Cycle routes: connecting the network

Jiaxin Wang

Supervisor: Dr Michael Young

University of St Andrews

August 7, 2023

Abstract

Declaration

I declare that the material submitted for assessment is my own work except where credit is explicitly given to others by citation or acknowledgement. This work was performed during the current academic year except where otherwise stated.

The main text of this project report is NN,NNN words long, including project specification and plan.

In submitting this project report to the University of St Andrews, I give permission for it to be made available for use in accordance with the regulations of the University Library. I also give permission for the title and abstract to be published and for copies of the report to be made and supplied at cost to any bona fide library or research worker, and to be made available on the World Wide Web. I retain the copyright in this work.

Contents

List of Figures

Chapter 1

Introduction

In this chapter, we would discuss the motivation behind this project. We would present a list of aims and objectives, and give a brief overview of the structure of this report.

1.1 Motivation

With the climate crisis, there has been a big push from many facets of society to reduce car usage [4]. Encouraging more people to cycle over short distances instead of driving would be a big step towards reaching that goal. A study by Sloman et al. suggests that investment in cycling leads to increased cycle levels [25]. Many cities and towns have invested money into creating new cycling infrastructure, such as cycle paths alongside roads or standalone cycleways. Public budgets are always squeezed, so it is important to prioritize infrastructure improvements in the places where they would have the most impact.

Given the street map data of an area, it is possible to analyse the existing cycle infrastructure and identify the places where improvement is most needed. For each path in the street map, we can evaluate its suitability for safe cycling using data such as speed limits and cycle lanes. For example, a designated cycle lane is considered to be more cycle-friendly than a busy main road. If we consider the subgraph which consists of cycle-friendly paths only, we could potentially identify isolated areas that are disconnected from the rest of the graph. It might be safe to cycle within each area, but it could be difficult for cyclists to commute from one area to another. An improvement to the cycle network would be to add cycle-friendly paths to connect

such areas.

There are many network analysis tools implemented in various programming languages, but users without any technical expertise could struggle to apply such tools to the map data. There is a need for a system which allows less technical users to analyse street maps with respect to cycling infrastructure.

1.2 Aims and Objectives

The aim of this project is to implement an automated system which provides recommendations for additional cycle paths by analysing the map of St Andrews. The system should be extensible to other areas without any manual data processing.

The primary objectives are listed below:

1. Develop an automated process that turns the OpenStreetMap data for an area into a graph annotated with data relevant to cycle accessibility and apply this to St Andrews
2. Develop a set of configurable heuristics to determine whether a route is cycle friendly
3. Apply the heuristics to highlight disconnected components in the cycle-friendly subgraph
4. Consider other properties of the graph, such as k-connectivity and induced subgraphs, to produce other findings relevant to cycling
5. Suggest the most efficient paths to add to increase the connectedness of the subgraph

A list of secondary objectives are considered, with lower priority than the primary objectives:

1. Apply this analysis process to another similar area and assess how well the automated process works for areas it was not designed for
2. Apply the analysis process to a larger area to evaluate the scalability of the technique
3. Consider the cost of adding new paths when making suggestions

1.3 Report Structure

Chapter 2 gives an overview of graph theory which is relevant to this project and related work in this area. Chapter 3 concerns the structure of the system. The implementation of the system is discussed in Chapter 4, with focus on high-level algorithms. In Chapter 5, we evaluate the success and limitations of the system and compare it against existing approaches. Chapter 6 summarizes the project and discusses potential future work.

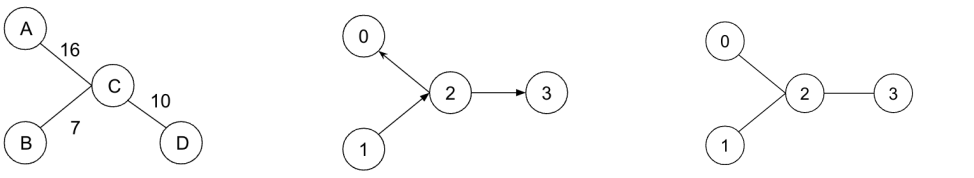
Chapter 2

Context Survey

This project is closely related to graph theory, and research has been done in this field. In Section 2.1, we would look into some important concepts in graph theory. Section 2.2 gives an overview of related work in this area. Tools and libraries that may be relevant to this project are discussed.

2.1 Graph Theory

A *graph* $G = (V, E)$ is a mathematical structure which consists of a set of vertices V and a set of edges E, where each edge is a pair of vertices in V [5]. A graph H is called a *subgraph* of G if $V(H) \subseteq V(G)$ and $V(H) \subseteq V(G)$. A *weighted* graph $G = (V, E, W)$ is a graph where each edge is assigned a weight value, and the weight mapping is stored in W. A *directed* graph is a graph where each edge is directed, whereas an *undirected* graph is a graph where edges have no direction. A *hypergraph* is denoted by $H = (V; \mathcal{E})$, where V is a set of vertices and $\mathcal{E}$ is a set of hyperedges. Each hyperedge joins any number of vertices.



(a) A weighted graph (b) A directed graph (c) An undirected graph

Figure 2.1: Three simple graphs

Three simple graphs are demonstrated in Figure 2.1. Figure 2.1a shows a weighted graph with $V = \{A, B, C, D\}$ and $E = \{AC, BC, CD\}$, where the weight of each edge is labelled next to the edge. An example of a directed graph and an undirected graph are shown in Figure 2.1b and 2.1c respectively. Figure 2.2 shows the Fano plane, which is an example of a hypergraph. It contains seven vertices as labelled on the diagram, and it has seven hyperedges: $\mathcal{E} = \{123, 145, 167, 246, 257, 347, 365\}$.

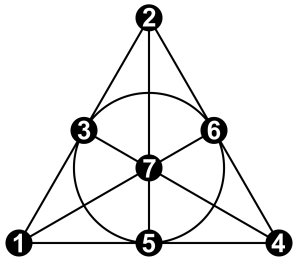


Figure 2.2: The Fano plane from Wikipedia [34]

Street maps can be considered as a type of hypergraph, where locations on the map are the vertices and streets joining various locations are the hyperedges. However, most implementations of graph algorithms are designed for graphs instead of hypergraphs. If we create a graph from the street map hypergraph, we can analyse the subgraph which only consists of cycle-friendly routes and identify isolated areas in the subgraph, where there are no connecting edges between the isolated area and other parts of the graph. In the following sections, we present some graph algorithms which can be useful in our task.

2.1.1 Connectivity

The *vertex-connectivity* of a graph is defined as the minimum number of vertices that need to be removed to result in a disconnected graph, and the *edge-connectivity* of a graph is the minimum number of edges that need to be removed to result in a disconnected graph [2]. A graph is *k-vertex-connected* if it has vertex-connectivity k.

Figure 2.3 shows a 4-vertex-connected graph. To separate the graph into two isolated subgraphs, the minimum number of vertices that need to be removed is four. For example, after removing the two left-most vertices and the two right-most vertices, we obtain two disconnected subgraphs, each of which consists of one vertex and no edges.

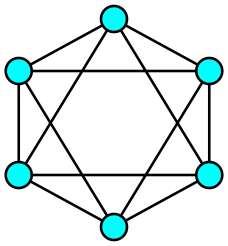


Figure 2.3: A 4-connected graph from Wikipedia [35]

The more connected the cycle-friendly subgraph is, the easier it is to get around the area with bicycles. An intuition to improve the cycle infrastructure would be to add paths that would increase the connectivity of the graph.

2.1.2 Graph traversal

If a graph S is a maximal connected subgraph of graph G, then S is a *component* of G [9]. We can analyse the connected components in the cycle-friendly graph and add edges between them to connect smaller cycle-friendly regions into a large cycle-friendly component. The set of components in an undirected graph can be found by traversing the graph. Running a search from a node n will give us all the reachable nodes from n, which form a component containing n. Therefore, running search algorithms from each unvisited node will yield the set of connected components in a graph.

There are two common examples of search algorithms, namely *breadth-first search* (BFS) and *depth-first search* (DFS). The BFS algorithm works by visiting each node in turn, starting with the closest nodes to origin and keeping the unvisited neighbours in a queue. The DFS algorithm runs in the same way, except it uses a stack to explore each branch in full before backtracking.

2.1.3 Shortest path

In order to measure the cost of cycling, we can look at the cost of the shortest paths. There are many algorithms for finding the shortest paths between nodes in a graph, perhaps the most famous of which is Dijkstra's shortest path algorithm [10]. It takes a starting point and works out the shortest path to all other reachable nodes. The algorithm is outlined in Algorithm 1 [32].

Consider the graph in Figure 2.4 [1], we can use Dijkstra's algorithm to find

Algorithm 1 Dijkstra's Algorithm

```
function DIJKSTRA(Graph, source)
    let Graph = (V, E, W)
    for v in V do
        dist[v] ← ∞
        prev[v] ← undefined
        add v to Q
    end for
    dist[source] ← 0
    while Q is not empty do
        u ← vertex in Q with min dist[u]
        remove u from Q
        for neighbour v of u still in Q do
            alt ← dist[u] + E(u, v)
            if alt < dist[v] then
                dist[v] ← alt
                prev[v] ← u
            end if
        end for
    end while
    return dist, prev
end function
```

the shortest distance from node A to every other node. During the process, we would keep a priority queue, which stores the shortest distance so far to a node from node A, with the smallest distance being at the front of the queue. After visiting the node A, the priority queue would look like this:

$$C = 2, B = 4, D = \infty, E = \infty$$

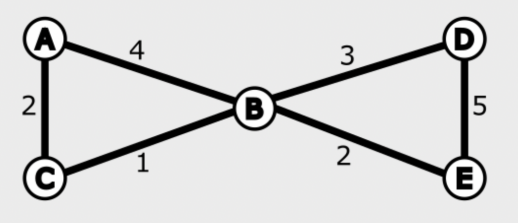


Figure 2.4: An undirected graph taken from freeCodeCamp [1]

This is because the shortest distance from A to B is 4, from A to C is 2. Then we choose a new unvisited node from the front of the priority queue, in this case, C. After visiting the node C, we can update the priority queue:

$$B = 3, D = \infty, E = \infty$$

C is now visited, so it is removed from the priority queue. Since $2 + 1 \leq 4$, the shortest distance between AB is updated to be 3. The next node to be visited is B, which is at the front of the priority queue. After visiting B, we have:

$$E = 5, D = 6$$

And after visiting E, we have:

$$D = 6$$

Finally, D is visited and there are no unvisited nodes left. We have computed that the shortest distance from A to every other node:

$$B = 3, C = 2, D = 6, E = 5$$

Another famous graph algorithm is Prim's algorithm, which finds the minimum spanning tree for a weighted undirected graph. The minimum spanning tree of a graph is defined to be the tree with the minimum total edge weights which connects all vertices [23]. An example of a minimum spanning tree is shown in Figure 2.5. In our task of improving the cycling infrastructure, we can use it to identify the paths that connect all areas with the lowest total distance and improve the graph based on this metric.

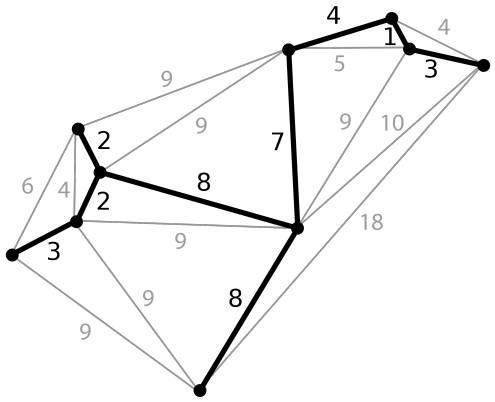


Figure 2.5: A minimum spanning tree example from Wikipedia [33]

2.2 Related Work

2.2.1 OpenStreetMap

OpenStreetMap [21] is an open, user-generated map database. It is licensed under the Open Data Commons Open Database Licence [14]. There are four types of data in OSM [27]:

- A node, which is a location on the Earth's surface with longitude and latitude
- A way, which is an ordered list of nodes representing roads and objects along the road
- A relation, which specifies the association among objects
- A tag, which is a key-value pair regarding information about an object

Figure 2.6 shows the map of St Andrews in CyclOSM [29], a bicycle map based on OpenStreetMap. It highlights bicycle infrastructure such as cycleways, and it includes information about highways such as speed limits. Ferster et al. [12] compared OSM data against municipal open data in six Canadian cities and concluded that OSM data was more extensive and detailed.

There are other maps that can be used to show bike lanes. For example, Google Maps (`https://www.google.com/maps`) is a widely used map. More than 1 billion people use it every month [24], and by 2018 it supported 40 languages [36]. It makes use of machine learning techniques to remove fake content and help keep the map accurate [16]. It also provides a good user

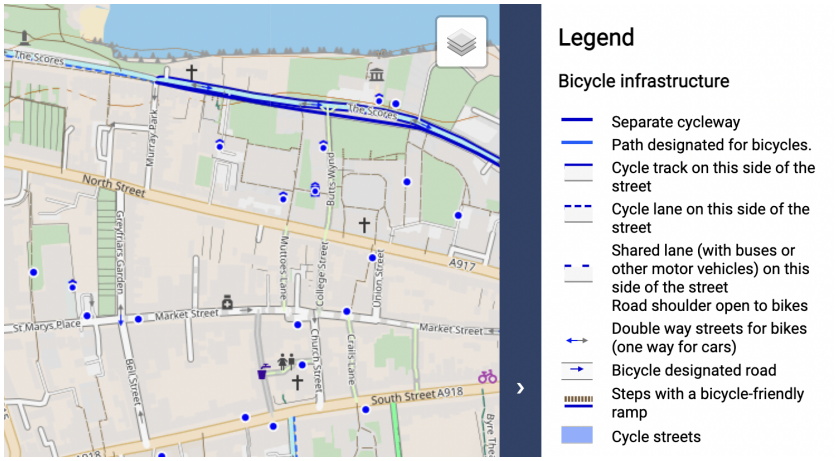


Figure 2.6: CyclOSM of St Andrews

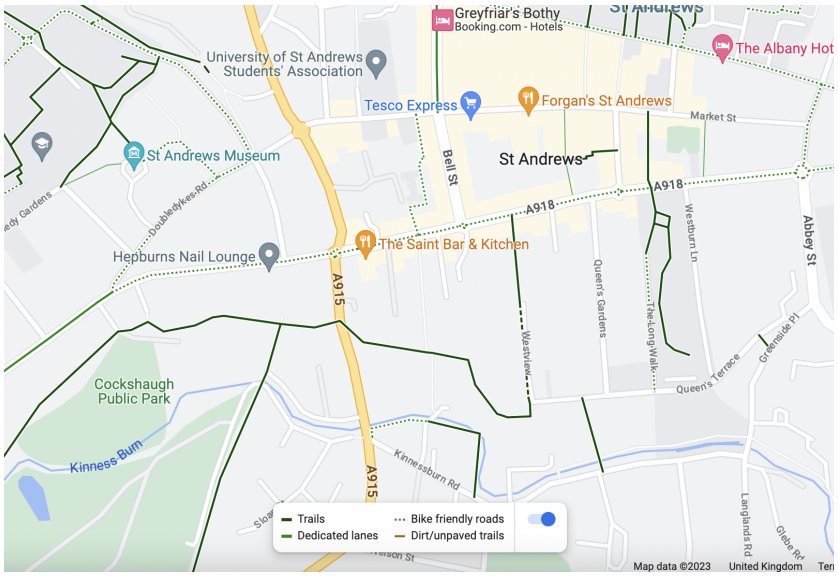


Figure 2.7: Google map of St Andrews

interface. Figure 2.7 shows the map of St Andrews in Google Maps in the cycling layer. One advantage that google have over OpenStreetMap is that the map data is not all contributed by volunteers, so it is much easier for them to keep their data consistent.

One disadvantage of Google Maps is that it is not open data. On the other hand, OpenStreetMap is open data, and it provides much more detailed information regarding bicycle infrastructure. The OpenStreetMap website [21] allows users to query features at any point on the map. Figure 2.8 shows

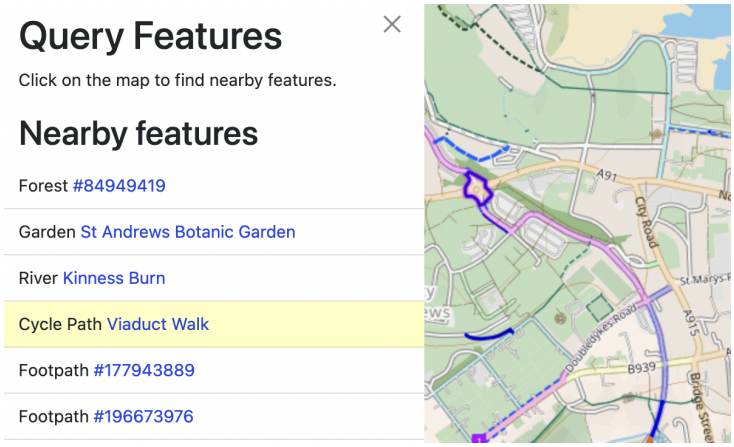


Figure 2.8: Querying features in OpenStreetMap

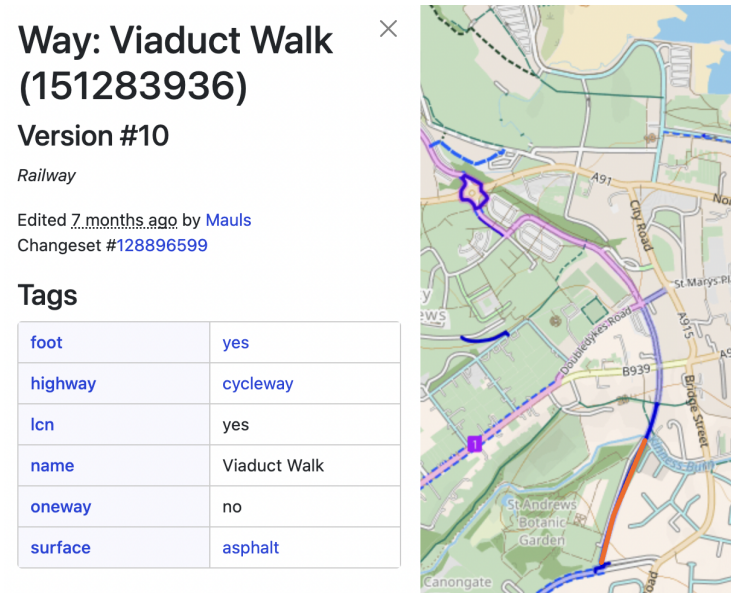


Figure 2.9: Details of Viaduct Walk

an example of queried features. Details of each feature can be viewed by clicking the blue text. For example, *Viaduct Walk* is a link to the information page of Viaduct Walk (Figure 2.9). It contains a series of tags which specify useful information such as the type of the road, whether pedestrians are allowed and whether it is one-way. It also contains a list of nodes which are on the path. Such information can be helpful for deciding whether a road is cycle-friendly.

2.2.2 Overpass API

OpenStreetMap exposes its data through an API, *Overpass API* [30]. This API means it is feasible to automate our system. It allows nodes, ways and relations to be queried according to a well-documented query language, *Overpass QL* [31].

For example, if we want to query nodes for post boxes within a certain area, the query would be

```
node["amenity"="post_box"]({{bbox}});
out;
```

where `bbox` is the bounding box for the area, specified by a 4-tuple `(south, west, north, east)` in longitude and latitude.

We could also query each OSM element by ID. To get the information associated with Viaduct Walk as shown in Figure 2.9, we could use

```
way(151283936);
out;
```

where 151283936 is the element ID of Viaduct Walk. This query returns the data in Figure 2.10.

We have access to the list of nodes on Viaduct Walk as well as the tags associated with it. Tags such as `highway=cycleway` are likely to be helpful in deciding whether an OSM way is cycle-friendly.

2.2.3 Applications of Graph Theory

In this section, we give an overview of how graph theory has been applied to transportation problems. Erath et al. [11] studied modern measures of the efficiency of transport metrics where they compared cost in the network against the straight-line distance, and showed that parts of the Swiss road network have reached growth limits, where it is hard to improve them any more owing to spatial limitations.

Derrible and Kennedy [8] described the history of graph theory with respect to transportation networks, paying particular note to the development of indicators describing the properties of transportation networks. They highlight the *scale-free* concept from network theory, where in some graphs the probability distribution of the number of connections a node has follows a power law and is not related to the number of edges or vertices in the graph. The process for adding cycle-friendly edges outlined in this project could promote

```
<way id="151283936">
    <nd ref="297685799"/>
    <nd ref="9097834868"/>
    <nd ref="9356781484"/>
    <nd ref="9356781485"/>
    <nd ref="287216154"/>
    <nd ref="1479980590"/>
    <nd ref="9356781486"/>
    <nd ref="1479980462"/>
    <nd ref="287216155"/>
    <nd ref="985936350"/>
    <nd ref="9356781487"/>
    <nd ref="9356781488"/>
    <nd ref="287216156"/>
    <nd ref="1478202890"/>
    <nd ref="1478202850"/>
    <nd ref="1478203126"/>
    <tag k="foot" v="yes"/>
    <tag k="highway" v="cycleway"/>
    <tag k="lcn" v="yes"/>
    <tag k="name" v="Viaduct Walk"/>
    <tag k="oneway" v="no"/>
    <tag k="surface" v="asphalt"/>
</way>
```

Figure 2.10: Query result of Viaduct Walk

a *scale-free* cycle-friendly network, since we are more likely to add edges to vertices which already have a high number of connections.

Zargham et al. [37] used spanning trees to quantify network reliability, another measure that may be of relevance to cyclists so they can be confident of an alternative route should there be any roadworks. They demonstrate an exact measurement of network reliability by looking at the set of all spanning trees in the graph and the probability of no failure for each spanning tree. This could be applied to road networks if we used historical data about roadworks along certain routes to estimate the probability of an edge being open.

2.2.4 Applications of Prim's Algorithm

There are many applications of Prim's algorithm. Wang et al. [28] developed an algorithm based on Prim's algorithm and used it to identify natural disasters in isolated areas. Fitina et al. [13] discussed applying Prim's algorithm in the context of tourism in Papua New Guinea, in order to reduce the expense of travelling between the islands. Iqbal et al. [19] used Prim's algorithm to optimize the planning of fibre optic cables to reduce the costs. We can see that Prim's algorithm is often used to find the most efficient overall routes, and we would like to improve these routes when building new cycling infrastructure.

2.2.5 Software and Tools

Below we discuss some libraries and tools that might be relevant to our project.

Overpass API

`overpy` [22] is a python wrapper to access the Overpass API. It supports querying with Overpass QL as well as parsing the response data to objects such as nodes and ways.

Graph visualization

`Matplotlib` [18] is a powerful tool for visualizing graphs in python. It was used to display the generated graphs in this project.

Graph analysis

`NetworkX` [17] is an open source python library for analysing graphs. It has implementations of standard graph algorithms such as graph traversal and cycle detection. Furthermore, it provides support for graph visualization with `Matplotlib`. It was used to perform graph analysis in this project.

`python-igraph` [7] is another useful library for graph manipulation. It also comes with standard graph algorithms and supports graph plotting with either `Matplotlib` or `Cairo` [3] which is a graphics library.

Version control and backup

`Git` was used for version control and the project was pushed to a private `GitHub` repository regularly so that it can be easily recovered in case of

machine failure.

Chapter 3

Design

In this chapter, we describe the structure of our system and outline some important design decisions.

3.1 System Structure

The system was divided into three layers: a **data fetcher** layer, a **model** layer and a **graph** layer. The data fetcher makes queries to the Overpass API and stores the street map data returned by the API. The model evaluates cycle-friendliness of the street map data based on certain criteria and uses it to construct nodes and edges. The graph layer builds a graph from the model and analyses the graph to suggest new paths. Figure 3.1 shows the structure of the system. This multi-layer structure also facilitates easier unit testing.

3.1.1 Data fetcher layer

All requests to the Overpass API are carried out by the data fetcher. An example query would be requesting all the ways within an area, specified by a bounding box. The result of the queries are stored in the data fetcher, and it exposes methods that are used by the **model** layer to select data.

3.1.2 Model layer

The model layer provides logical functions to process the data provided by the data fetcher. For example, given an OSM way, the model can evaluate its cycle-friendliness based on the tags associated with the way. In addition, the

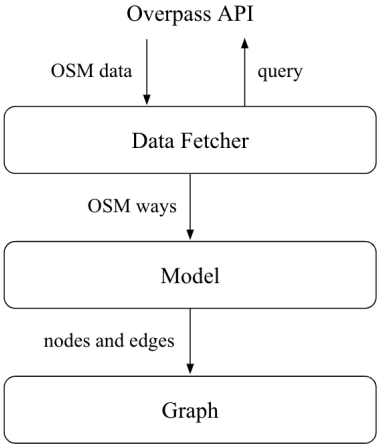


Figure 3.1: Structure of the system

model converts the OSM data to a graph structure which consists of nodes and edges. This step is necessary because each OSM way may contain more than two nodes, and most graph libraries such as `NetworkX` do not support edges with more than two nodes.

3.1.3 Graph layer

The graph layer constructs a `NetworkX` graph from nodes and edges provided by the model layer and performs analysis on it. It exposes methods to visualize and modify the graph, as well as to run various shortest path algorithms on the graph. If a framework other than `NetworkX` was preferred, all that would need to be replaced is the graph layer.

3.2 Data fetcher

The data fetcher contains the OSM data. This includes all the OSM nodes and ways within the area to be analysed. It should provide access to the relevant information regarding a node or a way. For example, we can store a mapping from node IDs to node objects, and in other layers (model and graph) we can use the lightweight node IDs to represent vertices, and we only access the actual node object when needed. This provides a form of abstraction: the model layer and the graph layer do not need to know the implementation details of OSM nodes and ways; they only need access to some properties such as node positions and way tags, which are provided by

the data fetcher.

Ideally, the OSM data returned by any requests can be cached in the data fetcher so that we do not make a new query for every OSM node we are interested in. The data fetcher should also provide some basic error handling, such as when some data is missing.

3.3 Model

The model processes the OSM data from the data fetcher. The main tasks of this layer include:

- selecting OSM ways based on their cycle-friendly scores, and
- converting OSM ways to vertices and edges to be used by the graph layer

The heuristic function to evaluate the cycle-friendliness of an OSM way lies in this layer. For the selection task, we keep a list of OSM ways whose scores are higher than some threshold. To convert OSM ways to a graph, we can process each OSM way in turn, convert OSM nodes to vertices and add edges to connect neighbouring nodes.

For path finding purposes, we would like to construct two graphs: a cycle-friendly graph from cycle-friendly OSM ways, and a city graph from all OSM ways. The logic to convert the OSM ways to a graph is the same; the only difference in the process is the selection of OSM ways. For the city graph, we simply do not perform the selection, or we select all OSM ways. The output of this layer should be nodes and edges of each graph.

There are several ways to represent a graph, using a list of edges or an adjacency list. We decided that the model layer should return an adjacency list, where each node ID is linked to a list of node IDs as its neighbours. This is the most natural way to represent our graph, as we are dealing with neighbouring nodes when processing OSM ways.

3.4 Graph

The graph receives an adjacency list from the model. To plot the graph, we need positions of each node, and this information is available in the data fetcher. The graph layer obtains node positions through the model layer, where the model requests this data from the data fetcher.

On receiving the layout of nodes, the graph layer computes edge lengths according to node positions. All graph analysis happens in this layer, including path finding and component analysis. It should also provide methods to display and save the plots.

3.4.1 Path finding strategies

In Section 1.2, we mentioned that we would like to suggest the most efficient path to add to the graph to improve the connectedness of the cycling network. However, the definition of the *most efficient* path varies depending on what the user needs the most. It could be that the user only wants to find the shortest path that would connect two big components, or maybe it is more important for the new path to be as central as possible. Three strategies with different priorities were designed to suggest paths.

Strategy A: Overall shortest path

A simple strategy to improve the connectedness of the cycle-friendly subgraph would be to add a path to connect the largest component (region 1) and the second largest component (region 2). Given the graph of the whole area, we search for the shortest path from any node in region 1 to any node in region 2. Making this path cycle-friendly would connect region 1 and region 2 in the cycle-friendly subgraph and hence improving connectedness.

Strategy B: Shortest path via town centre

One potential issue with strategy A is that the suggested path may be located at the edge of the town, which could be inconvenient for cyclists and its use rate might be low. To address this, strategy B attempts to find the shortest path from the most central node in region 1 to the most central node in region 2, where the most central node in a region is defined to be the node closest to the centre of the town.

Strategy C: Shortest path via local centre

Strategy C works in a similar fashion as strategy B. It searches for the shortest path from the most central node in region 1 to the most central node in region 2. However, the most central node in a region does not depend on the centre of the town. Instead, it is defined to be the node closest to the centre of the region, i.e. the centre of the component.

3.5 System Configuration

As discussed in Section 1.2, the system should be flexible so that it can be applied to another area. To achieve this, the following attributes were designed to be configurable:

- the bounding box of the area
- the heuristics to evaluate the cycle-friendliness of an OSM way
- the threshold for an OSM way to be considered cycle-friendly
- the strategy to find the most efficient way to improve cycle infrastructure

There should be a configuration file which is independent of the system code and can be passed to the system as an argument. In the following sections, we would discuss how each attribute can be configured.

3.5.1 Area specification

The area can be specified by two things: the *node ID* of the town or city, and the *bounding box* to run the analysis on. The node ID was required for uniquely identifying a place, and the bounding box was used to limit the data to the region.

3.5.2 Cycle-friendly heuristics

A heuristic function was designed to give a score to an OSM way depending on the tags associated with it to represent how cycle-friendly it is. The score was calculated by

$$h(way) = \frac{\sum_{t \in way.tags} w_t \times s_t}{\sum_{t \in way.tags} w_t} \tag{3.1}$$

where each tag t is a key-value pair, w_t is the weight assigned to tag t, s_t is the score gained for tag t to take its value, and $h(way)$ is the score given to the OSM way. This is essentially a weighted average of tag value scores, and both w_t and s_t are configurable.

Suppose in the configuration file we have two tags, *cycleway* and *smoothness*. We specify that the weight assigned to each tag is

$$w_{cycleway} = 1$$
$$w_{smoothness} = 0.5$$

And the score assigned to each tag value is

$$\begin{aligned} score(cycleway,\, track) &= 1 \\ score(cycleway,\, share\_busway) &= 0.6 \\ score(smoothness,\, excellent) &= 1 \\ score(smoothness,\, intermediate) &= 0.4 \end{aligned}$$

Now suppose we have an OSM way with the following tags:

$$cycleway = track, \quad smoothness = intermediate$$

The heuristic function should return

$$\begin{aligned} h(way) &= \frac{1 \times 1 + 0.5 \times 0.4}{1 + 0.5} \\ &= 0.8 \end{aligned}$$

Tags which do not appear in the configuration file are considered unimportant and have zero weight.

3.5.3 Cycle-friendly threshold

The heuristic function assigns a cycle-friendly score to each OSM way, and we would like to make the binary decision of whether a way should be classified as cycle-friendly based on its score. We could define a threshold in the configuration file, and consider a way to be cycle-friendly only if it gives a score higher than the threshold. The choice of the threshold value is important. If it is set too high, not many ways will be selected; if it is set too low, not much filtering will happen and the resulting graph will not give much useful information.

3.5.4 Strategies

User should be able to specify which strategies they prefer to use, as discussed in Section 3.4.1.

Chapter 4

Implementation

This chapter focuses on the implementation of our system.

4.1 Data Fetcher

All queries to the Overpass API were made in the data fetcher layer. Given the bounding box of an area to be analysed, we are interested in all the nodes and ways within this area. This data could be fetched using the query

```
nwr(south, west, north, east); out;
```

where the tuple `(south, west, north, east)` specifies the bounding box, and `nwr` stands for nodes, ways and relations.

We used `overpy` to perform the query and the result was cached in a `Result` object. The following was provided to the model layer:

1. All the OSM ways within a given area
2. A method to fetch all the nodes on a given OSM way
3. A method to get the location of a given OSM node, in longitude and latitude

The model needs access to all the OSM ways as well as all the nodes on each way so that it can analyse the cycle-friendliness of ways and convert ways to nodes and edges. The location of each node is helpful when plotting the graph.

The OSM ways in the area were stored in `Result.ways`, which is a list of `Way` objects. To get all the nodes on a given way, the `Way` object provides a

`get_nodes` method. The longitude and latitude of a node can be accessed by `Node.lon` and `Node.lat`.

It should be noted that when querying data within a bounding box, the returned `Way` may not be completely inside the bounding box, since Overpass API would return all the ways including the ones that are only partly inside the boundary. This means that when we fetch all the nodes on a given way, some nodes may be outside the bounding box. To keep the graph clean and focused, we would like to disregard nodes that are outside the boundary.

4.2 Model

The model layer handles the filtering of OSM ways. Since we were interested in the cycle-friendly subgraph, we implemented a heuristic function that gives a cycle-friendly score to each way by analysing the tags associated with it, as defined in Equation 3.1.

Two data classes, `Tag` and `WeightedTags`, were created to store the weighting of the tags. `Tag` has two attributes:

- `weight`, which is a float representing the tag weight
- `values`, which maps each tag value to a score

On the other hand, `WeightedTags` stores a mapping from a tag key to a `Tag`. It also provides a method, `weight_sum`, which calculates the sum of weight for all the tags. These data classes facilitated the implementation of Equation 3.1. With the score function, we could select cycle-friendly ways based on a given threshold.

The next step was to convert OSM ways to vertices and edges. Since each OSM way could consist of more than two nodes, a simple method to convert it into edges would be breaking the way at each intermediate node. For example, consider an OSM way represented by

$$A - B - C - D$$

Since each way is an ordered list of nodes, we could represent it using three edges:

$$A - B, B - C, C - D$$

The number of vertices and edges we obtained using this approach would greatly depend on the size of the area. To simplify the graph, we would like to keep nodes that are either at the beginning or the end of a way, or if they

appear on more than one way. We introduced a *link counter* to keep track of the number of links each node has. This is done by going through all the ways and incrementing the counter for each node on the way.

Then, we can use the *link counter* to convert OSM ways into edges, represented by an adjacency list. For each OSM way, we look at every intermediate node, i.e. any node that is not at the beginning nor the end of the way. If a node has more than one link, we keep the node and break the way at this node; otherwise, we ignore the node. The pseudocode for this algorithm is demonstrated in Algorithm 2.

Algorithm 2 An algorithm to convert hyperedges to an adjacency list

Input: $hyperEdges, counter$
Output: adj
  $adj \leftarrow \emptyset$
  for e **in** $hyperEdges$ **do**
    $currentPointer \leftarrow 0$
    $currentNode \leftarrow e[currentPointer]$
    $nextPointer \leftarrow currentPointer + 1$
    while $nextPointer < length(e)$ **do**
      $nextNode \leftarrow e[nextPointer]$
      if $counter[nextNode] > 1$ **or** $nextPointer = length(e) - 1$ **then**
        if $currentNode \neq nextNode$ **then**
          $neighbours \leftarrow$ empty list
          if $currentNode$ **in** adj **then**
            $neighbours \leftarrow adj[currentNode]$
          end if
          add $nextNode$ to $neighbours$
          $adj[currentNode] \leftarrow neighbours$
          $currentPointer \leftarrow nextPointer$
          $nextPointer \leftarrow currentPointer + 1$
        end if
      else
        $nextPointer \leftarrow nextPointer + 1$
      end if
    end while
  end for

4.3 Graph

In the graph layer, we construct a graph from vertices and edges provided by the model layer. Two graphs were created: a graph that only consists of cycle-friendly edges, known as the *cycle-friendly graph*, and a graph which consists of all edges regardless of their cycle-friendliness, known as the *city graph*. Our goal was to find a path in the city graph that if it were to be made cycle-friendly, the connectedness of the cycle-friendly graph would improve the most.

4.3.1 Choice of graph libraries

Two graph libraries, `NetworkX` and `python-igraph`, were experimented with during the implementation. We discovered that `NetworkX` produced better plots using `Matplotlib` than `python-igraph` did. Figure 4.1 and 4.2 shows a plot produced by `python-igraph` and `NetworkX` respectively. We can see that the `igraph` plot is much more compact, and it is very difficult to distinguish nodes next to each other. On the other hand, `NetworkX` provides better colouring settings and the resulting plot is tidier. Therefore, `NetworkX` was the main library used in the graph layer.

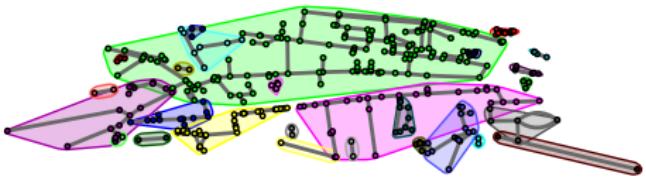


Figure 4.1: A plot of St Andrews produced by `python-igraph`

4.3.2 Plotting

We obtained nodes and edges from the OSM data in the model layer, and we would like to plot the graph to see how well it represents the St Andrews area. To achieve this, each node must be given a position. Since we had access to the longitude and latitude of a node, we created a mapping from node ID to node coordinates, and this was used as a position attribute assigned to each node. Figure 4.3 shows the plot of the cycle-friendly graph overlaying the OpenStreetMap of St Andrews.

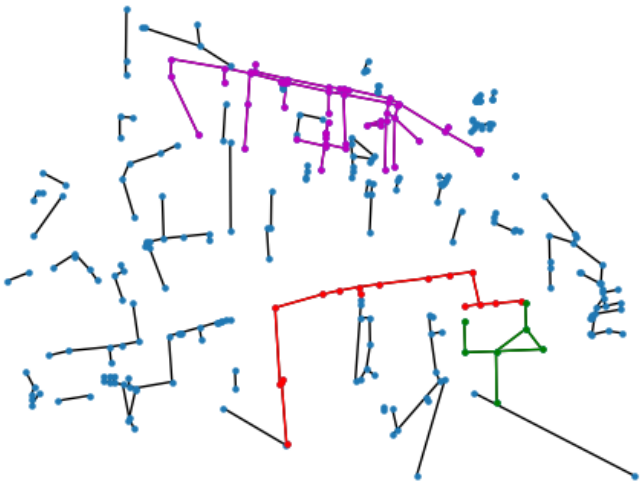


Figure 4.2: A plot of St Andrews produced by `NetworkX`

4.3.3 Geometric edges

A closer look at the constructed graph reveals a problem. Figure 4.4a shows part of the cycle-friendly graph, where the red region is the largest connected component in the graph. We can see that the right most part of the red region appears to be connected to the edge next to it, but they were actually not connected in the graph. Figure 4.5 shows the corresponding part in OpenSteetMap. Although Lade Braes Walk (vertical red dotted line) and Queen's Terrace (horizontal white solid line) do not share any nodes on OSM, it seems like they do have an intersection, so they should be considered as connected. In addition, Figure 4.6 shows the Google Maps street view of Queen's Terrace, where Lade Braes Walk is clearly accessible.

There are many other examples in our constructed graph where nodes are very close to each other but disconnected, even though in real life it is possible for a cyclist to move from one node to another. To address this, for every pair of nodes that are geographically close to each other, we added an edge to connect them. Conveniently, `NetworkX` provides a method called `geometric_edges` which computes the edge list of node pairs within a certain radius of each other. This radius is configurable in the configuration file. Figure 4.4b shows the graph after connecting close nodes, with radius set to 0.0005. The red region which represents the largest component is now larger, connecting nodes and edges that are nearby.

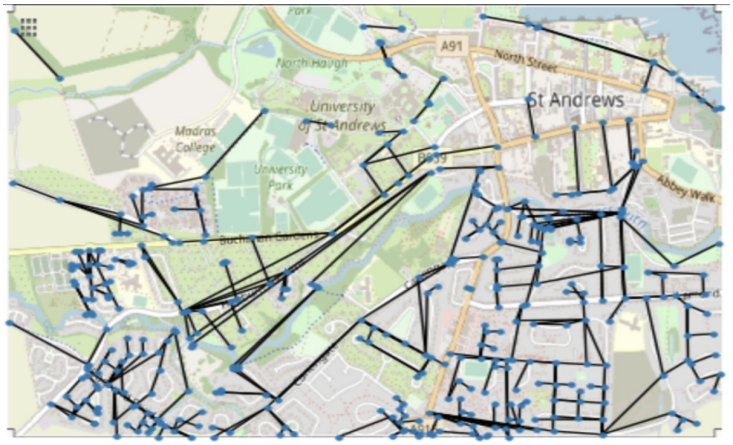


Figure 4.3: The cycle-friendly graph overlaying the OpenStreetMap

(a) Before adding geometric edges

(b) After adding geometric edges

Figure 4.4: Part of the cycle-friendly graph

4.3.4 Connected components

In order to identify isolated islands in the cycle-friendly graph, we would like to analyse the connected components in the graph. `NetworkX` provides methods to find connected components, and we are interested in connecting the largest component with the second largest component. There are a few possible definitions of *the largest* component; it could be the component that contains the most number of nodes, or the most number of edges, or the area it spans is the largest. We decided to use area as the metric as it would make the most sense - it is better to connect geographically big areas than to connect small areas with many nodes.

Now the question becomes, how do we measure the area of a component? We adapted a simple but effective method: for each component, we compute the total length of all the edges inside the component. The longer the edges are, the more likely it is that the component is going to span across a big

Figure 4.5: The corresponding part of Figure 4.4 in OpenStreetMap

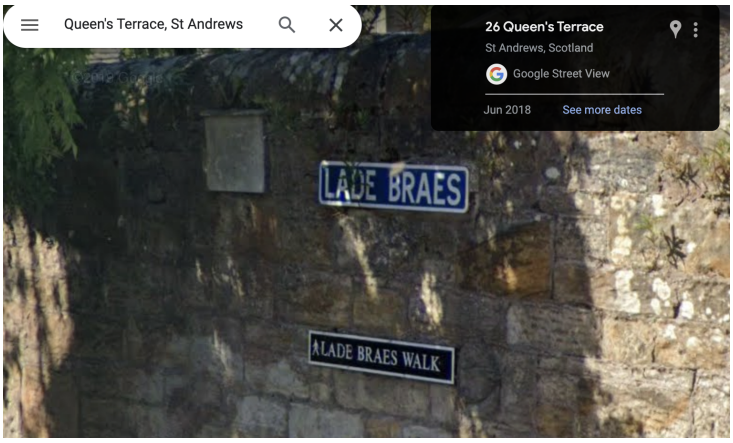


Figure 4.6: Google Maps street view of Queen's Terrace

area. Then, we sorted the connected components using this criterion and identified the two largest component, as demonstrated in Figure 4.7.

Another approach we considered was to measure the geographical area enclosed by the component. This was challenging because the formulae for calculating the area of an irregular polygon assume that its boundary is known. However, we have a large collection of nodes, and it is difficult to distinguish which nodes are on the boundary and which are not. One possible way to find the boundary was to take the extreme compass points, i.e. the nodes with the greatest and least longitude and latitude. The problem with this approach was that we may not always end up with more than two nodes, for example, when the north-most node is also the east-most node,

and the south-most node is also the west-most node. In this case, the nodes would form a line which has no area.

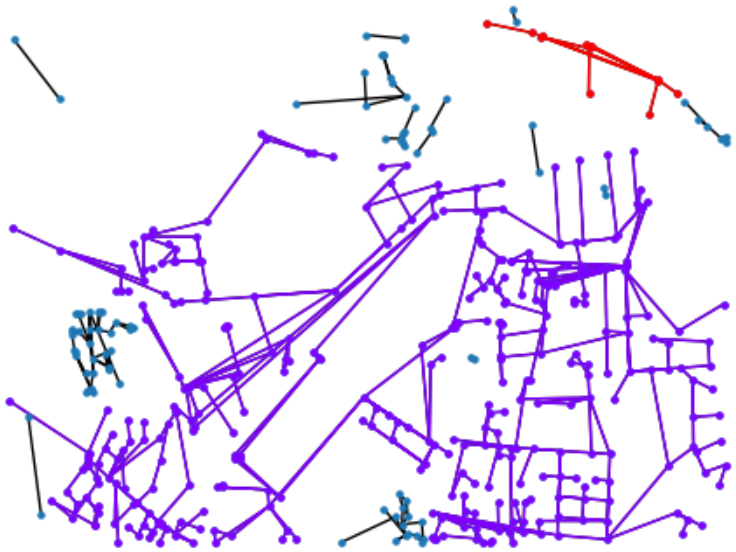


Figure 4.7: Connected components in the cycle-friendly graph

4.3.5 Weighted graph

Our strategies involve running the shortest path algorithm on the graph. This shortest path should reflect on the actual length of the path in the map, which means we have to assign weights to edges. This could be achieved easily if we were provided with the length of each OSM way, but this data is not available on the Overpass API. Hence, we would like to estimate the length of an edge based on node coordinates. Given two nodes, u and v, and their coordinates, $u = (u_0, u_1)$ and $v = (v_0, v_1)$, where u_0, v_0 is the latitude and u_1, v_1 is the longitude, we can compute the geodesic distance between the nodes using `geopy` [15], a python package for geocoding. This distance could be used as the weight of the edge uv.

4.3.6 Strategies

Recall that we discussed three strategies in Section 3.4.1. All these strategies involve finding the shortest path. `NetworkX` provides an implementation of Dijkstra's algorithm, and this was utilized differently for each strategy.

Strategy A: Overall shortest path

To find the overall shortest path between two regions, we ran Dijkstra's algorithm from every node in region 1 to every node in region 2. For each path found, we kept track of the length of the path as well as the path itself, in the form of a list of nodes on the path. Each time a shorter path was found, we replaced the stored path with the new shorter path. Eventually, we ended up with the overall shortest path between the two regions.

Strategy B: Shortest path via town centre

In OpenStreetMap, there is a node representing the centre of St Andrews. Since OSM nodes have coordinates, we used this to find the nearest node to the centre of St Andrews in each region. The distance from a node to the centre was calculated using the geodesic distance. After finding the most central node for each region, we used Dijkstra's algorithm to compute the shortest path between these nodes.

Strategy C: Shortest path via local centre

In Strategy C, we find the nodes that are central within each region. A method called `NetworkX.centre` was used to obtain the set of central nodes of a graph, where the maximal distance from the central nodes to other nodes in the graph is minimized. We could construct a subgraph for each region and find the central nodes. For simplicity, we took the first returned node as the centre of the region. After that, we ran Dijkstra's algorithm from the central node in region 1 to the central node in region 2.

4.4 Testing Approach

As our system was implemented in a multi-layer architecture, we were able to test each layer independently. The following python libraries were used for testing:

- `unittest` [26], which is a python framework for unit testing
- `unittest.mock` [20], which is used to mock objects
- `coverage` [6], which is a tool for measuring test coverage

The testing results are shown in Appendix A. In the following sections, we give an overview of how unit testing was carried out for each layer.

4.4.1 Data Fetcher

Most methods provided in the data fetcher are attribute getters, so unit tests were written to verify the correct attribute was returned by each method. We used mock OSM ways and nodes for testing so that we could set their attributes without the need to make a request to the Overpass API. In addition, the `patch` decorator was used to create a mock object of the Overpass API, so that the unit tests would not be dependent on the Overpass API server.

4.4.2 Model

Unit tests were written for each method in model, with edge cases considered where applicable. A mock data fetcher was used when creating the model so that we could focus on the behaviour of model without being affected by the data fetcher. Similar to data fetcher tests, we used mock OSM ways when testing methods such as `eval_way`, the heuristic function to give a score to an OSM way. This allows us to configure the tags associated with each way easily, and the method was tested with many configurations.

4.4.3 Graph

In the graph layer, most of the code were written to display a plot and make function calls to the `NetworkX` library. It was difficult to write unit tests for graph visualization, and widely-used libraries such as `NetworkX` are likely to be well-tested. Hence, tests were only written for auxiliary methods in this layer which do not involve plotting or graph analysis in `NetworkX`.

Chapter 5

Evaluation

In this chapter, we reflect on the goals and objectives of this project. We first present some plots produced by the system, and we compare our system against other approaches. In addition, we discuss the limitations of our system.

5.1 Configuration

As discussed in the design and implementation chapters, the system is designed in such a way that the choice of which tags contribute to the cycle-friendly score are the responsibility of the user. A default configuration file was created, using the legend and documentation of CyclOSM [29] as inspiration. CyclOSM uses OpenStreetMap data to highlight cycling infrastructure. Similarly to this project, it utilises the tags of ways, so it is a relatively complete reference for the most important tags to be considered. As mentioned in the context survey, one difficulty with OpenStreetMap is that since different groups of mappers have their own conventions, there are a wide variety of tag-value mappings that represent the same things. As a consequence, CyclOSM does not cover every relevant combination of tags. The extensible configuration means that a local town planner could adapt the system to suit the conventions of the local area.

5.2 Cycle-friendly Graph and CyclOSM

We compared the cycle-friendly graph generated by our system against CyclOSM [29]. Figure 5.1 shows the map of St Andrews in CyclOSM, and the

map legend is shown in Figure 5.2. The cycle-friendly graph produced by our system using the default configuration is shown in Figure 5.4b.

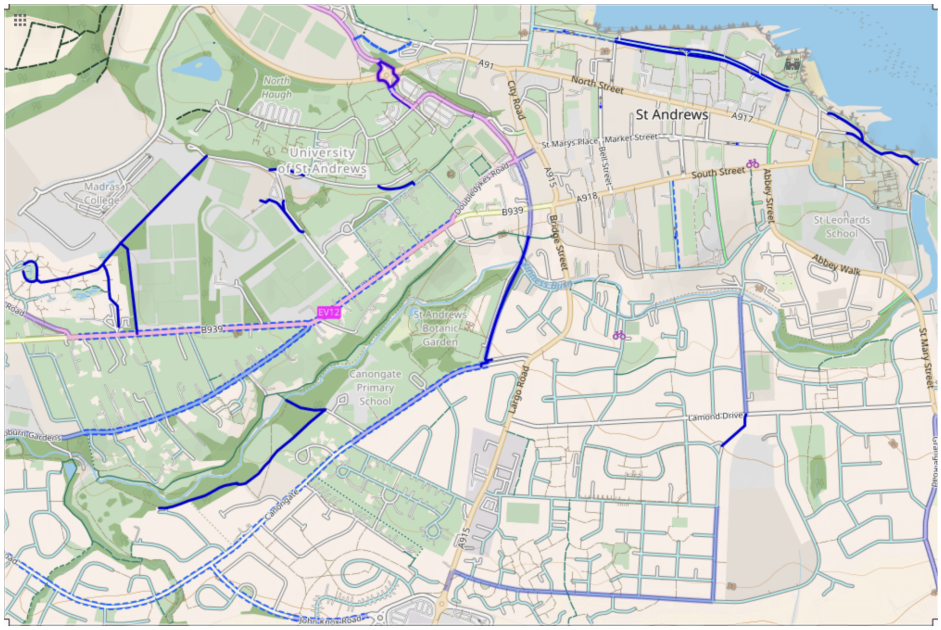


Figure 5.1: CyclOSM of St Andrews

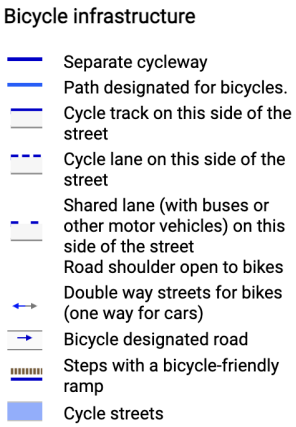


Figure 5.2: CyclOSM legend of bicycle infrastructure

An earlier produced graph is displayed in Figure 5.4a. Upon a closer look at some of the paths we discovered an omission in the CyclOSM view. There are a few streets which have a cycle path where the value for the `cycleway` tag is `opposite`, however this value is not considered by CyclOSM. Therefore,

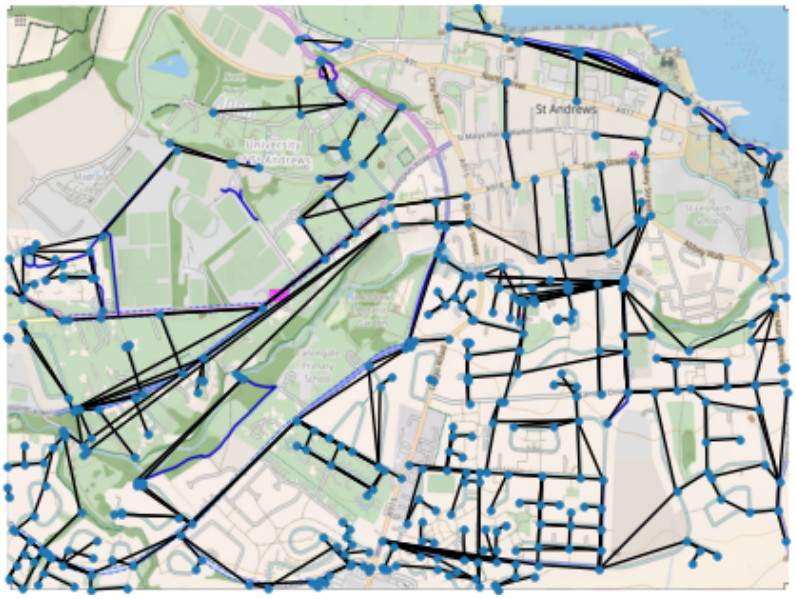


Figure 5.3: The cycle-friendly graph overlaying the CyclOSM

this tag was added to the default configuration file, and the more complete cycle-friendly graph was produced (Figure 5.4b).

Figure 5.3 shows the cycle-friendly graph overlaying the CyclOSM of St Andrews. We can see that ways with a separate cycle way and paths designated for bicycles are identified in our plot. On the other hand, the busy main roads such as *North Street* and *South Street* do not appear, which is sensible as they are not very safe to cycle on.

We also identified some additional cycle-friendly paths, which fall into two categories:

1. Roads that have a cycle path but were not highlighted by CyclOSM for the reasons discussed earlier, such as *Greyfriars Garden*.
2. Roads that do not have a cycle path, but which our heuristic deemed cycle-friendly, such as *Bell Street*. In this case it is a small tertiary road, so it would likely be acceptable for most cyclists to cycle through there.

Since *Bell Street* connects to *Greyfriars Garden*, it would seem reasonable for cyclists to use this route to go from *South Street* to *North Street* using the existing cycling infrastructure. This route is not highlighted in CyclOSM, but it is considered cycle-friendly in our plot. Note that one limitation of our system is that we don't consider direction - *Bell Street* is one way for cyclists

as well as cars, so it would not be so easy to use this route to go from *North Street* to *South Street*.

5.3 Connectivity Analysis

5.4 Application to Other Areas

In order to evaluate the scalability of our approach, we have applied the automated process to two other cities: Dundee and Cambridge.

5.5 Comparison to Other Approaches

5.6 Limitations

5.7 Success criteria

In Section 1.2, We presented a list of aims and objectives. We will look at each of the objectives in turn to evaluate the success of this project.

5.7.1 Primary objectives

1. *Develop an automated process that turns the OpenStreetMap data for an area into a graph annotated with data relevant to cycle accessibility and apply this to St Andrews* - **fully complete**

   We implemented a three-layer system: a data fetcher to request the OpenStreetMap data, a model to process the data into a graph based on cycling information, and a graph layer to display and analyse the graph. We also provided a script `main.py` as the entry point to the program, so that the above analysis can happen automatically in one command. This process was applied to the area of St Andrews, as discussed in Section 5.2.

2. *Develop a set of configurable heuristics to determine whether a route is cycle friendly* - **fully complete**

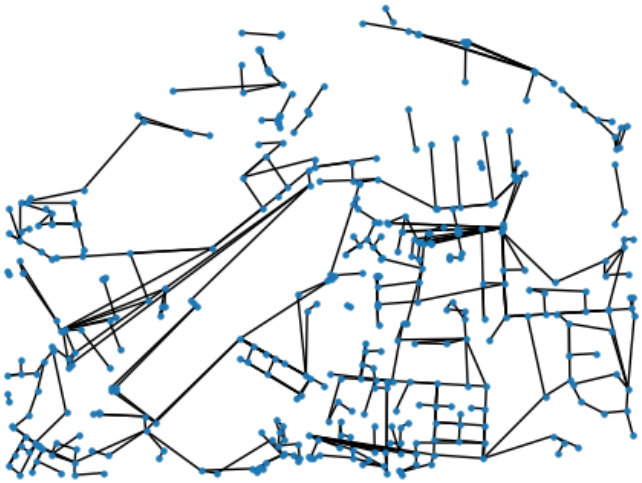
   A heuristic function was given in Section 3.5.2, where the weightings and scores can be configured in `configuration.json`. In addition, the threshold score for which a way is considered to be cycle-friendly is configurable.

3. *Apply the heuristics to highlight disconnected components in the cycle-friendly subgraph*
4. *Consider other properties of the graph, such as k-connectivity and induced subgraphs, to produce other findings relevant to cycling*
5. *Suggest the most efficient paths to add to increase the connectedness of the subgraph* - **fully complete**

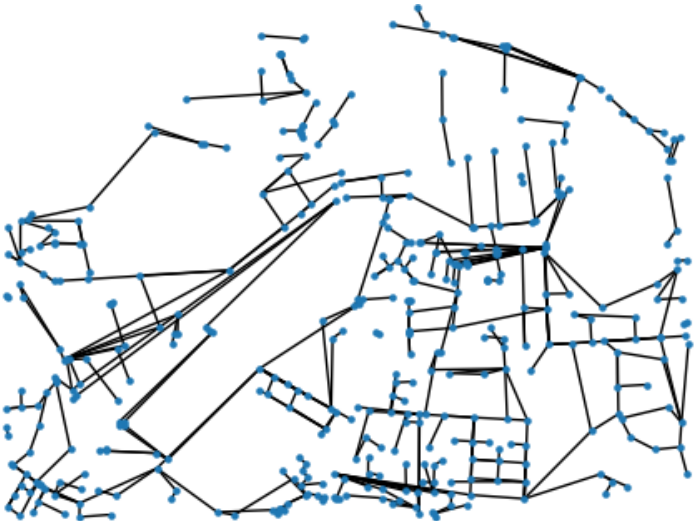
   We have developed three strategies to suggest new paths, as discussed in Section 3.4.1.

5.7.2 Secondary objectives

1. *Apply this analysis process to another similar area and assess how well the automated process works for areas it was not designed for*
2. *Apply the analysis process to a larger area to evaluate the scalability of the technique*
3. *Consider the cost of adding new paths when making suggestions*



(a) Before the configuration was modified



(b) After the configuration was modified

Figure 5.4: The cycle-friendly graph generated by our system

Chapter 6

Conclusions

Bibliography

[1] Ihechikara Vincent Abba. Dijkstra's algorithm – explained with a pseudocode example. `https://www.freecodecamp.org/news/dijkstras-algorithm-explained-with-a-pseudocode-example/`, 2022. Accessed: 2023-06-07.

[2] F. Buckley and F. Harary. *Distance In Graphs*. And Computer Engineering: Control. Basic Books, 1990.

[3] Cairo. `https://www.cairographics.org/`. Accessed: 2023-07-27.

[4] Lee Chapman. Transport and climate change: a review. *Journal of Transport Geography*, 15(5):354–367, 2007.

[5] Gary Chartrand and Ping Zhang. *A First Course in Graph Theory*. Dover Publications Inc., 2012.

[6] Coverage.py. `https://coverage.readthedocs.io/en/7.2.7/`. Accessed: 2023-08-05.

[7] Gabor Csardi and Tamas Nepusz. The igraph software package for complex network research. *InterJournal*, Complex Systems:1695, 11 2005.

[8] Sybil Derrible and Christopher Kennedy. Applications of graph theory and network science to transit network design. *Transport Reviews*, 31(4):495–519, 2011.

[9] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics)*. Springer, August 2005.

[10] Edsger W Dijkstra. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

[11] Alexander Erath, Michael Löchl, and Kay W. Axhausen. Graph-theoretical analysis of the swiss road and railway networks over time. *Networks and Spatial Economics*, 9(3):379–400, Sep 2009.

[12] Colin Ferster, Jaimy Fischer, Kevin Manaugh, Trisalyn Nelson, and Meghan Winters. Using openstreetmap to inventory bicycle infrastructure: A comparison with open data from cities. *International Journal of Sustainable Transportation*, 14(1):64–73, 2020.

[13] Lakoa Fitina, John Imbal, Vanessa Uiari, Nathaline Murki, and Elizabeth Goodyear. An application of minimum spanning trees to travel planning. *Contemporary PNG Studies: DWU Research Journal*, 12, 2010.

[14] Open Knowledge Foundation. Open Data Commons Open Database License (ODbL). `https://opendatacommons.org/licenses/odbl/`. Accessed: 2023-06-13.

[15] geopy contributors. Welcome to GeoPy's documentation! `https://geopy.readthedocs.io/en/stable/#module-geopy.distance`. Accessed: 2023-08-01.

[16] Ashish Gupta. New ways we tackle fake contributions on Google Maps. `https://blog.google/products/maps/google-maps-fake-contributions-ai-machine-learning/`, 2023. Accessed: 2023-06-07.

[17] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using networkx. In Gaël Varoquaux, Travis Vaught, and Jarrod Millman, editors, *Proceedings of the 7th Python in Science Conference*, pages 11 – 15, Pasadena, CA USA, 2008.

[18] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing in Science & Engineering*, 9(3):90–95, 2007.

[19] Muhammad Iqbal, Andysah Putera Utama Siahaan, Nathania Elizabeth, and Dedi Purwanto. Prim's algorithm for optimizing fiber optic trajectory planning. *International Journal of Scientific Research in Science and Technology*, 3:504–509, 08 2017.

[20] unittest.mock — mock object library. `https://docs.python.org/3/library/unittest.mock.html`. Accessed: 2023-08-05.

[21] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . `https://www.openstreetmap.org`, 2017.

[22] Welcome to Python Overpass API's documentation! `https://python-overpy.readthedocs.io/en/latest/index.html`. Accessed: 2023-07-31.

[23] Seth Pettie. *Minimum Spanning Trees*, pages 541–544. Springer US, Boston, MA, 2008.

[24] Ethan Russell. 9 things to know about Google's maps data: Beyond the map. `https://cloud.google.com/blog/products/maps-platform/9-things-know-about-googles-maps-data-beyond-map`, 2019. Accessed: 2023-06-07.

[25] L Sloman, N Cavill, A Cope, L Muller, and A Kennedy. Analysis and synthesis of evidence on the effects of investment in six cycling demonstration towns. Report for Department for Transport and Cycling England, 2009.

[26] unittest — Unit testing framework. `https://docs.python.org/3/library/unittest.html#module-unittest`. Accessed: 2023-08-05.

[27] John E. Vargas-Munoz, Shivangi Srivastava, Devis Tuia, and Alexandre X. Falcão. OpenStreetMap: Challenges and opportunities in machine learning and remote sensing. *IEEE Geoscience and Remote Sensing Magazine*, 9(1):184–199, 2021.

[28] Wen-Ching Wang and Ming-Che Hsieh. Applying Prim's algorithm to identify isolated areas for natural disaster prevention and protection. *Engineering*, 10:417–431, 01 2018.

[29] OpenStreetMap Wiki. Cyclosm — openstreetmap wiki,, 2022. Accessed: 2023-07-07.

[30] OpenStreetMap Wiki. Overpass api — openstreetmap wiki,, 2023. Accessed: 2023-07-10.

[31] OpenStreetMap Wiki. Overpass api/overpass ql — openstreetmap wiki,, 2023. Accessed: 2023-07-27.

[32] Wikipedia, the free encyclopedia. Dijkstra's algorithm. Accessed: 2023-08-01.

[33] Wikipedia, the free encyclopedia. Minimum spanning tree, 2005. Accessed: 2023-06-13.

[34] Wikipedia, the free encyclopedia. A nimber numbering of the fano plane, 2011. Accessed: 2023-07-31.

[35] Wikipedia, the free encyclopedia. 4-connected graph, 2018. Accessed: 2023-06-13.

[36] Atsuko Yamagami. Google Maps learns 39 new languages. https://blog.google/products/maps/google-maps-learns-39-new-languages/, 2018. Accessed: 2023-06-07.

[37] Seyed Ashkan Zarghami, Indra Gunawan, and Frank Schultmann. Exact reliability evaluation of infrastructure networks using graph theory. *Quality and Reliability Engineering International*, 36(2):498–510, 2020.

Appendix A

Testing summary

Unit test

```
test_bounding_box (tests.test_config.ConfigTestCase) ... ok
test_config (tests.test_config.ConfigTestCase) ... ok
test_config_no_such_area (tests.test_config.ConfigTestCase) ... ok
test_weighted_tags (tests.test_config.ConfigTestCase) ... ok
test_get_centre (tests.test_data_fetcher.DataFetcherTestCase) ...
    ok
test_get_node_pos_by_ids (tests.test_data_fetcher.
    DataFetcherTestCase) ... ok
test_get_nodes_on_ways (tests.test_data_fetcher.DataFetcherTestCase
    ) ... ok
test_get_ways (tests.test_data_fetcher.DataFetcherTestCase) ... ok
test_makes_query (tests.test_data_fetcher.DataFetcherTestCase) ...
    ok
test_node_in_area_boundary (tests.test_data_fetcher.
    DataFetcherTestCase) ... ok
test_node_in_area_false (tests.test_data_fetcher.
    DataFetcherTestCase) ... ok
test_node_in_area_true (tests.test_data_fetcher.DataFetcherTestCase
    ) ... ok
test_preprocessing_fully_connected (tests.test_graph.
    GraphProcessingTestCase) ... ok
test_preprocessing_not_fully_connected (tests.test_graph.
    GraphProcessingTestCase) ... ok
test_trim_path_many_head_cycles (tests.test_graph.
    GraphProcessingTestCase) ... ok
test_trim_path_many_tail_cycles (tests.test_graph.
```

```
    GraphProcessingTestCase) ... ok
test_trim_path_no_cycle (tests.test_graph.GraphProcessingTestCase)
    ... ok
test_trim_path_one_cycle (tests.test_graph.GraphProcessingTestCase)
     ... ok
test_count_node_links (tests.test_model.ModelTestCase) ... ok
test_eval_way_empty_tag (tests.test_model.ModelTestCase) ... ok
test_eval_way_maxspeed_has_unit (tests.test_model.ModelTestCase)
    ... ok
test_eval_way_maxspeed_in_range (tests.test_model.ModelTestCase)
    ... ok
test_eval_way_maxspeed_not_in_range (tests.test_model.ModelTestCase
    ) ... ok
test_eval_way_one_tag (tests.test_model.ModelTestCase) ... ok
test_eval_way_tag_does_not_exist (tests.test_model.ModelTestCase)
    ... ok
test_eval_way_value_does_not_exist (tests.test_model.ModelTestCase)
     ... ok
test_get_adj_list (tests.test_model.ModelTestCase) ... ok
test_get_node_pos (tests.test_model.ModelTestCase) ... ok
test_get_node_pos_empty (tests.test_model.ModelTestCase) ... ok
test_ways_to_adj_list (tests.test_model.ModelTestCase) ... ok
test_ways_to_adj_list_cycle_edge (tests.test_model.ModelTestCase)
    ... ok
test_ways_to_adj_list_empty_edge (tests.test_model.ModelTestCase)
    ... ok

----------------------------------------------------------------------
Ran 32 tests in 0.176s

OK
```

Test coverage

```
Name                       Stmts   Miss  Cover
----------------------------------------------
scripts/__init__.py            0      0   100%
scripts/config.py             51      0   100%
scripts/data_fetcher.py       30      0   100%
scripts/exception.py           2      0   100%
scripts/graph.py             109     48    56%
scripts/model.py              63      0   100%
```

```
tests/__init__.py              0      0   100%
tests/test_config.py          26      0   100%
tests/test_data_fetcher.py    61      0   100%
tests/test_graph.py           54      0   100%
tests/test_model.py          131      0   100%
-----------------------------------------------
TOTAL                        527     48    91%
```

Appendix B

User manual

Appendix C

Ethics