Bicycle Route Planning on a Dynamic Graph

April 13, 2017

Lewis McQuillan
Student ID: 1444167
Supervised by Alan P. Sexton

Submitted in conformity with the requirements
for the degree of BSc Computer Science
School of Computer Science
University of Birmingham

Declaration

The material contained within this thesis has not previously been submitted for a degree at the University of Birmingham or any other university. The research reported within this thesis has been conducted by the author unless indicated otherwise.

Signed .....

Abstract

When planning a bicycle tour, users are often concerned with more than just the shortest or even the fastest route. Safety is a major concern, especially for beginner cyclists, so there may be occasions when certain roads should be avoided. Furthermore, there are often times when cyclists could take shortcuts which are not represented in map data and are therefore not accounted for during the route planning.

This report will cover the design, implementation and evaluation of a web application that allows users to plan routes more suitable for cyclists through the use of soft constraints on the search, and by modifying map data. The search uses a bi-directional A* implementation on a contracted graph of England.

The project has been rather successful. The application allows users to plan routes across England, in fair time, using a series of waypoints. Users are able to save modifications to the graph, classified as 'shortcuts' and 'avoids', which will then be considered in the search. Users may then choose to publish their modifications which allows other users to use them during planning. If the graph were to be expanded to cover a larger area such as Europe, an alternative search technique would likely be required in order to return a longer path within feasible time.

Location of Git Repository

All software for this project can be found at:
https://git-teaching.cs.bham.ac.uk/mod-40cr-proj-2016/ljm467.git/

Acknowledgements

Firstly, I would like to thank my project supervisor, Alan P. Sexton, for his support and guidance throughout my project. I would also like to thank Robert Hendley for his feedback in the midterm inspection. Finally, I would like to thank my family and friends, who have supported me for the duration of this project.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement

Route planning for cycling is fundamentally different to other modes of transport, such as driving. Typically with route planning, users are interested in either the shortest or fastest path, while with cycling there are other considerations to be made.

In a recent government survey (Department of Transport, 2014) 64% of the respondents were deterred from cycling as they believed the roads were too dangerous. Clearly safety concerns are a significant factor for why many people avoid cycling, so it is important for route planners to accommodate for this. Adjusting the search to follow quieter and safer roads could address this issue, but it is important to consider the trade-off with additional distance by following these paths.

There may be times where a cyclist could access areas or make shortcuts which are not represented in the map data. An example of this scenario would be a park without any cycle paths, clearly a cyclist could still ride through the park, however a route planner will typically detour them around the park. Alternatively there may be areas cyclists wish to avoid such as a specific busy roundabout, or a particularly steep hill.

As with any route planning, the efficiency and speed of the search is critical. It is important to consider that the search will be carried out on a large transportation network. While it is possible to use Dijkstra's classical algorithm on a graph of this size, it would simply not be practical (Delling, 2009). It is therefore important to review what techniques are available in order to achieve acceptable search speeds.

1.2 Aim and Objectives

The aim of this project is to produce a web application which allows the planning of bicycle tours throughout England. The system should allow for route planning more tailored towards bicycles rather than a more general shortest path route. In turn, this should allow for current cyclists to take advantage of more convenient routes, while also encouraging non-cyclists to participate through a safer experience.

The methods that will accommodate for a more tailored search include the ability to follow safer routes by adapting the heuristic function, and allowing users to modify the map to add information which is not represented in the original data.

In order the achieve the project aim, the following objectives have been established:

- Understand how current solutions have attempted the problem, and identify the strengths, weaknesses and characteristics they possess.
- Investigate and review the speed-up techniques available for search over large transportation networks.
- Provide an analysis of the problem and realise the project requirements.
- Produce a structured design of the web application.
- Identify which technologies are to be used for the implementation of the solution, and justify their selection.
- Implement the web application to service potential users.
- Thoroughly test the implementation to check for any errors and unexpected behaviour.
- Evaluate the success of the solution in terms of usability, performance, and accuracy.
- Compare the web application to current solutions, in order to gain an understanding of the strengths and weaknesses of the solution.
- Discuss the achievements and limitations of the solution, and establish possible further work.

1.3 Report Structure

The report is structured into nine chapters.

2. **Background**
   This chapter provides research that has been carried out in order to gain a deeper understanding of the problem. The section includes a review of current solutions in order to explore what is currently offered, and where these systems excel and fall behind.

3. **Requirements and Specification**
   This chapter gives a detailed analysis of what is required of the software system. Through this analysis, the user requirements have been established and use cases have been written in order to describe how the system should behave.

4. **Design**
   This chapter provides an high-level structural overview of the system. The chapter documents and justifies design decisions, and provides a description of the algorithms used.

5. **Implementation**
   This chapter provides details on how the solution has been implemented. We explore the workings of the key aspects of the system, and what algorithms have been used.

6. **Testing**
   This chapter provides an overview of how the software has been validated and testing through means of white-box and black-box techniques.

7. **Project Management**
   This chapter explains what management strategies have been used to ensure the success of the project within the given time constraints.

8. **Evaluation**
   This chapter provides details on the process and results of evaluation. The success of the solution is evaluated, through the evaluation of algorithm choices, user feedback, comparisons to current solutions and the overall robustness of the system.

9. **Discussion**
   This chapter will summarise the achievements and deficiencies of the project. The inadequacies of the project will be identified, and an explanation will be provided for why they occurred, and how they could be resolved. Further work will also be explored and discussed.

10. **Conclusion**
    This chapter will give a summary of how the provided system addresses the stated problem.

CHAPTER 2

Background

2.1 Current Solutions

In this section, a set of current solutions will be examined. The study will consider the usability of the system, quality of the route, speed the route is delivered, among other novel features on a case-by-case basis. The same set of routes (Appendix B.1) will be tested for each solution in order to make a fair comparison of the performance of the search. Each route will be referenced by their reference in the table.

2.1.1 CycleStreets

CycleStreets (2017) is one of the leading bicycle route planners in the UK. The website offers an A-to-B planning system - to plan a route you simply enter a start location, an end location, and an expected speed. It is worth noting that at this current time, the system is in beta testing.

The start and end location can be searched for using an address, selected by clicking on the map, or by detecting your GPS location. The interface to do so is very intuitive, and further adjustments can be made by dragging the markers placed.

Upon clicking a "Plan this journey" button, the page is redirected and the system delivers three route options; the fastest route, a balanced route, and the quietest route. It appears that the fastest route prioritises primary roads to avoid many turns, the quietest route follows side streets until it is necessary to travel on a primary road, and the balanced route similarly follows sides streets but has a softer constraint on when to move to primary roads. The system provides a wide-range of information for a given route, including the estimated journey time, distance, elevation data, calories burnt and a scale of "quietness". The route also provides turn-by-turn directions, providing an overview at each point in the journey. Once the route has been calculated, no modifications can be made, so no further waypoints can be added and the current start and end can not be adjusted with ease. This means the user must return back to the start if changes are to be made.

The system handled route A very well. The routes were delivered in approximately a second, and each route was well representative of the "type", i.e. the fastest route followed primary roads, and the quietest route followed side-streets and was clearly safer. Producing the path for route B was rather slow, taking about fifteen seconds to produce the result,

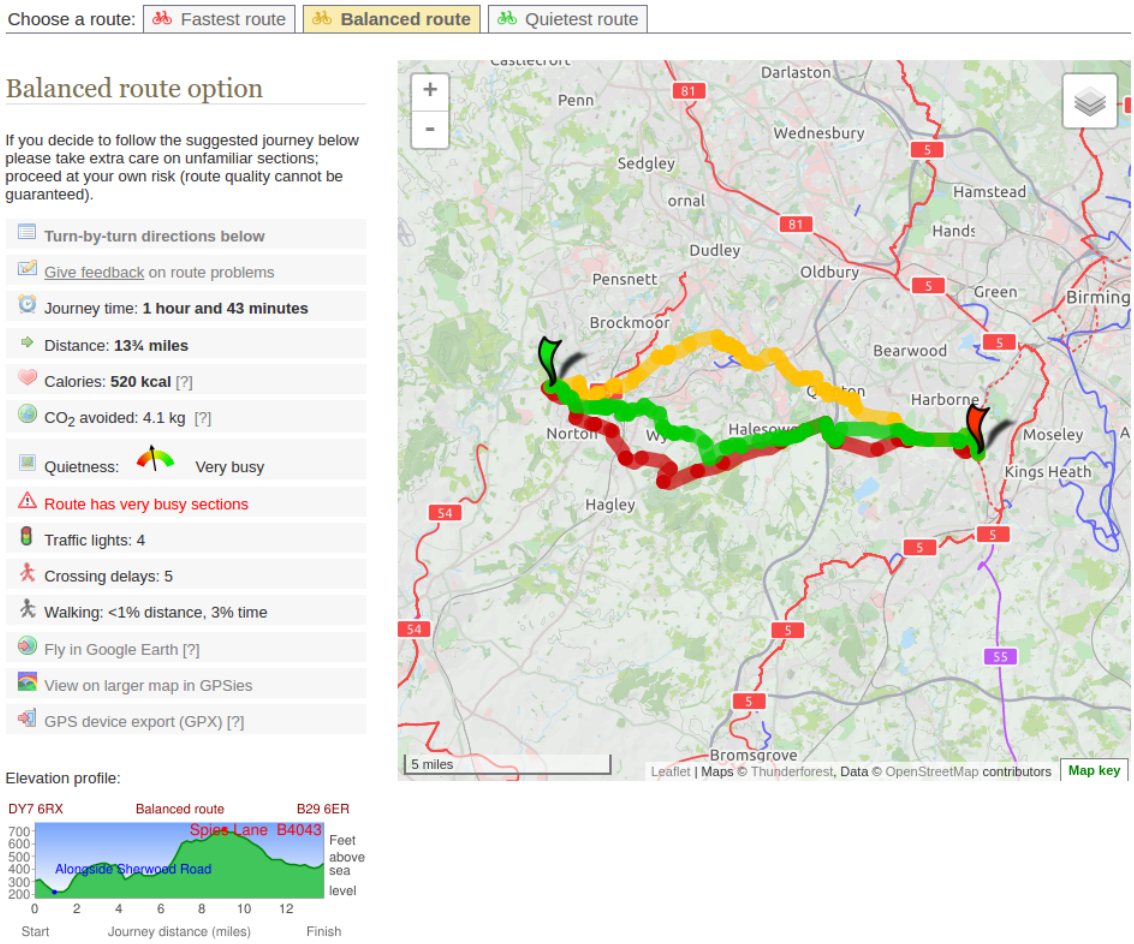


Figure 2.1: CycleStreets Route A

however the calculated routes, like route A, were impressive. The system would not allow route C to be planned, as it has a maximum distance of 400km. while this would not be a huge problem if the system allowed further waypoints to be added, the fact that it is an A-to-B planner means that it does not facilitate for journeys greater than 400km.

Overall this is a fairly robust solution, however it does have a few drawbacks. The quality of the routes produced is rather impressive, especially providing users with the option to choose routes of varying quietness. It is clearly targeted towards short-medium distance bicycle routes. While this is not necessarily a problem, it appears to be this way due to a slow search implementation, and it seems unfortunate to limit the good quality routes by distance. Some aspects of the UI are done well, while others could likely do with some work. The initial selection of the start and end points is easy to use, however it would be preferable to be able to modify the route without having to restart.

2.1.2 MapMyRide

MapMyRide (2017) is another leading bicycles route planner, which has worldwide coverage. A website is available to plan routes via a set of waypoints, while a mobile application has also been provided to follow routes through GPS tracking. The mobile application will not be considered, however, as the focus of this project is the route planning.

The website allows you to begin with a blank route, or by uploading a GPX file (see Section 2.5). However, if a GPX file is uploaded changes can not be made, so this feature is more targeted towards the route following. The UI for planning a route is intuitive; waypoints can be added by searching for an address, and then clicking on the map where you want the point to be added. Once a waypoint has been added, a path will be drawn from the previous waypoint. One noticeable drawback of the interface is that it is easy to mistakenly add waypoints while attempting to pan and zoom the map. While occasionally you are able to delete these waypoints, the functionality does not seem consistent, and can lead to having to restart the route from scratch. There is not a great deal of information provided about the route, other than overall distance and elevation, so perhaps a few more fields such as ETA, quietness, etc. would be useful.

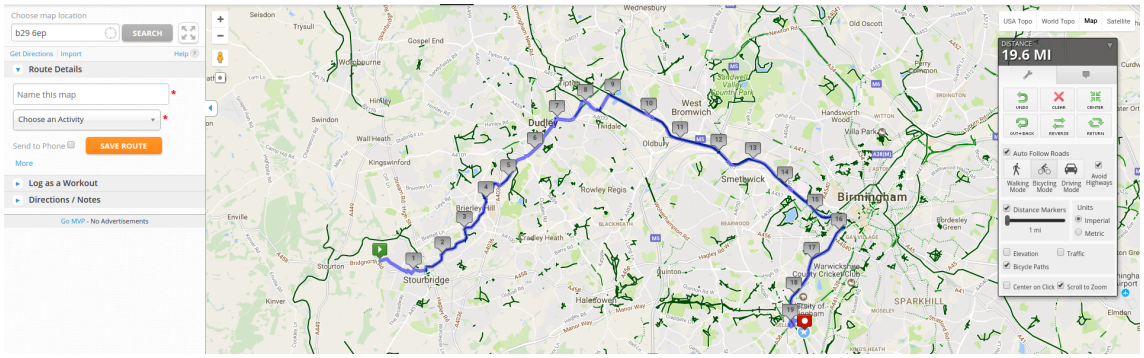


Figure 2.2: MapMyRide Route A

The search-speed for the solution is impressive, each route yielding a result in under a second. There was not a lot of flexibility for the route planning. For route A, the path followed the canals where possible, which would most likely by preferable, however this route does nearly double the distance, so the option to take a more direct route may be helpful. While the system does provide an "Avoid Highways" option, this appears to only avoid motorways, which are illegal to cycle on either way according to the Highway Code (Gov.uk, 2015). The generated paths for the longer routes were to a good standard, they were direct and followed secondary roads which were close to motorways. However, to achieve the impressive search speed, it appears that the system preprocesses the graph through contraction (see Section 2.2.2), and searches between the core nodes, while information is lost about the component nodes, following the route can then become little difficult at times. An example of this is shown in Appendix B.2.

To conclude, this solution handles the problem to a high standard. The UI is generally easy to use, despite the issue with accidentally adding waypoints. The implementation of the search provides a fast response and routes suitable for cyclists. It would be improved if

for longer distances the route kept all of the information for following the road, and users were given the option to take more direct routes in order to reduce the travel distance.

2.1.3 PlotARoute

PlotARoute (2017) is another leading bicycle route planner with worldwide coverage. The solutions takes a more similar form to MapMyRide as opposed to CycleStreets in terms of UI and functionality.

The website begins with a blank route and similarly to MapMyRide, waypoints can be added by searching for an address and clicking on the map. The website gives a wide range of data for a route, including a break down of directions, estimated time of arrival per waypoint, and elevation data.

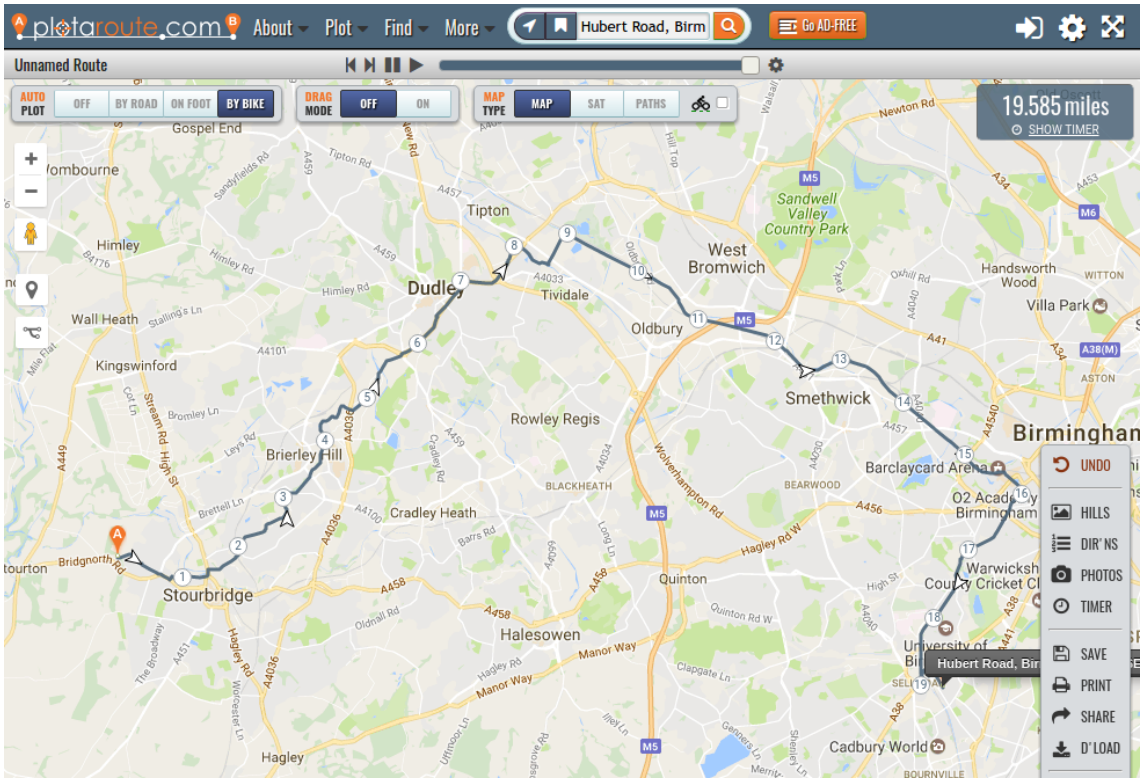


Figure 2.3: PlotARoute Route A

The planner calculated route A quickly, and returned the same path as MapMyRide. Again, while this route is probably preferable, the planner does not provide the option to take a more direct route. While the longer routes did take a little longer to process than MapMyRide, the quality of the routes were of a higher standard as they followed the roads exactly, leading to less ambiguity.

The solution is a good example of how the problem can be addressed. While it would be desirable to see more flexibility in terms of planning and perhaps an improvement to search speed, the system performs well with regard to usability and quality of routes delivered.

2.1.4 Summary

The reviewed solutions each have their strengths and weaknesses. It is clear that each of the systems acknowledge that cyclists desire safer routes, and they accommodate for this through favouring quieter roads.

It appears none of these systems address the problem of modifying map data in order to take shortcuts or avoid specific areas, so the introduction of this functionality would be novel.

2.2 Route Planning in Transportation Networks

It is important to carefully consider the search technique when working with a road network of great size. Schultes (2008) states that using a classical Dijkstra's (1959) approach would yield very slow query times, and would be computationally expensive. Given a graph $G = (V, E)$, Dijkstra's computes the shortest path from the start node s to all nodes $v \in V$, rather than just a target node t. This clearly is too much processing for such a large graph. On the other hand, using an aggressive heuristic to guide the search would produce inaccurate routes. It is difficult to find a suitable trade-off between speed and suboptimal routes when using this method, so there has been considerable research into techniques which can produce faster and more accurate results (Schultes, 2008).

Delling (2009) states that the majority of speed-up techniques split the work into two phases. The *preprocessing* stage computes additional data which can be used to accelerate the *query*. The preprocessing of the graph can take anywhere between a few minutes to a few days, depending on the technique used, however the work done at this stage can reduce query time to a matter of milliseconds. It is also important to make the distinction between a *dynamic* and *static* graph at this stage. A static graph consists of a fixed set of vertices and edges, whereas a dynamic graph is subject to modification. In the case of static graphs the preprocessing can be executed once, while for dynamic graphs each update requires further computation. As this solution aims to allow the user to modify the graph, it is important to consider a technique which allows for fast and regular preprocessing, while still providing considerable speed-ups for the querying.

Below we examine what techniques could be used to improve search-times, in regard to both the preprocessing and query stage.

2.2.1 Arc-Flags

To begin this technique, the graph is first partitioned into a set of regions. While these regions should be connected, each node should only belong to one region (Lauther, 2004). Then for each edge, we assign a set of flags - one for each region. If the edge lies on the shortest path to a region then we set that flag to *true* (Delling, 2009). This means we can use a modified implementation of Dijkstra's to follow only edges that lie on the shortest path to a given target nodes region.

This method has its advantages and its drawbacks. The technique has a considerable effect on query-time, Lauther (2004) notes an average speed-up of a factor of 64 compared to a graph without preprocessing. However, it is clear that this preprocessing is very computationally expensive both in terms of time and space. Furthermore, Delling (2009) describes the "coning" effect, where more edges lie on a shortest path as the target region is approached, reducing the speed-up of the technique. Eventually, when the target region is entered all edges are considered, as every edge has the own-region flag set to *true*. This effect can be eased through the use of bi-directional arc-flags or partitioning the graph more granularly, however this further increases the time and space required (Delling, 2009).

While this technique could be effectively utilised on a static graph, its use would be infeasible on a dynamic graph. Updating the graph would require a considerable amount of processing in order to recalculate which edges lie on the shortest path.

2.2.2 Contraction

The contraction technique extracts a subgraph from the input graph, which can be referred to as the *core* (Delling, 2009). The method is split into two stages; node reduction and edge reduction. During node reduction, we iteratively bypass nodes until no node is bypassable. To bypass a node x, we remove all of its incoming edges I and outgoing edges O, and for each node u in the tails of I we introduce new edges to each node v in the heads of O, with length *len(u,x) + len(x, v)*. The newly inserted edges are considered *shortcut edges*[1]. A node which has been bypassed is considered part of the *component*. A node x is only bypassable *iff*, given a tunable contraction parameter c, the following holds (Delling, 2009):

$$\text{No. shortcut edges} \leq c \cdot (deg_{in}(x) + deg_{out}(x)) \tag{2.1}$$

During the contraction of nodes, redundant shortcut edges can be added. Delling (2009) states that these edges are not required in order to keep the distances in the core correct, and can therefore be removed. Figure 2.4 illustrates the process of a node and edge reduction.

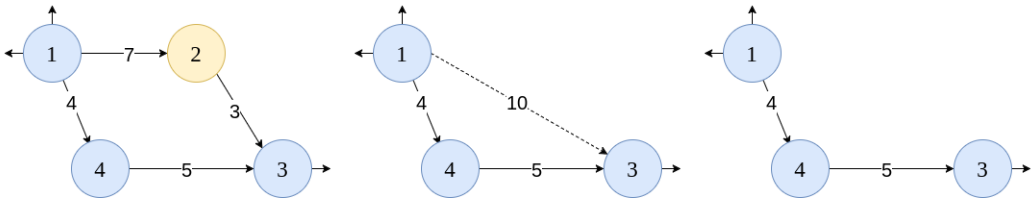


Figure 2.4: In this example, vertex 2 is being bypassed. The vertex and all of its incoming and outgoing edges are removed. A shortcut edge can then be added between vertex 1 and 3, with length $len(v_1, v_2) + len(v_2, v_3)$.

2.2.3 Query Optimisation

While the greatest speed-ups are achieved during the preprocessing, further improvements can be made during the query phase. In the previous sections we referenced Dijkstra's (1959) algorithm or modified versions thereof for querying the preprocessed graph. Dijkstra's algorithm takes a greedy approach to the shortest path problem, meaning "it

[1]Not to be confused with the shortcuts described in Chapter 1 for cyclists.

repeatedly selects from the unselected vertices, vertex v nearest to source s and declares the distance to be the actual shortest distance from s to v" (Reddy, 2013). A* search is a technique which takes a similar approach to Dijkstra's, however an additional heuristic $h(v)$ is calculated which provides an estimate on the distance from vertex v to a target t (Goldberg and Harrelson, 2004). The heuristic can therefore guide the search towards t, as opposed to evaluating every vertex.

Further improvements can be seen when bidirectional search is applied. This technique executes a forwards search from the source s, and a backwards search from the target t. The means both paths typically meet somewhere in the middle, requiring fewer vertices to be expanded.

2.2.4 ALT

Fuchs (2012) states the ALT algorithm is a variant of A* which uses a set of *landmark* vertices and *triangle inequalities* to compute a feasible *potential* function). During pre-processing, the distance for every vertex to every landmark must be calculated. Given the graph $G = (V, E)$, the triangle inequality $|x + y| \leq |x| + |y|$ can be used to derive $\text{dist}(u, v) \geq \text{dist}(l, v) - \text{dist}(l, u)$ and $\text{dist}(u, v) \geq \text{dist}(u, l) - \text{dist}(v, l)$ for any $u, v, l \in V$. Fuchs (2012) further notes that we can then define the two potential functions π:

$$\pi_t^{l+} \coloneqq \text{dist}(v, l) - \text{dist}(t, l) \qquad \pi_t^{l-} \coloneqq \text{dist}(t, l) - \text{dist}(v, l)$$

Using the *max* of these potential functions will the return the lowest bound for $\text{dist}(v, t)$. The speed-up benefits for ALT depend greatly on the selection of landmark vertices (Delling, 2009). The landmarks should be spread across the graph as much as possible. The *maxCover* (Goldberg and Harrelson, 2004) heuristic is one of the best-known techniques to do this, however it will not be covered in detail in this report.

Delling (2009) states that ALT is easy adaptable to dynamic graphs. Edge weights can be updated through the use of dynamic shortest path trees. Insertions and deletions can be reduced to edge weight changes; deleting an edge can be considered as setting its weight to infinity, while inserting an edge can be considered as setting its from infinity to a given value.

However, calculating the shortest path from every vertex to every landmark yields high memory consumption. Delling (2009) notes that 16 landmarks requires an additional 128 bytes for each vertex. Furthermore, the speed-ups can not compete with that of contraction.

2.3 OpenStreetMap Data

OpenStreetMap is a project to produce a free editable map of the world, using data provided and maintained by the community. The data contains information including roads, area boundaries, trails and more. It is licensed under the Open Data Commons Open Database License (ODbL), allowing it to be copied, distributed, transmitted and adapted as long as OpenStreetMap and its contributors are credited (OpenStreetMap, 2017). The data will be used for the graph for the search, and for the generation of the

map tile images, used to display the map.

The data is available to download via Geofabrik.de (2017), and can be downloaded by continent, country or administrative subdivision. The data is available in OSM XML format or OSM PBF (Protocol buffer Binary Format) format.

2.3.1 PBF Format

The PBF format is an alternative to the XML format, which is 30% smaller, about five times faster to write, and six times faster to read (OpenStreetMap, 2017).

The data is encoded using the Google Protocol Buffer. Protocol Buffers are used to serialise structured data; once a data structure is defined, it is simple to write and read data to the structure using generated source code, which then compiles to low-level serialisation code (Google Developers, 2017).

2.3.2 Elements

The OpenStreetMap data is comprised of elements categorised by nodes, ways, and relations. Each element can possess one or more tags, which each contain a key and a value. A tag describes a specific feature of an element (OpenStreetMap, 2017).

Nodes

A node represents a point on the planet. Each node has at least an ID, and a pair of coordinates, defined by latitude and longitude. A node represents a stand-alone feature, such as a point on a road, a park bench, etc. While nodes usually have no tags, some may possess tags such as `highway=traffic_signals`, which marks traffic signals (OpenStreetMap, 2017).

Ways

A way represents a polyline, which is an ordered list of nodes. A way can be defined as either a *closed way*, or an *open way*. An open way represents a linear feature, such as a road or a river. A closed way represents a polygon, such as an area boundary, or a building. Common tags for ways include `highway=<type of road>`, and `oneway=true`. A way also has an ID, which can be referenced in a relation (OpenStreetMap, 2017).

Relations

A relation documents the relationship between two or more elements. A relation can have different meanings, which is defined by its tags - typically the relation will have a *type* tag. A relation contains an ordered list of nodes, ways or tags, which are known as the members for the object. An example relation is a route, which lists the ways of a highway (OpenStreetMap, 2017).

2.3.3 Map Tiles

Tiles are used for a graphical representation of the map data. The tiles can be generated from OpenStreetMap data, using software packages such as Mapnik (2016), or can be obtained by various tile servers, such as those provided by OpenStreetMap (2017). The

tiles are usually 256×256 pixel bitmap images, which are arranged in a grid layout. Tiles are produced at different zoom levels, described in the following subsection.

Slippy Tilenames

The naming of tiles for OpenStreetMap data follows a standard convention. The filenames follow the format of `/path/to/tiles/zoom/x/y.png`

(a) A tile at zoom level 0 (b) A tile at zoom level 12

Figure 2.5: Tile examples for different zoom levels (OpenStreetMap, 2017)

The the x and y of the tile is derived from the zoom level. Tiles are typically generated for zoom levels 0 through to 19. At each zoom level z, the number of tiles t is calculated by (OpenStreetMap, 2017):

$$t = 2^z \cdot 2^z \tag{2.2}$$

So when $z = 0$, one tile will contain the whole planet, however when $z = 19$ more than 2.7×10^{12} tiles are required to cover the planet.

The x and y of the tile for given coordinates in degrees (lon, lat) can be calculated as follows (OpenStreetMap, 2017):

$$x = 2^z \cdot \frac{(lon + 180)}{360} \tag{2.3}$$

$$y = 2^{z-1} \cdot \left(1 - \frac{\ln(\tan(lat \cdot \frac{\pi}{180}) + \sec(lat \cdot \frac{\pi}{180}))}{\pi}\right) \tag{2.4}$$

Alternatively, you can calculate the coordinates (lon, lat) for a given tile (x, y) using the following equations (OpenStreetMap, 2017):

$$lon = \frac{x}{2^z} \cdot 360 - 180 \tag{2.5}$$

$$lat = \arctan\left(\sinh\left(\pi - \frac{y}{2^z}\right)\right) \cdot \frac{180}{\pi} \tag{2.6}$$

It is worth noting that $\{x, y\} \subset \mathbb{R}^+$, despite the fact that the tile images are named using integers. The decimal part of the tile numbers are used for the precise coordinates on that tile.

2.4 Other Data

While OpenStreetMap provides a comprehensive set of data, there are some elements which are not included, which may considered important for the project.

2.4.1 Postcode Data

Postcode data would be useful when querying locations for the search. The Office Of National Statistics (2015) provides the *National Statistics Postcode Lookup* dataset, which lists 1.75mil postcodes for the United Kingdom. Each postcode is listed alongside useful information for the project, including its respective coordinates, local authority name, region, and country. The file is available in a variety of formats, including CSV, JSON, and XML.

2.4.2 Elevation Data

Unfortunately, elevation data is very difficult to acquire. Ordnance Survey (2017) provides the *OS Terrain 50* dataset for free, however as suggested by the name, the elevation grid is separated by 50 metre posts, so would not be hugely helpful in regard to cycling. Precise data (5 metres) is available, but at considerable expense - typically £2,500+ for a one year license (Ordnance Survey, 2017). Another option would be to use Google Maps APIs (2017) to query elevation data at particular points, however this service has a 20,000 request limit per day, requires external API calls (which may be slow) and would require the usage of an embedded Google Map in accordance to their terms of service.

Due to these issues, elevation data will not be used in the project, but could be included as further work with greater funding.

2.5 GPX Files

The GPS Exchange Format (GPX) is an XML file containing coordinate data. Traditionally the GPX file format was designed for use for GPS devices, however it is now more generally used for exchanging route information. As the project will not be providing functionality for route following, the planner should be able to export to GPX so the routes can be followed using GPS devices. Descriptions of the key types in the schema are as follows (Topografix, n.d.):

Element	Description
gpx	The root element of the GPX file. The element must contain attributes describing the version of the schema used, and the name/URL of the software that produced the file.
metadata	This element contains information about the route. Optional attributes include author, description, copyright, timestamp and the coordinate bounds.
wpt	Each instance of this element represents a waypoint. A waypoint represents an independent point of interest or feature on the map.
rte	This element represents an ordered list of significant points for a route. GPS devices will use each item in the list to calculate the bearing and distance for the following point in the route, as the route is navigated.
trk	This element represents an ordered list of points which describe a path. Usually this data is collected from a GPS, after following a route.

Table 2.2: A description of the GPX file schema

CHAPTER 3

Requirements and Specification

3.1 Project Scope

3.1.1 User Identification

The system will accommodate for cyclists with a range of experience who are looking to plan journeys for commute, leisure or general travel. The system will assist users looking to plan routes of any distance within the bounds of England. Inexperienced cyclists should feel comforted with the ability to choose safer routes, while all users should benefit from the ability to modify the map in order to add shortcuts, or avoid particular areas.

The system is not targeted towards professional and competitive cycling, as that area involves keeping track of and analysing statistics of previous rides, which is more concerned with route following as opposed to the planning.

3.1.2 Assumptions

- The user will be using a desktop operating system, running one of the following web browsers: Google Chrome, Mozilla Firework, Internet Explorer 11, Microsoft Edge.
- The user is fairly computer-literate, and perhaps has experience with other online mapping tools.
- The user has an internet connection, required to interact with the web application.

3.2 Requirements

The research conducted on other solutions in Section 2.1 has provided some insight on what functionality is currently available, what aspects work well, and how the drawbacks of these solutions can be avoided. From the conclusions drawn from this research and personal ideas, a set of requirements have been established for the application.

3.2.1 Functional Requirements

Here a set of requirements are listed in regard to what the system should do:

1. The system must allow users to register using a username, email and password.
2. The system must allow users to login using their username and password.
3. The system must allow users to log out.
4. The system must explain any erroneous or invalid user input.
5. The system must only show the map to logged in users.
6. The system must allow users to pan and zoom a map of England.
   (a) Any objects drawn on top of the map must move in correspondence to this panning and zooming (e.g. waypoints and paths).
7. The system must allow users to search by address in order to add waypoints.
8. The system must draw clearly the path between waypoints.
9. The system must show all waypoints on the map.
10. The system must allow users to move waypoint positions.
11. The system must allow users to remove waypoints.
12. Upon the movements or deletion of a given waypoint, the system must recalculate the path(s) between the preceding and proceeding waypoints.
13. The system must allow routes to be loaded and saved to a user account.
    (a) The system must allow current routes to be loaded, edited and then overwritten.
    (b) The system must allow for routes to be shared with other users by providing a unique URL.
14. The system must allow users to export their routes to a valid GPX file, which can then be loaded onto GPS devices or apps.
15. The system must allow users to make modifications to the map.
    (a) The system must allow users to add personal modifications, which can only be used by them.
    (b) The system must allow users to add public modifications, which can be used by any user.
        i. The system must allow users to rate public modifications, so users can then filter modifications by rating.
    (c) The system must allow users to modify the map by adding an edge between two nodes, considered as a shortcut.
    (d) The system must allow users to modify the map by adding a set of edges which must be avoided by the search.
16. The system must allow users to choose to avoid primary roads.

   (a) The system must then follow secondary/tertiary roads until necessary to switch to primary roads.
17. The system must allow users to prefer roads with cycle paths.
   (a) The system must then follow cycle paths while they are available.

3.2.2 Non-Functional Requirements

1. The system must work on all major desktop web browsers.
2. The system must work on display sizes greater than 12 inches.
3. The system must return a search in less than 7 seconds for any two points.
4. The map panning and zooming must be responsive to user action.
5. The system must take measures to ensure user data is held securely.
6. The system should be able update the graph to accommodate for a modification in less than 1 second.

3.3 Constraints

The biggest constraint to this project is the limited funding which restricts what hardware and data is available. Due to amount of information in the map data, a system with high specifications would be required to represent the entire world or even a continent, which is why the solution will only cover the map of England. As stated in Section 2.4.2, elevation data is also expensive to obtain, while it would be a valuable addition for the project.

As with many software projects, this solution is constrained by time. The system will have $\approx$3 months of development time, so it is essential the project is managed well, and the goals are prioritised.

CHAPTER 4

Design

4.1 System Design

In this section we explore how the system will be defined in terms of architecture, components and interfaces, in order to satisfy the requirements outlined in the previous section.

4.1.1 Architecture

The system architecture will follow the client-server model. Due to the intensive memory and processing requirements, it would be infeasible for the computation of the search to be executed on client machines. Even if every client did have the required specifications, the graph would need to be preprocessed on each client before the search could begin. For this reason, any intense computation will be executed on the server based on user input, and a response will be returned to the client.

As the preprocessing of a graph can take a long time, the system will make use of a development and a production server. The development server will use a subgraph of England, namely the West Midlands. This will allow for changes to be made and tested quickly. On the completion of a feature and given the system is in a stable state, the code can then be deployed onto the production server, which will use the full graph of England.

The production environment will be hosted using a virtual private server (VPS) provided by OVH (2017). As the application will be extremely memory expensive, a VPS with 24GB of allocated RAM has been chosen, which should be more than enough the represent the graph of England.

MVC Pattern

The system will be developed using the model-view-controller pattern, which consists of:

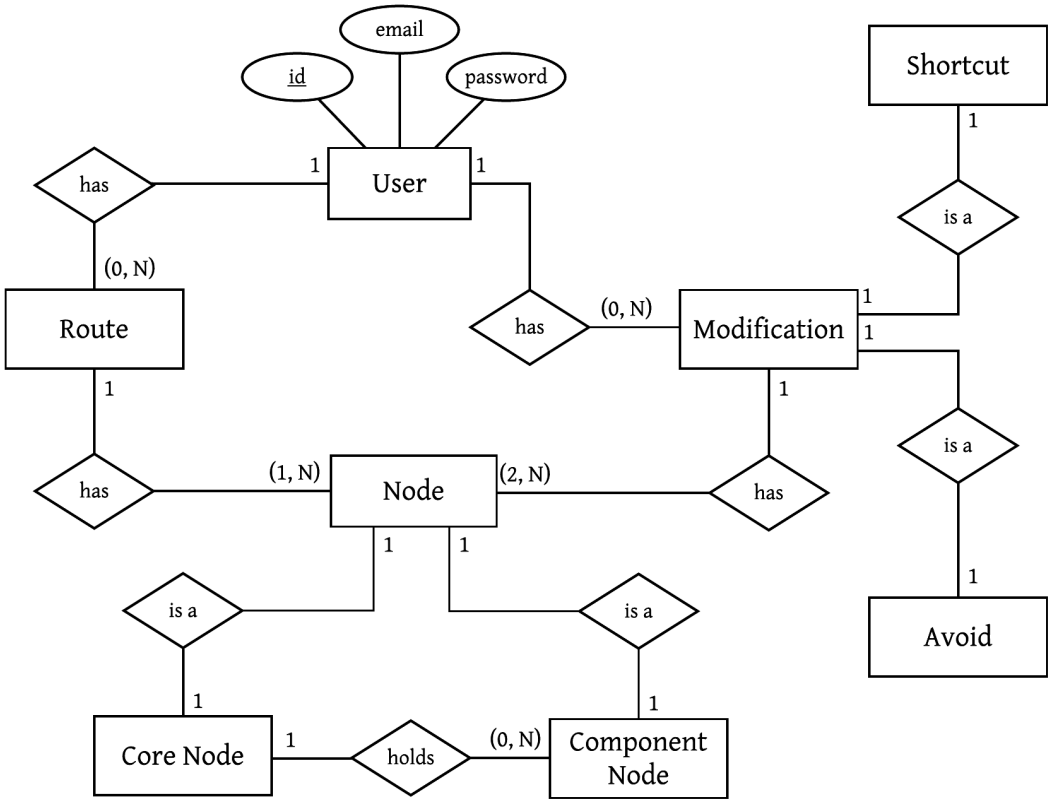
- **Model.** The objects which will be used on the server. The server will model the graph and map modifications.
- **Controller.** Used to update the models on the server. Upon user API requests, methods will be executed on the server in order to manipulate the graph, execute the search, etc.

- **View.** Upon updating the model, the server will return data to the client. The view will consist of a webpage comprised of HTML, CSS and JavaScript, which will represent the data returned.

4.1.2 Database Design

The system will require a database to store both user inputted data, and the graph data. While the search will use graph data stored in-memory, it will also be kept in a database which will allow the use of spatial functions, and additional information to be stored about the graph. Spatial functions are required for functionality such as finding the closest node for a given geocode. While this could be implemented in the server programming the spatial functions are extremely efficient with regard to the large dataset, so it would be logical to utilise what is provided. Furthermore, we need to be as efficient as possible with the resources allocated to the server container, so leveraging this workload onto the database will increase what is available for modules such as the search.

The ER diagram below shows the structure of the database.



4.2 Selection of Search Technique

In Section 2.2 various speed-up techniques were researched and we discussed the advantages and drawbacks they possess. While considerable speed-up would be preferable, it is also crucial to consider other factors such as hardware constraints and the time required to preprocess the graph. The method we use must also allow for quick updates to the graph as it is modified by the user.

While ALT seems interesting, providing decent query-times, and easy updates to the graph. Unfortunately the memory requirements are too high, thus being too expensive for the project.

Compared to the other reviewed methods of preprocessing it appears that contraction techniques typically put the least strain on memory, while providing excellent speed-ups, and will be used in the project.

At query-time, a bidirectional implementation of A* will be used. As discussed in Section 2.2, we can use the additional heuristic of A* to help guide the search towards the target, and used bidirectionally should provide noticeable improvements.

4.3 Technologies

4.3.1 Web Server

The chosen language for the back-end of this project is Java; mainly due to personal experience, and its object-oriented nature should *help* encourage good design patterns. Apache Tomcat (2017) is a servlet container for Java EE (2014), "which implements the Java Servlet, JavaServer pages, and Java WebSocket technologies", and will be used to run the web server.

In order to communicate with the front-end, an API is required. Jersey (2017) is a framework which simplifies the development of RESTful web services. While it is possible to develop an API purely through servlets, the framework simplifies the process greatly.

4.3.2 Database

A database management system (DBMS) will be required for the storage and retrieval of data within the system. The two DBMSs which will be examined are PostgreSQL (2012) and MySQL (2017). While I personally have more experience with MySQL, there are other considerations to be made.

As the solution will be working with geographical data, it is wise to consider which DBMS has the best support for spatial functions, especially given the vastness of the dataset. MySQL 5.7 has native support for spatial functions; while for PostgreSQL the PostGIS (2012) extension is available. At present, MySQL seems to be lacking some of the more advanced features that PostGIS provides, such as aggregate functions for spatial data

(BostonGIS, 2008). While it is unlikely these features will be used in the scope of this project, they may be required for any future work. Furthermore, more community support appears to be available for PostGIS; perhaps as it has been established for a longer time. For these reasons, PostgreSQL with the PostGIS extension has been chosen as the DBMS for this project.

4.3.3 Generating Map Tiles

In order to generate the map tiles, a series of software and libraries will be used. The process will be conducted as described on Switch2osm.org (2013), however instead of tiles being rendered dynamically i.e. as and when they are needed, the tiles will be pre-rendered in effort to reduce the workload on the server. First, the PBF file of England will be loaded into a PostgreSQL database with the PostGIS extension installed, using the software package Osm2pgsql (2017). A style sheet for the map images can then generated using OSM-Bright (Crosby, 2014). Finally, Mapnik (2016) can be used to read the database and generate the tiles with the generated style sheet. Tiles will be generated for zoom levels 5 through to 16, which should provide a sufficient amount of detail required for route planning.

4.3.4 Web Map

A slippy map is the name given to web maps, which allow panning and zooming (OpenStreetMap, 2017). External JavaScript libraries such as Leaflet (Agafonkin, 2017) and Slippy Map On Canvas (2012) have been considered to display a slippy map; however, it has been decided that the required functionality for this application is too bespoke and requires a greater degree of flexibility, so using these libraries may create more work in the long-run. Instead a slippy map will built from the ground up, taking inspiration from these libraries which will provide greater freedom of control.

4.4 User Interface

Building an effective user interface (UI) is an important aspect of this project. As the application will be open for public use, it is important that the interface is intuitive and easy-to-use. The interface will consist primarily of a single-page, which is reactive to user input.

The construction of the UI will follow Nielsen's (1995) 10 usability heuristics. Here we list each of the heuristics and how they will be addressed in the UI.

1. **Visibility of system status**
   The UI should indicate to the user when server-side processing is being executed, through the use of a "loading" animation.

2. **Match between system and the real world**
   The UI should avoid using technical words. The functionality to a plan a route should be presented in a logical order.

3. **User control and freedom**
   The UI should allow for the deletion and movement of any waypoints added.

4. **Consistency and standards**
   The flow of the the route planning should be consistent. The users should be able operate the application through moving down the page.

5. **Error prevention**
   The UI should be carefully designed to limit potential errors. Controls should be carefully selected so only valid input can be added.

6. **Recognition rather than recall**
   The UI should be self-explanatory. Typically, any functionality that may be used in the near-future should not be more than two clicks away.

7. **Flexibility and efficiency of use**
   The UI should guide the user through the planning of a route. Accelerators – unseen by the novice user – may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

8. **Aesthetic and minimalist design**
   The majority of the screen should just be of the map, presenting the user with a clear view of the route. Route planning should be executed through a sidebar, but should not detract from the map.

9. **Help users recognize, diagnose, and recover from errors**
   Any erroneous user input should return an error message explaining what the problem is, and how it can be solved.

10. **Help and documentation**
    While the UI should be able to be used without documentation, help should be provided for the more challenging aspects.

4.4.1 Frameworks

The use of icons can bring the advantages of being easily recognisable (Shneiderman, cited in Gatsou, Politis and Zevgolis 2012) and improving memorability of functions (Siau, cited in Gatsou, Politis and Zevgolis 2012). It is therefore wise to incorporate icons into the UI to aid the navigation of the application. FontAwesome (Gandy, 2017) provides a library of 675 clear and comprehensible icons and will be used in the development of the UI.

Material Design Lite (2017) is a framework containing templates, components, and controls based on the design of Material (2017). The framework will be used primarily to improve the aesthetics of the application and give it a more professional feel.

4.4.2 Wireframe

A wireframe of the main application screen is provided below. The wireframe has been created using the online tool myBalsamiq (2017). The wireframe has been annotated with yellow callouts which are described below.

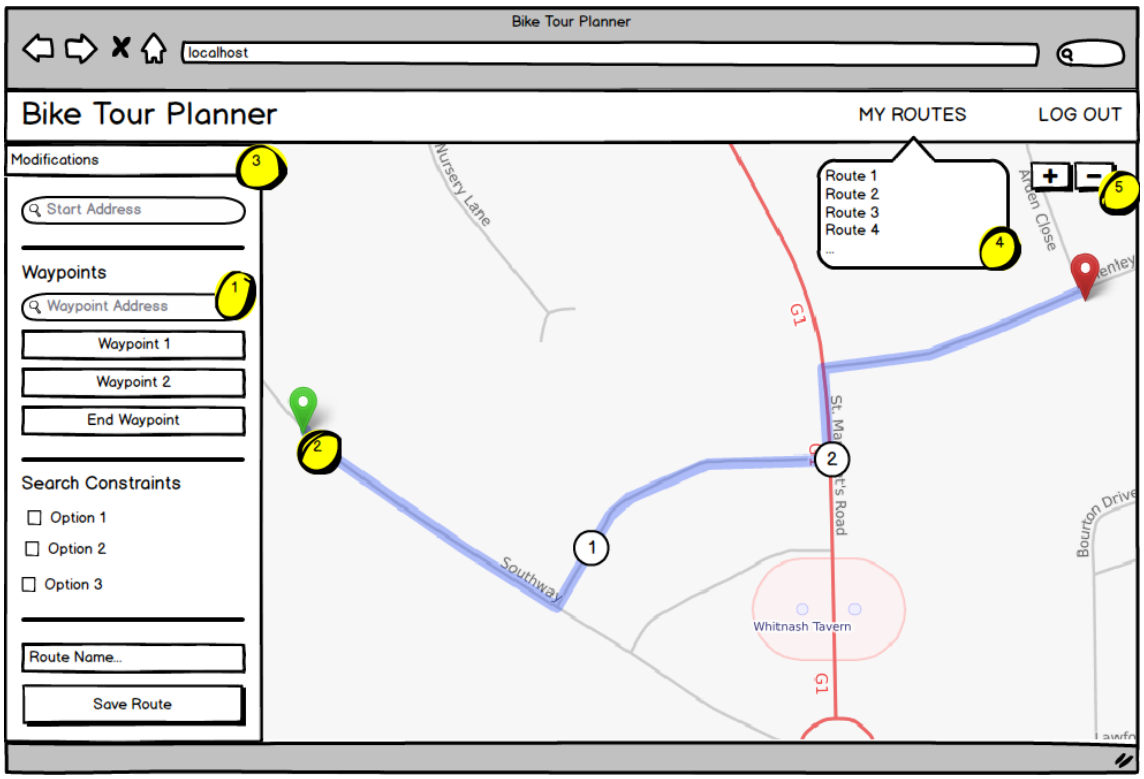


Figure 4.1: Map Image provided by OpenStreetMap (2017)

1. Upon entering an address, a new waypoint is added on the map and the sidebar listing.
2. Users will be able to move a marker on the map to recalculate a path.
3. This should expand to reveal user shortcuts and avoids.
4. Zooming should be possible through mousewheel or clicking on the zoom buttons.
5. Clicking on a route in the menu should load the path on the map.

CHAPTER 5

Implementation

5.1 User Authentication and Validation

The system allows users to register and log in. This functionality is added so users can save routes and map modifications to their account. The user can register using a username, email and password. Once a user has logged in, a session is generated on the server, which stores their user ID.

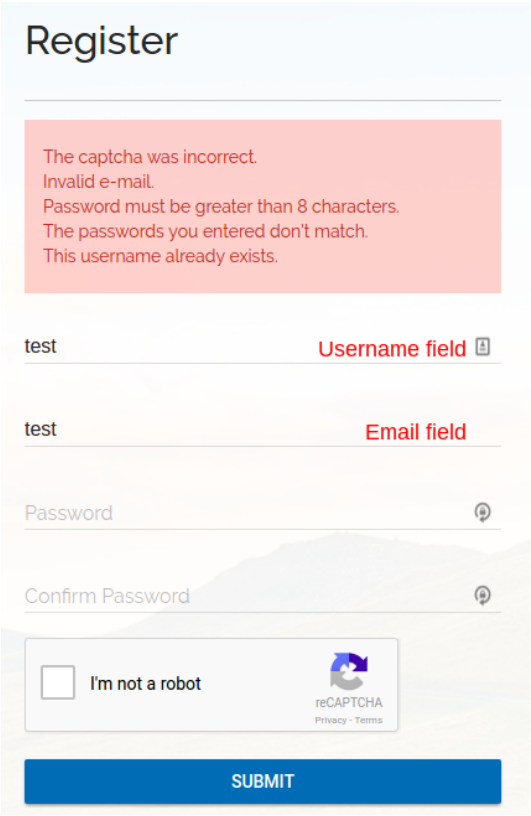


Figure 5.1: The slippy map in debug mode

Various measures have been taken in order to validate user input before it is inserted into the database. All validation is done server-side, and any errors are reported back to the

user, as shown in Figure 5.1. Both the username and email must be unique, i.e. not exist in the database. The email is validated against the RFC822 (2012) regular expression. In order to encourage password security, the password has a minimum length of eight characters.

5.2 Security

Despite not being a security-oriented project, it is important for any project to maintain solid security, especially those dealing with user data. While a breach of data for this project would not lead to any sensitive data being exposed, it is rather typical for users to use the same credentials over a variety of websites, which could lead to a more sinister compromise. This section details the security measures that have been taken, in response to the attack vectors listed in the OWASP Top 10 (2013).

In response to SQL injection, prepared statements have been used. Any execution of SQL statements is performed server side, so the `PreparedStatement` interface provided Java EE (2014) is used to bind any parameters into the queries.

In response to session management and broken vulnerabilities, a few steps have been taken. User passwords are hashed using a salt and SHA256, so they are stored as:

$$SHA256(salt + password)$$

In turn, if the database were to be compromised, attackers could not run the hashes against rainbow tables[1] as the same message will generate a different hash.

In response to data exposure, the site uses an SSL certificate. This means that any data transmitted is encrypted.

Finally to prevent XSS attempts, all untrusted data from user input is escaped before it is displayed on the page. This measure has been implemented using the Jsoup library (Hedley, 2016), and sanitises the input based on a given whitelist.

5.3 Slippy Map

The slippy map which has been implemented for this project uses a canvas object for the rendering, and is manipulated using jQuery. Figure 5.2 shows the slippy map in debug mode which may make it easier to visualise how it works.

The module uses event-driven programming. As the user interacts with the map, different functionality is triggered. The basic idea is that a grid of 256×256px map tiles, generated as per Section 4.3.3, is rendered onto the canvas. The map can be panned and zoomed, using mouse events, which will then re-render the canvas with the required tiles.

The module essentially uses three different unit systems. In order to load the correct tiles, the map tiles have to be tracked using the `z/x/y` system described in Section 2.3.3.

[1] Tables which contained precomputed hashes for a set of plaintext messages.

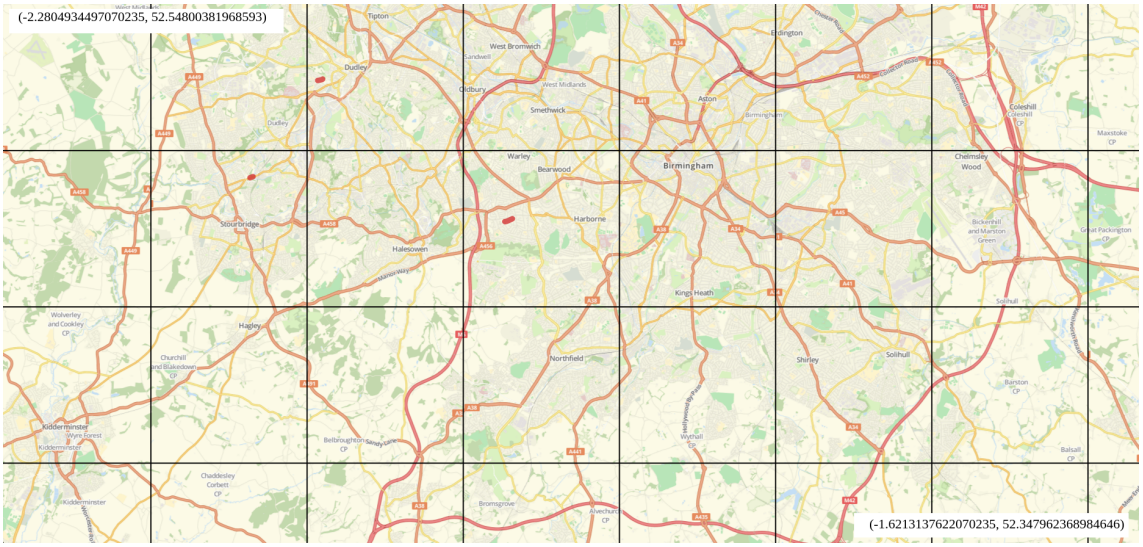


Figure 5.2: The slippy map in debug mode

We keep track of the `x` and `y` tile in the top left hand corner, and render a grid of tiles based on the size of the viewport (canvas). A further two tile border is also pre-rendered, to make panning more seamless. As the map is panned we increment/decrement an `x` and `y` *pixel offset* based on direction. Once an offset is equal to or greater than the map tile size we shift the `x` or `y` *tile* based on direction. As the map is zoomed the `z` is incremented/decremented. An algorithm for how these tiles are rendered is shown in Algorithm 1.

Algorithm 1 Tile Render

```
1:  procedure LoadTiles
2:      xTileCount ← ceil(viewportWidth / tileSize)
3:      yTileCount ← ceil(viewportHeight / tileSize)
4:      for i = −2 to (xTileCount + 2) do
5:          x ← i + mapXPos
6:          xPixelPos ← x × tileSize + xOffset
7:          for j = −2 to (yTileCount + 2) do
8:              y ← j + mapYPos
9:              yPixelPos ← y × tileSize + yOffset
10:             if tile (x, y, zoom) is loaded then
11:                 draw tile (x, y) for new zoom at pixel (xPixelPos, yPixelPos)
12:             else
13:                 draw tile (x, y) for previous zoom at pixel (xPixelPos, yPixelPos)
```

The map also needs to be able to convert any position to geocodes, which are used for drawing paths, markers, etc. To do this we can use the conversion functions listed in Section 2.3.3. We convert the tile number in the top-left corner of the viewport and the bottom-right corner of the viewport using these functions, so we have the bounds for the geocodes. Then if we are given a geocode, we can find the pixel position by calculating where that geocode is in relation to the bounds. The pixel position can then be used to draw the polylines and objects on the canvas.

On zooming in and out we draw the tiles from the previous zoom level but at an enlarged/shrunk resolution until the new layer has loaded the images. The old tiles are replaced by the new tiles at the correct resolution. This is done in effort to create a more seamless transition between layers, rather than wiping the whole layer until the new tiles are loaded.

5.4 API

The client communicates with the server using through an API. As a user interacts with the interface AJAX requests are made to the server returning a JSON response. When a user makes a request to the API, the server check their session in order to validate that they are logged in and the request is valid. If the request is not valid, the API returns a HTTP 403 Forbidden status. If the request is valid, the user ID variable from the session is typically used to retrieve data from the database for that user.

Requests typically containing data are made to the API through jQuery functions, and once the response is received a callback function is triggered which will use the data from the response. A full list and explanation of the API endpoints is given in Appendix C.

5.5 JSON Data Representation

In order for the view to represent data returned from the server, suitable data structures are required. We can then iterate through these structures, in order to draw them onto the canvas.

5.5.1 Waypoints and Routes

A route is essentially an array of paths. A waypoint is an image which shows the start and/or end of a path. They are stored in separate data structures as a waypoint can mark the end of one path and the beginning of another, and each path has two waypoints, so combining them into one structure would likely lead to redundant data.

```
// Array with paths
[
  // Single path
  {
    id: 1,
    // Array with path points
    path: [
       // Single point
      {
        lat: 90,
        lon: -90
      },
      ..
    ]
  },
  ..
]
```

```
// Array with waypoints
[
  // Single waypoint
  {
    id: 1,
    // Waypoint image properties
    height: 80,
    width: 40,
    img: Image()
    // Waypoint geocode
    location: {
      lat: 90,
      lon: -90
    },
    // Waypoint pixel position
    pixelLocation: {
      top: 100,
      left: 100,
      bottom: 100,
      right: 100
    }
  },
  ..
]
```

A path is simply an ordered list of geocodes, which means we can draw a polyline by converting each geocode to its pixel position (described in Section 5.3). So we can iterate through the paths and draw each one.

A waypoint shows a different image, of different size depending on which paths it lies on. The first and final waypoints in the list show a larger flag icon to denote the start and end of the route, while intermediate waypoints show a smaller number icon. We must store the pixel position because the image is loaded asynchronously. This means we build the

placeholder for an image, and continue to execute other tasks, such as drawing the path. Once the image has loaded, we can simply place it in the stored placeholder.

5.5.2 Shortcuts and Avoids

As described in Section 5.7, shortcuts are comprised of two nodes, while avoids are a set of edges.

```
// Array with shortcuts
[
  // Single shortcut
  {
    id: 1,
    name: "name",
    description: "desc",
    public: true,
    head: {
      lat: 90,
      lon: -90
    },
    tail: {
      lat: 90,
      lon: -90
    }
  },
  ..
]
```

```
// Array with avoids
[
  // Single avoid
  {
    id: 1,
    name: "name",
    description: "desc",
    public: true,
    // Array of avoid edges
    edges: [
      {
        head: {id, lat, lon},
        tail: {id, lat, lon},
        inters: [
          {id, lat, lon},
          ..
        ]
      },
      ..
    ]
  },
  ..
]
```

For each shortcut we can simply draw a single line from the geocode listed in the tail, to the geocode listed in the head. However, for each avoid we must iterate through each of its edges, beginning at the tail and draw a line to each intermediate and then finally to the head.

5.6 Route Finding

Due to hardware and software constraints and in attempt to build an optimal user experience, it was important to make the search as efficient as possible. Below the details and techniques used to implement the search are explained.

5.6.1 Constructing and Preprocessing the Graph

The process for which the graph is built also contracts the graph, in order to make the search faster. It takes about 20 seconds to process the map of the West Midlands on the development server, and about 5 minutes to process the map of England on the production server. Once the graph has been built and preprocessed on server start-up, it is stored as a servlet context variable so that it is kept in memory.

In order to parse the PBF file, the library OSM-Binary (Crosby, 2014) was used. This library parses all nodes in order of ID, followed by ways. The data parsed from this file is primitive, so a new graph data structure was required. The graph structure contains two primary data types, defined by a list of `Nodes` and a list of `Edges`. A `Node` contains an ID, geocode and a list of incoming and outgoing `Edges`. An `Edge` contains a weight, a head and tail `Node`, a set of component `Nodes` and various flags to note whether it is one-way, a shortcut, etc.

The PBF file is read twice. Once to check which nodes should be contracted, and then another time to build the contracted graph. On the first iteration we build a list of `Nodes` while reading the PBF nodes. While parsing the ways, instead of adding edges we simply count how many times each node is referenced. Nodes with more than one reference will be a junction so will be added to the core, while nodes with one reference is simply a node along a way so can be added to the component. Now we know which nodes belong to the core and component, we can parse the ways again to add edges. For each way, we begin a new edge and for each node along the way we check if the node is lies on the core. If the node is a not a core node, it is added as an "intermediate" for the edge. If the node is a core node, then the head of the edge is set to that node, and a new edge begins from that node. The weight of the edge is the cumulative haversine[2] distance between all component nodes. The process continues for the rest of the way. A basic representation of the procedure to parse a way is shown in Algorithm 2.

Because the ways give node references by ID when adding edges, an efficient method to find the `Node` in the list is required as we have no information about index. One proposed solution was to store the `Nodes` in a `HashMap` with the ID as the key, rather than a list. This would mean a retrieval of the `Node` could be done in $O(1)$ time, however the memory overhead was simply too large. Instead, the `Nodes` were stored in an `ArrayList` ordered by ID. We could then execute a binary search using the ID as the comparator field, yielding a worst-case time complexity $O(\log n)$, with the `ArrayList` consuming considerably less memory.

[2] A formula to calculate the great-circle distance between two points on a sphere.

Algorithm 2 Parsing a given way

```
1: procedure ParseWay
2:     for i = 0 to wayNodes.size do
3:         if wayNodes[i].isCore then
4:             intermediates ← []
5:             fromNode ← wayNodes[i]
6:             prev = fromNode
7:             weight = 0
8:             for j = i + 1 to wayNodes.size do
9:                 if wayNodes[j].isCore then
10:                    toNode = wayNodes[j]
11:                    weight+ = haversine(prev, toNode)
12:                    addEdge(fromNode, toNode, weight, intermediates)
13:                    if not way.isOneway then
14:                        addEdge(toNode, fromNode, weight, intermediates)
15:                    break
16:                else
17:                    intermediateNode = wayNodes[j]
18:                    weight+ = haversine(prev, intermediateNode)
19:                    prev = intermediateNode
20:                    intermediates.push(intermediateNode)
```

5.6.2 Querying

As selected in Section 4.2, the querying uses an implementation of bi-directional A* search. A start node and an target node is required in order to begin the search. For two given geocodes we can use a PostGIS spatial distance query (see Section 4.3.2) to obtain the closest core nodes. This typically takes about 500ms for each point, so about a second is added to the search, no matter how long the path is. Now we have the start and target node, we can begin a search on the core.

The search uses the haversine distance, to calculate distances for the the heuristic, so that the curvature of the Earth is accounted for. The search uses separate instances of data types for the forwards and backwards direction. The open sets are implemented using a `PriorityQueue`, which orders nodes by the value of $f(n)$ in ascending order. The $g(n)$ value of each node is stored in a `HashMap<Long, Double>`. The key is of type `Long` because it stores the `Node` ID, with the value given as a `Double`. The use of a `HashMap` means that during the search we can retrieve the value of *g(n)* in *O(1)* time for a given node.

5.7 Map Modifications

Map modifications allow users to add data to the graph, and can be categorised by shortcuts and avoids. While how each of these types are used in the search does not differ greatly, the way they are added and removed from the graph does. Shortcuts are essentially a pair of `Nodes`, while avoids are a set of `Edges`.

All modifications have a name, description and an associated user ID. Each modification is either personal which means they can only be used by the user who created it, or public so they can be used by anyone. The name, description and whether the modification is public or personal can be edited after insertion.

Before starting a search, a user can decide whether to use public and/or personal modifications of each type. If a user chooses not to use a modification of a particular type, the search will ignore `Edges` with the respective flag and the modifications will not be shown on the UI.

5.7.1 Shortcuts

Shortcuts involve the insertion of an `Edge` between two `Nodes`. It is important for users to be able to add shortcuts along a road rather than just at junctions, so the functionality to add shortcuts between core and component nodes is required. An example of a shortcut in use can been seen in Figure 5.3.

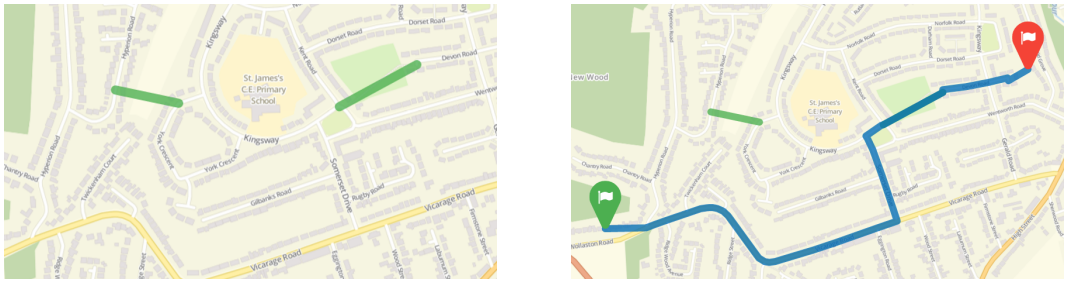


Figure 5.3: Example usage of a shortcut to cycle through a park. Shortcuts are represented with a green polyline.

To add a shortcut, a user can select any two points on the map and for each of the points, the server finds the closest core or component `Node` using a PostGIS spatial distance query. If a `Node` lies on the core, an edge can simply be added to it. However, if a `Node` is part of the component we must make it part of the core, and reassign the edges for the head and tail for the edge it sits on. A new `Edge` can then be introduced between the `Nodes`. The shortcut `Edge` is assigned the *shortcut* flag and a *shortcut user* variable, making it possible to distinguish which `Edges` are shortcuts during the search. Figure 5.4 shows how a shortcut is added between a core and component node.

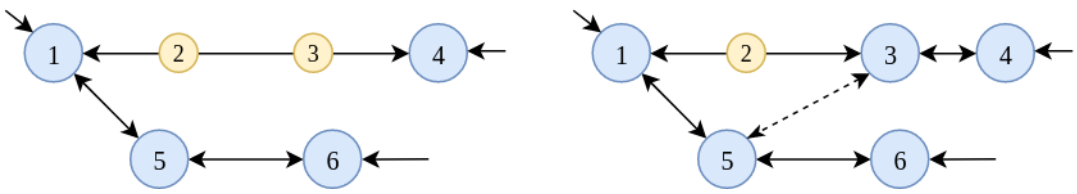


Figure 5.4: In this example, we are adding a shortcut between node 3 and 5. Any the smaller nodes are part of the component, while the larger nodes belong to the core. As node 3 is made part of the core, the edge between 1 and 4 is removed, introducing two new edges: $1 \leftrightarrow 3$ and $3 \leftrightarrow 4$. A new shortcut edge can then be added between 3 and 5.

5.7.2 Avoids

Avoids involve adding a flag to a current set of `Edges` which will then be ignored by the search. Because avoids are comprised of edges, we can ignore components nodes as the edges only lie between nodes on the core.

To begin adding an avoid the user first selects a road on the map, sending the geocode of the click to the server. The server then finds the two closest neighbouring points. Once the initial edge has been added, the user can then select more roads adding edges in a similar manner. The only difference is that now it finds the closest node out of the *subgraph* of selected edges, and the nearest neighbour of that node to the point. An example of an avoid in use can be seen in Figure 5.7

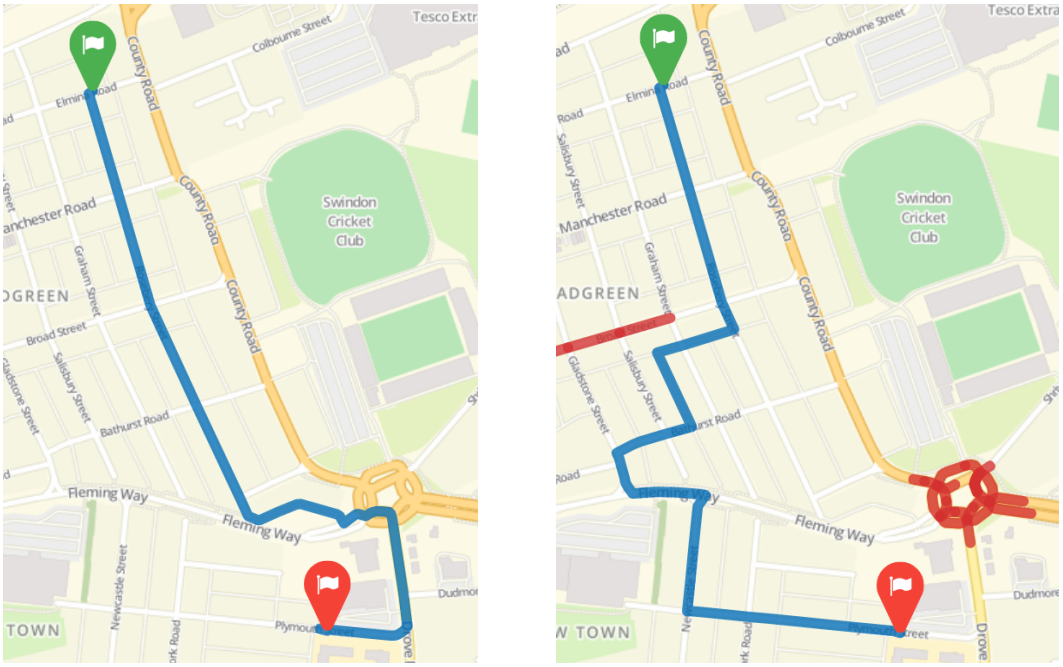


Figure 5.5: Example avoidance of a busy roundabout. Avoids are represented with a set of red polylines.

5.7.3 Loading and Publishing Modifications

It is important that only necessary data is kept in the graph so that search times are kept to a minimum. Unnecessary data refers to `Edges` which would be held in the graph, but no one could use them. The search algorithm would still need to check these `Edges` to see if they could be explored, even though we know they never could be. If the project were to be successful, given a user base of around 50,000 users each with 5 shortcuts, an additional 250,000 edges would be kept in the graph which would have a noticeable effect on the search. On the other hand, we can not temporarily insert modifications during search-time as the process of querying the database for the user modifications, inserting them into the graph, etc. would be even more detrimental to the speed of the search.

The way personal and public modifications are inserted and held in the graph differs. Public modifications are loaded into the graph during server start-up, and then kept in the graph. This is because every user is allowed to use them, so the data should be readily available. On the other hand, only one user can use personal modifications so it would not make sense to always keep these in the graph. One idea to combat this was to load a copy of the graph when a user logs in, with their personal modifications inserted. However, the graph simply requires too much memory to store a copy for every concurrent user. Instead, when a user logs in their modifications are loaded into the main graph, and removed when they log out. While not all logged in users are able to use these modifications, it is certainly a much smaller subset of redundant data than if all personal modification were always kept in the graph.

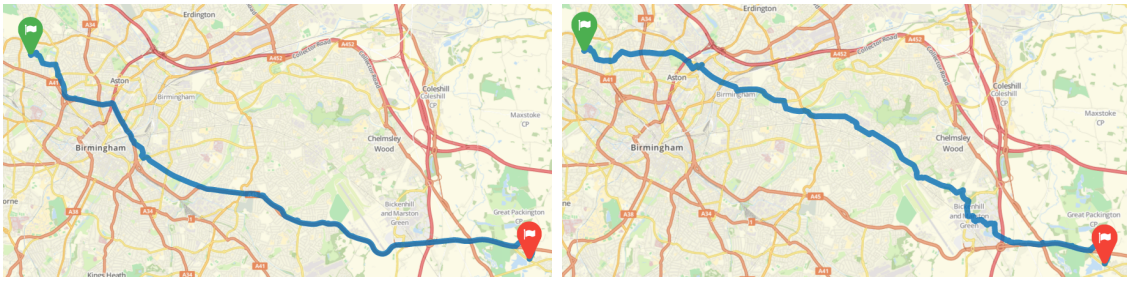
5.8 Search Constraints

Search constraints allow the user to make more general adjustments to the search. The basic idea is that a penalty or reward is given to different road types, which means the search will favour certain roads but will still consider the other types if necessary. This kind of soft constraint has been used over a hard constraint for a couple of reasons. First, the map of England is not fully connected without using primary roads or higher (see Figure 5.6), so completely ignoring these road types could make search impossible. Another reason is to prevent the search taking extremely long detours for the sake of avoiding a short stretch of road.

Figure 5.6: In this example, it would be impossible to navigate from Carlisle to Longtown without using an A roads or motorway for at least a short distance.

The reward or penalty is decided for an `Edge` when the ways are being parsed on server start-up. The algorithm checks the tags for each way, so the road type can be determined. Each road type has constants assigned to it, which can used as coefficients for the length function during the search. By multiplying the length function instead of adding to it, the penalty or reward is relative to the length of the road. Adding constants to the length would affect short roads too much, and longer roads too little.

The two options given to users are "avoid main roads" and "prefer cycle-ways". For each way, we check the value of the `highway` key. The different values for the tag have different constants; the larger, faster road types such as primary roads have higher constant values. We also check whether the way has the `cycleway` tag, which indicates a way has a cycle path. If the tag exists, then another constant is added which is ≤ 1.0, hence acting as a reward. When the constants have been chosen we can add them to the `Edges` of that way. It is worth noting that the two are not mutually exclusive; a road with a penalty can also have a reward for being a cycleway. If the user does not wish to use these options, they can be deselected and the coefficient(s) will not be applied to the length function.



(a) (b)

Figure 5.7: The same route when using main roads (a), and avoiding main roads (b).

CHAPTER 6

Testing

6.1 Structural Testing

Structural testing, also known as white-box testing, follows a "detailed investigation of internal logic and structure of the code" (Ehmer and Khan, 2012).

6.1.1 Unit Testing

The architecture of this project makes unit testing very difficult. The server for the application takes user input as the request, and typically returns JSON output. However, valid responses are only provided when there is an active session generated through the login procedure, which is difficult to mock through testing suites. JWebUnit (Dashorst and Wright, 2015) is a testing framework specifically for web applications, which can simulate browser behaviour to navigate through a website and set assertions. The framework has been used to validate user authentication, as shown in Figure 6.1, however once the user is authenticated testing again becomes difficult as a lot of the functionality lies within manipulating the canvas object.

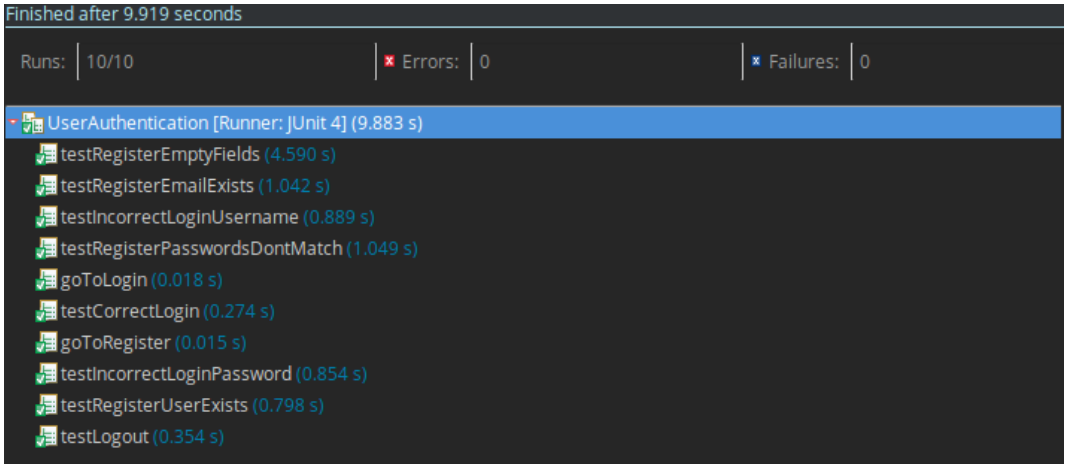


Figure 6.1: Unit tests for user authentication

6.2 Functional Testing

Functional testing, also known as black-box testing, assumes no knowledge of the internal workings of the system and investigates the functionality which is available to the user (Ehmer and Khan, 2012).

Test	Description	Expected Result	Actual Result
1	Logged out users should always be presented with login screen.	Attempted access to `index.jsp` with no valid session should redirect to `login.jsp`	**Pass**.
2	Logged in users should always be presented with map.	On site load, given an active session, the map is shown.	**Pass**.
3	Map should pan when dragged by the user.	While mousedown, the map should move in the direction of mousemove	**Pass**.
4	Map should zoom in and out on mouse wheel.	Mouse wheel up should zoom in on the map, mouse wheel down should zoom out.	**Pass**. The functionality works, however could be improved by reduced sensitivity to mousewheel.
5	Users should be able to search by postcode.	Upon entering a postcode in the format A9AA 9AA, a full address should be returned, and a new waypoint added.	**Pass**. The search returns a full address for correctly formatted postcodes and adds a new waypoint.
		Upon entering a postcode in the format A9AA9AA, a full address should be returned, and a new waypoint added.	**Fail**. The search returns the error message *Unable to parse your search.*
		Upon entering an invalid postcode, an error message should be returned.	**Pass**. The search returns the error message *Unable to parse your search.*
6	Users should be able to search by geocode.	Upon entering a geocode in the format `lat, lon`, a full address should be returned, and a waypoint added.	**Pass**. The search returns a full address for correctly formatted postcodes and adds a waypoint.

		Upon entering an invalid geocode, an error message should be returned.	**Pass**. The search returns the error message *Unable to parse your search.*
7	Users should be able to move waypoints.	While mousedown on a given waypoint, the waypoint should move, on mouse up it is dropped and a new address is found.	**Pass**.
8	A clear should be drawn between two successive waypoints.	A blue path should be draw between every successive waypoint.	**Pass**.
9	Users should be able to delete waypoints.	Upon deletion of a waypoint it should be removed from the sidebar listing, and a new path should be drawn between the preceding and proceeding waypoints.	**Pass**.
10	Users should be able to adjust search to avoid main roads.	With the option selected, any new paths or movement of current waypoints returns a new path avoiding main roads.	**Pass**.
11	Users should be able to adjust search to prefer cycleways.	With the option selected, any new paths or movement of current waypoints returns a new path preferring cycleways.	**Pass**.
12	Users should be able to create shortcuts.	Upon clicking "Set Points", a user clicks once to begin the shortcut and again to end the shortcut. Both points on shortcut should snap to the nearest node.	**Pass**.
		Users should be able to reset the points of their shortcut by re-clicking "Set Points"	**Pass**.
		Users should be able set the name and description for their shortcut by clicking on the text fields.	**Pass**.

		Users should be able to edit the name and description of previously added shortcuts.	**Pass**.
13	Users should be able to create avoids.	Upon clicking "Set Points", a user can click on a road and the edge will be added to the avoid. Users should then be able to click on neighbouring edges to add them.	**Fail**. While this works for the most part, there are times where clicking on a road will result in the wrong road being added to the avoid.
		Users should be able to reset the points of their avoid by re-clicking "Set Points"	**Pass**.
		Users should be able set the name and description for their avoid by clicking on the text fields.	**Pass**.
		Users should be able to edit the name and description of previously added avoids.	**Pass**.
14	Users should be able to toggle the use of public and/or personal modifications.	Only selected modifications should be drawn on the map, and used in the search.	**Pass**.
15	Users should be able to save routes.	Upon entering a name for the route the and clicking the save button, the route should be saved and the user presented with a share URL.	**Pass**.
16	Users should be able to view their saved routes.	Upon clicking "My Routes" the list of their routes should be shown.	**Pass**.
17	Users should be able to load saved routes.	Upon clicking a route in the "My Routes" the route should be loaded with the route data, name and share URL.	**Fail**. While the route is shown, the waypoint images are incorrect and is missing the route name and share URL.

18	Users should be able to load a shared route.	Upon loading the share URL the route data and name should be loaded.	**Fail**. While the route is shown, the waypoint images are incorrect and is missing the route name.
19	Users should be able to export routes to a GPX file.	Upon entering a route name and clicking the "Export GPX" file for a new route, a valid GPX file should automatically download.	**Pass**. The GPX file has been downloaded and tested using GPX Visualizer (Schneider, 2016).
		Upon entering a route name and clicking the "Export GPX" file for a loaded or shared route, a valid GPX file should automatically download.	**Fail**. The page redirects to a "404 Not Found" error page. This is due to relative urls being used on the button, so the /download path is appended to /share
20	Users should be able to log out.	Upon clicking "Log out" the user will be redirected to `login.jsp`.	**Pass**.

CHAPTER 7

Project Management

7.1 Changes to Original Proposal

The proposed project was to include a web application for route-planning and a mobile application route-following. It was soon decided that implementing both of this systems would simply be too much work for the given time constraints. Instead of trying to cover both systems to a poor standard, a more focused approach was taken for the planning aspect.

7.2 Schedule

Time-management was crucial for the success of the project, so it was important that milestones and their expected delivery dates were established. Appendix D.1 shows a Gantt chart for the project.

7.3 Weekly Meetings

During the course of this project, weekly meetings were held with my project supervisor, Alan Sexton. In these meetings, we discussed the progress of the given week, any outstanding issues, and work to be finished by the following week. The meetings were important to ensure goals were achieved and provided guidance for any problems which were encountered.

7.4 Git Version Control

Git version control (Torvalds, 2005) allows the tracking of changes for files within a system, so that they can be recalled at a later date if required. The use of git in this project meant that the files were backed-up, and if any severe software problems were encountered a previous version was available.

CHAPTER 8

Evaluation

In this chapter, the implemented solution is evaluated. The search performance is evaluated and compared against other solutions. A user feedback study has also been conducted, to grasp an understanding of how system performs overall - mainly in terms of usability.

8.1 Search Evaluation

The search used for the project has been tested against the solutions reviewed in Section 2.1. As per that section, the routes were tested using those listed in Appendix B.1. It is difficult to get completely fair results, as we are unsure what hardware the other solutions are running and how long the search takes with regard to the overall response time. When testing my search, I can output exactly how long it takes, whereas while testing the other solutions, it is more of an approximation. It is also worth noting that CycleStreets calculates three routes during the planning, hence increasing search-time. Therefore, these results should be used to gain a *general* idea of the search times, but should not be given too much consideration.

Route Ref.	This Solution	CycleStreets	MapMyRide	PlotARoute
A	59ms	≈3s	< 1s	< 1s
B	279ms	≈13s	< 1s	≈3s
C	3461ms	N/A	< 1s	≈8s

Table 8.2: Comparative search-time experiments between solutions.

The results show that the speed of the search is comparable to that of the current solutions. Clearly the search would have to be improved if it were to match the speed of MapMyRide, however it still performs to a good standard. The results seem to be growing exponentially with distance, which is understandable because as distance grows, more nodes are expanded which do not lie on the "shortest" path. With that in mind, if the data were to be expanded to cover a greater region, such as the entire planet, the search method may need to be revised to accommodate for even longer distances. However, complete cross-country bicycle routes are unlikely, and cross-continent routes even more so.

Of course speed is not the only metric when considering the performance of a search. The quality of the route with regard to cyclists is likely a more important factor, presuming

the search-speed is practical. If we recall back to Section 2.1, both MapMyRide and PlotARoute favoured the canal for route A. Figure 8.1 shows the route produced by this solution with the options to avoid main roads and prefer cycleways enabled.

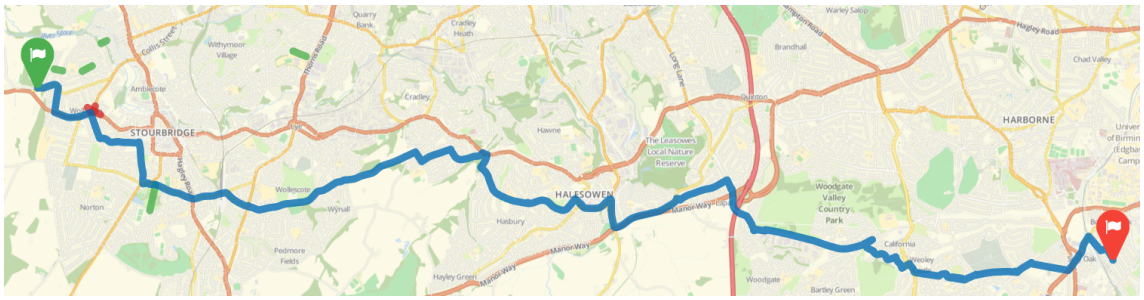


Figure 8.1: Route A planned with the implemented solution.

The route does take a more direct route, however the route still follows side-streets so some traffic would be present. This was perhaps an oversight during the design of the system, and the addition of another parameter to prefer canals would provide the user with routes of minimal traffic.

Routes B and C are handled well. When the "avoid main routes parameter" is enabled the planned routes follow countryside paths for the vast majority of the way, while still taking a relatively direct route. Problems may occur with the introduction of elevation data as these roads will likely contain more hills, which may not be desirable. With the "prefer cycleways" option enabled the routes similarly follow countryside roads between cities, however upon approaching cities the path follows roads with cycle-paths alongside them, as expected.

8.2 User Evaluation

A user feedback study has been conducted via a questionnaire. The study mainly addresses usability of the system as these issues can be difficult to grasp on a system you have built yourself. The study asked users to complete tasks on the application, and then questioned them on their user experience. The user is also provided with the option to leave additional comments at the end of the questionnaire.

As stated in Section 3.1.1, the system should accommodate for cyclists of varying experience, so the questionnaire was distributed to people who were known to cycle regularly, and others who rarely do. Unfortunately only 11 volunteers were found, so while some useful observations can be made, more respondents would lead to greater confidence in the conclusions drawn.

8.2.1 Discussion of Results

The results provided in Appendix E aid the understanding of the strengths and deficiencies of the usability of the system.

The majority of participants found the creation of routes straight forward. Multiple comments mentioned that searching for an address was limited, and requested the ability to search by other fields, such as town. This feature could have certainly been implemented if time permitted, but perhaps it should have been prioritised during the project management.

Most participants found that the route was delivered in sufficient time. However, the length of the route that users tested is unclear. This is a drawback in the design of the questionnaire, and we should have perhaps provided a longer test case to see if the users were still satisfied.

The results show that the average comfortability score for the default route is 3.4, while the modified search scores 4.0. It would therefore be wise the make the modified search the default, and users would have to disable the search constraints.

It appears that the participants in general found the insertion of shortcuts easier than avoids. It was noted in Functional Test 13 that the interface for adding avoids is a known issue, and upon the continuation of development, the fixing of this bug would be of high priority.

To conclude, the conducted user evaluation has helped to identify some outstanding issues, and reconsider the severity of others. On the other hand, we are more aware of what works well in the system.

CHAPTER 9

Discussion

9.1 Summary of Achievements

Overall the solution has been a success. The search speeds are comparable to those delivered by current leading solutions, and in some cases superior. The quality of routes provided are generally of high quality, but they could be improved further through additional search constraints such as guidance towards canals.

The ability to modify specific areas of the map is a novel feature which has not been encountered in the solutions reviewed in this report, or any other solutions which have been found. While this functionality has a lot of potential, the biggest drawback is that the modifications have to be added manually and thus, typically requires prior knowledge of the route. In aim to combat this, the system allows the use of public modifications, which in turn should prove more helpful as the database of users grows.

9.2 Further Work

The project has a lot of capacity to grow, and many ideas have been formed for how it could do so. I would like to continue working on this project, extending its functionality and realising its full potential.

The first area of development would be to introduce a route-following mobile application, and doing so would improve the work of the current solution, primarily in terms of map modifications. The automatic insertion of shortcuts could be achieved if a user takes a shortcut which is not currently represented in the system. It would be difficult to add this functionality for avoids, unless users consistently avoid a set of roads when they appear on routes. Users could also benefit by manually adding shortcuts and avoids as they appear while following the route.

Another area of development could be the expansion of the graph to cover more countries. This would require greater hardware in order to store the graph, and perhaps further contraction to speed up the search. While the search handles cross-country routes fairly well, it would most likely take a long time to calculate a cross-continent route, however it is highly unlikely that users would plan routes of that size.

With the introduction of elevation data, more functionality could be achieved with regard to the search constraints. Users could choose to take steeper or more levelled routes.

CHAPTER 10

Conclusion

In conclusion, the web application allows users to plan routes over England and efforts have been made to focus the route planning for bicycles. The user can either modify the search heuristic to plan routes based on road type, or can modify the map to add shortcuts which are not represented in the map data or avoid specific areas.

Overall, I have enjoyed working on this project. The development of this software has taught me an array of skills, while improving others. The project introduced me to new challenges, in both the engineering and development of the software. I look forward to continuing work on this project to see what can come of it.

Bibliography

[1] Agafonkin, V. (2017). *Leaflet — an open-source JavaScript library for interactive maps*. [online] Leafletjs. Available at: `http://leafletjs.com/`.

[2] Apache Tomcat. (2017). Apache Software Foundation.

[3] BostonGIS (2008). *Boston GIS: Geographic Information Systems Web Mapping OpenGIS and open source Solutions*. [online] Available at: `http://www.bostongis.com`.

[4] Crosby, S. (2014). *OSM-binary*. [online] GitHub. Available at: `https://github.com/scrosby/OSM-binary`.

[5] Cyclestreets. (2017). *CycleStreets: UK-wide Cycle Journey Planner and Photomap: Cycle journey planner*. [online] Available at: `https://www.cyclestreets.net/`.

[6] Dashorst, M. and Wright, J. (2015). *JWebUnit*. [online] JWebUnit. Available at: `https://jwebunit.github.io/jwebunit/`.

[7] Delling, D. (2009). *Engineering and Augmenting Route Planning Algorithms*. Ph.D. Karlsruhe Institute of Technology.

[8] Department of Transport, (2014). *British Social Attitudes Survey 2014: Public attitudes towards transport*. [online] Available at: `https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/481877/british-social-attitudes-survey-2014.pdf`.

[9] Dijkstra, E. (1959). A note on two problems in connexion with graphs. *Numerische Mathematik*, 1(1), pp.269-271.

[10] Ehmer, M. and Khan, F. (2012). A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications, 3(6)*.

[11] Ex-parrot.com. (2012). *RFC822*. [online] Available at: `http://www.ex-parrot.com/pdw/Mail-RFC822-Address.html`.

[12] Fuchs, F. (2010). *On Preprocessing the ALT-Algorithm*. Ph.D. Karlsruhe Institue of Technology. Gandy, D. (2017). *Font Awesome, the iconic font and CSS toolkit.* [online] Fontawesome.io. Available at: `http://fontawesome.io/`.

[13] Gatsou, C., Politis, A. and Zevgolis, D. (2012). The Importance of Mobile Interface Icons on User Interaction. *International Journal of Computer Science and Applications*, 9(3), pp.92 – 107.

[14] Geofabrik.de. (2017). *GEOFABRIK*. [online] Available at: https://www.geofabrik.de/data/download.html.

[15] Goldberg, A. and Harrelson, C. (2004). *Computing the Shortest Path: A* Meets Graph Theory*. Redmond, WA.

[16] Google Developers. (2017). *Protocol Buffers | Google Developers*. [online] Available at: https://developers.google.com/protocol-buffers/.

[17] Google Maps APIs. (2017). *Google Maps APIs | Google Developers.* [online] Available at: https://developers.google.com/maps/documentation/

[18] Gov.uk. (2015). *The Highway Code - Guidance - GOV.UK.* [online] Available at: https://www.gov.uk/guidance/the-highway-code.

[19] Hedley, J. (2016). *Jsoup*.

[20] Java EE. (2014). Oracle.

[21] Jersey. (2017). Oracle.

[22] jquery.org. (2017). *jQuery*. [online] Jquery.com. Available at: https://jquery.com/.

[23] Lauther, U. (2004). An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, 22, pages 219–230.

[24] Mapmyride.com. (2017). *MapMyRide*. [online] Available at: http://www.mapmyride.com [Accessed 28 Mar. 2017].

[25] Mapnik. (2016). *Mapnik.org - the core of geospatial visualization & processing.* [online] Available at: http://mapnik.org/.

[26] Material Design. (2017). *Material Design*. [online] Availableat:https://material.io.

[27] Material Design Lite. (2017). *Material Design Lite*. [online] Available at: https://getmdl.io.

[28] Mybalsamiq.com. (2017). *Painless Remote UX | myBalsamiq*. [online] Available at: https://www.mybalsamiq.com/.

[29] MySQL. (2017). Oracle.

[30] Nielsen, J. (1995). *10 Heuristics for User Interface Design*. [online] Nngroup.com. Available at: https://www.nngroup.com/articles/ten-usability-heuristics/.

[31] Office Of National Statistics. (2015). *National Statistics Postcode Lookup UK | Open Data Portal*. [online] Available at: https://opendata.camden.gov.uk/Maps/National-Statistics-Postcode-Lookup-UK/tr8t-gqz7.

[32] OpenStreetMap. (2017). *OpenStreetMap*. [online] Available at: https://www.openstreetmap.org/.

[33] Osm2pgsql. (2017). *GitHub*. [online] Available at: `https://github.com/openstreetmap/osm2pgsql`.

[34] OVH. (2017). *Web hosting, cloud computing and dedicated servers - OVH*. [online] Available at: https://www.ovh.co.uk/.

[35] OWASP Top 10. (2013). 1st ed. OWASP Foundation.

[36] PlotARoute. (2017). *plotaroute.com - Walking, Running, Cycle Route Planner*. [online] Available at: `https://www.plotaroute.com`.

[37] PostGIS. (2012). OSGeo.

[38] PostgreSQL. (2012). The PostgreSQL Global Development Group.

[39] Reddy, H. (2013). *PATH FINDING - Dijkstra's and A* Algorithm's*. Available at: `http://cs.indstate.edu/hgopireddy/algor.pdf`.

[40] Schneider, A. (2016). *GPS Visualizer*. [online] Gpsvisualizer.com. Available at: `http://www.gpsvisualizer.com`.

[41] Schultes, D. (2008). *Route Planning in Road Networks*. Ph.D. Karlsruhe Institute of Technology.

[42] Slippy Map On Canvas. (2012). GitHub. [online] Available at: `https://github.com/dfacts/Slippy-Map-On-Canvas`.

[43] Switch2osm.org. (2013). *Manually building a tile server (14.04) | switch2osm*. [online] Available at: https://switch2osm.org/serving-tiles/manually-building-a-tile-server-14-04/.

[44] Torvalds, L. (2005). *Git*.

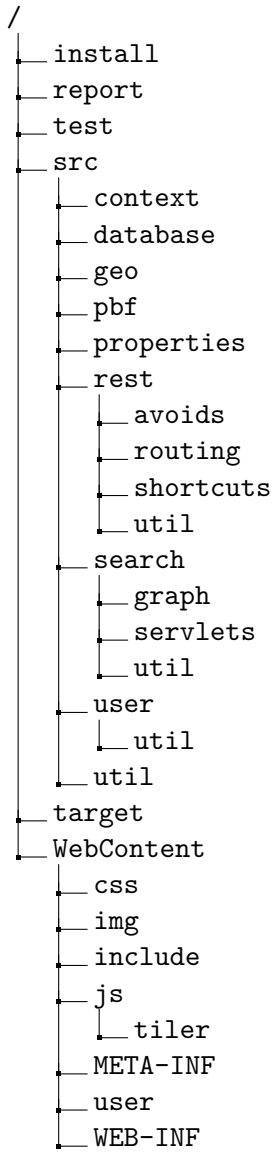
[45] Topografix. (n.d.). *GPX: the GPS Exchange Format*. [online] Available at: `http://www.topografix.com/gpx.asp`.

Appendices

CHAPTER A

Project Information

A.1 Directory Tree



Install. Provides a full installation website and the required files to install the system.
Report. The root directory for this report.
Test. Unit tests for the project.
Src. The source files for the Java back end.

- **Context.** Classes to manage context variables on server start-up/shutdown and session initialisation/destruction.
- **Database.** Classes to connect to and manage the database.
- **Geo.** Classes to manage context variables on server start-up/shutdown and session initialisation/destruction.
- **Pbf.** Classes to parse the PBF file(s).
- **Properties.** Classes to read in the configuration files.
- **Rest.** End points for the API.
- **Search.** Classes to the run the search.
- **User.** Classes to handle user authentication.
- **Util.** Miscellaneous utility classes.

Target. The compiled Java classes.
WebContent. Static web content.

- **Css.** Style sheets for the website.
- **Img.** Images for the website.
- **Include.** Files that at included throughout the site.
- **Js.** Classes to parse the PBF file(s).
- **META-INF.** Java meta directory.
- **User.** Pages for user authentication.
- **WEB-INF.** Contains configuration properties and required libraries.

A.2 Installation Information

The installation for the software is quite involved. If you wish to try the software for yourself, it is available via:

URL	`https://lewismcquillan.com/BikeTourPlanner/`
Username	`user`
Password	`password`

Tile Server

The tile server for this project is available at for the map of England has been provided at:

`https://lewismcquillan.com/tiles/z/x/y.png`

Alternatively, if you wish to show the entire planet (OpenStreetMap, 2017):

`https://a.tile.openstreetmap.org/z/x/y.png`

Prerequisites

The following software packages are required before installation:

- Java EE SDK 7+
- Apache Tomcat 8+
- PostgreSQL 9.2+
- PostGIS 2.0+
- Apache Ant 1.9+

In terms of hardware, the following is required:

- 4GB+ RAM if using the map of the West Midlands.
- 24GB+ RAM if using the map of England.
- At least 60MB of disk space + space for the pbf you wish to use.
  - 60MB for the map of West Midlands.
  - 713MB for the map of England

Installation Instructions

1. Configure the database

```
$ sudo -u postgres -i
$ createuser btp
$ createdb -E UTF8 -O btp planner
$ logout
```

2. Install PostGIS extension

```
$ sudo -u postgres psql
$ \c planner
$ CREATE EXTENSION postgis;
$ ALTER TABLE geometry_columns OWNER TO btp;
$ ALTER TABLE spatial_ref_sys OWNER TO btp;
$ \q
```

3. Import database tables

```
$ cd /path/to/install
$ psql -d planner -a -f sql/create_tables.sql
```

4. Download a PBF file, in order to obtain West Midlands (Geofabrik, 2017):

```
$ wget http://download.geofabrik.de/europe/great-
   britain/england/west-midlands-latest.osm.pbf
```

5. Configure properties files located at: `/path/to/install/config/`

6. Run install script

```
$ cd /path/to/install
$ ./install.sh
```

7. Deploy generated `BikeTourPlanner.war` to Tomcat.

CHAPTER B

Current Solution Study

B.1 Routes

Ref.	From	To	Distance (KM)
A	Stourbridge (DY7 6RX)	Selly Oak (B29 6EP)	≈18
B	Cambridge (CB3 0JN)	Nottingham (NG7 1BP)	≈150
C	Newcastle upon Tyne (NE13 6AS)	Exeter (EX4 4EA)	≈600

B.2 MapMyRide

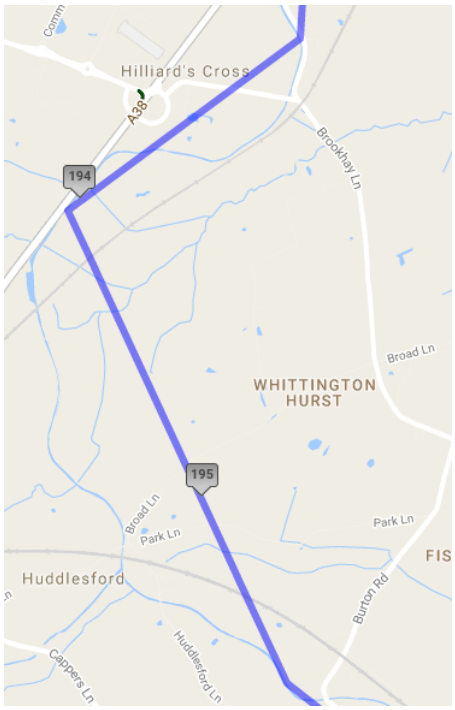


Figure B.1: Lost information for long routes.

CHAPTER C

API Documentation

Category	URL Pattern	Purpose	POST Body	Returns
User	/user/session	Checks if the user session is valid	-	JSON boolean
Routing	/route/checkExists	Checks if a user has a route saved under a given name.	The route name	JSON boolean
	/route/load	Load a saved route	The route ID	JSON object containing route data and name
	/route/loadRoutes	Loads all user routes	-	JSON array of all route metadata
	/route/save	Saves a new route	JSON object containing route data	JSON object containing success status and if successful, the route identifier
	/geo/find	Parses a search string and returns an address and geocode	A search string, either a postcode or geocode	JSON array containing formatted address and postcode
	/path	Searches between two geocode	A start and end geocode, and the search option	JSON object containing search path
	/share	Loads a shared route	The unique route identifier	JSON object containing the waypoints and path for the route

Shortcuts	/shortcuts/draw	Gets data required to draw all shortcuts	User selection to draw personal and/or public shortcuts	JSON object with all shortcuts geocode data
	/shortcuts/point	Gets a point for a new shortcut	The geocode of the point	JSON object of closest node
	/shortcuts/user/get	Gets a single shortcut for a user	The shortcut ID	JSON object of the shortcut
	/shortcuts/user/all	Gets all shortcuts for a user	-	JSON array of user shortcuts
	/shortcuts/user/new	Saves a new shortcut for a user	The data for the new shortcut	JSON object of the new shortcut
	/shortcuts/user/edit	Edits a current shortcut for a user	The ID of the shortcut to edit, alone with the new data	JSON object of the edited shortcut
Avoids	/avoids/draw	Gets data required to draw all avoids	User selection to draw personal and/or public avoids	JSON object with all avoids geocode data
	/avoids/edge	Gets a new edge while building an avoid	JSON object with the current edges in the avoid	JSON object of the avoid with the new edge
	/avoids/user/get	Gets a single avoid for a user	The avoid ID	JSON object of the avoid
	/avoids/user/all	Gets all avoids for a user	-	JSON array of user avoids
	/avoids/user/new	Saves a new avoid for a user	The data for the new avoid	JSON object of the new avoid
	/avoids/user/edit	Edits a current avoid for a user	The ID of the avoid to edit, along with the new data	JSON object of the edited avoid

CHAPTER D

Gantt Chart

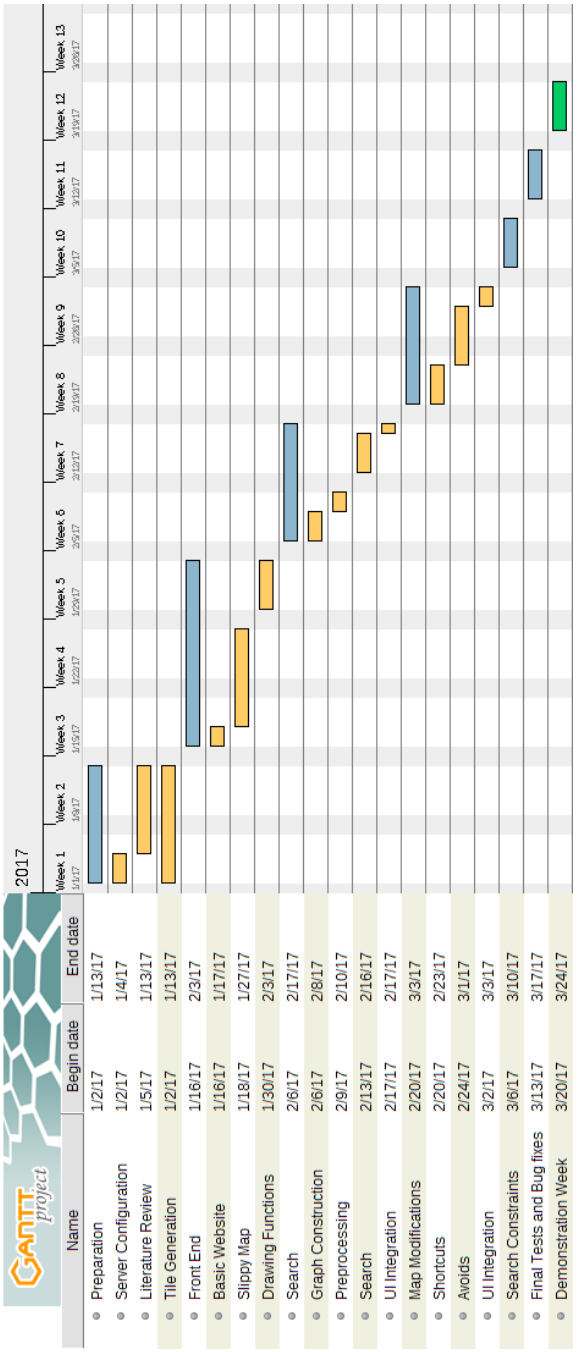


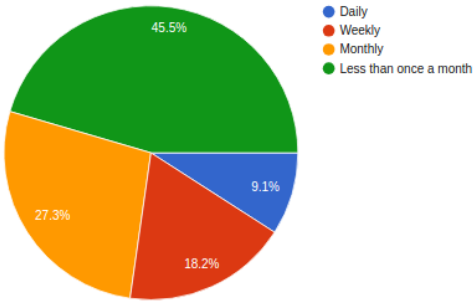
Figure D.1: caption

CHAPTER E

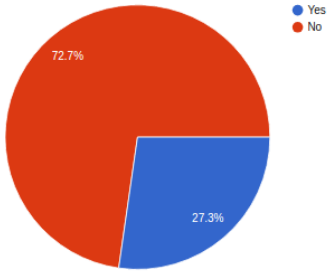
User Feedback Results

Note: A higher score on the bar chart represents more positive result.

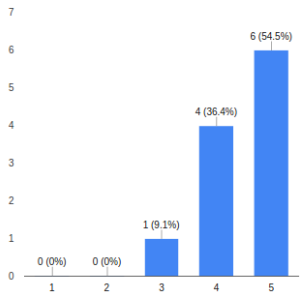
How frequently do you cycle? (11 responses)



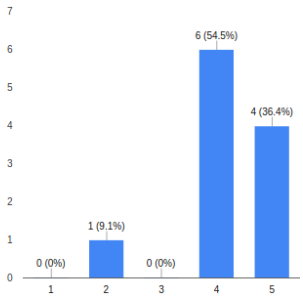
Have you used any route planners specifically for cycling in the past?
(11 responses)



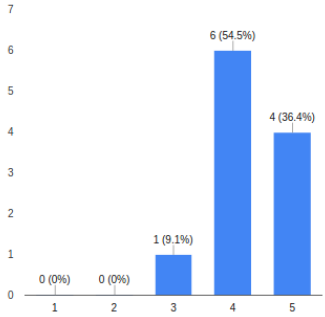
Try to plan a route between two waypoints. How easy did you find this task?
(11 responses)



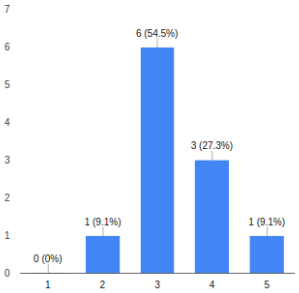
If you were successful in the previous task, try to adjust one of the waypoints you have added. How easy did you find this task?
(11 responses)



The route was delivered in sufficient time (11 responses)

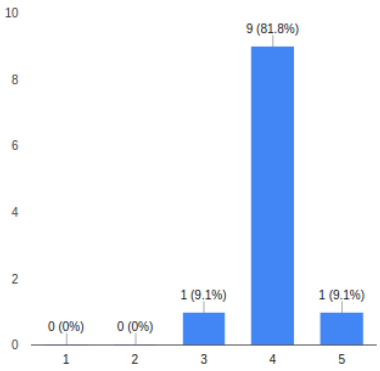


If you were able to plot a route, how comfortable would you feel cycling on this route?
(11 responses)



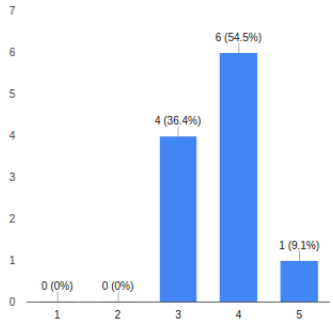
Now adjust the route using the "Options" on the lefthand side to avoid main roads, or prefer cycleways. Move a waypoint so the route is recalculated. How comfortable would you feel cycling this new route?

(11 responses)



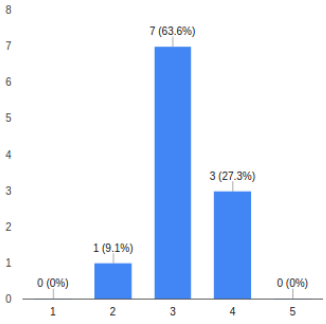
Try to insert a new shortcut onto the map. How easy did you find this task?

(11 responses)



Try to insert an area to avoid onto the map. How easy did you find this task?

(11 responses)



Have you got any other comments about the system? (7 responses)

It would be good to have some indication of which road will be added to the avoid, at the moment it's a bit of a shot in the dark and it doesn't seem very consistent.
not a huge cyclist so want to avoid roads as much as possible, would be better if it allowed me to just follow off route tracks/canals
no real explanation what shortcuts and avoids do?
I think giving an elevation profile of the route would be beneficial, it doesn't seem to account for steep hills
Avoids are a bit awkward to use, but otherwise a pretty good idea
should be an easier way to add waypoints rather than just putting postcode
Well laid out, easy to use. Would be useful to be able to search a town or city e.g Stourbridge, rather than using a postcode
great dude, flawless :D