Data Structures to Represent Public Transport Systems for Small Towns

Case Study Of Dharwad

Charu Agarwal

Abstract

Commuter bus systems plays a crucial role in the working of a city, especially for the non-metropolitan towns, where it is the primary mode of transport. For efficient planning, management and utilization of a citys transport system, a well-organized and structured database is necessary. While different cities follow different conventions to store and manage data, it is not clear if this representation allows for data analysis and optimization. This study outlines one possible methodology to formalize this data into structures which can be used for query and analysis purposes. The methodology is made simple enough to be implemented by the novice programmer. One unique feature is the unification of the tabular data of the schedules with geospatial data of routes, which produces a rich and powerful database. It is followed by some applications and algorithms briefly describing the arenas where this data can be utilized by researchers, depot managers, and everyday passengers.

Introduction

Imagine you are a student who wants to travel from Ganesh Temple to Central School, starting at 7:15 AM, in the minimum possible time. Or you are in a planning committee and would like to know which are the more profitable areas to add a new bus route. Or perhaps, you are a researcher who wants to study how much percentage of the rural population has access to a bus within 500m. Such fairly complex questions can be easily answered using graph theory and other tools. Now, some of us may be familiar with

the graph notations and algorithms and would be able to churn out a program in five minutes (or less) that would perform such computations for us. But to implement such nice algorithms, we require data in a form that is ready to use and understood by the computer. For the sake of specificity of purpose, the study has been sub-divided into four sections. Sections 1 and 2 describe the process of collection of data and the cleanup process. The utilization and applications are emphasized in sections 3 and 4, including a brief outline of the development of a mobile navigation application and a modified, optimized version of the shortest path algorithm suitable to be run on a mobile phone. Much of the algorithms described in this study are supported by pseudo code to provide easy implementation for future use.

1 Background And Collection Of Data

The Dharwad bus depot is one of the eight depots under NWKRTC (North Western Karnataka Road Transportation Corporation). It caters to local buses as well as nearby and long distance buses. For the purpose of this study, we restricted ourselves to local buses. There are around 65 local routes (considering to and fro as the same route), 250 bus stops and 2050 bus trips per day.

1.1 Acquisition 1: Form IV(The city bus schedule)

The most basic requirement to analyze a bus network is to begin with the operational representation used by the City Bus System. We began with the so-called Form IV, the city schedule, a format used by NWKRTC. This is organized in an unstructured 8000 line excel workbook, a snapshot of which is given below (see Fig. 1). Form IV is organized as a sequence of schedules, with each schedule consists of 10-20 trips. Each trip has a start destination, start time and an end-destination and end-time. Each schedule is typically a shift done by a crew consisting of a conductor and a driver. Thus, the schedule is a collection of 10-20 rows in Form IV. As mentioned, this was an 8000 line excel workbook where many non-functional schedules were also present. After verification from the Depot, we compressed it to a 2000 line excel sheet.

CC	F M	1 -	07:05	7:20	-	0:00	89	
DCS-3B	19	CITY	CBT	SOMESHWAR TMP	6	14:15	14:35	YAMMIKERE, R.MATH
	20	"	SOMESHWAR TMP	CBT	6	14:40	14:00	YAMMIKERE, R.MATH
	21	"	CBT	TEJASHWI NAGAR	5	15:05	15:20	YAMMIKERE, R.MATH
	22	"	TEJASHWI NAGAR	CBT	5	15:35	15:50	YAMMIKERE, R.MATH
	23	"	CBT	TEJASHWI NAGAR	5	15:55	16:10	YAMMIKERE, R.MATH
	24	"	TEJASHWI NAGAR	CBT	5	16:20	16:35	YAMMIKERE, R.MATH
	25	"	CBT	TEJASHWI NAGAR	5	16:40	16:55	YAMMIKERE, R.MATH
	26	"	TEJASHWI NAGAR	CBT	5	17:10	17:25	YAMMIKERE, R.MATH
	27	"	CBT	SARASWATIPUR	4	17:30	17:45	YAMMIKERE
	28	"	SARASWATHIPUR	CBT	4	17:50	18:05	YAMMIKERE
	29	"	CBT	TEJASHWI NAGAR	5	18:20	18:35	YAMMIKERE, R.MATH
	30	"	TEJASHWI NAGAR	CBT	5	18:40	18:55	REST
	31	"	CBT	TEJASHWI NAGAR	5	19:10	19:25	YAMMIKERE, R.MATH
	32	"	TEJASHWI NAGAR	CBT	5	19:40	19:55	YAMMIKERE, R.MATH
	33	"	CBT	TEJASHWI NAGAR	5	20:10	20:25	YAMMIKERE, R.MATH
	34	"	TEJASHWI NAGAR	CBT	5	20:30	20:45	REST
	35	"	CBT	S.R.NAGAR	5	21:15	21:30	YAMMIKERE, R.MATH
	36	"	S.R.NAGAR	CBT	5	21:35	21:50	YAMMIKERE, R.MATH
N/O CBT		1 -	07:35	6:00	-	0:00	90	179

Figure 1: A snapshot of Form IV

The information provided by the depot was not sufficient on several counts.

1. It did not contain any information about the route followed in a trip or the stops within the trip.

2. The names of the terminal destinations were not standard.

3. It did not enable other analysis, such as crew or vehicle utilization.

Dharwad Bus Depot did have the conductor carrying a ticket-vending machine for all routes but the stop-data was non-standard and did not have route/geo-tags. Whence it was decided that (a) route and stop data would be obtained, and (b) suitable schemas would be developed to represent Form IV and to perform other analysis.

1.2 Acquisition 2: KML Files of Routes

The second important task was to get the geospatial information of the routes. Now while the Dharwad depot has some newly launched buses which are equipped with GPS, most of the buses do not have this functionality. So it was decided to undertake field work and generate the data. Using the data from Form IV, we extracted the different routes and allotted them to the class of 39 students of the course CS213 (Jan. 2017-May. 2017), at IIT Dharwad,

to track the routes in pairs. The application used is called GeoTracker[1], available for free to download on the Google Play Store which generates a KML file (a text format) using the GPS position of the mobile phone. Each student team was required to hop on three bus routes, use the app to mark the stops and submit the KML files of the routes. The kml file produced for a route may be viewed on standard platforms, and consisted of two segments: (i) Part I. The stops as place-points (about 20) and its data, and (ii) Part II. A sequence of 1000-odd lat-longs which tracked the route. This will be called as the track of the route. See Fig. 2 for a snapshot of two of kml files. Note the irregular route due to tracking error and bus movements and also the two different names and locations given by the two teams travelling on the routes.

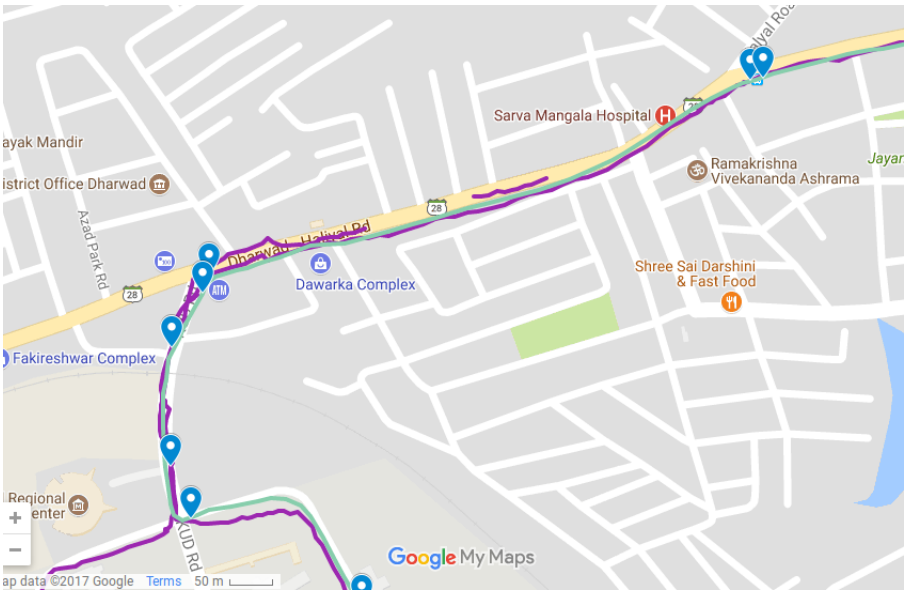

Figure 2: Two Routes.

2 The Representation

Now came the rather tedious task of formalizing the data we had in hand, so that it can be utilized.

2.1 Developing a data structure as a graph

It was decided to use the graph data structure as the basis for representing key concepts. A network graph $G(V, E)$ consists of a set of vertices V, and E, a set of edges. An edge is an ordered/unordered tuple (v, v), where v, v are vertices. A graph is eminently suitable to represent locations (as vertices) and paths between locations (as edges). Thus graphs were chosen as the basic structure which would cater to our requirements. We will now describe (i) the definition and construction of vertices, (ii) the definition and construction of edges, and finally, (iii) the use of paths, i.e., a connected sequence of edges, to represent routes. Here is a formal description of the data structure:
There are 4 key **structs**, viz. **place**, **edge**, **route** and **schedule**. The **place** represents an actual location and contains location data (latitude-longitude (lat-long)), name and other attributes specific to the place and the route on which it lies. An **edge** consists of two places $(p1, p2)$ and a track between them. A **route** consists of a sequence of places, connected by the edges. A **service** consists of a route followed by a bus along with the start time and the trip duration. Finally, a **schedule** is a set of services carried out by a single bus. A standard schedule is 8 hour long and typically consists of 20 services, which usually are to and fro trips along a single route.

```
struct place
{
double latitude,longitude;
String name;
int id;
int eindex[]; //stores the index of edges starting from the place
};
struct edge{
int from, to, duration, route_no, id;
String polyline;
};
struct Route{
int id;
```

```
int from, to; //stores the id of the terminal stops
};
struct Service{
int route_id, duration; //duration of each trip
String start_time; //stores the start times of the service in
    HH:MM format
int scheduleID;              //stores the id of the schedule it
    belongs to
};
struct Schedule{
    String sch_name;
    double actual_dur; //stores total schedule duration in hours
    double running_time; //stores the duration the bus operates on
        road (excluding breaks)
    String startTime; //stores starting time of schedule in HH:MM
        format
    String endTime;
    int nTrips; //stores number of trips in a schedule
    Service services[]; // array which stores the services catered
        to by the schedule
};
```

The remaining part of this section contains more technical details of the process for actual implementation and may be skipped without any loss of context.

2.2 Standardization of Names in Form IV

There is no standard algorithm for this part. We tried to compare the stop names based on difference in characters in name but it was not very reliable. (For eg, Bendre Bhavan and Bendre Bus Stop are completely different stops while Vidyagiri and JSS refer to same stop.) So we had to manually modify the ambiguous names using discretion. To verify we had corrected all the names, we then wrote a program which generated all the unique stop names and inspected them.

2.3 Extraction of Substops from the KML Files

KML files have a feature called Placemark, which stores attributes like name, timestamp, elevation and coordinates. To extract these placemarks, we used a web utility [2] which converted KML files into CSV files containing list of attributes.

2.4 Clustering of the Stops

The coordinates and names of the stops (extracted from the KML files) did not match exactly as the routes had been tracked by different students at different times. We used a clustering algorithm using connected components to synchronize stop names and coordinates. An auxiliary graph was constructed which used the stops as vertices. An edge was put between two stops if the distance between them was within xx meters. Each connected component of this auxiliary graph was converted into a place with its coordinate as the mean lat-long of its member stops. These lat-longs were temporary and were corrected later, see below. Standardized names obtained from 2.1 and others from Google Maps were used to name the places. This completes the preparation of the vertices of our final network graph.

2.5 Fitting the GPS generated tracks with the Google Map Roads

Now it is time to look at the generation of edges and paths in the network graph G. It was decided that the edges between places (whose preliminary lat-longs and names are now available) will follow existing roads, i.e., polylines as shown by Google Maps. For this, we began with Part II of the kml files, i.e., the tracks, which consist of a sequence of lat-longs followed by the bus. As the tracks were generated from real time tracking, they were quite disorderly and deviated from the roads a lot. To align the tracks with the official road polylines, we used a Google API[3]. From documentation: This service takes up to 100 GPS points collected along a route, and returns a similar set of data with the points snapped to the most likely roads the vehicle was traveling along. Thus, what could have been done in one go, had to be done by feeding a selection of 100 points from the track.

Algorithm 1 Clustering Algorithm

procedure CLUSTER(G) ▷ G is a graph with vertices as stops and there exists edge between vertices u and v iff $dist(u, v) < \epsilon$

```
for all v in G do
    label as undiscovered
end for
for all v in G do
    if v is undiscovered then
        sumCoordinates ← 0
        numberOfStops ← 0
        Q ← queue initialized with v
        while Q is not empty do
            current ← Q.dequeue()
            sumCoordinates ← sumCoordinates+currentCoordinates
            increase numberOfStops by 1
            for each node n that is adjacent to current do
                if n is not labeled as discovered then
                    label n as discovered
                    Q.enqueue(n)
                end if
            end for
        end while
        meanCoordinates ← sumCoordinates/numberOfStops
        update all stops in the current connected component with
meanCoordinates
    end if
end for
end procedure
```

We first converted the KML files into GeoJSON format using a web utility [4], wrote a program which extracted the track T_i from the file and made a selection of 100 points from the track at regular intervals. This selection I_i was fed as input to the Google API to obtain O_i, the output, which consisted of a set of 100 points, defining a track aligned with the road. We then compared the track O_i returned by the API with the original track T_i. Care was taken in the selection of input I_i to ensure that O_i closely followed T_i.

Finally, O_i was replaced by T_i.

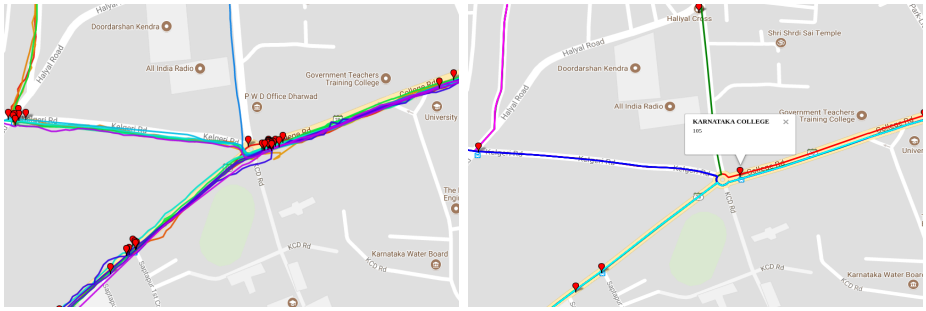

Figure 3: Here is a before v/s after for comparison after steps 3.2, 3.3 and 3.4. All stops have been clustered and the tracks have been aligned to the roads.

2.6 Inserting stops into polylines

To generate edges between vertices, we needed polylines. As the data was generated from one trip only, many buses did not stop at all the stops which lie on the route depending on the ridership that day. So, we then wrote a program which took all the stops, computed its minimum distance from the track and inserted it into the track O_i if the distance was less than some chosen epsilon. Thus, at the end of this phase, all the data from the various original kml files has been processed. For each original track T_i, we have a new track O_i which follows Google Map roads. All stops which were marked on this or any other track, have been inserted into the new tracks O_i . Note that now O_i conceptually consisted of a sequence of places $(p(i,1), p(i,2), ..., p(i,k))$ and for each consecutive places $(p(i,j), p(i,j+1))$, there was a track which had been validated by the Google API. This was used to generate the edges of the graph G. For two places p and q, we put an edge $e = (p,q)$ if there was a track O, in which these were consecutive. Moreover, the attribute polyline of the edge was what was obtained from the Google's polyline encoding algorithm [5] to encode the polylines into strings, which is easy to store and can be decoded later as needed. Please note that if there were two tracks O_i and O_j on which p and q were consecutive, the polylines obtained would be identical.

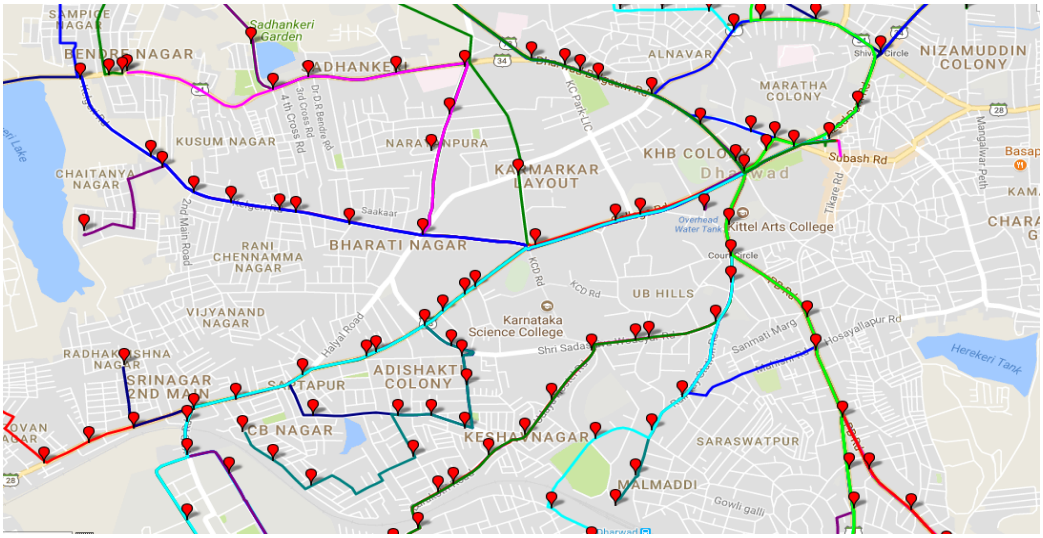

Figure 4: Final bus network after cleanup.

The Route data-structure was now constituted from edges constructed as above. The duration field was computed from the original track. Finally, Services were formed and Form IV was re-configured into a database as a collection of Schedules, where each Schedule was a collection of services and start-times. Further, there was a database for all stops which acted as vertices in the graph representation, and a database which stored the route information.

_id	route	start	duration	sch_no
875	44	6:15	20	21
876	44	7:55	20	21
877	44	8:45	20	21
878	44	9:35	20	21
879	44	17:05	20	33
880	44	17:55	20	33

Figure 5: Sample service database. Here the service is shown for the route CBT-UAB (Route no. 44). Also, reverse routes have been marked by adding a minus sign to the route number (eg. UAB-CBT will be route no. -44 in the service database).

_id	name	longitude	latitude
1	9THCROSSKALYANNAGAR	74.9953	15.4382
2	ANJANEYA NAGAR	74.9648	15.46
3	ADARSHA NAGAR	74.9809	15.4743
4	AGRI COLLEGE	75.0058	15.4613
5	AGRICULTUREUNIVERSITYSTOP	74.9765	15.4869
6	AIRTECH	74.970665	15.492198

Figure 6: Sample database to identify stops (vertices).

_id	route	duration	start	end	polyline
1	1	1	143	146	sdh}AsfrhMKG??b@u@??h@w@??d@aA??b@u@??`@
2	1	1	146	18	eqg}AoyrhM??]w@_A{BEM??i@iA??OYc@}@??O[??
3	1	1	18	212	wwg}AoeshM??GQa@gAIY??Og@AC??i@oC_@eB??
4	1	2	212	221	g|g}AershM??ESo@mC[{A??GO??Uy@CK??Sm@]qAAG
5	1	1	221	153	}ah}AobthM??Ga@Kk@??YwA??Ks@Im@?? @wB??

Figure 7: Sample database to store route data (edges). Duration is stored in minutes. The ids in the stops database are used to mark the start and end stops. The polyline has been encoded.

_id	mFROM	mTO	km
1	144	33	6
2	33	3	7
3	33	233	10

Figure 8: Sample database for route ids.

sch_no	running_time	start_time	end_time	actual_dur	trips
DCS -49	7.3	8:00	18:05	10.08	24
DCS 19A	6	7:00	14:35	7.58	18
DCS 19B	6	15:00	22:45	7.75	18
DCS 1A	5	6:30	14:25	7.91	15
DCS 1B	5.67	14:45	22:30	7.75	17

Figure 9: Sample database for schedules.

Results of sample queries on the database appear at the end.

3 Development of the mobile navigation app

Our next goal was to develop a mobile application that would display the schedule to the passengers, show the route on map and furthermore, suggest routes to go from Place X to Place Y(not necessarily on the same route) in minimum time, given a starting time. Now, we could have simply implemented the shortest path algorithm in its original form, but as this app was to be developed with more rural users in mind, it was desirable that the app worked offline. This posed a significant challenge as we were very restricted in terms of memory resources as the entire computation has to be done on an average mobile phone instead of a server. So, we modified the algorithm as follows:

In the standard shortest path algorithm, at every step, we select the minimum cost vertex (here stop) and update the distance, if more, to every unvisited vertex that is directly connected to it, with the cost being the time taken to reach the current stop plus the time taken to reach the adjacent stop using a particular route. Suppose there are k routes having n stops each. The total number of edges in this case would be (n-1) *k. If we identify each unique event of arrival/departure as a distinct edge, and suppose 1 route has m trips to and fro per day, the number of edges would increase to (n-1) *m*k.

We note that we are only interested in the next bus. So instead of explicitly storing the edges for all buses, we simply compute the next bus from the current time and then compare our options.

The present cost of an edge depends on the current time also, along with the edge duration. For example, suppose there are two buses which go from stop A to stop B: bus X starts at 7:30 AM while bus Y starts at 8:00 AM. For the sake of simplicity, assume that bus X takes 50 minutes to complete its journey, while bus Y takes 25 minutes to complete its journey. Which bus should you take to reach earliest? The shorter duration one? Well, the feasibility and preference would depend on what time you start your journey. If you arrive at 7:10AM, your expenditure would be 20 minutes (waiting time) + 50 minutes (travel time) if you take bus X, and 50 minutes (waiting time) + 25 minutes (travel time) to reach from stop A to stop B, by bus Y. Clearly, bus X is preferred in this situation and the expenditure would be 70 minutes. However, if you arrive at stop A at 7:35 AM, you would take bus Y and your expenditure would be 50 minutes only.

Hence, the cost of an edge at the present event depends on the cost of all the previous edges used to reach the current stop and the starting time (as they

determine the present time). So, instead of creating all the time dependent edges in the beginning, we created only route edges and recomputed the cost of each edge at runtime as Current Cost= waiting time + edge duration; where waiting time is defined as the minimum non-negative difference between the starting time of the bus and the current time; infinity if no such time exists. Now that our algorithm appears to be correct intuitively, we give the pseudo code. (See next page)

Algorithm 2 Algorithm for finding bus route

procedure RoutePlanner(Graph, source, destination, start time)
create vertex set Q
for each vertex v in Graph **do** ▷ Initialization
$dist[v] \leftarrow \infty$ ▷ Unknown distance from source to v
$prev[v] \leftarrow UNDEFINED$ ▷ Previous edge in optimal path from source
$arrival\_time[v] \leftarrow \infty$ ▷ Stores the best arrival time at each vertex
end for
add v to Q ▷ All nodes initially in Q (unvisited nodes)
$dist[source] \leftarrow 0$ ▷ Distance from source to source
$arrival\_time[source] \leftarrow start\ time$
while Q is not empty **do**
$u \leftarrow$ vertex in Q with min dist[u] ▷ Node with the least distance will be selected first
remove u from Q
if u is destination **then**
break ▷ no need of further computation
end if
for each edge starting from u **do** ▷ where v is still in Q
$length(u, v) \leftarrow waitingtime(arrival\_time[u]) + length(u, v)$ ▷ re compute edge cost
$alt \leftarrow dist[u] + length(u, v)$
if alt < dist[v] **then** ▷ A shorter path to v has been found
$dist[v] \leftarrow alt$
$prev[v] \leftarrow e(u, v)$
$arrival\_time[v] \leftarrow arrival\_time[u] + length(u, v)$
end if
end for
end while
return dist[ ], prev[ ]
end procedure

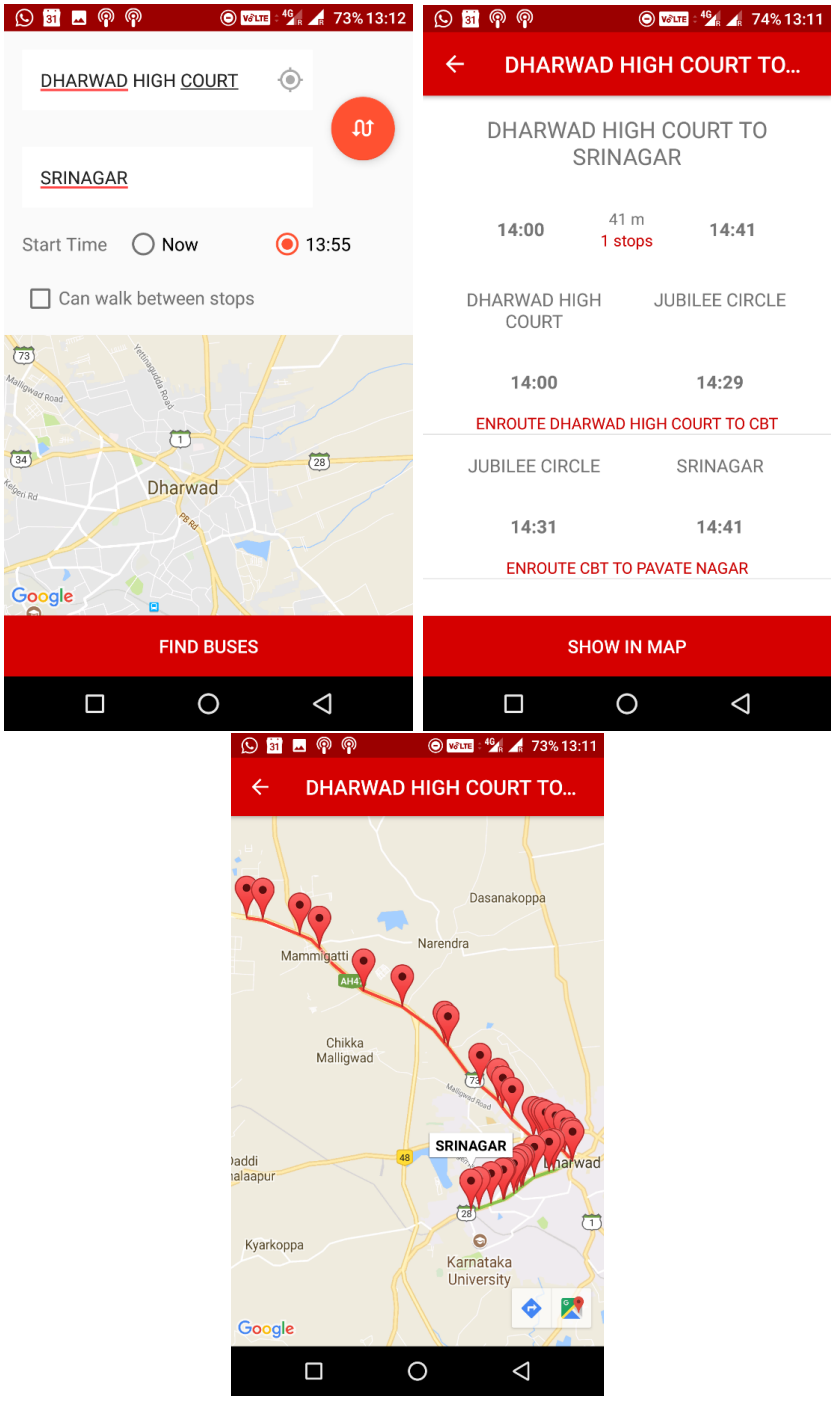

Figure 10: Working sample of the app: Query from High Court to Sringar starting at 13:55.

3.1 Adding Walking Edges

We noticed that in certain situations, it may be more advantageous to walk a few hundred meters than to take a bus as it covered a larger loop to come to the same point. So we added a feature to enable walking where a user can give as input the maximum distance he/she is willing to walk between two stops and then our program adds additional walking edges between the stops so that an even more optimized route is generated.

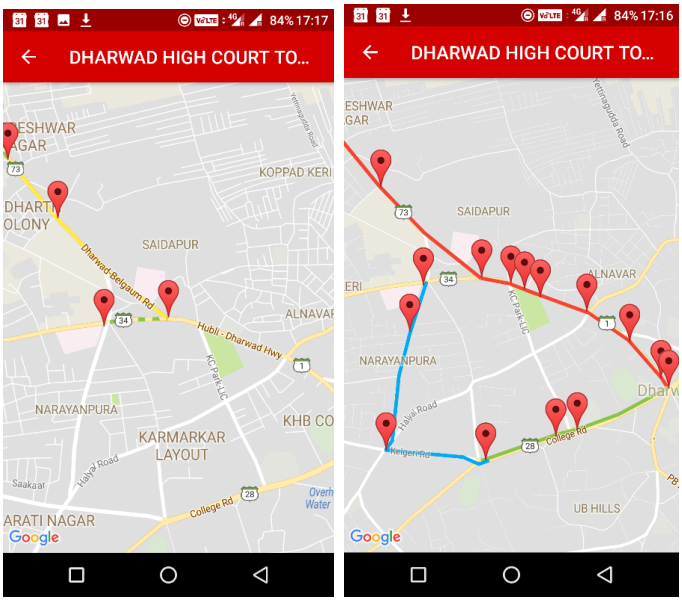

Figure 11: Sample query for Dharwad High Court to German Hospital with(left) and without(right) walking feature enabled. (All walking routes are dashed)

3.2 Current Location Feature

Quite often the user will not be at a bus stop and would like to know how to reach the nearest bus stop by walking and then catch a bus. We have added a feature which uses the current location of the user to compute the nearest bus stop and gives directions accordingly.

4 Further Analysis and Limitations

The shape files of the routes generated were loaded up into a GIS application QGIS, along with the ward data from the municipal corporation to estimate how many wards had access to a bus.

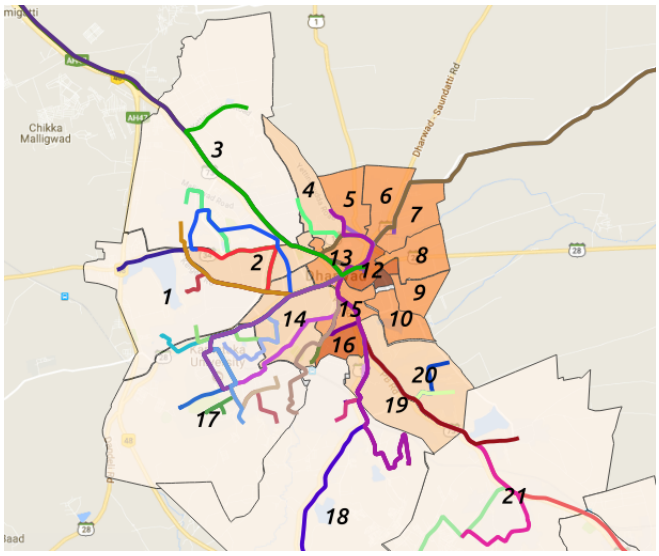

Figure 12: The bus route network has been superimposed over the wards, classified by population density.

Using the database, several queries were made to generate plots to analyze crew and resource utilization.
We plotted the number of trips made by the buses as a function of time. (See Fig. 14) As expected, a standard bell curve with slight dip between 13:00-15:00hrs. It seems they have planned the frequency of buses to meet the demand reasonably well. A plot of distribution of the schedule duration (see Fig. 13) shows that for 29 out of 101 schedules, the schedule duration is more than 8 hours. The depot is paying overtime charges for over 82 hours every day. For the 72 schedules whose duration is less than or equal to 8h, a total of 35 working hours are wasted every day. Comparing the time bus is stationary v/s the time bus is moving during the schedule (see Fig. 15), over 39% buses are lying idle for more than 40% of their running duration, especially for shorter routes for which the idle time is comparable to running time. This is as the depot has allotted standard breaks in multiples of 5

minutes between the trips. But when the bus trip is only 10 minutes long, it is quite wasteful to make it wait for 10 or more minutes. However, the break time cannot be reduced too much also, as some time is needed to allow new passengers to climb the bus, before the bus begins the next trip.

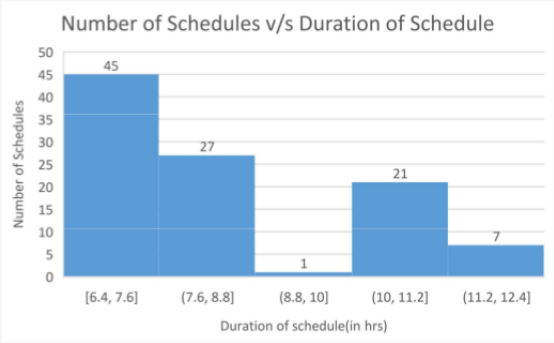

Figure 13: Plot of Number of Schedules v/s the schedule duration. (This duration is the cumulative duration of the schedule, including breaks.)

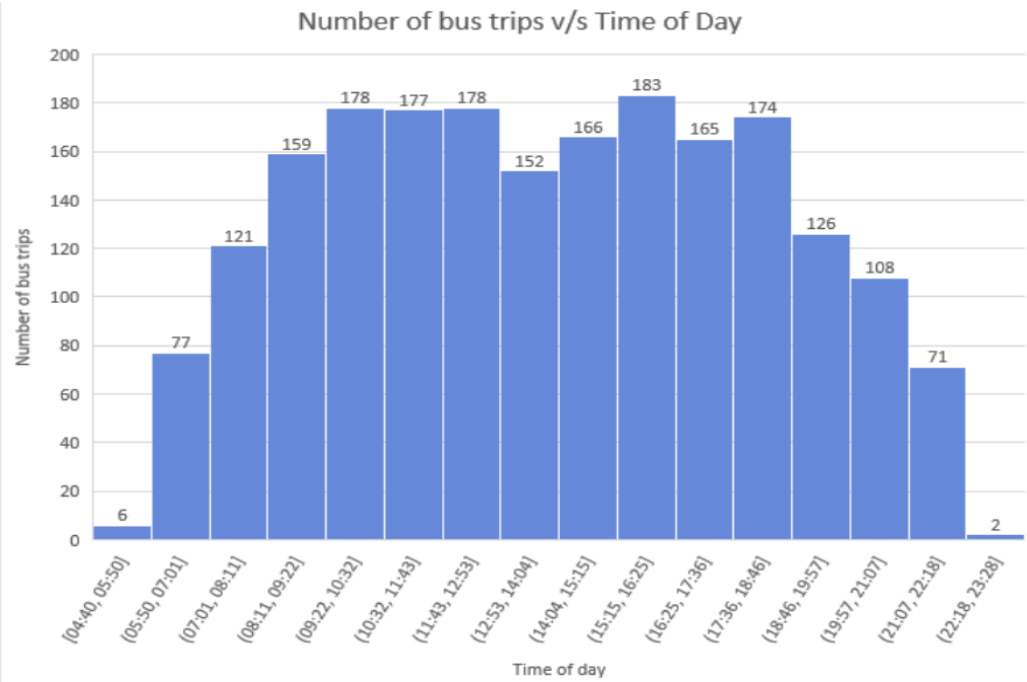

Figure 14: Distribution of the number of bus trips throughout the day.

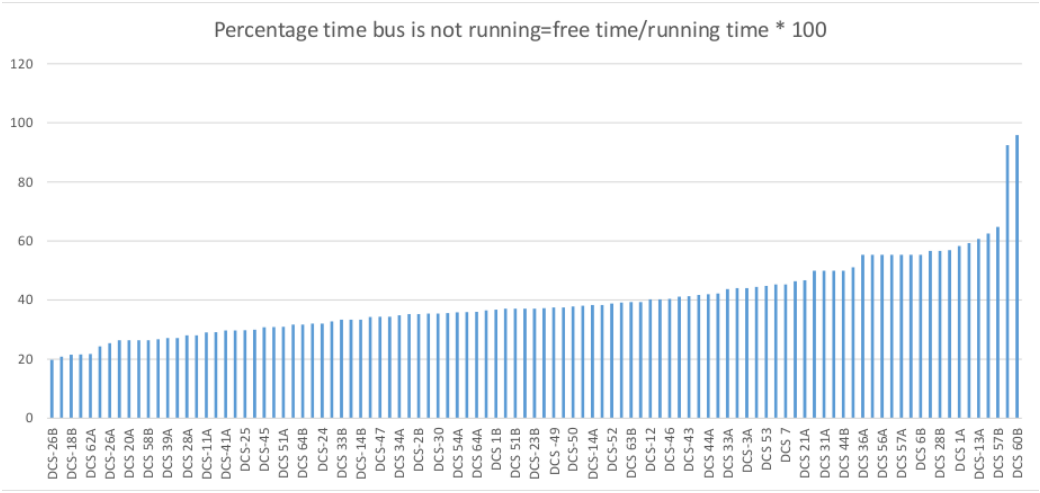

Figure 15: Plot showing the percentage of time bus is staying idle. This time can be utilized for accommodating other trips.

Although we were able to generate a self-sufficient database, there were some drawbacks. We did not have access to the fare data, which is an important factor for analysis purposes. Also, not all routes were tracked as some ran only in very early morning. This model works on static data of the schedule. While this might be useful in rough planning of a trip, it will not account for unexpected delays, which is a more prevalent problem in larger cities, where traffic and congestion is more. In such scenarios, using real time data would be much more useful. But for that, we need GPS to be installed in all buses, or create an app which tracks the positions of passengers all over the city to estimate the position of the bus.

Acknowledgements

The author would like to thank Prof. Milind Sohoni (CSE Department, IIT Bombay) under whose guidance this project was done, Prof. Seshu (Director, IIT Dharwad) for his support and encouragement, Mr. Deepak Jadhav (Depot Manager, Dharwad) for providing the datasets and the students of CSE Department, IIT Dharwad for collecting the data.

References

[1] I. Bogdanovich, *Geo Tracker - GPS tracker*. Available at: https://play.google.com/store/apps/details?id=com.ilyabogdanovich.

[2] Unknown, *ConvertCSV*. Available at: https://www.convertcsv.com.

[3] Google, *Google Maps Roads API*. Available at://developers.google.com/maps/documentation/roads/snap.

[4] Unknown, *MyGeodata Cloud-GIS data warehouse, converter, maps*. Available at: https://mygeodata.cloud/converter/kml-to-geojson.

[5] Google, *Google's polyline encoding algorithm*. Available at: https://developers.google.com/maps/documentation/utilities/polylinealgorithm.