Trip planning using past data

MTP Thesis

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

JUNE, 2020

Acknowledgment

I am extremely grateful to my guide, Prof. Abhiram Ranade and research scholar Dr. Sumit Sen for their unwavering guidance and support. I would also like to thank Mr. Dillip Rout for his help throughout. The meetings I had with them were integral in giving me direction and clarifying any doubts or queries I had. I would also like to thank my parents for their constant support and encouragement.

Abstract

Trip planning means finding optimal travel plans between two locations on a given day at given departure time such that certain criteria like travel time, travel cost and transfers are optimized. There are different trip planners that optimizes different criteria. Some planners are independent of specific departure time (Mumbai Navigator) , some uses historical data to leverage time-dependency (Mutli label correcting algorithm), whereas some planners use both real time data and past data (MLC planner using real time data) to generate time dependent, fast and reliable plans. We planned to develop such planners, by using the large ticketing data of Mumbai public transport buses. The data being incomplete and noisy, we moved on to creating our own ticketing like synthetic data to use with the trip planners. We made modifications in the Mumbai Navigator to incorporate the departure time of the user to generate time interval specific plans. We implemented two variants of MLC planners, one that uses only past data and other that uses real time data and past data. Best time estimator using RAPTOR algorithm is developed to find the best possible plan for the given day, time, source and destination. Experiments are performed to measure the travel time prediction accuracy of the planners and its ability to generate best plans. The results show that using real time data improves the predicted travel time accuracy of planners and also helps in generating best possible plans for the day.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

What is the optimal way to travel from point A to point B in a city using its public transport service?

Public transportation network consists of bus stops and bus lines where buses follow specific routes as per schedule. Trip planning means finding optimal travel plans from a given source to a given destination at a given day and departure time. Travel plans may require changing of buses at intermediate stops to reach the destination. These plans are considered optimal if they optimize certain criteria like travel time, transfers or travel cost.

Formally, for a given source s, destination d and departure time t, the goal is to generate travel plan tp which specifies the routes to be taken with their associated pickup and drop points along with the estimated time to reach the destination.

Travel plans can be a sequence of trips or can be represented as trees.
Sequence of trips looks like:

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

where $v1$ is the source and v_n is the destination and each element of the set specifies intermediate stops to be travelled along with the bus that needs to be taken.
Travel plans represented as trees are of the form shown in figure 1.1. Here the plans are adaptive, picking buses which comes first at a stop. Travel plan tree contains routes and stops that results in minimum average travel time.

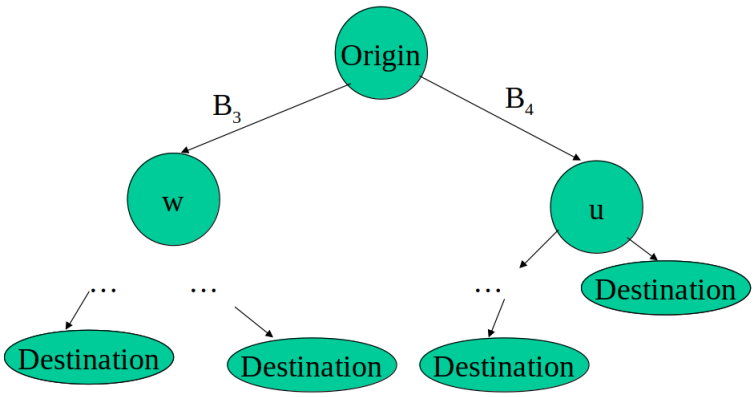


Figure 1.1: Travel plan tree

In the above figure 1.1, user picks from bus B3 and B4, whichever comes first at the origin and gets down at the mentioned drop point and picks next bus that comes first at this drop point. This continues until destination is reached.

This chapter is an introduction to Trip Planning in Public Transport Networks. It answers why simple approaches (like shortest path algorithms) do not work for Public Transport Trip Planning. It discusses different variants of Trip Planning and how to model the timetable of Public transport. Lastly, a brief introduction to few Trip Planning approaches that are used in this project is given.

Public transport network can be considered as a graph with bus stops as vertices and there is an edge between two vertices when a bus line connects them. Shortest path algorithms for graphs can not be used to find optimal travel plans for public transports because:

- Public transportation network is time-dependent. *Time dependence* means that the best travel plans for the same source-destination may vary depending on the departure time of the user from the source. This happens due to traffic congestion and the fact that buses can travel only at specific points of time. Figure 1.2 shows different plans for source A and destination B for different departure times.

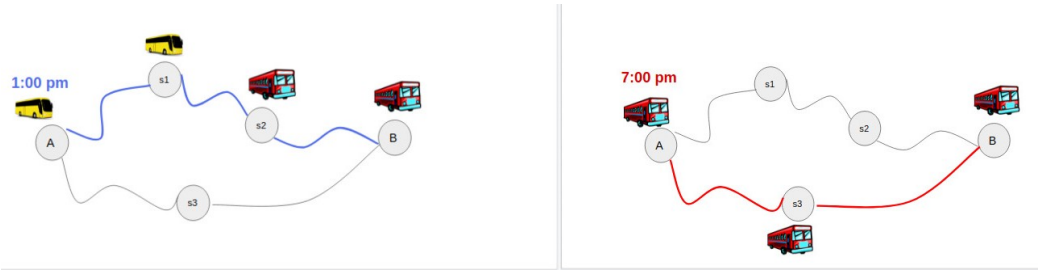


Figure 1.2: Time-dependent nature of public transports

- While travelling by public transport, one may want to minimize travel time, number of transfers or monetary travel cost. The optimal travel plans may change with change in optimization criteria. Figure 1.3 shows different travel plans for different optimizing criteria for same source and destination.

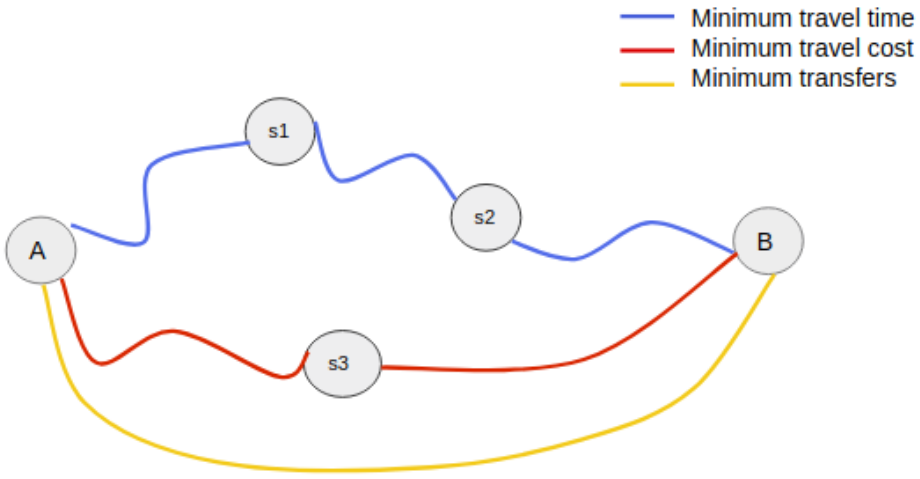


Figure 1.3: Multicriteria nature of public transports

1.1 Variants of Trip Planning

There are different problem variants of Trip Planning:

Earliest Arrival Problem: Given source, 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 source, 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 source, destination, departure time and criteria to be optimized, the problem asks for all non dominating travel plans optimizing given criteria. 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. For example, Let plans optimize two criteria: travel time and number of transfers. A plan $tp1(30,1)$ with 30 minutes travel time and 1 transfer, dominates plan $tp2(32,2)$ with 32 minutes travel time and 2 transfers. Whereas $tp1(30,1)$ and $tp3(25,2)$ are non-dominating set of plans because $tp1$ is better in one criterion and $tp3$ is better in other criterion.

1.2 Approaches to Trip Planning in Public Transit Networks

For trip planning in Public Transit Networks, the input is given by a timetable. A timetable consists of a set of stops (bus stops or train platforms) and a set of routes (like bus or train lines). With each route we have set of trips which corresponds to vehicles visiting stops along a route at a specific time of the day. Trips have arrival times associated with each stop of the route.

1.2.1 Modeling

Modeling [4] is to create a graph from timetable so that shortest paths in graph correspond to optimal plans.

- ***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). To enable transfers between vehicles, all vertices at the same stop are interlinked by transfer (or waiting) edges. 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.

- ***Frequency-Based Model:*** Trips on a route often run according to specific frequencies at different times of the day. A bus on a route may run in every 10 minutes during peak hours, and every 15 minutes otherwise. The modeling here is same as time-dependent model, but arrival times of trip at each stop is calculated based on the frequency of buses at a particular time.

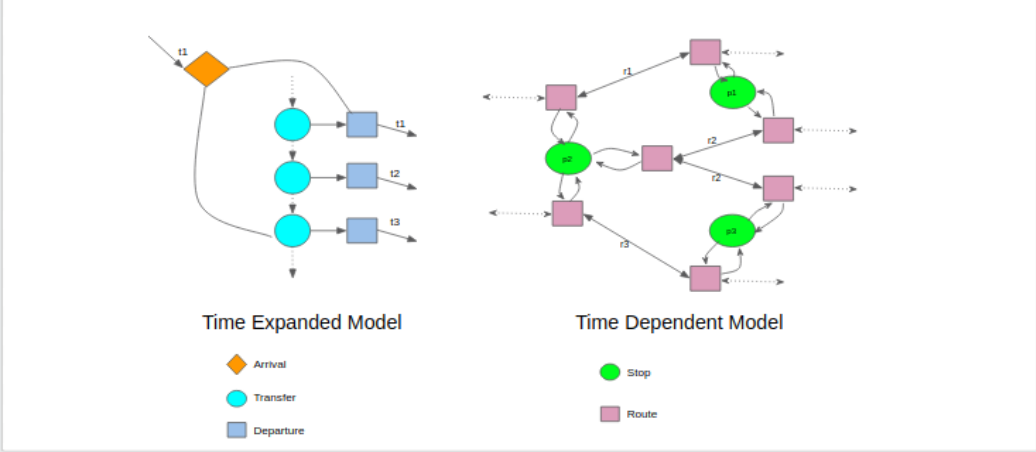


Figure 1.4: Time-expanded and Time-dependent models. Connection arcs in the time-expanded model are annotated with its trips t_i , and route arcs in the time-dependent model with its routes r_i [4]

1.2.2 Approaches

Some approaches to Trip planning in Public Transit Networks is discussed briefly here.

- ***Mumbai Navigator:*** Mumbai Navigator [3] takes set of stops, set of routes, routes frequencies and travel time between consecutive stops of the route as database input; takes source and destination as user input and outputs a travel plan tree with minimum average travel time (refer fig 1.1) for each number of transfer. This plan is adaptive, based on taking buses (from suggested buses in the plan) that arrive first at each stop and getting down at mentioned stop until we reach the destination. This algorithm assumes bus arrivals follow Poisson distribution with expected time between arrivals being 1/F(b) where F(b)

is the frequency of bus b. The expected time taken by a plan is evaluated recursively, since sub trees of a given plan tree are themselves plan trees.

- ***RAPTOR:*** RAPTOR [5] is a Round-bAsed Public Transit Optimized Router which takes bus timetables and trips information (arrival/departure times) as database input. Given source stop, destination stop and departure time of the user, RAPTOR computes optimal travel plans for public transport networks based on two criteria: arrival time and number of transfers. This algorithm requires no preprocessing. This Bi-criteria Problem (optimizing arrival time and number of transfers) can be interpreted as a series of Earliest Arrival Problems: For each number of transfers k, we pick plans with minimal arrival time among all feasible plans with k+1 trips. Thus, the resulting non-dominating set contains at most one optimal solution for each number of transfers. The RAPTOR algorithm exploits this observation with a dynamic programming approach that operates in rounds. In round k, it computes optimal solutions with k trips (k-1 transfers). This is done by taking the solutions from the previous round and extending them by one trip. In each round, each route is processed at most once.

- ***Multi-label correcting (MLC) algorithm:*** This algorithm [1] solves the Multicriteria problem on time-dependent model. It extends Dijkstra's algorithm by keeping set of non-dominated labels for each vertex. Each label contains multiple values, one for each optimization criteria. Priority queue maintains unprocessed labels and in each step extracts the minimum label and scan its outgoing edges. For every edge (u,v), a new label is created by adding edge costs (from past or real time data) to the extracted label values and this new label is added to the queue if is not dominated by other labels of vertex v (eliminating other dominating labels). A specific case of this algorithm is when arrival time and its variance is minimized to ensure speediness and reliability and all non-dominated plans minimizing both arrival time and variance are outputted.

1.3 Contributions

The objective of this project is to develop Trip planners that generates optimal plans based on the optimizing criteria for a given source, destination and departure time. All planners use some data (past data, real time data) to estimate travel time and generate best plans.

In this project we have evaluated three trip planners: Modified Mumbai Navigator, MLC planner that uses only past data and MLC planner that uses both real time data and past data. We have made modifications in the original Mumbai Navigator and implemented MLC planners from scratch.

Modified Mumbai Navigator generates plan trees that minimizes expected travel time and number of transfers. It takes the interval (morning, afternoon or evening) of departure time into account and uses appropriate database to generate best plans in that interval.

MLC planner generates multiple non-dominating travel plans that minimizes travel time, variance and the number of transfers. MLC planner has two variants based on the type of data used for the waiting time for buses at the source stop. First variant uses only past data to generate plans. Second variant uses past data as well as real time data. It uses the real time data to predict the waiting time for a bus at the source stop. We have developed both the variants of MLC planner.

For the MLC planner that uses real time data, since we don't have GPS data, we pick a random day from the past and consider that day to be the current travel day and use all the bus information till the given departure time of that day to simulate GPS. This gives the position of all the buses at the given departure time.

We evaluate the planners by comparing the time it actually takes in following the plan on a day to the best possible time for the day, and the time predicted by the planners. The best possible plan is the best plan that can be followed on the travel day to arrive earliest at the destination with minimum number of transfers. To generate the best plan, we assume to know the future, i.e. provide the complete schedule of the travel day and use RAPTOR algorithm to get the best plan.

Trip planners need public transport timetable as input. A timetable consists of routes, stops, frequency of routes, stop to stop travel time of buses, etc. This timetable is then modeled into appropriate form required by the trip planner to generate travel plans and estimate travel time.

BEST (Brihanmumbai Electricity Supply and Transport) provided us with tickets data containing the ticket records issued on Mumbai buses in the month of July 2017. We were hopeful to use this ticket data information

as input data for different trip planners. But due to its incompleteness and noisiness, we had to drop it and generate our own synthetic data to be used for trip planners. Our code is modular and can be used with the ticket data, if provided in future.

We compare the results of different planners using this data with the actual time it takes to follow a particular plan on a random day and also with the best possible plan for that random day.

1.4 Overview of the Report

Chapter 2 discusses the format of BEST ticket data and the information extraction process. Information extraction process includes assigning trip IDs to tickets and calculating the occupancy and arrival time of trips at each stop. For trips with no tickets issued at stops, we need to predict the arrival time at stops based on the data provided by other trips.

Chapter 3 discusses the Trip data set, which is the compressed data set generated from ticket data, which can be used as an input for different planners. It basically contains all the past trips, its occupancy and arrival time at different stops. Chapter 3 also mentions various queries that can be answered using this Trip data set.

Chapter 4 explains Mumbai Navigator in detail. It discusses the input data used by it and the plan tree generation process. Since Mumbai Navigator does not consider the departure time of the user to generate plans, we had to make changes in it to generate plans according to the departure time interval of the user.

Chapter 5 discusses the changes made in MN, the conversion of Trip data to MN data format and the adding a Simulator code that calculates the actual time taken to follow the Mumbai Navigator generated plan on a day.

Due to inconsistencies found in the given BEST data, we generated our own clean Trip data in chapter 6. The variations in speed of buses and frequencies was introduced to model the real behaviour of buses.

Finally, Chapter 7 discusses different planners implemented like MLC planner using past data and MLC planner using both real time data and past data. Best time estimator is also discussed in this chapter which uses RAP-

TOR algorithm to generate best possible plans for the travel day (knowing the future). It also gives the comparison of plans generated by all the planners (MN, MLC) with the actual time taken by a plan and the best possible time to reach the destination. We use RMSE (Root mean squared error) to compare the accuracy of different planners in estimating the travel time and to know how often it suggests best plans to users.

Chapter 2

BEST data

The Brihanmumbai Electricity Supply and Transport (BEST) is a civic transport and electricity provider public body based in Mumbai, India. BEST provided us with the tickets data generated from the Electronic Ticket Issuing Machines (ETIM) in buses over the month of July 2017. Each ticket belongs to a trip and contains the ticket issue time, stops for which the ticket was bought, etc. Information needed by the planners can be extracted from this ticket data and used to create a smaller trip data set. Trip data set contains the required information for trip planning which can be used to generate efficient plans.

The ticket data is huge and consists of approximately 52 million ticket records of size 14 GB. Querying this data is very time consuming. The aim is to create a "Trip data" which is much smaller in size and answers queries efficiently. The trip data set contains information about each trip made by a vehicle and its arrival time and occupancy on each stop. To generate Trip data from ticket data, we need to identify and group tickets that belong to the same trip and then calculate the occupancy and arrival time at each stop. We assume that the time at which the first ticket is issued at a stop of a trip is the arrival time of the trip, but there are cases when no ticket is issued at a stop or a ticket is issued much later.

This chapter majorly discusses the process of information extraction from the BEST data. In section 2.1, it discusses the format of BEST data and its various fields. Information extraction in section 2.2 includes finding routes and its corresponding stops, grouping tickets into trips, finding occupancy and arrival time of trips at stops. The size of BEST data is huge and can be reduced to a smaller data set called trip data. This data set can be given as input to different trip planners as it contains all the required information

and is compressed in size. In section 2.3, we look at the few problems identified in the algorithms used to extract information and find new fields in the ticket data that can be used to correct them.

2.1 Format of data provided by BEST

Ticket data consists of multiple ticket records and each ticket record has 35 fields. The field headers were provided separately, which did not entirely match with the fields of the data provided. After figuring out, what each field in data corresponds to, some important fields are shown in table 2.5.

Other important fields identified in the data are: *ticket sequence number*, *single ticket cost*, *"from" stage number* and *"to" stage number*. Each ticket record tells us about the route, day and time at which the ticket was bought, from which stage-stop to which stage-stop was the ticket issued, the bus number on which the ticket was bought, the cost of the ticket and so on..

Ticket records Example								
no. of tickets bought	total cost	route no.	bus no.	ticket date	ticket time	"from" stage name	"to" stage name	trip direction
1	8	440	196589	01/07/17	07:00	SHRAWAN YESH-WANTE CHOWK	LOWER PAREL STA-TION	U
2	20	390	114866	04/07/17	20:46	SEEPZ BUS STATION	CHAKALA	D
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

Table 2.1: Example of BEST data showing 9 fields

Note: The ticket issue time at a stage-stop for a bus maybe different from the arrival time of a bus at that stage-stop. Also, there may be some routes or stage-stops that are not present in the ticket data.

From the data, it was found that there are 514 routes and 986 stage-stops.

There are 52146199 ticket records in total. Out of these records, 7 ticket

records have *ticket time* out of range; 153014 tickets records does not have *ticket date* between 01/07/2017 and 31/07/2017; In 1457 tickets records *(single ticket cost * no. of tickets bought)* don't match the *total cost*. In total, there are 154477 inconsistent ticket records and these records are filtered out of the data. After removing inconsistent tickets and tickets containing NULL fields, we now have 51985932 ticket records.

2.2 Information extraction from BEST data

The BEST data we have is very large (~14GB) which is very difficult to handle and query. In order to get information like occupancy of a particular bus at a stop, average occupancy of buses on a route at all stops at given times, arrival time of buses on stops, etc.. we need to read a huge database and perform computations to get the result. These computations take lot of time. For example, consider this query: *Find the average occupancy of buses at all stops on route 396 between 10:00 a.m. and 11:00 a.m.*. Here we need to extract all tickets bought on route 396 between 10:00 a.m. and 11:00 a.m. and calculate the occupancy of bus at a particular stop by looking at the number of tickets bought on previous stops. We need to predict the arrival time of bus at a stop where ticket was not bought, which indeed requires finding the average travel time between consecutive stops of the route. Now the average of occupancy of buses passing between 10:00 am and 11:00 am should be calculated at each stop of the route. Thus, finding answer to a simple query requires lot of computations.

In order to save space and time, we create ***trip-data*** (refer 2.2.5) which is much smaller than the original data set and gives results to our queries in few seconds. To generate trip-data, we need to assign trip ID to each ticket record, find occupancy and arrival time of trips at each stop.

2.2.1 Assigning Trip ID's

We have tickets data but we don't know which ticket belongs to which trip as the data of same trips are not present together. For example: On a particular route on a particular day if tickets are sorted based on ticket time, Ticket i may belong to one trip, Ticket i+1 may belong to some other trip and so on.., as multiple trips can be running on a route at the same time.

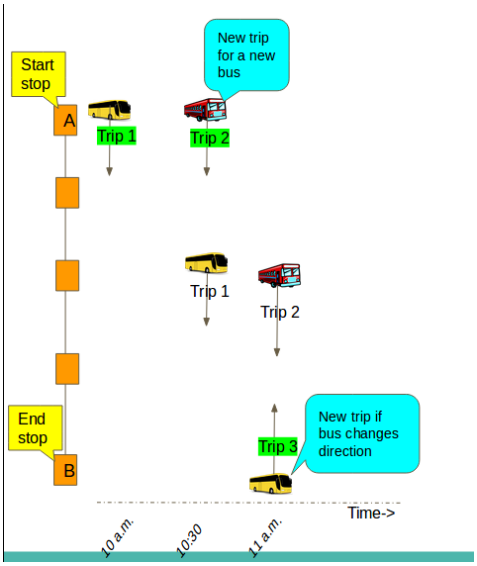


Figure 2.1: Trips made on route x on day d

We need to find all trips made in a day and the tickets issued on that trips. When a new bus starts its journey or the same bus changes its direction, a new trip is created, see figure 2.1.

Algorithm 1 Assign Trip ID

Input: route number

```
Fetch all the tickets issued on given route number
Sort tickets based on ticket date and ticket time
for each date do
    for each ticket of a date do
        Find bus number and trip direction of that ticket
        if bus number already present that day then
            if direction of bus changed then
                Update the current direction of bus
                Assign new trip ID to this ticket
                 Save the trip ID assigned to this bus
            else
                trip ID for this ticket is same as the current trip ID assigned for this bus
            end if
        else
            Save bus number and its direction
            Assign new trip ID to this ticket
            Save the trip ID assigned to this bus
        end if
    end for
end for
```

Execution time of Algorithm 1 is 50 seconds
Output: Every ticket is now associated with a trip ID which tells the trip on

which the ticket was issued, see table 2.2.

Ticket records Example							
trip ID	route no.	bus no.	ticket date	ticket time	...	trip direction	...
440_01/07/17_1	440	198403	01/07/17	05:13	...	D	...
440_01/07/17_2	440	197334	01/07/17	05:27	...	U	...
440_01/07/17_2	440	197334	01/07/17	05:43	...	U	...
440_01/07/17_3	440	197334	01/07/17	06:00	...	D	...
...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...

Table 2.2: Trip IDs assigned to tickets

Trip IDs are unique and of the form *routenumber_ticketdate_tripoftheday*.

2.2.2 Order of bus stops in a route

From the data, we know the bus stage-stops present in a route. From now on, we will refer stage-stops as stops. We need to find the order in which these stops are travelled in a route. Order of the stops of a route get reversed when direction is changed.

We need to consider all the trips taken on a route to get the order, since picking a single trip might not assign order to stops where tickets were not bought.

Create a graph with bus stops in a route as nodes. Edge exists from stop s_i to stop s_j, if in some trip, ticket was bought from stop s_i to stop s_j or a ticket at stop s_j was bought after a ticket issued at stop s_i. We perform topological sort on the graph to get the order of stops. See algorithm 2. Note that garbage data where tickets at s_j were issued before s_i for some trips must be removed by maintaining count for edges and reverse edges.

Algorithm 2 Order of bus stops in a route

Input: route number, trip direction

Fetch all the tickets issued on given route number on a given direction
Sort tickets based on trip ID and ticket time
for each trip **do**
    $prevStop = NULL$
    for each ticket of a trip **do**
        $fromStop$ = ticket["from" stage number]
        $toStop$ = ticket["to" stage number]
        Create edge in graph from node *fromStop* to node *toStop*, if not already there
        $count[fromStop][toStop]++$
        if $prevStop$ not NULL and $prevStop$!=$fromStop$ **then**
            Create edge in graph from node *prevStop* to node *fromStop*, if not already there
            $count[prevStop][fromStop]++$
        end if
        $prevStop = fromStop$
    end for
end for
for each pair of nodes **do**
    if edge exists from node i to node j and node j to node i **then**
        Remove edge with smaller count
    end if
end for
Perform Topological sort on graph to get the order

Execution time: 5 seconds
Output: Ordered bus stops in a route, see figure 2.2.

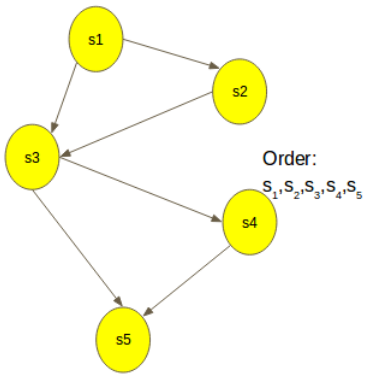


Figure 2.2: Topological sort on bus stops in a route

2.2.3 Occupancy of trips at stops

Occupancy of buses matter a lot to users. User may want to board a bus where he can find a seat. Therefore, it is important to get the occupancy of trip at a stop. Here we calculate occupancy of buses for our given ticket data, which later will be useful to generate optimal plans.

Algorithm 3 Occupancy of trips at stage-stops

Input: route number

1: Fetch all the tickets issued on given route number
2: Sort tickets based on trip ID and ticket time
3: Get stops present in route in order
4: **for each** trip **do**
5: Maintain a 2D array *tripwise_occupancy* initialized with 0.
6: $tripwise\_occupancy[i][j]$ represent count of tickets bought from stop i to stop j.
7: **for each** ticket of a trip **do**
8: $tripwise\_occupancy[ticket[fromstageno.]][ticket[tostageno.]]++$
9: **end for**
10: **for each** stop P of a trip **do**
11: $occ[P] = occ[P-1] + boardingAtP - alightingAtP$
12: **end for**
13: **end for**

Execution time: 1 min 40 seconds
Output: Occupancy of every trip at every stop is found, refer table 2.3.

To get the occupancy of a trip at stop P, we add the number of people boarding at P to the occupancy of trip at P-1 and subtract the number of people alighting at P.

Occupancy Example			
tripID	stop no.	occupancy	trip direction
4941_01/07/17_11	333	1	U
4941_01/07/17_11	254	5	U
...	...	...	...
4941_01/07/17_13	412	23	U
...	...	...	...

Table 2.3: Occupancy of trips at stops

2.2.4 Arrival time of trips at stops

If we assume that tickets were issued immediately after person boarded the bus, then the arrival time of trip at a stop will be the time of first ticket issued at that stop on that trip. But it's possible that no ticket was issued at a particular stop in that trip. Here we need to estimate the arrival time of trip at that particular stop. To do this we need to find the time it takes to travel between stops in a route.
Time interval of the day also matters while calculating travel times. For example, To go from stop i to stop $i+1$ it might take 3 minutes in non peak

Algorithm 4 Travel time between stops

Input: route number, trip direction

```
Fetch all the tickets issued on given route number
Sort tickets based on trip ID and ticket time
Get stops present in route in order
for each trip do
    Find arrival time of trip at stops where tickets were bought
    for each consecutive stop pairs (i, i+1) do
        if arrival time at both stops is known then
            Find the time window in which it belongs
            Calculate the arrival time difference (t_{i+1} − t_i)
            Find the mean travel time between (i, i+1) in that time window
        end if
        if arrival time at second stop of the pair is not known then
            Find the next closest stop (j) where arrival time is known
            Find the time window in which it belongs
            Find the time difference (t_j − t_i)
            Find number of stops that lie in between
            Divide the time difference among those stop pairs
            Find the mean travel time for all such stop pairs in that window
        end if
    end for
end for
```

hours whereas it might take 8 minutes in peak hours. Thus travel time between stops must be found at all times of the day. We have window wise travel time between stops. Window can be hourly windows. In case of missing times in a window look at the time in closest window.

Now that we have travel time between consecutive stops of a route, we can find the arrival time of trip at stops where ticket was not bought.

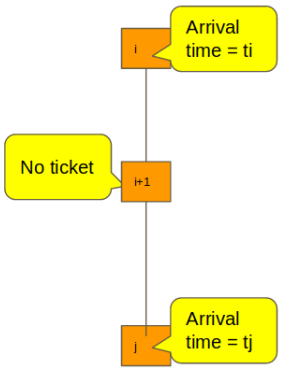


Figure 2.3: Arrival time at stop $i+1$ with no ticket

Refer figure 2.3. Use calculated mean travel times to find the ratio of time

taken between stop i and stop $i+1$ to time taken between stop $i+1$ and stop j in that time window. Divide the time difference ($tj - ti$) according to the ratio to get the arrival time at stop $i+1$.

Algorithm 5 Arrival time of trips at all stops

Input: route number, trip direction

Fetch all the tickets issued on given route number on given direction
Sort tickets based on trip ID and ticket time
Get stops present in route in order
for each trip **do**
    Find arrival time of trip at stops where tickets were bought
    if arrival time at start stop is missing **then**
        Find first stop (say f) where arrival time is present
        Use calculated mean travel times to find arrival times at all stops before f
    end if
    for all stops $i+1$ with missing arrival time **do**
        Find stop after stop $i+1$ where arrival time is present (say stop j)
        Find ratio of time between $(i, i+1)$ and $(i+1, j)$ in appropriate time window to estimate arrival time at $i+1$ from calculated mean times
        Divide the time diff $t_j - t_i$ according to the ratio and get arrival time at stop $i+1$
    end for
end for

Execution time of both codes: 3 mins
Output: Arrival time of trips at all stops, refer table 2.4.

Arrival times Example			
tripID	stop no.	arrival time	trip direction
4941_01/07/17_11	333	08:45	U
4941_01/07/17_11	254	08:52	U
...	...	...	...
4941_01/07/17_13	412	09:54	U
...	...	...	...

Table 2.4: Arrival time of trips at stops

Problem: The python code written for calculating occupancy and arrival times of trips at stops took 20 hours to run for 514 routes. The python code used pandas Data frame to store all tickets of a route. A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns, which makes it very easy to process all tickets of a route by reading data just once from the file. Though it is easy but it is very time consuming.

Solution: Therefore, these codes are converted to C++ which uses vectors for storing tickets information. To access date and time related functions and structures, we need to include <ctime> header file in our C++ program. There are four time-related types: clock_t, time_t, size_t, and tm. The types - clock_t, size_t and time_t represents the time and date as some sort of integer.

The structure type tm holds the date and time in the form of a C structure having the following elements [8]

```
struct tm {
    int tm_sec; // seconds of minutes from 0 to 61
    int tm_min; // minutes of hour from 0 to 59
    int tm_hour; // hours of day from 0 to 24
    int tm_mday; // day of month from 1 to 31
    int tm_mon; // month of year from 0 to 11
    int tm_year; // year since 1900
    int tm_wday; // days since sunday
    int tm_yday; // days since January 1st
    int tm_isdst; // hours of daylight savings time
}
```

The important functions, used while working with date and time in C++ are:

*char * asctime (const struct tm * time);*
This returns a pointer to a string that contains the information stored in the structure pointed to by time converted into the form: day month date hours:minutes:seconds year.

*struct tm *gmtime(const time_t *time);*
This returns a pointer to the time in the form of a tm structure. The time is represented in Coordinated Universal Time (UTC), which is essentially Greenwich Mean Time (GMT).

*time_t mktime(struct tm *time);*
This returns the calendar-time equivalent of the time found in the structure pointed to by time.

double difftime (time_t time2, time_t time1);
This function calculates the difference in seconds between time1 and time2.

size_t strftime();
This function can be used to format date and time in a specific format.

The total time taken by these algorithms to create trip data set for all routes is 20 mins.

2.2.5 Trip data

From the information extracted, we create a new data set of form Table 2.5.

This new data set (called trip data set) is much smaller in size as compared to original data set.

Trip data set Example				
tripID	stop no.	arrival time	ocupancy	trip direction
4941_01/07/17_11	333	08:45	1	U
4941_01/07/17_11	254	08:52	5	U
...	...	...	...	...
4941_01/07/17_13	412	09:54	23	U
...	...	...	...	...

Table 2.5: Trip data set

2.3 Problems identified

There were some issues identified during and after the generation of trip data.

2.3.1 Order of bus stops in a route

From the trip data set generated earlier, it was found that the time taken by bus to travel between consecutive stops was too high for some routes. Also, negative occupancy at stage-stops was noticed for some trips.

For example: For ***route 126***, the order of stage-stops generated using topological sort was 113, 114, 115, 507, 566, 3413, 1368, 1586, 438, 509, 531, 534, 295, 296, 969, 971, 565, 4326, 3610, 3611, 2281, 4223, 3750, 1411, 4224, 1329, 1315. The average time taken for bus to go from stop 507 to stop

566 was 1 hr 4 mins, which is very high as compared to time taken between other consecutive stops in the route.

It was also observed that routes such as 420 and 424 have some common stops but relative order of those stops was not followed. Also route 424 is a circular route which starts and end with same stops but was not identified.

Order of stops for ***route 422***: 665, 670, 671, 674, 688, 14, 692, 666, 807, 501, 751, 867, 863, 980
Order of stops for **route *424***: 867, 501, 980, 863, 14, 692, 666, 4400, 686, 751, 807, 688
Note: Look at the position of stop 688 in both routes.

Problem: Order of stops generated using topological sort is incorrect for some routes due to the timings at which the tickets were issued at some stops. It is possible that the tickets from earlier stops were issued much later in the journey.

Solution: Out of the 35 fields in ticket data, two new fields (*fromStageindex* and *toStageindex*) were identified in ticket record that corresponded to the sequence number of "from" stage-stop and sequence number of "to" stage-stop in a route. Algorithm 6 was used to identify order of stage-stops in a route.

Algorithm 6 Order of stops in a route

Input: route number
Fetch all the tickets issued on given route number
for each ticket **do**
  if $ticket[fromStageindex]$!=null and $ticket[fromStageno]$!=null **then**
    if $fromStopindex$ is already in $routeOrder$ and $routeOrder[fromStageindex]$!=$fromStageno$ **then**
      print Error message
      return
    else
      $routeOrder[fromStageindex] = fromStageno$
    end if
  end if
  if $ticket[toStageindex]$!=null and $ticket[toStageno]$!=null **then**
    if $toStopindex$ is already in $routeOrder$ and $routeOrder[toStageindex]$!=$toStageno$ **then**
      print Error message
      return
    else
      $routeOrder[toStageindex] = toStageno$
    end if
  end if
end for
Sort $routeOrder$ on key (i.e. Stageindex)
return $routeOrder$

New order of stops for the problematic routes are as follows-
route *126*: 113, 114, 115, 507, 438, 509, 531, 534, 295, 296, 969, 971, 565, 4326, 566, 3610, 3611, 2281, 3413, 1368, 1586, 4223, 3750, 1411, 4224, 1329, 1315
route *422*: 665, 670, 671, 674, 688, 14, 692, 666, 807, 501, 751, 867, 863, 980
route *424*: 980, 863, 867, 751, 501, 807, 666, 692, 14, 4400, 686, 688, 14, 692, 666, 807, 501, 751, 867, 863, 980

Note: route 424 is identified as a circular route, which was impossible to identify by performing topological sort because stops can't repeat if topological sort is performed.

Note: It was identified that there were ticket records with improper indices of source ("from" Stage) and destination ("to" Stage). In upward direction trip, the *fromStageindex* should be less than or equal to *toStageindex*. In downward direction trip, the *toStageindex* should be less than or equal to *fromStageindex*. There were 1611 records that violated this condition. These records were removed from the ticket data, therefore the total number of ticket records are 51984321 now.

2.3.2 Assigning Trip ID's

Problem: Trip ID initially assigned is of the form *routenumber_ ticketdate_ tripoftheday*. On careful observation of the generated trip data set, it was found that the Trip ID generation process is problematic for circular routes. According to old Trip ID assigning algorithm, a new trip is created when a new bus starts its journey or when the same bus changes its direction. But in the case of circular routes, the direction of buses remain same for different trips. Due to which, many ticket records were assigned same trip ID even though it belonged to different trips.

Let's consider a bus B on route R, which is a circular route. Let's say B started its first trip $T1$ from first stop at 10:00 a.m. and no ticket was issued on trip $T1$ at second stop and then a ticket was issued at third stop at 10:15 a.m. Suppose trip $T1$ finished and trip $T2$ started. Since $T2$ has same direction as $T1$, algorithm considers it as same trip and ticket issued by $T2$ at second stop at 11:35 a.m. seems to be the ticket issued at second stop on trip $T1$. One possible way to distinguish between the two is by looking at the huge time gap, but it might be the case that this ticket at second stop actually belonged to $T1$ and was issued much later by the conductor.

There is a field in ticket data which corresponds to the $tripnumber$ of a bus on a day on a particular route. So the combination of $routeno.$, $busno.$, $ticketdate$ and $tripnumber$ can be used as a trip ID. But, this Trip ID is not unique. For example, for $routeno.$ 10, $busno.$ 193374, $ticketdate$ 01/07/2017, two different trips, starting at different times of the day had $tripnumber$ as 1.

Solution: Now, combination of both ways of assigning trip ID is used to generate trip ID's for ticket records. Trip ID is of the form *routenumber_ ticketdate_ tripoftheday_ busnumber_ tripnumber*.

The trip ID formed is huge in size as it contains combination of five fields. To reduce its size, we replace the last three fields by a unique number given to the combination of these three fields. Now trip ID is of form *routenumber_ ticketdate_ i*, where *i* specifies that this specific trip is the *ith* trip made on a route on a day.

Chapter 3

Trip data set

Table 3.1 shows the trip data set generated from the information extracted in chapter 2. The size of this data set is 460 MB, which is much smaller than the original ticket data size. This data set is used as the database input for all trip planners as it contains all the information that can be used for trip planning. Also, this data can be used to answer various queries easily.

Trip data set Example				
tripID	stop no.	arrival time	occupancy	trip direction
4941_1_11	333	08:45	1	U
4941_1_11	254	08:52	5	U
...	...	...	...	...
4941_1_13	412	09:54	23	U
...	...	...	...	...

Table 3.1: Trip data set

In this chapter we look at various queries that can be answered with trip data. There are many queries that can be answered with our trip data in few seconds. Some of them are:

- What is the average occupancy of buses on route 3961 at I.I.T Market at 8:00 a.m.?
- What is the average occupancy of buses on route 3961 at all stops at all times?

- What is the average occupancy of all buses passing through I.I.T Market on different routes?
- Which buses run empty?
- Bus starting at 09:00 a.m. from Thane reaches I.I.T Market at what time?
- What is the frequency of buses on a particular route?

3.1 Query Outputs

Here are the outputs of some queries.

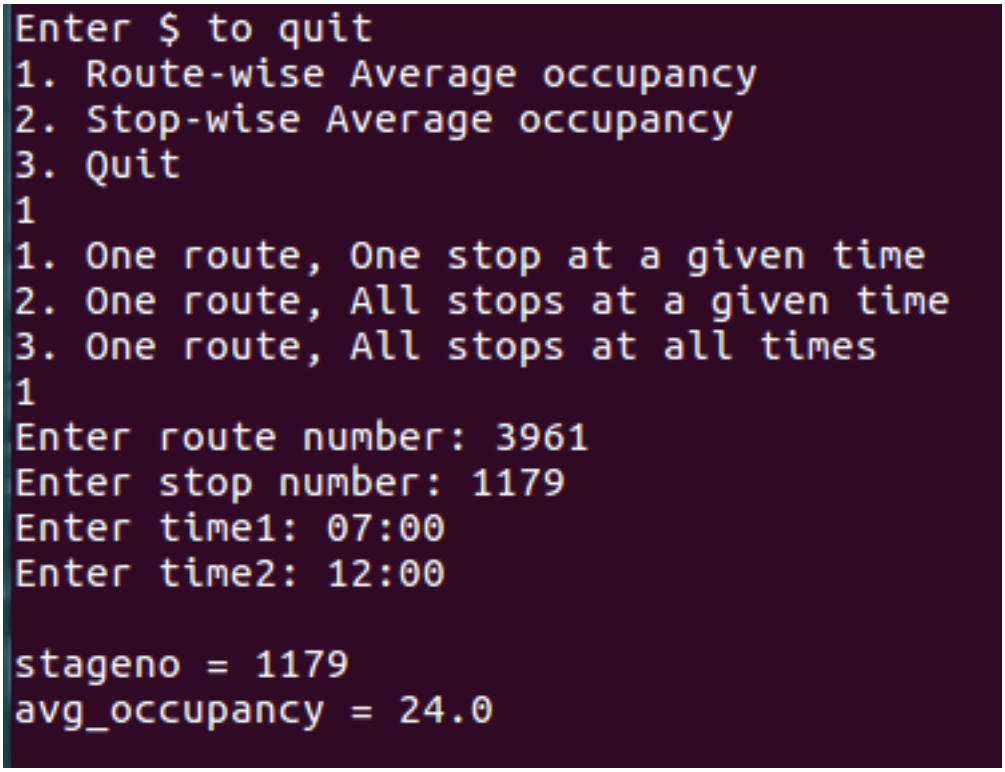


Figure 3.1: Avg occupancy of buses on route 3961 from 07:00 to 12:00 pm at I.I.T Market

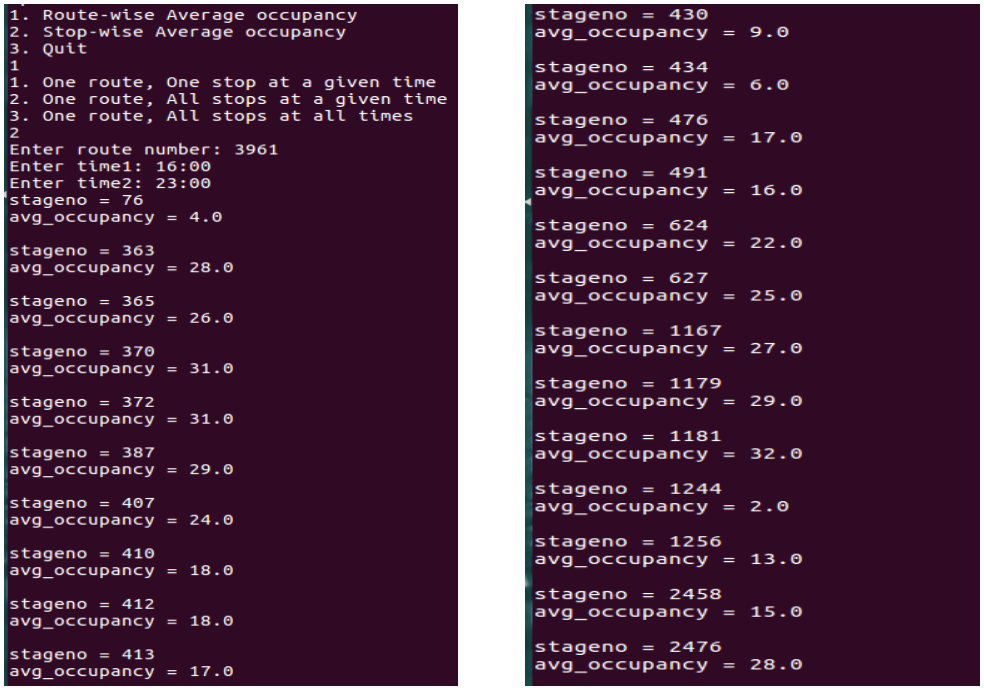


Figure 3.2: Avg occupancy of buses on route 3961 from 16:00 to 23:00

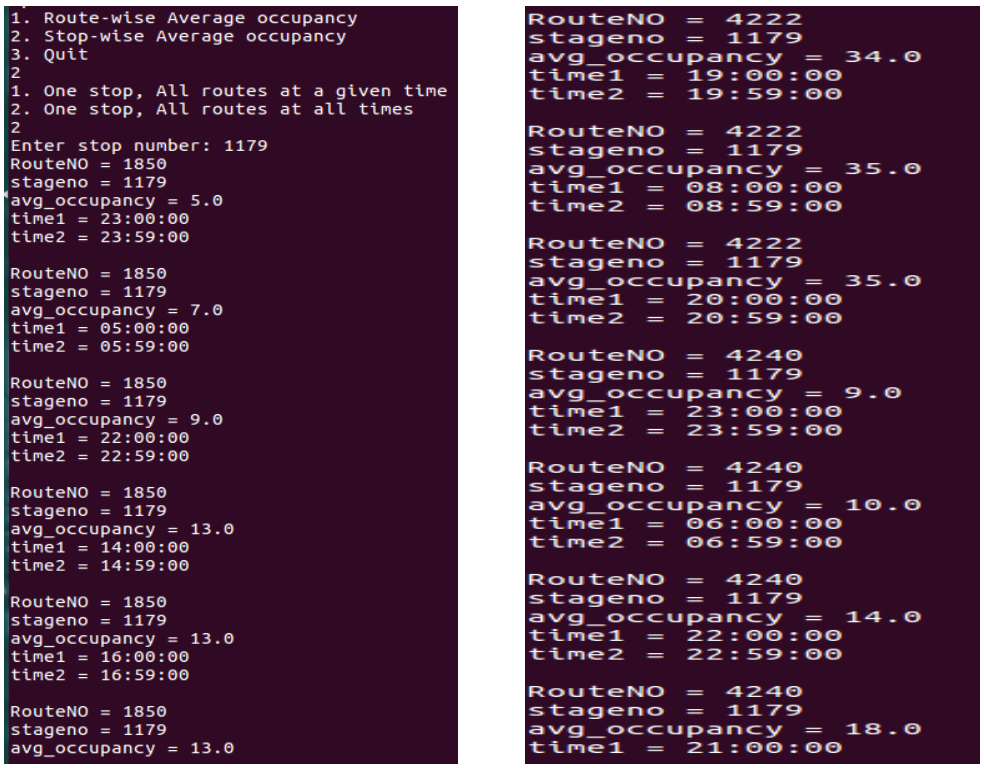


Figure 3.3: Avg occupancy of buses at I.I.T Market on all routes at all times

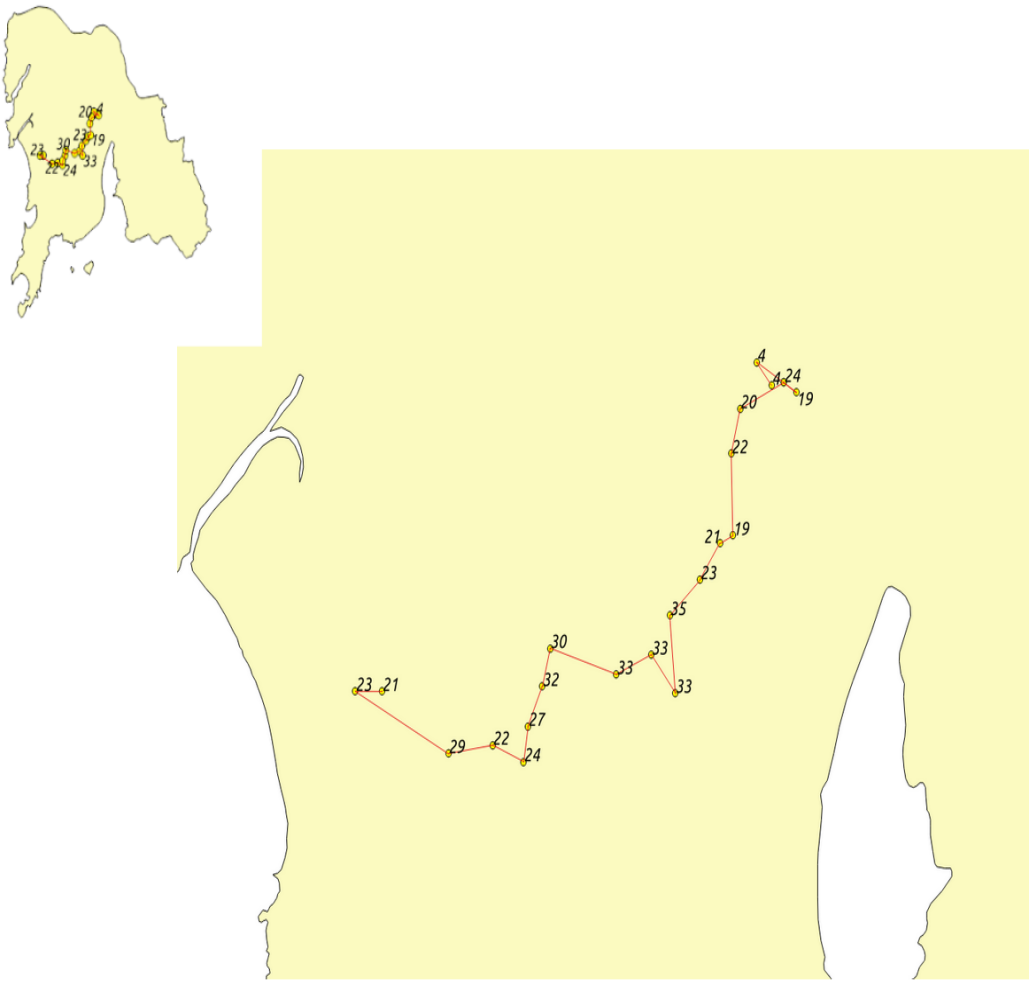


Figure 3.4: Avg occupancy of buses on route 3961 from 10:00 to 11:00 am

Figure 3.4 shows Average occupancy of buses on route 3961 from 10:00 a.m. to 11:00 a.m. on Mumbai map. Line in figure shows route 3961 and numbers on map represent the average occupancy of buses.

Chapter 4

Mumbai Navigator

Mumbai Navigator is a trip planner which determines the fastest way to reach destination from source. Since buses hardly run according to schedule, this problem can not be solved using Shortest path algorithms. 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 Mumbai Navigator algorithm gives travel plan with minimum expected time. It represents all travel plans as trees and produces least cost tree in polynomial time.

In this chapter we look at the algorithm of Mumbai Navigator and how it generates optimal plan trees for each number of transfers in section 4.1. In section 4.2, we move on to the understanding of the MN code written in Java. Mumbai Navigator generates same plan trees for given source and destination, irrespective of the departure time of the user. The understanding of the MN code was necessary to introduce changes into it, so that the generated plan trees differ based on the departure time of the user. MN uses a "busdata" file as database input. Thus we need to understand its format to convert our trip data into that form.

4.1 Algorithm

The problem is to go from point A to point B in the fastest way possible using public transport services [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 4.1), 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 generated by the algorithm 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.

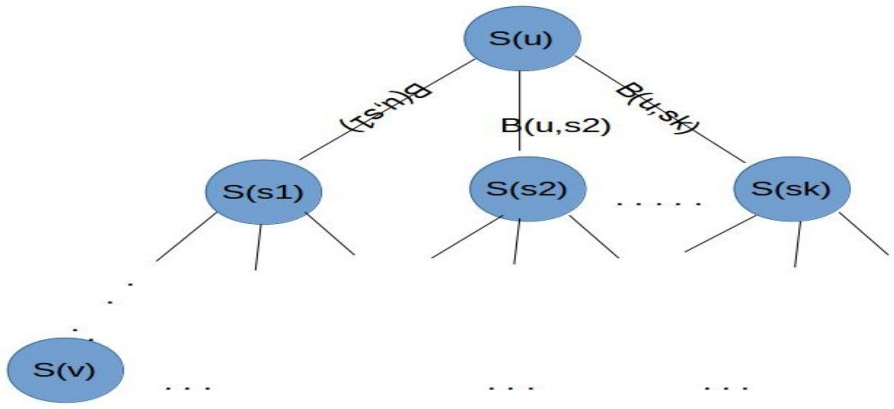


Figure 4.1: Travel Plan Tree

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

$$T(u) = \frac{1}{\sum_i F(b_i)} + \sum_i \frac{F(b_i)}{\sum_i F(b_i)}(R_{bi}(u,si) + T(si)) \tag{4.1}$$

In this equation, the first term $\frac{1}{\sum_i F(b_i)}$ represents the expected waiting time at stop u. The second term which is $\sum_i \frac{F(b_i)}{\sum_i F(b_i)}(R_{bi}(u,si) + T(si))$ represents expected travel time to reach the destination from stop u. This second term gives the time after getting into some bus. For example: After getting into bus b_i, the time taken is $(R_{bi}(u,si) + T(si))$, which is the sum of travel time from u to si with bus b_i and $T(si)$ which is the time taken to reach destination starting at stop si. The probability of getting into bus b_i is proportional to its frequency.

Maximum number of transfers given by the 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.

$$T_b(u,h) = min(T_b(u,h-1), min_i(R_b(u,si) + T(si,h-1))) \tag{4.2}$$

The optimal plan with height h starting with bus b is either same as the optimal height h-1 bus restricted plan or it involves picking the best stop to get down at (si) and finding expected time to reach destination from si with height h-1.

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

$$T(s,h) = min_{B \subseteq B_s}\left(\frac{1}{\sum_{b \in B} F(b)} + \sum_{b \in B} \frac{F(b)}{\sum_{b \in B} F(b)} T_b(s,h)\right) \tag{4.3}$$

In above equation, the first term is the waiting time for buses and the second term is the time spent in the execution of bus restricted plans.

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.

4.2 Understanding MN code

We look at the important classes of Mumbai Navigator's Java code. See figure 4.2 for the Class diagram.

Database class takes a *"busdata"* file as input. The format of *"busdata"* file is as follows:

route: *route_number* "both/single" *bus_frequency*
stop_number1 *stop_number2* *time_taken* *distance* T/F
stop_number2 *stop_number3* *time_taken* *distance* T/F
stop_number3 *stop_number4* *time_taken* *distance* T/F
.
.
.
route: *route_number* "both/single" *bus_frequency*
stop_number1 *stop_number2* *time_taken* *distance* T/F
stop_number2 *stop_number3* *time_taken* *distance* T/F
.

Here "both" means that buses on that route runs in both direction. "T/F" specifies whether the first stop of the row is a stage-stop or not. Stop numbers are the consecutive stops in route and *time_taken* specifies the time taken by buses to go from first stop of the row to second stop of the row and distance means the distance between two stops.

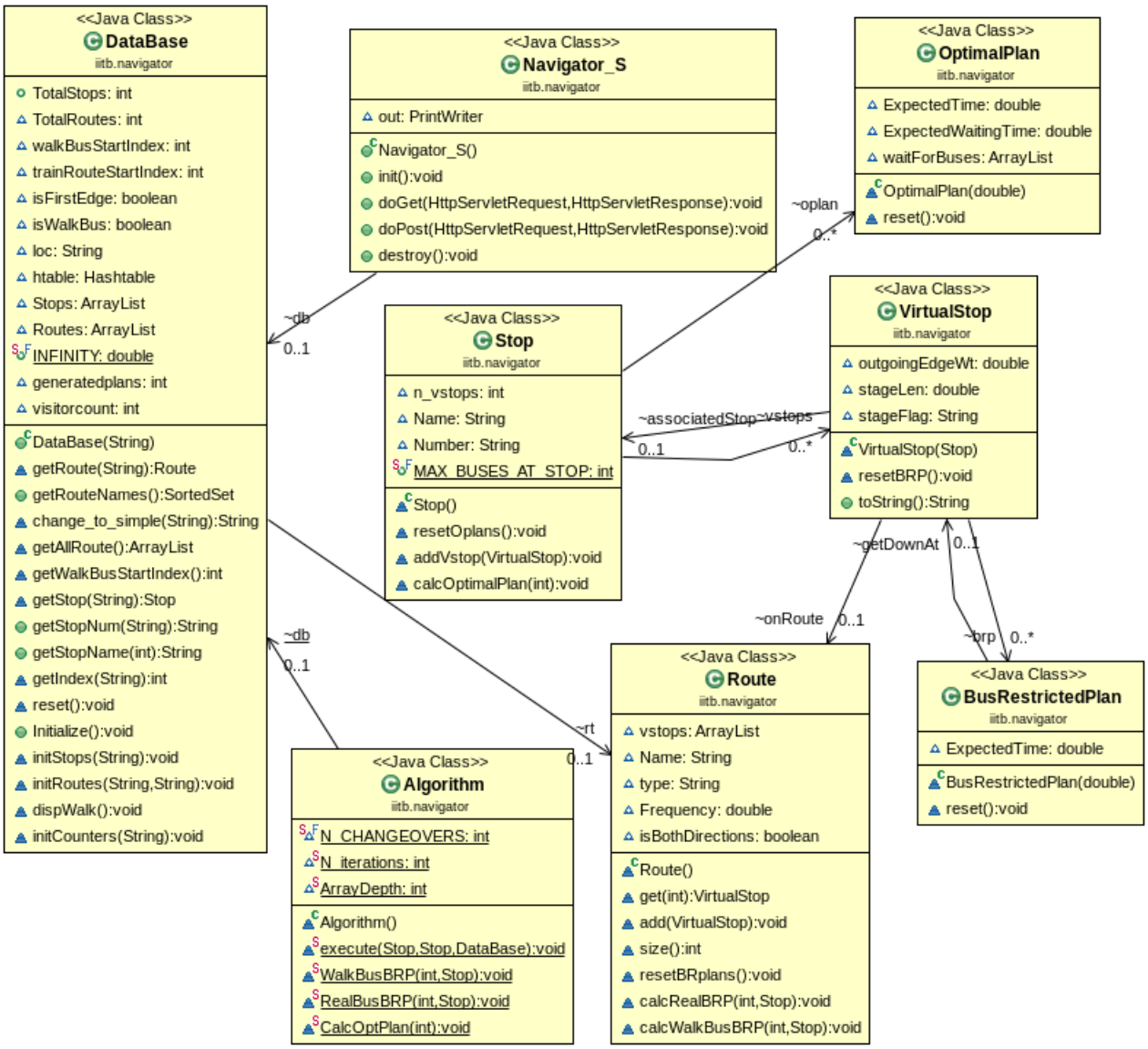


Figure 4.2: Class diagram of MN

"Stop" class represents a bus stop which contains bus stop name and number. It also contains object of class "VirtualStop" which represents buses that pass through that stop.

"Route" class represents a bus route which contains route name, route number, route frequency and direction. It contains list of "VirtualStops" which represents the stops present in the route.

A "VirtualStop" is a stop associated with a bus (attribute: *onRoute*). It also contains *outgoingEdgeWt* which represents the time taken to go from this particular stop to the next stop on the route. Next stop on the route is obtained by the object *onRoute* of class "Route".

"Database" class uses the *busdata* file to initalize "Route", "Stop" and "VirtualStop" information. It also initializes a hash table that stores mapping between bus stop number and a "Stop".

"Navigator_S" class gets a HTTP Post request containing origin and a destination from the user. It creates a "Database" object *db* to obtain the initial information. It uses *getRoute()* method of "Database" to obtain the "Stop" object associated with origin and destination parameters. It then calls *execute()* method of "Algorithm" which executes the MN algorithm to obtain the optimal plans for given origin-destination pair.

As mentioned in section 4.1, "Algorithm" executes the algorithm and generates optimal plans with zero, one and two changeovers between origin and destination. Changeovers represents maximum height of the plan tree. *"BusRestrictedPlan"* is associated with "VirtualStop" for each height which gives the *ExpectedTime* and the information about which "VirtualStop" to *getDownAt*.

Optimal Plan is associated with "Stop"s and is stored in form of a tree. Plan starts at origin Stop and contains *ExpectedTime* and a List of *waitForBuses* which is a list of "VirtualStop"s which specifies which route to take and where to get down. User takes the first bus that arrives at origin and gets down at specified stop mentioned in the plan. This new stop becomes the new origin and plan is obtained recursively until we reach the destination.

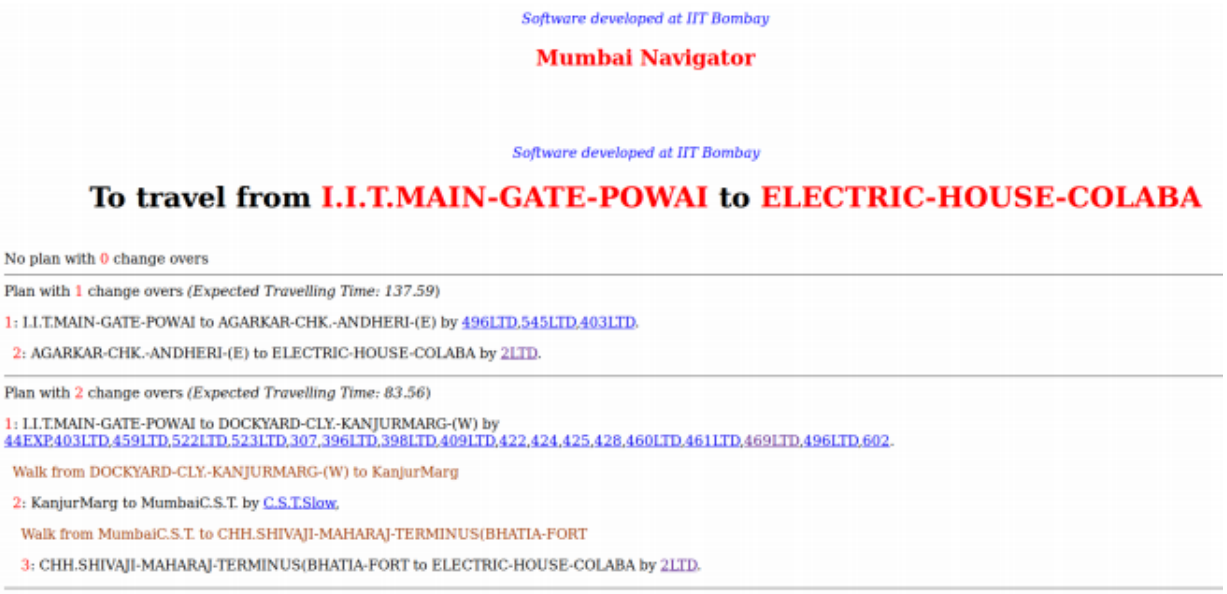


Figure 4.3: Generated plan for given origin and destination

Figure 4.3 shows the plans generated by Mumbai Navigator for origin "IIT Main Gate" and destination "Electric House Colaba".

Chapter 5

Modifying Mumbai Navigator

Mumbai Navigator takes one "busdata" file as input. We have to use trip data set and convert it into the format of "busdata" for trip planning. Trip data contains arrival times of past trips at stops which can be used to divide those trips into different intervals of the day like morning, afternoon and evening. This division leads to more accurate frequencies of routes and stop to stop travel times for each interval of the day. This can lead to better travel time estimation depending on the interval in which the departure time of the user falls. These 3 intervals will require 3 different "busdata" files containing frequency and travel time of buses in that particular interval. Since MN only deals with one "busdata" file, we need to modify MN code to deal with multiple data files (one for each interval of the day). We added a simulator code to Mumbai Navigator which gives the actual time taken when the generated plan is followed on a travel day. Actual time taken by the generated plan gives an idea of how close the time predicted by the planner is to the time taken when a plan is actually followed. To compare the predicted time with the actual time taken in following a plan, a random day for travel is picked from the past days and the predicted and actual time are compared.

This chapter looks at the changes made in the Mumbai Navigator code to generate plans and estimate the travel time for different intervals of the day like morning, afternoon and evening. MN takes "busdata" file as input, therefore, the generated trip data is converted into the format of "busdata" by finding frequencies and stop to stop travel time of buses in different intervals in section 5.1. Comparison is made in section 5.2 between plans generated using different intervals database but same source and destination. To calculate the actual time taken by a plan on a day, Simulator class is added to the MN code in section 5.3 which runs the generated plan on a random day. Lastly, in section 5.4, we look at the incorrectness and errors in

BEST tickets data, which led to the need for Synthetic data generation.

Problem faced in modifying MN code:
Mumbai Navigator is an Eclipse project. Any change made in Java classes was not being reflected. After looking into the matter it was found that the Launch Configuration used to run application was referencing a JAR instead of the project directly. Eclipse doesn't generally build JARs, so if the runtime classpath of a Launch Configuration is pointed at a JAR, it won't see coding changes unless you rebuild the JAR it's pointed to. The solution is to delete and re-create the Launch configuration.

5.1 Frequency of buses in route

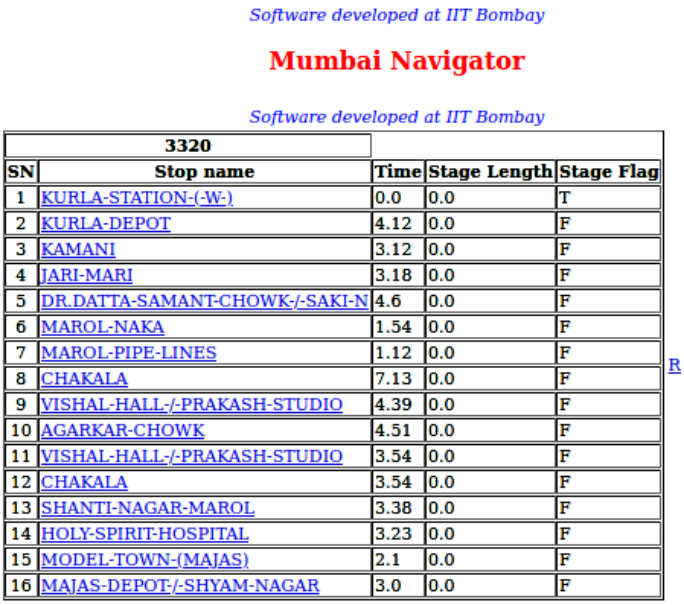
In section 4.2 it was mentioned that "Database" class uses *"busdata"* file as input which contains frequency of buses in routes, their direction and their stop to stop travel times.
We used Algorithm 4 to generate travel time between stops. The frequency of buses in routes can be calculated using the Trip data set 3.1. We create four sets of *"busdata"* referred to as *"day_ busdata"*, *"morning_ busdata"*, *"afternoon_ busdata"* and *"evening_ busdata"*. The *"morning_ busdata"* contains frequency of routes in morning hours and average time taken by buses in routes to travel between stops in morning hours. Similarly *"afternoon_ busdata"* and *"evening_ busdata"* contains afternoon and evening information of routes. We use the arrival times of buses at stops to categorize data into morning, afternoon or evening. The *"day_ busdata"* is just the average *"busdata"* containing average frequency of routes and average travel times between stops throughout the day. We have considered morning hours from 06:00 to 11:59, afternoon hours from 12:00 to 16:00 and evening hours from 16:01 to 23:59.

Algorithm 7 Frequency of route

Input: route number

1: Fetch all trips on given route number
2: Sort trips based on trip date and trip start time
3: $avg\_freq = INFINITY$
4: **for each** day **do**
5: Find first and last trip of the day
6: Count number of trips in the day
7: $freq[day] = (last\_trip\_time - first\_trip\_time)/count - 1$
8: $avg\_freq += freq[day]$
9: **end for**
10: $avg\_freq /= 31$
11: return avg_freq

The algorithm 7 can be used similarly for morning, afternoon and evening trips. Once we have these 4 databases, we can use it for comparison of the travel plans generated using different databases. Figure 5.1 shows route information of a bus in Mumbai Navigator.



Software developed at IIT Bombay

Mumbai Navigator

Software developed at IIT Bombay

3320				
SN	**Stop name**	**Time**	**Stage Length**	**Stage Flag**
1	KURLA-STATION-(-W-)	0.0	0.0	T
2	KURLA-DEPOT	4.12	0.0	F
3	KAMANI	3.12	0.0	F
4	JARI-MARI	3.18	0.0	F
5	DR.DATTA-SAMANT-CHOWK-/-SAKI-N	4.6	0.0	F
6	MAROL-NAKA	1.54	0.0	F
7	MAROL-PIPE-LINES	1.12	0.0	F
8	CHAKALA	7.13	0.0	F
9	VISHAL-HALL-/-PRAKASH-STUDIO	4.39	0.0	F
10	AGARKAR-CHOWK	4.51	0.0	F
11	VISHAL-HALL-/-PRAKASH-STUDIO	3.54	0.0	F
12	CHAKALA	3.54	0.0	F
13	SHANTI-NAGAR-MAROL	3.38	0.0	F
14	HOLY-SPIRIT-HOSPITAL	3.23	0.0	F
15	MODEL-TOWN-(MAJAS)	2.1	0.0	F
16	MAJAS-DEPOT-/-SHYAM-NAGAR	3.0	0.0	F

R

Figure 5.1: Route description of selected bus

Figures from 5.2 till 5.5 show the travel plans generated by Mumbai Navigator for origin "IIT Market" and destination "Marol Pipelines" using 4 different databases namely *"day_ busdata"*, *"morning_ busdata"*, *"afternoon_ busdata"* and *"evening_ busdata"*. The expected time of the corresponding plans is also shown. Notice the difference in expected travel times and travel plans when different databases are used.

We can see that in MN plans with 0 changeovers, expected time to travel from source (I.I.T Market) to destination (Marol Pipelines) over a day is 14.11 minutes. For morning the expected time is 21 mins, in afternoon the expected time is 28 mins and in evening the expected time is 20 mins. The difference in the specific interval plan time and whole day plan time shows that the departure time of user matters.

Mumbai Navigator

Software developed at IIT Bombay

To travel from I.I.T.MARKET to MAROL-PIPE-LINES

Using day data
Plan with 0 change overs *(Expected Travelling Time: 14.11)*

1: I.I.T.MARKET to MAROL-PIPE-LINES by 4220,3961.

Using morning data
Plan with 0 change overs *(Expected Travelling Time: 21.96)*

1: I.I.T.MARKET to MAROL-PIPE-LINES by 4220,3961,1850.

Using afternoon data
Plan with 0 change overs *(Expected Travelling Time: 28.79)*

1: I.I.T.MARKET to MAROL-PIPE-LINES by 1850,4220.

Using evening data
Plan with 0 change overs *(Expected Travelling Time: 20.98)*

1: I.I.T.MARKET to MAROL-PIPE-LINES by 4220.

Using day data
Plan with 1 change overs *(Expected Travelling Time: 12.37)*

1: I.I.T.MARKET to DR.DATTA-SAMANT-CHOWK-/-SAKI-N by 3981,4091.

2: DR.DATTA-SAMANT-CHOWK-/-SAKI-N to MAROL-PIPE-LINES by 3400,2261,4220,3320,4881.

1: I.I.T.MARKET to MAROL-DEPOT by 5451.

2: MAROL-DEPOT to MAROL-PIPE-LINES by 3490,2261.

1: I.I.T.MARKET to MAROL-PIPE-LINES by 4220.

Using morning data
Plan with 1 change overs *(Expected Travelling Time: 17.86)*

1: I.I.T.MARKET to DR.DATTA-SAMANT-CHOWK-/-SAKI-N by 3981,4781.

2: DR.DATTA-SAMANT-CHOWK-/-SAKI-N to MAROL-PIPE-LINES by 3400,2261,4881,4701,3320,3961.

Figure 5.2: Generated plan for given origin and destination using different databases (1)

Using afternoon data
Plan with 1 change overs *(Expected Travelling Time: 26.23)*

1: I.I.T.MARKET to CHAKALA by 5451.

2: CHAKALA to MAROL-PIPE-LINES by 3400,3490,4100,3360,4220,3591.

1: I.I.T.MARKET to DR.DATTA-SAMANT-CHOWK-/-SAKI-N by 3981.

2: DR.DATTA-SAMANT-CHOWK-/-SAKI-N to MAROL-PIPE-LINES by 3400,2261,1850,4220,3961,4881.

1: I.I.T.MARKET to MAROL-DEPOT by 5221.

2: MAROL-DEPOT to MAROL-PIPE-LINES by 3490,2261.

Using evening data
Plan with 1 change overs *(Expected Travelling Time: 20.50)*

1: I.I.T.MARKET to DR.DATTA-SAMANT-CHOWK-/-SAKI-N by 3981.

2: DR.DATTA-SAMANT-CHOWK-/-SAKI-N to MAROL-PIPE-LINES by 3400,4220,2261,3320,1850,4881.

1: I.I.T.MARKET to MAROL-PIPE-LINES by 4220.

Using day data
Plan with 2 change overs *(Expected Travelling Time: 12.18)*

1: I.I.T.MARKET to DR.AMBEDKAR-UDYAN-(POWAI) by 4611,4891.

2: DR.AMBEDKAR-UDYAN-(POWAI) to DR.DATTA-SAMANT-CHOWK-/-SAKI-N by 3200,3981,4781.

3: DR.DATTA-SAMANT-CHOWK-/-SAKI-N to MAROL-PIPE-LINES by 3400,2261,4220,3320,4881.

2: DR.AMBEDKAR-UDYAN-(POWAI) to MAROL-DEPOT by 5451.

3: MAROL-DEPOT to MAROL-PIPE-LINES by 3490,2261.

2: DR.AMBEDKAR-UDYAN-(POWAI) to MAROL-PIPE-LINES by 4220.

1: I.I.T.MARKET to DR.DATTA-SAMANT-CHOWK-/-SAKI-N by 3981.

2: DR.DATTA-SAMANT-CHOWK-/-SAKI-N to MAROL-PIPE-LINES by 3400,2261,4220,3320,4881.

1: I.I.T.MARKET to MAROL-DEPOT by 5451.

2: MAROL-DEPOT to MAROL-PIPE-LINES by 3490,2261.

1: I.I.T.MARKET to MAROL-PIPE-LINES by 4220.

Figure 5.3: Generated plan for given origin and destination using different databases (2)

Figure 5.4: Generated plan for given origin and destination using different databases (3)

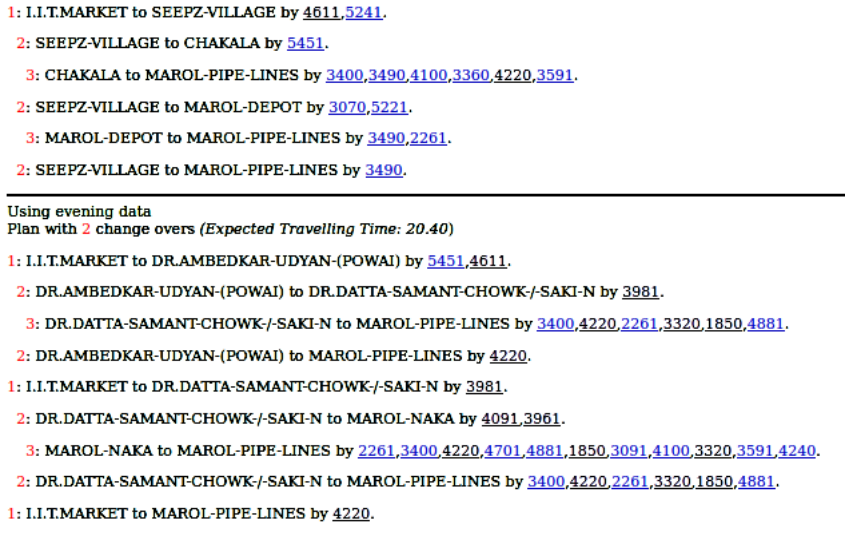


Figure 5.5: Generated plan for given origin and destination using different databases (4)

5.2 Travel plans comparison

We might wonder whether to use the whole day data to generate optimal plans or to use the most appropriate data depending on the journey time to generate travel plans.

Therefore, a comparison between expected travel time of plans generated using whole day database and time-based database is necessary to answer the question "Is time of the day really important?". This thought required Mumbai Navigator to work with multiple databases instead of one, therefore few changes are made in the MN code.

For example, four different "Database" class objects are created each using four different data files namely *"day_busdata"*, *"morning_busdata"*, *"afternoon_busdata"* and *"evening_busdata"*. MN algorithm is executed for all 4 databases and expected times of the plans are compared.

A python code sends multiple HTTP requests with random origin-destination pairs to Mumbai Navigator. Mumbai Navigator generates travel plans using our four databases and sends Expected Time of travel plans as response.

```
import requests
import pandas as pd

data = pd.read_csv("stop_list.csv",names=['stop',' '])
data.columns = ["stop",""]
rawlist = list(data.stop)

url =' http://localhost:8080/MumbaiNavigator/Navigator_S'
for i in rawlist:
for j in rawlist:
print(i+" "+j)
myobj = 'Origin': i, 'Destination': j
x = requests.post(url, data=myobj)
```

We randomly generate origin-destination pairs and send request to Mumbai Navigator code. For comparison purpose, 208681 unique origin-destination pairs are generated.

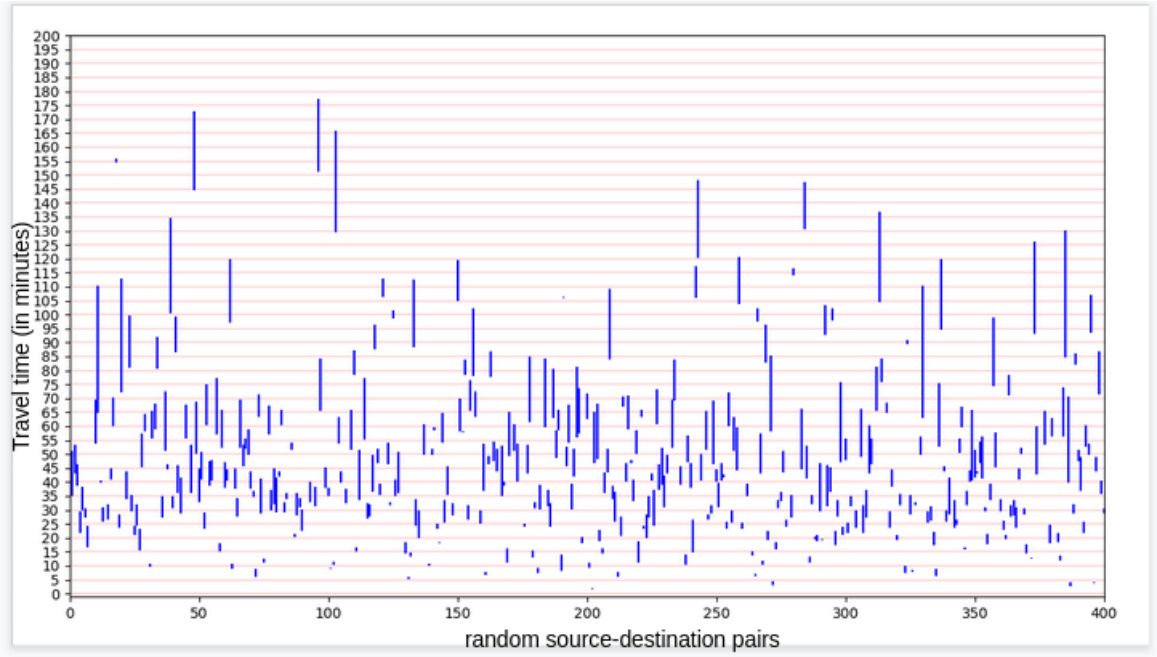


Figure 5.6: Day and morning travel plans comparison

Figure 5.6 shows the difference between expected time of plan with 2 change overs generated using day bus data and morning bus data. In figure 5.6, sample of 400 origin-destination pairs is selected for display. In figure, x axis represents different stage-stop pairs and y axis represents the expected time (in mins) of plans. The vertical blue lines in the figure shows the difference between expected time of 2-changeover plans generated using whole day data and morning data.

Note: The plans generated for same origin-destination pairs may be different based on the database used as it might be possible that for some route, buses take more time in certain interval of the day and run less frequently.

In figure 5.6, we see that the largest difference in expected travel times is of 45 mins. In this case the origin is "DIAMOND-INDUSTRIES" and destination is "GURUNANAK-VIDYALAYA". Let us look at the plans generated in this case.

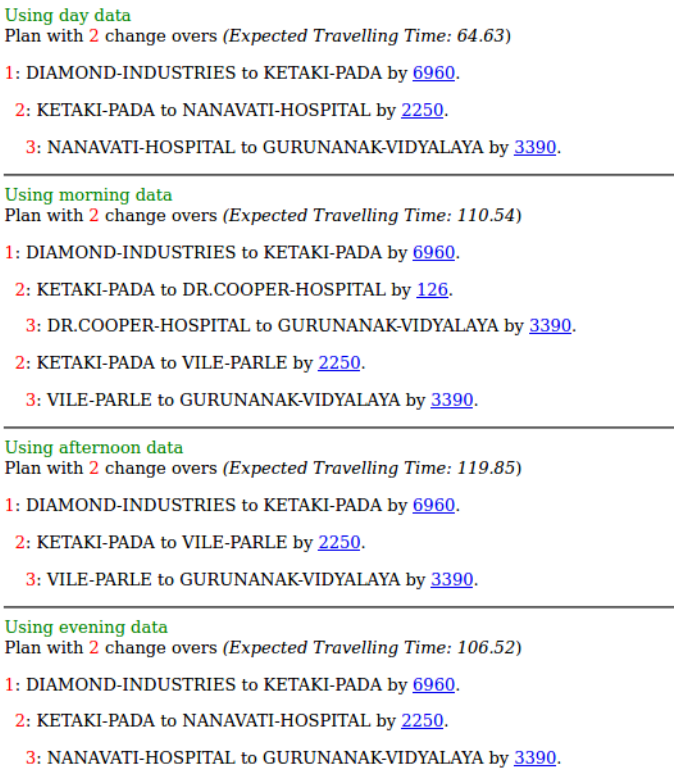


Figure 5.7: Travel plans from DIAMOND-INDUSTRIES to GURUNANAK-VIDYALAYA

In figure 5.7, although the plans are almost same, but the average frequencies and travel time between stops are varying for different time of the day, which changes the expected travel times for different departure time. See table 5.1 for frequencies.

route	frequency in day_ busdata (mins)	frequency in morning_ busdata	frequency in afternoon_ busdata	frequency in evening_ busdata
6960	47	37	35	38
2250	88	94	69	62
3390	19	16	21	22
126	41	31	43	48

Table 5.1: Routes and its frequencies

5.3 Simulation of plans

We need to find out the actual time taken when a plan generated between origin and destination is followed. For this, we pick a random day and do simulation of plans for that day. Time difference between actual time taken and the expected time given by the algorithm should be low, to prove that the generated optimal plans can be followed in real life scenario.

For simulation purpose, we create a new class which takes day, current time, source and destination as parameters and returns the time taken by following the generated plan on that day. The constructor function initializes parameters and opens the Trip data-set file for that day. Function *Time_taken* follows the optimal plan tree generated by Mumbai Navigator algorithm for a source-destination pair. The code assumes that Trip data set is sorted on arrival time.

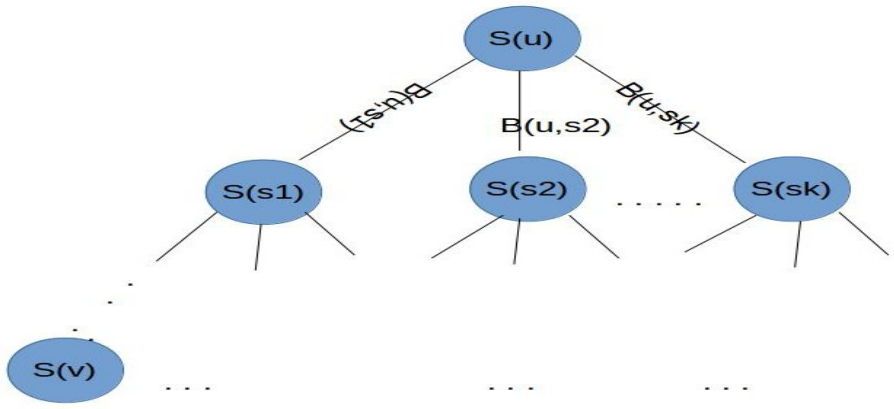


Figure 5.8: Travel Plan Tree

In figure 5.8, root node S(u) represents the source vertex and leaves represent the destination stop. The generated MN 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. While calculating the actual time taken, the waiting time at a bus stop for the first bus to arrive is calculated by the difference of current time and the arrival time of first bus. Initially the current time is same as the departure time. Similarly, travel time taken by a bus from source stop to the stop where user is supposed to get down at is calculated by the difference of arrival times of bus in trip data set at those two stops. Upon getting down at a stop, this stop becomes

the new source stop and the same process is followed until destination is reached. The new class diagram of MN is shown in figure 5.9. It contains new attributes and new class (Simulator class) added in the code and the dependencies between them. The code for Simulator class is shown below from 5.10 to 5.12.

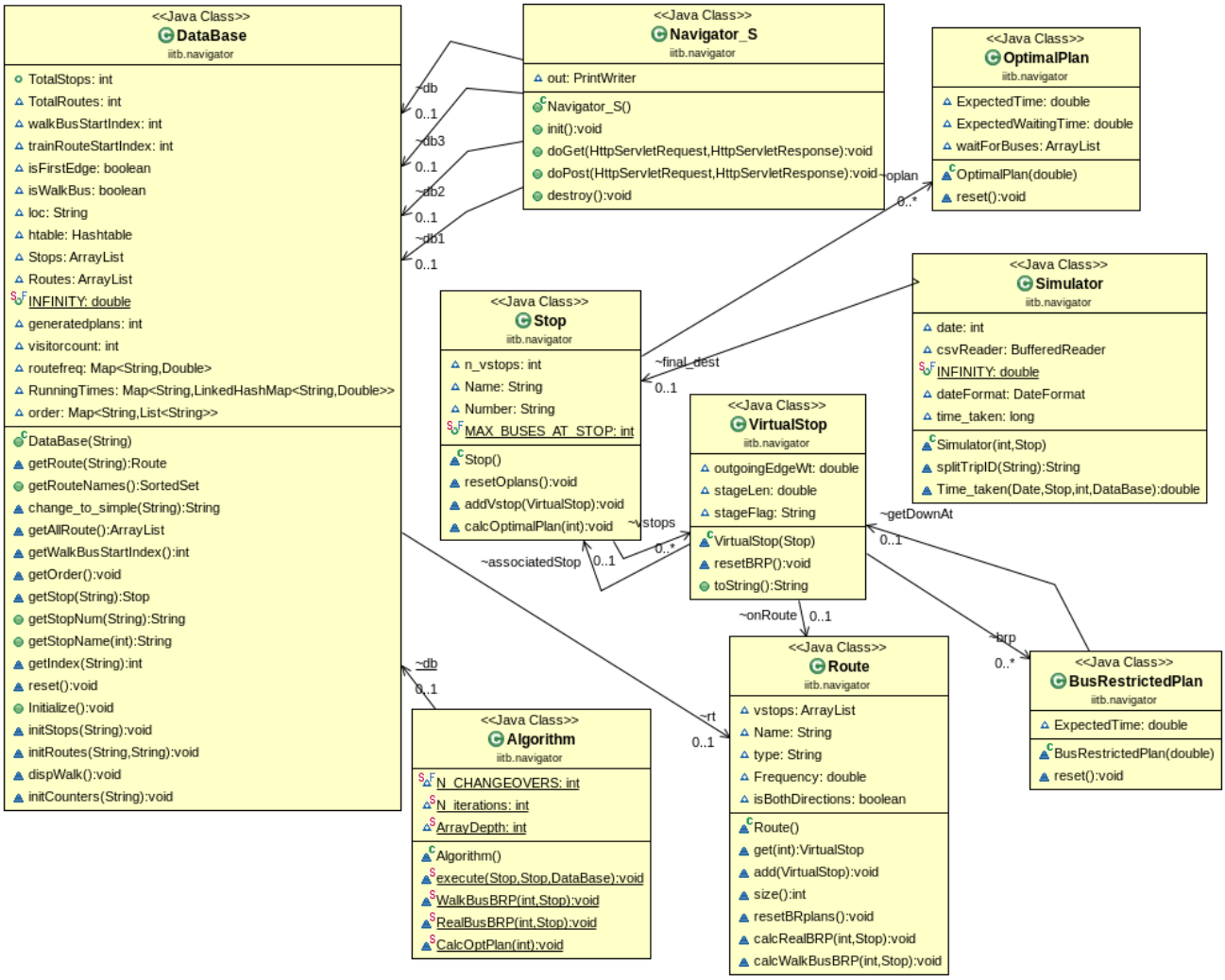


Figure 5.9: New Class diagram of MN

```
1  package iitb.navigator;
2  import java.text.DateFormat;
25
26 public class Simulator  {
27
28     Simulator(int date,Stop dest) throws IOException{
29         //Open trip database file corresponding to date.
30         this.date=date;
31         this.final_dest=dest;
32         this.csvReader = new BufferedReader(new FileReader("/home/user/Desktop/MTP-ws/"
33                 + "MumbaiNavigator/WebContent/database/tripdata"+date+".csv"));
34         this.dateFormat = new SimpleDateFormat("HH:mm:ss");
35         this.time_taken = 0;
36     }
37
38     String splitTripID(String ID) {
39         String[] row = ID.split("_");
40         return row[0];
41     }
42
43     double Time_taken(Date current_time,Stop sourcestop,int j, DataBase db) throws
44     IOException, ParseException {
45         /*Traverse file, From current_time, find bus from list that starts earliest at source
46          * Travel database after current time
47          * For each row of database , store stop,arrival,tripID.
48          * Split tripID to get bus no.
49          */
50
51         VirtualStop[] wfb;
52         int n_vstops = sourcestop.oplan[j].waitForBuses.size();
53         wfb = new VirtualStop[n_vstops];
54
55         String line;
56         HashMap<String, Integer> waitingBusIndex = new HashMap<String, Integer>(); ;
57         for(int i=0; i<n_vstops; i++) {
58             wfb[i] = (VirtualStop)sourcestop.oplan[j].waitForBuses.get(i);
59             waitingBusIndex.put(wfb[i].onRoute.Name,i);
60         }
61         while((line = csvReader.readLine()) != null) {
62             String[] row = line.split(",");
63             String stop = row[2];
64             String tripID = row[0];
65             String bus = splitTripID(tripID);
66             Date arrival_time = dateFormat.parse(row[3]);
67             Date reach_time = null;
68             String direction = row[1];
69
70             if(sourcestop.Number.equals(stop) && waitingBusIndex.containsKey(bus) &&
71                     arrival_time.getTime()>=current_time.getTime()) {
72                 int index = waitingBusIndex.get(bus);
73
74                 if(direction.equals("0")) {
75                     if(db.order.get(bus).indexOf(sourcestop.Number)>db.order.get(bus).
76                             indexOf(wfb[index].brp[j].getDownAt.associatedStop.Number))
77                         continue;
78                 }
79                 else {
80                     if(db.order.get(bus).indexOf(sourcestop.Number)<db.order.get(bus).
81                             indexOf(wfb[index].brp[j].getDownAt.associatedStop.Number))
82                         continue;
83                 }
84                 //Take that bus
85                 time_taken+=arrival_time.getTime()-current_time.getTime();  //waiting time
86                 //Traverse database to find arrival time of bus B at dest(B). Let arrival time be reach_time
87                 String line2;
88                 while((line2 = csvReader.readLine()) != null) {
89                     String[] row2 = line2.split(",");
90                     reach_time = dateFormat.parse(row2[3]);
91
92                     if(wfb[index].brp[j].getDownAt.associatedStop.Number.equals(row2[2])
93                             && row2[0].equals(tripID)) {
94                         break;
```

Figure 5.10: Simulator code (1)

Figure 5.11: Simulator code (2)

```
95                      }
96                  }
97                  if(line2==null) {
98                      return INFINITY;
99
100                 }
101                 time_taken+=reach_time.getTime()-arrival_time.getTime();
102                 current_time=reach_time;
103                 sourcestop=wfb[index].brp[j].getDownAt.associatedStop;
104                 break;
105
106             }
107         }
108         if(line==null) {
109             return INFINITY;
110
111         }
112
113         if(sourcestop==final_dest) {
114             return time_taken/60000;
115         }
116         else {
117             //Traverse plan Tree. Find new buses and destinations after taking B
118             return Time_taken(current_time, sourcestop,j-1,db);
119         }
120     }
121
122     int date;
123     Stop final_dest;
124     //File descriptor
125     BufferedReader csvReader;
126     public static final double INFINITY = 100000.0;
127     DateFormat dateFormat;
128     long time_taken;
129
130 }
131
```

Figure 5.12: Simulator code (3)

5.4 Need for Synthetic Data generation

The BEST ticket data is incomplete and noisy. Due to following inconsistencies in BEST data, we found the need to make a model and generate cleaner ticketing like data:

- trip number field in BEST data is not unique for given route, date and vehicle number
- For some routes there are very few tickets bought over a month. This was especially due to wrong route number associated with tickets.
- Error in trip direction field of tickets ('U' written instead of 'D') which lead to trips running in other direction also although the route was only single directional.
- For some routes there were no trips made in morning hours but were made in afternoon and evening hours.
- In 11% of the trips, either incorrect time was written with the ticket or ticket was issued much later, which led to the arrival time at later

stops to be before the arrival time at initial stops.

Example 1: The difference between expected time of plans with source "DHARAVI-POLICE-STATION" and destination "TARUN-BHARAT-SOCIETY" using day bus data and afternoon bus data was found to be 254 mins. See figure 5.13.

After checking, it was found that route 4354 had no buses running in the afternoon (from noon to 4 p.m.), therefore the frequency of that route is infinity (100000), leading to large expected travel time in the afternoon.

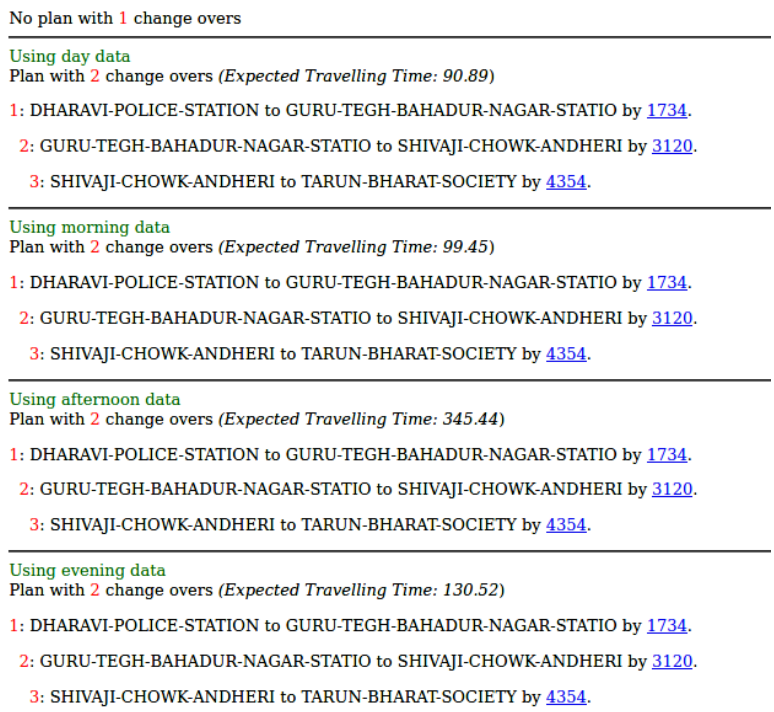


Figure 5.13: Travel plans from DHARAVI-POLICE-STATION to TARUN-BHARAT-SOCIETY

Example 2: We compare the expected time given by MN and actual time taken by following that plan. Figure 5.14 shows the difference in expected travel times given by MN and the actual time taken to go from source to destination on a day at a given time. In the figure, we have shown random 200 source-destination pairs with departure time as 09:00 a.m and day as July

25. In figure, x axis represents different stage-stops pairs and y axis represents the travel times (in mins). The vertical blue lines shows the difference between expected time of 2-changeover plan generated using whole day data by MN and the actual time taken by following the same plan to travel on that day using Trip data set. In figure, we see that there are long vertical lines crossing the graph in upward direction. This is because buses suggested by the algorithm are not running on that day which takes the actual time taken to Infinity. If we ignore such data, we see that the minimum difference in expected time and actual travel time is as little as 1 min from "ACHARYA-GARDEN/DIAMOND-GARDEN" to "DR.AMBEDKAR-GARDEN" and is as large as 867 mins from "A.H.ANSARI-CHOWK" to "DR.BHADKAMKAR-ROAD". When plans are generated using specific time based data, although the difference improves for some route, but overall there is not much change observed.

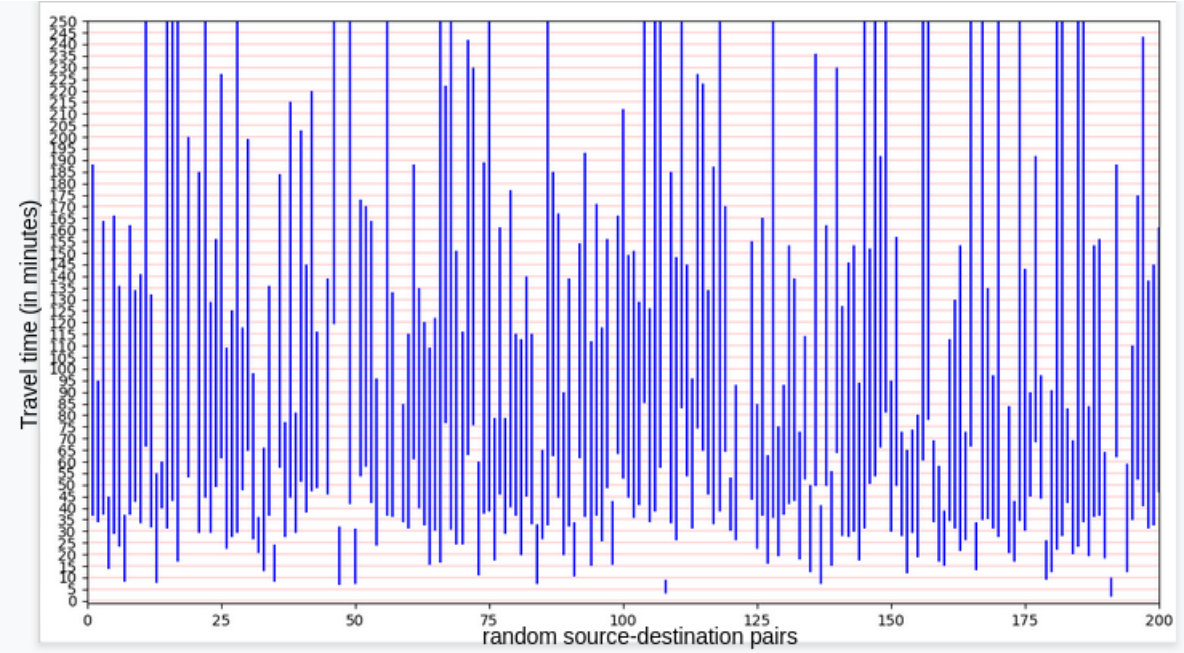


Figure 5.14: Comparison of expected and actual time taken by plans

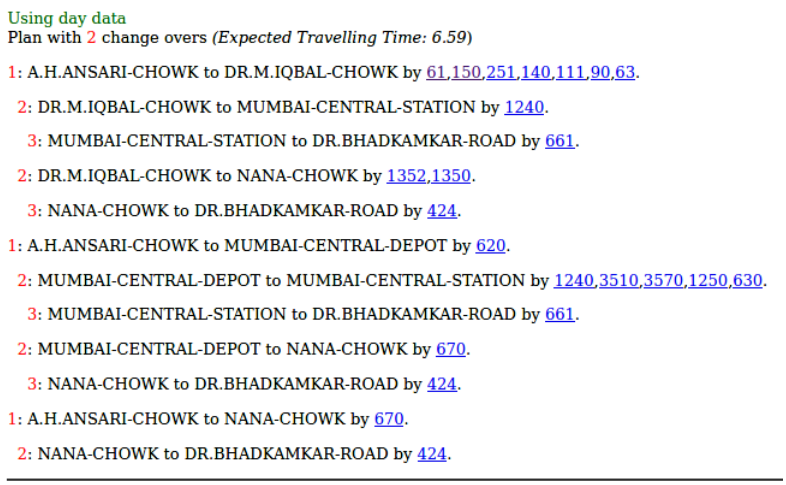


Figure 5.15: Travel plans from A.H.ANSARI-CHOWK to DR.BHADKAMKAR-ROAD

In figure 5.15, we see that plan suggests taking route 61,150,251,140,111,90,63, 620 or 670 whichever comes first at A.H. Ansari Chowk. According to Trip Data set at 09:00 a.m., the first bus to arrive was bus 150. Plan tells to get down at DR.M.IQBAL-CHOWK, catch bus 1240 and get down at MUMBAI-CENTRAL-STATION and then take bus 661 from there. On investigation, it was found that bus 661 ran only in the night on July 25th, which greatly affects the actual travel time.

Thus, we needed to generate our own data due to the incorrectness and incompleteness of BEST ticket data.

Chapter 6

Synthetic Data Generation

Synthetic data generation means constructing ticketing like data synthetically by taking a base frequency of routes and base speed of buses at different stops and introducing modifications into it at different times of the day. For example: we can decrease the speed of buses during peak hours by some amount. The need for synthetic data generation aroused due to inconsistencies in the BEST ticket data. These inconsistencies were brought into light while using the trip data generated from ticket data on Mumbai Navigator. We decided to create our own data that is clean and can be used with different trip planners to get proper results. To generate this synthetic data, the routes, base frequencies of routes and average time taken between stops information was taken from original Mumbai Navigator. The variations in speed of buses and frequencies was introduced to model the real behaviour of buses.

We start with some base parameters. For every route r there is a base frequency F_r. For every stage s, there is some base speed of buses SP_s running around it. The frequency at time t on route r is $F_r + f_{tr}$, where f_{tr} is uniformly distributed between 0 to 3 minutes. Similarly, speed of buses at a stage s at time t is $SP_s + sp_{ts}$, where sp_{ts} is uniformly distributed between (-0.25*SP_s) and 0 in peak hours and between (-0.01*SP_s) and (0.1*SP_s) in non peak hours. A schedule Generator program generates trips for each route using appropriate frequencies and speeds corresponding to the time of the day. Speed function is called to predict the arrival time of the trip at the next stage-stop. Frequency function is called every time we want to start a trip.

In this chapter, we discuss the synthetic data generation process which involves creating 3 files: Network Desc file, Speed Modifications File and Fre-

quency Modifications file. Network desc file stores the routes and stop to stop distances. Speed Modifications File and Frequency Modifications file stores the variations in speed of buses and its frequency at different times of the day. Schedule generator then takes all these files information and generates clean Trip data. The public transport network structure, base speed and base frequency of buses is obtained from the old input file of original Mumbai Navigator.

6.1 Data Generation process

We use the routes, distances, frequency and time taken by buses between stops information from original Mumbai Navigator to generate trips data. Original Mumbai Navigator takes input files called *"Busdata"* and *"stop_list"* to generate travel plans. *"stop_list"* contains stop numbers along with stop names of Mumbai. *"Busdata"* contains route information like route numbers, stops present in routes, frequency of routes, distance between stops and travel time taken by bus between consecutive stops of routes.

The *"Busdata"* file looks as follows:

route: 698 both 2.3333333333333335
1943 369 0.84 0.3 T
369 2180 0.84 0.0 F
2180 695 0.93 0.5 T
695 2169 0.93 0.0 F
2169 193 0.93 0.0 F
.
.
route: 700LTD both 14.666666666666666
1504 2163 1.53 0.0 F
2163 611 1.53 0.0 F
611 390 1.53 1.7 T
390 1978 2.45 0.9 T

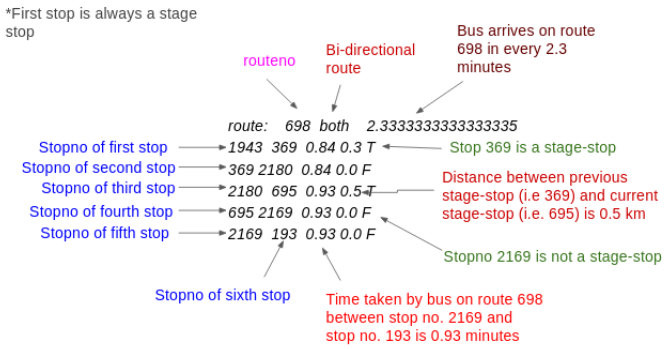


Figure 6.1: Fields in "Busdata" file

6.1.1 Generating Network Description File

A network description file is generated from the MN files which describes the public transport network. This file stores route, direction, frequency, stages and stage to stage distances. We have taken stages instead of stops because stop to stop distances are not available in original Mumbai Navigator. Also because ticketing data had only stages data in it, so to make it similar to the ticketing data we use stages instead of stops.

The Network description file looks as follows:

route: 1_U both 9.666666666666666
2220 43 0.9 —>(stage1, stage2, distance in km)
43 549 0.9
549 2615 1.0
2615 550 0.3
550 737 0.4
.
.
route: 1_D both 9.666666666666666
376 86 0.8
86 2702 0.8
2702 1565 1.4
1565 2483 0.8
.
.

Here we added direction to the route numbers for easy distinction. Route number 1_U is actually route number 1 in upward direction.

6.1.2 Creating Frequency Modifications File

Base frequency for each route is obtained from the Network Description File. **Note:** Frequency (in minutes) here means the time after which next bus in the route arrives. If frequency of route is 6 minutes it means that buses in this route arrives in every 6 minutes.

A day is divided into 18 one hour intervals like 6 a.m. to 7 a.m. ; 7 a.m. to 8 a.m.; ..... 23 to 00. Variation in frequencies of each route is induced in each interval randomly between 0 to 3 minutes.

The format of Frequency Modifications file is as follows:

route no.	base frequency	freq variation from 06:00 to 07:00	freq variation from 07:00 to 08:00	..	freq variation from 23:00 to 00:00
1_U	9.66	0.425968	1.08234	..	0.0101881
2LTD_D	10.0	0.615814	1.27887	..	2.36252
...	...	...	...	...	...
700LTD_U	14.66	1.17284	1.13905	..	0.37412
...	...	...	...	...	...

Table 6.1: Frequency Modifications file

The frequency of route 700LTD_U at 07:30 a.m. is sum of base frequency (i.e. 14.66) and freq variation from 07:00 to 08:00 (i.e. 1.13905) which is 15.7 minutes.

6.1.3 Creating Speed Modifications File

We saw that the MN file contains time taken by route buses between consecutive stops. Using the "T/F" field we can determine the stage stops of route and calculate time taken by route buses between consecutive stage-stops. Our network description file contains the distance between stage stops of the routes. Using distance and time, speed of route-buses at each stage stop of the route is calculated. We can find the average speed of buses at a stage stop by taking the average of speeds of all routes passing through that stage

stop. This average speed becomes the base speed of buses at a stage-stop.

A day is divided into 18 one hour intervals like 6 a.m. to 7 a.m. ; 7 a.m. to 8 a.m.; ..... 23 to 00. Variation in speed of buses at each stage stop in each interval is induced randomly. It is assumed that in peak hours the speed variation at stage stops will be some random number generated between (-0.25*base_speed, 0). In non peak hours the speed variation at stage stops will be some random number generated between (-0.01*base_speed, 0.1*base_speed). Peak hours are assumed to be from 08:00 to 14:00 and from 16:00 to 21:00.

The format of Speed Modifications file is as follows:

stage no.	base speed (kmph)	speed variation from 06:00 to 07:00	speed variation from 07:00 to 08:00	..	speed variation from 23:00 to 00:00
4	25.9585	2.21301	1.08466	..	-0.21035
10	22.5983	1.79897	1.68023	..	1.21722
...	...	...	...	...	...
2967	6.68112	0.594409	0.57806	..	0.331412
...	...	...	...	...	...

Table 6.2: Speed Modifications file

Thus, the speed of buses around stage-stop no. 10 at 07:30 a.m. is the sum of base speed (i.e. 22.5983) and speed variation from 07:00 to 08:00 (i.e. 1.68023) which is 24.28 kmph.

6.1.4 Schedule Generator

A schedule Generator program generates trips for each route using appropriate frequencies and speeds corresponding to the time of the day. The starting time of the first trip of each route is taken as 6 a.m.

Frequency and Speed are the functions which we evaluate whenever we need it by giving appropriate parameters. Frequency function takes route number and time of the day as parameters. Speed function takes stage no and time of the day as parameters.

Speed (stop,time interval) = base_speed+dw

frequency (route,time interval) = base_freq+dv
dw and dv are variations in speed and frequency respectively based on time of the day.

Speed function is called to predict the arrival time of the trip at the next stage-stop. For this, we use the speed of buses at current stage-stop for the given time interval and the speed of buses at the next stage-stop for the given time interval to find the average speed of buses around the next stage-stop in a particular interval. Using this average speed and the distance (between current and next stage-stop) information from the Network Description file we find the arrival time of trips at stage stops.

Frequency function is called every time we want to start a trip. To the last trip of the route, we add the frequency as returned by the function to get the start time of the new trip on a route.

Trip data generated by Schedule Generator:
1_U_1_1, 2220, 06:00:00
1_U_1_1, 43, 06:02:57
1_U_1_1, 549, 06:05:28
1_U_1_1, 2615, 06:07:51
1_U_1_1, 550, 06:08:29
1_U_1_1, 737, 06:09:28
.
.
28_U_1_23, 2650, 09:37:41
28_U_1_23, 508, 09:42:06
28_U_1_23, 2807, 09:45:34
.
.
Each line consists of trip ID, stage number and arrival time of trip at that stage number. Trip ID itself is a combination of route no, direction, date and trip number.

We have generated Trip data containing trips information of all the routes. This data is clean and complete and variations in speed and frequency is introduced in the data to induce bus delays or speedups at some intervals of the day.

Chapter 7

Planners Implemented

Multi-label correcting (MLC) algorithm (section 7.1) minimizes total time, variance and number of transfers for a given source, destination and departure time. It outputs multiple non-dominating plans, where each plan is a sequence of trips with drop and pickup points. This planner uses historical data to generate efficient plans.

Second planner is a variant of the MLC planner (section 7.2). It uses real time data as well as past data to generate non-dominating plans minimizing travel time, variance and number of transfers for a given day. Real time data is used to predict bus waiting time in the near term and past data is more useful in estimating bus travel time in the long term.

Best time estimator (section 7.3) gives the best possible plan between source and destination at a departure time on a given day. This estimator is implemented as a planner using RAPTOR algorithm.

We perform experiments (section 7.4) to compare the plans of planners with their actual time and best possible time. We also compare different planners based on the accuracy of the predicted time, i.e. how close is the predicted time to the actual time taken. We then compare different planners based on how often it suggests the best possible plans to users.

7.1 Multi-Label Correcting (MLC) algorithm

Given source stop, destination stop and departure time, we compute optimal travel plans for public transport networks based on criteria: arrival time, reliability and number of transfers. Number of transfers was introduced as

an criteria to limit the maximum number of transfers to 2.

The assumption that buses follow a static schedule does not hold true in a realistic transit network because bus arrival times depend on traffic conditions and are time dependent.

In this planner, we take the generated synthetic trip data and Network description file as database input. Network description file is useful to know the number of routes, number of stops in the network. Synthetic trip data is used as the past data to know the behaviour of buses on different routes at different times in the past to better estimate the travel time and generate reliable travel plans.

- ***Database input:*** Synthetic trip data, Network description file, interval duration
- ***User input:*** source, destination and departure time
- ***Planner output:*** Non dominated plans with up to 2 changeovers minimizing total time and variance.

Network description file is used to generate a time-dependent graph (figure 7.1).

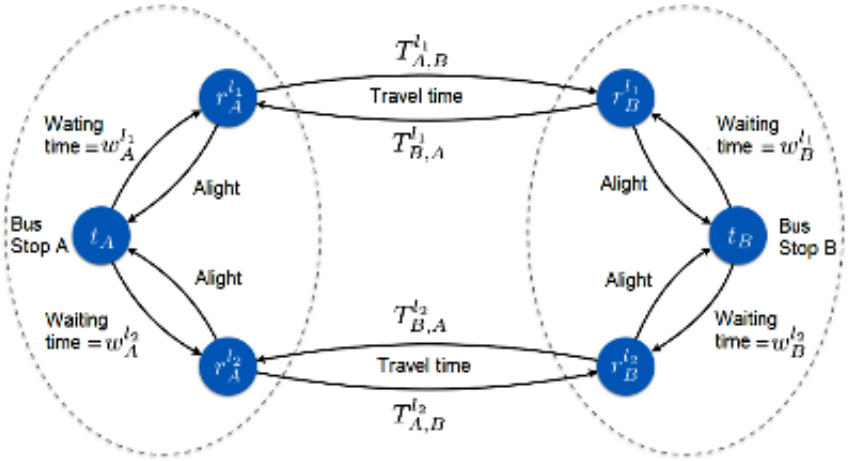


Figure 7.1: Time-dependent transportation graph [1]

We have two types of nodes: transfer nodes t_A .. t_B and route nodes r_A^{l1} .. r_A^{lk}. Multiple route nodes correspond to a transfer node if there are multiple routes containing that stop. Edge from transfer node t_A to route node r_A^{l1} represents the bus boarding process. The edge cost is the waiting time for bus $l1$ at stop A. Edge from route node r_A^{l1} to transfer node t_A represents alighting process and edge cost is negligible but increments the number of transfers by 1. Edge between route node r_A^{l1} to route node r_B^{l1} for the same bus $l1$ represents the travel process and the edge cost is the travel time taken between stop A to stop B by bus $l1$.

Using network description file, we create transfer nodes for each stop and route node for all stops of the route. We connect each transfer node to its route node in both direction to represent boarding and alighting process. Two route nodes are connected if they are the consecutive stops of the same route and the direction of edge is from previous stop of the route to the next stop of the route.

Interval duration describes the length of each interval of the day. If interval duration is taken to be 60 mins, then the day is divided into intervals of 60 mins. For example: 06:00 to 07:00 represents an interval. Similarly 07:00 to 08:00, 08:00 to 09:00 and so on.. The cost (time and variance) selected for an edge depends on the time interval in which we arrived at an edge.

The edge weights are calculated based on the departure time of the user and the type of edge (boarding, travelling or alighting). Past data gives average travel time and its variance between any two stops of a route during each time interval of the day. These are maintained as multi-dimensional costs. A k-dimension cost c of an edge in the graph can be defined as an array of k cost elements [c[0], c[1], ..., c[k 1]]. When combining two travel times, the aggregate mean can be calculated by the sum of individual means and also the sum of variances can be used as an approximation for the reliability of routes.

Past data is also used to obtain the frequency of buses during each time interval of the day to estimate the waiting time of buses at stops during a given time interval of the day. This data is used to update waiting time in the graph.

The output of the planner is a set of non-dominated travel plans. A travel plan is defined as a sequence of trips. Each trip contains route, pickup and

dropping points. Each travel plan has the arrival time at the destination stop and variance which tells by how much time can the arrival time at the destination vary and also the number of transfers required in that plan.

Example:
v_s = source stop
v_d = destination stop
t_ij = jth trip that is taken in ith travel plan
v_s , v_i1, v_i2, .. v_d are the transfer stops for plan i
Travel plans are of form:
Travel plan tp1 = { { ($t_1$1, v_s, v_d) } , $\tau_1(v_d)$, $var_1(v_d)$ }
Travel plan tp2 = { { ($t_2$1, v_s, $v_2$1) , ($t_2$2, $v_2$1, v_d) }, $\tau_2(v_d)$, $var_2(v_d)$ }
Travel plan tp3 = { { ($t_3$1, v_s, $v_3$1) , ($t_3$2, $v_3$1, $v_3$2), ($t_3$3, $v_3$2 , v_d) }, $\tau_3(v_d)$, $var_3(v_d)$ }

Thus, a travel plan tp = { { sequence of trips with pickup and drop points} , arrival time at destination, variance }. Number of transfers in the plan is one less than the number of trips in a plan.

Travel plans are non dominating i.e. no travel plan is better or worse than any other travel plan in all criteria.

Example: Let these two be the travel plans
Travel plan 1: { { ...} , 10:45 , 2 mins, transfers = 1 }
Travel plan 2: { { ... } , 10:40 , 5 mins , transfers = 1 }
A plan p: { { ... } , 10:43 , 10 mins, transfers = 1 } cannot be a travel plan because though it is better than travel plan 1 in arrival time but it is worse than travel plan 2 in both the criteria (arrival time and variance). This means that there are better plans available with earlier arrival times and more reliability.

7.1.1 Algorithm

The algorithm is similar to Dijkstra's shortest path algorithm. A node label l is defined as a pair (ni,ci) where ni is the node and ci is the multidimensional cost storing time, variance and number of transfers. This cost is the cost to reach node ni from the source node. There can be multiple labels associated with each node with multiple non dominating costs.

Algorithm 8 MLC [1]

```
Input: source ns, destination nd, departure time t

Initialize priorityQueue pq, predecessorMap pm, nodeCostsList cl to null
▷ Create a label for source node, with cost as 0. Cost to reach source from source is 0
label ls = createLabel (ns , cs = 0)
▷ Insert this label into priority queue
pq.insert(ls)
▷ Main loop, runs till priority queue is empty
while pq != null do
    ▷ pops label with minimum travel time to reach from source
    label lu = pq.pop()
    node nu = lu.getNode()                                  ▷ find node corresponding to this label
    for each outgoing edge e(u,v) do
        ▷ find if edge is waiting edge or travel edge
        ▷ get the edge cost using time interval of the day
        ▷ find cost for node nv by adding cost of label lu with edge cost
        cost cv = getAccurateCost (lu , e(u,v), t)
        ▷  get all the existing costs obtained till now for node nv
        List <cost> costs = cl.getCosts(nv)
        if cv is dominated by costs then
            drop cv
        else
            cl.put(nv,cv)
            cl.removeDominated()                            ▷ Remove all dominated costs from node
            ▷ Create new label and insert into priority queue
            Label lv = createLabel(nv,cv)
            pm.put(lv,lu)
            pq.insert(lv)
        end if
    end for
end while
```

A priority queue pq is maintained to determine which node and label to explore next. It is sorted based on the mean travel time of the multi-dimensional cost. A predecessor map pm keeps track of the predecessors of the labels. For every node, there is a node cost list cl. This list stores all non-dominated costs. Initially priority queue, predecessor map, node cost list is empty. A label with node as source node and cost as 0 is created and inserted into priority queue.

The main loop in the algorithm runs until the priority queue is empty and there are no more labels left to visit in the priority queue. In each iteration, the label with the lowest cost is retrieved and all of its outgoing edges are examined. Here, we find the correct cost (of the incoming node) depending on the edge type:

1. If it is a waiting edge, we use the frequency of the route in that interval from the past data to obtain the waiting time. The frequency of bus arrivals at a station can be defined as the number of buses that are expected to arrive during this period divided by the length of the time

interval. The expected waiting time at a bus stop i for a specific time interval j can be calculated as: wij = 0.5 / fij.

2. If it is a transfer edge, we use past data to find mean time and its variance taken by bus to go from one stop to another in that interval.

3. If it is an alight edge, we increment the number of transfers.

Once we get the cost, it is compared with existing costs of the same node. If it is better than any, a new label is created and inserted into the priority queue, and dominated costs are removed from the node. The algorithm stops when all labels are processed and the queue becomes empty.

This is a label correcting algorithm. After i iterations of the main loop, we have processed i labels (at most i nodes). After i iterations we have found a temporary set of non-dominating costs (cost from source to this node) for nodes. Set of non dominating costs found for the nodes after i iterations is not the final set because there may be other paths through which we can obtain some more costs that are non-dominating with the existing costs or costs that are better than existing costs. Only at the end of the algorithm, we find the permanent cost list for nodes. At the end of the algorithm for each vertex u we have nodecostsList which stores all the non-dominated costs and predecessor map to get all the stops and routes in the plans.

7.1.2 Output

Source stop number = 2431
Destination stop number = 432
Departure time of user = 13:00

Non-dominating plans are generated by MLC using past data for given source, destination and departure time. With each plan, it specifies the estimated travel time, variance and number of transfers. It then gives the description of plan, with buses to take and stops to get down at.

Actual time taken specified after each plan is the time taken by following that plan on a random day (from past). This time is used to find the accuracy of the predicted plan. To calculate the actual time taken, a travel day is selected from the past and the trip data corresponding to that travel day is provided to the code. This trip data contains the actual arrival times of buses at stops on the travel day. The planner suggested plan is followed on that day and actual time taken is printed with each plan.

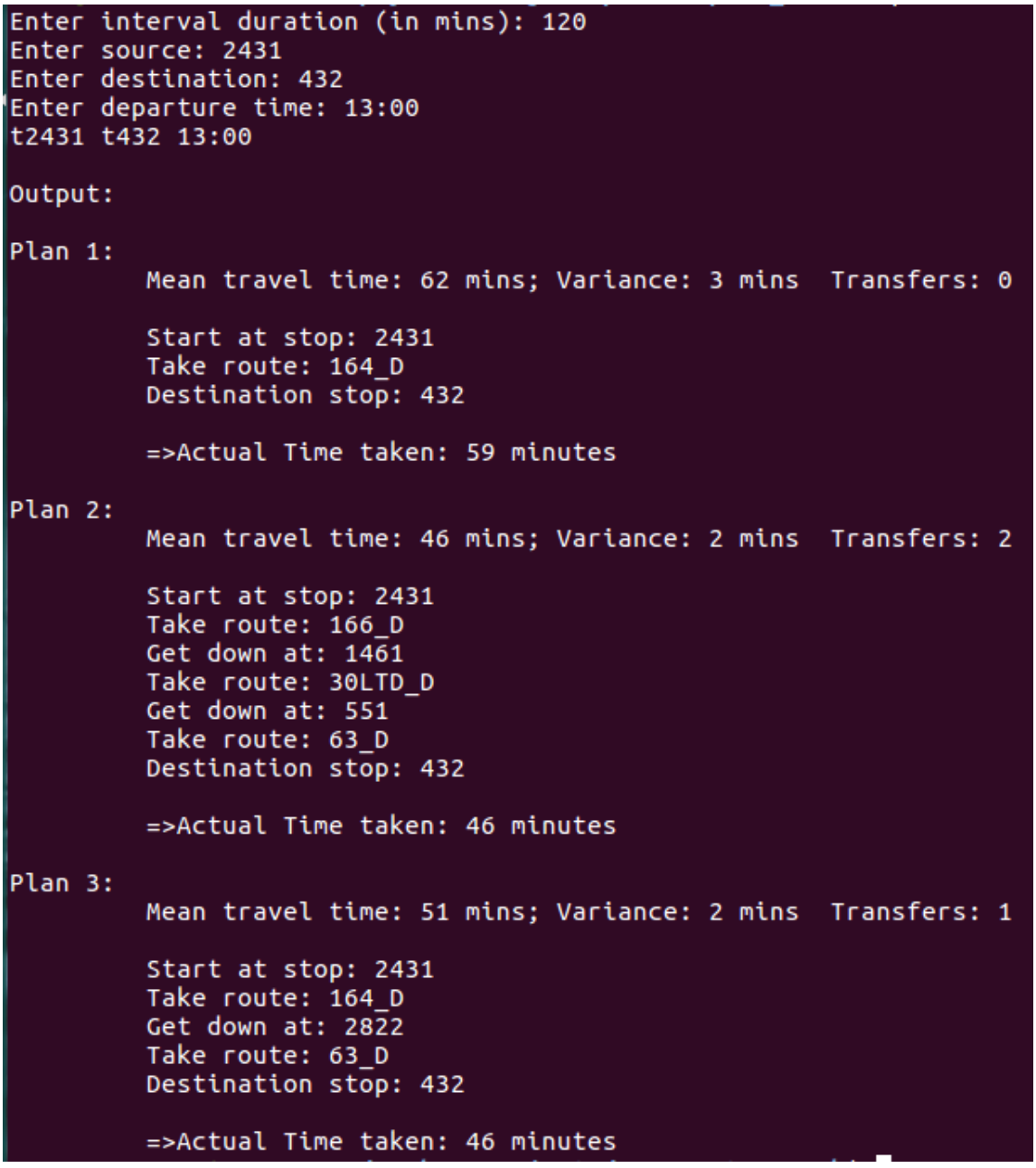


Figure 7.2: MLC plan using past data

7.2 MLC planner using real time data

This planner uses the same transportation graph (figure 7.1) and algorithm (Algorithm 8) as MLC planner with past data. The only difference is that it uses real time data at source stop to estimate the waiting time for buses at source. For travel time and waiting time at later stops, past data is used. The real time data gives the position of buses at the departure time, which lets you predict the arrival of buses at source more accurately. In particular, real-time transit data can provide quite accurate bus waiting time in the near-term, while historical data are more useful in estimating bus travel time in the long-term.

Since, we don't have GPS data, to find the real waiting time, we take the synthetic schedule of the travel day till the departure time. The algorithm gets to see the state of the buses at the beginning of the journey, and knows about the expected delay.

- ***Database input:*** Synthetic trip data (past data), Travel day data up til departure time (real time data), Network description file, interval duration
- ***User input:*** source, destination and departure time
- ***Planner output:*** Non dominated plans with up to 2 changeovers minimizing total time and variance.

7.2.1 Output

Source stop number = 2431
Destination stop number = 432
Departure time of user = 13:00

Non-dominating plans are generated by MLC using real time data and past data for given source, destination and departure time. With each plan, it specifies the estimated travel time, variance and number of transfers. It then gives the description of plan, with buses to take and stops to get down at.

Actual time taken specified after each plan is the time taken by following that plan on the travel day. This time is used to find the accuracy of the predicted plan. To calculate the actual time taken, we use the travel day whose real time data was used and complete trip data corresponding to that travel

day is provided to the code. This trip data contains the actual arrival times of buses at stops on the travel day. The planner suggested plan is followed on that day and actual time taken is printed with each plan.

```
Enter interval duration (in mins): 120
Enter source: 2431
Enter destination: 432
Enter departure time: 13:00
t2431 t432 13:00

Output:

Plan 1:
        Mean travel time: 59 mins; Variance: 4 mins  Transfers: 0

        Start at stop: 2431
        Take route: 164_D
        Destination stop: 432

        =>Actual Time taken: 59 minutes

Plan 2:
        Mean travel time: 44 mins; Variance: 2 mins  Transfers: 2

        Start at stop: 2431
        Take route: 164_D
        Get down at: 1461
        Take route: 30LTD_D
        Get down at: 551
        Take route: 63_D
        Destination stop: 432

        =>Actual Time taken: 46 minutes

Plan 3:
        Mean travel time: 48 mins; Variance: 3 mins  Transfers: 1

        Start at stop: 2431
        Take route: 164_D
        Get down at: 2822
        Take route: 63_D
        Destination stop: 432

        =>Actual Time taken: 46 minutes
```

Figure 7.3: MLC plan using real time data at source

We see that expected time given by the plans are more closer to the actual time taken by the plan when real time data is use to predict waiting time at source stop (figure 7.3).

7.3 Best time estimator

We use RAPTOR algorithm to find the best possible plan between source and destination for a given departure time, assuming we know the future, i.e. the entire travel day synthetic schedule is given to us and not just the schedule until the departure time.

Given source stop, destination stop and departure time, RAPTOR computes optimal travel plans for public transport networks based on two criteria: arrival time and number of transfers.

It works on timetable (S,T, R, F), where S is a set of bus stops, R is a set of routes. A route consists of a sequence of bus stops. T is a set of trips made by buses on a route. A trip represents a sequence of stops that a vehicle visits along with its arrival time and departure time at that stop. We use the trips information from the Trip data set of the travel day. Since the actual arrival time of the trips on the travel day will be used, it will generate best plan with minimum travel time that could be followed on that day. The best plan will be used to compare the results of different planners.

- ***Database input:*** Entire synthetic trip data of the travel day and Network Description file
- ***User input:*** source, destination and departure time
- ***Planner output:*** Best possible plans between source and destination with 0, 1 and 2 transfers. A travel plan is defined as a sequence of trips in order of travel. Each trip contains pickup and dropping points. Each travel plan has the arrival time at the destination stop and number of transfers associated with the plan. A plan containing k trips has exactly k-1 transfers.

Using Trip data and N/w Description file we obtain information like set of bus stops, set of routes, set of trips made by buses on a given day with arrival time of trips at each stop of the route. We store these information in data structures as shown in figure 7.4.

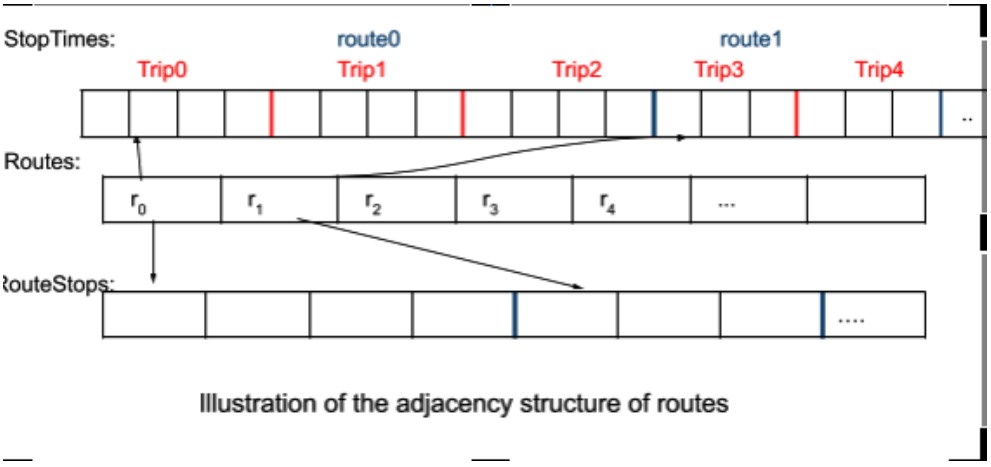


Figure 7.4: RAPTOR data structure

Routes array holds the route information like number of stops, number of trips and pointer to its corresponding stops and trips data structures. RouteStops stores the stops of routes. StopTimes stores all the trip information (arrival time at each stop) of each route.

7.3.1 Algorithm

The algorithm operates in rounds, one per transfer and computes arrival times by traversing every route at most once per round.

Let K be the maximum transfer limit given by the user (here: 3). Each stop p has a multilabel ($\tau_0(p)$, $\tau_1(p)$, ... $\tau_K(p)$), where $\tau_i(p)$ means the earliest arrival time at stop p with i trips. The algorithm finds the multilabel values for each stop roundwise.

In round 1 it finds the earliest time to reach each stop from source stop ps with 0 transfers. Thus, after round 1, we have a value obtained for quantity $\tau_1(p)$ of the multilabel for all p's (stops) that can be reached from ps (source) using direct trips. Those stops that cannot be reached from ps by direct trips (no transfers) has $\tau_1(p)$ as undefined.

Similarly in round 2, we get the earliest arrival time at each stop ($\tau_2(p)$) with 1 transfer, i.e. with 2 trips. This value will be better than $\tau_1(p)$ if it is defined.

Round k computes the fastest way to reach every stop with k-1 transfers (taking k trips). Quantity $\tau_k(p)$ for each stop p is computed in round k.

Algorithm 9 RAPTOR [5]

Input: source ps, destination pd, departure time t

Initialize multi label values for each stop to infinity
▷ Fastest way to reach source with 0 trips
$\tau_0(ps)$ = t ▷ t is the departure time
for k = 1,2 ... K **do**
  Assign $\tau_k(p)$ = Infinity
  ▷ Process each route in each round
  for each route r in R **do**
    Find first stop p' in r where $\tau_{k-1}(p')$ is defined
    current_trip = earliest trip t in r that we can catch at p' ▷ s.t. departure time of t at p' is > $\tau_{k-1}(p')$
    for each stop p_i of r beginning with p' **do**
      ▷ Update $\tau_k(pi)$ to the arrival time of current_trip at p_i if it is better
      if $\tau_{arr}(current\_trip, p_i) < \tau_k(p_i)$ **then**
        $\tau_k(p_i) = \tau_{arr}(current\_trip, p_i)$
      end if
      If we can catch earlier trip at p_i , update the current_trip
    end for
  end for
end for

The main loop of the algorithm runs for round 1 to K. After the end of each round k, we obtain the earliest time to reach every stop p (i.e. $\tau_k(p)$) with k trips and making k-1 transfers.

In each round, each route is processed exactly once. When processing route r in round k, we consider journeys where the last (k'th) trip taken is on route r. At each stop of route r , we find the first trip that we can catch on route r whose departure time from stop p is after the earliest arrival time at stop p with k-1 trips and update the arrival time (for round k) for next stop if it is better than the existing arrival time. This step is basically taking the solutions from the previous round and extending them by one trip if it is improving the arrival time at stops. After processing n number of routes in round k, the best arrival times at stops is either not defined or is better than round k-1, improved by taking the last trip of the journey on any of the n processed routes.

At the end of the algorithm, at each stop, for each k (from 1 to K = max transfers) we have best arrival times obtained with k trips. Parent pointers are associated with each taken trip to obtain a full travel plan.

Optimizations used in the implementation:

- If there is no improvement in arrival times of any of the stops in round

k (from round k-1), the algorithm stops.

- Instead of processing every route once in each round, only selected routes can be processed. During round k-1 we mark those stops pi for which the arrival time $\tau_{k-1}(p_i)$ was improved. Before starting round k, we find all routes containing marked stops. Only routes from the resulting set are considered for scanning in round k.

The worst-case running time of this RAPTOR algorithm is as follows. Every round scans each route r ∈ R at most once. Let |r| be the number of stops along r, then there are $\sum_{r \in R} |r|$ stops in total. We look at every trip t of a route at most once, since the earliest trip in r only decreases while scanning a route. In total, it takes O (K (($\sum_{r \in R} |r|$) + |T |+|F|) time, where K is the number of rounds.

7.3.2 Output

Source stop number = 2431; Destination stop number = 432; Departure time of user = 13:00

```
Enter source: 2431
Enter destination: 432
Enter departure time: 13:00

Source: 2431 Destination: 432 Departure_time: 13:00

Plan with 1 trips
        Best time taken to reach destination: 59 minutes

        Take bus 164_D at stop 2431 then
        Get down at destination: 432

Plan with 2 trips
        Best time taken to reach destination: 46 minutes

        Take bus 164_D at stop 2431 then
        Take bus 63_D at stop 2822 then
        Get down at destination: 432

Plan with 3 trips
        Best time taken to reach destination: 41 minutes

        Take bus 164_D at stop 2431 then
        Take bus 30LTD_D at stop 1461 then
        Take bus 172_D at stop 2364 then
        Get down at destination: 432
```

Figure 7.5: Best plans

Above figure shows best possible plans generated by RAPTOR using complete travel day information. In figure 7.5 we see different plans are generated with transfers 0, 1 and 2 with the best time taken to reach the destination.

7.4 Experimental results

We perform various experiments to evaluate different planners.

7.4.1 Plans comparison with actual and best time

We can compare the predicted time of the plan with the actual and best times.

Figure 7.6, 7.7, 7.8 compares real-time MLC predicted time, Actual time taken by the plan and Best possible plan time for different departure times. The x axis represents different source-destination pairs and y-axis represents the travel time in minutes. If MLC generates m different plans for same source-destination, m different points are made on the x axis, one for each plan. Orange plot line is the predicted time by the MLC planner which uses real time data. Blue line is the worst predicted time by the real time MLC planner, obtained by adding variance to the mean predicted time. Green plot line plots the actual time taken by the plan on the travel day and Red line is the best possible plan time for that travel day given by RAPTOR. Interval duration for MLC planner is taken to be 60 minutes.

Figure 7.6 is the comparison between plans when departure time is 9 a.m. We see that the predicted time by MLC is slightly overestimated as compared to the actual time taken. The actual time plot and the best time plot overlaps for most of the part, suggesting that best plans are generated by MLC that uses real time data at first stop.

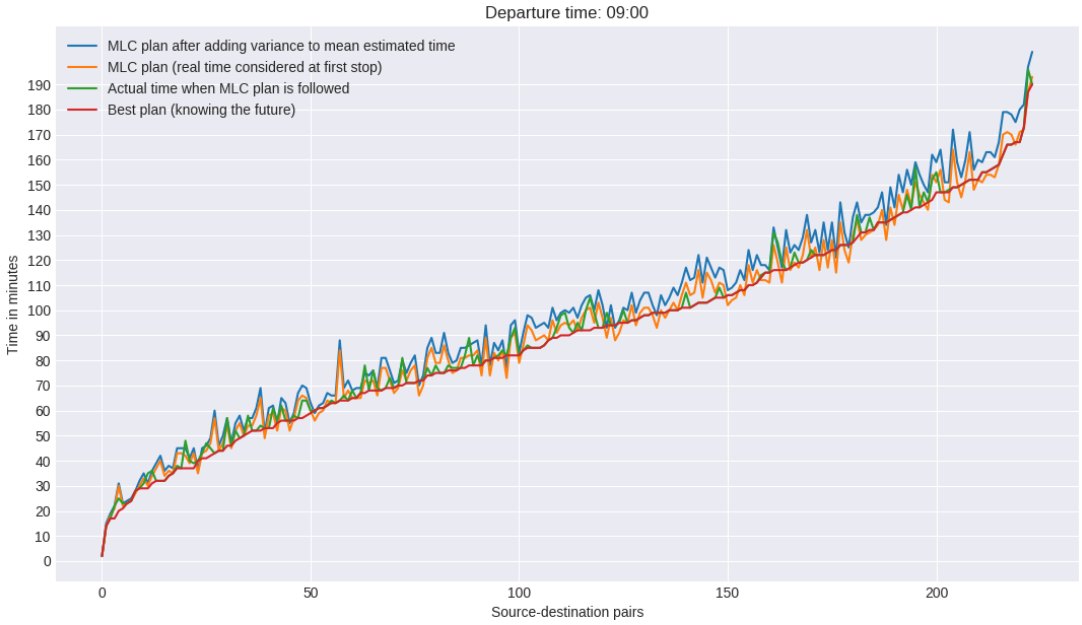


Figure 7.6: Plans comparison at 09:00

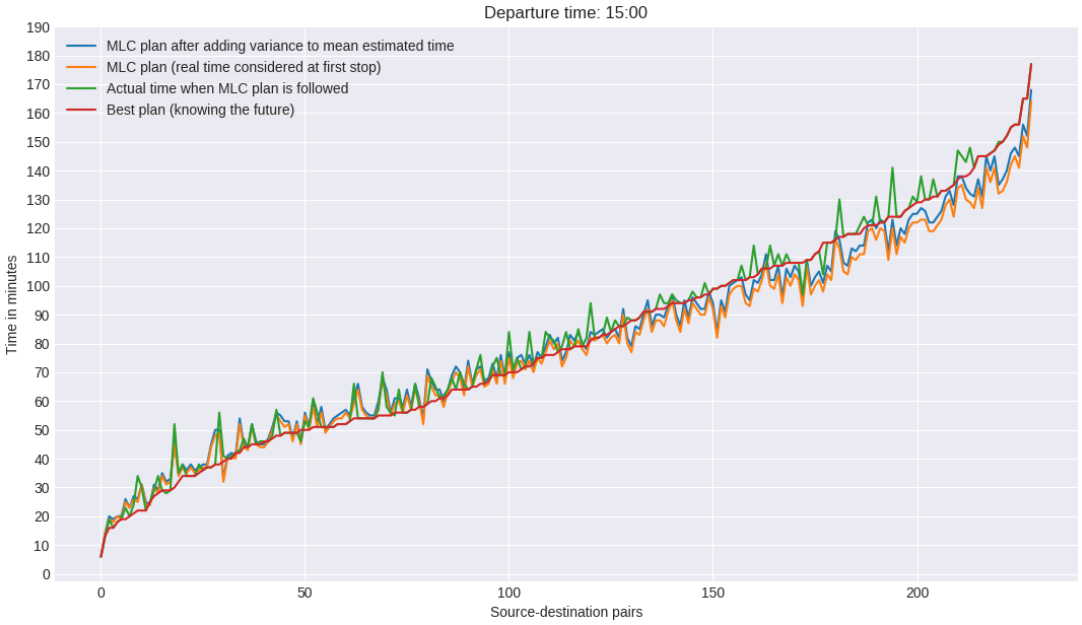


Figure 7.7: Plans comparison at 15:00

Figure 7.7 compares plans when departure time is 3 pm. In figure, we see that the actual time taken by the plan overlaps with the predicted time of the plan if the journey time is within one hour. If it is more than one hour, MLC underestimates the travel time. This is because 3 pm to 4 pm is a non peak hour but after 4 pm, peak hours start, according to our data generation model. Since MLC considers the weighted average of two intervals (best and second best interval) while calculating edge weights, it underestimates the travel time due to the non-peak hour interval timings also taken into account.

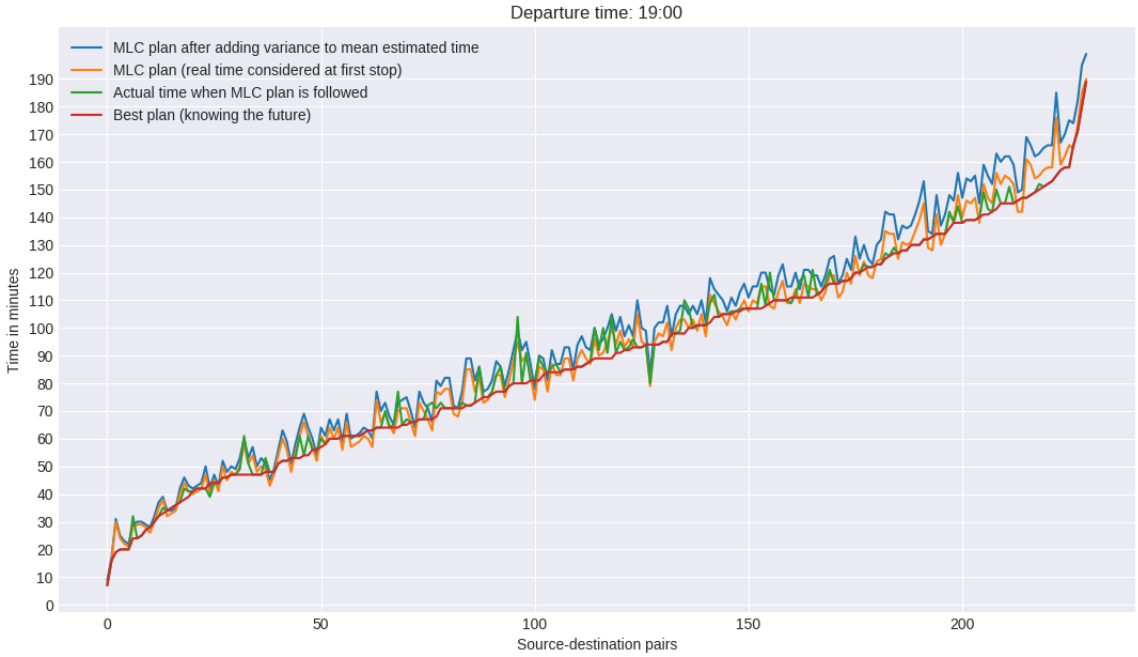


Figure 7.8: Plans comparison at 19:00

Above figure compares plans when departure time is 7 pm. In figure 7.8, we see that variance given by MLC planner is high. The actual plan time seems to overlap with the best time for most of the plans.

7.4.2 Accuracy of expected total time

In this experiment, 200 different source destination pairs are used to measure the accuracy of the expected total travel times returned by different planners. Root mean square error (RMSE) is used as a metric. It calculates the difference between predicted travel time by the planner and actual

time taken on following the plan. Different planners that are compared are MLC that uses real time data to predict waiting time at first stop, MLC that uses only past data, Mumbai Navigator that picks the appropriate database based on the departure time and Mumbai Navigator that uses whole day average frequency and stop to stop times to estimate travel time.

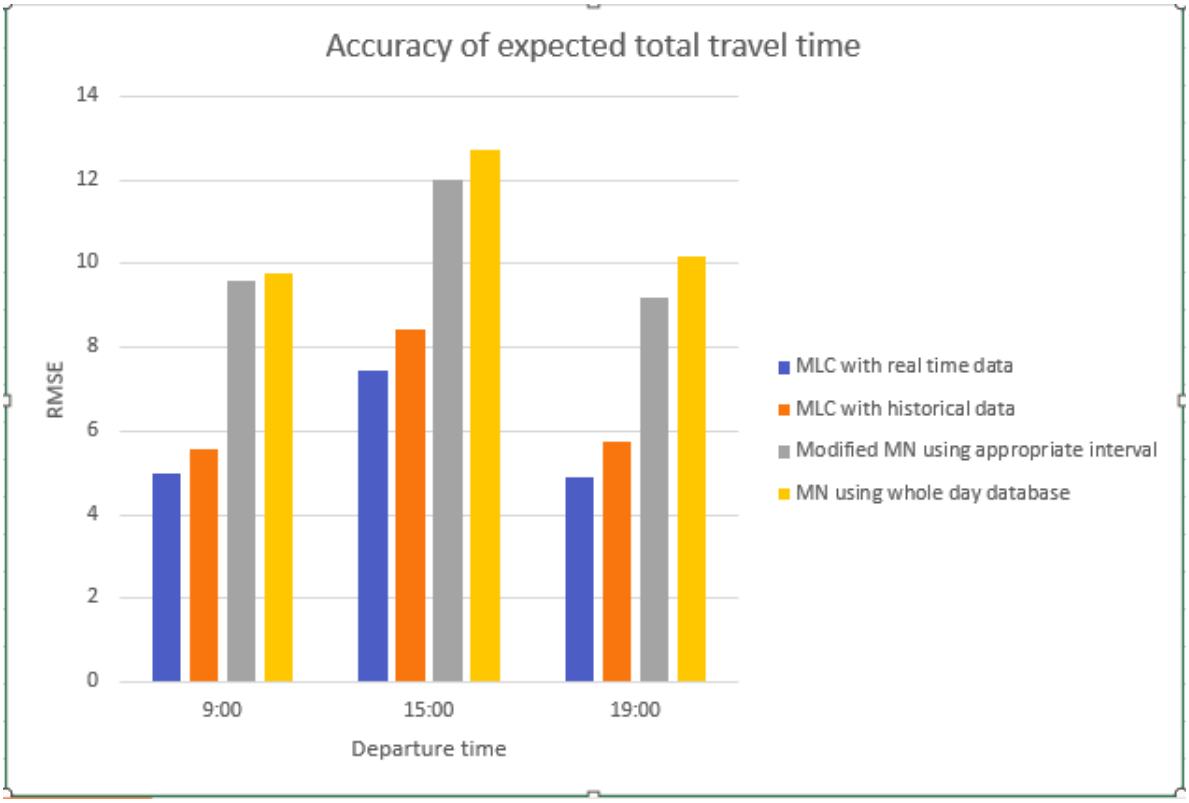


Figure 7.9: Accuracy of expected total travel time

In figure 7.9, we see that MN using whole day database suffers from the largest error at all departure times. MN that uses appropriate database based on departure time is better than the MN that does not. MLC planner that uses only past data, predicts travel time better than the Mumbai Navigator. MLC that uses real time data, suffers from least error and it estimates the travel time most accurately.

7.4.3 Are best plans generated?

Figure 7.10, calculates the difference in actual time taken by planner's plan on the travel day and the best time possible on the travel day to go from

source to destination at a given departure time. This tells us if the planner is giving the best plan. We see that MLC with real time data is best in generating best possible plans. Both the variants of Mumbai Navigator has almost same error, but worse than real time MLC planner. MLC with only past data gives maximum error, i.e. this planner generated best plans least number of times as compared to other planners.

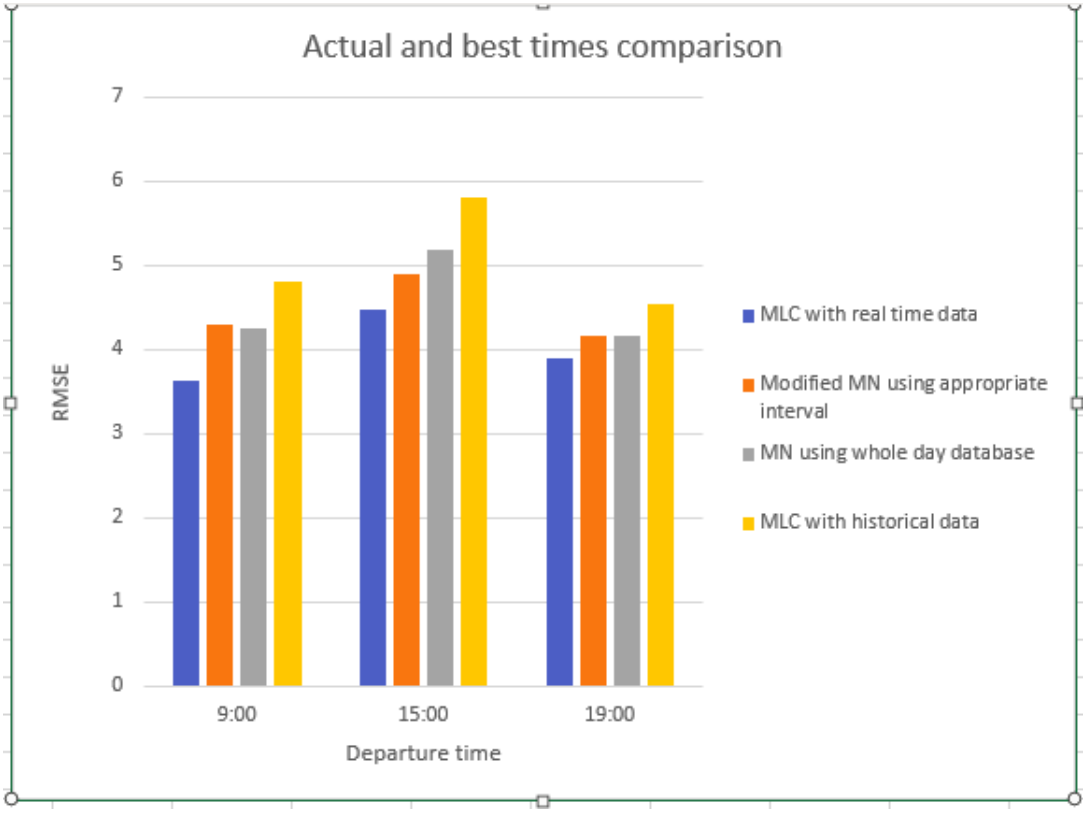


Figure 7.10: Actual and best times comparison

From the above two figures we see that MLC with real time is best in estimating the travel time and also best in giving the best possible plans as compared to other planners.

7.4.4 Plans to pick in MLC

MLC planners generates multiple non dominating plans minimizing multiple criteria. The question arises which plan should be picked by the user so that the best plan is selected.

Say, MLC1 = MLC planner using only past data, generating multiple non-dominating plans
MLC 1a = MLC planner using only past data, generating only minimum expected travel time plan
MLC 2 = MLC planner using real time data, generating multiple non-dominating plans
MLC 2a = MLC planner using real time data, generating only minimum expected travel time plan

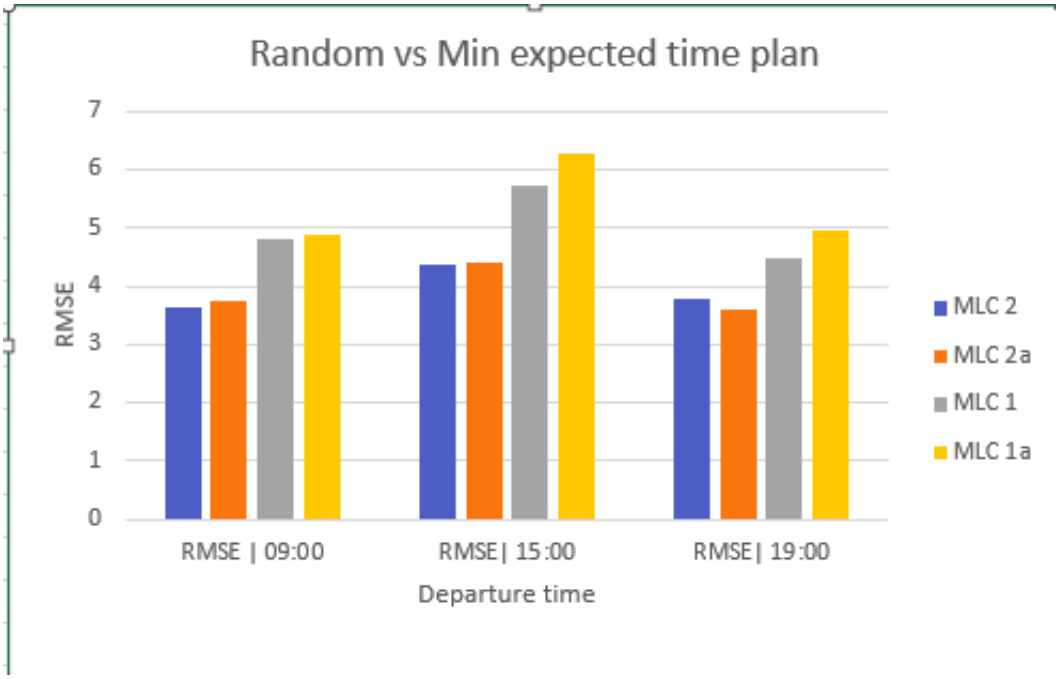


Figure 7.11: Random plan vs Minimum expected time plan: RMSE

Figure 7.12: Random plan vs Minimum expected time plan: MAE

When we use RMSE and mean absolute error as metrics, both gives same results. Picking min travel time plan over other plans does not assure that it is the best plan. Therefore, picking plans at random is better than picking plan with minimum expected travel time

Chapter 8

Conclusion and Future Work

In this project we looked at different variants of Trip planning in public transport and how it is different from simple shortest path problems. We looked at the BEST ticketing data and the process of information extraction to generate a smaller and common dataset for all Trip planners called "Trip data". We understood the Mumbai Navigator algorithm and the changes made in it to use the time interval that departure time falls in to generate better plans. Inconsistencies in BEST data led to the synthetic data creation. The synthetic data was created to generate Trip data with frequency and speed variations in buses at different times of the day. The generated synthetic Trip data was used with different planners as an input to generate plans. Mumbai Navigator minimizes expected travel time and generates plan trees. MLC planner uses past data to obtain plans minimizing time, variance and transfers. MLC planner developed with real time data generates fast and reliable plans with minimum transfers. Experiments were performed to evaluate different planners. Important results show that using real time data is beneficial in estimating travel times and generating best possible plans.

We would like to run these planners on the actual Mumbai bus data and compare the performance of these planners. In future, we can use real time data at later stops also to generate adaptive plans.

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] Richard P. Guenthner and Kasimin Hamat. Distribution of Bus Transit On-Time Performance. Transportation Research Record 1202, Transit Issues and Recent Advances in Planning and Operations Technique, 1988.

[8] https://www.tutorialspoint.com/cplusplus/cpp_date_time.htm

[9] Peng Ni, Hoang Tam Vo, Daniel Dahlmeier. DEPART: Dynamic Route Planning in Stochastic Time-Dependent Public Transit Networks.

IEEE 18th International Conference on Intelligent Transportation Systems, 2015.