# Finding Multi-criteria Optimal Paths in Multi-modal Public Transportation Networks using the Transit Algorithm

**Leonid Antsfeld** **Toby Walsh**
School of Computer Science and Engineering, UNSW, Sydney Australia
NICTA, Sydney, Australia
Leonid.Antsfeld@nicta.com.au Toby.Walsh@nicta.com.au

**Abstract**

We present an algorithm to find optimal routes in a multi-modal public transportation network. Our model takes into account many realistic features such as walking between multi-modal stations, transfer times, traffic days, multiple objectives and finding connections between geographical locations rather than just source and destination stations. In order to provide useful routing directions, we consider the robustness of the provided solutions. In addition we present numerous speed up techniques that reduce both the preprocessing time and storage. Our preliminary experiments on the Sydney transit network are promising.

**Keywords:** shortest path algorithm, public transport network, intermodal journey planner, trip planning

## 1 Introduction

There exist many different systems that provide users with transit information, e.g. NSW TranportInfo [8], Google Transit [6]. Such systems are often available as a Web or smart phone application. The application ask a user to input an origin, destination and expected departure or arrival time and provides the user with recommended travel routes. Such systems are useful in encouraging people to switch from their private cars to use public transport services, thus reducing congestion, CO2 emission and providing the travelers a better experience.
We present here an algorithm to find optimal paths in a multi-modal public

transportation network. Our algorithm extends the Transit algorithm [3, 4] using an improved time expanded graph of the multi-modal public network. The original Transit algorithm is one of the best methods for finding shortest paths in very large road networks, [5], but was previously limited to a single mode of transport and static and undirected graphs. The nature of public network is very different from road network in many ways, e.g., links are directional, time/day dependent, etc. In addition, the number of nodes is very large, especially if, as is the usual case, we deal with time dependency by constructing a time expanded graph. In practice, simply applying Transit to a time expanded graph does not scale. To deal with this problem, we will apply it to two-layers model of the public network. We start with a single objective problem and show how to extend it to multi-objective criteria, such as travel time, tickets cost and hassle of interchanges according to user preferences.

## 2 Related Work

In recent years several algorithms have been developed that use precomputed information to obtain a shortest path in a road network in a few microseconds, [10]. However there has been less progress on public transport networks. In [1] H. Bast discusses why finding shortest paths in public transport networks is not as straight forward as in road networks. There are several issues that arise in public networks, which are not encountered in road networks. For example, source and target of the query are geographical locations, and we need to walk first to some nearby station. A priori it is not so clear what station should we start our search from, so here we have set of source and target stations. Other issues we need to consider are *transfer time safety buffers*, *tickets cost*, *operating days*, etc. The recent and the most prominent result in this area is by H. Bast et al [2]. They report a time of 10ms for station-to-station query for a North America public transportation network consisting of 338K stations and more than 110M events. Besides being relatively complicated, the main drawback of their algorithm is the large computational resources required for precomputation. The authors report requirements of 20-40 (CPU core) hours per 1 million of nodes. For the Swiss transit network, which is comparable in size to the Sydney network, the reported precomputation time was between $560 - 635$ hours. Our approach is more intuitive, has much less hardware requirements and precomputation time. In addition we show how to provide multiple results incorporating user preferences.

## 3 Modeling the Network

Currently, there are two main approaches to model a public transport networks, known as *Time-Dependent* and *Time-Expanded* models. For an exhaustive description of the models and existing techniques we address the readers to [7, 9]. Typically the *Time-Expanded* model has three types of nodes: *arrival node*, *departure node* and *transfer node*. In our new model we eliminate *transfer nodes* and all the links from *transfer nodes* to *departure nodes*. Instead we connect *arrival* and *departure* nodes directly. For the Sydney public transport network containing $9.3K$ stations and $4M$ events, the space reduces from $600Mb$ to $480Mb$. In addition to the storage saving this modification also speeds up precomputation time. The model consist of two layers: *station graph* and *events graph*. The *events graph* nodes are arrival and departure events of a station and are interconnected by four types of links: *departure links*, *continue links*, *changing links*, *waiting links*. The *station graph* nodes are the stations and it has two types of links *station links* and *walking links*. In our experiments we assume that we can walk from every station to every other station within 10 minutes walking radius. The described graph is illustrated below.

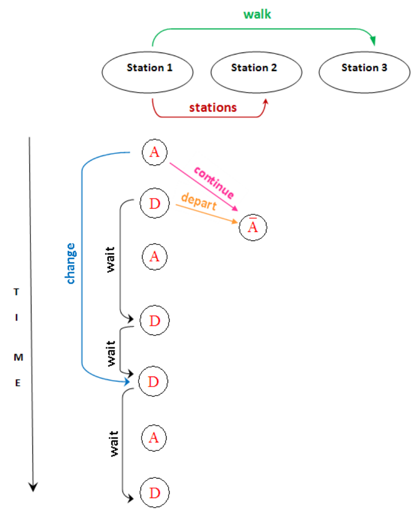

Figure 1: The two layered, time expanded graph with three stations.

## 4 The Modified Transit Algorithm

Assume for a moment that our world is a public transport network, i.e. we start and finish our journey at *event nodes*. We start with solving a single objective problem, e.g. we are interested to find the fastest way to get from station $A$ to station $B$ starting our journey at time $t$. Being consistent with [2] we denote this problem as $A@t \to B$. This problem is equivalent to finding an earliest arrival time, given the departure time. In reality the user may be interested in finding a route with the latest departure time, given an arrival time, which we will denote as $A \to B@t$. This query can be answered efficiently by simply applying exactly the same algorithm but using backward Dijkstra. Similarly to the original Transit algorithm [3] our algorithm also consists of two phases - precomputation and query.

**Precomputation**
Similarly to the original transit algorithm, in order to identify *transit nodes* we exploit the fact that the public transport stations have a geographical coordinates associated with them and use a rectangular grid.
At first stage of this phase, we identify a set of *transit nodes*. Intuitively a *transit node* is a node through which at least one long distance shortest path pass. Since the time expanded graph is large, both precomputation time and storage space will be prohibitive. In order to overcome this issue we define transit nodes to be stations rather than events and determine them in the *stations graph*, rather than in the *events graph*. This approach very successfully solves the scalability issue, but introduces several complications. Between two stops, one path can be a shortest path, say in the morning and another one in the evening. Therefore we need to make our transit nodes time dependent. We partition the day into segments that reflects public services daily patterns, say 6am-9am, 9am-12am, 12pm-4pm, 4pm-7pm, 7pm-10pm, 10pm-6am. Since the *station graph* is relatively small (10K-40K nodes), this is done relatively fast and can be executed in parallel.
The next stage is performed on the *events graph* layer of our network. We precompute the following shortest routes and store them in three tables: (i) *node-to-transit*: for every station $S$, from every departure event of $S$ to every transit station node of $S$, according to the time of the day, (ii) *transit-to-transit*: from every event of transit station node to every other transit station node and (iii) *transit-to-node*: for every node $S$, from every event of its associated transit station node to $S$.

**Query (time only)**
Given the query $A@t \to B$, if $A$ and $B$ are a "long distance" [1] away we find the fastest journey time as follows. We fetch transit station nodes of $A$ and $B$, $T_A$ and $T_B$ accordingly. Let $\tau_A \in T_A$ and $\tau_B \in T_B$ be transit nodes of $A$ and $B$. For every $\tau_A$ we fetch $c_1 = cost(A@t, \tau_A)$. Let's assume that this route arrived to $\tau_A$ at time $t_1$. Then we fetch $c_2 = cost(\tau_A@t_1, \tau_B)$. Finally, assuming that the fastest route from $\tau_A$ at time $t_1$ arrived to $\tau_B$ at time $t_2$, we fetch $c_3 = cost(\tau_B@t_2, B)$. Eventually the total travel time will be $c_1 + c_2 + c_3$. Iterating over all $\tau_A \in T_A$ the cost of the fastest route will be the one that yields minimal $c_1 + c_2 + c_3$. In case $A$ and $B$ are not "long distance" away, we just apply any efficient search algorithm, A* for example.

**Query (actual path)**
Unlike road networks, where for many applications returning the time (or distance) is good enough, in public transport networks users will be usually interested in concrete route directions. More precisely, besides knowing on which service to board at the start of their journey, we need to provide instructions where they should change services. Luckily, the vast majority of our journeys consist of very small number of transfers, 2-3, 4 and more on very rare cases. Observing this, during precomputation phase for every precomputed pair we can store the instructions where to change services. Alternatively we can store only the first transfer instruction and reconstruct the whole journey by iteratively applying the Query from the next transfer station. Again the number of iterations should be very small.

## 4.1 Dealing with Multiobjectiveness

In addition to find the fastest connection between two points, the user may consider also other criteria, such as the cost of tickets, hassle of interchanging between services, etc. Moreover, different users can have different preferences over these criteria. For example a businessmen may try to optimize his travel time, a student may try to minimize his costs and a visitor may wish to avoid changing services in order not to get lost. There are several ways to deal with multiobjective optimization [7], in this work we choose the normalization approach, by introducing a linear utility function. This approach reduces the multi-criteria problem to a single-criterion optimization, which we can solve as described.

[1]"long distance" will be precisely defined in the full paper

## 4.2 Providing multiple results in the real world

Until now we have assumed that our world is a public transport network, i.e. the start and end point of the journey is always a station node and we are departing exactly at time $t$. The result we obtain is theoretically optimal, but of course in the real world most of the time this is not the case. We may need to walk from our home to our first station and from the last station to our final destination. Moreover, in real life we would prefer to wait a little bit now if we know that eventually we may arrive earlier, for example to wait for an express bus. In addition we all like choices, therefore the system will be more user friendly if it could provide multiple attractive alternatives to the user. In order to cover those real life scenarios and provide the user several attractive suggestions we will run our Query starting from different stations around the user's starting location and different times around his specified departure/arrival time $t$. From those we will choose, say the best five.
Another important aspect we consider is robustness of the provided solutions. In practice, public transport often runs late due to traffic congestion or accidents or other unpredicted events. Missing a connection by one minute may cost us an hour in our total journey time, if the next connection is infrequent and departs say only once an hour. In order to minimize such occurrences and to make the system more reliable and user friendly we identify those trips and warn the user about risky connections. Then the user will choose his preferred trip according to his risk adverseness.

## 4.3 Complexity Analysis

We will start with complexity analysis of the original Transit algorithm for road networks. The complexity of the algorithm depends on many factors, such as graph nodes distribution, node connectivity, etc. Intuitively we can see that if we choose $S_{inner}$ to be very small (say containing only one node), then every node will be a transit node and the precomputation will compute all the shortest paths. In this case a query will be a simple lookup in a large precomputed table. On the other extreme, suppose we choose $S_{inner}$ to contain all the nodes. In this case, every query is local and we do no precomputation. So we can observe that there is a clear tradeoff between size of the squares, precomputation time, storage and query time. In what follows, we will start assuming a simplified "grid world" graph layout, where nodes are equally distributed and every node is connected to its four neighbors by a link of unit cost. Let $k$ denote the number of cells and $n = |V|$ denote the

number of nodes. Consider cell $C$. Then $|V_C| = \frac{n}{k}$. Let $S_{inner}$ be a square centred on $C$ consisting of some constant number of cells. In the worst case, the number of transit nodes for cell $C$ equals the number of border nodes of $S_{inner}$, which is $O(\sqrt{\frac{n}{k}})$. Since we have $n$ nodes, the storage space for the *node-to-transit* table will be $O(n\sqrt{n})$ for any choice of $k$, which may be prohibitive.
In real life networks, not every road has the same travel time. There are highways, major roads, minor roads, etc. In order to make our simplified "grid-world" graph resemble a real life road network we assume that every, say, 10th vertical and horizontal road is a highway. We will model this by assigning zero cost to such highways. Since every cell $C$ is of bounded size, it follows that only a constant number of highways cross every cell. Consequently the number of transit nodes for every cell is $O(1)$. This gives $O(n)$ storage space for the *node-to-transit* table. Now, the total number of transit nodes is $O(k)$. Therefore, if we choose $k$ to be $O(\sqrt{n})$ we need $O(k^2) = O(n)$ storage space for the *transit-to-transit* table.
In a public transport network, there is an additional factor - events. Let $e$ be an average number of events per station. Then, the total number of nodes is $ne$. When applying the proposed algorithm on public transportation network, eventually we need to run a Dijkstra from every event. Since there are totally $ne$ nodes and the running time of one Dijkstra is $O(ne \log(ne))$, the total precomputation time is $O(n^2e^2 \log(ne))$. Similarly to the original Transit algorithm, the storage space is $O(ne)$. While asymptotical precomputation time may look somewhat discouraging, by using the speed up techniques described in 4.4, in practise the precomputation time is much faster.

### 4.4 Speed up techniques

One significant improvement was achieved by noticing that there is no need to precompute and store distances from/to every node in the network. Without loss of accuracy we can ignore stations that are visited only by one service. Given a query where source or destination one of this nodes we can just simply follow the only path from this node until we encounter first station node that is visited by more than one service and was precomputed. We observed about 40% reduction in the number of nodes for which we should perform a precomputation stage 4.
Another significant speed up was achieved by noticing that same service running, say 10 minutes later, will eventually take me to the same stations just 10 minutes later. Since there is no point for the user to wait on the sta-

tion for exactly the same service which will arrive 10 minutes later, during the precomputation stage, when running Dijkstra this whole branch can be pruned. This gave us about 20% speed up in time performance.
Similarly, we can notice that a journey from station $A$ to a *nearby* station $B$, say at 9am and 9:30am has actually the same pattern. On the other hand traveling from $A$ to $B$, say at 6pm may have a different pattern. In order to capture this, we divide the day into segments (e.g. 6am-9am, 9am-12pm, 12pm-4pm, 4pm-7pm, 7pm-10pm, 10pm-6am) and precompute the *node-to-transit* table only once for every time range, for a single event of every service. This short-cut may sacrifice a small amount of optimality, but makes a significant improvement in precomputation time.
Another 'trick' to make precomputation twice as fast is to precompute the tables only for departure events. During the query time, if we need to continue on the same service through one of the transit stations, the departure event associated with the arrival event of these service can be momentarily accessed by an auxiliary link connecting between those two events.
Finally, an additional significant speed up is achieved by noticing that the process of determining transit nodes for a cell is completely independent of other cells and therefore can be parallelized. Similarly we can observe that during Stage 2 the precomputation and storage of the three database (*node-to-transit*, *transit-to-transit*, *transit-to-node*) tables is also independent and can be performed in parallel. The query between different source and destination pairs, as described in section 4.2 can be also easily parallelized.

## 5 Implementation and Experiments

The proposed algorithm was implemented using the Java programming language and the experiments were performed on PowerEdge 1950 server, with those specifications: CPU: 2 x 2.00GHz Intel Quad Core Xeon E5405, Memory: 16 GB. We present some preliminary tests results obtained using the Sydney public transport multimodal network consisting of buses, trains, ferries, lightrail and monorail modes. There are $9.3K$ station nodes, $2.1M$ event nodes and $8.1M$ links in the underlying graph of that network.

| Inner/Outer squares size | Grid size | $\lvert T\rvert$ | Storage Mb. |
|---|---|---|---|
| 3/5 | 32 x 32 | 51 | 685 |
| 5/9 | 32 x 32 | 9 | 36 |
| 3/5 | 64 x 64 | 234 | 1510 |
| 5/9 | 64 x 64 | 104 | 747 |
| 3/5 | 128 x 128 | 1065 | 9914 |
| 5/9 | 128 x 128 | 496 | 3028 |

**Table 1:** Experimental results using different grid sizes, and different sizes of $S_{inner}$ and $S_{outer}$ squares comparing between $|T|$ number of transit nodes, and estimated storage space.

In what follows we present results for grid size of 100x100 cells and using $S_{inner}$ consisting of $5 \times 5$ cells and $S_{outter}$ of $9 \times 9$ cells. In this setup, we have 319 transit nodes, the offline, precomputation stage, including presented speed-up techniques took us ~20 hours and ~1.2Gb of storage space including full transit instructions. Without any speed-ups the estimated time of this stage is more than 100 hours. On average, running 1000 random queries a single 'station-to-station' query time is ~0.4ms and 'location-to-location' query time is ~20ms.

## 6 Conclusions and Future work

In this work we presented a novel approach for finding optimal connections in public transit network. We presented an experimental results using the Sydney public transportation network. Our future work includes improving the preprocessing time and reducing the database tables storage space, if possible whilst preserving optimality. There are several promising directions we are interested to investigate in order to achieve this, such as using an adaptive grid rather than simple, fixed grid or partitioning the underlying graph into clusters in completely different matter. In addition we would like to extend this idea to fully realistic inter modal journey planer which includes combination of private transport (e.g. car, motorbike, bicycle) and public transport and incorporates real time updates for both traffic conditions and public transport actual location and estimated time of arrival.

**Acknowledgments.**

This work was partially supported by Buzzhives. The first author would like to thank Dr. Phil Kilby, Dr. Tim Cooper, Adrian Schoenig Ph.D. student under Toby Walsh's supervision for helpful discussions.

## References

[1] Hannah Bast. *Car or Public Transport–Two Worlds*, pages 355–367. Springer-Verlag, Berlin, Heidelberg, 2009.

[2] Hannah Bast, Erik Carlsson, Arno Eigenwillig, Robert Geisberger, Chris Harrelson, Veselin Raychev, and Fabien Viger. Fast routing in very large public transportation networks using transfer patterns. In *Proceedings of the 18th annual European conference on Algorithms: Part I*, ESA'10, pages 290–301, Berlin, Heidelberg, 2010. Springer-Verlag.

[3] Holger Bast, Stefan Funke, and Domagoj Matijevic. Transit ultrafast shortest-path queries with linear-time preprocessing. In *9th DIMACS Implementation Challenge*, 2006.

[4] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In transit to constant time shortest-path queries in road networks. *Proc. 9th Workshop on Algorithm Engineering and Experimentation (ALENEX)*, 2007.

[5] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Workshop on Experimental and Efficient Algorithms*, pages 319–333.

[6] Transit Google. *http://www.google.com/intl/en/landing/transit.*

[7] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Timetable information: models and algorithms. In *Proceedings of the 4th international Dagstuhl, ATMOS conference on Algorithmic approaches for transportation modeling, optimization, and systems*, ATMOS'04, pages 67–90, Berlin, Heidelberg, 2007. Springer-Verlag.

[8] TransportInfo NSW. *http://www.131500.com.au/.*

[9] Evangelia Pyrga, Frank Schulz, Dorothea Wagner, and Christos D. Zaroliagis. Experimental comparison of shortest path approaches for timetable information. In *Algorithm Engineering and Experimentation*, pages 88–99, 2004.

[10] Peter Sanders and Dominik Schultes. Robust, almost constant time shortest-path queries on road networks. In *IN: 9TH DIMACS IMPLEMENTATION CHALLENGE*, 2006.