KINGS MAPS

“Real world” application



2022 A-Level Computer Science

By Oscar Pearce, 8196

Analysis:

A look into the travelling salesman problem

The travelling salesman problem (TSP) has been a classic problem for mathematicians and computer scientist since the 1930s. While there exists no efficient algorithm to solve real world networks, a heuristic approach leads to a solution in most cases. This problem involves finding a path that visits any finite number of locations at least once, before returning to the starting point. There exist two variations of the TSP, classical and practical. The classical approach does not allow for locations to be visited more than once, where the practical allows multiple visits to a single location.

Hailing from pathfinding and optimisation, the TSP has obvious real-world parallels. One such example would be, believe it or not, a travelling salesman. The example that I am using for my project is an infamous postman working for in the Wedmore area, Lee Gadd. Applying the TSP to this client will reduce the overall time spend driving, as the total distance travelled will be minimised. This will not only save time, but money for the client’s employer (I’m sure he needs it).

Interview:

* What software is currently in use for delivery?

“There is currently no software used, we receive a list of addresses and have to improvise the route that we take. Honestly this can get a bit overwhelming on a busy day!”

* What software feature would you find most useful?

“An option to input my entire list of addresses for the day, so I would not have to worry about keeping track of where I’ve been.”

* How easy is it to visualise the routes you are taking?

“Well, I’ve been in this job for 20 years so I have developed quite the mind for visualising my routes. However, sometimes I experience a lapse in my judgement and miss a turning or two!”

* How often do you run the same, or very similar, route in a week?

“Often we end up running the same route 12 times a week, 2 times a day for 6 days a week. It would be helpful to save a route for later use.”

* Do you consider the routes you take efficient?

“Yes. However sometimes I find myself backtracking, but not very often.”

An implementation of the TSP for this problem would ensure that the client takes only the most efficient route, reducing wasted fuel and saving the client time. My solution would also allow the client to save previously calculated routes as well as load them for repeated use. In order to abstract real world locations and roads to a model that can be computed, data needs to be extracted from the real world. This will be done through Google Maps distance matrix API, which takes an array of starting latitudes and longitude, returning the distance between each combination.

Locations in the real world will be abstracted to “nodes” with an associated latitude and longitude, so that they can be displayed on a map once calculation is complete. Roads in the real world are abstracted to an integer value, an “arc”, that can represent the time taken to travel along it or the distance to travel along it from start to end. Connections between nodes, arcs, can be represented through an adjacency matrix. This is a dictionary type structure that lists all a node’s connections, arcs, and the associated weight. An adjacency matrix defines a network, or graph. The TSP algorithm that I will implement involves performing many algorithms on a graph to find upper and lower bounds, and testing if they are optimal.

My aims for this project are:

* Produce an algorithm to solve the TSP for any adjacency matrix
  + Users must select a starting node
  + Must be fast
* Produce an adjacency matrix representative of a set of real-world points
  + Users will be able to click on an interactive map to add nodes
  + Users will be able to remove and change the names of nodes
  + Interacting with the map and nodes must be straight forward and feel smooth
* Produce an intuitive and responsive UI
  + Must be easy to use
  + Must be aesthetically pleasing
  + Animations are something to think about
* Save solutions for repeated use
  + These will be saved to a local JSON file that can be read by the program
  + Allow previous solutions to be shown on a map

A graph will be represented in my project as a class, with various methods that will perform an algorithm on the graph. Arcs will also be represented as a class, with attributes detailing the nodes connected by it, and an attribute defining its weight. A graph can have many arcs, however, is not dependent on them to be classed as a graph. The general process of creating a new solution will be as follows:

* User selects “new solution”
* User clicks on map a desired number of times, adding a node each time
* Latitude and Longitude of where the user clicked is stored
* User selects start node
* User selects “calculate”
* Google maps distance API queried with Lat/Lng of nodes
* Adjacency matrix created with information from distance API
* Adjacency matrix sent to back-end TSP solver
* Sequence of nodes to visit sent back to front-end
* Sequence displayed on map

Design:

The following diagram shows the data flow in the system, from user clicking to a finished solution:

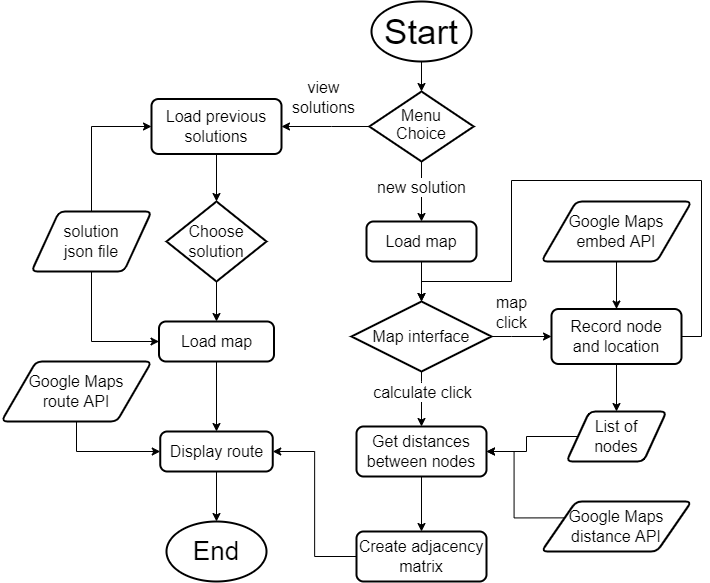
## 

There will be three main areas in the application that the user can interact with: the landing page, the new solution page, and the previous solutions page. The following diagram shows the relationship between the three:

Graphical user interface, text, application, chat or text message

Description automatically generated

System flowchart indicating major stages in the flow of control:



IPSO chart of system inputs / outputs:

|  |  |  |  |
| --- | --- | --- | --- |
| **Input** | **Process** | **Storage** | **Output** |
| Map click | * Get lat/lng from google maps embed API * Display node in node menu and map | Node object/class | Marker on map representing node |
| Dropdown menu | * Select start node | React state variable | Selection on dropdown menu |
| Calculate button | * Create adjacency matrix out of nodes * Fill distances between nodes using distance matrix api * Save matrix to json file * Execute TSP solver * Read json file | * Network class * Json files | Canvas drawing of optimal network |
| Save route button | * Write current route to json file * Redirect to previous solution page * Display route | * Json file | Optimal route displayed on map |
| Select route button | * Configure list of waypoints and starting lat/lngs * Get route data from google directions API * Display route on map | Route class | Selected route displayed on map |

User Interface:

|  |  |  |
| --- | --- | --- |
| Data item | Data type | Restrictions |
| Node name | string | * No special characters * Must be unique to route |
| Starting node | As above | * Selected from dropdown |
| Route name | String | * No special characters * Must be unique |
| Current route | Object | * Selected from list of routes |

Mock up of the landing page UI:

Graphical user interface, application

Description automatically generated

Mock up of the “new solution” page:

Text

Description automatically generated

Mock up of the “previous solutions” page:

Text

Description automatically generated

The Google Maps distance matrix api takes an array of lat/lng objects (Origins) and returns the shortest distance between each pairing between another array of lat/lng objects (Destinations). This is the equivalent of populating a complete graph with connections. However, upon researching how this API works, I had discovered that N-1 pairs are repeated. It is because of being frugal that I had decided to implement an algorithm that removes the need to repeat these pairings, by iterating over each node and only requesting the remaining connections. By default, this would be N^2. This is the pseudocode for said algorithm:

All Nodes = [N1, N2, N3,…,Nn]

All Arcs = []

For node in all nodes:

Send request(

Origin: node

Destinations: [all nodes after node]

) then (

For connection in results:

Append connection to All Arcs

)

This algorithm will reduce the program from calling the API with N^2 nodes to (N^2-N)/2 nodes, as one less node is queried each time. While this is the same order as before, it will result in significantly less requests to the Google Maps distance matrix api, saving me money.

Once the graph created from the connections provided by the Google Maps distance matrix api has been loaded, the Floyd-Warshall algorithm will be used to find the shortest distance between any two nodes:

For node in all nodes:

For row in all nodes:

For column in all nodes:

If row != column:

newDistance = distanceTable[node][column] + distanceTable[row][node]

If newDistance < distanceTable[row][column]:

distanceTable[row][column] = newDistance

routeTable[row][column] = routeTable[row][node]

While this algorithm is N^3, it is the best option going forward as alternatives (Dijkstra’s) will require the algorithm to be repeated multiple times throughout the program. The optimal distanceTable and routeTable will be saved to the Graph Class for later use when traversing the graph.

An algorithm will be required to traverse the graph once Floyd’s algorithm has been completed. This is not as simple as iterating over the route table, as each connection defined in the route table can often be split into shorter routes also defined in the route table. This is a mock up of a recursive algorithm, pathBetweenNodes(startNode, endNode), that I have designed to get the shortest sequence of nodes between startNode and endNode, based on floyd’s algorithm being completed:

If floyd’s not complete:

Complete floyd’s

new path initialPath(start = startNode, end = endNode)

initialPath.sequence.insert(startNode, endNode)

If areAdjacent(startNode, endNode):

Return initialPath

Else:

nextNode = routeTable[startNode][endNode]

new path subPath1 = pathBetweenNodes(startNode, nextNode)

new path subPath2 = pathBetweenNodes(nextNode, endNode)

initialPath.splice(subPath1)

initialPath.splice(subPath2)

return initialPath

The method of creating an upper bound will use a modified version of the nearest neighbour algorithm, that first removes a node before finding the MST. Once the MST (Minimum Spanning Tree) is found, the removed node is then added back in via the shortest path. The following pseudocode represents the process:

Initialise list of graphs BoundGraphs

For node in all nodes:

Initialise new graph NNGraph

Initialise list of nodes BannedNodes

NNGraph.addNode(node)

BannedNodes.addNode(node)

currentNode = node

While size of BannedNodes != size of all nodes:

Initialise list of arcs possibleArcs

For adjacentNode in adjancencyMatrix[currentNode]:

If adjacentNode not in BannedNodes:

arcWeight = adjacencyMatrix[currentNode][adjacentNode]

possibleArcs.addArc(arcWeight, currentNode, adjacentNode)

sort possibleArcs according to arcWeight

currentNode = possibleArcs[0].node2

BannedNodes.addNode(currentNode)

NNGraph.addNode(currentNode)

NNGraph.addArc(possibleArcs[0])

returnArc = new arc PathBetweenNodes(currentNode, node)

BoundGraphs.addGraph(NNGraph)

Sort BoundGraphs according to total weight

Return BoundGraphs[01]

Testing:

The entire system consists of three main components:

* Backend executable that will read a json file as input, and write a sequence of nodes as output
* Frontend interface that will display the map and take input directly from the user
* Middleware that will read / write json files, acting as an API

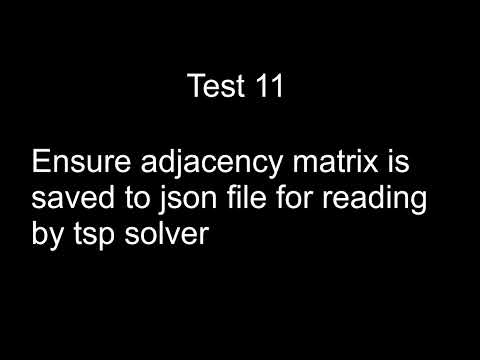
Each aspect of the system will be tested:

* Tests 001 through 004 show the function of the map and are intended to show the proper recording of point data. This shows the function of the BetterMap module and successful communication between the NewSol module.
* Tests 005 through 007 show the function of the route options and the proper recording of these options for later use.
* Tests 008 and 009 show the appearance of a pop-up upon a valid selection of the “calculate” button. This shows the function of the NetworkDisplay module and the interface between it and the NewSol module.
* Tests 010 and 011 show the completeness of the adjacency matrix using the Google maps distance matrix API, as well as communication to the back-end server where the abstraction takes place. This shows successful parsing of data between front and back end.
* Tests 012 to 014 show the successful calculation of an optimal sequence of nodes to visit from the C++ TSP solving algorithm. I admit that I should have shown more testing with the C++ side here, however you can see the output from the algorithm as it runs, showing some level of module testing,
* Test 015 shows the communication from the C++ algorithm to the back-end server using a JSON file, as well as parsing the node sequence back down to the front-end.
* Test 016 shows proper input sanitation from the user in a stylish manner.
* Tests 017 through 020 show proper communication between front-end, back-end, and a JSON file to store previous solutions. This includes error handling and clean-up.

## Test plan:

|  |  |  |  |
| --- | --- | --- | --- |
| **Test ID** | **Description** | **Data** | **Expected results** |
| 001 | Check that latitudes and longitudes of map clicks are recorded and can be accessed | * Click from map * Latitude * Longitude | React state variable with node name, latitude and longitude |
| 002 | Check nodes are properly removed on user input | * Click of button * Node name | Node removed from node menu, as well as map. Node is node included in solution |
| 003 | Check nodes are renamed and that the new name is consistent across node menu and map | * Click of button * Previous node name * “” * “test!\*” * “test” * “\*&((“ * “………..” * Some node name that has already been used | Node name is replaced with the user’s input, stripped of whitespace and disallowed characters – disallowed characters, null input, and non-unique node names should all display an error message |
| 004 | Check starting node is consistent with nodes on map | * Click on dropdown menu | Start node starts as a placeholder message, will hold value of selected node, returns to placeholder message when node from map is deleted or renamed – nodes on map must be consisted with nodes in dropdown |
| 005 | Route options are recorded according to selection | * Click on option buttons | User clicks are recorded using a react state variable to represent the route options |
| 006 | Ensure only one of each type of option can be selected at once | * Clicks on option buttons | Whichever option was clicked on remains highlighted, one per type of option |
| 007 | Ensure traffic options can only be selected when driving is the transit method | * Clicks on option buttons | Traffic options are greyed out and cannot be clicked on when any transit option other than driving is selected |
| 008 | Ensure calculation only available when more than 2 nodes are present and start node is selected | * Less than 2 nodes, no start node * More than 2 nodes, no start node * Less than 2 nodes, start node * More than 2 nodes, start node | The calculate button can only be clicked when more than two nodes are selected as well as a start node, otherwise the step (s) needed to make calculation are displayed to the user |
| 009 | Ensure modal is displayed when calculation button is clicked with more than 2 nodes and a start node selected | * More than 2 nodes * Start node | Modal pops up with either loading message or the optimal network |
| 010 | Ensure Adjacency matrix is filled up using google distance matrix service on valid calculation button click | * Arbitrary amount of nodes, each with latitudes and longitudes | 2 dimensional dictionary representing a complete adjacency matrix for the network |
| 011 | Ensure adjacency matrix is saved to a json file so that it can be read later | * Adjacency matrix | Successful saving of adjacency matrix, display error message otherwise |
| 012 | Confirm C++ solver returns a sequence of nodes representing optimal solution | * Minimum adjacency matrix (3 x 3) * Maximum adjacency matrix (10 x 10) | Sequence of nodes that represents optimal solution, must be checked by solving the network manually using upper and lower bounds |
| 013 | Confirm C++ solver writes node sequence to same file it read from | * As above | Path attribute added to data.json with node sequence |
| 014 | Confirm that after C++ solver executes, the json file is read by internal server | * Sequence of nodes * Minimum: [“”, “”, “”] * Maximum: [“”, “”, …, “”] x 10 | Array of strings is read by internal server and sent back using api |
| 015 | Ensure optimal network is displayed once sequence of nodes is returned | * As above | Drawing using js canvas with connections between nodes in order of the optimal route |
| 016 | Confirm network name is unique and contains no special characters | * User input * Previously used names * “” * “test” * “\*&(“ * “test\*&” | Error message displayed until name is valid. Trailing and leading whitespace is stripped, empty string is not allowed |
| 017 | Confirm network data is saved to prevData.json with user input name, redirect to page where route will be shown | * User input as above * Minimum and maximum networks * Adjacency matrix (3x3 through 10x10) * Node information (min 3, max 10) * Start Node information (required) | Addition to the solutions array in prevData.json, with all structure and attributes preserved. If saving fails, an error message should be shown. If saving succeeds, user should be redirected to another page to display the route |
| 018 | Confirm network renames are unique and contain no special characters | * Previous route name * “” * “test” * “&(\*&” * “test”\*(&” | Name of route is changed in current instance as well as prevData.json, if the route is currently selected then the title of the map must also change. Leading whitespace, trailing whitespace, and special characters must be removed |
| 019 | Confirm route is selected and loaded onto map on button press | * Button press | Previously selected route (if exists) is removed from map, and user selected route is loaded onto map |
| 020 | Confirm route is deleted on button press | * Button press | Route is deleted from prevData.json, if it is currently selected then the map must be cleared |

# Tests:

[](https://www.youtube.com/embed/ZAyXyuUwgC4?feature=oembed)

Evaluation:

In conclusion, I believe my project to effectively meet the user’s needs for a fast and efficient route-finding algorithm with an intuitive and responsive UI/UX. The TSP algorithm that I have designed uses an upper bound in a heuristic approach to the problem, so that the solution that it reaches is always a real-world solution that the user can physically drive down.

Even when the location the user selected is in a field, and the route takes them to the nearest road, I have added a polyline (google maps line on the map) that shows where each point connects to the end of the road. This effectively shows any user the association between each destination on the calculated route and the physical point that they have selected. This is an improvement over most existing solutions, as the typical sat-nav solution will not show allow the user to select any point, only enter an address.

One limitation of the google maps distance matrix API that I have used is that the API sometimes returns an error when inter-continental points are selected. This error is handled in the solution pop-up with an appropriate message and warning. Following this warning, the user can close down the pop-up and remove the problematic points before selecting more appropriate points and attempting another calculation.

My client did provide feedback that the system cannot handle inter-continental journeys, one of the main weaknesses of the system. This can be remedied by using a different distance matrix api, for example the MapBox or Apple Maps apis. However, both of these do not offer a free option for amateur developers, so they were not considered in analysis.

The glaring problem with the routes that the system returns is that they are only upper bounds and may not be the optimal solution. This was considered when designing the algorithm that I will use, however I believe that the implementation that I chose provided the best trade off between assuredness of optimality, time, and viability. If I were to design a more complex algorithm that ensured an optimal solution every time, the system would be much slower and may not even be able to find an optimal solution with every set of points. It is for these reasons that I believe my Travelling Salesman Problem solving algorithm is the best option for the client at this scale, for the purposes of delivering parcels.