Facultat d'Informàtica de Barcelona

Asymmetric Traffic Assignment on Large-Scale Networks

Autor:
Sivam Pasupathipillai

Director:
Esteve Codina Sancho

Titulació:
Grau en Enginyeria Informàtica

Departament:
EIO

FIB - Informatics Engineering
Final degree project
21 June, 2014

Contents

Chapter 1

Introduction

1.1 Project definition

The traffic assignment problem [TA] is a recurring problem in urban transportation network analysis. The problem can be formulated as: given a directed graph and an origin-destination trip rate matrix, find a set of paths and their relative flows, such that an equilibrium situation is reached and the demand of travelers from each origin to each destination is satisfied. Each link in the graph has an associated cost function, which in urban transportation network usually models the travel time on a particular street. If this function depends only on the link's flow, the problem takes the name of symmetric, or diagonal, traffic assignment problem. Otherwise, i.e. when link costs also depends on other links' flows, the problem is called asymmetric traffic assignment [ATA]. Traffic assignment has applications in various fields including communication networks routing, urban transportation networks planning as well as economics and game theory. In our project, we have developed an algorithm for the asymmetric traffic assignment problem using a gradient projection technique. The algorithm was then tested on various medium to large scale network in order to evaluate its performance. Finally, a modified version of the algorithm was implemented and a comparative analysis of the results was executed.

1.2 Document structure

The first part of this document introduces some relevant theory about the traffic assignment problem. In section 2.3 the concept of equilibrium is defined and different examples are presented. In section 2.4 the basic formulation of the traffic assignment problem as a non linear program is presented while section 2.7 describes some extensions to the basic model. In section 2.5 an interesting urban networks paradox is described. Section 2.8 describe other relevant concepts about the problem statement and in the following sections various algorithm design schemes are explained.

The second part of this document focuses on the development of our project. In section 3.1 a detailed description of the algorithm implemented is carried on. Sections 3.2 and 3.3 analyse the algorithm's complexity and correctness. In section 3.4 an analysis of the shortest path problem and algorithms is carried on.

The last part of the document contains design considerations and implementation details of the developed software. In section 4.4 computational results are presented and in section 4.5 possible improvements are proposed. Section 4.6 contains alternative available software packages to solve the problem while section 4.7 contains an economical analysis of the project.

Finally in section 4.8 conclusions are drawn on the results obtained and on the developed project as a whole.

Chapter 2

The Traffic Assignment Problem

2.1 Notation

$G(N, A)$	Directed graph of nodes and links
N	Set of nodes
A	Set of links
W	Set of origin-destination pairs
Γ	Set of paths, $\otimes_{w \in W} \Gamma_w$
Γ_{pq}	Set of paths from p to q for $(p, q) \in W$
d_{pq}	Trip rate from p to q for $(p, q) \in W$
v_a	Link flow on $a \in A$
t_a	Link cost of $a \in A$
h_γ	Path flow on $\gamma \in \Gamma$
c_γ	Path cost of $\gamma \in \Gamma$
$\mathbf{v}$	Vector of link flows
$\mathbf{t}$	Vector of link costs
$\mathbf{h}$	Vector of path flows
$\mathbf{c}$	Vector of path costs

2.2 Equilibrium description

When modeling complex systems such as traffic patterns in an urban transportation network, assumptions must be made in order to reduce the model complexity and facilitate analysis. In the case of traveler's behaviour, one might assume that each traveler on the network will try to minimize his personal travel time.

Considering a single origin-destination pair, this assumption implies that the system will tend to an equilibrium situation in which every used path from the origin to the destination will have the same travel time, while all the other paths remain unused. This conclusion stems from the fact that if a traveler is traveling on a path and there is another path with a lower

travel time, he is encouraged to change his route and to take the better path, thus increasing that path's travel time.

This tendency will naturally stop when the following statement is verified:

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

Notice that the equilibrium statement also implies that the average travel time from an origin to a destination is equal to the minimum travel time. The following paragraph introduce the notion of equilibrium and how it can be formalized mathematically.

2.3 Equilibrium definition

The equilibrium conditions described in the previous paragraph are usually attributed to J.G. Wardrop and are therefore known as the Wardrop equilibrium conditions. Let $c_{pq\gamma}$ be the cost of the simple path γ between origin p and destination q and let m_{pq} be the cost of the shortest path between p and q, then Wardrop user equilibrium conditions can be stated as

$$\begin{aligned}
h_{pq\gamma}(c_{pq\gamma} - m_{pq}) &= 0 && \forall \gamma \in \Gamma_{(p,q)}, \forall (p,q) \in W && (2.1)\\
c_{pq\gamma} - m_{pq} &\geq 0 && \forall \gamma \in \Gamma_{(p,q)}, \forall (p,q) \in W && (2.2)\\
\sum\nolimits_{\gamma \in \Gamma_{pq}} h_{pq\gamma} &= d_{pq} && \forall (p,q) \in W && (2.3)\\
h_{pq\gamma} &\geq 0 && \forall \gamma \in \Gamma_{(p,q)}, \forall (p,q) \in W && (2.4)\\
m_{pq} &\geq 0 && \forall (p,q) \in W && (2.5)
\end{aligned}$$

where (2.3) states that the sum of the path flows must be equal to the trip rate from p to q while (2.4) and (2.5) state the nonnegativity of the path flows and corresponding costs thus ensuring the physical legitimacy of the solution.

A vector of path flows that satisfies the Wardrop conditions for a specific network, assuming that travelers behave the way we have described, is a solution for the traffic assignment problem on that network. The equilibrium situation described is usually called user equilibrium in contrast with the concept of system equilibrium.

2.3.1 System equilibrium

Another possible assumption about traveler's behaviour is that travelers choose their path so that the system total travel time is minimized. Total travel time can be expressed as

$$\sum\nolimits_{a \in A} t_a(v_a) v_a = \sum\nolimits_{(p,q) \in W} \sum\nolimits_{\gamma \in \Gamma_{pq}} c_{pq\gamma}(\mathbf{h}) h_{pq\gamma} \tag{2.6}$$

If we define the marginal travel cost of link a at flow v_a as the increase of the link's travel time caused by an additional (marginal) traveler

$$\bar{t}_a(v_a) := \frac{d}{dv_a}(t_a(v_a)v_a) = t_a(v_a) + t'_a(v_a)v_a \tag{2.7}$$

Then the difference between personal cost and system cost is $t'(v_a)v_a$. If travelers could perceive $\bar{t}$ instead of t, when making their route choice, the system would naturally tend to an equilibrium where total travel time is minimized.

One way to enforce this situation is by introducing taxation to make travelers aware of this cost discrepancy. The equilibrium situation just described is referred to as system equilibrium.

2.4 Beckmann's transformation

It is possible to represent the traffic assignment problem as a nonlinear mathematical program whose solution satisfies the Wardrop conditions. The program formulation is the following

$$\min T(\mathbf{f}) := \sum\nolimits_{a\in A} \int_0^{v_a} t_a(s)\, ds \tag{2.8}$$

subject to

$$\sum\nolimits_{\gamma\in\Gamma_{pq}} h_{pq\gamma} = d_{pq} \qquad \forall (p,q) \in W \tag{2.9}$$

$$h_{pq\gamma} \geq 0 \qquad \forall \gamma \in \Gamma_{pq},\ \forall (p,q) \in W \tag{2.10}$$

$$\sum\nolimits_{(p,q)\in W} \sum\nolimits_{\gamma\in\Gamma_{pq}} \delta_{pq\gamma a} h_{pq\gamma} = v_a \qquad \forall a \in A \tag{2.11}$$

where

$$\delta_{pq\gamma a} = \begin{cases} 1, & \text{path } \gamma \text{ contains link } a \\ 0, & \text{otherwise} \end{cases} \tag{2.12}$$

This mathematical program is referred to as Beckmann's transformation. The program's nonlinearity is caused by the nonlinear volume-delay function contained in (2.8). The objective function itself does not have a meaningful representation in the physical network, but, using this function, it is possible to prove the equivalency between first-order optimality conditions of this program and user equilibrium conditions, i.e. each optimal solution of the program satisfies the Wardrop user equilibrium conditions.

The function $\mathbf{t}$ must be integrable in order for the objective function to be well defined. If $\mathbf{t}$ is differentiable, the integral is well defined if and only if the Jacobian matrix $\nabla\mathbf{t}(\mathbf{f})$ is symmetric everywhere. Obviously, this is not the case for the asymmetric variant of the problem and therefore there is no direct correlation between this variant and the program defined in this paragraph.

On the other hand the equilibrium conditions define a solution in both the diagonal and asymmetric version of the problem. In the following section the so called Braess paradox is presented.

2.5 Braess's paradox

Consider the network described in figure 2.1 without the dashed link. Each link has an associated cost function which represents the travel time on the link in minutes. N represents the number

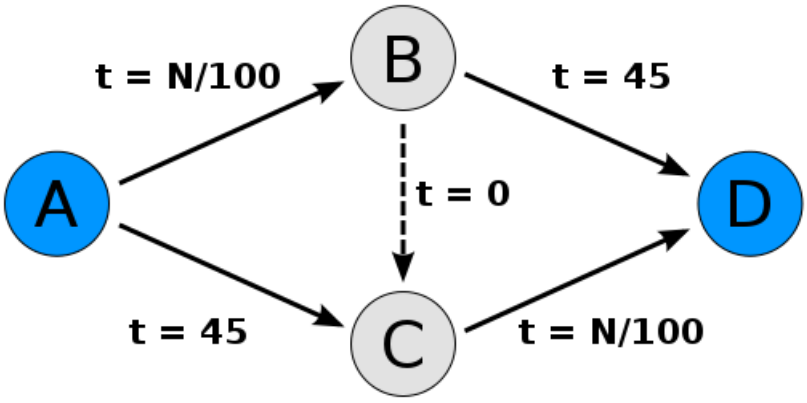

Figure 2.1: Example of Braess's paradox

of travelers that choose to travel on the link. If we assume a trip rate of 8000 travelers from A to D, then the user equilibrium flows are

$$h_{abd} = 2000 \quad c_{abd} = 65$$
$$h_{acd} = 2000 \quad c_{acd} = 65$$

and the average travel time from A to D is 65 minutes.

Now consider adding the dashed link B - C to the network. With this new configuration the new equilibrium flows are

$$h_{abd} = 4000 \quad c_{abd} = 80$$

with average travel time of 80 minutes. Interestingly enough, adding capacity to the network increase the average travel time. This is due to the fact that travelers tend to minimize personal travel times instead of total travel time.

This paradox, attributed to Dietrich Braess, has extreme relevance in urban transportation network planning as it proves that increasing the network capacity, for example building a new road, may lead to worse traffic congestion.

2.6 Volume-delay functions

Usually in a transportation network, link costs are functions of link flows, i.e. $\mathbf{t}(\mathbf{v})$. Moreover this type of network usually features some kind of congestion effect so that the flow cost dependency is not linear.

In order to represent such systems, we have to define a suitable function to model both of these aspects. The cost functions, usually called volume-delay functions, are chosen to be continuous, nonnegative and slowly increasing as the link flow increases. After a certain threshold,

which represents the link's maximum capacity, the link cost starts to increase dramatically. An example of a typical volume delay function is represented in figure 2.2.

Usual parameters for volume-delay functions are link's free-flow travel time, i.e. the link's travel time when link flow is zero, link's capacity and modeling time. The latter is the length of the period of time which is interesting to the modeler, e.g. the morning peak hour. We assume that trip rates and flows are nonnegative and real valued. Therefore, since the actual flows of travelers from on origin to a destination are naturally discrete, e.g. number of vehicles that traverse a certain road, all the results obtained in our model refers to averages trip rates and flows over the modeling time period.

In the development of our project we considered two types of volume-delay functions, one for priority links and another for non priority links. The travel time on priority links does not depend on other link's flow. Therefore if a network contained only priority links the problem would be in diagonal form. The volume-delay function used for this type of links is

$$t_a(v_a) = t_0 \left(1 + \alpha \left(\frac{v_a}{H c_a}\right)^{\beta}\right) \tag{2.13}$$

where t_0 is the free-flow travel time, H is the modeling time and c_a is the link's capacity. α and β are parameters that defines a specific volume-delay function. In the implementation these were chosen to be 0.1 and 1.5 respectively. The travel time is computed from the link's length and the maximum speed allowed on the network.

Non priority links are links whose travel time depends generally on the whole link flow vector $\mathbf{v}$. In our model non priority links depend only on the subset of flows corresponding to the links that share the same ending node with the non priority link. This models the traffic interactions at an urban intersection. The volume delay function defined for non priority links is

$$t_a(x_a(\mathbf{v})) = t_0 + \frac{1}{\theta} \ln\left(1 + e^{\theta b (x_a(\mathbf{v}) - 1)}\right) \tag{2.14}$$

where

$$x_a(\mathbf{v}) = \frac{v_a + \sum_{a' \in X(a)} k_{a'} v_{a'}}{H c_a} \tag{2.15}$$

and $X(a)$ is the set of priority links that enters the ending node of link a, $k_{a'}$ is defined as $\frac{c_a}{c_{a'}}$.

2.7 Extensions

The traffic assignment problem has been studied in many variants. Apart from the already described asymmetric case, the most important extensions have been the elastic demand variant and stochastic models for route choices.

The elastic demand variation consists on considering the trip rates between origin and destination as functions of the least travel cost. A traveler can choose where to travel depending on the perceived benefit of the trip. This variant is more easily interpreted in economical terms considering the network as a supplier of transportation services at a given cost, i.e. travel times, and

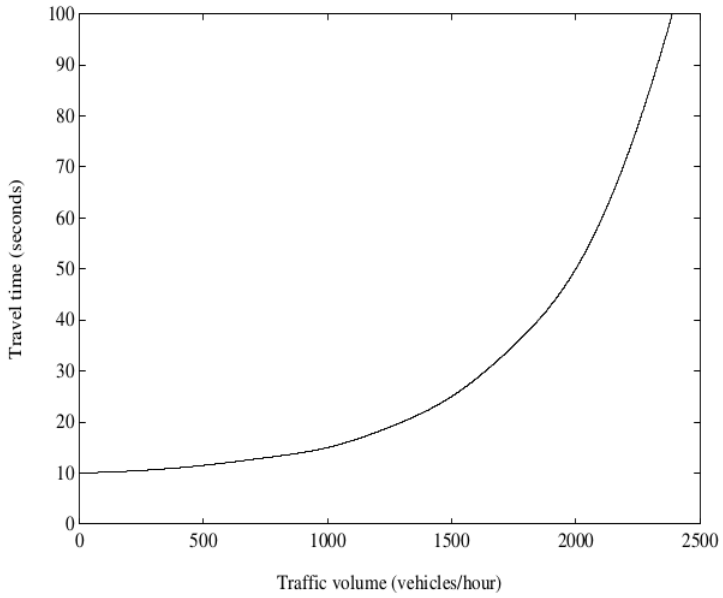

Figure 2.2: Example of volume-delay function

considering trip rates as the variable demand from travelers which consume the transportation services.

In the stochastic traffic assignment model, the assumption that travelers can perceive travel times with absolute precision when making their route choice is dropped. In this formulation travel times depend on probability distributions. In this extension the user equilibrium definition is reformulated as

> *No traveler* ***believes*** *that he can reduce his journey time by unilaterally choosing another route*

That means that all used paths have equal *perceived* costs.

2.8 Variational inequality problem

We now define a variational inequality problem which is a mathematical problem with relevant applications in the design of algorithms for the asymmetric traffic assignment problem.

DEF Let $X \subseteq \Re^n$ be a nonempty, closed and convex set, and $F : X \mapsto \Re^n$ a continuous mapping on X. The *Variational Inequality Problem* [VIP] then is to find an $\mathbf{x}^* \in X$ such that

$$F(\mathbf{x}^*)^T(\mathbf{x} - \mathbf{x}^*) \leq 0, \quad \forall \mathbf{x} \in X \tag{2.16}$$

It can be proven that, if F is continuous and $\nabla F(\mathbf{x})$ is symmetric on an open convex subset of X then [VIP] can be restated as a mathematical program with an objective function of the

same form as the one described in the Beckmann formulation. When $\nabla \mathbf{F}$ is asymmetric though it is still possible to define a merit function, usually called a gap function, which estimate the violation of [VIP] at any point $x \in X$.

DEF Let Ω be the set of solutions to [VIP]. A function $\psi : X \mapsto \Re \cup \{-\infty, +\infty\}$ is a gap function for [VIP] if
(1) ψ is restricted in sign on X, and
(2) $\psi(\mathbf{x}) = 0 \iff \mathbf{x} \in \Omega$

It can be proven that by minimizing a gap function over X, a solution of the [VIP] is obtained. In the context of traffic equilibrium it can be proven that if we minimize the gap function

$$G(\mathbf{v}) = \mathbf{t}(\mathbf{x})^T \mathbf{v} - \min_{\mathbf{y} \in V} \mathbf{t}(\mathbf{v})^T \mathbf{y} \tag{2.17}$$

where V is the space of feasible link flows, we obtain a flow pattern $\mathbf{v}^*$ which satisfies the Wardrop conditions and we consequently have a solution for traffic assignment problem. This conclusion can be drawn intuitively by analyzing the gap function structure. Equation (2.17) defines the gap function as the difference between the current total travel cost and the one obtained by assigning all the flows to the shortest paths. When the gap function is equal to zero all the flow is assigned to the shortest paths and we have an optimum solution.

2.9 Iterative method and convergence

Many optimization techniques are based on the so called iterative method. This method consists of starting from a valid solution to a problem and iteratively improve the solution by means of a mapping onto the space of solutions, i.e. an iterative algorithm.

One important property of iterative algorithms is convergence, which is the algorithm's ability the generate a sequence of solutions which converges to an optimal solution. Another important factor is the speed at which the sequence converges. It is possible to define conditions under which an iterative algorithm converges to a solution. This conditions are usually defined over the functions used to map a solution to the following in the sequence.

Unfortunately, in the case of asymmetric traffic assignment, some of the algorithms do not comply with the convergence conditions and therefore there is no theoretical evidence that the algorithm will find an optimal solution.

In practice, most of the algorithm designed for the problem seems to converge to a solution and therefore, it may be the case that the convergence conditions are too strict and could be relaxed.

2.10 Step size

The mapping from a solution to another used in an iterative algorithm usually implies two steps: first, define a descent direction for the objective or merit function and second, find an

appropriate step size so that by moving in the descent direction we reach an improved solution. This can be formulated as

$$\mathbf{x}^{k+1} = \mathbf{x} + \alpha_k(\mathbf{y}^k - \mathbf{x}^k), k = 1, 2, ... \tag{2.18}$$

where $\mathbf{y} - \mathbf{x}$ denotes a descent direction and α_k is so called step size modifier. It can be shown that if $\sum \alpha^k = \infty$ and $\sum(\alpha^2) < \infty$ the method almost surely converges to a solution.

There are many techniques in order to choose the step size. A common choice is to choose a predefined step size α_k in the form $1/k$ where k is the current iteration of the algorithm. This method is called MSA, or Method of Successive Averages. The biggest drawback of this method is that the convergence speed rapidly decreases with the number of iterations as the step size becomes smaller and smaller. Other possible sequences are

$$\alpha^k = \frac{1}{\sqrt{k+1}} \tag{2.19}$$

$$\alpha^k = \left\{ 1, \frac{1}{2}, \frac{1}{2}, \frac{1}{3}, \frac{1}{3}, \frac{1}{3}, ..., \overbrace{\frac{1}{k}, \frac{1}{k}, ..., \frac{1}{k}}^{k}, ... \right\} \tag{2.20}$$

The step size α_k can be interpreted as the weight that every intermediate solution has on the final solution. Considering this, one may assign a larger weight to later solution as they are closer to the optimal solution.

One method that takes advantage of this consideration is the MSWA, or Method of Successive Weighted Averages. In this method the step size is defined as

$$\alpha^k = \frac{k^d}{\sum_1^k i^d} \tag{2.21}$$

where d is a measure of the weight assigned to later iterations. In our second implementation we used a step size corresponding to (2.19) as it seems that it achieves good convergence speed on the networks we have tested.

2.11 Gradient projection method

The gradient projection method is a technique used to find a feasible descent direction in the solutions space of an optimization problem. The technique consists of projecting the negative gradient of the objective function onto the feasible set using the active constraints of the program, thus obtaining a new feasible point with an improved objective.

In the space of path flows, a valid projection can be defined as the difference between the cost of a path and the minimum path cost. This algorithm can be extended to the asymmetric variant of the problem.

Chapter 3

ATA algorithm

3.1 Description

This section contains a description of the gradient projection algorithm implemented in the context of the project. The algorithm was derived from the gradient projection algorithm described in [5] for the diagonal traffic assignment problem. After implementing the first version, a second version proposed by the project director Esteve Codina was developed in order to compare the different implementations.

The algorithm is a path-based gradient projection algorithm for the asymmetric traffic assignment problem. Algorithm 1 contains the pseudocode of the algorithm. It can be divided into four main procedures: shortest path tree computation, path costs update, convergence criterion testing and path flows update. These procedures are described in details in the following paragraphs.

Algorithm 1 Pseudocode for the gradient projection algorithm

```
- Find the shortest path for each origin destination pair
- Assign the trip rate on the shortest path
while True do
    - Update path costs
    - Find the shortest path for each origin destination pair
    if Convergence == True then
        return
    else
        - Update path flows
    end if
end while
```

3.1.1 Shortest path tree computation

The algorithm uses the shortest path cost in the paths flows update procedure. In order for this information to be available the shortest path tree for each origin must be computed at each

iteration of the algorithm. This tree could be different in each iteration since, even though the network topology does not change, the link costs, i.e. volume-delay functions, may vary due to the changes in path flows.

In order to solve the problem on large networks, where $|W|$ could be in the order of 10^5, a careful implementation of this procedure is necessary. We have decided to use label correcting algorithms as even though their worse case performance is inferior to label setting algorithms, e.g. Dijkstra's algorithm, they seems to achieve better performance on sparse graphs as those representing urban transportation networks.

We have implemented three different algorithms from this class and it is possible to switch between them during configuration. Section 3.4 describes the algorithms in more details.

3.1.2 Path costs update

The path costs update procedure consists of updating the path costs after the path flows have changed. This update is necessary because path costs are defined addictively as the sum of the costs of the links that the path traverses. If path flows are updated, link costs will change and therefore path costs must be updated as well.

From the performance evaluation executed at the end of the project we found out that this procedure consumes an amount of CPU time comparable to the shortest path tree procedure. In the implementation of the algorithm described in [5], the shortest path tree information was stored at each iteration and the path costs update procedure could be optimized exploiting the tree data structure. In our implementation we choose to store path information directly and could not find a beneficial optimization scheme.

3.1.3 Convergence criterion testing

The definition of a convergence criterion is necessary in order to stop the algorithm execution after a "good enough" solution has been found. In order to do that it is common to define an $\epsilon > 0$ such that the algorithm stops when

$$|\mathbf{x}^* - \mathbf{x}| \leq \epsilon \tag{3.1}$$

where $\mathbf{x}$ is the current solution and $\mathbf{x}^*$ is the optimal solution.

Usually the difference is defined by mean of a gap function over the space of solutions. In our first implementation we defined this gap function as

$$\frac{|\mathbf{c}(\mathbf{h})^T(\bar{\mathbf{h}} - \mathbf{h})|}{|\mathbf{c}(\mathbf{h})^T(\mathbf{h})|} \tag{3.2}$$

which represent the total travel time difference between current flows $\mathbf{h}$ and shortest path flows $\bar{\mathbf{h}}$. The corresponding convergence criterion is

$$\frac{|\mathbf{c}(\mathbf{h})^T(\bar{\mathbf{h}} - \mathbf{h})|}{|\mathbf{c}(\mathbf{h})^T(\mathbf{h})|} \leq \epsilon \tag{3.3}$$

3.1.4 Path flows update

At each iteration paths are updated in a descent direction in order to get closer to the optimal solution. Considering one origin-destination pair (p, q), the path flows update equation for non-shortest paths is

$$h_\gamma^{k+1} = \max\left\{0, h_\gamma^k - \alpha^k(c_\gamma - m_{pq})\right\} \quad \forall (p,q) \in W \tag{3.4}$$

where α is the step size and m_{pq} is the cost of the shortest path from p to q.

After all the non-shortest paths have been updated, the difference between the total demand d_{pq} and the demand assigned to the non-shortest paths is assigned to the shortest path, in order not to violate the problem constraints. From (3.4) we can conclude that in every iteration the flow on non-shortest path is decreased or set to zero while the flow on the shortest path is increased.

This lead to an increase in the shortest path cost and could potentially lead to the discovery of a new shortest path by the shortest path algorithm. As described in the first part of this document, the difference between the path costs and the minimum path costs define a valid descent direction. The step size can be defined in various ways as described in section 2.10. In this particular design, the step size depends on the second order derivatives of the path's volume-delay functions.

The algorithm described seems to converge on most of the networks we have tested. In some simulations though, the algorithm has an oscillating behaviour since there are two or more paths which become shortest path and receive all of the flow h_{pq} in alternative iterations.

3.2 Correctness

The correctness of the algorithm described can be derived from the discussion on equivalency between the variational inequality problem and user equilibrium conditions. In the algorithm we use a gap function of the form described in 2.8. This implies that if the iterative algorithm is convergent, it will minimize the gap function over the set of possible solution thus obtaining the optimal solution.

3.3 Complexity

Even though the algorithm balances all the origin-destination pairs conceptually 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 $(p, q) \in W$ are sorted so that pairs with the same origin are evaluted one after another, 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 simply computes a value from 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 $O(N_o O(spt)+O(N_o N_p N^{1/2})+ 2O(N_o N_p))$.

3.4 Shortest path tree problem

Given a directed and strongly connected graph $G(N, A)$ and an origin o, the shortest path tree problem 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 $G(N, A)$, $\forall n \in N$. Solving the shortest path tree problem efficiently is really important in order to find a solution for the asymmetric traffic assignment problem using the algorithm we have described in the previous paragraphs.

Many algorithms exist for the shortest path tree. All these algorithms have in common some form of labeling scheme for the node of the graph. A node's label usually represent the lower bound on the cost of the path between the origin and the node. The main distinction is between label setting and label correcting algorithms.

3.4.1 Label setting algorithms

The most famous label setting algorithm is Dijkstra's algorithm. It utilizes a data structure to maintain a list of the nodes to visit. Different implementations of this data structure result in different complexities for the algorithm.

The algorithm initializes all the labels to ∞ except the label of the origin, which is set to zero. Then it select the node which has the minimum label and set it as current node. For each neighbour of the current node the algorithm updates the neighbour's label if its value is greater than the sum of the current node label and the cost of the link between the current node and the neighbour. After all neighbours have been visited, the current node is removed from the list and the following current node,i.e. the node with the minimum label value in the list, is selected.

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, implemented for example with a heap. The main drawback of Dijkstra's algorithm is that at every iteration it must find the label with the minimum value, thus adding considerable overhead to the computation.

3.4.2 Label correcting algorithms

In label correcting algorithm it is possible to ignore the minimum label at every iteration at the cost of increasing the number of iteration. This is because an already visited node can reenter the candidates data structure. In this case the data structure used to store the candidates set is usually a queue.

The main algorithm of this family is the Bellman-Ford's algorithm. This algorithm, instead of removing a node from the candidate set in every iteration, checks if the label of a neighbour of the current node can be decreased, if that is the case, the neighbour is added at the end of the candidate set. This continues until the candidate set is empty. The worse case complexity of the Bellman-Ford's algorithm is $O(|A||N|)$.

Many extension of this simple algorithm exists. Most of them differ from the original version in the insertion policy of the nodes in the candidates set. The D'Esopo-Pape algorithm is based on the consideration that whenever a node's label is updated, there is an high chance that the node's neighbours labels will be modified as well. Therefore it is better if the original node is updated before it's neighbours. This consideration is translated in a different insertion policy which states the following

> If a node has never been in the candidates set before add it to the bottom, otherwise 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 consideration that can be made 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 before. 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 candidates set, the LLL, or Large Label Last, policy operates when extracting a node from the candidates set. 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 action is repeated until a node with a label smaller or equal to the mean value is found.

Both the SLF and the LLL strategy have worse complexities than the original formulation of the algorithm, but again they seems to work well in practice. In our project we have implemented the Bellman-Ford's algorithm as well as the D'Esopo-Pape method and a combination of the SLF and LLL policies.

3.5 Modified algorithm

After implementing the first version of the gradient projection algorithm, we implemented a second gradient projection algorithm proposed by the project director Esteve Codina Sancho. The algorithm is analysed in the following paragraphs.

Algorithm 2 Pseudocode for the modified version of the projection algorithm

```
- Find the shortest path from each origin destination pair
- Initialize the active paths set with the shortest path
- Assign the trip rate on the shortest path
while True do
    - Update path costs
    - Compute c̄_w and σ_w for each w ∈ W
    - Compute η 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
```

3.5.1 Algorithm description

The second algorithm we implemented is still a path based gradient projection algorithm for the asymmetric traffic assignment problem. One difference is that, instead of using the merit function described previously we defined the gap function

$$G(\mathbf{x}) = \sum\nolimits_{w \in W} \frac{\sigma_w}{\bar{c}_w} \tag{3.5}$$

If we consider the user equilibrium conditions where the shortest travel time is equal to the average travel time we can see that (3.5) defines a valid gap function and by minimizing it on the solutions space we obtain an optimal solution for the problem. The pseudocode for the algorithm is showed in Algorithm 2. We define the set of active paths A_w^+ for each origin-destination pair which contains the path with positive flow and possibly the shortest path which have no flow assigned yet. All the following steps are performed for each origin-destination pair.

We initialize the active set of the pair with the shortest path on an empty network, i.e. a network without any flow assigned on its links. We perform an all-or-nothing assignment of the shortest path and update the path cost. We find the new shortest path and add it to the active paths set. Next we compute mean and standard deviation of the costs over the active set and define η_γ as

$$\eta_\gamma = \begin{cases} c_\gamma - \bar{c}_w, & \gamma \in A_w^+ \\ 0, & \gamma \notin A_w^+ \end{cases} \tag{3.6}$$

Then we check for the convergence criterion. If the criterion is false we update the path flows with

$$h_\gamma^{k+1} = h_\gamma^k - \beta_w \eta_\gamma \tag{3.7}$$

where β is an appropriate step size limited by

$$\beta_w \leq \min \left\{ \frac{h_\gamma}{\eta_\gamma} \mid \eta_\gamma > 0, \gamma \in A_w^+ \right\} \tag{3.8}$$

Finally we update the set of active paths to include all the paths with nonnegative flow. If the convergence criterion is false, we define Z as $c_\gamma - \bar{c}_w, \forall \gamma \notin A_w^+$. Then we check if any of the inactive paths has a $Z < 0$. If we find such a path we add it to the active set and repeat, otherwise we stop.

The limitations on β_w are necessary in order to maintain the path flows nonnegativity.

3.5.2 Algorithm complexity

We can make the same considerations we made for the first algorithm implemented. One difference is that, in the second algorithm, the convergence criterion can be computed in $O(N_o)$. Therefore the complexity of this second implementation is $O(N_o O(spt) + O(N_o N_p N^{1/2}) + O(N_o N_p) + O(N_o))$.

Chapter 4

Project development

4.1 Design

Keeping in mind the defined goals of the project, the following considerations were considered during the design phase.

4.1.1 SPT algorithms

During the design phase an in-depth analysis of state-of-the-art shortest path algorithms was performed. This analysis can be found in 3.4. Apart from the algorithms already described the Threshold algorithm's family was also considered. This algorithm inserts the nodes in the candidates set in different positions depending on a threshold on the node label. Since there aren't effective schemes to choose the threshold parameters without testing on a particular network, this algorithms were not considered in the implementation phase.

4.1.2 Path storage and identification

We have already mentioned that the number of paths discovered by the algorithm can grow exponentially. Naturally this brought about concerns about which data structure to use for storing the paths.

Another concern was that paths need to be identified and retrieved efficiently in order to avoid storing multiple copy of the same path and to increase the algorithm performance. We decided to use an hash map to store the paths for each origin-destination. An hash map enables key retrieval and access in $log(n)$ where n is the size of the container. The hash value of a path is defined on the path's sorted link list. The hash function pseudocode is shown in Algorithm 3.

This function has the property that if two paths are composed of the same links, then their hash values will be equal. On the other hand if two paths are different, there is a low probability that their hash values will be equal.

We tested these assertions with a large number of similar paths, in the order of 10^6, without detecting any collision.

Algorithm 3 Path hash function

```
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
```

4.1.3 Numerical precision

Numerical precision must be considered in any computer implementation of a mathematical model. Floating point binary representation is subject to rounding and overflow errors which must be taken into account in order to yield valid results during a computation. To account for these deficiencies, we have decided to fix an arbitrary precision in all the arithmetic computation of the program so that any value less than the precision is considered zero. This value was set to 10^{-12}.

4.2 Implementation

4.2.1 Language choice

The language chosen for the implementation was C++. The version used was the latest C++ version released which is known as C++11. We choose C++ for its support for lower level programming while still providing higher level data abstractions. The whole project was developed in Linux using the Eclipse IDE and compiled with the GNU C++ compiler.

4.2.2 Path identifiers

The implementation of the hash function described in XXX is strictly dependent on the implementation of the system rand() function. This function returns an uniformly distributed integer between 0 and MAX_RAND where the latter is an implementation dependent integer value. Therefore on certain implementation of the language the algorithm could present unexpected behaviour regarding the generation of different paths with the same hash value.

4.2.3 Cached shortest path tree

As already mentioned, if the origin-destination pairs are ordered with respect to the origin a single shortest path tree can be cached and reused for all the pairs that share the same origin. To exploit this fact we implemented a shortest path solver that simply offers a getter for the shortest path, parameterized with the origin and the destination in its interface. This enables internal management of the cached shortest path tree and can save some resources at runtime.

4.2.4 Identifiers mapping

Another optimization we have implemented in our project is the direct mapping of nodes identifiers. Node identifiers can be defined to be integers on any range and therefore some kind of mapping between integer and nodes is necessary in order to maintain a consistent nodes structure. In order to avoid this inconvenient we reassigned node identifiers and link identifiers on the range $0, ..., |N|$ and $0, ..., |A|$ respectively.

In this way it is possible to use a simple array, in which every index corresponds to the node id, as a node container, thus greatly improving the runtime performance of the algorithm. The same technique was used for the origin-destination pairs.

4.3 Validation

Google Test

Unit code testing was performed using the Google C++ Test framework. Google Test 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 and finally a mock network was defined in order to test the full algorithm's execution.

Valgrind

One of the main characteristics of C++ is the absence of a default garbage collector. Designing C++ program therefore require a deep understanding of the underlying memory abstractions and can lead to unexpected memory related error such as memory leakage or segmentation faults. In order the validate the project against this type of errors a useful tool called Valgrind was used. Valgrind is a code analysis tool that should be able to detect 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.

gprof

In order to improve the algorithm's performance the GNU profiling tool, i.e. gprof, was used. This tool is a free profiling tool available for the Linux operating system. Like other profilers it runs the code and output valuable information about the estimated running time of specific methods in the code. From the analysis of the profiler's results the identifier mapping technique was implemented and the internal container for the shortest path tree procedure was changed from an hash map to a vector. Both these changes improved significantly the algorithm's performance.

4.4 Results

In this section we discuss the experimental results we obtained with the implemented algorithms. These results are obtained executing the two versions of the gradient projection algorithm on the networks of Terrassa, Winnipeg and Hessen. The characteristics of these networks are described in the following table.

Network	$\|N\|$	$\|A\|$	$\|W\|$
Terrassa	1609	3264	2215
Winnipeg	1057	2535	4345
Hessen	4660	6674	17213

Shortest path tree algorithm

We begin our analysis with the results on the shortest path algorithms. Figure 4.1 shows a bar chart resuming the performance on the three algorithms tested.

The best performance is obtained using the D'Esopo-Pape insertion method on all the networks tested for both the original version and the modified version of the algorithm. The graph shows the average iteration time across 200 iterations.

Original gradient projection algorithm

We now describe the results obtained for the original version of the gradient projection algorithm. Figure 4.2 shows the trend of the error function defined in (3.5) as the algorithm executes. We can notice that the decrease of the natural logarithm of the error function is in the order of 10^{-2} per iteration.

This results in a slow convergence speed especially on larger networks where each iteration of the algorithm can take up to 10 seconds to execute. This results is pretty poor compared to other algorithm for the traffic assignment problem. We must report that this version of the algorithm also presents an oscillating behaviour in which the error function does not steadily decrease.

Modified gradient projection algorithm

In figure 4.3 we can see the performance analysis for the modified version of the gradient projection algorithm. We can notice that this second version tends to converge more rapidly in the first iterations while slowing down when we approach the optimal solution. This is an expected result given the dependence between the step size and the distance to the optimal solution. We can notice that both versions of the algorithm have comparable running speeds but the modified version reach better solutions in a smaller amount of time.

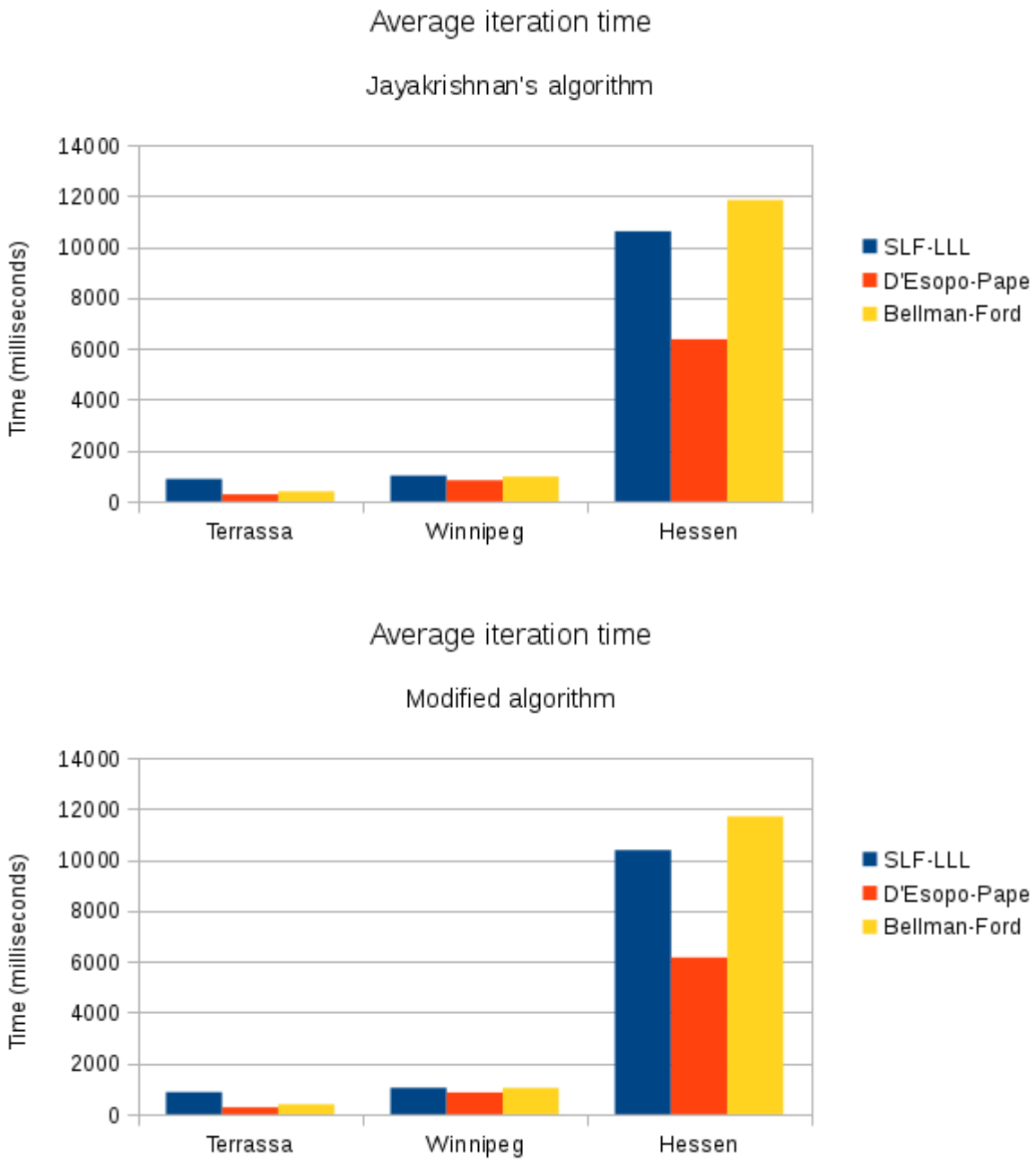

Figure 4.1: Shortest path tree analysis

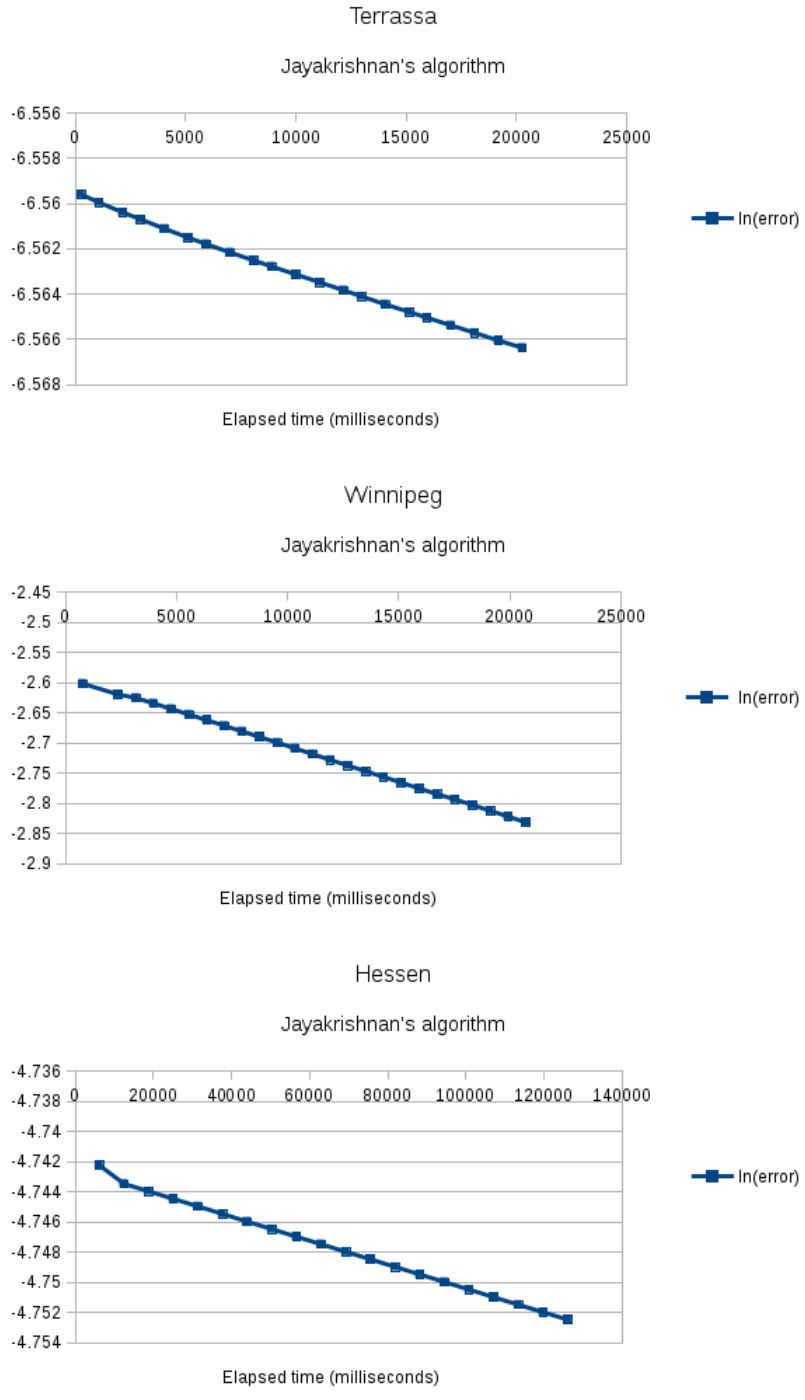

Figure 4.2: Original version

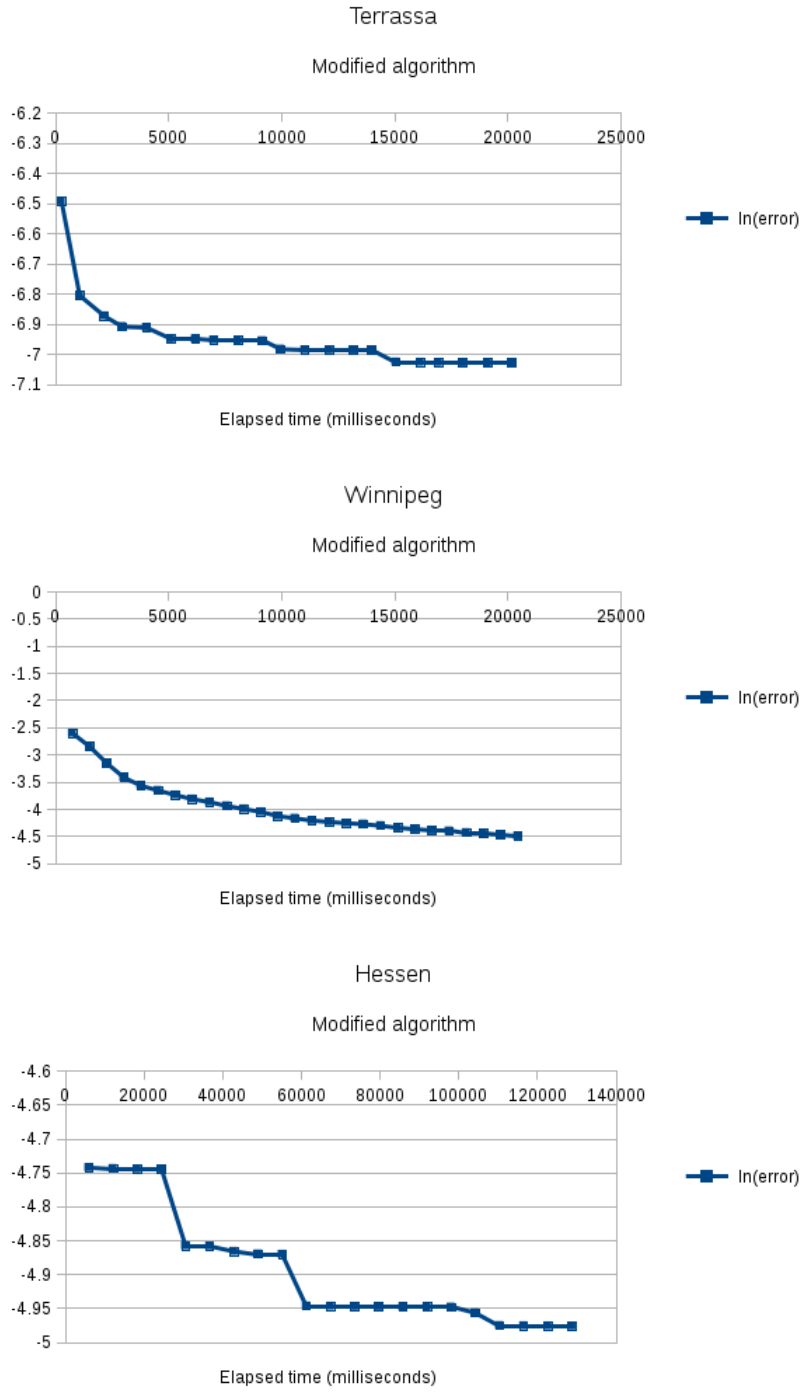

Figure 4.3: Modified version

4.5 Future improvements

In this section we suggest possible future improvements to the project developed.

4.5.1 Polimorphic volume-delay functions

The current implementation of volume-delay functions only defines two types of volume-delay functions. One for priority links and one for non priority links. This definition could be extended to include a broader set of function classes. This implementation could exploit polymorphism of C++ classes.

4.5.2 Path costs efficiency

We have already mentioned that the path costs assignment is one of the more time consuming procedures. The current implementation is fairly simple and it is possible that it could be optimized in order to obtain better running time.

4.5.3 Parallelization

The current implementation does not exploit the parallel architecture. A multi-threaded application could improve the algorithm performance. In the eventuality of this refactoring, shared data structures should be kept in mind in order to maintain the algorithm correctness

4.6 Alternatives

INRO EMME

EMME is a commercial software package by INRO. It features modeling and forecasting of transportation networks. Once a model is defined, various traffic assignment algorithms can be run on it to forecast traffic on the network. These algorithm include the Frank-Wolfe's algorithm, which is a link-based iterative algorithm for the traffic assignment problem. The EMME software package offers highly optimized performance and a rich set of useful modeling features.

4.7 Economical analysis

The following economical analysis gives an estimated value of the commercial value of the project developed.

Hardware

The project was developed on a middle tier 2-years-old laptop computer. The laptop was bought at a commercial value of €800. Considering a personal computer's life expectancy of 4 years we can compute the cost of the hardware as a percentage of its total life expectancy. The project was developed during 6 months, therefore the hardware cost associated with it is €100.

Software

All the software used in this project is free software. For the main code of the project the Eclipse IDE was used. The programming of the format converter in python was done using the Emacs editor. This document was produced and formatted using LaTeX. Therefore the total software cost for the project was €0.

Staff

The project implied two main phases: design and implementation. The specification is not included since the algorithm was derived from an already specified algorithm. Considering 6 months as the project development time and considering 6 hours a week for the design and 4 hours a day for the programming phase. The costs associated with the staff are resumed in the following table.

Role	Price/hour	Total hours	Total cost
Software engineer	55 €/hour	144	7920€
Programmer	30 €/hour	672	20160€

Therefore the final project cost was €28180.

4.8 Conclusions

In this project we have implemented two versions of a gradient projection algorithm for the asymmetric traffic assignment problem. The first version was based on the algorithm proposed in [5] for the diagonal traffic assignment problem while the second was a modification of it.

Even though in theory the number of paths can grow exponentially during the problem's resolution, the implementation has shown that path-based algorithms can be considered in order to solve the problem efficiently, even on large scale networks.

We have found that the best practical performance for the shortest path tree computation is obtained using the D'Esopo-Pape extension to the Bellman-Ford's algorithm. On the other hand, we have shown that the SLF-LLL extension does not achieve good results in the experiments and its performance is comparable to the basic version of the label-correcting algorithm.

The profiling of the application has shown that the most time consuming procedures are the shortest path tree search and the path cost update method. Therefore optimizing these procedures could lead to a significant speedup in the algorithm's performance.

We have noticed that the choice of the step size modifier is critical for the convergence speed of the algorithm. Most of the step sizes described in this document are selected a priori, usually as a function of the current iteration number. We argue that an intelligent choice on the step size, maybe defined as a function of the current solution, could show better convergence results since it could directly exploit the problem structure. An attentive analysis should be necessary while defining this new function so that the convergence conditions remain valid.

In the first version of the algorithm implemented the step size is dependent on the second order derivatives of volume-delay functions. We have found though that this step size definition leads to oscillations in the gap function trend during the execution of the algorithm.

Finally we have noticed that unfortunately in neither of the algorithms implemented the convergence speed is highly remarkable.

4.9 User manual

This section describes the steps necessary in order to execute the algorithm.

4.9.1 Compilation

The project must be compiled from source code. This can be done by simply compiling all the source code files and header contained in the **src** folder with any C++11 compatible compiler. The C++11 version flag must be passed as a parameter to the compiler. Using the gnu-c++ compiler the following command compiles the project to the **ata** executable file

g++ std=c++11 src/*.cpp src/*.h -o ata

4.9.2 Parameters

The program expects a valid configuration file as parameter. If no configuration file is provided the file **ata.conf** in the current directory is used as the default configuration file. If no such file exists an error is thrown.

4.9.3 Configuration file format

A demo configuration file can be found in **conf/demo.conf**. The configuration is composed of (keyword,value) pairs. Each pair must span a single line and must be separate by a whitespace character. The keywords are

output filepath to the file where the results should be saved. Results are saved in .csv format.

network filepath to the file containing the network representation.

matrix filepath to the file containing the trip rate matrix representation.

max-speed maximum speed in kilometers per hour to be considered when calculating link's travel times.

priority-alpha parameter for the priority volume-delay functions.

priority-beta parameter for the priority volume-delay functions. Beta must be greater then one.

np-theta parameter for the non priority volume-delay functions.

np-b parameter for the non priority volume-delay functions.

modeling-time length in hours of the modeling time period considered.

epsilon required precision to reach convergence.

ata-algorithm gradient projection algorithm selected. 0 for the original, 1 for the modified version.

spt-algorithm shortest path tree algorithm selected. 0 for SLF-LLL, 1 for D'Esopo-Pape, 2 for Bellman-Ford.

4.9.4 Input file format

The program implemented reads the network structure and the trip rate matrix from external input file. These files must be specified in a particular format in order to be parsed correctly.

Input .ata file format

Contains a list of the nodes and links of the network. A demo .ata file can be found in **data/demo.ata**. Each node corresponds to a node record. Each link corresponds to a link record. The fields of both of these records are defined in the following tables

Node record = type id

Field	Type	Description
type	char	c for centroid nodes, t for normal nodes
id	int	unique node identifier

Link record = from to length type capacity

Field	Type	Description
from	int	identifier of link's starting node
to	int	identifier of link's ending node
length	double	link's length in kilometers
type	int	1 for priority links, 0 for non priority links
capacity	double	link's capacity

Input .mat file format

The program uses the input matrix format defined in the INRO/EMME user manual as the trip rate matrix format.

Input file converter

One of the initial decisions in the project was to use the INRO/EMME format also for the network representation. Due to the unnecessary parsing of a large portion of the input files we decided to define the .ata data format described in 4.9.4. Though, since most of the network representations are expressed in the INRO/EMME format, we implemented a converter from the original to the .ata format. The converter, implemented in python, can be found in **data/ata-converter**.

The converter takes as input a network structure file as defined in the INRO/EMME user manual and outputs a file in the .ata format corresponding to the same network.

One important thing must be noticed in order to convert successfully between the two formats: the user defined data 2 for the link records in the INRO/EMME format must contain the link's capacity.

A third file containing the non priority links can be passed as input in order to define link priorities in the network. The priority file should contain an header and then a list records defined as a pair (starting node id, ending node id) for each non priority link in the network. The starting and ending node identifiers must be separated by a whitespace character. Each pair must span a single line.

Chapter 5

References

[1] M. Patriksson. *The Traffic Assignment Problem - models and methods.* V.S.P. Intl Science, 1994.

[2] Yossi Sheffi. *Urban Transportation Networks: Equilibrium Analysis With Mathematical Programming Methods.* Prentice Hall, 1985.

[3] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[4] Dimitri Bertsekas. *Network Optimization: Continuous and Discrete Models.* Athena Scientific, 1998.

[5] R. Jayakrishnan, Wei T. Tsai, Joseph N. Prashker, and Subodh Rajadhyaksha. A faster path-based algorithm for traffic assignment. *University of California Transportation Center*, 1994.

[6] Anthony Chen, Lee Der-Horng, and R. Jayakrishnan. Computational study of state-of-the-art path-based traffic assignment algorithms. *Matematics and Computers in Simulation*, 2002.

[7] David G. Luenberger. *Introduction to Linear and Nonlinear Programming.* Addison-Wesley, 1973.

[8] HenryX. Liu, Xiaozheng He, and Bingsheng He. Method of successive weighted averages (mswa) and self-regulated averaging schemes for solving stochastic user equilibrium problem. *Networks and Spatial Economics*, 9(4):485–503, 2009.