Travel Route Planner: Application of Travelling Salesman Problem using Brute Force and Heuristic approach with Google Maps API

Eigel Asinas, Jean Pierre Domic Contreras, Marco Noel Aonan, Mark Edison Rosario
Bachelor of Computer Science, Polytechnic University of the Philippines
Sta. Mesa, Manila, Philippines

I. INTRODUCTION AND BACKGROUND

Travelling has always been a part of our lives. It is the action where a person moves from one place to another in order to fulfill a particular task. It can be as simple as a kid going to a nearby store or as complicated as tourists visiting heritage sites, delivery services going to houses, school bus drivers dropping off students to their houses. As travelling becomes more repetitive and complicated, most people would want the fastest route which can save them a sizable amount of time and effort.

To cite an example, a person can easily figure out the fastest route leading to his destination especially if he has in-depth knowledge of roads based on his past experiences. The complication occurs when a person tries to figure out the most efficient route to visit multiple destinations. To solve such optimization problems will require intensive processes.

This type of problem is similar to the popular Travelling Salesman Problem (TSP) as it aims to find the optimal path. The TSP is an optimization problem in graph theory in which the nodes (cities) of a graph are connected by directed edges (routes), where the weight of an edge indicates the distance between two cities. The problem is to find a path that visits each city once, returns to the starting city, and minimizes the distance traveled. [1]

This study aims to develop a Travel Route Planner, an application software featuring an interactive map wherein users can mark multiple destinations. It will try to find the most optimal route from the given origin and list of destinations. It is a tool that can be used for general purposes as long as the problem requires optimization. The optimal route will help users to save a lot of time, effort and resources. It can help boost productivity for work-related scenarios as it identifies the most effective route.

The following scenarios are instances where the optimization tool is applicable:

- Delivery to multiple houses.
- Re-supply of multiple fast food chains.
- Family visiting multiple places.
- School Bus Driver with multiple drop locations.

The application will also have a map interface. Through the use of a web mapping interface, the creation and visualisation of locations and paths will be more convenient. Google Maps is a popular web mapping platform and consumer application offered by Google. The current Google Maps only offers web mapping services and no automatic route optimization features are featured. The Travel Route Planner will utilize the Google Map API to compute for the most efficient path.

II. RELATED WORKS

According to Naher (2011), the Travelling Salesman Problem (TSP) is one of the most famous and most studied problems in combinatorial optimization. It is defined as follows: A travelling salesman has to visit n cities in a round trip (often called tour). He starts in one of the n cities, visits all remaining cities one by one, and finally returns to his starting point. [1]

The actual optimization problem is to find a tour of minimal total length. For this purpose the distances between all pairs of cities are given in a table or matrix. Besides the exact geometric distance values other values may be used, such as travel time or the cost of the required amount of fuel. The goal is to plan the tour in such a way that the total distance, travel time, or the total cost is minimized, respectively. [1]

There are obviously a lot of different routes to choose from, but finding the best one—the one that will require the least distance or cost—is what mathematicians and computer scientists have spent decades trying to solve for.

TSP has commanded so much attention because it's so easy to describe yet so difficult to solve. In fact, TSP belongs to the class of combinatorial optimization problems known as NP-complete. This means that TSP is classified as NP-hard because it has no "quick" solution and the complexity of calculating the best route will increase when you add more destinations to the problem. [5]

The problem can be solved by analyzing every round-trip route to determine the shortest one. However, as the number of destinations increases, the corresponding number of roundtrips surpasses the capabilities of even the fastest computers. With 10 destinations, there can be more than 300,000 roundtrip permutations and combinations. With 15 destinations, the number of possible routes could exceed 87 billion. [5]

According to Abid et al. (2015), TSP can be classified as symmetric, asymmetric and multiple TSP based on the distance and direction between two cities in a graph (Figure 1). If distance between two cities is same in each direction it is symmetric with undirected nature otherwise it is asymmetric. [3]

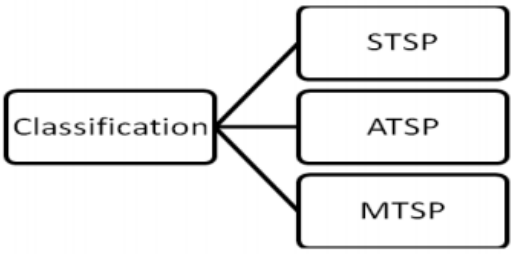

Figure 1: Classification of TSP

In order to understand TSP, let us explore the given example below. Figure 2 shows the road distance between the three towns i.e. ABC. Here the decimal values near the line edges in the diagram are the driving distances between the cities. In this example, we are assuming that we have a symmetric TSP. [3]

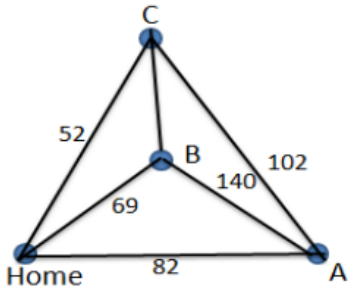

Figure 2: A 4 city sTSP

The best path from the problem above is HACBH where the least total distance is 343.

HACBH → 82 + 102 + 90 + 69 = 343

Thus, going from H to A, then to C, then to B and then back H could be the best choice.

According to Sahalot et al. (2014), on their Comparative Study of different approaches to solve TSP, a TSP can be solved using different types of approaches which have different performances to accuracy and efficiency. Some of the most popular solutions to TSP include: bruteforce, and heuristic.[2]

A. BRUTE FORCE

When one thinks of solving TSP, the first method that might come to mind is a brute-force method. The Brute Force approach, also known as the Naive Approach, calculates and compares all possible permutations of routes or paths to determine the shortest unique solution.[2] The shortest tour is thus the optimal tour. To solve TSP using Brute-force method we can use the following steps:

Algorithm 1: TSP using Brute Force Method

Step 1: calculate the total number of tours (where cities represent the number of nodes).
Step 2: draw and list all the possible tours.
Step 3: calculate the distance of each tour.
Step 4: choose the shortest tour; this is the optimal solution.

Unfortunately, the total number of all possible tours grows extremely fast with increasing numbers of cities. It is easy to see that there are $(n-1)! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots\cdot(n-1)$ different ways to visit n cities by a tour. Each tour has to start in one city (e.g., the first one), then it has to visit all $n-1$ remaining cities in an arbitrary order, and finally return to the first city. However, there are exactly $(n-1)!$ possible orderings or permutations of the remaining $n-1$ cities. [1]

The pseudocode below represents the Brute Force algorithm. [7]

```
get an initial tour; call it T
      best_tour ⇦ T
      best_score ⇦ score(T)
while there are more permutations of T do the following
      generate a new permutation of T
      if score(T) < best_score then
            best_tour ⇦ T
            best_score ⇦ score(T)
print best_tour and best_score
```

B. HEURISTIC: Nearest-Neighbor Algorithm

Unlike Bruteforce, heuristic is an approach to problem-solving that uses a practical method or various shortcuts in order to produce solutions that may not be optimal but are sufficient.

The nearest neighbour heuristic, is a simple approach for solving the travelling salesman problem. To solve TSP with a Nearest Neighbour heuristic we look at all the arcs coming out of the city (node) that have not been visited and choose the next closest city, then return to the starting city when all the other cities are visited. [2]

To solve TSP using Nearest Neighbour Heuristic we can use the following steps:

Algorithm 3: TSP using Nearest Neighbor Heuristic

Step 1: Pick any starting node.
Step 2: Look at all the arcs coming out of the starting node that have not been visited and choose the next closest node.
Step 3: Repeat the process until all the nodes have been visited at least once.
Step 4: Check and see if all nodes are visited. If so return to the starting point which gives us a tour.
Step 5: Draw and write down the tour, and calculate the distance of the tour.

The pseudocode below represents the Repetitive Nearest-Neighbor algorithm.

```
Select an arbitrary node j
Set l = j and W = {1, 2, ..., n} \ {j}
      While W ≠ ∅ do
            Let j ∈ W such that clj = min{cli | j ∈ W}
            Connect l to j and set W = W \ {j} and l = j.
Connect l to the first node to form a cycle.
```

The time complexity of the nearest-neighbor algorithm is $O(n^2)$. It is relatively more efficient than Brute Force which has a time complexity of O(n!).

The Brute Force solution is optimal but inefficient. It is guaranteed to find a solution, but it will take an incredibly long time. On the other hand, the Nearest-Neighbor Algorithm is efficient but not optimal. It may not be optimal but it is still an approximation method which returns an answer that is close to being accurate.

Often, Google Maps is confused with being a route optimization tool. It can definitely be used for planning a route with multiple stops.

The distinction here is that while Google Maps is a tool that can be used to find the shortest route between multiple stops, it was never designed to find the optimal order of those stops in your route. The person planning routes will need to plot addresses in Google Maps, and spend the time manually determining the most efficient order to serve them in. A person needs to look at the map, and drag and drop the addresses into the best order you can come up with.[6]

III. OBJECTIVES

General Objective:

This study aims to develop a web application, Travel Route Planner, that could aid users in planning their route. The application which will find the optimal route from the given locations provided by the users thru the digital map user interface. This will help users in determining the best route to take, saving them a lot of time, effort and resources.

Specific Objectives:

(1) To create a digital map interface where users can add and delete locations, from which a complete directed weighted graph can be derived inorder to generate a distance adjacency matrix.

(2) To implement the Brute force algorithm, which computes for the optimal route of the given graph problem if the number of nodes is less than equal to 10 ($n <= 10$). And, implement the Repetitive Nearest Neighbor algorithm, a Heuristic approach, to approximate the optimal route if the number of nodes is greater than 10 ($n > 10$).

(3) To display the optimal route in the digital map and create a timeline where the distance and time of each travel is presented.

IV. SCOPES AND LIMITATIONS

The Travel Route Planner will be developed as a web application. The application will be built using React JS, an open-source front-end JavaScript library. Most of the application's features will rely on the Google Maps API. The application will only refer to the APIs' internal distance and time information and will not be able to use very accurate and real time information.

The application's digital map interface will primarily cover the National Capital Region (NCR) of the Philippine country, but not in any way will be limited to this area. The route mode is set to driving mode by default, thus travel between waters will not be accepted. The user will be able to add and remove locations. The locations can be added via searching or direct marking. The user can also pre-select the starting and ending location.

The application's subscription type to Google Map API's service is only free, hence there are several limitations to the API usage. The API only allows 10 requests at a time which limits the maximum number of locations to 10. Using a divide and conquer approach, the requests are divided into multiple instances of requests. From a limit of 10 locations, the application can now cater up to 25 locations.

Upon obtaining a better subscription to Google Map API, the previous restrictions will be removed and commercialized use will be possible. The application will be able to fully utilize the algorithm without restrictions, thus providing better performance.

The researchers implemented two algorithms: Brute Force and Heuristic: Repetitive Nearest-Neighbor. If the number of nodes in a given problem is less than equal to 10 ($n <= 10$) then the Brute Force algorithm will be used, otherwise the Heuristic: Repetitive Nearest-Neighbor Algorithm.

V. SYSTEM DESIGN AND METHODOLOGY

The Figure 3 below shows the system architecture used in this study. The whole system will be developed using React technology, one of the popular front end technologies and one of the most powerful javascript libraries.

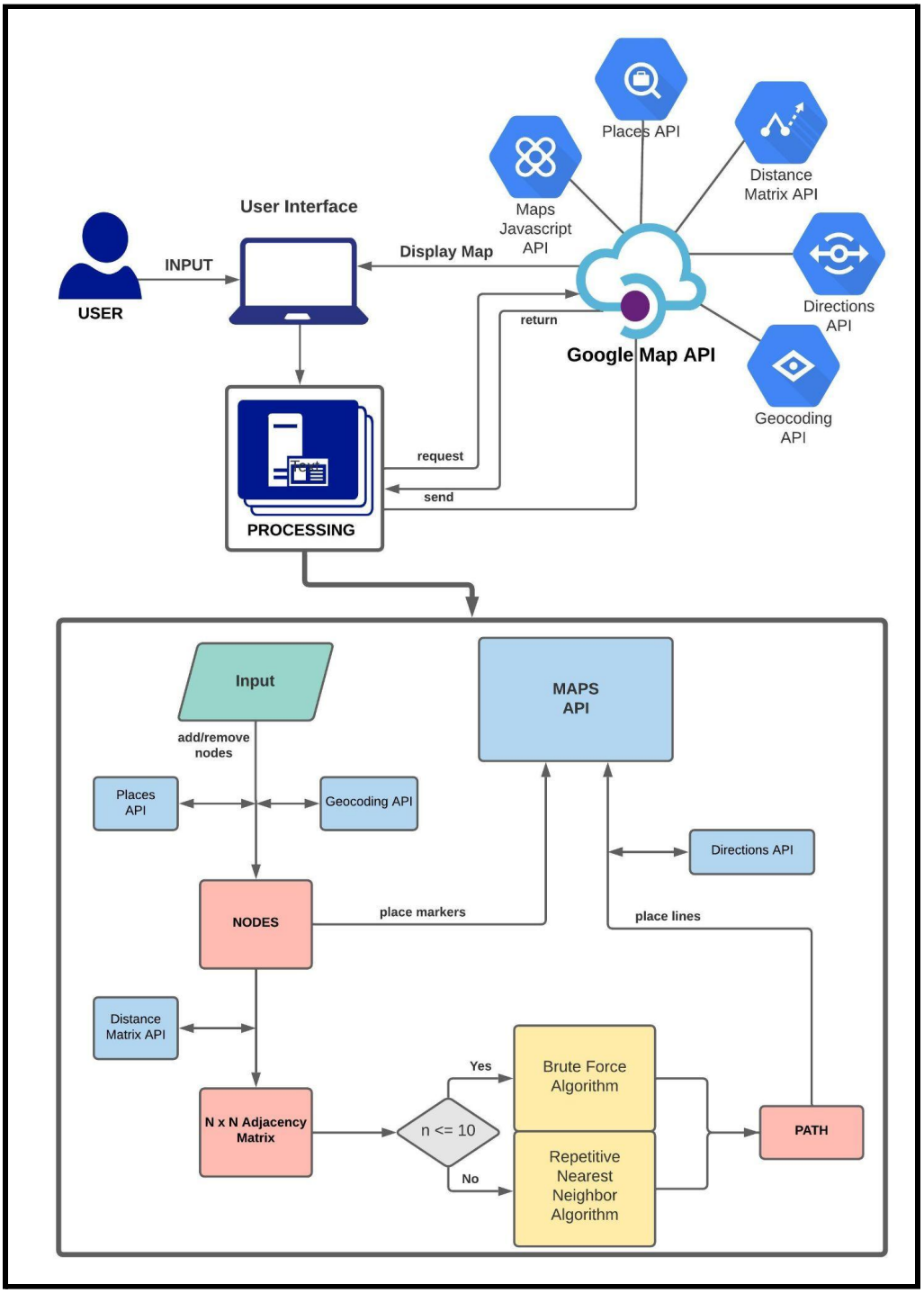

Figure 3: System Architecture

One of the major components of the system is the Google Map API. It provides most of the functionalities of the system. The API is divided into multiple APIs with particular functions.

The following APIs were used in the system.

a. **Maps Javascript API**: Add a map to the website, providing imagery and local data from the same source as Google Maps. Style the map to suit your needs by visualizing the data on the map.
b. **Places API**: Get data from the same database used by Google Maps. Places features over 100 million businesses and points of interest that are updated frequently through owner-verified listings and user-moderated contributions.
c. **Geocoding API**: Convert addresses into geographic coordinates (geocoding), which you can use to place markers or position the map. This API also allows you to convert geographic coordinates into an address (reverse geocoding).
d. **Distance Matrix API:** Access travel distance and time for a matrix of origins and destinations with the Distance Matrix API. The information returned is based on the recommended route between start and end points.
e. **Directions API:** Access driving, cycling, walking and public transportation routing with the Directions API using an HTTP request. Waypoints offer the ability to alter a route through a specific location. Specify origins, destinations and waypoints either as text strings (e.g. "Chicago, IL" or "Darwin, NT, Australia") or as latitude/longitude coordinates.

Following are the major processes involved in the system architecture (Figure 3):

1. **MAP INTERFACE**:

The Google Map API provides the digital map in the user interface from which the user can use. It is accomplished by rendering the Maps Javascript API into the application.

2. **COLLECTION OF INPUT: Add/remove locations**

Using the digital map interface, the user will be able add locations by searching or marking any places. It is accomplished by utilizing the Places and Geocoding API. The Places API provides millions of places which the user can search or locate in the digital map, and the

Geocoding API can convert coordinates into addresses. Using the addresses and coordinates, we are able identify the locations then mark them to the Map Javascript API. The system will then render the map interface with the added markers.

3. DATA STRUCTURE AND PROBLEM:

Using the list of locations, an adjacency matrix can be created to represent the graph. The adjacency matrix contains the distances of each location provided by the Distance API.

The Figure 4 below shows the type of marked locations in the map.

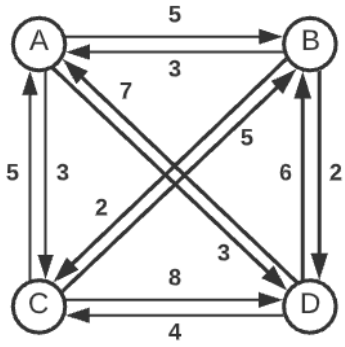

Figure 4: A Complete Directed and Weighted Graph

The adjacency matrix will represent a complete directed and weighted graph. All vertices of the graph are connected to each other making it complete. It is also a directed and weighted graph since every edge has direction and weight. The weight of each edge corresponds to the distance between the two vertices. [8]

The given type of graph can be classified as an Asymmetric Travelling Salesman Problem (aTSP) where the distance between two points is different for each direction. The problem aims to find the most optimal route from the given list of destinations which yields the least total distance. The problem is also further classified into 3 kinds, this is where some parts of the route will be derived from. The application offers the following options:

A. **Start-To-Start**: This is the classic TSP in which the route starts and ends in the same location, thus creating a circular tour. The selected locations will already be positioned in the route.
B. **Start-To-Any**: This is also an optimization problem but unlike the usual TSP, the route will not start and end in the same location. The ending location can be any of

the locations as long as the optimal path is obtained. The selected starting location will already be positioned in the route while the rest will be used for processing.

C. **Start-To-Node** (*Selected*): This is also unlike the usual TSP. In this type of problem, the ending location is already specified by the user. The system will find the most optimal path that ends in the selected location. The selected starting and ending is already positioned in the route.

4. ALGORITHM:

The system implemented two distinct algorithms to solve the route optimization problem: Brute Force and Heuristic: Repetitive Nearest-Neighbor Algorithm.

If the number of nodes in a given problem is less than equal to 10 (n <= 10) then the Brute Force approach will be used, otherwise the Heuristic: Repetitive Nearest-Neighbor Algorithm.

A. **BRUTE FORCE:** It is a simple algorithm that is easy to understand and implement. The brute force method is to simply generate all possible routes and compute their distances. The shortest route is thus the optimal tour.

The algorithm was implemented in the JavaScript Language in order to be integrated with the web application. The code implementation is as follows:

```
export async function getRoute(graph, endPos=0, labels){
    const n = graph.length;
    let currPos = 0;
    let circular = false;
    let any = false;

    let ans = 99999999999;
    const v = graph.map(_ => false);
    v[currPos] = true;

    let smallest_path = [];
    let cur_path = [];
    cur_path[0] = currPos;

    if(endPos === currPos){
        circular = true;
        cur_path[n] = currPos;
    }else if(endPos < currPos){
        any = true;
    }else if(endPos > currPos){
        cur_path[n-1] = endPos;
        v[endPos] = true;
    }

    tsp(graph, currPos, n, 1, 0);
    function tsp(graph, currPos, n, count, cost){
        if(count == n){
            if(circular  && graph[currPos][0]){
                var total = cost + graph[currPos][0];
                if(ans > total){
                    ans = total;
                    smallest_path = cur_path.slice();
                }
                return;
            }else if(any){
                var total = cost;
                console.log(total);
                if(ans > total){
                    ans = total;
                    smallest_path = cur_path.slice();
                }
                return;
            }
        }else if (count == n-1 && graph[currPos][endPos] && endPos
> 0) {
            var total = cost + graph[currPos][endPos];
            console.log(total);
            if(ans > total){
                ans = total;
                smallest_path = cur_path.slice();
            }
            return;
        }

        for (let i = 0; i < n; i++) {
            if (!v[i] && graph[currPos][i]) {
                v[i] = true;
                cur_path[count] = i;
                        tsp(graph, i, n, count + 1, cost +
graph[currPos][i]);

                v[i] = false;
            }
        }
}
return {
     path: labels ? smallest_path.map(index => labels[index]) :
 smallest_path,
    total: ans    }
```

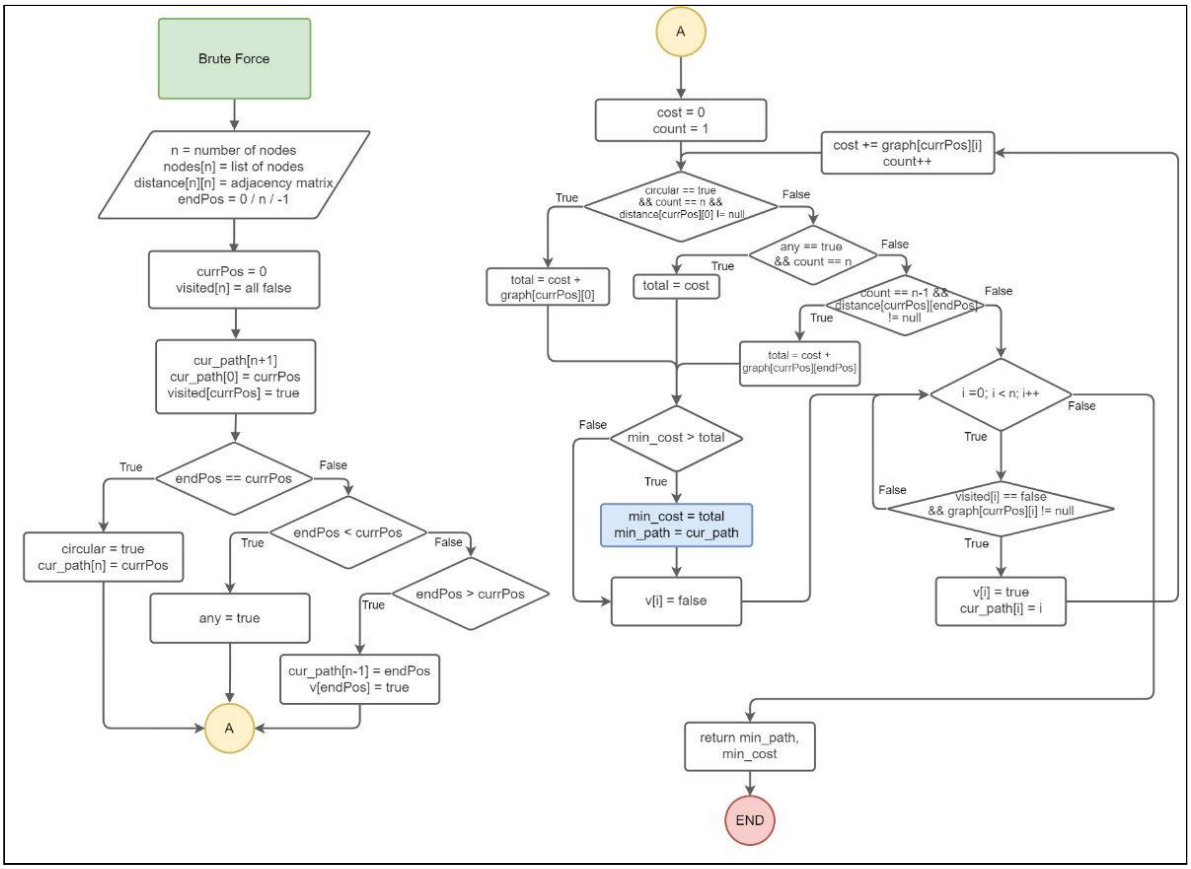

Figure 5: Flowchart of Brute Force Implementation

The time complexity of this algorithm is O(n!). The algorithm uses permutation to draw all the possible paths thus having a factorial time complexity. It is extremely inefficient as it grows incredibly high.

B. **HEURISTIC Repetitive Nearest-Neighbor:** Heuristic approach has different algorithms that are used for approximation. The system implemented the Repetitive Nearest-Neighbor Algorithm (RNNA), a slight modification of the Nearest-Neighbor Algorithm (NNA). In NNA, the path is selected by picking a reference vertex and at each step, walk to the nearest unvisited vertex until all vertices are visited. On the other hand, RNNA performs the NNA from every possible reference vertex, obtaining N different paths then choosing the cheapest one. [2]

The time complexity of the nearest-neighbor algorithm is O(n^2). While the Repetitive NNA grows to O(n^3). The NNA was executed repeatedly on different starting vertices. The enhancement of the NNA to RNNA is to increase the accuracy of the approximation method. Increased efficiency results in decreased optimality, and increased optimality results in decreased efficiency.

To solve TSP using Repetitive Nearest-Neighbour Heuristic we can use the following steps:

1. Pick a starting node.
2. Look at all nodes connected to the starting node that have not been visited and choose the next closest node.
3. Repeat the process until all the nodes have been visited at least once.
4. Check and see if all nodes are visited. If so, return to the starting point which gives us a route.
5. Draw and write down the route, and calculate the distance of the route.
6. Repeat the previous steps until all nodes are used as starting nodes.
7. Choose the shortest route; this is the approximate solution.

The algorithm was implemented in the JavaScript Language in order to be integrated with the web application. The code implementation is as follows:

```
export async function getRouteApprox(graph, endPos=0,
    labels){
    let n = graph.length;
    let currPos = 0;

    let k = (currPos < endPos) ? n - 3 : n - 2 ;

    let smallest_path = []
    let min_ans = 999999999999999;

    for(let z = 1; z < n; z++){
       if( (endPos <= currPos) || (endPos > currPos &&
    z !== endPos) ){
        find_path(z);
      }
    }
    function find_path(start){
      let node_count = 2;
      let t_cost = 0;
      let v = Array(n).fill(false);
      v[currPos] = true;
      v[start] = true;

      if(endPos > currPos){
          v[endPos] = true;
      }

      let cur_path = []
      cur_path[currPos] = currPos;
      cur_path[1] = start;

      t_cost += graph[currPos][start];
      for(let i = 1; i <= k; i++){
          let cost = 999999;
          let index = 0;
          for(let j = 0; j < n; j++){
            if(!v[j] && cost > graph[start][j]){
              cost = graph[start][j];
              index = j;
            }
          }
          start = index;
          v[start] = true;
          t_cost += cost;
          cur_path[node_count++] = start;

          if( i === k && endPos === currPos ){
            cur_path[node_count] = 0;
            t_cost += graph[start][0];
          }else if(i === k && endPos > currPos){
            cur_path[node_count] = endPos;
            t_cost += graph[start][endPos];
          }
      }

      if(min_ans > t_cost){
          min_ans = t_cost;
          smallest_path = cur_path.slice();
      }
    }

    return {
           path: labels ? smallest_path.map(index =>
    labels[index]) : smallest_path,
        total: min_ans
    }

}
```

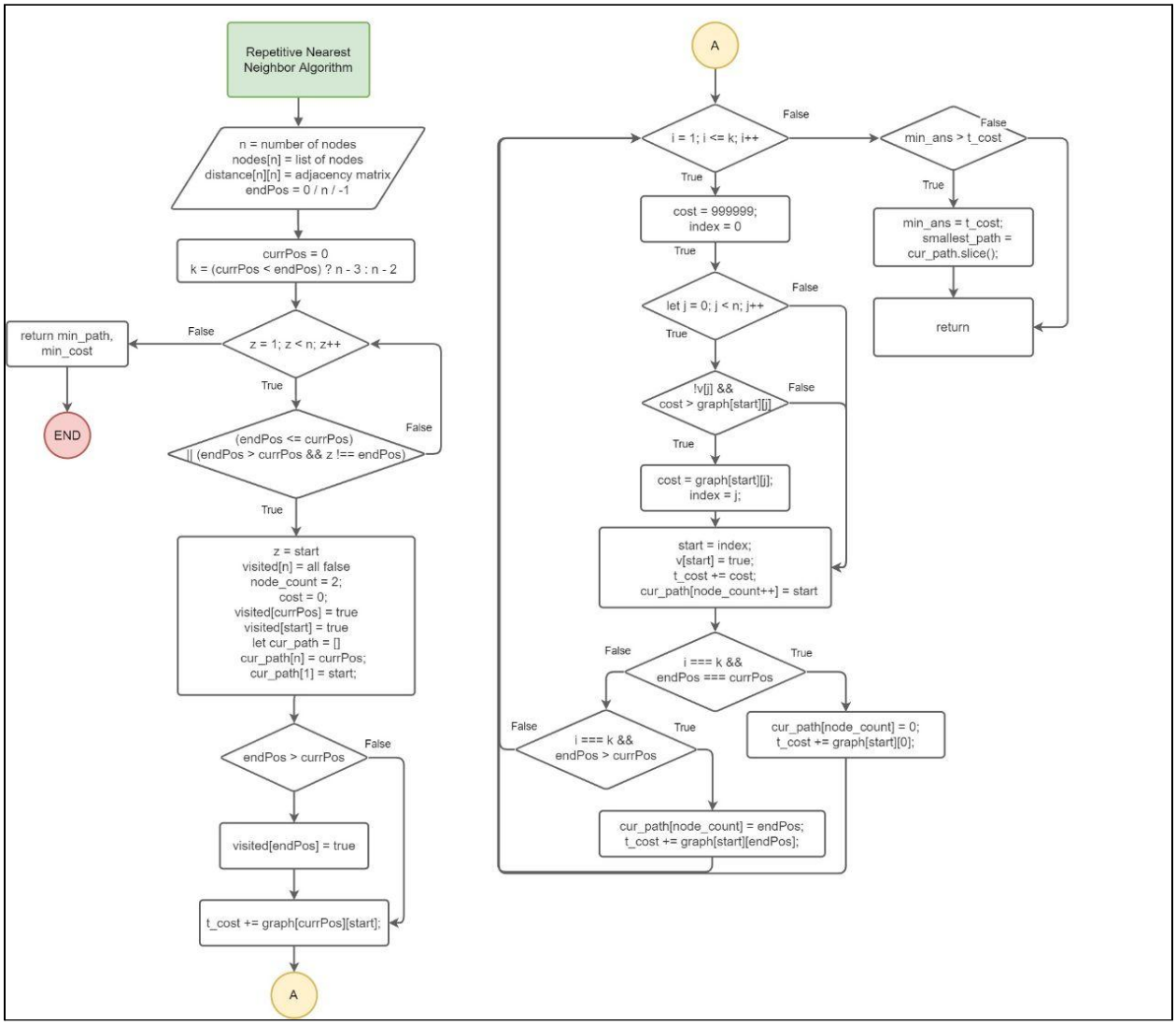

Figure 6: Flowchart of RNNA Implementation

5. ROUTE AND TIMELINE:

A panel containing the timeline of the route will be displayed. It will contain the total distance and time of the whole route. The digital map will be rendered with the added markers and line, representing the obtained route. It is accomplished by utilizing the Directions API. It returns the path between two locations according to the transportation mode. Each location will be connected one-by-one by drawing a path in the Map Javascript API.

The user will be able to view the generated route and its corresponding information from the timeline.

VI. SCHEDULE OF ACTIVITIES

The Figure below shows the schedule of tasks using a GANTT chart. It starts in the first week of June and is expected to be completed in the month of July. It also represents the Linear Responsibility Chart of the study. A linear responsibility chart or LRC, describes the participation by various roles in completing tasks for a project.

Legends:

- P1: Mark Edison Rosario
- P2: Jean Pierre Dominic Contreras
- P3: Eigel Asinas
- P4: Marco Noel Aoanan

- R: Responsible
- A: Accountable
- C: Consulted
- I: Informed

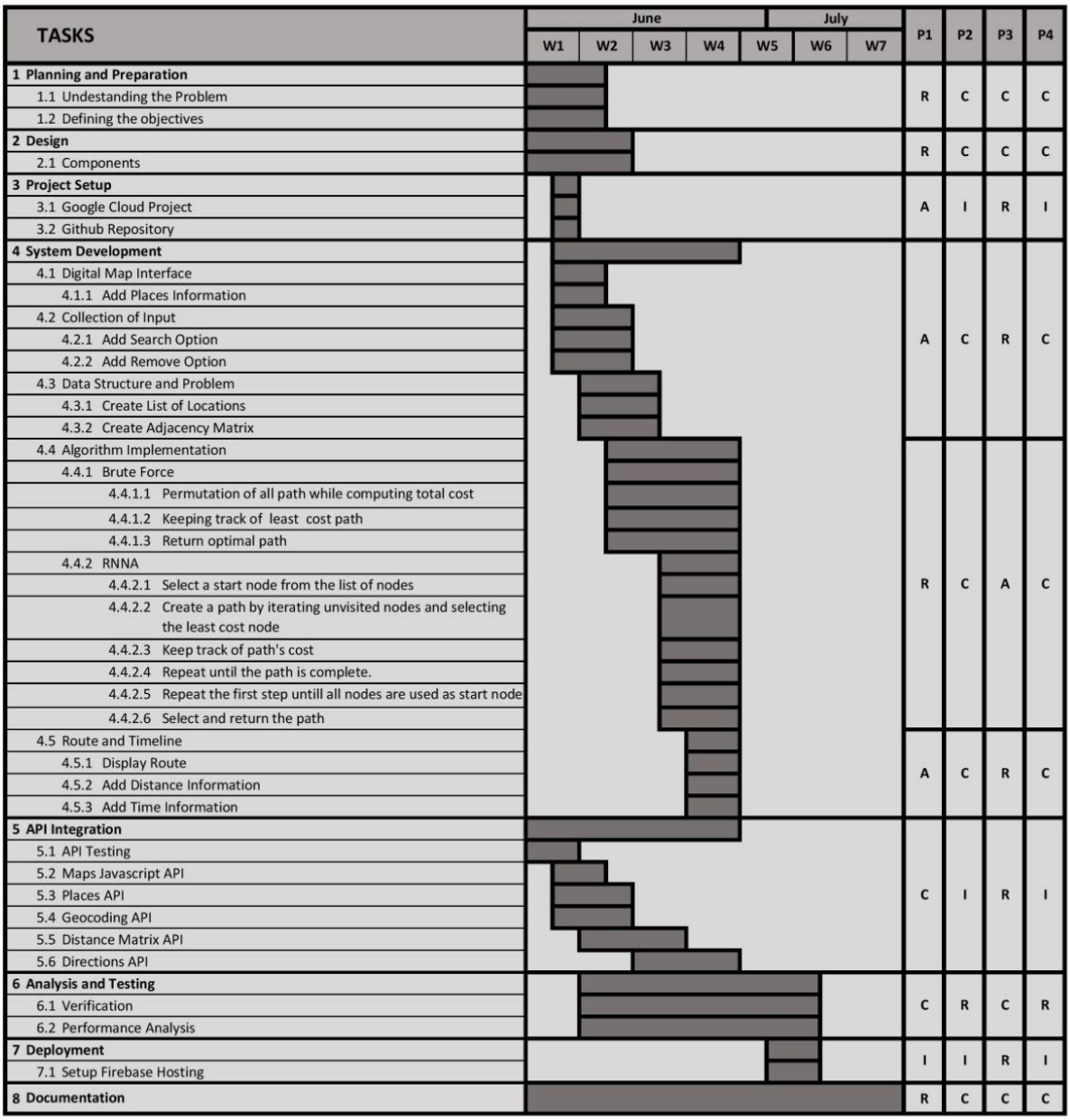

Figure 7: GANTT Chart and LRC

VII. REFERENCES

[1] Näher S. (2011) The Travelling Salesman Problem. In: Vöcking B. et al. (eds) Algorithms Unplugged. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-15328-0_40

[2] Sahalot, A., & Shrimali, S. (2014). A comparative study of brute force method, nearest neighbour and greedy algorithms to solve the travelling salesman problem. International Journal of Research in Engineering & Technology, 2(6), 59-72.

[3] Abid, M. M., & Muhammad, I. (2015). Heuristic approaches to solve traveling salesman problems. TELKOMNIKA Indonesian Journal of Electrical Engineering, 15(2), 390-396

[4] https://jlmartin.ku.edu/~jlmartin/courses/math105-F14/chapter6-part5.pdf

[5] Suzanne Ma, January 2020, Understanding The Travelling Salesman Problem (TSP), https://blog.routific.com/travelling-salesman-problem

[6] David Williams, March 2021, ,Route Optimization And Planning With Google Maps, https://blog.routific.com/route-optimization-google-maps-313a45e13d27

[7] Case Study: Solving the Traveling Salesman Problem https://www2.cs.sfu.ca/CourseCentral/125/tjd/tsp_example.html

[8] https://jlmartin.ku.edu/courses/math105-F11/Lectures/chapter6-part3.pdf