# Max Bowman NEA

## Optimised Safe Route Finder For Cyclists

Last updated 07:44, Friday 25th March, 2022
Pages: 70
Written in LaTeX

./images//coverimage.png

# Contents

# 1 Analysis

## 1.1 Problem Area

There exist a plethora of route finding services for finding the shortest route between two points. These typically optimise for the shortest time taken to travel between two points. Provision for cyclists can be lacking, because it typically consists of an alteration to the existing code for cars, slightly modified to allow for cycle paths and different timings for cyclists.

Many of these route finding applications don't take into account safety considerations. This is shown in Figure 1.1 which shows the route Google Maps suggests that I cycle to school by. The route includes the A4, which is a very dangerous stretch of road for cyclists as it is a 3 lane road. My plan is to use accident statistics to work out which roads are dangerous and then avoid them. This would be done by imposing a cost for going somewhere that there had been an accident.

## 1.2 Client

I interviewed Yuvraj Dubey, one of my classmates, to talk about whether they would be interested in a product that attempted to calculate safe routes to cycle.

MAX: Do you own a bicycle?

YUVRAJ: Yes

MAX: Do you cycle regularly?

YUVRAJ: Yes

MAX: Do you ever feel in danger while cycling?

YUVRAJ: Yes

MAX: When do you feel in most danger while cycling?

YUVRAJ: At a Junction near Victoria, where there is a right turn and no cycle lane to do it in. I attempted to go this way once and was almost hit by a bus.

MAX: Are you aware of your surroundings while cycling?

YUVRAJ: Not really

MAX: Do you feel aware of the places which to cycle in safely, does this change when you are in places you commonly go?

YUVRAJ: No, but especially where I don't know where I am because then I do not know where it is safe to cycle. When I cycle home I have learnt a safe route, but if asked to cycle somewhere I did not know I would most likely end up in a dangerous place.

MAX: Would you use an application that tells you safe routes to get places?

YUVRAJ: Yes

MAX: Would you be happy using a command line application for this?

YUVRAJ: Yes

MAX: If said application took more than 60 seconds to calculate a route would you lose patience?

YUVRAJ: Yes, probably

MAX: Have you found that cycling directions generated by current products are safe and efficient?

YUVRAJ: They are definitely efficient, but they often take me onto busy roads and junctions which can be less safe.

./images//route.jpg ./images//dangerous.png

Figure 1.1: Route suggested by google maps

### 1.2.1 Interview Analysis

From this interview and my own experiences cycling in London I determined that there was an application for my idea, but only if it ran in a reasonable amount of time and was otherwise easy to use, as people struggle to find safe routes themselves, especially when going somewhere they had not previously been, and commercial products such as Google Maps don't generate safe routes. Add a little bit more analysis to this bit

## 1.3 Similar Systems

./images//citymapper.png ./images//googlemaps.png

Figure 1.2: Citymapper and Google Maps respectively, generating the same directions

There exist similar systems for route finding such as Google Maps and Citymapper. These have advantage's over what I will be able to offer such as being able to use live traffic data, and having a good user interface. Citymapper has a option to choose between "Quiet", "Regular", and "Fast" routes, which seems to be based on the type of road you are taken down. However they don't really take accident density into account which will be my aims.

## 1.4 Features

### 1.4.1 Map data

I need a data source which can provide data on roads that are legal to cycle on as well as being freely available for me to use. I settled on Open Street Map, a project which combines data gathered by volunteers into one massive freely available map. The map is downloadable in the form of a large XML file or PBF file. A PBF file is just a binary version of the same thing. I will write code to parse one of these myself.

### 1.4.2 Accident Data

Transport for London has an excellent API which you can download accident data from. The data comes in JSON files and contains information about what type of vehicle was involved in each accident, the severity, and the coordinates of the accident. I need to find a way of mapping the coordinates onto the road network so that the danger of roads can be calculated as accurately as possible. Originally I was looking at adding the accident to the nearest edge in the graph but I decided against this after considering

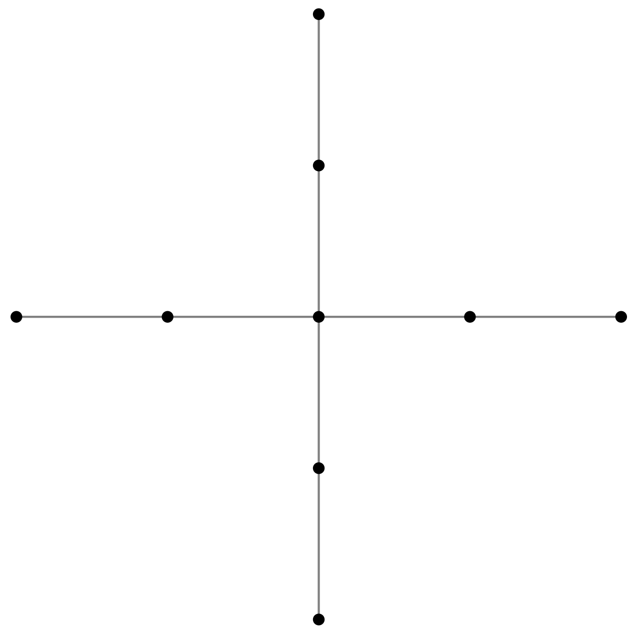

Figure 1.3: Typical OSM representation of an intersection

how intersections are typically represented in OSM. Road intersections typically look something like what is represented in Figure 1.3. If an accident were to occur at the intersection it would end up being added to one of the segments leading into the intersection, so the danger would not be properly calculated if not passing through that segment. Instead I thought about adding the danger to the nearest node, so that the accident would always be counted when a route passes through that intersection. The other problem that I have to deal with, if adding data to the nearest node is that the density of nodes is not constant. Straighter roads will not need to use as many OSM nodes as curved roads, so i might incorrectly add cost to the wrong node. My proposed solution to this is to interpolate in nodes to a very high density to deal with this problem.

### 1.4.3 Traffic Data

The main problem with this is that the accident data is absolute and can thus not be used to calculate probabilities. For example, more accidents happen on King's Street than the dangerous road I showed earlier, but this doesn't mean that King's Street is more dangerous merely that more cyclists travel on it. This means that I need to get accurate cyclist traffic data for the whole of London in order to turn my accident statistics into accident probabilities. The Department for Transport and the Office for National Statistics both keep data on traffic, but it isn't applicable because cycle data is only given as a total [1] and at specific count points. This means that I will either need to work out traffic data or get it from some dataset, such as Strava's Global Heatmap.
Luckily the rest of the application can work without including traffic data so my plan is to deal with this problem at a later time or hope that the avoidance of accidents alone will be enough.

### 1.4.4 Finding the shortest route

Algorithms for finding the shortest path in a Graph are abundant. The most well known is Dijkstra's algorithm, and it's variant A*. In large Graphs, both Dijkstra's algorithm and A* can be very slow. I will most likely use preprocessing based algorithms such as ALT*/Contraction Hierachies to make

[1] 0.6 billion miles per year in London

./images//classdiagram.png

Figure 1.4: Rough idea of the class diagram I will use

my program run a lot faster. This preprocessing will take a long time, but when completed it will significantly reduce the time taken to complete searches.

## 1.5 Class Layout

As seen in Figure 1.4, I think I will use 3 major classes which will deal with parsing, storing, and preprocessing Graph data respectively. These will be linked by object composition rather than inheritance. Where possible I will use private fields and methods.

## 1.6 Critical Path

- Download accidents from the TFL API, parse for accidents that are of interest, and store it in a format that can be easily parsed later
- Import OSM map of London as a Graph which can be easily queried.
- Write code that imports the accident data and connects it to given Graph nodes.
- Write a simple Dijkstra's Algorithm method of finding the shortest path.
- Write a preprocessing based method of finding the shortest path.
- Write a frontend for the input of user options.

## 1.7 Specification

1. Accident Download System
   - 1.1 The System must be capable of interfacing with the TFL API to download accidents
   - 1.2 The System must be capable of parsing accident data to determine which accidents are relevant
   - 1.3 The System must be capable of storing accident data in a easily accessible file for the route finding algorithm to parse
2. Route Finding System
   - 2.1 The System should be capable of processing an Open Street Map map file into a suitable data structure
   - 2.2 The System should be capable of processing the accident data from the Accident Download System
   - 2.3 The System should be capable of attaching the data from the Accident Download System to the suitable data structure from specification point 2.1
   - 2.4 The System should be capable of using Accident Data combined with other suitable cost estimation functions to find a directed route between two points that are places on roads in London
   - 2.5 The System should always suggest a route that can be followed while observing all currently known laws of physics
   - 2.6 The System should always suggest a route that can be followed while observing all international, national, local laws and all releveant bylaws
   - 2.7 The System should suggest a safe route wherever possible
   - 2.8 The System should have guards in place for certain types of roads which are deemed too dangerous to consider
   - 2.9 The System should be able to generate this route quickly
   - 2.10 The System should have a mode for preprocessing to make the route generation even quicker
   - 2.11 When using the same parameters, the route generated using a preprocessed graph and associated algorithms should be the same as that generated by the non-preprocessing based approach
   - 2.12 The System should be easy to use
   - 2.13 The System should not crash, and should give appropriate non crashing errors if user data is determined to be bad
   - 2.14 The parameters for the cost estimation functions should be user definable, but sensible defaults should also be defined
   - 2.15 The Route should be output in a format that is easy for the user to follow on a mobile device
   - 2.16 The System should have a system for saving all data from intensive calculation to disk

## 1.8 Specification Justification

1. Accident Download System
    1.1 This point is required in order to obtain the necessary accident data
    1.2 This point is required to reduce the file sizes needed to be packaged with the project, and make it easier to use later
    1.3 This point is required as otherwise the system would not do much good in making the Route Finding System simpler
2. Route Finding System
    2.1 The suitable data structure is required for all further queries
    2.2 The accident data is required for route finding applications
    2.3 This is required to tell where the costs for the accidents should be applied in shortest path queries
    2.4 This is the main function of the project; if it can't do that it will be worthless
    2.5 The Route should be realistic and connected; it can't ask people to teleport to their destination or phase through walls
    2.6 The Route should only take people onto roads which are publicly accessible and legal to cycle on, as obviously I do not want to encourage people to break laws
    2.7 The Route should attempt to be as safe as possible. Of course in some situations there is only one route which could be unsafe. In that situation that route would be suggested
    2.8 Some roads can be very dangerous to cyclists, such as canal paths, and these won't necessarily be represented through the data
    2.9 If the route generation takes too long the user will get bored. Other products offering the same features can generate routes very quickly
    2.10 London is very large, and route generation may be slow without preprocessing, so the preprocessing mode will help with that
    2.11 This makes sure that the preprocessing based approach is not losing information about the best route in any way
    2.12 If the system was not easy to use, people would not use it
    2.13 This feeds into the previous specification point
    2.14 The customisability would allow users to make their own decisions about tradeoffs between time and safety, and sensible defaults should generate sensible routes
    2.15 Especially with longer routes, the probability of the user being able to remember the route is slim, so they will need a mobile device to be able to guide them
    2.16 The parsing of XML files may take a while, and the preprocessing for query speedup definitely will. If the results of this can be saved to disk repeating this every time the program is run will not be necessary.

# 2 Design

## 2.1 Accident Download System

As outlined in the Specification, this system should be capable of interfacing with the TFL API, downloading the accidents, deciding which ones are relevant, and storing this in a useful format.

### 2.1.1 Pulling from the TFL API

The Transport For London API is an excellent API which can provide information on many different aspects of the Transport For London network. The specific API which I used is the AccidentStats API. This API is very simple, you simply request a given year and a JSON object is returned which contains all of the accidents that happened in london that year. These consist of all the accidents that were reported tocdepth the police as happening in that year. Of course, there will be many more accidents than are on the API, but these will mostly be more minor accidents. The Data is returned as a list of accidents, formatted as shown in Figure 2.1.
As the data about accidents that happened in the past is not going to change any time soon, and TFL only updates this API every year, it is simplest just to download the files once, parse it once, and then use that result in the route finder. I used two scripts to do this.
TFL started gathering this data in 2005, and the most recent update was in 2019, so the first script downloads all the data from 2005 to 2019 and save it in a subfolder called `accidents`. The script loops through all the years between 2005 and 2019 and creates a JSON file for that year.

### 2.1.2 Parsing the Data

The next step is to go through the data from all the years[1], and save all the accidents that are pertinent to my project. As seen in Figure 2.1, each accident listing has information on casualties,
of which there may be many. As this is a cycling application, I only want data on accidents where at least one of the casualties was a cyclist. The API provides a lot of data, but all that is relevant is the latitude and longitude, as well as the severity of the incident. All of this data was ultimately saved to a file called `output.json`. The intended format of the file data will be saved in is shown in Figure 2.2.

[1] I later decided to remove some years see Section 3.2

```json
[{
    "id": 0,
    "lat": 0.0,
    "lon": 0.0,
    "location": "string",
    "date": "string",
    "severity": "string",
    "borough": "string",
    "casualties": [{
        "age": 0,
        "class": "string",
        "severity": "string",
        "mode": "string",
        "ageBand": "string"
    }],
    "vehicles": [{
        "type": "string"
    }]
}]

```

Figure 2.1: The Default output from the API [1]

```json
[
    ["Latitude","Longitude","Severity"]
]
```

Figure 2.2: Intended format for parsed accident file

## 2.2 Route Finding System

### 2.2.1 Overview and Design Choices

I decided to use Kotlin for this project. Kotlin is related to Java in that it allows access to all the Standard Java Libraries, as well as external ones, and it runs in the Java Virtual Machine. Kotlin does not need to maintain backwards compatability with old Java code, so it has a much cleaner API and has nice syntax. As this was a new project not dependent on using Java code Kotlin was an obvious choice. Furthermore, Java code can be easily translated into Kotlin code, so I could code things in Java then translate over if necessary. Like Java, kotlin allows Object-Oriented-Programming, so that is the approach I will be taking in this project.
As seen in Figure 2.3, I decided to define 3 main classes for my NEA. `OpenStreetMap` contains methods for parsing OSM files and creating a graph. `OpenStreetMap` contains a single instance of the `GeographicGraph` class by object composition. This is because there will only be one `GeographicGraph` class associated with a given OSM file, and this system could be easily extended to hold information on two separate graph areas at the same time, which could be used if I wanted to extend the system to more cities. The `GeographicGraph` can contain an instance of the `ContractableGraph` class, if it has been contracted. If it has not been contracted, this class will not exist.
Upon further thought, I decided to change this to allow for many `ContractableGraph` instances to be contained within a single instance of `GeographicGraph`, as there is no reason that a new `GeographicGraph` should need to be created if a contracted graph at a different distance cost was required.

./images//classdiagram.png

Figure 2.3: Rough UML diagram for the project

### 2.2.2 Open Street Map Data

Open Street Map is a world map generated by user mapping. It contains multitudes of data on all sorts of mappable things, from the height of stories to the roads that encompass them. Files can be in one of two formats; PBF [2] which is a highly efficient binary format, and XML[3] which is a markdown language. I tried to use a library to parse the PBF file into a Graph, but the main library for that did not work. So instead I decided to parse it manually as an XML file.

#### 2.2.2.1 Getting the relevant file

Openstreetmap has an API for requesting parts of the map, but it does not allow requesting large areas, such as the whole of London. There are many mirrors from which you can download large parts of the world. I decided to use geofabrik [2] because it allows you to download single countries. Next I extracted London from this dataset. This is not strictly necessary, as my program can still deal with large areas such as the whole of the UK, but it does not have accident data for such an area. Furthermore it would increase the file size needed on disk and increase the running time of any preprocessing. In order to extract London, I used a GeoJSON[3] file, which is essentially a list of coordinates. I found a GeoJSON file on the internet [4] of the M25 boundary, and used the following command[4] to cut out London:

```
$osmium extract -p course_m25_boundary.json united_kingdom.osm.pbf -o london.osm
```

This file could be used by my program, as it intelligently avoids parsing non-routable ways and, but I decided to further reduce the file size by removing all nodes and ways that were not part of highways. This was just for quality of life, as it makes parsing the file much faster.

```
$osmium tags-filter london.osm nw/highway -o ways.osm
```

This created a 370MB file called `ways.osm`. I could have reduced this further by removing the ways that are irrelevant to the program, but I decided to do that in the parser so that it would be more easily configurable.

#### 2.2.2.2 Parsing the file

Open Street Map files are built out of 3 main elements [5]:

- Nodes, which contain longitude and latitude.
- Ways, which contain between 2 and 2000 ordered nodes. These are used to form roads, but also all kinds of other polygons, such as buildings, rivers, and walls.
- Relations, which contain many ways, nodes and relations. These are used to connect things together, like all roads on a certain bus route. These are not relevant to my project.

All of these elements can also contain additional information on what type of node or way it is, as well as a unique id. As I only care about nodes that are part of roads I can ignore the relation element completely. In a `.osm` file, the nodes all come first. This means that I can parse the nodes, and then figure out how all the nodes are connected using the ways.

After some research, I determined that the best way to parse the XML was to use the library `dom4j` [6].

[2]Protocol Buffer Format
[3]Extensible Markup Language
[4]I used the OSM manipulation tool Osmium

```xml
<way id="1202" version="32" timestamp="2019-05-15T08:24:07Z" uid="7105697" user="_Garrison_" changeset="70264933">
    <nd ref="5335693253"/>
    <nd ref="5335693250"/>
    <nd ref="104429"/>
    <nd ref="3330244696"/>
    <nd ref="5335691516"/>
    <tag k="bicycle" v="no"/>
    <tag k="foot" v="no"/>
    <tag k="highway" v="trunk"/>
    <tag k="horse" v="no"/>
    <tag k="lanes" v="3"/>
    <tag k="lit" v="yes"/>
    <tag k="maxspeed" v="40 mph"/>
    <tag k="maxspeed:enforcement" v="average"/>
    <tag k="name" v="North Circular Road"/>
    <tag k="oneway" v="yes"/>
    <tag k="operator" v="Transport for London"/>
    <tag k="ref" v="A406"/>
    <tag k="sidewalk" v="none"/>
    <tag k="surface" v="asphalt"/>
</way>
```

Figure 2.4: One of the Ways in `ways.osm` extracted

I decided that the `OpenStreetMap` class needs 3 functions: `parseXML`, `processNode`, and `processWay`. The function that is called with the direct input file is `.parseXML` Essentially what it does is iterate through the items in the XML file, calling either `processNode` or `processWay`. `processNode` simply adds a new node to the `GeographicGraph`. The `processWay` function is more complex as it has to work out whether ways should be added to the graph, and if additional data needs to be stored about them. Within each way stored in the `.osm` file, there will be additional tags that hold more data about the way [7]. An example way, which is part of the North Circular road, is shown in Figure 2.4.
First there are a ordered list of nodes which are sequentially connected, then there is a collection of optional Key Value pairs which form tags. The `processWay` function analyzes these tags in order to determine if the road is acceptable for cyclists. The example in Figure 2.4 would probably not be a good way to include, because the tag `bicycle` is set to `no`, and it is a 3 lane road with a 40mph speed limit. The progam uses accident data to work out which roads are dangerous, but some roads will not have many accidents on them simply because they are so dangerous nobody would ever cycle on them.

**2.2.2.2.1 Parsing the ways** In order to convert from XML to a data representation in memory, we simply iterate through all the subelements in the way. If it is a reference to a node, the id is appended to a ordered list, and if it is a tag, it is added to a HashMap so it can be more easily queried.

#### 2.2.2.2.2 Deciding what is allowed

- Highway tag
  - This tag is present in all roads mapped in OSM, so it's presence must be checked for.
  - This tag represents the "the importance of the highway within the road network as a whole"[7].
  - This does not normally represent anything about the road quality, safety, usage, layout, or maximum speed. The exceptions to this are
    - `motorway` which is obviously not wanted.

  * `living_street` which represents places where pedestrians and cyclists have legal priority, such as Low Traffic Neighbourhoods.
  * `cycleway` which encompasses segregated areas for cyclists.
  * `bridleway` which is a segregated area for horses, where cyclists may be allowed.
  * `footway` which is a footpath which cannot be cycled on.

- Access tag. This tag represents what the access arangements for the area in question are. If it is `no` or `private`, the way would not be wanted.
- Bicycle and Motor tags. These tags are the same as the access tag, but for specific types of vehicle.
- Surface tag. This represents the surface the route is made out of. Ways that are made out of dirt or similar materials will not be parsed.
- Note tag. This can contain lots of different notes, but the one `processWay` looks for is the `towpath`, as these can be dangerous in a way that is not represented in the TFL accident data
- Oneway tag. If true, the route will only be connected oneway.
- Maxpseed tag. This can be used to exclude dangerous roads.

After analyzing these tags `processWay` has multiple outcomes.

- Way is included in the graph in both directions.
- Way is included in the graph only in the same direction as the ordered list of nodes, because `oneway` is set to `yes`.
- Way is considered too dangerous to include or access to cyclists is not guaranteed to be legal so specification point 2.5 would not be followed.
- Way is included in the graph and the nodes are added to a set of safe nodes because it is in a cycle route or footpath.
- Way is included in the graph and the nodes are added to a set of nodes that will be slower to travel on because of a footpath.

Some of these outcomes can overlap, such as being added to the group of safe nodes and having edges only added in one direction.

1. The way is parsed in some form if:

   The `highway` value is not null, and is within a set of allowed highway types

   The `access` and `bicycle`[5] value are not within a set of disallowed access types.

   The `surface` value is not within a list of disallowed surfaces

   The `maxspeed` value is not more than 30 miles per hour

   The `note` value does not say that the path is a towpath

2. The way is only parsed in one direction if:

   The `oneway` tag is set to yes

[5]In places where cycling is not permitted but walking with a bike is this will be dismount instead of no

3. The way is marked as a safe node if it is

   Marked as a `cycleway`, `footway` ,or `pedestrian` or `motor_vehicle` is set to private

4. The way is marked as a slow way if it is

   Marked as a `footway` or `pedestrian` and `bicycle` is not set to `designated`

Of course, it will be very easy to iteratively develop this when the whole system is finished. If a certain way that looks like it should be routed down is not routed down or vice versa the function can be adjusted. It is important to note that this function is designed to be quite open in deciding which ways are safe. It only excludes ways that are plainly very dangerous. The actual decision of where to go based on danger will be made by the route finding algorithms.

### 2.2.3 Representing and Querying the Graph

As shown in Figure 2.3, my design uses a class called `GeographicGraph` to represent London[6]. There are many graph libraries available to java programs, but creating my own implementation will allow me more flexibility in implementation of methods that relate to real world applications. As shown in Figure 2.3, the `GeographicGraph` class has some major methods that need to be implemented. `gatherWeights` will read in a accident file in the format specified in Figure 2.2, and attach the accident to the nearest node in the graph. `findRoute` will find the best route between two nodes in the graph. There will also need to be other functions for maintaining and querying the data structure.

#### 2.2.3.1 Representing the Graph

There are many possible methods of representing Graphs, such as using an adjacency matrix, or an adjacency list. An adjacency matrix would not work in this case, as the graph is very sparse. An adjacency list could work, but maintaining information about the nodes separately from the adjacency list would be difficult. Instead, my design uses an OOP approach, where every relevant node in the OSM representation is converted into an equivalent node represented by a subclass called `GeographicNode`. This is advantageous because it allows associated data, such as the number of accidents attached and the coordinates, to be stored in the same place. As shown in Figure 2.5, the connections would be represented as a HashSet. Ideally, this would contain other `GeographicNode` objects that the node in question was connected to. However this would lead to problems with serialisation as explained in Section 2.2.5 so instead the unique ids from the OSM representation are used, along with a lookup table for converting these back to `GeographicNode` objects.
Another feature that the `GeographicGraph` class needs to provide is facility for the conversion of coordinates to the id of the closest node. This will be useful for allowing the user to ask for directions in terms of coordinates, as well as for attaching accidents to the nearest node. In order to do this quickly and efficiently my design calls for the use of an R*Tree[8]. This is a special type of data structure for holding coordinates that groups coordinates into a tree of rectangles. This allows much faster queries when looking for nodes that satisfy certain geometric properties, such as asking for the closest node. The * in R*Tree means that a heuristic is used to try an optimise query times, by minimising overlap between rectangles. The implementation of an R*Tree is beyond the bounds of this project, and any I could implement would be less powerful than one from a library. To that end, I decided to use the library `rtree2`[9]. I have used it in other projects before and found it to be very powerful. The R*-Tree will be set up when the graph is created, and a function called `getNearestNode` will convert between coordinates and OSM node ids.

#### 2.2.3.2 Creating the Graph

The `OpenStreetMap` class deals with parsing the XML, but it has to interface with the `GeographicGraph` class. When adding new nodes to the Graph, it can simply create a new instance of `GeographicNode` and add it to the relevant lookup table. For adding an edge, the functionality can be provided by a `addEdge` function.

**2.2.3.2.1 Pruning the Graph** Due to the sequential order of parsing the OSM representation of the Graph, some nodes may be left disconnected because the way that connects them together was not parsed. If the system is asked to find a route from a node that is not connected to any other node, it will of course fail. This could lead to a crash, which would be in contravention of specification point 2.13, or at least a bad user experience, if given an error for seemingly sensible inputs. Another problem could be that the Graph might have multiple components, perhaps because access to an area was cut

[6]Other areas could be represented as well

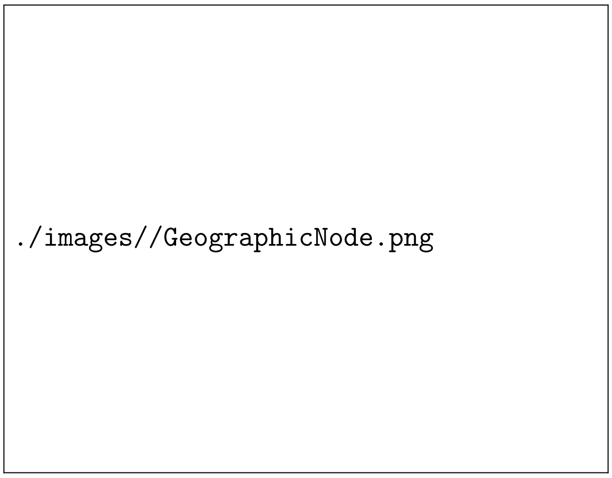


Figure 2.5: Rough UML class diagram for the `GeographicNode` subclass

off by safety protocols. Whatever the cause, both of these problems can be averted by removing all nodes that cannot be routed to by one location in the main component. This can be done by picking any node from which all other nodes can be connected to, finding the whole component, then removing any `GeographicNode` objects that are not in that component from the lookup table they are stored in. However there are two problems with this approach: the node that all other nodes are connected to must be picked manually, and the component is not guaranteed to be strongly connected. What this means is that while it is possible to reach any node in the component from the start node it is not necessarily for every node in the component. A component for which this would work is called a strongly connected component. If the function finds all strongly connected components and then assumes that the largest one is the one that is wanted the chance of allowing unsolvable queries is eliminated. One algorithm for finding all strongly connected components is Kosaraju's Algorithm[10]. This algorithm uses the fact that strongly connected components remain when all edges are reversed, and relies on a post-order DFS. The DFS is run first in the right direction, then in reverse to find the strongly connected components. The following steps are followed.

```
L <- []
visited <- []
FOR u in G:
    Visit(u)
Subroutine Visit(u):
    if u NOT IN visited:
        ADD u TO visited
        for each neighbour v:
            Visit(v)
        prepend u to L
FOR u in L:
    Assign(u,u)
Subroutine Assign(u,v):
    if u NOT assigned:
        assign u to v
        for each incoming neighbour p:
            Assign(p,v)
```

The ordering of L is important as it represents the direction in which connections are made in the components. These components are then checked to find the strongly connected components. Each strongly connected component will take a continuous region of L. Implementation of this may require modifying the `GeographicNode` class to also keep track of incoming connections.

```
function Dijkstra(Graph, source, target):
    for each vertex v in Graph.Vertices:
        dist[v] <- INFINITY
        prev[v] <- UNDEFINED
        add v to Q
    dist[source] <- 0
    while Q is not empty:
         u <- vertex in Q with min dist[u]
         remove u from Q
         if u == target: break
         for each neighbor v of u still in Q:
             alt <- dist[u] + Graph.Edges(u, v)
             if alt < dist[v]:
                 dist[v] <- alt
                 prev[v] <- u
  S <- empty sequence
  u <- target
  if prev[u] is defined or u = source:
      while u is defined:
          insert u at the beginning of S
          u <- prev[u]
```

Figure 2.6: Dijkstra's algorithm modified from Wikipedia[11]

#### 2.2.3.3 Querying the Graph

The main query that has to be provided for is finding the shortest route between two points. This functionality can be provided either by a simple Dijkstra's Algorithm implementation, or through a different query for the preprocessed Graph. If the graph has been preprocessed, that mode should be used. Otherwise Dijkstra's shortest path algorithm should be used.

**2.2.3.3.1 Dijkstra's Algorithm** Dijkstra's Algorithm[12] is a immensely popular algorithm for finding routes in graphs. It builds up a tree of nodes that it knows the shortest route to until said tree contains the target node. It does this by iterative "settling" of nodes. The known unsettled node that has the shortest route cost to the start node is settled, and all of it's neighbours are added to a list of nodes to be settled. Every node has a tentative `dist` value and a tentative `prev` value, defined by the currently known shortest path to that route. When the node is settled this value is known to be true. This process is repeated until the algorithm is completed. Pseudocode for Dijkstra's algorithm is shown in Figure 2.6. A Priority Queue can be used to find the next node to process. This is detailed in the Final Design section.
A variant of Dijkstra's algorithm called A*[13] uses heuristics to reduce the required search space. The heuristic estimates the distance to the end node, meaning that nodes are vaguely settled in the right direction. However this heuristic has to be "admissible" if the shortest possible route is to be found. This means that it never makes overestimates. As the density of accidents cannot be predicted between any point and the end node, the heuristic would be extremely loosely fitting, and would fail to significantly reduce the search space.

**2.2.3.3.2 Cost Function** Any route finding algorithm needs a function or set of functions for the cost that would be incurred if going a certain way. If going from node $u$ to node $v$, the function $c$ would

./images//cosinediagram.png

Figure 2.7: Triangle used to illustrate the cosine rule [14]

be:

$$
\begin{aligned}
c(u,v) &= w(v) + a \times s(u,v) \times d(u,v) + b \times t(p(u),u,v) \\
w(x) &= \text{slight}(x) + 2 \times \text{serious}(x) + 3 \times \text{fatal}(x) \\
t(a,b,c) &= \begin{cases} 1 & \text{if } \angle abc < c \\ 0 & \text{otherwise} \end{cases} \\
s(u,v) &= \begin{cases} \text{d} & \text{if both nodes are safe nodes} \\ \text{e} & \text{if both nodes are slow nodes} \\ \text{f} & \text{otherwise} \end{cases}
\end{aligned}
$$

where $a$,$b$,$c$,$d$,$e$,and $f$ are variables that can be tweaked. The turn function $t(a,b,c)$ can be used to impose a cost on turning too often, as this is makes routes harder to follow and increases danger. $\angle abc$ can be calculated using the cosine rule. For the triangle shown in Figure 2.7

$$
\begin{aligned}
a^2 &= b^2 + c^2 - 2bc\cos\alpha \\
\therefore \cos\alpha &= \frac{b^2 + c^2 - a^2}{2bc} \\
\therefore \alpha &= \arccos\left(\frac{b^2 + c^2 - a^2}{2bc}\right)
\end{aligned}
$$

**2.2.3.3.3 Calculating Distances** To calculate real world distances[7] from coordinates, Great Circle distance is required. This is the shortest possible distance across the surface of a sphere. The haversine formula[15] is one formula that calculates this distance. For two points $(\phi_1, \lambda_1)$ and $(\phi_2, \lambda_2)$ where $\phi$ and $\lambda$ are latitude and longitude respectively, the great circle distance can be calculated as

$$
d = 2r \arcsin \sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos\phi_1 \cos\phi_2 \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}
$$

where $r$ is the radius of the earth, so approximately 6371 km.

### 2.2.4 Preprocessing and Subsequent Querying of the Graph

There are many different methods of route finding[16]. In the Analysis section I considered both Contraction Hierachies[17] and ALT*. Upon further research, I determined that contraction hierachies would be more efficient than ALT* in this situation.

[7] Assuming the world is a sphere

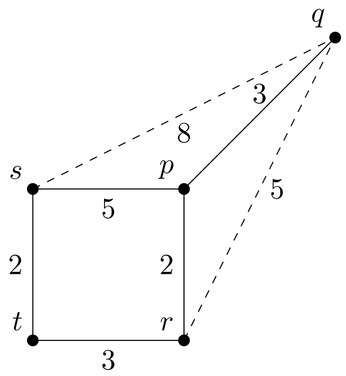


Figure 2.8: The shortcuts that would be added while contracting node $p$

#### 2.2.4.1 Contraction Hierachies

In most road networks there is an inherently hierachical structure. When a human plots a long distance route, they don't consider small dirt tracks, and instead choose the best motorway. This is because in most cases motorways are faster. I do not expect the same level of hierachy in figuring out where is cyclable, because cyclists do not have the same limiting factor of the speed limit. However there will still be some routes that are obviously the best, at least to my program.

Contraction hierachies work by the process of iterative contraction of nodes. Some order of contraction is defined, in order of least to most important, and the nodes are contracted in this order. When a node is contracted the shortest paths are calculated between all pairs of neighbour nodes. If the shortest path goes through the node to be contracted a new shortcut is added. This maintains all shortest paths in the Graph. An example in an undirected graph is shown in Figure 2.8.

**2.2.4.1.1 Bidirectional Dijkstra Queries** The bidirectional variant of dijkstra is a useful speedup technique. A forward search from the source node is interlaced with a backward search from the target node. When a node is settled by both searches all nodes on the shortest path are known to have been settled. This variant can reduce the search space, and ideas from this search are used when querying the contracted graph.

**2.2.4.1.2 Querying the Contracted Graph** When contraction is complete, a new graph is created called G*. This Graph contains all of the nodes and edges in the original graph, as well as all of the shortcuts created during the contraction process. All nodes have a new item of information attached to them, known as the hierachy. The first node to be contracted will have a hierachy of 1, and the last node to be contracted will have a hierachy of $n$. Two new graphs can be developed from the hierachy: $G{*}\uparrow$ and $G{*}\downarrow$. $G{*}\uparrow$ contains only edges where the hierachy increases, and $G{*}\downarrow$ contains the opposite. In order to find the shortest path between two nodes $u$ and $v$, a bidirectional search consisting of a forward search in $G{*}\uparrow$ and a backward search in $G{*}\downarrow$. In an undirected graph this is the same as two forward searches in $G{*}\uparrow$, but in a directed graph the backwards search must go backwards on directed edges. Implementing this will require the `GeographicNode` equivalent class in `ContractableGraph` to contain incoming connections and shortcuts. It has been proven[17] that this method of query will leave at least one node on the shortest path settled by both the forward and backward searches.

**2.2.4.1.3 Finding the shortest route** Once the query is completed, a node on the shortest path will be in the intersection of both searches. The total distance of a route containing a given node is the sum of the distance from the forward search and the distance from the backward search. So to find a node on the shortest path, the minimum total distance cost in the intersection of both searches must be found. From there the `prev` values can be used to backtrack to the start and end of the route. At this point the route will almost certainly contain shortcuts, which must be somehow expanded into their original form.

**2.2.4.1.4 Limitations** When using contraction hierachies, turn costs cannot be as easily integrated because the cost function must be per edge not for collections of edges. It is possible to use a second graph which has the original graph's edges as nodes, but this approach would slow down contraction, and add more complexity to the project. Therefore the ability to consider turn costs will only be added to the simple dijkstra searches and any turn costs will be ignored in the preprocessing mode, and queries will only be made if the input turn cost is set to 0.

**2.2.4.1.5 Node Order** The optimal node ordering is the one that minimises the expected query time. This can be estimated to be the same as the ordering that leads to the lowest total amount of shortcuts being added, which will in turn minimise the expected search space size. Finding the best ordering is an NP-hard problem, as it requires simulating every possible node ordering to find which one gives the best results. Instead the contraction algorithm will use a heuristic based approach. There are many heuristics of varying complexity, but the two that my design uses are the difference between the amount of shortcuts added and edges removed, and the number of deleted neighbours. Keeping the number of deleted neighbours low makes sure the contraction is spread out across the graph. When contraction is not uniform, very poor performance is attained. The edge difference is essentially a metric of how simple the graph would be after the contraction, so it makes sense to greedily make the graph as simple as possible. The edge difference heuristic can be calculated by simulating the contraction of a node. The deleted neighbours heuristic can only go up as the graph is contracted, and the same tends to happen for the difference of edges. Therefore it makes sense to load all of the nodes into a priority queue by their heuristic values, and pull out the minimum node. If the heuristic value has changed since the heuristic was placed into the priority queue the node is reinserted into the queue with the new value, otherwise it is contracted.

**2.2.4.1.6 Optimising Contraction** In order to contract a node, and compute the edge difference, it needs to be ascertained which pairs of neighbour nodes have their shortest path going through the node. If checking individually for each pair, there would be $O(n^2)$ queries. Instead a variant of dijkstra's algorithm can be used, with a stopping condition of settling all the other neighbour nodes instead of just one. These queries might take slightly longer, but they would find all $n$ neighbour nodes. Another useful optimisation technique is to refill the priority queue every so often, for instance when lots of re-insertions have to happen.

### 2.2.5 Saving the work done to disk

So that intensive workloads such as contracting the graph do not need to be repeated a facility for saving and loading this information must exist. It would be possible to write some manual serialisation code, but it is simpler to just save the whole class objects. Kotlin has a fairly easy to use serialisation library that can serialise whole objects to disk[18]. The standard mode for this is serialising to JSON, but as my files will likely be very large, my design uses the alternate CBOR encoding instead. As mentioned previously, `GeographicNode` objects cannot contain other `GeographicNode` objects. This is because the kotlin serialiser will get stuck recursively serialising the same nodes. CBOR[8] can be used for more consise binary representation of JSON objects.

[8]Concise Binary Object Representation

### 2.2.6 User Interface

My design calls for a simple terminal based user interface. Users should be able to load maps and preprocessed data from pre existing files, and trigger addtional preprocessing at different levels. This can be simply implemented by means of a loop where users can choose options or end the program.

### 2.2.7 Route Output

In order to fulfil specification point 2.15, the route must be output in a format that is not just a list of coordinates. Writing software that can use a route to give turn by turn directions, along with route recalculation when straying off the route is beyond the bounds of this project. However such software does already exist. One example of this is Osmand, an application that uses Open Street Map data to provide a routing application. Osmand can generate it's own routes, but it can also be used to follow routes stored in gpx files. GPX files are a subset of XML that can be used to store geographic data such as routes. Many other applications accept these, so a user would not be limited to Osmand. Writing to a GPX file manually would be boring, so my design calls for the use of a library.

## 2.3 Data Structure Overview

# 3 Technical Implementation

## 3.1 Overview

Figure 3.1 shows the complete file system. The `accidents` subfolder is used to generate `output.json`, and the `maps` folder contains different `osm` files for different areas. All of the `.py` files are used in the creation of `output.json` and all of the files in `src/main/kotlin` are make up the program as a whole, with `Main.kt` as the entry point.

## 3.2 Accident Download System

As this system only really needs to be run once, I just wrote some hacky python scripts to do this. As mentioned in Section 2.1.1, I used a simple script called `download.py` to pull all the relevant accident data from the TFL API and store it in the `accidents` folder.

```
import requests
import json
for year in range(2005,2020):
    print(year)
    response = requests.get(f"https://api.tfl.gov.uk/AccidentStats/{year}")
    with open(f"accidents/{year}.json","w") as yearfile:
        json.dump(response.json(),yearfile)
```

In order to extract useful information from these files, I used `parse.py` to extract all those accidents where one of the listed casulaties was a cyclist. While data exists between 2005 and 2019, when looking at some of the areas with high accident density, I found that the accident density had greatly decreased in recent years. This led to the realisation that the TFL accident data was not solely collected for the purpose of providing interesting data for A-Level NEA coursework, and that TFL was probably looking at the data as well and fixing those areas which were lethal. Thus I decided to only use accidents from after 2010.

```
import json
#we want to parse for accidents that happened to cyclists and store the severity and the location
accidents = []
for year in range(2010,2020):
    print(len(accidents), year)
    with open(f"accidents/{year}.json","r") as outfile:
        data = json.load(outfile)
    for x in data:
        try:
            if x["casualties"]["mode"] == "PedalCycle":
                accidents.append([x["lat"],x["lon"],x["severity"]])
        except:
            if len([person for person in x["casualties"] if person["mode"] == "PedalCycle"]) >= 1:
                accidents.append([x["lat"],x["lon"],x["severity"]])
with open("output.json","w") as outfile:
    json.dump(accidents,outfile)
```

I also wrote a small visualisation script using matplotlib to verify that the data was dense enough to be useful and imported correctly. I called this `display.py`.

```
import json
import matplotlib.pyplot as plt
```

```
$ tree
.
├── accidents
│   ├── 2005.json
│   ├── 2006.json
│   ├── 2007.json
│   ├── 2008.json
│   ├── 2009.json
│   ├── 2010.json
│   ├── 2011.json
│   ├── 2012.json
│   ├── 2013.json
│   ├── 2014.json
│   ├── 2015.json
│   ├── 2016.json
│   ├── 2017.json
│   ├── 2018.json
│   └── 2019.json
├── build.gradle.kts
├── download.py
├── gradlew
├── gradlew.bat
├── maps
│   ├── course_m25_boundary.json
│   ├── cyclable.osm
│   ├── cyclable.osm.pbf
│   ├── goldhawk.osm
│   ├── testarea.json
│   ├── testarea.osm
│   └── ways.osm
├── output.json
├── parse.py
├── display.py
├── route.gpx
├── savedGraph.bin
├── settings.gradle.kts
└── src
    └── main
        └── kotlin
            ├── ContractableGraph.kt
            ├── ContractableGraphTest.kt
            ├── DoubleTuple.kt
            ├── GeographicGraph.kt
            ├── GeographicGraphTest.kt
            ├── IntTuple.kt
            ├── Main.kt
            ├── MainKtTest.kt
            ├── OpenStreetMap.kt
            └── OpenStreetMapTest.kt
6 directories, 42 files
```

Figure 3.1: Project file structure. Automatically generated files have been omitted.

```python
x = []
y = []
with open("output.json","r") as inputfile:
    data = json.load(inputfile)
for accident in data:
    x.append(accident[1])
    y.append(accident[0])
    print(accident[0],accident[1])
plt.scatter(x,y,0.05)
plt.show()
```

./images//chart.png

Figure 3.2: Plot of all accidents in which there were injured cyclists in London since 2010

This generated the plot shown in Figure 3.2. Major roads are clearly visible, so a route finding algorithm will be able to avoid these. One problem that was apparent from the data was that there are significantly more accidents in central London then in the suburbs. Central London may be more dangerous, but there are also far more cyclists, and as I cannot get road usage data easily routes will probably be biased against central london, possibly causing them to go around. Hopefully, the distance cost will pull routes back towards central London, and they will instead pick between roads for safety on a more local level.

## 3.3 Route Finding System

The accident download system of the previous section only really serves to provide the input for this system. This system *is the project*. Figure 3.3 shows the completed UML diagram. `IntTuple` and `DoubleTuple`, which are used in priority queues, have been added since the class hierachy was designed but overall the structure has remained the same.

### 3.3.1 Parsing Open Street Map Data

The `OpenStreetMap` class contains the methods for parsing `.osm` files and forming a new `GeographicGraph` class. Some of these methods cannot be called in the init method of `GeographicGraph` because they operate on the graph after it has been populated by `parseXML`, so they are placed in the constructor for `OpenStreetMap`. This constructor calls `parseXML`.

```
private fun parseXML(filename: String) {
    val stream = File(filename).inputStream()
    val saxReader = SAXReader()
    val cyclableDocument = saxReader.read(stream)
    val root: Element = cyclableDocument.rootElement
    val it: Iterator<Element> = root.elementIterator()
    while (it.hasNext()) {
        val element: Element = it.next()
        when (element.qName.name) {
            "node" -> processNode(element)
            "way" -> processWay(element)
        }
    }
}
```

All that `parseXML` does is iterate through all of the subelements in the input xml file, then calls either `processNode` or `processWay`. The `processNode` function just adds all nodes it sees to the `cyclableGraph`, and the real logic is dealt with in `processWay`.

```
private fun processWay(way: Element) {
    var oneWay = false
    var cycleWay = false
    var slowWay = false
    val nodes = mutableListOf<Long>()
    val acceptedRoads = mutableListOf(
        "bridleway",
        "trunk",
        "pedestrian",
        "service",
        "primary",
        "secondary",
        "tertiary",
        "unclassified",
        "residential",
        "primary_link",
        "secondary_link",
        "tertiary_link",
        "living_street",
        "cycleway",
        "footway"
    )
    val disallowedSurfaces = mutableListOf("unpaved", "fine_gravel", "gravel", "dirt",
 "grass", "pebblestone")
    val disallowedAccess = mutableListOf("no")
    val highSpeed = mutableListOf("40 mph", "50 mph", "60 mph", "70 mph")
    val tags = HashMap<String, String>()
```

./images//finaluml.png

Figure 3.3: Final Project UML Diagram

```
    val it = way.elementIterator()
    while (it.hasNext()) {
        val subElement = it.next()
        if (subElement.qName.name == "nd") {
            nodes.add(subElement.attribute("ref").value.toLong())
        }
        if (subElement.qName.name == "tag") {
            tags[subElement.attributeValue("k")] = subElement.attributeValue("v")
        }
    }
    val highwayType = tags["highway"].toString()
    val access = tags["access"].toString()
    val surface = tags["surface"].toString()
    val note = tags["note"].toString()
    val bicycle = tags["bicycle"].toString()
    val oneway = tags["oneway"].toString()
    val maxspeed = tags["maxspeed"].toString()
    val towpath = tags["towpath"].toString()
    val motor = tags["motor_vehicle"].toString()
    if (!acceptedRoads.contains(highwayType)) return
    if (highSpeed.contains(maxspeed)) return
    if (oneway == "yes") oneWay = true
    if (highwayType == "cycleway") cycleWay = true
    if (highwayType == "footway") {
        cycleWay = true
        slowWay = true
    }
    if (highwayType == "pedestrian") {
        cycleWay = true
        if (bicycle != "designated") {
            slowWay = true
            }
    }
    if (note == "towpath") return
    if (disallowedAccess.contains(access) && bicycle != "yes") return
    if (disallowedAccess.contains(bicycle)) return
    if (disallowedSurfaces.contains(surface)) return
    if (!acceptedRoads.contains(highwayType)) return
    if (towpath == "yes") return
    if (motor == "private") cycleWay = true
    if (cycleWay) {
        cyclableGraph.safeNodes.addAll(nodes)
    }
    if (slowWay) {
        cyclableGraph.slowNodes.addAll(nodes)
    }
    for (i in 1 until nodes.size) {
        cyclableGraph.addEdge(nodes[i - 1], nodes[i], oneWay)
    }
}
```

All of the relevant values from `tags` are converted to string in order to eliminate the need for any null checks. If a way doesn't have a certain tag the value will be `"null"`, which is of course not in any of the allow or disallow lists. If the way is determined to be "safe" or "slow", all the nodes in the way are added to the relevant set in `cyclableGraph`.

## 3.4 Graph Representation

As shown in Figure 3.3, the `GeographicGraph` class is used to represent London. This is through the `GeographicNode` inner class, which stores both incoming and outgoing connections as well as the weight of the node.

#### 3.4.0.1 Graph Setup

Once all the nodes and connections have been added and the `safeNodes` and `slowNodes` sets have been filled up by the parsing functions in `OpenStreetMap`, some other functions need to be run to set up the graph for operations and otherwise clean up. These functions are called from the intialisation method of `OpenStreetMap`.

**3.4.0.1.1 Graph Pruning** The first function called is `pruneDisconnected`. This function ensures that the graph is strongly connected by finding the largest strongly connected component and deleting all nodes not in it. This ensures that the program can easily conform to Specification Point 2.13, as it ensures that it is possible to find a route between any two points in the graph.

```
fun pruneDisconnected() {
    fun getPostOrderTraversal(vertice : Long, visited : HashSet<Long>, getNeighbours :
     (Long) -> HashSet<Long>): List<Long> {
        var searchStack = Stack<Long>()
        var postOrder = LinkedList<Long>()
        searchStack.add(vertice)
        while (searchStack.size != 0)
        {
            var current = searchStack.pop()
            if (!visited.contains(current))
            {
                visited.add(current)
                postOrder.add(current)
                for (neighbour in getNeighbours(current))
                {
                    searchStack.push(neighbour)
                }
            }
        }
        return postOrderStack
    }
    var visited = HashSet<Long>()
    val L = LinkedList<Long>()
    for (vertice in vertices)
    {
        for (item in getPostOrderTraversal(vertice.key,visited) { id -> vertices[id
    ]!!.connections })
        {
            L.addFirst(item)
        }
    }
    visited = HashSet<Long>()
    val components = HashMap<Long, HashSet<Long>>()
    for (vertice in L)
    {
        var postOrder = getPostOrderTraversal(vertice,visited) { id -> vertices[id]!!.
    incomingConnections }
        if (postOrder.size != 0)
        {
            components[vertice] = HashSet<Long>()
            for (item in postOrder)
```

```
            {
                components[vertice]?.add(item)
            }
        }
    }
    val mainComponent = components.maxByOrNull { component -> component.value.size }!!
    val before = vertices.size
    vertices = vertices.filter { item -> mainComponent.value.contains(item.key) } as
    HashMap<Long, GeographicNode>
    val after = vertices.size
    println("Shrunk vertices from $before to $after")
    for (vertice in vertices.values)
    {
        vertice.connections = vertice.connections.filter { id -> vertices.containsKey(
    id) }.toHashSet()
        vertice.incomingConnections = vertice.incomingConnections.filter { id ->
    vertices.containsKey(id)}.toHashSet()
    }
}
```

I had to slightly modify Kosaraju's Algorithm[10], as it called for the use of recursive functions. As the strongly connected component that my program uses to represent London has more than a million nodes, any recursive solution was clearly going to run out of stack depth or stack space. Therefore I modified it to use a iterative approach, using the stack `searchStack`, and the list `postOrder`.
In order to avoid rewriting the subfunction `getPostOrderTraversal` for the case in which the search is performed backwards, I passed in a lambda function that is used to find either outgoing or incoming neighbours. Once all the strongly connected components are placed in the HashMap `components`, the largest one can be quickly ascertained. From there, all that is required is the actual pruning of the process. First, the `vertices` are filtered for nodes that are in the largest strongly connected component, then all of these vertices have their connections filtered to ensure that traversal algorithms do not attempt to lookup vertices which no longer exist.

**3.4.0.1.2 Setting up the R*Tree** As discussed in Section 2.2.3.1, I used an R*Tree to facilitate fast lookup of nodes from nearby coordinates. The library I used[9] supports both iterative loading of the rtree and bulk loading. Bulk loading is faster and can result in a more efficient R*Tree, so `setupRTree` uses that. The R*Tree generated is visualised in Figure 3.4.

```
fun setupRTree() {
    if (nodeTree.size() != 0) {
        return
    }
    var nodeList = mutableListOf<Entry<Long, Geometry>>()
    for (node in vertices) {
        nodeList.add(Entries.entry(node.key, Geometries.point(node.value.longitude,
    node.value.latitude)))
    }
    nodeTree = RTree.star().maxChildren(28).create(nodeList)
}
```

The function also includes a safety check to make sure that the R*Tree has not been loaded before, as if it had the entries would be added twice.

./images//rtree.png

Figure 3.4: The RTree containing all of the OSM nodes in London

**3.4.0.1.3 Gathering Weights** This system simply reads the output from the Accident Download System, and attaches weights to the nearest node to every accident. I decided on the arbitrary scoring system of 1 point for Slight accidents, 2 points for Severe accidents, and 3 points for Fatal accidents.

```
fun gatherWeights() {
    val accidentFile = File("output.json")
    val accidents = JSONArray(accidentFile.inputStream().readBytes().toString(Charsets.UTF_8))
    for (accident in accidents) {
        val parsedAccident = accident as JSONArray
        val latitude: Double = (parsedAccident[0] as BigDecimal).toDouble()
        val longitude: Double = (parsedAccident[1] as BigDecimal).toDouble()
        val severity = parsedAccident[2] as String
        try {
            val additionNode = vertices[getNearestNode(latitude, longitude)]!!
            when (severity) {
                "Slight" -> additionNode.weight += 1
                "Severe" -> additionNode.weight += 2
                "Fatal" -> additionNode.weight += 3
            }
        } catch (e: InvalidParameterException) {
            //In this case we can still apply all the other accidents, even if this accident doesn't match well.
            continue
        }
    }
}
```

In order to find the nearest node, this function calls another function called `getNearestNode`. This is shown below.

```kotlin
private fun getNearestNode(latitude: Double, longitude: Double): Long {
    val closestNodeObserver = nodeTree.nearest(Geometries.point(longitude, latitude),
    0.005, 1)
    try {
        val closestNode = closestNodeObserver.first()
        return closestNode.value()
    } catch (e: NoSuchElementException) {
        //If this happens the coordinates are very far from the nearest node, so a
    appropriate node cannot be found.
        //Therefore an error should be thrown
        throw InvalidParameterException()
    }
}
```

The `InvalidParameterException` is thrown by `getNearestNode` if there is no node within 0.005 degrees. This is roughly half a kilometer, so all reasonable ranges of nodes can be found. Selecting a smaller value for the maximum search distance makes the query faster. In `gatherWeights` this can be safely ignored as while a few coordinates might be too far away from nodes the vast majority will not be.
Finally, all known safe nodes have their weights set to 0. This completes the graph setup.

#### 3.4.0.2 Graph Queries

`GeographicGraph` has 3 main methods that will be queried by the user facing code. These are `getRandomId`, `getNearestNode`, and `findRoute`. `getRandomId` just returns a random vertice id from the vertices HashMap, and `getNearestNode` has already been explained. `findRoute` provides functionality for deciding which route finding method to use, then executing it. In my design I allowed only one instance of `contractedGraph`, but when implementing I decided to change this to allow as many contractions as wanted. This was implemented through a HashMap that holds all the instances of `contractedGraph`, which are used to find routes when matching parameters are given. So the `findRoute` function calls the corresponding function in the preprocessed graph, or the private function `findRouteNonContracted`. This is shown below.

```kotlin
/**
* Basic dijkstra
*/
private fun calculateRoute(
    start: Long,
    end: Long,
    accidentsPerKilometre: Double,
    accidentsPerTurn: Double
): Pair<MutableList<Long>, Double> {
    val F = PriorityQueue<DoubleTuple>()
    val dist = HashMap<Long, Double>()
    val prev = HashMap<Long, Long>()
    dist[start] = 0.0
    prev[start] = -1
    var u: Long
    val toAdd = DoubleTuple(start, 0.0)
    F.add(toAdd)
    while (F.size != 0) {
        val item = F.poll()
        u = item.id
        if (dist[u] != item.dist) continue
        if (u == end) {
            return Pair(solution(end, prev), dist[end]!!)
        }
        for (neighbour in vertices[u]?.connections!!) {
            var alt = dist[u]?.plus(vertices[neighbour]?.weight!!)
            alt = alt?.plus(getDistanceCost(u, neighbour) * accidentsPerKilometre)
```

```
                alt = alt?.plus(getTurnCost(prev[u]!!, u, neighbour, accidentsPerTurn))
                if (alt != null) {
                    if (!dist.containsKey(neighbour) || alt < dist[neighbour]!!) {
                        dist[neighbour] = alt
                        prev[neighbour] = u
                        val toAdd = DoubleTuple(neighbour, dist[neighbour]!!)
                        F.add(toAdd)
                    }
                }
            }
        }
    throw InvalidParameterException()
}

fun findRouteNonContracted(
    start: Long,
    end: Long,
    accidentsPerKilometre: Double,
    accidentsPerTurn: Double
): MutableList<Long> {
    return calculateRoute(start, end, accidentsPerKilometre, accidentsPerTurn).first
}

fun solution(end: Long, prev: HashMap<Long, Long>): MutableList<Long> {
    val route = mutableListOf<Long>()
    var current = end
    while (current.toInt() != -1) {
        route.add(current)
        current = prev[current]!!
    }
    return route.asReversed()
}
```

The notable difference between this code and what was designed is that when a tentative distance is updated the old value in the priority queue is not removed. This is because removing items from the middle of priority queues can be quite expensive, and it is actually cheaper just to leave it in there and just ignore it when eventually polled.

The `DoubleTuple` class is used to allow nodes weights in the priority queue to be based on their distance from the start node rather than the value of the node. The `DoubleTuple` class implements `Comparable`, so that it can be compared to other `DoubleTuple` objects by the distance. This allows the tuples to be packed into the priority queue with a weighting based on distance.

```
class DoubleTuple(val id : Long, val dist : Double) : Comparable<DoubleTuple>
{
    override fun compareTo(other: DoubleTuple): Int {
        if(dist > other.dist)
        {
            return 1
        }
        else if(dist == other.dist)
        {
            return 0
        }
        else
        {
            return -1
        }
    }
 }
```

./images//ContractableNode.png

Figure 3.5: UML class Diagram for the `ContractableNode` subclass

## 3.5 Graph Contraction and Querying

#### 3.5.0.1 Graph Contraction

In order to create a contracted graph, the two functions `createGraph` and `contractGraph` must be run. `createGraph` simply copies the relevant information from the `GeographicNode` objects in the `GeographicGraph` to `ContractableNode` objects in the new object. `contractGraph` runs the graph contraction algorithms.

```
fun contractGraph(inputGraph: GeographicGraph) {
    println("Contracting Graph of size ${vertices.size}")
    var current = 1
    var contractionQueue = PriorityQueue<IntTuple>()
    var count = 0
    for (vertice in vertices.keys) {
        contractionQueue.add(IntTuple(vertice, getHeuristicValue(vertice, inputGraph)))
        println("There are ${contractionQueue.size} nodes left to contract")
    }
    while (contractionQueue.size != 0) {
        if (count == 40) {
            println("Recalculating Queue")
            val newQueue = PriorityQueue<IntTuple>()
            while (contractionQueue.size != 0) {
                val current = contractionQueue.poll()
                val edgeDifference = getHeuristicValue(current.id, inputGraph)
                newQueue.add(IntTuple(current.id, edgeDifference))
            }
            contractionQueue = newQueue
            count = 0
        }
        val next = contractionQueue.poll()
        val oldHeuristic = next.dist
        val newHeuristic = getHeuristicValue(next.id, inputGraph)
        if (oldHeuristic != newHeuristic) {
```

```
            count += 1
            contractionQueue.add(IntTuple(next.id, newHeuristic))
            continue
        } else {
            count = 0
        }
        contractNode(next.id, current, inputGraph)
        current += 1
        println("There are ${contractionQueue.size} nodes left to contract")
    }
    println("Contraction finished")
    for (vertice in vertices) {
        vertice.value.allOutgoingConnections =
            vertice.value.shortcutConnections.keys.union(vertice.value.connections.
    keys).toList()
        vertice.value.allIncomingConnections =
            vertice.value.incomingShortcuts.keys.union(vertice.value.
    incomingConnections.keys).toList()
    }
}
```

As mentioned in Section 2.2.4, this function uses a priority queue and heuristics to determine which node to contract next. The items in the Queue are held in `IntTuple` objects, which are the same as `DoubleTuple` objects, but use integers instead of doubles.If the heuristic of the item has changed since being inserted into the queue, the item is reinserted to the queue. If this happens 40 times in a row, the entire queue is recalculated. When all the nodes are contracted, they have their shortcut and normal connections unioned together. This is further preprocessing for later route finding, as before implementing this finding unions of these sets was taking up the majority of the time used to find routes. The heuristic function `getHeuristicValue` is the sum of the edge difference and the number of deleted nodes connected to a node considered for contraction. It calls `getShortest` for every incoming node to the given node. `getShortest` gives the list of outgoing nodes such that the shortest path between the incoming and outgoing node goes through the given node.

```
/**
* Generates a tree which can then be checked for shortest routes
* The tree is built until all neighbours of the about node are found, then queried to
   find if the shortest route goes through the about node
* The estimation feature enables faster estimations of edge differences by terminating
    the search after a given number of nodes,
* but can be disabled for the real contraction operation
*/
private fun getShortest(
    from: Long,
    about: Long,
    inputGraph: GeographicGraph,
    estimation: Boolean
): MutableList<Long> {
    val toFind = vertices[about]!!.connections.keys.union(vertices[about]!!.
    shortcutConnections.keys)
        .filter { x -> !vertices[x]!!.deleted }.toHashSet()
    HashSet<Long>()
    val F = PriorityQueue<DoubleTuple>()
    val dist = HashMap<Long, Double>()
    val prev = HashMap<Long, Long>()
    var numsettled = 0
    dist[from] = 0.0
    prev[from] = -1
    var u: Long
    var toAdd = DoubleTuple(from, 0.0)
    F.add(toAdd)
```

```
    while (F.size != 0) {
        val item = F.poll()
        u = item.id
        if (dist[u] != item.dist) continue
        toFind.remove(u)
        numsettled += 1
        for (neighbour in vertices[u]!!.connections.keys) {
            var alt = dist[u]!! + vertices[u]!!.connections[neighbour]!!
            if (!dist.containsKey(neighbour) || alt < dist[neighbour]!!) {
                toFind.remove(neighbour)
                dist[neighbour] = alt
                prev[neighbour] = u
                toAdd = DoubleTuple(neighbour, dist[neighbour]!!)
                F.add(toAdd)
            }
        }
        if (toFind.size == 0 || (numsettled == 10000 && estimation)) break
    }
    val shortestThrough = mutableListOf<Long>()
    for (i in vertices[about]!!.connections.keys.union(vertices[about]!!.shortcutConnections.keys)
        .filter { x -> !vertices[x]!!.deleted }) {
        if (i == from) continue
        if (!prev.containsKey(i)) shortestThrough.add(i) //i has not been reached because contraction has been terminated early, therefore a shortcut is necessary
        else {
            val route = inputGraph.solution(i, prev)
            if (route.contains(about)) shortestThrough.add(i) //The shortest route goes through the node therefore a shortcut is necessary
        }
    }
    return shortestThrough
}
```

In order to speed up the contraction process, a settled node limit of 10000 is imposed for estimating the edge difference, but not for the actual contraction of nodes, as this limit would result in unecessary shortcuts.

#### 3.5.0.2 Graph Querying

Once the graph has been sucessfully contracted, all that remains is to allow for route finding. The `findRoute` function implements a complete bidirectional dijkstra search in the reduced search space to find a node that is known to be on the minimum path.

```
/**
    * Finds the route using contraction hierachies.
    * First the set of nodes with a higher hierarchy from the from node are calculated
    * Then the set of nodes with a higher hierachy from the to node (backwards) are calculated
    * Then the intersection of the set of settled nodes is found and the minimum found
    * Then the route is created by iteratively looking up the shortcuts
    */
fun findRoute(from: Long, to: Long, inputGraph: GeographicGraph, showVisited: Boolean): List<Long> {
    //first we generate the upwards and downwards search spaces using simple queues
    val Q: Queue<Long> = LinkedList<Long>()
    val upwardsSpace = HashSet<Long>()
    val downSpace = HashSet<Long>()
    Q.add(from)
    while (Q.size != 0) {
        val current = Q.poll()
        upwardsSpace.add(current)
        for (neighbour in vertices[current]!!.allOutgoingConnections) {
            if (vertices[neighbour]!!.hierachy > vertices[current]!!.hierachy && !upwardsSpace.contains(neighbour)) {
                Q.add(neighbour)
            }
        }
    }
    Q.add(to)
    while (Q.size != 0) {
        val current = Q.poll()
        downSpace.add(current)
        for (neighbour in vertices[current]!!.allOutgoingConnections) {
            if (vertices[neighbour]!!.hierachy > vertices[current]!!.hierachy && !downSpace.contains(neighbour)) {
                Q.add(neighbour)
            }
        }
    }
    var F = PriorityQueue<DoubleTuple>()
    val settledFrom = HashSet<Long>()
    val dist = HashMap<Long, Double>()
    val prev = HashMap<Long, Long>()
    var u: Long
    var toAdd = DoubleTuple(from, 0.0)
    dist[from] = 0.0
    F.add(toAdd)
    while (F.size != 0) {
        val item = F.poll()
        u = item.id
        if (dist[u] != item.dist) continue
        settledFrom.add(u)
        for (neighbour in vertices[u]!!.allOutgoingConnections) {
            if (!upwardsSpace.contains(neighbour)) continue
            var cost: Double =
                if (vertices[u]!!.connections.containsKey(neighbour)) vertices[u]!!.connections[neighbour]!!
                else vertices[u]!!.shortcutConnections[neighbour]!!
            val alt = cost + dist[u]!!
```

```
            if (!dist.containsKey(neighbour) || alt < dist[neighbour]!!) {
                dist[neighbour] = alt
                prev[neighbour] = u
                toAdd = DoubleTuple(neighbour, dist[neighbour]!!)
                F.add(toAdd)
            }
        }
    }
    F = PriorityQueue<DoubleTuple>()
    val settledTo = HashSet<Long>()
    val distTo = HashMap<Long, Double>()
    val prevTo = HashMap<Long, Long>()
    distTo[to] = 0.0
    toAdd = DoubleTuple(to, 0.0)
    F.add(toAdd)
    while (F.size != 0) {
        val item = F.poll()
        u = item.id
        if (distTo[u] != item.dist) continue
        settledTo.add(u)
        for (neighbour in vertices[u]!!.allIncomingConnections) {
            if (!downSpace.contains(neighbour)) continue
            var cost: Double =
                if (vertices[u]!!.incomingConnections.containsKey(neighbour)) vertices[u]!!.incomingConnections[neighbour]!!
                else vertices[u]!!.incomingShortcuts[neighbour]!!
            val alt = cost + distTo[u]!!
            if (!distTo.containsKey(neighbour) || alt < distTo[neighbour]!!) {
                distTo[neighbour] = alt
                prevTo[neighbour] = u
                toAdd = DoubleTuple(neighbour, distTo[neighbour]!!)
                F.add(toAdd)
            }
        }
    }
    var minimumNode: Long = -1
    var minimumCost = Double.MAX_VALUE
    for (i in settledTo.intersect(settledFrom)) {
        if (dist[i]!! + distTo[i]!! - inputGraph.vertices[i]!!.weight < minimumCost) {
            minimumNode = i
            minimumCost = dist[i]!! + distTo[i]!! - inputGraph.vertices[i]!!.weight
        }
    }
    println(minimumCost)
    return if (showVisited) settledTo.intersect(settledFrom).toList()
    else deContractRoute(minimumNode, prev, prevTo)
}
```

The implementation of this function is fairly standard, it just consists of two dijkstra searches in reduced search spaces and a call to `deContractRoute`. `deContractRoute` works both backwards and forwards from the minimum node to gather the whole route, then it fills in all of the shortcuts.

```
private fun deContractRoute(minimumNode: Long, prev: HashMap<Long, Long>, prevTo: HashMap<Long, Long>): List<Long> {
    val route = mutableListOf<Long>(minimumNode)
    while (prev.containsKey(route[0]) && prev[route[0]]!! != -1L) {
        route.add(0, prev[route[0]]!!)
    }
    while (prevTo.containsKey(route.last()) && prevTo[route.last()]!! != -1L) {
        route.add(prevTo[route.last()]!!)
    }
    return unpackRoute(route)
```

./images//interface.png

Figure 3.6: The interface for my program

```
}

private fun unpackRoute(route: MutableList<Long>): MutableList<Long> {
    while (true) {
        var finished = true
        for (i in 1 until route.size) {
            if (route[i - 1] == -1L) continue
            val current = vertices[route[i - 1]]!!
            if (current.shortcutConnections.containsKey(route[i])) {
                route.add(i, current.shortcutRoutes[route[i]]!!)
                finished = false
                break
            }
        }
        if (finished) break
    }
    return route
}
```

`unpackRoute` operates in a similar manner to a bubble sort, repeatedly iterating through the list until it is known that it contains no shortcuts.

## 3.6 User Interface

The user interface is written in the `Main.kt` file. It is defensively programmed, to protect from user error, and the code is shown in the Appendix. It allows for an input loop. An example usage is shown in Figure 3.6.

## 3.7 Serialisation

In order to serialise objects of a certain class in kotlin, the class and all of the properties of the class must be serialisable. Kotlin data structures, like HashMaps and HashSets are all serialisable. The only data structure that was not serialisable was the R*Tree, so this had to be marked as Transient so that the serialiser would not complain, and then regenerated by calling `setupRTree` upon deserialisation.

```kotlin
@Serializable
class GeographicGraph {
    var vertices: HashMap<Long, GeographicNode> = HashMap()
    var safeNodes = HashSet<Long>() //nodes that are known to be safe
    var slowNodes = HashSet<Long>()
    var contractedGraphs = HashMap<Double, ContractableGraph>()

    @Transient
    lateinit var nodeTree: RTree<Long, Geometry>
```

When a method is deserialised, the intialisation method is run, so I had to move the functionality of these into other functions to stop things incorrectly being added twice upon deserialisation. The `OpenStreetMap` class has a init method, but this checks to see if the `cyclableGraph` has already been created, and if so does nothing.

```kotlin
init {
    if (cyclableGraph.vertices.size == 0) {
        println("Parsing XML")
        parseXML(filename)
        println("Pruning Disconnected")
        cyclableGraph.pruneDisconnected()
        println("Creating rTree")
        cyclableGraph.setupRTree()
        println("Gathering weights")
        cyclableGraph.gatherWeights()
        for (node in cyclableGraph.safeNodes)
            cyclableGraph.vertices[node]?.weight = 0.0
    }
}
```

4

# Testing

I employed 3 major strategies to test my Project.

1. Comprehensive unit testing. Tests are defined for every major function in code, and run regularly. This helps ascertain that the code works, and also helps to ensure that bugs are not introduced during development.

2. Human Testing. Essentially a human uses the program and ascertains that the program complies with specification points and does not crash when the user does unexpected things.

3. Performance Testing. In order to ascertain that the program runs at an acceptable speed, automated performance tests measure the speed of certain functions.

## 4.1 Creating a test area

Some operations, especially preprocesing, take a long time to execute. As the unit tests will be run many times, it makes sense to define a smaller test area for some operations. I created a GeoJSON file, and used it to cut out a smaller area of london, called `testarea.osm`.

## 4.2 Unit Testing

The basic idea of unit testing is to test that all functions do what they should do. Where possible, I manually worked out what they should do and set this as the condition for the test.

| Unit Testing Table | | | |
|---|---|---|---|
| Test Coverage | Description | Type | Works? |
| `GeographicGraph.getDistance` | At the equator, a degree of longitude is approximately 111km. This function calculates a degree of longitude at the equator and checks that it is close to 111km | Equals | ✓ |
| `GeographicGraph.getRandomId` | A random id is sucessfully generated which is in the graph set of vertices | Boolean | ✓ |
| `GeographicGraph.addEdge` | Add both directed and undirected edges to a graph, and makes sure they are added correctly | Boolean | ✓ |
| `GeographicGraph.gatherWeights` | Set all weights in the graph to 0, gather them again, and check for right value | Equals | ✓ |
| `GeographicGraph.getDistanceCost` | Checks that the calculated distance cost between two close nodes is as expected | Equals | ✓ |
| `GeographicGraph.pruneDisconnected` | Runs the function on the graph, and checks that nothing changes as it has been run before | Equals | ✓ |
| `GeographicGraph.getTurnCost` | Runs the turn cost function on nodes in a straight road, and around a 90 degree turn | Equals | ✓ |
| `GeographicGraph.contractGraph` | Runs the contraction function on the graph | No Errors | ✓ |
| `GeographicGraph.findRoute` | Runs the route finding algorithm and checks the total distance | Equals | ✓ |

| Continuation of Unit Testing Table | | | |
|---|---|---|---|
| Test Coverage | Description | Type | Works? |
| `ContractableGraph.createGraph` | Creates a new contractable graph | No Errors | ✓ |
| `ContractableGraph.findRoute` | Creates and contracts a `ContractableGraph` instance, then checks the route found has the right cost. | Equals | ✓ |
| `OpenStreetMap` | Creates a new `OpenStreetMap` instance | No Errors | ✓ |
| `MainKt.writeMapToDisk` | Writes an OSM file to disk, reads it back, and checks that the size of the graphs is the same | Equals | ✓ |
| `MainKt.writeObjectToDisk` | Writes an OSM file to disk, reads it back, and checks that the size of the graphs is the same | Equals | ✓ |

As shown in Figure 4.1, all of these tests ran successfully, and had relatively high code coverage considering that most of the lines in `Main.kt` cannot be easily unit tested as they rely on user input.

## 4.3 Human Testing

I also used human testing to make sure that the program runs as specified. These tests mostly check that code does not break when given difficult inputs, but also that it conforms to other specification points, and that the code I know is working due to unit testing is correctly connected to the user interface. I used two types of tests: Normal tests, where everything should work, and breaking tests, where the program should not crash and give an expected output.

| Human Testing Table | | | | |
|---|---|---|---|---|
| Covered Specification Point | Description | Type | Evidence | Works? |
| 1.1 | Run the `download.py` script and check that accidents are properly stored. | Normal | Figure A.1 | ✓ |
| 1.2 & 1.3 | Run `parse.py` and verify that the `output.json` file is created correctly using `display.py` | Normal | Figure A.2 | ✓ |
| 2.1 & 2.2 & 2.3 & 2.4 | Run the program, and select a `.osm` file to parse, and verify it is parsed correctly by finding a route between two points. | Normal | Figure A.3 | ✓ |
| 2.5 & 2.6 & 2.7 & 2.8 & 2.15 | Find a route between two points, and evaluate whether following it would be safe, legal, &possible, by following the route. | Normal | Appendix B | ✓ |
| 2.10 | Load a `.osm` file, contract the graph, then execute and check a `findRoute` query on that graph. | Normal | Figure A.4 | ✓ |
| 2.11 | Run a contracted and non contracted query on the same graph, and verify that they overlap. | Normal | Figure A.6 | ✓ |
| 2.13 & 2.14 | Attempt to cause a crash by inputting illegal inputs. Verify that this fails | Illegal | Figure A.5 | ✓ |
| 2.16 | Create and then save a map. Load the map back into memory and verify that route finding on it still works | Normal | Figure A.7 | ✓ |

./images//test.png

./images//coverage.png

Figure 4.1: Unit tests running successfully, as well as code coverage stats

./images//profiling.png

Figure 4.2: Profiling of the `ContractableGraph.findRoute` method

## 4.4 Performance Testing

Performance testing is needed to test specification point 2.9. I added a test option to the user interface that runs 1000 searches between east london and heathrow airport. As these nodes are very far apart the time taken to find routes between them should be at its maximum. So long as this number is low enough, we can assume that all other routes would also be findable in a sufficient time period. I decided to test all algorithms with a distance cost of 15.0 and a turn cost of 0.0.

### 4.4.1 Testing Without Contraction

Finding a route between Heathrow and East London takes on average 1.34563138954 seconds. This is quite a long time when dealing with computers, but users will be able to wait that long, so I say it is ok. Another useful metric is the total number of nodes settled. For this contraction it was 783382 nodes, which is a sizeable proportion of the total number.

### 4.4.2 Testing with Contractions

When querying a contracted graph, finding a route between Heathrow and East London takes on average 0.11923476453. This is quite a speedup on 1.35 seconds, but not as much as could be expected. Looking at Figure 4.2 most of the time being spent is on route decontraction. This is the part of the algorithm that is run to find a whole route from a single part. In order to find out how long the main algorithm is taking, looking at the number of settled nodes is useful. In this search a total of 406 nodes are settled, which is massively reduced. This isn't directly comparable to the figure from the standard search, as the degree of these nodes will have been increased by adding shortcuts, and many nodes will have been settled by both searches, but the idea remains the same.

`./images//settled.png`

Figure 4.3: The set of settled nodes in a `ContractableGraph.findRoute` search

From this small search space we can conclude that if the size of the graph was increased, for example to include the rest of the world, standard dijkstra's time taken would grow out of control, while contraction hierachies would not. The set of settled nodes are visualised in Figure 4.3.

# 5 Evaluation

# Appendices

A

# Evidence for Human Testing

./images//downloadrun.png ./images//downloadproof.png

Figure A.1: Evidence that `download.py` works as expected

./images//parserun.png ./images//chart.png

Figure A.2: Evidence that `parse.py` works as expected

./images//parsegraph.png

Figure A.3: Evidence that `.osm` files can be successfully parsed

| | |
|---|---|
| ./images//contractiontest.png | ./images//contractionroute.png |

Figure A.4: Evidence that contracting a graph works

./images//distancecost.png

./images//coordinateinput.png

./images//loadfile.png

Figure A.5: Program not accepting illegal inputs

./images//checkContraction.png

./images//overlap.png

Figure A.6: Evidence that the routes generated by both approaches are the same. Both tracks are displayed in the second picture, but the first is hidden behind the second

./images//savemapone.png

./images//savemaptwo.png

./images//savemapthree.png

Figure A.7: Evidence that saving and reloading a map to file does not break it

B

# Real World Testing

./images//googlemapsroute.png

Figure B.1: The route suggested by Google Maps for this journey

In order to test specification points 2.5 & 2.6 & 2.7 & 2.8 & 2.15 complete usage of the program for its intended purpose is required. I decided to cycle into Central London at 17:00 on a Thursday, going from 38 Thornfield Rd, London W12 8JQ to The Mall, London SW1A 1BB. This time is very busy, so it was a good test of my program. The route suggested by Google Maps for this journey is shown in Figure B.1. I imported the route into Osmand, and used it's route matching feature to allow turn by turn directions to be given. As seen in Figure B.2, I managed to accurately follow the route apart from a brief section in the middle where I turned off the route. This was because the gps lock on my phone was lost, and so was not really the fault of my program. Apart from that the route was easy to follow, as Osmand generated useful turn by turn directions from my route, and there were no turns that proved practically impossible. There were also several long straight sections, which are of course easy to navigate along.
In terms of safety I thought that the route was pretty good, as it took me onto many empty residential streets in Notting Hill and near Paddington. A very large proportion of the route was on segregated cycle paths or low traffic neighbourhoods. Other parts of the route at least had some cycle lanes. When returning from central London, I used google maps and found myself in heavy traffic on kensington high street. While the situation was not extremely dangerous it at least required more attention than was required on the way there. On the other hand the Google Maps route was significantly quicker, and did not go up a large hill. It also had less turns because it used major roads.

./images//testroutegpxsee.png ./images//strava.png

Figure B.2: The route generated by my program and my attempt at following it

## B.1 Subjective Analysis

| Specification Point | Evaluation |
|---|---|
| 2.5 | Excluding the one exception caused by lost gps, I managed to exactly replicate in real life the route suggested by my program, so I would say that this specification point has been met. |
| 2.6 | The route only led me the right way down one way roads, and did not lead me into any private areas. To the best of my knowledge my cycling of this route was legal, so I would say that this specification point has been met. |
| 2.7 | As previously discussed, the route generated was safe at the expense of some speed. A very large proportion of the total distance travelled was in segregated cycle lanes, or low traffic residential neighbourhoods, and the rest still felt quite safe. Obviously working out how safe one thing is compared to another is highgly subjective, but I felt that the route generated was more safe than the route generated by Google Maps. |
| 2.8 | The route did not direct me onto any roads which would be too dangerous to consider, such as the A4. Obviously this route might still have been considered, just decided against, but my other testing makes me reasonably confident this is not so. |
| 2.15 | The `.gpx` file was easily imported into Osmand as a route from which turn by turn instructions could be generated. |

All in all I believe this test was a success. Obviously it is possible that some other routes may fail to meet the specification, but this one certainly did.

# C Program Source Code

The structure of the source code is shown in Figure 3.1

download.py

```
import requests
import json
for year in range(2005,2020):
    print(year)
    response = requests.get(f"https://api.tfl.gov.uk/AccidentStats/{year}")
    with open(f"accidents/{year}.json","w") as yearfile:
        json.dump(response.json(),yearfile)
```

parse.py

```
import json
#we want to parse for accidents that happened to cyclists and store the severity and
    the location
accidents = []
for year in range(2010,2020):
    print(len(accidents), year)
    with open(f"accidents/{year}.json","r") as outfile:
        data = json.load(outfile)
    for x in data:
        try:
            if x["casualties"]["mode"] == "PedalCycle":
                accidents.append([x["lat"],x["lon"],x["severity"]])
        except:
            if len([person for person in x["casualties"] if person["mode"] == "
    PedalCycle"]) >= 1:
                accidents.append([x["lat"],x["lon"],x["severity"]])
with open("output.json","w") as outfile:
    json.dump(accidents,outfile)
```

display.py

```
import json
import matplotlib.pyplot as plt
import numpy as np
x = []
y = []
with open("output.json","r") as inputfile:
    data = json.load(inputfile)
for accident in data:
    x.append(accident[1])
    y.append(accident[0])
    print(accident[0],accident[1])
plt.scatter(x,y,0.05)
plt.show()
```

src/main/kotlin/Main.kt

```
import io.jenetics.jpx.GPX
import io.jenetics.jpx.Track
import io.jenetics.jpx.TrackSegment
import kotlinx.serialization.ExperimentalSerializationApi
import kotlinx.serialization.cbor.Cbor
import kotlinx.serialization.decodeFromByteArray
import kotlinx.serialization.encodeToByteArray
import org.dom4j.DocumentException
import java.io.File
import java.io.FileNotFoundException
```

```kotlin
import java.nio.file.InvalidPathException
import java.nio.file.Path
import java.security.InvalidParameterException


fun writeNewScatter(scatter: List<Long>, cyclableGraph: GeographicGraph, gpx: GPX):
    GPX {
    var workingGpx = gpx
    for (node in scatter) {
        val nodeVertice = cyclableGraph.vertices[node]!!
        workingGpx =
            workingGpx.toBuilder().addWayPoint { p -> p.lon(nodeVertice.longitude).lat
    (nodeVertice.latitude) }.build()
    }
    return workingGpx
}

@OptIn(ExperimentalSerializationApi::class)
fun writeNewTrack(route: List<Long>, cyclableGraph: GeographicGraph, gpx: GPX): GPX {
    var track: TrackSegment = TrackSegment.builder().build()
    for (point in route) {
        val node = cyclableGraph.vertices[point]
        track = track.toBuilder().addPoint { p -> p.lon(node!!.longitude).lat(node.
    latitude) }.build()
    }
    val segment: Track = Track.builder().addSegment(track).build()
    return gpx.toBuilder().addTrack(segment).build()
}

@OptIn(ExperimentalSerializationApi::class)
fun writeMapToDisk(filename: String, toWrite: OpenStreetMap) {
    val saveFile = File(filename)
    saveFile.writeBytes(Cbor.encodeToByteArray(toWrite))
}

@OptIn(ExperimentalSerializationApi::class)
fun readMapFromDisk(filename: String): OpenStreetMap {
    val readFile = File(filename)
    return Cbor.decodeFromByteArray(readFile.readBytes())
}

fun getMapObject(): OpenStreetMap {
    var map: OpenStreetMap
    while (true) {
        print("Load from osm file (o) or load from preprocessed data (p):")
        val input = readln()
        print("File name:")
        val filename = readln()
        if (input == "p") {
            try {
                map = readMapFromDisk(filename)
                println("Loaded map corresponding to ${map.filename}")
                println("This map has contractions for ${map.cyclableGraph.
    contractedGraphs.keys}")
                map.cyclableGraph.setupRTree()
            } catch (e: FileNotFoundException) {
                println("File not found please try a different file")
                continue
            } catch (e: java.lang.Exception) {
                println("File is not of the right type, please fix or try a different
    file")
```

```
                continue
            }
            map.cyclableGraph.setupRTree() //re establish R*Tree
            break
        }
        if (input == "o") {
            try {
                map = OpenStreetMap(filename)
            } catch (e: FileNotFoundException) {
                println("File not found please try a different file")
                continue
            } catch (e: DocumentException) {
                println("File is not of the right type, please fix or try a different
 file")
                continue
            }
            break
        }
    }
    return map
}

@OptIn(ExperimentalSerializationApi::class)
fun main(args: Array<String>) {
    fun getDouble(prompt : String, default : Double) : Double
    {
        var distanceCost = -1.0
        while (distanceCost < 0) {
            try {
                print(prompt)
                val input = readln()
                if (input == "d"){
                    println("Using default value of $default")
                    return default
                }
                distanceCost = input.toDouble()
            } catch (e: NumberFormatException){
                continue
            }
        }
        return distanceCost
    }
    var gpx = GPX.builder().build()
    val map = getMapObject()
    while (true) {
        print("Contract (c), Find Route(r), Find Random Route (rr), Exit (e), Display
 Contraction Levels (d), Test (t) or Save(s):")
        val input = readln()
        when (input) {
            "e" -> break
            "d" -> println(map.cyclableGraph.contractedGraphs.keys)
            "s" -> {
                print("Save file path:")
                val filename = readln()
                try {
                    writeMapToDisk(filename, map)
                } catch (e: InvalidPathException) {
                    continue
                }
            }
            "c" -> {
```

```
            try {
                val distanceCost = getDouble("Distance cost (d for default):",30.0)
                map.cyclableGraph.contractGraph(distanceCost)
            } catch (e: NumberFormatException) {
                continue
            }
        }
        "r" -> {
            try {
                val distanceCost = getDouble("Distance cost (d for default):",30.0)
                val turnCost = getDouble("Turn cost (d for default):",2.0)
                print("Coordinate One:")
                val coordinateOne = readln()
                print("Coordinate Two:")
                val coordinateTwo = readln()
                val latOne = coordinateOne.split(",")[0].toDouble()
                val longOne = coordinateOne.split(",")[1].toDouble()
                val latTwo = coordinateTwo.split(",")[0].toDouble()
                val longTwo = coordinateTwo.split(",")[1].toDouble()
                val startTime = System.nanoTime()
                gpx = writeNewTrack(
                    map.cyclableGraph.findRoute(
                        latOne,
                        longOne,
                        latTwo,
                        longTwo,
                        distanceCost,
                        turnCost,
                        false
                    ), map.cyclableGraph, gpx
                )
                val endTime = System.nanoTime()
                println("Route finding completed in ${(endTime - startTime) / (1000000000.0)}")
            } catch (e: NumberFormatException) {
                continue
            } catch (e: InvalidParameterException)
            {
                println("Check your coordinates")
                continue
            }
        }
        "rr" -> {
            try {
                val distanceCost = getDouble("Distance cost (d for default):",15.0)
                val turnCost = getDouble("Turn cost (d for default):",10.0)
                val first = map.cyclableGraph.getRandomId()
                val second = map.cyclableGraph.getRandomId()
                val startTime = System.nanoTime()
                gpx = writeNewTrack(
                    map.cyclableGraph.findRoute(first, second, distanceCost, turnCost, false),
                    map.cyclableGraph,
                    gpx
                )
                val endTime = System.nanoTime()
                println("Route finding completed in ${(endTime - startTime) / (1000000000.0)}")
```

```
                } catch (e: NumberFormatException) {
                    continue
                }
            }
            "t" -> {
                //repeated query path between 2 far away nodes and record the average time difference
                val first = 68248591L
                val second = 117869324L
                val distanceCost = getDouble("Distance cost (d for default):",15.0)
                val turnCost = getDouble("Turn cost (d for default):",15.0)
                val startTime = System.nanoTime()
                gpx = writeNewScatter(map.cyclableGraph.findRoute(first,second,distanceCost,turnCost,true),map.cyclableGraph,gpx)
                for (i in 1..100) map.cyclableGraph.findRoute(first, second, distanceCost, turnCost, false)
                val endTime = System.nanoTime()
                println("Route finding completed in average time of ${(endTime - startTime) / (1000000000.0 * 100.0)}")
            }
        }
    }
    print("Do you want to save the GPX file (y):")
    if (readln() == "y") {
        print("Filename:")
        GPX.write(gpx, Path.of(readln()))
    }
}
```

src/main/kotlin/OpenStreetMap.kt

```
import com.github.davidmoten.rtree2.geometry.Geometries
import kotlinx.serialization.Serializable
import org.dom4j.Element
import org.dom4j.io.SAXReader
import java.io.File


@Serializable
class OpenStreetMap constructor(val filename: String) {
    var cyclableGraph = GeographicGraph()

    init {
        if (cyclableGraph.vertices.size == 0) {
            println("Parsing XML")
            parseXML(filename)
            println("Pruning Disconnected")
            cyclableGraph.pruneDisconnected()
            println("Creating rTree")
            cyclableGraph.setupRTree()
            println("Gathering weights")
            cyclableGraph.gatherWeights()
            for (node in cyclableGraph.safeNodes)
                cyclableGraph.vertices[node]?.weight = 0.0
        }
    }

    private fun parseXML(filename: String) {
        val stream = File(filename).inputStream()
        val saxReader = SAXReader()
        val cyclableDocument = saxReader.read(stream)
        val root: Element = cyclableDocument.rootElement
```

```
        val it: Iterator<Element> = root.elementIterator()
        while (it.hasNext()) {
            val element: Element = it.next()
            when (element.qName.name) {
                "node" -> processNode(element)
                "way" -> processWay(element)
            }
        }
    }

    private fun processNode(node: Element) {
        val longitude = node.attribute("lon").value.toDouble()
        val latitude = node.attribute("lat").value.toDouble()
        cyclableGraph.vertices[node.attribute("id").value.toLong()] =
            GeographicGraph.GeographicNode(longitude, latitude)
    }

    private fun processWay(way: Element) {
        var oneWay = false
        var cycleWay = false
        var slowWay = false
        val nodes = mutableListOf<Long>()
        val acceptedRoads = mutableListOf(
            "bridleway",
            "trunk",
            "pedestrian",
            "service",
            "primary",
            "secondary",
            "tertiary",
            "unclassified",
            "residential",
            "primary_link",
            "secondary_link",
            "tertiary_link",
            "living_street",
            "cycleway",
            "footway"
        )
        val disallowedSurfaces = mutableListOf("unpaved", "fine_gravel", "gravel", "
    dirt", "grass", "pebblestone")
        val disallowedAccess = mutableListOf("no")
        val highSpeed = mutableListOf("40 mph", "50 mph", "60 mph", "70 mph")
        val tags = HashMap<String, String>()
        val it = way.elementIterator()
        while (it.hasNext()) {
            val subElement = it.next()
            if (subElement.qName.name == "nd") {
                nodes.add(subElement.attribute("ref").value.toLong())
            }
            if (subElement.qName.name == "tag") {
                tags[subElement.attributeValue("k")] = subElement.attributeValue("v")
            }
        }
        val highwayType = tags["highway"].toString()
        val access = tags["access"].toString()
        val surface = tags["surface"].toString()
        val note = tags["note"].toString()
        val bicycle = tags["bicycle"].toString()
        val oneway = tags["oneway"].toString()
        val maxspeed = tags["maxspeed"].toString()
```

```
        val towpath = tags["towpath"].toString()
        val motor = tags["motor_vehicle"].toString()
        if (!acceptedRoads.contains(highwayType)) return
        if (highSpeed.contains(maxspeed)) return
        if (oneway == "yes") oneWay = true
        if (highwayType == "cycleway") cycleWay = true
        if (highwayType == "footway") {
            cycleWay = true
            slowWay = true
        }
        if (highwayType == "pedestrian") {
            cycleWay = true
            if (bicycle != "designated") {
                slowWay = true
            }
        }
        if (note == "towpath") return
        if (disallowedAccess.contains(access) && bicycle != "yes" && highwayType != "pedestrian") return
        if (disallowedAccess.contains(bicycle) && highwayType != "pedestrian") return
        if (disallowedSurfaces.contains(surface)) return
        if (!acceptedRoads.contains(highwayType)) return
        if (towpath == "yes") return
        if (motor == "private") cycleWay = true
        if (cycleWay) {
            cyclableGraph.safeNodes.addAll(nodes)
        }
        if (slowWay) {
            cyclableGraph.slowNodes.addAll(nodes)
        }
        for (i in 1 until nodes.size) {
            cyclableGraph.addEdge(nodes[i - 1], nodes[i], oneWay)
        }
    }
}
```

src/main/kotlin/ContractableGraph.kt

```
import kotlinx.serialization.Serializable
import java.util.*
import kotlin.collections.HashMap
import kotlin.collections.HashSet

@Serializable
class ContractableGraph(private var distanceCost: Double) {
    /**
        * In the initalisation phase we create a new graph from the previous graph
        */
    private var vertices = HashMap<Long, ContractableNode>()
    private var noShortcuts = 0

    /**
        * A vertice that represents two connected edges and their associated connections (given as id's)
        */
    @Serializable
    class ContractableNode(val id: Long) {
        var deleted = false
        var hierachy = 0
        var connections = HashMap<Long, Double>()
        var incomingConnections = HashMap<Long, Double>()
        var shortcutConnections =
```

```
            HashMap<Long, Double>() //gives the weights of a shortcut based connection to another node
        var incomingShortcuts = HashMap<Long, Double>()
        var shortcutRoutes = HashMap<Long, Long>() //gives the node through which the shortcut travels
        var allOutgoingConnections = listOf<Long>()
        var allIncomingConnections = listOf<Long>()
    }


    /**
        * Finds all edges
        * Checks every edge to see if it is reversible, if so it is added twice
        */
    //TODO edges are not symetrical i.e node1 to node2 is different from node2 to node1 so use the edgelookup system to fix this
    fun createGraph(inputGraph: GeographicGraph) {
        println("Creating Graph")
        for (vertice in inputGraph.vertices) {
            vertices[vertice.key] = ContractableNode(vertice.key)
            for (connection in vertice.value.connections) {
                vertices[vertice.key]!!.connections[connection] =
                    inputGraph.routeCost(vertice.key, connection, distanceCost, 0.0)
            }
            for (incoming in vertice.value.incomingConnections) {
                vertices[vertice.key]!!.incomingConnections[incoming] =
                    inputGraph.routeCost(incoming, vertice.key, distanceCost, 0.0)
            }
        }
    }

    /**
        * Finds the route using contraction hierachies.
        * First the set of nodes with a higher hierarchy from the from node are calculated
        * Then the set of nodes with a higher hierachy from the to node (backwards) are calculated
        * Then the intersection of the set of settled nodes is found and the minimum found.
        * Then the route is created by iteratively looking up the shortcuts
        */
    fun findRoute(from: Long, to: Long, inputGraph: GeographicGraph, showVisited: Boolean): List<Long> {
        //first we generate the upwards and downwards search spaces using simple queues
        val Q: Queue<Long> = LinkedList<Long>()
        val upwardsSpace = HashSet<Long>()
        val downSpace = HashSet<Long>()
        Q.add(from)
        while (Q.size != 0) {
            val current = Q.poll()
            upwardsSpace.add(current)
            for (neighbour in vertices[current]!!.allOutgoingConnections) {
                if (vertices[neighbour]!!.hierachy > vertices[current]!!.hierachy && !upwardsSpace.contains(neighbour)) {
                    Q.add(neighbour)
                }
            }
        }
        Q.add(to)
        while (Q.size != 0) {
```

```
            val current = Q.poll()
            downSpace.add(current)
            for (neighbour in vertices[current]!!.allOutgoingConnections) {
                if (vertices[neighbour]!!.hierachy > vertices[current]!!.hierachy && !downSpace.contains(neighbour)) {
                    Q.add(neighbour)
                }
            }
        }
        var F = PriorityQueue<DoubleTuple>()
        val settledFrom = HashSet<Long>()
        val dist = HashMap<Long, Double>()
        val prev = HashMap<Long, Long>()
        var u: Long
        var toAdd = DoubleTuple(from, 0.0)
        dist[from] = 0.0
        F.add(toAdd)
        while (F.size != 0) {
            val item = F.poll()
            u = item.id
            if (dist[u] != item.dist) continue
            settledFrom.add(u)
            for (neighbour in vertices[u]!!.allOutgoingConnections) {
                if (!upwardsSpace.contains(neighbour)) continue
                var cost: Double =
                    if (vertices[u]!!.connections.containsKey(neighbour)) vertices[u]!!.connections[neighbour]!!
                    else vertices[u]!!.shortcutConnections[neighbour]!!
                val alt = cost + dist[u]!!
                if (!dist.containsKey(neighbour) || alt < dist[neighbour]!!) {
                    dist[neighbour] = alt
                    prev[neighbour] = u
                    toAdd = DoubleTuple(neighbour, dist[neighbour]!!)
                    F.add(toAdd)
                }
            }
        }
        F = PriorityQueue<DoubleTuple>()
        val settledTo = HashSet<Long>()
        val distTo = HashMap<Long, Double>()
        val prevTo = HashMap<Long, Long>()
        distTo[to] = 0.0
        toAdd = DoubleTuple(to, 0.0)
        F.add(toAdd)
        while (F.size != 0) {
            val item = F.poll()
            u = item.id
            if (distTo[u] != item.dist) continue
            settledTo.add(u)
            for (neighbour in vertices[u]!!.allIncomingConnections) {
                if (!downSpace.contains(neighbour)) continue
                var cost: Double =
                    if (vertices[u]!!.incomingConnections.containsKey(neighbour)) vertices[u]!!.incomingConnections[neighbour]!!
                    else vertices[u]!!.incomingShortcuts[neighbour]!!
                val alt = cost + distTo[u]!!
                if (!distTo.containsKey(neighbour) || alt < distTo[neighbour]!!) {
                    distTo[neighbour] = alt
                    prevTo[neighbour] = u
                    toAdd = DoubleTuple(neighbour, distTo[neighbour]!!)
                    F.add(toAdd)
```

```
                }
            }
        }
        var minimumNode: Long = -1
        var minimumCost = Double.MAX_VALUE
        for (i in settledTo.intersect(settledFrom)) {
            if (dist[i]!! + distTo[i]!! - inputGraph.vertices[i]!!.weight <
minimumCost) {
                minimumNode = i
                minimumCost = dist[i]!! + distTo[i]!! - inputGraph.vertices[i]!!.
weight
            }
        }
        return if (showVisited) settledTo.intersect(settledFrom).toList()
        else deContractRoute(minimumNode, prev, prevTo)
    }

    /**
        * With given conditions, decontracts the route
        */
    private fun deContractRoute(minimumNode: Long, prev: HashMap<Long, Long>, prevTo:
HashMap<Long, Long>): List<Long> {
        val route = mutableListOf<Long>(minimumNode)
        while (prev.containsKey(route[0]) && prev[route[0]]!! != -1L) {
            route.add(0, prev[route[0]]!!)
        }
        while (prevTo.containsKey(route.last()) && prevTo[route.last()]!! != -1L) {
            route.add(prevTo[route.last()]!!)
        }
        return unpackRoute(route)
    }

    private fun unpackRoute(route: MutableList<Long>): MutableList<Long> {
        while (true) {
            var finished = true
            for (i in 1 until route.size) {
                if (route[i - 1] == -1L) continue
                val current = vertices[route[i - 1]]!!
                if (current.shortcutConnections.containsKey(route[i])) {
                    route.add(i, current.shortcutRoutes[route[i]]!!)
                    finished = false
                    break
                }
            }
            if (finished) break
        }
        return route
    }

    /**
        * Creates another graph: G*, which is contracted
        * Uses a edge change heuristic, which aims to minimise the number of edges
reduced
        */
    fun contractGraph(inputGraph: GeographicGraph) {
        println("Contracting Graph of size ${vertices.size}")
        var current = 1
        var contractionQueue = PriorityQueue<IntTuple>()
        var count = 0
        for (vertice in vertices.keys) {
            contractionQueue.add(IntTuple(vertice, getHeuristicValue(vertice,
```

```
inputGraph)))
            println("There are ${contractionQueue.size} nodes left to contract")
        }
        while (contractionQueue.size != 0) {
            if (count == 40) {
                println("Recalculating Queue")
                val newQueue = PriorityQueue<IntTuple>()
                while (contractionQueue.size != 0) {
                    val current = contractionQueue.poll()
                    val edgeDifference = getHeuristicValue(current.id, inputGraph)
                    newQueue.add(IntTuple(current.id, edgeDifference))
                }
                contractionQueue = newQueue
                count = 0
            }
            val next = contractionQueue.poll()
            val oldHeuristic = next.dist
            val newHeuristic = getHeuristicValue(next.id, inputGraph)

            if (oldHeuristic != newHeuristic) {
                count += 1
                contractionQueue.add(IntTuple(next.id, newHeuristic))
                continue
            } else {
                count = 0
            }
            contractNode(next.id, current, inputGraph)
            current += 1
            println("There are ${contractionQueue.size} nodes left to contract")
        }
        println("Contraction finished")
        //work out the unions now so the program doesn't have to later when route finding
        for (vertice in vertices) {
            vertice.value.allOutgoingConnections =
                vertice.value.shortcutConnections.keys.union(vertice.value.connections.keys).toList()
            vertice.value.allIncomingConnections =
                vertice.value.incomingShortcuts.keys.union(vertice.value.incomingConnections.keys).toList()
        }
    }

    /**
        * Performs the act of contraction on the input node
        */
    private fun contractNode(node: Long, current: Int, inputGraph: GeographicGraph) {
        val nodeObj = vertices[node]!!
        for (from in nodeObj.incomingConnections.keys.union(nodeObj.incomingShortcuts.keys)
            .filter { x -> !vertices[x]!!.deleted }) {
            for (to in getShortest(from, node, inputGraph, false)) {
                if (from != to) {
                    val fromVertice = vertices[from]!!
                    val nodeObj = vertices[node]!!
                    val toVertice = vertices[to]!!
                    noShortcuts += 1
                    var connectionWeight = 0.0
                    connectionWeight += if (fromVertice.connections.containsKey(node)) fromVertice.connections[node]!! else fromVertice.shortcutConnections[node]!!
                    connectionWeight += if (nodeObj.connections.containsKey(to))
```

```
nodeObj.connections[to]!! else nodeObj.shortcutConnections[to]!!
                    fromVertice.shortcutConnections[to] = connectionWeight
                    fromVertice.shortcutRoutes[to] = node
                    toVertice.incomingShortcuts[from] = connectionWeight
                }
            }
        }
        nodeObj.deleted = true
        nodeObj.hierachy = current
    }

    /**
        * Generates a tree which can then be checked for shortest routes
        * The tree is built until all neighbours of the about node are found, then queried to find if the shortest route goes through the about node
        * The estimation feature enables faster estimations of edge differences by terminating the search after a given number of nodes,
        * but can be disabled for the real contraction operation
        */
    private fun getShortest(
        from: Long,
        about: Long,
        inputGraph: GeographicGraph,
        estimation: Boolean
    ): MutableList<Long> {
        val toFind = vertices[about]!!.connections.keys.union(vertices[about]!!.shortcutConnections.keys)
            .filter { x -> !vertices[x]!!.deleted }.toHashSet()
        HashSet<Long>()
        val F = PriorityQueue<DoubleTuple>()
        val dist = HashMap<Long, Double>()
        val prev = HashMap<Long, Long>()
        var numsettled = 0
        dist[from] = 0.0
        prev[from] = -1
        var u: Long
        var toAdd = DoubleTuple(from, 0.0)
        F.add(toAdd)
        while (F.size != 0) {
            val item = F.poll()
            u = item.id
            if (dist[u] != item.dist) continue
            toFind.remove(u)
            numsettled += 1
            for (neighbour in vertices[u]!!.connections.keys) {
                var alt = dist[u]!! + vertices[u]!!.connections[neighbour]!!
                if (!dist.containsKey(neighbour) || alt < dist[neighbour]!!) {
                    toFind.remove(neighbour)
                    dist[neighbour] = alt
                    prev[neighbour] = u
                    toAdd = DoubleTuple(neighbour, dist[neighbour]!!)
                    F.add(toAdd)
                }
            }
            if (toFind.size == 0 || (numsettled == 10000 && estimation)) break
        }
        val shortestThrough = mutableListOf<Long>()
        for (i in vertices[about]!!.connections.keys.union(vertices[about]!!.shortcutConnections.keys)
            .filter { x -> !vertices[x]!!.deleted }) {
            if (i == from) continue
```

```
            if (!prev.containsKey(i)) shortestThrough.add(i) //i has not been reached because contraction has been terminated early, therefore a shortcut is necessary
            else {
                val route = inputGraph.solution(i, prev)
                if (route.contains(about)) shortestThrough.add(i) //The shortest route goes through the node therefore a shortcut is necessary
            }
        }
        return shortestThrough
    }

    /**
        * Works out the edge difference and the number of deleted neighbours and uses these to calculate a heuristic
        * The edge difference is  equivalent to the number of shortcuts that would be added if the node were to be deleted
        * or the number of the routes between adjacent nodes that pass through this node
        * minus the number of routes that originally go through this node.
        */
    private fun getHeuristicValue(node: Long, inputGraph: GeographicGraph): Int {
        val nodeObj = vertices[node]!!
        var count = 0
        for (from in nodeObj.incomingConnections.keys.union(nodeObj.incomingShortcuts.keys)
            .filter { x -> !vertices[x]!!.deleted }) {
            count += getShortest(from, node, inputGraph, true).size
        }
        count -= (nodeObj.incomingConnections.keys.union(
            nodeObj.incomingShortcuts.keys.union(
                nodeObj.connections.keys.union(
                    nodeObj.shortcutConnections.keys
                )
            )
        )).filter { x -> !vertices[x]!!.deleted }.size
        count += (nodeObj.incomingConnections.keys.union(nodeObj.connections.keys)).count { x -> vertices[x]!!.deleted }//maintains uniformity
        return count
    }
}
```

src/main/kotlin/IntTuple.kt

```
class IntTuple(val id : Long, val dist : Int) : Comparable<IntTuple>
{
    override fun compareTo(other: IntTuple): Int {
        if(dist > other.dist)
        {
            return 1
        }
        else if(dist == other.dist)
        {
            return 0
        }
        else
        {
            return -1
        }
    }
}
```

src/main/kotlin/DoubleTuple.kt

```
class DoubleTuple(val id : Long, val dist : Double) : Comparable<DoubleTuple>
```

```
{
    override fun compareTo(other: DoubleTuple): Int {
        if(dist > other.dist)
        {
            return 1
        }
        else if(dist == other.dist)
        {
            return 0
        }
        else
        {
            return -1
        }
    }
}
```

src/main/kotlin/GeographicGraphTest.kt

```
import org.junit.Test
import org.junit.jupiter.api.Assertions.*
import kotlin.math.abs

internal class GeographicGraphTest {
    var testMap = OpenStreetMap("maps/testarea.osm")
    var testGraph = testMap.cyclableGraph

    @Test
    fun addEdge() {
        var testNodeOne: Long = 1767962872
        var testNodeTwo: Long = 8883453446
        testGraph.addEdge(testNodeOne, testNodeTwo, false)
        assert(
            testGraph.vertices[testNodeOne]!!.connections.contains(testNodeTwo) &&
   testGraph.vertices[testNodeTwo]!!.connections.contains(
                testNodeOne
            )
        )
        testNodeOne = 1767962860
        testNodeTwo = 8889691941
        testGraph.addEdge(testNodeOne, testNodeTwo, true)
        assert(
            testGraph.vertices[testNodeOne]!!.connections.contains(testNodeTwo) && !
   testGraph.vertices[testNodeTwo]!!.connections.contains(
                testNodeOne
            )
        )
    }

    @Test
    fun gatherWeights() {
        for (vertice in testGraph.vertices.keys) {
            testGraph.vertices[vertice]!!.weight = 0.0
        }
        testGraph.gatherWeights()
        var total = 0.0
        for (vertice in testGraph.vertices.keys) {
            total += testGraph.vertices[vertice]!!.weight
        }
        assertEquals(3570.0, total)

    }
```

```

    @Test
    fun getRandomId(){
        assert(testGraph.getRandomId() in testGraph.vertices)
    }

    @Test
    fun contractGraph(){
        testGraph.contractGraph(10.0)
    }

    @Test
    fun getDistance() {
        assert(abs(testGraph.getDistance(0.0, 0.0, 1.0, 0.0)-111)<1)
    }
    @Test
    fun getDistanceCost() {
        assertEquals(0.1868205192755297, testGraph.getDistanceCost
    (1767962851,1767962820))
    }


    @Test
    fun pruneDisconnected() {
        val size = testGraph.vertices.size
        testGraph.pruneDisconnected()
        assertEquals(size, testGraph.vertices.size)//everything should already have
    been pruned that can be pruned
    }

    @Test
    fun getTurnCost() {
        assertEquals(testGraph.getTurnCost(1767962851, 1767962853, 1767962820, 10.0),
    10.0)
        assertEquals(testGraph.getTurnCost(8883476987, 1767962831, 1767962838, 10.0),
    0.0)
    }

    @Test
    fun findRoute() {
        var route = testGraph.findRoute(8861811044, 8864562252, 10.0, 10.0, false)
        var distance = 0.0
        for (i in 1 until route.size)
            distance += testGraph.getDistanceCost(route[i - 1], route[i])
        assertEquals(0.14516271195954372, distance)
    }
}
```

src/main/kotlin/ContractableGraphTest.kt

```
import org.junit.jupiter.api.Test

import org.junit.jupiter.api.Assertions.*

internal class ContractableGraphTest {
    var testGraph = OpenStreetMap("maps/testarea.osm").cyclableGraph

    @Test
    fun createGraph() {
        val contractedGraph = ContractableGraph(15.0)
        contractedGraph.createGraph(testGraph)
    }
```

```

    @Test
    fun findRoute() {
        val contractedGraph = ContractableGraph(15.0)
        contractedGraph.createGraph(testGraph)
        contractedGraph.contractGraph(testGraph)
        val route = contractedGraph.findRoute(8861811044, 8864562252,testGraph,false)
        var distance = 0.0
        for (i in 1 until route.size)
            distance += testGraph.getDistanceCost(route[i - 1], route[i])
        assertEquals(0.14516271195954372,distance)
    }
}
```

src/main/kotlin/OpenStreetMapTest.kt

```
import org.junit.jupiter.api.Test

import org.junit.jupiter.api.Assertions.*

internal class OpenStreetMapTest {
    @Test
    fun createNewMap(){
        val map = OpenStreetMap("maps/ways.osm")
    }
}
```

MainKtTest.kt

```
import kotlinx.serialization.ExperimentalSerializationApi
import kotlinx.serialization.cbor.Cbor
import kotlinx.serialization.encodeToByteArray
import org.junit.jupiter.api.Test

import org.junit.jupiter.api.Assertions.*

internal class MainKtTest {

    @OptIn(ExperimentalSerializationApi::class)
    @Test
    fun writeMapToDisk() {
        val map = OpenStreetMap("maps/testarea.osm")
        writeMapToDisk("/tmp/savedMap.bin",map)
        var newMap = readMapFromDisk("/tmp/savedMap.bin")
        assertEquals(map.cyclableGraph.vertices.size,newMap.cyclableGraph.vertices.
size)
    }

    @Test
    fun readMapFromDisk() {
        val map = readMapFromDisk("savedGraph.bin")
        map.cyclableGraph.findRoute(map.cyclableGraph.getRandomId(),map.cyclableGraph.
getRandomId(),10.0,2.0,false)
    }

}
```

D

# Evidence of Specific Marking Criteria

# Bibliography


[1] Transport For London. *AccidentStats API Documentation Page*. URL: https://api-portal.tfl.gov.uk/api-details#api=AccidentStats&operation=AccidentStats_Get. (accessed: 17.03.2022).

[2] geofabrik. *Open Street Map mirror*. URL: https://download.geofabrik.de.

[3] GeoJSON developers. *GeoJSON website*. URL: https://geojson.org/.

[4] Stuart Grange. *GeoJSON download site*. URL: https://skgrange.github.io/data.html.

[5] Open Street Map Wiki. *Open Street Map Wiki Elements Page*. URL: https://wiki.openstreetmap.org/wiki/Elements. (accessed: 17.03.2022).

[6] dom4j. *dom4j documentation site*. URL: https://dom4j.github.io.

[7] Open Street Map Wiki. *Open Street Map Wiki Key:highway Page*. URL: https://wiki.openstreetmap.org/wiki/Key:highway. (accessed: 18.03.2022).

[8] Norbert Beckmann et al. "The R*-tree: an efficient and robust access method for points and rectangles". In: *Proceedings of the 1990 ACM SIGMOD international conference on Management of data - SIGMOD '90*. ACM Press, 1990. DOI: 10.1145/93597.98741. URL: https://doi.org/10.1145/93597.98741.

[9] David Moten. *Library Github Page*. URL: https://github.com/davidmoten/rtree2.

[10] *Kosaraju's algorithm*. Jan. 2022. URL: https://en.wikipedia.org/wiki/Kosaraju%27s_algorithm.

[11] Wikipedia Editors. URL: https://en.wikipedia.org/wiki/Dijkstra's_algorithm#Pseudocode. Accessed (19.03.2022).

[12] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. DOI: 10.1007/bf01386390. URL: https://doi.org/10.1007/bf01386390.

[13] Peter Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/tssc.1968.300136. URL: https://doi.org/10.1109/tssc.1968.300136.

[14] Dweisman. URL: https://commons.wikimedia.org/wiki/File:Triangle_with_notations.svg.

[15] J. de Mendoza y Ríos. *Memoria sobre algunos metodos nuevos de calcular la longitud por las distancias lunares y explicaciones prácticas de una teoría para la solución de otros problemas de navegación*. Imp. Real, 1795. URL: https://books.google.co.uk/books?id=030t0OqlX2AC.

[16] Timothy Langer. *How do travel planners work?* URL: https://www.youtube.com/watch?v=wLNai11U2tI.

[17] Robert Geisberger et al. "Exact Routing in Large Road Networks Using Contraction Hierarchies". In: *Transportation Science* 46.3 (Aug. 2012), pp. 388–404. DOI: 10.1287/trsc.1110.0401. URL: https://doi.org/10.1287/trsc.1110.0401.

[18] *Serialization: Kotlin*. URL: https://kotlinlang.org/docs/serialization.html.