Delhi Technological University

Real Time Navigation with Implementation of Traveling Salesman on Maps

Report File

—

Naman Gupta
2K19/MC/082
(Department of Applied Mathematics)

Index

Introduction

Let's begin by discussing what a Hamiltonian cycle is. It's a cycle or closed loop that covers all the nodes of a graph (or say cities) by visiting exactly once and returning to the start node. **A minimum weighted Hamiltonian cycle is the Traveling Salesman Problem.**

The Traveling Salesman problem asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city ? ".

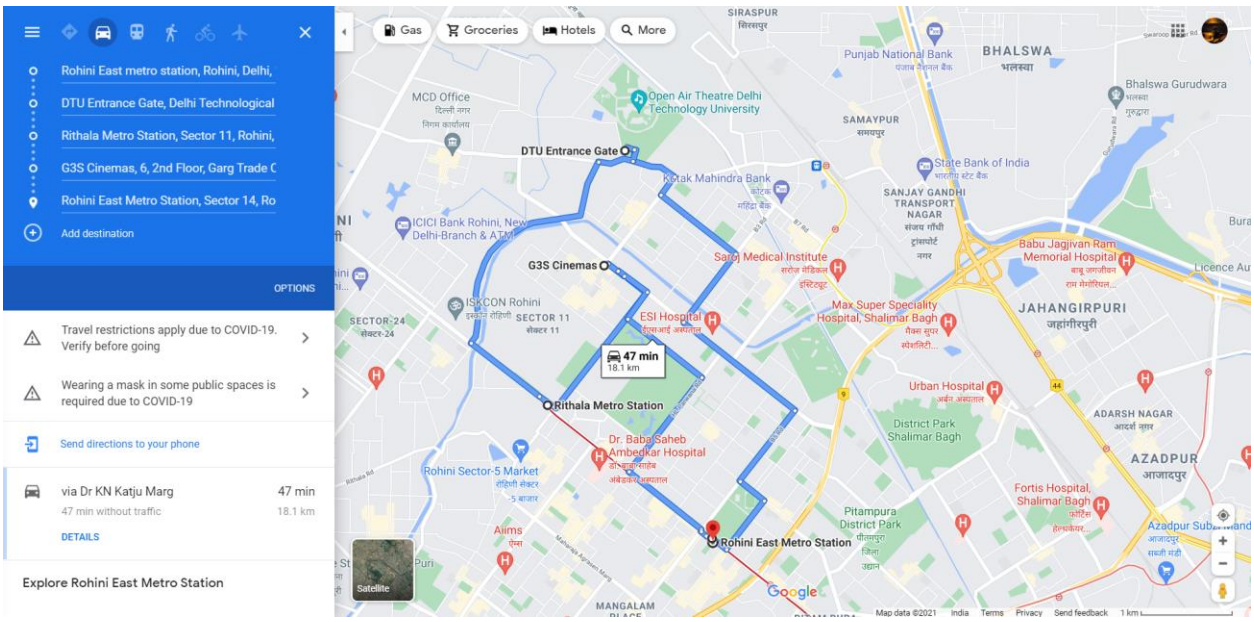
In this project, the underlying algorithm behind the traveling salesman problem has been implemented on real time maps that can be used for navigation and effective commuting, all put together in a website. Using the website, one can plan an effective journey schedule with multiple waypoints or destinations routing effectively back to the start point, thus increasing one's overall productivity. It deals with factors like real time traffic, distance, time, traveling mode and other details thus suggesting the best suitable path.

Here's the link to the site : https://naman-traveling-salesman.netlify.app/

Background

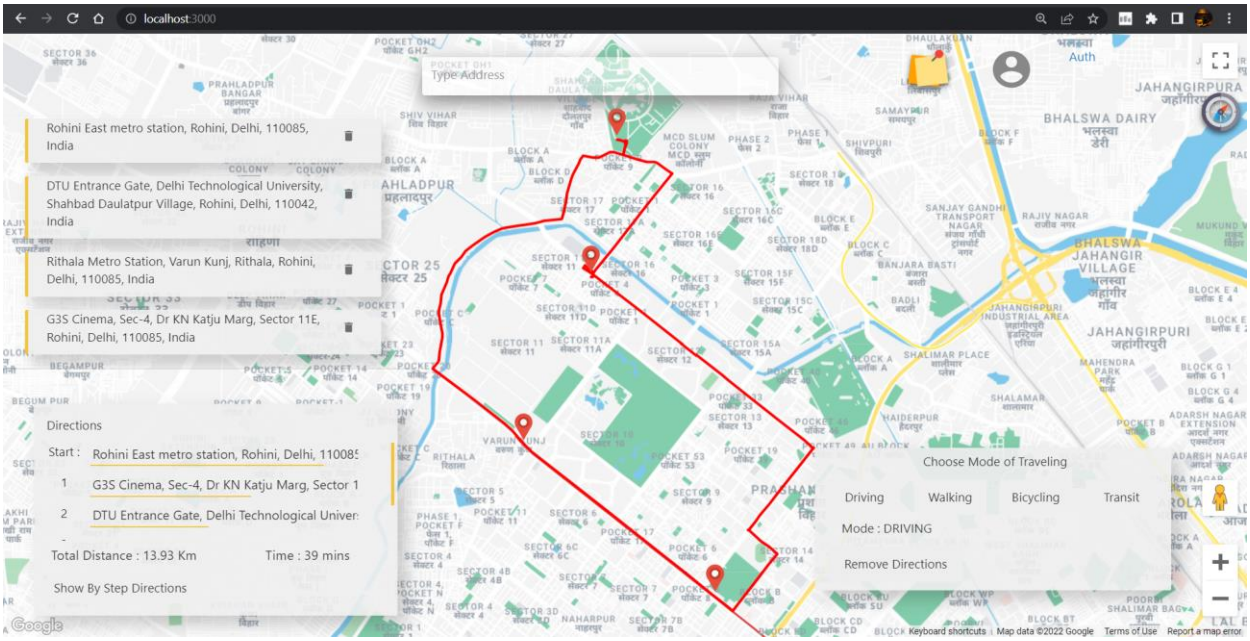
The project aims to develop a website that can produce an effective economical route as per user's marked input points or markers using the concept and algorithm behind the traveling salesman problem.

Now the question arises, why not just use google maps? Google map tends to optimize the route sequentially between two waypoints, not every point, like an example shown below:



Therefore, google maps is not of much use when it comes to efficient multi-waypoint routing. This project intends to cover up the existing deficiency.

Now the solution my website provides:



Clearly, the route provided is optimized, less time consuming as compared to the one by google maps.

Regarding technical aspects, the website has been made using React Js, a JavaScript framework and the following Google Maps APIs have been used:

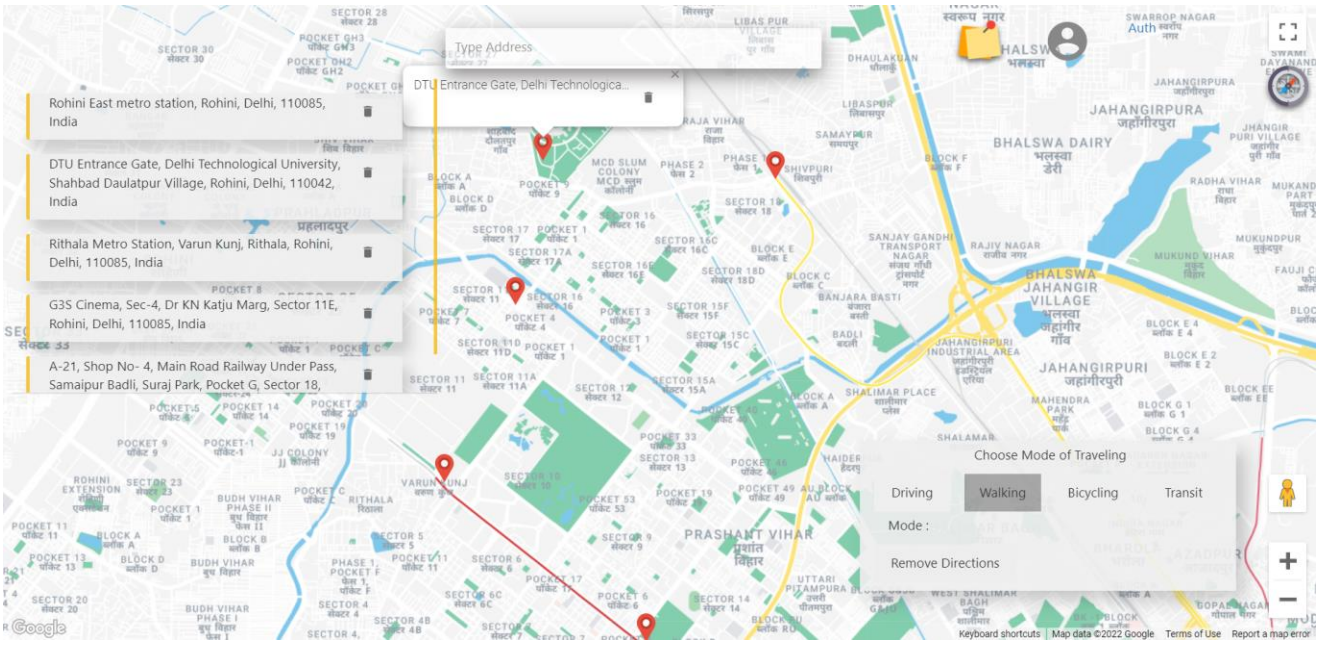
- Maps JavaScript API – An API used to load maps
- Geocoding API – API used to convert latitudinal and longitudinal coordinates to addresses and vice-versa
- Places API – API used to get information about places.
- Directions API – API used to get the directions between two localities.
- Distance Matrix API – An API that responds with the travel times and distances for a matrix of origins and destinations of latitudes and longitudes.

Better and effective commuting is one's need, for saving time. Therefore, instead of taking the usual distance factor into consideration, time is used to determine the fastest or most effective possible route between multiple points, as a result of computations (as per Traveling Salesman Algorithm) done on the distance matrix API response, discussed ahead.

Specifications

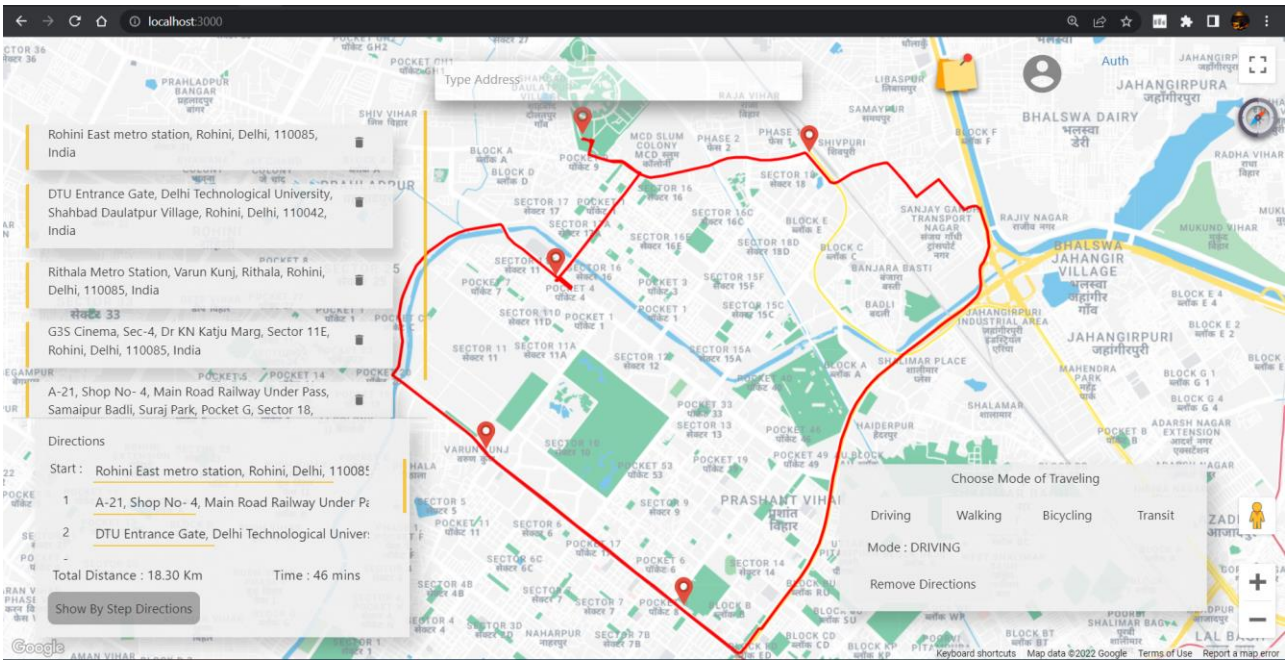
About the website's user interface

The webpage allows a user to mark multiple points, provides an option box with the feature to choose traveling mode

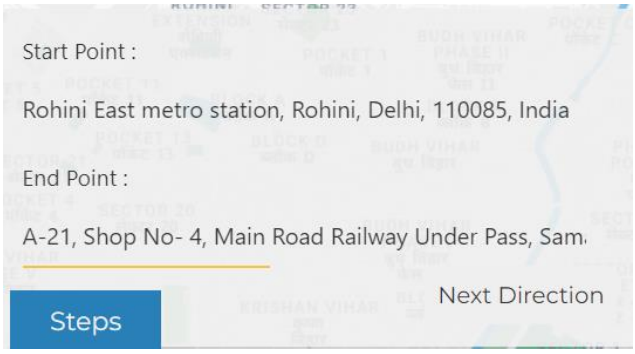


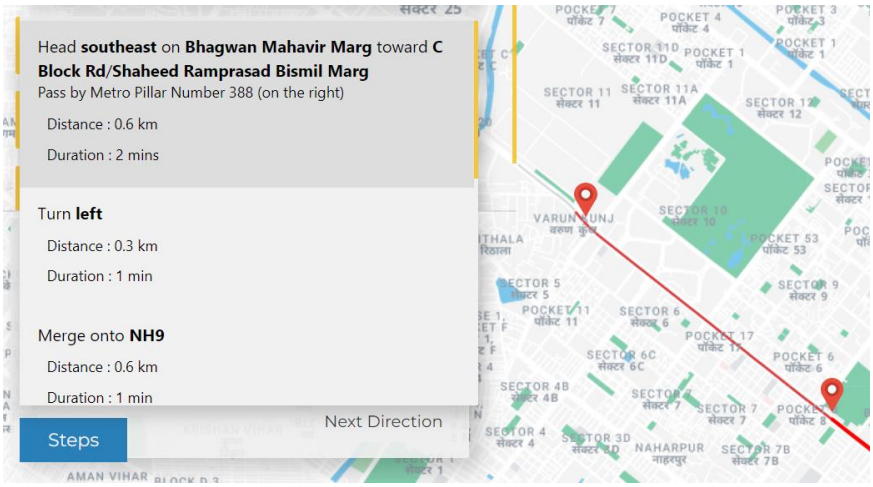
A search box for doing places search. It is implemented using an npm react package known as PlacesAutoComplete which uses the google places API to show suggestions based on user search. Further on suggestion click, map relocates to the selected address, done using the geocoding API.

Upon mode selection, the user will get animated directions on the map, with a directions box stating the waypoints to follow, total time and distance to cover.

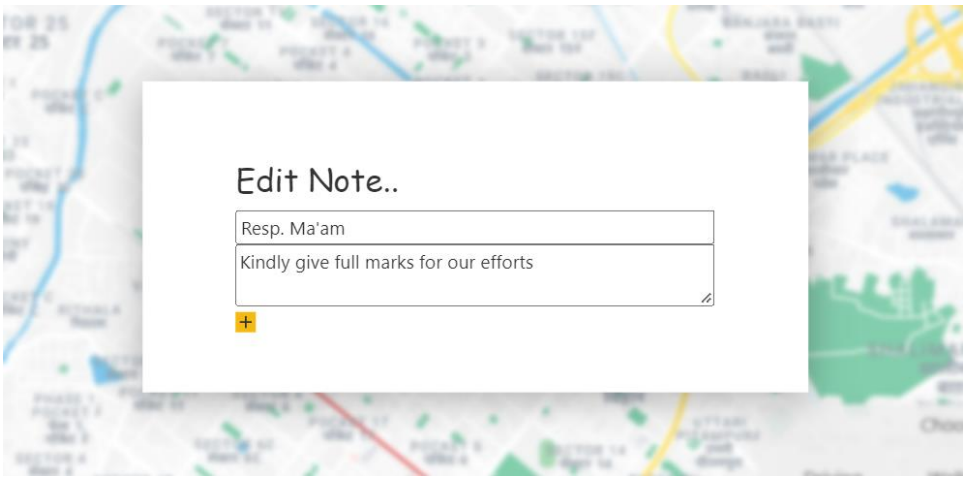


Upon clicking the button “show by step directions”, user gets detailed navigational information for the current sub path.

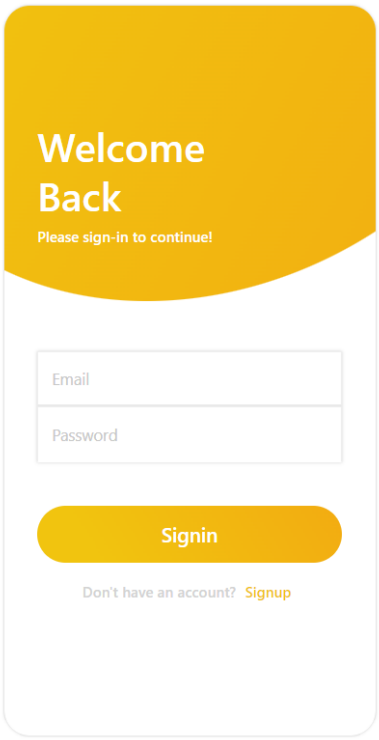


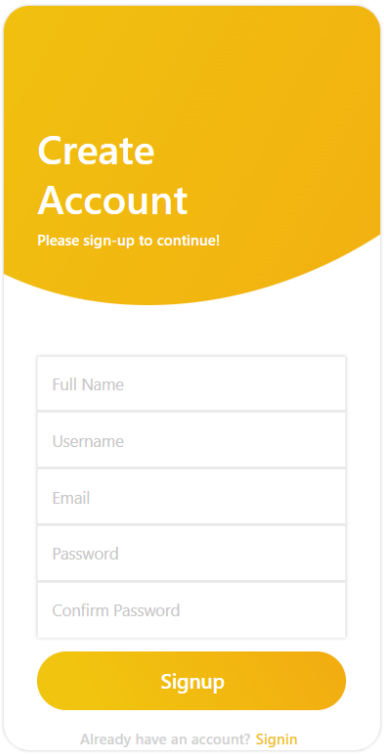


One can create, read, edit or update, and delete notes (all the CRUD operations).



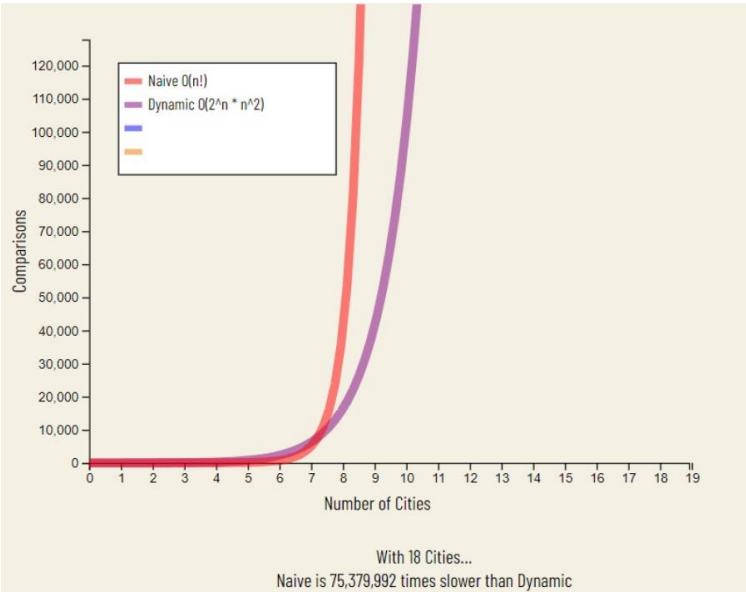
The website also consists of user authentication, in order to store notes, favorite places and history of recently configured routes, using MongoDB as database and mongoose, used to establish connection between MongoDB and the backend framework, express.





Algorithm Design

The diagram shows computational difference between naive and the approach using dynamic programming.

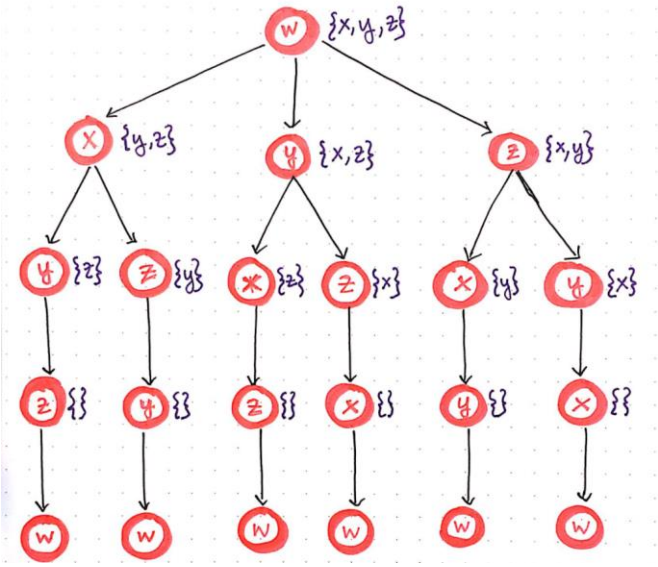


The naive approach compares all possible permutations of routes or paths to determine the shortest unique solution, calculate the

distance (here time) of each permutation and then choose the shortest one which is the optimal solution. This approach works with time complexity of big O(n!) and space complexity O(1)

where n is the total number of cities.

The recurrence tree diagram below represents all the permutations possible for 4 cities named w, x, y and z that has w as it's starting and ending point.



For every recursive call, keep track of minimum cost (distance or time) permutation.

To implement TSP on maps, the best route should also be stored along with the minimum cost, further adding the factor of n (to update the path array at each node) to the time complexity and increasing the space complexity to big O(n!*n) (for each node, the maximum length of the array can be n).

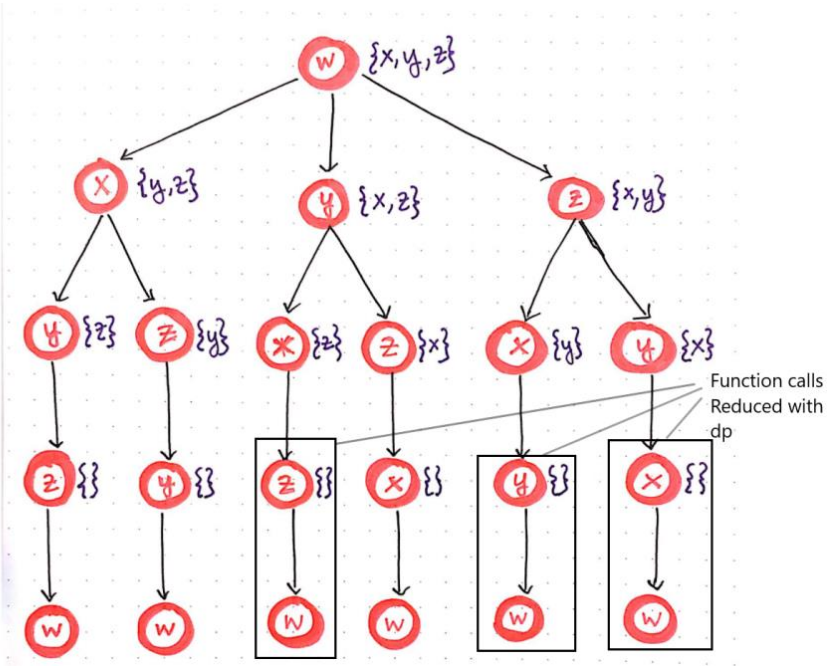
Therefore time complexity : O(n!*n)

Space complexity : O(n!*n)

Dynamic Programming can further optimize the approach by reducing repetitive function calls. There are at most $O(n*2^n)$ subproblems, and each one takes linear time to solve, and linear time to update the array storing path. The total running time is therefore $O(n^3*2^n)$. The time complexity is much less than O(n!), but still exponential. Now the space complexity for storing path at all nodes is $O(n*2^n)$ added to the space required to store the values of repeating calls which is $O(n*2^n)$.

Therefore time complexity : $O(n^3*2^n)$

Space complexity : $O(n*2^n + n*2^n) = O(n*2^n)$

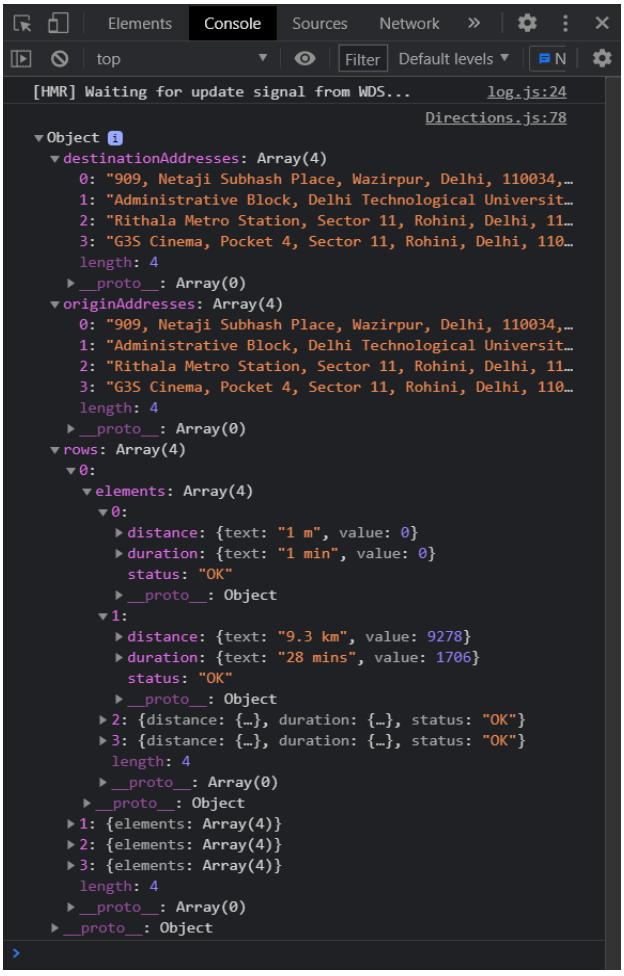


Implementation

All the working has been done with the use of JavaScript framework, React.js. Reason being that React is component-based that makes use of class components as well as functional components that increases the reusability of code, hence increasing flexibility. Large chunks of code can be easily broken down into smaller pieces, that enhances readability and the code seems to be neat and clean.

Coming to the code part, Google's distance matrix API takes an object of matrix of origins and destinations along with mode as request and returns the necessary information.

Here's the data API provides:



The response consists of marked point addresses, duration and distance between every point, which is used to implement the problem and routing.

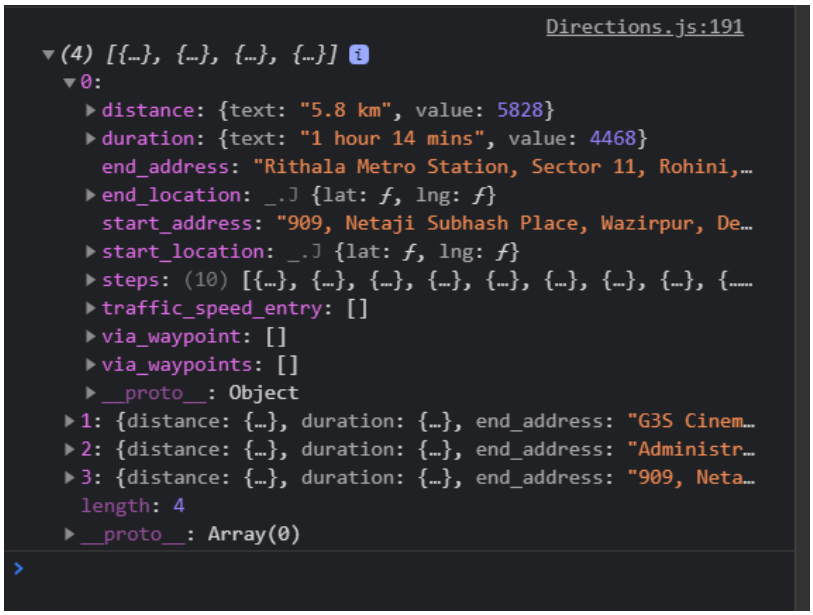
Below is the code implementation for Traveling salesman problem using dynamic programming :

```
let TSP = function(timeMatrix, maskVal, currPos, checkMask){
    if(maskVal === checkMask){
        return {
            currTime : timeMatrix[currPos].elements[0].duration.value,
            currPath : [0]
        };
    }
    if(dp[currPos][maskVal] !== -1){
        return dp[currPos][maskVal];
    }

    let ansTime = Number.MAX_VALUE;
    let ansPath = [currPos];

    for(let i=0;i<timeMatrix.length;i++){
        if((maskVal&(1<<i)) === 0){
            let {currTime, currPath} = TSP(timeMatrix,maskVal|(1<<i),i,checkMask);
            currTime += timeMatrix[currPos].elements[i].duration.value;
            if(currTime < ansTime){
                ansTime = currTime;
                ansPath = [i, ...currPath];
            }
        }
    }
    return dp[currPos][maskVal] = {currTime : ansTime, currPath : ansPath}
}
```

As discussed earlier, function TSP takes the distance time matrix as an argument and returns the minimum time valued route points. The path points are then passed to directions API as a request that responds with the data given below:



The data is then used to provide necessary information on the website and route steps are used to create animated polylines or directions on the map.

Results And Evaluation

Underlying is the result or site link of the project:

https://naman-traveling-salesman.netlify.app/

The platform intends to cover up one of the proportion google maps lack, i.e. efficient multipoint routing, using concept of Traveling Salesman Problem, the issue is resolved using most optimal approach, dynamic programing.

Link for the code work:

https://github.com/N-a-m-a-n/Traveling-Salesman

https://github.com/N-a-m-a-n/TravelingSalesman-Backend

Conclusion

Currently, there is no system that can effectively plan a route for multipoint routing, a platform that can help one save time. Google maps optimize routes but only those with a single start and end point. This project tends to solve the issue. The underlying algorithm behind the traveling salesman problem is used and implemented on real time maps on a website that can be used for navigation and effective commuting. Using the website, one can plan an effective journey schedule with multiple waypoints or destinations. For better results, time is considered as the main factor over distance. For the data part, Google's multiple APIs are used.

Future Work

- Making optimizations to the code, instead of using inbuilt JS functions, implementing binary search for index based searching, increasing overall efficiency.
- Implementation of multiple TSP for enhanced faster delivery system.
- Adding features like forget password (updation) to the system.

References

https://developers.google.com/maps/documentation/javascript/overview

https://developers.google.com/maps/documentation/places/web-service/overview

https://www.npmjs.com/package/react-places-autocomplete

https://en.wikipedia.org/wiki/Travelling_salesman_problem

https://developers.google.com/maps/documentation/directions/overview

https://developers.google.com/maps/documentation/distance-matrix/overview