Abstract

Acknowledgement

I acknowledge ...

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

CHECK LaTeXLOG!!

talk about things that doesn't work as well

when do we use italics?

italic all the "Names"

check sp time only vs total time

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 work place and residence twice a day, and buy goods from shops where the goods need to be delivered across the city. Meanwhile transportation networks expand and improve constantly to cater people's demand for an efficient transportation network, the rate of improvement does not confront with the rate of population increase. As a result, many obstacles caused by congestion have arisen.

Congestion lead to major economical losses due to time delays and increase usage of petrol. Congestion also cause air pollutions which increase respiratory problems such as asthma, and the exhaust gas exacerbates global warming. It also increases noise pollution and cause frustration, which in turn accelerates accidents. It is important for traffic designers to be able to reduce congestion problems, and eliminate the negative effects of congestion. This include introducing road tolls to diverge traffic to less congested roads, or educating people to use public transportation instead of travelling by car.

Since traffics improvements and expansions tend to be very costly, it is always necessary to make an optimal plan: use the least amount of investment for the greatest change. In order to

make the optimal plan for traffic design, different mathematical models have been built in the past 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, e.g. upgrading new roads, changing roundabouts to traffic lights or adding new public transports.

The transportation forecasting model is divided into 4 steps (Figure 1.1): trip generation, trip distribution, mode choice and traffic assignment. In summary, data about traffic demand is collected and the model generates origins and destinations for travellers in the road network (trip generation), it then calculates the number of trips that are required for between each origin and destination (trip distribution), which transportation method should be used for each trip is then decided (mode choice), and finally it decides the shortest path for each trip to take (traffic assignment).

The traffic assignment (TA) problem in the last stage of the forecast model is a very complicated problem, this is because when traffic is assigned onto the network, congestion occur and it is very difficult to find an equilibrium situation where everybody in the network finds their shortest path. Methods for solving the traffic assignment are mostly iterative, where shortest path between each origin and destination pair in the network need to be solved. This requirement lead to traffic assignment algorithms that spend most of their computational time on finding the shortest path.

1.2 Purpose of this Project

There exist different algorithms for solving the traffic assignment problem, one particular algorithm called the path equilibration method requires the calculation of shortest path between a specific origin and destination namely the point to point shortest path problem. Thus the aim of this project is to find the fastest algorithm which solves the point to point shortest path algorithm. As a result, traffic assignment would be solved faster, and larger and more complicated road

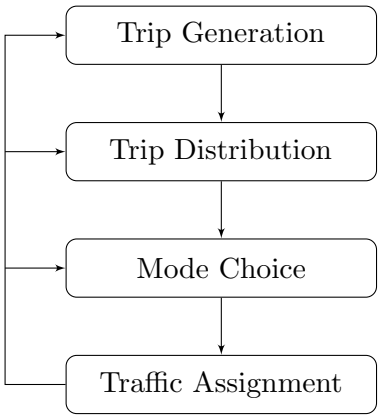


Figure 1.1: Transportation forecasting model

networks would able to be tested in a shorter amount of time, which allows city designers estimate traffic flows further into the future and make better decisions on road network design.

1.3 Structure of the Report

Chapter 3 discusses the theory of finding the shortest path, and presents the description, run time analysis and pseudocode for each algorithm mentioned in the project aims. Chapter 4 presents the specific implementation details. Chapter 5 shows results.

Chapter 5 discussion chapter 6 conclusion ...

Chapter 2

The Traffic Assignment Problem

In a transportation network, every traveller wish to travel between different pairs of origins and destinations. As travellers start travelling in the network, congestion happen as traffic volume increase. The travelling speed on a given road tend to decrease more rapidly as traffic increase, due to more and more interactions between the travellers. 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.

Traffic equilibrium models are commonly in use to solve the Traffic Assignment Problem. 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 solve for traffic flows that are "system optimal", which are characterized by Wardrop's second principle, which states the "the average journey time is minimum".

In this chapter, the network equilibrium model is first stated. Then one particular solution algorithm for solving such model is described. Finally the shortest path problem which arise from the solution algorithm 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 from Florian & Hearn (1995, 2008). The model assumes deterministic traffic demands, where they are fixed during the traffic assignment process. The model also assumes the network is symmetric. This is not assuming the underlying graph of

the network is symmetric or bidirectional. But it means that the change on travel time of any link does not depend on the change in traffic flows of any other links. These assumptions may not be realistic but they result a set of simple algorithms that are easier for analysis.

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.

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

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

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

The corresponding link flows v_a are given by:

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

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 user equilibrium condition (Wardrop 1952) is applied.

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

where

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

and

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

To elaborate, the principle 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. Further more, the principle 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.

ref

To model congestion effects, the Bureau of Public Road link cost function $t_a(v_a)$ is used for the travel time on link $a \in A$. The function is given by:

$$t_a(v_a) = t_f \left(1 + 0.15 \left(\frac{v_a}{C_a}\right)^4\right) \tag{2.9}$$

where t_f and C_a are the free-flow travel time and link capacity. This cost function only depends on the link flow, and is strictly monotonic, continuous and differentiable, An example of this link cost function is shown in Figure 2.1

add in capacity line

2.2 Path Equilibration Algorithms

The symmetric network equilibrium model is equivalent of 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, and path equilibration algorithm.

ref these methods

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

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

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

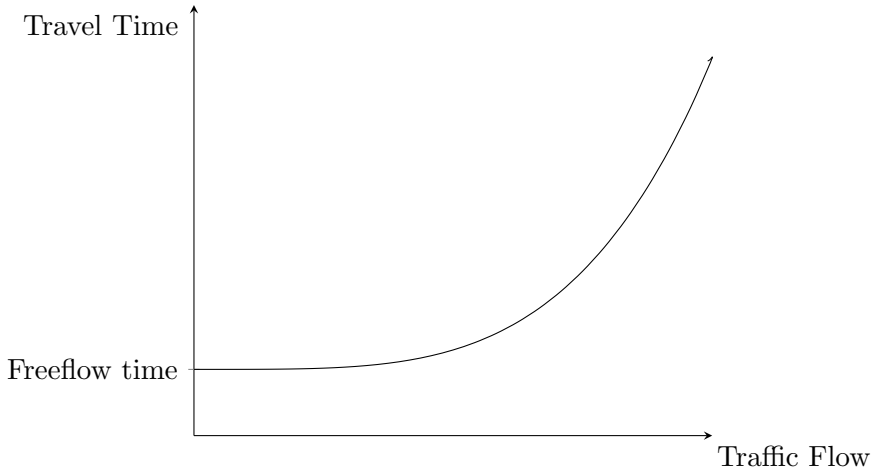


Figure 2.1: Travel time function.

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.

> *Step 0.* Given initial solution, set $l = 0$, $l' = 0$.
> *Step 1.* If $l' = |I|$, stop; otherwise set $l = l \mathrm{mod} |I| + 1$ 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 flows, set $l' = 0$ and return to step 1.

end The objective function of the problem is convex (any local minimum is also the global minimum) so the convergence of the Gauss-Seidel strategy can be ensured.

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.

> *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

$$\min_{\lambda} \quad \max_{a \in A} \int_0^{y_a} s_a(x)\mathrm{d}x \tag{2.15}$$

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

$$y_a = v_a^i + (\delta_{ak_1} d_{k_1} - \delta_{ak_2} d_{k_2})\lambda + \bar{v}_a. \tag{2.17}$$

Step 3. Using the λ obtained, update $h_k = h_k + \lambda d_k, k = \{k_1, k_2\}; v_a^i = v_a^i + (\delta_{ak_1} d_{k_1} - \delta_{ak_2} d_{k_2})\lambda; s_a = s_a(v_a^i + \bar{v}_a$.
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. 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.

Given a large network and assume many iterations of the algorithm are required to find the optimal solution, it can be shown that the shortest path calculation in step 4 would require a huge amount of computational time. For example if it takes 20 iterations to solve for a small network with $100,000$ O-D pairs, and each shortest path calculation takes 0.01 second, it would still take more than 6 minutes to solve. Thus for the rest of the report, we will investigate and find a faster shortest path algorithm for the traffic assignment problem.

2.3 Convergence criterion

For completeness, the convergence criterion for the traffic assignment is discussed in this section. The convergence criterion of traffic assignment algorithms are normally measured by the notion of relative gap. The relative gap is a measure of how close the current traffic assignment solution is to the user equilibrium solution.

"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."Rose et al. (1988)

"The "relative gap" is the aforementioned difference between the cost of the current UE solution and the cost of the AON solution divided by the cost of the current UE solution. This is a fairly sensitive measure of convergence and is superior to many other stopping criteria such as simple functions of the differences between assignment iterates (Rose et al., 1985)" Slavin et al. (2006)

In our path equilibrium algorithm, the relative gap (RGAP) is computed as

$$\text{RGAP} = 1 - \frac{\text{Minimum Travel Time}}{\text{Total Total Time}} \qquad = \frac{\sum x_{UE} \cdot c(x_{UE}) - \sum x_{AON} \cdot c(x_{UE})}{\sum x_{UE} \cdot c(x_{UE})} \tag{2.18}$$

where

$$\text{Minimum Travel Time} = \sum_{i \in I} g_i t_{\tilde{k}_i} \tag{2.19}$$

$$\text{Total Travel Time} = \sum_{a \in A} v_a t_a \tag{2.20}$$

The Minimum Travel Time is the sum of travel time $t_{\tilde{k}_i}$ for each user g_i travelling on their shortest path $\tilde{k}_i$, which is the current user equilibrium solution. The Total Travel Time is the sum of travel time the network is experiencing in the current iteration, which is obviously different from the user equilibrium solution because travel flows are not on their shortest path yet. The traffic solution converges when the Minimum Travel Time and the Total Travel Time becomes the same, resulting the relative gap converging to 0.

In conclusion, the speed of convergence is highly depend 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 for solving it. Algorithms that are applicable for road networks 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 Frank-Wolfe algorithm in the TA involves solving the single-source SPP by finding shortest path going from one origin to every other destinations of the network. The Path Equilibration method in the TA Solving the point to point SPP solves from one origin to a specific destination and is used in the Path Equilibration method.

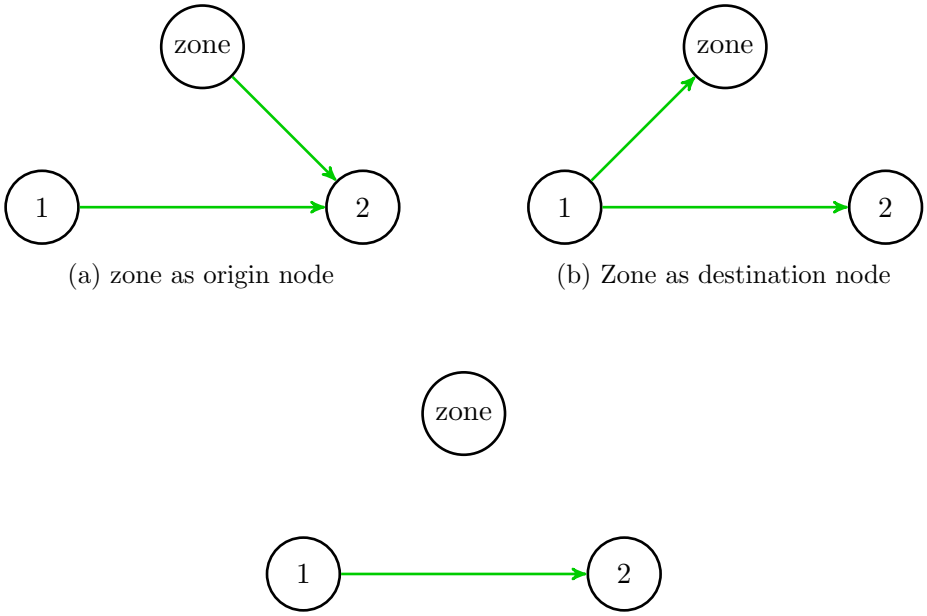
When solving SPP for a normal road network, different measurements such as distance and travel exist for the road length. But in traffic assignment, the road length is measured in a non-decreasing travel time function, which encapsulates information such as traffic flow, road capacity and travel speed. This travel time function (Figure 2.1) is always non-negative so taking advantage of this helps the selection of algorithms that uses this property.

Here we present the notations mainly borrowed from Cormen et al. (2001) and Klunder & Post (2006), we denote $G = (V, E)$ a weighted, directed graph, where V is the set of nodes (origins, destinations, and intersections) and E the set of edges (roads). We say E is a subset of the set $\{(u, v) \mid u, v \in V\}$ of all ordered pairs of nodes. We denote the link cost function $c : E \to \mathbb{R}$ which assigns a cost (travel time) to any arc $(u, v) \in E$ 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 $(u_i, u_{i+1}) \in E$ 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 a transportation network, the origins and destinations are often called centroids or zones. They are used for generating trip demands and supplies and hold information such as household income and employment information. These information helps to understand trips that are produced and attracted within the zone. The zones are conceptual nodes in the network and are untravellable, which means a path between two zone nodes must not contain another zone node. Figure 3.1 demonstrates how a zone node behaves under different conditions.



(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 exists for solving SPP with directed non-negative length edges. In this section we describe the generic case for these algorithms, the generic shortest path algorithm (GSP).

This family of algorithms aims at finding a vector $(d_1, d_2, \ldots d_v)$ of distance labels and its corresponding shortest path (Klunder & Post 2006). Each d_v keeps the least distance of any path going from s to v, $d_v = \infty$ if no path has been found. A shortest path is optimal when it

satisfies the following conditions:

$$d_v \leq d_u + c_{uv}, \quad \forall (u,v) \in E, \tag{3.1}$$
$$d_v = d_u + c_{uv}, \quad \forall (u,v) \in P. \tag{3.2}$$

The inequalities (3.1) are called Bellman's condition (Bellman 1958). In other words, we wish to find a label vector d which satisfies Bellman's condition for all of the vertices in the graph. To maintain the label vector, the algorithm uses a queue $\mathcal{Q}$ to store the label distances.

In the label vector, a node is said to be unvisited when $d_u = \infty$, scanned when $d_u \neq \infty$ and is still in the queue, and labelled when the node has been retrieved from the queue and its distance label cannot be updated 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, we start by putting the origin node in the queue, and then iteratively find the arc that violates the Bellman's condition (i.e., $d_v > d_u + c_{uv}$). Distance labels are set to a value which satisfies condition (3.1) to the corresponding node of that arc. Shortest path going from s to all other nodes in V is found when (3.1) is satisfied for all edges in E. It may not be obvious but negative costs are permitted in the GSP but not negative cost cycles.

We use p_u to denote 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. p_s is often set to -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 checks the violation of Bellman's condition for all emanating edges of node u.

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 next two sections describes these rules and implementations.

3.3 Label Correcting Algorithm

Check if its FIFO or double ended queue

pseudo code

The GSP is addressed as a label correcting algorithm when the queue is a first in first out (FIFO) queue. Given the arc costs can be negative in the GSP, and in order to satisfy the Bellman's conditions for all edges, the algorithm has to scan all edges $|V| - 1$ times, giving a run time of $O(|V||E|)$.

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 ∈ V : 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) ∈ E 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
```

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. This algorithm is also called the BellmanFordMoore algorithm credited to Bellman (1958), Ford (1956) and Moore (1959).

3.4 Label Setting Algorithm

The classical 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 shortest length. In order to achieve label setting, the queue $\mathcal{Q}$ is modified to always have the minimum distance label in front of the queue, hence the algorithm iterates through every node in the graph exactly once, labelling the next node u in the order of non-decreasing distance labels.

The advantage of this algorithm over the label correcting algorithm is that all nodes in the graph are only visited once; the shortest path tree grows radially outward from the source node. It is clear that when the next node in the queue is the destination node, the algorithm can be stopped for the point to point SPP case, which is desirable for the Path Equilibration method.

3.4.1 Priority Queue Implementations

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) suggest the use of a min-priority queues. Min-priority queues are a collection of data structures that always serve elements with higher priorities. The priority in SPP are the distance labels: smaller distance label have a 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 in Algorithm 2) 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 is used for updating the distance 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 ∈ V : 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) ∈ E 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 or a binary min-heap, 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 $O(|V|)$ time (searching through the entire array), giving a overall time of $O(|V|^2 + |E|)$.

A binary min-heap is a binary tree which satisfies the min-heap property: the value of each node

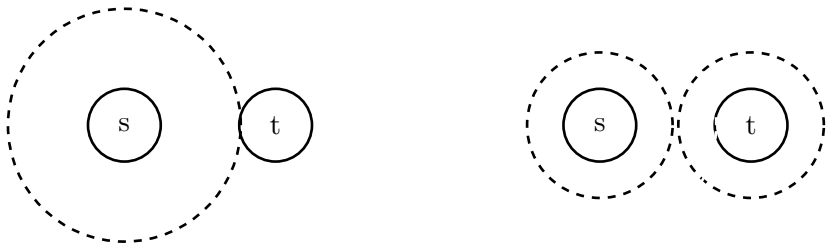
is smaller or equal to the value of its child nodes. Cormen et al. (2001) shows that if the graph is sufficiently sparse (in particular $E = o(|V|^2/\log(|V|))$, Dijkstra's algorithm can be improved with a binary min-heap. In this implementation, the binary tree takes $O(|V|)$ time, Extract-Min takes $O(\log(|V|))$ time for $|V|$ operations and Decrease-Key takes $O(\log(|V|))$ time for each $|E|$. The total running time is therefore $O((|V| + |E|)\log(|V|))$, which improves the array implementation.

The running time can be improved further using a Fibonacci heap developed by Fredman & Tarjan (1987). Where historically, the development of Fibonacci heaps was motivated by the observation that Dijkstra's algorithm typically makes many more Decrease-Key calls than Extract-Min. In Fibonacci heap, each of the $|V|$ Extract-Min operations take $O(\log(V))$ amortized time, and each of the $|E|$ Decrease-Key operations take only $O(1)$ amortized time, which gives a total running time of $O(V\log(V) + E)$.

Min-priority queue can also be implemented as a binary search tree, where the worst case for insertion, deletion and search for an element in the tree all run in $O(log(n))$ time. Dijkstra's algorithm (Algorithm 2) can be modified for a binary search tree implementation: when a label distance of node can be updated, we remove that node from the tree and insert a new one with the update value, which is analogous to the Decrease-Key operation. Dijkstra's algorithm using a binary search also runs $O((|V|+|E|)\log(|V|))$ in the worst case compared to the min-binary heap. 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, since we just delete the node from the tree and add the node with a different value, and there is no harm deleting a non-existent node from the tree.

3.5 Bidirectional Label Setting Algorithm

Dijkstra's algorithm can be imagined to be searching radially outward like 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. (Figure 3.2) Thus Dijkstra's algorithm can be run on the origin and destination simultaneously 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.



(a) Dijkstra's algorithm (b) bidirectional

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

show proof?

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. The correct termination criteria was first designed and implementation by Pohl (1971) based on researches presented by Dantzig (1963), Nicholson (1966) and Dreyfus (1969). Klunder & Post (2006) summarises the procedure and algorithm (Algorithm 3) for the termination criteria presented by Pohl (1971).

In Algorithm 3, two independent Dijkstra's algorithms are alternatively run on the forward and reverse graph (forward and backward algorithm), the algorithms terminate when a node is permanently labelled in both directions. Once the algorithms have 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 condition increases the run time significantly, searches have to be done for all edges that connect all labelled nodes in the forward search to all labelled nodes in the backward search.

Show theorem and proof?

Note in Algorithm 3, $\mathcal{R}^s$ is the subset of nodes that are permanently labelled from s with labels d_v^s in the forward search, and $\mathcal{R}^t$ is the subset of nodes that are permanently labelled from s with labels d_v^t in the backward search.

Algorithm 3 Bidirectional Label Setting 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 $\min\{\min\{d_v^s + c_{vw} + d_w^t | v \in \mathcal{R}^s, w \in \mathcal{R}^t, (v, w) \in E\}, d_u^s + d_u^t\}$, 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 embed during the searches. The termination condition is summarized as the following. During the forward and backward search, we maintain the length of the shortest path seen so far, μ, and its corresponding path. Initially, $\mu = \infty$. When an arc (v, w) is scanned by the forward search and w has already been scanned in the reverse search (or vice versa), we know the shortest $s - v$ and $w - t$ path have lengths d_v^s and d_w^t respectively. If $\mu > d_v^s + c_{vw} + d_w^t$ then this path is shorter than the one detected before, so we update u and its path accordingly. The algorithm terminates when the search in one direction selects a node already scanned in the other direction.

Goldberg et al. (2006) showed and proved a stronger termination condition on top of his previous one. The searches can be stopped if the sum of the top priority queue values is greater than μ:

Show theorem and proof as well?

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

where top_f and top_r are the top priority queue values in the forward and reverse search, they the next minimum distance label that have not been labelled.

3.6 A* Search

Up until now, Dijkstra's algorithm does not take into account the location of the destination, the shortest path tree is grown out radially until the destination is labelled. In a traditional graph where actual distances are used for the distance labels, a heuristic can be used to direct the shortest path tree to grow toward the destination (an ellipsoid in stead of a circle). If the heuristic estimate is the distance from each node to the destination, and the estimate is smaller than or equal to the actual distance going to that destination, then a shortest path can be found. This is called A* search or goad directed search, first described by Hart et al. (1968).

Formally we define the following. Let h_v be a heuristic estimate from node v to destination t, and apply Bellman's condition such that an optimal solution exist, that is

$$h_v \leq h_u + c_{uv} \quad \forall (u,v) \in E, \tag{3.3}$$
$$h(t) = 0, \tag{3.4}$$

where t is the destination node. This means the heuristic function h must be admissible and consistent. The heuristic must never overestimate the actual path length and the estimated cost of a node reaching its destination node must not be greater than the estimated cost of its predecessors. Note a consistent heuristic is also admissible but not the opposite. Hart et al. (1968) proves if the heuristic function (such as using geographical coordinates and Euclidean distance) is admissible and consistent, then A* is guaranteed to find the correct shortest path with a better time performance by scanning less nodes and edges.

To implement A* search, Dijkstra's algorithm is modified. Instead of selecting the node with the minimum distance label in the priority queue, we select the next node u that has the minimum distance label added with the heuristic value, which is $d_u + h_{ut}$ where h_{ut} is the estimated distance from node u to destination t.

In the Path Equilibration method, geographical coordinates and Euclidean distances can not be used for the heuristic estimate because a travel time function is used for the length of the edges. Instead, zero-flow travel time from every node to the destination can be used for the heuristic. Zero-flow travel time admissible and consistent and can be shown by analysing the travel times function (Figure 2.1). The travel times function is a non-decreasing function with the lowest value being the zero-flow travel times. This means using zero-flow travel times as the heuristic estimate is assured to be admissible as no travel time can be lower than the zero flow travel at any time. The heuristic function is consistent because the travel time from a node to the destination must be no longer than all its predecessors.

3.7 Bidirectional A* Search

Bidirectional search can also be applied to A* search, where two ellipsoids are extended from the origin and destination respectively (Figure 3.4. One may construct the shortest path with the

same termination condition described in section 3.5. But this would not work. This is due to fact that A* search does not label the nodes permanently in the order of their distance from the origin (Klunder & Post 2006), the heuristic estimations are no longer consistent.

The strategy for the correct use of heuristic estimates and termination criterion has first been published by Pohl (1971). The use of heuristic estimates is later improvement by Ikeda et al. (1994) and the termination criterion is improved by Goldberg et al. (2006).

The strategy is as follows. The heuristic estimates need to translated to consistent functions first. We denote $\pi_f(v)$ the estimate on distance from node v to the destination t in the forward search and $\pi_r(v)$ the estimate on distance from origin s to node v in the backward (reverse) search (Figure 3.3)

add $\pi_f(t) and \pi_r(s)$ for figure

In general two arbitrary feasible functions π_f and π_r are not consistent, but their average is both feasible and consistent (Ikeda et al. 1994):

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

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

where the two constants $\frac{\pi_r(t)}{2}$ and $\frac{\pi_f(s)}{2}$ are added by Goldberg et al. (2006) to provide better estimates. Note the modified consistent heuristic p provides worse bounds than the original π values.

mention network is not symmetric so they don't meet, π values are different

Finally Goldberg et al. (2006) shows and proves the stopping criterion:

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

where is μ the best $s - t$ path seen fast, top_f the length of the path from s to the top node (minimum distance label) in the forward search priority queue and top_r the length of the path from t to the top node in the backward search priority queue.

3.8 Preprocessing Algorithms

In the last 2 decades, extensive research has been done on the idea of speeding up shortest path calculations using pre-calculated data. These preprocessing algorithms generally need to to spend a long time pre-calculating in order to speed up the subsequent shortest path queries.

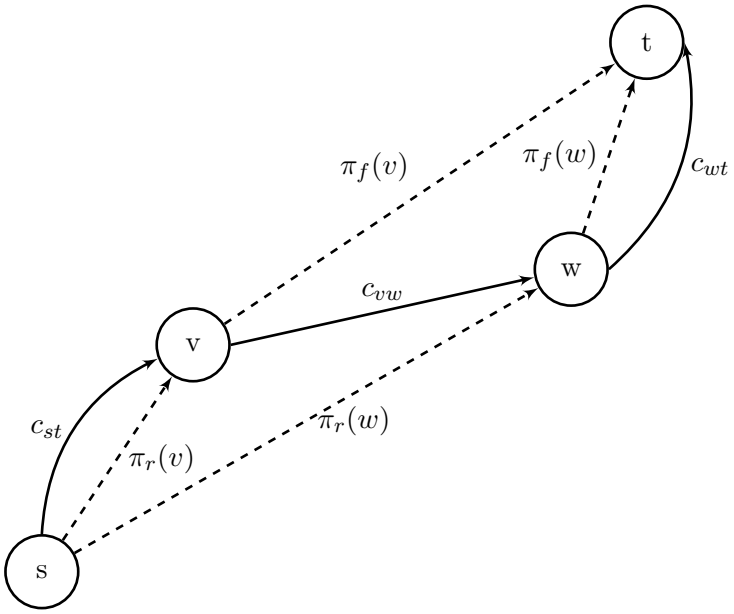


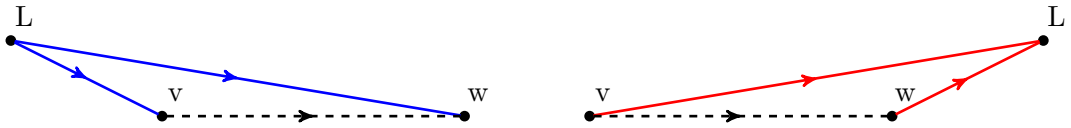
Figure 3.3: Heuristic values for bidirectional A* search

(a) A* search (b) bidirectional

Figure 3.4: Difference between the scan area of A* search and its bidirectional version

3.8.1 A* Search with Landmarks

In section 3.6, zero flow travel times is used for the heuristic estimate. Better heuristic estimate can also be used. One such estimate comes from the landmarks method developed by Goldberg & Harrelson (2005). In fact the class of A* search algorithms that use a feasible function (heuristic estimate) π_t with $\pi_t(t) = 0$ as lower-bounding algorithms. Goldberg & Harrelson (2005) concludes that better lower bounds give better performance, where landmarks with triangle inequalities can give a better lower bounds. This method works as follow. First we select a small set of landmarks and place them around the graph, and pre-compute distances to and from every landmark for each node. Then we calculate lower bounds by triangle inequality: $\text{dist}(L, w) - \text{dist}(L, v) \leq \text{dist}(v, w)$ and $\text{dist}(v, L) - \text{dist}(w, L) \leq \text{dist}(v, w)$. The tightest lower bound is the maximum over all landmarks $\max\{\text{dist}(L, w) - \text{dist}(L, v), \text{dist}(v, L) - \text{dist}(w, L)\} \leq \text{dist}(v, w)$.



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

Figure 3.5: Explanatory diagram for triangle inequality

Goldberg & Harrelson (2005) outlines the procedure of selecting how many landmarks and where to place them. And states that, for transportation networks, having a landmark geometrically lying infront of the origin or behind the destination tends to give good bounds. And the selection of how many landmarks is dependent on the size of the graph. More landmarks means more preprocessing time, and doe not necessary give better lower bounds.

3.9 Techniques for Iterative Calculations

In the path equilibration method of traffic assignment, shortest path has to be repeatedly calculated between every O-D pair while arc costs change between each iteration. Thus we can take advantage of information calculated from the previous iteration to speed up subsection calculations.

3.9.1 Lifelong Planning A*

A family of algorithms exist for dynamically changing graphs, for example moving node locations, changing arc lengths etc. One particular algorithm which particularly tackles this problem 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 graph change only slightly and the changes are close to the goal. This means Lifelong Planning A* can be used for our problem if we

can prove that the graph only changes slightly between each iteration of the path equilibration method.

3.9.2 Avoiding Shortest Path Calculations

explain why

. All of the algorithms mentioned so far need to calculate shortest path between every O-D pair in each iteration. But because of the nature of the path equilibration method, calculations can be avoided.

We can hope the shortest path for the current iteration is the same as the previous iteration, so avoid calculating it can reduce the computational time.

We are able to skip shortest path calculations in the Path Equilibration algorithm mainly due to how the algorithm and the convergence criterion works. The convergence criterion described in Section 2.3 is depend on the current user equilibrium solution, so it is okay if we compute an inaccurate solution that does not converge for the current iteration but make a better solution a few iterations later, the only change is the increase number of iterations, which may increase the computational time.

If we can prove that shortest path do not change most of the time, then we can apply some strategies to avoid the calculation. Overall the computational time is reduced because calculations are avoided on the shortest paths that are not going to change often.

The first strategy is as follows. For each O-D pair, if its shortest path do 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 normal 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. Or if we choose to skip too few iterations, then there will be not much of an affect.

The other strategy is as follows. For each O-D pair, we skip the current calculation by an random probability. The advantage of this strategy is that we do not need to know how many iterations the algorithm will take. But the computational time will vary from different runs, resulting it to be unpredictable.

explain total number of iters will increase

Chapter 4

Implementation Details

this chapter has to be more formal, it is too colloquial at the moment. And I am not sure but some of the content.

The previous chapter has described all the algorithms that are going to be implemented for this report. In this chapter, we seek and research specific implementation details that make the algorithms perform faster.

Note the traffic assignment algorithms have already been implemented by the co-supervisor of this report in a Object Oriented C++ program. The programs includes Frank-Wolfe, Path Equilibration, label correcting algorithm and many more.

4.1 Graph Storage

The network graph storage is implemented as a Forward Star data structure. Information about Forward Star can be found in (Sheffi 1985). In summary, Forward Star stores a network compactly with $O(|V| + |E|)$ space. It allows $O(1)$ access for any nodes in the graph and $O(1)$ access for all edges emanating from a random node, which are the requirements for the generic shortest path algorithms. Using Forward Star ensures that run time of accessing the graph can be neglected when analysing the shortest path algorithms.

4.2 Traffic Assignment Implementation

Convergence and Rel gap! 1e-6 for all networks

Implemented by Olga, the co-supervisor.

4.3 Priority Queue Implementations

Various implementations of the priority queues exist, they include:

- std::priority_queue - an array based heap implementation from the C++ standard library,
- std::set - a red and black binary search tree from the C++ standard library,
- boost::heap - pointer based heap implementations from the boost library.

Each priority queue implementation have some advantages than the other. For example faster tree balancing, faster Extract-Min or Delete etc.

We first examine the 6 variants of Heap implementations from the C++ Boost Heap Library shown in Table 4.1 (Blechmann 2013). Where N is the number of elements in the Heap tree, and all time complexities are measured in amortized time, i.e. the average run time if the operation is run for a long period of time, average out worse case and best case.

Table 4.1: C++ Boost Heap Implementations with Comparison of Amortized Complexity

	top()	push()	pop()	increase()	decrease()
d-ary (Binary)	O(1)	O(log(N))	O(log(N))	O(log(N))	O(log(N))
d-ary (Ternary)	O(1)	O(log(N))	O(log(N))	O(log(N))	O(log(N))
Binomial	O(1)	O(log(N))	O(log(N))	O(log(N))	O(log(N))
Fibonacci	O(1)	O(1)	O(log(N))	O(1)	O(log(N))
Pairing	O(1)	$O(2^{2*\log(\log(N))})$	O(log(N))	$O(2^{2*\log(\log(N))})$	$O(2^{2*\log(\log(N))})$
Skew	O(1)	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 (or increase) function. The decrease (or increase) function is referred as the decrease-key (or increase-key) operation mentioned in Section 3.4, which updates the value of the key in the Heap tree. Decrease-key is used for a min-heap and increase-key for a max-heap tree. For Dijkstra's algorithm, often nodes are scanned multiple times in the label updating step, instead of adding the node again into the Heap tree, we can use decrease-key on the node, updating its distance label. This means we can reduce the size of the Heap tree and run time by using decrease-key rather than adding the same node with different distance label in the queue again.

this is going to be hard to find a good reference, all reports are from Stackoverflow.com

In table 4.1, we observe the Fibonacci Heap has a very interesting time complexity: constant amortized time for the push, pop and increase-key operation time. But the fact is, we do not know how much constant time it really uses behind its big O. It is reported that Fibonacci Heaps only outperforms other Heaps when the graph is very dense, but it is worth to experiment Fibonacci Heap as well as all other Heaps.

heap from boost are slower, as they are pointer based, not array based

The C++ standard library Heap is still going to be implemented and tested. The C++ standard library Heap does not support the decrease-key operation but careful implementation would not give wrong result. This is due to the fact that if a node has been added to the Heap more than once, the node with smaller distance label is always going to be removed from the queue and update its successors first, the same node with larger distance label will therefore not update its successors.

C++ Boost Library Heaps are implemented as max-heaps, which means in order to use the Fibonacci O(1) increase-key function, we need to negate the distance labels when we add them into the Heap.

Chapter 5

Computational Results

This chapter shows the results from testing all the shortest path algorithms detailed in Chapter 3 using the specific implementation described in the previous chapter.

The results are generated from using the g++ compiler using the -O3 optimise for speed option on Ubuntu 12.04 operating system, which has a Intel Core i5-3317U CPU with 3.8GiB RAM.

5.1 Problem Data and Result Explanation

The problem data for solving the TA problems are retrieved from Transportation Network Test Problems (Bar-Gera 2013). Table 5.1 shows the data that are going to be tested with, where the network name, numbers nodes, traffic analysis, origin-destination (OD) pairs and edges are given.

Table 5.1: Network Problem Data

Network	Nodes	Zones	OD pairs	Edges
Anaheim	416	38	1406	914
Barcelona	1020	110	7922	2522
Winnipeg	1052	147	4344	2836
ChicagoSketch	933	387	93135	2950

By examining the network problem data, we can see that the number of OD pairs increase significantly respect to the number of zone nodes, this is important because it indicates how many point to point SPPs need to be solved for each iteration of the PE method. We can also roughly tell that these networks are very sparse; for a complete graph (every node is connected to every other node) of 1000 nodes have 499500 edges ($n(n-1)/2$), but the larger networks in our problem data only have about 0.4% to 0.6% of edges in the corresponding complete

graph. Analysing the graph shows the degree of any vertex in the graph is no more than 5. This information is useful for choosing the best algorithm and data structure.

The correctness of the final shortest path trees are checked by comparing to the label correcting algorithm that is implemented by the co-supervisor of this project, which is guaranteed to be correct.

5.2 Discussion of Computational Results

Figure 5.1 and 5.2 shows the performance of Dijkstra's algorithm on both Winnipeg and ChicagoSketch network using priority queue implementations from the C++ Boost library and the Standard Template Library.

On both networks, the Heap implementation from the C++ Standard Template Library out performs all other implementations. And the Fibonacci Heap does not have the promised runtime behaviour described in Section 4.3. What is interesting is that for the Heaps with $O(\log(N))$ behaviour, their run time vary a lot, especially for the Binomial Heap.

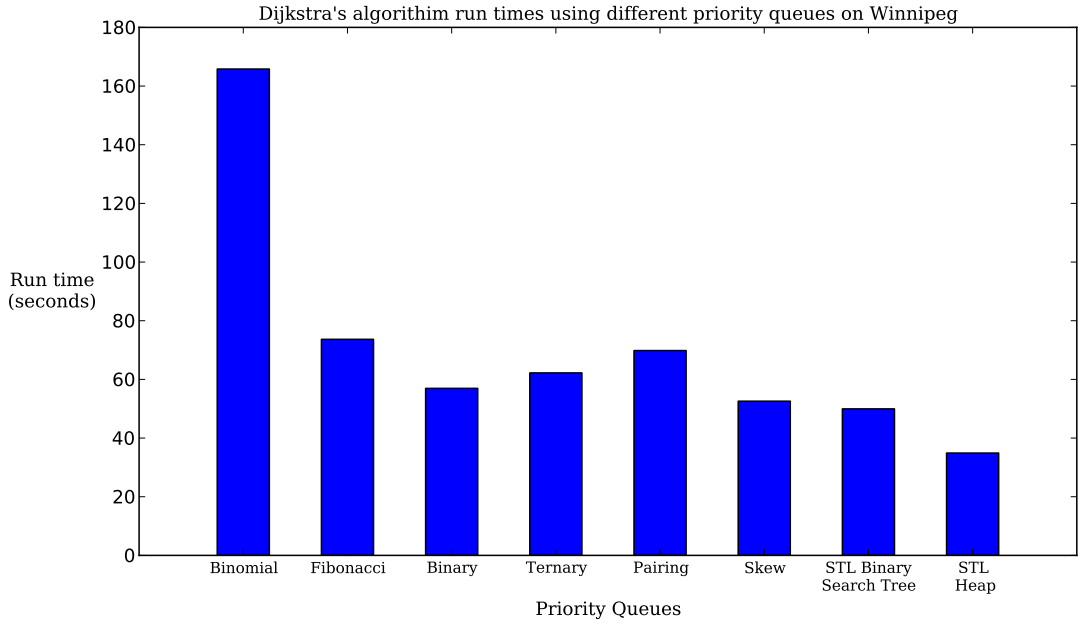


Figure 5.1: Dijkstra's algorithm run times using different priority queues on Winnipeg

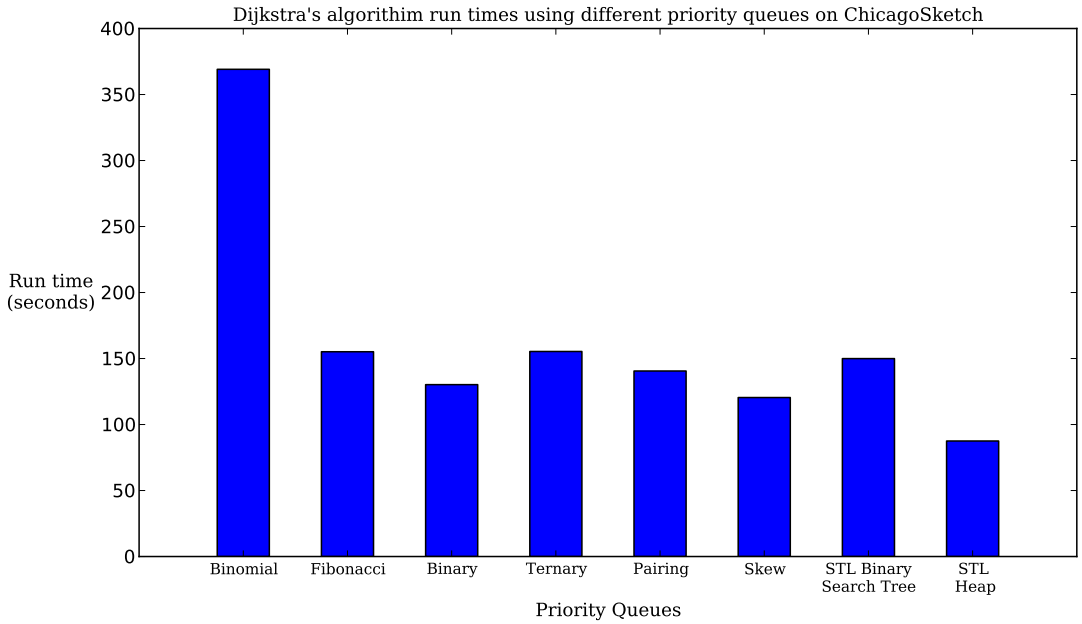


Figure 5.2: Dijkstra's algorithm run times using different priority queues on ChicagoSketch

Figure 5.3 shows the performance of each of the algorithms

- label correcting Bellman-Ford,
- point to point Dijkstra using heap from C++ standard library,
- point to point Dijkstra using binary search tree,
- bidirectional Dijkstra,
- A* search,
- bidirectional A* search,

used on each of the networks shown in Table 5.1.

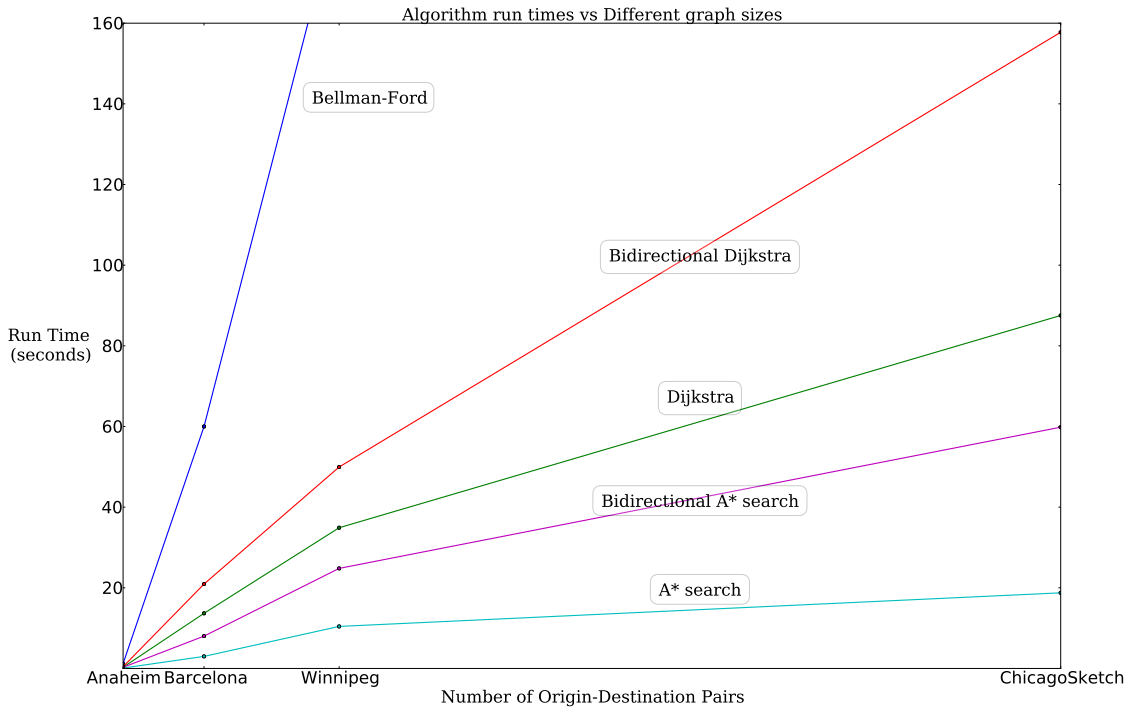
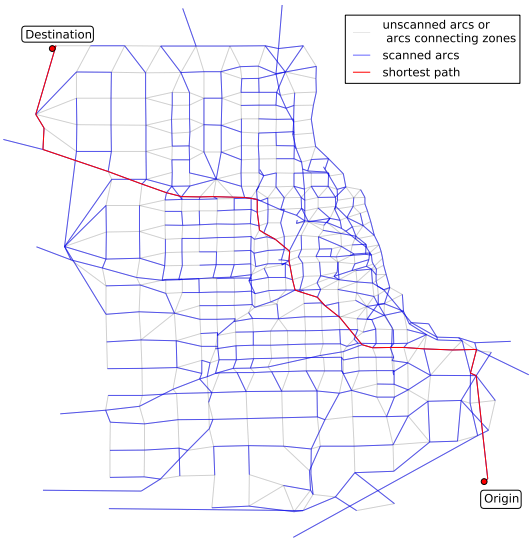


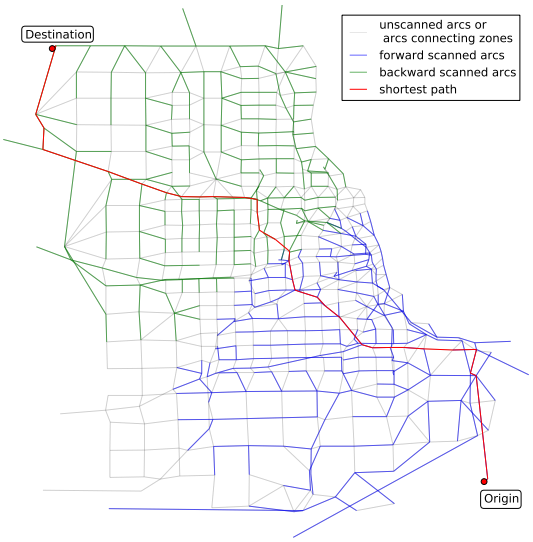
Figure 5.3: Run time performances of different algorithms on different networks

Now we investigate the reason for the run time behaviours of the shortest path algorithms shown in Figure 5.3. Figure 5.4 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; the Bidirectional Dijkstra scans almost the whole network but with a few nodes left out; The Bidirectional A* search scans a slightly larger region near the origin and destination nodes; and the A* search only scans a few nodes along the shortest path. The behaviour of the algorithms is shown further in Figure 5.5, where the origin and destination is placed close to each other. The Dijkstra and Bidirectional Dijkstra algorithm need to scan many nodes before termination; And the A* and Bidirectional A* search only scan a small portion of network, and they do not scan the area behind the origin and destination node.

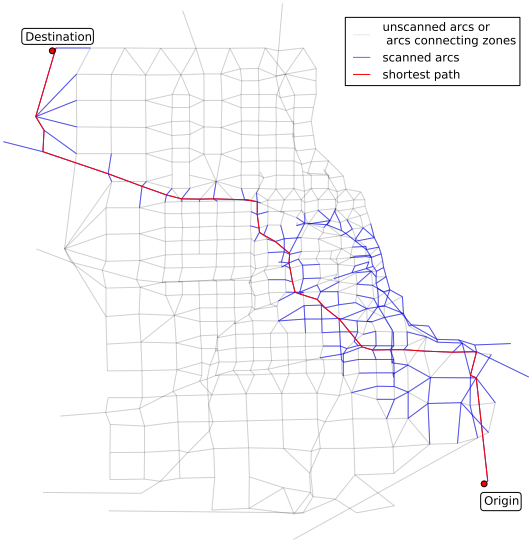
The run time of the shortest path algorithms are heavily dependent on the number of nodes they scan. The process of scanning a node consists of placing the node into a priority queue, taking it out sometime later, and check all its out going arcs. So cumulatively the less nodes an algorithm scans the faster it runs. And by examining Figure 5.4 and Figure 5.5, it is obvious to see A* search is the fastest.



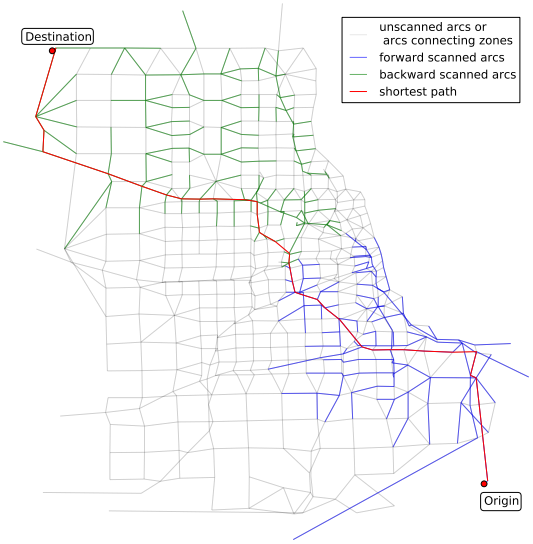
(a) Dijkstra



(b) Bidirectional Dijkstra

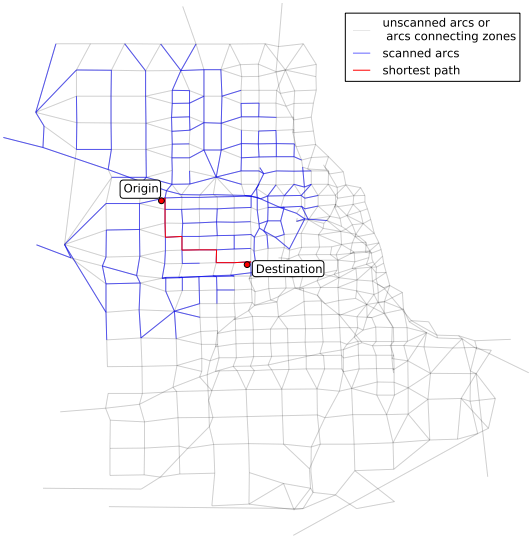


(c) A* Search

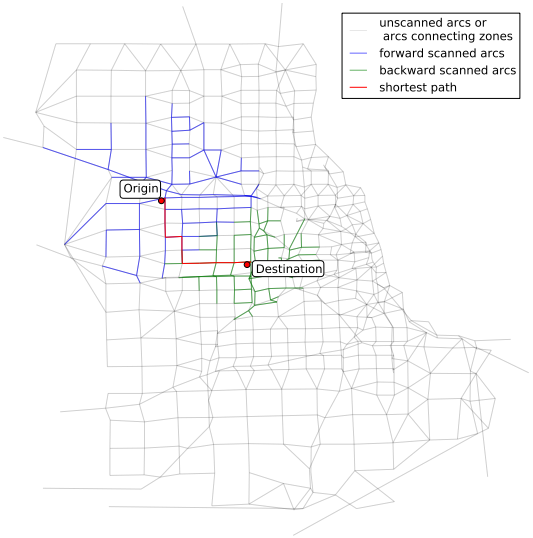


(d) Bidirectional A* Search

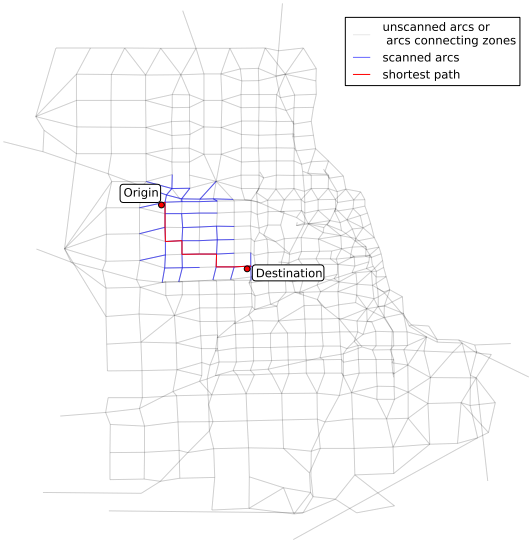
Figure 5.4: Shortest path tree between two distant nodes in the ChicagoSketch Network



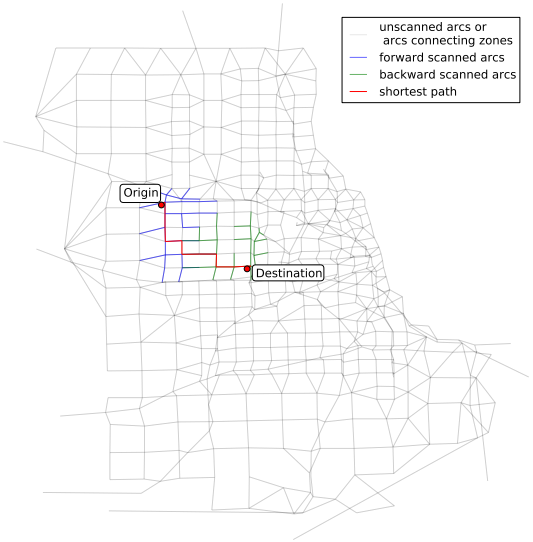
(a) Dijkstra



(b) Bidirectional Dijkstra



(c) A* Search



(d) Bidirectional A* Search

Figure 5.5: Shortest path tree between two close nodes in the ChicagoSketch Network

5.3 Results on skipping shortest path calculations

In this section we consider skipping shortest path calculations in the iterative Path Equilibration algorithm discussed in Section 3.9.2.

First we investigate how the shortest paths change during a complete run of the Path Equilibration algorithm. Figure 5.6 shows the percentage of how many times each O-D pairs' shortest path change during the 26 iterations of the algorithm run on the ChicagoSketch network. It is shown that most of the shortest paths only change a few times during the complete run of the Path Equilibration algorithm. This entails that after the first dozen of iterations, the rest of the algorithm spends its time balancing only a few shortest paths. From this observation, we can develop a strategy of skipping some shortest path calculations if they do not change in the previous two iterations. This result is shown in Figure 5.7 for the Terrassa and ChicagoSketch network, where an arbitrary numbers of skipping are chosen.

The Terrassa network mentioned above is a small network with 55 zones, 1609 nodes, 3264 links and 2215 O-D pairs. Using A* search, the algorithm takes 415 iterations to converge due to its complicated supply and demand description.

Figure 5.7a shows the A* search run time on Terrassa network. The strategy is to skip the next 15, 25, 50, 100 and 200 calculations for the O-D pair that did not change its shortest path in previous two iterations. This strategy worked very well as all of the run times are reduced by half, with only some change in the total number of iterations. Figure 5.7b shows the same information on the ChicagoSketch network, with skipping the next 5, 10 15 and 20 iterations. There are only a slight reduce of run times, with a relatively large increase in the total number of iterations. The strategy did not work well due to its large network size.

The problem with this strategy is that we have to know how many iterations the algorithm will take and choose an appropriate number that is less than that, this is because choosing a 'bad' skipping number will result either unnecessary computations or have no effect on reducing the run time.

The next strategy we explore is to skip the next shortest path calculation randomly. The advantage of this strategy is that it does not need the number of iterations it will compute, but the disadvantage is that the run time may vary between different runs. Figure 5.8a shows the A* search run time on the Terrassa network with skipping the current O-D pair shortest path calculation using probabilities of 0.3, 0.4, 0.5, 0.6 and 0.7. In this case the run time dropped quite significantly for 0.5 probability, while the other probabilities all have reduced the run time by some amount. Figure 5.8b shows the same strategy on the ChicagoSketch network. Again this time there is no significant reduction in run time but all of the run times are reduced slightly.

Since the skipping shortest path strategy seems to work well, and because only small networks have been experimented so far, we will now experiment this strategy along with A* search on two more huge networks.

The networks that is going to be experimented are the Philadelphia and ChicagoRegional network.

The Philadelphia network consists of 1,525 zones, 13,389 nodes, 40,004 links and 1,149,795 O-D pairs. The ChicagoRegional network consists of 1,790 zones, 12,982 nodes, 39,018 links and 2,296,227 O-D pairs. It is interesting to note that these two networks consist of 1 and 2 millions of O-D pairs, which means they require 1 and 2 millions shortest path calculations just for 1 iteration of the Path Equilibration algorithm. The run time comparisons are shown in Table 5.2.

Table 5.2 show both the A* search and A* with 0.5 probabilities of randomly skipping an shortest path calculation on the two networks mentioned above. The random skipping strategy has approximately 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* with 0.5 probability of randomly skipping	5.75 hours	24.18 hours

maybe need random analysis

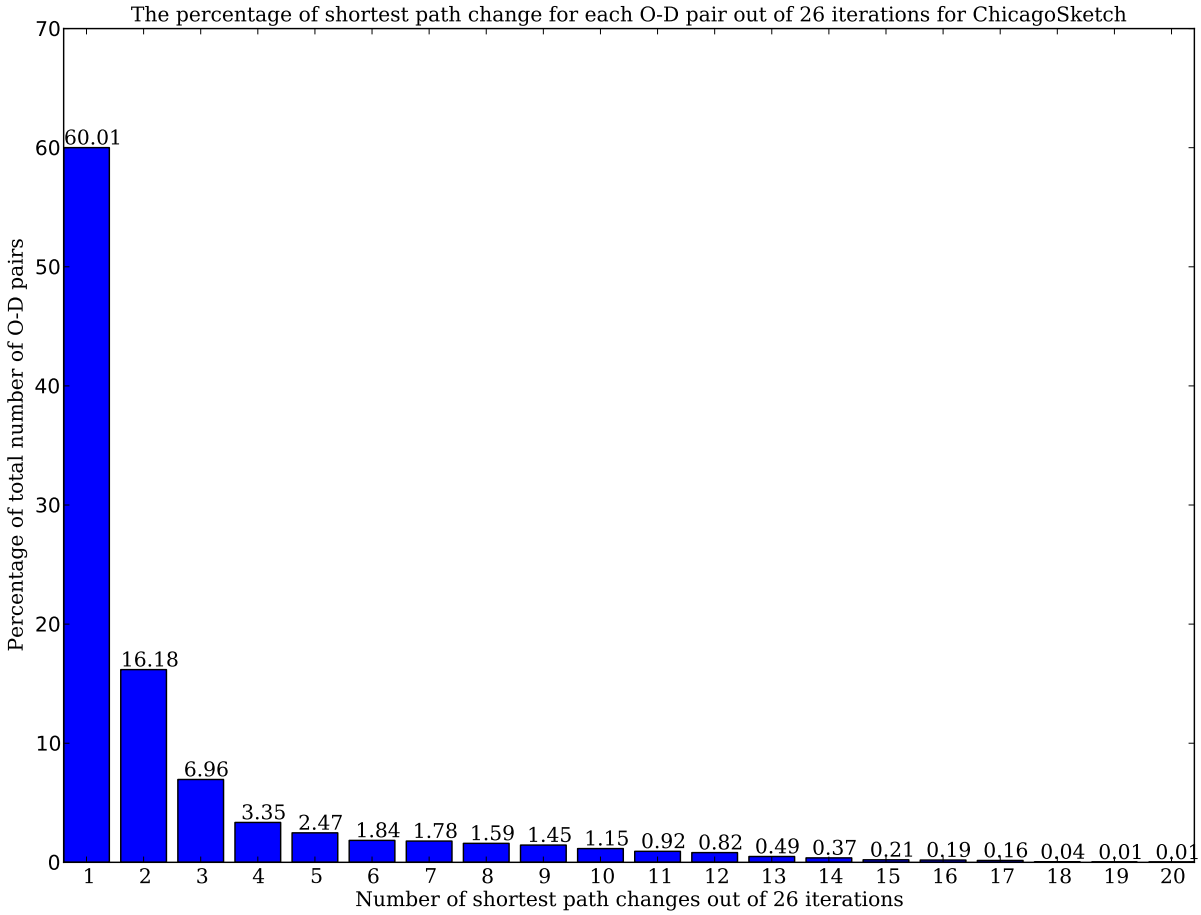
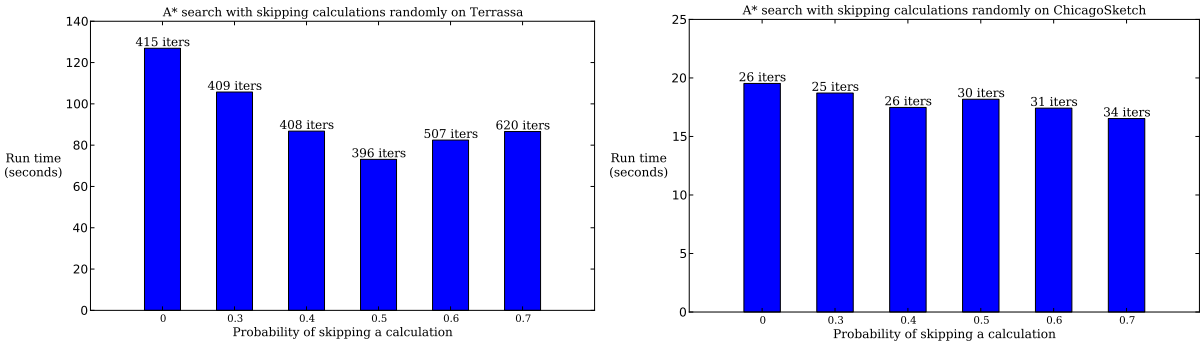


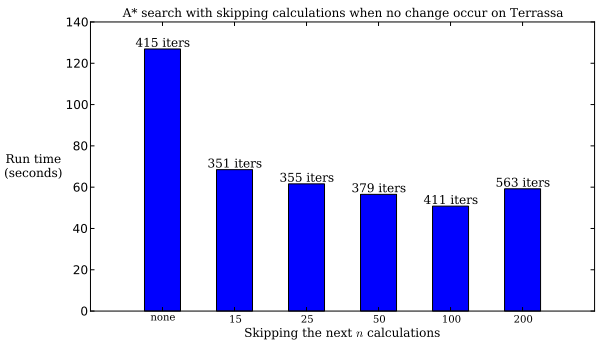
Figure 5.6: The percentage of shortest path change for each O-D pair out of 26 iterations for ChicagoSketch



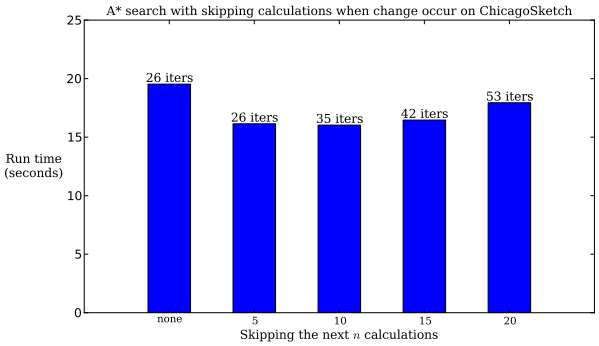
(a) Terrassa network

(b) ChicagoSketch network

Figure 5.7: Run time for skipping shortest path calculations randomly



(a) Terrassa network



(b) ChicagoSketch network

Figure 5.8: Run time for skipping shortest path calculations if the previous 2 did not change

Chapter 6

Conclusions

A* search out performs all other algorithms ...

Chapter 7

Future Work

Appendix A

Priority queue data structure 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
	STL Binary Search Tree	127	66.89
	STL Heap	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
	STL Binary Search Tree	25	145.95
	STL Heap	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 (Heap)	10	0.30
	Dijkstra (Tree)	10	0.62
	Bidirectional Dijkstra	10	0.43
	A*	10	0.10
	Bidirectional A*	10	0.28
Barcelona	Bellman-Ford	28	60.00
	Dijkstra (Heap)	27	13.67
	Dijkstra (Tree)	27	27.36
	Bidirectional Dijkstra	27	20.93
	A*	28	2.99
	Bidirectional A*	30	8.02
Winnipeg	Bellman-Ford	129	190.00
	Dijkstra (Heap)	127	34.89
	Dijkstra (Tree)	127	66.89
	Bidirectional Dijkstra	126	49.95
	A*	126	10.42
	Bidirectional A*	127	24.81
ChicagoSketch	Bellman-Ford	25	500.00
	Dijkstra (Heap)	25	87.52
	Dijkstra (Tree)	25	149.95
	Bidirectional Dijkstra	25	157.75
	A*	26	18.75
	Bidirectional A*	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`.

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).

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. & 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.

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.

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.

Slavin, H., Brandon, J. & Rabinowicz, A. (2006), An empirical comparison of alternative user equilibrium traffic assignment methods, Technical report, Caliper Corporation.

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