to the number of edges in the transition graph. When the optimal values are obtained, constructing a set of patrol schedules that match the marginal coverage is simple. We will see how the marginal coverage can be determined and subsequently how patrols can be constructed shortly.

The passenger inspection probability resulting from a patrol schedule probability distribution $\boldsymbol{\pi}$ is denoted by $q_{\lambda|\boldsymbol{\pi}}$ and can be expressed using the law of total probability:

$$q_{\lambda|\boldsymbol{\pi}} := \sum_{\mathbf{P} \in \mathcal{P}_r} \pi_{\mathbf{P}}\, q_{\lambda|\mathbf{P}}. \tag{2.5}$$

By inserting Equation (2.2) into (2.5) and using that $\sum_{e \in P \cap \lambda} = \sum_{e \in \lambda} \theta(P, e)$, we

can obtain an upper bound for $q_{\lambda|\boldsymbol{\pi}}$:

$$\begin{aligned} q_{\lambda|\boldsymbol{\pi}} &= \sum_{\mathbf{P}\in\mathcal{P}_r} \pi_{\mathbf{P}} \min\left\{1, \sum_{i=1}^{\gamma} \sum_{e\in P_i\cap\lambda} f_e\right\} \\ &\le \sum_{\mathbf{P}\in\mathcal{P}_r} \pi_{\mathbf{P}} \sum_{i=1}^{\gamma} \sum_{e\in\lambda} \theta(P_i, e) f_e \\ &\le \sum_{e\in\lambda} f_e \underbrace{\sum_{\mathbf{P}\in\mathcal{P}_r} \pi_{\mathbf{P}} \sum_{i=1}^{\gamma} \theta(P_i, e)}_{x_e} \\ &\le \sum_{e\in\lambda} f_e x_e. \end{aligned} \tag{2.6}$$

The definition of the marginal coverage (2.4) is also inserted. We can use this upper bound to obtain a similar upper bound for the income per passenger given a patrol probability distribution. We denote the true income per passenger of type λ given $\boldsymbol{\pi}$ by $u_{\lambda|\boldsymbol{\pi}}$ and its upper bound by u_λ. We write:

$$u_{\lambda|\boldsymbol{\pi}} := \min\{\rho, \beta \cdot q_{\lambda|\boldsymbol{\pi}}\} \le \min\left\{\rho, \beta \cdot \sum_{e\in\lambda} f_e x_e\right\} =: u_\lambda. \tag{2.7}$$

2.4 Linear optimization problem (LP) formulation

Finding the optimal marginal coverage is referred to as the *compact formulation*, as the number of variables to be determined is far less than the number of possible patrol schedules. We now consider how the optimal values for $\mathbf{x} := [x_e]_{e\in E}$ can be determined.

When we say that a marginal coverage vector $\mathbf{x}$ is *optimal* we could mean different things depending on what we are trying to accomplish. The goal of the train operator is usually to recover as much of the revenue lost to fare evasion as possible, and thus an optimal marginal coverage is one that maximizes revenue. To find the optimal marginal coverage we turn to mathematical optimization. At this point we would like to formulate a linear optimization problem that maximizes the total revenue. Maximizing the total revenue is equivalent to maximizing the income of a random passenger, which can be expressed as a sum over passenger types in the following way:

$$\sum_{\lambda\in\Lambda} p_\lambda u_{\lambda|\boldsymbol{\pi}}.$$

Here p_λ is the probability that a random passenger is of type λ, which crucially is assumed to be known.

The true income for a random passenger does not depend on x_e, but rather on $q_{\lambda|\boldsymbol{\pi}}$. However, its upper bound u_λ does. If we maximize instead the upper bound

we can find hopefully *near-optimal* values for x_e. To do this we solve the following LP:

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{u}}{\text{maximize}} \quad & \sum_{\lambda\in\Lambda} p_\lambda u_\lambda && && (2.8a)\\
\text{subject to} \quad & u_\lambda \le \rho, && \forall\, \lambda \in \Lambda, && (2.8b)\\
& u_\lambda \le \beta \sum_{e\in\lambda} f_e x_e, && \forall\, \lambda \in \Lambda, && (2.8c)\\
& \sum_{v\in V^+} x_{(v^+,v)} = \sum_{v\in V^-} x_{(v,v^-)} \le \gamma, && && (2.8d)\\
& \sum_{(v',v)\in E} x_{(v',v)} = \sum_{(v,v')\in E} x_{(v,v')}, && \forall\, v \in V, && (2.8e)\\
& 0 \le x_e \le \alpha, && \forall\, e \in E. && (2.8f)
\end{aligned}$$

Here we have used the vector form $\mathbf{u} := [u_\lambda]_{\lambda\in\Lambda}$. The objective function (2.8a) is the sum of passenger type income upper bounds times passenger type probabilities, which is equal to the expected income of a random passenger. Equations (2.8b) and (2.8c) ensures that u_λ behaves according to (2.7). Each u_λ will always find itself at one of the two upper bounds in an optimal solution, as the objective function is a positive linear combination of the u_λ-variables. The marginal coverage x_e behaves as a flow through the transition graph between the source and sink vertices v^+ and v^-. Equations (2.8d) and (2.8e) ensure flow conservation as well as source and sink conditions of $\mathbf{x}$. Finally equation (2.8f) forces $\mathbf{x}$ to be positive and less or equal to a parameter α. The α-parameter is set somewhere in the range $[1, \gamma]$, and could prevent edges from being overly covered. This is done to reduce the likelihood of having many teams use the same edge at the same time, in which case the probability model in Equation (2.2) may no longer be valid.

Solving linear optimization problems are done using classic algorithms like the Simplex method. More modern methods also exist. We will not dwell on how LPs are solved, and refer to Chvátal [2] for a background on the Simplex method. After solving the above optimization problem the resulting optimal marginal coverage $\mathbf{x}^*$ can be used to construct explicit patrol schedules. While there is no guarantee that maximizing the income upper bound will yield optimal patrol schedules in practice, numerical results presented in [9] suggest that the resulting schedules give a total income of 96 % or better of the theoretical upper bound.

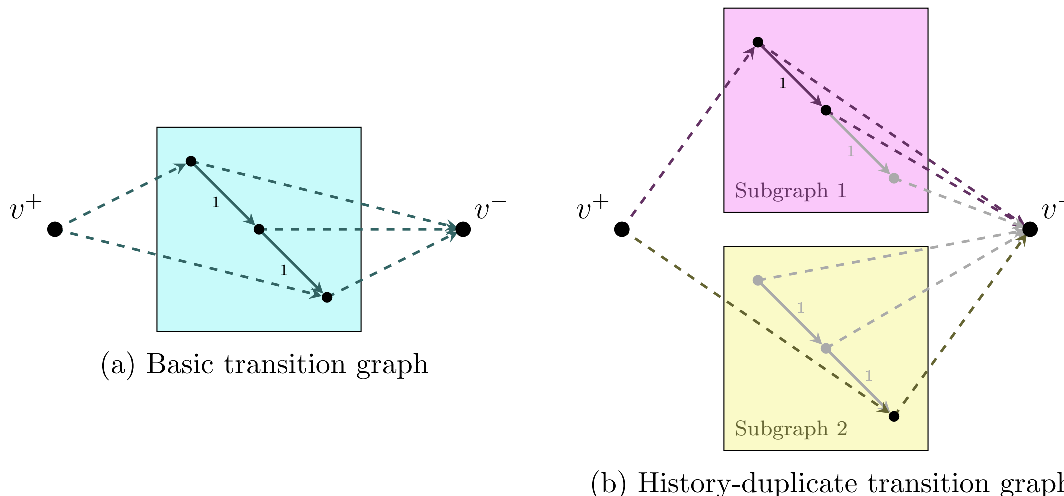
A weighted set Υ of patrol paths can be constructed using Algorithm 1. A complete patrol schedule can created daily by sampling independently from Υ random patrols for each patrol unit. A patrol is sampled with a probability equal to the normalized patrol weight. The patrols provided by Algorithm 1 all start and end at prescribed times and places but are otherwise unconstrained. A potential problem lies in that patrol paths can vary in length when different start and end times are possible. For the practical execution of patrol schedules it will often be necessary to impose certain conditions on the patrol paths. In the next subsection we look at how the problem formulation above can be altered to accommodate a maximum duration condition for the individual patrol paths.

Algorithm 1 Constructing patrol paths from $\mathbf{x} = [x_e]_{e \in E}$

Construct the empty weighted set Υ of patrol paths
while $\min_{e \in E}\{x_e\} > 0$ **do**
  Find a path P from v^+ to v^- such that $x_e > 0$ for all $e \in P$
    ▷ If this is not possible, then $x_e = 0$ for all $e \in E$ due to (2.8e)
  Add to Υ path P with weight $\tilde{x} = \min_{e \in P}\{x_e\}$
  Let $x_e \leftarrow x_e - \tilde{x}$ for all $e \in P$
end while
Normalize path weights

2.5 Constraining patrol duration

A patrol is only usable if it can be executed within the fare inspectors designated working hours. For this reason it is often necessary to impose a duration restriction on the patrol paths. Algorithm 1 finds a set of patrols that match the optimal marginal coverage and can not readily be modified to accommodate duration restrictions on paths. Fortunately this can be facilitated within the given framework by altering the transition graph.



(a) Basic transition graph

(b) History-duplicate transition graph

Figure 2.4: In (a), a trivial transition graph where all non-dummy edges have duration 1. For maximum patrol length $\kappa = 1$, the HDT graph in (b) is constructed. The vertices in edges in gray are left out of the graph to avoid long patrols.

Let us suppose that no patrol is allowed to have a duration that exceeds κ time units. If there exists two vertices $v_{start} \in V^+$ and $v_{end} \in V^-$ with time distance greater than κ, Algorithm 1 could conceivably produce a patrol between these two vertices that violate the duration requirement. This can be prevented by introducing the *history-duplicate transition graph* (HDT graph) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ to replace the basic transition graph. The HDT graph consists of multiple restricted copies of the original transition graph, one for each potential starting time. We refer to one of these copies

as a *subgraph*. We could say that vertices in the HDT graph carries one extra piece of information, namely the patrol starting time. For a subgraph corresponding to the potential start time t_{start}, only vertices $v = (s, t)$ with t such that $0 \leq t - t_{start} \leq \kappa$ are kept. The sets of vertices where patrols may start an end are now denoted by $\mathcal{V}^+$ and $\mathcal{V}^-$ respectively. A trivial transition graph with edges of duration 1 and an accompanying HDT graph for $\kappa = 1$ is illustrated in Figure 2.4.

Roughly the same procedure as before can now be used on the HDT graph to find the marginal coverage that maximizes the income upper. The notation is slightly changed by letting the marginal coverage of an edge $e \in \mathcal{E}$ in the HDT graph be denoted by y_e, and $\mathbf{y} := [y_e]_{e\in\mathcal{E}}$. Sets containing the copies of edges e from the original transition graph in the HDT graph are denoted by $\Gamma(e) \subset \mathcal{E}$. The actual marginal edge coverage x_e is thus equal to $\sum_{e'\in\Gamma(e)} y_e$ for all $e \in E$, and the near-optimal marginal coverage can be found by solving the LP:

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{y},\mathbf{u}}{\text{maximize}} \quad & \sum_{\lambda\in\Lambda} p_\lambda u_\lambda && && \text{(2.9a)}\\
\text{subject to} \quad & u_\lambda \leq \rho, && \forall\, \lambda \in \Lambda, && \text{(2.9b)}\\
& u_\lambda \leq \beta \sum_{e\in\lambda} f_e x_e, && \forall\, \lambda \in \Lambda, && \text{(2.9c)}\\
& \sum_{v\in\mathcal{V}^+} y_{(v^+,v)} = \sum_{v\in\mathcal{V}^-} y_{(v,v^-)} \leq \gamma, \leq \gamma && && \text{(2.9d)}\\
& \sum_{(v',v)\in\mathcal{E}} y_{(v',v)} = \sum_{(v,v')\in\mathcal{E}} y_{(v,v')}, && \forall\, v \in \mathcal{V}, && \text{(2.9e)}\\
& x_e = \sum_{e'\in\Gamma(e)} y_e && \forall\, e \in E, && \text{(2.9f)}\\
& 0 \leq y_e, && \forall\, e \in \mathcal{E}, && \text{(2.9g)}\\
& 0 \leq x_e \leq \alpha, && \forall\, e \in E. && \text{(2.9h)}
\end{aligned}$$

Here the objective function (2.9a) and u_λ-constraints (2.9b), (2.9c) are unchanged from the previous LP. Constraints (2.9d) and (2.9e) are analogous to constraints (2.8d) and (2.8e), this time for the y_e-variables. Constraint (2.9f) enforces the aforementioned relationship between x_e and y_e, and constraints (2.9g) and (2.9h) keeps the variables within the acceptable ranges as before. Patrol schedules can again be generated by using Algorithm 1.

The HDT graph is at most a factor $|V^+|$ (the number of possible starting vertices) larger than the transition graph. The new LP (2.9) should still be solvable by any current LP solver. Adding new restrictions on patrol paths can be done if the graph can be modified in such a way that all possible paths satisfy them. The trade-off is usually that the size of the graph increases. In [9] one additional feature is added by modifying the transition graph. The fare inspectors of the LA Metro prefer simple patrol path that do not involve excessive amounts of train switches or short in-station inspections. By modifying the HDT graph and including a penalty term for switches a balance between revenue-maximizing and simple patrol paths can be found. Numeric test results are presented in [9].

A game theoretic refinement of the above approach is presented and applied to the LA Metro setting by Jiang et al. [5]. The main idea of this paper is to make

patrols robust to unforeseen events that force patrols to deviate from their paths. A smart phone application based on [5] lets patrol units register deviations and get updated patrols during their working days. While dynamic updating of patrols is beyond the scope of this master thesis, we refer to Luber et al. [6] for details on the smart phone application and to Fave et al. [4] for recent real-world testing results.

2.6 Discussion

Before we proceed to the next chapter where we consider the inspector scheduling problem in the NSB local train network we will first discuss some of the assumptions made in TRUSTS during the problem modelling.

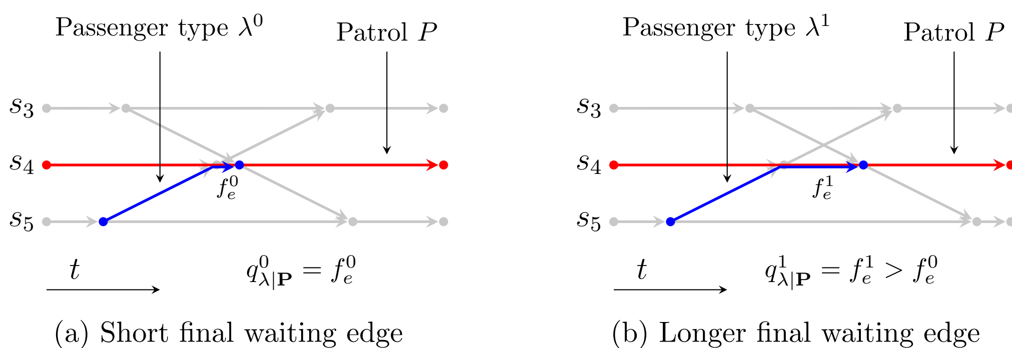
TRUSTS is designed for train networks consisting of lines that are largely independent such as the LA Metro. In networks that are more intertwined the possibility of transferring between lines becomes more relevant. This area is mentioned as a topic for future work in [9], and we will consider this in the next chapter.

A key assumption made above is that passengers behave *rationally* and minimize their expected travel costs. In real life passengers are different, some accepting more risk than others. To get a more detailed view passengers can be divided into groups that respond differently to their inspection probabilities. This is indeed done in [9]. Furthermore passengers are assumed to be able to estimate their inspection probability. This is only be realistic for passengers that take the same journey often. A possible addition to the model could be passengers that are unaware of their specific inspection probability, but know the overall rate of inspection and base their ticket purchasing choices on this. This would only be a minor addition to the model that we will not pursue any further.

More deserving of additional discussion is the probability model given in Equation (2.2). The model is based on the assumptions that patrol units inspect passengers in sequence such that the inspection fractions can be added rather than multiplied, as would be the case if passengers were sampled at random. We refer again to Figure 2.2.

The situation becomes less intuitive when a patrol unit has covered the entire train, and it is unclear exactly how inspectors should behave to comply the probability model. For example, let us consider Figure 2.3. After the inspection on edge e_1 the patrol unit has covered over half of the train. On the next edge the unit has the capacity to inspect more than the remaining part of the train. When the unit reaches the end of the train the inspectors could turn around and continue inspecting. If a fraction f_{e_2} of passengers of type λ_3 are to be inspected the unit would have to remember where in the train they started after e_1 and continue their inspection there. Alternatively the unit could split up before the inspection at e_2. If the model is to hold then inspectors now moving backwards in the train must re-inspect passengers of type λ_1, being unable to distinguish between inspected and newly arrived passengers. If inspectors could somehow avoid re-inspecting passengers then the effective number of passengers on board the train is reduced, and new passengers could be inspected with a higher efficiency. In the current model this is not reflected, and including it would most likely be impossible without destroying the linearity of the model.

Another point worth considering is how in the current model each passenger type path ends with a single waiting edge that corresponds to the exiting of the destination stations. During in-station inspections fare inspectors are posted at the station exits, inspecting passengers as they leave. If a passenger encounters both an on-train inspection and an in-station inspection their contributions are added even though these two inspections are independent. Also, the length of the single final waiting edge in each passenger type path can influence the inspection probability in an undesirable way. Consider Figure 2.5. Here a passenger type ends in a short waiting edge in the middle of a long and ongoing in-station inspection. While the patrol passenger should surely be inspected by a unit when exiting the station if no passengers can slip past the inspectors, the model inspection probability will be small due to the short duration of the waiting edge.



(a) Short final waiting edge

(b) Longer final waiting edge

Figure 2.5: The length of the final waiting edge influences the inspection probability when it should not.

While we have now raised some concerns about the probability model, the effects of the items we have mentioned may be small in practice. It is important to note that *some* model must be used for the inspection probability, as in the real world there are two many variables to determine the probability exactly. The decisive advantage of the current probability model lies in its linearity which allows for efficient computation of the optimal marginal coverage.

In the next chapter we consider a transportation system different form the LA Metro in many ways, and a slightly different approach will be required. As we will see, TRUSTS will provide a solid starting point and many elements will remain unchanged.

Chapter 3

Explicit formulation

We now turn to the problem of scheduling optimal inspections for NSB's local trains around Oslo. Unfortunately there are certain differences between the LA Metro and the NSB local train system that makes applying TRUSTS directly problematic. First and foremost, the local trains around Oslo travel much less frequently than the trains in the LA Metro. Secondly, many train lines overlap by sharing the same train tracks, and passengers can change between lines on the same ticket. Studying each train line separately is not as meaningful in this case. Additionally, NSB have a different inspection policy that calls for a revised problem formulation.

In this chapter we will attempt to solve the inspector scheduling problem for a *train network* as opposed to a single line, and we will devote an entire section to introducing terminology and notation suited to this setting. We begin the chapter with a description of the NSB's current fare inspection policy and working procedure, and explain in greater detail why the method described in the previous chapter is difficult to apply. Finally the basis of a different solution approach is given.

3.1 NSB local train network

Like the LA Metro the NSB local train network is a proof-of-payment system, and passengers can purchase tickets in different ways. Primarily passengers are encouraged to purchase tickets before boarding the train on dedicated machines at stations or using a smart phone application. Single-trip tickets, multi-trip tickets or tickets that allow for unlimited travel for certain periods of time are available. Alternatively, passengers may also purchase single-trip tickets from a guard on board the train in certain cars. The trains consist of connected cars of two different types, 'Serviced' (Betjent) cars and 'Unserviced' (Ubetjent) cars. The majority of cars are unserviced. Passengers are free to use unserviced cars, but are obliged to purchase a tickets in advance.

NSB's fare inspectors work in teams of 6–7 inspectors and have shifts that are 5–10 hours long, often including a lunch break in break rooms available at certain stations. Contrary to in the in LA metro, inspections are only carried out on board trains between stations. Inspections are normally only performed in the unserviced cars as the train guard are responsible for the serviced ones.

During inspections teams move through the trains while inspecting passengers

sequentially along the way. Each train car is only inspected once to disturb passengers as little as possible. This means that if a team of inspectors has covered the entire train once before the team is scheduled to disembark, then the inspection is finished and inspectors remain idle until leaving the train. New passengers that may have boarded the train since the beginning of the inspection will not be inspected due to the policy of not re-inspecting passengers.

If passengers are found without valid tickets they are given a choice between paying a fine immediately or receiving a slightly larger fine later in the mail. In any case processing the fine takes a few minutes and delays further inspections for the inspector issuing the fine. NSB estimates that on average it takes a team of six inspectors 2–3 minutes to inspect a car with 100 passengers, which equates to an inspection rate of 5–9 passengers per minute per inspector.

3.1.1 Differences from the LA Metro

As we are now considering a network of lines rather than a single train line we have to take train transfers for passengers and inspectors into consideration. This is not supported by TRUSTS. Another crucial difference lies in the differing inspection policies. In the LA metro fare inspectors are allowed to re-inspect trains as new passengers embark during inspections, an assumption which is built into the inspection probability model in Equation (2.2). For example, a patrol unit could conceivably stay on a train from one terminal station to another, continuously inspecting new passengers that board the train along the way. The inspectors on NSB's trains are generally not allowed inspect passengers more than once and will avoid re-inspecting cars, making it unfavourable for patrol units to stay on board trains for extended periods of time. This makes switching trains often more desirable and we need a model that reflects this aspect.

The inspection policy also makes it disadvantageous to assign multiple patrol units to the same trains or have overlapping patrols. In TRUSTS patrols are sampled independently for each team each day, something which could lead to interfering patrols in our case. One could apply some rejection based sampling technique to avoid patrol interference, but this could potentially change the effective marginal coverage of certain edges and lead to sub-optimal results.

To handle all these difficulties we will in the following sections try to develop an alternate approach to solving the inspector scheduling problem. In an attempt to be rigorous we will in the next section build a framework for describing train networks in a more general perspective. However, the notation and definitions we will introduce are adapted to the particular problem we will be considering in this thesis and not necessarily in line with any established train terminology.

3.2 Train Networks

We will now be considering general train networks. Instead of concerning ourselves with actual rail road infrastructure like tracks and switches, we will regard the network from the perspective of the user. Passengers and fare inspectors only see stations and trains travelling between them according to a given train schedule, and

this is exactly the level of detail we need when considering the inspector scheduling problem.

Stations and lines

In our train network we have a set of stations. We let the number of stations be denoted by N_s, and define the *station set*:

$$S := \{s_i\}_{i=1}^{N_s} . \tag{3.1}$$

Here s_i identifies a particular station just like a station name would, such as for example 'Trondheim Central Station'.

Trains travel in the network along different *train lines.* A train line is sequence of stations that trains visit in order. Trains regularly travel back and fourth along one designated line each according to a daily schedule. The two stations at each end of a line are called the *terminal stations* of the train line. We let the number of train lines in the train network be denoted by N_ℓ, and define the set of train lines

$$L := \left\{\ell^j\right\}_{j=1}^{N_\ell} . \tag{3.2}$$

Here train line ℓ^j is a of a sequence of n_j stations:

$$\ell^j := \left\langle s_k^j \right\rangle_{k=1}^{n_j} , \tag{3.3}$$

where s_k^j refers to the kth station along line ℓ^j. Note that the station sequence could be reversed without changing the definition of the line. Also note that the subscript k refers to the order of the station along the line when a superscript line index is present. When we write s_i with just a subscript we mean a specific station.

A pair of a station set and a corresponding set of lines (S, L) is what we will refer to a train network in the following. Train networks can be illustrated with *train maps*, similar to those usually found in train stations and on trains in real life train networks. Often each train line is given an identifying color and number or letter combination. A very simple example of an illustrated train map is given in Figure 3.1. A more elaborate real-world example is the map of local trains around Oslo that is shown in Figure 1.1.

Train traffic

Trains travel along their respective train lines according to a daily time table. We do not concern ourselves with the practical realization of the train time table, we only consider the scheduled arrivals and departures of trains that are available to the passengers. We do not keep track of the actual train vehicles, and the time table will always be given and beyond our control. For the sake of simplicity we assume that every train operating on the same line spends the same amount of time between two stations, and that all trains wait for the same amount of time at each station. A complete time table can then be specified by lists of when trains leave which terminal stations for each line.

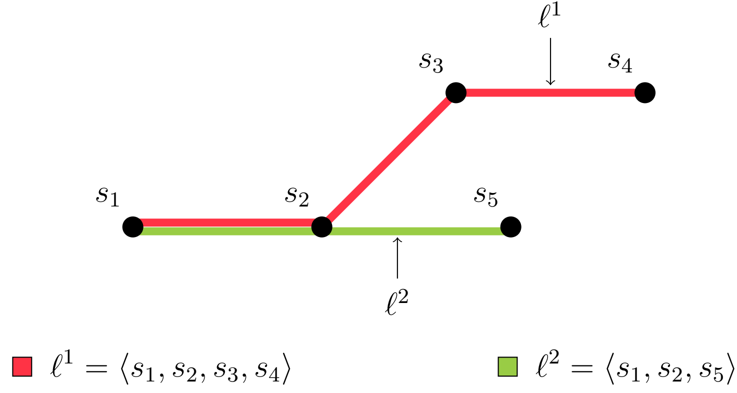


Figure 3.1: A train map with five stations and two lines. Both lines start in s_1, thus $s_1^1 = s_1^2 = s_1$. The third station of line ℓ^1 is s_3 and the third station of line ℓ^2 is s_5, thus $s_3^1 = s_3$ and $s_3^2 = s_5$.

We let the travel time between stations s and s' for a train on line ℓ be given by $\tau(s, s', \ell)$ where

$$\tau : S \times S \times L \to \mathbb{R}^+$$

is the *travel time function*. All trains wait for the same amount of time t_{wait} at each station between arrival and departure regardless of which line they travel on. In the previous chapter waiting times at stations were not included.

We define a *departure* d to be the event of a train leaving a terminal station of a train line at a scheduled time. Sometimes we also refer to a specific train as a departure. The set of all departures D is a subset of all possible departures:

$$D \subset \left\{ d = (\ell, t, \sigma) : \ \ell \in L, \ t \in \mathbb{R}^+, \ \sigma \in \{-1, 1\} \right\}. \tag{3.4}$$

Here ℓ is the line the train is operating on and t the time at which the train starts. The travel direction of the train is given by σ. If the train starts at the last station of the train line and travels in the reverse direction then $\sigma = -1$, otherwise $\sigma = 1$.

Transition graph

To model the movement of trains in the train network we again introduce a *transition graph* $G = (V, E)$, similar to the one found used in TRUSTS. The event of a train either arriving at or departing from a station at a certain time corresponds to a vertex or node v in the vertex set V. A vertex $v \in V$ is a unique pair of a station and a time, thus

$$V \subset \left\{ v = (s, t) : \ s \in S, \ t \in \mathbb{R}^+ \right\}. \tag{3.5}$$

In addition to the vertices that correspond to departures and arrivals of trains we often also include start and end vertices $(s, 0)$ and (s, t_{max}) for each station s in S. Here

$$t_{max} = \max\left(\{t : (s, t) \in V\}\right)$$

is the *end time*.

Like in the previous chapter edges $e \in E$ represent possible movements in the network. *Train edges* are edges that correspond to a train travelling between two stations. For example, a train departing from station s at time t and arriving at station s' at time t' corresponds to train edge $e = (vv') = ((s,t)(s',t'))$. We also have *waiting edges* that correspond to staying at stations between two train related events, like a train arriving at or departing from the station. The edges in the transition graph are directed as it is only possible to move forward in time. We have

$$E \subset \{e = (vv') = ((s,t),(s',t')) : \; v, v' \in V, \; t < t'\} \,. \tag{3.6}$$

An example of a transition graph is given in Figure 3.2. Here vertices are shown as black dots. Waiting edges are grey arrows and waiting edges are arrows coloured according to the train line they belong to. To construct a complete transition graph for a train network (S, L), all we need is a set of departures D along with the relevant travel time function τ. The exact procedure we use to construct the transition graph is given in Algorithm 6 in Appendix B.

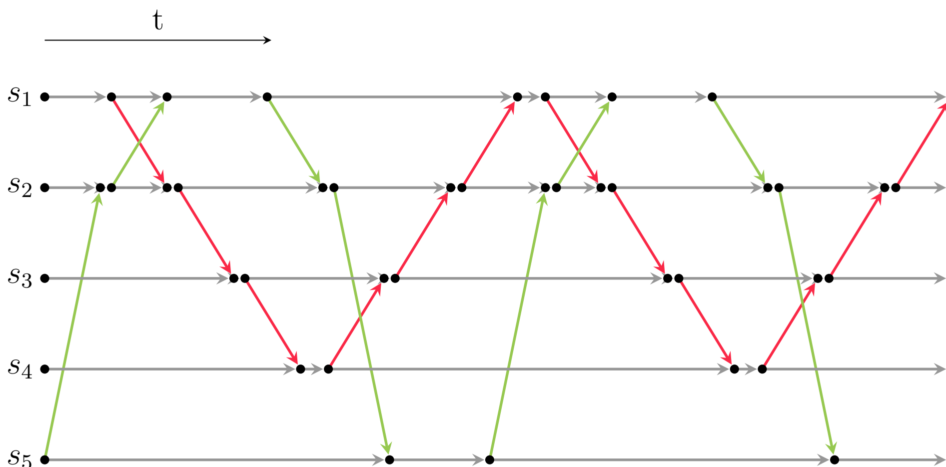


Figure 3.2: A transition graph using the train network in Figure 3.1. Time increases towards the right. Eight departures are illustrated, two in each direction on both lines.

Passengers

All passengers in the network travel from one specific station to another, possibly switching trains during their journey. Like before the travel plans of passengers are fixed, and we say that a *passenger type* λ is a path in the transition graph that correspond to a possible journey. A few examples of passenger type paths can be found in Figure 3.3. Note that we are now dealing with networks, and passengers may transfer between trains during their journey.

We denote the number of passengers of type λ by d_λ, and often refer to this as the *demand* of λ. Approximate values for the demand can often be deduced from real world data. The *passenger type set* Λ contains all *sensible* journeys that passengers may take. We define a journey from $v = (s,t)$ to $v'' = (s'', t'')$ to be sensible if no

shorter alternative exits, i.e. there is no path from some $v' = (s', t')$ to v'' with $t' > t$. Since passenger types are paths, we have that $\lambda \subset E$. To find all sensible passenger types in some transition graph G we use Algorithm 7 given in Appendix B.

Each passenger need purchase only a single ticket that depends only on the start and end station of the journey, that may include train transfers. The ticket price is given by the *ticket price function*

$$\rho : \Lambda \to \mathbb{R}.$$

Passengers may also opt to not purchase a ticket before travelling, in which case they travel for free unless inspected by fare inspectors. When passengers are caught without tickets they must pay a fine β. In contrast to the ticket price, the fine size equal for all passengers. After paying the fine a fare evading passenger continues his or her journey. Passengers can receive one fine per train, meaning that passengers on journeys that involve train transfers risk receiving more than one fine. Even though passengers could possibly purchase tickets for only parts of their journeys, we do not include this in our model.

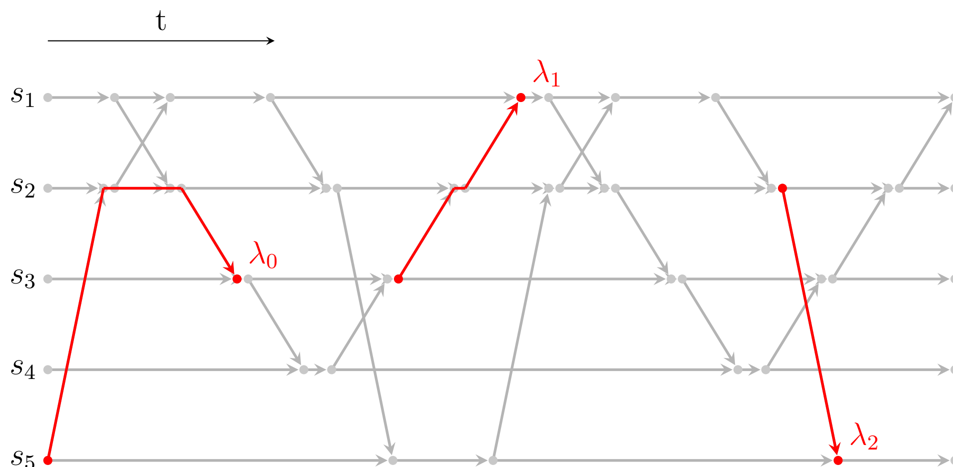


Figure 3.3: The transition graph from Figure 3.2. Highlighted in red are three different passenger types λ_0, λ_1 and λ_2.

Inspectors

Fare inspectors employed by the train service operator also travel within the train network as they inspect passengers. The fare inspectors work in teams we call *patrol units*, and there are usually multiple patrol units operating within the system at the same time. We want to make sure that the inspection plans of the patrol units are not in conflict with each other and work well together every work day. Instead of treating each unit separately we look for good collections of patrols, one for each team. We will refer to such a collection of patrols as a *joint patrol*.

Let us denote a joint patrol by $\boldsymbol{\psi}_i$ and suppose we have a *restricted* set

$$\boldsymbol{\Psi}_r := \{\boldsymbol{\psi}_i\}_{i=1}^N, \tag{3.7}$$

containing N different joint patrols

$$\boldsymbol{\psi}_i := [\psi_{i,j}]_{j=1}^T . \tag{3.8}$$

Here $\psi_{i,j}$ is the patrol used by team j in joint patrol i. The number of patrol units is T.

In the previous chapter, patrols were simply paths in the transition graph that represented to the travel plans of constantly inspecting patrol units. In this chapter patrol units must stop their inspections when all train cars are inspected, and we will now incorporate this in the model. We allow patrol units to travel with trains *without* inspecting passengers by defining a *patrol* as a transition graph path with an accompanying specification of action for each edge in the path.

Suppose we have a *patrol path*, which is just the transition graph path corresponding to a patrol. The path can be represented as a collection $\mathbf{x}(\psi_{i,j}) := [x(\psi_{i,j}, e)]_{e\in E}$ of indicator variables:

$$x(\psi_{i,j}, e) := \begin{cases} 1 & \text{if } e \text{ is used in patrol } \psi_{i,j} \\ 0 & \text{otherwise.} \end{cases}$$

On each train edge e the patrol unit may plan to inspect a number of passenger between zero and their maximum capacity $h_e\mu$, where again h_e is the duration of the edge. We let the planned number of inspected passengers on an edge e divided by the total number of passengers on board the train at this edge be the *inspection fraction* which we denote by $w(\psi_{i,j}, e)$. We also define $\mathbf{w}(\psi_{i,j}) := [w(\psi_{i,j}, e)]_{e\in E}$. If the unit inspects passengers at its maximum capacity then $w(\psi_{i,j}, e) = f_e$, the *inspection effectiveness* for that edge. We can write

$$\psi_{i,j} \in \Psi_j \subset \{0, 1\}^{|E|} \times \mathbb{R}^{|E|}$$

and

$$\boldsymbol{\psi}_i \in \boldsymbol{\Psi} \subset \Psi_1 \times \ldots \times \Psi_T,$$

where $|E|$ is the number of edges in the transition graph. Here Ψ_j is the set of all valid patrols for unit j and $\boldsymbol{\Psi}$ the set of all valid joint patrols. The restricted set $\boldsymbol{\Psi}_r$ is a subset of $\boldsymbol{\Psi}$.

3.3 Inspection model

It is now time to choose a model for the inspection probability. Let us first enumerate the set of passenger types by writing

$$\Lambda := \{\lambda_k\}_{k=1}^M, \tag{3.9}$$

where M is the number of different passenger types. Again, we have that

$$\lambda_k \subset E.$$

The probability of being inspected varies between the passenger types and depends the joint patrol used. When joint patrol $\boldsymbol{\psi}_i$ is used, a passenger of type λ_k is

inspected with a certain probability. As passengers can receive one fine per train they use during their journeys we instead consider the expected number of times passengers are inspected. We denote the expected number of times a passenger is inspected given that joint patrol $\boldsymbol{\psi}_i$ is used by $r_{i,k}$ and refer to this as the *inspection rate*.

Assuming that passengers are distributed randomly within the trains we express the inspection rate as a sum inspection fractions:

$$r_{i,k} := \sum_j \sum_{e \in \lambda_k} w(\psi_{i,j}, e). \tag{3.10}$$

The difference from the previous model in Equation (2.2) is that instead of taking the sum of inspection effectiveness parameters f_e we take the sum of the inspection fractions $0 \leq w(\psi_{i,j}, e) \leq f_e$. It can be helpful to think of the inspection fractions as the fraction of the trains length that is covered in an inspection rather than the fraction of passengers, as the total number of passengers varies from edge to edge. When passengers are evenly distributed in the train inspecting a fraction x of the total passengers is equivalent to covering the same fraction x of the length of the train.

We think of inspection patrol units as moving from one end of the train to the other. When the other end is reached, we must have that the sum of inspection fractions is equal to 1, as the whole length of the train is covered. Thus, we require that

$$\sum_j \sum_{e \in \Gamma(d)} w(\psi_{i,j}, e) \leq 1 \qquad \text{for all departures } d, \tag{3.11}$$

i.e. that the sum of inspection fractions is less or equal to 1 for each train departure. Here where $\Gamma(d)$ is the set of transition graph edges corresponding to departure d. Provided that (3.11) holds, we know that $r_{i,k}$ is less or equal to the number of trains used by passenger type λ_k.

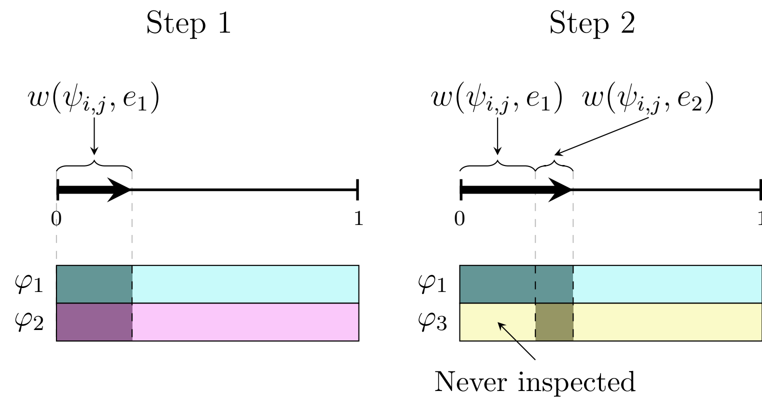


Figure 3.4: Trains are only inspected once, leaving some passengers that board the train during an inspection uninspected.

The new situation is illustrated in Figure 3.4. A patrol unit moves from one end of the train to the other. Passengers that enter the train behind the patrol unit

will not be inspected. As trains can only be covered once, the timing of inspections becomes more important than in the previous chapter. In the next section we consider how optimal joint patrols can be constructed and applied.

3.4 Optimization modelling

As previously mentioned it is important to prevent passengers from recognizing and exploiting any regularity in the fare inspections by introducing some random element in the procedure. In the previous chapter we saw a method of generating random patrol routes day by day. In this chapter we have stricter patrol requirements, and we will instead sample joint patrols each day from some pool of acceptable joint patrols. We call this the *explicit formulation* as we treat patrols directly.

Our main challenge is to construct such a pool or set of joint patrols and to determine the optimal probability distribution for the joint patrol sampling. The naïve approach would be to include all possible joint patrols with the hope of assigning a probability to each one. Unfortunately the space of possible transition graph paths increases exponentially with the size of the graph, and the number of possible joint patrol paths is simply too vast in almost all cases. Furthermore the accompanying inspection rates are continuous variables, making the set of possible joint patrols infinite.

However, of all the possible joint patrols there is only a handful that is worthy of our consideration. The key is to identify a sufficient subset joint patrols such that an optimal inspection strategy can be realized. In the following we will outline an approach of finding such a subset based on an iterative optimization technique called *column generation*. The basic idea of column generation is to iteratively generate useful joint patrols until an optimal subset is be found. Column generation is explained in [2], and we explain the procedure step by step with our problem in mind below.

3.4.1 Master problem

In column generation the main optimization problem to be solved is often referred to as the *master problem*, and the problem of finding a *variable* (in this case a joint patrol) to be added to the subset is called the *subproblem*. When the master problem is solved for a restricted set of variables this is referred to as the *restricted master problem*. Before we can discuss the subproblem we need to clearly define the master problem.

The master problem in our case is to determine the optimal sampling probabilities for a set of joint patrols such that fare evasion is minimized. Suppose for now that we have a subset $\boldsymbol{\Psi}_r$ with N distinct joint patrols. The to-be-determined probability of choosing joint patrol $\boldsymbol{\psi}_i$ is denoted by by q_i. We define the vector form

$$\mathbf{q} := [q_i]_{i=1}^{N} .$$

Using the law of total probability we can state the effective inspection rate r_{λ_k} for

passenger type λ_k given $\mathbf{\Psi}_r$ and a corresponding probability distribution $\mathbf{q}$:

$$r_{\lambda_k} := \sum_{i=1}^{N} r_{i,k}\, q_i. \tag{3.12}$$

The average amount paid by a *rational* passenger of type λ_k is thus

$$u_k := \min\left\{\rho_k,\ \beta \sum_{i=1}^{N} q_i\ r_{i,k}\right\}, \tag{3.13}$$

where $\rho_k := \rho(\lambda_k)$ is the ticket price for passenger type λ_k. Rational passengers are passengers that minimize their expected cost as explained in Section 2.3. We define the vector form

$$\mathbf{u} = [u_k]_{k=1}^{M}.$$

The probabilities $\mathbf{q}$ that maximize overall income can be found by solving the following LP:

$$\begin{aligned}
\underset{\mathbf{q},\mathbf{u}}{\text{maximize}} \quad & \sum_{k=1}^{M} d_k u_k && && (3.14a)\\
\text{subject to} \quad & \sum_{i=1}^{N} q_i = 1, && && (3.14b)\\
& u_k \le \beta \sum_{i=1}^{N} r_{i,k}\, q_i, && \forall\, k, && (3.14c)\\
& u_k \le \rho_k, && \forall\, k, && (3.14d)\\
& q_i \ge 0, && \forall\, i, && (3.14e)\\
& u_k \ge 0, && \forall\, k. && (3.14f)
\end{aligned}$$

The objective (3.14a) is the expected total income per day, where d_k are the known demand of λ_k-passengers. Constraint (3.14b) ensures that the sum of all probabilities is 1. Equations (3.14c) and (3.14d) sets the upper bounds for the passenger type income u_k, which always takes on the value of the lowest upper bound in an optimal solution. All variables are kept positive through the constraints in Equations (3.14e) and (3.14f).

An optimal solution of (3.14) is a probability distribution over the (current) set $\mathbf{\Psi}_r$ of joint patrols that gives the highest income. However, they may exist a different set $\mathbf{\Psi}_r'$ that makes a higher income possible. We refer to the *true optimum* as the best possible solution of (3.14). This can be found either by including all possible joint patrols by using $\mathbf{\Psi}$ (which would be impractical), or by finding an *optimal subset* $\mathbf{\Psi}_r^*$ that can be proved to be sufficient. Solving this optimization problem is a relatively easy task for any LP-solver as long as the number of different passenger types is manageable.

3.4.2 Dual problem

To generate new joint patrols we need to formulate and solve the subproblem. The first step towards formulating the subproblem is to express the *dual* of the master problem. In mathematical optimization, *duality* is a concept that lets optimization problems be viewed from two perspectives: Any linear optimization problem, called a *primal problem*, has a unique corresponding *dual problem*. If a bounded optimal solution exist for either problem then there also exists one for the other, and the optimal objective values for both problems are equal. This statement is widely known as *The Duality Theorem*, which can be found for example in [2]. If the primal is a minimization problem then the dual is a maximization problem and vice versa.

The dual problem arises when trying to find an upper bound for the objective value of a maximization problem (Or the lower bound for a minimization problem). For illustration let us now derive the dual to (3.14). We start by making a linear combination of the constraints in Equations (3.14b)–(3.14d). The constraint in Equation (3.14) is multiplied with a coefficient y^q, and the constraints in Equations (3.14c) and (3.14d) are multiplied with the non-negative[1] coefficients y_k^β and y_k^ρ respectively for $k \in \{1, \dots, M\}$. Taking the sum of all constraints we get

$$y^q \sum_{i=1}^{N} q_i + \sum_{k=1}^{M} y_k^\beta u_k + \sum_{k=1}^{M} y_k^\rho u_k \leq y^q + \sum_{k=1}^{M} y_k^\beta \beta \sum_{i=1}^{N} r_{i,k}\, q_i + \sum_{k=1}^{M} y_k^\rho \rho_k.$$

By rearranging terms we obtain

$$\sum_{i=1}^{N} q_i \left(y^q - \beta \sum_{k=1}^{M} y_k^\beta\, r_{i,k} \right) + \sum_{k=1}^{M} u_k \left(y_k^\beta + y_k^\rho \right) \leq y^q + \sum_{k=1}^{M} y_k^\rho \rho_k. \tag{3.15}$$

We may now take a look at the primal objective function in Equation (3.14a) and realize that if $y^q - \beta \sum_{i=1}^{N} y_k^\beta\, r_{i,k} \geq 0$ and $y_k^\beta + y_k^\rho \geq d_k$, then we have found the upper bound we were looking for:

$$\sum_{k=1}^{M} d_k u_k \leq \sum_{i=1}^{N} q_i \underbrace{\left(y^q - \beta \sum_{k=1}^{M} y_k^\beta r_{i,k} \right)}_{\geq 0} + \sum_{k=1}^{M} u_k \underbrace{\left(y_k^\beta + y_k^\rho \right)}_{\geq d_k} \leq y^q + \sum_{k=1}^{M} y_k^\rho \rho_k. \tag{3.16}$$

Finding the coefficients y_k^β, y_k^ρ and y^q that give the lowest upper bound for the primal objective function value is known as the dual problem:

$$\underset{y^q, \mathbf{y}^\beta, \mathbf{y}^\rho}{\text{minimize}} \quad y^q + \sum_{k=1}^{M} \rho_k\, y_k^\rho \tag{3.17a}$$

$$\text{subject to} \quad y_k^\beta + y_k^\rho \geq d_k, \qquad \forall\, k, \tag{3.17b}$$

$$y^q - \beta \sum_{k=1}^{M} r_{i,k}\, y_k^\beta \geq 0, \qquad \forall\, i, \tag{3.17c}$$

$$y_k^\beta, y_k^\rho \geq 0, \qquad \forall\, k. \tag{3.17d}$$

[1] To not reverse the inequalities

Here we have used vector forms

$$\mathbf{y}^\beta := \left[y_k^\beta\right]_{k=1}^M \qquad \text{and} \qquad \mathbf{y}^\rho := [y_k^\rho]_{k=1}^M . \tag{3.18}$$

The objective function in Equation (3.17a) is the upper bound for the primal problem. To make the objective an upper bound the conditions in (3.17b) and (3.17c) must be satisfied, as shown in Equation (3.16). The y_k^β and y_k^ρ coefficients must also be positive, as enforced by (3.17d).

The Duality Theorem states that lowest upper bound found in this way (the optimal solution to the dual problem) is equal to the highest possible objective fuction value in the primal problem (the optimal solution to the primal). This means that for the optimal solution of (3.17), the constraints (3.17b) and (3.17c) all hold with equality. In addition, if either solution is known then finding the other is trivial. The dual problem in (3.17) can also be obtained by writing the primal on a standard form and using a known primal-dual problem pair. This is done in Appendix A.

3.4.3 Column Generation

Solving the primal problem defined in (3.14) yields the so-called *dual variables* y^q, y_k^β and y_k^ρ corresponding to the optimal primal solution $\mathbf{q}_0, \mathbf{u}_0$. Let us denote the corresponding primal objective function value by P_0 and the dual objective function value by D_0. From the duality theorem we have that $P_0 = D_0$.

Suppose we now add one joint patrol with related inspection rates $r_{N+1,k}$ to our subset $\mathbf{\Psi}_r$ and let the corresponding sample probability q_{N+1} be zero. We call this the *updated* problem. This gives us a feasible solution $\mathbf{q}_1 = [\mathbf{q}_0, 0], \mathbf{u}_1 = \mathbf{u}_0$ and objective value $P_1 = P_0$ to the updated primal problem. Note that the dual variables are unchanged until we re-solve the primal problem. We now ask the question: Is this new primal solution $\mathbf{q}_1, \mathbf{u}_1$ optimal? If so, adding the new joint patrol did not lead to a possible increase in the primal objective value, and there was seemingly no point in adding that particular joint patrol. If on the other hand it turns out that $\mathbf{q}_1, \mathbf{u}_1$ is a sub-optimal solution to the updated problem, then adding the new joint patrol was beneficial as we can re-solve the primal to obtain a better objective value.

To check whether the new primal solution is optimal or not, we consider the dual problem. Adding the new joint patrol introduces one additional constraint to the dual, namely

$$y^q - \beta \sum_{k=1}^M r_{N+1,k}\, y_k^\beta \geq 0. \tag{3.19}$$

This can be seen from Equation (3.17c). Note that the values of $r_{N+1,k}$ belong to the new joint patrol and are known. Let consider the two possible cases:

1. **The constraint holds.** Then we see from (3.16) that the new dual objective value D_1 is unchanged since $q_{N+1} = 0$. We have $D_1 = D_0 = P_0$. Thus by the Duality theorem, the new solution is still optimal.

2. **The constraint does not hold.** Then y^q, y_k^β and y_k^ρ do not correspond to an optimal solution of the primal problem, and the primal solution may be sub-optimal.

We can conclude that when looking for a new joint patrol to add to our subset, we should choose one with inspection rates $r_{N+1,k}$ such that constraint (3.19) is *violated*. Moreover, if no such joint patrol exists we have a sufficient subset of joint patrols to find the true optimum of the master problem.

3.4.4 Subproblem

Before formulating the subproblem we make one final observation. Supposing still that we have an optimal solution of the primal (and dual) problem, the left-hand side of (3.19) corresponds to the negative derivative of the primal objective function with respect to the variable q_{N+1}. To see this, we consider equation (3.15) and recall that for an optimal solution equations (3.17b) and (3.17c) hold with equality. After adding the new joint patrol we get

$$\sum_{k=1}^{M} d_k u_k = y^q + \sum_{k=1}^{M} y_k^\rho \rho_k - \underbrace{q_{N+1}}_{=0} \underbrace{\left(y^q - \beta \sum_{k=1}^{M} y_k^\beta r_{N+1,k} \right)}_{\text{l.h.s. of (3.19)}} .$$

Here we see that if q_{N+1} is increased from zero (without considering constraint (3.14b)) then the momentary increase in the primal objective function value is equal to the left-hand side of (3.19) times -1. Hence, we should add the joint patrol that violate constraint (3.19) the most as the primal objective value increases the most in this direction.

To find the new joint patrol we must solve the following optimization problem:

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{w},\mathbf{r}_{N+1}}{\text{minimize}} \quad & y^q - \beta \sum_{k=1}^{M} r_{N+1,k}\, y_k^\beta && && (3.20a)\\
\text{subject to} \quad & \sum_{e \in \delta^+(v)} x_{j,e} + a_{j,v} = \sum_{e \in \delta^-(v)} x_{j,e} && \forall\, j,\ v, && (3.20b)\\
& w_{j,e} \le f_e\, x_{j,e} && \forall\, j, e, && (3.20c)\\
& \sum_{j=1}^{T} \sum_{e \in \Gamma(d)} w_{j,e} \le 1 && \forall\, d, && (3.20d)\\
& r_{N+1,k} = \sum_{j=1}^{T} \sum_{e \in \lambda_k} w_{j,e} && \forall\, k, && (3.20e)\\
& w_{j,e} \ge 0 && \forall\, j, e, && (3.20f)\\
& x_{j,e} \in \{0,1\} && \forall\, j, e. && (3.20g)
\end{aligned}$$

Here we have used the following shorthand notation:

$$x_{j,e} := x(\psi_{N+1,j}, e) \qquad \text{and} \qquad w_{j,e} := w(\psi_{N+1,j}, e) \qquad (3.21)$$

with vector forms

$$\mathbf{x} := [x_{j,e}]_{j=1,\dots,T, e\in E} \qquad \text{and} \qquad \mathbf{w} := [w_{j,e}]_{j=1,\dots,T, e\in E}. \tag{3.22}$$

The objective function (3.20a) is the left-hand side of dual problem constraint (3.17c), which is the negative derivative of the primal objective function with respect to the new joint patrol. Constraint (3.20b) enforces flow conservation such that $\mathbf{x}$ defines continuous paths in the transition graph between the designated start and end vertices of each patrol. We use $\delta^{\pm}(v)$ to denote the sets of edges that enter or leave vertex v, and define

$$a_{j,v} := \begin{cases} 1 & \text{if unit } j \text{ is to start at vertex } v, \\ -1 & \text{if unit } j \text{ is to end at vertex } v, \\ 0 & \text{otherwise.} \end{cases} \tag{3.23}$$

The variables in $\mathbf{w}$ are the inspection fractions belonging to the patrol paths. If edge e is not in use by team j the inspection fraction must be zero, and if it is then it can not exceed f_e as discussed in Section 3.3. This is enforced by constraint (3.20c). The condition that each train only can be inspected once is contained in constraint (3.20d). Lastly the inspection rates $\mathbf{r}_{N+1} := [r_{N+1,k}]_{k=1}^{M}$ follows the previously discussed probability model by constraint (3.20e). Constraints (3.20f) and (3.20g) makes $w_{j,e}$ and $x_{j,e}$ non-negative and binary respectively.

3.5 Solution methods

Using the procedure we present algorithmically in Algorithm 2 we are theoretically able to find the true optimal solution of the master problem in (3.14) without considering all possible joint patrols. Starting with an arbitrary joint patrol, we can iteratively generate new joint patrols that improve the primal objective value until the optimal solution of the subproblem in (3.20) has a non-negative subproblem solution. When this happens there are no joint patrols that can improve the primal objective value, and we know that the solution is optimal.

As we have mentioned the master problem is a linear optimization problem and solvable when the number of joint patrols is reasonable. The subproblem on the other hand is not a standard linear optimization problem as it contains binary variables, but a so-called *mixed integer optimization problem* (MIP). To solve the subproblem we can apply a general MIP-solver. Like for LPs we will not go into detail on general solution algorithms in this thesis, and refer to Wolsey [8] for a guide to different approaches. The problem with MIPs is that they in general are solved in *exponential time*, meaning that the computation time needed to solve the problem increases exponentially with the problem instance size. What this means to us is that solving the subproblem may only be feasible when the transition graph is small, as we will see evidence of in the next chapter.

Fortunately, we do not need to solve the subproblem exactly to improve the primal objective function value in every iteration. Solving the subproblem yields the joint patrol which looks to improve the function value *the most*, but we could

add any joint patrol that violate (3.19) and still improve the solution. Finding such a good joint patrol using *heuristic* methods can often be done significantly faster than finding the optimal solution. Heuristics may also be the only feasible way to solve to solve the subproblem for larger transition graphs. When this is the case we can not hope to find the true optimal subset $\mathbf{\Psi}_r^*$, but we may be able to find a good approximation. We will study a few basic heuristics later.

Another problem with Algorithm 2 is that the number of iterations needed to find the optimal solution is unknown and potentially large. As solving the subproblem is computationally expensive we should take steps to reduce the required number of iterations as much as possible. In our case there are no single superior strategy to do this, but there are a few simple techniques we can employ that will turn out to reduce the overall computation time considerably.

In the next chapter we look at some of the different acceleration strategies and heuristic methods for solving the subproblem, before we in the subsequent chapter gauge their performance by presenting some relevant numerical results.

Algorithm 2 Finding an optimal subset $\mathbf{\Psi}$ of joint patrols and corresponding probabilities $\mathbf{q}$ using column generation

---

$\mathbf{\Psi} \leftarrow \emptyset$
$\boldsymbol{\psi} \leftarrow$ arbitrary joint patrol
$D \leftarrow -\infty$
while $D < 0$ **do**
  $\mathbf{\Psi} \leftarrow \mathbf{\Psi} \cup \{\boldsymbol{\psi}\}$
  $(y^q, \mathbf{y}^\beta, \mathbf{y}^\rho) \leftarrow$ MASTER PROBLEM DUAL($\mathbf{\Psi}$)
  $(\boldsymbol{\psi}, D) \leftarrow$ SUBPROBLEM($y^q, \mathbf{y}^\beta, \mathbf{y}^\rho$)
end while
$(\mathbf{q}, \mathbf{u}) \leftarrow$ MASTER PROBLEM PRIMAL($\mathbf{\Psi}$)
Optimal solution is found.

---

function MASTER PROBLEM DUAL($\mathbf{\Psi}$)
  $(y^q, \mathbf{y}^\beta, \mathbf{y}^\rho) \leftarrow$ optimal solution of the master problem dual (3.17) using $\mathbf{\Psi}$
  return $(y^q, \mathbf{y}^\beta, \mathbf{y}^\rho)$
end function

---

function SUBPROBLEM($y^q, \mathbf{y}^\beta, \mathbf{y}^\rho$)
  $(\mathbf{x}, \mathbf{w}) \leftarrow$ optimal solution of the subproblem (3.20) with current $y^q, \mathbf{y}^\beta, \mathbf{y}^\rho$
  $D \leftarrow$ optimal value of subproblem objective function (3.20a)
  $\boldsymbol{\psi} \leftarrow (\mathbf{x}, \mathbf{w})$
  return $(\boldsymbol{\psi}, D)$
end function

---

function MASTER PROBLEM PRIMAL($\mathbf{\Psi}$)
  $(\mathbf{q}, \mathbf{u}) \leftarrow$ optimal solution of the master problem primal (3.14) with current $\mathbf{\Psi}$
  return $(\mathbf{q}, \mathbf{u})$
end function

Chapter 4

Heuristics and acceleration strategies

As we have now discussed we find the true optimal solution of the master problem by iteratively generating joint patrols using column generation. We have also mentioned that while the restricted master problem is linear and easy to solve, the subproblem is a mixed integer optimization problem that is likely to be very time consuming to solve in many cases. Speeding things up may not only be desirable but also necessary.

When we discuss ways to speed up the algorithm we distinguish between *heuristics* and *acceleration strategies*. We refer to techniques that reduce the number of required iterations and/or reduce the total computation time needed to find a true optimal solution as acceleration strategies. Techniques used to find *approximations* to the true optimal solution are called heuristics or heuristic methods. Acceleration strategies usually only lead to modest time savings, but are generally useful as the only associated cost is the work required to implement them. Heuristics on the other hand can lead to substantial and necessary time savings, but should be used with more care as they primarily give sub-optimal results.

4.1 Interpreting the subproblem

Many acceleration strategies and heuristics are products of practical insights about the relevant problem, and thus it can be helpful for us to re-formulate the subproblem in a more familiar form. We will do this in the following.

Let us consider the sum in the second term of the subproblem objective function (3.20a), and insert for $r_{N+1,k}$ using (3.20e). We can then write the objective as a sum over edges instead of passenger types with the help of the indicator function

$$\theta(\lambda_k, e) := \begin{cases} 1 & \text{if } e \in \lambda_k, \\ 0 & \text{otherwise.} \end{cases}$$

We rewrite:

$$
\begin{aligned}
\sum_{k=1}^{M} r_{N+1,j} y_k^{\beta} &= \sum_{k=1}^{M} \sum_{j=1}^{T} \sum_{e \in E} \theta(\lambda_k, e)\, w_{j,e}\, y_k^{\beta} \\
&= \sum_{e \in E} \sum_{j=1}^{T} w_{j,e} \underbrace{\sum_{k=1}^{M} \theta(\lambda_k, e)\, y_k^{\beta}}_{:= y_e} \\
&= \sum_{e \in E} \sum_{j=1}^{T} w_{j,e}\, y_e, \qquad (4.1)
\end{aligned}
$$

where we have defined *edge weights*

$$
y_e := \sum_{k=1}^{M} \theta(\lambda_k, e)\, y_k^{\beta}. \qquad (4.2)
$$

The edge weights reflect the dual variables. Inserting back into the objective function yields

$$
\underset{\mathbf{x},\mathbf{w}}{\text{minimize}} \quad y^q - \beta \sum_{e \in E} \sum_{j=1}^{T} w_{j,e}\, y_e. \qquad (4.3)
$$

Naturally a joint patrol that minimizes the function (4.3) also maximizes the sum in the following objective:

$$
\underset{\mathbf{x},\mathbf{w}}{\text{maximize}} \quad \sum_{e \in E} \sum_{j=1}^{T} w_{j,e}\, y_e. \qquad (4.4)
$$

By now we see that the subproblem is set in the more familiar setting of a weighted graph, as each edge transition graph edge can be given a weight y_e. Many well-studied graph problems revolve around finding paths with certain properties in weighted graphs. When we design heuristic methods we try to decompose our problem into smaller sub-problems that hopefully resemble some more well-known problem for which an algorithm exits.

A classical example of a well-studied graph problem is the *Shortest path problem*, which is the problem of finding a path between two vertices in a graph such that the sum of weights over edges in the path is minimized. As long as the graph does not contain a *negative cycle*[1], the shortest path problem is solvable and can be solved in *polynomial time*[2] (quickly). A related problem is the *Longest path problem*, which is finding a path between two vertices such that the path weights are maximized. The longest path problem is more complex than the shortest path problem. While the

[1] A sequence of vertices starting and ending in the same vertex, such that there are edges from each vertex to the next consecutive vertex (a cycle) and the sum of edge weights are negative (a negative cycle). If a negative cycle exists, a path could be infinitely short.

[2] The number of required steps is $\mathcal{O}(n^k)$ for some integer k where n is the input size, for example number of vertices or edges.

problem is solvable as long as a positive cycle does not exist, it can only be solved in polynomial time in *directed acyclic graphs*[3] (DAGs).

We may now draw some similarities between our subproblem (3.20) and the longest path problem. First of all we note that transition graphs are DAGs, as all edges point forwards in time and no cycles can exist. The longest path between two vertices in a transition graph is a path that maximizes the sum of edge weights y_e over path edges. On the other hand the solution of the subproblem is an *edge weighted-path* (a patrol), that is a path (a patrol path) in which each edge has an associated weight (the inspection fraction). Instead of maximizing the sum of edge weights y_e over path edges, we maximize *weighted sum* over path edges. The sum is weighted by inspection fractions $w_{j,e}$ for the path edges, as we can see from (4.4). We recall that $w_{j,e}$ are non-zero only if edge e is used in patrol path e.

If the requirement that each train only can be inspected once was to be disregarded, individual patrols could be found independently by solving longest path problems. We refer to this requirement simply as the *inspection requirement* in the following. Without the requirement patrol units should always inspect at their maximum capacity, i.e. $w_{j,e} = f_e\, x_{j,e}$. Inserting this into (4.4) we get

$$\sum_{e \in E} \sum_{j=1}^{T} w_{j,e}\, y_e = \sum_{e \in E} \sum_{j=1}^{T} x_{j,e}\, f_e\, y_e \qquad (4.5)$$

The objective function is then the right hand side of (4.5), which clearly makes the problem equivalent to T longest path problems where $f_e\, y_e$ are the edge weights.

To find patrols we find paths between source and sink nodes v_j^+ and v_j^- that maximize the sum in (4.5) for each patrol unit. A longest path algorithm is given as a function LONGEST PATH in Algorithm 8, found in Appendix B. The algorithm is based on the classical *DAG shortest path algorithm* that can be found in [3].

When the inspection requirement is included the problem becomes much more complex. Let us suppose for simplicity that there is only one patrol unit ($T = 1$) such that patrols can not interfere with one another. An optimal patrol path allows the sum of accompanying inspection fractions $w_{j,e}$ times edge weights y_e to be maximized. Finding such a patrol is more involved than solving a longest path problem in the transition graph and an interesting problem of its own.

If more then one patrol unit are present ($T > 1$) the situation is complicated further. The responsibility for inspecting different trains must now be delegated between patrol units and patrol paths must be coordinated such that the objective is maximized. It seems unrealistic that this problem is solvable in polynomial time, but we will not prove this claim.

4.2 Acceleration strategies

We are ready to look at ways to speed up the solution approach to the inspector scheduling problem we presented in Algorithm 2. When attempting to improve the performance of LPs and especially MIPs a natural first step is to reduce the

[3]Directed graphs that does not contain cycles.

number of variables as much as possible. A very simple time-saving measure is to insert equation (3.20e) into the subproblem objective (3.20a) to eliminate the $r_{N+1,k}$ variables, as these are easily re-computed after the problem is solved. Another potential measure is to experiment with adding different redundant constraints to the optimization problem. Sometimes adding additional constraints to MIPs can help solvers find the optimum faster. In our case such formulation improvements may be possible to implement, but we will not focus on this area in the following.

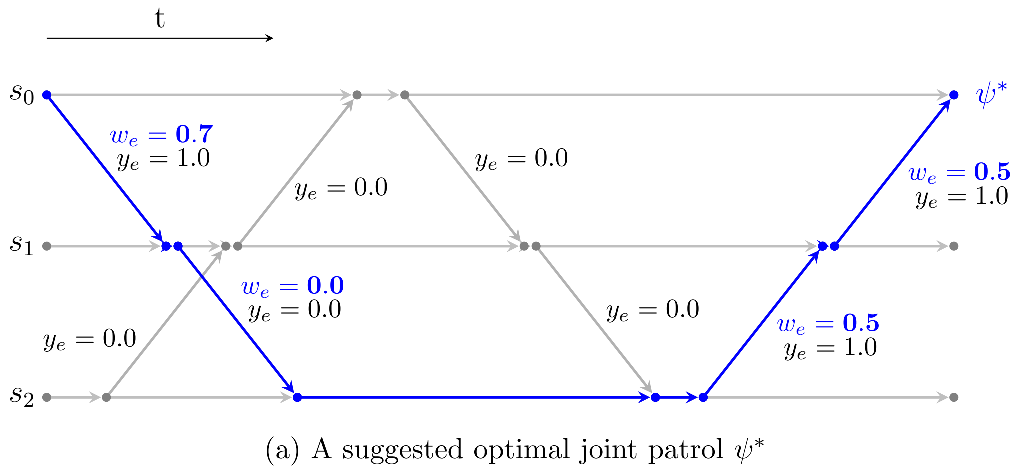
Instead we will focus on reducing the number of required subproblem iterations. In each iteration of the algorithm a joint patrol that corresponds to an optimal subproblem solution is included in the subset $\boldsymbol{\Psi}_r$. Often there are more than one optimal solution of the subproblem. Adding a different optimal solution than the one initially returned by the subproblem solver may lead to faster convergence. Optimal solutions of the subproblem are joint patrols that give the highest possible *momentary* increase in the primal objective function, and are in that sense all equally desirable. Yet, by using common sense we can say that some optimal joint patrols are better than others. We can not know in advance which optimal joint patrol to add to get the fastest possible convergence, but we can use our knowledge of the problem setting to potentially improve joint patrols suggested by the solver.

For example, suppose our subproblem solver suggests joint patrol $\boldsymbol{\psi}^* = (\mathbf{x}^*, \mathbf{w}^*)$. If $y_e = 0$ for some edge e, then from the perspective of the subproblem solver there is no point to performing inspections on this edge, even if the edge is used in the suggested optimal joint patrol. This happens for edges that are sufficiently covered such that passengers purchase tickets given the current primal solution. From our perspective it is always better to inspect than not to inspect, and as such we would always like to increase the inspection levels. If we can increase w_e^* for some train edge without violating constraints then $\boldsymbol{\psi}^*$ can be *augmented*. We increase $\mathbf{w}^*$ in a manner that maximizes some secondary objective function. For example we can maximize the total number of inspected passengers by solving the following LP:

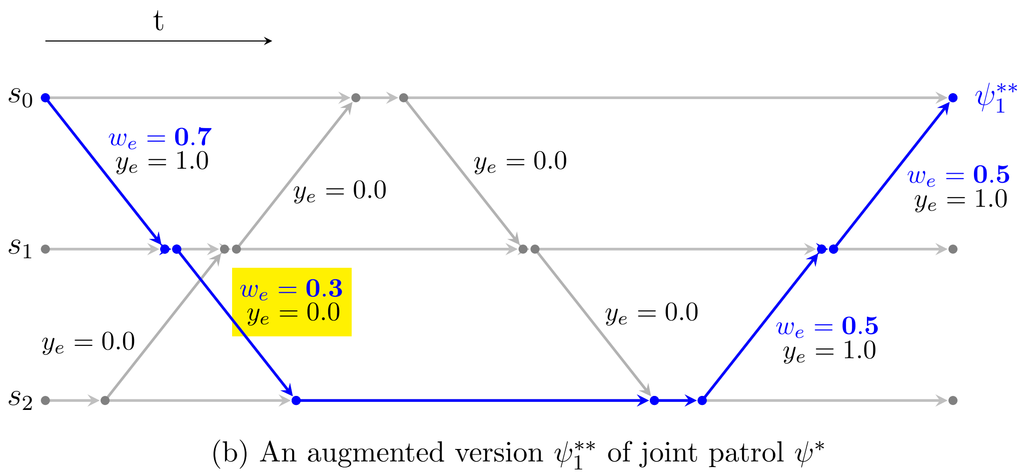
$$
\begin{aligned}
\underset{\mathbf{w}}{\text{maximize}} \quad & \sum_{e \in E} \sum_{j=1}^{T} w_{j,e}\, u_e && && (4.6\text{a}) \\
& w_{j,e} \leq f_e\, x_{j,e}^* && \forall\, j, e, && (4.6\text{b}) \\
& w_{j,e} \geq w_{j,e}^* && \forall\, j, e, && (4.6\text{c}) \\
& \sum_{j=1}^{T} \sum_{e \in \Gamma(d)} w_{j,e} \leq 1 && \forall\, d. && (4.6\text{d})
\end{aligned}
$$

Here $(\mathbf{x}^*, \mathbf{w}^*)$ is the given optimal solution of (3.20). The objective function (4.6a) is the total number of inspected passengers as u_e is the passenger volume or the number of passengers using edge e, and $w_{j,e}$ the fraction that is inspected. Constraint (4.6b) prevents units from inspect beyond their capacity and is analogous to (3.20c). To not spoil the optimality of the solution all inspection fractions in the augmented version must be greater or equal than in the suggested optimal solution. This is ensured by constraint (4.6c). Finally the inspection condition remains and is contained in (4.6d).

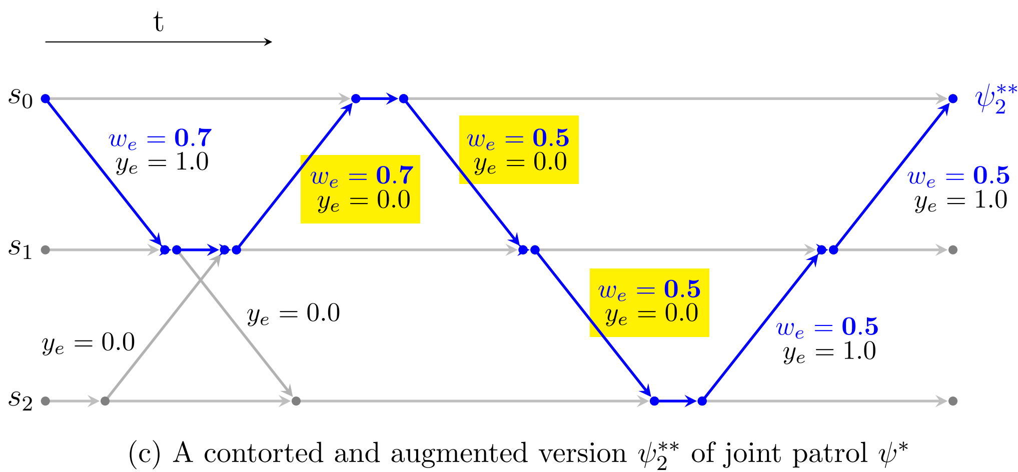
Another situation that can occur when $y_e = 0$ for some edge or edges is that joint



(a) A suggested optimal joint patrol ψ^*



(b) An augmented version ψ_1^{**} of joint patrol ψ^*



(c) A contorted and augmented version ψ_2^{**} of joint patrol ψ^*

Figure 4.1: An optimal patrol suggested by the subproblem solver and two improved versions, one augmented and one contorted. In this example $f_e = 0.7$ for all $e \in E$. All three patrols give the same objective value, but patrols in (b) and (c) are subjectively better.

patrols suggested by the solver contain unnecessary waiting edges. We prefer to have patrol units inspecting on edges with $y_e = 0$ rather than waiting, and would like to alter the patrol paths to increase the number of inspected passengers if possible. To do this we can solve the MIP given by:

$$\begin{aligned}
\underset{\mathbf{x},\mathbf{w}}{\text{maximize}} \quad & \sum_{e \in E} \sum_{j=1}^{T} w_{j,e}\, u_e && && \text{(4.7a)} \\
& x_{j,e} \geq \delta(y_e > 0)\, x^*_{j,e} && \forall\, j, e, && \text{(4.7b)} \\
& w_{j,e} \geq \delta(y_e > 0)\, w^*_{j,e} && \forall\, j, e, && \text{(4.7c)} \\
& w_{j,e} \leq f_e\, x_{j,e} && \forall\, j, e, && \text{(4.7d)} \\
& \sum_{j=1}^{T} \sum_{e \in \Gamma(d)} w_{j,e} \leq 1 && \forall\, d, && \text{(4.7e)} \\
& \sum_{e \in \delta^+(v)} x_{j,e} + a_{j,v} = \sum_{e \in \delta^-(v)} x_{j,e} && \forall\, j,\ v, && \text{(4.7f)} \\
& x_{j,e} \in \{0, 1\} && \forall\, j, e. && \text{(4.7g)}
\end{aligned}$$

The objective (4.7a) is again the total number of inspected passengers. In constraints (4.7b) and (4.7c) we use the indicator function $\delta(y_e > 0)$ which is 1 if $y_e > 0$ and zero otherwise. If $y_e = 0$ and e is used in the suggested joint patrol then this edge is fixed, otherwise the path is free to be changed. Similarly the inspection fraction is lower bounded by $w^*_{j,e}$ only if $y_e > 0$ to ensure that optimality is conserved. Constraints (4.7d), (4.7e), (4.7f) and (4.7g) are analogous to (4.6b), (4.6d), (3.20b) and (3.20g) and have all been explained previously.

Unfortunately (4.7) is a MIP and therefore as complicated as the subproblem itself. We should therefore avoid spending time on solving this problem unless it reduces the number of required iterations substantially. Alternatively an approximate solution could be found using heuristic methods, but we leave this as an area of future work. A *contorted* patrol ψ_2^{**} is shown in Figure 4.1c.

In the next chapter we will test the methods we have now described on a few constructed examples to evaluate their performance. We note that there likely are many other avenues worth exploring to improve the performance of the algorithm and reduce the number of required iterations. Without dwelling further we move on to consider heuristic methods in the next section.

4.3 Heuristic methods

We now consider ways of solving the subproblem approximately. As indicated in Section 4.1 where the problem was interpreted as a weighted graph problem, solving the problem is made complex by the inspection requirement. We also indicated that without this requirement the subproblem is reduced to T longest path problems that can be solved in polynomial time due to the fact that the transition graph is a DAG.

We will use a similar approach also when the requirement *is* included. Our plan is decompose the problem by treating each patrol unit separately and construct a

joint patrol by assigning patrols to units one by one in such a way that the inspection requirement is not violated. In this way we sacrifice optimality to find a decent joint patrol significantly faster than we could solve the subproblem.

Before we begin we create an empty joint patrol $\boldsymbol{\psi} := (\mathbf{x}, \mathbf{w})$ where $\mathbf{x} := [\mathbf{x}_j]_{j=1}^T$ and $\mathbf{w} := [\mathbf{w}_j]_{j=1}^T$. We write a single patrol as $\psi_j := (\mathbf{x}_j, \mathbf{w}_j)$, where $\mathbf{x}_j := [x_{j,e}]_{e \in E}$ and $\mathbf{w}_j := [w_{j,e}]_{e \in E}$. We let the function INITIALIZE JOINT PATROL initialize $\boldsymbol{\psi}$ and set all $x_{j,e}$ and $w_{j,e}$ to zero. Patrols $(\mathbf{x}_j, \mathbf{w}_j)$ are later updated iteratively as they are determined.

Single patrol

First we consider a heuristic for finding a patrol $\psi_j := (\mathbf{x}_j, \mathbf{w}_j)$ for a single unit j. Edges in the transition graph receive weights $\mathbf{y} := [y_e]_{e \in E}$ as defined by (4.2) in each iteration. When we find patrols we use *relevant* edge weights denoted $\bar{\mathbf{y}} := [\bar{y}_e]_{e \in E}$. These are updated several times while we construct the joint patrol as we will explain later. To obtain a single patrol path, we disregard the inspection requirement and find a path between source and sink vertices for unit j. The path can be found using the longest path algorithm in Appendix B with current relevant edge weights $\bar{\mathbf{y}}$. We find the path by calling the function LONGEST PATH with $\bar{\mathbf{y}}$,v_{source} and v_{sink} as inputs. The function returns a vector $\mathbf{x}_j$ that represents the path. Again, we have that

$$x_{j,e} := \begin{cases} 1 & \text{if edge } e \text{ is used in patrol } j, \\ 0 & \text{otherwise.} \end{cases}$$

A path found in this manner is likely to be sub-optimal due to prolonged stays on trains as the algorithm is unaware of the inspection requirement. The best possible inspection fractions $\mathbf{w}_j$ to go along with $\mathbf{x}_j$ can be found by a similar procedure as we explained in the previous section. Initially we can set all $w_{j,e}$ to zero and *augment* the patrol with respect to relevant edge weights $\bar{\mathbf{y}}$. By this we mean finding $\mathbf{w}_j$ such that the sum

$$\sum_{e \in E} w_{j,e} \, \bar{y}_e$$

is maximized. Instead of solving an LP similar to (4.6) we can apply our own procedure given as a function AUGMENT PATROL in Algorithm 10 in Appendix B. The function returns an updated vector of inspection fractions $\mathbf{w}_j$. For reasons we come back to shortly the function takes three inputs: The path $\mathbf{x}_j$, the relevant edge weights $\bar{\mathbf{y}}$ and a vector $\bar{\mathbf{z}} := [\bar{z}_d]_{d \in D}$. We here use $0 \leq \bar{z}_d \leq 1$ which is fraction of departure d that is currently uninspected by the joint patrol $\boldsymbol{\psi} = (\mathbf{x}, \mathbf{w})$ under construction.

The steps we have outlined for finding a patrol for a single unit is presented in Algorithm 3. We present the procedure as a function HEURISTIC PATROL that takes as input the relevant edge weights $\bar{\mathbf{y}}$, the current uninspected fractions of departures $\bar{\mathbf{z}}$ and the patrol number j. The resulting patrol $\psi = (\mathbf{x}_j, \mathbf{w}_j)$ is returned.

Algorithm 3 Find an approximation to the best patrol for unit j

```
function HEURISTIC PATROL(ȳ, z̄, j)
    x_j ← LONGEST PATH(ȳ, v_j^+, v_j^-)                ▷ Algorithm 8
    w_j ← AUGMENT PATROL(x_j, ȳ, z̄)                    ▷ Algorithm 10
    return (x_j, w_j)
end function
```

Joint patrol

After finding a single patrol using HEURISTIC PATROL we repeat the procedure for the next patrol unit. When we commit to using a certain patrol we say that the patrol is *fixed*. Other patrols must then be designed to complement previously fixed patrols by avoiding previously covered trains as these can not be inspected again. To discourage the path finding algorithm of using edges corresponding to these trains we update the relevant edge weights $\bar{\mathbf{y}}$ using the function UPDATE PARAMETERS in Algorithm 9. In the same function the uninspected fractions of departures $\bar{\mathbf{z}}$ are updated. Relevant edge weights $\bar{y}_e$ are set to $f_e y_e$ multiplied with the uninspected fraction $\bar{z}_d$ for their corresponding departure. If the departure is entirely uninspected then $\bar{z}_d = 1 \implies \bar{y}_e = f_e y_e$, the maximal contribution from this edge as we can see from (4.5). If the departure is completely inspected then $\bar{z}_d = 0 \implies \bar{y}_e = 0$, and the incentive to use this edge is removed. The algorithm is given in Algorithm 9 in Appendix B. The function takes edge weights $\mathbf{y}$ and current inspection fraction vector $\mathbf{w}$ as inputs, returning relevant edge weights $\bar{\mathbf{y}}$ and uninspected fractions $\bar{\mathbf{z}} := [\bar{z}_d]_{d \in D}$.

Applying HEURISTIC PATROL for all T patrol units we get a joint patrol $\boldsymbol{\psi} = (\mathbf{x}, \mathbf{w})$. In each repetition a new patrol is designed to complement previously fixed patrols. If patrol units have different working hours and/or starting places, the order in which patrols are fixed can influence the result. To determine the order patrols is fixed we employ the simple strategy of always fixing the patrol that provides the biggest increase in the objective function. In each iteration we find suggested patrols $(\hat{\mathbf{x}}_j, \hat{\mathbf{w}}_j)$ for all remaining units that have not yet been assigned patrols, and fix the one with the highest *utility* $\hat{U}_j$. HEURISTIC PATROL is thus called $T + (T-1) + \ldots + 1 = \frac{1}{2}T(T-1)$ times.

The complete heuristic method is presented in Algorithm 4 as the function HEURISTIC JOINT PATROL. The function takes edge weights $\mathbf{y}$ as input and returns the joint patrol $\boldsymbol{\psi} = (\mathbf{x}, \mathbf{w})$. We can use this heuristic in the place of the subproblem solver in each iteration of Algorithm 2 to obtain a sub-optimal subset $\boldsymbol{\Psi}$, see Algorithm 5. As the heuristic joint patrol give a sub-optimal subproblem objective function value the algorithm terminates before the optimum is found.

Before we proceed to numerical experiments in the next chapter we point out that the heuristic given in this chapter is basic, and that more advanced heuristic methods could be developed in the future. This is an interesting field of study that we unfortunately are not able to explore further in this master thesis.

Algorithm 4 Find an approximation to the optimal joint patrol $\boldsymbol{\psi} = (\mathbf{x}, \mathbf{w})$

```
function HEURISTIC JOINT PATROL(y)
    (x, w) ← INITIALIZE JOINT PATROL
    Q ← {1, ..., T}
    while |Q| > 0 do
        (ȳ, z̄) ← UPDATE PARAMETERS(y, w)
        for j ∈ Q do
            (x̂_j, ŵ_j) ← HEURISTIC PATROL(ȳ, z̄, j)
            Û_j ← Σ_{e∈E} ŵ_{j,e} y_e        ▷ The utility of suggested patrol (x̂_j, ŵ_j)
        end for
        j ← arg max_{j∈Q} {Û_j}
        (x_j, w_j) ← (x̂_j, ŵ_j)
        Q ← Q \ {j}
    end while
    ψ ← (x, w)
    return ψ
end function
```

Algorithm 5 Finding a sub-optimal subset $\boldsymbol{\Psi}$ of joint patrols and corresponding probabilities $\mathbf{q}$ using column generation and a heuristic method to solve the sub-problem

```
Ψ_r ← ∅
ψ ← arbitrary joint patrol
D ← −∞
while D < 0 do
    Ψ_r ← Ψ_r ∪ {ψ}
    (y^q, y^β, y^ρ) ← MASTER PROBLEM DUAL(Ψ_r)
    y ← Edge weights computed from y^β according to (4.2)
    ψ ← HEURISTIC JOINT PATROL(y)
    D ← Value of ψ computed by inserting into (4.3)
end while
(q, u) ← MASTER PROBLEM PRIMAL(Ψ_r)
Sub-optimal solution is found.
```

Chapter 5

Numerical results

It is now time to test and evaluate the solution approaches to the inspector scheduling problem we presented in the preceding chapters. We will perform various experiments to examine the behaviour of the column generation procedure and to determine when finding optimal solutions is computationally feasible. To evaluate our heuristic method we will compare the heuristic solutions with their optimal counterparts.

For our experiments we will use three different example networks inspired by the NSB local train network in Figure 1.1. The networks are designed to expose strengths and weaknesses of our solution method and will have varying levels of detail and realism. We implemented our solution in Python using the GNU Linear Programming Kit (GLPK)[1] to solve optimization problems with the PyMathProg modelling language[2]. The code is found in Appendix C. All experiments were performed on Intel(R) Xeon(R) CPU X7542 @ 2.67 GHz processors. When we present computation times of optimization problems we count the time spent setting up and solving the problems.

5.1 Basic example network

The first example network we consider is the relatively simple train network shown in Figure 5.1. We call this the basic example network. In this example there are six stations that correspond to stations in the south eastern part of the NSB network. The three train lines ℓ^0, ℓ^1, ℓ^2 represent real world lines *L14*, *L21* and *L22* respectively. As this simple network is only meant to give us a basic sense of how the procedure performs we have left out several stations and parts of the train lines.

For all example networks in this chapter we use train time tables based on current time tables for non-peak hours found on NSB's home page[3]. We let all trains wait for $t_{wait} = 1$ minute at each stop. The time tables we use are only close to the actual time tables but serve a realistic examples. All trains travel at regular intervals of either $10, 20, 30$ or 60 minutes such that all time tables are repeated hourly. We will study the example for different durations as this affects the size of the transition

[1] `http://www.gnu.org/software/glpk/`
[2] `http://pymprog.sourceforge.net/`
[3] `www.nsb.no`

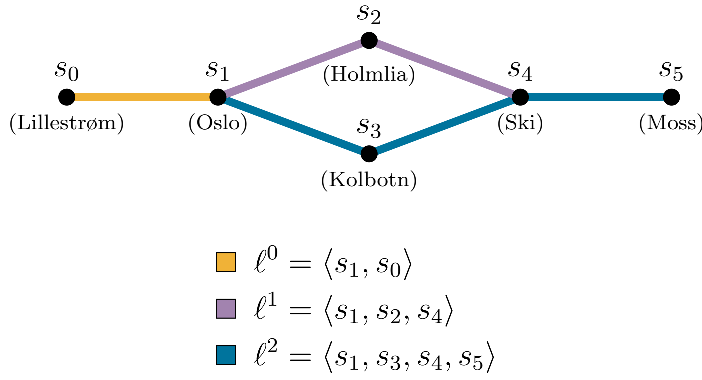


Figure 5.1: The basic example network, inspired by the real-world network in Figure 1.1. There are six station $s_0, \ldots s_5$ and three lines ℓ_0, ℓ_1, ℓ_2.

Table 5.1: An hourly train time table used with the example network in Figure 5.1.

Line	Direction	Departure time	Departure frequency
ℓ_0	$+1$	xx:04	20 minutes
ℓ_0	-1	xx:06	20 minutes
ℓ_1	$+1$	xx:18	60 minutes
ℓ_1	-1	xx:31	60 minutes
ℓ_2	$+1$	xx:31	60 minutes
ℓ_2	-1	xx:38	60 minutes

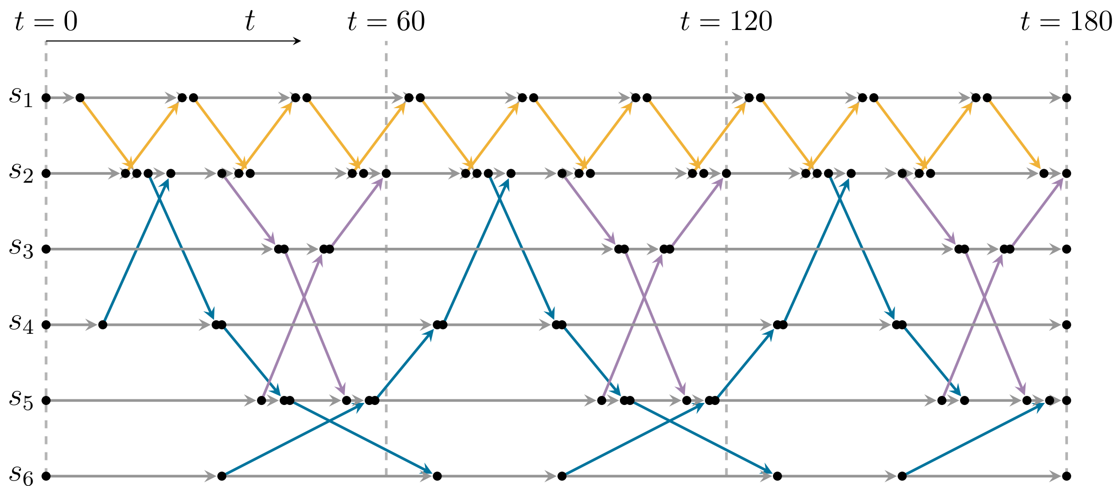


Figure 5.2: The transition graph corresponding to the train network in Figure 5.1 and three repetitions of the time table in Table 5.1.

graph and space of possible patrols. In Figure 5.2 we see the transition graph corresponding to three repetitions of the time table given in Table 5.1.

Passenger types λ_k are found using Algorithm 7 in Appendix B and corresponding demands d_{λ_k} are set to realistic values as real world data is unavailable to us. The ticket pricing structure used by NSB is hard to mimic, and in all our examples we use a fixed ticket price $\rho = 1$ for any journey. We will try different fine sizes β and observe the effects. We let patrol units consist of teams of three inspectors that each can inspect seven passengers per minute, i.e. $\mu = 3 \cdot 7 = 21$. The inspection effectiveness f_e follows the simple model in (2.1). Unless stated otherwise we always use two patrol units ($T = 2$). In the basic example units have overlapping working hours, both starting in $v^+ = (s_1, 0)$ and ending in $v^- = (s_1, t_{max})$.

5.2 Acceleration strategies

To validate the acceleration strategies presented in Section 4.2 we run the column generation procedure from Algorithm 2 on the basic test problem with time windows of 3, 4 and 5 hours. (The transition graph for three hours is the one shown in Figure 5.2.) To test the acceleration strategies we find the optimum with three different strategies:

- None: No acceleration strategy is applied.
- Augment: Augmenting joint patrols by solving (4.6).
- Contort/Aug.: Contort joint patrols by solving (4.7) until stagnation, then use Augment.

Our initial experiments revealed that a pure contortion strategy can cause the algorithm to stagnate. This happens when a newly added contorted joint patrol gets a sampling probability of zero in the restricted master problem. When this happens the solution is effectively unchanged from the previous iteration, returning identical dual variables. This causes the subproblem solver to generate the same joint patrol again in the next iteration, at which point the same thing happens again. Why this occurs is unclear to us as contortion does was verified to not affect the subproblem objective value. To avoid this behaviour we switch from contortion to augmentation whenever $q_N = 0$ in the master problem solution.

The results from all three strategies are shown in Table 5.2, and we a sample joint patrol is illustrated Figure 5.3. We see from the table that both acceleration strategies reduce the number of iterations and computation time required to find optimal solutions. Using the augmentation strategy almost halves the required number of iterations and more than halves the total computation time in all three cases. The contortion strategy yields slightly higher iteration numbers and computation times in two of three experiments and does slightly better in the experiment with a four hour time window. Given the overall performance of the strategy we are lead to conclude that the contortion strategy is inferior to the augmentation strategy. We will use augmentation in all the following numerical tests.

Table 5.2: Testing acceleration strategies on the basic example network with $\beta = 3$ and three different durations.

Hours	Strategy	Iterations	Comp. time	Optimal value
3	None	66	23.7 s	4842.4
	Augment	35	10.8 s	4842.4
	Contort/Aug.	36	11.1 s	4842.4
4	None	109	170.0 s	6824.0
	Augment	56	66.2 s	6824.0
	Contort/Aug.	54	66.0 s	6824.0
5	None	139	1851.8 s	8807.2
	Augment	77	711.6 s	8807.2
	Contort/Aug.	103	1004.4 s	8807.2

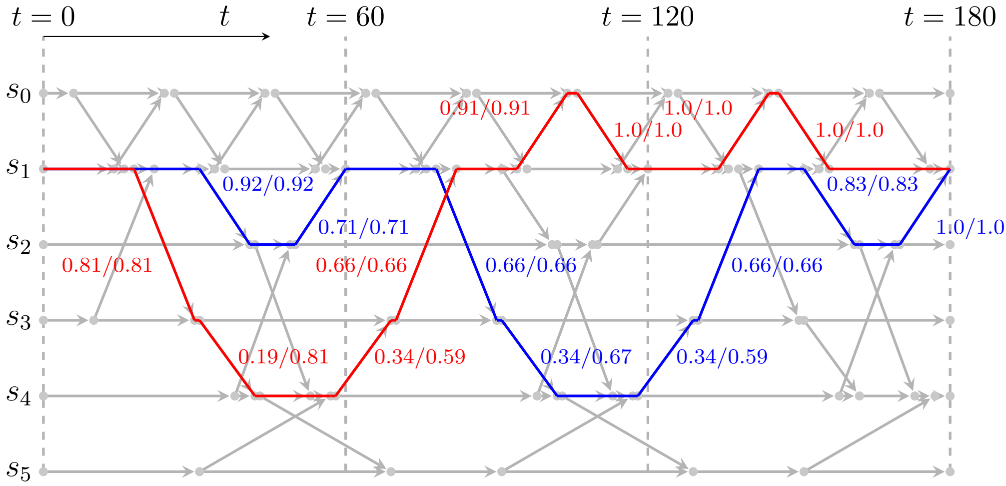


Figure 5.3: A joint patrol for the basic example problem with a three hour time window. Two units are illustrated, one in red and one in blue. Inspection fractions $w_{j,e}$ and their upper bounds (the edge effectiveness) f_e are presented in the format "$w_{j,e}/f_e$".

5.3 Scalability and behaviour

Another observation that can be made from Table 5.2 is that the time required to find optimal solutions greatly increases when the size of the transition graph increases. When time window is increased from three to five hours, the size of the transition graph (the number of edges) almost doubles. At the same time we notice that the computation time increases from 10.8 seconds to 711.6 seconds. This indicates that the computation time is exponential, and we will most likely hit a computational wall when we increase the problem size. We will examine the scalability of the solution approach further below.

In Table 5.3 we present iteration numbers and computation times for the basic example with durations of 3 to 5 hours with one to three patrol units. Units always start and end their patrols at the same times and places. We also present the number of joint patrols that are given nonzero sampling probabilities, denoted by $|\mathbf{\Psi}_{>0}|$.

Table 5.3: Testing different numbers of patrol units and time windows in the basic example network with $\beta = 3$.

Hours	$\lvert E\rvert$	Patrol units	Iter.	$\lvert\mathbf{\Psi}_{>0}\rvert$	Comp. time	Optimal value
3	140	1	13	3	2.0 s	3967.8
		2	35	19	10.8 s	4842.4
		3	15	13	6.6 s	4957.0
4	188	1	23	6	6.2 s	5537.8
		2	54	29	55.1 s	6824.0
		3	72	25	599.3 s	7034.5
5	236	1	51	8	23.0 s	7106.6
		2	77	36	707.5 s	8807.2
		3	121	30	27446.1 s	9109.7

In all cases the number of joint patrols used in the optimal solution is considerably lower than the number of iterations. The unused joint patrols have served their purpose by helping the column generation find an optimal solution an can now be discarded from $\mathbf{\Psi}_r$ as they are no longer needed.

Again we see the trend that computation times increase drastically as the problem size increases. We have included number of transition graph edges $|E|$ in each experiment to see how it plays a role. The number of patrol units T appears to be critical for the complexity of the problem. Computing the optimal solution for $T = 3$ with a five hour duration took over seven hours. Interestingly when the time window is three hours the optimum is found faster for $T = 3$ than for $T = 2$. This is due to the fact that the network is adequately saturated when three patrol units are present, i.e. there are no fare evaders on edges patrol units can reach.

We prove this last claim by running the experiment with $T = 2$ for different the fine sizes. The results of this experiment are shown in Table 5.4. For a three hour duration we see that as the fine size increases the optimal value converges to the same value as for $T = 3, \beta = 3$. As the fine size increases the impact of the patrol units increases until eventually all passengers pay for their tickets. The fact that the objective value does not increase when the fine size increases indicates strongly that the maximum value has been reached.

We can also notice from the results that the computation time varies significantly for different fine sizes. The fine size we have used in the above experiments appears to cause the longest computation times. When the impact of the inspectors are either very low or very high (as measured by the expected cost of fare evading) the computation is completed relatively quickly. The closer the optimal value is to the theoretical maximum the longer the computation takes as larger numbers of joint patrols are needed. When the fine size is higher than necessary to discourage all fare evaders the problem becomes easier to solve as few or no joint patrols have to be discarded.

To find out what causes the high computation times we take a closer look at how the procedure performs iteration by iteration. We run the algorithm on the basic example with two patrol units and fine size $\beta = 3$ for different time windows. The master problem and subproblem objective values and computation times in each

Table 5.4: Testing different fine sizes and time windows for the basic network with $T = 2$.

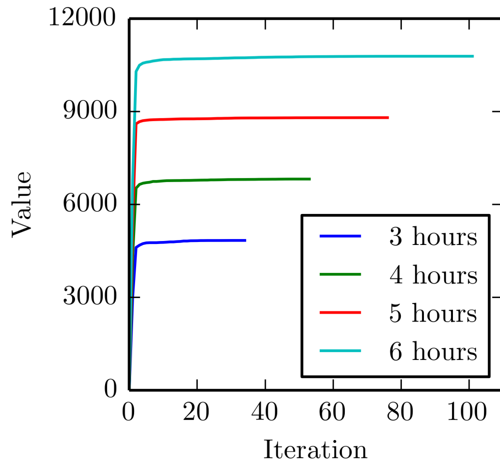
Hours	$\|E\|$	Fine size β	Iter.	$\|\mathbf{\Psi}_{>0}\|$	Comp. time	Optimal value
3	140	1	15	7	3.9 s	2928.7
		2	12	8	3.2 s	4297.4
		3	35	19	10.8 s	4842.4
		4	28	15	7.2 s	4946.2
		5	10	8	2.0 s	4957.0
		6	7	7	1.4 s	4957.0
4	188	1	17	10	11.2 s	4116.9
		2	25	11	19.9 s	6287.3
		3	54	29	57.8 s	6996.5
		4	57	26	52.3 s	7043.0
		5	14	12	4.3 s	7043.0
		6	7	7	2.0 s	7043.0
5	236	1	20	13	52.8 s	5300.8
		2	44	15	194.8 s	8077.2
		3	77	36	705.6 s	8077.2
		4	59	29	382.6 s	9046.7
		5	25	22	14.1 s	9129.0
		6	8	8	5.1 s	9129.0

iteration are illustrated in Figure 5.4.

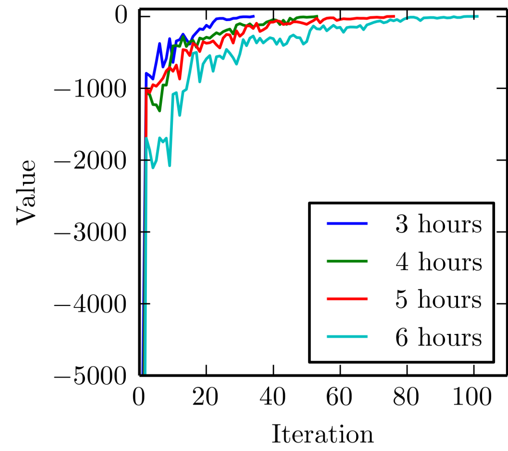
The master problem objective values are given in Figure 5.4a. We see that the value increases quickly at first before flattening out significantly. Almost all the progress are made in the first few iterations, close inspection reveals that the objective value is within 98 % of the optimum after just 5 iterations in all four cases. The objective increases very slowly towards the en until finally reaching the optimum. In Figure 5.4b we can see the subproblem objective values which exhibit a similar but more erratic behaviour. The values vary from iteration to iteration but are tending towards zero from below.

In Figure 5.4c we see that the master problem computation times. The computation time increases almost linearly with the iteration number (the number of joint patrols). In this example the longest master problem computation time is close to 1.5 seconds. On the other hand the longest observed subproblem computation time is just above 450 seconds, as we can see from Figure 5.4d. We see that the computation time varies from iteration to iteration and behaves unpredictably. In the final experiment with a six hour time window the computation time rises sharply towards the end.

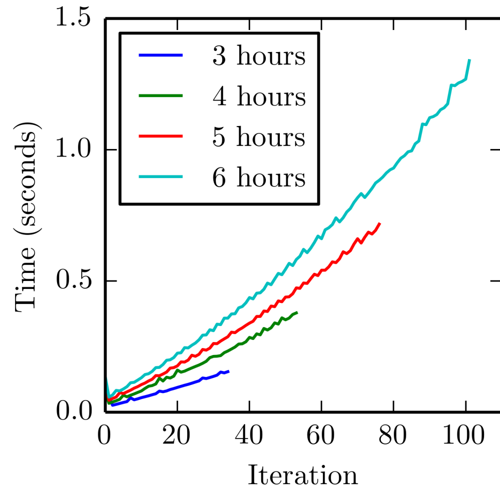
To summarize, most of the progress in the column generation procedure is made in the first few iterations. The computation time per iteration may also increase drastically towards the end. To illustrate the situation further we present in Table 5.5 the computation time until 99 % of the optimum is reached. Compared to the time required to find optimal solutions close-to-optimal solutions are found very quickly. We do also notice that the time required to find 99 % optimal solutions seem



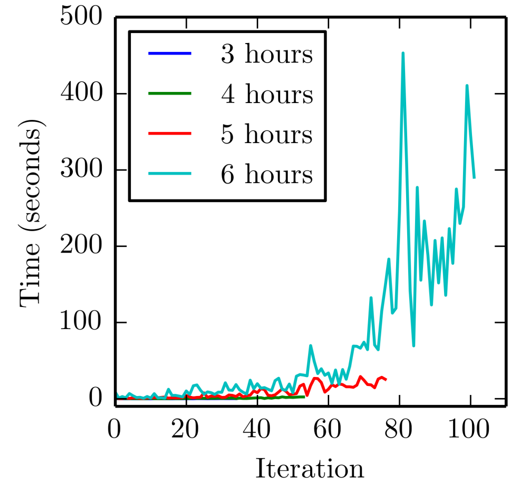
(a) Master problem objective values



(b) Subproblem objective values



(c) Master problem computation times



(d) Subproblem computation times

Figure 5.4: Performance data for column generation procedure applied to the basic example with $T = 2$, $\beta = 3$ and four different time windows.

Table 5.5: Time required to find a solution within 99 % of the optimal solution in the basic example with $T = 2$, $\beta = 3$ for different time windows.

Hours	Optimal value	99 % optimal time	Total time	Time fraction
3	4842.4	2.6 s	10.7 s	24.2 %
4	6824.0	4.1 s	57.6 s	7.2 %
5	8807.2	5.7 s	705.6 s	0.8 %
6	10789.8	35.4 s	7499.1 s	0.5 %

to increase quite fast with the transition graph size, but not relative to the total computation time. This means that the tailing off effect observed for the master problem objective values becomes more severe when the problem size increases, increasing the number of required iterations. Finding near-optimal solutions by performing only a few iterations may eventually also become infeasible.

The results we have presented so far indicate that the complexity of the subproblem is substantial, and it appears that applying the procedure directly to real-world scenarios where the level of detail is much greater will be problematic. The computation time varies depending on a number of factors and has a generally unpredictable nature. We see that near-optimal solutions can be found with only a few column generation iterations, but we fear that the computation time of even a single iteration will increase beyond acceptable limits as the problem size increases. Heuristic methods are necessary to solve these problems.

5.4 Heuristic methods

We will now measure the performance of the heuristic method presented in Chapter 4. First we compare heuristic results with optimal solutions from the previous section. We apply the heuristic to the same basic example problem with two patrol units and fine size $\beta = 3$ for different durations. The results are presented in Table 5.6. Here OPTIMAL refers to finding solution using the AUGMENT acceleration strategy from Section 5.2 and HEURISTIC to the heuristic method.

Table 5.6: Testing the heuristic on the basic example network with $T = 2$, $\beta = 3$ and different time windows.

Hours	$\|E\|$	Solver	Iter.	$\|\boldsymbol{\Psi}_{>0}\|$	Comp. time	Value	% optimal
1	44	OPTIMAL	3	3	0.2 s	824.0	
		HEURISTIC	3	3	< 0.1 s	824.0	100 %
2	92	OPTIMAL	13	8	1.8 s	2857.0	
		HEURISTIC	15	10	0.4 s	2856.9	> 99.9 %
3	140	OPTIMAL	35	19	10.8 s	4842.4	
		HEURISTIC	15	10	0.9 s	4801.5	99.2 %
4	188	OPTIMAL	56	29	66.2 s	6824.0	
		HEURISTIC	29	15	3.1 s	6771.5	99.0 %
5	236	OPTIMAL	77	36	771.6 s	8807.2	
		HEURISTIC	35	21	5.9 s	8706.0	98.9 %
6	284	OPTIMAL	102	42	7499.1 s	10789.7	
		HEURISTIC	30	23	5.5 s	10666.8	98.9 %
7	332	OPTIMAL	N.A.	N.A.	> 7 hours	N.A.	
		HEURISTIC	46	28	13.8 s	12569.6	N.A.
8	380	OPTIMAL	N.A	N.A.	N.A	N.A.	
		HEURISTIC	84	37	54.3 s	14573.9	N.A.

The results reveal that the heuristic works well on the basic example, finding solutions that are around 99 % of optimal in very reasonable amounts of time. Un-

fortunately finding the optimal solution for durations of seven hours or longer turned out to be infeasible, and we can not evaluate the performance of the heuristic objectively in these cases. We see that the heuristic uses fewer iteration and produces fewer joint patrols than the optimal solver in each experiment. This is because the algorithm terminates early when the heuristic is unable to find a negative solution of the subproblem.

To expose shortcomings of the heuristic we perform new test on two more elaborate example networks. The first network consists of the line *L1* and the east half of line *L13* from Figure 1.1, and is illustrated in Figure 5.5. We refer to the network as the medium sized example network, and show the transition graph for two hours in Figure 5.6. The time table mimics the real time table currently in use on these lines. The third network includes all regular stations east of Oslo S on lines *L1*, *L12*, *L13*, *L14* and *R10* from Figure 1.1. We call this the large example network. For space reasons we omit the train map and only show the transition graph for one hour in Figure 5.7. In both examples there are two patrol units, both starting and ending their patrols in $t = 0$ and $t = t_{max}$. One unit starts and ends at s_0 (Oslo S) while the other starts and ends at s_{12} (Lillestrøm).

In Table 5.7 we present optimal and heuristic solutions for both networks. This time the optimal solution can only be computed for very modest time durations as the level of detail is higher. To verify that the heuristic indeed also works on longer time windows, we perform test with eight and twelve hour durations. This gives us an idea of how the heuristic performs, but we can not measure its degree of success in these cases.

Table 5.7: Testing the heursitic on more detailed example networks with $T = 2$, $\beta = 4$ and different time windows. Comparisons are made when optimal solutions can be found.

Network	Hours	$\|E\|$	$\|\Lambda\|$	Solver	Iter.	Time	Value	% opt.
Medium	1	421	709	Opt.	34	23.0 s	1227.0	
				Heur.	8	1.5 s	1212.6	98.8 %
	2	843	2567	Opt.	93	864.8 s	3563.4	
				Heur.	10	8.6 s	3357.9	94.2 %
	8	1993	6500	Heur.	24	356.1 s	15832.8	N.A.
	12	2985	9956	Heur.	23	705.5 s	24235.0	N.A.
Large	1	421	709	Opt.	16	15.1 s	2368.9	
				Heur.	11	4.1 s	2328.0	98.3 %
	2	843	2567	Opt.	68	2015.7 s	7713.0	
				Heur.	10	25.1 s	6403.6	83.0 %
	8	3375	16094	Heur.	7	474.9 s	28229.1	N.A.
	12	5063	25182	Heur.	16	2704.6 s	43794.0	N.A.

First we consider the results from the medium sized example network. For both small time windows the heuristic finds a solution relatively quickly. When the time window is two hours the heuristic solution value is within 94.2 % of the optimal value. When we use time window of eight and twelve hours the computation time increases up to 705.5 seconds. For the large example network we see similar results.

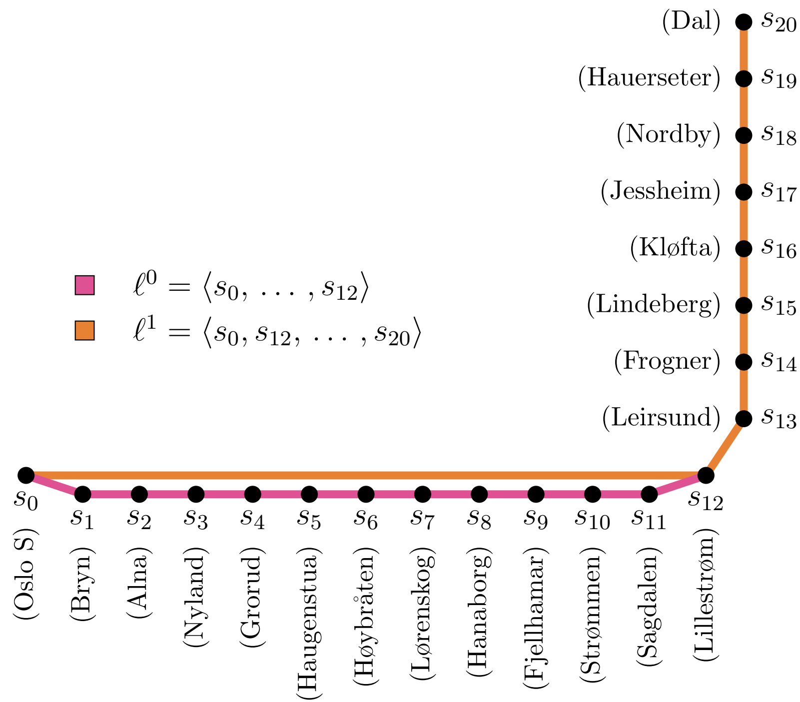


Figure 5.5: The medium example network.

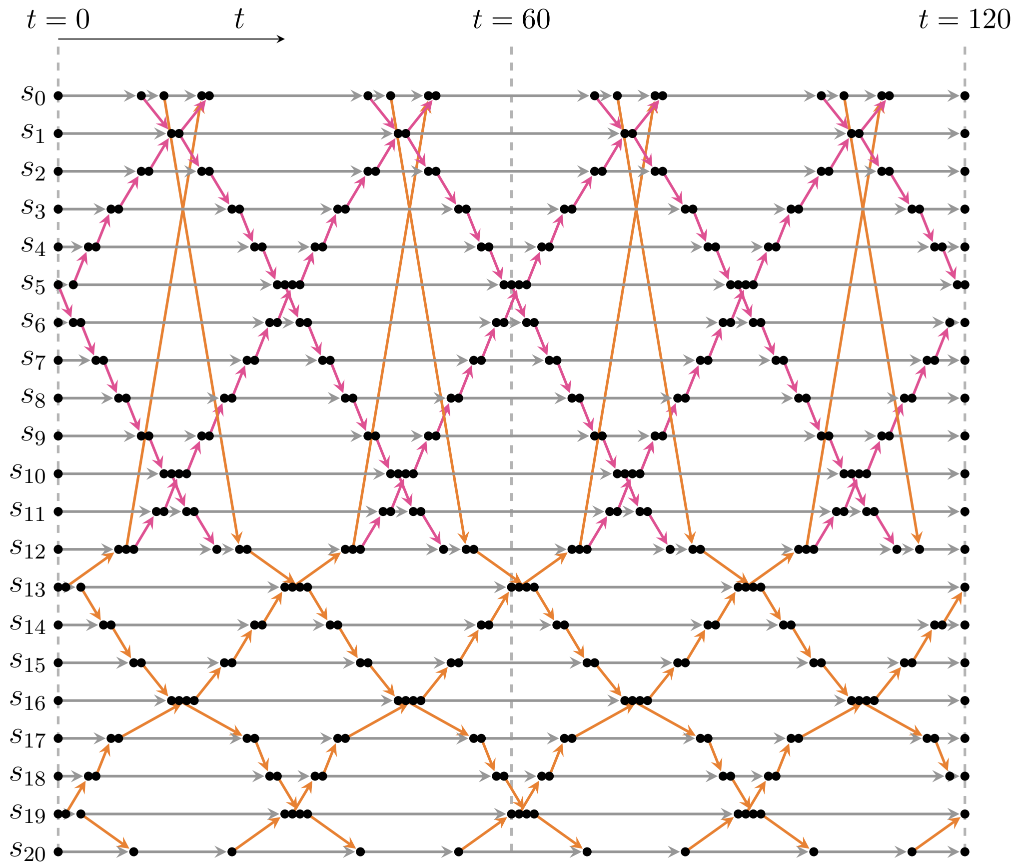


Figure 5.6: Transition graph for two hours of the time table corresponding to the medium example network in Figure 5.5.

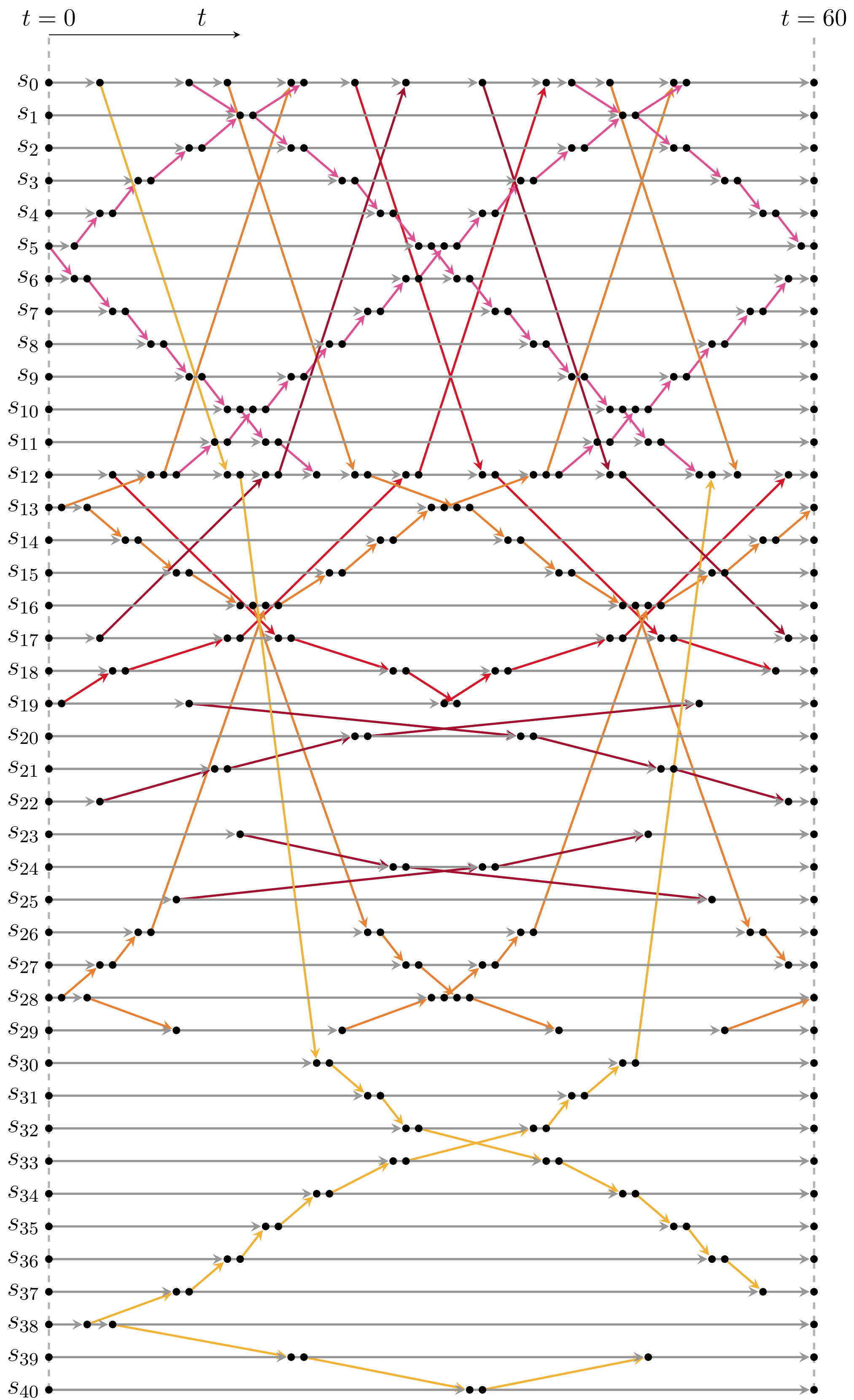


Figure 5.7: Transition graph for one hour of the time table corresponding to the large example network.

The heuristic solutions are found within seconds for small time windows, but when the time window is twelve hours the computation time rises to about 45 minutes. We also notice that the heuristic does poorly for the two hour time window, achieving a value that is 83 % of the optimum. Although we were unable to judge the optimality of the heuristic solutions for the larger examples, we can always shed light on the quality of solutions by for example computing the resulting fare evasion rates or the average income per passenger.

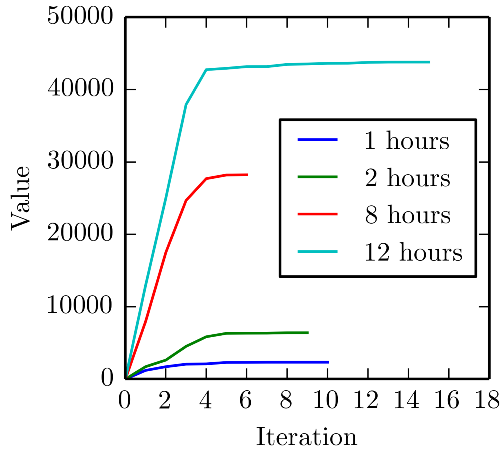
The reason for the observed drop in performance for the large example network is unclear, but could be related to the path finding procedure used in the heuristic. When paths are generated in the heuristic method the inspection requirement is neglected. This can cause patrols to stay on trains longer than they should when trains only can be inspected once. In the basic example network train lines are very short and thus the effects of this are minimal. We will discuss a potential way of dealing with this issue in the next chapter.

To find the source of the higher heuristic computation times we plot the master problem and subproblem values and computation times in each iteration for the experiments on the large example problem. The results are illustrated in Figure 5.8. The objective values for both problem are shown in Figures 5.8a and 5.8b and behave similarly to when the optimal solver is applied. We do however notice that the subproblem objective value spikes in the last iteration in the last two experiments. The heuristic is unable to generate good joint patrols here, causing a premature termination of the column generation procedure.

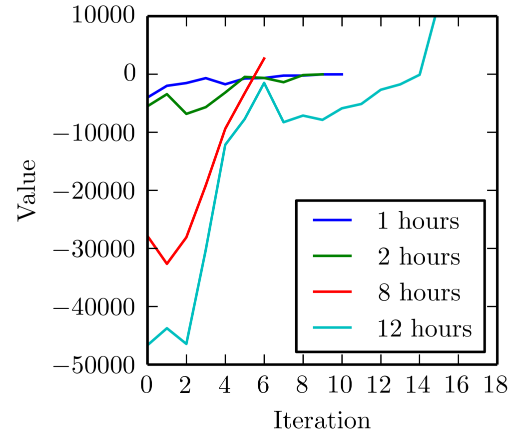
In Figures 5.8c and 5.8d the computation times for both problems in each iteration is shown. This time it is the master problem that is the most time consuming to solve. The cause of this is most likely the high number of passenger types $|\Lambda|$ in this network. We must point out that in our implementation the master problem must be set up from scratch in every iteration, and this causes the computation time to be artificially high. Using more elegant solvers this problem can be avoided.

As for the subproblem we see that even for the largest transition graph the problem is solved in less than half a second, suggesting that there is a considerable amount of leeway for more advanced heuristic methods. It is worth pointing out that our chosen optimization problem solver is currently among the slowest MIPs solvers available according to Mittelmann [7]. We see that commercial solvers outperform GLPK which is freely available. We expect this trend to hold true for LPs as well, and that finding heuristic solutions for large transition graphs is possible with the appropriate tools.

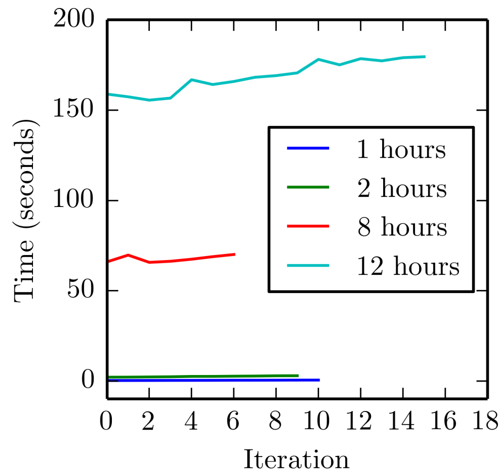
The results presented in this section indicate that solving the inspector scheduling problem using heuristic methods is a viable option. We were able to generate joint patrols for a section of the NSB local train network with realistic levels of detail, but we have refrained from attempting to find solutions for the entire network or a complete working day. It is uncertain whether this would be computationally feasible even with the simple heuristic we have presented.



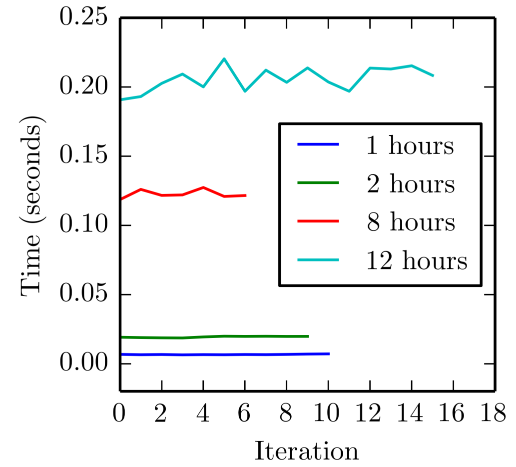
(a) Master problem objective values



(b) Subproblem objective values



(c) Master problem computation times



(d) Subproblem computation times

Figure 5.8: Performance data for column generation procedure with the heuristic subproblem solver applied to the large example with $T = 2$, $\beta = 4$ and four different time windows.

Chapter 6

Concluding remarks and further work

In this thesis we have presented the inspector scheduling problem and a recent solution method applied in the Los Angeles Metro Rail system. With the aim of solving the same problem in the NSB local train network in the south eastern part of Norway we have suggested a different solution method based on column generation. To deal with the computational complexity of finding optimal solutions we have developed and presented a heuristic method. Both methods have been tested on three different example network with promising results.

6.1 Model improvements

The procedure we have presented can be used as the foundation for a future inspector scheduling application. In its present state the solution approach is very basic but there are many additional features that could be incorporated to widen its applicability. Due to the fact that patrols are represented explicitly we can add different kinds of restrictions with relative ease. As a few examples we list support for lunch breaks, time buffers between inspections and flexibility in starting and ending positions for patrol units.

Our method can also serve as a decision making tool for service operators. Parameters can be varied in experiments to investigate the effects of for example different patrol unit constellations and fine sizes. Passenger behaviour can be made more realistic by introducing different groups of passengers as mentioned in Chapter 2. We could also apply different objective functions to design patrols with other goals in mind.

We have only given passengers the choice of either purchasing single tickets or fare evading in this thesis, but other types of tickets could also be included. For example a commuter would base his or hers decision of purchasing a subscription ticket on the expected cost of fare evading over a period of time rather than for a single trip. Giving passengers different ticket options may be possible by introducing different passenger types for different tickets.

6.2 Heuristic improvements

The key area of future work lies in ensuring that high quality solutions can be found in realistic settings. To achieve this we feel that more advanced heuristic methods should be developed. A weakness of the current heuristic is the way in which patrol paths are determined. To find patrol paths the inspection requirement is neglected, potentially leading to inadequate paths. We have seen that the performance of the heuristic drops on networks with realistic levels of detail and suspect that this is a contributing cause.

We suggest a way of dealing with this issue. Paths are currently found using a version of the DAG shortest path algorithm applied on the weighted transition graph. Edges are given weights based on the restricted master problem dual variables, and paths that maximize the sum of weights are suggested as patrol paths. We would like the path finder algorithm to take the inspection requirement into consideration to prevent patrol units from needlessly staying on inspected trains.

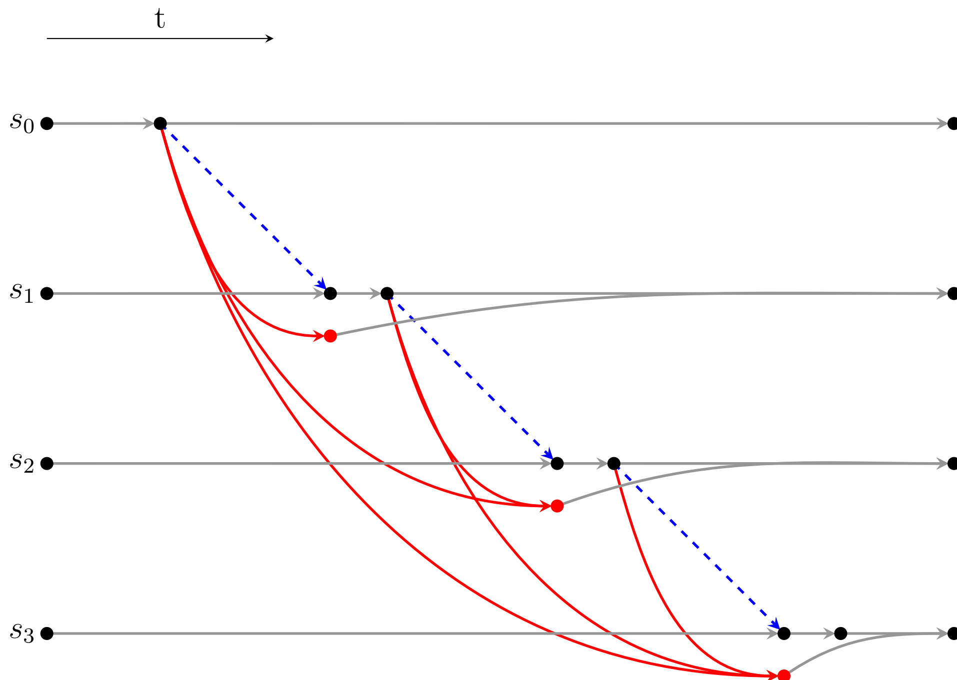


Figure 6.1: An altered transition graph where long edges and exiting vertices (in red) replace ordinary train edges (dashed blue edges). The path finding algorithm could be applied to this graph instead of the ordinary transition graph to potentially obtain better results.

To do this we propose applying the same path finding algorithm on an altered transition graph. An example of such a graph is given in Figure 6.1. Suppose we are considering a vertex v in the transition graph that corresponds to a specific departure d. The main idea is replacing the train edges with *long edges* from v to all consecutive vertices v' that correspond to arrivals of the same train. These are shown in red in the figure. The long edges represent stays of different duration and can be given weights according to the optimal inspection on corresponding edges in

the original transition graph. In this way the inspection requirement is incorporated. To prevent patrol units of using more than one long edge per departure we introduce *exiting vertices* along side regular vertices corresponding to train departures. Exiting vertices are connected to the next consecutive vertices for their respective stations as indicated in Figure 6.1, where the exiting vertices are illustrated as red dots. Ordinary train edges are shown as dashed blue arrows.

By letting all long edges start in regular vertices and end in exiting vertices, using more than one long edge per departure is impossible. After using the path finding algorithm on the altered transition graph we can translate the path back to the original transition graph, hopefully obtaining a better patrol path. The downside of this approach is that the number of transition graph edges increases drastically. It may also be time consuming to assign the correct weights to all edges. However, given that the computational bottleneck of the current heuristic is solving the master problem and not the subproblem, the impact of this drawback may not be very big in practice.

Another possible cause of poor performance for the heuristic could be that patrols are not coordinated very well. A potential improvement to consider would be a mechanism that lets patrol units cooperate to a larger extent when generating patrol paths. Currently patrols are generated greedily one by one by letting each new patrol work around previously fixed patrols, and it may be possible to device some way for patrols to be adapted to one another dynamically.

6.3 Other further work

As we commented in the introduction of this thesis, the inspector scheduling problem is ambiguous and approachable from different angles. We have assumed that passengers know the probability of being inspected and make rational decisions based on this knowledge. The former assumption is rather strong, and can only hold true when the same scheduling scheme is used over long periods of time. An interesting idea would be to device a system for registering fare evasion levels and dynamically shifting inspection levels to different areas as needed. Alternatively one could model the *perceived probability* of being inspected and let passengers base their actions on this instead.

A problem related to scheduling inspections is challenge of designing a work schedule for a crew of inspectors. We assumed that patrol unit starting times and places were fixed, but this requirement could be relaxed. Usually working hours are determined separately as working hours are regulated by strict laws. Combining this problem with the inspector scheduling problem is conceivable and could potentially lead to better results.

Bibliography

[1] A. Caprara, L. Kroon, M. Monaci, M. Peeters, and P. Toth. Passenger railway optimization. In C. Barnhart and G. Laporte, editors, *Handbook in OR & MS*, volume 14, pages 129–187. Elsevier B. V., 2007.

[2] V. Chvátal. *Linear Programming*. W. H. Freeman and Company, 1983. ISBN 9780716715870.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction To Algorithms*. MIT Press, 2001. ISBN 9780262032933.

[4] F. M. D. Fave, M. Brown, C. Zhang, E. Shieh, A. X. Jiang, H. Rosoff, M. Tambe, and J. Sullivan. Security Games in the Field: an Initial Study on a Transit System. In *Proc. of the 13th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, May 2014. Available at: `http://teamcore.usc.edu/papers/2014/dellefave_aamas_2014_camera_ready.pdf`.

[5] A. X. Jiang, Z. Yin, C. Zhang, M. Tambe, and S. Kraus. Game-theoretic Randomization for Security Patrolling with Dynamic Execution Uncertainty. In *Proc. of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, May 2013. Available at: `http://teamcore.usc.edu/papers/2013/aamas13-execution.pdf`.

[6] S. Luber, Z. Yin, F. D. Fave, A. X. Jiang, M. Tambe, and J. P. Sullivan. Game-theoretic Patrol Strategies for Transit Systems: the TRUSTS System and its Mobile App (Demonstration). In *Proc. of the 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS)*, May 2013. Available at: `http://teamcore.usc.edu/papers/2013/AAMAS_2013_camera_ready.pdf`.

[7] H. Mittelmann. Mixed integer linear programming benchmark (miplib2010), March 2014. URL `http://plato.asu.edu/ftp/milpc.html`.

[8] L. Wolsey. *Integer Programming*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 1998. ISBN 9780471283669.

[9] Z. Yin, A. X. Jiang, M. P. Johnson, M. Tambe, C. Kiekintveld, K. Leyton-Brown, T. Sandholm, and J. P. Sullivan. TRUSTS: Scheduling Randomized Patrols for Fare Inspection in Transit Systems. In *Proc. of the 24th Conference on Innovative Applications of Artificial Intelligence (IAAI)*, 2012. Available at: `http://teamcore.usc.edu/papers/2012/iaai12-trusts.pdf`.

Appendix A

Primal and dual problem

In this section we derive the dual LP to the primal LP defined by (3.14). To do this we first write the primal on the *standard form*

$$\begin{aligned} \max_x \quad & c^T x \\ \text{such that} \quad & Ax = b, \\ & x \geq 0. \end{aligned} \tag{A.1}$$

The dual of a maximization problem in standard form is given by

$$\begin{aligned} \min_y \quad & y^T b \\ \text{such that} \quad & y^T A \leq c^T. \end{aligned} \tag{A.2}$$

To explicitly state the dual of (3.14) we start by rewriting the inequality constraints (3.14c) and (3.14d) as equality constraints by introducing *slack variables* s_k^β and s_k^ρ such that

$$u_k + s_k^\beta = \beta \sum_{k=1}^{M} r_{i,k}\, q_i \qquad \forall\, k, \tag{A.3}$$

$$u_k + s_\varphi^\rho = \rho_k \qquad \forall\, k. \tag{A.4}$$

The slack variables have vector forms

$$\mathbf{s}^\beta = \left[s_k^\beta \right]_{k=1}^{M} \qquad \text{and} \qquad \mathbf{s}^\rho = [s_k^\rho]_{k=1}^{M}$$

and must all be positive for the original constraints to be satisfied:

$$s_k^\beta, s_k^\rho \geq 0 \qquad \forall\, k.$$

The variable vector x and the cost vector c in (A.1) can now be specified as

$$x = \begin{bmatrix} \mathbf{u} \\ \mathbf{q} \\ \mathbf{s}^\beta \\ \mathbf{s}^\rho \end{bmatrix} \qquad \text{and} \qquad c = \begin{bmatrix} \mathbf{d} \\ \mathbf{0}_{N\times 1} \\ \mathbf{0}_{M\times 1} \\ \mathbf{0}_{M\times 1} \end{bmatrix}, \tag{A.5}$$

where

$$\mathbf{d} := [d_k]_{k=1}^{M} \tag{A.6}$$

and $\mathbf{0}_{m\times n}$ is matrix of zeros with dimensions $m \times n$, in this case a column vector. We now have that $x, c \in \mathbb{R}^{N_x}$ with $N_x := N + 3M$ and

$$c^T x = \sum_{k=1}^{M} d_k u_k.$$

The next task is to write the constraints (3.14b), (A.3) and (A.4) on the form $Ax = b$. We begin by introducing

$$A_1 := \begin{bmatrix}\mathbf{0}_{1\times M} & \mathbf{1}_{1\times N} & \mathbf{0}_{1\times M} & \mathbf{0}_{1\times M}\end{bmatrix} \qquad \text{and} \qquad b_1 := [1],$$

where similarly $\mathbf{1}_{m\times n}$ is a matrix of ones with dimensions $m \times n$, in this case a row vector. This way $A_1 x = b_1$ is equivalent to (3.14b). Next we write (A.3) in matrix form

$$\begin{bmatrix}I_{M\times M} & \mathbf{0}_{M\times N} & I_{M\times M} & \mathbf{0}_{M\times M}\end{bmatrix}\begin{bmatrix}\mathbf{u}\\ \mathbf{q}\\ \mathbf{s}^{\beta}\\ \mathbf{s}^{\rho}\end{bmatrix} = \begin{bmatrix}\mathbf{0}_{M\times M} & \beta P & \mathbf{0}_{M\times M} & \mathbf{0}_{M\times M}\end{bmatrix}\begin{bmatrix}\mathbf{u}\\ \mathbf{q}\\ \mathbf{s}^{\beta}\\ \mathbf{s}^{\rho}\end{bmatrix}.$$

Here we have used

$$R := \begin{bmatrix} r_{1,1} & \dots & r_{N,1}\\ \vdots & \ddots & \vdots \\ r_{1,M} & \dots & r_{N,M}\end{bmatrix} \in \mathbb{R}^{M\times N}. \tag{A.7}$$

Moving everything over to the left hand side we can write

$$A_2 := \begin{bmatrix}I_{M\times M} & -\beta R & I_{M\times M} & \mathbf{0}_{M\times M}\end{bmatrix} \qquad \text{and} \qquad b_2 := \begin{bmatrix}\mathbf{0}_{M\times 1}\end{bmatrix}$$

such that $A_2 x = b_2$ is equivalent with (A.3). Finally we define

$$A_3 := \begin{bmatrix}I_{M\times M} & \mathbf{0}_{M\times N} & \mathbf{0}_{M\times M} & I_{M\times M}\end{bmatrix} \qquad \text{and} \qquad b_3 := [\boldsymbol{\rho}]$$

where

$$\boldsymbol{\rho} := [\rho_k]_{k=1}^{M}, \tag{A.8}$$

such that $A_3 x = b_3$ is equivalent to (A.4). Using

$$A = \begin{bmatrix}A_1\\ A_2\\ A_3\end{bmatrix} = \begin{bmatrix}\mathbf{0}_{1\times M} & \mathbf{1}_{1\times N} & \mathbf{0}_{1\times M} & \mathbf{0}_{1\times M}\\ I_{M\times M} & -\beta R & I_{M\times M} & \mathbf{0}_{M\times M}\\ I_{M\times M} & \mathbf{0}_{M\times N} & \mathbf{0}_{M\times M} & I_{M\times M}\end{bmatrix} \quad \text{and} \quad b = \begin{bmatrix}b_1\\ b_2\\ b_3\end{bmatrix} = \begin{bmatrix}1\\ \mathbf{0}_{M\times 1}\\ \boldsymbol{\rho}\end{bmatrix} \tag{A.9}$$

we finally have that $Ax = b$ and $x \geq 0$ is equivalent to the original constraints given by Equations (3.14b) - (3.14f). We can now write out the dual problem (A.2) using

the A-matrix in (A.9):

$$
\begin{aligned}
\underset{y^q, \mathbf{y}^\beta, \mathbf{y}^\rho}{\text{minimize}} \quad & y^q + \sum_{k=1}^{M} \rho_k \, y_k^\rho && && \text{(A.10a)} \\
\text{subject to} \quad & y_k^\beta + y_k^\rho \geq d_k, && \forall \, k, && \text{(A.10b)} \\
& y^q - \beta \sum_{k=1}^{M} r_{i,k} \, y_k^\beta \geq 0, && \forall \, i, && \text{(A.10c)} \\
& y_k^\beta, y_k^\rho \geq 0, && \forall \, k. && \text{(A.10d)}
\end{aligned}
$$

Appendix B

Algorithms

Algorithm 6 Constructing V, E

```
V ← ∅                                                        ▷ Start with empty sets
E ← ∅
for (ℓ^j, t_0, σ) ∈ D do
    t' ← t_0
    if σ = +1 then
        s^j_m ← s^j_k for m = k from 1 to n_j                  ▷ Keep the station order
    else if σ = −1 then
        s^j_m ← s^j_{n_j+1−k} for m = k from 1 to n_j          ▷ Reverse the station order
    end if
    for m from 1 to n_j − 1 do
        v  ← (s^j_m, t')
        v' ← (s^j_{m+1}, t' + τ(s, s', ℓ^j)
        V ← V ∪ {v, v'}
        E ← E ∪ {(vv')}
        t' ← t' + t_wait
    end for
end for
V ← V ∪ {(s, 0) : s ∈ S} ∪ {(s, t_max) : s ∈ S}              ▷ Adding start/end vertices
for s_i ∈ S do                                                ▷ Adding waiting edges
    V_i ← {v = (s, t) : v ∈ V, s = s_i}
    T̃_i ← {t : (s, t) ∈ V_i}
    t_1 ← 0
    repeat |V_i| − 1 times
        T̃_i ← T̃_i \ {t_1}
        t_2 ← min({t : t ∈ T̃_i})
        E ← E ∪ {(vv') = ((s_i, t), (s_i, t')) : v, v' ∈ V, t = t_1, t' = t_2}
        t_1 ← t_2
    end repeat
end for
```

Algorithm 7 Constructing Λ

```
Λ ← ∅
V_sorted ← {v_k = (s_k, t_k) : k = 1, ... |V|, v_k ∈ V, t_k ≥ t_{k+1}}
                                              ▷ Reverse topological sort of V
for k = 1, ... , |V_sorted| do                ▷ Iterate over vertices in backwards order
    for s' ∈ S \ {s_k} do                     ▷ Find paths to all other stations
        λ ← shortest path from (s_k, t_k) to some v' = (s', t')
        if there is a λ' ∈ Λ starting in some v = (s_k, t) ending in v' then
            Do nothing                        ▷ λ is not sensible
        else
            Λ ← Λ ∪ {λ}
        end if
    end for
end for
```

Algorithm 8 Find the longest path from v_{source} to v_{sink} in the transition graph $G = (V, E)$ with edge weights $\bar{\mathbf{y}}$

```
function LONGEST PATH(ȳ, v_source, v_sink)
    V_sorted ← {v_i = (s_i, t_i) : i = 1 ..., n, v_i ∈ V, t_i ≤ t_{i+1}, v_0 = v_source, v_n = v_sink}
                                                        ▷ Topological sort of vertices
    for i = 1, ..., n do
        dist_{v_i} ← −∞                                 ▷ Distance from source v_j^+ to v_i
        pred_{v_i} ← (÷)                ▷ Predecessor to vertex v_i, if unknown then (÷)
    end for
    dist_0 ← 0
    pred_0 ← (+)                        ▷ No predecessor to the source, signify by (+)
    for i = 0, ..., n do
        for e = (v_i, u) ∈ δ^+(v_i) do
            if dist_u < dist_{v_i} + ȳ_e then
                dist_u ← dist_{v_i} + ȳ_e
                pred_u ← v_i
            end if
        end for
    end for
    u ← v_n
    while pred_u ≠ (+) do   ▷ Constructing the path backwards from predecessors
        e ← (pred_u, u)
        path ← [ e | path ]
        u ← pred_u
    end while
    x_j ← [δ(e ∈ path)]_{e∈E}
    return x_j
end function
```

Comments:
Note that a longest path problem in a DAG G is equivalent to the shortest path problem in $-G$, where all edge weights are multiplied with -1. Also, in this algorithm vertices are *topologically sorted* in V_{sorted}. A topological sort is an ordering of vertices such that for any vertex in the ordering none the preceding vertices can be *reached*. That is, there exits no path from this vertex to preceding vertices in the topological sort.

Algorithm 9 Update edge weights $\bar{\mathbf{y}} = [\bar{y}]_{e \in E}$ and uninspected fractions $\bar{\mathbf{w}} = [\bar{w}_d]_{d \in D}$

```
function UPDATE PARAMETERS(y, w)
    for d ∈ D do
        z̄_d ← 1 − Σ_{e∈Γ(d)} Σ_{j=1}^{T} w_{j,e}
        for e ∈ Γ(d) do
            ȳ_e ← f_e y_e z̄_d
        end for
    end for
    return (ȳ, z̄)
end function
```

Algorithm 10 Find the optimal inspection fractions $\hat{\mathbf{w}}_j$ for a patrol path $\hat{\mathbf{x}}_j$ by maximizing the sum of $\bar{\mathbf{y}} = [\bar{y}_e]_{e \in E}$

```
function AUGMENT PATROL(x̂_j, ȳ, z̄)
    ŵ_j ← 0
    for d ∈ D do
        Q ← {e ∈ Γ(d) ∩ {e : x̂_{j,e} = 1}
        ẑ_d ← z̄_d                      ▷ Temporary uninspected fraction of departure d
        while ẑ_d > 0 and |Q| > 0 do
            e ← arg max_{e∈Q}{ȳ_e}
            Q ← Q \ {e}
            w̃ ← min{ẑ_d, f_e − ŵ_{j,e}}          ▷ Maximum legal increase in ŵ_{j,e}
            ẑ_d ← ẑ_d − w̃             ▷ Update temporary uninspected fraction ẑ_d
            ŵ_{j,e} ← ŵ_{j,e} + w̃       ▷ Update potential inspection fraction ŵ_{j,e}
        end while
    end for
    return ŵ
end function
```

Appendix C

Python code

Here we include the Python code written to perform the numerical experiments. We have the following files:

1. `solve_single.py` – The main script.
2. `optimization.py` – Solving of optimization problems, used by 1
3. `heuristics.py` – The heuristic method, used by 1
4. `classes.py` – Objects used in 2 and 3
5. `NSBbasic.py` – Initializes the basic example network, used in 1
6. `NSBmedium.py` – Initializes the medium sized example network, used in 1
7. `NSBlarge.py` – Initializes the large example network, used in 1
8. `mapconstruct.py` – Constructs transition graphs, used by 4, 5 and 6.

solve_single.py

```
import pickle
import time
import sys, os
# Local:
import optimization as opt
import heuristics as heu
# Maps
import NSBbasic
import NSBmedium
import NSBlarge

# Output levels
showlvl  = 0
printlvl = 0

if showlvl > 0: import trainplot as tplot

def main():

  fine = 3 # Fine size
  reps = 3 # Number of hours
  insp = 3 # Number of inspectors per unit
  ntms = 2 # Number of teams for the basic network
```

```
  # Solver: 'opt' or 'heur'
  solver   = 'opt'

  # Acceleration strategy: 'none', 'augment' or 'contort'
  strategy = 'contort'

  # Network: 'basic', 'medium' or 'large'
  network     = 'basic'

  solve(fine, reps, insp, ntms, solver, strategy, network)

def solve(fine, reps, insp, ntms, solver = 'heur', strategy = 'augment', network =
    'basic'):

  postfixstr = ""
  warning = ""
  # Get the map
  if network == 'basic':
    postfixstr += "B" + str(ntms) + "_"
    (M,G,Dlists,Tlists,Phi,lookup,param) = NSBbasic.get_map(fine, reps, insp, ntms)
  elif network == 'medium':
    postfixstr += "M_"
    (M,G,Dlists,Tlists,Phi,lookup,param) = NSBmedium.get_map(fine, reps, insp)
  elif network == 'large':
    postfixstr += "L_"
    (M,G,Dlists,Tlists,Phi,lookup,param) = NSBlarge.get_map(fine, reps, insp)
  else:
    raise Exception("Invalid map name!")

  Psi = []
  P   = []

  x0 = [[0] * param.num_edges for i in xrange(param.num_teams)]
  w0 = [[0] * param.num_edges for i in xrange(param.num_teams)]
  bpsi = [ (x0[t],w0[t]) for t in xrange(param.num_teams) ]
  p     = [0] * param.num_types

  candidate_bpsi = [bpsi]
  candidate_p    = [p]

  # Master problem
  loop = True
  it   = 0

  mpval, spval, mptime, sptime = [],[],[],[]

  t0 = time.time()
  told, tnew = 0, 0
  while loop:

    # Updating sets
    for i in xrange(len(candidate_bpsi)):
      Psi.append(candidate_bpsi[i])
      P.append(candidate_p[i])

    told = time.time()
    (qval,mpobj,yb,yq) = opt.masterproblem(Psi, P, Phi, param, printlvl = printlvl)
    tnew = time.time()
    mptime.append(tnew-told)  # Time spent solving the master problem
    mpval.append(mpobj)       # Value of the master problem objective

    told = time.time()
    if solver == 'opt': (bpsi,p,spobj,status) = opt.subproblem(G, Dlists, Tlists,
        Phi, lookup, param, yb, yq, post = strategy, printlvl = printlvl )
    if solver == 'heur': (bpsi,p,spobj,status) = heu.subproblem(G, lookup, Tlists,
        Dlists, Phi, param, yb, yq )
    tnew = time.time()
    sptime.append(tnew-told)  # Time spent solving the subproblem
    spval.append(spobj)       # Value of the subproblem objectuve
```

```
    if status.flag == 0:
      print "Exiting main loop: ", status.message
      break
    candidate_bpsi = [bpsi]
    candidate_p    = [p]

    # Print status
    print it, ", Master: ", mpobj, ", Sub: ", spobj, "nz: ", sum([int(i>1e-10) for
        i in qval])

    # Emergency brake
    if qval[-1] == 0 and strategy == 'contort':
      status.message += "Contort caused stalling, switched to augment. "
      print "Warning: Switched from contort to augment"
      strategy = 'augment'
      warning = 'stall'
    it += 1

  print ""
  print "Val: ", mpobj
  print ""
  t1 = time.time()
  print "Time spent: ", t1-t0
  pass

if __name__ == '__main__':
  main()
```

optimization.py

```
import pymprog
import time
# Local
import classes

def masterproblem(Psi, P, Phi, param, printlvl = 3):
    """Solve the restricted master problem with subset Psi"""
    mp = pymprog.model('masterproblem')
    u = mp.var(xrange(param.num_types),'U')
    q = mp.var(xrange(len(P)),'Q',bounds=(0,1))

    mp.max( sum(u[j]*Phi.demand[j] for j in xrange(param.num_types)) )
    constraint = []
    # q-constraints
    qsum = mp.st( sum(q[j] for j in xrange(len(P))) == 1 )
    tt = time.time()
    # u-constraints
    for i in xrange(param.num_types):
      mp.st( u[i] <= param.ticket )
      c = mp.st( u[i] <= param.fine*sum(q[j]*P[j][i] for j in xrange(len(P))) )
      constraint.append(c)

    mp.solve()

    mpobj = mp.vobj()
    qval = [ q[i].primal for i in xrange(len(P)) ]
    # Dual variables
    yb = [constraint[i].dual for i in xrange(param.num_types)]
    yq = qsum.dual
    return(qval, mpobj, yb, yq)

def subproblem(G, Dlists, Tlists, Phi, lookup, param, yb, yq, post = 'augment',
    printlvl = 3):
  """Solve the subproblem for dual variables yb, yq"""
  status = classes.StatusMessage()
  # Find y_e
  ye = [0] * param.num_edges
  for k in xrange(param.num_types):
    for i in Phi.Plists[k]:
```

```
      ye[i] += yb[k]
  # Define problem and variables
  sp = pymprog.model('subproblem')
  x = sp.var(xrange(param.num_teams),'X')
  w = sp.var(xrange(param.num_teams),'W')
  for t in xrange(param.num_teams):
    x[t] = sp.var(xrange(param.num_edges),kind=bool)
    w[t] = sp.var(xrange(param.num_edges),bounds=(0,1))
  # sum of y not greater than 1 on each departure
  for i in [a for a in xrange(len(Dlists)) if len(Dlists[a].d) > 0 ]:
    sp.st( sum(sum(w[t][j] for j in Dlists[i].d) for t in xrange(param.num_teams))
        <= 1 )
  # w is less than x and f
  for i in xrange(param.num_edges):
    for t in xrange(param.num_teams):
      sp.st( w[t][i] <= lookup.effect[i]*x[t][i] )
  # x defines a path
  for t in xrange(param.num_teams):
    for i in xrange(param.num_nodes):
      sp.st( sum(x[t][j] for j in lookup.e_in[i]) + int(Tlists[t][0] == i) - int(
          Tlists[t][1] == i) == sum(x[t][j] for j in lookup.e_out[i]) )
  # Primary objective function
  sp.min( yq - param.fine * sum( sum(w[t][i] * ye[i] for i in xrange(param.
      num_edges)) for t in xrange(param.num_teams) ) )
  tt = time.time()
  sp.solve()
  spobj = sp.vobj()

  tol = 1e-9
  # Check whether solution is worth adding to Psi or not
  if sp.vobj() < -tol:
    status.flag = 1
    status.message = "Found negative subproblem solution"
    if post == 'augment':
      ## Augment the joint patrol
      # Secondary objective
      sp.max( sum( sum(w[t][i] * lookup.evolume[i] for i in xrange(param.num_edges)
          ) for t in xrange(param.num_teams) ) )
      # Additional constraints
      [  sp.st( w[t][i] >= w[t][i].primal for t in xrange(param.num_teams) for i in
           xrange(param.num_edges) if w[t][i].primal < 1 ) ]
      [  sp.st( w[t][i] == 1 for t in xrange(param.num_teams) for i in xrange(param
          .num_edges) if w[t][i].primal >= 1 ) ]
      [  sp.st( x[t][i] == x[t][i].primal for t in xrange(param.num_teams) for i in
           xrange(param.num_edges) ) ]
      tt = time.time()
      sp.solve()
    elif post == 'contort':
      ## Contort the joint patrol
      ye = [ int( lookup.e2d[i] != -1 ) * sum( yb[k] for k in lookup.e2t[i] ) for i
           in xrange(param.num_edges)]
      # Secondary objective
      sp.max( sum( sum(w[t][i] * lookup.evolume[i] for i in xrange(param.num_edges)
          ) for t in xrange(param.num_teams) ) )
      # Additional constraints
      [ sp.st( w[t][i] >= int( ye[i] > 0 ) * w[t][i].primal for t in xrange(param.
          num_teams) for i in xrange(param.num_edges) if w[t][i] < 1 ) ]
      [ sp.st( w[t][i] == 1 for t in xrange(param.num_teams) for i in xrange(param.
          num_edges) if w[t][i].primal >= 1 and ye[i] > 0 ) ]
      [ sp.st( x[t][i] >= int( ye[i] > 0 ) * x[t][i].primal for t in xrange(param.
          num_teams) for i in xrange(param.num_edges) ) ]
      tt = time.time()
      sp.solve()

    # Get the solution on the right form
    xvec = [ [x[t][i].primal for i in xrange(param.num_edges) ] for t in xrange(
        param.num_teams) ]
    wvec = [ [w[t][i].primal for i in xrange(param.num_edges) ] for t in xrange(
        param.num_teams) ]
    bpsi = [ (xvec[t],wvec[t]) for t in xrange(param.num_teams) ]
```

```
    pvec = [ sum( sum(w[t][i].primal for i in Phi.Plists[k] ) for t in xrange(param
        .num_teams) ) \
             for k in xrange(param.num_types) ]
  else:
    status.flag = 0
    status.message = "Positive solution of subproblem"
    bpsi, pvec = -1, -1  # Return _something_
  return (bpsi,pvec,spobj,status)
```

heuristics.py

```
from heapq import heappush, heappop, heapify
from sys import maxint
import copy
# Local
import classes
import trainplot as tplot # Used to make plots

def subproblem(G, lookup, Tlists, Dlists, Phi, param, yb, yq,):
  """Find a heuristic solution to the subproblem"""
  status = classes.StatusMessage()
  # Find y_e
  ye = [ int( lookup.e2d[i] != -1 ) * sum( yb[k] for k in lookup.e2t[i] ) for i in
      xrange(param.num_edges)]
  # Initialize variables
  bpsi = [ ([0] * param.num_edges, [0] * param.num_edges) for t in xrange(param.
      num_teams)]
  Q = set(range(param.num_teams))
  while Q:
    H = []  # Empty heap
    for t in Q:
      # Source and sink
      iv_so = Tlists[t][0]
      iv_si  = Tlists[t][1]
      # Update parameters
      (ybar, zbar) = update_parameters(lookup, param, Dlists, bpsi, ye)
      #  Heuristic path
      xj = dagshortestpath(G, lookup, param, iv_so, iv_si, ybar)
      wj = augment_patrol(param, lookup, Dlists, xj, ybar, zbar)
      psi = (xj, wj)
      Uj = sum( wj[i] * ye[i] for i in xrange(param.num_edges))     # Find utility
          of psi
      # Push into heap
      heappush(H, (-Uj, psi, t))
    # Find the psi with highest utility
    (dummy, psi, a) = heappop(H)
    # Add this psi to bpsi
    bpsi[a] = copy.deepcopy(psi) # Fix the psi
    Q.remove(a)
  # Objective value
  spobj = yq - param.fine * sum( sum( bpsi[t][1][i] * ye[i] for i in xrange(param.
      num_edges) ) for t in xrange(param.num_teams) )
  # Find probability
  pvec = [ sum( sum(bpsi[t][1][i] for i in Phi.Plists[k] ) for t in xrange(param.
      num_teams) ) for k in xrange(param.num_types) ]
  if spobj > -1e-10:
    status.flag = 0
    status.message += "Positive heuristic solution of subproblem"
  else:
    status.flag = 1
    status.message = "Found negative heuristic subproblem solution"
  return (bpsi, pvec, spobj, status)

def update_parameters(lookup, param, Dlists, bpsi, ye):
  zbar = [ 1 - sum(sum(bpsi[t][1][i] for t in xrange(param.num_teams)) \
        for i in Dlists[nd].d) for nd in xrange(param.num_deps) ]
  ybar = [0] * param.num_edges
  for nd in xrange(param.num_deps):
    for i in Dlists[nd].d:
      r = 0
```

```
      ybar[i] = zbar[nd] * ( r * ye[i] + (1-r) * ye[i] * lookup.effect[i] )
  return (ybar, zbar)

def augment_patrol(param, lookup, Dlists, xj, ybar, zbar):
  # Paths in bpsi
  pathset = set([i for i in xrange(param.num_edges) if xj[i] == 1])
  wj = [0] * param.num_edges
  for nd in xrange(param.num_deps):
    Q = pathset.intersection( set(Dlists[nd].d) )
    zhat = zbar[nd]
    while zhat > 0 and Q:
      (dummy, ie) = max( sorted([(ybar[i], i) for i in Q]) )
      Q.remove(ie)
      use = min(zhat, lookup.effect[ie])
      zhat   -= use
      wj[ie] += use
  return wj

def dagshortestpath(G, lookup, param, start, stop, ybar):
  first = lookup.order.index(start)
  last  = lookup.order.index(stop)
  dist = {}
  pred = {}
  for iv in lookup.order:
    dist[iv] = -maxint
    pred[iv] = -1
  dist[lookup.order[first]] = 0
  for iv in lookup.order[first:last]:
    for ie in lookup.e_out[iv]:
      iu = lookup.v2i[G.Elist[ie][1]]
      addition = ybar[ie]
      if dist[iu] < dist[iv] + addition:
        dist[iu] = dist[iv] + addition
        pred[iu] = iv
  # Backtrace the path
  node = stop
  path = [stop]
  while not pred[node] == -1:
    node = pred[node]
    path.append(node)
  path.reverse()
  # Make edge path from vertex path
  path_edge = [ lookup.e2i[(G.Vlist[path[i]],G.Vlist[path[i+1]])] for i in xrange(
      len(path)-1)]
  # Make x_j-vector from edge path
  xj = [int(i in path_edge) for i in xrange(param.num_edges)]
  return xj
```

classes.py

```
class StatusMessage(object):

  def __init__(self):
    self.flag    = -1
    self.message = ""
class Job(object):
  def __init__(self,level,value,fixstatus,fixedpaths):
    self.level = level
    self.value = value
    self.fixstatus = fixstatus
    self.fixedpaths = fixedpaths
class TrainLine(object):
  def __init__(self,l,c):
    self.l = l
    self.c = c
class TrainDeparture(object):
  def __init__(self,d,c):
    self.d = d
    self.c = c
class TrainMap(object):
```

```
  def __init__(self,S,L):
    self.S = S
    self.L = L
class TransitionGraph(object):
  def __init__(self,Vlist,Elist):
    self.Vlist = Vlist
    self.Elist = Elist
class Lookup(object):
  def __init__(self):
    self.e_out, self.e_in, self.v2i, self.e2i, self.e2d, self.e2t, self.evolume,
        self.effect, self.order = [], [], [], [], [], [], [], [], []
class Parameters(object):
  def __init__(self):
    self.num_edges, self.num_teams, self.num_types, self.num_nodes, self.num_stats,
         self.num_deps, self.ticket, self.fine,  self.insprate, self.tmax, self.
        twait = [], [], [], [], [], [], [], [], [], [], []
class PassengerInfo(object):
  def __init__(self):
    self.Plists = []
    self.demand = []
```

NSBbasic.py

```
# Local
import classes
import mapconstruct

def get_map(finesize, repetitions, inspectors, nteams, printlvl = 0):

  S = range(6)
  busyness = [7, 10, 5, 3, 5, 6]
  l21 = classes.TrainLine([1,3,4,5], [0,116,157])
  dl21 = [12,11,26]
  l22 = classes.TrainLine([1,2,4], [162,131,175])
  dl22 = [10,11]
  lx = classes.TrainLine([1,0], [240,178,55])
  dlx = [10]
  L  = [l21,l22,lx]
  Ldist = [dl21,dl22,dlx]

  M  = classes.TrainMap(S,L)

  ticket = 1
  fine   = finesize
  tau = {}
  for il,l in enumerate(L):
    for i in xrange(len(l.l)-1):
      tau[(l.l[i], l.l[i+1], l)] = Ldist[il][i]
      tau[(l.l[i+1], l.l[i], l)] = Ldist[il][i]
  tw = 1

  D = []
  starthour = 8
  endhour   = starthour + repetitions
  for i in xrange(-2,3*(endhour-starthour)):
    D.append((lx, 14+i*20,  1)) #L13
    D.append((lx,  6+i*20, -1)) #L13
  for i in xrange(-2,1*(endhour-starthour)):
    D.append((l21, 18+i*60,  1)) #L21
    D.append((l21, 31+i*60, -1)) #L21
    D.append((l22, 31+i*60,  1)) #L22
    D.append((l22, 38+i*60, -1)) #L22

  insprate = 7*inspectors
  starttime = 0
  endtime   = 60 * repetitions
  (G,lookup,param,Dlists,Phi) = mapconstruct.build_G(S, D, tau, tw, starttime,
      endtime, printlvl = printlvl)
  (Phi,lookup) = mapconstruct.populate_G(G, lookup, param, D, Dlists, Phi, insprate
      ,  busyness, printlvl = printlvl)
```

```
  param.ticket = ticket
  param.fine   = fine

  # Make teams
  T = [ ((1, 0),(1, param.tmax)) for t in xrange(nteams) ]

  Tlists = []
  for t in xrange(len(T)):
    Tlists.append( (lookup.v2i[T[t][0]], lookup.v2i[T[t][1]]) )
  param.num_teams = len(Tlists)


  return (M,G,Dlists,Tlists,Phi,lookup,param)
```

NSBmedium.py

```
import numpy as np
# Local
import classes
import mapconstruct
def get_map(finesize, hours, inspectors, printlvl = 0):
  ticket = 1
  fine   = finesize

  S = range(21)
  sizes = {0:10, 1:2, 2:2, 3:1, 4:3, 5:1, 6:1, 7:1, 8:1, 9:1, 10:1, 11:1, 12:5,
      13:1, 14:1, 15:1, 16:3, 17:3, 18:1, 19:1, 20:3}
  L1  = classes.TrainLine([0,1,2,3,4,5,6,7,8,9,10,11,12], [222,81,146])
  dL1 = [4,3,3,2,2,2,2,2,2,2,2,2,3]
  L13 = classes.TrainLine([0,12,13,14,15,16,17,18,19,20], [231,129,51])
  dL13 = [10,7,3,3,4,9,2,3,7]
  L   = [L1,L13]
  Ldist = [dL1,dL13]

  M   = classes.TrainMap(S,L)

  tau = {}
  for il,l in enumerate(L):
    for i in xrange(len(l.l)-1):
      tau[(l.l[i], l.l[i+1], 1)] = Ldist[il][i]
      tau[(l.l[i+1], l.l[i], 1)] = Ldist[il][i]
  tw = 1

  D = []
  starthour = 8
  endhour    = starthour + hours
  for i in xrange(-2,2*(endhour-starthour)):
    D.append((L1, 11+i*30,  1))
    D.append((L1, 10+i*30, -1))
    D.append((L13, 14+i*30,  1))
    D.append((L13, 23+i*30, -1))

  insprate = 7 * inspectors
  starttime = 0
  endtime    = 60 * hours
  (G,lookup,param,Dlists,Phi) = mapconstruct.build_G(S, D, tau, tw, starttime,
      endtime)
  (Phi,lookup) = mapconstruct.populate_G(G, lookup, param, D, Dlists, Phi, insprate
      , sizes)
  param.ticket = ticket
  param.fine   = fine

  # Make teams
  T1 = ((0,0),(0,param.tmax))
  T2 = ((12,0),(12,param.tmax))
  T = [T1,T2]

  Tlists = []
  for t in xrange(len(T)):
    Tlists.append( (lookup.v2i[T[t][0]], lookup.v2i[T[t][1]]) )
```

```
  param.num_teams = len(Tlists)

  return (M,G,Dlists,Tlists,Phi,lookup,param)
```

NSBlarge.py

```
import numpy as np
# Local
import classes
import mapconstruct

def get_map(finesize, hours, inspectors, printlvl = 0):
  ticket = 1
  fine   = finesize

  S = range(41)
  sizes = {0:10, 1:1, 2:1, 3:1, 4:2, 5:1, 6:1, 7:2, 8:1, 9:1, 10:1, 11:1, 12:7,
      13:1, 14:1, 15:1, 16:3, 17:3, 18:1, 19:5, 20:1, 21:1, 22:1, 22:4, 23:1, 24:1,
       25:3, 26:3, 27:1, 28:1, 29:4, 30:1, 31:1, 32:1, 33:2, 34:1, 35:1, 36:1,
      37:1, 38:1, 39:1, 40:5}

  L1  = classes.TrainLine([0,1,2,3,4,5,6,7,8,9,10,11,12], [222,81,146])
  dL1 = [4,3,3,2,2,2,2,2,2,2,2,3]
  R10 = classes.TrainLine([0,12,17,19,20,21,22,23,24,25], [163,17,48])
  dR10 = [10,13,11,26,10,9,15,12,24]
  L12 = classes.TrainLine([0,12,17,18,19], [215,22,42])
  dL12 = [10,13,8,4]
  L13 = classes.TrainLine([0,12,13,14,15,16,26,27,28,29], [231,129,51])
  dL13 = [10,7,3,3,4,9,2,3,7]
  L14 = classes.TrainLine([0,12,30,31,32,33,34,35,36,37,38,39,40],[240,178,55])
  dL14 = [10,6,3,2,10,5,3,2,3,7,14,13]
  L  = [L1,R10,L12,L13,L14]
  Ldist = [dL1,dR10,dL12,dL13,dL14]

  M  = classes.TrainMap(S,L)

  tau = {}
  for il,l in enumerate(L):
    for i in xrange(len(l.l)-1):
      tau[(l.l[i], l.l[i+1], l)] = Ldist[il][i]
      tau[(l.l[i+1], l.l[i], l)] = Ldist[il][i]
  tw = 1

  D = []
  starthour = 8
  endhour   = starthour + hours
  for i in xrange(-3, 2*(endhour-starthour)):
    D.append((L1, 11+i*30,  1))
    D.append((L1, 10+i*30, -1))
    D.append((L12, 54+i*30,  1))
    D.append((L12, 31+i*30, -1))
    D.append((L13, 14+i*30,  1))
    D.append((L13, 23+i*30, -1))
  for i in xrange(-3, 1*(endhour-starthour)):
    D.append((R10, 34+i*60,  1))
    D.append((R10, 10+i*60, -1))
    D.append((L14,  4+i*60,  1))
    D.append((L14, 34+i*60, -1))

  insprate = 20
  starttime = 0
  endtime   = 60 * hours
  (G,lookup,param,Dlists,Phi) = mapconstruct.build_G(S, D, tau, tw, starttime,
      endtime)
  (Phi,lookup) = mapconstruct.populate_G(G,lookup,param,D,Dlists,Phi,insprate,sizes
      )
  param.ticket = ticket
  param.fine   = fine

  # Make teams
```

```
  T1 = ((0,0),(0,param.tmax))
  T2 = ((12,0),(12,param.tmax))
  T = [T1,T2]

  Tlists = []
  for t in xrange(len(T)):
    Tlists.append( (lookup.v2i[T[t][0]], lookup.v2i[T[t][1]]) )

  param.num_teams = len(Tlists)

  return (M,G,Dlists,Tlists,Phi,lookup,param)
```

mapconstruct.py

```
import os
import pickle
import copy
from operator import itemgetter
from sys import maxint
from heapq import heappush, heappop
import numpy as np
from scipy.optimize import minimize
import pymprog
# Local
import classes

def build_G(S, D, tau, tw, starttime, endtime, printlvl = 0):
  """Construct the transition graph"""

  lookup = classes.Lookup()
  param  = classes.Parameters()

  param.num_stats = len(S)
  param.num_deps  = len(D)

  Vset = set([])
  Eset = set([])

  Dedges = []  # Dlists, containing edges

  if printlvl >= 3: print "Starting part 1"

  tmax = 0

  # Algorithm 1
  for (line, t2, dway) in D:

    seq  = copy.deepcopy(line.l)
    dedge = [] # List of edges (v1,v2) in the departure

    if dway == -1:
      seq.reverse()

    for i in xrange(len(seq)-1):

      # Make vertices v1, v2 and edge e
      s1, s2 = seq[i], seq[i+1]
      t1, t2 = t2, t2 + tau[s1, s2, line]
      v1, v2 = (s1, t1), (s2, t2)
      e      = (v1, v2)

      if t1 >= starttime and t2 <= endtime:
        Vset.add(v1)
        Vset.add(v2)
        Eset.add(e)
        dedge.append(e)
        if t2 > tmax: tmax = t2  # Update tmax
      t2 = t2 + tw               # Add waiting time

    # Add the departure edge list
```

```
  Dedges.append(dedge)

param.tmax  = tmax
param.twait = tw

if printlvl >= 3: "Starting part 2"

# Adding start time and end time vertices
for i in xrange(param.num_stats):
  Vset.add((i,0))
  Vset.add((i,param.tmax))

# Adding waiting edges
for i in xrange(param.num_stats):
  times = sorted([y for (x,y) in Vset if x == i])
  for j in xrange(len(times)-1):
    v1 = (i,times[j])
    v2 = (i,times[j+1])
    e  = (v1,v2)
    Eset.add(e)

param.num_nodes = len(Vset)
param.num_edges = len(Eset)

if printlvl >= 3: "Starting part 3"

# Assign indices to vertices and edges, make dictionaries
v2i = {}
e2i = {}
Vlist = []
Elist = []
for i,v in enumerate(Vset):
  v2i[v] = i
  Vlist.append(v)

e_out = [set([]) for i in xrange(param.num_nodes)]
e_in  = [set([]) for i in xrange(param.num_nodes)]
for i,e in enumerate(Eset):
  # Edge
  e2i[e] = i
  Elist.append(e)
  # Vertex
  (u,v)  = e
  iu, iv = v2i[u], v2i[v]
  e_out[iu].add(i)
  e_in[iv].add(i)

G = classes.TransitionGraph(Vlist, Elist)
lookup.v2i = v2i
lookup.e2i = e2i
lookup.e_out = e_out
lookup.e_in  = e_in

if printlvl >= 3: "Starting part 4"

# Find the indices of the ordered edges for each departure
Dlists  = []
e2d     = [-1] * param.num_edges
for nd,dedge in enumerate(Dedges):
  (line, t2, dway) = D[nd]
  dedge_idx = []
  for i,e in enumerate(dedge):
    ie = e2i[e]
    dedge_idx.append(ie)
    e2d[ie]  = nd
  dlist = classes.TrainDeparture(dedge_idx, line.c)
  Dlists.append(dlist)

lookup.e2d = e2d
```

```
  if printlvl >= 3: "Starting part 5 (find paths)"

  # Make types with transfers
  sortVlist = sorted(G.Vlist,key=itemgetter(1))

  # Save the chronological order for later
  order = [ v2i[sortVlist[i]] for i in xrange(param.num_nodes) ]
  lookup.order = order


  sortVlist.reverse()
  journeymap = {}


  for io,v in enumerate(sortVlist):
    if printlvl >= 4: "Iteration ", io, "/", len(sortVlist)
    (s0, t0) = v
    iv = v2i[v]
    # Find potential paths
    path = shortest_train_rides(G, lookup, param, iv)
    # Filter paths
    for s1,p in enumerate(path):
      add = True
      if len(p) > 0: # There is a path
        # keys to previously existing paths between s0 and s1
        prekeys = [(a,b,c) for (a,b,c) in journeymap if a == s0 and b == s1]
        # r are those paths
        # if the last edge in new path p is equal to the last edge in an existing
        # path, do not add.
        if p[-1] in [r[-1] for r in [journeymap[k] for k in prekeys]]:
          add = False
      else: # Empty path, do not add
        add = False

      if add:
        journeymap[s0,s1,t0] = p

  if printlvl >= 3: "Starting part 5.5 (append paths)"

  Plists = []
  for key in journeymap:
    path = journeymap[key]
    trainpath = [ ie for ie in path if not e2d[ie] == -1 ]
    Plists.append(trainpath)

  param.num_types = len(Plists)

  e2t = [ [k for k in xrange(len(Plists)) if i in Plists[k]] for i in xrange(len(
      Elist)) ]

  lookup.e2t = e2t

  Phi = classes.PassengerInfo()
  Phi.Plists = Plists

  return (G,lookup,param,Dlists,Phi)

def populate_G(G,lookup,param,D,Dlists,Phi,insprate,busyness, printlvl = 0):

  if printlvl >= 3: "Starting part 6 (determine passenger type demand)"

  param.insprate = insprate

  demand = [ max(1, round(0.2 * ((busyness[G.Elist[Phi.Plists[k][0]][0][0]] *
      busyness[G.Elist[Phi.Plists[k][-1]][1][0]]) ** 1.5 ))) for k in xrange(param.
      num_types) ]

  evolume = [0] * param.num_edges
  effect   = [0] * param.num_edges
  for i in xrange(param.num_edges):
```

```
    etime = G.Elist[i][1][1] - G.Elist[i][0][1]
    if not lookup.e2d[i] == -1:
      evolume[i] = sum( demand[k] for k in lookup.e2t[i] )
      effect[i] = min(1, (insprate * etime)/float(evolume[i]) )

  # Assigning edge passenger volumes

  lookup.evolume = evolume
  lookup.effect = effect
  Phi.demand = demand

  return (Phi,lookup)

def shortest_train_rides(G, lookup, param, iv_start):
  """Find the shortest path from a vertex other stations"""

  (stat_start, time_start) = G.Vlist[iv_start]

  dist = [maxint] * param.num_nodes
  pred = [-1] * param.num_nodes
  statdist = [maxint] * param.num_stats
  statdist[stat_start] = 0
  endnode = [-1] * param.num_stats

  dist[iv_start] = 0
  statdist[stat_start] = 0

  Q = []
  heappush(Q,(0,iv_start))

  while Q:
    q = heappop(Q)
    iu = q[1]
    curdist = G.Vlist[iu][1] - time_start

    if max(statdist) <= curdist:
      # Found all other stations
      break

    for ie in lookup.e_out[iu]:
      iv = lookup.v2i[G.Elist[ie][1]]
      nustat = G.Vlist[iv][0]
      nutime = G.Vlist[iv][1]
      nudist = nutime - time_start

      if dist[iv] > nudist:
        dist[iv] = nudist
        pred[iv] = iu
        heappush(Q,(nudist,iv))
      if statdist[nustat] > nudist:
        statdist[nustat] = nudist
        endnode[nustat] = iv

  vpath = [[] for i in xrange(param.num_stats)]
  epath = [[] for i in xrange(param.num_stats)]
  for i in xrange(param.num_stats):
    iu = endnode[i]
    if iu != -1:
      vpath[i].append(iu)
      while iu != iv_start:
        iu = pred[iu]
        vpath[i].append(iu)
      vpath[i].reverse()

      for j in xrange(len(vpath[i])-1):
        epath[i].append(lookup.e2i[G.Vlist[vpath[i][j]],G.Vlist[vpath[i][j+1]]])

  return epath
```