THE UNIVERSITY OF AUCKLAND
NEW ZEALAND

Engineering Science

Faster Shortest Path Computation for Traffic Assignment

Author:
Boshen Chen

Supervisors:
Dr. Andrea Raith
Olga Perederieieva

May 24, 2013

Abstract

Acknowledgement

I acknowledge . . .

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

CHECK LATEX LOG!!

probability return infinite

italic jargons

write chapter outline brief

1.1 Project Motivation

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

show forecast model figure

A traffic model called the transportation forecasting model is built with the aim of reducing congestion and predicting future traffic response when the behaviour of the traffic is changed. This model estimates traffic flows with the following four stages: trip generation, trip distribution, mode choice and traffic assignment. In short, this model generates origins and destinations (or traffic analysis zones) 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.

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

ref user equilibrium - John Glen Wardrop principles of equilibrium

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 means 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 zero-flow travel times, traffic flows are then assigned to these shortest paths and updates the new travel times accordingly. New shortest paths are re-identified and travel flows are re-assigned until user equilibrium is reached.

write about Frank-Wolfe

Both of these methods are iterative methods that require many shortest paths for each trip origin in the network. Sheffi (1985) states that finding the shortest path is the most computation-intensive component of each iteration compared to other components such as updates and convergence checks that require no more than a few percentages of the total running time. Thus speeding up the shortest path calculation will significantly speed up the traffic assignment algorithms. As a result, traffic forecasting is solved faster for larger and more complicated road networks, predicting longer into the future and allow better designed roads.

1.2 Project Aims

This project aims to embed well known shortest path algorithms in the traffic assignment methods and find the fastest algorithm. 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.

incomplete

This project also aims to find and discuss the possibility of preprocessing the network or using data calculated from the previous iteration in traffic assignment methods such that the shortest path algorithms have more information to speed up their calculations.

1.3 Report Overview

incomplete

This report continues in Chapter 2 with the theory behind finding the shortest path under different conditions, and also the description, analysis and pseudocode for each algorithm mentioned in the project aims. Chapter 3 presents the specific implementation details used to give the fastest algorithm possible. Chapter 4 shows and illustrates the results from each algorithm mentioned in the project aims.

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 in different situations. In this chapter, notations and definitions for the shortest path problem is stated first, the theory for solving the shortest path problem is described next, algorithms that are applicable for road networks are then summarised, including the discussion of their advantages and drawbacks.

Big O analysis for all algorithms

Need to talk about results, what should the reader pay attention to? What should they conclude?

assume a path always exist between an OD pair

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 hat 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 of shortest path going from one origin to every other destinations 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.

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 $\mathcal{P}$ inside a transportation network has to be a directed simple path, which is a sequence of nodes and arcs $(u_1, (u_1, u_2), u_2, \cdots, (u_{k-1}, u_k), u_k)$ 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 $\mathcal{P}$, u_1 and u_k together is called an O-D pair for this path.

In a transportation network, the origins and destinations are often called centroids or zones. They are traffic analysis zones for generating trip demands and supplies and hold information such as household income and employment information, these information helps the understanding of 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.

Maybe a picture of the network explain what the zones are.

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 nonlinear monotonic increasing travel time function, encapsulating information such as traffic flow, road capacity and travel speed. This travel time function is always non-negative so taking advantage of this helps the selection of better algorithms that uses this property.

2.2 Generic Shortest Path Algorithm (GSP)

state SPP from node s to node t

explain pivot node

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

All of these algorithms aim 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 paths has been found. A shortest path is optimal when it satisfies the following conditions:

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

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

what is $\mathcal{P}$?

The inequalities (2.1) is called Bellman's conditions (Bellman 1958). In other words, we wish to find a label vector d which satisfies 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.

this means it 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 $\mathcal{V}$ is found when (2.1) 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.

diagram showing u, v, c_{vw} etc.

Algorithm 1 describes the generic shortest path algorithm mentioned above, with an extra constraint required when solving a TA problem: travelling through zone nodes are not allowed.

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

procedure GenericShortestPath(s)
  $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{s\}$ ▷ initialisation
  $p_s \leftarrow -1$ ▷ origin has no predecessor
  $d_s \leftarrow 0$
  for all $u \in \mathcal{V} : u \neq s$ **do** ▷ all nodes are unvisited except the source
    $d_u \leftarrow \infty$
  while $\mathcal{Q} \neq \emptyset$ **do**
    $u \leftarrow \text{top}(\mathcal{Q})$
    $u \leftarrow \mathcal{Q} \setminus \{u\}$
    if $u \neq \text{zone}$ **then**
      for all $v : (u, v) \in \mathcal{A}$ **do** ▷ for all outgoing arcs from u
        if $d_u + c_{vw} < d_v$ **then**
          $d_v \leftarrow d_u + c_{vw}$
          $p_v \leftarrow u$
          if $v \notin \mathcal{Q}$ **then** ▷ add node v to queue if it is unvisited
            $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{v\}$

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

2.3 Label Correcting Algorithm

pseudo code

Check if its FIFO or double ended queue

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).

say Dijkstra discovered this

2.4 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. Algorithm 2 shows the Dijkstra's algorithm where the algorithm can be terminated in line 10.

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 OD pair.

2.4.1 Priority Queue and Heap

how do you deal with one node is STL multiple times

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. 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.

Algorithm 2 Point to Point Label Setting Algorithm (Dijkstra)

procedure DIJKSTRA(s, t)
  $\mathcal{Q} \leftarrow \{s\}$ ▷ initialisation
  $p_s \leftarrow -1$
  $d_s \leftarrow 0$
  for all $u \in \mathcal{V} : u \neq s$ **do** ▷ all nodes are unvisited except the source
    $d_u \leftarrow \infty$
  while $\mathcal{Q} \neq \emptyset$ **do**
    $u \leftarrow \text{top}(Q)$ ▷ get the node with the minimum distance label
    $\mathcal{Q} \leftarrow \mathcal{Q} \backslash \{u\}$
    if $u = t$ **then**
      Terminate Procedure
    if $u \neq$ zone **then**
      for all $v : (u, v) \in \mathcal{A}$ **do** ▷ For all outgoing arcs from u
        if $d_u + c_{vw} < d_v$ **then**
          $d_v \leftarrow d_u + c_{vw}$
          $p_v \leftarrow u$
          $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{v\}$

2.4.2 Heap Implementation

explain how this relates to previous results (one is STL and one is Boost, need to be much clearer

what are these procedures?

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):

Table 2.1: 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))

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.

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 with different distance label in the queue again.

In table 2.1, 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.

more details about Fibonacci, why good in theory not in practice

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

not actual count yet

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.

2.5 Bidirectional Dijkstra

2.6 A* Algorithm

describe in LP sense, solving the dual, potentials etc.

potential instead of heuristic?

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, a 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* search. Formally we define the following: Let h_v be a heuristic estimate from node v to t, we apply Bellman's condition such that 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.

comment on gurantee of optimal shortest path

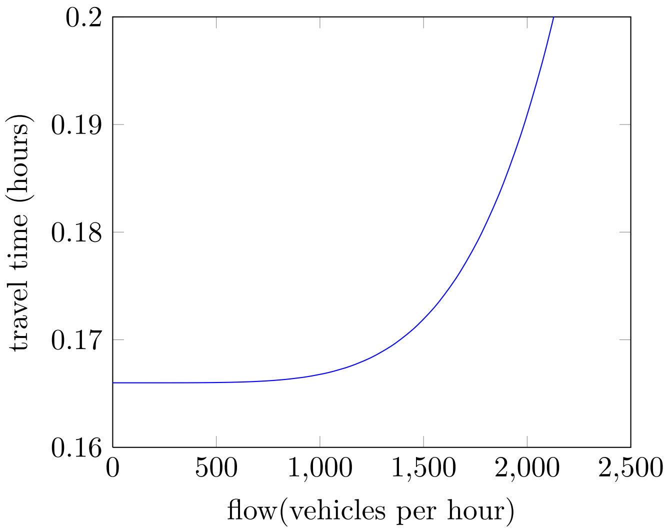
reference!

It is proven using geographical coordinates and Euclidean distance as the heuristic estimate, A* search 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.

could we use Euclidea distance * min speed ?

By analysing the travel times function (Figure 2.1), 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.



comment and correct graph

Figure 2.1: Travel time function.

Modifying Step 1 of the GSP for A* search:

where are my h? heuristic?

Algorithm 3 A* Search Algorithm

procedure ASTAR(s, t)
    $\mathcal{Q} \leftarrow \{s\}$ ▷ Add node s with $d_s = h_s$
    $p_s \leftarrow -1$
    $d_s \leftarrow 0$
    for all $u \in \mathcal{V} : u \neq s$ **do** ▷ All nodes unvisited except the source
        $d_u \leftarrow \infty$
    while $\mathcal{Q} \neq \emptyset$ **do**
        $u \leftarrow \text{top}(Q)$ ▷ Remove u such that $d_u + h_u = \min\limits_{v \in \mathcal{Q}}\{d_v + h_v\}$
        $\mathcal{Q} \leftarrow \mathcal{Q} \backslash \{u\}$
        if $u = t$ **then**
            Terminate Procedure
        if $u \neq$ zone **then**
            for all $v : (u, v) \in \mathcal{A}$ **do** ▷ For all successors of u
                if $d_u + c_{vw} < d_v$ **then**
                    $d_v \leftarrow d_u + c_{vw}$
                    $p_v \leftarrow u$
                    $\mathcal{Q} \leftarrow \mathcal{Q} \cup \{v\}$ ▷ Add node v with $d_v = d_u + c_{vw} + h_v$

2.7 Bidirectional A*

2.8 Preprocessing

Chapter 3

Implmentation Details

Chapter 4

Results

results interpretation

results speed up in percentages

talk about existing code

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).

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 "OD 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 whole 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.

need to explain what an iteration where it is introduced

Table 4.1: Network Problem Data

Network	Nodes	Zones	OD pairs	Arcs
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

the number of nodes listed in the table includes traffic zones

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 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.

mention node degree

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.

wrong, the smaller data sets have same data but not the large ones

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).

rewrite, this is wrong

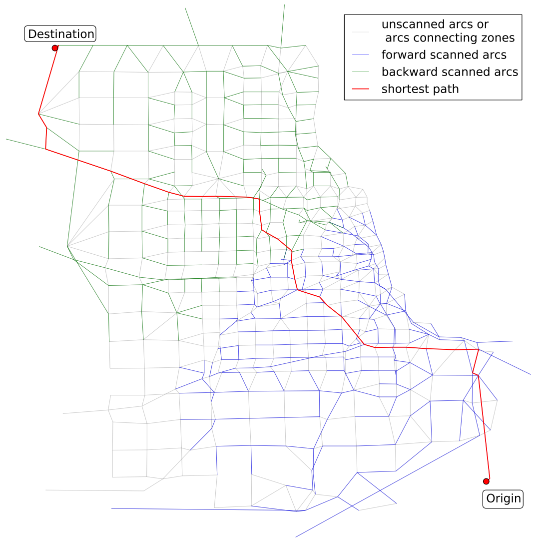
The accuracy of all results are checked by comparing the traffic flows from the traffic assignment output, as well as the final shortest path for every OD pairs.

Table 4.2: Results for all test networks. Showing the number of iterations per graph (ITERS), max number of scans (COUNT) and the speed up respect to the label correcting algorithm (SPD).

			Max Scans		Time	
Graph	Algorithm	ITERS	COUNT	SPD	SEC	SPD
SiouxFalls	B	69			0.25	
	AP-D	69			0.24	
	P2P-D	64			0.15	
	Bi-D					
	A*	85			0.16	
	Bi-A*					
Anaheim	B	10			1.20	
	AP-D	10			1.20	
	P2P-D	10			0.67	
	Bi-D					
	A*	10			0.15	
	Bi-A*					
Barcelona	B	28			60.00	
	AP-D	28			43.00	
	P2P-D	27			27.71	
	Bi-D					
	A*	27			6.10	
	Bi-A*					
Winnipeg	B	129			190.00	
	AP-D	129			137.00	
	P2P-D	129			70.00	
	Bi-D					
	A*	128			21.85	
	Bi-A*					
ChicagoSketch	B	25			500.00	
	AP-D	25			541.00	
	P2P-D	25			204.00	
	Bi-D					
	A*	26			42.90	
	Bi-A*					

Result : average number of scans

(a) Dijkstra



(b) Bidirectional Dijkstra

(c) A* Search

(d) Bidirectional A* Search

Figure 4.1: Shortest Path Tree for ChicagoSketch Network with Two Distant OD Pair

2 nodes close to each other

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`.

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.
URL: *http://www.sciencedirect.com/science/article/B7P6G-4FKY22V-S/1/2d71e5d83c74e838b7b6b583a3c488e9*

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. (1985), *Urban Transportation Networks: Equilibrium Analysis With Mathematical Programming Methods*, Prentice Hall.
URL: *http://web.mit.edu/sheffi/www/urbanTransportation.html*