Department of Engineering Science

Part IV Project

Faster Shortest Path Computation for Traffic Assignment

Boshen Chen

Supervisors:
Dr. Andrea Raith
Olga Perederieieva

September 2013

Abstract

The Traffic Assignment (TA) problem involves the selection of optimal path for every vehicle in a transportation network subject to congestion. One method for solving TA is the Path Equilibration (PE) algorithm. PE requires to find the shortest path repeatedly between every origin and destination pair in the network for a large number of iterations, while the travel times of the roads change due to congestion. The aim of this project is to find a faster shortest path algorithm for PE. Dijkstra's algorithm, A* search and their bidirectional versions and eight different versions of priority queue data structures that improve these algorithms' performance are implemented. Two strategies for avoiding shortest paths altogether in the iterative environment of PE are also developed. The first strategy is to avoid the next few iterations when the shortest path of the previous two iterations are the same. The second strategy is to randomly skip the next shortest path calculation in the hope for the situation where the previous and current iteration is going to be the same. We present experimental results that demonstrate the run time differences between the priority queues, shortest path algorithms and path calculation avoiding strategies. We conclude A* search algorithm using the priority queue implementation from the C++ standard template library with random avoiding strategy has the best performance.

Acknowledgements

I would like to thank my supervisors, Dr Andrea Raith and Olga Perederieieva from the Department of Engineering Science at the University of Auckland, for their patient guidance, assistance and useful critiques throughout this project.

I would also like to thank my fellow part IV students, especially Danny Tsai for his insightful ideas and kind help.

Table of Contents

1 Introduction **1**
- 1.1 Introduction to traffic modelling . . . 1
- 1.2 Purpose of this project . . . 2
- 1.3 Structure of the report . . . 3

2 The Traffic Assignment Problem **4**
- 2.1 The deterministic network equilibrium model . . . 4
- 2.2 Solving the traffic assignment problem . . . 6
- 2.3 The path equilibration algorithm . . . 7
- 2.4 Convergence and stopping criterion . . . 9

3 Solving the Shortest Path Problem **10**
- 3.1 Notations and definitions . . . 10
- 3.2 Generic shortest path algorithm . . . 11
- 3.3 Bellman-Ford-Moore algorithm . . . 13
- 3.4 Dijkstra's algorithm . . . 14
  - 3.4.1 Priority queues . . . 14
- 3.5 Bidirectional Dijkstra's algorithm . . . 16
- 3.6 A* Search . . . 18
  - 3.6.1 A* search in traffic assignment . . . 19
- 3.7 Bidirectional A* search . . . 20
- 3.8 Pre-processing algorithms . . . 21
  - 3.8.1 A* search with landmarks . . . 21

4 Implementation Details **23**
- 4.1 Traffic assignment implementation . . . 23
- 4.2 Graph storage . . . 23
- 4.3 Priority queue implementations . . . 24

5 Experimental Results **26**
- 5.1 Road networks . . . 26
- 5.2 Results on priority queues . . . 27
  - 5.2.1 Discussion on priority queues . . . 27
  - 5.2.2 Discussion on Fibonacci heap . . . 28
- 5.3 Results on shortest path algorithms . . . 29
  - 5.3.1 Discussion on shortest path algorithms . . . 30

5.4 A* search with landmarks 31

6 Solving Shortest Path in Traffic Assignment **34**
6.1 Avoiding shortest path calculations 34
6.1.1 Avoiding a number of iterations 35
6.1.2 Avoiding the next shortest path calculation randomly 36
6.2 Results on avoiding shortest path calculations 36

7 Conclusions and Future Work **39**
7.1 Conclusions 39
7.2 Future work 40

Appendices **41**

A Run time results **41**
A.1 Priority queue results 41
A.2 Shortest path algorithms results 42
A.3 Large network results using random skipping strategy 42

References **43**

Chapter 1

Introduction

1.1 Introduction to traffic modelling

Nowadays a large portion of people's daily lives involve activities that relate to transportation. For example, most people need to travel between their workplace and residence twice a day, or buy goods from shops that need to be delivered across the city. In the meanwhile, transportation networks expand and improve constantly to cater for people's demand for efficient travelling, but the rate of improvement does not confront with the rate of population growth. As a result, the network becomes inefficient, resulting in traffic congestion.

Congestion leads to major economical losses due to time delays and increased usage of petrol. Congestion causes air pollution that increases respiratory problems such as asthma, and the exhaust gas exacerbates global warming. Congestion also increases noise pollution and causes frustration, which in turn accelerates traffic accidents. It is important for traffic designers to be able to reduce congestion problems, and eliminate its negative effects.

Making improvements to the transportation network tends to be very costly, so a good plan is necessary: use the least amount of investment for the greatest change. In order to make better plans for traffic design, different mathematical models have been built to estimate the current and future behaviour of the transportation system. One particular model called the transportation forecasting model is commonly used. The aim of this model is to estimate future traffic usage when the system is changed. For example, changes to a transport system can include upgrading or adding roads, changing roundabouts to traffic lights, or providing better public transport.

The transportation forecasting model has 4 stages (shown in Figure 1.1): trip generation, trip distribution, mode choice and traffic assignment (Sheffi, 1985). In the model, the solution of each stage can be passed to the previous ones to improve transportation forecasting and traffic design. This means the model can be calibrated numerous times to establish an accurate model which can estimate future traffic behaviour. In summary, the model collects traffic demand data and generates origins and destinations for travellers (trip generation). It then calculates the number of trips that are required between each origin and destination (trip distribution), and decides which transportation method should be used for each trip (mode choice). Finally it decides the route (e.g. the shortest path) that each trip needs to travel on (traffic assignment), where traffic

flow is estimated for every modelled road in the network. The traffic assignment problem in the last stage of the forecast model is a complicated problem. This is because congestion occurs as traffic flows are assigned onto the network, and it is very difficult to find an equilibrium situation where everybody in the network finds their best route. Generally the equilibrium situation happens when it is assumed that people are selfish, everyone tries to minimise travel times by travelling along their shortest paths.

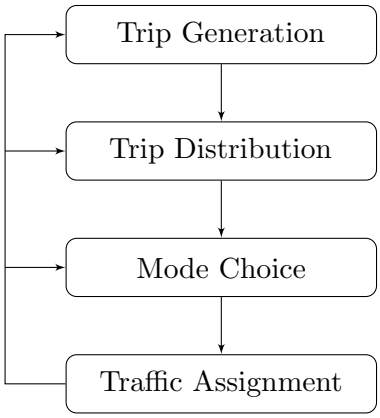


Figure 1.1: Transportation forecasting model

1.2 Purpose of this project

The transportation forecasting model has been implemented in many software packages for traffic design. One key observation is that, the traffic assignment problem can take hours, or even days to solve. It turns out that, the bottleneck is in the algorithm for solving the shortest path problem. This is because algorithms for solving the traffic assignment problem are usually iterative, where each iteration requires to find millions of shortest paths between every origin and destination in the network. Although each shortest path calculation may take only a fraction of a second, cumulatively the computational time can be very long. Therefore the purpose of this project is to find a faster algorithm for solving the shortest path problem in an iterative environment. As a result, traffic assignment will be solved faster for larger and more complicated road networks. This will also allow city designers to estimate traffic flows further into the future and make better decisions on road network design.

In summary, The algorithms that are going to be studied are:

- Bellman-Ford's label correcting algorithm,
- Dijkstra's label setting algorithm (using different data structures),
- Bidirectional Dijkstra's algorithm,

- A* search,
- Bidirectional A* search.

We also aim to find and discuss the following techniques that can speed up shortest path calculations in an iterative algorithm that solves the traffic assignment problem:

- pre-processing algorithms,
- use information from previous iterations,
- avoid shortest path calculations.

1.3 Structure of the report

Chapter 2 gives a short description of the traffic assignment problem and presents a specific algorithm called the path equilibration algorithm that solves it. This algorithm is the begin of our experiments for this project. Chapter 3 introduces the shortest path problem and presents some of the well established algorithms that solve it. Implementation details are presented in Chapter 4. Chapter 5 shows the results and discussions for the algorithms described in Chapter 3. Chapter 6 presents strategies and results for solving the shortest path problem in path equilibration algorithm of the traffic assignment. Finally conclusions and future work are drawn in Chapter 7.

Chapter 2

The Traffic Assignment Problem

In a transportation network, every traveller wishes to travel between different pairs of origins and destinations. As travellers start to travel in the network, congestion happens and travel speeds tend to decrease rapidly due to more and more interactions between the travellers and the increase in traffic volume. This leads to the problem of travellers wishing to find the fastest route to travel on, meanwhile taking account of congestion while every other traveller is trying to do the same. In transportation modelling point of view, we wish to find a traffic flow pattern in the network with a given travel demand between the origin and destination pairs. This is called the *traffic assignment problem*.

The traffic assignment problem is commonly solved by traffic equilibrium models. The notion of traffic equilibrium was first formalized by Wardrop (1952), where he introduced the postulate of the minimisation of the total travel costs. His first principle states that "the journey times on all routes actually used are equal and less than those which would be experienced by a simple vehicle of any unused route." The traffic flows that satisfy this principle are referred to as "user optimal" flows, as each traveller chooses the route that is the best for them.

In this chapter, the network equilibrium model is first stated. Then a particular solution algorithm for solving such a model is described. Finally the motivation of this project in the context of solving the traffic assignment problem is brief explained.

2.1 The deterministic network equilibrium model

In order to formulate a mathematical model for solving the traffic assignment problem, two assumptions are commonly made. We assume deterministic traffic flows and fixed demand, and the change of travel time on any link does not depend on the change of traffic flows on the other links.

Using notations from Florian and Hearn (1995, 2008), we consider a transportation network represented as a graph $G = (N, A)$, where N is a set of nodes and A a set of directed links in the network. The number of vehicles (or flow) on link a is v_a $(a \in A)$, and the cost of travelling on a link is given by the link cost function $c_a(v)$ $(a \in A)$, where v is the vector of link flows over the entire network.

Let I be the set of origin-destination (O-D) pairs and K_i be the set of cycle-free paths for O-D pair $\forall i \in I$. The origin to destination traffic demands g_i $(i \in I)$ are distributed over directed paths $k \in K_i$, and it is assumed $K_i \neq \emptyset$ and $K = \cup_{i \in I} K_i$. Let h_k be the flows on paths k that satisfy the conservation of flow and non-negativity constraints:

$$\sum_{k \in K_i} h_k = g_i \quad \forall i \in I, \tag{2.1}$$

$$h_k \geq 0 \quad \forall k \in K. \tag{2.2}$$

The corresponding link flows v_a are given by:

$$v_a = \sum_{k \in K} \delta_{ak} h_k \quad \forall a \in A, \tag{2.3}$$

where

$$\delta_{ak} = \begin{cases} 1 & \text{if link } a \text{ is on path } k, \\ 0 & \text{otherwise.} \end{cases} \tag{2.4}$$

In order to solve for user-optimal flows, the Wardrop (1952) user-equilibrium condition is applied:

$$c_k(v^*) - u_i^* \begin{cases} = 0 & \text{if } h_k^* > 0 \\ \geq 0 & \text{if } h_k^* = 0 \end{cases} \quad \forall k \in K_i, i \in I, \tag{2.5}$$

where

$$c_k(v) = \sum_{a \in A} \delta_{ak} c_a(v) \quad \forall k \in K, \tag{2.6}$$

and

$$u_i = \min_{k \in K_i} [c_k(v)] \quad \forall i \in I. \tag{2.7}$$

To elaborate, this condition means that the traffic is in equilibrium when no traveller in the network can find a faster route than the one that is already being travelled on. Furthermore, this condition assumes that the travellers have complete knowledge about the network, they always choose the best route to travel based on the current information about the network. This means the equilibrium is the result of everybody simultaneously attempting to minimise their own travel costs.

With constraints (2.1) - (2.4), the user optimal optimisation problem has objective

$$\min S(v) = \sum_{a \in A} \int_0^{v_a} c_a(x) \mathrm{d}x. \tag{2.8}$$

It has been shown in Florian and Hearn (1995) that optimisation problem (2.8), (2.1)-(2.4) has the user-equilibrium solution as the optimal solution.

The *Bureau of Public Roads* (1964) (BPR) link cost function $c_a(v_a)$ is often used to model the travel time on link $a \in A$. The function is given by

$$c_a(v_a) = c_f \left(1 + B \left(\frac{v_a}{C_a}\right)^{\alpha}\right), \tag{2.9}$$

where B and α are the parameters for the level of congestion that can be experienced, and c_f and C_a are the free-flow travel time and link capacity. It is important to note this cost function only depends on traffic flow on its own link, and it is strictly monotonic, continuous and differentiable. An example of this link cost function is shown in Figure 2.1

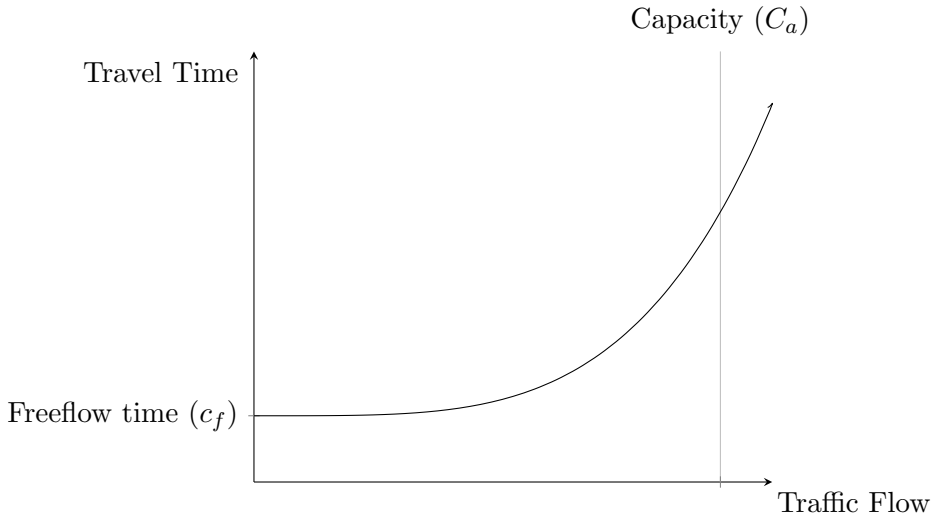


Figure 2.1: Travel time function.

2.2 Solving the traffic assignment problem

Different methods exist for solving the traffic assignment problem. They are classified by how the solution is presented. They include link-based, path-based and origin-based. The solution variables for link-based are link flows, path-based are path flows, and origin-based are link flows coming from a particular origin. Summaries of these methods can be found in Zhou et al. (2010). These solution methods, which have been widely studied, result different numerical accuracies and vary greatly in run time and memory requirement (Perederieieva et al., 2013).

The deterministic network equilibrium model described in the previous section is equivalent to a convex cost differentiable optimisation problem, where a wide range of algorithms exists for solving such problem. They include, the linear approximation method, the linear approximation with parallel tangents method, the restricted simplicial decomposition method, and the path equilibration algorithm (Florian and Hearn, 1995).

In this project, we focus on the path-based algorithm (also known as path equilibration algorithm) proposed in Dafermos and Sparrow (1969). The algorithm has traditionally been discarded by transportation researchers as too memory-intensive and slow for large networks (Jayakrishnan et al., 1994). As a result, the development of path-based algorithm has only began in the last decade, and there exist lots of areas for improvement (Perederieieva et al., 2013).

2.3 The path equilibration algorithm

The general approach of the path equilibration algorithm is equivalent to Gauss-Seidel decomposition, which is an iterative method for solving a linear system of equations. In a step of the algorithm, path flows h_k ($\forall k \in K_i$) between a single O-D pair $i \in I$ are considered by keeping the flows of all other O-D pairs fixed. The algorithm iteratively solves each O-D pair sub-problem until any of them cannot be improved.

The sub-problem for solving each O-D pair $i \in I$ is another fixed-demand network equilibrium problem:

$$\min \quad \sum_{a \in A} \int_0^{v_a^i + \bar{v}_a} c_a(x) \mathrm{d}x. \tag{2.10}$$

$$\text{s.t.} \quad \sum_{k \in K_i} h_k = \bar{g}_i, \quad i \in I, \tag{2.11}$$

$$h_k \geq 0, \quad k \in K_i, \tag{2.12}$$

where flows $\bar{v}_a$ are fixed:

$$\bar{v}_a = \sum_{i' \neq i} \sum_{k \in K_i} \delta_{ak} h_k, \tag{2.13}$$

and flows v_a^i are varied:

$$v_a^i = \sum_{k \in K_i} \delta_{ak} h_k. \tag{2.14}$$

The Gauss-Seidel decomposition (or 'cyclic decomposition' by O-D pair) is stated as follows.

> *'cyclic decomposition' by O-D pair* (Florian and Hearn, 1995)
>
> *Step 0.* Given initial solution, set $l = 0$, $l' = 0$.
>
> *Step 1.* If $l' = |I|$, stop; otherwise set $l = l \bmod |I| + 1$ and continue.
>
> *Step 2.* If the current solution is optimal for the l^{th} sub-problem (2.10)-(2.14), set $l' = l' + 1$ and return to step 1; otherwise solve the l^{th} sub-problem, update the path flows, set $l' = 0$ and return to step 1.

The path equilibration algorithm for solving the sub-problem (2.10)-(2.14) finds the solution of the traffic assignment problem by balancing path flows between each O-D pair, it finds the shortest and longest path and equalizes the travel times between them. Let $K_i^+ = \{k \in K_i | h_k > 0\}$ be the set of paths with positive flows.

The algorithm for solving the l^{th} sub-problem of each O-D pair $i \in I$ is stated as follows.

Path Equilibration Algorithm (Florian and Hearn, 1995)

Step 0. All-or-nothing assignment: find an initial solution v_a^i and $\bar{v}_a$; $c_a = c_a(v_a^i + \bar{v}_a)$ and the initial K_i^+.

Step 1. Compute the costs of the currently used paths c_k, $k \in K_i^+$. Find k_1 such that $c_{k_1} = \min_{k \in K_i} [c_k]$ and k_2 such that $c_{k_2} = \max_{k \in K_i^+} [c_k]$.

If $(c_{k_2} - c_{k_1}) \leq \varepsilon$, go to next O-D pair.

Step 2. Compute the shortest path $\tilde{k}$ with cost $\tilde{s}_k = \min_{k \in K_i} [c_k]$; If $\tilde{s}_k < \min_{k \in K_i^+} [c_k]$, then the path $\tilde{k}$ is added to the set of kept paths, $K_i^+ = K_i^+ \cup \tilde{k}$.

Step 3. Define the direction $d_{k_1} = (c_{k_2} - c_{k_1})$ for path flow k_1 and $d_{k_2} = (c_{k_1} - c_{k_2})$ for path flow k_2. Find the step size λ which redistributes the flow $h_{h_1} + h_{k_2}$ between the paths k_1 and k_2 using the defined directions d_{k_1} and d_{k_2} in such a way that their costs become equal. That is, shift flow from the longest path to the shortest path in the set of kept path K_i^+.

Step 4. Using the λ obtained, update $h_k = h_k + \lambda d_k, k = \{k_1, k_2\}; v_a^i = v_a^i + \lambda y_a; c_a = c_a(v_a^i + \bar{v}_a)$.

In step 0, a so called all-or-nothing assignment is performed for each of the O-D pairs, where it finds the shortest path for O-D pair i and assigns all traffic supplies along that path to the destination. In step 1, the two paths that have the minimum and maximum costs are found. If costs of the two paths are the same, then we can go to the next O-D pair because all travellers have minimised their travel times. In step 2, since travel times have not been minimised, we improve the currently used paths set from step 1 by computing a new shortest path. In steps 3 and 4, traffic flows are redistributed between the longest and shortest paths, these steps are equivalent to solving the Wardrop equilibrium shown in Equations (2.5)-(2.7).

Keep in mind the all-or-nothing assignment is different from the traffic assignment solution. The all-or-nothing assignment gives the solution where between every O-D pair, all traffic flows are assigned along the shortest path. Meanwhile the traffic assignment gives the solution where every individual traveller travels along their shortest path, so between O-D pairs, there is a set of paths with different traffic flows. This is presented by the set of kept path K_i^+ in the path equilibration algorithm.

In summary, in each iteration of the path equilibration algorithm, it first computes the shortest

path with fixed traffic flows and link costs for O-D pair $i \in I$, moves the traffic flow closer to satisfying the Wardrop equilibrium, updates all the link costs and finally continues with the next O-D pair.

Path equilibration is said to be memory intensive because for each O-D pair, it has to store all of the kept path K_i^+. On networks with millions of O-D pairs, the algorithm may require more than 10 gigabytes of memory, as our algorithms show.

If we are given a large network and assume it requires many iterations to find the optimal solution, it can be shown that a large number of shortest path calculations are needed. For example, given the algorithm takes 20 iterations to solve a small network with $100,000$ O-D pairs, it require more than 6 minutes to solve if each shortest path calculation consumes 0.01 second. In reality, networks may contain millions of O-D pairs. Thus for this project we wish to investigate and find a faster shortest path algorithm, and also investigate other ways of improving solution time for the traffic assignment problem.

2.4 Convergence and stopping criterion

For completeness, the convergence criterion for the traffic assignment is discussed in this section. It it known that the objective function (2.8) of the problem is convex (any local minimum is also the global minimum), so the convergence of the Gauss-Seidel strategy is ensured (Florian and Hearn, 2008). The convergence criterion of traffic assignment algorithms is normally measured by so called relative gap (Rose et al., 1988). Rose et al. (1988) states that "the relative gap is expressed by the difference between the current value of the objective function and the lower bound as a percentage of the current objective function." Here the objective function is the user equilibrium (UE) solution of the traffic assignment problem. And the lower bound refers to the all-or-nothing (AON) assignment in step 0 of the path equilibrium algorithm.

The relative gap (RGAP) is computed as

$$\text{RGAP} = \frac{\text{UE} - \text{AON}}{\text{UE}} = \frac{\sum_{a \in A} v_a c_a - \sum_{i \in I} g_i c_{\tilde{k}_i}}{\sum_{a \in A} v_a c_a}. \tag{2.15}$$

The AON solution is the sum of travel times $c_{\tilde{k}_i}$ of each traffic demand g_i travelling on their shortest path $\tilde{k}_i$. The UE solution is the sum of travel times over the entire network. These two values are improved from iteration to iteration during the path equilibration algorithm until they become identical, i.e. RGAP converges to 0. The traffic assignment problem is solved when RGAP is 0, but normally we stop the algorithm at some tolerance such as 10^{-6}.

It is worth to note that the speed of convergence is highly dependent on how small the relative gap is set, as well as the size and complexity of the network. A smaller relative gap will result in more iterations for the traffic assignment algorithms.

Chapter 3

Solving the Shortest Path Problem

The previous chapter describes the details of the traffic assignment problem. Algorithms such as the path equilibration algorithm used to solve the traffic assignment require to solve the shortest path problem on a static graph, where the graph does not change while the shortest path is getting solved. Thus in this chapter, we investigate various standard algorithms which have been developed to address the problem of finding the shortest path. This chapter states notation and definitions for the shortest path problem and discusses the theory of solving it. Shortest path algorithms that are applicable for the traffic assignment problem are summarised, including the discussion of their advantages and drawbacks.

3.1 Notations and definitions

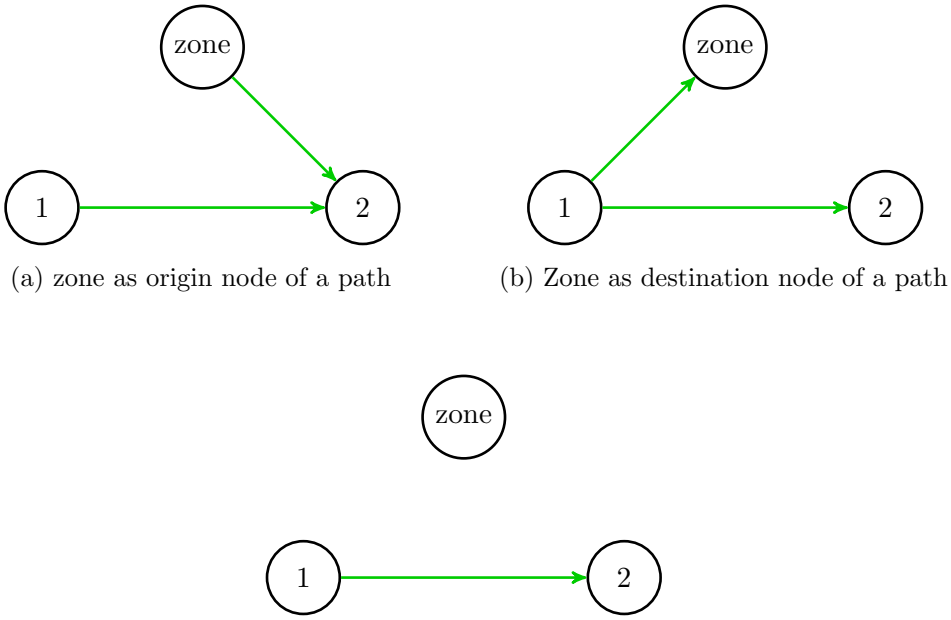
The Shortest Path Problem (SPP) is the problem of finding the shortest path from a given origin to some destination. There are two types of SPP that are going to be analysed in this chapter: a single-source and a point-to-point SPP. The single-source SPP wishes to find all of the shortest paths originating from a origin, and the point-to-point SPP wishes to find one shortest path between an origin and destination. More emphasis is going to be put on the point-to-point SPP required by the path equilibration algorithm described in the previous chapter.

We present the notation mainly borrowed from Cormen et al. (2001) and Klunder and Post (2006). We denote $G = (N, A)$ a weighted, directed graph, where N is the set of nodes (origins, destinations, and intersections) and A the set of arcs (links). We say A is a subset of the set $\{(u, v) \,|\, u, v \in N\}$ of all ordered pairs of nodes. We denote the link (arc) cost function $c : A \to \mathbb{R}$ which assigns a cost (travel time) to any arc $(u, v) \in A$. We write the costs of link (u, v) as: $c((u, v)) = c_{uv}$.

The path P inside a transportation network has to be a directed simple path, which is a sequence of nodes and arcs $(u_1, (u_1, u_2), u_2, \ldots, (u_{k-1}, u_k), u_k)$ such that $(u_i, u_{i+1}) \in A$ for $i = 1, \ldots, k-1$ and $u_i \neq u_j$ for all $1 \leq i < j \leq k$. Note u_1 is the origin and u_k is the destination of the path P, u_1 and u_k together is called an O-D pair for this path. For simplicity, we denote s to be the source (origin) and t to be the target (destination) for any path P.

In traffic assignment, the origin and destination nodes used for traffic supply and demand are

referred as zone nodes. The zones are conceptual nodes that are untravellable in the network, which means a path between two zone nodes must not pass through another zone node. Figure 3.1 demonstrates how a zone node behaves under different conditions. If the zone is an origin node, then only out-going arcs are allowed. If it is a destination node, then only in-coming arcs are allowed. And if it is neither, then no arcs can pass through it, otherwise an incorrect path is going to be created.



(a) zone as origin node of a path

(b) Zone as destination node of a path

(c) Zone as neither origin nor destination node of a path

Figure 3.1: Behaviour of arcs on zone nodes acting as origin, destination or neither.

3.2 Generic shortest path algorithm

A family of algorithms exists for solving the shortest path problem. In this section the generic case for these algorithms is described.

This family of algorithms aims to find a distance label vector $(d_1, d_2, \ldots, d_v)$, and the corresponding shortest path between each origin and destination (Klunder and Post, 2006). Each d_v tracks the least distance of the path going from the origin s to a node v. A shortest path P is optimal

when it satisfies the following conditions:

$$d_v \leq d_u + c_{uv}, \quad \forall (u,v) \in A, \tag{3.1}$$

$$d_v = d_u + c_{uv}, \quad \forall (u,v) \in P. \tag{3.2}$$

The inequalities (3.1) are called Bellman's condition (Bellman, 1958). To solve the shortest path problem, we wish to find a label vector d which satisfies Bellman's condition for all of the nodes in the graph. In the generic shortest path algorithm, it continuously finds a node that violates Bellman's condition (3.1) and updates its distance label with the path that has a shorter distance that connects to it. All shortest paths connecting the origin s to all other nodes in N are guaranteed to be found when both Equation (3.1) and (3.2) hold. It is important to note that arcs with negative costs are permitted, but the graph must not contain negative cycles.

Algorithms for solving the SPP generally use some kind of queue $\mathcal{Q}$ to store the label distances d_v ($\forall v \in N$). In the label vector, a node v is said to be

- *unvisited* when $d_v = \infty$,
- *scanned* when $d_v \neq \infty$ and it is still in the queue,
- *labelled* when the node has been retrieved from the queue and its distance label cannot be improved further.

If a node v is labelled then its distance value is guaranteed to represent the shortest path distance from s to v.

To keep track of the shortest path found so far for node v, we denote p_v the predecessor of node v. The shortest path can be constructed by following the predecessor of the destination node t back to the origin node s. We set $p_s = -1$ to indicate that node s does not have a predecessor.

Algorithm 1 (Klunder and Post, 2006) describes the generic shortest path algorithm mentioned above, with an extra constraint required when solving a TA problem: travelling through zone nodes is not permitted. In essence, this algorithm repeatedly selects node $v \in \mathcal{Q}$ and updates its distance label if Bellman's condition is violated for all its out-going arcs.

Algorithm 1 is generic because the rule for selecting the next node u (the 'next' function in line 8) and the implementation of the queue $\mathcal{Q}$ are unspecified. Different algorithms use different rules and implementations to give either the one-source or the point-to-point shortest path algorithm (Pallottino and Scutellà, 1997). The rules and implementations are described in the following sections to give concrete algorithms for solving the SPP.

Algorithm 1 The Generic Shortest Path Algorithm

```
1: procedure GenericShortestPath(s)
2:     Q ← Q ∪ {s}                                  ▷ initialise queue with source node
3:     p_s ← −1                                     ▷ origin has no predecessor
4:     d_s ← 0
5:     for all u ∈ N : u ≠ s do                     ▷ all nodes are unvisited except the source
6:         d_u ← ∞
7:     while Q ≠ ∅ do
8:         u ← next(Q)                              ▷ select next node
9:         Q ← Q \ {u}
10:        if u is not zone or u = s then
11:            for all v : (u, v) ∈ A do            ▷ check Bellman's condition for all successors of u
12:                if d_u + c_vw < d_v then
13:                    d_v ← d_u + c_vw
14:                    p_v ← u
15:                    if v ∉ Q then
16:                        Q ← Q ∪ {v}              ▷ add node v to queue if unvisited
```

3.3 Bellman-Ford-Moore algorithm

When some specific strategy is applied to maintain the queue $\mathcal{Q}$ and arc costs are allowed to have negative values, the generic shortest path algorithm is addressed as the label correcting algorithm, or Bellman-Ford-Moore algorithm (credited to Bellman (1958); Ford (1956) and Moore (1959)).

The algorithm is addressed as the label correcting algorithm because the distance labels do not get permanently labelled when a node u is retrieved from the queue, the distance label of node u may be 'corrected' by some other node v when v is retrieved from the queue and re-updates u.

One specific strategy for maintaining the queue is described in Sheffi (1985). This strategy is shown to be very effective for computing shortest path on transportation networks. It avoids duplicating computation by not physically moving (or reordering) nodes in the queue, as well as not adding nodes to the queue if they are already in it. The nodes in the queue are simply processed from front to end. Scanned nodes are firstly added to the end of the queue, and if the scanned node is already in the queue, then the node is put in front of the queue so they can be processed first. This strategy, developed by Pape (1974), transforms the queue into a *double-ended queue* (or *deque*).

In order to satisfy the Bellman's condition for all arcs, the algorithm has to scan all arcs $|N| - 1$

times, resulting in a run time of $O(|N||A|)$[1].

3.4 Dijkstra's algorithm

The classic algorithm for solving the single-source shortest path problem is the label setting algorithm published in Dijkstra (1959). The algorithm is addressed as label setting because when the next node u is retrieved from the queue, it gets permanently labelled, The shortest path going to this node is solved and the distance label represents the length of this shortest path. In order to achieve label setting, it is assumed that all arc costs are non-negative, and the queue $\mathcal{Q}$ is modified to always have the minimum distance label in the front. This modification allows the algorithm to visit every node in the graph exactly once, where the next node is labelled in the order of non-decreasing distance labels.

The advantage of this algorithm is that, when the next labelled node is the destination node, the algorithm can be stopped for the point-to-point shortest path problem. This reduces the total run time as the algorithm does not have to scan the entire graph, which is desirable for the point-to-point shortest path problem involved in the path equilibration algorithm described in the previous chapter.

3.4.1 Priority queues

The run time performance of Dijkstra's algorithm depends heavily on the implementation of the queue $\mathcal{Q}$ for storing the scanned nodes. Cormen et al. (2001) suggests the use of priority queues. Priority queues are a collection of data structures that keep track of elements using an extra priority attribute. This allows an element with a higher priority to be processed before an element with a lower priority when iterating through the queue. In the shortest path problem, the priority is measured by the distance labels, where smaller distance labels have higher priority. Priority queues that have smaller values come first are name min-priority queues, whereas larger values come first are named max-priority queues.

Algorithm 2 shows the use of the min-priority queue in Dijkstra's algorithm. The min-priority queue has three main operations: Insert, Extract-Min and Decrease-Key. The Insert operation (line 2 and 17) is used for adding new nodes to the queue. The Extract-Min operation (line 8) is used for getting the element with the minimum distance label. The Decrease-Key (line 19) operation is used to decrease the value of the distance label if the node is already in the queue.

[1]Big-O notation: if $f \in O(g)$, then $f \leq g$. This means f does not grow asymptotically faster than g, it describes the worst-case scenario of f. In context, run time of $O(|N||A|)$ means the performance of the algorithm is directly proportional to the number of nodes multiplied by the number of arcs.

Algorithm 2 Point to Point Dijkstra's Algorithm

```
1: procedure DIJKSTRA(s, t)
2:     Insert(Q , u)                          ▷ initialise priority queue with source node
3:     p_s ← −1                                ▷ origin has no predecessor
4:     d_s ← 0
5:     for all u ∈ N : u ≠ s do                ▷ all nodes are unvisited except the source
6:         d_u ← ∞
7:     while Q ≠ ∅ do
8:         u ← Extract-Min(Q)                  ▷ select next node with minimum value
9:         if u = t then
10:            Terminate Procedure             ▷ finish if next node is the destination
11:        if u is not zone or u = s then
12:            for all v : (u, v) ∈ A do       ▷ check Bellman's condition for all successors of u
13:                if d_u + c_vw < d_v then
14:                    d_v ← d_u + c_vw
15:                    p_v ← u
16:                    if v ∉ Q then
17:                        Insert(Q, v)        ▷ add node v to queue if unvisited
18:                    else
19:                        Decrease-Key(Q, v)  ▷ else update value of v in queue
```

According to Cormen et al. (2001), a min-priority queue can be implemented via an array, a binary min-heap or a binary search tree, where each implementation gives different run time performances.

In the array implementation, the distance labels are stored in an array where the n^{th} position gives the distance value for node n. Each Insert and Decrease-Key operation in this implementation takes $O(1)$ time, and each Extract-Min takes $O(|N|)$ time (searching through the entire array), giving a overall time of $O(|N|^2 + |A|)$.

A binary min-heap is a binary tree that satisfies the min-heap property: the value of each node is smaller or equal to the value of its child nodes. Cormen et al. (2001) shows that the performance of Dijkstra's algorithm can be improved with a binary min-heap if the graph is sufficiently sparse. In this implementation, each Insert and Extract-Min operation takes $O(\log(|N|))$ time in total of $|N|$ times, and the Decrease-Key operation takes $O(\log(|N|))$ time in total of $|A|$ times. The total running time of Dijkstra's algorithm using min-priority is therefore $O((|N| + |A|)\log(|N|))$, which is an improvement compared to the array implementation.

The running time can be further improved using a Fibonacci heap developed by Fredman and Tarjan (1987). Historically, the development of the Fibonacci heap was motivated by the observation

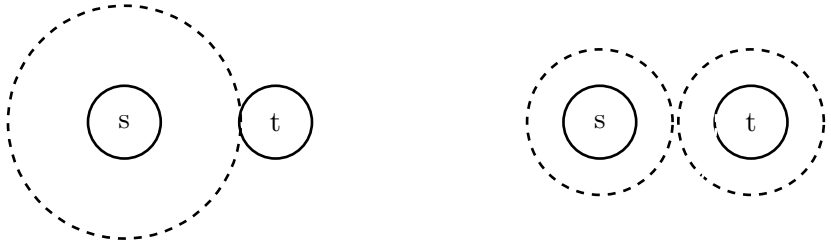
that Dijkstra's algorithm typically uses more Decrease-Key operations compared to Extract-Min operations. In Fibonacci heap, each of the $|N|$ Extract-Min operation takes $O(\log(|N|))$ amortized time[1] and each of the $|A|$ Decrease-Key operation takes only $O(1)$ amortized time. The total running time is therefore $O(|N|\log(|N|) + |A|)$, which is a further improvement.

Min-priority queue can be implemented as a binary search tree. In a binary search tree, the worst case for insertion, deletion and search for an element all have $O(\log(|N|))$ time. Dijkstra's algorithm can easily be modified to accommodate a binary search tree: when label distance of a node needs to be updated, we remove that node from the tree and insert a new one with the updated value (this is analogous to the Decrease-Key operation). Dijkstra's algorithm using a binary search tree runs $O((|N| + |A|)\log(|N|))$ time in the worst case, which is the same compared to the min-binary heap implementation.

In this project, all of the mentioned priority queue implementations are tested. The results can be found in Section 5.2.

3.5 Bidirectional Dijkstra's algorithm

Dijkstra's algorithm can be imagined to be searching radially outward in a circle with the origin in the centre and destination on the boundary. Likewise, Dijkstra's algorithm can be used on the reverse graph (all arcs reversed in the graph) from the destination node. Thus Dijkstra's algorithm can be started from the origin and destination at the same time. The motivation for doing this is because the number of scanned nodes can be reduced when searching bidirectionally: two smaller circles growing outward radially instead of a larger one. Figure 3.2 demonstrates the difference in search area between Dijkstra's algorithm and its bidirectional version on a planar graph using Euclidean distances. It is easy to see that the total search area of the bidirectional version is a lot smaller.



(a) Dijkstra's algorithm (b) bidirectional

Figure 3.2: Difference between the scanned area of label setting and its bidirectional version

[1] Amortized time gives how much time is taken in total when an operation is repeated a million times. For example with different inputs, run time is averaged between the worst-case and the best-case.

It is common to conclude that the shortest path is found when the two searches meet somewhere in the middle, but this is not actually the case. There may exist another arc connecting the two frontiers of the searches that has a shorter path (proof can be found in Klunder and Post (2006)). The correct termination criteria was first designed and tested by Pohl (1971) based on research presented by Dantzig (1963); Nicholson (1966) and Dreyfus (1969). The procedure and algorithm are summarised in Klunder and Post (2006), and the termination criteria is presented by Pohl (1971).

The bidirectional search algorithm is shown in Algorithm 3. Two independent Dijkstra's algorithms are performed alternatively on the forward and reverse graph (forward and backward search). The two searches terminate when some node is permanently labelled in both directions. Once the searches is terminated, the correct shortest path is found by looking for an arc connecting the frontiers of the two searches that may yield a shorter path. This extra requirement increases the run time significantly, where searches are needed for all arcs that connect all labelled nodes in the forward search to all labelled nodes in the backward search.

Note in Algorithm 3, $\mathcal{R}^s$ contains the set of nodes v in the forward search that are permanently labelled from the origin node s, $\mathcal{R}^s$ have corresponding label distances d_v^s. Similarly, $\mathcal{R}^t$ contains the set of nodes v in the backward search that are permanently labelled from the destination node t, $\mathcal{R}^t$ have corresponding label distances d_v^t.

Algorithm 3 Bidirectional Dijkstra's Algorithm

1: **procedure** BIDIRECTIONAL(s, t)
2: Execute one iteration of the forward algorithm. If the next node u is already labelled permanently by the backward algorithm ($u \in \mathcal{R}^t$), go to step 3. Else, go to step 2.
3: Execute one iteration of the backward algorithm. If the next node u is already labelled permanently by the forward algorithm ($u \in \mathcal{R}^s$), go to step 3. Else, goto step 1.
4: Find $\min\{\min\{d_v^s + c_{vw} + d_w^t | v \in \mathcal{R}^s, w \in \mathcal{R}^t, (v, w) \in A\}, d_u^s + d_u^t\}$, which gives the length of the correct shortest path between s and t.

In recent years, Goldberg and Werneck (2005) improved the bidirectional algorithm using a better termination condition, where step 3 of Algorithm 3 is embedded during the searches. The termination condition is summarised as follows. During the forward and backward search, we maintain an extra variable, μ, to present the length of the shortest path seen so far during the forward and backward search. Initially $\mu = \infty$. When an arc (v, w) is visited by the forward search and node w has been scanned in the backward search, or vice versa, we know the shortest path $s - v$ and $w - t$ have lengths d_v^s and d_w^t respectively. During the search, if $\mu > d_v^s + c_{vw} + d_w^t$ then the current connected path $s - v - w - t$ is shorter than the one before, so we update $\mu = d_v^s + c_{vw} + d_w^t$. The algorithm terminates when a node is permanently labelled in both directions, where μ gives the length of the shortest path.

Goldberg, Harrelson, Kaplan and Werneck (2006) showed and proved a stronger termination condition on top of the previous one. The searches can be stopped if the sum of the two top priority queue values is greater than μ

$$\text{top}_f + \text{top}_r \geq \mu.$$

Here top_f and top_r are the next minimum distance labels that have not been labelled in the forward and backward search.

Overall, the bidirectional version of Dijkstra's algorithm can be faster than the single direction one if it is implemented correctly using the best termination criterion.

3.6 A* Search

Dijkstra's algorithm can be imagined as growing the shortest path tree radially out from the origin, the location of the destination does not affect how the shortest path tree is grown. In fact, heuristic estimates can be used to guide the shortest path tree towards the destination, forming an ellipsoid shape. The use of heuristic estimates was first described by Hart et al. (1968), where the algorithm is given the name A* search. A* search is a goal directed search where the direction of search is aimed towards the destination.

Let h_v be the heuristic estimate for the shortest path distance between node v to destination t. We apply Bellman's condition such that an optimal solution exists, that is

$$h_v \leq h_u + c_{uv} \quad \forall (u, v) \in A, \tag{3.3}$$

$$h(t) = 0. \tag{3.4}$$

Although h_v is a heuristic function, optimal solutions (i.e. shortest path) can still be obtained under the following conditions. Hart et al. (1968) state that the heuristic estimate function h must be admissible and consistent. Admissible means that the heuristic must never over-estimate the distance to the destination. Consistent means that the estimated path length of a node to the destination must not be greater than the estimated path length of its predecessors (i.e. Equations (3.3) and (3.4)). These two conditions mean that if the heuristic estimate is smaller than or equal to the actual distance going to the destination, then the optimal shortest path can always be found. In graphs that are measured by Euclidean distances, the heuristic estimate h_u is the straight Euclidean distance between node u and destination t.

To implement A* search, Dijkstra's algorithm is modified. When a node v is about to be added to the priority queue, the heuristic estimate h_v is calculated. Compared to Dijkstra's algorithm, instead of inserting node v with its distance label d_v, we now insert with $d_v + h_v$. By doing so, nodes that are closer to the destination are now labelled first.

3.6.1 A* search in traffic assignment

In our traffic assignment problem, geographical coordinates and Euclidean distances cannot be used. This is because the length of the arcs are determined by the BPR link cost function, where travel time on the link is based on the traffic flow in the current iteration. It is important to note here that in the path equilibration algorithm for solving the traffic assignment problem, the traffic flow changes in each iteration of the algorithm and so does the travel time. When the traffic assignment is getting solved, there are only two ways to obtain the travel time estimate from any given node to the destination. They are either to use the travel time from the previous iteration, or use zero-flow travel times.

At first glance one may use travel times from the previous iteration, but this does not work. This is because traffic flows can decrease from one iteration to the next. The decreased traffic flows result in non-admissible travel times, where some of the arc travel times from the previous iteration is longer than the current travel times. Using these longer travel times from the previous iteration will overestimate the travel time for the current iteration, the resulting shortest path is not guaranteed to be optimal any more.

Another option is to use zero-flow travel times for the heuristic estimates. The estimates h_v can be obtained from computing the shortest path tree for every node v where the traffic flow of the entire network is set to 0. So when solving for the shortest path of any O-D $(s-t)$ pair when there are traffic flows, h_v is equivalent to the zero-flow travel time of the shortest path from node v to destination t. These computed zero-flow travel times are both admissible and consistent. This can be shown by analysing the BPR link cost function shown in Figure 2.1. The function is a monotonic non-decreasing function with the lowest value being the zero-flow travel times. As traffic flows change from iteration to iteration, the travel times can never be smaller than the zero-flow travel times so they will never be an overestimate. This means zero-flow travel times can be used for the heuristic estimates, and the shortest path is guaranteed to be optimal.

Overall, A* search does not improve the worst case time complexity compared to Dijkstra's algorithm, but it can improve the average case by scanning less nodes in the network. It can be see that when $h_v = 0$ $(\forall v \in N)$, A* search is equivalent to Dijkstra's algorithm. A* search has been tested in various studies. In the case of using road networks, Goldberg and Harrelson (2005) conclude that A* search with Euclidean distance estimates does not improve the run time compared to Dijkstra's algorithm when using Euclidean distance estimates. However, it is still worth a try in the context of solving the traffic assignment problem using zero-flow travel times for the heuristic estimates.

3.7 Bidirectional A* search

Since bidirectional search can be applied to Dijkstra's algorithm, it can also be applied to A* search. The bidirectional A* search can be described by two ellipsoids extending from the origin and destination respectively. This is shown in Figure 3.3, where bidirectional A* search potentially has a smaller search area than the unidirectional version.

One may construct the algorithm with the same termination condition described in the bidirectional Dijkstra's algorithm section (Section 3.5), that is to stop the algorithm when the two frontiers of the searches meet. But the problem with bidirectional A* search is that, due to its two independent searches, it does not label the nodes permanently in the order of their distance from the origin (Klunder and Post, 2006). This can cause either wrong shortest path to be calculated or having the two search frontiers never meet.

The correct strategy for calculating the heuristic estimates and termination criterion is first published in Pohl (1971). The heuristic calculation was later improved in Ikeda et al. (1994). Ikeda et al. (1994) demonstrates that two distinct admissible heuristic functions π_f and π_r for the forward and backward search are not consistent for the bidirectional search, but their average is both admissible and consistent. Here $\pi_f(v)$ is the estimate of distance from node v to destination t in the forward search and $\pi_r(v)$ is the estimate of distance from origin s to node v in the backward search. The new heuristic estimate functions are:

$$p_f(v) = \frac{1}{2}(\pi_f(v) - \pi_r(v)) + \frac{\pi_r(t)}{2}, \tag{3.5}$$

$$p_r(v) = \frac{1}{2}(\pi_r(v) - \pi_f(v)) + \frac{\pi_f(s)}{2}. \tag{3.6}$$

The constants $\frac{\pi_r(t)}{2}$ and $\frac{\pi_f(s)}{2}$ are added by Goldberg, Harrelson, Kaplan and Werneck (2006) to provide better estimates. These two modified heuristic estimates are now consistent and the frontiers of the two searches are guaranteed to meet. The drawback of this modification is that they now provide worse bounds compared to the original π values, and the search area may now be larger than the unidirectional A* search.

(a) A* search (b) bidirectional A* search

Figure 3.3: Difference between the scanned area of A* search and its bidirectional version

Goldberg, Harrelson, Kaplan and Werneck (2006) showed and proved a better stopping criterion compared to the one published in Pohl (1971), where bidirectional Dijkstra's termination criterion is extended. Bidirectional A* search has to be stopped when

$$\text{top}_f + \text{top}_r \geq \mu + p_r(t). \tag{3.7}$$

Here μ is the best $s - t$ path seen so far during the search, top_f and top_r are the minimum distance labels in the forward and backward search respectively (they are minimum valued elemented at the top (front) of the priority queues).

Bidirectional A* search with Euclidean distance heuristic estimates has been tested by different academics, e.g. Klunder and Post (2006) and Goldberg and Werneck (2005). It is not evident whether bidirectional A* search can guarantee significant improvement, as the result is heavily dependent on the configuration of the road network and the quality of the heuristic estimates. Nevertheless, bidirectional A* search is implement and tested, the results can be found in Section 5.3.

3.8 Pre-processing algorithms

In the last two decades, extensive research has been done on the idea of speeding up shortest path calculations using pre-calculated data. They include A* search with landmarks (Goldberg and Harrelson, 2005), reach-based pruning (Goldberg, Kaplan and Werneck, 2006) and Hierarchical search (Ertl, 1998; Pearson and Guesgen, 1998), etc. These pre-processing techniques generally need to spend some time to pre-calculate the data required for speeding up the sub-sequent shortest path queries.

The next section discusses the A* search with landmarks, and its drawback when applied to the traffic assignment problem.

3.8.1 A* search with landmarks

From Section 3.6 follows that different methods can be used to estimate the shortest path distance h_v from node v to a specified destination. The landmarks algorithm developed by Goldberg and Harrelson (2005) is another way to obtain the heuristic estimates. First a small set of landmarks need to be positioned in different locations on the graph, shortest path trees are calculated for every landmark using Dijkstra's algorithm. Heuristic estimates can now be obtained for every node in the graph using the shortest path trees of the landmarks, Figure 3.4 demonstrates the calculation of the estimates using the two triangle inequalities when a landmark is either placed

in front of the scanned node v or behind the destination t. The two triangle inequalities are:

$$\text{dist}(v,t) \geq \text{dist}(L,t) - \text{dist}(L,v), \tag{3.8}$$

$$\text{dist}(v,t) \geq \text{dist}(v,L) - \text{dist}(t,L). \tag{3.9}$$

The heuristic estimate to be used during A* search is the maximum over all landmarks:

$$\text{dist}(v,t) \geq \max\{\text{dist}(L,t) - \text{dist}(L,v), \text{dist}(v,L) - \text{dist}(t,L)\}. \tag{3.10}$$

With these formulations, Goldberg and Harrelson (2005) concluded that it is best to place the landmarks in front of the origins or behind the destinations, i.e. around the outer boundary of the graph.

(a) Landmark placed before v (b) Landmark placed after t

Figure 3.4: Explanatory diagram for triangle inequality

The landmarks algorithm requires to store all shortest path distances for all landmarks, memory may be an issue if there are many landmarks and the network is large. Although the pre-processing stage of the landmarks algorithm can take a long time, it was shown that the shortest path solving times are significantly faster than A* search using Euclidean distance estimates (results can be found in Goldberg and Harrelson (2005)).

For the equilibration algorithm of the traffic assignment, it is unknown whether landmarks using zero-flow travel times can provide better estimates compared to Euclidean distances. Furthermore, both pre-processing and shortest path solving time of this algorithm are heavily dependent on the quantity and placement of the landmarks, which is another optimisation problem where different strategies and algorithms need to be studied. Due to these two problems, landmarks algorithm is not investigated further for this project.

Chapter 4

Implementation Details

The previous chapters have described all the algorithms that are considered for this project. In this chapter, specific implementation details of the algorithms are discussed.

4.1 Traffic assignment implementation

Path equilibration algorithm and other algorithms for solving the traffic assignment problem have already been implemented by Olga Perederieieva, the co-supervisor of this project. The algorithms are implemented in the C++ programming language.

The existing implementation of the path equilibration algorithm uses Bellman-Ford-Moore algorithm for its shortest path calculations as suggested by Sheffi (1985). When solving the traffic assignment problem, the algorithm spends most of its time computing shortest paths. Significant amount of time is also spent on the convergence check step described in Section 2.4, where the algorithm requires to run the Bellman-Ford-Moore algorithm on all of the zones for the all-or-nothing solution.

4.2 Graph storage

To obtain information from the graph when running shortest path algorithms, the storage of the underlying graph has been implemented in such a way that it can provide efficient access to its nodes and arcs. The current implementation uses the Forward Star data structure described in Sheffi (1985). The data structure compactly stores graphs in $O(|N| + |A|)$ spaces, and provides $O(1)$ random access to all of its nodes, it also provides $O(1)$ access to all out-going arcs of a randomly chosen node. Using the Forward Star ensures the run time of accessing the graph can be neglected when analysing the shortest path algorithms.

4.3 Priority queue implementations

As mentioned in Dijkstra's algorithm section (Section 3.4), the performance of shortest path algorithms is heavily dependent on the implementation of the priority queue data structure. Various priority queue implementations exist, they include:

- ⟨priority_queue⟩ from the C++ standard template library,
- ⟨set⟩ from the C++ standard template library,
- ⟨heap⟩ from the C++ Boost library.

Here we use the angled brackets to denote the names are specific implementations of the priority queue data structure.

Each priority queue implementation has some advantages and disadvantages. For example some provide faster tree balancing while others provide faster Extract-Min or Insert operation. All of these implementations are studied for this project.

First we examine six variants of heap implementations from the C++ Boost ⟨heap⟩ library shown in Table 4.1. In the table, N is the number of elements in the priority queue and the time complexities are measured in amortized time. Extract-Max and Increase-Key are referred as Extract-Min and Decrease-Key in Chapter 3. Extract-Max and Increase-Key are used because all Boost library heaps are implemented as max-heap trees, elements are ordered from large to small in the tree so the largest element is extracted first.

Table 4.1: C++ Boost heap implementations with run time in amortized complexity (Blechmann, 2013)

	Insert	Extract-Max	Increase-Key
Binary	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$
Ternary	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$
Binomial	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$
Fibonacci	$O(1)$	$O(\log(N))$	$O(1)$
Pairing	$O(2^{2\log(\log(N))})$	$O(\log(N))$	$O(2^{2\log(\log(N))})$
Skew	$O(\log(N))$	$O(\log(N))$	$O(\log(N))$

Theories and analysis of these heap implementations can be found in Johnson (1975) for Binary and Ternary, Vuillemin (1978) for Binomial, Fredman and Tarjan (1987) for Fibonacci, Fredman et al. (1986) for Pairing, and Sleator and Tarjan (1986) for Skew.

We are interested in using Boost library heaps rather than the C++ standard template library heap due to one reason: the Increase-Key operation. The operation is used to change the distance

labels in the priority queue when a node is scanned and updated during shortest path calculations. In Dijkstra's algorithm, nodes are often scanned multiple times in the label updating step, so instead of adding the node again into the heap tree with a difference value, we can use the Increase-Key operation to increase the value of the node's distance label. The advantage of using this operation is that we can reduce the size of the heap tree, which results in a performance improvement as it takes less time to search and insert nodes to smaller heap trees. Since Boost library heaps are max-heap trees, all distance labels need to be negated when inserting them into the heap in order for them be used in our shortest path algorithms.

In Table 4.1, we observe that the Fibonacci heap has a very interesting time complexity. It has a constant amortized $O(1)$ time for the Insert and Increase-Key operation. This is very attractive for us, but the problem is that we do not know how much constant time it really uses behind its big O notation.

Next we examine ⟨priority_queue⟩ from the C++ standard template library. ⟨priority_queue⟩ is implemented as binary min-heap tree. It provides $O(\log(N))$ Insert and Extract-Min operation. This implementation neither has the Decrease-Key operation nor a way to change node values or delete a node somewhere other than the top of the heap tree. So when solving the shortest path problem, the priority queue is going to contain the same nodes several times but with different distance labels. This is not a problem for our shortest path algorithms. When a node is added to the queue more than once with different distance labels, the one with the smaller distance label is always going to be in front of the queue waiting to be extracted and labelled first, so when that node is labelled, all the other same nodes with larger distance labels in the queue will not affect finding the shortest path, this is because the successors of those nodes have already been assigned with smaller distance labels.

Finally we examine ⟨set⟩ from the C++ standard template library. A *set* is a data structure used to store unique elements that follow a specific order. In the C++ standard template library, it is implemented as a red-black binary search tree. This data structure can be used for our shortest path algorithms because it provides $O(\log(N))$ insert, search and delete operations. For our shortest path algorithms, we can modify them to accommodate the unique elements and specific ordering requirement. To meet the unique elements requirement, instead of using the Increase-key operation whenever a node needs to be updated, we simply delete that node and insert the one with the new value. And for the ordering requirement, we can just order the nodes non-decreasingly by their distance labels, so the node with the minimum distance label always comes first. The advantage of ⟨set⟩ compared to ⟨priority_queue⟩ is that nodes can be removed from anywhere in the data structure, since the performance of the operations is heavily dependent on the number of elements in the data structures., ⟨set⟩ can be faster than ⟨priority_queue⟩ and require less memory compared to the other priority queues.

Chapter 5

Experimental Results

In this chapter we present experimental results. All experiments are run under 12.04 Ubuntu Linux on an ASUS K46C laptop, which has four Intel Core i5-3317U CPU and 3.8GiB RAM.

Various algorithms (including the path equilibration algorithm) for solving the traffic assignment problem has already been implemented by Olga Perederieieva, the co-supervisor of this project. Specifically, only Bellman-Ford-Moore algorithm is available for computing the shortest path.

All results presented in this chapter are based on newly developed code. The algorithms are written in C++, compiled by the g++ compiler using the '-O3' optimisation flag. All timed results are measured for a complete run of traffic assignment, where all algorithms are run on a single core of the CPU. All solutions use 10^{-6} as the relative gap for the stopping criterion.

In this chapter, we are going to

1. first identify the best priority queue data structure to use in Dijkstra's algorithm in Section 5.2,
2. then compare different shortest path algorithms: Dijkstra's algorithm, A* search and their bidirectional versions described in Section 5.3.

5.1 Road networks

Table 5.1 shows the road networks used for this project, the network details are retrieved from Bar-Gera (2013). The table has an extra column showing the minimum number of iterations the path equilibration algorithm takes to solve the networks. Compared to other networks in the table, Anaheim, Barcelona and Winnipeg are small networks. Chicago Sketch is a medium sized network that is part of the Chicago Regional network. Berlin Center is a medium sized network. Philadelphia and Chicago Regional are two large networks that have over a million O-D pairs.

Note the complexity of the traffic assignment problem and the run time of the path equilibration algorithm are heavily influenced by the number of O-D pairs in the networks. For example, although Philadelphia and Chicago Regional networks are similar in size, Chicago Regional is much more complex and hard to solve than Philadelphia as it has twice as many O-D pairs.

Table 5.1: Network Problem Data

Network	Category	Nodes	Arcs	Zones	O-D pairs	Iterations
Anaheim	small	378	914	38	1,406	10
Barcelona	small	910	2,522	110	7,922	27
Winnipeg	small	905	2,836	147	4,344	126
Chicago Sketch	medium	546	2,950	387	93,135	25
Berlin Center	medium	12,116	28,376	865	49,688	23
Philadelphia	large	11,864	40,004	1,525	1,149,795	81
Chicago Regional	large	11,192	39,018	1,790	2,296,227	152

5.2 Results on priority queues

In this section we test different priority queues discussed in Section 4.3. The priority queues are used in Dijkstra's algorithm on the Winnipeg and Chicago Sketch networks. The results are shown in Figure 5.1 and 5.2, where the run times are measured from a complete run of the traffic assignment. On the Winnipeg and Chicago Sketch network, a total of 547,344 and 2,328,375 shortest paths are solved respectively. The exact numerical results can be found in Table A.1 of Appendix A.

On both networks, ⟨priority_queue⟩ has the best performance and the Binomial heap has the worst performance. Skew heap has the best performance among the six heap implementations from the Boost library. Fibonacci heap has worse performance compared to some of the other implementations despite of its $O(1)$ Insert and Increase-Key operation. ⟨set⟩ is significantly slower than ⟨priority_queue⟩.

5.2.1 Discussion on priority queues

The priority queue implementation results show that all of the heap implementations from the Boost library and ⟨set⟩ are worse than ⟨priority_queue⟩. The reason behind this can be explained using the Binary heap implementation. It turns out that the implementation of ⟨priority_queue⟩ is similar to Binary heap from the Boost library, the only difference is their underlying storage of node information. Nodes are stored using an array in the standard library version, whereas the Boost library uses pointer references to keep track of the nodes. Due to computer cache coherence, it is known that accessing data from a nearby random access memory (RAM) locations in a short period of time is faster than accessing from distant memory locations. This is due to cache memory access being much faster than RAM access, and internally a block of memory is pre-fetched into the cache in the hope they will be accessed in a short period of time. In

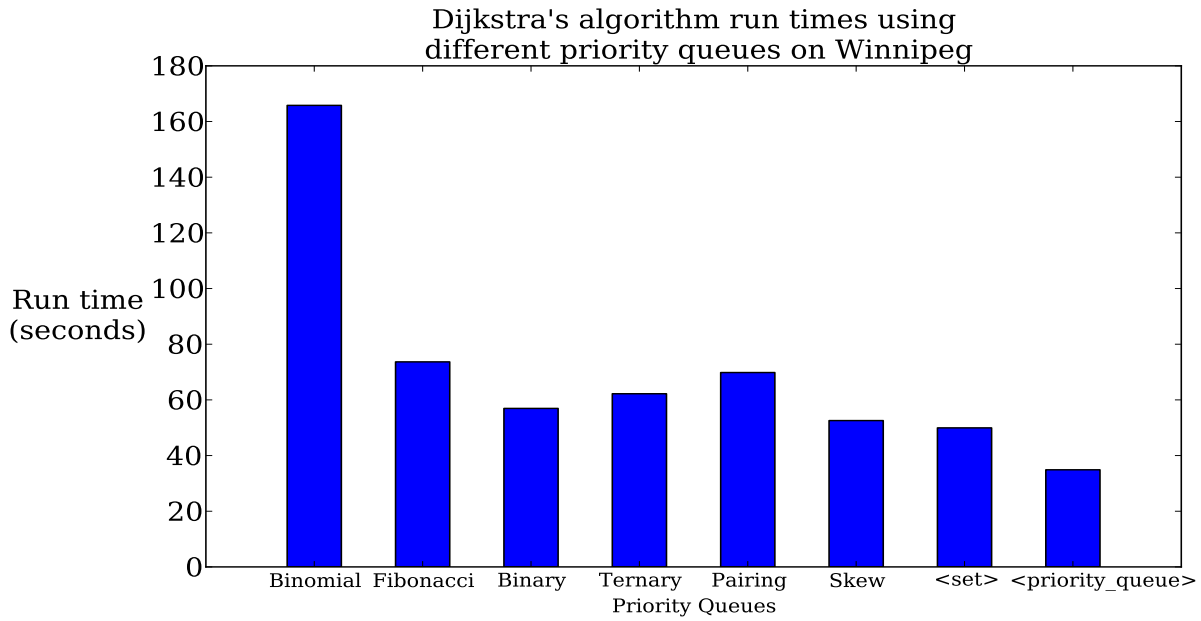


Figure 5.1: Full traffic assignment run times using Dijkstra's algorithm with different priority queues on Winnipeg, a total number of 547,344 shortest paths are solved.

the shortest path algorithms, the heap tree needs to be searched over and over again in a short period of time when nodes are being scanned and inserted. The standard library version uses an array where data are stored linearly in a nearby location, so it is much faster than the pointer based version where memory is allocated in random locations when nodes are inserted.

5.2.2 Discussion on Fibonacci heap

Here we discuss the reason behind Fibonacci heap not performing well despite its $O(1)$ amortized time Increase-Key operation. As described in the two priority queue sections (Section 3.4.1 and 4.3), the Increase-Key operation is used to increase the value of distance label of a node when the node is already in the max-heap. It was discovered that the $O(1)$ time has a very high constant factor, and Fibonacci heap only works well if the underlying graph is large and dense (i.e. every node connects to almost every other node). This discovery comes from the fact that the Increase-Key operation is only used frequently when the graph is dense, so cumulatively its high constant $O(1)$ time will perform better compared to $O(\log(N))$ time in other heap implementations, where N need to be a large number.

We confirm our graph is indeed not dense and the Increase-Key is not used frequent. We find that all of our graphs are very sparse. The degree of any node of any graph is no more than

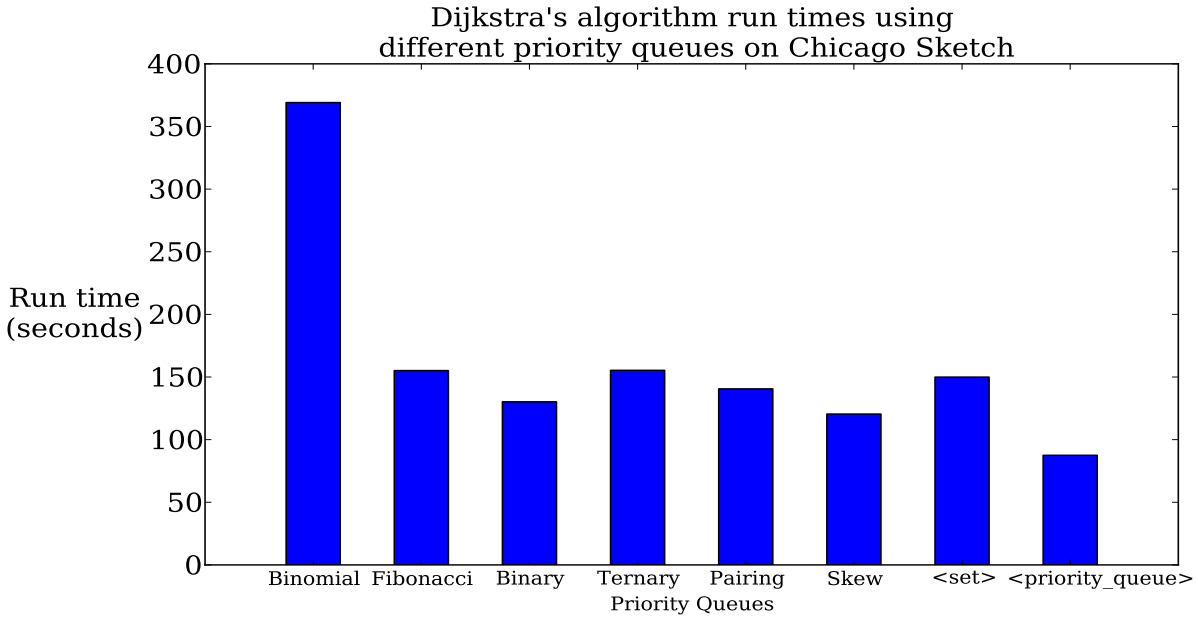


Figure 5.2: Full traffic assignment run times using Dijkstra's algorithm with different priority queues on Chicago Sketch, a total number of 2,328,375 shortest paths are solved.

5, as it is already really rare to have an intersection with 5 roads connected. The graphs only have about 0.4% to 0.6% of arcs in the corresponding complete graph (every node connects to every other node). We also find that in all of the experimented graphs when using Dijkstra's algorithm, the probability of using Increase-Key on any node is only around 1 to 5 percent.

5.3 Results on shortest path algorithms

In Section 5.2 we have identified ⟨priority_queue⟩ to be the most suitable (efficient) data structure to be used in Dijkstra's algorithm. We use ⟨priority_queue⟩, implement and test the following algorithms:

- Bellman-Ford-Moore algorithm (existing),
- Dijkstra's algorithm,
- Bidirectional Dijkstra's algorithm,
- A* search,
- Bidirectional A* search.

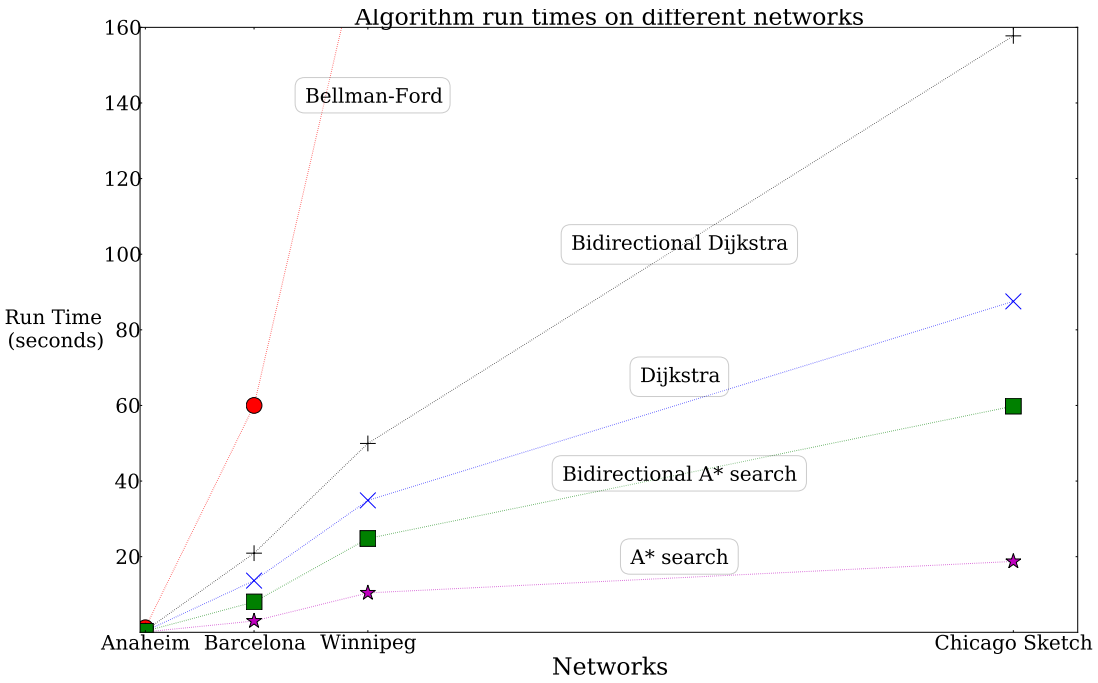


Figure 5.3: Run time performances of different algorithms on different networks

Figure 5.3 shows the performance of the mentioned algorithms on the Anaheim, Barcelona, Winnipeg and Chicago Sketch networks (see Table A.2 of Appendix A for exact numerical results). The networks are spaced out on the horizontal axis to show their relative sizes, i.e. the number of O-D pairs.

Bellman-Ford-Moore algorithm has the worst performance while A* search has the best performance on all networks. The bidirectional versions of Dijkstra's algorithm and A* search are more than twice slower than their unidirectional versions.

5.3.1 Discussion on shortest path algorithms

In this section we investigate the reason for the worse performances of the bidirectional algorithms compared to their unidirectional versions.

First we examine whether search areas of the algorithms are what is expected. Figure 5.4 shows the shortest path trees of the point-to-point algorithms, where the origin and destination nodes are placed on opposite sides of Chicago Sketch network. It can be seen that Dijkstra's algorithm scans the entire network. Bidirectional Dijkstra scans almost the entire network with some nodes left out on the sides of the network. Bidirectional A* scans a slightly larger region near the origin

and destination nodes, and the A* search scans just a few nodes along the shortest path. The behaviour of the algorithms are shown further in Figure 5.5, where the origin and destination are placed close to each other. It can be seen that both Dijkstra's algorithm and its bidirectional version scan almost half of the graph, where the bidirectional version scans fewer nodes. A* search and its bidirectional version scan a small portion of the graph, and they do not scan the area behind the origin and destination node compared to the Dijkstra's algorithm.

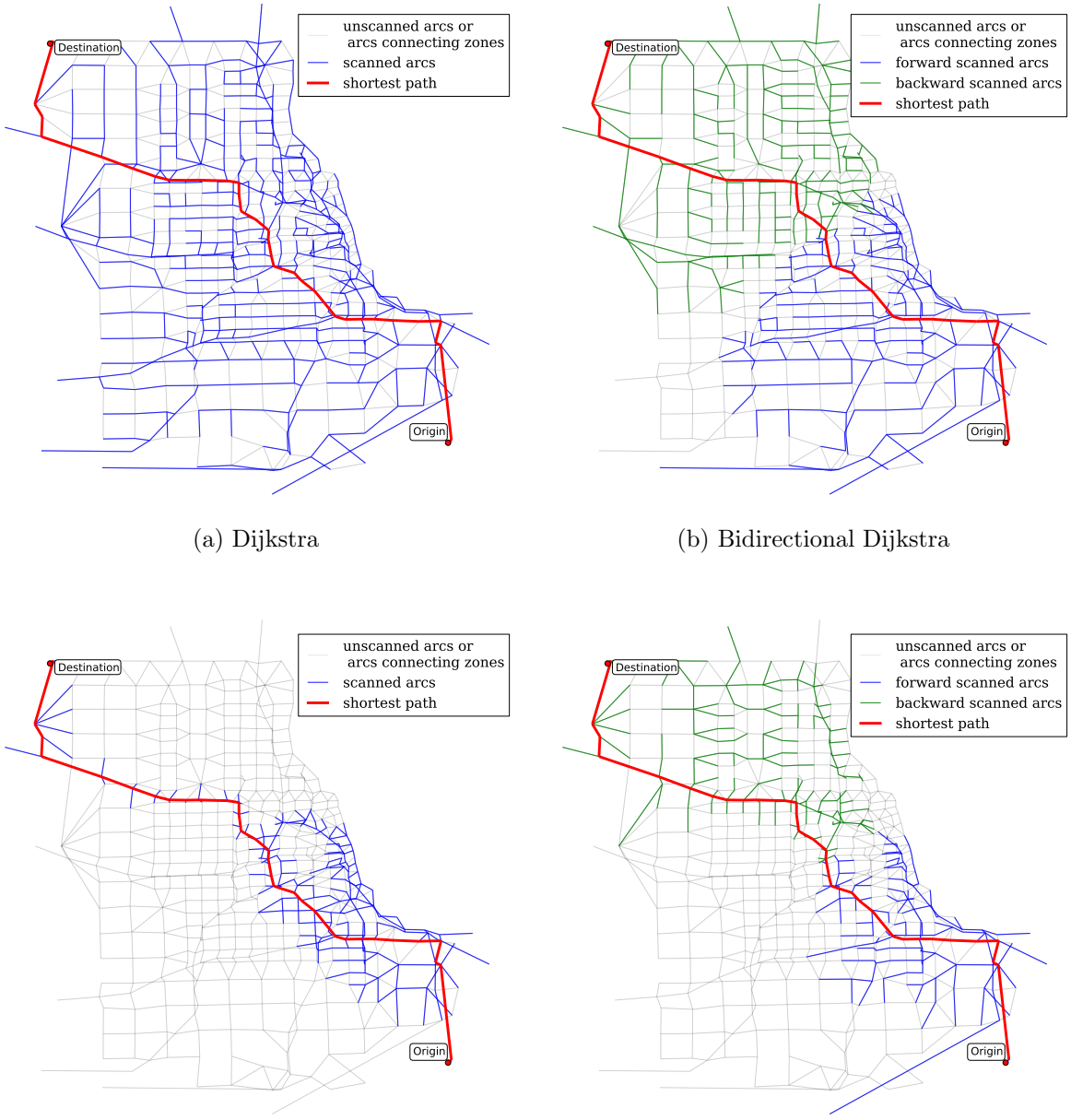
The search areas of the bidirectional Dijkstra's algorithms match what is expected, but not the run times when compared to the unidirectional version. The reason for reduction in run time is due to our implementations. In both forward and backward search, the current shortest path μ needs to be updated every time a node is scanned, and the stopping criterion needs to be checked when a node is labelled. Furthermore, once the algorithm terminates, we need to retrieve the shortest path in both directions by following node predecessors and concatenate them together for the full shortest path. So it is concluded that these additional computations slowed down the run times.

For A* search, the bidirectional version always scans more nodes than the unidirectional version. This is due to the worse bounds of heuristic estimates calculated in the bidirectional search compared to the unidirectional version, That is, heuristic estimates need to be averaged from the forward and backward search in order to find the correct shortest path. Since the bidirectional version has similar stopping criterion compared to the bidirectional Dijkstra's algorithm, it is easy to understand why bidirectional A* has a worse performance.

Finally, there is one thing that needs to be pointed out for A* search. A* search requires to run Dijkstra's algorithm on all nodes (not zones) in the network to obtain the heuristic estimates (zero-flow travel times from every node to every destination) and store the travel time values (not the paths). This procedure is similar to that of pre-processing algorithms, where we need to pre-process the network so subsequent queries can be made faster. In our A* search results, the run times included the pre-processing and the overall run time is still faster than all the other tested algorithms.

5.4 A* search with landmarks

A* search with landmarks algorithm is not implemented for this project. This is due to its sophisticated graph dependent implementation, where we need to either manually or dynamically decide the number of landmarks and their location placements. And also there is a high chance of not being able to work efficiently, as the algorithm is aimed at geographic node locations and Euclidean distances, not travel times based on traffic flows. Using zero-flow travel times may work but it was decided not to investigate this further. Instead, we focus on strategies that avoid some of the shortest path computations altogether in the path equilibration algorithm.



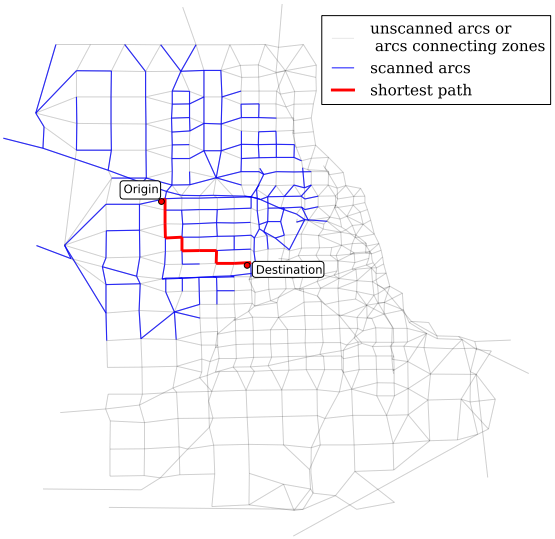
(a) Dijkstra

(b) Bidirectional Dijkstra

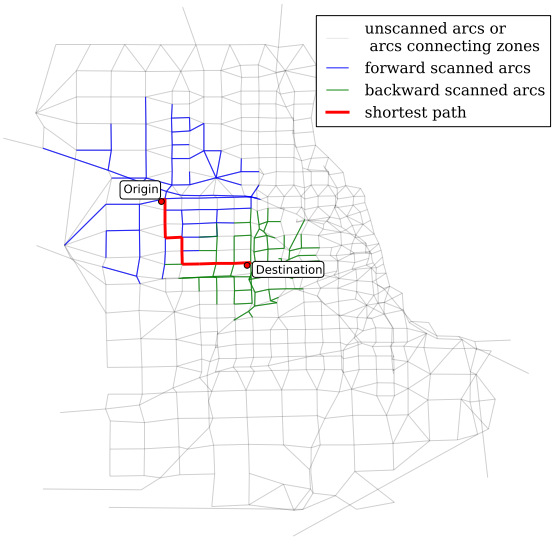
(c) A* Search

(d) Bidirectional A* Search

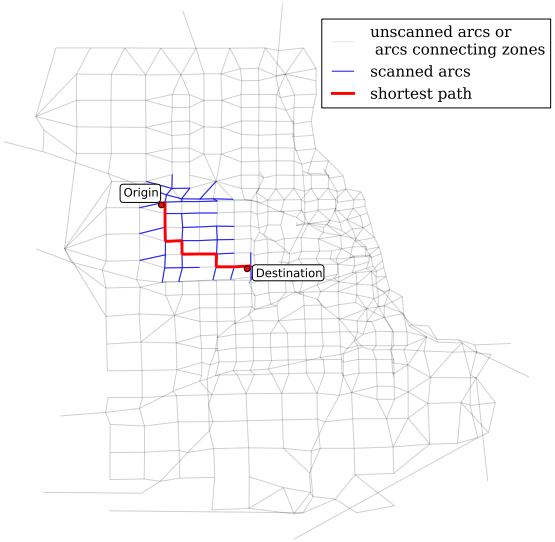
Figure 5.4: Shortest path tree between two distant nodes in the Chicago Sketch network



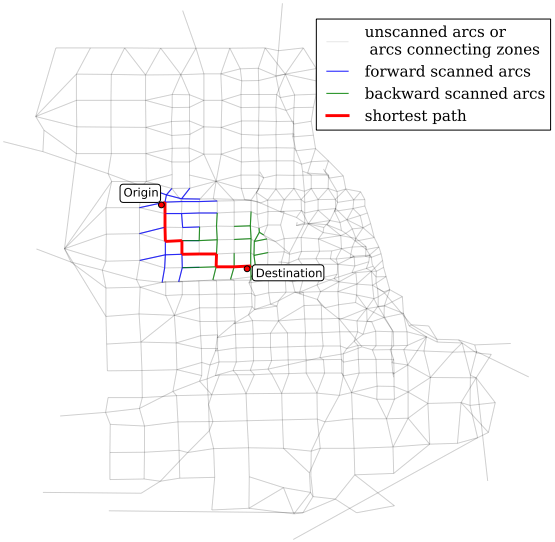
(a) Dijkstra



(b) Bidirectional Dijkstra



(c) A* Search



(d) Bidirectional A* Search

Figure 5.5: Shortest path tree between two close nodes in the Chicago Sketch network

Chapter 6

Solving Shortest Path in Traffic Assignment

The previous chapter suggests the use of A* search with zero-flow travel times as heuristic estimates and the min-heap priority queue implementation from the C++ standard template library for solving the shortest path problem.

So far we have only been dealing with solving the shortest path problem on static graphs for travel time based on fixed flow. In this chapter we discuss solving the problem in the path equilibration algorithm where the graph dynamically changes its arc costs between iterations. Better performance can be achieved if we are able to use information from previous iterations, for example reusing shortest path from the previous iteration.

6.1 Avoiding shortest path calculations

All of the shortest path algorithms mentioned so far need to fully calculate the shortest path between every O-D pair in each iteration. It turns out for every O-D pair in the path equilibration algorithm, their shortest path calculation can be avoided to reduce computational time by using solution from the previous iteration in the current iteration.

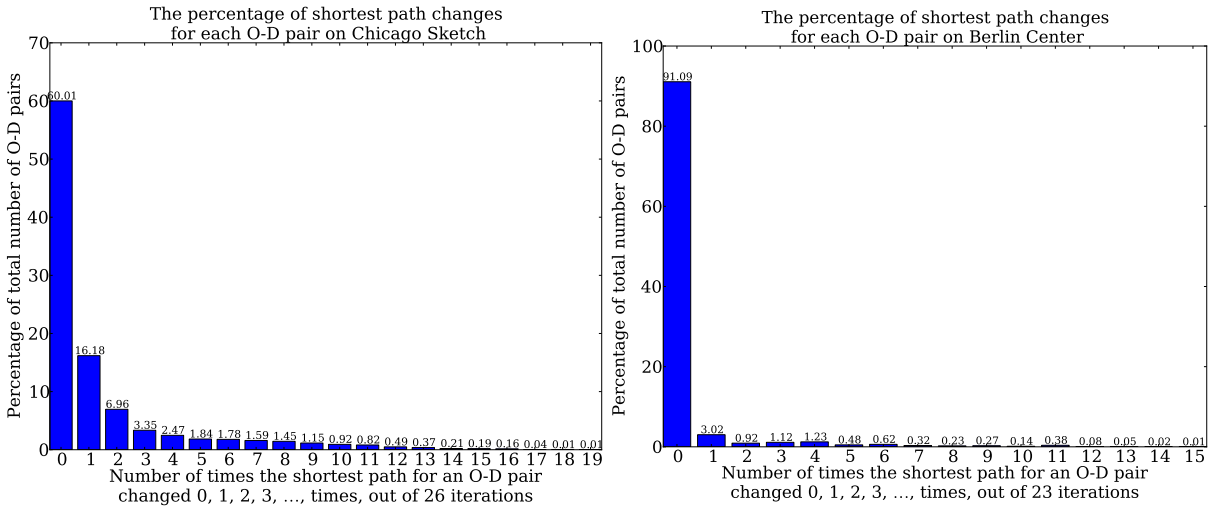
The basic framework is as follows. First a complete iteration of the path equilibration algorithm is performed and all of the shortest paths are stored. Then for every subsequent iterations, we either perform a shortest path calculation or skip the calculation by some strategy. If a shortest path is skipped, the stored path from the last iteration is used.

Two situations can occur if we choose to do so. The first situation is when the shortest path between the previous and current iteration are the same, then we have successfully avoided the calculation and reduced the computational time. The second situation is when the paths are different, then the path equilibration algorithm may not move closer to its equilibrium state after the current iteration, which causes a possible increase in the total number of iterations and computational time. Although the total number of iterations may vary, the path equilibration will still converge. This is because with a proper defined strategy, where a new shortest path will

be calculated eventually, non-converging iterations will get corrected and converge when new shortest path is calculated.

While traffic flows and arc costs change between iterations, if we can prove that shortest path do not change often between iterations, then some strategies can be used to avoid the calculations. The overall computational time is reduced when most shortest path calculations are avoided on the O-D pairs that are not going to change often.

Figure 6.1 shows how often shortest paths change between iterations for the Chicago Sketch and Berlin Center network (details of the networks can be found in Table 5.1). The histogram show the total number of times the shortest path for an O-D pair changed 0,1,2,..., times (in percentage out of all O-D pairs). The figure shows that on both networks, 60% and 91% of all O-D pairs have not changed their shortest path after the initial iteration, and 16% and 3% of all O-D pairs changed their shortest path only once. This means that after the first iterations, the algorithm spends most of its time changing only a dozen of O-D pairs' shortest path. From these observations, it is assured that run time can be reduced if we avoid shortest path calculations that do not change between iterations.



(a) Out of 26 iterations on Chicago Sketch

(b) Out of 23 iterations on Berlin Center

Figure 6.1: The percentage of shortest path change for each O-D pair

6.1.1 Avoiding a number of iterations

The first strategy is as follows. For each O-D pair, if its shortest path did not change in the previous two iterations, then we can delay the next shortest path calculation by a few iterations. This strategy requires prior knowledge of how many iterations there is going to be during a

standard run. This is because if we choose to skip more calculations than the total number of iterations, then there will be excessive iterations resulting in wasted time. If we skip too few iterations, then there may not be any impact on the computational time.

This strategy requires to store shortest path between iterations, memory may become an issue if the number of O-D pairs is large. This strategy also requires to compare the stored paths when iterations are not skipped and copy the stored path to the current iteration when iterations are skipped, this will increase the run time.

6.1.2 Avoiding the next shortest path calculation randomly

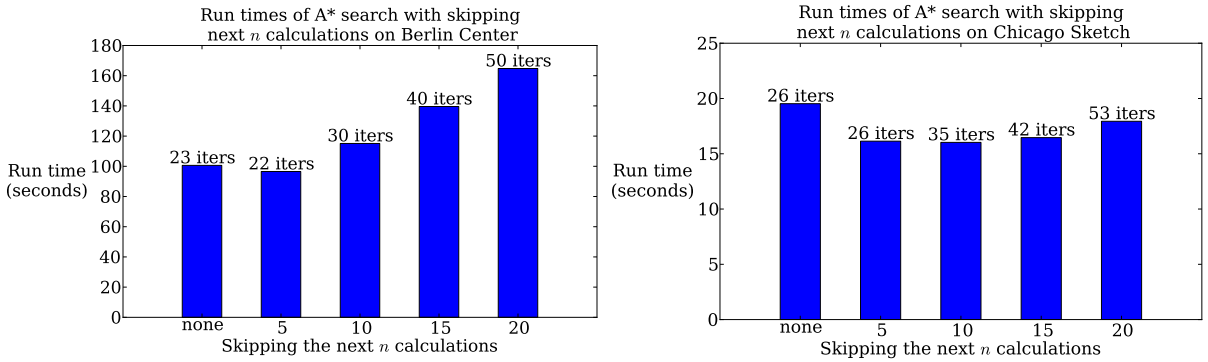
The other strategy is to skip shortest path calculations randomly. That is, when it comes to calculate the shortest path for a given O-D pair, we generate a random number and decide whether to perform the calculation based on that number. The advantage of this strategy is that we do not need to know how many iterations the algorithm will take. The disadvantage is that the computational time can vary between different runs, resulting in unpredictable run times.

This strategy also requires to store and copy shortest paths between iterations, but paths do not need to be compared, so this strategy is faster than the first strategy.

6.2 Results on avoiding shortest path calculations

In this section, we consider avoiding shortest path calculations in the iterative path equilibration algorithm using strategies described in the previous sections. A* search is used to generate results for this section as it is identified to be the most efficient algorithm in Section 5.3.

We present the strategy of avoiding a pre-defined number of shortest path calculations for each O-D pair if the previous two iterations result identical shortest path. The results are shown in Figure 6.2, where the strategy is tested on the Berlin Center and Chicago Sketch network. On both networks, we choose to avoid the next 5, 10, 15 and 20 iterations, these numbers are sensibly chosen because in a normal run of the path equilibration algorithm, it takes 23 and 26 iterations on the Berlin Center and Chicago Sketch network respectively. The strategy does not work well on the Berlin Center network, where run time is only decreased slightly when 5 iterations are skipped, and all other skips increased the run time significantly. Due to the size of the network, the run time is heavily affected by the increase in total number of iterations. The strategy worked well on the Chicago Sketch network. Skipping 5 iterations resulted the same 26 iterations and the run time is decreased by 4 seconds. Run times are still reduced in cases where there is an increase in total number of iterations. Overall the run times become unpredictable when more iterations are skipped.



(a) Berlin Center network

(b) Chicago Sketch network

Figure 6.2: Run time for avoiding shortest path calculations if the previous two iteration did not change

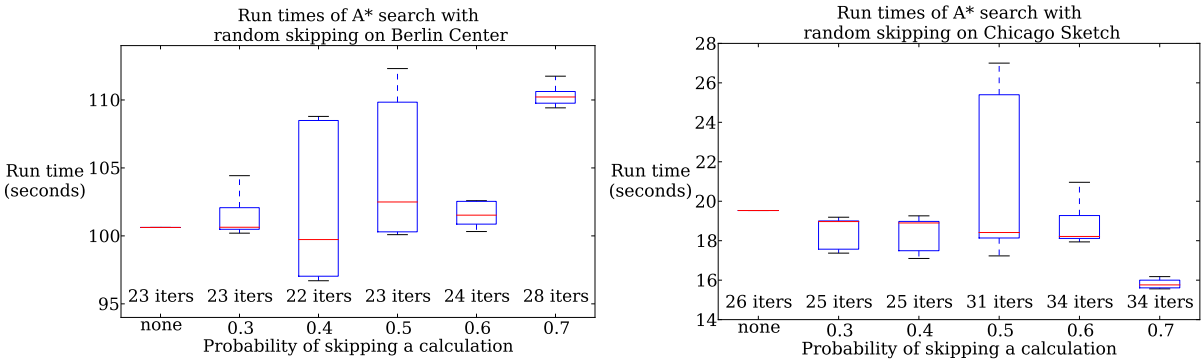
The other strategy is to skip the next shortest path calculation randomly. Figure 6.3 shows the results on both Berlin Center and Chicago Sketch, where probabilities of 0.3, 0.4, 0.5, 0.6 and 0.7 for skipping the next shortest path calculation are chosen. All of the probabilities are run 10 times so the average and extremes of the run times can be determined. On both networks, run times are unpredictable, there are extremes where run times are increased by 10% on Berlin Center and 30% on Chicago Sketch. With 0.7 probability, the run time increased significantly on Berlin Center meanwhile decreased significantly on Chicago Sketch. Overall, the average run times can be reduced for probabilities 0.3 to 0.6.

The random skipping strategy has a better performance than the other strategy, and since only small networks have been tested so far, we now present the random skipping strategy on the Philadelphia and Chicago Regional networks (details in Table 5.1) that have over a million O-D pairs.

Due to memory requirement, we switch to a computer with Intel Xeon CPU and 11.7 GiB RAM for the upcoming results. The run time comparisons are shown in Figure 6.4 (see Table A.3 of Appendix A for extract results). The random skipping strategy uses 50% probability to skip a shortest path calculation. The strategy has a 25% and 27% run time improvement on the Philadelphia and Chicago Regional networks respectively.

From this chapter, we conclude the following:

- the random skipping strategy works well on the tested graphs, but there are extremes where the algorithm takes significantly longer to run using high probability such as 0.7.
- the skipping next few number of iterations strategy works well when just a few iterations are skipped (e.g. 5 iterations), the run time becomes unpredictable when more iterations are skipped.



(a) Berlin Center network

(b) Chicago Sketch network

Figure 6.3: Run time for skipping shortest path calculations randomly

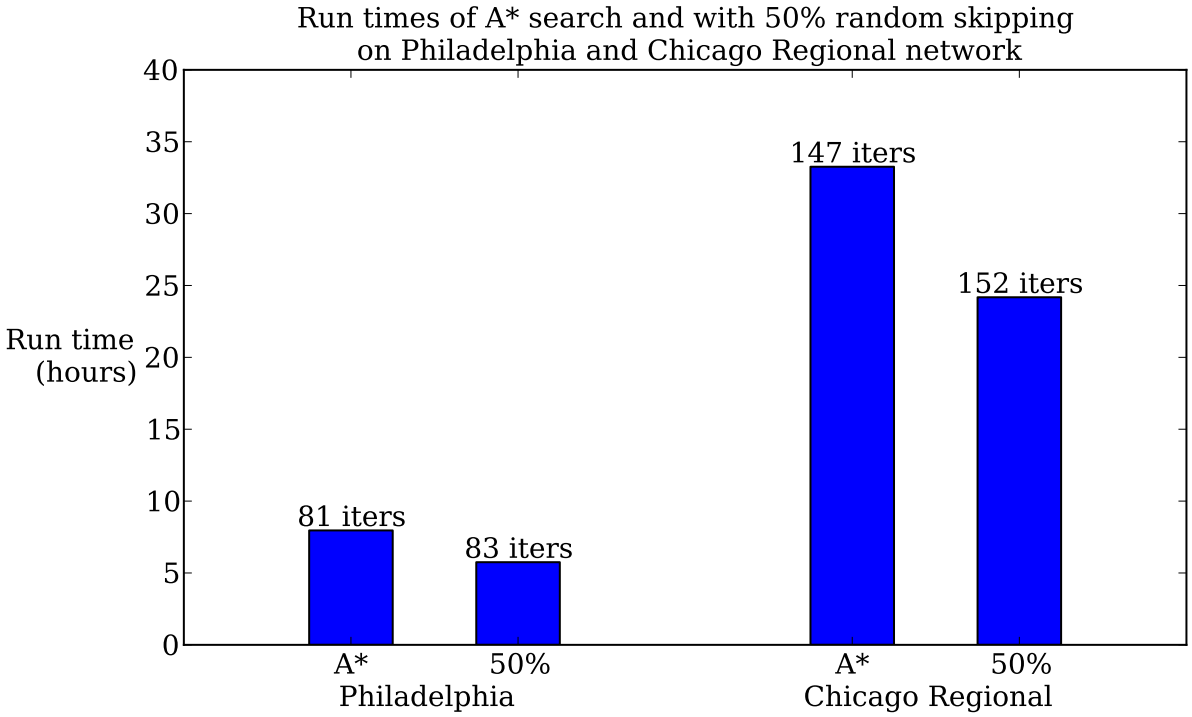


Figure 6.4: Run times of A* search and 50% random skipping on large networks

Chapter 7

Conclusions and Future Work

To summarise, in this project we have studied the point-to-point shortest path problem embedded in the path equilibration algorithm for solving the traffic assignment problem. We have implemented Dijkstra's algorithm, A* search and their bidirectional versions. We also tested eight different priority queues for the shortest path algorithms. Two strategies were developed to improve the performance of these shortest path algorithms when they are used in the iterative path equilibration algorithm. The first strategy is to avoid the next few iterations when the shortest path of the previous two iterations are the same. The second strategy is to randomly skip the next shortest path calculation, with the hope that the shortest path in the current and previous iteration is going to be the same. In addition, we have also investigated the possibility of using pre-processing methods such as A* search with landmarks.

7.1 Conclusions

We now conclude this project with the following results:

- The priority queue implementation from the C++ standard template library has the best performance compared to the six implementations from the C++ Boost library and binary search tree from the C++ standard template library.
- A* search algorithm using zero-flow travel times as heuristic estimates has the best performance.
- Bidirectional versions of Dijkstra's algorithm and A* search have worse performances. Bidirectional Dijkstra is worse because it has to check the stopping criterion at each step. Bidirectional A* is worse because its search area is larger than that of unidirectional A*.
- The strategy of avoiding the next few iterations of shortest path calculations is viable when small number of iterations are skipped.
- The strategy of avoiding the shortest path calculations randomly is viable. By using A* search and 50% random skipping on large networks that require millions of shortest path calculations in each iteration, the run times are further improved by about 25% compared to just using A* search.

7.2 Future work

- In this project, only the path equilibration algorithm has been used to test the strategies of avoiding shortest path calculations. The strategies may also be applicable for other iterative algorithms that solve the traffic assignment problem.
- The current A* search algorithm only runs on a single thread. The algorithm can be improved by implementing a multi-threaded version developed by Inam (2009). The algorithm will run extremely fast as it is designed for GPGPU (General Purpose Graph Processing Unit) run on multi processors using many threads concurrently. The main modification of the algorithm is that instead of sequentially updating all out-going arcs from the labelled node, we update them in parallel using multiple threads.
- A* search with landmarks algorithm may be applicable for our problem but needs to be studied further. This algorithm is difficult to implement, this is because the number of landmarks to use and deciding their placements is another optimisation problem. Tests need to be done in order to find the best performance.
- A family of algorithms exists for dynamically changing graphs, including graphs that have moving nodes or changing arcs. One particular algorithm that tackles the problem of changing arc costs is the Lifelong Planning A* (LPA*), developed by Koenig et al. (2004). Koenig et al. (2004) were able to show experimentally that LPA* is more efficient than A* if the change in arc costs are close to the destination. This means LPA* may be applicable for traffic assignment but more study needs to be done.

Appendix A

Run time results

A.1 Priority queue results

Table A.1: Priority queues run time results on Winnipeg and Chicago Sketch network for Figure 5.2

Network	Priority Queue	Iterations	Time (seconds)
Winnipeg	Fibonacci	128	73.66
	Binary	131	56.94
	Ternary	128	62.21
	Skew	127	52.57
	Pairing	131	69.83
	Binomial	127	165.8
	⟨priority_queue⟩	127	66.89
	⟨set⟩	127	34.89
Chicago Sketch	Fibonacci	25	155.14
	Binary	25	130.22
	Ternary	25	155.34
	Skew	25	120.45
	Pairing	25	140.55
	Binomial	25	369.10
	⟨priority_queue⟩	25	145.95
	⟨set⟩	25	87.52

A.2 Shortest path algorithms results

Table A.2: Shortest path algorithms run time results on all test networks for Figure 5.3

Network	Algorithm	Iterations	Time (seconds)
Anaheim	Bellman-Ford	10	1.20
	Dijkstra ⟨priority_queue⟩	10	0.30
	Dijkstra ⟨set⟩	10	0.62
	Bidirectional Dijkstra	10	0.43
	A* search	10	0.10
	Bidirectional A* search	10	0.28
Barcelona	Bellman-Ford	28	60.00
	Dijkstra ⟨priority_queue⟩	27	13.67
	Dijkstra ⟨set⟩	27	27.36
	Bidirectional Dijkstra	27	20.93
	A* search	28	2.99
	Bidirectional A* search	30	8.02
Winnipeg	Bellman-Ford	129	190.00
	Dijkstra ⟨priority_queue⟩	127	34.89
	Dijkstra ⟨set⟩	127	66.89
	Bidirectional Dijkstra	126	49.95
	A* search	126	10.42
	Bidirectional A* search	127	24.81
Chicago Sketch	Bellman-Ford	25	500.00
	Dijkstra ⟨priority_queue⟩	25	87.52
	Dijkstra ⟨set⟩	25	149.95
	Bidirectional Dijkstra	25	157.75
	A* search	26	18.75
	Bidirectional A* search	26	59.82

A.3 Large network results using random skipping strategy

Table A.3: Run time of A* search and the randomly skipping strategy on Philadelphia and Chicago Regional network for Figure 6.4

	Philadelphia	Chicago Regional
A* search	7.69 hours	33.26 hours
A* search with 50% random skipping	5.75 hours	24.18 hours

References

Bar-Gera, H. (2013), 'Transportation network test problems', `http://www.bgu.ac.il/~bargera/tntp/`.

Bellman, R. (1958), 'On a routing problem', *Quarterly of Applied Mathematics*, 16, 87–90.

Blechmann, T. (2013), 'Boost c++ libraries', `http://www.boost.org/doc/libs/1_53_0/doc/html/heap/data_structures.html`.

Bureau of Public Roads (1964), Traffic Assignment Manual, U.S. Dept. of Commerce, Urban Planning Division, Washington D.C.

Cormen, T. H., Stein, C., Rivest, R. L. and Leiserson, C. E. (2001), *Introduction to Algorithms*, 2nd edn, McGraw-Hill Higher Education.

Dafermos, S. C. and Sparrow, F. T. (1969), *The traffic assignment problem for a general network*, Vol. 73B, Journal of Research of the National Bureau of Standards, pp. 91–118.

Dantzig, G. (1963), *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey.

Dijkstra, E. W. (1959), 'A note on two problems in connexion with graphs', *Numerische Mathematik*, 1(1), 269–271.

Dreyfus, S. E. (1969), 'An appraisal of some shortest-path algorithms', *Operations Research*, 17(3).

Ertl, G. (1998), 'Shortest path calculation in large road networks', *Operations-Research-Spektrum*, 20(1), 15–20.

Florian, M. and Hearn, D. (1995), *Handbooks in Operations Research and Management Science*, Vol. Volume 8, Elsevier, chapter Chapter 6 Network equilibrium models and algorithms, pp. 485–550.

Florian, M. and Hearn, D. W. (2008), Traffic assignment: Equilibrium models, *in* A. Chinchuluun, P. Pardalos, A. Migdalas and L. Pitsoulis, eds, 'Pareto Optimality, Game Theory And Equilibria', Vol. 17, Springer New York, chapter Springer Optimization and Its Applications, pp. 571–592.

Ford, L. R. (1956), 'Network flow theory', Report P-923, The Rand Corporation.

Fredman, M. L., Sedgewick, R., Sleator, D. and Tarjan, R. (1986), 'The pairing heap: A new form of self-adjusting heap', *Algorithmica*, 1(1-4), 111–129.

Fredman, M. L. and Tarjan, R. E. (1987), 'Fibonacci heaps and their uses in improved network optimization algorithms', *J. ACM*, 34(3), 596–615.

Goldberg, A. V. and Harrelson, C. (2005), Computing the shortest path: A search meets graph theory, *in* 'Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms', SODA '05, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, pp. 156–165.

Goldberg, A. V., Harrelson, C., Kaplan, H. and Werneck, R. F. (2006), 'Efficient point-to-point shortest path algorithms', `http://www.cs.princeton.edu/courses/archive/spr06/cos423/Handouts/EPP%20shortest%20path%20algorithms.pdf`.

Goldberg, A. V., Kaplan, H. and Werneck, R. F. (2006), Reach for a*: Efficient point-to-point shortest path algorithms, *in* 'SIAM Workshop on Algorithms Engineering and Experimentation', pp. 129–143.

Goldberg, A. V. and Werneck, R. F. (2005), Computing point-to-point shortest paths from external memory, *in* 'Proceedings of the Seventh Workshop on Algorithm Engineering and Experiments (ALENEX'05)', pp. 26–40.

Hart, P., Nilsson, N. and Raphael, B. (1968), 'A formal basis for the heuristic determination of minimum cost paths', *IEEE Transactions on Systems Science and Cybernetics*, 4(2), 100–107.

Ikeda, T., Hsu, M.-Y., Imai, H., Nishimura, S., Shimoura, H., Hashimoto, T., Tenmoku, K. and Mitoh, K. (1994), A fast algorithm for finding better routes by ai search techniques, *in* 'Vehicle Navigation and Information Systems Conference', pp. 291–296.

Inam, R. (2009), A* algorithm for multicore graphics processors, Master's thesis, Chalmers University of Technology.

Jayakrishnan, R., Tsai, W. K., Prashker, J. and Rajadhyaksha, S. (1994), 'A faster path-based algorithm for traffic assignment', *Transportation Research Record*, 1443, 75–83.

Johnson, D. B. (1975), 'Priority queues with update and finding minimum spanning trees', *Information Processing Letters*, 4(3), 53–57.

Klunder, G. A. and Post, H. N. (2006), 'The shortest path problem on large-scale real-road networks.', *Networks*, 48(4), 182–194.

Koenig, S., Likhachev, M. and Furcy, D. (2004), 'Lifelong planning a*', *Artif. Intell.*, 155(1-2), 93–146.

Moore, E. F. (1959), The shortest path through a maze, *in* 'Proc. Internat. Sympos. Switching Theory 1957, Part II', Harvard Univ. Press, Cambridge, Mass., pp. 285–292.

Nicholson, T. A. J. (1966), 'Finding the shortest route between two points in a network', *The Computer Journal*, 9(3), 275–280.

Pallottino, S. and Scutellà, M. G. (1997), Shortest path algorithms in transportation models: classical and innovative aspects, Technical Report TR-97-06.

Pape, U. (1974), 'Implementation and efficiency of moore-algorithms for the shortest route problem', *Mathematical Programming*, 7(1), 212–222.

Pearson, J. and Guesgen, H. W. (1998), Some experimental results of applying heuristic search to route finding., *in* D. J. Cook, ed., 'FLAIRS Conference', AAAI Press, pp. 394–398.

Perederieieva, O., Ehrgott, M., Wang, J. Y. T. and Raith, A. (2013), 'A computational study of traffic assignment algorithms'.

Pohl, I. (1971), Bi-directional and heuristic search in path problems, PhD thesis, Stanford University, Stanford, California.

Rose, G., Daskin, M. S. and Koppelman, F. S. (1988), 'An examination of convergence error in equilibrium traffic assignment models', *Transportation Research Part B: Methodological*, 22(4), 261–274.

Sheffi, Y. (1985), *Urban Transportation Networks: Equilibrium Analysis With Mathematical Programming Methods*, Prentice Hall.

Sleator, D. D. and Tarjan, R. E. (1986), 'Self adjusting heaps', *SIAM J. Comput.*, 15(1), 52–69.

Vuillemin, J. (1978), 'A data structure for manipulating priority queues', *Commun. ACM*, 21(4), 309–315.

Wardrop, J. (1952), 'Some theoretical aspects of road traffic research', *Proceedings of the Institution of Civil Engineers, Part II*, 1(36), 352–362.

Zhou, Z., Brignone, A. and Clarke, M. (2010), 'Computational study of alternative methods for static traffic equilibrium assignment'.