Mirroring of Google Maps

***Abstract*—**

Creating a mirror of Google maps involves managing a map database which consists of city names and and neighbouring cities to each city and distance between neighbouring cities and retrieving information from this database to find the shortest path between two cities. Here a user will input the source city and the destination city,using the suffix tree and indexed value of each city from the graph will be accessed to find out the shortest path between the source and destination cities. And this shortest path will be determined using Dijkstra's algorithm.

I. INTRODUCTION

Shortest Path problems are inevitable in road network applications such as city emergency handling and drive guiding system, in where the optimal routings have to be found. As the traffic condition among a city changes from time to time and there are usually a huge amounts of requests occur at any moment, it needs to quickly find the solution. Therefore, the efficiency of the algorithm is very important.

Some approaches take advantage of preprocessing that compute results before demanding. These results are saved in memory and could be used directly when a new request comes up. This is inapplicable if the devices have limited memory and external storage. This project aims only at investigate the single source shortest path problems and intends to obtain the shortest path between source and destination cities using Dijkstra's shortest path algorithm.

The Dijkstra's shortest path algorithm is the most commonly used to solve the single source shortest path problem today. For a graph G(V, E), where V is the set of vertices and E is

the set of edges, the running time for finding a path between two vertices varies when different data structure are used. Although there are some data structures that may slightly improve the time complexity, such as Fibonacci heap that can purchase time complexity of O(V*log(V)).

II. PROBLEM STATEMENT

Given an adjacency matrix graph which is bidirectional, and the cities which should not be in the path(optional), the algorithm generates the path to be taken and the distance of travel for the same path. All the distances are given in the unit of Kilometers. The adjacency matrix has all its adjacent cities as a linked list, sorted based on the relative order in which they were added to the database.

III. APPROACH

We have used five main(important) functions and many helper functions, those are

1) **int add_city(GRAPH* G,char* city);**

This function should add the 'city' to the TRIE. so it has to update both TRIE and INDEX_ARRAY. And return the index of the city in the INDEX_ARRAY. If the city name already exists, it should just return the index of the city, need not add to TRIE.

Time complexity of this function is **O(m)**.where 'm' is number of characters in the city name.

2) **NODE* insert(GRAPH* G,char *city,NODE* neighbours);**

Insert function will insert the city to the graph. it has to add the neighbours of the city to the graph, neighbours will contain NULL terminated linked list of it's neighbours. Also it has to update the neighbours city to point to this city as well.

Time complexity of insert function is

O(m * nlogn + m*n^2)

where 'm' is number of characters in the city name and 'n' is number of cities in the database.

3) **char* index_to_city(GRAPH* G,int idx);**

This function returns the corresponding index for the given cities. This will make use of TRIE.Time complexity of index_city_function is **O(1).**

4) **int city_to_index(GRAPH* G,char* city);**

This function returns the corresponding index for the given cities. This will make use of TRIE. Time complexity of this function is **O(m)**.

where 'm' is number of characters in the city name.

5) **CITY* shortest_path(GRAPH* G, char* src, char*dst ,NODE *remove);**

This shortest_path function will return the shortest path between the source and destination cities. It should be NULL terminated.the last node dst in the returned list. This function will make use of city_to_index and index_to_city function to map between cities and index. As decided this function has to use dijkstra algorithm. Before we apply dijkstra's algorithm we mask all the cities which are present in the linked list remove, which is NULL terminated. This make sure that the dijkstra's algorithm do not use these cities.

Algorithm:

a)First we need to visit all the cities and mark the distance to be infinity except for the source city.And insert into the queue.

b)Extract the city with minimum distance(for the first time it will be the source city).

c)From the minimum_distanced(extracted) city we need to check the distance need to cover it's neighours ,if that distance if lesser than the current distance then update the distance with smaller one and make it's parent as minimum_distanced(extracted) city.

d)Repeat above two steps until queue is empty(which indicates that there is no path from source to destination) or if the extracted city is destination .

e)After that move to the destination city in the queue and trace back it's parent until it is source city.And keep track of the path.

f)Return the path.

IV. RESULT AND ANALYSIS

The average time taken to build the adjacency matrix is 0.32 seconds. The dataset had 133 cities of India and it's distances from various other parts of India. Each city was connected to around 90 to 100 different other cities. The data was stored in a comma separated values in a file in the format:

city
$neighbour_1$,$distance_1$,
$neighbour_2$,$distance_2$,

….
….
$neighbour_n$, $distance_n$

The average time to generate the shortest path between the two given cities depends on how close the two cities are geographically. But on an average the time taken to compute the path was 0.09 seconds on the same dataset.

V. CONCLUSION

The project helped us in realizing the importance of keeping things ordered and the advantage we might have by doing the same. We also had an opportunity to look into various ways of storing a graph semantically, and eventually decided upon adjacency matrix.

The project gives desirable output for any valid input in reasonable amount of time. The project can be extended to find the shortest k path between the two given cities and through appropriate user interface.