Engineering Science

Faster Shortest Path Computation for Traffic Assignment

Author:
Boshen Chen

Supervisors:
Dr. Andrea Raith
Olga Perederieieva

May 6, 2013

Abstract

Acknowledgement

I acknowledge ...

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

TODO: italic “names”

1.1 Traffic Assignment Models

In today's world, cities are becoming larger, road networks are becoming more complex, and city designers are facing difficulties of forecasting, building and resigning the roads. Road networks often face congestions that are hard to predict, leading to unpredictable travel times for people to travel from places to places.

forecast model 4 stage zones O-D pair TA

In order to solve these problems, different traffic models are built. One particular model is called the transportation forecasting model. This model solves traffic flows in a typical road network, of which involves four stages of process: trip generation, trip distribution, mode choice and traffic assignment. In short, this model generates traffic demands and supplies in different traffic zones, having them act as origins and destinations for the travellers. this model then selects a transportation method for travellers to use, and sends them to their destination. This final stage of sending travellers to their destination is addressed as the traffic assignment (TA) problem. TA is a complex task, which involves the selection of routes to use by considering information such as congestions.

PE Shortest Path travel time

One method of solving the traffic assignment is called equilibrium assignment or the path equilibration method. In this method, shortest path for every origin-destination pair (O-D pair) are calculated, and traffic flows are assigned to these shortest path. The traffic assignment is said to be solved when equilibrium occur when no traveller can find a shorter path. As every time traffic flows are reassigned, congestion will happen differently and travel time for every path will change respectively, thus a number of iterations is needed to settle the traffic flows to equilibrium. The travel times for the path are no longer expressed solely by their distance, but instead they are expressed by a formulation which considers parameters such as the travel time from the previous iteration, the number of travellers and the capacity of the road.

time consuming

mention Frank-Wolfe algo

Transportation networks are typically very large, which means it may take a very large numbers of iterations for the traffic flows to settle, and meanwhile in each iteration, shortest path for each O-D pair are calculated repeatedly, which is a very time consuming step for large networks with lots of roads and intersections. Speeding up the shortest path calculation would significantly speed up the TA algorithms, as a result, larger and more complex networks can then be solved faster. And when this fast TA algorithm is put back inside the traffic forecasting model, we can predict longer into the future by changing traffic demands and supplies or modifying the road network design.

Chapter 2

Problem Description

In the previous chapter we have briefly talked about the shortest path and the traffic assignment problem. In this chapter, we give formal definition to these concepts, as well as the idea of a data structure.

2.1 Notations and Definitions

TODO ref properly-¿

Using notations from (Klunder & Post 2006) and in the context of transportation networks, we denote $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ for a directed graph, where $\mathcal{V}$ denotes the set of nodes (origins, destinations, and intersections) and $\mathcal{A}$ the set of arcs (roads); we say $\mathcal{A}$ is a subset of the set $\{(u, v) \mid u, v \in \mathcal{V}\}$ of all ordered pairs of nodes. We denote the cardinality of $\mathcal{V}$ be V and $\mathcal{A}$ be A. We assume that $1 \leq V < \infty$ and $0 \leq A < \infty$, and that a function $c : \mathcal{A} \to \mathbb{R}$ is given that assigns a cost (travel time) to any arc $(u, v) \in \mathcal{A}$. We write the costs of arc (u, v) as: $c((u, v)) = c_{uv}$.

The path inside a transportation network has to be a directed simple path, which is a sequence of nodes and arcs $(u_1, (u_1, u_2), u_2, \cdots, (u_{k-1}, u_k), u_k)$ such that $(u_i, u_{i+1}) \in \mathcal{A}$ for $i = 1, \cdots, k-1$ 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. In a transportation network, these O-D pairs are often traffic zones for generating supplies and receiving demands, indicating they are untravellable nodes. Finally we denote cost of the whole path $C(P) := \sum_{(u,v)\in P} c_{vw}$.

2.2 The Shortest Path Problem

SPP

The Shortest Path Problem (SPP) is the problem of finding the shortest path from a given origin s (source) to some destination t (target). There are two types of SPP hat are going to be analysed in this report: a single-source SPP and a point to point SPP. A single-source SPP solves

the shortest path going from one origin to every other destinations in the network, meanwhile a point to point SPP solves from one origin to a specific destination.

2.3 Data Structures

A Data structure is a way for computers to store, update and manipulate intermediate results. A good data structure can efficiently speed up the access of the stored data, which means the performance of solving our shortest path problem does not only depend on the algorithm it self but also the data structure. In this project, two data structures are combined to improve the solving speed:

- Forward Star (for the label correcting algorithm in Section 3.2),
- Heap (for the label setting algorithms in Section 3.3 and onwards).

2.3.1 Forward Star

There is one common characteristic for all of the shortest path algorithms that is going to be discussed in Section 3: all arcs emanating from a given node need to be accessed. An efficient way of visiting every node inside a network and access all their emanating arcs is to use the Forward Star data structure. The advantage of using this data structure for storing the network is that the time complexity for finding any node in the network and iterating through all the emanating arcs is $O(1)$. The exact implementation of Forward Star is not going to be detailed in this report, but the information can be found in Chapter 5.3 of Urban Transportation Networks (Sheffi 1984). This data structure has already been implemented by the second supervisor.

TODO reword

2.3.2 Priority Queue and Heap

A priority queue is a data structure which sorts elements by their priority, element with high priority is always retrieved first before an element with a lower priority. In section 3.3 and on wards, a specific implementation of the priority queue is used: min-heap tree (of which also have different types of implementation). The idea of the min-heap tree is that the values of a parent node is always less or equal to its parent node, by maintaining this property, the minimum valued element will always be on top, and retrieving it has only $O(1)$ time complexity. And for other operations such as adding, removing and updating a node in the heap is at most $O(\log(N))$. So the heap data structure is very efficient at constantly adding a element and finding the current minimum value. The impact and usage of this data structure will be mentioned again for label setting algorithm in section 3.3.

Chapter 3

Solving the Shortest Path Problem

In this chapter we discuss the algorithms that solve the shortest path problem that are applicable for the transportation network.

TODO Big O Analysis

3.1 Generic Shortest Path Algorithm (GSP)

A family of algorithms exist for solving SPP, in this section we describe the generic case for these algorithms.

All of these algorithms depend on solving a label vector $(d_1, d_2, \ldots d_v)$ (Klunder & Post 2006). Each d_v keeps the least distance of of any path going from s to v, $d_v = \infty$ if no paths has been found. A shortest path is optimal when it satisfies the following conditions:

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

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

Bellman's conditions

The inequalities (3.1) is called the Bellman's conditions (Bellman 1958). In other words, we wish to find a label vector d which satisfy the Bellman's conditions for all of the vertices in the graph. To maintain the label vector, the algorithm uses a candidate list $\mathcal{Q}$ 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 candidate list, and labelled when the node has been retrieved from the candidate list and its distance label cannot be updated further.

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 satisfy condition (3.1) to the corresponding node of that arc. Shortest path going from s to all other nodes in $\mathcal{V}$ is found when (3.1) is satisfied for all arcs in $\mathcal{A}$. 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; shortest path can be constructed by following the predecessor of destination node t back to origin node s.

TODO

The following pseudo code describes the generic shortest path algorithm mentioned above, with an extra constraint than a normal GSP: travelling through zone nodes are not allowed.

Algorithm 1 The Generic Shortest Path Algorithm (Klunder & Post 2006)

```
1:  procedure GenericShortestPath(s)
2:      Q ← Q ∪ {s}                                  ▷ Initialisation
3:      p_s ← −1                                     ▷ Origin has no predecessor
4:      d_s ← 0
5:      for all u ∈ V : u ≠ s do                     ▷ All nodes unvisited except the source
6:          d_u ← ∞
7:      while Q ≠ ∅ do
8:          u ← top(Q)                               ▷ get pivot node
9:          u ← Q \ {u}
10:         if u ≠ zone then
11:             for all v : (u, v) ∈ A do            ▷ For all outgoing arcs from 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                ▷ Include node v if unvisited
16:                         Q ← Q ∪ {v}
```

TODO check correctness

Note GSP is the generic case for a family of algorithms that use different implementations of the candidate list $\mathcal{Q}$ (Pallottino & Scutellà 1997), of which solve either the one-source or the point-to-point shortest path problem.

3.2 Label Correcting Algorithm

The GSP is addressed as a label correcting algorithm when the candidate list is changed to a first in first out (FIFO) queue. Given the arc costs can be negative (no negative cycle), and in order to satisfy the Bellman's conditions for all arcs, the algorithm has to scan all arcs in $\mathcal{A}$ $V - 1$ (number of nodes-1) times, giving a time complexity of $O(mn)$.

In this algorithm, the distance labels do not get permanently labelled when a pivot node is retrieved from the queue, 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, Ford and Moore (Bellman 1958, Ford 1956, Moore 1959).

3.3 Label Setting Algorithm

The classical algorithm for solving the single-source shortest path problem is the Label Setting Dijkstra's algorithm. Conceptually the algorithm grows a shortest path tree from the source node radially outward. The algorithm is said to be label setting as when the pivot node is retrieved from the queue, the node gets permanently labelled, the shortest path going to this node is then solved, the distance label on this pivot node gives the length of the shortest path. In order to do this, the priority queue is modified to always have the minimum distance label in front of the queue. Hence the algorithm will iterate through all successive pivot nodes exactly once, labelling pivot nodes in the order of non-decreasing distance labels.

The advantage of this algorithm over the label correcting algorithm is that all nodes are only visited once, and the shortest path tree grows outward radially. Combining these two feature, it is clear that when the pivot node is the destination node and is labelled, we can stop the algorithm for the point to point SPP case, which is desirable for the Path Equilibration method. Modifying Step 1 of the Dijkstra Algorithm gives

Algorithm 2 Point to Point Label Setting Algorithm (Dijkstra)

1: **procedure** DIJKSTRA(s, t)
2: $\mathcal{Q} \leftarrow \{s\}$ ▷ Initialisation
3: $p_s \leftarrow -1$
4: $d_s \leftarrow 0$
5: **for all** $u \in \mathcal{V} : u \neq s$ **do** ▷ All nodes unvisited except the source
6: $d_u \leftarrow \infty$
7: **while** $\mathcal{Q} \neq \emptyset$ **do**
8: $u \leftarrow \text{top}(Q)$
9: $\mathcal{Q} \leftarrow \mathcal{Q} \backslash \{u\}$
10: **if** $u = t$ **then**
11: Terminate Procedure
12: **if** $u \neq$ zone **then**
13: **for all** $v : (u, v) \in \mathcal{A}$ **do** ▷ For all outgoing arcs from u
14: **if** $d_u + c_{vw} < d_v$ **then**
15: $d_v \leftarrow d_u + c_{vw}$
16: $p_v \leftarrow u$
17: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{v\}$

Note a path will not be found if the queue becomes empty, but this stopping condition is safe because we known a path always exist between an O-D pair.

3.3.1 Heap Implementation

Various implementations of the Heap data structure exist, with each implementation have some advantages than the other, for example faster tree balancing, faster push or pop.

We examine 6 different Heap implementations from the C++ Boost Heap Library (Blechmann 2013):

	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))

Table 3.1: C++ Boost Heap Implementations with Comparison of Amortized Complexity

Where N is the number of elements in the Heap tree, and all time complexities are measured in amortized time. (the average run time if the operation is run for a long period of time, average out worse case and best case).

We are interested in using these Heap data structures rather than the standard STL priority queue is because of one reason: the decrease (or increase) function. The decrease (or increase) function is referred as the decrease-key (or increase-key) operation, 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 the 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 (different distance label) in the queue again.

In table 3.1, we can 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. This also applies to all the other operations. And since we do not know the time constant for all the operations, and with different sparsity of the networks, we need to experiment with all of them.

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

TODO actual count

All of these run times are slower than the STL version of the Heap. Upon inspection, it is found that the increase-key operation is used about between 5% to 10% of the time, which means the graphs are not dense enough for these Heap structures to outperform a simple array based priority queue.

3.4 Bidirectional Dijkstra

3.5 A* Algorithm

TODO diagram

Up until now, the 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, heuristic can be used to direct the shortest path tree to grow toward the destination. 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*. Formally we define the following: Let h_v be a heuristic estimate from node v to t, apply Bellman's condition an optimal solution exist, that is $h_v \leq h_u + c_{uv}, \quad \forall (u,v) \in \mathcal{A}$. In other words, the heuristic estimate for each node need to always under estimate the actual distance going from the node to the destination.

It is proven using geographical coordinates and euclidean distance as the heuristic estimate, A* is guaranteed to work with a huge run time speed up by scanning very few nodes.

In our Path Equilibration method, we can no longer use geographical coordinates and euclidean distance for the heuristic estimate, this is because we use travel times as the distances for the arcs. Although distance is included in the calculation, we cannot guarantee it to under estimate the travel times.

By analysing the travel times function (Figure 3.1), we can see that it is a non-decreasing function with the lowest value being the zero flow travel times, which means if we use the zero flow travel times as the heuristic estimate, it is assured that it will always under estimate the travel time for that arc, because no travel time can be lower than the zero flow travel at any time.

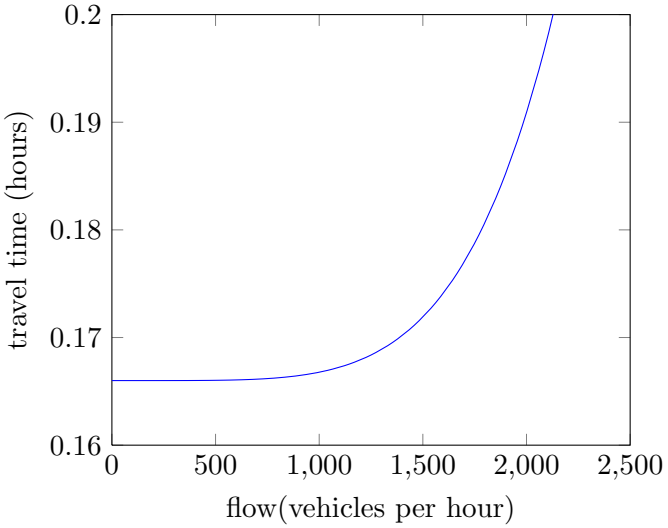

Figure 3.1: Travel time function.

TODO correct diagram show zero flow

Modifying Step 1 of the GSP for A*:

Algorithm 3 A* Algorithm

1: **procedure** ASTAR(s, t)
2: $\mathcal{Q} \leftarrow \{s\}$ ▷ Add node s with $d_s = h_s$
3: $p_s \leftarrow -1$
4: $d_s \leftarrow 0$
5: **for all** $u \in \mathcal{V} : u \neq s$ **do** ▷ All nodes unvisited except the source
6: $d_u \leftarrow \infty$
7: **while** $\mathcal{Q} \neq \emptyset$ **do**
8: $u \leftarrow \text{top}(Q)$ ▷ Remove u such that $d_u + h_u = \min_{v \in \mathcal{Q}}\{d_v + h_v\}$
9: $\mathcal{Q} \leftarrow \mathcal{Q} \backslash \{u\}$
10: **if** $u = t$ **then**
11: Terminate Procedure
12: **if** $u \neq$ zone **then**
13: **for all** $v : (u, v) \in \mathcal{A}$ **do** ▷ For all outgoing arcs from u
14: **if** $d_u + c_{vw} < d_v$ **then**
15: $d_v \leftarrow d_u + c_{vw}$
16: $p_v \leftarrow u$
17: $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{v\}$ ▷ Add node v with $d_v = d_u + c_{vw} + h_v$

Comparing the Dijkstra and A* algorithm's result (Table 4.5 and 4.6), we see an approximately 5 times improvement. By looking at the shortest path tree generated by the ChicagoSketch network, there are only a few scanned nodes, the path goes straight to the destination. (TODO

reference) says the closer the heuristic is to the actual distance, the better/faster shortest path calculation, by looking at the travel time function (Figure 3.1, we can see the slope is really
TODO shallow near the start, and by comparing the initial flow and final flow (TODO, data), they are very close so the final flow is very close to the initial flow, which means the heuristic is a very good estimation, which is our A* is very fast.

3.6 Bidirectional A*

3.7 Preprocessing

Chapter 4

Results

4.1 Problem Data and Result Explanation

talk about existing code

The problem data for solving the TA problems are retrieved from Transportation Network Test Problems (Bar-Gera 2013).

Through out the report, Table 4.1 is used to show the run time and number of iterations for solving one particular network. In the table, the "O-D pairs" column gives the number of pairs of origin and destination in the network. The "zone" column gives the number of traffic zones, in some cases, the nodes in the network also include the traffic zones. The "Run time (seconds)" gives is measured from executing the path equilibration algorithm from start to finish. The "Iterations" column gives how many times the whole network gets solved to settle the traffic flows to equilibrium.

Network	Nodes	Zones	O-D pairs	Arcs	Iterations	Run Time (s)
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		

Table 4.1: Network Problem Data

time per iteration?

By examining the network problem data, we can see that the number of O-D pairs increase significantly respect to the number of zone nodes, this is important because it indicates how many SPPs need to be solved for each iteration of the PE. We can also roughly tell that these networks are very sparse, as a complete graph (every node is connected to every other node) of 1000 nodes have 499500 arcs ($n(n-1)/2$), and the larger networks in our problem data only have about 0.4% to 0.6% of arcs in a complete graph, this information is useful when we start tuning the algorithms for solving SPP.

Network	Nodes	Zones	O-D Pairs	Arcs	Iterations	Run Time (s)
SiouxFalls	24	24	528	76	69	0.25
Anaheim	416	38	1406	914	10	1.20
Barcelona	1020	110	7922	2522	28	60.00
Winnipeg	1052	147	4344	2836	129	190.00
ChicagoSketch	933	387	93135	2950	25	500.00

Table 4.2: One Source Label Correcting Algorithm Result

Network	Nodes	Zones	O-D Pairs	Arcs	Iterations	Run Time (s)
SiouxFalls	24	24	528	76	64	0.24
Anaheim	416	38	1406	914	10	1.20
Barcelona	1020	110	7922	2522	27	43.00
Winnipeg	1052	147	4344	2836	129	137.00
ChicagoSketch	933	387	93135	2950	25	541.00

Table 4.3: C++ STL One-Source Label Setting Algorithm Result

Most of the data does not resemble a real world transportation network, for example sometimes all roads have the same speed limit, road type and capacity.

In this report, all problem data are solved on a Intel i5 1.78GHz CPU computer with 4GB RAM, which runs the Ubuntu 12.04 Linux operating system. And the code is compiled with the g++ compiler with the -O3 optimisation flag (i.e. optimise for speed).

CPU param

The accuracy of all results are checked by comparing the traffic flows from the traffic assignment output, as well as the final shortest path for every O-D pairs.

Using the standard C++ standard template library (STL) priority queue (implemented as a Heap tree) using std::vector as the underlying storage, the following results are generated.

The following table shows the result for point to point Dijkstra's algorithm.

Network	Nodes	Zones	O-D Pairs	Arcs	Iterations	Run Time (s)
SiouxFalls	24	24	528	76	64	0.15
Anaheim	416	38	1406	914	10	0.67
Barcelona	1020	110	7922	2522	27	27.71
Winnipeg	1052	147	4344	2836	129	70.00
ChicagoSketch	933	387	93135	2950	25	204.00

Table 4.4: Point to Point Label Setting Algorithm (Dijkstra) Result

The following table shows the run time and iterations for all the networks.

Network	Iterations	Binary	Ternary	Binomial	Fibonacci	Pairing	Skew
SiouxFalls	85	0.17	0.17	0.29	0.18	0.17	0.16
Anaheim	10	0.88	0.81	2.12	1.05	1.02	0.83
Barcelona	27	34.00	33.00	85.00	46.00	44.00	34.00
Winnipeg	128	83.00	86.00	202.00	107.00	97.00	83.00
ChicagoSketch	26	233.00	229.00	472.00	264.00	231.00	209.00

Table 4.5: Point to Point C++ Boost Label Setting Algorithm (Dijkstra) Result

Results:

Network	Iterations	STL	Binary	Ternary	Binomial	Fibonacci	Pairing	Skew
SiouxFalls	85	0.16	0.14	0.16	0.22	0.22	0.14	0.14
Anaheim	10	0.15	0.19	0.19	0.33	0.22	0.18	0.17
Barcelona	27	5.44	6.53	6.54	11.45	7.62	6.56	6.10
Winnipeg	128	19.49	24.34	24.86	44.41	27.93	24.23	21.85
ChicagoSketch	26	38.92	46.00	44.00	78.02	53.28	45.10	42.90

Table 4.6: A* Algorithm Result

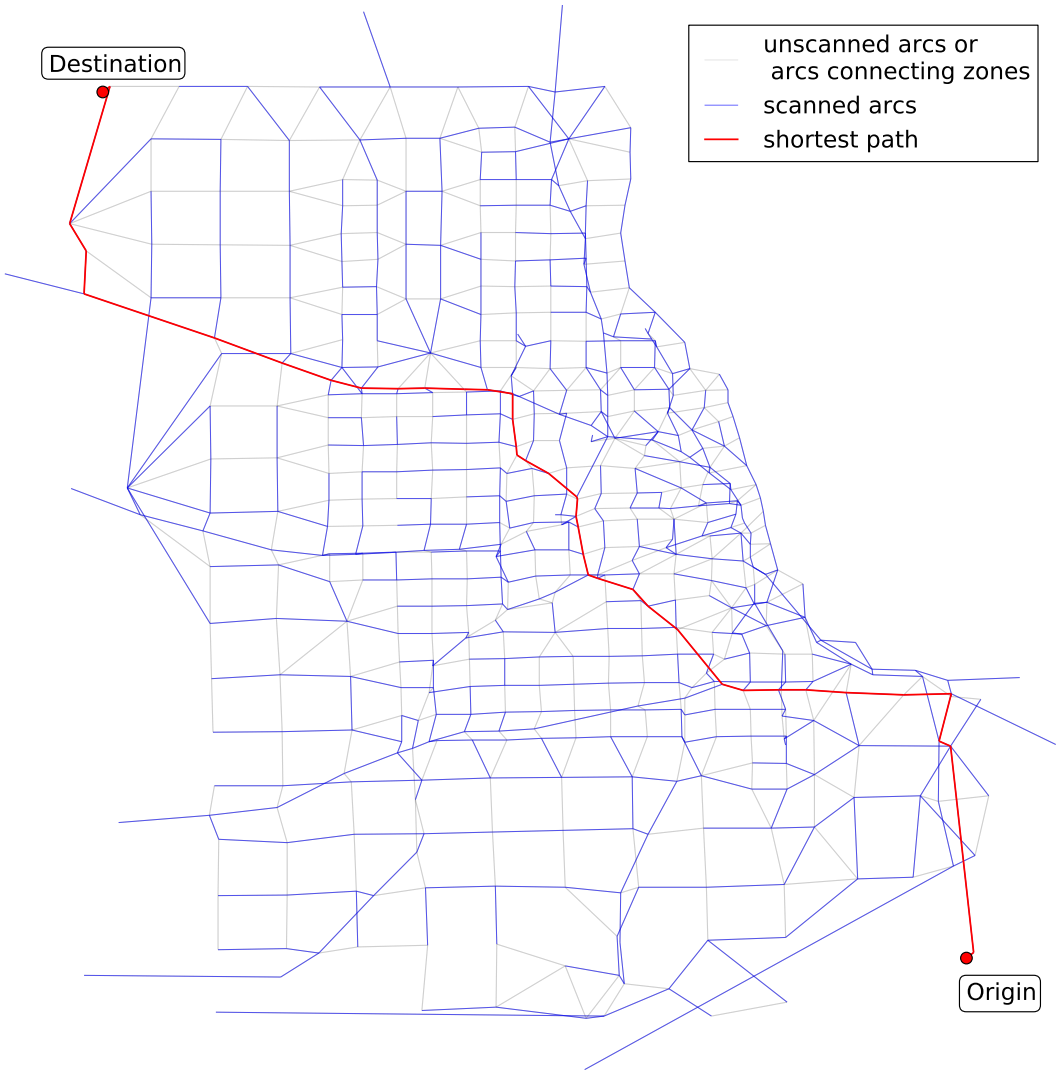

Figure 4.1: Dijkstra Path Tree for ChicagoSketch network

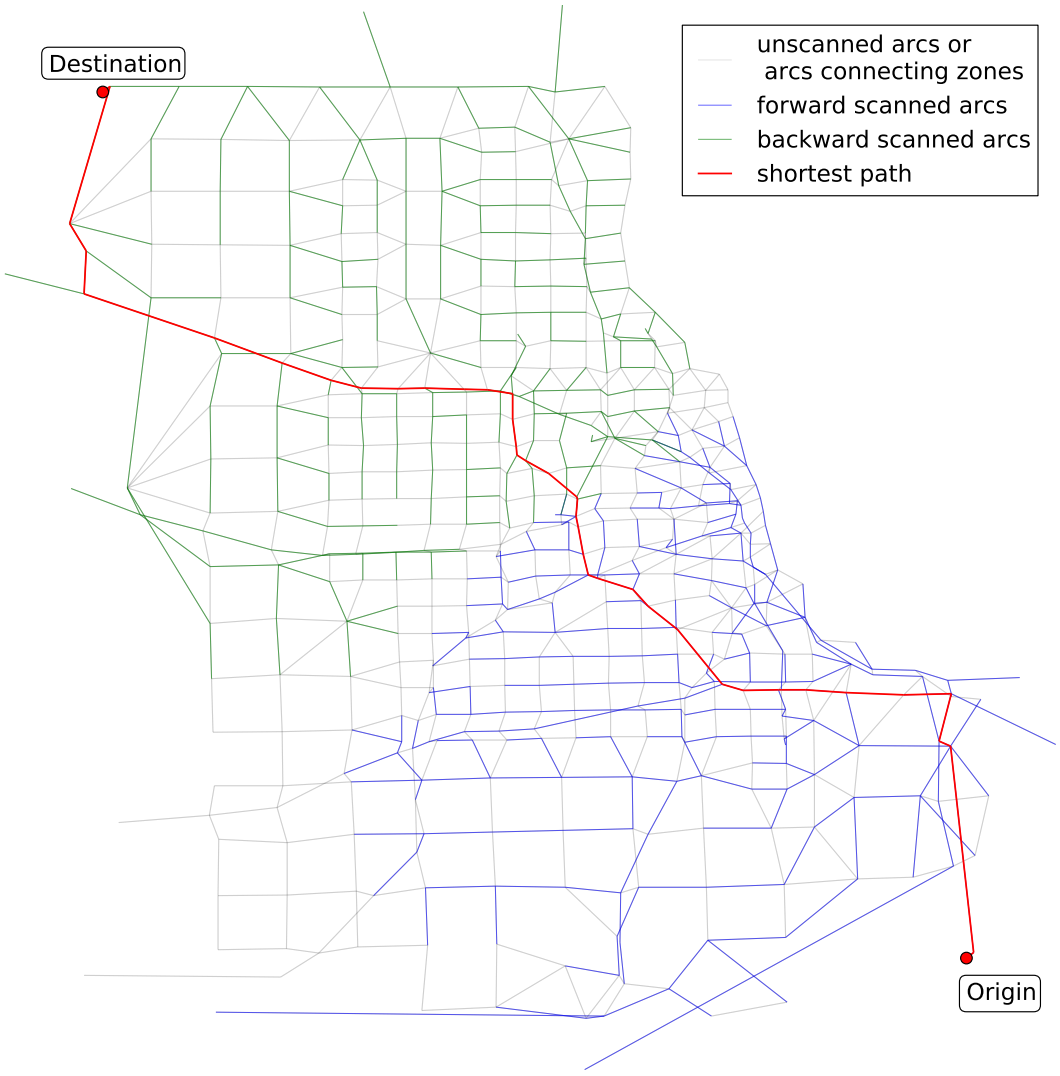

Figure 4.2: Bidirectional Dijkstra Shortest Path Tree for ChicagoSketch network

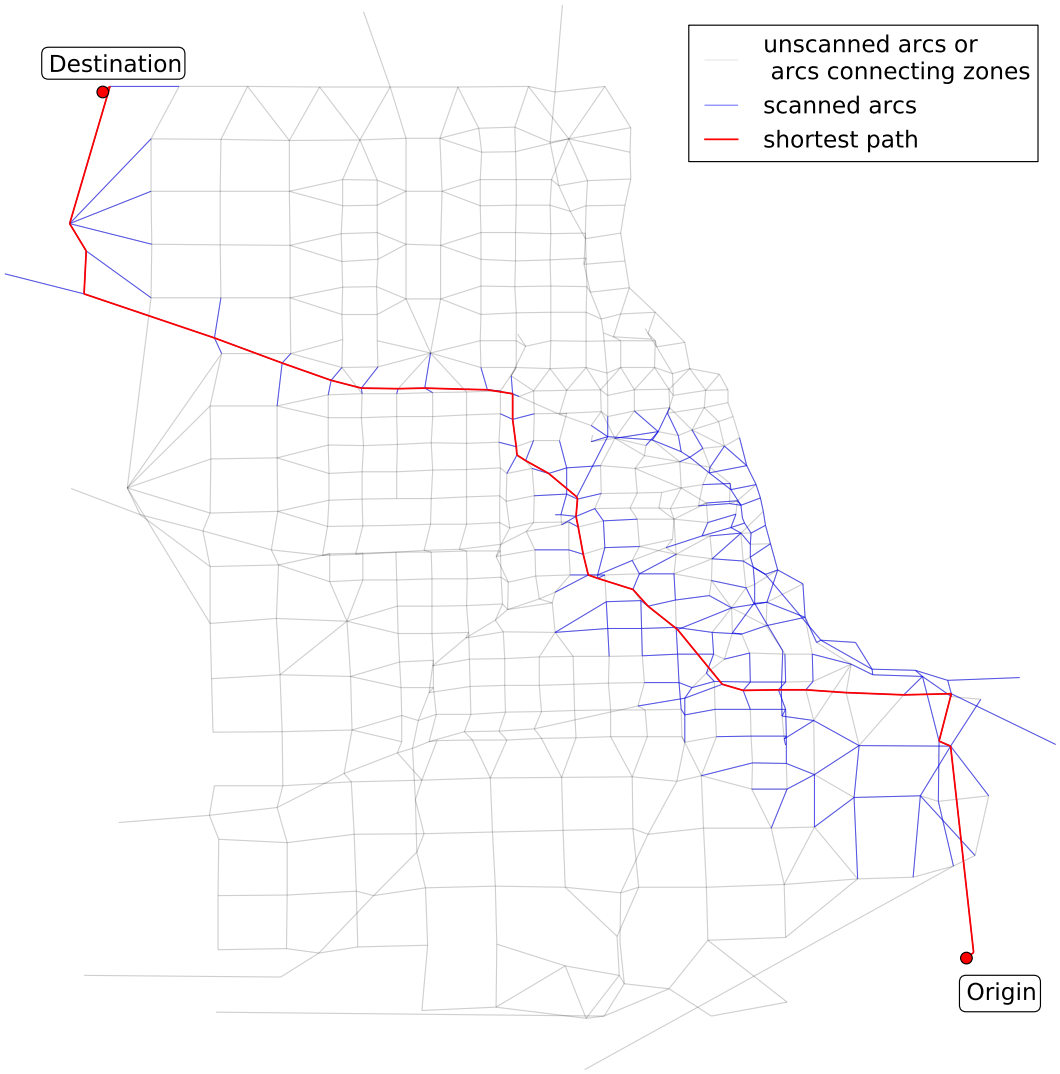

Figure 4.3: A* Shortest Path Tree for ChicagoSketch network

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`.

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

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.

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

Sheffi, Y. (1984), *Urban Transportation Networks: Equilibrium Analysis With Mathematical Programming Methods*, Prentice Hall.
URL: *http://web.mit.edu/sheffi/www/urbanTransportation.html*