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:

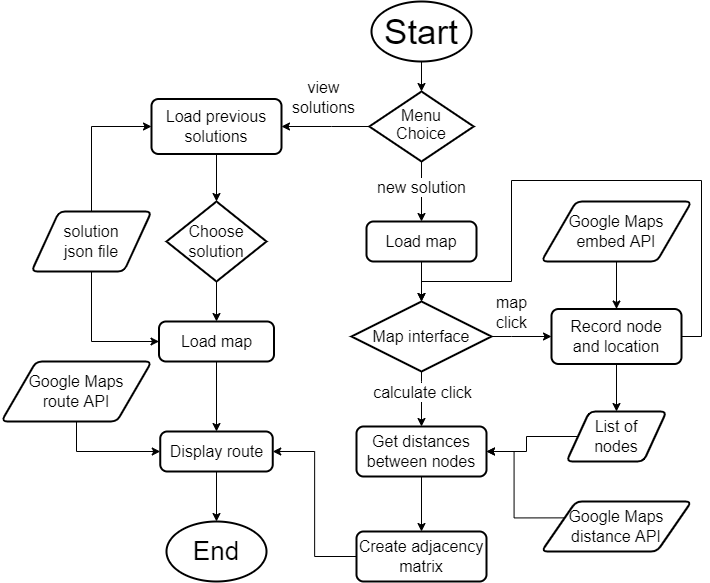
## 

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

Testing:

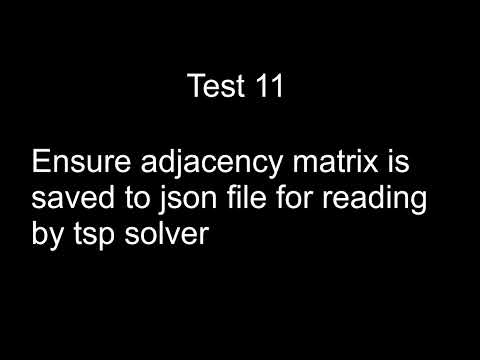
The entire system consists of three main components:

* 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

## 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: