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 the optimal path for every vehicle in a transportation network. One algorithm for solving TA is the Path Equilibration (PE) algorithm. PE requires solving shortest path repeatedly between every origin and destination pair in the network for a large number of iterations, while the edge costs change between iterations. The aim of this project is to find a faster shortest path algorithm for PE. We implement Dijkstra's algorithm and A* search, their bidirectional versions and 8 different versions of priority queue data structures that improve these algorithms' performance. We develop two strategies for using these algorithms in the iterative environment of PE. The first strategy is to avoid next few numbers of iterations when the shortest path of the previous two iterations is the same. The second strategy is to randomly skip the next shortest path calculation, where we 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 these priority queues, algorithms and strategies and show that A* search algorithm with random skipping strategy has the best performance.

Acknowledgements

I would like to thank my superviosrs, 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 and Corey Kok, for their insightful ideas and kind helps, some of the algorithms would take much longer time to write without them.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Introduction to traffic modelling

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

Congestion lead to major economical losses due to time delays and increase usage of petrol. Congestion cause air pollutions that increase respiratory problems such as asthma, And the exhaust gas exacerbates global warming. Congestion also increase noise pollution and cause frustration, which in turn accelerate traffic accidents. It is important for traffic designers to be able to reduce congestion problems, and eliminate its negative effects. The transportation network can be improved by for example, introduce road tolls to diverge traffic to less congested roads, or educate people to use public transports.

Making improvements to the transportation network tend to be very costly, so an optimal plan is always necessary: use the least amount of investment for the greatest change. In order to make optimal plans for traffic design, different mathematical models have been built to simulate 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, upgrading or adding roads, changing roundabouts to traffic lights or provide better public transports.

The transportation forecasting model has 4 stages (shown in Figure 1.1): trip generation, trip distribution, mode choice and traffic assignment. In the model, each of the next stage can pass information to the previous stages to improve traffic design. 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 best route (e.g. shortest path) that each trip need to travel on (traffic assignment). The traffic assignment problem in the last stage of the forecast model is a very complicated problem. This is because congestion occur as traffic flows are assigned onto the network, and it is very difficult to find an equilibrium situation where everybody in the network find their best route.

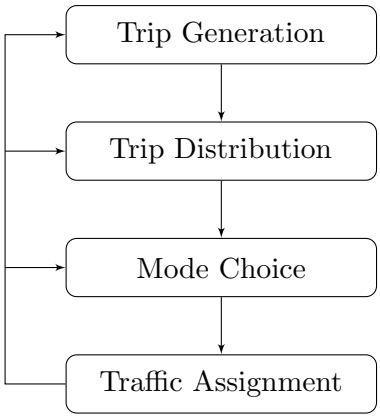


Figure 1.1: Transportation forecasting model

1.2 Purpose of this project

The transportation forecasting model has been implemented in many software for traffic design. One key observation from these software is that, the traffic assignment problem takes days, or even weeks 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 (sometimes there are hundreds of iterations) require 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 the time, but cumulatively the computation is huge. So the purpose of this project is to find a faster algorithm for solving the shortest path problem in an iterative environment. As a result, the traffic assignment will be solved faster for larger and more complicated road networks, and this will allow city designers estimate traffic flows further into the future and make better decisions on road network design.

1.3 Structure of the report

Chapter 2 gives a short description of the traffic assignment problem and presents a specific algorithm (The Path Equilibration Algorithm) that solves it. This algorithm is going to be experimented for this project. Chapter 3 introduces the shortest path problem and presents some of the well established algorithms that solves it very fast. Strategies for solving the shortest path problem faster in an iteration environment is also describe in this chapter. Implementation details are presented in Chapter 4. And Chapter 5 shows the results and their discussions. Finally conclusions 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 happen and travel speed tend to decrease rapidly due to more and more interactions between the travellers and increase in traffic volume. This lead to the problem of travellers wishing to find the fastest route to travel on, meanwhile taking account of congestion as every other traveller is trying to do the same. From the road design point of view, we wish to find a flow pattern in the network with a given travel demand between the origin-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 which is the best for them. On the other hand, we can also solve for traffic flows that are "system optimal", which is characterized by Wardrop's second principle, stating that "the average journey time is minimum".

In this chapter, the network equilibrium model is first stated. Then a particular solution algorithm for solving such model is described. Finally the reason for needing a faster algorithm for solving the shortest path problem is briefly explained.

2.1 The network equilibrium model

In this section, the deterministic symmetric network equilibrium model for solving the user optimal of the traffic assignment is summarised. This model assumes deterministic traffic demands, where the demands are fixed when the traffic assignment problem is getting solved. This model also assumes the network is symmetric, which does not mean the underlying graph of the network is symmetric or bidirectional, it means the change of travel time on any link does

not depend on the change of traffic flows on the other links. These assumptions may not be realistic but they result a set of simple algorithms that are easier to analyse.

Using notations from Florian & 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 a user cost function $s_a(v)$ $(a \in A)$, 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 $i \in I$. 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:

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

$$h_k \geq 0. \tag{2.2}$$

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

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

With constraints 2.1 - 2.4, the user optimal objective is

$$\min S(v) = \sum_{a \in A} \int_0^{v_a} s_a(x) \mathrm{d}x. \tag{2.5}$$

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

$$s_k(v^*) - u_i^* \begin{cases} = 0 & \text{if } h_k^* > 0 \\ \geq 0 & \text{if } h_k^* = 0 \end{cases}, \quad \forall k \in K_i, i \in I, \tag{2.6}$$

where

$$s_k(v) = \sum_{a \in A} \delta_{ak} s_a(v), \quad \forall k \in K \tag{2.7}$$

and

$$u_i = \min_{k \in K_i} [s_k(v)], \quad \forall i \in I. \tag{2.8}$$

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 is under the assumption 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 minimize their own travel times.

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

$$s_a(v_a) = s_f \left(1 + B \left(\frac{v_a}{C_a}\right)^{\alpha}\right) \tag{2.9}$$

where B and α are the parameters for the level of congestion that can can be experienced, and s_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

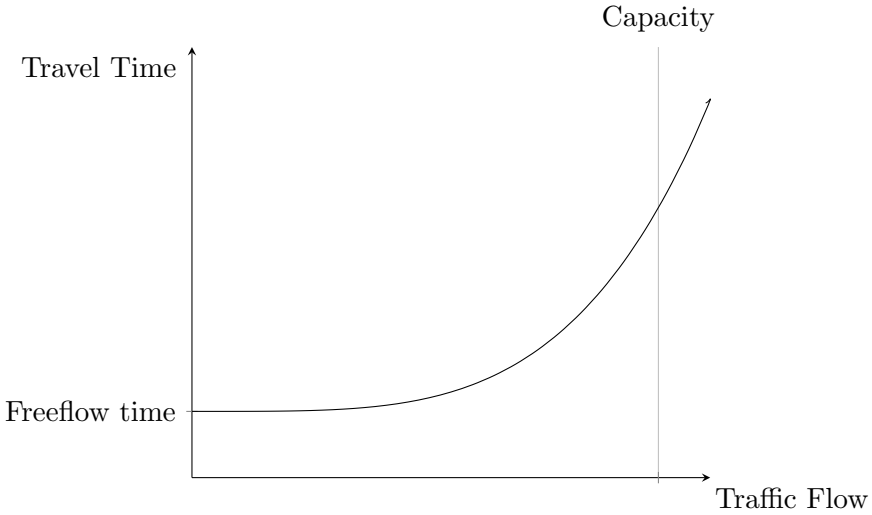


Figure 2.1: Travel time function.

2.2 The path equilibration algorithms

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

For this project, we focus on the path equilibration algorithm. The general approach of the algorithm is equivalent to Gauss-Seidel decomposition (an iterative method for solving a linear system of equations). In a step of the algorithm, path flow h_k between a single O-D pair is solved by keeping the flows of all other O-D pairs fixed. The algorithm iteratively solves each O-D pair sub-problem until all of the path flows cannot be improved.

The sub-problem for solving each O-D pair i is another fixed-demand network equilibrium problem.

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

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

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

where

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

and

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

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

'cyclic decomposition' by O-D pair

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 ith sub-problem (2.10)-(2.14), set $l' = l' + 1$ and return to step 1; otherwise solve the lth sub-problem, update the path flows, set $l' = 0$ and return to step 1.

The path equilibration algorithm for solving 2.10-2.14 obtains the solution by balancing path flows between each O-D pair. One such algorithm, proposed by Dafermos (1971), finds the shortest and longest path and equalizes the flows between them. Let $K_i^+ = \{k \in K_i | h_k > 0\}$ be the set of positive flows. The algorithm for solving each O-D pair i is stated as follows.

Path Equilibration Algorithm

Step 0. Find an initial solution v_a^i; $s_a = s_a(v_a^i + \bar{v}_a)$ and the initial K_i^+.

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

If $(s_{k_2} - s_{k_1}) \leq \epsilon$, go to step 4; otherwise define the direction $d_{k_1} = (h_{k_2} - h_{k_1})$ for path flow k_1 and $d_{k_2} = (h_{k_1} - h_{k_2})$ for path flow k_2

Step 2. Find the step size λ which redistributes the flow $h_{h_1} + h_{k_2}$ between the paths k_1 and k_2 in such a way that their costs become equal, that is, solve

$$\min_{\lambda} \quad \max_{a \in A} \int_0^{v_a^i + \lambda y_a + \bar{v}_a} s_a(x) \mathrm{d}x \tag{2.15}$$

$$\text{s.t.} \quad 0 \leq \lambda \leq \left(\frac{-h_{k_2}}{d_{k_2}} \right), \tag{2.16}$$

$$\text{where} \quad y_a = \delta_{ak_1} d_{k_1} - \delta_{ak_2} d_{k_2}. \tag{2.17}$$

Step 3. Using the λ obtained, update $h_k = h_k + \lambda d_k, k = \{k_1, k_2\}$; $v_a^i = v_a^i + \lambda y_a$; $s_a = s_a(v_a^i + \bar{v}_a)$.

Step 4. Compute the shortest path $\tilde{k}$ with cost $\tilde{s}_k = \min_{k \in K_i} [s_k]$; if $\tilde{s}_k < \min_{k \in K_i^+} [s_k]$, then the path $\tilde{k}$ is added to the set of kept paths, $K_i^+ = K_i^+ \cup \tilde{k}$ and return to step 1; otherwise stop.

In step 0, an all-or-nothing assignment is performed for each of the O-D pairs, where it finds the shortest path and assigns all traffic flows along that path. In step 1 and 2, the algorithm finds the two paths that have the minimum and maximum cost, and balances the flow between them to equalize their cost. These two steps are equivalent of solving the Wardrop equilibrium shown in Equation (2.6)-(2.8). In step 4, the shortest path between the O-D pair is computed and added to the set of used paths for the all-or-nothing assignment and Wardrop equilibrium.

Now if we are given a large network and assume it requires many iterations to find the optimal solution, it can be shown that a huge number of shortest path calculations are needed in step 4 of the path equilibration algorithm. For example given the algorithm takes 20 iterations to solve

for a small network with $100,000$ O-D pairs, it can take more than 6 minutes to solve if each shortest path calculation consumes 0.01 second, And 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 for the traffic assignment problem.

2.3 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 of the problem is convex (any local minimum is also the global minimum), so the convergence of the Gauss-Seidel strategy is ensured (Florian & Hearn 2008). The convergence criterion of traffic assignment algorithms are normally measured by the notion of relative gap. 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

$$\text{RGAP} = \frac{\text{UE} - \text{AON}}{\text{UE}} \tag{2.18}$$

$$= \frac{\sum_{a \in A} v_a s_a - \sum_{i \in I} g_i s_{\tilde{k}_i}}{\sum_{i \in I} g_i s_{\tilde{k}_i}}. \tag{2.19}$$

The AON solution is the sum of travel time $s_{\tilde{k}_i}$ of each traffic demand g_i travelling on their shortest path $\tilde{k}_i$. The UE solution is the sum of travel time over the entire network. These two values are improved from iteration by iteration during the path equilibration algorithm until they become identical, i.e. RGAP converging toward 0. The traffic assignment problem is solve when RGAP is 0, but normally we stop the algorithm at some tolerance such as $1e-6$.

It is worth to notice that the speed of convergence is highly dependent on how small we set the relative gap to be, as well as the size and complexity of the network and number of supply and demand nodes. A smaller relative gap will result more iterations for the traffic assignment algorithms.

Chapter 3

Solving the shortest path problem

Over the years, various algorithms have been developed to address the problem of finding the shortest path. This chapter states notations and definitions for the shortest path problem and discusses the theory of solving it. Algorithms that are applicable for the traffic assignment problem are summarised, including the discussion of their advantages and drawbacks.

3.1 Notations and definitions

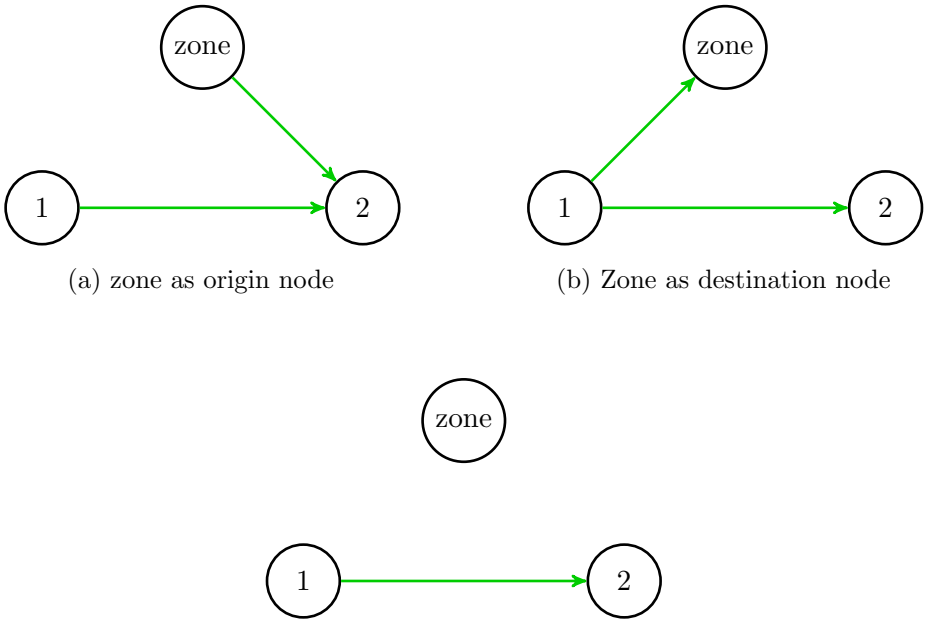
The Shortest Path Problem (SPP) is the problem of finding the shortest path from a given origin to some destination. There are two types of SPP that are going to be analysed in this chapter: a single-source and a point-to-point SPP. More emphasise is going to be put on the point-point-point SPP for the path equilibration algorithm described in the previous chapter.

Now we present the notations mainly borrowed from Cormen et al. (2001) and Klunder & 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 (roads). 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 cost function $c : A \to \mathbb{R}$ which assigns a cost (travel time) to any arc $(u, v) \in A$ depending on traffic flow on that arc. We write the costs of arc (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 edges $(u_1, (u_1, u_2), u_2, \ldots, (u_{k-1}, u_k), u_k)$ 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 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 emanating arcs are allowed. If it is a destination node, then only emerging arcs are allowed. And if it is neither, then no arcs can pass through it.



(a) zone as origin node

(b) Zone as destination node

(c) Zone as neither origin nor destination node

Figure 3.1: Zone node and its allowable arc flows

3.2 Generic shortest path algorithm

A family of algorithms exist for solving the shortest path problem. In this section the generic case for these algorithms are 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 & Post 2006). Each d_v tracks the least distance of the path going from the origin s to an destination v. We denote $d_v = \infty$ if no path has been found. A shortest path 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. Algorithms for solving the SPP generally uses some kind of queue $\mathcal{Q}$ to store the label distances d.

In the label vector, a node is said to be unvisited when $d_u = \infty$. Scanned and still in the queue when $d_u \neq \infty$. Labelled when the node has been retrieved from the queue and its distance label cannot be improved further. If a node is labelled then its distance value is guaranteed to represent the minimal distance from s to t.

In the generic shortest path algorithm, the algorithm continuously finds the node that violets the Bellman's condition (3.1) and updates its distance label with the path that have a shorter distance that connects to it. All shortest paths connecting the origin s to all other nodes in N is 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.

To keep track of the shortest path found so far for node u, we denote p_u the predecessor of node u. 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 it does not have a predecessor.

Algorithm 1 (Klunder & 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 $u \in \mathcal{Q}$ and updates its distance label if the Bellman's condition is violated for all its emanating edges.

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 ≠ zone 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
```

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

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)).

In this algorithm, the distance labels do not get permanently labelled when the next node in the queue is retrieved. Another node may 'correct' this node's distance label again, thus the name label correcting algorithm.

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 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 processed first. This strategy transforms the queue into a first-in-first-out (FIFO) queue.

In order to satisfy the Bellman's condition for all edges, the algorithm has to scan all edges $|N| - 1$ times, resulting a run time of $O(|N||A|)$.

3.4 Dijkstra's algorithm

The classic algorithm for solving the single-source shortest path problem is the label setting algorithm published by 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 its shortest path. In order to achieve label setting, it is assumed that all arc costs are non-negative, and queue 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 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 for storing the scanned nodes. Cormen et al. (2001) suggests the use of min-priority queues. Min-priority queues are a collection of data structures that always serve the element with highest priority. In the shortest path problem, the priority is measured by the distance labels, where smaller distance label have higher priority.

Algorithm 2 shows the use of the min-priority queue in Dijkstra's algorithm. The min-priority queue has 3 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. And the Decrease-Key (line 19) is used for updating the distance label if the node is already in the queue.

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 ≠ zone 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 implemented via an array, a binary min-heap or a binary search tree, where each implementation give 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 the Dijkstra's algorithm can be improved with a binary min-heap if the graph is sufficiently sparse (in particular $A = o(|N|^2/\log(|N|))$. In this implementation, each Insert and Extract-Min operation takes $O(\log(|N|))$ time for each $|N|$, and the Decrease-Key operation takes $O(\log(|N|))$

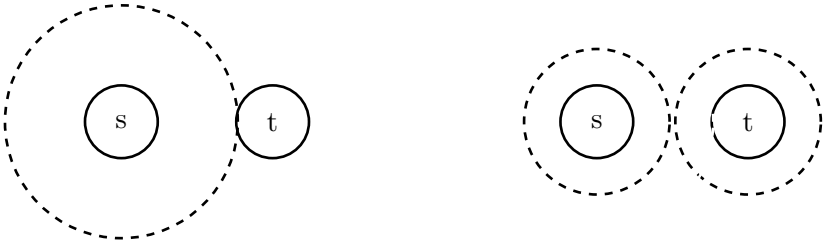
time for each $|A|$. The total running time of Dijkstra's algorithm using min-priority is therefore $O((|N| + |A|) \log(|N|))$, which is an improvement compared to the array implementation.

The running time can be further improved using a Fibonacci heap developed by Fredman & Tarjan (1987). Historically, the development of the Fibonacci heap was motivated by the observation that Dijkstra's algorithm typically does more Decrease-Key operation compared to the Extract-Min operation. In Fibonacci heap, each of the $|N|$ Extract-Min operation takes $O(\log(|N|))$ amortized time 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 also 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 need 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 runs $O((|N| + |A|) \log(|N|))$ in the worst case, which is the same compared to the min-binary heap implementation. The advantage of using a binary search tree is that we do not have to keep track of information about whether a node is in the queue, this is because when performing the Decrease-Key operation, we simply delete the node and add a new node with the updated value.

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 edges 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 a normal Dijkstra's algorithm and its bidirectional version, 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 scan area of label setting and its bidirectional version

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 (see Klunder & Post (2006) for the proof). The correct termination criteria was first designed and implementation by Pohl (1971) based on researches presented by Dantzig (1963), Nicholson (1966) and Dreyfus (1969). The procedure and algorithm is summarised in Klunder & 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 alternatively run on the forward and reverse graph (forward and backward algorithm). The algorithm terminates when some node is permanently labelled in both directions. Once the algorithm has terminated, the correct shortest path is found by looking for a 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 edges 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 in the forward search that are permanently labelled from the origin node, which have corresponding label distances d_v^s. Similarly, $\mathcal{R}^t$ contains contains the set of nodes in the backward search that are permanently labelled from the destination node, which 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 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 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 correct shortest path between s and t.

In recent years, Goldberg & 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 summarized as the following. During the forward and backward search, we maintain an extra variable, μ, to present the length of 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 the 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 shortest path length.

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

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

where 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 is 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 toward 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 toward the destination.

Formally we define the following. Let h_v be the heuristic estimate for the shortest path distance between node v to destination t. We apply Bellman's conditions such that an optimal solution exist, 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 solution can still be achieved under the following conditions. Hart et al. (1968) states that the heuristic estimate function h must be admissible and consistent. In context of road networks that use geographical coordinates and Euclidean distances, admissible means that the heuristic must never over-estimate the distance to the goal. And consistent means that the estimated length of a node reaching its destination must not be greater than the estimated length of its predecessors (i.e. Equation 3.3 and 3.4). This two conditions mean that if the heuristic estimate is the shortest path distance from each node to the destination, and the estimate is smaller than or equal to the actual distance going to that destination, then the optimal shortest path can always be found.

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.

In graphs that are measured by Euclidean distances, the heuristic estimate h_u is the straight Euclidean distance between node u and destination t. But in our traffic assignment problem, geographical coordinates and Euclidean distances can not be used. This is because the length of the arcs are determined by the BPR link cost function, where it determines the travel time on the link based on the traffic flow. 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 the zero-flow travel times.

It is obvious that using travel times from the previous iteration do not work. This is because traffic flows can decrease between iterations. The decreased traffic flows result not admissible travel times, where some of the arc travel times from the previous iteration is now longer than the current travel times. Using these longer travel times from the previous iteration will overestimate

the travel time for the current iteration, which results the shortest path not being able to be calculated.

Another option is to use zero-flow travel times for the heuristic estimates. The estimates can be obtained from computing the shortest path tree for every node where the traffic flow of the entire network set to 0. So for any O-D pair, the corresponding h_u is equivalent to the travel time of the shortest path from node u to its destination. 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 can be guaranteed to be calculable.

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 does not improve worst case is because when $h_u = 0, \forall u \in N$, A* search is equivalent to Dijkstra's algorithm. A* search has been experimented in various situations. In the case of using road networks, Goldberg & Harrelson (2005) concludes that A* search with Euclidean distance estimates does not improve the run time compared to Dijkstra's algorithm when using Euclidean distance estimates, but it is still worth a try to use 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 imagined to be having two ellipsoids extending from the origin and destination respectively. This can be shown in Figure 3.3, where Bidirectional A* search has a smaller search area than the unidirectional version.

(a) A* search (b) bidirectional

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

One may construct the algorithm with the same termination condition described in the bidirectional Dijkstra's algorithm section (Section 3.5), that is stop the algorithm when the two frontiers of the searches meet. But the problem with A* search is that, it does not label the nodes permanently in the order of their distance from the origin (Klunder & Post 2006). In other words, the forward and backward heuristic estimates are no longer consistent, it cause either no solution or having the two frontiers never meet.

The correct strategy for calculating the heuristic estimates and termination criterion is first published by Pohl (1971). The heuristic calculation is later improved by Ikeda et al. (1994).

Ikeda et al. (1994) demonstrates that, two arbitrary feasible functions π_f and π_r are not consistent, but their average is both feasible and consistent. Here $\pi_f(v)$ is the estimate on distance from node v to the destination t in the forward search and $\pi_r(v)$ is the estimate on distance from origin s to node v in the backward search. The new heuristic 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 two constants $\frac{\pi_r(t)}{2}$ and $\frac{\pi_f(s)}{2}$ are added by Goldberg, Harrelson, Kaplan & Werneck (2006) to provide better estimates. These two modified heuristic estimates are now consistent and the frontiers of the two searches is guaranteed to meet. The drawback of this modification is that they now provide worse bounds compared to the original π values, the search area may now be large than the unidirectional A* search.

Goldberg, Harrelson, Kaplan & Werneck (2006) showed and proved a better stopping criterion compared to the one published by Pohl (1971), where they extended the bidirectional Dijkstra's termination criterion, bidirectional A* search now need 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 the nodes at the top (front) of the priority queues).

Bidirectional A* search with Euclidean distance heuristic estimates has been experimented by different academics, e.g. Klunder & Post (2006) and Goldberg & 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.

3.8 Preprocessing algorithms

In the last 2 decades, extensive researches have been done on the idea of speeding up shortest path calculations using pre-calculated data. They include reach-based pruning (Goldberg, Kaplan & Werneck 2006), A* search with landmarks (Goldberg & Harrelson 2005) and Hierarchical search (Ertl 1998, Pearson & Guesgen 1998) etc. These preprocessing techniques generally need to spend quite a long time to pre-calculate the required data for in order to speed up the subsequent shortest path queries.

The next section discusses the A* search with landmarks algorithm, and its drawback when used for the traffic assignment problem.

3.8.1 A* search with landmarks

It is shown in the A* search section (Section 3.6), different methods can be used to obtain the heuristic estimates of the shortest path distance between a node to the destination. The landmarks algorithm developed by Goldberg & 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, where 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:

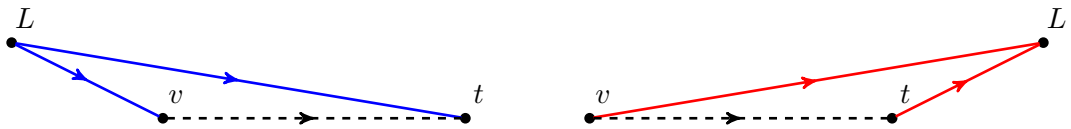
$$\text{dist}(v,t) \geq \text{dist}(L,t) - \text{dist}(L,v) \tag{3.8}$$

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

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 & Harrelson (2005) concluded that it is best to place the landmarks in front of the origins or behind the destinations, i.e. around the edges of the graph.



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

Figure 3.4: Explanatory diagram for triangle inequality

Although the preprocessing stage of the landmarks method can take a very long time, it is proven that the shortest path solving times are significantly faster than A* search using Euclidean distance estimates (see results in Goldberg & Harrelson (2005)). For our traffic assignment's path equilibration algorithm, it is unknown whether using zero-flow travel times can provide better estimates compared to Euclidean distances.

The landmarks method and similar preprocessing techniques have a few drawbacks that need to be considered. The first drawback is that the preprocessing time may be longer than the actual shortest path solving time, so combined combined running time may take longer. What follows this drawback is that the preprocessing stage need to be re-run whenever there is a physical change in the road network, which is undesirable for transportation planning because the road network need to be constantly changed to improve congestion. The second drawback is that both preprocessing and shortest path solving time are heavily dependent on the quantity and placement of the landmarks, which is another optimization problem where different strategies and algorithms need to be experimented.

3.9 Techniques for iterative calculations

So far we have only been dealing with solving the shortest path problem on static graphs. In this section 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.

3.9.1 Lifelong planning A*

A family of algorithms exist for dynamically changing graphs, for example they can be used on graphs that have moving nodes or changing arcs. One particular algorithm that tackles the problem of changing arc costs is the Lifelong Planning A*, 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 Lifelong Planning A* can be used for our problem if we can show the only changes are close to the destination.

3.9.2 Avoiding shortest path calculations

All of the algorithms mentioned so far need to fully calculate the shortest path between every O-D pair in each iteration. It turns out that 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. Two situations can occur if we choose to do so. The first situation is when the shortest path between the previous and current iteration is going to be the same, then we have successfully avoided the calculation and reduced the computational time. The second situation is when they are going to be different, then the path equilibration algorithm is not going to converge for the current iteration, which causes an increase in total number of iterations and computational time.

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.

The first strategy is as follows. For each O-D pair, if its shortest path did not change in the last 2 iterations, then we can delay the 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 calculations that are larger than the total number of iterations, then there will be excessive iterations resulting wasted time. And if we skip too few iterations, then there may not be any impact on the computational time.

The other strategy is to skip shortest path calculations randomly. That is, when it comes to calculate the shortest path for every O-D pair, we generate an random number and decide whether to do 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 unpredictable run times.

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 that provide better performance are discussed.

4.1 Traffic assignment implementation

The path equilibration algorithm and other algorithms for solving the traffic assignment problem has already been implemented by Olga Perederieieva, the co-co-supervisor of this project. The algorithms are implemented in the C++ programming language, where the language has a superior run time performance compared to the others.

The current implementation of the path equilibration algorithm uses Bellman-Ford-More algorithm for its point-to-point shortest path calculations. When solving the traffic assignment problem, the algorithm spends most of its time computing shortest paths. Time is also spent on other parts of the algorithm, the majority of it is dedicated to the convergence check step mentioned in Section 2.3, where the algorithm require to run the Bellman-Ford-More 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 the most 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 emanating arcs of that 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 the 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.

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

First we examine the 6 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[1].

Table 4.1: C++ Boost Heap Implementations with Comparison of Amortized Complexity (Blechmann 2013)

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

We are interested in using Boost library Heaps rather than the C++ standard library Heap is due to one reason: the Decrease-Key (or Increase-Key) operation. The operation is used to change the distance labels in the priority queue when a node is scanned and updated during the shortest path calculations. The Decrease-Key operation is used for min-heap trees (minimum value on top) and the Increase-key operation is used for max-heap trees (maximum value on top). In Dijkstra's algorithm, nodes are often scanned multiple times in the label updating step,

[1] Amortized time: how much time is taken in total when an operation is repeated a millions for example, with different inputs. Run time is averaged out between the worst-case and the best-case.

so instead of adding the node again into the Heap tree with a difference value, we can use the Decrease-Key operation. The advantage of using this operation is that we can reduce the size of the Heap tree, which results a performance improvement as it takes less time to search and insert nodes to smaller heap trees.

In table 4.1, we observe 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. C++ Boost Library Heaps are implemented as max-heaps, which means in order to use the Fibonacci $O(1)$ Increase-Key operation, all of the distance labels need to be negated when inserted into the heap.

Next we examine ⟨priority_queue⟩ from the C++ standard library. This implementation also provides $O(\log(N))$ push and pop operation, but it does not have the decrease-key operation nor does it have a way to change node values somewhere else in the tree. So when solving the shortest path problem, the priority queue is going to have many nodes that have the same 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 labelled first, so once that node is labelled, all the other same node will simply be ignored in the algorithm.

Finally we examine ⟨set⟩ from the C++ standard library. A *set* is a data structure used to store unique elements that follow a specific order. In the C++ standard 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 Decrease-key operation whenever a node need 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 label always come first. The advantage of ⟨set⟩ compared to ⟨priority_queue⟩ is that nodes can be removed from anywhere in the data structure, so ⟨set⟩ may be faster than ⟨priority_queue⟩ because the performance of their operations are heavily dependent by the number of nodes in the data structures.

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. The algorithms are written in C++, compiled by the g++ compiler using the '-O3' optimization flag.

All timed results are measured for a complete run of the traffic assignment. All solutions will use $1e^{-6}$ as the relative gap for the stopped criterion.

5.1 Road networks

Table 5.1 shows the road network used for this project, the network detaisl 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. ChicagoSketch is a medium sized network that is part of the ChicagoRegional network. Terrassa is a small network that suffers high congestion, which takes more than 400 iterations to solve. Philadelphia and ChicagoRegional are two large networks that have over a million number of O-D pairs.

Table 5.1: Network Problem Data

Network	Nodes	Arcs	Zones	O-D pairs	Iterations
Anaheim	416	914	38	1,406	10
Barcelona	1,020	2,522	110	7,922	27
Winnipeg	1,052	2,836	147	4,344	126
ChicagoSketch	933	2,950	387	93,135	25
Terrassa	1,609	3,264	55	2,215	415
Philadelphia	13,389	40,004	16,525	1,149,795	81
ChicagoRegional	12,982	39,018	16,790	2,296,227	152

5.2 Results on priority queues

In this section we experiment with all the priority queues discussed in Section 4.3. The priority queue are used in Dijkstra's algorithm on the Winnipeg and ChicagoSketch network. The results are shown in Figure 5.1 and 5.2. The extract numerical results can be found in 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 in the 6 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 comparably a lot slower than ⟨priority_queue⟩.

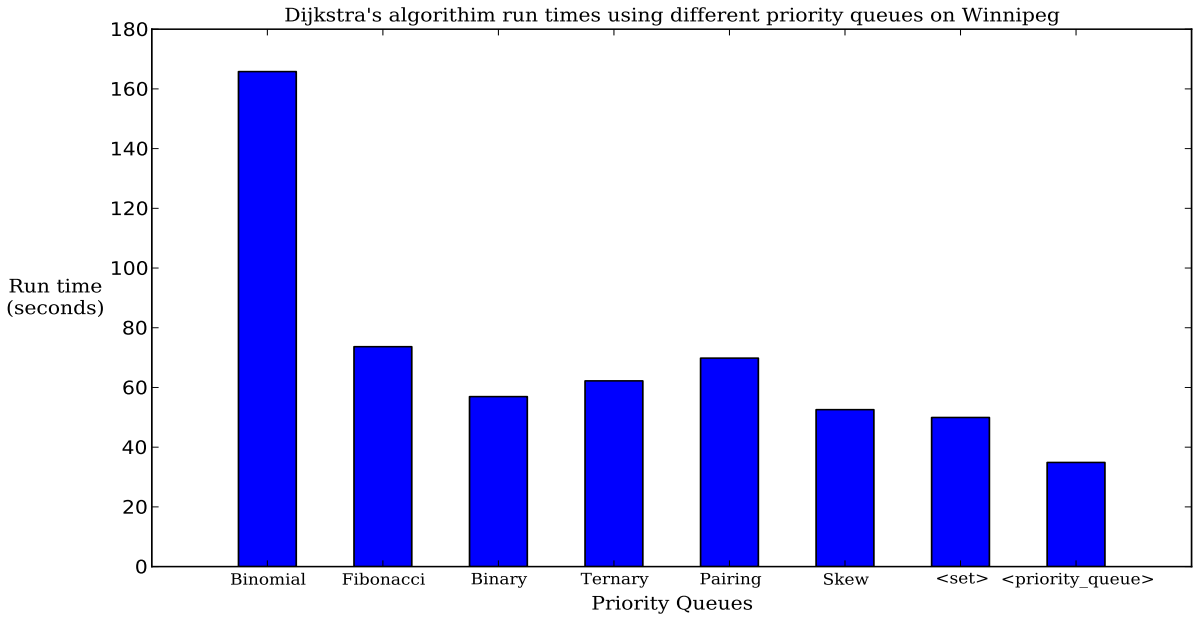


Figure 5.1: Dijkstra's algorithm run times using different priority queues on Winnipeg

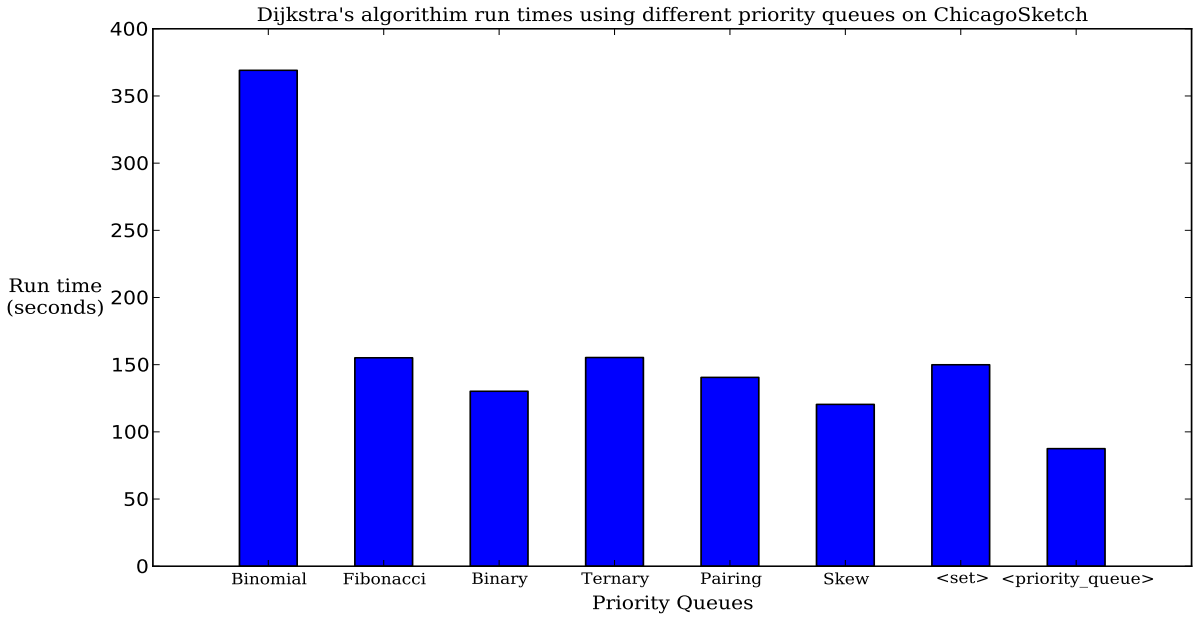


Figure 5.2: Dijkstra's algorithm run times using different priority queues on ChicagoSketch

5.3 Results on shortest path algorithms

Now we use ⟨priority_queue⟩ from the C++ standard template library and implement Dijkstra's algorithms and A* search, as well as their bidirectional versions. Figure 5.3 shows the performance of the mentioned algorithms on the Anaheim, Barcelona, Winnipeg and ChicagoSketch networks (please see Appendix B for exact numerical results). Bellman-Ford-More algorithm is also run to show the base line for the implemented algorithms. The networks are spaced out on the horizontal axis to show their relative sizes.

Bellman-Ford-More algorithm has the worst performance while the 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.

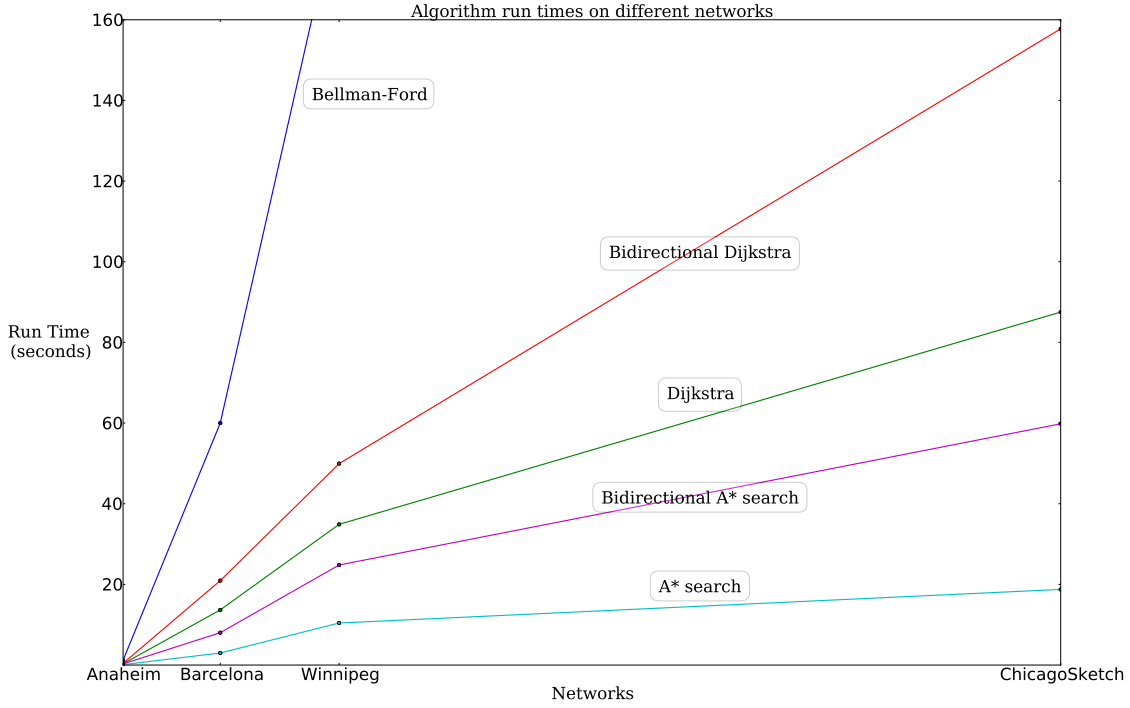


Figure 5.3: Run time performances of different algorithms on different networks

5.4 Results on avoiding shortest path calculations

In this section, we consider avoiding shortest path calculations in the iterative path equilibration algorithm discussed in Section 3.9.2. A* search is used to generate results for this section.

As mentioned in Section 3.9.2, in order to apply shortest path avoiding strategies, we need to first prove that most of the shortest paths do not change often.

Figure 5.4 shows that on the ChicagoSketch network, out of 26 iterations, the percentage of O-D pairs that changed their shortest path once, twice, three times etc. The figure shows that 60% of O-D pairs have not changed their shortest path after the initial iteration, and 16% of 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 on the paths that do not change between iterations.

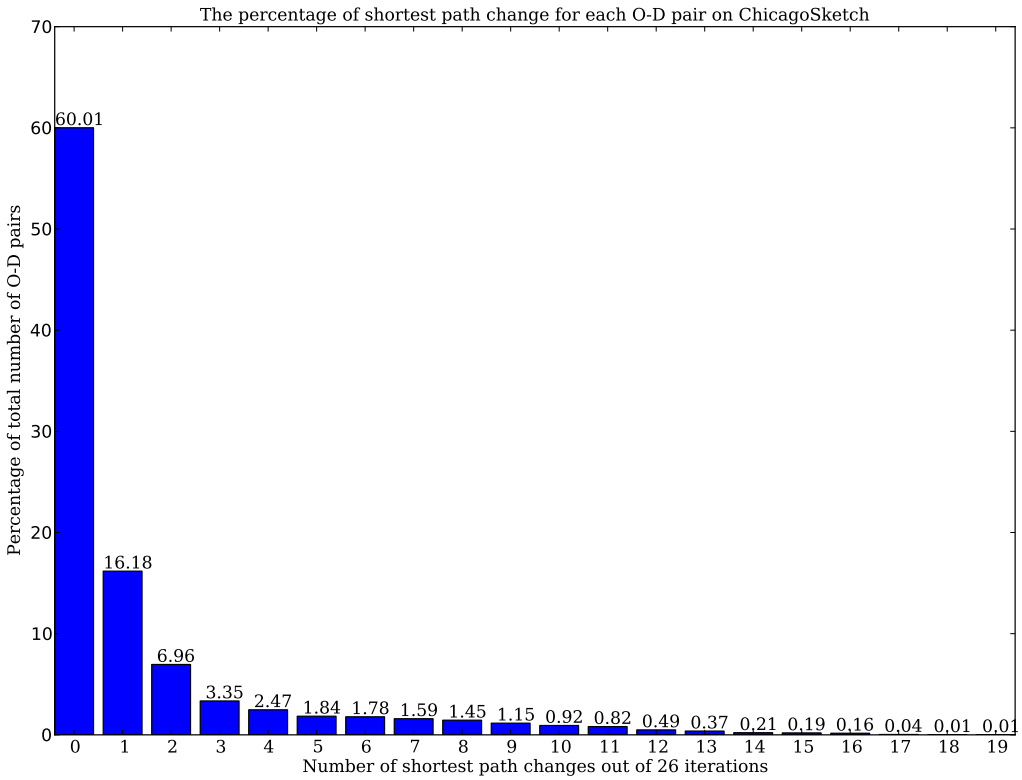
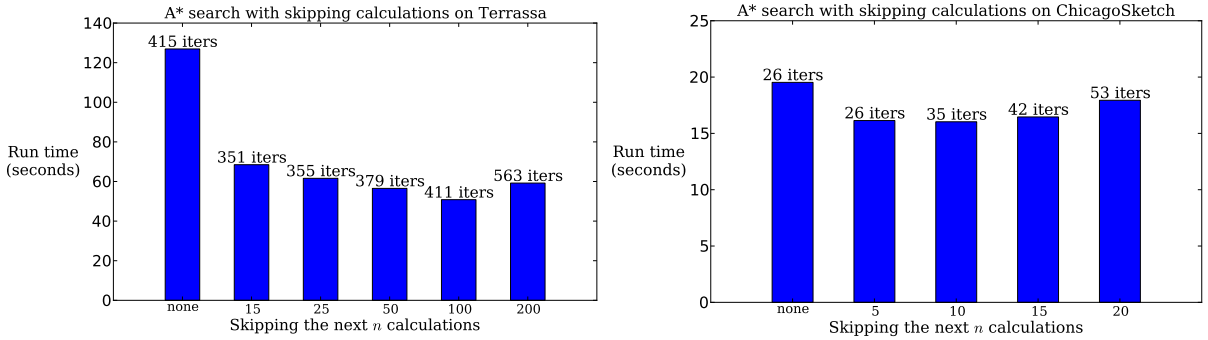


Figure 5.4: The percentage of shortest path change for each O-D pair out of 26 iterations for ChicagoSketch

We now present the strategy of avoiding a pre-defined number of shortest path calculations for each O-D pair if the previous two iterations are the same. The results are shown in Figure 5.5, where the strategy is experimented on the Terrassa and ChicagoSketch network. On the Terrassa network, we choose to avoid the next 15, 25, 50, 100 and 200 iterations of shortest path calculations if the previous two are the same. And on the ChicagoSketch network, we choose to avoid the next 5, 10, 15 and 20 iterations. The strategy worked well on the Terrassa network, where run time is decreased by half for all of the chosen number of avoiding iterations. Due to small size of the network, the run time is not affected even though the total number of iterations increased to 563 when calculations are avoided by 200 iterations. The strategy also worked well on the ChicagoSketch 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 a increase in total number of iterations.

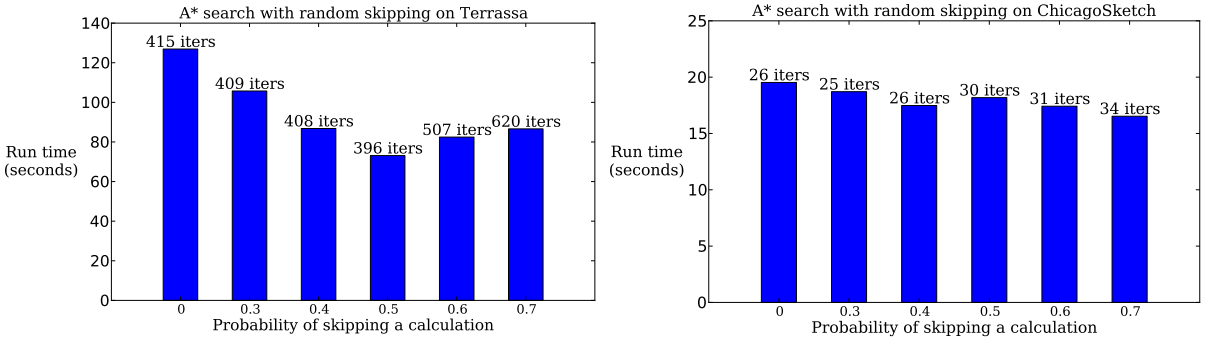


(a) Terrassa network

(b) ChicagoSketch network

Figure 5.5: 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 5.6a shows the results on the Terrassa network, probabilities of 0.3, 0.4, 0.5, 0.6 and 0.7 for skipping the next calculation. The strategy reduced the run time quite significantly for all probabilities, especially 0.5. Figure 5.6b shows the same strategy on the ChicagoSketch network. This time all run times are reduced slightly, with 0.4 having the most reduction.



(a) Terrassa network

(b) ChicagoSketch network

Figure 5.6: Run time for skipping shortest path calculations randomly

The random strategy can be used before knowing the total number of iterations the path equilibration algorithm going to produce, And because only small networks have been tested so far, we now experiment the random strategy on the Philadelphia and ChicagoRegional network, where they have over a million number of O-D pairs. The run time comparisons are shown in Table 5.2, where 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 ChicagoRegional respectively.

Table 5.2: Run time of A* search and the randomly skipping strategy on Philadelphia and ChicagoRegional network

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

Chapter 6

Discussion of results

The previous chapter presented results that indicated A* search using priority queue from the C++ standard template library has the best performance. The results also shows that shortest path calculations can be speed up further when it is in an iterative environment. In this chapter we discuss the details behind the mentioned results, including why some of the algorithms that should have performed better but failed to do so.

6.1 Priority queues

The priority queue implementation results showed that all of the heap implementations from the Boost library is 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 almost 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, where as the Boost library uses pointers to keep track of the nodes. Due to computer cache coherence, it is known that accessing data from a nearby 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 are pre-fetched into the cache in a hope they will be accessed in a short period of time). In the shortest path algorithms, the Heap tree need to be searched over and over 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 are allocated in random locations when nodes are inserted.

6.1.1 Fibonacci heap

Here we discuss the reason behind Fibonacci heap not performing well despite its $O(1)$ amortized time Decrease-Key operation. As described in the two priority queue sections (Section 3.4.1 and

4.3), the Decrease-Key operation is used to change the distance label of a node when the node is already in the 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 Decrease-Key operation is only used frequent 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.

Now we confirm our graph is indeed not dense and the Decrease-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 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 Decrease-Key on any node is around 1 to 5 percent.

6.2 Bidirectional algorithms

In this section we investigate the reason for the worse performances of the bidirectional algorithms.

First we examine whether search areas of the algorithms are what is expected. Figure 6.1 shows the shortest path trees of the point-to-point algorithms, where the origin and destination node is placed on the opposite side of the ChicagoSketch network. It can be seen that Dijkstra's algorithm scans the entire network. Bidirectional Dijkstra scans almost the entire network with some nodes on the side left out. 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 is shown further in Figure 6.2, where the origin and destination is placed close to each other. It is shown that both Dijkstra's algorithm and its bidirectional version scan almost half of the graph, where bidirectional search scans less. A* and its bidirectional search 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 Dijkstra's algorithms match what is expected, but not the run times. The reason for reduction in run time is due to our implementations. In both forward and backward search, the current shortest path μ need to be updated every time a node is scanned, and the stopping criterion need 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 their predecessors and concatenate them together for the full shortest. So it is concluded that these

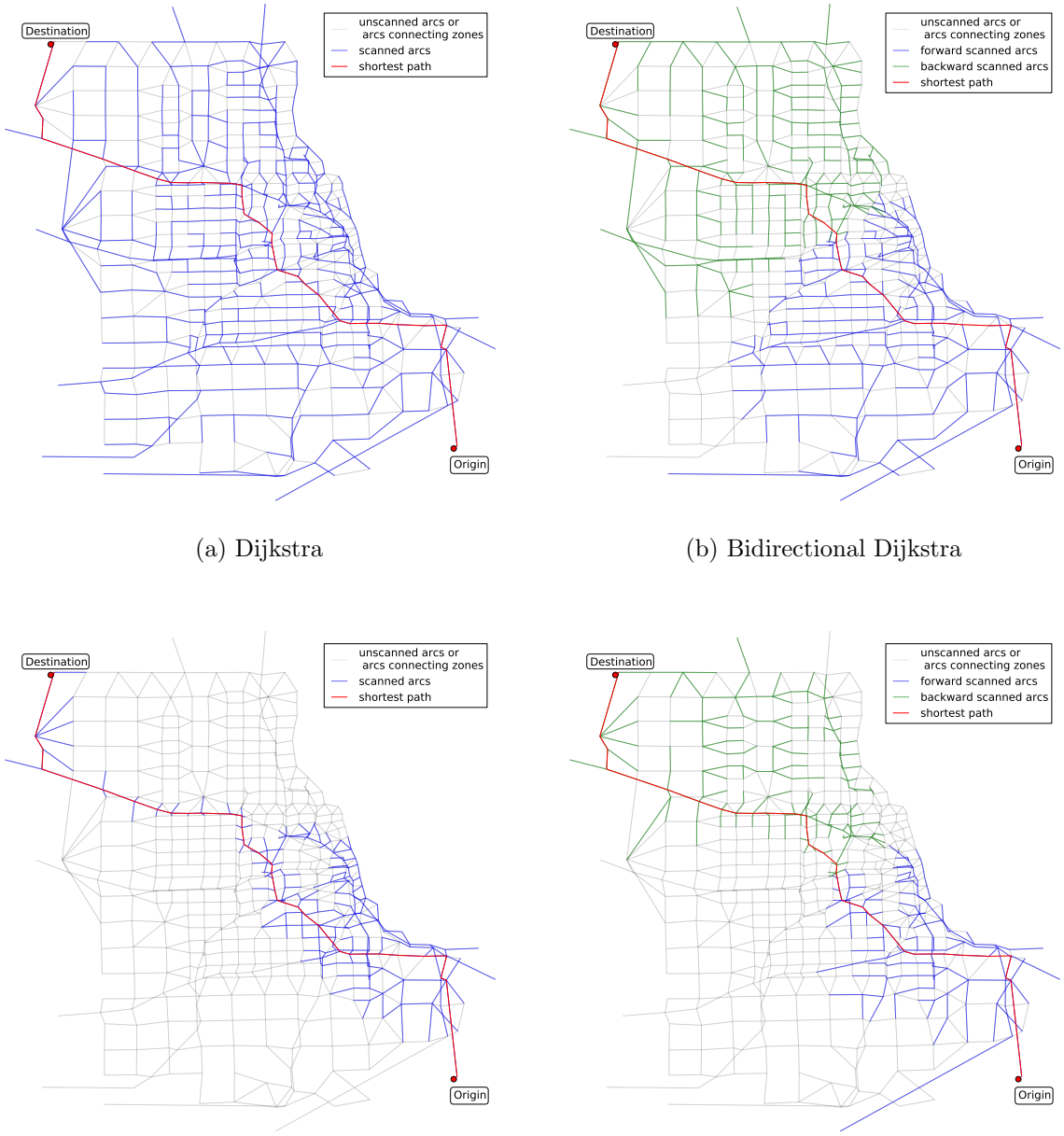
addition computations slowed down the run times.

For A* search, the bidirectional version scans more nodes than the unidirectional version. And since the bidirectional version has similar stopping criterion compared to the bidirectional Dijkstra's algorithm, it is easy to understand why bidirectional A* performed worse.

6.3 A* search with landmarks

A* search with landmarks algorithm is not implemented for this project. This is due to two reasons. The first reason is due to its sophisticated graph dependent implementation, where we need to either manually or dynamically decide the number of landmarks and their placement locations. The second reason is due to its high chance of not being able to work, as the algorithm is aimed at geographic node locations and Euclidean distances, not our travel times that based on traffic flows.

Preprocessing algorithms are not really practical for the traffic assignment problem and the transportation forecasting model. This is because normally these algorithms tend to spend much longer time than just running its standard version. And since the purpose of the traffic assignment is to modify the network and resolve the problem to see its affect on congestion, so it is not ideal to rerun the preprocessing algorithm every time the network is modified.



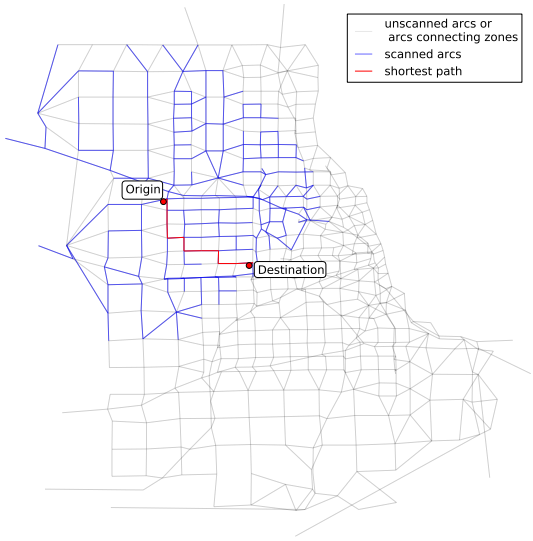
(a) Dijkstra

(b) Bidirectional Dijkstra

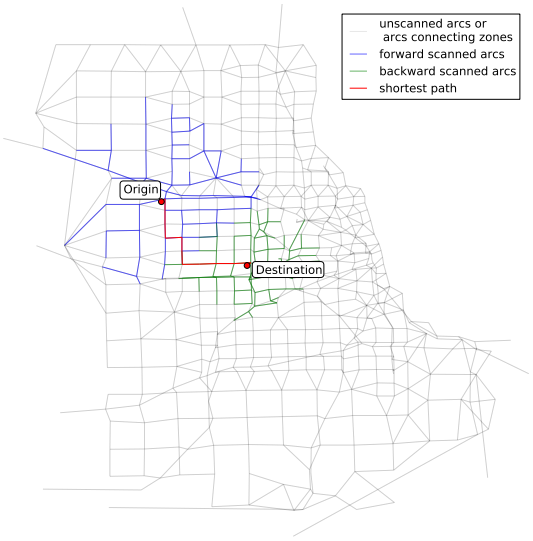
(c) A* Search

(d) Bidirectional A* Search

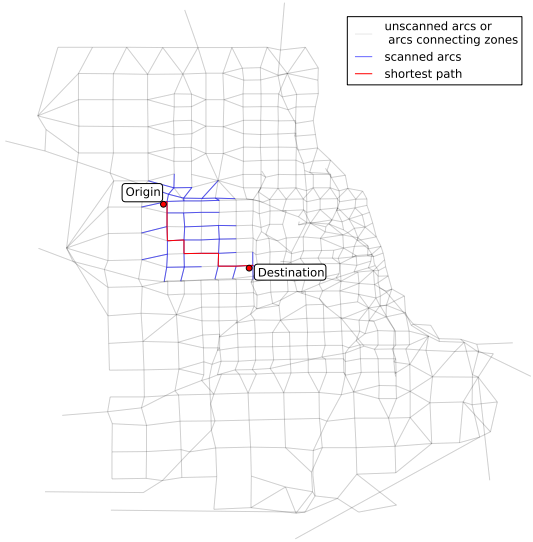
Figure 6.1: Shortest path tree between two distant nodes in the ChicagoSketch Network



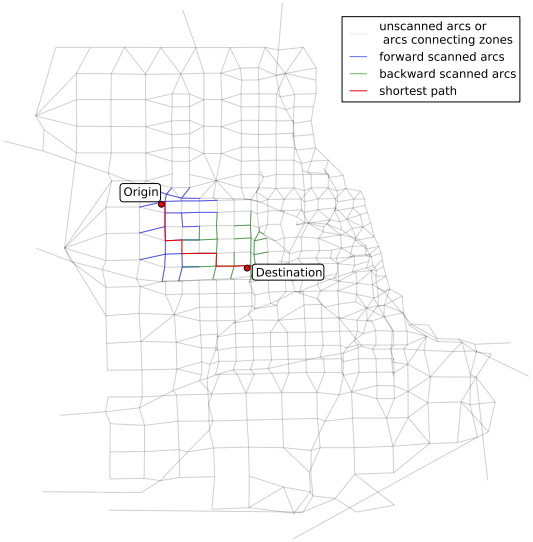
(a) Dijkstra



(b) Bidirectional Dijkstra



(c) A* Search



(d) Bidirectional A* Search

Figure 6.2: Shortest path tree between two close nodes in the ChicagoSketch Network

Chapter 7

Summary and conclusions

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. 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 number of 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, where we hope for 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 preprocessing methods such as A* search with landmarks algorithm.

We now conclude this project with the following points:

- The A* search algorithm using zero-flow travel times as heuristic estimates has the best performance.
- Bidirectional versions of Dijkstra's algorithm and the A* search have worse performances. Bidirectional Dijkstra is worse because its need to check the stopping criterion at each step. Bidirectional A* is worse because its search area is larger than unidirectional A*.
- The strategy of avoiding next few iterations of shortest path calculations is not viable as it is difficult to find the best number of iterations to skip.
- The strategy of avoiding 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.
- A* search with landmarks is not applicable unless we decide to concentrate on a specific road network, for example the Auckland Regional Transport model.

7.1 Future work

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 GPU) run on multi processors using many threads concurrently. The main modification of the algorithm is that instead of sequentially update all emanating arcs from the labelled node, we update them in parallel using multiple threads.

A* search with landmarks algorithm can be investigated if we decide to concentrate on a specific road network. How many landmarks need to be used and where to place them need to be experimented in order to find the best combination of preprocessing run time and actual query time.

Appendix A

Priority queue results

Table A.1: Priority queues run time results on Winnipeg and ChicagoSketch network

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
ChicagoSketch	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

Appendix B

Shortest path algorithms results

Table B.1: Shorest path algorithms run time results on all test networks

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
ChicagoSketch	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

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. & Leiserson, C. E. (2001), *Introduction to Algorithms*, 2nd edn, McGraw-Hill Higher Education.

Dafermos, S. C. (1971), 'An extended traffic assignment model with applications to two-way traffic', *Transportation Science*, 5(4), 366–389.

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. & 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. & Hearn, D. (2008), Traffic assignment: Equilibrium models, *in* A. Chinchuluun, P. Pardalos, A. Migdalas & 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. & Tarjan, R. E. (1987), 'Fibonacci heaps and their uses in improved network optimization algorithms', *J. ACM*, 34(3), 596–615.

Goldberg, A. V. & 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. & 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. & 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. & 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. & 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. & 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.

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

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

Pearson, J. & 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.

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

Rose, G., Daskin, M. S. & 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.

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