Engineering Science

Faster Shortest Path Computation for Traffic Assignment

Author:
Boshen Chen

Supervisors:
Dr. Andrea Raith
Olga Perederieieva

May 2, 2013

Abstract

One of the major concerns of commuters in Auckland (and all over the world!) is road traffic and traffic congestion. Both the design of the road network and determining the best way to use it are complex questions that can be addressed based on so-called traffic assignment (TA) models, which are network equilibrium problems. These models can be used to assess the benefit of road upgrades or adding new roads to the network. A TA problem models the route choice of users of a network with nonlinear travel time functions to capture congestion effects as more traffic flow on a road means slower travel time. The aim of TA is to identify how many network users end up choosing to travel along each individual arc in the network. This is done by assuming network users are selfish and choose to travel along their shortest path between origin and destination. In solving such a TA equilibrium problem, an iterative approach is used. Initially all travel demand is assigned to the shortest paths computed for between each trip origin and destination based on zero-flow travel times. Then, travel times are updated based on the new flows. New shortest paths are identified and travel demand is re-assigned. This process continues until it eventually reaches the equilibrium solution. The iterative scheme used by TA algorithms requires many shortest path calculations. Speeding up these shortest path calculations can lead to a significant speed up of TA algorithms, but the speed up technique used would differ for particular traffic assignment algorithms. One idea is to use A* search for TA algorithms that repeatedly compute shortest paths between single origin destination pairs. A* should lead to a significant improvement in runtime over using a standard label setting algorithm. Other speed-up techniques for shortest path algorithms will be identified and tested, exploiting the fact that while shortest paths change in every iteration of the algorithm, it may be possible to avoid fully re-computing shortest path trees. Avoiding re-computation of shortest path trees should benefit TA algorithms that require shortest paths between a single origin and all destinations.

I acknowledge ...

I acknowledge Olga for her bug-free, no momory leak, well constructed, robust, self documented OOP C++ code ... as well her patience for my poorly written shortest paths codes...

Contents

List of Figures

List of Tables

Chapter 1

Introduction

SPP has been mentioned countless times, most materials are very repetitive, what kind of info do i need to make a reference of?

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

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 theses 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.

travel
time

time consuming

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, when everything is sped up, 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.

existing code

Existing computer code (written in C++) already exist for the TA algorithms, including the path equilibrium method and many others. A simple shortest path calculation algorithm is currently used by the path equilibrium method, which runs very poorly given a small network.

zone

The existing algorithm is referred as the label-correcting algorithm, it is modified and implemented as the well known label-setting Dijkstra's algorithm using a special data structure for storing information, A* algorithm is them implemented to speed the Dijkstra's algorithm for the point to point shortest problem.

no travel

Special cares are taken when implementing these algorithms, namely the origins and destinations mentioned sometimes do not act as intersections for the travellers (to be included in the shortest path calculation), since these origins and destinations are conceptual traffic zones for generating demands and supplies.

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 an data structure. The network problem data that is going to be tested on is also described in this chapter.

2.1 Notations and Definitions

In the context of transportation networks, we use the notation $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ for a directed graph, where $\mathcal{V}$ denotes the set of nodes (origins, destinations, and intersections) and $\mathcal{A}$ the set of arcs (roads); so $\mathcal{A}$ is a subset of the set $\{(u, v) \,|\, u, v \in \mathcal{V}\}$ 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 called an O-D pair; in a transportation network, these O-D pairs are often traffic zones for generating supplies and receiving demands, this means the nodes for these O-D pairs are not travelable, traffic flows cannot go through these zone nodes. Finally we denote cost of the whole path $C(P) := \sum_{(u,v)\in P} c_{vw}$.

TODO diagram explain

Shortest Path Problem

The Shortest Path Problem (SPP) is the problem of finding the distance for a given origin s (source) and a destination t (target). We assume the graph contains a path from node s to node t, as well as all arc lengths are positive. For a real transportation network, all these assumptions are satisfied naturally; any transportation networks will have at least one O-D pair and all arc lengths are travel times that are naturally positive.

SPP

Adapted from (How do I reference for all these paragraphs?)

```
The Shortest Path Problem on Large-Scale Real-Road Networks
NETWORKS2006DOI 10.1002/net
```

There are two types of SPP that 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 all other destinations in the network, meanwhile a point to point SPP solves from one origin to a specific destination.

Traffic Assignment

Traffic Assignment (TA) is the problem of selecting paths between origins and destinations in a transportation network, and identify how many travellers use each path. The difficulties of TA lies within a realistic model of the travel times, travel times are modelled by a non-linear function to capture congestion effects: more traffic flow means more congestion leading to slower travel times.

TA

Path Equilibration

mention user/system optimal?

Path Equilibration (PE) is a method for solving the TA problem. This method assumes all travellers are selfish and will always find the shortest (least travel time) path. Initially all travel demand is assigned to the shortest path between each O-D pair based on zero-flow travel times. Then the travel times are updated iteratively based on the new flows, assigning new shortest path and new flows in each iteration, eventually the travel times will reach equilibrium and no better shortest path can be calculated, resulting optimality. For each iteration of PE, the point to point SPP is solved for all O-D pairs in the network.

PE

2.1.1 Forward Star

Data structure for storing the network, useful for label correcting algorithm.

2.1.2 Priority Queue

A priority queue is a data structure which sorts elements by its priority, element with high priority is always retrieved first before an element with lower priority. Useful for label setting algorithm.

2.2 Problem Data and Result Explanation

Data used for solving the TA problems are retrieved from (http://www.bgu.ac.il/ bargera/tntp/),

Through the report the following Table is used to show the run time and number of iterations took. O-D pairs are the pair of origin and destination pair for the zone nodes in the network, zones are sometimes only a origin or a destination, thus we Run time (seconds) is measured from

executing the path equilibration algorithm. Iterations is how many times the whole network get solved to settle the traffic flows to equilibrium.

Network	Nodes	Zones	O-D pairs	Arcs	Run Time (s)	Iterations
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 2.1: Network Problem Data

The nodes the table include the traffic zones. 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.

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

All results are from an Intel i5 1.78ghz CPU computer, 4GB RAM, running Ubuntu 12.04 Linux. Code is compile with g++ with -O3 optimisation flag.

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.

Chapter 3

Solving the Shortest Path Problem

In this chapter we will solve the shortest path problem for transportation networks and sequentially improve the existing shortest path algorithm implemented. Run time improvement will be shown and discussed.

TODO

TODO: run time (Big O) analysis for all algorithms.

3.1 General Shortest Path Algorithm

Most algorithms for SPP depend on solving a label vector $(d_1, d_2, \ldots d_v)$.

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 Bellman's 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

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. 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 general 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., $d_w > d_u + c_{uv}$), 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 $\mathcal{A}$. 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

TODO: better format add diagram showing u, v, c_{uv} etc.

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

```
Initially,
    Q:={s}, p_s:=-1, d_s:=0,d_u:=\infty, \forall u \neq s.
Step 1: Node selection
    Remove a node u from the candidate list Q.
    Node u is called the pivot node.
Step 2: Label update
    if u is not a zone node then
        for each outgoing arc (u,v) \in \mathcal{A} do
            if d_u + c_{uv} < d_v then
                d_v := du + c_{uv}
                p_v := u
                if v \notin Q then
                    add v to Q
                end if
            end for
    end if
Step 3: Stop condition
    if Q \neq \emptyset
        goto Step 1
    else
        terminate algorithm
```

3.2 Label Correcting Algorithm

The existing shortest path algorithm in the traffic assignment code is called the label correcting algorithm. The code is adapted from (Sheffi 1984), which uses a first in first out queue for the candidate list. The algorithm maintains the pivot node in step 1 of GSP in a way such that it is always the top node, utilising the forward star data structure for storing the network mentioned in Chapter 2.1.1.

TODO TODO: pseudo code?

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.

Network	Nodes	Zones	O-D Pairs	Arcs	Run Time (s)	Iterations
SiouxFalls	24	24	528	76	0.25	69
Anaheim	416	38	1406	914	1.2	10
Barcelona	1020	110	7922	2522	60	28
Winnipeg	1052	147	4344	2836	190	129
ChicagoSketch	933	387	93135	2950	500	25

Table 3.1: Label Correcting Algorithm Result

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.

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.

Network	Nodes	Zones	O-D Pairs	Arcs	Run Time (s)	Iterations
SiouxFalls	24	24	528	76	0.24 ± 0.02	64
Anaheim	416	38	1406	914	1.2 ± 0.2	10
Barcelona	1020	110	7922	2522	43	27
Winnipeg	1052	147	4344	2836	137	129
ChicagoSketch	933	387	93135	2950	541	25

Table 3.2: Label Setting Algorithm (Dijkstra) Result

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

Thus we can change the stop condition in the GSP:

```
Step 3: Stop condition
    if u is the destination node
        terminate algorithm
    end if
    if Q \neq \emptyset
        goto Step 1
    end if
```

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.

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

Network	Nodes	Zones	O-D Pairs	Arcs	Run Time (s)	Iterations
SiouxFalls	24	24	528	76	0.15 ± 0.02	64
Anaheim	416	38	1406	914	0.67 ± 0.2	10
Barcelona	1020	110	7922	2522	27.71	27
Winnipeg	1052	147	4344	2836	70	129
ChicagoSketch	933	387	93135	2950	204	25

Table 3.3: Point to Point Label Setting Algorithm (Dijkstra) Result

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:

	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 3.4: 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.

Thus we change Step 2 of the GSP

```
Step 2: Label update
    if u is not a zone node then
        for each outgoing arc (u,v) \in \mathcal{A} do
```

```
            if d_u + c_{uv} < d_v then
                d_v := du + c_{uv}
                p_v := u
                if v \notin Q then
                    add v to Q
                else
                    decrease-key(v)
                end if
            end for
    end if
```

In table 3.4, 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

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

Network	Binary	Ternary	Binomial	Fibonacci	Pairing	Skew	Iterations
SiouxFalls	0.17	0.17	0.29	0.18	0.17	0.16	85
Anaheim	0.88	0.81	2.12	1.05	1.02	0.83	10
Barcelona	34	33	85	46	44	34	27
Winnipeg	83	86	202	107	97	83	128
ChicagoSketch	233	229	472	264	231	209	26

Table 3.5: Label Setting Algorithm (Dijkstra) Result

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 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.

TODO shotest path tree

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.

Need Proof?

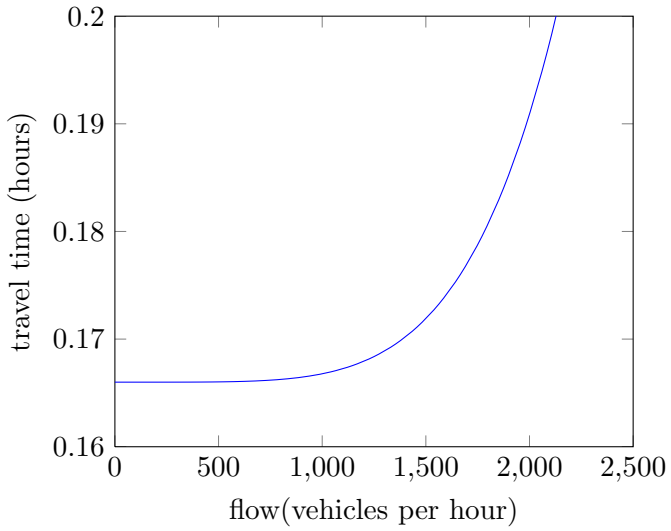

Figure 3.1: Travel time function.

TODO correct diagram show zero flow

Modifying Step 1 of the GSP for A*:

```
Step 1: Node selection
    Remove a node u from the candidate list Q
    such that
    d_u + h_u = \min_{v \in Q} \{ d_v h + h_v \}
```

Results:

Network	STL	Binary	Ternary	Binomial	Fibonacci	Pairing	Skew	Iterations
SiouxFalls	0.16	0.14	0.16	0.22	0.22	0.14	0.14	85
Anaheim	0.15	0.19	0.19	0.33	0.22	0.18	0.17	10
Barcelona	5.44	6.53	6.54	11.45	7.62	6.56	6.1	27
Winnipeg	19.49	24.34	24.86	44.41	27.93	24.23	21.85	128
ChicagoSketch	38.92	46	46	78.02	53.28	45.1	42.9	26

Table 3.6: Label Setting Algorithm (Dijkstra) Result

Comparing the Dijkstra and A* algorithm's result (Table 3.5 and 3.6), we see an approximately 5 times improvement. By looking at the shortest path tree generated by (TODO:one of the) 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 shallow near the start, and by comparing the initial flow and final flow, the 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.

TODO

TODO TODO: is it still a sp tree? Draw sp tree for A*.

3.4.1 Bidirectional Dijkstra

3.4.2 Bidirectional A*

Appendix A

Code

Appendix B

More Code

Bibliography

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