Trip Planning using Public and Private Transport

A Seminar Report

Submitted in partial fulfillment of requirements for the degree of

Master of Technology

By

Tasneem Lightwala
Roll No.: 183050004

under the guidance of

Prof. Abhiram Ranade

Department of Computer Science and Engineering
INDIAN INSTITUTE OF TECHNOLOGY, BOMBAY

MAY, 2019

Acknowledgment

I am extremely thankful to my seminar guide **Prof. Abhiram Ranade** for helping me throughout the research seminar. The seminar meetings have been very crucial in resolving doubts related to the Trip Planning in various scenarios. I can't thank enough my parents for their constant encouragement and support. I am also very grateful to my peers for all the discussions held through out the semester.

Abstract

Trip planning means finding paths between two locations such that certain criterion like travel time, travel cost and transfers are optimized. There are various techniques that can efficiently find optimal paths between two locations. Some techniques may require minimal preprocessing effort, some may result in suboptimal paths, while others may deal efficiently with real time data. Route planning in road networks is similar to finding the shortest path in the graph. The trip planning problem in public transportation network although conceptually similar, is a more complex problem than the route planning problem in road networks because of the time-dependent and multicriteria nature of the public transportation network. While some techniques rely on static schedules of public transport to generate optimal travel plans, some use real time data of buses to accurately generate routes that are adapted to traffic situations. Need for incorporating real time data in our trip planner arises due to inaccuracies in prediction of travel time due to unreliable bus schedules.

Contents

List of Figures

List of Algorithms

Chapter 1

Introduction

In this report, we consider the problem of trip planning for road network and public transportation network.

Road network graph consists of junctions as the vertices of the graph and there is an edge between two vertices if a road connects them. Now that we have a graph, applying shortest path algorithms on it will give the optimal route between origin and destination for road networks.

Public transportation network consists of bus stops and bus lines where buses follow specific routes as per schedule. Trip planning means finding an optimal travel plan or finding multiple non dominating travel plans between origin and destination. Travel plans are usually of the form

$$\{\{(v1,v2),b1\},\{(v2,v3),b2\},...\{(v_{n-1},v_n),b_n\}\}$$

where each element of the set specifies intermediate stops to be travelled along with the bus that needs to be taken. Thus, travel plans may require changing of buses at intermediate stops. These plans are considered optimal if they optimize certain criteria like minimizing travel time, minimizing number of transfers or minimizing fare cost.

The key difference between the road networks and public transportation networks is the time-dependent and multicriteria nature of public transports.

Time dependence: Time dependence means that the best travel plans for same origin-destination may vary depending on the departure time of the user from the origin, see figure 1.1. This happens due to traffic congestion and the fact that buses can travel only at specific points of time.

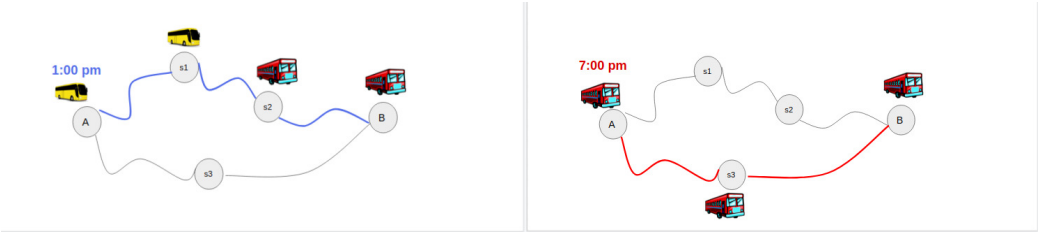

Figure 1.1: Time-dependent nature of public transports

MultiCriteria: Unlike road networks where travel time is the only criteria to be optimized, one may want to minimize travel time, number of transfers and monetary travel cost while travelling by public transports. Also, the optimal travel plans may change with changing of optimization criteria. See figure 1.2.

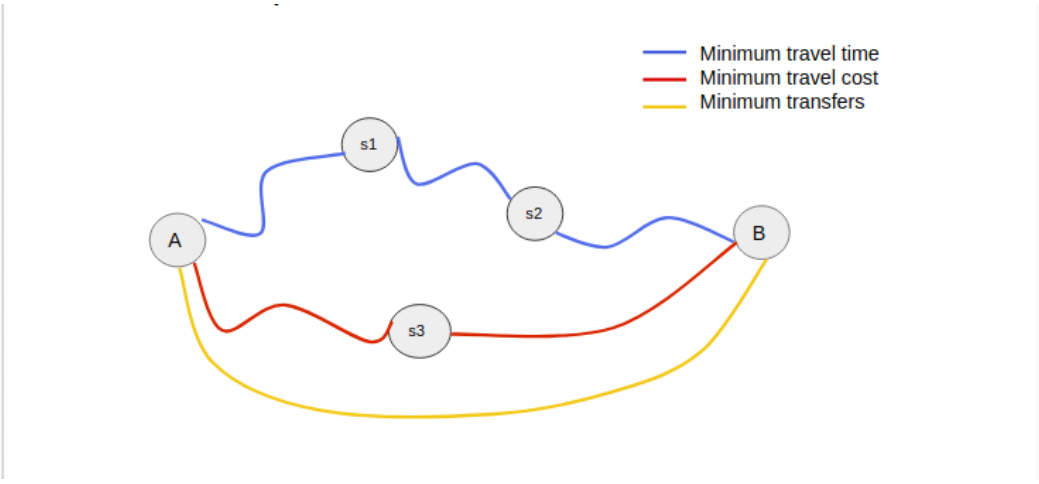

Figure 1.2: Multicriteria nature of public transports

There are different problem variants of Trip Planning:

Earliest Arrival Problem: Given origin, destination and departure time, the problem asks for travel plan that leaves origin at time no earlier than the given departure time and reaches destination as early as possible.

Range Problem: Given origin, destination and range of departure time, the problem asks for travel plan that leaves origin at any time in the given time range and minimizes the travel time.

Multicriteria Problem: Given origin, destination, departure time and criterion to be optimized, the problem asks for all non dominating travel plans optimizing given criterion. Plans T1 is said to dominate plan T2 i.e. $T1 \leq T2$, if plan T1 is no worse in any criteria than plan T2.

This report goes through different techniques of finding optimal route in road networks. Our focus in this report will be on Multicriteria problem, optimizing travel time and bus transfers in Public Transportation Network.

We discuss Route Planning in Road Networks in Chapter 2. Chapter 3 discusses two algorithms for Trip Planning using Public Transport. In Chapter 4, the report illustrates the need for real time data in trip planning and suggests two algorithms using real time data for trip planning. Conclusion at the end gives brief summary of the report.

Chapter 2

Route Planning in Road Networks

Road maps can be represented as graphs where junctions are the nodes, roads between junctions are the edges and time taken to travel each road is the cost of the edge. Therefore, finding the fastest route between two locations is same as finding shortest path between two points in a graph.

In this chapter, we discuss various techniques of finding optimal route in road networks [4]. This chapter firstly discusses basic techniques like Dijkstra's and Bellman-Ford and then moves on to advanced techniques. Goal-directed techniques such as A* search, Geometric Containers and Arc Flags are discussed. Then we move on to more optimal techniques like Separator based techniques, Bounded-Hop techniques and Hierarchical techniques. These techniques are more commonly used today by route planners such as Google Maps. Lastly we compare all the algorithms on basis of precomputation time and query time.

2.1 Basic Techniques

Dijkstra's algorithm is the most common solution for the one to all shortest path problem. It has a label-setting property i.e. once a vertex is scanned, its shortest path distance value from origin is correct. Therefore, algorithm may stop once it scans the destination node. Simultaneous forward and backward searches can reduce the search space for the algorithm.

A label-correcting algorithm scans each vertex multiple times. Bellman-Ford

is one such algorithm. Unlike Dijkstra's algorithm, Bellman-Ford works on graphs with negative weights. Floyd Warshall algorithm calculates distances between all pairs of vertices.

Goal directed techniques guide the search towards the destination node by avoiding nodes that are not in the direction of destination node. A* search is the classical goal-directed shortest path algorithm.

Separator based techniques exploit the fact that a road network graph can be split into smaller graphs by removing small number of vertices $S \subset V$ called separators. These separators can be used to compute an overlay graph over S such that distances between any pair of vertices from S is preserved. Thus much smaller graph is used which accelerates the query processing.

Hierarchical methods exploit the inherent hierarchy of road networks. Contraction Hierarchies is one such algorithm.

2.2 Goal-Directed Techniques

Goal-directed techniques are useful in directing search towards the destination by avoiding the vertices that are not in the direction of destination t.

2.2.1 A* search

A* search uses a potential function π:V→R on vertices which is the lower bound on the distance dist(u,t) from vertex u to vertex t. Modified version of Dijkstra's algorithm is run where the priority of vertex u is dist(s,u) + π(u). The vertices closer to destination are scanned earlier and if the lower bound is exact (i.e. π(u) = dist(u,t)) then only vertices along the shortest path from origin to destination are scanned. One can use geographical distances between two vertices divided by the maximum travel speed as the potential function. Unfortunately, the bounds are poor.

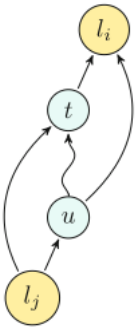

Figure 2.1: Triangle inequaties for ALT [4]

One can obtain better lower bounds with *ALT (A*, landmarks and triangle inequality)* algorithm. During preprocessing phase, well spaced landmarks are picked from the set of vertices. It stores the distances between all landmarks and vertices of the graph. Triangle inequality is used to obtain the lower bound for dist(u,t) (See figure 2.1). For any landmark l_i, dist(u,t) ≥ dist(u,l_i) - dist(t,l_i) and dist(u,t) ≥ dist(l_i,t) - dist(l_i,u). If several landmarks are available, one can use the maximum bound. The quality of the lower bounds depends on which vertices are chosen as landmarks during preprocessing.

2.2.2 Geometric Containers

For each edge e = (u,v), this algorithm computes label L(e) which stores the set of vertices to which shortest path from u begins with edge e. Labels store the coordinates of the vertices. During a query, if detination vertex is not in L(e), the search can be pruned at e. The preprocessing phase is very costly since it requires computation of all-pairs shortest path.

2.2.3 Arc flags

During preprocessing, it partitions the graph into K cells having roughly same number of vertices. Each edge maintains a K-bit vector with i-th bit set if the arc lies on shortest path to some vertex of cell i. The search can thus prune edges not having bit set to the cell containing destination vertex.The arc flags for a cell i are computed by growing a backward shortest path tree from each boundary vertex (of cell i), setting the i-th flag for all arcs of the tree. Arc Flags currently have the fastest query times among purely goal-directed methods for road networks.

2.3 Separator-Based Techniques

Separator based techniques exploit the fact that a road network graph can be split into smaller graphs by removing small number of vertices $S \subset V$ or edges called separators.

2.3.1 Vertex Separators

Separators can be used to compute an overlay graph over S such that distances between any pair of vertices from S is preserved. Subset $S \subset V$ can be the set of carefully chosen important vertices (not necessarily separators). For each pair of vertices u,v ∈ S, an edge (u,v) is added to the overlay graph if shortest path from u to v in original graph does not contain any other vertex w from S. Thus much smaller graph is used which accelerates the (parts of) query processing.

2.3.2 Arc Separators

This class of algorithm uses arc separators to create overlay graphs. See figure 2.2. In first step, one computes a partition C = (C_1 , . . . , C_k) of the vertices into balanced cells while attempting to minimize the number of cut arcs (which connect boundary vertices of different cells). Shortcuts are then added to preserve the distances between the boundary vertices within each cell. The query algorithm then (implicitly) runs Dijkstra's algorithm on the subgraph induced by the cells containing s and t plus the overlay.

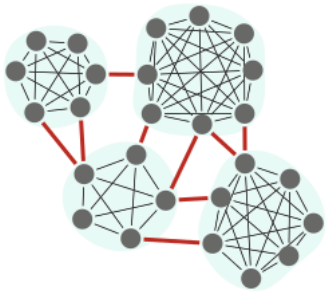

Figure 2.2: Overlay graph constructed from arc separators [4]

2.4 Hierarchical Techniques

Hierarchical methods exploit the inherent hierarchy of road networks. Contraction Hierarchies is one such important algorithm which we shall discuss.

2.4.1 Contraction Hierarchies

There are algorithms which are faster and memory efficient than Dijsktra's algorithm and A* algorithm like ***Contraction Hierarchies*** algorithm. They do time consuming preprocessing on graphs but once the preprocessing is done, individual queries run very fast on the graph.

Google Maps use something similar to Contraction Hierarchies to calculate driving directions.

Preprocessing the graph: Preprocessing is done by contracting the nodes one by one in some order. To perform contraction of a node, find shortest path between node's neighbors and insert shortcuts between them iff the shortest path includes the node to be contracted. See figure 2.3. Contracting a node means ignoring that node from now on because whenever we see an edge going towards the contracted node, we know that a shortcut edge exists if it was in a shortest path. Therefore we can ignore that node. Usually the node which reduces the number of edges in the graph is preferred for contraction.

Finding the shortest path: Two searches are performed, one from origin node and one from destination node and then the intersection point is found. Search from origin node is similar to Dijkstra's algorithm but only the edges that goes towards higher contraction order node from the current node are considered. Search process from destination node also works in same way. The intersection of list of nodes visited in forward and reverse searches is taken and one route is put after another to get the entire route. If many routes are possible, one with the lowest cost is preferred.

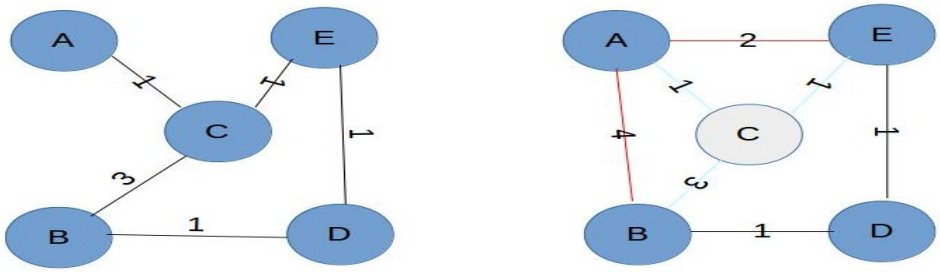

Figure 2.3: Contraction Hierarchies

Experimental Results: The UK map data contains 3,478,051 nodes and 7,646,738 (non-shortcut) edges. The contraction process added 6,186,168 shortcuts. The searches from London and Edinburgh visited at 3,968 and 3,113 nodes respectively, and the solution path found passed through 3,617 nodes.

2.5 Bounded-Hop Techniques

The idea is to precompute distances between pairs of vertices, implicitly adding "virtual shortcuts" to the graph. Queries then returns the length of a virtual path with very few hops.

2.5.1 Transit Node Routing

During its preprocessing it, it carefully selects a small set of vertices $T \subset V$ (usually vertex separators used) and computes all pairwise distance between them and stores it in distance table. It computes for each vertex $u \in V \setminus T$, set of access nodes $A(u) \subset T$. A transit node $v \in T$ is an access node of u if there is a shortest path P from u in G such that v is the first transit node contained in P. For origin (s) - destination (t) query a path is chosen that minimizes s-a(s)-a(t)-t distance where a(s) and a(t) are access nodes of s and t repectively. If shortest path does not involve vertex from T, CH algorithm can be used to get the correct path.

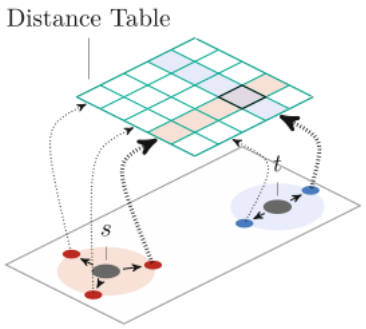

Figure 2.4: Illustrating a TNR query [4]

See figure 2.4. Here there are 3 access nodes of s and two access nodes of t. The highlighted entries in table represents access nodes that minimized the combined s-t distance.

2.6 Experimental Results

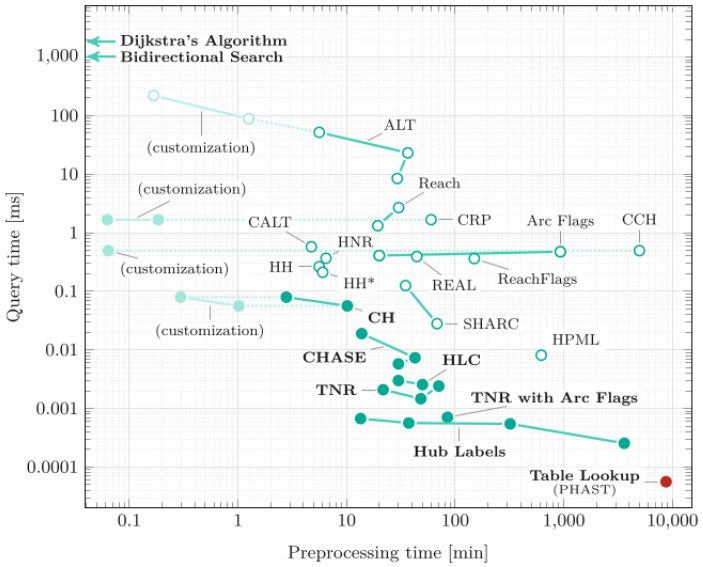

Figure 2.5: Preprocessing and average query time performance for algorithms [4]

See figure 2.5. It shows preprocessing and average query time performance for algorithms with available experimental data on the road network of Western Europe, using travel times as edge weights. Connecting lines indicate different trade-offs for the same algorithm. The figure includes two versions of Dijkstra's algorithm, one Dijkstra-based hierarchical technique (CH), three non-graph-based algorithms (TNR, HL, HLC), and two combinations (CHASE and TNR+AF). CHASE is the combination of CH and Arc flags. For reference, it also includes a goal-directed technique (Arc Flags) and a separator-based algorithm (CRP), even though they are dominated by other methods. Separator-based (CRP), hierarchical (CH), and goal-directed (Arc Flags) methods do not use much more space than Dijkstra's algorithm, but are three to four orders of magnitude faster. Finally CHASE improves query times by yet another order of magnitude, visiting little more than the shortest path itself.

Chapter 3

Trip Planning using Public Transport

In this chapter, we discuss two ways to model our data namely Time-Expanded model and Time-Dependent model. Then we discuss two algorithm for Trip planning using Public Transport. Frequency based algorithm represents all travel plans as trees and produces least cost tree in polynomial time. It uses frequency of buses to estimate the arrival time of buses. Second algorithm i.e. RAPTOR applies shortest path algorithm on a different kind of graph where edges represent different routes in the graph.

Route planning in Road networks is similar to finding shortest path in graphs. But for Public Transports, it is not this simple. The shortest path may not be the optimal path. Time dependence nature and Multicriteria nature of Public transport requires something more than simple shortest path algorithms.

Modeling: Our algorithms for public transports work on timetable. Timetable consists of a set of stops, a set of routes, and a set of trips. Trips correspond to individual vehicles that visit the stops along a certain route at a specific time of the day.Since the shortest-path problem is well understood in the literature, it seems natural to build a graph G = (V, A) from the timetable such that shortest paths in G correspond to optimal journeys. There are two main approaches of modeling the timetable.

Time-Expanded Model: This model creates a vertex for every event (like departures, arrivals) of timetable. Edges are the elementary connections between two events and cost of each edge $e_{u,v}$ is Time(v)-Time(u).The main disadvantage of this model is the large size of resultant graphs.

Time-Dependent Model: In this model, the vertices of the graph corresponds to the bus stops. Edge $e_{u,v}$ exists between vertices u and v if there exists a bus serving u and v in this particular order. The departure and arrival times of such buses are encoded with each edge of the graph.

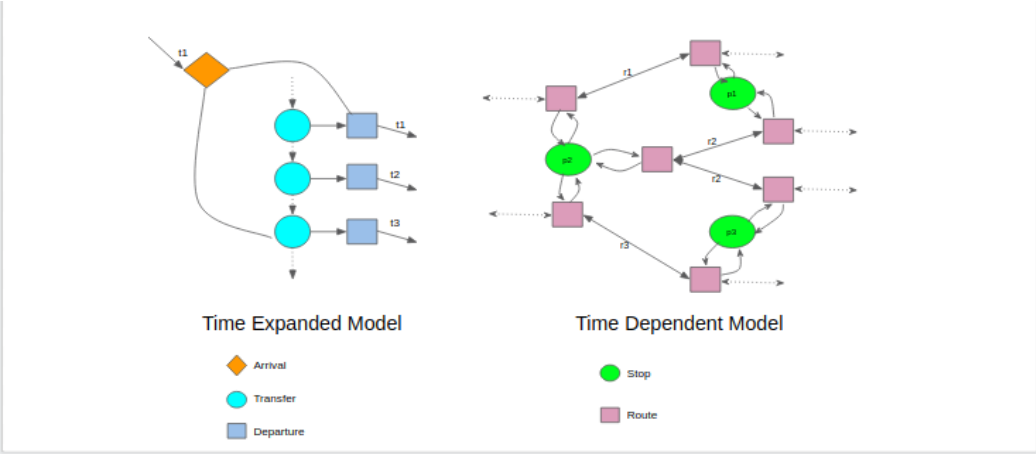

Figure 3.1: Time-expanded and Time-dependent models. Connection arcs in the time-expanded model are annotated with its trips t_i , and route arcs in the time-dependent model with its routes r_i.

3.1 Frequency based algorithm

We want to go from point A to point B in the fastest way possible. It would have been a classical shortest path problem if buses ran according to a fixed schedule. But buses don't do that usually due to external factors like traffic jams or bus breakdowns.

The travel time is usually predictable once you hop on a bus but the waiting time for a bus is random and can only be estimated statistically. This algorithm gives travel plan with minimum expected time. It represents all travel plans as trees and produces least cost tree in polynomial time [3].

Assumptions made:

- Assumes travel time to be constant.
- It does not take time-dependence nature of public transport into account.

Bus route refers to the sequence of bus stops. If a bus route b with frequency $F(b)$ is considered then the expected time between their arrivals is $1/F(b)$, assuming arrivals of bus on route b to be Poisson distributed.

$R_b(u,v)$ is the travel time i.e. the time taken to travel from stop u to stop v by taking bus b. It should be noted that $R_b(u,v)$ may not be equal to $R_{b'}(u,v)$ if halting stops of bus b and bus b' are different.

Adaptive travel plans are better in delay prone world. In the travel plan tree (See figure 3.2), nodes represent the bus stops and edges represent bus routes between two stops. Root node S(u) represents the source vertex and leaves represent the destination stop.

The plan is as follows: Wait at S(u) for buses B(u,s1),B(u,s2),...B(u,sk). If B(u,si) arrives first, take it till stop S(si) and execute the action indicated by si. Subtrees also represent plan trees, therefore calculation of expected time taken by a plan happens recursively.

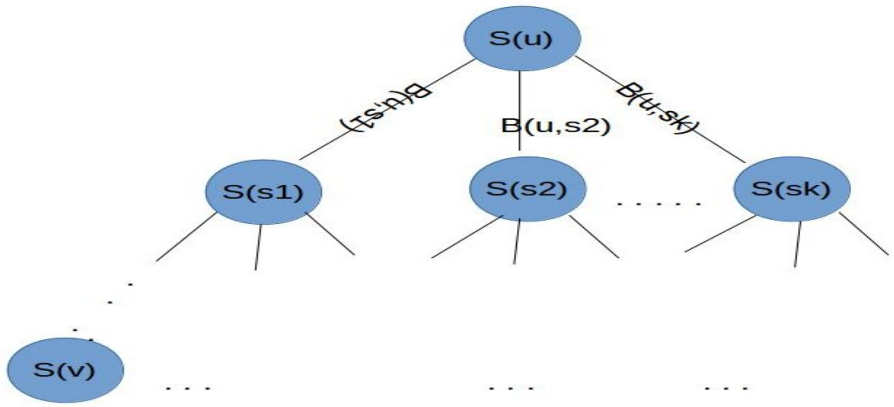

Figure 3.2: Travel Plan Tree

$T(u)$ is the time taken to reach destination starting at stop u and b_i = B(u,si).

$$T(u) = \frac{1}{\sum_i F(b_i)} + \sum_i \frac{F(b_i)}{\sum_i F(b_i)}(R_{bi}(u,si) + T(si)) \tag{3.1}$$

Maximum number of transfers given by user can be incorporated in the travel time by using height of the travel plan tree. $T(u,h)$ is the expected time to reach destination starting at u using an optimal plan of height h. $T_b(u,h)$ is a bus restricted plan of height h starting at stop u in bus b.

$$T_b(u,h) = min(T_b(u,h-1), min_i(R_b(u,si) + T(si,h-1))) \tag{3.2}$$

This algorithm considers only a subset of buses B (say $B1$ to B_k) from all buses B_s that goes through stop s and takes bus from B1,.. B_k whichever arrives first.

- Sort all B_s buses based on running time
- Compute waiting time , running time for first i buses
- Find j s.t. running time for bus B_{j+1} is greater than sum of waiting and running time for bus B_j
- Use buses 1..j as B

$$T(s,h) = min_{B \subseteq B_s}(\frac{1}{\sum_{b \in B} F(b)} + \sum_{b \in B} \frac{F(b)}{\sum_{b \in B} F(b)} T_b(s,h)) \tag{3.3}$$

This is a polynomial time algorithm with complexity $\mathcal{O}(HLlogC)$ where C is the maximum number of buses at each stop, L is the sum of lengths of all bus routes and H is the height provided by user.

Preprocessing is done to find time between consecutive stops from the given end-to-end time for each route. This algorithm does not use real time data of buses which would provide more detailed information about the waiting time for other buses especially during peak hours. Also, it does not optimize criteria like travel cost.

3.2 Round-based public transit routing

RAPTOR, Round bAsed Public Transit Optimized Router deals with trip planning problem in public transports [5]. It's main focus is on minimizing the travel time and minimizing number of transfers made while travelling. This algorithm is not Dijkstra's based and requires no preprocessing.

Drawbacks of Dijkstra based alorithms:

- hard to parallelize
- criteria other than travel time are also important when travelling is done using public transport.
- preprocessing becomes impractical while dealing with public transit systems due to delays and cancellations.

RAPTOR is a dynamic program with simple data structures. It organizes the input as simple arrays of trips and routes and has excellent memory locality.

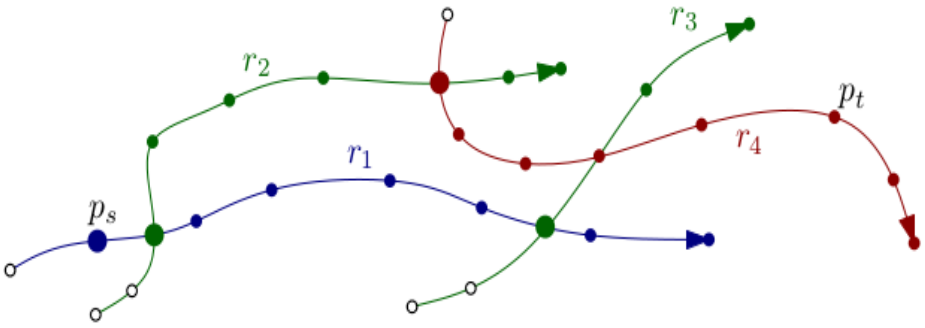

Figure 3.3: Scanning routes for a query from p_s to p_t [5]

Algorithm 1 RAPTOR [5]

Input: Source and target stops p_s , p_t and departure time τ .

```
1: //Initialization of the algorithm
2: for each i do
3:     Set τ_i(.) to INF
4: end for
5: Set τ*(.) to INF
6: Set τ_0(p_s) to τ
7: Mark p_s
8: for each k = 1,2 ... do
9:     //Accumulate routes serving marked stops from previous round
10:    Clear Q
11:    for each marked stop p do
12:        for each route r serving p do
13:            if (r,p') ∈ Q for some stop p' then
14:                Substitute (r,p') with (r,p) in Q if p comes before p' in r
15:            else
16:                Add (r,p) to Q
17:            end if
18:        end for
19:        unmark p
20:    end for
21:
22:    //Traverse each route
23:    for each route (r,p) ∈ Q do
24:        Set t to ⊥ //the current trip
25:        for each stop p_i of r beginning with p do
26:    //Can the label be improved in this round? local and target pruning
27:            if t ≠⊥ and arr(t,p_i) < min(τ*(p_i),τ*(p_t)) then
28:                τ_k(p_i) = τ_arr(t,pi)
29:                τ*(p_i) = τ_arr(t,pi)
30:                mark p_i
31:            end if
32:            //Can we catch an earlier trip at p_i?
33:            if τ_{k-1}(p_i) ≤ τ_dep(t,p_i) then
34:                t = et(r,p_i)
35:            end if
36:        end for
37:    end for
38:
39:    //Look at foot-paths
40:    for each marked stop p do
41:        for each footpath (p,p') ∈ F do
42:            τ_k(p') = min(τ_k(p'),τ_k(p)+l(p,p'))
43:            mark p'
44:        end for
45:    end for
46:
47:    //Stopping-criterion
48:    if no stops are marked then
49:        stop
50:    end if
51: end for
```

See Algorithm 3.2. It works in rounds where round k computes earliest arrival times to each stop p with exactly k transfers $\tau_k(p)$. It also keeps track of earliest arrival time achieved so far for each stop from origin $\tau^*(p)$. Each round takes as input only the marked stops from previous round. Marked stops are those whose arrival time improved in the previous round. It then scans the routes served by these stops by traversing its stops in order and updating the arrival times whenever better arrival times are found. Each round scans each route at most once.

See figure 3.3. For a query from p_s to p_t, route r_1 is first scanned in round 1, routes r_2 and r_3 in round 2, and finally, route r_4 in round 3. Scanning a route begins at the earliest marked stop (bold). Hollow stops are never visited.

The worst-case running time of our algorithm can be bounded as follows. In every round, we scan each route r ∈ R at most once. If | r | is the number of stops along r, then we look at $\sum_{r \in R} |r|$ stops in total. Caching et(r, ·), we look at every trip t of a route at most once. Thus, the total running time of our algorithm is linear per round. In total, it takes $\mathcal{O}(K(\sum_{r \in R} |r| + |T| + |F|)$ time, where K is the number of rounds.

Improvements such as local pruning and target pruning makes this algorithm very efficient, even better than Dijkstra's algorithm with single criteria. Multiple CPU cores can be used to parallelize RAPTOR where each core can handle different subset of non-conflicting routes.

Experimental results: Experiments happened on the transit network of London. The resulting graph had 100,878 vertices and 283,587 edges. 10,000 queries were run between random origin-destination pairs. RAPTOR benefits from its simpler data structures with an average query time of 7.3ms which was 9 times faster than Multi-Label-Correcting(MLC) and 6 times faster than Layered Dijkstra(LD).

Chapter 4

Using Real time data in Trip Planning

In this chapter, we first discuss the need of real time data in trip planning using Public Transports. Then we discuss two algorithms which incorporates this real time data to accurately predict travel times using public transports including time spent waiting for the buses. First algorithm uses Time-Expanded modeling and updates the graph every time it obtains real time data. Whereas second algorithm uses Time-Dependent modeling to build a graph using static, real-time and historical data to predict fast and reliable routes.

4.1 Need for Real time data in Trip Planning

The need for real time data arises because of the unreliable nature of bus schedules. Buses may get delayed due to traffic congestion on roads or due to bus breakdowns, due to which algorithms not using real time location of buses may predict inaccurate travel times. The problem of bus bunching may arise where many buses arrive at a particular stop simultaneously and no buses arrive afterwards for a long period of time. We need real time information of buses to reduce the amount of time wasted waiting for delayed vehicles. Algorithms using real time data change the travel plans according to whether the departure time falls in peak or off-peak hours of the day or falls on weekdays or weekends. It also provides user with the choice of picking better and faster alternative routes.

4.2 Time-Expanded model algorithm

The algorithm [2] fuses real time data with existing schedule based trip planners. System architecture (see figure 4.1)consists of client, server and third party information provider. Client selects his origin and destination and the server determines k-shortest paths between origin and destination. The third party information providers provide static data (such as bus schedules and route configuration) and dynamic data (such as bus arrival time). Static data include names of routes which serve the bus stop, scheduled arrival time of each bus at stops, etc. Given static data is used to build transit network graph whereas dynamic data is used to update the wait and travel time between links.

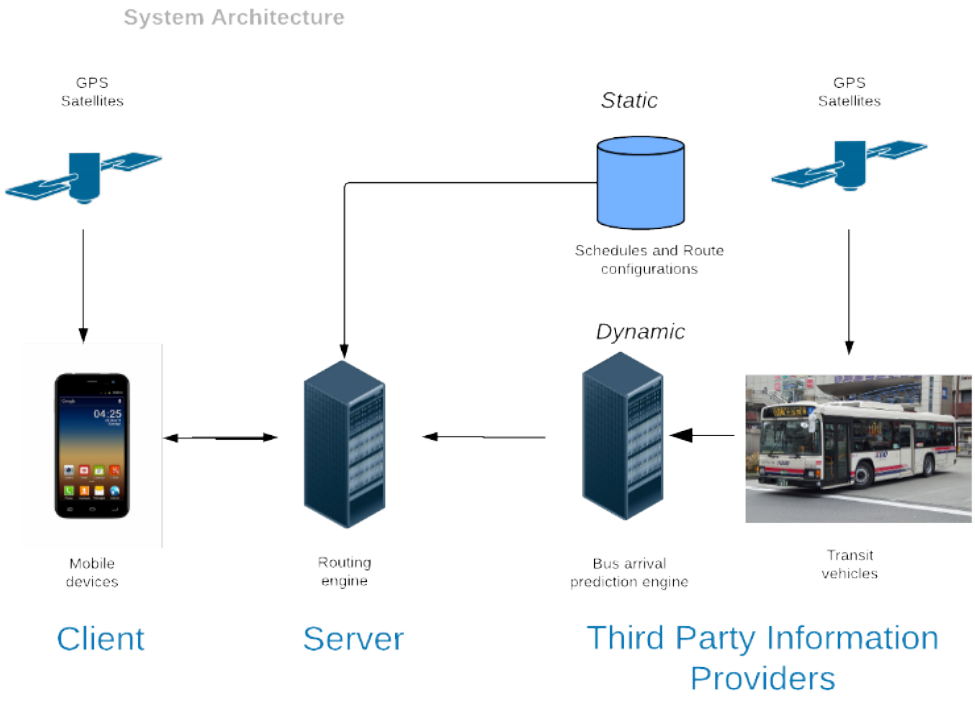

Figure 4.1: System Architecture

The algorithm determines k-shortest paths instead of a single shortest path in order for the user to decide between a set of optimal routes, using personal preferences. The set of optimal routes is constructed in two steps. First, we construct a graph representing all possible ways to go from any point in the network to any other point. The construction of this graph requires static schedules and the resulting graph is stored in a database. Next, the dynamic updates of the graphs are constructed using real-time information in the form of updates to the static graph.

Static Network Flow Framework: Time expanded graphs are used for modeling here. Time indexed vertices $V\ x\ T$ are used . Vertex (v,t) corresponds to stop v at time t and $c_{(v,t),(v',t')}$ corresponds to the cost of an edge between vertex (v,t) and (v',t'). The transit network graph is the union of waiting subgraph, walking subgraph and riding subgraph.

Let us look at these subgraphs:

Waiting subgraph: As it is possible to wait everywhere, it assigns $c_{(v,t),(v',t+1)} = 1, \forall v \in V, \forall t < |T| - 1$.

Walking subgraph: For all pair of walkable stops (v,v') it assigns $c_{(v,t),(v',t+d(v,v')/w)} = d(v,v')/w\ \forall t < |T| - d(v,v')/w$. where d(v,v') is the distance between the stops and w is the walk speed.

Riding subgraph: For every pair of the graph connected by transit option at time t, it assigns $c_{(v,t),(v',t+d(v,v')/r_{v,v,t})} = d(v,v')/r_{v,v',t}$ where $r_{v,v',t}$ is the average ride speed.

Dynamic Network Flow Problem: As time is marched and delay information of buses is obtained (lets say at time t), the edges in the time expanded graph are replaced with new edges. Suppose vehicle that left the stop u at time $\theta < t$ scheduled to arrive at time θ' = θ + d(u,v)/$r_{u,v,\theta}$ at destination, is known to be delayed by time t_d, dynamic updates take place where edge $e_{(u,\theta)(v,\theta')}$ is removed and new edge $e_{(u,\theta)(v,\theta'+t_d)}$ is added. Similarly all adjacent edges are updated.

All feasible paths between all origin-destination pairs can be precomputed using static schedules and these paths can be modified whenever updated

edge data becomes available. Best k paths are outputted to user and user can pick any one according to his preference.

Feasible paths where number of transfers exceed a maximum number of transfers preferred by users can be dropped.

Experimental results: Over 600 trips were planned over schedule based transit trip planner and the real-time transit trip planner using real time data from San Francisco Municipal Transit Authority buses running over 87 routes across the city.

- Accuracy of the estimated transit vehicle arrival engine: Estimated arrival times between 0 and 10 minutes have a median error of zero, and the interquartile range tend to fall within a minute of the true travel time.
- Accuracy of the estimated trip travel time: The median error of the schedule-based TTP is 14.9% vs 11.7% for the real-time TTP.
- Route selection optimality: In most cases, the real-time TTP outperformed the schedule-based TTP by a small margin.

4.3 Time-Dependent model algorithm

This algorithm [1] uses two types of data to dynamically predict routes based on departure time of users.

- Real time bus arrival time which accurately predicts bus waiting time at first stop
- Historical smart card data to predict travel time and also waiting time at future stops.

The system includes three types of data sources:

- Historical data includes past travel data used to build models for bus travel time and bus waiting time in form of probability distributions.
- Real time data gives the expected bus arrival times.
- Static data such as bus schedules, number of stops and buses are used to create a time-dependent graph.

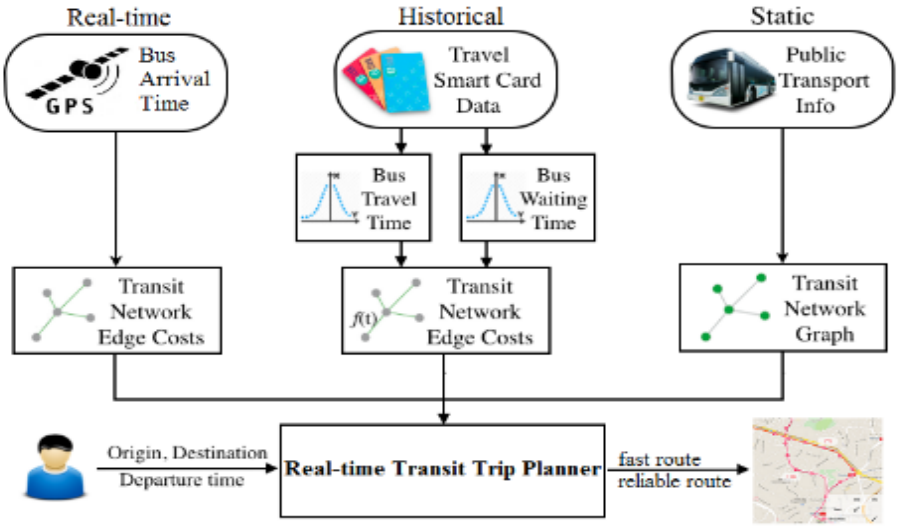

Figure 4.2: Architecture of a real-time trip planner[1]

This algorithm recommends routes depending on whether the departure time falls in peak or off-peak hours of the day or falls on weekday or weekend. This algorithm favors both fast and reliable routes.

The key information that we need to extract from the historical data is the estimated travel time (including boarding and alighting of passengers) and its variance between any two stops of a bus line during each time interval of a day. Specifically, every record of the data is treated as a sample point. It includes information such as boarding/alighting stops, boarding time and total travel time of that trip. These trip records are grouped according to the boarding stops, alighting stops, bus line number, and time interval within a day (by default every 30 minutes). Given all entries within a group, it is possible to calculate the mean and standard deviation of the corresponding travel time.In particular, instead of choosing the best matching time interval that the given departure time falls into, we take the best matching and the second best matching intervals, then use their weighted average as the expected travel time.

Another information we need to extract is estimated waiting time by the frequency of bus arrivals during given time interval of the day.

Time dependent graph is the main data structure used here and multiple criterion costs are associated with each edge of the graph.The stochastic models of bus travel time, bus waiting time and real time bus arrival times are included as cost of edges in this graph.

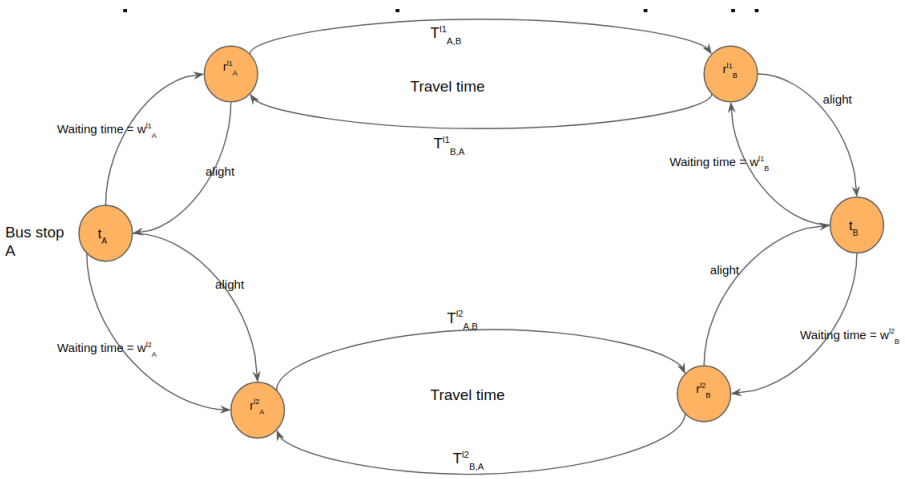

Figure 4.3: Time-dependent transportation graph

Time-dependent transportation graph: In this graph (see figure 4.3) we have two types of nodes: transfer nodes t_A and route nodes $r_A^{l1}..r_A^{lk}$. There are multiple route nodes corresponding to a transfer node if there are multiple buses going through that stop. There are three kinds of edges in this graph:

- Edge from transfer node t_A to route node r_A^{l1} represents bus boarding process. The edge cost is the waiting time for bus $l1$ at stop A. Waiting time at first stop is updated based on real time data whereas historical data is used to update waiting time at future transfers.
- Edge from route node r_A^{l1} to transfer node t_A represents alighting process and edge cost is usually taken to be 30s.
- Edge between route node r_A^{l1} to route node r_B^{l1} for same bus represents the travel process and its cost is the travel time estimated using historical data.

Bus waiting time and travel time have two properties namely, mean and variance of the distribution, representing speediness and reliability of routes respectively. In this time-dependent graph, edges have multi-dimensional cost where each dimension corresponds to each optimization criteria such as speediness and reliability.

Algorithm 2 Modified multi-criteria shortest path [1]

Input: graph g, source node n_s , destination node n_d , departure time t
Output: shortest paths and their expected costs

priorityQueue $pq = \phi$
predecessorMap $pm = \phi$
nodeCostsList $cl = \phi$
label l_s = createLabel($n_s, c_s = 0$)
pq.insert(l_s)
while pq!=ϕ **do**
    label l_u = pq.pop()
    node n_u = l_u.getNode()
    for each outgoing edge $e_{u,v}$ **do**
        cost $c_v = getCost(l_u, e_{u,v}, t)$
        List<cost> *costs* = cl.getCosts(n_v)
        if c_v *is not dominated by costs* **then**
            /* c_v is not dominated */
            cl.put(n_v, c_v)
            cl.removeDominated()
            label l_v = createLabel(n_v, c_v)
            pm.put(l_v, l_u)
            pq.insert(l_v)
        else
            /* drop c_v since it is not smaller than any of existing costs */
            continue
        end if
    end for
end while

See Algorithm 4.3. A node label l_i is defined by (n_i, c_i) where c_i is the multi-dimension cost to reach n_i from origin. Priority queue, which is sorted on mean travel time of c_i, determines the nodes to be explored next. Predecessor Map keeps track of the predecessors of nodes and cl is the list of all non dominated costs. In this algorithm, in each iteration the outgoing edges

of all lowest cost labels are examined. $getCosts()$ adds edge costs (real time waiting time or expected waiting/travel time) depending on the edge type to current cost. If this new cost is better than the existing costs of that particular node, then all dominated costs are removed and new label is created and added to the priority queue. Algorithm stops when queue becomes empty.

This algorithm gives quite accurate expected travel times and suggests fast and reliable dynamic routes based on departure times.

Experimental results: The data used for building stochastic models of bus travel time and waiting time are collected from real smart card integrated bus network in Singapore and comprise of the recorded usage of all bus lines in a period of three months, which includes about 300 million of trip records. Real-time transit trip planner is compared with Google Maps 2 and Gothere.sg 3.

- Accuracy of expected total travel time: A set of trip instances are selected from the historical smart card data as the ground truth. Gothere.sg suffers from the largest error with travel time 40% different from ground truth. The travel time estimation by Google Maps during peak hours drops sharply with error as high as 22%. Accuracy of Real-time Trip planner is best of all three and error is usually below 10%.
- Real-time transit trip recommendation: Gothere.sg and Google Maps are static and return same routes in spite of different departure times whereas RTTP returns routes adapted to traffic situations.
- Favoring both fast and reliable routes: RTTP utilizes multi-criteria algorithm and finds both least expected travel time and most reliable routes.

Chapter 5

Conclusion

In this report we discussed the unreliable nature of public transport. We discussed various shortest path algorithms for Road Networks and compared different algorithms based on their preprocessing and query times. Things get complicated when time-dependent and multicriteria nature of Public transports come into picture.

Static algorithms for trip planning using public transports like Frequency based algorithm or RAPTOR, suggest same routes for all departure times and may give incorrect expected travel times. Two algorithms were given for Trip Planning in Public transports using real time data. While one used time-expanded modeling, the other used time-dependent modeling of graphs. These algorithms predicted accurate expected travel times and gave fast and reliable routes.

With real time data, we get more accurate predictions for total trip times using public transports and there are large number of cases in which optimal route are suggested. But the routes given by these algorithms may not be optimal for longer trips as optimal route may change with time and recalculation of optimal route may be required at different stages of trip.

Bibliography

[1] Hoang Tam Vo.Enabling Smart Transit with Real-time Trip Planning.SAP Innovation Center, Singapore, 2015.

[2] J. Jariyasunant, D. Work, B. Kerkez, R. Sengupta, S. Glaser, and A. Bayen. Mobile Transit Trip Planning with Real–Time Data. Transportation Research Board 89th Annual Meeting, Washington, D.C., Jan. 10–14, 2010.

[3] M. Datar and A. Ranade. Commuting with delay prone buses. In proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms, pages 22-29, 2000.

[4] Hannah Bast, Daniel Delling, Andrew Goldberg, Matthias Müller-Hannemann, Thomas Pajor, Peter Sanders, Dorothea Wagner, and Renato F. Werneck. Route Planning in Transportation Networks, 2015.

[5] Delling, D., Pajor, T., Werneck, R.F. Round-based public transit routing. In Proceedings of the 14th Meeting on Algorithm Engineering and Experiments (ALENEX 2012), pp. 130–140. SIAM (2012).

[6] Justin Boyan, Michael Mitzenmacher. Improved Results for Route Planning in Stochastic Transportation Networks. In Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms Pages 895-902, 2001.

[7] Konstantinos G. Zografos and Konstantinos N. Androutsopoulos. Algorithms for Itinerary Planning in Multimodal Transportation Networks. IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS, VOL. 9, NO. 1, MARCH 2008.

[8] Yu ZHANG, Jiafu TANG, Lily CHEN. Itinerary Planning with Deadline in Headway-based Bus Networks Minimizing Lateness Level. In Proceedings of the 11th World Congress on Intelligent Control and Automation Shenyang, China, June 29 - July 4 2014.