The Asymmetric Traffic Assignment Problem on Large-Scale Networks

Author: Sivam Pasupathipillai
799352

Supervisor: Prof. Edoardo Amaldi

Advisor: Prof. Mauro Passacantando

Advisor: Prof. Esteve Codina Sancho

Academic year 2014-2015
To Elisa
Abstract

The traffic assignment problem is a network equilibrium problem with applications in various fields. Given a directed and weighted graph and a trip rate matrix, the problem consists of determining a flow assignment on the network that satisfies the demand from each origin to each destination under a properly defined equilibrium concept. The most widely used equilibrium concept is the user equilibrium, which corresponds to the situation in which users try to minimize their personal travel times. One of the most common applications of the problem is to transportation planning. In this context, link weights are defined as functions of link flows to model the congestion phenomenon of urban transportation networks. If link weights depend on the interactions between different link flows the problem is known as asymmetric traffic assignment. Traffic assignment models are used to predict the behavior of travelers in a transportation network.

The goal of this thesis is to implement and compare three different path-based algorithms for the asymmetric traffic assignment problem. Path-based algorithms consider path flows as variables. The methods we have implemented and investigated are: the gradient projection algorithm of Jayakrishnan, a heuristic method proposed by Sancho and the double projection algorithm of Panicucci, Pappalardo and Passacantando. The gradient projection algorithm was published by Jayakrishnan in 1994 for the diagonal traffic assignment problem and we extend it to the asymmetric version. The heuristic method was proposed by Sancho in 2014 and it includes a projection scheme based on the mean path cost difference between used paths. The double projection algorithm proposed by Panicucci, Pappalardo and Passacantando in 2007 is the only one among the three methods that is proven to converge on asymmetric problem instances. We compare the three algorithms on different medium to large-scale networks, representing actual transportation networks, to evaluate their performance and to verify their applicability for solving real-world problem instances. Our results show that path-based algorithms are viable for solving the asymmetric traffic assignment problem. In particular, the gradient projection algorithm and the double projection algorithm exhibit the best performance on the considered networks.
Sommario

Il problema dell’assegnamento del traffico è un problema di equilibrio su reti con applicazioni in diversi campi. Dato un grafo diretto e pesato e una matrice di frequenza di viaggi, il problema consiste nel determinare un assegnamento di flusso sulla rete che soddisfi la domanda di viaggi da ogni origine ad ogni destinazione e verifichi una condizione di equilibrio. Il concetto di equilibrio più usato è il cosiddetto user equilibrium, che corrisponde alla situazione in cui i viaggiatori cercano di minimizzare il proprio tempo di viaggio. Il problema si applica comunemente alla pianificazione del trasporto urbano. In questo contesto i pesi degli archi si definiscono come funzioni dei flussi sulla rete per modellare il fenomeno di congestione delle reti di trasporto urbano. Nel caso in cui i pesi dipendano dalle interazioni tra i flussi su archi diversi il problema è noto come assegnamento del traffico asimmetrico. I modelli di assegnamento del traffico vengono utilizzati per prevedere il comportamento dei viaggiatori in una rete di trasporto.

Ringraziamenti

Vorrei ringraziare innanzitutto il Professor Edoardo Amaldi per la sua cortesia e disponibilità nel farmi da relatore ed il Professor Mauro Passacantando per la sua disponibilità e per i numerosi consigli che mi hanno aiutato a migliorare e completare questa tesi. Inoltre vorrei ringraziare il Professor Esteve Codina Sancho per avermi assistito nella prima fase di questo progetto di tesi svoltasi presso l’Universitat Politècnica de Catalunya di Barcellona.

Un ringraziamento particolare va alla mia famiglia per tutto il supporto dimostratomi in questi anni, che mi ha permesso di inseguire i miei sogni e realizzarmi come persona.

Ovviamente un grazie speciale va ad Elisa, che mi ha supportato, e sopportato, in questi cinque anni e che mi ha aiutato a superare i momenti più duri.

Infine, vorrei ringraziare tutte le persone che ho incontrato in questo percorso e che, nel bene e nel male, mi hanno aiutato a crescere ed a diventare una persona migliore.
## Contents

### 1 Introduction
- 1.1 The traffic assignment problem
  - 1.1.1 Notation
  - 1.1.2 Equilibrium models
  - 1.1.3 Modeling congestion
  - 1.1.4 Problem statement
  - 1.1.5 Example
  - 1.1.6 Variational inequality formulation
  - 1.1.7 Optimization problem formulations
- 1.2 Document structure and contributions

### 2 State of the art
- 2.1 Equivalent formulations
  - 2.1.1 Fixed point problem
  - 2.1.2 Existence and uniqueness of equilibria
- 2.2 Algorithm concepts
  - 2.2.1 Partial linearization
  - 2.2.2 Decomposition
  - 2.2.3 Column generation and simplicial decomposition
  - 2.2.4 Projection algorithms
  - 2.2.5 Other considerations
- 2.3 Relevant algorithms
  - 2.3.1 Symmetric algorithms
  - 2.3.2 Asymmetric algorithms

### 3 Asymmetric traffic assignment algorithms
- 3.1 Common procedures
  - 3.1.1 Shortest path tree problem
  - 3.1.2 Convergence check
  - 3.1.3 Cost update
- 3.2 Jayakrishnan algorithm
# CONTENTS

A ANNEX: Source code

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 Solver.h</td>
<td>75</td>
</tr>
<tr>
<td>A.2 Solver.cpp</td>
<td>78</td>
</tr>
<tr>
<td>A.3 Network.h</td>
<td>93</td>
</tr>
<tr>
<td>A.4 Network.cpp</td>
<td>96</td>
</tr>
</tbody>
</table>

List of Figures 108

List of Tables 109

Bibliography 110
Chapter 1

Introduction

From the second half of the twentieth century an increasing interest has been addressed toward the study of transportation planning.

Transportation planning includes the evaluation, assessment, design and planning of transportation facilities, such as streets, highways, public transport lines, etc. Transportation planning relies on a collection of mathematical models which are used to simulate the behavior of real transportation systems.

These models can be categorized in trip generation and distribution models, modal split models and traffic assignment models. Each of these models can be described as an input-output process in which the output of the first model is fed as input to the second model and so on.

Trip generation and distribution models are used to determine the number of daily travelers from a given point to another in a transportation system. These two points are respectively known as the trip’s origin and destination. Due to the high level of complexity of the modeled phenomenon, considering each household as a trip generator would produce an intractable model. Therefore, contiguous areas of a transportation system are clustered into zones and each of the presented models works at a zone level granularity.

Trip generation models try to estimate the number of trips starting from and ending at a zone, for each zone of the system. These models take into account social, economic, land use and other relevant characteristics of each zone in the estimation.

The number of trips computed by trip generation models is used as input for trip distribution models. Trip distribution models try to assign to each trip its starting and terminating zone, thus obtaining the number of trips from each origin to each destination in the network. These results are usually expressed using the so called trip rate matrix $D$. Each element of the matrix $d_{pq}$ is the average number of trips, also known as demand, from origin $p$ to destination $q$, where the average is computed with respect to a defined time period known as the modeling horizon. Considering average times we are not limited to integer variables, i.e. number of trips, but we can use instead real-valued measures like vehicles per hour (vph).

Given the availability of different transport facilities in a transportation system, for example private cars and public transport, modal split models are used to estimate the distribution of trips on these transportation facilities; namely which transport will travelers choose in order to
go from their origin to their destination. Each facility is denoted as a mode. According to this analysis models are categorized as single-mode or multi-modal.

Finally, traffic assignment models are used to estimate how trips are allocated to paths in the transportation network, i.e. which street or public line starting from their origin will travelers choose in order to reach their destination. In these models a common assumption is that travelers try to minimize their personal travel time.

When we have a plausible traffic assignment over the network we can simulate the behavior of the network. Moreover, we can forecast how this behavior will react to changes in the model. This result can be very useful in order to evaluate modifications on the network topology, e.g. in forecasting of future projects, or to develop real-time traffic management systems.

In the following we will focus on traffic assignment models and in particular on the traffic assignment problem which is the main interest of this thesis.

1.1 The traffic assignment problem

Given a directed weighted graph, representing a transportation network, and a trip rate matrix, representing the number of travelers from a given origin to a given destination in the network, the traffic assignment problem (TAP) consists of determining a flow assignment on the links of the network which satisfies the demand for each origin-destination pair and minimizes each traveler’s travel time.

A link’s weight represents the cost of the link. The term cost is not to be intended literally as monetary cost, but instead as a general disutility function perceived by users that travel on the link. In our discussion the term cost will always be associated with the travel time on a link, i.e. the amount of time required for a vehicle to go through the link, thus the terms cost and travel time will be used equivalently.

We must point out that the problem just described is not formally well-defined. This is because we are modeling a dynamic system in which each user’s choice influences the other users’ choices. In this situation the system may never reach a stable state, in which case a solution to TAP would not be defined. To rigorously define the problem we must introduce the concept of equilibrium.
CHAPTER 1. INTRODUCTION

1.1.1 Notation

Network
- $G(N, A)$: Directed weighted graph representing the transportation network
- $N$: Nodes of the network, each node represents a crossroad
- $A$: Links of the network, each link represents a street
- $A_p$: Priority links of the network, $A_p \subset A$
- $A_n$: Non-priority links of the network, $A_n \subset A$
- $C$: Centroids of the network, each centroid represents a zone, $C \subset N$

Demand
- $W$: Set of origin-destination pairs $(p, q)$ with $p, q \in C$
- $D$: Trip rate matrix, square matrix $|W| \times |W|$ of demand, from $p$ to $q$ for $(p, q) \in W$

Link formulation
- $v_a$: Flow, i.e. number of travelers per hour, on link $a \in A$
- $t_a$: Cost, i.e. travel time per unit of flow, of link $a \in A$
- $\mathbf{v}$: Aggregated column vector of link flows, $|A| \times 1$
- $\mathbf{t}$: Aggregated column vector of link costs, $|A| \times 1$
- $V \equiv \mathbb{R}^{|A|}$: Space of link flow vectors

Path formulation
- $\Gamma_{pq}$: Set of discovered paths from $p$ to $q$ for $(p, q) \in W$
- $\Gamma$: Set of all discovered paths, $\otimes_{w \in W} \Gamma_w$
- $h_\gamma$: Flow on path $\gamma \in \Gamma$
- $c_\gamma$: Cost of path $\gamma \in \Gamma$
- $\pi_{pq}$: Minimum path cost from $p$ to $q$ for $(p, q) \in W$
- $\mathbf{h}$: Aggregated column vector of path flows, $|\Gamma| \times 1$
- $\mathbf{c}$: Aggregated column vector of path costs, $|\Gamma| \times 1$
- $H$: Space of path flow vectors

Volume-delay function
- $c_a$: Capacity of link $a \in A$
- $t_a^0$: Free-flow travel time on link $a \in A$
- $M$: Modeling horizon

1.1.2 Equilibrium models

As already mentioned, the goal of traffic assignment models is to provide a macroscopic description of a transportation system. It is important to point out that the behavior of this system depends on the decision-making process of each traveler in the network. It is obvious that in order to obtain a useful model, realistic assumptions must be made about this process. Different behavioral assumptions will define different solution concepts. A solution concept in traffic
assignment models is called equilibrium. Several equilibrium definitions have been defined over the years.

The most widely used equilibrium definitions come from the work of J.G. Wardrop. We have already mentioned that the most common assumption is that travelers minimize their personal travel time. Considering the behavior of individuals in a transportation system this seems a very sound assumption. The corresponding equilibrium is called user equilibrium and can be formalized by two principles known as Wardrop conditions.

To obtain some insight on the decision-making process behind travelers choices, consider a traveler which is traveling on path $A$ and has the alternative of traveling on a different path $B$. It is obvious that he or she will choose the path with the lowest travel time, lets assume it is $B$. Assuming that the travel time of a street depends on the amount of vehicles on that street, since there is one more traveler on path $B$, the travel time on $B$ will increase while the travel time on $A$ will decrease.

By looking at this simple example it is clear that travelers will continue to change their paths as long as there exists a path with lower travel time. Therefore the equilibrium can only be reached if the following statement is verified.

No traveler can reduce his journey time by unilaterally choosing another path

The Wardrop conditions can be stated as follows

**Principle 1.1** (Wardrop’s first principle). The journey times on all the paths actually used are equal, and less than those which would be experienced by a single vehicle on any unused path.

This is obvious from the simple example we have considered. As long as there exists a favorable path, travelers will move to that path and the system will not reach an equilibrium. Therefore all used paths must have the same travel time which must be strictly lower (better) than that on all other available paths, which of course remain unused.

**Principle 1.2** (Wardrop’s second principle). The average journey time is a minimum.

In other words, the average travel time on all used paths corresponds to the minimum travel time. This can be derived from the first principle. Since all trips are distributed on paths with the least travel time, at equilibrium the average travel time will correspond to the minimum travel time.

It must be noticed that both of these principles assume that each traveler has complete and accurate information about the available paths and their cost at any given time. This is a model approximation since it is unrealistic that travelers know exactly how long does it take to reach a certain destination at any given time.

Another concept of equilibrium proposed by Wardrop is the system equilibrium. This situation corresponds to the traffic assignment that minimizes the total travel time on the network without considering individual travel times, where the total travel time is defined as

$$\sum_{a \in A} t_a(v) \cdot v_a$$
CHAPTER 1. INTRODUCTION

This situation can be beneficial from a system administration point of view, but can be non-optimal from a single traveler’s perspective. Since in real life travelers are selfish and they are free to choose their paths, the system does not naturally converge to a system equilibrium. Nonetheless, this equilibrium has been studied since it can be enforced by the system administration; this can be done by introducing some form of road taxation or dynamic load balancing.

1.1.3 Modeling congestion

Another important assumption in traffic assignment models is that link travel times depend on link flows, i.e. the number of vehicles traveling on the network. This phenomenon is known in everyday life as traffic congestion. Traffic jams are a common example of this phenomenon.

The usual way to model congestion in traffic assignment models is to define link weights as functions of link flows, i.e. \( t_a = t_a(v) \) where \( v \) is the aggregated link flow vector. These flow-cost relations are usually called \textit{volume-delay functions}, \textit{link performance functions} or \textit{travel time functions}. For a detailed discussion on the definition and evaluation of volume-delay functions see Branston [10]. A well-defined volume-delay functions has the following properties:

- A volume-delay function is a function of the link flow vector, \( t_a(v) \)
- A volume-delay function is strictly positive on its domain, \( t_a(v) > 0 \)
- A volume-delay function is nonlinear
- A volume-delay function is monotonically increasing

An example of volume-delay function is plotted in figure [11]. The first two properties are needed in order to have a consistent model. The third and fourth properties are derived from empirical considerations and direct observation of traffic phenomena. Direct observation of traffic phenomena has also lead researchers to define volume-delay functions so that the link’s cost increases slowly at first and very rapidly after a certain flow amount is reached. This flow threshold is known as the link’s \textit{capacity}, denoted by \( c_a \). Intuitively, the link’s capacity is the maximum number of vehicles the street can contain. Capacity is only one of many parameters used to define volume-delay functions. Another important parameter is free-flow travel time, \( t^0_a \) for \( a \in A \), which represents the travel time on a link corresponding to zero flow, i.e. an empty street, \( t^0_a = t_a(0) > 0 \). The following are commonly used volume-delay formulas:

a) \( t_a(v) = t^0_a \cdot e^{\frac{\alpha_a}{v_a - t^0_a}} \)

b) \( t_a(v) = t^0_a \cdot 2^{\frac{\alpha_a}{v_a - t^0_a}} \)

c) \( t_a(v) = \left( 1 + \alpha_a \left( \frac{v_a}{c_a} \right)^\beta_a \right) \)

d) \( t_a(v) = t^0_a + \log(c_a) - \log(c_a - v_a) \)

e) \( t_a(v) = \beta_a - c_a \left( \frac{t^0_a - \beta_a}{v_a - c_a} \right) \)
where $\alpha$ and $\beta$ are real valued parameters. Even though all these formulas are defined as function of the link’s flow $v_a$ in our discussion we will consider volume-delay functions defined over the whole flow vector $v$. We will see that this choice is crucial and it leads to very different problem formulations.

It is important to notice that travel time is not the only factor that travelers consider when choosing their path. Distance, presence of tools and frequency of traffic signals are other factors that influence travelers’ choices. Moreover, travelers cannot correctly predict the travel time of each available path and therefore the perfect knowledge assumption does not apply in many scenarios. Thus, it is obvious that volume-delay functions can only give a rough estimate of the cost perceived by the user even in models where the complete information assumption is dropped.

![Figure 1.1: Typical volume-delay function](image)

### 1.1.4 Problem statement

We now formalize mathematically the user equilibrium principles and present the problem statement for TAP. The model we present here is taken from Beckmann[5]. In the following all defined vectors are assumed to be column vectors. Consider a transportation network described by a directed weighted graph $G(N, A)$ in which each node $n \in N$ represents a crossroad and each link $a \in A$ represents a street. A *centroid* node $c \in C \subset N$ is an artificial node representing a zone, i.e. an aggregated location in which a travelers flow is generated or terminates. For each pair of
centroids in $W = \{(p, q) : p, q \in C, p \neq q\}$ we have a nonnegative demand, or trip rate, $d_{pq}$ which represents the average number of travelers from $p$ to $q$ in the period of time $M$ considered. With this demand we can construct the trip rate matrix $D$, which is a square matrix of dimension $|W| \times |W|$. We denote the set of simple, directed paths from origin $p$ to destination $q$ as $\Gamma_{pq}$, or $\Gamma_w$. Each path $\gamma \in \Gamma_w$ can carry a nonnegative flow $h_\gamma$ and has a nonnegative cost $c_\gamma(h)$. We denote with $h$ the aggregated path flow vector and with $c(h)$ the aggregated path cost vector.

We assume that the cost on a path depends in general on the whole flow vector $h$. We denote $\pi_w$ the cost of the shortest path between $p$ and $q$, computed with respect to the cost vector $c(h)$. The aggregated minimum path cost vector is denoted with $\Pi$.

To create a model as general as possible, we assume that the travel demand between origin and destination depends on the minimum path cost vector, i.e $d_w = d_w(\Pi)$, this assumption is justified in certain applications where travelers can choose different, equivalent service providers in the network. Thus they can choose one destination instead of another depending on the shortest path cost associated with the journey for reaching that destination.

With these definitions the Wardrop conditions can be formalized using the following model

**Definition 1.1. Network equilibrium model (NEM)**

\[
\begin{align*}
h_\gamma(c_\gamma(h) - \pi_w) & = 0 & \forall \gamma \in \Gamma_w, \forall w \in W \\
c_\gamma(h) - \pi_w & \geq 0 & \forall \gamma \in \Gamma_w, \forall w \in W \\
\sum_{\gamma \in \Gamma_w} h_\gamma & = d_w(\Pi) & \forall w \in W \\
h_\gamma & \geq 0 & \forall \gamma \in \Gamma_w, \forall w \in W \\
\pi_w & \geq 0 & \forall w \in W
\end{align*}
\]

A user equilibrium flow vector $(h^*, \pi^*)$ is a vector in $\mathbb{R}^{|\Gamma| \times |W|}$ that satisfies NEM. A vector that satisfies NEM is a solution to TAP. Since variables correspond to path flows this formulation is known as path-based TAP.

Under the assumption that path costs are additive, i.e. the cost of a path is the sum of the costs on the links composing the path, we can restate NEM in term of link flow variables. Denoting with $v_a$ and $t_a(v)$ respectively the flow and cost of link $a$, for each $a \in A$, we can represent a link flow assignment in term of path flows as

\[
v_a = \sum_{w \in W} \sum_{\gamma \in \Gamma_w} \delta_{a,\gamma} h_\gamma
\]

or in compact form

\[
v = \Delta h
\]

with $\Delta$ being the link-path incidence matrix with elements defined as

\[
\delta_{a,\gamma} = \begin{cases} 
1, & \text{if link } a \text{ belongs to path } \gamma \\
0, & \text{otherwise}
\end{cases}
\]
With this notation we can express the cost additivity constraints as
\[ c_{\gamma}(h) = \sum_{a \in A} \delta_{\alpha\gamma} t_a(v) \]
or in compact form
\[ c_{\gamma}(h) = \Delta^T t(\Delta v) \]
Replacing these equations in the general NEM model leads to the so-called, link-based formulation of the problem. Both path-based and link-based formulations can be useful in different contexts such as determining existence and uniqueness condition for TAP or designing algorithms for its resolution.

One drawback of path-based formulation is that the number of possible paths is exponential in the size of the input and in general cannot be enumerated a priori. This fact implies that both sets \( \Gamma_w \) and \( \Gamma \) are just theoretical artifacts and cannot be directly used in the design of solutions algorithms.

On the other hand, in link-based formulation the solution of the model does not directly provide path flow information since a link flow assignment can correspond to multiple path flow assignments. Moreover, link-based algorithms usually provide less accurate solutions with respect to path-based algorithms.

Although many more extensions and variants of the model proposed in this section exist, we will only deal with either the symmetric or asymmetric, fixed demand traffic assignment problem. For a complete discussion on the traffic assignment problem and its variants see Sheffi\[29\].

1.1.5 Example
In this section we present one iteration of simple algorithm on a small problem instance. This is intended to give an example of the steps that a resolution algorithm performs in each iteration to solve the traffic assignment problem. The network corresponding to the problem instance is shown in figure 1.2. The network is composed of 9 nodes and 12 links. Nodes colored in grey are centroids. Each node that is not a centroid represents a road intersection. Each link represents a street. Links colored in green are priority-links, i.e. streets that have precedence at intersections. There is a bidirectional street between nodes 5 and 8 and nodes 7 and 9. We assume that volume-delay functions for priority links are defined as
\[ t_a(v_a) = g(v_a) \]
where
\[ g(x) = 1 + x + x^2 \]
For non-priority links we have
\[ t_a(v) = g(v_a) + g(\bar{v}) \]
where $\bar{v}$ is defined as

$$\bar{v} = \sum_{a \in X_a} v_a$$

where $X_a$ is the set of priority links that share the ending node with $a$.

There is a positive demand of 20 vph from 1 to 2 and another 20 vph from 3 to 4. The corresponding trip rate matrix $D$ is

$$D = \begin{bmatrix}
-20 & 0 & 0 \\
-0 & 0 & 0 \\
-20 & -20 & -
\end{bmatrix}$$

A common initialization of TAP algorithms corresponds to assigning all the demand on the shortest path between each origin-destination pair. Each shortest path is computed considering an empty network. In our example the link costs corresponding to an empty network are

<table>
<thead>
<tr>
<th>Link</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 5)</td>
<td>2</td>
</tr>
<tr>
<td>(3, 8)</td>
<td>1</td>
</tr>
<tr>
<td>(5, 6)</td>
<td>1</td>
</tr>
<tr>
<td>(5, 8)</td>
<td>1</td>
</tr>
<tr>
<td>(5, 9)</td>
<td>1</td>
</tr>
<tr>
<td>(6, 7)</td>
<td>2</td>
</tr>
<tr>
<td>(7, 2)</td>
<td>1</td>
</tr>
<tr>
<td>(7, 9)</td>
<td>1</td>
</tr>
<tr>
<td>(8, 5)</td>
<td>2</td>
</tr>
<tr>
<td>(8, 6)</td>
<td>1</td>
</tr>
<tr>
<td>(9, 4)</td>
<td>1</td>
</tr>
<tr>
<td>(9, 7)</td>
<td>1</td>
</tr>
</tbody>
</table>

The shortest paths between the origin-destination pairs corresponding to this cost vector are

$$spt(1 \to 2) = [1, 5, 9, 7, 2] = \gamma_1 \quad cost = 5$$
$$spt(3 \to 4) = [3, 8, 5, 9, 4] = \gamma_2 \quad cost = 4$$

where each path is denoted with the ordered list of nodes it traverses. After the assignment of the whole demand on these paths, the link flows are

<table>
<thead>
<tr>
<th>Link</th>
<th>Flow</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 5)</td>
<td>20</td>
</tr>
<tr>
<td>(3, 8)</td>
<td>20</td>
</tr>
<tr>
<td>(5, 6)</td>
<td>0</td>
</tr>
<tr>
<td>(5, 8)</td>
<td>0</td>
</tr>
<tr>
<td>(5, 9)</td>
<td>40</td>
</tr>
<tr>
<td>(6, 7)</td>
<td>0</td>
</tr>
<tr>
<td>(7, 2)</td>
<td>20</td>
</tr>
<tr>
<td>(7, 9)</td>
<td>20</td>
</tr>
<tr>
<td>(8, 5)</td>
<td>0</td>
</tr>
<tr>
<td>(8, 6)</td>
<td>20</td>
</tr>
<tr>
<td>(9, 4)</td>
<td>20</td>
</tr>
<tr>
<td>(9, 7)</td>
<td>20</td>
</tr>
</tbody>
</table>

and the corresponding link costs are

<table>
<thead>
<tr>
<th>Link</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1, 5)</td>
<td>842</td>
</tr>
<tr>
<td>(3, 8)</td>
<td>421</td>
</tr>
<tr>
<td>(5, 6)</td>
<td>1</td>
</tr>
<tr>
<td>(5, 8)</td>
<td>1</td>
</tr>
<tr>
<td>(5, 9)</td>
<td>1641</td>
</tr>
<tr>
<td>(6, 7)</td>
<td>422</td>
</tr>
<tr>
<td>(7, 2)</td>
<td>421</td>
</tr>
<tr>
<td>(7, 9)</td>
<td>1</td>
</tr>
<tr>
<td>(8, 5)</td>
<td>421</td>
</tr>
<tr>
<td>(8, 6)</td>
<td>2</td>
</tr>
<tr>
<td>(9, 4)</td>
<td>421</td>
</tr>
<tr>
<td>(9, 7)</td>
<td>421</td>
</tr>
</tbody>
</table>

The new shortest paths are

$$spt(1 \to 2) = [1, 5, 6, 7, 2] = \gamma_3 \quad cost = 1686$$
$$spt(3 \to 4) = [3, 8, 6, 7, 9, 4] = \gamma_4 \quad cost = 1267$$

while the cost of the paths discovered in the previous step is updated according to

$$\gamma_1 = [1, 5, 9, 7, 2] \quad cost = 3325$$
$$\gamma_2 = [3, 8, 5, 9, 4] \quad cost = 2904$$
In the following chapters we will discuss different flow update strategies. For this simplified example assume that the flow is updated according to the following equation

\[ h_{k+1}^\gamma = \max\left\{ 0, h_k^\gamma - \alpha(c_\gamma - \pi_{pq}) \right\} \quad \forall (p, q) \in W \]

for non-shortest paths, with \( \alpha \in (0, 1) \), and all the remaining demand is assigned to the shortest path. Taking \( \alpha = 0.01 \), the path flows after the update are

<table>
<thead>
<tr>
<th>Path</th>
<th>( \gamma_1 )</th>
<th>( \gamma_3 )</th>
<th>( \gamma_2 )</th>
<th>( \gamma_4 )</th>
</tr>
</thead>
</table>

We can notice that the flow on the shortest path is increased, while the flow on non-shortest path is decreased. This flow update implies new link cost and flow vectors. Ideally, the path costs after the update are identical, however this is not guaranteed by our simple flow update strategy.

The flow update completes one iteration of the algorithm. Then, a new shortest path for each origin-destination pair is computed and the execution goes on. The algorithm continues until a flow assignment is found that is close enough to satisfying the user equilibrium conditions presented in definition 1.1.

![Example network](image)

**Figure 1.2: Example network**

### 1.1.6 Variational inequality formulation

The NEM model provides a characterization of flow vectors satisfying the Wardrop user equilibrium conditions. Even though this characterization is an important theoretical result it is not straightforward to develop resolution algorithms based on it. Therefore, different formulations
have been used over the years which were found more useful in order to provide uniqueness and existence conditions of solutions to TAP and to develop algorithms for its resolution.

One of the most important formulations for TAP is through a variational inequality problem. A variational inequality problem (VIP) is a mathematical problem that have been extensively used in order to study equilibrium problems in many different fields. For an overview of the most relevant results regarding VIP see Harker et al.\[17\]. The equivalency between TAP and a VIP was originally pointed out by Dafermos\[12\]. A variational inequality problem can be defined by the following

**Definition 1.2 (Variational Inequality Problem).** Let $X \subseteq \mathbb{R}^n$ be a nonempty, closed and convex set, and $F : X \mapsto \mathbb{R}^n$ a mapping on $X$. The variational inequality problem is to find an $x^* \in X$ such that

$$F(x^*)^T(x - x^*) \geq 0, \quad \forall x \in X$$ (1.6)

Geometrically, VIP is the problem of finding a vector $x^*$ such that the vector $F(x^*)$ is at an acute angle with all the vectors in $X$ stemming from $x^*$.

If we define $F(x) = F((h, \Pi))$ as

$$F(x) = \left\{ c_\gamma(h) - \pi_w \mid \sum_{\gamma \in \Gamma_w} h_\gamma - d_w(\Pi) \right\}^T$$

we have the following

**Proposition 1.1 (Equivalency between NEM and VIP).** The NEM and VIP formulations are equivalent, provided that $c_\gamma(h) > 0$ and $d_w(\Pi) \geq 0$, for all $\gamma \in \Gamma_w, w \in W, h \in \mathbb{R}^{|\Gamma|}, \Pi \in \mathbb{R}^{|W|}$.

For the details about the proof see Section 2.1.

**Link flow formulation with fixed demands**

In order to derive existence conditions for TAP we now restate the problem considering a link-based VIP formulation with fixed demands, i.e. $d(\Pi) = \bar{d}$. The following result was originally presented by Smith\[30\].

**Proposition 1.2.** A link flow vector $v^*$ solves NEM if and only if $v^*$ solves

$$\sum_{a \in A} t_a(v^*)(v_a - v_a^*) \geq 0$$

for all $v_a$ such that

$$\sum_{\gamma \in \Gamma_w} h_\gamma = d_w, \quad \forall w \in W$$ (1.7)

$$h_\gamma \geq 0, \quad \forall \gamma \in \Gamma$$ (1.8)

$$v_a = \sum_{\gamma \in \Gamma} \delta_{a\gamma} h_\gamma, \quad \forall a \in A$$ (1.9)

This formulation can be extended to elastic demand models, however in the rest of this thesis we will focus on the fixed demand version of the problem.
CHAPTER 1. INTRODUCTION

1.1.7 Optimization problem formulations

With the results discussed so far it is possible to derive equivalent optimization problems for NEM, and consequently for TAP. However, before continuing with our discussion it is important to consider separately two versions of the problem for which different results have been derived, namely the symmetric and asymmetric TAP.

This categorization takes into account the structure of volume-delay functions considering link flows $t(v)$, or equivalently path flows $c(h)$. In the following we denote with $\nabla t(v)$ the Jacobian matrix of volume-delay functions.

**Symmetric traffic assignment**

If $\nabla t(v)$ is symmetric, that is

$$\frac{\delta t_a(v)}{\delta v_b} = \frac{\delta t_b(v)}{\delta v_a}, \quad \forall a, b \in A$$

we refer to the problem as symmetric TAP. Moreover, if volume-delay functions are separable, i.e. $t_a(v_a), \forall a \in A$, the problem is known as diagonal TAP. Consider the following optimization theory result

**Proposition 1.3.** If $f : \mathbb{R}^n \to \mathbb{R}$ is a convex, differentiable function, and $X$ is a nonempty, convex set in $\mathbb{R}^n$, then $x^* \in X$ is an optimal solution to the problem of minimizing $f(x)$ subject to $x \in X$ if and only if

$$\nabla f(x^*)^T(x - x^*) \geq 0, \quad \forall x \in X$$

Intuitively, if $F(x)$ is the gradient of a convex, differentiable function $f$, that is

$$F(x) = \nabla f(x)$$

a VIP can be stated as an equivalent convex optimization problem.

With this result and considering a diagonal TAP it is possible to prove the equivalency between finding an equilibrium flow vector for NEM and solving the following convex optimization problem

$$\min_{v \in V} S(v) = \sum_{a \in A} \int_0^{v_a} t_a(x) dx$$

subject to (1.7), (1.8) and (1.9). This formulations was originally proposed by Beckmann et al.\[5\] and is known as Beckmann’s transformation.

Due to its equivalency to an optimization problem, symmetric traffic assignment has been extensively studied and many algorithms have been proposed for its resolution. In fact this equivalency provides TAP with the entire framework of resolution methods developed in the convex optimization field. However, the applicability of symmetric methods to real problem instances is limited due to the fact the symmetric and diagonal assumption are seldom satisfied in practice.
Asymmetric traffic assignment

When $F(x)$ does not correspond to a gradient mapping, the line integral in (1.10) is path-dependent and the optimization problem equivalent to TAP is not well-defined. If $F$ is differentiable this corresponds to saying that Jacobian matrix of volume-delay function is not symmetric.

In this case the problem is known as asymmetric TAP (ATAP). This version of the problem is more challenging than the symmetric version and it also has a broader range of applications. The methods for ATAP mostly rely on results in the field of variational inequalities. In order to solve VIP it is common to define a gap function and minimize it over the solutions space. Consider the following

**Definition 1.3 (Gap function).** Let $\Omega$ be the set of solutions to VIP. A function $\psi : X \mapsto \mathbb{R} \cup \{-\infty, +\infty\}$ is a gap function for VIP if

1. $\psi \geq 0$, $\forall x \in X$ and
2. $\psi(x) = 0 \iff x \in \Omega$

A gap function is a measure of the distance between the current point and the points that solves VIP. This means that if we minimize a gap function over $X$ we obtain a solution for VIP. Thus, many algorithms for VIP try to minimize a properly defined gap function on the space of feasible link or path flows. For an overview on the most relevant concepts regarding gap functions see Larsson et al. [20].

One of the first gap functions in the literature, the *primal gap function* [3], can be defined considering VIP in the space of link flows

$$t(v^*)^T(v - v^*) \geq 0, \quad \forall v \in \Theta$$  \hspace{1cm} (1.11)

where $\Theta$ is the set of feasible link flows, i.e. link flows that verify conditions (1.7), (1.8) and (1.9). Define $G(\tilde{v})$ for all $\tilde{v} \in \Theta$ as

$$G(\tilde{v}) = \max_{v \in \Theta} t(\tilde{v})^T(\tilde{v} - v)$$

It is easy to see that for an optimal assignment $v^*$ we have $t(v^*)^T(v^* - v) \leq 0, \forall v \in \Theta$. Then, solving (1.11) is equivalent to finding $v^*$ that solves

$$v^* = \arg\min_{v \in \Theta} G(\tilde{v})$$

which can be rewritten as

$$v^* = \min_{\tilde{v} \in \Theta} \{\max_{v \in \Theta} t(\tilde{v})^T(\tilde{v} - v)\}$$

where, for each $\tilde{v}$ the inner maximization problem is a linear program. Function $G(\tilde{v})$ is the primal gap function.

Over the years researchers have defined many different classes of gap functions and their development is still an active argument of research in the field of variational inequalities. With
the introduction of the concept of gap function for the asymmetric TAP it is possible to reduce the
problem to an optimization problem, however other resolution strategies exist.

The most well-known algorithms proposed in literature are iterative algorithms that perform
a search in the feasible solutions space while trying to improve a properly defined merit function
representing the quality of the current solution with respect to the optimum defined by the
Wardrop conditions. The step forward in each iteration consist of shifting flow between the
possible paths according to a search direction and proportionally to a step-size. The details
regarding the flow balancing procedures, the search direction and the step-sizes differentiate one
algorithm from another.

In our project we have focused our attention of projection algorithms. These algorithms
include a projection scheme in the flow balancing procedure so that the feasibility of the solution
is maintained between successive iterations.

1.2 Document structure and contributions

During the development of this thesis we have implemented three different path-based algo-
rithms for the asymmetric traffic assignment problem. The first algorithm is an adaptation of
the Jayakrishnan’s gradient projection algorithm for TAP \cite{15}. The algorithm was originally
proposed for diagonal TAP and it was adapted to the asymmetric version of the problem using a
diagonalization scheme. The second algorithm is an heuristic procedure proposed by Sancho\cite{27}
and includes an original projection scheme based on the mean path cost difference between
used paths. The third algorithm implemented is the double projection algorithm of Panicucci,
Pappalardo and Passacantando\cite{24} which features a double projection scheme with an adaptive
step-size.

In this chapter we introduced some relevant theory about about the traffic assignment prob-
lem. In chapter 2 we review the state of the art of TAP. We focus on providing existence and
uniqueness conditions for solutions to the problem and we describe the main algorithms that
have been proposed for its resolution. In chapter 3 we describe in details the three algorithms
implemented during the project and in chapter 4 we detail the main development phases of
the project and in particular the implementation choices and the software tools we used. In
chapter 5 we report the experimental results we have obtained and in chapter 6 we present the
conclusions of the thesis and we propose future work related to the project.

The main purpose of this thesis is to study the behavior of different resolution methods for
the asymmetric TAP on large-scale networks and establish whether it is the case that one of the
algorithms outperforms the others. Even though TAP is a very well-studied problem and many
algorithms for its resolution exist in the literature, the comparative analysis of such methods
has not yet been the focus of researchers in this field. We object that a serious effort in the
evaluation and comparison of the available methods could be beneficial to the field and drive
future research.
Chapter 2

State of the art

In this chapter we will discuss the current state of the art regarding the traffic assignment problem and its resolution methods. We begin the chapter by presenting some relevant theory results that are needed to derive uniqueness and existence conditions for the solutions to TAP. Then, we present the main algorithmic concepts that have been used in the design of algorithms for the problem and finally we present some of the most well-known algorithms proposed in the literature. The following sections will complete the discussion presented in the first chapter. In particular we use the same notation introduced in Section 1.1.1.

2.1 Equivalent formulations

We begin the chapter with the demonstration of the equivalency between NEM and VIP presented in Section 1.1.6. The corresponding proof was originally presented in Dafermos[12]. For clarity we repeat

**Proposition 2.1** (Equivalency between NEM and VIP). The NEM and VIP formulations are equivalent, provided that $c_\gamma(h) > 0$ and $d_w(\Pi) \geq 0$, for all $\gamma \in \Gamma_w, w \in W, h \in \mathbb{R}^{\Gamma}_+, \Pi \in \mathbb{R}^{|W|}_+$.

**Proof.** Suppose that $x^* = (h^*, \Pi^*)$ solves NEM. Then, from the definition of $F(x)$ and equations (1.1) and (1.3) we have

$$F(x^*)^T x^* = (c_\gamma(h^*) - \pi^*_w, \sum_{\gamma \in \Gamma_w} h^*_\gamma - d_w(\Pi^*))^T (h^*, \Pi^*)$$

$$= \sum_{w \in W} \sum_{\gamma \in \Gamma_w} (c_\gamma(h^*) - \pi^*_w) h^*_\gamma + \sum_{w \in W} \left( \sum_{\gamma \in \Gamma_w} h^*_\gamma - d_w(\Pi^*) \right) \pi^*_w$$

$$= 0$$

Similarly, considering a generic vector $x = (h, \Pi) \in X$ we can write

$$F(x)^T x = \sum_{w \in W} \sum_{\gamma \in \Gamma_w} (c_\gamma(h) - \pi^*_w) h^*_\gamma \geq 0$$
since \( h_\gamma \geq 0 \) and \((c_\gamma (h^*) - \pi^*_{w}) \geq 0, \forall \gamma \in \Gamma_w, w \in W \).

Thus, the vector \( x^* = (h^*, \Pi^*) \)

\[
F(x^*)^T(x - x^*) = F(x^*)^T x - F(x^*)^T x^* \geq 0, \quad \forall x \in X
\]
solves VIP.

Now, suppose \( x^* \) solves VIP, let \( \hat{x} = x^* + e_i \) where \( e_i \) is the \( i \)-th unit vector of \( \mathbb{R}^n \). Then, \( \hat{x} \geq 0 \) and

\[
0 \leq F(x^*)^T(\hat{x} - x^*) = F(x^*)^T e_i = \text{the } i \text{-th component of } F(x^*)
\]

Since \( i \) can be chosen arbitrarily, every component of \( F(x^*) \) is nonnegative, i.e. \( \forall \gamma \in \Gamma_w, \forall w \in W : c_\gamma (h^*) - \pi^*_w \geq 0 \) and \( \sum_{\gamma \in \Gamma_w} h_\gamma - d_w(\Pi^*) \geq 0 \).

Now assume that every component of \( x^* \) is positive, i.e. \( x^*_i > 0 \), let \( \hat{x} = x^* - \alpha e_i \) with \( x^*_i - \alpha > 0 \), and \( \alpha > 0 \). Then

\[
0 \leq F(x^*)^T(\hat{x} - x^*) = (-\alpha)F(x^*)^T e_i
\]

which implies that \( F(x^*)^T e_i \leq 0 \). Thus, from the previous step it must be \( F(x^*)e_i = 0 \), i.e. every component of \( F(x^*) \) is equal to zero provided that \( x^* \) solves VIP and for each \( i, x^*_i > 0 \).

Therefore, assuming that \( x^* = (h^*, \Pi^*) \) solves VIP we have derived the following conditions

\[
\begin{align*}
\alpha \sum_{\gamma \in \Gamma_w} h_\gamma^* &= 0, \quad \forall \gamma \in \Gamma_w, \forall w \in W \quad \text{(2.1)} \\
\alpha \sum_{\gamma \in \Gamma_w} h_\gamma^* - d_w(\Pi^*) &= 0, \quad \forall w \in W \quad \text{(2.2)} \\
\sum_{\gamma \in \Gamma_w} h_\gamma^* &- d_w(\Pi^*) \geq 0, \quad \forall w \in W \quad \text{(2.3)} \\
\Pi^* \geq 0 \quad \text{(2.4)} \\
h^* \geq 0 \quad \text{(2.5)}
\end{align*}
\]

Comparing this system with the NEM definition, the only equation that could be violated is (1.3). Then, assuming that \( x^* \) solves [VIP] but does not solve NEM, we have that for some \( \tilde{w} \in W, \sum_{\gamma \in \Gamma_{\tilde{w}}} h_\gamma^* > d_{\tilde{w}}(\Pi^*) \) which from equation (2.3) implies \( \pi^*_{\tilde{w}} = 0 \). Since demands are nonnegative by assumption, we have that there exist one path \( \tilde{\gamma} \in \Gamma_{\tilde{w}} \) with positive flow. By (2.1) this implies \( c_{\tilde{\gamma}}(h^*) = \pi^*_{\tilde{w}} \). Therefore, we have \( c_{\tilde{\gamma}}(h^*) = 0 \), which contradicts the hypothesis. This concludes the proof.

The following section introduces another equivalency relation, this time between VIP and a fixed point problem. The following results were originally published by Aashtiani[1]. This equivalency is important in determining existence and uniqueness conditions for TAP.
2.1.1 Fixed point problem

In general, a fixed point problem (FPP) is the mathematical problem of finding a solution to the equation \( x = H(x) \) for some function \( H \). The intrinsic difficulty of TAP is due to the fact that demand functions are dependent on minimum path costs, which in turn are dependent on the demand vector. Thus, if we consider a fixed demand vector \( \bar{d} \), it may be the case that the demand function \( d(\cdot) \) evaluated at the minimum path cost vector \( \Pi \) corresponding to the fixed demand, may be different from \( \bar{d} \). In fact an equilibrium occurs when

\[
\bar{d}(\Pi) = \bar{d}
\]

If we define the function \( H : \mathbb{R}_+^n \to \mathbb{R}_+^n \) with \( n = |\Gamma| + |W| \) as

\[
H(h, \Pi) = \text{Proj}_{\mathbb{R}_+^n}[(h, \Pi) - F(h, \Pi)]
\]

where \( h \in \mathbb{R}_+^{|\Gamma|}, \Pi \in \mathbb{R}_+^{|W|} \) and \( \text{Proj}_X[y] = z \) means that \( z \) is the closest point to \( y \) in set \( X \). If we consider the FPP of finding a vector \( x^* \) such that

\[
x^* = H(x^*) = \text{Proj}_{\mathbb{R}_+^n}[x^* - F(x^*)]
\]

we have the following proposition

**Proposition 2.2** (Equivalency between FPP and VIP). \( x^* \) solves FPP if and only if \( x^* \) solves VIP.

**Proof.**

\[
x^* = H(x^*) = \text{Proj}_{\mathbb{R}_+^n}[x^* - F(x^*)] \iff (x^* - (x^* - F(x^*)))^T(x - x^*) \geq 0, \quad \forall x \geq 0
\]

\[
\iff F(x^*)^T(x - x^*) \geq 0, \quad \forall x \geq 0
\]

2.1.2 Existence and uniqueness of equilibria

The Brouwer fixed-point theorem states that every continuous function from a convex compact subset \( K \subset \mathbb{R}^n \) to \( K \) itself has a fixed point. Then, to ensure existence of a solution for NEM we have

**Proposition 2.3** (Existence of solution to NEM). If \( c_\gamma(h) \) is positive and continuous and \( d_w(\Pi) \) is nonnegative, continuous and bounded, for each \( \gamma \in \Gamma \) and \( w \in W \), the NEM has a solution.

Consider the following

**Definition 2.1** (Monotone function). A vector function \( f : X \subseteq \mathbb{R}^n \to \mathbb{R}^n \) is monotone on \( X \) if

\[
(f(x_1) - f(x_2))^T(x_1 - x_2) \geq 0, \quad \forall x_1, x_2 \in X
\]

It is strictly monotone if for any distinct \( x_1, x_2 \in X \) we have

\[
(f(x_1) - f(x_2))^T(x_1 - x_2) > 0
\]
Uniqueness of the solution of NEM can be stated considering link flow variables as

**Proposition 2.4.** If \( t(v) \) and \(-d(\Pi)\) are strictly monotone, then \([NEM]\) has a unique solution in the arc flows \( v^* \) and in the minimum path costs \( \Pi^* \).

In general a link flow assignment does not correspond to a unique path flow assignment, thus in path-based TAP the uniqueness of the optimal solution is not usually guaranteed.

For completeness we also report the definitions of a Lipschitz continuous function and a pseudomonotone function, which will be used in the following chapters.

**Definition 2.2 (Lipschitz continuous function).** A vector function \( f : X \subseteq \mathbb{R}^n \rightarrow \mathbb{R}^n \) is Lipschitz continuous with constant \( L \) on \( X \), if there exists a positive constant \( L \) such that

\[
||f(x_1) - f(x_2)|| \leq L||x_1 - x_2||, \quad \forall x_1, x_2 \in X
\]

**Definition 2.3 (Pseudomonotone function).** A vector function \( f : X \subseteq \mathbb{R}^n \rightarrow \mathbb{R}^n \) is pseudomonotone on \( X \), if

\[
f(x_2)^T(x_1 - x_2) \geq 0 \Rightarrow f(x_1)^T(x_1 - x_2) \geq 0, \quad \forall x_1, x_2 \in X
\]
2.2 Algorithm concepts

In this section we report the most important results achieved in the algorithmic resolution of TAP. The literature on this topic is extremely wide, however it is possible to categorize algorithms using only a few main concepts which characterize them.

These key concepts are partial linearization, decomposition and column generation. The vast majority of algorithms proposed for TAP make use of one or more of these concepts. Thus, even though describing every proposed algorithm is outside the scope of this thesis, by considering these few concepts we can have an idea of the evolution of TAP algorithms in time. For a more detailed review of algorithms for TAP see Florian et al.\cite{Florian} and Patriksson\cite{Patriksson}.

For each concept we will consider its application to both symmetric and asymmetric TAP. The main difference is that in the symmetric version we are considering the problem stated as a mathematical program (1.10), while in the asymmetric version we are considering a VIP based formulation.

2.2.1 Partial linearization

Partial linearization is a resolution method which can be applied to nonlinear, convex optimization problems. This technique consists of an iterative descent method, where the search direction is generated by solving an auxiliary linear program obtained by linearizing the objective function of the original problem.

When applied to TAP, the solution of the auxiliary linear program corresponds to the flow vector obtained by assigning all demands on the shortest path for each origin-destination pair. The procedure of finding the shortest path and assigning all the corresponding demand on it is known as \textit{all-or-nothing} assignment. The search direction is the difference between the current flow vector and the all-or-nothing flow vector. Then, a step-size is computed in order to select a new feasible point in the search direction. This computation can be performed optimally by solving a linear program, or approximately, by using a heuristic such as the Armijo-Goldstein rule. A new solution is obtained by moving in the search direction using the selected step-size.

As already mentioned, asymmetric TAP does not feature an explicit objective function and therefore the linear approximation concept must be refined in order to be applied to this variant of the problem. In this context the concept is more generally known as cost approximation. The main idea behind cost approximation is to generate iteratively a descent direction with respect to a gap function for VIP. Depending on the characteristics of the mapping $F$ of VIP different resolution strategies exist. Using some approximation schemes it is possible to solve a convex quadratic program equivalent to VIP. In other cases projection methods can be used. Convergence of cost approximation algorithms for VIP usually depends on the monotonicity and Lipschitz continuity of $F$.

2.2.2 Decomposition

From the particular structure of the constraint set it is possible to decompose the feasible region into an independent set of simplices, one for each origin-destination pair. With this method we are able to decompose the original problem into a set of independent smaller-sized problems. In
order to perform this decomposition it is however necessary to modify the objective function, which in general is not separable as the feasible region.

Decomposition algorithms are composed of two main steps: linearize the objective function so that it becomes separable and solve the separated subproblems in an efficient way. The main techniques used to solve the subproblems are the Gauss-Seidel method and the Jacobi method. Both of these methods were developed to solve efficiently systems of linear equations in an iterative manner. The Gauss-Seidel method works sequentially on the system while the Jacobi method can be implemented to work in parallel. Both approaches can be valid depending on the context. Sometimes the speedup obtained through parallelization is nullified by the overhead needed to synchronize different parallel computations.

Every subproblem fixes the flows corresponding to the other origin-destination pairs and optimizes the flow for the pair corresponding to the subproblem. Since decomposition algorithms remove asymmetries between link costs, they are also referred to as diagonalization or relaxation algorithms.

Convergence and performance characteristics of these methods depend intuitively on the strength of the dependencies between the volume-delay functions. Decomposition algorithms can be directly applied to ATAP, without the need of additional considerations.

2.2.3 Column generation and simplicial decomposition

It is well known that the feasible region of a convex program can be represented as the convex hull of its extreme points. The main idea behind column generation algorithms is to iteratively generate solutions and then using the convex combinations of these solutions to find a point which minimizes the objective.

When the procedure of generating new solutions resolves to computing the shortest path between each origin-destination pair the method is referred to as simplicial decomposition. This term was originally coined by Von Hohenbalken[31]. The name derives from the simplex algorithm, and the simplicial decomposition method can be seen as a generalization of such an algorithm. In this version non-basic variables can have non-zero values and a change of basis correspond to moving a non-basic variable to a constraint bound. In this context it is possible to show that a basis corresponds to a shortest path and a feasible descent direction is defined by the difference between path cost and shortest path cost for each path. For a concrete application to traffic assignment see Nguyen[23].

In order to find the solution as a convex combination of previous solutions the vector of coefficients must be computed; this can be done using heuristics or approximation techniques. Most of the algorithms consider only a fixed number of previously generated points in order to find new solutions. A method that considers only a finite number of points in the convex combination is known as restricted simplicial decomposition. This number is an important parameter for this class of algorithms and can determine convergence of the method. Sometimes it is also important to define solution dropping rules, so that points that are not used in the combination are removed from the working set. This consideration is also important since in many cases the number of generated points is related to the number of possible paths between an origin and a destination and is therefore exponential in the size of the input.
When developing simplicial decomposition algorithms it is possible to distinguish between at least two levels of aggregation. The first level is known as aggregated simplicial decomposition. It consist of generating intermediate solutions corresponding to complete flow vectors, that is without considering an origin-destination pair decomposition. In this version the new iterate is obtained as the convex combination of path flow vectors of dimension $|\Gamma| \times 1$, i.e. considering all discovered paths. Another approach consists of considering one origin-destination pair at a time and combining only the path flow vectors corresponding to that pair, i.e. combining vectors with dimensions $|\Gamma_w| \times 1$. These two approaches are respectively referred to as aggregated simplicial decomposition (ASD) and disaggregated simplicial decomposition (DSD).

As with decomposition techniques, also column generation algorithms can be applied to ATAP. However, in this case it is not possible to limit a priori the number of points used in the convex combination in order not to lose convergence.

### 2.2.4 Projection algorithms

Projection algorithms are the first resolution method proposed for the variational inequality formulation of TAP [12]. These algorithms are an example of cost approximation. The main idea is to replace the original cost function with a linear one defined using an appropriate symmetric positive definite matrix $Q$. The corresponding variational inequality can be solved by iteratively finding the projection of the point

$$v - \rho Q^{-1} t(v)$$

onto the feasible set $\Omega$, where $\rho$ is a positive constant. This approach can also be extended to the path-based VIP formulation. In fact, in this case the projection can be computed more efficiently, as was shown by Bertsekas and Gafni[9]. The main concerns in order to design a projection algorithm are the definition of the projection matrix to be used and the choice of the step-size, i.e. the length of the step to be taken in the descent direction. The step-size is especially relevant in order to prove the convergence of a projection algorithm. The gradient projection method is a particular instance of a projection algorithm. The method in its simplest version consists of moving from an initial feasible solution in the direction of steepest descent of the objective function, which is represented by the negative gradient. In many cases this movement leads to a point outside of the feasible region. If this happens a projection is performed so that the new feasible point is the projection of the new point onto the feasible set, i.e. the feasible point closest to the new point. With some modification this method can be extended to nonlinear problems, both constrained and unconstrained. For more details and extensions see Luenberger[22].

### 2.2.5 Other considerations

With a combination of the algorithmic concepts presented it is possible to characterize the majority of the algorithms for TAP proposed in the literature. The intrinsic complexity of the problem has lead to the development of iterative algorithms that terminate when the current solutions is close enough to an optimum, where the distance is defined using a merit function of some sort. In order to apply these algorithms to large-scale instances of the problem it
is important to take into consideration the structure of the problem. By structure we mean
for example the separable nature of the feasible region into origin-destination pairs, the linear
constraints and the characteristics of the objective function. An algorithm not explicitly designed
to consider these aspects will never be good enough to outperform algorithms developed to
exploit the problem structure.

At the core of each algorithm defined for TAP there is some procedure to transfer flow from
one path to another. These flow equilibration techniques vary from algorithm to algorithm. One
simple method is to consider only the longest and shortest path and move their flows from one to
the other until their cost is the same. Other techniques consider the simultaneous equilibration
of all the active paths; these methods use a path equilibration step in which the amount of flow
moved is proportional to the difference in path costs and to path flows. Other algorithms, such
as decomposition algorithms, cycle through the origin-destination pairs and optimize one pair
at a time, considering as fixed the traffic generated by the other pairs. Different algorithmic
concepts generate different path equilibration techniques, and thus algorithms with different
performance and characteristics.

2.3 Relevant algorithms

2.3.1 Symmetric algorithms

The literature regarding resolution methods for symmetric TAP is extremely vast. The historical
reference algorithm in this field is the Frank-Wolfe method. This algorithm it is still nowadays
the state of the practice for traffic assignment in several commercial software packages. The main
formulation behind symmetric algorithm is the Beckmann’s transformation. Thus symmetric
algorithms can only be applied to instances where volume-delay functions are separable or
symmetric. Historically, link-based algorithms were proposed first since path-based methods
were considered to have too excessive space requirements. With the developments in computer
technology at the end of the century, path-based algorithms became a viable alternative. Even
more recently, origin-based algorithms have been proposed. These algorithms consider a multi-
commodity flow assignment, where each commodity corresponds to an origin. We present the
Frank-Wolfe method and the OBA algorithm by Bar-Gera, which was one of the first origin-
based algorithm proposed in literature.

Frank-Wolfe algorithm

The Frank-Wolfe algorithm is still nowadays the state of the practice algorithm for traffic equi-
librium problems. It was proposed in 1956 as a general resolution algorithm for quadratic
programming problems \([15]\). The algorithm works in the space of link flows \(V\) and has the
convenient advantage of a very limited storage need. The algorithm steps are the following

1. (Initialization) Select a feasible solution \(v^0 \in V\), \(LBD = 0\), \(\epsilon > 0\) and \(k = 0\).

2. (Search direction generation) Define \(\tilde{S}\) as

\[
\tilde{S}(v) = S(v^k) + \nabla S(v^k)^T (v - v^k)
\]
where $S(v)$ is defined in [1,10], and solve the linear program

$$P : \min \tilde{S}(v)$$

subject to

$$\sum_{\gamma \in \Gamma_{pq}} h_{\gamma} = d_{pq} \quad \forall (p, q) \in W \quad (2.8)$$

$$h_{\gamma} \geq 0 \quad \forall \gamma \in \Gamma_{pq}, \forall (p, q) \in W \quad (2.9)$$

$$\sum_{(p, q) \in W} \sum_{\gamma \in \Gamma_{pq}} \delta_{\gamma a} h_{\gamma} = v_{a} \quad \forall a \in A \quad (2.10)$$

Let $y^k$ be the solution to $P$, define the search direction as $p^k = y^k - v^k$

3. (Convergence test) Let $LBD = \max\{LBD, \tilde{S}(y^k)\}$, if

$$\frac{S(v^k) - LBD}{LBD} < \epsilon$$

then terminate using $v^k$ as solution. Otherwise, continue

4. (Line search) Find a step-size, $\alpha^k$, that solves the line search problem

$$\min\{S(v^k + \alpha p^k) \mid 0 \leq \alpha \leq 1\}$$

5. (Flow update) Let

$$v^{k+1} = v^k + \alpha^k p^k$$

6. (Convergence test) If

$$\frac{S(v^{k+1}) - LBD}{LBD} < \epsilon$$

then terminate using $v^{k+1}$ as solution. Otherwise, let $k = k + 1$ and go to 2.

The Frank-Wolfe algorithm solves a linearized version of the program to find a feasible descent direction (2), then it finds the optimal step-size in that direction (4) and finds a new link flow assignment by moving in the descent direction using the step-size (5). The convergence test stops the algorithm when the desired precision $\epsilon$ is obtained. An initial feasible solution can be generated considering an all-or-nothing assignment on an empty network.

Apart from the already mentioned drawbacks of link-based algorithms, the Frank-Wolfe method also shows a tailing, or jamming, behavior when it gets close to the optimal solution. This means that after the initial iterations the algorithm starts to generate solutions which are only slightly better than the previous ones. Therefore, even if the convergence of the algorithm to the optimal solution set is guaranteed, its convergence rate has been showed to be sublinear. This problem is due to the fact that the local descent direction obtained in the subproblem resolution is used as a global descent direction in the line search. This algorithm has been studied extensively and over the years a great number of extensions and improvements have been proposed. Due to the absence of a robust benchmark for traffic equilibrium problems the Frank-Wolfe algorithm is often used to evaluate newly proposed algorithms.
OBA algorithm

Origin-based algorithms consider a multi-commodity flow assignment. In these algorithms the link flow is separated according to its origin, i.e. the origin from which the flow was generated. This concept is known as aggregated link flow. With this abstraction we have an aggregated link flow vector for each origin and we can develop origin-based algorithms focused on balancing this aggregated flow. Here we report the OBA algorithm proposed by Hillel Bar-Gera[4]. A restricting subnetwork is a spanning tree from an origin containing all links that carry positive aggregated flow for the origin. During execution a restricting subnetwork is kept for each origin. The algorithm correctness relies on the fact that restricting subnetworks are kept acyclic during the execution of the algorithm. This algorithm is one of the first algorithms of this class and it can be described as follows

1. (Initialization) Compute the minimum spanning tree for each origin and perform all-or-nothing assignments for each destination.
2. Update restricting subnetworks.
3. Find a feasible descent direction $g^k$ and an optimal step-size in the descent direction $\lambda^k$.
4. Update link flows according to
   \[
   v^{k+1} = v^k + \lambda^k \cdot g^k
   \]
5. If restricted convergence is met continue, otherwise go to step 3.
7. If global convergence is met stop, otherwise go to step 2

The restricting subnetwork update procedure consists of removing unused links, compute the maximum distance from the origin to each node and update the data structures accordingly. The maximum distance is kept as an upper bound to the size of paths so that it is possible to detect links that could be part of a cycle.

The flow update procedure consists of shifting flow from one link in the restricting subnetwork to another, while maintaining feasibility of the solution. The update considers the approaching links of each node. Flow is shifted from one link to another considering the approach proportion of each link, defined as the proportion of total flow that comes to a node from the considered adjacent link.

The stopping criterion for this algorithm is based on the concept of social pressure. This is defined whenever we have a flow shift from $a_1$ to $a_2$, for $a_1, a_2 \in A$, as the cost difference of the links

\[
sp = t_1(v_1) - t_2(v_2)
\]

Therefore, the algorithm continues as long as the social pressure of flow updates is positive, i.e. the objective function is improving.
2.3.2 Asymmetric algorithms

After the publication of the equivalency result between VIP and TAP, algorithms for the asymmetric version of the problem have followed the developments in this field. We present two different approaches, one featuring a simplicial decomposition scheme and another based on the minimization of a properly defined gap function.

Lawphongpanich-Hearn algorithm

In this section we present the SDVI algorithm proposed by Lawphongpanich and Hearn\cite{21}. The algorithm is an example of simplicial decomposition for the ATAP formulated as a variational inequality problem.

The algorithm generates iteratively aggregated extreme points of the feasible set. As already mentioned, each extreme point corresponds to an all-or-nothing assignment vector on shortest paths for each origin-destination pair.

Once the shortest paths have been computed a solution to VIP can be represented in term of a vector of coefficients $\lambda^* \in \Lambda$ such that

$$A^T t(A\lambda^*)(\lambda - \lambda^*) \geq 0, \quad \forall \lambda \in \Lambda$$

where $A$ is the matrix having all-or-nothing assignment flow vectors as its columns and

$$\Lambda = \{\lambda : \sum_{i=1}^{m} \lambda_i = 1, \lambda_i \geq 0\}$$

The number $m$ is the number of extreme points of the feasible region. Since the extreme points are impractical to enumerate the algorithm generates them as needed. The convex combination of previously generated extreme points using $\lambda^*$ forms the new solution.

Since we do not have a properly defined objective function to monitor the algorithm convergence, the primal gap function is used instead. After each iteration the function is evaluated and according to its value different actions are taken

- If the function is equal to zero the solution is optimal and the algorithm terminates.
- If the function was increased with respect to the previous iteration, we include the current solution in the extreme points set.
- If the function was decreased, we drop from the extreme points set the flow vectors with a coefficient equal to zero.

The SDVI algorithm is proved to convergence for the asymmetric TAP under mild assumptions on volume-delay functions.

Fukushima algorithm

In this section we present the algorithm of Fukushima\cite{16} for asymmetric variational inequalities. The paper introduces an original gap function which has been known later as the Fukushima’s
gap function. With this function it is possible to reformulate VIP as an equivalent optimization problem even without the assumption that $F$ is a gradient mapping. Additionally the Fukushima’s gap function is differentiable so that the optimization problem can be solved using one of the many descent methods available for minimizing an objective function.

The Fukushima’s gap function is defined as

$$f(x) = -F(x)^T (H(x) - x) - \frac{1}{2}(H(x) - x)^T (G(H(x) - x))$$

where $G$ is a positive definite matrix and $H(x)$ is the mapping that maps $x$ onto its projection of the feasible set with respect to $G$, that is

$$H(x) = \text{Proj}_{S,G}(x - G^{-1}F(x))$$

with $S$ being the feasible set of VIP. In other words $H$ defines the FPP equivalent to VIP.

The optimization problem equivalent to VIP is

$$x^* = \arg\min_{x \in S} f(x) \quad (2.11)$$

The Fukushima’s gap function $f$ has the convenient property that it is continuous if $F$ is continuous. Moreover, if $F \in C^1$, i.e. if $F$ is continuously differentiable, and $\nabla F(x)$ is positive definite $\forall x \in S$, if $x$ is a stationary point of problem (2.11), $x$ is also a global optimal solution of the problem. Thus, descent methods can be applied since they will not get stuck on local optima.

Apart from descent methods, which involve the evaluation of the gradient of $F$, Fukushima proposes to use the distance between $x$ and its projection as a descent direction, that is

$$d = H(x) - x$$

The corresponding algorithm is proved to converge under continuity assumptions on the operator $F$ even if the step length used in the descent is computed heuristically.

The algorithm is an interesting example of a resolution method for asymmetric TAP from the variational inequality field. For a discussion on direct application of the Fukushima algorithm and other similar projection methods based on different gap functions to asymmetric TAP refer to Sancho et al.\cite{28}.
Chapter 3

Asymmetric traffic assignment algorithms

In this chapter we describe in details the three algorithms implemented during the development of the project. Each algorithm can be divided into the following main procedures

- shortest path tree computation
- cost update
- flow update
- convergence check

In the following section we will point out the main differences among the three algorithms. Apart from the flow update procedure, these procedures are very similar in the three implementations and we describe them in the beginning of the chapter.

3.1 Common procedures

3.1.1 Shortest path tree problem

Given a directed and strongly connected graph $G(N, A)$ and an origin $o \in N$, the shortest path tree problem (SPT) consists of finding a spanning tree $T$ of $G(N, A)$ such that the path from $o$ to $n$ on $T$ corresponds to the shortest path from $o$ to $n$ in $G(N, A)$, $\forall n \in N$. As already mentioned, in many algorithm for TAP a feasible solution is found by computing the shortest paths between each origin-destination pair. Moreover, the computation of shortest path trees account for a large portion of the computational time required to solve the problem. Thus, it is very important to solve the shortest path tree problem efficiently in order to find a solution for the asymmetric traffic assignment problem using the algorithms we will describe in the following sections. For a detailed review of network problems and algorithms please refer to Ahuja et al.\cite{2} and Bertsekas\cite{8}.
CHAPTER 3. ASYMMETRIC TRAFFIC ASSIGNMENT ALGORITHMS

Many algorithms exist for SPT. All of these algorithms have in common some form of labeling scheme for the nodes of the graph. A node’s label represents the lower bound on the cost of the path between the origin and the node. According to the way labels are managed, SPT algorithms can be distinguished in label setting and label correcting algorithms. In the following we denote with \( l_n \) the label value of node \( n \in N \), and \( c_{ij} \) the cost of link \( (i, j) \in A \).

**Label setting algorithms**

Label setting algorithms take their name from the fact that once a node label is set, it is never modified again. In this algorithms each node is considered exactly once and at each iteration one node is permanently labeled.

The most famous label setting algorithm is Dijkstra’s algorithm \([13]\). It utilizes a data structure to maintain a list of the nodes to be labeled; we will refer to this structure as the candidate list. Different implementations of the candidate list result in different worst case complexities for the algorithm.

The algorithm initializes all the labels to \( \infty \), except for the label of the origin node, which is set to zero. The origin is then inserted in the candidate list. At each iteration, it selects the node in the candidate list which has the minimum label, it sets it as current node and removes it from the list. For each neighbor \( n \) of the current node \( c \), if its label is greater than the sum of the current node label and the cost of the link between the current node and the neighbor, that is

\[
l_n > l_c + c_{cn}
\]

the algorithm updates the neighbor’s label and inserts it into the candidate list, if it is not already present. After all neighbors have been visited, and possibly added to the candidate list, a new iteration begins. The algorithm continues until the candidate list is empty.

Using a simple list as container and linearly searching the minimum label value in the list in every iteration results in a complexity of \( O(|N|^2) \). This limit can be improved to \( O = (|A| + |N|\log|N|) \) by using a priority queue. The main drawback of Dijkstra’s algorithm is that in every iteration it must find the label with the minimum value, thus adding considerable overhead to the computation.

**Label correcting algorithms**

In label correcting algorithm the optimal search of the minimum label at each iteration is replaced by a greedy approach. The greedy approach consists of selecting at each iteration the top of the candidate list as the current node. With this method a node that was already visited can reenter the candidate list, thus increasing the total number of iterations. In this case the data structure used to store the candidate list is usually a queue, i.e. a LIFO structure.

The main algorithm of this family is the Bellman-Ford’s algorithm \([6]\). This algorithm is similar to Dijkstra’s algorithm, however at each iteration the node removed from the candidate list is simply the top of the list and when a node is inserted it is placed at the bottom. The label update procedure is identical to that of Dijkstra. The algorithm keeps iterating until the candidate list is empty. As already mentioned the main difference between the two algorithms
is that nodes already removed from the candidate list can be reinserted if a better path to reach
them is found. The worse case complexity of the Bellman-Ford’s algorithm is \(O(|A||N|)\).

Many extensions of this simple algorithm exist. Most of them differs from the original version
by the insertion policy of the nodes in the candidate list. The D’Esopo-Pape algorithm \[25\] is
based on the intuition that whenever a node’s label is updated, there is an high chance that its
neighbors’ labels will be modified as well. Therefore, the algorithm should perform better if the
node is inserted before its neighbors in the candidate list. This consideration is translated in
the following insertion policy

\[
\text{If a node has never been in the candidate list before add it to the bottom, otherwise}
\text{add it to the top}
\]

The worst-case complexity of this algorithm is worse than that of the Bellman-Ford algorithm
but in practice it seems to perform better, especially on sparse networks.

Another heuristic approach that can be considered is that nodes with low labels should be
inserted near the top of the candidates list. This is because low labeled nodes will probably
affect higher labeled nodes and should therefore be considered first. This lead to the variant
of the Bellman-Ford algorithm known as SLF, or Small Label First. In this variant whenever
a node is about to be inserted in the candidates set, if the node’s label is less than or equal to
the top element’s label, then the new node is inserted at the top, otherwise it is inserted at the
bottom.

The algorithms described differ from the insertion policy of a node in the candidate list, the
LLL, or Large Label Last, heuristic operates when extracting a node from the candidate list.
Instead of always extracting the node on top of the queue, the LLL strategy compares the top
node’s label with the mean value of the labels in the queue. If the top node’s label is higher
than the mean value, the top element is removed from the top and inserted at the bottom. This
procedure is repeated until a node with a label smaller or equal to the mean value is found; this
is the node extracted from the list. Both the SLF and the LLL strategy have worse worst case
complexities than the original formulation of the algorithm, but again they seem to work well
in practice.

In our project we have implemented Dijkstra’s algorithm, Bellman-Ford’s algorithm as well as
the D’Esopo-Pape method and a combination of the SLF and LLL policies. All TAP algorithms
implemented make use of the same shortest path tree solver.

\subsection{3.1.2 Convergence check}

Each iterative resolution method needs to have a stopping criterion. A stopping criterion is a
logical condition that, when verified, forces the algorithm to terminate. In the case of TAP a
stopping criterion usually checks that the current solution satisfies some property, for example
if it is optimal or “close enough” with respect to the optimum. For ATAP many convergence
criteria have been defined. The majority of these verify \(\epsilon\)-optimality, i.e. if the current solution is
within an \(\epsilon\) to the current best optimum estimate, with \(\epsilon\) being a small nonnegative tolerance, e.g.
\(\epsilon = 10^{-6}\). Other widely used criteria rely on the difference between an error measure between two
successive solutions generated by the iterative method. Using this type of convergence criteria
is however dangerous since in many resolution methods a small difference between successive feasible points does not mean we are close to an optimal solution.

The definition of a stopping criterion usually requires the definition of an error function over the feasible solutions which represents the distance of the current solution from the optimum, this concept is related to the concept of gap function defined in Section 1.1.7. In fact, many algorithms based on VIP use a gap function both to provide a search direction in the solutions space and to verify the quality of the current solution.

Since the purpose of this project was to compare the performance of the algorithms, we implemented three different convergence criteria, namely the relative gap, the variance criterion and the relative flow error. The first criterion is widely used in literature and is based on the primal gap function, the second was proposed by Sancho[27] and the third was taken from Panicucci et al.[24].

We implemented multiple criteria in order to verify that our comparative analysis was consistent independently on the error measure utilized. It is easy to verify that all of the following error measures are equal to zero when evaluated at an optimal solution of the problem, i.e. a solution that satisfies the Wardrop conditions.

Relative gap

The relative gap criterion is based on the primal gap function. Intuitively, it compares the total travel time evaluated at the current solution with the travel time corresponding to an all-or-nothing assignment on the shortest paths. The expression for the criterion is

$$\Phi(h) = \frac{|c(h)^T(\bar{h} - h)|}{|c(h)^T(\bar{h})|}$$

where $\bar{h}$ is the flow vector obtained by assigning the whole demand on the current shortest paths, i.e. evaluated at the current cost. The formula is normalized with respect to the shortest travel time. The corresponding convergence criterion is

$$\Phi(h) \leq \epsilon$$

Variance criterion

The variance criterion uses the information about the mean path costs and cost standard deviations for each origin-destination pair to derive an error measure. From Wardrop first principle (1.1) we know that at equilibrium all used path costs are equal and correspond to the minimum path cost. Therefore, we can define an error measure as

$$\Psi(h) = \max_{w \in W} \frac{\sigma_w}{\bar{c}_w}$$

where $\sigma_w$ is the standard deviation of the path costs and $\bar{c}_w$ is the mean cost for each $w \in W$. Since all paths share the same cost, the standard deviation of used paths costs must tend to zero when we approach a user equilibrium. In order to evaluate this error measure we have to compute the mean cost and standard deviation for each origin-destination pair since this
information is not available directly in most resolution methods. This criterion was proposed for the Sancho’s algorithm, since the mean path cost information plays a central role in this algorithm.

**Maximum flow error**

The maximum flow error considers the maximum amount of flow that is not traveling on shortest paths among all origin-destination pairs. It is defined as

$$
\varphi(h) = \max_{w \in W} \left( \frac{1}{d_w} \sum_{\gamma \in \Gamma_w} h_\gamma \right)
$$

where

$$
\Gamma_w = \left\{ \gamma \in \Gamma_w : \frac{c_\gamma(h) - \pi_w(h)}{\pi_w(h)} > \epsilon \right\}
$$

This error measure does not require additional information with respect to the one already available during the problem resolution. Thus, it can be used without significant overhead in all three algorithms.

### 3.1.3 Cost update

The cost update procedure is used to maintain consistency on the network during the resolution of the problem. As already mentioned, we assume that link costs, and consequently path costs, depend on the link flow vector. Since the flow vector changes as the algorithm progresses, link costs must be updated in each iteration to reflect the state of the network. It turns out that this procedure is expensive in terms of computational cost since each path flow updated implies a cost update for each link composing the path. Moreover, several paths may share the same links thus making this procedure $O(|\Gamma||A|)$ in the worst case scenario.

With respect to volume-delay functions we have two types of links, priority and non-priority links. Priority links represent streets with priority at the road intersection in which they enter. Non-priority links represent streets without priority. Priority and non-priority links form a partition on the links set $A$, that is

$$
A_p \cup A_n \equiv A \quad \text{and} \quad A_p \cap A_n \equiv \emptyset
$$

The cost update procedure is performed after each flow update. The procedure considers each path sequentially and recomputes the path cost as the sum of the cost of links composing the path. Thus we assume that path costs are additive.

**Priority links**

Assuming that vehicles on non-priority links must give way to vehicles on priority links at intersections, we can assume that the travel time on a priority link only depend on the vehicles
on the link, i.e. on the link’s flow. Therefore, for priority links we have the following formula

\[ t_a(v) = t^0_a \cdot \left[ 1 + \alpha \left( \frac{v_a}{M c_a} \right)^\beta \right], \quad \forall a \in A_p \]

where \( M \) is the modeling horizon and \( c_a \) is the link capacity, \( \alpha \) and \( \beta \) are nonnegative parameters. As already mentioned in Section 1.1.3 a volume-delay function is a nonlinear, positive function of flow. The cost corresponding to zero flow is the free-flow travel time \( t^0_a = t_a(0) \).

### Non-priority links

Travel time on non-priority links depends on the number of vehicles traveling on the corresponding priority links. To reflect this fact, the volume-delay function defined for these links depends on both the non-priority link flow and the corresponding priority links flows. The volume-delay formula for this type of links is

\[ t_a(v) = t^0_a + \frac{1}{\theta} \ln \left( 1 + e^{\theta b(x_a(v)-1)} \right), \quad \forall a \in A_n \]

where \( \theta \) and \( b \) are nonnegative parameters and \( x_a(v) \) is the priority factor defined as

\[ x_a(v) = \frac{v_a + \sum_{\bar{a} \in X(a)} k_{\bar{a}} v_{\bar{a}}}{M c_a} \]

where \( X(a) \) is the set of priority links sharing the ending node with \( a \). The relative capacity \( k_{\bar{a}} \) is defined as

\[ k_{\bar{a}} = \frac{c_a}{c_{\bar{a}}} \]

The asymmetric version of TAP is due to the fact that the dependency between priority and non-priority link costs is not equivalent. In other words, the cost of a priority link will never be affected by the flow on a non-priority link, while the inverse is not true. Therefore the Jacobian matrix of the volume-delay functions we have defined is not symmetric.

### Volume-delay function parameters

Both of the defined volume-delay functions depend on different parameters. The parameter values we used in this project are showed in table 3.1 for each of the networks we considered. The capacity and free-flow travel time parameters are defined for each link in the input files. The free-flow travel time in particular is computed as a function of the maximum legal speed and of the link length and it is expressed in minutes.
### 3.2 Jayakrishnan algorithm

The first algorithm we implemented in our project was the Jayakrishnan gradient projection (GP) algorithm. The algorithm was originally proposed by Jayakrishnan et al. [18] for the diagonal version of the problem. This paper was one of the first to reconsider path-based formulations of the problem, since the initial motivations that lead to their disuse, for example their excessive space requirements, were to be reexamined in light of the huge improvements in computational power. This consideration was supported by the experimental results of Chen [11].

The algorithm contains a gradient projection scheme derived from the one proposed by Bertsekas [7] for optimal control problems. In the algorithm, the shortest path flow variables are rewritten with respect to non-shortest paths variables. The projection can then be performed only with respect to these variables. Then, the remaining flow is assigned to the shortest path to maintain the feasibility of the solution.

Algorithm 1 contains the pseudocode for the algorithm we have implemented. One difference with respect to the original algorithm is that the convergence criterion is evaluated before updating the flow vector and not as the final step of the algorithm. Since the original algorithm does not define explicitly a convergence criterion we cannot tell which of the two approaches is better. However, we object that the evaluation of a convergence criterion after a flow update without the corresponding cost update might prevent the discovery of new shortest paths that could improve the solution.

The procedure that characterize the GP algorithm, and many other algorithms, is the flow update procedure, or *path equilibration step*. The projection in the GP algorithm is performed considering the demand satisfaction constraints

\[
\sum_{\gamma \in \Gamma_w} h_{\gamma} = d_w, \quad \forall w \in W
\]

It is important to notice that these constraints define a simplex for each origin-destination pair, thus the projection can be performed independently for each pair and this improves the efficiency of the algorithm. At each iteration paths are updated in the descent direction corresponding to the negative gradient, which in this case is equivalent to the vector having the differences between path costs and shortest path cost as elements.

Considering one origin-destination pair \(w\) at a time, the path flows update equation for non-shortest paths is

\[
h_{\gamma}^{k+1} = \max \left\{ 0, h_{\gamma}^k - \frac{\alpha^k}{s^k_{\gamma}} (c_{\gamma}(h^k) - \pi_w) \right\}, \quad \forall \gamma \in \Gamma_w
\]

(3.1)
The step-size in the descent direction is the ratio
\[
\frac{\alpha^k}{s^k}\gamma
\]
where \(\alpha\) is a nonnegative parameter and \(s^k\gamma\) depends on the Jacobian of volume-delay functions.

After all the non-shortest paths have been updated, the difference between the total demand \(d_w\) and the demand assigned to non-shortest paths is assigned to the shortest path, in order not to violate the demand constraints. From (3.1) we can conclude that in every iteration the flow on non-shortest paths is decreased or set to zero while the flow on the shortest path is increased.

In the original paper it is suggested to take \(\alpha\) equal to one, as it is reported to achieve good empirical convergence. The step-size used in the method is an approximation of a quasi-Newton step-size. The term \(s^k\gamma\) is defined as the sum of the derivatives of volume-delay functions of the uncommon links between the shortest path and \(\gamma\). We consider only uncommon links since the the flow shift between the shortest path and \(\gamma\) does not change the resulting flow on the common links.

In diagonal TAP the Jacobian matrix of volume-delay functions is diagonal. Thus, the derivatives of volume-delay functions correspond to the diagonal elements of the Jacobian. In the case of ATAP the off-diagonal elements might not be equal to zero and if we consider only diagonal elements we introduce an approximation in the step-size. In other words, we are ignoring the dependency between asymmetric cost functions, i.e. we are diagonalizing the problem.

Algorithm 1 Pseudocode for the GP algorithm

- Find the shortest path for each o-d pair
- Assign the demand on the shortest path for each o-d pair

while True do
  - Update path costs
  - Find the shortest path for each o-d pair
  if Convergence == True then
    Return
  else
    - Update path flows
  end if
end while

3.2.1 Complexity

Even though the algorithm conceptually balances all the origin-destination pairs at the same time, the implementation is restricted by the procedural nature of the hardware and therefore the computation cannot be parallelized completely. We can consider the complexity of each of the aforementioned procedures in order to evaluate the performance of the algorithm as a whole.

If the pairs \(w \in W\) are sorted so that pairs with the same origin are evaluated one after the other, it is possible to cache the latest computed shortest path tree instead of generating
the same tree for each different destination. Considering this optimization, if we let \( O(spt) \) be the complexity of the chosen shortest path tree algorithm, we find that the complexity of the shortest path tree procedure is \( O(N_o O(spt)) \) where \( N_o \) is the number of different origins.

The path cost update complexity depends on the number of paths that the algorithm generates during its execution. This number can grow exponentially but in practice only from 5 to 10 paths, \( N_p \), are discovered for each origin-destination pair near the equilibrium conditions. Assuming that each path has an average of \( O(N^{1/2}) \) links, the complexity of the path costs update procedure is \( O(N_o N_p N^{1/2}) \).

The convergence criterion computes a value considering each path of each origin-destination pair, therefore its complexity is simply \( O(N_o N_p) \). Similarly the path flows update procedure computes the updated flow for each path of each origin-destination pair and its complexity is also \( O(N_o N_p) \).

Summarizing the total complexity of the implemented algorithm is

\[
O(N_o O(spt) + O(N_o N_p N^{1/2}) + 2O(N_o N_p))
\]

### 3.2.2 Convergence

The convergence properties for the Jayakrishnan’s algorithm derive from the convergence proof of the gradient projection method of Bertsekas\[7\]. It can be shown that starting from any feasible solution there exists an \( \bar{\alpha} \) such that the sequence of flow assignments generated by the algorithm converges to an optimal solution to the problem, assuming that \( \alpha \in (0, \bar{\alpha}] \) and that the volume-delay functions are convex.

These assumptions hold in diagonal problem instances, however when applied to asymmetric problem instances the algorithm is not guaranteed to converge.
3.3 Sancho algorithm

The second algorithm implemented during the project is the mean projection (MP) algorithm proposed by Sancho[27]. The algorithm was proposed in May 2014 as an extension to the project. The method is a path-based projection algorithm for asymmetric TAP. In order to describe the algorithm we have to define the concept of active set.

For each origin-destination pair we define an active set as the set containing the paths which carry positive flow and eventually the shortest path. In each iteration the flow is shifted between paths in the active set. The active set for pair $w$ is denoted as $A^+_w$. Each path in the active set is known as an active path.

The algorithm pseudocode can be found in Algorithm 2. In the initialization, the active set for each origin-destination pair is initialized with the shortest path. Then we perform an all-or-nothing assignment on shortest path and update the path costs. For each pair $w$ we then compute the mean cost $\bar{c}_w$ and standard deviation $\sigma_w$ among the active paths. Then, we define $\eta$ as the difference between path cost and mean cost, if a path is active, or zero if a path is not active, that is

$$\eta_\gamma = \begin{cases} c_\gamma - \bar{c}_w, & \gamma \in A^+_w \\ 0, & \gamma \notin A^+_w \end{cases}$$

Then, we check convergence using one of the criteria defined in Section 3.1.2. If the convergence criterion is not satisfied, we update path flows and continue, otherwise we check for a path outside of the active set whose cost is lower than the mean cost. If such a path exists we add it to the active set and continue, otherwise the algorithm terminates.

The flow update formula for the algorithm is

$$h^{k+1}_\gamma = h^k_\gamma - \beta^k_w \eta^k_\gamma, \quad \forall \gamma \in \Gamma, \forall w \in W$$

The step-size $\beta$ is a nonnegative constant. However, before using it the step-size is bounded according to

$$\beta^k_w \leq \min \left\{ \frac{f^k_\gamma}{\eta^k_\gamma} : \eta_\gamma > 0, \gamma \in A^+_w \right\}$$

in order to maintain the nonnegativity constraints on path flows. We call this limited value of $\beta$ the step-size threshold.

If the convergence criterion is met we need an additional step in order to be sure that the solution cannot be improved further. We consider the paths outside of the active set and verify if there exist a path $\gamma$ such that

$$z_\gamma < 0$$

where $z_\gamma$ is defined as

$$z_\gamma = c_\gamma - \bar{c}_w$$
If we found such a path we add it to the active set and continue with the resolution, otherwise we report the current solution.

The projection scheme utilized in this algorithm derives directly from the Wardrop second principle \[1.1.2\]. Since at equilibrium the mean travel time is minimum, the algorithm tries to iteratively shift flows between active paths proportionally to the difference between the path cost and the mean cost for the origin-destination pair. The flow on paths with a cost lower than the mean cost is increased and the flow on paths with a cost higher than the mean cost is decreased. Thus, the algorithm moves the flow assignment towards a situation in which all active paths share the same cost, which is at the same time the mean and the minimum path cost. This situation corresponds to the user equilibrium.

**Algorithm 2** Pseudocode for the MP algorithm

- Find the shortest path for each o-d pair
- Initialize the active path set with the shortest path for each o-d pair
- Assign the demand on the shortest path for each o-d pair

**while** True **do**
- Update path costs
- Find the shortest path for each o-d pair
- Compute \( \bar{c}_w \) and \( \sigma_w \) for each \( w \in W \)
- Compute \( \eta \) for each path

**if** Convergence == False **then**
- Update path flows
**else**
- Define \( z \) for each non-active path
  **if** There is a path with \( z < 0 \) **then**
    - Update active set
  **else**
    Return
**end if**
**end if**
**end while**

### 3.3.1 Complexity

Even though from our empirical observations we have notice a convergent behavior, there is not a convergence proof for the MP algorithm, which should therefore be treated as an heuristic method. We can make the same considerations we made for the GP algorithm. One difference is that, if we consider the variance convergence criterion, this can be computed in \( O(N_o) \). Therefore the complexity of this second implementation is

\[
O(N_oO(spt)) + O(N_oN_pN^{1/2}) + O(N_oN_p) + O(N_o))
\]
3.3.2 Convergence

The mean projection algorithm is based on the intuition that if path flows are balanced in order to obtain paths with the same cost the flow assignment tends to an equilibrium assignment.

Even though this observation is correct, there does not exist a convergence proof for this method. Therefore, we will consider this algorithm as an heuristic procedure for the traffic assignment problem.
3.4 Panicucci et al. algorithm

The last algorithm we implemented is the double projection (DP) algorithm proposed by Panicucci et al. The algorithm was originally proposed for ATAP and it includes a double projection scheme based on the extra-gradient method of Khobotov. As already mentioned, due to the separable nature of the feasible region of the problem we can perform projections very efficiently considering one origin-destination pair at a time. In the following we denote with \( X_w \) the feasible region for each pair \( w \).

The pseudocode for the algorithm can be found in Algorithm 3. The algorithm is initialized as usual performing an all-or-nothing assignment on the shortest path. The original paper also present an all-at-once assignment strategy. This strategy consists of computing the shortest path trees sequentially for each origin-destination pair, while assigning the demand on the shortest path after each computation. The difference with the all-or-nothing procedure is that only the first shortest path is computed on an empty network. All other paths are computed on a congested network, where the congestion derive from the flows of the pairs that were considered previously in the procedure. Even though the paper shows that this procedure generates better feasible solution, we did not consider it in the implementation of the project. The main reason is that in our project we were not interested in implementing the fastest algorithm possible, but instead in comparing different algorithms implemented in a similar way.

The algorithm depends on three input parameters, namely \( \beta, \epsilon \in (0,1) \), and \( \bar{\alpha} > 0 \). The first two are step-size scaling parameters; they define the ratio at which the step-size is reduced in each iteration. Instead, \( \bar{\alpha} \) is the default initial step-size. After the initialization we search for shortest paths and check convergence. If we have not reached convergence we enter in the inner loop where the first projection is performed. This loop iteratively updates the step-size until it is greater than the current best approximation of the Lipschitz constant of the volume-delay functions. The step-size \( \alpha \) is updated according to

\[
\alpha^k = \min \left\{ \epsilon \cdot \alpha^k, \beta \cdot \frac{\|h^k - \bar{h}^k\|}{\|c(h^k) - c(\bar{h}^k)\|} \right\}
\]

until

\[
\alpha^k \leq \beta \cdot \frac{\|h^k - \bar{h}^k\|}{\|c(h^k) - c(\bar{h}^k)\|}
\]

where \( \bar{h}^k \) is the aggregated projected flow vector. Each disaggregated flow vector is computed as

\[
\bar{h}^k_w = \text{Proj}_{X_w}(h^k_w - \alpha^k c(h^k))
\]

where the projection operator is described in Section 3.4.1. After a suitable step-size has been found the inner loop terminates and we update the flow vector using

\[
\bar{h}^{k+1}_w = \text{Proj}_{X_w}(h^k_w - \alpha^k c(\bar{h}^k))
\]
CHAPTER 3. ASYMMETRIC TRAFFIC ASSIGNMENT ALGORITHMS

Notice that in this second projection costs are evaluated using the vector obtained in the first projection, $\bar{h}^k$. After the flow update the step-size is reinitialized according to

$$\alpha^{k+1} = \min \left\{ \epsilon \cdot \bar{\alpha}, \beta \cdot \frac{\|h^k - \bar{h}^k\|}{\|c(h^k) - c(\bar{h}^k)\|} \right\}$$

and the algorithm continues.

The convergence of the algorithm is guaranteed under the assumptions that the cost operator are pseudo-monotone and Lipschitz continuous on the feasible region.

3.4.1 Projection algorithm

The projection algorithm used in the paper is able to efficiently perform projections on the feasible region for each origin-destination pair. Assuming we want to project a vector $z \in \mathbb{R}^n_+$ onto the feasible region defined by

$$\sum_{\gamma \in \Gamma_w} h_{\gamma} = d_w, \quad \forall w \in W$$

we have

1. Set $k = 0$. Set each element $x_i$ of the projected vector $x$ to

$$x_i^k = z_i + \frac{1}{n} \left( d - \sum_{j=1}^{n} z_j \right)$$

2. If the projected vector is positive, $x^k \geq 0$, stop. Otherwise, compute the set $I$ of the indexes of positive elements of $x^k$, that is

$$I \equiv \{ i : x_i^k > 0 \}$$

3. Update the projected vector as

$$x_i^{k+1} = \begin{cases} 0, & \text{if } i \notin I \\ x_i^k + \frac{1}{|I|} \left( d - \sum_{j \in I} x_j^k \right), & \text{if } i \in I \end{cases}$$

and go to 2.

The algorithm performs simple operations and finds a feasible projected vector in at most $n$ iterations.
CHAPTER 3. ASYMMETRIC TRAFFIC ASSIGNMENT ALGORITHMS

Algorithm 3 Pseudocode for the DP algorithm
- Find the shortest path for each o-d pair
- Assign the demand on the shortest path for each o-d pair
while True do
  - Update path costs
  - Find the shortest path for each o-d pair
  if Convergence == True then
    - Return
  else
    while Step size is too large do
      - Compute the projected flow vector
      - Update step size
    end while
    - Update path flows
    - Reinitialize step size
end if
end while

3.4.2 Complexity

We now give an upper bound on the complexity of the double projection algorithm. Using the same assumptions we considered for the other algorithms we have that the shortest path computation, the cost update and the convergence test complexities are, respectively \( O(N_o O(spt)) \), \( O(N_o N_p N^{1/2}) \) and \( O(N_o N_p) \).

The inner loop complexity depends on the projection algorithm. Assuming \( \bar{n} \) is the maximum number of iterations of this algorithm, we have that the inner loop worst case complexity is

\[
O(in) = O(N_o \bar{n} + 2(N_o N_p N^{1/2}))
\]

The second term is due to the fact that in each iteration of the inner loop, the cost vector needs to be evaluated considering two different flow levels, the current flow and the projected flow.

Assuming that \( N_{in} \) is the maximum number of inner loop cycles we have that the overall complexity per iteration of the algorithm is

\[
O(O(N_o O(spt)) + O(N_o N_p N^{1/2}) + N_{in} O(in) + O(N_o N_p))
\]

3.4.3 Convergence

The double projection method is proven to converge assuming that volume-delay functions are Lipschitz continuous and pseudomonotone on the set of feasible flow assignments. The convergence proof derives from taking a step-size always smaller than an approximation of the Lipschitz constant of the volume-delay functions.

Therefore, this is the only algorithm that is proven to converge on instances of the asymmetric traffic assignment problem.
Chapter 4

Project development

The project started in February 2014 at Universitat Politècnica de Catalunya under the supervision of Professor Esteve Codina Sancho. The initial goal of the project was to implement Jayakrishnan’s algorithm \(3.2\) and analyze its performance for solving the asymmetric TAP. Later, the mean projection algorithm \(3.3\) was proposed by professor Sancho for a direct comparison between the methods. The project was then extended between years 2014 and 2015 at Politecnico di Milano under the supervision of Professor Edoardo Amaldi and Professor Mauro Passacantando, from Università di Pisa. The double projection algorithm \(3.4\) was implemented in this period for an additional comparison. In this chapter we describe the development phases of the project.

4.1 Software architecture

The software modules developed in the project were designed with simplicity and expandability in mind. The code was intended to be reasonably fast since we planned to test the algorithms on large-scale networks with thousands of nodes.

The programming language used for the project was C++, standard version 2011. The reason for this choice is twofold. The main reason is that C++ is highly versatile and supports high level data abstractions, while still naturally providing access to low level memory management. This versatility is a big advantage over other programming languages, especially in projects where low level memory management can make a great difference in term of performance. The second reason is that we had already access to some legacy code for TAP, written in C++. Even though in the end this legacy code was not used, its existence was a good source of inspiration in the early stages of the project. Another reason for the language choice was the compiled nature of C++, which makes it one of the fastest out-of-the-box programming language available.

In the following sections we describe the software modules implemented, their role in the complete software architecture and their interrelations. Each module was implemented as an object using the abstract data types provided by the language, i.e. a C++ class.
4.1.1 Solver module

The solver module is the main part of the project. This module is the first one to be created during initialization. Only one instance of solver exist in memory at any given time. The solver is responsible for the execution of the algorithms for asymmetric TAP. It contains all the information about the algorithm to be executed and its parameters, the convergence criterion to be used and the problem instance to be solved. This module implements all the logic necessary to solve the problem using the algorithms described in chapter 3. The resolution method used depends on the configuration. The public interface of class solver contains only two method: the public constructor and the method solve(). The constructor is used to configure the object and initialize all its members. The method solve() is used to solve the problem according to the configuration passed to the constructor. The source code for this object can be found in appendices A.1 and A.2.

4.1.2 Network module

As already mentioned each solver contains all the necessary information about the problem instance. This information is managed by the network module. The network module is responsible for managing the information about the network topology and it is used by the solver during the execution to update the network, e.g. updating the link cost vector after a flow update, or to extract relevant information, e.g. the active paths for an origin-destination pair. The network module is also responsible for all the logic associated with the shortest path tree subproblem. The module contains the node list, the link list and the set of origin-destination pairs. The public interface of the network module let users retrieve the current aggregated cost or flow vector, update path flows and link costs and discover the shortest path for each origin-destination pair. These methods are common to each of the three algorithms implemented and were grouped in a single interface for this reason. The source code for the network module can be found in appendices A.3 and A.4.

4.1.3 Other modules

Other modules were implemented to represent the data types related to the project. We implemented a separate class for each one of the following: node, link, origin-destination pair, path and volume-delay function (vdf). The node and link modules contain simply an identifier and some simple data structure, for example each node contains the list of its incoming and outgoing links. Each origin-destination pair object contains the list of discovered paths between its origin and destination and each path contains the list of links composing the paths as well as its current cost and current flow. The vdf object contains the parameters and the methods used to compute a link cost starting from its current flow. There is only one instance of vdf at any given time. Each link contains an handle to this vdf object, which is used to update the link cost when it is needed. The vdf object uses two different methods to compute the link cost depending on the caller being a priority link or non-priority link. The vdf module is also responsible for computing the cost derivatives used in the Jayakrishnan’s gradient projection algorithm.
4.2 Implementation details

In the following sections we describe in details the implementation choices we made during the project and the considerations behind them.

4.2.1 Initialization

The asymmetric TAP solver developed in the project can be run as a single executable file from command line. For the initialization we decided to use input text files. Each file has its own data format which was defined during the project. The configuration file is the only direct input the program expects. The configuration file contains one key-value pair per line. Each pair corresponds to an input parameter. These parameters include the algorithm to solve the problem, the step-size parameters, the convergence criterion to use, etc. The configuration file also includes the filepaths to the other files needed to run the program, the network file and the matrix file. The network file contains the topology of the network as a list of nodes and links. Each node has an unique identifier, not necessarily sequential, and a type, either centroid or not. Each link has a starting and ending node, a length, a capacity and a type, either priority or non-priority. The presence of non sequential identifiers was a problem that we solved by assigning a sequential id to every parsed node and link. This little trick also simplified the storage and access of the data in the corresponding data structures. The matrix files contains the trip rate matrix. For each element of this matrix the program generates the corresponding origin-destination pair. The initialization is performed by parsing these input files and it creates the corresponding data structures in memory. We chose this approach to initialization since it is really flexible, especially with respect to the experiments that were executed. With external configuration files we could run different algorithms on different networks by simply modifying one or two lines of the configuration file.

4.2.2 Data structures

The C++ language offers a rich standard library, known as standard template library or STL. All the software modules developed in the project make use of different basic data structures in the STL. We briefly describe the ones we have used in the project

- **array** - an array is a homogeneous fixed-size container stored in contiguous memory. Array in C++ are directly derived from the C language and therefore they are very basic. This drawback is compensated by the fact that they offer the fastest access time among all containers. Arrays were used for every fixed-size list, for example for the node and link lists.

- **vector** - a vector is a dynamically allocated array that can grow in size. Vectors are one of the most useful general purpose containers. One limitation of vectors is that elements can only be inserted on one end. In the project a vector was used wherever we needed a size varying container and a vector was not outperformed by another container.
• **deque (double-ended queue)** - a deque is a dynamically allocated array that can grow on both ends. The use of a double-ended queue was needed in the shortest path tree algorithms for storing the candidate list in the D’Esopo-Pape method (3.1.1).

• **valarray** - a valarray is a standard data structure representing a vector in mathematical sense, i.e. an ordered collection of a fixed number of elements from a certain domain. The advantage of valarrays is that they are optimized for mathematical operation on its elements or between two valarrays. For example, with valarrays it is possible to easily perform vector arithmetics by using the default language operators. Valarrays were used in vector equations, for example in the path equilibration step of the algorithms.

• **map** - a map is an associative array, i.e. an array where keys are not limited to be integer numbers. A map is a complex data structure with high management cost. We used a map for storing the path set for each origin-destination pair. This choice was due to the fact that paths needs to be uniquely identified in order for the algorithms to solve the problem correctly.

Apart from the standard template library, many C++ projects make use of the Boost C++ library. The Boost library provides a collection of community driven C++ implementations which try to extend the standard C++ library. In our project we decided to not use the Boost libraries since we evaluated that the integration effort would be superior to the effort of directly implementing the features we needed.

### 4.2.3 Projection operators

As already mentioned in chapter 3, the algorithms implemented during the project share most of their procedures. The characterizing factor in each algorithm is typically the way in which path flows are equilibrated. In the implementation we tried to standardize this procedure in order to obtain a more organized, cleaner code. We designed a standard projection method with the following signature

```cpp
vector projection(vector);
```

where the parameter is the vector to project and the return value is the projected vector. We then specialized the projection to the three algorithms by implementing the actual projection schemes. The actual data type used in the implementation of the projection is a valarray of doubles. This choice leads to a cleaner interface in the solver module and to a similar implementation of the three algorithms.

### 4.2.4 Path identification and storage

Identifying correctly each path on the network is a prerogative for the correctness of every algorithm considered in this project. This is due to the fact that the shortest path found as a solution to the subproblem must be added to the path set only if it is not already present. The problem of uniquely identifying paths was one of the main obstacles encountered during the implementation of the algorithms.
Since each path is unambiguously identified by its links, the first approach that was considered was to generate a string from the links identifiers and use it as path identifier. This solution seemed good since the corresponding hash function is a perfect hash and there cannot be any collision. Unfortunately, the implementation of this method showed that on large networks, the storage of the string identifiers requires a large number of bytes since each path has many links. Since the number of paths can grow exponentially during the execution of the algorithms this perfect hash approach was reconsidered for a less space demanding method.

The hash function that was actually implemented is still based on the path links identifiers, but instead of generating a string it generates an integer value. The pseudocode for this function can be found in Algorithm 4. The function initializes the system random number generator (rng) with a deterministic seed, in this case 0, and the hash is initialized with a random number generated by the rng. Then, for each link in the path the rng is reinitialized using the sum between the current hash and the link identifier as seed and finally the next hash is computed as the binary XOR of two random numbers extracted from the rng. Assuming that links identifiers are sorted for each path, it is easy to see that two link lists will generate the same hash value. However, since we do not have a perfect hash we are not guaranteed that two different link lists will generate different hash values. We tested this function on 600 thousands similar paths, i.e. paths sharing all links except one, without detecting collisions.

As already mentioned paths are stored in a map for each origin-destination pair. The key in the map record corresponds to the hash value just defined. We decided to use a map since the insertion operator of this container takes care of managing duplicates, that is, if a duplicate key is found the insertion fails. This was convenient from our perspective since we did not have to manage directly duplicate paths.

As a side note, Jayakrishnan [18] proposes an original tree structure to store paths with limited space requirements. We did not investigate this implementation, since our own implementation did not show particular weaknesses even when solving large problem instances and we decided to favor simplicity over performance when we had the choice.

Algorithm 4 Hash function for paths

```plaintext
function PATH_HASH(links)
    rand_seed(0);
    hash = rand();
    for all i in links do
        rand_seed(hash + i);
        hash = rand() ⊕ rand();
    end for
    return hash;
end function
```

4.2.5 Numerical precision

Each model introduces an approximation. In particular, representing real numbers in a binary computer introduces approximation and rounding errors. In numerical applications these errors
can be detrimental to the correctness of the model. As many other programming languages, C++ is susceptible to floating point errors derived from the limited precision inherent to binary hardware architectures. Arbitrary precision libraries exist in the standard, however the use of these libraries introduces additional overhead to every computation. For this reason, in our project we decided to deal with approximation errors directly, without using a specialized library. The numerical precision is configurable in the input files. Each value smaller in absolute value than this parameter is considered equal to zero. Even though these errors are difficult to identify, we tested our project thoroughly and we are confident that the results of the executions are not influenced by the numerical approximation we have enforced.

4.2.6 Functions and derivatives

Following the discussion on numerical precision, the problem is particularly relevant when we need to evaluate arbitrary real-valued functions. In the project we were in this situation when computing link costs and link derivatives from volume-delay formulas. The development of a general purpose solver for computing arbitrary function derivatives was outside the scope of this project. In fact, we only needed to implement two function derivatives, one for priority links and another for non-priority links. We decided to compute the derivative formulas and to hardcode them in the volume-delay function object.

For priority links we implemented the derivatives expression

$$\frac{\delta t_a(v)}{\delta v_a} = \frac{\gamma_a \alpha \beta}{(Mc_a)\beta} \cdot v_a^{\beta-1}$$

and for non-priority links we have

$$\frac{\delta t_a(v)}{\delta v_a} = \frac{be^{\theta_b(x_a(v)-1)}}{Mc_a(1 + e^{\theta_b(x_a(v)-1)})}$$

Notice that these formulas correspond to diagonal elements of the Jacobian matrix of volume-delay functions. We do not need to consider off-diagonal elements since in the Jayakrishnan’s algorithm [3.2] the step-size depends on diagonalized link costs.

4.2.7 Software tools

In this section we describe the software tools we used during the project development.

- **Eclipse IDE** - the main part of the project was implemented using the Eclipse IDE. Eclipse is an open source IDE with support for many programming languages. In particular we used the Eclipse CDT plugin for C/C++ development. Eclipse includes automatic build files generation, inline code hinting and many other useful features.

- **Google Test** - the testing of the project was performed using the Google Test framework for C++. Google Test is a complete framework for unit testing based on the xUnit architecture. It features a complete testing environment which enables testing through C macros definition. Another advantage of Google Test is the seamless way in which tests
are initialized and ran. Basic unit testing was performed on all the algorithm classes and
methods. We also defined a mock network in order to test the full algorithm’s execution.
Google Test does not come integrated with the Eclipse IDE, so we had to write our own
scripts for initializing and executing the project test suite. Google Test is freely available
and can be downloaded from its Google code repository.

- **Valgrind** - one of the main characteristics of C++ is the absence of a default garbage col-
lector. Designing C++ program therefore require a deep understanding of the underlying
memory abstractions and can lead to unexpected memory related errors such as memory
leaks or segmentation faults. In order the validate the project against this type of errors we
used the Valgrind tool. Valgrind is a code analysis tool that detects memory management
issues such as using a pointer to unallocated memory or losing a reference to an object
without deallocating it, thus causing a memory leak. Valgrind is easy to use and it is a
useful validation tool. Valgrind is available under the GNU General Public License 2 and
can be downloaded from the project website.

- **GNU gprof** - the profiling of the application was done using the GNU gprof profiler.
GNU gprof is a command-line based profiler that analyzes the execution of a program and
identifies possible bottlenecks. The profiling phase was really important for optimizing
the code and obtaining an efficient implementation of the algorithms. Like other profilers
GNU gprof runs the application code and outputs information about the estimated running
time of specific methods in the code. This information includes the number of calls to a
specific method, the percentage of computing time spent executing a function and/or its
subroutines, etc. This tool can be downloaded freely and its available for all major Linux
distributions.

- **gnuplot** - all the charts and plots presented in this document are realized using the gnuplot
program. Gnuplot is an interactive program for graph drawing based on a command-line
interface. With gnuplot it is possible to create 2d and 3d graphs of arbitrary functions.
Chapter 5

Computational results

In this chapter we present the computational results of the project. We tested the developed algorithms on different problem instances and compared their performances. In the following we denote with GP, MP and DP, respectively the gradient projection, mean projection and double projection algorithm.

All the graphs represent a merit function plotted against execution time in milliseconds. The merit functions defined in Section 3.1.2 are denoted respectively as Gap1, Gap2, and Gap3. In all the experiments we considered $\epsilon = 10^{-4}$, i.e. the execution terminates when the corresponding merit function, evaluated at the current solution, becomes less than $\epsilon$. In some experiments we have also fixed the maximum iteration number, so that if we reach the maximum number of iterations without having an $\epsilon$-optimal solution the execution also terminates.

As a side note, we point out that a merit function does not have to necessarily decrease in each iteration. This is because the value of the merit function plotted is evaluated after each shortest path search, but before the flow equilibration step. Thus, an increase in the merit function value might simply correspond to a new shortest path being discovered. This is especially true for Gap2 and Gap3, since both of these functions are defined using the maximum operator.

All the experiments were executed on a laptop PC running Fedora Linux with an Intel Core i3-380M processor and 4 GB of RAM.

5.1 Network instances

To evaluate the implemented algorithms we run experiments on six different problem instances each one corresponding to a network. The Sioux-Falls network is a well-known small-sized network in literature. Terrassa, Winnipeg and Hessen networks are medium to large-scale networks respectively representing the cities of Terrassa in Spain and Winnipeg in Canada and the region of Hessen in Germany. The characteristics of these networks are reported in table 5.1.

For each network we report the number of nodes, links and origin-destination pairs and the asymmetric link ratio $r$ defined as

$$r = \frac{|A_n|}{|A|}$$
The ratio represents the amount of asymmetric volume-delay functions in the network with respect to the number of links. This ratio is especially relevant when evaluating the Jayakrishnan’s algorithm since, as already mentioned, this method was originally proposed for the symmetric version of the problem and its performance has never been tested on asymmetric problem instances.

## 5.2 Shortest path tree algorithms

We begin this chapter with the performance analysis of the shortest path tree algorithms we implemented for solving the shortest path tree subproblem.

As already mentioned we compared Dijkstra’s algorithm, Bellman-Ford algorithm, D’Esopo-Pape algorithm and the Slf-Lll method. For the details about the algorithms see Section 3.1.1. Each of these methods generates a shortest path tree from a given origin assuming link costs are nonnegative.

To compare the algorithms we executed the solver on the networks of Sioux-Falls, Terrassa and Winnipeg using a different shortest path tree algorithm in each experiment. The average iteration time obtained with the different algorithms is reported in figures 5.1, 5.2 and 5.3 for the three networks.

From the histograms it can be seen that the D’Esopo-Pape method outperforms the other algorithms on all the considered networks. Notice that on the Sioux-Falls network Dijkstra’s algorithm is slower than the others, but on larger network the worse average iteration time is obtained with the Slf-Lll method. The Bellman-Ford method has performance similar to the D’Esopo-Pape method. This is not surprising considering that these algorithms differ only by the insertion policy in the candidate list. Another interesting thing that can be noted is that Dijkstra’s algorithm is outperformed by other methods with a worse worst-case complexity. The D’Esopo-Pape method was selected in all consequent experiments to solve the shortest path tree subproblem.

## 5.3 Step-size analysis

As already mentioned, the step-size is one of the most important choices when designing a projection scheme. Each of the algorithms implemented in this project requires an initial step-size choice. For many algorithms there is not an effective way to choose the step-size apriori, even though there are theoretical conditions that must be satisfied in order to guarantee convergence of the algorithm. In order to compare the algorithms in the best possible way, we performed

<table>
<thead>
<tr>
<th>Network</th>
<th>N</th>
<th>A</th>
<th>W</th>
<th>r</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sioux-Falls</td>
<td>48</td>
<td>124</td>
<td>552</td>
<td>0</td>
</tr>
<tr>
<td>Terrassa</td>
<td>1609</td>
<td>3264</td>
<td>2215</td>
<td>0.086</td>
</tr>
<tr>
<td>Winnipeg</td>
<td>1057</td>
<td>2535</td>
<td>4345</td>
<td>0.279</td>
</tr>
<tr>
<td>Hessen</td>
<td>4660</td>
<td>6674</td>
<td>17213</td>
<td>0.064</td>
</tr>
</tbody>
</table>

Table 5.1: Networks characteristics
Figure 5.1: Shortest path analysis - Sioux-Falls

Figure 5.2: Shortest path analysis - Terrassa
a preliminary step-size analysis to determine the best value of this parameter for each problem instance.

5.3.1 Jayakrishnan algorithm

With respect to the gradient projection algorithm, the original paper by Jayakrishnan proposes a step-size

\[ \alpha^k \frac{s^k}{s^k} \]

where \( \alpha^k = \bar{\alpha} > 0 \) is a positive constant and \( s^k \) depends on the volume-delay functions derivatives as described in Section 3.2. Our first choice for \( \bar{\alpha} \) was

\[ \bar{\alpha} = 1 \]

as it is suggested in the original paper.

Empirical observations pointed out that this choice was not suitable for the problem instances we considered. With the original value, and even smaller values, the method showed a zigzagging behavior in the value of the merit function. We found out that this behavior was caused by the fact that the path equilibration procedure continues to switch the whole demand between two paths. The path that is not used becomes the shortest path and this causes the whole demand to
be assigned to it in the following iteration. This continues for the majority of origin-destination pairs and it generates a cyclic behavior.

One common choice in literature is to choose a decreasing step-size defined as a function of the current iteration. We considered the choice

\[ \alpha_k = \frac{1}{\sqrt{k+1}} \]

where \( k \) is the iteration number. Since the flow shifts decrease as the step-size decreases, this approach should inhibit the oscillating behavior of constant step-sizes, at least in later iterations. Unfortunately this choice introduces one drawback: when the execution reaches an high iteration number, the flow shift is limited by the step-size and the quality of the solution improves very little between consecutive iterations. In addition to the choice \( \bar{\alpha} = 1 \), we experimented with various other constant values of \( \bar{\alpha} \in (0, 1) \). We found that the step-size for which the gradient projection shows the divergent behavior depends on the size of the network.

The graphs corresponding to these step-size choices are reported in figures 5.4, 5.5 and 5.6 respectively for the network of Terrassa, Sioux-Falls and Winnipeg; the experiments report 100 iterations of the gradient projection algorithm with various values for the step-size. The zigzagging behavior is reported for the network of Terrassa with a step-size of \( \bar{\alpha} = 0.5 \) or greater. For clarity we do not show this behavior in the other graphs, but it can be easily reproduced. It can be noticed that the solution with the decreasing step-size shows a similar behavior on Sioux-Falls and Terrassa, while diverges on Winnipeg and is not reported. On the Winnipeg network, step-sizes as small as \( \bar{\alpha} = 0.1 \) cause the zigzagging behavior.

The results show that for each network there is a range of step-size values that achieves good convergence. Unless otherwise noted, we selected the step-size value of \( \bar{\alpha} = 0.015 \) for all the following experiments with the GP algorithm.

### 5.3.2 Sancho algorithm

The mean projection algorithm \( (3.3) \), similarly to the GP algorithm, requires the definition of a default step-size. However, in this method the only constraint on \( \beta \) is nonnegativity. In our analysis we considered the same approach as in Section 5.3.1. Results are presented in figures 5.7, 5.8 and 5.9.

This algorithm does not show the zigzagging behavior described for the previous method. In fact, in this method the amount of flow shifted between paths is only limited by the nonnegativity constraints on path flows and the procedure does not admit the cyclic behavior exhibited by the gradient projection algorithm when the step-size is too large, since only one path flow can be set to zero for each origin-destination pair in one iteration.

Even in this case the performance of the method depends on the step-size choice. Choosing a large step-size means that the algorithm moves as much flow as possible in a promising direction in each iteration, however we can notice that a step-size too large, for example \( \beta = 10^6 \) causes the method to reduce its convergence speed after a few iterations. On larger networks like
CHAPTER 5. COMPUTATIONAL RESULTS

Figure 5.4: Step-size analysis for GP algorithm - Terrassa

Figure 5.5: Step-size analysis for GP algorithm - Sioux-Falls
Winnipeg the difference in the behavior of the algorithm with different step-size values tends to decrease. This is caused by the fact that with a larger number of origin-destination pairs the threshold on the flow shifted between paths is more often identified by an actual flow and not by the default step-size. We found that a step-size of 

$$\beta = 1000$$

was a reasonable choice in all the networks we considered.

### 5.3.3 Panicucci et al. algorithm

The double projection method of Panicucci et al. (3.4) uses an adaptive step-size limited by the best known approximation of the Lipschitz constant of volume-delay functions. This method did not show anomalous behavior on the considered networks so in our analysis we used the parameter values proposed in the original paper; these are $$\beta = 0.2$$, $$\xi = 0.5$$ and $$\bar{\alpha} = 10^6$$.

### 5.4 Algorithms comparison

In the following sections we present the computational results obtained comparing the three implemented algorithms on the networks of Sioux-Falls, Terrassa, Winnipeg and Hessen. On each network the algorithms are compared using the three convergence criteria defined in Section 3.1.2. Each method terminates when a solution is found within $$\epsilon$$ to the optimum, where $$\epsilon = 10^{-4}$$.
CHAPTER 5. COMPUTATIONAL RESULTS

Figure 5.7: Step-size analysis for MP algorithm - Terrassa

Figure 5.8: Step-size analysis for MP algorithm - Sioux-Falls
5.4.1 Sioux-Falls network

As already mentioned the Sioux-Falls network is a small-sized network that is widely used in literature. The network does not contain any asymmetric volume-delay function, i.e. is an instance of symmetric TAP. The maximum iteration number for this network was set to 1000. The results of the execution are reported in figures 5.10, 5.11 and 5.12.

Numerical results corresponding to merit function Gap1 and Gap2 are reported in table 5.2; we show for each method the number of iterations needed to reach convergence, the total elapsed time in seconds and the value of the merit function evaluated at the final solution. We do not report the numerical results for Gap3, since its definition as the maximum flow error, makes it highly sensible to the discovery of new paths, as it can be noticed from its graph. The $\epsilon$-optimal solution is reached by all three algorithms.

It can be noticed that on this problem instance the GP algorithm of Jayakrishnan is better then the other two methods, since it reaches a solution in less time. The DP method performs the least number of iterations but it generally takes more time per iteration. The quality of the solution found depends on the merit function used. Considering Gap1 we have that the quality is similar in all three methods, while considering Gap2 we can see that the MP and DP methods find a solution that is slightly more precise. In general, we can notice that the convergence criterion based on Gap2 is more difficult to satisfy since all the methods perform a larger number of iterations with respect to Gap1.
CHAPTER 5. COMPUTATIONAL RESULTS

Figure 5.10: Comparison on Sioux-Falls network - Gap1

Figure 5.11: Comparison on Sioux-Falls network - Gap2
5.4.2 Terrassa network

Terrassa network is a medium-sized network representing the city of Terrassa, Spain. Its asymmetric link ratio $r$ is 0.086, thus only a low percentage of links has asymmetric volume-delay functions. For these experiments we set the maximum iteration number to 200. The numerical results are reported in table 5.3. The symbol $-$ in the table denotes that the execution was stopped due to the maximum iteration being reached. The execution plots are showed in figures 5.13 and 5.14 for Gap1 and Gap2.

The only algorithm to reach convergence before the maximum iteration number is reached is the GP algorithm. The MP and DP methods achieve a reasonably precise solution below $10^{-3}$ for Gap1 and below $10^{-2}$ for Gap2. From the execution time comparison the GP algorithm is still the fastest, followed by the MP algorithm. Even though this instance is asymmetric, the gradient projection algorithm does not exhibit anomalous behavior during the resolution.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Iter</th>
<th>Time(s)</th>
<th>Gap1</th>
<th>Iter</th>
<th>Time(s)</th>
<th>Gap2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>33</td>
<td>1.268</td>
<td>9.93e-5</td>
<td>250</td>
<td>8.569</td>
<td>9.92e-5</td>
</tr>
<tr>
<td>MP</td>
<td>79</td>
<td>3.624</td>
<td>9.75e-5</td>
<td>357</td>
<td>17.319</td>
<td>4.19e-5</td>
</tr>
<tr>
<td>DP</td>
<td>32</td>
<td>2.781</td>
<td>9.78e-5</td>
<td>120</td>
<td>8.912</td>
<td>4.36e-5</td>
</tr>
</tbody>
</table>

Table 5.2: Numerical results for Sioux-Falls network
Figure 5.13: Comparison on Terrassa network - Gap1

Figure 5.14: Comparison on Terrassa network - Gap2
5.4.3 Winnipeg network

Winnipeg network is a medium-sized network representing the city of Winnipeg, Canada. Even though this network is smaller than Terrassa network the number of origin-destination pair is doubled. Since all developed algorithms decompose the problem with respect to these pairs, solving asymmetric TAP on the Winnipeg network is more demanding than on the Terrassa network. The maximum number of iteration for these experiments was set to 200. The GP step-size for this method was set to 0.005. The results for the Winnipeg network are reported in figures 5.15 and 5.16 and in table 5.4.

All three methods reach the maximum number of iterations without achieving convergence. The best solution found is the one provided by the DP algorithm, but the execution time is worse than the other methods. The fastest algorithm is MP, however the solution provided by this method on this problem instance is around two orders of magnitude worse than the other methods. This is also true considering Gap2 as merit function.

![Relative gap on Winnipeg network](image)

Figure 5.15: Comparison on Winnipeg network - Gap1

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Iter</th>
<th>Time(s)</th>
<th>Gap1</th>
<th>Iter</th>
<th>Time(s)</th>
<th>Gap2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>118</td>
<td>70.219</td>
<td>9.73e-5</td>
<td>-</td>
<td>120.530</td>
<td>7.02e-4</td>
</tr>
<tr>
<td>MP</td>
<td>-</td>
<td>128.683</td>
<td>8.37e-4</td>
<td>-</td>
<td>126.794</td>
<td>9.77e-3</td>
</tr>
<tr>
<td>DP</td>
<td>-</td>
<td>182.710</td>
<td>7.16e-4</td>
<td>-</td>
<td>184.437</td>
<td>1.00e-2</td>
</tr>
</tbody>
</table>

Table 5.3: Numerical results for Terrassa network
5.4.4 Hessen network

The Hessen network is the largest network we have considered in our experiments. The number of origin-destination pairs is 4 times larger than that of Winnipeg. The average iteration time on this instance is around 15 seconds per iteration. The maximum number of iterations considered for these experiments is 50. We kept the step-size for the gradient projection method to 0.005 as for the Winnipeg network. The results of these experiments are reported in figures 5.17 and 5.18 and in table 5.5.

The resolution of this problem instance shows that the MP algorithm is again outperformed by the other two methods. The solution provided by this algorithm is one order of magnitude worse than those provided by the GP and DP algorithms, considering both convergence criteria. The double projection algorithm requires almost double the execution time of the other methods and it obtains the same solution quality as the one obtained by the gradient projection algorithm.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Iter</th>
<th>Time(s)</th>
<th>Gap1</th>
<th>Iter</th>
<th>Time(s)</th>
<th>Gap2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>-</td>
<td>413.366</td>
<td>2.35e-4</td>
<td>-</td>
<td>443.782</td>
<td>2.17e-2</td>
</tr>
<tr>
<td>MP</td>
<td>-</td>
<td>389.357</td>
<td>4.43e-2</td>
<td>-</td>
<td>394.358</td>
<td>0.304</td>
</tr>
<tr>
<td>DP</td>
<td>-</td>
<td>618.943</td>
<td>1.98e-4</td>
<td>-</td>
<td>614.071</td>
<td>2.86e-2</td>
</tr>
</tbody>
</table>

Table 5.4: Numerical results for Winnipeg network
CHAPTER 5. COMPUTATIONAL RESULTS

Figure 5.17: Comparison on Hessen network - Gap1

Figure 5.18: Comparison on Hessen network - Gap2
CHAPTER 5. COMPUTATIONAL RESULTS

5.5 Summary

In the following section we summarize the results obtained by focusing on each of the considered algorithms one at a time. We will describe for each method the main advantages and drawbacks we have pointed out with our analysis and how each method compares against the others. In the next section we make some considerations about the merit functions considered.

5.5.1 Merit functions

In the development of our project we considered different merit functions in order to assess the quality of a solution. For our experiments we mainly used Gap1 and Gap2.

As already pointed out, Gap3 was not reported since it is defined as the maximum flow error. This definition implies that whenever a new path is found Gap3 is equal to one even if we are close to an optimal solution. Due to this fact plotting Gap3 against the execution time does not provide much insight since, if the algorithm does not converge completely, i.e. if the maximum iteration number is reached, the function is equal to one for most of the execution. This behavior can be noticed in figure 5.12.

Considering Gap1 and Gap2, we have already pointed out that Gap2 is more “demanding” than Gap1. In other words if we consider two solutions $a$ and $b$ such that $Gap1(a) = Gap2(b)$, then $b$ is closer to the optimum than $a$. This is an empirical consideration based on the results presented in this chapter. Additionally, we have notice that Gap1 presents a smoother behavior with respect to Gap2 for all the experiments we have considered. Apart from this behavior, we have found that the relative comparison of the algorithms does not depend on the particular merit function considered.

5.5.2 Jayakrishnan algorithm

The Jayakrishnan’s algorithm is a gradient projection algorithm for diagonal TAP. In each iteration the flow is shifted between non-shortest paths according to equation

\[ h^{k+1}_\gamma = \max \left\{ 0, h^k_\gamma - \frac{\alpha^k}{s^k_\gamma} (c_\gamma - \pi_{pq}) \right\} \quad \forall (p, q) \in W \]  

(5.1)

where $\alpha^k$ is a positive step-size and $s^k_\gamma$ is a scaling factor defined as a function of volume-delay function derivatives. For all the details refer to Section 3.2.

The computational results obtained show that the GP algorithm is better or equivalent to the DP algorithm in all the instances of the problem considered. However, with respect to the

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Iter</th>
<th>Time(s)</th>
<th>Gap1</th>
<th>Iter</th>
<th>Time(s)</th>
<th>Gap2</th>
</tr>
</thead>
<tbody>
<tr>
<td>GP</td>
<td>-</td>
<td>660.673</td>
<td>2.93e-3</td>
<td>-</td>
<td>669.881</td>
<td>5.96e-2</td>
</tr>
<tr>
<td>MP</td>
<td>-</td>
<td>598.428</td>
<td>4.09e-2</td>
<td>-</td>
<td>621.061</td>
<td>0.588</td>
</tr>
<tr>
<td>DP</td>
<td>-</td>
<td>1142.812</td>
<td>2.69e-3</td>
<td>-</td>
<td>1127.663</td>
<td>7.64e-2</td>
</tr>
</tbody>
</table>

Table 5.5: Numerical results for Hessen network
DP algorithm, GP requires less time to achieve the same level of precision since its time per iteration is lower.

The main drawback of this method, which was pointed out by the step-size analysis in Section 5.3.1, is that the performance of the method depends on the parameter $\alpha^k$ and there is not an effective technique to choose it without a preliminary analysis. We have shown that a bad step-size choice can even produce a method that does not converge. This behavior is shown in figure 5.4. Thus, in order to properly apply this method we need to perform an expensive analysis based on trial-and-error for each problem instance. This approach can be difficult to apply to large-scale instances, where each execution may require a significant amount of time.

5.5.3 Sancho algorithm

The Sancho’s algorithm is a path equilibration algorithm that uses the following flow update equation

$$h^{k+1}_\gamma = h^k_\gamma - \beta^k_w \eta^k_\gamma \quad \forall \gamma \in \Gamma_w, \forall w \in W$$

(5.2)

where $\beta^k_w$ is a nonnegative constant, which for each origin-destination pair is limited by

$$\beta^k_w \leq \min \left\{ \frac{f^k_\gamma}{\eta^k_\gamma} : \eta_\gamma > 0, \gamma \in A_w^+ \right\}$$

(5.3)

For the details about the method see Section 3.3. The analysis in Section 5.3.2 shows that the step-size choice is crucial also in this method. In this case however the step-size choice does not depend on the particular problem instance. We have shown that a choice of $\beta = 1000$

performs well enough on all the networks considered. This behavior can be explained analyzing equations (5.2) and (5.3). The limit on the step-size is computed for each origin-destination pair so that the flow update procedure does not generate negative flows. By choosing a large value for $\beta$ we are forcing the algorithm to use the limiting value for the majority of origin-destination pairs. In this way we are speeding up the method, since in general we want to move as much flow as possible in the descent direction defined by the method.

However, from the analysis has emerged that choosing a step-size too large is not beneficial to the convergence of the method near the optimal solution. From our understanding this is due to the fact that near the optimal solution the difference between path cost and mean path is small and the scaling factor provided by the step-size is too large to precisely adjust the flow vector. Thus, the convergence speed is penalized.

From the results it can be noticed that the solution provided by this method are less precise than the ones provided by the GP and DP method. This is especially clear on the network of Hessen. In general, the MP algorithm spend less time per iteration than the other two algorithms, but its convergence speed is lower. On small problem instances, for example the Sioux-Falls network, the MP algorithm tends to be slower than the other two since it needs more iterations to reach the same level of precision.
One advantage of this method is that its step-size is not sensible to the problem instance. One major drawback is that the descent direction of this method is generally worse than the other algorithms. This can be seen from the results we have reported, especially on larger networks. Another drawback is that there is not a convergence proof for the method in the conditions we have experimented.

5.5.4 Panicucci et al. algorithm

The Panicucci et al. algorithm is a double projection path-based algorithm for asymmetric TAP. The method is described in details in Section 3.4. The step-size in this method is computed repeatedly in each iteration, until it is smaller than the current best approximation of the Lipschitz constant of the volume-delay functions. This method shows a good behavior in all the experiments we have performed.

In many experiments the DP method is outperformed by the gradient projection method. However, it is important to point out that the double projection algorithm does not need the step-size analysis to determine the correct step-size for the given instance. As already mentioned this is a major drawback of the GP method with respect to the double projection algorithm. One major advantage of the method is that its convergence is guaranteed if the volume-delay functions are Lipschitz continuous and pseudomonotone.

On larger problem instances the double projection algorithm takes double the time per iteration needed by the other methods. However, this is compensated by the better descent direction generated by this algorithm. On smaller instances the algorithm outperforms the mean projection method. In conclusion, this method is better than the MP algorithm and shows a better out-of-the-box performance with respect to the GP algorithm. This is obtained by spending more time per iteration.
Chapter 6

Concluding remarks

In this project we have implemented and compared three path-based algorithms for the asymmetric traffic assignment problem. The traffic assignment problem is a network equilibrium problem that is found in various fields such as transportation planning, game theory and economics. Given a directed weighted graph and a trip rate matrix the problem consists of finding a flow assignment on the network that satisfies the demand for each origin-destination pair, under the user equilibrium conditions (1.1.4). The formulation of the problem depends on the cost operators that determine the link weights.

If each link weight depends in general on the whole link flow vector the problem is known as asymmetric traffic assignment. The traditional formulation of the asymmetric traffic assignment problem is based on a variational inequality problem (1.1.6). A variational inequality problem is a mathematical problem particularly useful when dealing with equilibrium problems. The resolution of the traffic assignment problem usually implies the resolution of the shortest path problem as a subproblem. To solve this subproblem we implemented various algorithms including the Dijkstra’s algorithm and the Bellman-Ford algorithm.

The algorithms we implemented for the traffic assignment problem are the Jayakrishnan’s gradient projection algorithm, Section 3.2, the mean projection algorithm of Sancho, Section 3.3 and the double projection algorithm of Panicucci, Pappalardo and Passacantando, Section 3.4. Each one of them can be categorized as a path-based projection algorithm. Until recently path-based algorithms were considered to be unpractical due to their large memory requirements. All three algorithms were implemented in C++ starting from scratch.

Computational experiments were executed on networks of various sizes. We considered the Sioux-Falls, Terrassa, Winnipeg and Hessen network. The Sioux-Falls network is a symmetric instance of the problem very well-known in the literature. The other networks are real-world networks representing the Terrassa and Winnipeg cities and the Hessen region.

We initially compared the shortest path tree algorithms. The analysis showed that the D’Esopo-Pape algorithm achieves the best results on the considered networks.

We noticed that the performances of the Jayakrishnan’s and Sancho’s algorithm are sensible to their parameters. For this reason we performed a comparative analysis using different configurations. We have shown that in the gradient projection algorithm the optimal parameters depend on the problem instance considered, while in the mean projection algorithm the param-
eters are not sensible to the problem instance. We then selected the parameters that achieved the best performance and carried out the final analysis considering all three algorithms.

In the experiments we used three different convergence criteria for comparing the algorithms’ performance. We showed that the choice of convergence criterion does not affect the relative performance comparison of the algorithms.

Although the only algorithm proven to converge on asymmetric problem instances is the double projection algorithm, we showed that both the mean projection and gradient projection methods exhibit empirical convergence in the computational experiments we have performed. One exception is the gradient projection algorithm of Jayakrishnan that loses convergence when the step-size is too large.

Our experiments showed that the gradient projection algorithm obtains the best convergence speed on all the considered networks. The mean projection algorithm is generally slower in achieving convergence with respect to the other methods and it generates less precise solutions. The double projection algorithm is the slowest of the considered methods. However this method is the most consistent when considering different networks. In particular, the double projection method does not require an initial analysis to determine the best parameters for each problem instance.

This thesis has shown that path-based projection algorithms are viable to solve the asymmetric traffic assignment problem on medium to large-scale networks. Among the considered algorithms, the gradient projection algorithm of Jayakrishnan and the double projection algorithm of Panicucci et al. show the best performance. The gradient projection algorithm needs an initial parameter study for each problem instance in order to achieve good performance. Without this analysis the method does not perform well, and for some parameter choices it can be shown that the method does not converge. On the other hand the double projection method is more consistent and is convergent under mild assumptions. We object that this method is preferable on instances where a parameter study is impractical and convergence of the method is required. Finally, even if our analysis shows that the mean projection algorithm is outperformed by the other methods in the majority of the experiments, this algorithm can still be useful in problems where inaccurate fast solutions are preferable to precise solutions obtained in more time.

This project could be extended in various ways. The analysis of the algorithms could be extended by considering various congestion levels. This can be done multiplying the trip rate matrix by a scaling factor and comparing the algorithm performances in these conditions.

From the results we have obtained we think that the gradient projection method could be improved with the inclusion of an adaptive step-size computed automatically in each iteration.

The study of evaluation methods for algorithms for the traffic assignment problem has not yet been addressed much by researchers. We think that the development of robust evaluation methods could drive this extremely wide research field to more meaningful and interesting results.
Appendix A

ANNEX: Source code

We report the source code for the solver and network modules. These modules represent the main part of the project. We do not claim the implementation to be optimal in any way. In fact, in many cases we decided to favor readability and simplicity over code optimization. Nonetheless, we have tried to be fair in the implementation of the three algorithms so that computational results would not be affected.

A.1 Solver.h

/∗ Solver.h ∗
∗ Author: sivam pasupathipillai
∗ Email: sivam.pasu@gmail.com
∗/

#ifndef SOLVER_H
#define SOLVER_H

#include "Network.h"

enum class AtaAlgorithm {Gp, Mp, Dp};
enum class ConvergenceTest {Gap, Var, Max};

class Solver {
private:
    // Ata algorithm used
    AtaAlgorithm ATA_ALG;
    // Convergence test used
    ConvergenceTest CONV_TEST;
    // Required precision for the solution
    double EPSILON;
    // Alpha parameter for stepsize
    double STEP_ALPHA;
    // Beta parameter for stepsize
    double STEP_BETA;
};
// Xi parameter for stepsize
double STEP_XI;
// Filepath for output
std::string OUT_FILENAME;
unsigned int MAX_ITER;
double PRECISION;
// Network instance
Network net;

// Solve problem using GP algorithm
void solve_gradient_projection();
// Solve problem using MP algorithm
void solve_mean_projection();
// Solve problem using DP algorithm
void solve_double_projection();
// Print the current solution in a readable format
void print_solution(std::vector<OdPair>∗ ods);

// Perform gradient projection
std::valarray<double> gradient_projection(std::vector<OdPair>∗ ods, unsigned int iteration);
// Perform mean projection
std::valarray<double> mean_projection(double step_beta, std::valarray<double> gammas, const std::vector<OdPair>∗ ods);
// Perform double projection
std::valarray<double> double_projection(double step_alpha, const std::vector<OdPair>∗ ods);
// Return default stepsize as a function of the iteration
double get_default_stepsize(int iter);

// Gradient projection
std::vector<int> find_uncommon_links(const std::vector<int>∗ a, const std::vector<int>∗ b);

// Mean projection
// Contains active paths ids or each od
std::vector<OdPair>∗ mean_active_ids;
// Update active path structure
void mean_set_active_paths(std::vector<OdPair>∗ ods);
// Clear active path structure
void mean_clear_active_paths();
// Return od active paths
std::valarray<bool> get_active_paths(unsigned int od_index);
// Checks if unused paths are convenient
bool mean_check_unused_paths(const std::valarray<double>∗ costs, std::vector<OdPair>∗ ods);
// Get vector of gammas
std::valarray<double> mean_get_gamma_vector(const std::valarray<double>∗ costs, const std::vector<OdPair>∗ ods);
// Get stepsize threshold
std::valarray<double> mean_get_step_threshold(const std::valarray<double>∗ costs, const std::vector<OdPair>∗ ods);
// Get mean costs for each od
std::valarray<double> mean_get_active_mean_costs(const std::vector<OdPair>∗ ods);
// Get std devs for each od
std::valarray<double> mean_get_active_std_devs(const std::valarray<double>∗ costs, const std::vector<OdPair>∗ ods);
// Get mean cost on active paths for od
# Appendix A. Annex: Source Code

```cpp
double mean_get_od_mean_cost(unsigned int od_index, const std::vector<OdPair>* ods);
// Get std dev on active paths for od
double mean_get_od_std_dev(unsigned int od_index, double od_mean, const std::vector<OdPair>* ods);

// Double projection
std::valarray<double> double_fast_projection(const std::valarray<double>& to_project, double demand);
double double_get_step_threshold(const std::valarray<double>& projected, double step_beta);
unsigned int get_min_index(const std::valarray<double>& cost_vector, double min_cost);

// Convergence criteria
std::valarray<double> get_mean_costs(const std::vector<OdPair>* ods);
std::valarray<double> get_std_devs(const std::valarray<double>& mean_costs, const std::vector<OdPair>* ods);
double get_relative_gap_criterion(const std::vector<OdPair>* ods);
double get_variance_criterion(const std::valarray<double>& mean_costs, const std::valarray<double>& std_devs);

// Printing functions
void print_init();
void print_header(unsigned int iter);
void print_footer(unsigned int iter, double merit, double prev, long it_time, long el_time);

public:
Solver(const std::string& conf_filename);
void solve();
};

#endif /* SOLVER.H */
```
A.2 Solver.cpp

/*
 * Solver.cpp
 *
 * Author: sivam pasupathipillai
 * Email: sivam.pasu@gmail.com
 */

#include <algorithm>
#include <cmath>
#include <fstream>
#include <iostream>
#include <sstream>
#include <chrono>
#include <limits>
#include "Solver.h"
#include "Util.h"

std::vector<int> Solver::find_uncommon_links(const std::vector<int>& a, const std::vector<int>& b) {  
    // Vector with the sum of lengths as length
    std::vector<int> uncommon(a.size()+b.size());
    // itr to use the STL method
    std::vector<int>::iterator itr;
    // Find uncommon links
    itr = std::set_symmetric_difference(a.begin(),a.end(),
        b.begin(),b.end(),
        uncommon.begin());
    // Resize vector to actual size
    uncommon.resize(itr-uncommon.begin());
    return uncommon;
}

/*
 * Computes the mean active path cost for each od pair
 */
std::valarray<double> Solver::get_mean_costs(const std::vector<OdPair>* ods) {

    // Compute the mean costs only for active paths
    std::valarray<double> mean_costs(ods->size());
    for (unsigned int i = 0; i < ods->size(); i++) {
        // Compute mean cost
        std::valarray<double> od_costs{(*(ods)[i]).get_costs()};
        std::valarray<double> od_flows{(*(ods)[i]).get_flows()};
        double cost_sum{0.0};
        int count{0};
        for (unsigned int i = 0; i < od_costs.size(); i++) {
if ((od_flows[i] > 0) || (od_costs[i] == od_costs.min())) {
    cost_sum += od_costs[i];
    count++;
}
mean_costs[i] = cost_sum/count;
return mean_costs;
/*
* Computes the std deviation of the active path costs for each od pair
*/
std::valarray<double> Solver::get_std_devs(const std::valarray<double>& mean_costs, const std::vector<OdPair>* ods) {
    // Compute the std devs only for active paths
    std::valarray<double> std_devs(ods->size());
    for (unsigned int i = 0; i < ods->size(); i++) {
        std::valarray<double> od_costs(*ods)[i].get_costs();
        std::valarray<double> od_flows(*ods)[i].get_flows();
        double od_mean{mean_costs[i]};
        double sum_sqr_err{0.0};
        int count{0};
        for (unsigned int i = 0; i < od_costs.size(); i++) {
            if (((od_flows[i] > 0) || (od_costs[i] == od_costs.min()))) {
                sum_sqr_err += std::pow(od_costs[i] - od_mean, 2);
                count++;
            }
        }
        std_devs[i] = std::sqrt(sum_sqr_err/count);
    }
    return std_devs;
}

/*
* Convergence criterion based on
* differences between minimum path flow
* assignment and current assignment.
*/
double Solver::get_relative_gap_criterion(const std::vector<OdPair>* ods) {
    // Initialize flows to zero
    std::valarray<double> perfect_flows(0.0, net.get_path_dimension());
    // Aggregated path flows
    std::valarray<double> current_flows{net.get_flow_vector()};
    // Aggregated path costs
    std::valarray<double> current_costs{net.get_cost_vector()};
    // Offset to traverse aggregated vectors
    unsigned int od_offset{0};
    // For each od pair
    for (auto& od : *od) {

// Od cost vector
std::valarray<double> od_costs{od.get_costs()};

// Find shortest path index
unsigned int min_index{get_min_index(od_costs, od_costs.min())};

// Assign all flow to shortest path
perfect_flows[od_offset + min_index] = od.get_demand();
od_offset += od_costs.size();

// Compute the modules and evaluate the ratio
return std::abs(dot_product(current_costs,(perfect_flows−current_flows)))/
    std::abs(dot_product(current_costs, perfect_flows));

/*
* Computes the convergence criterion as
* the biggest ratio between mean cost and
* std dev for each od pair
*/
double Solver::get_variance_criterion(const std::valarray<double>& mean_costs, const std::valarray<double>& std_devs) {
    // Maximum deviation error
    double max_error{0.0};
    // For each od pair
    for (unsigned int i = 0; i < mean_costs.size(); i++) {
        // Compute deviation error
        double od_error{std_devs[i]/mean_costs[i]};
        // Update maximum
        if (od_error > max_error) {
            max_error = od_error;
        }
    }
    return max_error;
}

/*
* Computes the convergence criterion as
* the maximum ratio between non optimal flow
* and od demand
*/
double Solver::get_max_flow_criterion(const std::vector<OdPair>* ods) {
    // Maximum error
    double max_err{0.0};
    // For each od
    for (const auto& od : *ods) {
        // Od flows
        std::valarray<double> od_flows{od.get_flows()};
        // Od costs
        std::valarray<double> od_costs{od.get_costs()};
        // Od min cost
        double min_cost{od_costs.min()};
        // Sum of non optimal flows
        double od_no_opt_flow_sum{0.0};
// For each path
for (unsigned int i = 0; i < od_flows.size(); i++) {
    // If path is non optimal
    if ( ((od_costs[i] - min_cost)/min_cost) > EPSILON ) {
        // Add flow to total sum
        od_no_opt_flow_sum += od_flows[i];
    }
}

// Od error
double od_err = od_no_opt_flow_sum/od.get_demand();
// If od error is maximum
if ( od_err > max_err ) {
    // Update maximum
    max_err = od_err;
}
return max_err;

/*
 * Returns index of shortest path from the cost vector
 */
unsigned int Solver::get_min_index(const std::valarray<double>& cost_vector, double min_cost)
{
    // Find the shortest path index
    unsigned int min_index = 0;
    while (cost_vector[min_index] != min_cost) min_index++;
    return min_index;
}

/*
 * Computes the projected vector using
 * gradient projection.
 */
std::valarray<double> Solver::gradient_projection(std::vector<OdPair>* ods, unsigned int iteration)
{
    // Vector to store the projected vector
    std::valarray<double> projected(net.get_path_dimension());
    unsigned int od_offset = 0;
    // For each od pair
    for (auto& od : *ods) {
        // Get the cost and flow vectors
        std::valarray<double> od_flows{od.get_flows()};
        std::valarray<double> od_costs{od.get_costs()};
        // Check if the sizes make sense
        if (od_flows.size() != od_costs.size()) {
            PRINT("ERROR:gradient_projection():\nPath flows do not match path costs.");
            throw std::invalid_argument("ERROR:gradient_projection():\nCost and flow sizes do not match");
        }
        // Find the minimum cost
        double min_cost = od_costs.min();
        // Find the shortest path index
        unsigned int min_index = get_min_index(od_costs, min_cost);
    }
/\* Find the links composing the shortest path \*/
std::vector<Path *> od_paths{od.get_paths()};
std::vector<int> min_links{od_paths[min_index]->get_path_links()};

/* Compute the scaling parameter for each path */
std::valarray<double> s_vec(od_paths.size());

/* Compute the s vector for stepsize */
for (unsigned int i = 0; i < od_paths.size(); i++) {
    Path * path_ptr{od_paths[i]};
    s_vec[i] = net.get_derivatives_sum(find_uncommon_links(path_ptr->get_path_links(), min_links));
}

/* Initialize parameters */
double alpha{STEP_ALPHA};
double assigned_dem{0.0};

/* Compute projection for the od pair */
for (unsigned int i = 0; i < od_flows.size(); i++) {
    if (i != min_index) {
        // Update flow
        double new_flow = nonnegative(od_flows[i] - ((alpha/s_vec[i])*(od_costs[i] - min_cost));
        projected[od_offset+i] = new_flow;
        // Keep track of demand assigned to paths
        assigned_dem += new_flow;
    }
}

/* Update shortest path flow with remaining demand to maintain feasibility */
projected[od_offset + min_index] = od.get_demand() - assigned_dem;

/* Update offset to keep the projected vector consistent */
od_offset += od_flows.size();

/* Return projected flow vector */
return projected;

void Solver::solve_gradient_projection() {
    unsigned int iteration{0};
    // Initially flows are zero from the link default constructor
    // Find the spt for each od pair and add it to the active set
    net.spt_search();
    // Assign the whole demand to the only path in the active set
    // which is the shortest path
    net.update_flows(net.get_demand_vector());
    // Cost update
    double currentMerit{0.0};
double previousMerit{0.0};
    std::vector<OdPair> * ods{net.get_od_pairs()};
    while (iteration <= MAX_ITER) {
        // Update costs
net.update_costs();
// Direction finding
net.spt_search();
// Compute gradient projection and
// update flows to compute the convergence criterion
// Get merit function for convergence check
if (CONV_TEST == ConvergenceTest::Gap) {
    current_merit = get_relative_gap_criterion(ods);
}
else if (CONV_TEST == ConvergenceTest::Var) {
    std::valarray<double> mean_costs{get_mean_costs(ods)};
    current_merit = get_variance_criterion(mean_costs, get_std_devs(mean_costs, ods));
}
else if (CONV_TEST == ConvergenceTest::Max) {
    current_merit = get_max_flow_criterion(ods);
}
else {
    PRINT("ERROR solve gradient projection(): \\
           invalid convergence test: \\
           " << CONV_TEST);
    throw std::invalid_argument("ERROR solve gradient projection(): invalid convergence test");
}
if (current_merit < EPSILON) {
    break;
}
net.update_flows(gradient_projection(ods, iteration));
// Store previous merit
previous_merit = current_merit;
iteration++;}

/*
 * Returns a vector containing an element
 * for each path corresponding to the difference
 * between path cost and mean od cost.
 * Defined only for active paths
 */
std::valarray<double> Solver::mean_get_gamma_vector(const std::valarray<double> & mean_costs, const std::vector<OdPair> * ods) {
    // Initialize vector
    std::valarray<double> gammas(net.get_path_dimension());
    // Aggregated vector offset
    unsigned int od_offset{0};
    // For each od pair
    for (unsigned int i = 0; i < ods->size(); i++) {
        std::valarray<double> od_costs{(od)[i].get_costs()};
        std::valarray<bool> active_paths{get_active_paths(i)};
        // For each path in od pair
        for (unsigned int j = 0; j < od_costs.size(); j++) {
            if (active_paths[j]) {
                gammas[od_offset + j] = od_costs[j] - mean_costs[i];
            }
        }
    }
    return gammas;
}
APPENDIX A. ANNEX: SOURCE CODE

gammas[od_offset + j] = 0;
}
}
// Update index
od_offset += od_costs.size();
return gammas;
}

/**
 * Computes the flow threshold for each od pair. The flow threshold is the maximum stepsize that maintains a positive flow
 */
std::valarray<double> Solver::mean_get_step_threshold(const std::valarray<double>& gammas, const std::vector<OdPair>* ods)
{
// Initialize vector with correct size
std::valarray<double> od_thresholds(ods->size());
// Aggregated index vector
unsigned int od_offset{0};
// For each od pair
for (unsigned int i = 0; i < ods->size(); i++) {
    double min_threshold{POS_INF};
    std::valarray<double> od_flows(*(ods)[i].get_flows());
    std::valarray<bool> active_paths{get_active_paths(i)};
    // For each path in od
    for (unsigned int j = 0; j < od_flows.size(); j++) {
        // If the path is active
        if (active_paths[j] && (gammas[od_offset + j] > 0)) {
            // Path threshold
            double path_threshold{od_flows[j] / gammas[od_offset + j]};
            // Update minimum
            if (path_threshold < min_threshold) {
                min_threshold = path_threshold;
            }
        }
    }
    // Fill threshold vector
    od_thresholds[i] = min_threshold;
    od_offset += od_flows.size();
}
return od_thresholds;
}

/**
 * Computes the projected flow vector for the mean projection algorithm
 */
std::valarray<double> Solver::mean_projection(double step_beta, std::valarray<double> gammas, const std::vector<OdPair>* ods)
{
    // Gamma vector

std::valarray<
  double
>
od_thresholds(mean_get_step_threshold(gammas, ods));
// Projected vectors
std::valarray<
  double
>
projected(net.get_path_dimension());
unsigned int
od_offset{0};
// For each od pair
for (unsigned int
  i = 0; i < ods->size(); i++) {
  // Od flow vector
  std::valarray<
    double
  >
od_flows{*ods[i].get_flows();}
  // Minimum threshold in od pair
  step_beta = min(step_beta, od_thresholds[i]);
  // For path in od
  for (unsigned int
    j = 0; j < od_flows.size(); j++) {
    // Compute projected flow
    double
    new_flow{od_flows[j] – step_beta*gammas[od_offset + j]};
    projected[od_offset + j] = round_zero(new_flow, PRECISION);
  }
  od_offset += od_flows.size();
}
return projected;
*/
/* Returns an array of booleans where
true corresponds to an active path
for the selected od pair */
std::valarray<
  bool
>
Solver::get_active_paths(unsigned int
  od_index)
{
  OdPair
od{net.get_od_pair(od_index)};
  std::vector<
    Path>
  od_paths{od.get_paths();}
  std::vector<int>
  od_active_ids{&mean_active_ids[od_index]};
  std::valarray<
    bool
  >
active_paths(false, od_paths.size());
  // For each path in od
  for (unsigned int
    i = 0; i < od_paths.size(); i++) {
    // If is active
    if (std::find(od_active_ids->begin(), od_active_ids->end(), od_paths[i]->get_id())
      != od_active_ids->end()) {
      // Set path to true
      active_paths[i] = true;
    }
  }
  return active_paths;
}
/* Computes the mean cost considering only
active paths (carrying positive flows) and the
shortest path */
double
Solver::mean_get_od_mean_cost(unsigned int
  od_index, const
  std::vector<
    OdPair>
  * ods)
{
  double
  sum_cost{0.0};
  unsigned int
  count{0};
std::valarray<double> od_costs{(*ods)[od_index].get_costs()};
std::valarray<bool> active_paths{get_active_paths(od_index)};

// For each od path
for (unsigned int i = 0; i < od_costs.size(); i++) {
  if (active_paths[i]){
    sum_cost += od_costs[i];
    count++;
  }
}
return sum_cost/count;

/*
 * Computes the standard deviation
 * considering only active paths and
 * the shortest path
 */
double Solver::mean_get_ods_std_dev(unsigned int od_index, double od_mean, const std::vector<OdPair>* ods)
{
  double sum_sqr_err{0.0};
  unsigned int count{0};
  std::valarray<double> od_costs{(*ods)[od_index].get_costs()};
  std::valarray<bool> active_paths{get_active_paths(od_index)};
  // For each od path
  for (unsigned int i = 0; i < od_costs.size(); i++) {
    if (active_paths[i]){
      sum_sqr_err += std::pow(od_costs[i] - od_mean, 2);
      count++;
    }
  }
  return std::sqrt(sum_sqr_err/count);
}

/*
 * Computes the active mean cost
 * for each od pair
 */
std::valarray<double> Solver::mean_get_active_mean_costs(const std::vector<OdPair>* ods)
{
  std::valarray<double> mean_costs(ods->size());
  for (unsigned int i = 0; i < ods->size(); i++) {
    // Compute mean cost
    mean_costs[i] = mean_get_od_mean_cost(i, ods);
  }
  return mean_costs;
}

/*
 * Computes the active mean cost
 * for each od pair
 */
```cpp
std::valarray<double> Solver::mean_get_active_std_devs(  
    const std::valarray<double>& mean_costs,  
    const std::vector<OdPair>* ods)
{
    std::valarray<double> std_devs(od_size);
    for (unsigned int i = 0; i < od_size; i++) {
        std_devs[i] = mean_get_od_std_dev(i, mean_costs[i], ods);
    }
    return std_devs;
}

void Solver::mean_clear_active_paths()
{
    unsigned int od_size{net.get_od_dimension()};
    for (unsigned int i = 0; i < od_size; i++) {
        mean_active_ids[i].clear();
    }
}
```
/∗
 * Computes the active set for each od pair
 */
void Solver::mean_set_active_paths(std::vector<OdPair>* ods) {
    // For each od pair
    for (unsigned int i = 0; i < ods->size(); i++) {
        // Od path costs
        std::valarray<double> od_costs{(*ods)[i].get_costs()};
        // For each od path
        for (const auto& path_ptr : (*ods)[i].get_paths()) {
            // If path carries positive flow or is minimum
            if (path_ptr->get_flow() > 0 || (path_ptr->get_cost() == od_costs.min())) {
                // Set it as active
                mean_active_ids[i].push_back(path_ptr->get_id());
            }
        }
        if (mean_active_ids[i].empty()) {
            PRINT("ERROR mean_set_active_paths():\nEmpty active set for od <" << (*ods)[i].get_origin() << "," << (*ods)[i].get_destination() << ";\nAn active set is empty");
            throw std::runtime_error("ERROR mean_set_active_paths():\nAn active set is empty");
        }
    }
}

/*
 * Returns stepsize decreasing step-size
 */
double Solver::get_default_stepsize(int iter) {
    return 1.0/std::sqrt(iter + 1.0);
}

/*
 * Solves the problem instance with the
 * mean projection algorithm
 */
void Solver::solve_mean_projection() {
    unsigned int iteration{0};
    double current_merit{0.0};
    double previous_merit{0.0};
    std::vector<OdPair>* ods{net.get_od_pairs()};
    net.spt_search();
    net.update_flows(net.get_demand_vector());
    while (iteration <= MAX_ITER) {
        net.update_costs();
        net.spt_search();
        mean_set_active_paths(ods);
        std::valarray<double> mean_costs{mean_get_active_mean_costs(ods)};
        // Get merit function for convergence check
        if (CONV_TEST == ConvergenceTest::Gap) {
            current_merit = net.merit();
            previous_merit = current_merit;
        }
    }
}
APPENDIX A. ANNEX: SOURCE CODE

```cpp
current_merit = get_relative_gap_criterion(ods);
}
else if (CONV_TEST == ConvergenceTest::Var) {
    current_merit = get_variance_criterion(mean_costs, mean_get_active_std_devs(mean_costs, ods));
}
else if (CONV_TEST == ConvergenceTest::Max) {
    current_merit = get_max_flow_criterion(ods);
} else {
    PRINT("ERROR solve mean projection():\nInvalid convergence test: ") << CONV_TEST);
    throw std::invalid_argument("ERROR solve_mean_projection():\nInvalid convergence test");
}
if (current_merit > EPSILON) {
    // Update flows
    net.update_flows(mean_projection(STEP_ALPHA, mean_get_gamma_vector(mean_costs, ods), ods));
    mean_clear_active_paths();
} else {
    // Check if unused paths are convenient
    // returns true if a path is found
    if (!mean_check_unused_paths(mean_costs, ods)) {
        break;
    }
}
previous_merit = current_merit;
iteration++;
}

/*
* Computes the threshold for the step size
* in the double projection algorithm
*/
double Solver::double_get_step_threshold(const std::valarray<double>& projected_flows, double step_beta) {
    // Get current flow and cost vectors
    std::valarray<double> current_flows{net.get_flow_vector()};
    std::valarray<double> current_costs{net.get_cost_vector()};
    // Computes projected costs
    net.update_flows(projected_flows);
    net.update_costs();
    std::valarray<double> projected_costs{net.get_cost_vector()};
    // Restore flows and costs on the network
    net.update_flows(current_flows);
    net.update_costs();
    // Compute the threshold and return it
    return step_beta * modulus(current_flows - projected_flows) / modulus(current_costs - projected_costs);
}

/*
* Computes the fast projection on the
* od flow space for the double
* projection algorithm
*/
```
std::valarray<double> Solver::double_fast_projection(const std::valarray<double>& to_project, double demand)
{
    // Compute projection on one od pair
    std::valarray<double> projected(to_project.size());
    // Initialize projection
    projected = to_project + (demand - to_project.sum()) / to_project.size();
    // Check feasibility of the projected point
    while (!is_nonnegative(projected)) {
        // Adjust projection to maintain feasibility
        std::vector<unsigned int> positives;
        for (unsigned int i = 0; i < projected.size(); i++) {
            if (projected[i] > 0) positives.push_back(i);
            else {
                projected[i] = 0;
            }
        }
        double pos_sum{0.0};
        for (auto pos_ind : positives) {
            pos_sum += projected[pos_ind];
        }
        for (auto pos_ind : positives) {
            projected[pos_ind] = projected[pos_ind] + (demand - pos_sum) / positives.size();
        }
        positives.clear();
    }
    // Return projected vector
    return projected;
}

void APPENDIX A. ANNEX: SOURCE CODE
APPENDIX A. ANNEX: SOURCE CODE

for (unsigned int i=0; i < od_projected.size(); i++) {
    projected[od_offset + i] = od_projected[i];
}  
od_offset += od_projected.size();  
// Return projected flow vector  
return projected;

/*  
* Solves the problem instance using the  
* double projection algorithm  
*/
void Solver::solve_double_projection()
{
    net.spt_search();
    net.update_flows(net.get_demand_vector());
    double currentMerit{0.0};
    double previousMerit{0.0};
    unsigned int iteration{0};
    std::vector<OdPair>∗ ods{net.get_od_pairs()};
    while(iteration <= MAX_ITER ) {
        net.update_costs();
        net.spt_search();
        // Get merit function for convergence check
        if (CONV_TEST == ConvergenceTest::Gap) {
            currentMerit = get_relative_gap_criterion(ods);
        }  
        else if (CONV_TEST == ConvergenceTest::Var) {
            std::valarray<double> meanCosts{get_mean_costs(ods)};
            currentMerit = get_variance_criterion(meanCosts, get_std_devs(meanCosts, ods));
        }  
        else if (CONV_TEST == ConvergenceTest::Max) {
            currentMerit = get_max_flow_criterion(ods);
        }  
        else {
            PRINT("ERROR solve_double_projection():\nInvalid convergence test:" << CONV_TEST);
            throw std::invalid_argument("ERROR solve_double_projection():\nInvalid convergence test");
        }
        if (currentMerit < EPSILON) {
            break;
        }
        // Compute projection
        double stepAlpha(STEP_ALPHA);
        double stepThreshold{0.0};
        std::valarray<double>∗ projected;
        while(true) {
            projected = double_projection(stepAlpha, ods);
            stepThreshold = double_get_step_threshold(projected, STEP_BETA);
            if (stepAlpha <= stepThreshold) break;
            stepAlpha = min(STEP_XI*stepAlpha, step_threshold);
        }
        // Set up flows and costs to get the correct assignments
    }
```cpp
auto current_flows = net.get_flow_vector();
net.update_flows(projected);
net.update_costs();
net.update_flows(current_flows);
auto new_flows = double_projection(step_alpha, ods);
net.update_flows(new_flows);
step_alpha = min(STEP_ALPHAA, step_threshold);
previous_merit = current_merit;
iteration++;
}
}

/*================================================================*/
/* Solves the problem                                              */
/* using the configured                                            */
/* algorithm                                                       */
/*================================================================*/
void Solver::solve() {
  switch (ATA_ALG) {
    case AtaAlgorithm::Gp:
      solve_gradient_projection();
      break;
    case AtaAlgorithm::Mp:
      solve_mean_projection();
      break;
    case AtaAlgorithm::Dp:
      solve_double_projection();
      break;
    default:
      PRINT("ERROR Solver::solve():invalid_projection_algorithm");
      throw std::invalid_argument("ERROR Solver::solve():invalid_projection_algorithm");
      break;
  }
}
A.3 Network.h

/*
 * Network.h
 *
 * Author: sivam pasupathipillai
 * Email: sivam.pasu@gmail.com
 */

#ifndef NETWORK_H
#define NETWORK_H

#include <string>
#include <map>
#include <vector>
#include <queue>

#include "Vdf.h"
#include "Link.h"
#include "Node.h"
#include "OdPair.h"
#include "Path.h"

/* Enums defining the valid algorithms for ata and spt problems. */
enum class SptAlgorithm {Djk, BeF, DeP, Slf};

/* Wrapper for parameters read from input files */
struct Parameters {
    double ALPHA;
    double BETA;
    double THETA;
    double B;
    double MOD_TIME;
    double MAX_SPEED;
    SptAlgorithm SPT_ALG;
    double PRECISION;
    // Constructor
    Parameters();
};

/* Labeled node used by spt algorithms */
struct LabelNode {
    int node_id;
    int link_id;
}
double label;
bool is_in;
bool was_in;

// Constructor
LabelNode(const int id);
}

/*! * Represents the network */
class Network {
private:
    // Spt algorithm to use
    SptAlgorithm SPT_ALGORITHM;
    // Numerical precision
double PRECISION;
    // Volume–delay function object
    Vdf vdf;
    // Map original node ids with assigned ids
    std::map<int, int> node_id_map;
    // Contains network nodes
    std::vector<Node> nodes;
    // Contains network links
    std::vector<Link> links;
    // Contains network o–d pairs
    std::vector<OdPair> od_pairs;
    // Origin of the most recently computed spt
    int cached_origin;
    // Most recently computed spt
    std::vector<LabelNode> cached_spt;

    // Compute spt with Bellman–Ford
    void compute_spt_bellman_ford(int origin);
    // Compute spt with D’Esopo–Pape
    void compute_spt_desopo_pape(int origin);
    // Compute spt with Dijkstra
    void compute_spt_dijkstra(int origin);
    // Compute spt with SLF–LLL
    void compute_spt_slflll(int origin);
    // Extract shortest path from cached spt
    Path extract_shortest_path(int origin, int dest);
    // Return the cached spt to its initial state
    void reset_cached_spt();
    // Return the index of the min label candidate
    int get_min_label_index(const std::vector<int>& candidates);
    // Return the mean label among candidates
    double get_mean_label(const std::deque<int>& candidates);
    // Return the priority factor of a non–priority link
    double compute_priority_factor(int link_id);
    // Return additive cost of path
    double compute_path_cost(const std::vector<int>& path_links);
    // Return shortest path from origin to dest
}
Path get_shortest_path(int origin, int dest);

public:
  // Constructors
  Network(); // Default constructor
  Network(const std::string& net_filename, const std::string& mat_filename, const Parameters& params);

  // Backport for testing
  void set_spt_algorithm(SptAlgorithm spt) { SPT_ALGORITHM = spt; }
  // Return spt algorithm
  SptAlgorithm get_spt_algorithm() const { return SPT_ALGORITHM; }
  // Return denominator of stepsize for GP
  double get_derivatives_sum(const std::vector<int>& link_ids);
  // Return current number of paths
  unsigned int get_path_dimension();
  // Return the od corresponding to the index
  OdPair& get_od_pair(int od_index);
  // Return the od corresponding to the index
  std::vector<OdPair> get_od_pairs() { return &od_pairs; }
  // Update the flow vector with the new flows
  void update_flows(const std::valarray<double>& new_flows);
  // Update path costs corresponding to new path flows
  void update_costs();
  // Search for spt for each od pair and add it to active set
  void spt_search();
  // Return demand vector
  std::valarray<double> get_demand_vector();
  // Return path flow vector
  std::valarray<double> get_flow_vector();
  // Return path cost vector
  std::valarray<double> get_cost_vector();
};

#endif /* NETWORK_H */
A.4 Network.cpp

/*
 * Network.cpp
 *
 * Author: sivam pasupathipillai
 * Email: sivam.pasu@gmail.com
 */

#include <limits>
#include <algorithm>
#include <fstream>
#include <sstream>
#include "Network.h"
#include "Util.h"

/*
 * Computes the shortest path tree from o, if not
 * already computed, and returns the shortest path
 * to d
 */
Path Network::get_shortest_path(int o, int d)
{
    // If I do not have a the correct cached spt
    if (o != cached_origin) {
        // Recompute the spt using the given algorithm
        switch (SPT_ALGORITHM) {
            case SptAlgorithm::BeF:
                compute_spt_bellman_ford(o);
                break;
            case SptAlgorithm::DeP:
                compute_spt_desopo_pape(o);
                break;
            case SptAlgorithm::Djk:
                compute_spt_dijkstra(o);
                break;
            case SptAlgorithm::Slf:
                compute_spt_slflll(o);
                break;
            default:
                PRINT("ERROR get_shortest_path(): invalid shortest path algorithm");
                throw std::invalid_argument("ERROR get_shortest_path(): invalid shortest path algorithm");
                break;
        }
    }
    // Extract shortest path from cached spt
    return extract_shortest_path(o, d);
}

/*
 * Bellman–Ford algorithm for the shortest path tree
 */
void Network::compute_spt_bellman_ford(int origin)
{
    // Initialize tree
    reset_cached_spt();
    cached_origin = origin;
    // Select the origin labeled node
    LabelNode* current = &cached_spt[origin];
    // Initialize the empty candidate queue
    // (LIFO structure)
    std::queue<int> candidates;
    // Put origin in candidates vector
    current->is_in = true;
    current->label = 0;
    candidates.push(current->node_id);
    // While candidates is not empty
    while(!candidates.empty()) {
        // Extract head from candidates
        current = &cached_spt[candidates.front()];
        candidates.pop();
        current->is_in = false;
        // For each outgoing link
        for (auto& link_id : nodes.at(current->node_id).get_out_links()) {
            // Get outgoing link
            Link* out_link = &links.at(link_id);
            // Retrieve neighbour labeled node
            LabelNode* neighbour = &cached_spt[out_link->get_to()];
            // Variable to compare costs
            double link_cost(0);
            // Get link cost
            if (out_link->is_prio()) {
                link_cost = out_link->get_cost();
            } else {
                link_cost = out_link->get_cost(compute_priority_factor(link_id));
            }
            // If neighbour's label doesn’t verify Bellman’s equation
            if (round_grt(neighbour->label, (current->label + link_cost), PRECISION)) {
                // Update neighbour’s label and predecessor
                neighbour->label = current->label + link_cost;
                neighbour->link_id = link_id;
                // If neighbour is not already in candidates
                if (!neighbour->is_in) {
                    // Put neighbour in candidates
                    neighbour->is_in = true;
                    candidates.push(neighbour->node_id);
                }
            }
        }
    }
}
* D’Esopo–Pape variation of the Bellman–Ford algorithm
* for the shortest path tree
*/

```cpp
void Network::compute_spt_desopo_pape (int origin)
{
    // Initialize tree
    reset_cached_spt();
    cached_origin = origin;
    // Select the origin
    LabelNode* current = &cached_spt[origin];
    std::deque<int> candidates;
    // Put origin in candidates vector
    current->is_in = true;
    current->was_in = true;
    current->label = 0;
    candidates.push_back(current->node_id);
    // While candidates is not empty
    while (!candidates.empty()) {
        // Extract head from candidates
        current = &cached_spt[candidates.front()];
        // Remove first node from candidates
        candidates.pop_front();
        current->is_in = false;
        // For each outgoing link
        for (auto& link_id : nodes.at(current->node_id).get_out_links()) {
            // Get outgoing link
            Link* out_link = &links.at(link_id);
            // Retrieve neighbour node
            LabelNode* neighbour = &cached_spt[out_link->get_to()];
            double link_cost = 0;
            // Get link cost
            if (out_link->is_prio()) {
                link_cost = out_link->get_cost();
            } else {
                link_cost = out_link->get_cost(compute_priority_factor(link_id));
            }
            // If neighbour’s label doesn’t verify Bellman’s equation
            if (round_grt(neighbour->label, (current->label + link_cost), PRECISION)) {
                // Update neighbour’s label and predecessor
                neighbour->label = current->label + link_cost;
                neighbour->link_id = link_id;
                // If neighbour is not already in candidates
                if (!neighbour->is_in) {
                    // Put neighbour in candidates D’Esopo–Pape
                    if (neighbour->was_in) {
                        candidates.push_front(neighbour->node_id);
                    } else {
                        candidates.push_back(neighbour->node_id);
                    }
                    neighbour->is_in = true;
                    neighbour->was_in = true;
                } else {
                    // Update neighbour’s label and predecessor
                    neighbour->label = current->label + link_cost;
                    neighbour->link_id = link_id;
                    // If neighbour is not already in candidates
                    if (!neighbour->is_in) {
                        // Put neighbour in candidates D’Esopo–Pape
                        if (neighbour->was_in) {
                            candidates.push_front(neighbour->node_id);
                        } else {
                            candidates.push_back(neighbour->node_id);
                        }
                        neighbour->is_in = true;
                        neighbour->was_in = true;
                    } else {
```
Returns the index of the candidate with minimum label

```cpp
int Network::get_min_label_index(const std::vector<int> & candidates) {
  double min = POS_INF;
  int min_index = -1;
  // For each candidate
  for (auto i : candidates) {
    // If the label is smaller than the min
    if (cached_spt[i].label < min) {
      // Record min index
      min_index = i;
      // Update min
      min = cached_spt[i].label;
    }
  }
  return min_index;
}
```

Dijkstra’s algorithm for the shortest path tree

```cpp
void Network::compute_spt_dijkstra(int origin) {
  // Initialize tree
  reset_cached_spt();
  cached_origin = origin;
  // Select the origin
  LabelNode* current = &cached_spt[origin];
  std::vector<int> candidates;
  // Put origin in candidates vector
  current->is_in = true;
  current->label = 0;
  candidates.push_back(current->node_id);
  // While candidates is not empty
  while (!candidates.empty()) {
    // Find min label node in candidates
    int min_index = get_min_label_index(candidates));
    // Get min label node from spt
    current = &cached_spt[min_index];
    // Remove min label node from candidates
    candidates.erase(std::find(candidates.begin(), candidates.end(), min_index));
    current->is_in = false;
    // For each outgoing link
    for (auto& link_id : nodes.at(current->node_id).get_out_links()) {
```
/ Get outgoing link
Link* out_link = &links.at(link_id);
// Retrieve neighbour node
LabelNode* neighbour = &cached_spt[out_link->get_to()];
double link_cost(0);
// Get link cost
if (out_link->is_prio()) {
    link_cost = out_link->get_cost();
} else {
    link_cost = out_link->get_cost(compute_priority_factor(link_id));
}
// If neighbour’s label doesn’t verify Bellman’s equation
if (round_grt(neighbour->label, (current->label + link_cost), PRECISION)) {
    // Update neighbour’s label and predecessor
    neighbour->label = current->label + link_cost;
    neighbour->link_id = link_id;
    // If neighbour is not already in candidates
    if (!neighbour->is_in) {
        // Put neighbour in candidates
        candidates.push_back(neighbour->node_id);
        neighbour->is_in = true;
    }
}
/*/ Returns the mean label among candidates */
double Network::get_mean_label(const std::deque<int>& candidates)
{
    double sum_label{0.0};
    for (auto i : candidates) {
        sum_label += cached_spt[i].label;
    }
    return sum_label / candidates.size();
}

/*/ Computes the spt using SLF-LLL heuristics (see doc for details) */
void Network::compute_spt_slflll(int origin)
{
    // Initialize tree
    reset Cached_spt();
    cached_origin = origin;
    // Select the origin labeled node
    LabelNode* current = &cached_spt[origin];
    // Initialize the empty candidate queue
    // (LIFO structure)
    std::deque<int> candidates;
// Put origin in candidates vector
current->is_in = true;
current->label = 0;
candidates.push_back(current->node_id);
// While candidates is not empty
while(!candidates.empty()) {
    double mean_label = get_mean_label(candidates); // Mean label value
    // LLL heuristic
    while (true) {
        current = &cached_spt[candidates.front()];
        if (round_grt(current->label, mean_label, PRECISION)) {
            candidates.pop_front();
            candidates.push_back(current->node_id);
        } else {
            break;
        }
    }
    // Remove head from candidates
    candidates.pop_front();
current->is_in = false;
    // For each outgoing link
    for (auto& link_id : nodes.at(current->node_id).get_out_links()) {
        // Get outgoing link
        Link* out_link = &links.at(link_id);
        // Retrieve neighbour labeled node
        LabelNode* neighbour = &cached_spt[out_link->get_to()];
        // Variable to compare costs
        double link_cost = 0;
        // Get link cost
        if (out_link->is_prio()) {
            link_cost = out_link->get_cost();
        } else {
            link_cost = out_link->get_cost(compute_priority_factor(link_id));
        }
        // If neighbour's label doesn't verify Bellman's equation
        if (round_grt(neighbour->label, (current->label + link_cost), PRECISION)) {
            // Update neighbour's label and predecessor
            neighbour->label = current->label + link_cost;
            neighbour->link_id = link_id;
            // If neighbour is not already in candidates
            if (!neighbour->is_in) {
                // Put neighbour in candidates using SLF
                if (candidates.empty()) {
                    candidates.push_back(neighbour->node_id);
                } else {
                    double top_label = cached_spt[candidates.front()].label;
                    // SLF insertion
                    if (neighbour->label > top_label) {
                        candidates.push_back(neighbour->node_id);
                    }
                }   
            } else {
                double top_label = cached_spt[candidates.front()].label;
                // SLF insertion
                if (neighbour->label > top_label) {
                    candidates.push_back(neighbour->node_id);
                }
            }
        }
    }
}

else {
    candidates.push_front(neighbour->node_id);
}
}
neighbour->is_in = true;
}
}
/
* Extracts a path from origin to destination from the cached SPT *
*/
Path Network::extract_shortest_path(int origin_id, int destination_id) {
    // Pointer to destination node
    LabelNode* path_walker = &cached_spt[destination_id];
    // If destination link is not set the algorithm
    // has not reached the destination
    if (path_walker->link_id == (-1)) {
        PRINT("ERROR extract_shortest_path(): cannot reach destination","+std::to_string(destination_id)+","+
        from_origin","+std::to_string(origin_id));
        throw std::invalid_argument("ERROR extract_shortest_path(): unreachable destination.");
    }
    // Pointer to link used to reach destination in spt
    Link* current_link = &links[path_walker->link_id];
    // Links of path to be returned
    std::vector<int> path_links;
    // Traverse links backward until you reach the first link in the path.
    // While we have not reached the origin
    while (current_link->get_from() != origin_id) {
        // Add current link to path
        path_links.push_back(current_link->get_id());
        // Move walker to previous node in path
        path_walker = &cached_spt[current_link->get_from()];
        // Selected previous link
        current_link = &links[path_walker->link_id];
    }
    // Finally, add first path link (not selected in the while)
    path_links.push_back(current_link->get_id());
    // Sort paths
    std::sort(path_links.begin(), path_links.end());
    // Create new path from links and return it
    return Path(path_links, compute_path_cost(path_links));
}
/
* Returns the path cost as the sum of
* link costs.
*/
double Network::compute_path_cost(const std::vector<int>& path_links) {
    double path_cost{0.0};
For each link
for (auto link_id : path_links) {
    // Pointer to the link
    Link* link = &links[link_id];

    // Add link cost to path cost
    if (link->is_prio()) {
        path_cost += link->get_cost();
    } else {
        path_cost += link->get_cost(compute_priority_factor(link_id));
    }
}
return path_cost;

/*
 * Returns the priority factor for a non priority link
*/
double Network::compute_priority_factor(int np_link_id) {
    // See the equation in the document
    double pri_factor{0.0};
    double k{0.0};
    Link* np_link = &links[np_link_id];

    // Compute the sum of the relative capacities
    for(auto link_id : nodes[np_link->get_to()].get_in_links()) {
        auto link = &links[link_id];
        if (link->is_prio()) {
            k = np_link->get_capacity()/link->get_capacity();
            pri_factor += k*link->get_flow();
        }
    }
    // Add link flow
    pri_factor += np_link->get_flow();
    // Normalize with modeling time
    pri_factor /= (vdf.get_modeling_time()*np_link->get_capacity());
    // Check consistency
    if (pri_factor < 0) {
        LOG(("ERROR:compute_priority_factor():\nNegative priority factor."));
        throw std::invalid_argument("ERROR:compute_priority_factor():\nNegative priority factor.");
    }
    return pri_factor;
}

/*
 * Returns the number of currently discovered paths
*/
unsigned int Network::get_path_dimension() {
    int dimension{0};
    for(const auto& od : od_pairs) {
        dimension += od.get_paths_size();
    }
}
APPENDIX A. ANNEX: SOURCE CODE

```c++
return dimension;
}

/* Computes the shortest path for each origin destination pair using the current link costs. If the path is new it adds it to the corresponding od pair */
void Network::spt_search()
{
    for (auto& od : od_pairs)
    {
        Path shortest_path = get_shortest_path(od.get_origin(), od.get_destination());
        od.add_path(shortest_path);
    }
}

/* Updates the path flows with the new flows. It assumes that paths are stored and retrieved in the same order. It automatically updates link flows to maintain consistency. */
void Network::update_flows(const std::valarray<double>& new_flows)
{
    std::valarray<double> rounded_flows(new_flows.size());
    for (unsigned int i = 0; i < rounded_flows.size(); i++)
    {
        rounded_flows[i] = round_zero(new_flows[i], PRECISION);
    }
    // Check that the vector size is correct
    if (get_path_dimension() != rounded_flows.size())
    {
        PRINT("ERROR update_flows(): Path flow vector dimension does not match path number.");
        throw std::invalid_argument("ERROR update_flows(): Vector sizes do not match");
    }
    // Update flows consuming the flow vector
    unsigned int index(0);
    for (auto& od : od_pairs)
    {
        index = od.update_flows(rounded_flows, index);
    }
    // Check again consistency of index
    if (index != rounded_flows.size())
    {
        PRINT("ERROR update_flows(): Recursive index and vector size do not match path number.");
        throw std::runtime_error("ERROR update_flows(): Recursive index and vector size do not match");
    }
    // Compute link flow vector
    std::vector<double> link_flows(links.size(), 0.0);
    for (auto& od : od_pairs)
    {
        for (const auto& path_ptr : od.get_paths())
        {
            for (auto link_id : path_ptr->get_path_links())
            {
                link_flows[link_id] += path_ptr->get_flow();
            }
        }
    }
    // Assign link flow to links
```

for (auto& link : links) {
    link.set_flow(link_flows[link.get_id()]);
}

/*
 * Returns a vector containing the demands for each
 * od pair ordered.
 */
std::valarray<double> Network::get_demand_vector() {
    std::valarray<double> dem_vec(od_pairs.size());
    for (unsigned int i = 0; i < dem_vec.size(); i++) {
        dem_vec[i] = od_pairs[i].get_demand();
    }
    return dem_vec;
}

/*
 * Updates path costs considering current flows.
 */
void Network::update_costs() {
    for (auto& od : od_pairs) {
        for (auto& path_ptr : od.get_paths()) {
            path_ptr->set_cost(round_zero(compute_path_cost(path_ptr->get_path_links()), PRECISION));
        }
    }
}

/*
 * Returns the sum of derivatives of
 * links at current flow.
 */
double Network::get_derivatives_sum(const std::vector<int>& link_ids) {
    double sum_der{0.0};
    // For each link
    for (auto id : link_ids) {
        auto link = &links[id];
        // Add the derivative to the sum
        if (link->is_prio()) {
            sum_der += links[id].get_derivative();
        } else {
            sum_der += links[id].get_derivative(compute_priority_factor(id));
        }
    }
    // Return computed sum
    return sum_der;
}
/∗
∗ Returns the aggregated path flow vector
∗/
std::valarray<double> Network::get_flow_vector()
{
    // Aggregated flow vector
    std::valarray<double> flow_vector(get_path_dimension());
    // Number of paths already considered
    double od_offset{0.0};
    // For each o−d pair
    for (const auto& od : od_pairs) {
        // Get o−d flow vector
        auto od_flows = od.get_flows();
        // For each path of o−d
        for (unsigned int i = 0 ; i < od_flows.size(); i++) {
            // Add flow to aggregated flow vector
            flow_vector[od_offset+i] = od_flows[i];
        }
        // Update o−d offset
        od_offset += od_flows.size();
    }
    return flow_vector;
}

/*/ 
* Returns the aggregated path cost vector
*/
std::valarray<double> Network::get_cost_vector()
{
    std::valarray<double> cost_vector(get_path_dimension()); // Aggregated cost vector
    double od_offset{0.0}; // Index to keep track of considered paths
    // For each o−d pair
    for (auto& od : od_pairs) {
        // Get o−d cost vector
        auto od_costs = od.get_costs();
        // For each o−d path
        for (unsigned int i = 0 ; i < od_costs.size(); i++) {
            // Add path cost to aggregated cost vector
            cost_vector[od_offset+i] = od_costs[i];
        }
        // Update index
        od_offset += od_costs.size();
    }
    return cost_vector;
}
List of Figures

1.1 Typical volume-delay function ................................................. 6
1.2 Example network .............................................................. 10

5.1 Shortest path analysis - Sioux-Falls ........................................ 55
5.2 Shortest path analysis - Terrassa ........................................... 55
5.3 Shortest path analysis - Winnipeg .......................................... 56
5.4 Step-size analysis for GP algorithm - Terrassa .............................. 58
5.5 Step-size analysis for GP algorithm - Sioux-Falls ......................... 58
5.6 Step-size analysis for GP algorithm - Winnipeg ......................... 59
5.7 Step-size analysis for MP algorithm - Terrassa .............................. 60
5.8 Step-size analysis for MP algorithm - Sioux-Falls ......................... 60
5.9 Step-size analysis for MP algorithm - Winnipeg ............................ 61
5.10 Comparison on Sioux-Falls network - Gap1 ................................. 62
5.11 Comparison on Sioux-Falls network - Gap2 ................................. 62
5.12 Comparison on Sioux-Falls network - Gap3 ................................. 63
5.13 Comparison on Terrassa network - Gap1 .................................. 64
5.14 Comparison on Terrassa network - Gap2 .................................. 64
5.15 Comparison on Winnipeg network - Gap1 .................................. 65
5.16 Comparison on Winnipeg network - Gap2 .................................. 66
5.17 Comparison on Hessen network - Gap1 .................................... 67
5.18 Comparison on Hessen network - Gap2 .................................... 67
### List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Volume-delay function parameters</td>
<td>35</td>
</tr>
<tr>
<td>5.1</td>
<td>Networks characteristics</td>
<td>54</td>
</tr>
<tr>
<td>5.2</td>
<td>Numerical results for Sioux-Falls network</td>
<td>63</td>
</tr>
<tr>
<td>5.3</td>
<td>Numerical results for Terrassa network</td>
<td>65</td>
</tr>
<tr>
<td>5.4</td>
<td>Numerical results for Winnipeg network</td>
<td>66</td>
</tr>
<tr>
<td>5.5</td>
<td>Numerical results for Hessen network</td>
<td>68</td>
</tr>
</tbody>
</table>
Bibliography


