Engineering Science

Faster Shortest Path Computation for Traffic Assignment

Author:
Boshen Chen

Supervisors:
Dr. Andrea Raith
Olga Perederieieva

August 5, 2013

Abstract

Acknowledgement

I acknowledge ...

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

CHECK LATEX LOG!!

use less ‘ we ’

1.1 Project Motivation

As the result of constantly increasing population, cities worldwide and their road networks are becoming more complex and difficult to navigate, leading to traffic congestions that are more problematic than ever for traffic designers and road users.

A traffic model called the transportation forecasting model is built with the aim to reduce congestion and predict future traffic response when the behaviour of the traffic is changed. This model solves and estimates traffic flows for a given time period with the following four stages: trip generation, trip distribution, mode choice and traffic assignment (Figure 1.1). In short, this model generates origins and destinations for travellers to travel from and to in different parts of the road network, it then calculates the number of trips that are required for each origin and destination pair and computes the proportion of trips between each pair that use a particular transportation method, in the end it assumes all travellers choose the best trip with the least transportation cost and best transportation method (e.g. shortest path, least travel time or cheapest route) and assigns each traveller to their destination considering traffic congestion.

The traffic assignment (TA) problem in the last stage of the forecast model is a very complicated problem, this is because the problem is only solved when the network reaches user equilibrium, which means no traveller can lower their transportation cost through unilateral action: every traveller will strive to find the shortest path while ignoring all other travellers.

reference user equilibrium - John Glen Wardrop principles of equilibrium

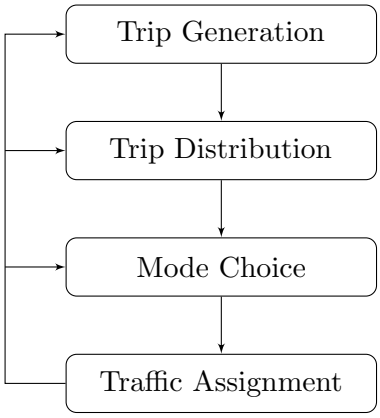


Figure 1.1: Transportation forecasting model

User equilibrium is difficult to find because in traffic assignment, travel times on different roads are modelled as nonlinear functions to capture congestion effects (more traffic flow results slower travel time). As different routes are assigned to the travellers, congestion happen differently for each road in a nonlinear manner, making the result of relocation of travellers hard to calculate.

One method of solving the traffic problem is the Path Equilibration (PE) method (Florian & Hearn 1995). This method initially calculates the shortest paths between each trip origin and destination based on the zero-flow travel times. Traffic flows are assigned to these shortest paths and new travel times are updated accordingly. The method iteratively re-identifies new shortest path based on the new travel times and re-assigns traffic flows until user equilibration is reached.

write about Frank-Wolfe

Both of these methods are iterative methods that require shortest path calculations for every trip origins in the network. It is not difficult to imagine that there would be many shortest path calculations if the network has hundreds of origins and destinations and takes some iterations to solve. Each shortest path calculation would also be very hard to solve if the network has a few hundred intersections and a few thousand roads for a realistic city road network. Sheffi (1985) states that finding the shortest path is the most computation intensive component for the PE or Frank-Wolfe algorithms, other components in the algorithms such as updating new values and convergence check only requires a few percentages of the total running time. Overall, speeding up shortest path calculations would significantly speed up the traffic assignment algorithms. As a result, traffic forecasting would be solved faster for larger and more complicated road networks, which allow city designers predict traffic further into the future and make better decisions on road network design.

1.2 Project Aims

This project aims to embed well known shortest path algorithms that are applicable for traffic assignment methods, and find the fastest. The algorithm that are going to be tested are:

- Bellman-Ford Label Correcting Algorithm,
- Dijkstra Label Setting Algorithm (using different data structures),
- Bidirectional Dijkstra,
- A* Search,
- Bidirectional A* search.

This project also aims to find and discuss techniques that can speed shortest path calculations in an iterative environment:

- network preprocessing,
- using information from the previous iteration for the current iteration to avoid recalculating shortest paths that are not going to change.

1.3 Report Overview

Chapter 2 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 3 presents the specific implementation details. Chapter 4 shows results.

Chapter 5 discussion chapter 6 conclusion ...

Chapter 2

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.

2.1 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 $i = 1, \ldots, k-1$

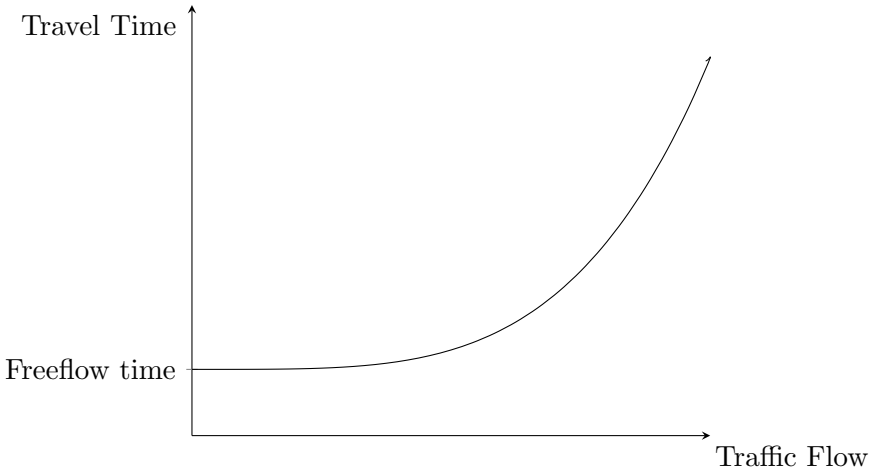
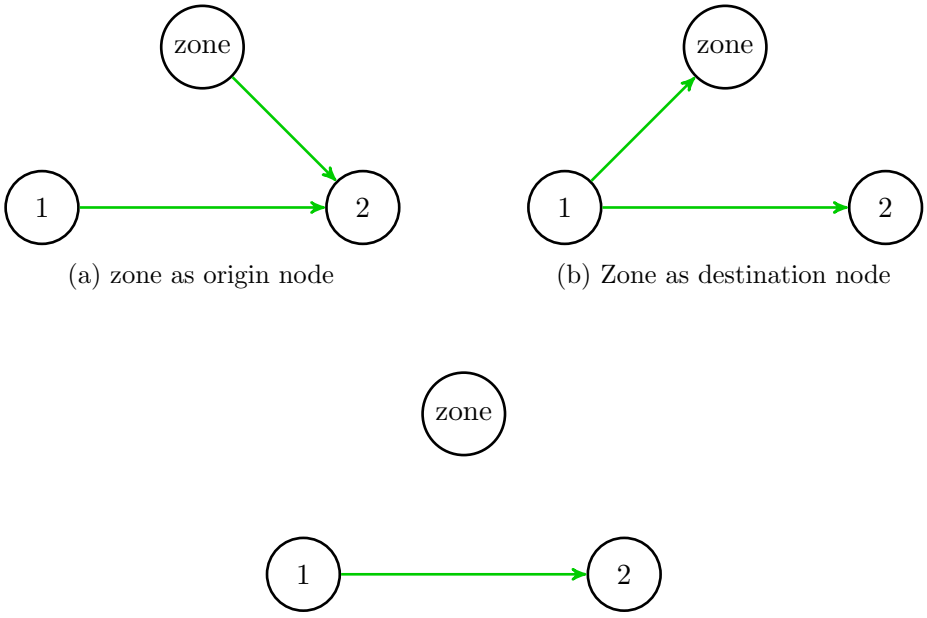


Figure 2.1: Travel time function.

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 2.2 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 2.2: Zone node and its allowable arc flows

2.2 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:

$$d_v \leq d_u + c_{uv}, \quad \forall (u, v) \in E, \tag{2.1}$$

$$d_v = d_u + c_{uv}, \quad \forall (u, v) \in P. \tag{2.2}$$

The inequalities (2.1) 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 (2.1) to the corresponding node of that arc. Shortest path going from s to all other nodes in V is found when (2.1) 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 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
```

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.

2.3 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 $O(|V||E|)$.

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).

2.4 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.

2.4.1 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.

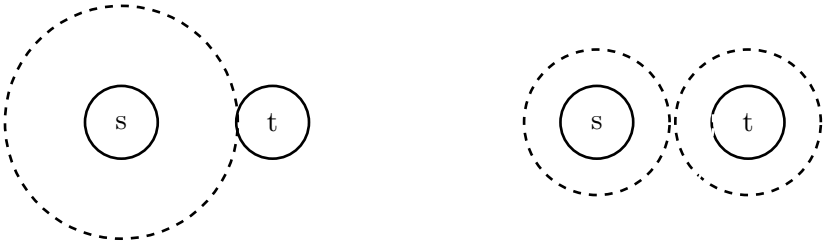
The running time can be improved further using a Fibonacci heap developed by Fredman & Tarjan (1987). Where historically, the development of Fibonacci heaps was motivated by the observation that Dijkstra's algorithm typically makes many more Decrease-Key calls than Extract-Min. In Fibonacci heap, each 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.

2.5 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 2.3) 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 2.3: 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

Show theorem and proof?

be done for all edges that connect all labelled nodes in the forward search to all labelled nodes in the backward search.

Note in Algorithm 3, $\mathcal{R}^s$ 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.

2.6 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

$$h_v \leq h_u + c_{uv} \quad \forall (u,v) \in E, \tag{2.3}$$
$$h(t) = 0, \tag{2.4}$$

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.

2.7 Bidirectional A* Search

Bidirectional search can also be applied to A* search, where two ellipsoids are extended from the origin and destination respectively. One may construct the shortest path with the same termination condition described in section 2.5. 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.

illustrate!

(a) A* search (b) bidirectional

Figure 2.4: Difference between the scan area of A* search and its bidirectional version

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):

$$p_f(v) = \frac{1}{2}(\pi_f(v) - \pi_r(v)) + \frac{\pi_r(t)}{2} \tag{2.5}$$

$$p_r(v) = \frac{1}{2}(\pi_r(v) - \pi_f(v)) + \frac{\pi_f(s)}{2} \tag{2.6}$$

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.

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

$$\text{top}_f + \text{top}_r \geq \mu + p_r(t), \tag{2.7}$$

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.

2.8 Preprocessing and More

this section is in draft stage

Preprocessing - trade memory to get faster time. We can either do a fast preprocessing between iterations to make query in each iteration (so combined speed is still faster) or do a long preprocessing at the start and use the computed heuristic values

- A* landmarks and triangle inequality (ALT)
- Reach-based routing
- ALT + Reach
- Geometric Containers

- Arc Flags

If we have more data on the network we can use algorithms that use hierarchies. Consider roads with higher speed first: use a hierarchy of subgraphs.

- Radius search.
- multi-level approach
- highway hierarchies

Extract from: Speed-Up Techniques for Shortest-Path Computations by Dorothea Wagner, Thomas Willhalm, and Fast Shortest Path Algorithms for Large Road Networks by Faramroze Engineer

We can also try Lifelong Planning A* (LPA*), using heuristic from previous each iteration, but the original paper says only a few percent arc change can boost run time, not idea if it is more than that.

If the edge lengths are whole numbers then we can use multi-level bucket for the priority queue.

Chapter 3

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.

3.1 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.

3.2 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 3.1 (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 3.1: 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 2.4, 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 3.1, 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.

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 4

Computational Results

This chapter shows the results from testing all the shortest path algorithms detailed in Chapter 2 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.

4.1 Problem Data and Result Explanation

The problem data for solving the TA problems are retrieved from Transportation Network Test Problems (Bar-Gera 2013). Table 4.1 shows the data that are going to be tested with, where the network name, numbers nodes, traffic analysis, origin-desitination (OD) pairs and edges are given.

Table 4.1: Network Problem Data

Network	Nodes	Zones	OD pairs	Edges
SiouxFalls	24	24	528	76
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.

4.2 Discussion of Computational Results

Figure 4.1 shows the performance of Dijkstra's algorithm on the ChicagoSketch network with the following priority queue implementations from the C++ Boost library:

- Binomial
- Ternary
- Fibonacci
- Pairing
- Binary
- Skew

and from the C++ standard library:

- set - implemented as red and black binary search tree
- priority_queue - implemented as heap tree.

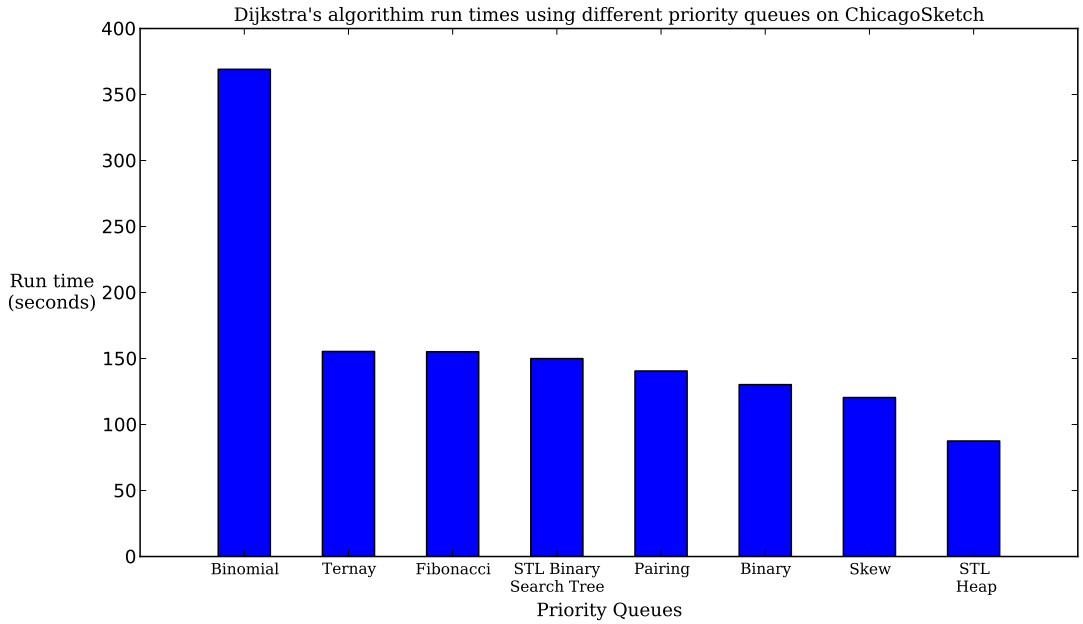


Figure 4.1: Dijkstra's algorithm run times using different priority queues on ChicagoSketch

Figure 4.2 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 4.1.

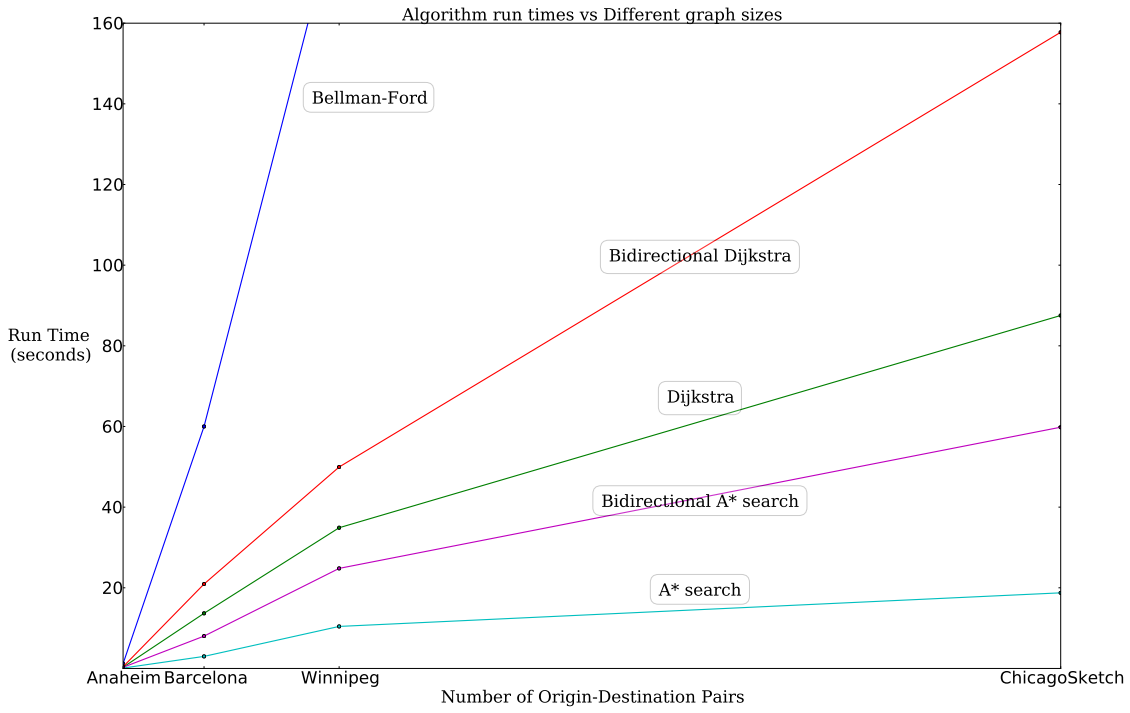


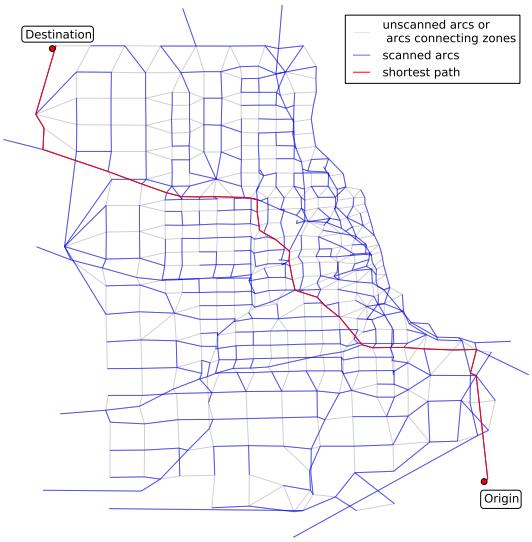
Figure 4.2: Performance of different algorithms on different networks

Figure 4.3 shows the shortest path tree between two distant nodes on the ChicagoSketch network. We can see Dijkstra's algorithm scans the entire network. Bidirectional Dijkstra scans almost the whole network with a few nodes not being scanned. A* search scans only a small region of the network.

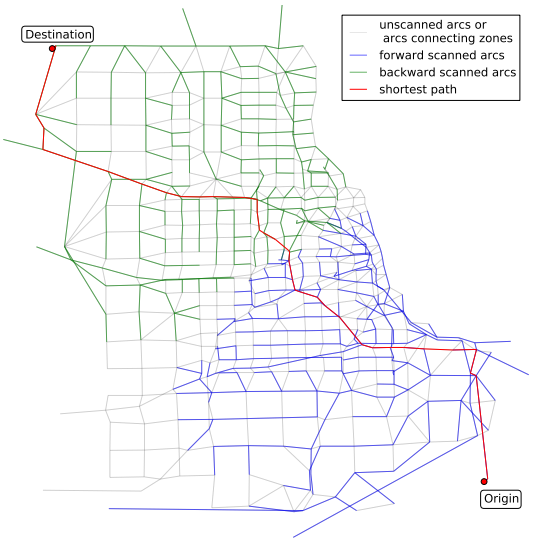
incomplete

draw 2 nodes that are close to each other

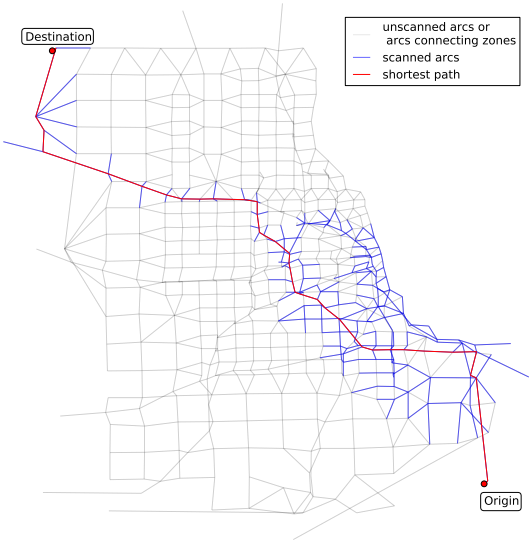
give results interpretation



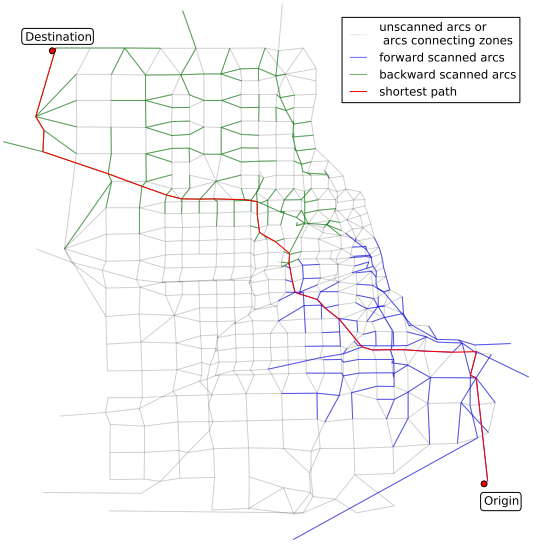
(a) Dijkstra



(b) Bidirectional Dijkstra



(c) A* Search



(d) Bidirectional A* Search

Figure 4.3: Shortest Path Tree between Two Distant Nodes in the ChicagoSketch Network -D Pair

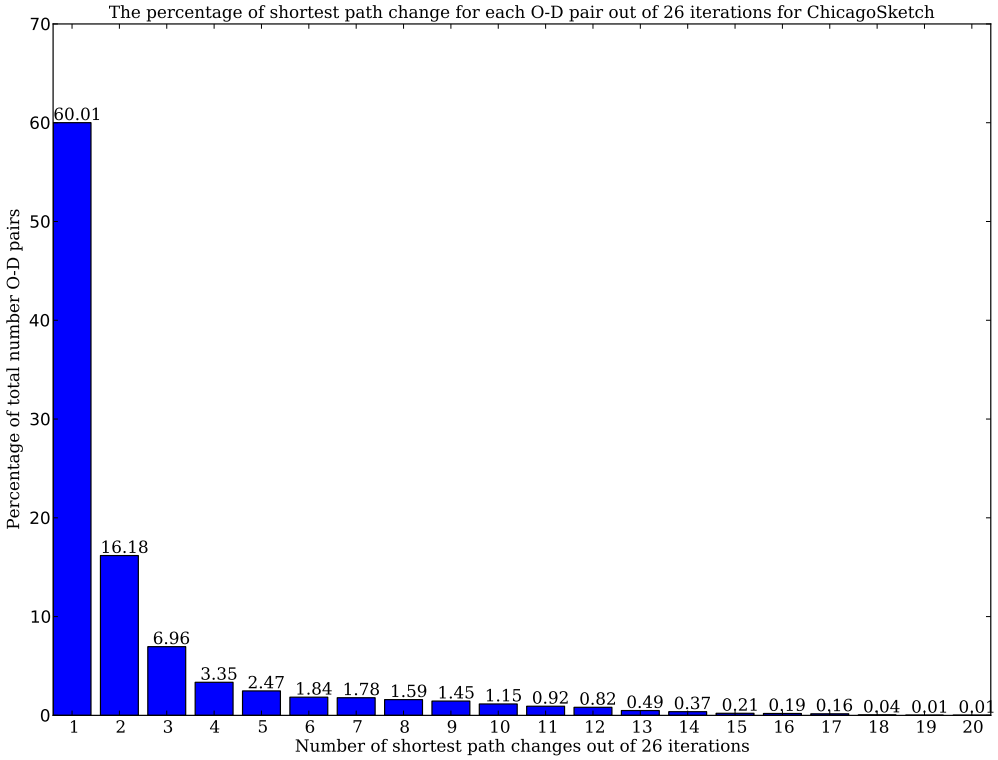


Figure 4.4: The percentage of shortest path change for each O-D pair out of 26 iterations for ChicagoSketch

Chapter 5

Conclusions

A* search out performs all other algorithms ...

Chapter 6

Future Works

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.

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.

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., 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.

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.

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