TDplanner: Public Transport Planning System with Real-time Route Updates Based on Service Delays and Location Tracking

D. Nguyen, T. MacDonald, Z. Xu
Department of Mathematics and Computer Science
Suffolk University
Boston, USA
{dnguyen, tmacdona, zxu}@mcs.suffolk.edu

***Abstract*—This paper proposes an alternative to the problem of finding optimal paths in public transportation networks. The authors present a system architecture that has been implemented on the bus system in Boston using their new solution and completed for mobile devices. On user's request the system proposes an optimal path to a destination and provides with alerts during the journey. To the best of their knowledge, it is the first planning system with the ability to respond to route updates in real-time. If there are changes due to user's errors (user misses a bus, etc.) or service delays, an updated or a new route is offered.**

I. Introduction

The key issue in any public transport planning system is the shortest path problem and many algorithms have been developed. In its most trivial form, we can use the Dijkstra's algorithm [1] that is simple, well-known and optimal, however only for static graphs. In a time-dependent model of public transport, the schedule makes a graph dynamic. The time of the next departure is statically determined but the wait time depends on the time of arrival which itself depends on the path traversed.

Dreyfus [2] developed many methods for graphs with time-dependent links, although, the algorithms do not allow users' preferences such as walk-links or the number of resulting routes.

In the work by *Koszelew* [3], the author proposes a different shortest path algorithm that considers two most important users' preferences: minimal travel time and minimal number of bus transfers. The algorithm minimizes hops to a destination at each step but we found cases where it failed to find the shortest path for certain graphs with waiting time during transfers. Also, the approach is only an approximation of the shortest path. From the computation point of view, it requires 1 $N*N$ distance matrix, k of $N*N$ transfer matrices and $N*N*k$ path templates which present high pre-computation and storage costs. The runtime for graphs with more than 10,000,000 edges is relatively high.

In this paper, we present our own solution to this shortest path problem. We propose an improved model of graph representation, so called *'Exploded graph'* in which we *'explode'* each bus station which has multiple routes into one node per route and we connect nodes in the same station to each other with dynamically weighted *'transfer'* edges. This enhancement allows us to use the standard Dijkstra's algorithm that naturally guarantees a path with the lowest cost and with the exploded graph it also minimizes number of transfers.

Outline: The remainder of this article is organized as follows. Section II gives account of previous work and its drawbacks are shown in Section II-A. Our new solution to the problem is described in Section III. We present the use of our approach on a proposed system architecture in Section III-A. Next, we give detailed implementation in Section IV. In Section V, we compare our approach with other work from the past. Section VI gives the conclusions and outlines prospective work on this research project.

II. Method Based on Transfer Matrices

First, we review a previous work by *Koszelew* [3], the method based on transfer matrices. The network model is represented as a weighted graph *G=(V, E, t)* where *V* is a set of nodes, in our case bus stops, *E* is a set of edges and *t* is a function of weights. Next, we assume that bus stops are represented by numbers from *1* to *n* and a directed edge *(i,j)* $\in E$ exists if there is at least one bus direct connection from station *i* to station *j*. Each edge has a weight of t_{ij} that is determined by travel time from node *i* to *j* from the schedule. The algorithm consists of several steps. In the pre-computation, we determine transfer matrices $Q_0, Q_1, ..., Q_{maxt}$ and the minimal distance matrix *D* for a given graph *G*. Next, we use transfer matrices to generate templates of optimal paths. Finally, at runtime, we find all templates between source *s* and destination *t*, walk along each template and minimize the number of hops at each step.

A transfer matrix Q_k ($k = 0, 1, ..., maxt$) is a two-dimensional matrix where each its element is equal to 1 if there exists at least one route from *s* to *t* with exactly *k* transfers, otherwise it equals to 0. Transfer matrices can be determined using the standard breadth first search algorithm.

The minimal distance matrix *D* saves the minimal distance between two nodes where d_{ij} = 0 if $i = j$, $d_{ij} = \infty$ if there is no path from *i* to *j*, otherwise d_{ij} equals to the length of a

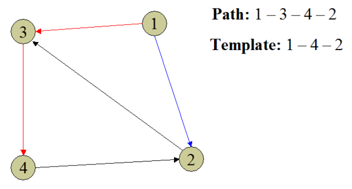


Fig. 1. Path Templates

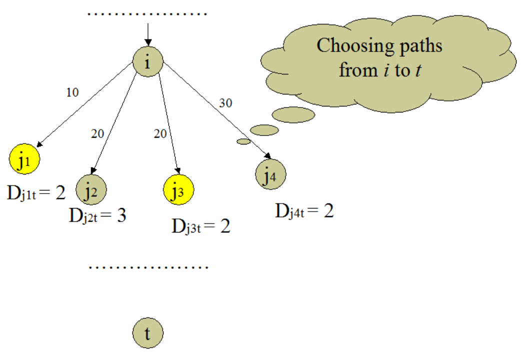


Fig. 2. Exploration

shortest path from *i* to *j*. The matrix can be determined again using the breadth first search algorithm.

A path template (Figure 1) is a sequence of nodes from *s* to *t* and includes only those nodes where we have to change the bus. We can determine all templates using transfer matrices.

After having generated transfer matrices, the minimal distance matrix and all path templates for the graph, only the last step is left and that is determining details of paths. We generate direct paths with no transfer first, next, paths with one transfer, paths with 2 transfers, etc. Since there might be many of such paths, the authors use an approximation based on the *D* minimal distance matrix. Let us assume that we are in node *i* and we need to determine all possible steps forward to the next bus stop *j* where we will change the line, that is to the next node of the path template. If there are two or more ways to move from node *i* to node *j* with the same or similar travel time that differs for each such way by a constant ϵ, the algorithm chooses the nodes *j* that have minimal value of d_{jt} (number of bus stops to the destination). This is called *Exploration*. See Figure 2.

Finally, we sort the resulting paths according to the travel time primarily and further according to the number of transfers.

A. Drawbacks of Previous Work

The algorithm presented in the previous work has serious drawbacks. First of all, it claims the ability to deal with a time-dependent model of transportation network and can satisfy users' preferences of minimal travel time and minimal number of transfers. But it fails to do so and in certain cases the path with the shortest travel time is not even included among its resulting set of quasi-optimal paths. For instance, let us assume

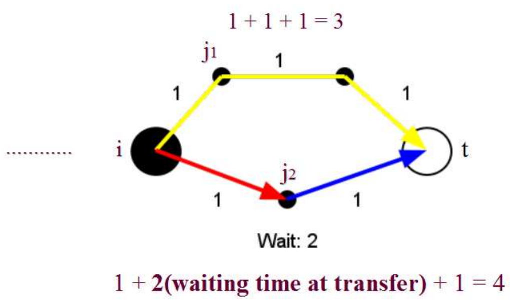


Fig. 3. Previous Work Fails to Find the Shortest Path

a simple situation as in Figure 3.

In node *i*, no matter how small the ϵ value is, both node j_1 and node j_2 are taken into account by the approximation algorithm. And since the distance d_{j_2t} from node j_2 to the destination node t is smaller than the distance d_{j_1t} from node j_1 to the destination node t, the next chosen node is j_2. And the next node is the final node t. Therefore, the chosen route by the algorithm is taking the red bus line and then transfer to the blue one. We can see that the algorithm fails to find a path with the shortest travel time, in fact, it is not even considered as one of the quasi-optimal one. Instead, it chooses the route with travel cost of 4, including the waiting time at transfer.

The pre-computation and storage requirements are also relatively high. Altogether the algorithm needs 1 $N * N$ distance matrix, k of $N * N$ transfer matrices and $N * N * k$ path templates that certainly is not a negligible cost, taking into account the fact that nowadays, world cities' public transport systems include many more nodes than 1500, as the authors considered for a city of 250,000 population. Not to mention that only bus transport is considered.

The idea of getting quasi-optimal routes is interesting, although, we think that users would appreciate more if they are offered with exact shortest paths in time for certain preferences.

III. Our Solution

We propose our own solution to this shortest path problem. We introduce a so called 'Exploded graph' in which each station having multiple routes is 'exploded' into one node per route. Next, the nodes in the same station are connected to each other with dynamically weighted 'transfer' edges and they represent the waiting time for the next closest departure. See Figure 4.

Let us assume that node *i* is an intermediate and we choose a route from node *i* to destination node *t*. Since there are two routes passing through node *i*, we 'explode' node *i* into two nodes, i_1 and i_2. There is a blue bus line going to node *k* and it takes 40 minutes. At node *k* we have to transfer to the yellow bus line and the waiting time for the departure is 20 minutes. Therefore we create a transfer edge with the weight of 20. The second route starts from the newly exploded node i_2. First, we take the red bus line to station *j* and there we have to change to the orange line, thus, we create another transfer

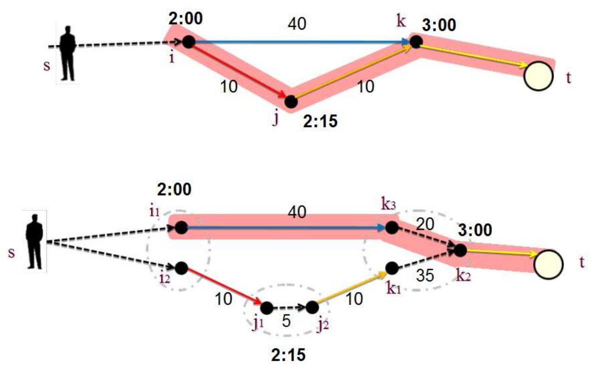


Fig. 4. Exploded Graph

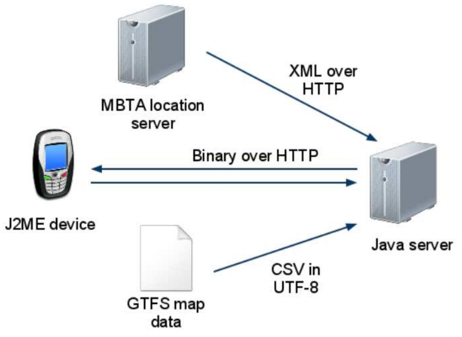


Fig. 5. System Architecture

edge for node *j* that is exploded to two new nodes, j_1 and j_2. At station *k* we wait for 35 minutes, therefore, we add a transfer edge with a value of 35 between two exploded nodes k_1 and k_2. The yellow bus line from station *k* to destination *t* is shared by both routes.

Such exploded graph allows us to use the standard Dijkstra's algorithm that naturally guarantees a path with the lowest cost. Furthermore, it also minimizes the number of transfers.

A. System Architecture

Our system architecture is divided into four key parts, the Java Server, the Boston's MBTA Public Transport Location Server, the GTFS Map Data and the J2ME Device. See Figure 5 for details.

The Java Server receives the source coordinates and the destination address. Next, it geocodes the destination address in order to get its latitude and longitude. It locates the stations nearby the source and the destination, computes the shortest path in time and finally, returns an optimal path and a session ID to a mobile device.

During travel, the server receives periodic location updates from the mobile and does the checking process. First, it checks if the user is far off the chosen shortest path, if so, it computes a new route and alerts the user. Next, it checks the user's movement against the bus movement and if the user is aboard a bus and nearing the destination, it sends an alert for user to disembark. Last, the server checks if the user is behind the schedule, if so, it computes an updated schedule or a new route and alerts the user.

The MBTA Location Server during our development could be queried for the locations of all buses in the Boston Area, recently it has been updated also for the subway. That can be easily updated in our system. A query is in a form of an HTTP get request and returns XML data consisting of a location of each bus, a route number and a direction. The server can also be queried for arrival time at a specific stop. A query is again an HTTP get request with parameters of a stop, a route and a direction. For such query, we get XML data that consists of an arrival time of the next few buses.

The GTFS Map Data (General Transit Format Specification) uses the common interchange format for public transit data. It was originally developed for Google Maps and it stores stations, routes and schedules stored as Comma Separated Values (CVS) in UTF-8. The relevant files that have been used are following:

- $stops.txt$ contains stop IDs, names, coordinates
- $routes.txt$ contains route IDs, names, types
- $calendar.txt$ contains calendar IDs, days of service
- $trips.txt$ associates routes to calendars
- $stop-times.txt$ associates trips to stations and times

The J2ME Device is implemented as a MIDlet. It retrieves a destination address from the user, retrieves its own GPS location and transmits both to the server. Next, it changes to the poling stage and receives a route back from the server, if one exists. Afterwards, it enters a GPS tracking loop. In the loop it sends its actual location to the server periodically and retrieves any alerts at the same time. The alerts are displayed to the user when nearing a station where the user must disembark or it could be information on a change of route due to user error or delay. Or the alert could let the user know that the bus is approaching to the destination.

IV. Implementation

Implementation details of each algorithm are given in this section.

A. Shortest Path Algorithm

First, we review the necessities of the shortest path algorithm. We need to consider paths between all stations S near the source, all stations D near the destination. S and D may be fairly large and particularly near transit centers. The Dijkstra's algorithm has a complexity of $O(E + VlogV)$ where V is the number of nodes and E is the number of edges. Since we have to run Dijkstra once for each pair in S, D, that gives us a complexity of $O(SD[E + VlogV])$. Instead, we temporarily modify the graph by inserting nodes to represent source src and destination $dest$, inserting edges between src and the elements of S, and between the elements of D and dst. This allows us to run Dijkstra exacly once, albeit on a graph of a slightly increased size. Our complexity is then $E+S+D+(V+2)log(V+2)$ which is $O(E+S+D+VlogV)$ and that is still dominated by $O(E + VlogV)$.

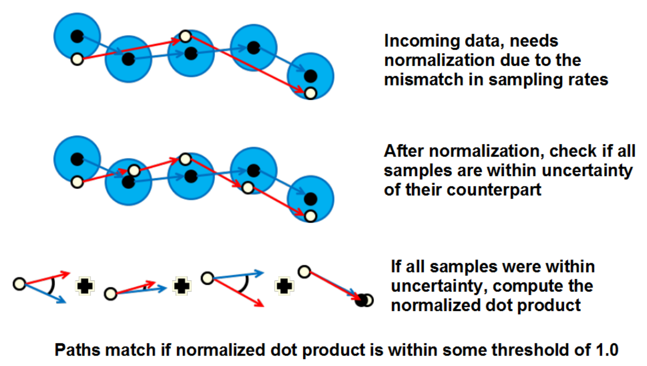

Incoming data, needs normalization due to the mismatch in sampling rates
After normalization, check if all samples are within uncertainty of their counterpart
If all samples were within uncertainty, compute the normalized dot product
Paths match if normalized dot product is within some threshold of 1.0

Fig. 6. Vector Normalization

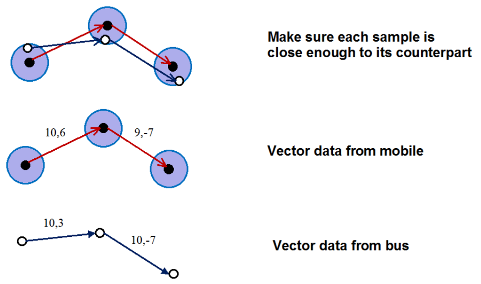


Fig. 7. Vector Matching

B. Determining Bus Boarding

When we try to get to know when the user boards a bus, we watch the location data from the mobile device and the bus, and match the two streams using vector matching. If we know the user is aboard, we can provide with an alert before the user must alight and we can update the route if the user's bus is delayed. However, not always it is possible to determine whether the user is aboard or not, for instance, the GPS signals may be blocked by buildings. In such situations we ask the users to confirm that they are aboard.

C. Vector Matching

We have two streams of location data, one from the mobile device and the other from the bus. These streams may arrive at different intervals, mobile GPS may be blocked or the connection may be lost. Therefore, the sample rate of incoming data needs to be normalized, see Figure 6.

If the location distances are not within an uncertainty, the user is too far away from the bus to be aboard. We treat each stream of data as an N-dimensional vector and the scalar product gives the similarity, if it is close to 1, streams match, see Figures 7 and 8.

D. Walking Distance Estimation

We need to estimate the walking distance from the current position to the starting station, from the end station to the destination and the distance between nearby stations when transferring. We assume that we do not know anything about the topology and do not have street maps or turn-by-turn directions. Thus, approximations are needed. We can use, for

$$\frac{a \cdot b}{|a||b|}$$

Formula for dot product

$$\frac{10\cdot10+6\cdot3+9\cdot10+-7\cdot-7}{\sqrt{10\cdot10+3\cdot3+10\cdot10+-7\cdot-7}\sqrt{10\cdot10+6\cdot6+9\cdot9+-7\cdot-7}}$$

$$\frac{257}{\sqrt{258}\sqrt{266}} = 0.981$$

Result is close to one, therefore it is a match

Fig. 8. Vector Matching

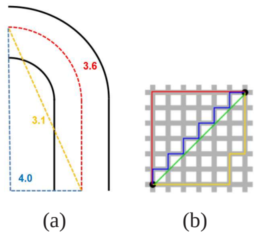


Fig. 9. (a) Walking Distance Estimation (b) Manhattan Distance(the red, blue and yellow paths share the same length)

instance, the Euclidean distance or the Manhattan distance. See Subfigures 9(a) and 9(b).

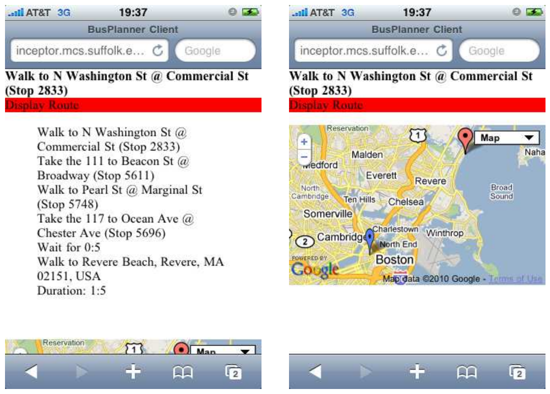
The actual distance (red) is an irregular path and is impossible to derive without a detailed map. Thus, the Euclidean distance (orange) can be used but it severely underestimates the distance. The Manhattan distance (blue) overestimates the distance by a little but is still more convenient. The Manhattan distance has good approximation for grid-layout cities and is less beneficial for irregular ones. However, it is generally better, since the Euclidean severely underestimates curves and corners, and underestimation is bad for causing missed buses. The Manhattan distance can approximate even complex paths.

The running application is demonstrated on the iPhone in Subfigures 10(a) and 10(b). It can run on all personal device assistants and smartphones. For older mobile devices with poor graphical support we developed a light-weight version of our application which is available on the research project's website [4], along with the shown version.

V. Related Work

In recent years, several algorithms using precomputation have been developed and they can find shortest paths on the road network of a whole continent in a few microseconds, which is a million times faster than Dijkstra's algorithm. However, none of them yields similar speed-ups for public transportation networks of comparable sizes. Without assuming anything about the nature of the network and without any precomputation, we would have to do a Dijkstra-like search and explore nodes to compute an optimal path as in our work.

Berger et al. [5] assumes the model of railways and forbids dynamic updates which insert new arcs into the model. The work argues that it is no strong restriction since new rail road tracks will not be provided all of sudden and will usually be known in advance to be included into the preprocessing. However, for the bus transport, new or emergency routes are



Walk to N Washington St @ Commercial St (Stop 2833)
Display Route

Walk to N Washington St @ Commercial St (Stop 2833)
Take the 111 to Beacon St @ Broadway (Stop 5611)
Walk to Pearl St @ Marginal St (Stop 5748)
Take the 117 to Ocean Ave @ Chester Ave (Stop 5696)
Wait for 0:5
Walk to Revere Beach, Revere, MA 02151, USA
Duration: 1:5

(a) (b)

Fig. 10. TDplanner on iPhone: (a) Route Proposal to User (b) Displaying Chosen Route, Actual Position and Alerting Route Updates

added and deleted extremely often, especially in big cities such as New York or Tokyo. This problem is taken into account in our work.

In general, public transport cannot be solved similarly as long distance transport such as cars or railways. On large municipal areas, query processing times can even be worse than a well-tuned implementation of Dijkstra's algorithm.

Hierarchy speed-up methods use a simple routing heuristic: Roads are divided into levels of importance and when searching, it takes into account all the roads in close proximity to the source and target, but once a certain distance from the source or target is reached, it leaves less important roads etc. In the public transportation network you will hardly find any hierarchy.

Methods using shortcuts/contraction replace a sequence of short straight-line road segments by a single arc, thus significantly reducing the number of nodes and arcs in the graph. But contraction and/or shortcuts are only effective for nodes of low degree which is not the case of big cities again.

Goal direction approach augments Dijkstra and partitions nodes into regions with similar distance to destination and the heuristic considers only nodes on the boundary of each region. Although, in public transportation there is very often bad connectivity to villages surrounding a big city and we might be late for the last bus and have to wait overnight for the next one, while simple Dijkstra takes a bus directly to the destination. Distance table methods share similar problems, we do not have efficient algorithms for local searches and it can take hours to a nearby village.

There have been some implementations in mobile devices that unfortunately cover only road networks. Goldberg at al. [6] implemented the *ALT* algorithm on a Pocket PC. The *RE* algorithm [7] [8] has been implemented with query times of 'a few seconds including path computation and search animation' [9] for '2-3GB of USA/Europe'. Contraction hierarchies in [10] require 69ms query time for the European network.

In one of the most recent implementations, *Martinek et al.* [11] considered both path walks and public transport. Although, the work does not consider real-time updates in traffic where just a small change (service delay, user misses bus) can invalidate most of precomputed information.

VI. Conclusion and Future Work

In this paper, we proposed a new approach to the shortest-path problem in public transportation networks based on a graph representation called 'exploded graph' that allows us to use a well-tuned Dijkstra's algorithm to search paths with the lowest cost and least transfers. Next, we introduced a system architecture and its implementation on the bus system for the Boston Area with an application developed for users of mobile devices. Main computational responsibilities lie on the Java Server that communicates with a mobile client and the Location Server. When the server receives the user's initial location and a destination address, it returns an optimal path. During travel, the client sends its actual location to the server periodically and retrieves any alerts. To the best of our knowledge, it is the first system to be able of providing the users with real-time route updates based on service delays and actual location of the user. If the user misses a bus or disembarks at a wrong station etc., an updated or a new route is computed. Similarly, the system responds if a bus gets behind or ahead of the schedule.

In the nearest future we plan to undertake more experiments in order to evaluate the system performance thoroughly. We also want to accommodate newly available data for other means of public transport from the Boston's Transport Authority. Afterwards, we want to fine-tune our system for its expansion to other cities and make the application widely available for users.

References

[1] E. Dijkstra, "A note on two problems in connection with graphs", Numerische Mathematik, 1, 1959, pp. 269-271.
[2] S. E. Dreyfus, "An Appraisal of Some Shortest-path Algorithms", Operations Research, 17, 1969, pp. 395-412.
[3] J. Koszelew, "Two Methods of Quasi-optimal Routes Generation in Public Transportation Network," in CISIM '08: Proceedings of the 2008 7th Computer Information Systems and Industrial Management Applications. Washington, DC, USA: IEEE Computer Society, 2008, pp. 231236.
[4] D. Nguyen, T. MacDonald, Z. Xu, "TDplanner Research Project Homepages", http://inceptor.mcs.suffolk.edu/∼TDplanner, 2010.
[5] A. Berger, M. Grimmer, M. Muller-Hannemann, "Fully Dynamic Speed-Up Techniques for Multi-criteria Shortest Path Searches in Time-Dependent Networks", P. Festa (Ed.): SEA 2010, LNCS 6049, pp. 35-46, 2010.
[6] A. V. Goldberg, R. F. Werneck, "Computing Point-to-Point Shortest Paths from External Memory," in Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX 2005), pp. 26-40. SIAM, Philadelphia (2005).
[7] R. J. Gutman, "Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks," in Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX 2004), pp. 100-111. SIAM, Philadelphia (2004).
[8] A. V. Goldberg, H. Kaplan, R. F. Werneck, "Better Landmarks Within Reach," in: Demetrescu, C. (ed.) WEA 2007. LNCS, vol. 4525, pp. 38-51. Springer, Heidelberg (2007).
[9] A. V. Goldberg, "Personal communication", (2008).
[10] P. Sanders, D. Schultes, C. Vetter, "Mobile Route Planning," in: Halperin, D., Mehlhorn, K. (eds.) ESA 2008. LNCS, vol. 5193, pp. 732-743. Springer, Heidelberg (2008).
[11] V. Martinek, M. Zemlicka, "Some Issues and Solutions for Complex Navigation Systems: Experience from the JRGPS Project," icons, pp.92-98, 2010 Fifth International Conference on Systems, 2010