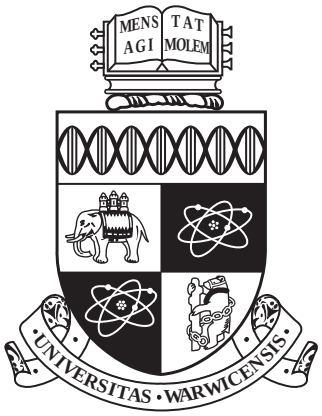


Route Adaptation in London through Graph Optimisation and Reinforcement Learning

by

Richard Collins

Supervisor

Dr. Theo Damoulas

Department of Computer Science

University of Warwick

September 2018

Abstract

Route planning applications that consider a single distance constraint may generate routes are inappropriate for a number of users. This project aims to compute routes through London that that are adaptive to features of the roads. Formulating the problem as a graph optimisation task, routes that minimise air pollution exposure, and that are similar to previously traversed routes are generated. Utilising historic trajectory data through the road network, bespoke preferences for road features are calculated via Inverse Reinforcement Learning with heuristically set feature weights. A forward Q-Learning control algorithm is also developed and applied to individual reward functions to personalised produce optimal routes.

Acknowledgements

I would first like to thank my supervisor, Dr. Theo Damoulas, for offering me this research opportunity and for his support over the past nine months. I would also like to thank Patrick O'Hara for providing me with the fruits of his efforts on his bachelor's thesis, the absence of which would have likely made this project impossible.

Abbreviations

LAEI	London Atmospheric Emissions Inventory
QGIS	Quantum Geographic Information Systems
WHO	World Health Organisation
EU	European Union
RL	Reinforcement Learning
IRL	Inverse Reinforcement Learning
MaxEntIRL	Maximum Entropy Inverse Reinforcement Learning
MDP	Markov Decision Process
PROCAB	Probabilistically Reasoning from Observed Context-Aware Behaviour
GPS	Global Positioning System
DP	Dynamic Programming
MC	Monte-Carlo
TD	Temporal Difference
ED	Edge Data
RDM	Ramer-Douglas-Peucker

Notations

V	A set of vertices/nodes
v	A vertex/node
E	A set of edges
e	An edge
$d(u, v)$	Length of edge between vertices u and v
$p(u, v)$	Pollution penalty between vertices u and v
γ_{avg}	Average pollution of edge
w_e	Edge weight
f_e	Scaling factor of edge weight
n_e	Traversal count of edge
$\mathcal{Z}$	Set of all trajectories
β	Rate of exponential decay
$\mathcal{S}$	Set of environment states
s	State
$\mathcal{A}$	Set of actions
a	Action
$\mathcal{P}$	Transition probabilities
γ	Discount factor
$\mathcal{R}$	Reward function
r	Reward
π	Policy
V^π	Value function under a policy
Q^π	Action-value function under a policy
η	Learning rate
α	Feature weight
ϕ	Basis vector
ζ	Trajectory
μ	Feature expectation
κ	Adjustment term
δ	Distance parameter

List of Figures

Contents

Chapter 1

Introduction

1.1 Motivation

The first fledgeling ideas for this research project were the result of a group project for the IM913 - Spatial Methods and Practice in Urban Science module offered by the Centre for Interdisciplinary Methodologies [8]. In this project, our group investigated the relationship between cycle hire schemes in London and air pollution. The data for air pollution was sourced from the London Atmospheric Emissions Inventory [32]. The data consisted of shape files with annual average concentrations for four pollutants: nitrous oxides (NO_x - including nitrogen monoxide and nitrogen dioxide), carbon dioxide (CO_2), $PM_{2.5}$ (particle matter with a radius of 2.5μm or less) and PM_{10} (particle matter with a radius of 10μm or less). An example of such a shape file is shown in Fig. 1.1.

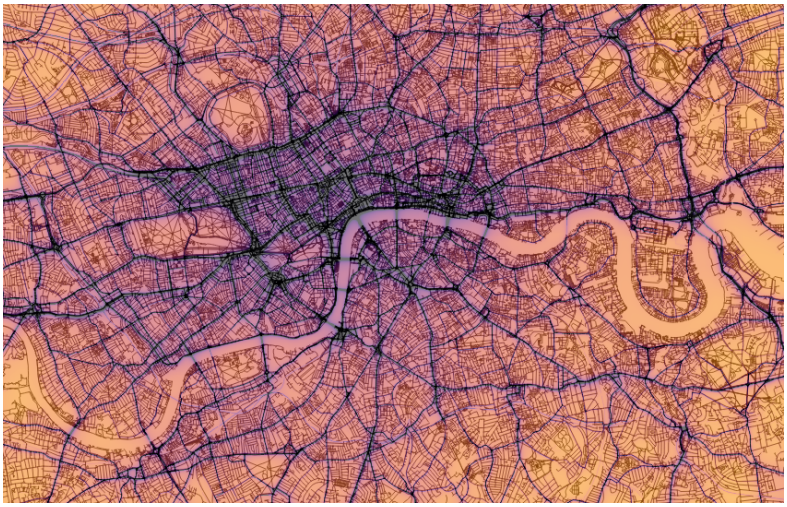


Figure 1.1: A raster map of NO_x concentration (μg m^{-3}) generated in QGIS based on data from the LAEI model described above. The resolution is 1 pixel = $20m^2$. Darker colours represent areas of higher concentration.

Investigating this data, we were able to establish a positive correlation between the activity of cycle stations and air pollution. This was unsurprising since the most active cycle stations are more likely to be located in the centre of London where air pollution is also higher. For the majority of users, the issue of air pollution is not a concern when picking up and dropping off their bike, but rather while cycling on busy streets with other road users. It is during the journey that the user experiences the highest levels of pollution. It was therefore decided that a more useful metric would be air pollution encountered *during* a cyclist's journey. The broad steps taken were:

- Identify typical routes which cyclists take
- Calculate the mean pollution level encountered along these paths
- Compare to other similar routes with reference to the pollution encountered

After generating example paths through the road network, we were able to show that route selection can reduce average[1] exposure of NO_x by 7% - 22% [8]. An example of three paths from Waterloo train station to St. Peter's Cathedral is shown in Fig. 1.2.

Figure 1.2: Three example cycle paths. Note how the green and blue lines pass through areas of lower air pollution.

After receiving positive feedback from fellow students and academics in the Centre for Interdisciplinary Methodologies, initial ideas were formulated to implement a routing application that not only considers fastest/shortest routes, but also routes that are the least polluted. Work was ongoing on this very subject in the form of the Clean Air in London project [16]. In particular, this dissertation builds upon the master's thesis by Green *et al.* [13] which created

[1]This was taken from only a very small number of potential paths, and should not represent the average reduction one would expect throughout London. The aim of the study was simply to show that path selection *does* reduce air pollution exposure.

mathematical models to predict air quality, and works alongside the final-year project by Patrick O'Hara [27], who tackled a similar problem to the one described in this section. It also picks up one of the initial aims of Patrick's thesis, which was to incorporate historic data to find similar paths. This aim was not met due to the changing focus of his particular project.

1.2 Problem Statement

There are three broad problems that this project seeks to solve.

1.2.1 Computing Least Polluted Routes

Starting with a graph representing the road network of London, where each edge weight signifies the local air quality along that edge, and given a start and end point, compute the path that minimises an individual's exposure to air pollution. This will be achieved by using synthetic air quality data generated by the model described in Green *et al.* [13], and the road network provided by Ordnance Survey [35]. The computation of the paths will be carried out by applying Dijkstra's algorithm [9], a graph optimisation algorithm for finding shortest paths in a network.

1.2.2 Adapted Routes with Historical Data

With the above system in place, and given historical trajectory data from an individual, adapt the edge weights of the graph to reflect the popularity of each road. A scaling factor will be incorporated in the calculation of air pollution. The scaling factor will be a function of the number of times the edge is traversed over an individual's trajectory history. Special selection of the function will have the effect of artificially lowering or increasing edge weights such that highly traversed edges will have a lower edge weight value (and vice versa). The routes generated by optimising against this new weighting scheme will be a compromise between minimising air pollution exposure and those the individual would habitually take.

1.2.3 Reinforcement Learning Formulation

A number of edge features will be added in conjunction with the features of distance and pollution. These features will be chosen to reflect some measure of preference for an individual (for example, type of road, number of lanes etc.). Then, by applying inverse reinforcement learning techniques and heuristics with historical trajectory data as training examples, the importance, or desirability, of each feature will be calculated. Depending on the feature mix of each edge, a value for the reward can be calculated. Generating routes will be achieved by applying a reinforcement learning algorithm to the newly set reward function with a source and target state.

This project will establish a proof-of-concept, with the view that a simple web or mobile application would be developed in the future. Throughout this project, we will keep the end user in mind; an individual who seeks to compute

adapted routes through London. Although initially a system such as this was conceived with a cyclist in mind, in principle, the system could work for anyone as long as the user has provided past examples of trajectories through the network.

1.3 Document Overview

This report begins by exploring related literature in graph optimisation and reinforcement learning. Then, the theoretical groundwork will be set by presenting key concepts in their general form first, before more specific cases related to the exact problems being addressed are presented. The implementation section details how theory was applied, and how any system limitations were overcome. Results and analysis follow highlighting the most important findings, as well as evaluations of the system's performance. Final remarks and future work are left for the conclusion chapter.

Chapter 2

Background

2.1 Clean Air in London

As mentioned in section 1.1, this report builds on work conducted by Green *et al.* [13] and Patrick O'Hara [27]. The former deals with building machine learning models to predict air quality in London, and the latter deals with generating running routes in London that minimises a person's exposure to air pollution.

2.1.1 Modelling Air Pollution

Approaches to modelling air pollution fall into two categories: deterministic approaches and statistical approaches. On the determinist front, models are based on mathematical descriptions of atmospheric processes [43]. However, these models suffer from being computationally expensive due to the high resolutions needed for then to be effective predictors. Improving the run-time of models necessarily incurs a loss of resolution, and therefore increased uncertainty. Statistical approaches model the relationships between various data sets and measurements. Common statistical air pollution models are: land use regression, artificial neural networks, and interpolation methods such as kriging.

Land use regression models offer a simple, easy-to-interpret method for predicting air pollution at locations without access to direct monitoring. As the name suggests, the predictor variables for such regressions are connected with the manner in which the land is used. For example, traffic, population, physical geography etc. [15][5]. Kinging is a linear interpolation method, combining spatially local points to predict values at a given point. The method differs from traditional linear interpolation by not only using distance for weighting schemes, but also leveraging the data itself. L. Mecer *et al.* compared the performance of kringing and land use regression when predicting the air pollution in Los Angeles [22]. Comparing the R^2 value of each after testing with 10-fold cross validation, they found that kringing consistently outperformed the land use regression models.

Green *et al.* chose to base their master's thesis on the U-Air model; a model of air quality in the city of Beijing [46]. The model is based on two classifiers: an artificial neural network for spatial data, and a linear-chain conditional random field for temporal data. A city is first divided into grid cells measuring 1km $\times$ 1km, of which some air pollution values are known and others unknown.

Training consists of using the features from the grid cells and inferring the pollution level where it is know. The trained model is then used to predict values for unknown grid cells.

2.1.2 Graph Optimisation

The principle optimisation algorithm used in this project is Dijkstra's algorithm. First conceived by Edsger W. Dijkstra in 1956 and published three years later in 1959 [9], all research efforts since then for single source shortest path problems have been based on a variation of this algorithm [38]. As Dijkstra's algorithm is applied to real-life networks, it becomes necessary to adapt the algorithm to perform optimally with respect to the graph in question. In the case of road networks, F. Benjamin Zhan and Charles E. Noon [44] provide an objective evaluation of 15 shortest path algorithms when applied to road networks. From this evaluation, they identified a set of recommended algorithms for use in shortest path problems on road networks.

Patrick O'Hara, in his bachelor's thesis *"Running from Air Pollution"* combined synthetic air pollution data with optimisation algorithms to compute running routes in London that minimise a runner's exposure to air pollution. Formulated as a graph optimisation problem, the main algorithm computed closed walks from a starting node with a fixed length. Continuous and discrete cases of the problem were constructed by modifying the multiplicity constraint. The continuous case produces walks that are "boring" for most runners, following the least polluted path to a region of low pollution, repeating the head edge a number of times to satisfy the distance constraint, then returning to the start node. The discrete case constrains edge multiplicity to 1 or 0, forcing the problem to be solved through approximate methods due to its $\mathcal{NP}$-hard property (a property that is proved in the final manuscript).

2.2 Air Quality and Exercise

Poor air quality is an ever increasing problem in large towns and cities. It is the cause of debilitating health effects such as respiratory diseases, impaired lung function and in severe cases premature death [10]. Air pollution is particularly marked in cities and other densely populated areas. It is estimated that over 80% of residents living in urban environments are exposed to air pollution that exceeds the levels set out by the WHO. This has become a major public health issue with 54% of the globe living in urban areas [3]. With ever larger and more densely populated cites comes with it higher levels of air pollution. This is borne from many different sources including traffic, construction, and the energy industry. For frequent cyclists and runners, this increase in air pollution is confronted during daily commutes and exercise, where road vehicles and bikes vie for space on crowded city highways. London, due to its great size and population, also suffers from poor air quality. It is regularly one of the most polluted areas of the country, failing to meet WHO and EU guidelines for pollution levels [24].

M. Tainio *et al.* [37] first noted that health risks of air pollution increase *linearly* with increased exposure for low to moderate levels of air pollution, whereas the benefits of physical activity increase *curvy-linearly* with increasing duration.

It follows that at a certain level of background air pollution and of activity level, risks could outweigh benefits; exercising outside is not recommended. The *tipping point* (duration of activity beyond which increased activity no longer has greater health benefits) and the *break-even point* (duration of activity beyond which increased activity has adverse health risks) were calculated for different background levels of $PM_{2.5}$. It was found that in at least 15 cities the break-even point is reached after 30 minutes of cycling, and 120 minutes for at least 62 cities [14]. Although London was not on this list, the study only considered the annual average of $PM_{2.5}$ levels and did not consider episodic spikes in concentration. If this were the case, London would join this list for as long as the spike lasts (during morning and evening rush-hours, for example).

The principle underlying this project is that the choice of path one takes reduces exposure to air pollution. Since much of a city's air pollution is generated by traffic, it seems like a reasonable conjecture that travelling on roads with lighter traffic would reduce one's exposure to air pollution. The preliminary study [8] described in section 1.1 showed that path selection does indeed reduce air pollution exposure. P. Apparicio *et al.* were able to draw the same conclusion through direct measurement of concentration of pollutants along cycle paths in Montreal [2]. A total of 85 bicycle trips were analysed, representing 422km of travel and nearly 25 hours of data collection. This study showed that the type of cycle path impacts the concentration of air pollution encountered by the rider, with on-road cycle lanes and shared cycle lanes being associated with higher levels air pollution, and off-street cycle paths being associated with lower levels of air pollution. P. MacNaughton *et al.* came to a similar conclusion in a study of cycle paths in Boston [29]. They concluded that cycle paths should be developed on roads with little traffic, or better still, off-street paths that are far from major roads.

2.3 Applications of Reinforcement Learning

Reinforcement learning (RL) centres around an imagined agent moving through a state space. It does this by selecting actions that allow for a transition from one state to the next. Rewards are yielded from the environment as the agent attempts to reach a predetermined goal state while maximising the total reward along the way. The goal for RL problems is to find the policy (mapping from state to action) that describes the optimum movement through the state space. RL methods can be utilised to solve a number of theoretical and real-world problems by adapting the state space and reward signal to reflect the particular problem.

2.3.1 Canonical Examples

Also called "benchmark" examples, they are often used to evaluate novel algorithms under precisely defined environments. The dimensionality of problems are typically low, with no more than a few dozen states, but usually lower. This allows for their behaviour to be studied easily, as well as being convenient to conceptualise. Benchmark problems almost exclusively deal with a single agent.

The most famous of these examples is the *grid world*. In its simplest incarnation, the grid world consists of a finite, planar surface divided into square cells,

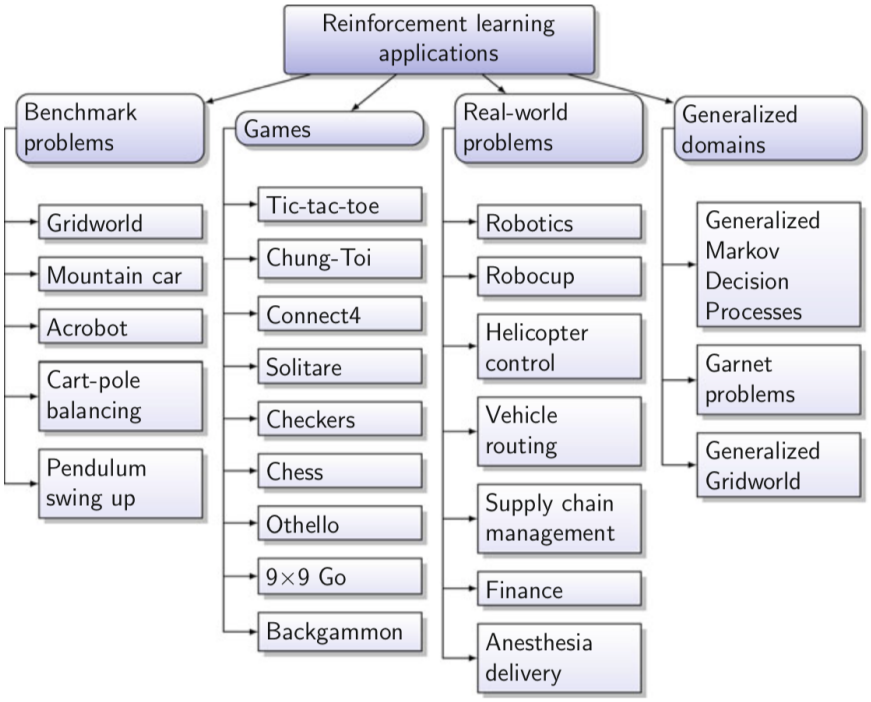


Figure 2.1: Overview of the main disciplines of RL [11].

much like a chess board (but not necessarily with sides of equal length). The agent is placed at some starting square, and is able to choose from four actions (north, south, east, and west) to move from one square (state) to the next. The aim is to reach some terminal state while collecting the maximum reward, as set by the experimenter. A number of modifications can be applied to this simple domain. R. Sutton and A. Barto [36] introduce an imagined "wind" that acts on the agent as it moves in the grid world, affecting the environment dynamics and, ultimately, the behaviour of the agent. Larger gird worlds have been used to develop and evaluate RL algorithms that can handle sub goals [4].

Though usually abstract constructions, benchmark problems have been devised to mimic real-world physical systems, often moving from a discrete state space to a continuous representation. An example of this is the mountain car problem. First introduced by A. Moore in 1990 in his PhD thesis [41], and later developed in the standard text *"Reinforcement Learning, An Introduction"* by R. Sutton and A. Barto [36], the problem concerns a car at the base of a valley that must reach the peak of the adjacent mountain by using the actions *accelerate*, *reverse*, and *do nothing*. A state is defined by the car's position and velocity, and rewards are $r = -1$ for every time step, meaning the optimal solution is that which allows the car to reach its goal in the fastest time possible.

This problem shares some basic similarities with the one being addressed in this project. Namely, solution methods revolve around approximating the value function with a set of basis vectors, and saving traces of previously found solutions. This second point is particularly pertinent to this project because

the agent is not able to make optimum choices until it has completed the task at least once. Many successful completions drive the agent to ever better, more optimal solutions.

2.3.2 Games

An area of active research is that of RL in games. Most noteworthy is the advancement in computer Go systems. As recently as March 2016, Google's AlphaGo system was able to beat one of the greatest human players of the game [25]. The development of AlphaGo utilises a Monte-Carlo tree search algorithm to evaluate and update the value function [31], a technique that first gained traction in the area of computer Go in 2011 after an article on the same subject by S. Gelly and D. Silver [12]. The authors note that a Monte-Carlo search method is suited to Go because it evaluates the current position dynamically, rather than storing knowledge about all positions in a static evaluation function. This makes it suitable choice for Go, where the number of possible positions is particularly large[1]. It may also therefore be an appealing method for evaluating optimum paths in the context of this project.

2.3.3 Real-World Problems

A more abstract application of RL has been explored by A. Karal and I. Dogan [19]. They modelled an inventory of perishable goods using a replenishment policy based on the age of the product. The environment was represented as the stochastic demand of customers. Their results demonstrated that the ordering policies that consider both the age information and the inventory quantity give better results when compared with policies that consider only the inventory quantity, as is the case for the majority of stock management systems.

2.4 Applications of Inverses Reinforcement Learning

2.4.1 Principles

First introduced by A. Ng and S. Russell in their seminal work titled *"Algorithms for Inverse Reinforcement Learning"* [26], Inverse Reinforcement Learning (IRL), as the name suggests, is the opposite process to RL. The aim is to recover the reward function for which an agent is observed to behave optimally. There are a number of motivations that lead to the study of IRL. The first is that for many real-life RL problems there is no natural reward function; it is engineered to represent the task at hand in the best way possible, continually being manipulated so as to observe desired behaviour from the agent. In domains such as games, natural reward functions can be clearly constructed from the rules of the game. But in other domains, construction of a bespoke reward function is often difficult. Imagine for example the difficulty of defining a reward signal associated with an autonomous driving system. In this case, it would be much simpler to instead observe expert behaviour (a human driver) and then attempt to learn the underlying reward function which is being optimised.

[1] According to [39], approximately 2×10^{170}.

2.4.2 Apprenticeship Learning

Apprenticeship learning is a related topic in IRL. It is the task of teaching an agent, also known as the *apprentice*, to learn from demonstration of the "correct" behaviour by an expert. In contrast to general unsupervised learning (for example classification problems) tasks are often multi-decision problems. The behaviour to be learned comprises of a sequence of decisions towards a goal, or in other words, a mapping from states to actions. P. Abbeel and A. Ng were the first to suggests solution methods under the name of apprentice learning [1]. They approximated policies using Monte-Carlo methods.

2.4.3 Route Modelling

In the domain of route modelling, much of the solution methods involve maximum entropy IRL (MaxEntIRL). The motivation for MaxEntIRL is that it deals with the ambiguity of potentially sub-optimal expert policies, and the fact that many policies can lead to the same feature counts. In addition, demonstrated behaviour has a propensity to be noisy and is often an imperfect representation of the true underlying behaviour of the expert. MaxEntIRL provides a principled method of dealing with this uncertainty.

MaxEntIRL has been successfully implemented by B. Ziebart *et al.* to predict routes given a driver's historic preferences [47] [48]. For the first paper, the road network surrounding Pittsburgh, Pennsylvania, was represented as a deterministic MDP with over 300,000 states and over 900,000 actions. (road segments and intersections, respectively). Historic data were collected from 25 taxi drivers over a 12 week duration. Removing overly noisy and short trajectories, they were left with a training and testing set of 7403 trajectories. Their probabilistic approach was able to model route preferences, infer destinations, and complete routes based on partial trajectories with 78.79%. The second paper presents an efficient method for Probabilistically Reasoning from Observed Context-Aware Behaviour (PROCAB). The system includes features that pertain to the context in which the trajectories were recorded, for example time of day, weather conditions etc. Using a similar set of historic taxi journeys, the PROCAB model was able to give the probability distribution over actions at intersections, predict the most likely route to a user-specified destination, and calculate the most probable destination given partially travelled routes.

Chapter 3

Theoretical Framework

3.1 Graph Optimisation

3.1.1 London as a Graph

As mentioned in section 1.2.1, this problem can be formulated as a graph optimisation task. We represent the road network of London as an undirected, weighted graph $G(V, E)$. The set of edges, E, represents all roads in the network, and the set of vertices[1] V represents the intersections between the roads. We may attribute data to each $v \in V$ and $e \in E$ representing edge and vertex ID, road distance, vertex degree etc. More complete lists of attributes are shown in Fig. 4.1 and 4.2 in section 4.1. We first present the edge data representing distance as $d(u, v)$; the physical length of the edge between vertices u and v, and the *pollution penalty* $p(u, v)$; the exposure to air pollution one would encounter traversing the edge between vertices u and v.

3.1.2 Dijkstra's with respect to Pollution

Some assumptions must be made at this point. Firstly, the pollution penalty incurred upon traversing an edge is uniform. Depending on the resolution of the air pollution model, this will be the average pollution exposure along an edge (high resolution), or, if an edge exists entirely within a pollution grid cell (low resolution), $p(u, v)$ will take the value of that grid cell. In the synthetic data used for this study, the value of air pollution (NO_2 in this case[2]) is denoted with γ and measured in μg m^{-3}. The total pollution penalty of an edge is given by $p(u, v) = \gamma_{\text{avg}} \cdot d(u, v)$, where γ_{avg} is the average of the γ values of the grid cells that the edge intersects. The second assumption is that the pollution penalty is direction-independent, that is to say $p(u, v) = p(v, u)$.

We seek the lowest-polluted path Q through the network from source to goal; the path that will minimise a cyclist's/runner's exposure to air pollution. This path is the path that has the lowest possible sum of pollution penalties between two user-defined vertices. Mathematically, we can express this in the following equation:

[1] The terms *vertex* and *node* will be used interchangeably.

[2] While we are only considering one of the four commonly measured air pollutants, the other three would scale in the same way. So, for simplicity's sake, we do not include them.

$$\text{cost}(Q) = \min\left(\sum_{i=0}^{n-1} p(u_i, u_{i+1})\right) \tag{3.1}$$

where $\text{cost}(Q)$ is the total air pollution encountered. In practice, we are less concerned with the total pollution penalty, and more with the optimum path Q as this would be the quantity of greatest interest to users of an application such as this.

Optimisation problems such as equation 3.1, seeking a path while minimises edge weights, can be solved using Dijkstra's algorithm. For a given starting vertex, the algorithm finds the path that minimises the sum of edge weights between that vertex and every other in the graph. The algorithm, while finding the minimum-cost path between source and every other vertex, can also be truncated when a target vertex is reached. Dijkstra's algorithm is greedy, meaning that the exploration of vertices in the graph is carried out by considering only the next closest to the source vertex. Knowing this, we can understand how the algorithm would necessarily find the shortest path. However, its greatest drawback is that, for some graph topologies, the run time can be quite slow.

The time complexity of Dijkstra's algorithm, in its original conception, completes in $O(n^2)$ where $n = |V|$ is the number of vertices in the graph. This can be improved by using a Fibonacci heap implementation, reducing time complexity to $O(m + n \log n)$ where $m = |E|$ is the number of edges in the graph. These are the *worst case* run times, where the sets V and E must be explored entirely before the shortest path is found. In practice, as explained above, Dijkstra's algorithm will be truncated once the shortest path between source and terminal vertices is found. A form or Dijkstra's algorithm is shown, for completeness, in appendix A.

3.1.3 Dijkstra's with respect to Edge Data

The above formulation can be extended to include any number of data layers, each contributing to the final weight of each edge in the network. Let w_e be the edge weight of edge e. Presently, we have the following expression for edge weights: $w_e = p(u, v) = \gamma_{\text{avg}} \cdot d(u, v)$. Further data layers can be incorporated through linear combination:

$$w_e = f_1 + f_2 + ... = \sum_{i=1}^{N} f_i \tag{3.2}$$

where f_i is some scalar representation of the ith feature of the edge. This formulation will be re-examined in section 3.5 when setting the foundations for IRL. When incorporating a single extra data layer it is more natural to treat it as a scaling factor rather than an additional term. This is partly due to the impracticality of scaling the added data to match the magnitude of the already present edge weight, but also due to the ease of conceptualising how the new data affects the edge weight. The above equation would then be transformed to:

$$w_e \leftarrow w_e \cdot f_{\text{new data}} \tag{3.3}$$

3.2 Incorporating Road Popularity

3.2.1 Popularity as a Scaling Factor

We wish to scale the current edge weights to include the effect of the popularity of a road segment. To do this, we adopt the method of multiplying the pollution penalty by a scaling factor. The scaling factor should produce the desired effect that, when a road is more popular, the overall edge weight is decreased. This would mean that more popular roads have a scaling factor closer to 0, and would therefore be more likely to be included in an optimum path between two nodes. In contrast, roads that are unpopular would have a scaling factor closer to 1, meaning they would be less likely to be included in an optimum path. We restrict the values the scaling factor can take to be $0 \leq f \leq 1$. This is for two reasons. The first reason is because negative values would produce an overall negative value for the edge weight, which cannot be handled by Dijkstra's algorithm[3]. The second reason is that while it is theoretically possible to "tax" unpopular edges by assigning them scaling factors greater than 1, in practice, as we shall see, measuring exactly how unpopular an edge is is difficult. Therefore, we assign the least popular roads (i.e. roads that have not been traversed in the trace data, see section 3.2.2) a scaling factor $f_e = 1$.

The measure of the popularity will be taken as the number of times a road is traversed by a single user. This edge count will be denoted by n_e. For reasons that will be explored more fully in section 4.4.1, the edge count will indicate the boolean value of whether that edge had been traversed in a trajectory or not. Multiple edge traversals in a single trajectory are still counted as unity. The number of trajectories under consideration will be denoted as $|\mathcal{Z}|$ meaning the cardinality of the set of all trajectories. Note that $\forall n_e : n_e/|\mathcal{Z}| \leq 1$.

With a definition of popularity, we now want to convert that into a scaling factor. We must ensure that f_e only takes values between 0 and 1, and that a higher popularity translates to a lower f_e. Two functions have been proposed to produce this desired behaviour:

$$f_e = \begin{cases} 1 - \exp\left(-\frac{\beta|\mathcal{Z}|}{n_e}\right) & \text{for } n_e > 0 \\ 1 & \text{for } n_e = 0 \end{cases} \tag{3.4}$$

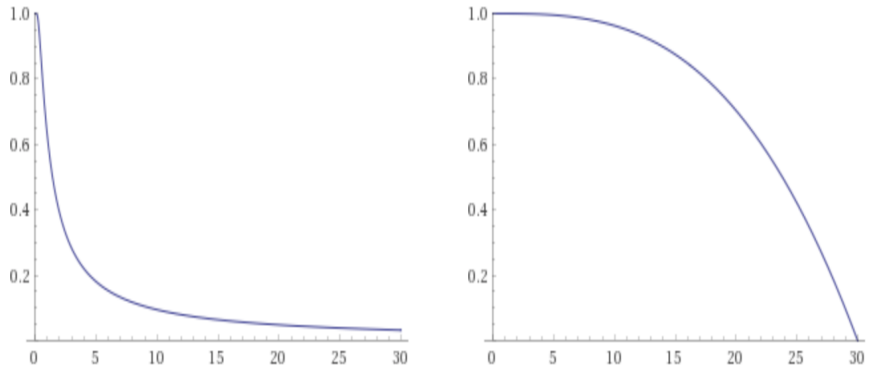
and

$$f_e = 1 - \left(\frac{n_e}{|\mathcal{Z}|}\right)^a \tag{3.5}$$

where β is a constant that dictates the rate of decay of the exponential, and a is a constant dictating the length of the plateaued region before the curve turns. The shape of these functions are shown in Fig. 3.1.

The exponential function is a piece-wise function defined for values of $n_e \geq 0$. At $n_e = 0$ we impose the value $f_e = 1$ due to the function becoming impossible to evaluate at this point. The effect of this function on the popularity scaling factor, and therefore the final edge weight, is as follows: low edge traversals n_e

[3]Other graph optimisation algorithms can handle negative edge weights such as the Bellman-Ford algorithm, but there is still a risk that the presence of negative edges could set up a negative cycle; a cycle whose edges sum to a negative number. If such cycles exist then an optimum path cannot be computed. Indeed, the Bellman-Ford algorithm is often employed in cases where negative cycles are the target to be found.



(a) An example curve of the form of equation 3.4.

(b) An example curve of the form of equation 3.5.

Figure 3.1: Proposed relationships between edge count n_e (x-axis) and scaling factor f_e (y-axis).

initially lead to a sharp drop in f_e, but as edge traversals increase, the change in f_e becomes less severe. In terms of an end user, they should notice that generated routes start to more closely match the most popular roads after only a few traversals, and that any further traversals have minimal effect on the routes. The choice of β depends on the desired output from a potential user; larger β would lead to a shallower curve and smaller β to a steeper curve.

The second function behaves in a manner where, at low n_e the scaling factor remains constant at unity, until a turning point. At this point the f_e decreases ever more rapidly with increasing n_e until $n_e = |\mathcal{Z}|$ where the function reaches zero. The effect for an end user is that any generated paths will not be affected by road popularity if the traversal frequency is low. Only highly traversed roads are likely to be included in generated routes. The constant a can be adjusted to control the length of the initial plateau and the sharpness of the fall. For $a = 1$ the function becomes linear, whereas at the other extreme where $a \to \infty$ the function becomes a step function.

3.2.2 Map-Matching from Trace Data

We have so far assumed that data regarding the frequency of edge traversals exists for a given user of the above system. In practice, calculating edge counts represents a significant part of the preprocessing work. We have access to trajectories from an individual that have been recorded while exercising and commuting. The trajectories move through the same physical space as the data graph structure of the London road network, but are totally unconnected to it. The task at hand is to identify which road segment the individual was traversing given the coordinate point of a trajectory, and then build a connected sequence of edges for the whole trajectory. The task is known as map-matching, of which a simple method is described in this section.

A naïve approach to map-matching is to simply find the edge that has the shortest Euclidean distance between it and the trajectory coordinate. Algorithm 1 describes how this would be achieved. It consists of a double-nested for-loop

that loops through every coordinate point in a trajectory. It then loops through every node in the graph[4] and checks whether it is closer than the current closest node. If a node is closer, it then becomes the next *"closest node"*. After all nodes have been looped through, the closest node is appended to the set of all closest nodes found so far.

Algorithm 1 Naïve Map-Matching Algorithm

Require: Trajectory coordinates $c_i \in \{c_1, c_2, ..., c_l\}$
Require: Node coordinates of a graph $v \in V$
Ensure: Set of nodes T that match the trajectory

```
for i = 1 to l do
    minDist ← ∞
    for all v ∈ V do
        if d(c_i, v) < minDist then
            v_closest ← v
            minDist ← d(c_i, v)
        end if
    end for
    T ← T ∪ {v_closest}
end for
return T
```

The time complexity of algorithm 1 is $O(nl)$ where $n = |V|$ is the number of vertices in the graph, and l is the length of the trajectory (or the number of coordinates). There are a number of issues with this simple approach. The most obvious is that we assume that proximity is the best indicator to determine which edge is being traversed. In practice, because the coordinates are gathered from a GPS device, the accuracy in terms of location and time stamping is sometimes poor. An improved method for map-matching will be presented in section 4.4.2.

3.3 Reinforcement Learning

Using the graph optimisation frameworks described above, it is possible to compute the least polluted path through a road network. This method can be extended to include any edge data, combined in an additive fashion or as a scaling factor to the edge weights. Road popularity can be incorporated to alter the edge weights to make them more likely to be included in any paths generated by any optimisation algorithm.

However, there exists two main weaknesses in terms of a user-facing application:

- Non-generalisability: Applying a popularity scaling factor to each edge only informs us of those edges that have been traversed. If the source and target node fall in an area where there is low (or no) previous traversal data, any graph optimisation algorithm will infer (perhaps incorrectly) that there is no preference between roads other than their edge weight.

[4]Nodes are chosen instead of edges because there is easy access to coordinate information of the nodes, see table in Fig. 4.1.

- Does not capture human behaviour: When selecting a path, humans do not only minimise a single attribute such as air pollution or distance, but rather a much more complex cost function, incorporating many features, each with a unique and personal importance. The above frameworks account for only a single attribute.

In order to correct for these shortcomings, we turn to the subject area of reinforcement learning. Manipulating the notation of our graph representation of the London road network, we now talk in terms of an agent moving through an environment composed of discrete *states*, selecting *actions* to transition between states.

3.3.1 MDP Representation

The mathematical framework underlying the behaviour of our agent is called a Markov Decision Process (MDP), of which a simple example is shown in Fig. 3.2

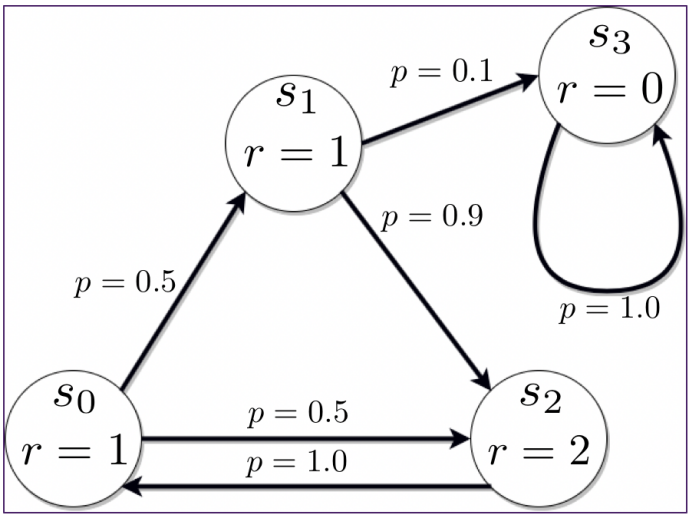


Figure 3.2: A simple example of an MDP with four states and six actions.

Formal Notation and Definitions

MDPs are defined by the tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \gamma, \mathcal{R})$ where:

- $\mathcal{S}$ is the set of environment states.
- $\mathcal{A}$ is the set of actions.
- $\mathcal{P} \in [0, 1]$ contains transition probabilities for $s \to a \to s'$.
- $\gamma \in [0, 1]$ is the discount factor for future rewards.
- $\mathcal{R}$ is the reward function.

The environment is represented by a set of states $s \in \mathcal{S}$ which is also known as the *state space*. In each state, the agent may choose an action $a \in \mathcal{A}(s)$ drawn from the subset of all actions available to it in the state s. This is a discrete process, such that a time-step t elapses every time the agent selects an action and makes the transition $s \to s'$. Upon completion of a transition, the environment emits a reward $\mathcal{R}^a_{ss'}$ for the transition $s \to s'$ via the action a. This will often be contracted to r_t meaning the reward at time-step t[5].

The mapping of state to action is governed by the agent's policy π where $\pi(a|s) = P[a_t = a|s_t = s]$, or, the probability that a particular action is taken given the current state[6]. The principle problem that RL attempts to solve is to find an optimal policy π^* that allows the agent is to move through the state space while maximising its cumulative reward, the *return*, in the long run. The return is given by the sum of all rewards from time-step $t+1$ to T, the terminal time-step[7].

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{T} \gamma^k r_{t+k+1} \tag{3.6}$$

The parameter γ is the *discount factor* taking values $0 \leq \gamma \leq 1$. It is introduced to control the level of foresight the agent has when calculating the return; when $\gamma = 0$, the agent only considers immediate rewards, when $\gamma = 1$ it treats all future rewards equally[8].

At each state, the agent must have a mechanism for evaluating the *value* of being in that state. This is a function of its *expected* cumulative rewards given a particular policy π. We therefore define V^π as the *value function* for policy π, which evaluates "how good" being in state s is.

$$\begin{aligned} V^\pi(s) &= E_\pi[R_t \mid s_t = s] \\ &= E_\pi\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \;\middle|\; s_t = s\right] \end{aligned} \tag{3.7}$$

We may define a further utility measure called the *action-value* function. The action-value function for policy π, $Q^\pi(s, a)$, is defined as the value of taking action a in state s at time t:

$$\begin{aligned} Q^\pi(s,a) &= E_\pi[R_t \mid s_t = s, a_t = a] \\ &= E_\pi\left[\sum_{k=0}^{T} \gamma^k r_{t+k+1} \;\middle|\; s_t = s, a_t = a\right] \end{aligned} \tag{3.8}$$

[5]The reward may be attributed to different elements of the state-action cycle, for example $\mathcal{R}_s$ would signify the reward for *being in* state s, or $\mathcal{R}_a$ the reward for taking action a. The exact characterisation will depend on how the problem is modelled. For now, we will assume that only a single reward is emitted *at some point* in each time-step.

[6]This is the stochastic case. The deterministic case is simply $\pi(s) = a$.

[7]Depending on the context, the agent may encounter a natural terminal state, which transitions only to itself and yields zero reward. Tasks like this are called *episodic*. Tasks where $T = \infty$ are called *continuous*.

[8]Note that $\gamma \neq 1$ for continuous tasks because the return would tend to infinity.

Notice that the only difference in equations 3.7 and 3.8 is the inclusion of an action. This can be interpreted as the value of being in a particular state and taking a particular action, and then continuing under the policy π thereafter.

It is possible, and indeed necessary, that value functions satisfy particular recursive relationships [36]. Following from equation 3.7, we can show that:

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\left[r_{t+1} + \gamma \sum_{k=0}^{T} \gamma^{k} r_{t+k+2} \;\middle|\; s_t = s \right] \\ &= \sum_{a \in \mathcal{A}(s)} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma E_{\pi} \left\{ \sum_{k=0}^{T} \gamma^{k} r_{t+k+2} \;\middle|\; s_{t+1} = s' \right\} \right] \\ &= \sum_{a \in \mathcal{A}(s)} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^{\pi}(s') \right] \end{aligned} \tag{3.9}$$

Following a similar derivation to the one above, we can arrive at the same recursive relationship for the action-value function:

$$Q^{\pi}(s,a) = \sum_{s' \in \mathcal{S}} P^{a}_{ss'} \left[R^{a}_{ss'} + \gamma \sum_{a' \in \mathcal{A}(s')} \pi(s',a') Q^{\pi}(s',a') \right] \tag{3.10}$$

Equations 3.9 and 3.10 are known as the Bellman equations. First identified by R. Bellman in 1957 [6], they average over all future the possibilities, stating that the value of the initial state s must equal the (discounted) value of the expected next state s', plus the reward expected along the way [36].

Since we seek an optimal policy, we want a policy that yields the most reward in the long run. A policy is therefore better than another if and only if the value of being in a state s under policy π is greater than or equal to the value of being in the same state under a different policy π'. With a slight abuse of notation, mathematically this is expressed as:

$$\begin{aligned} &\pi \geq \pi' \\ \text{iff} \quad &V^{\pi}(s) \geq V^{\pi'}(s) \end{aligned} \tag{3.11}$$

Therefore, the optimal value function and action-value function can be defined in the following way:

$$V^{*}(s) = \max_{\pi} V^{\pi}(s) \tag{3.12}$$

$$Q^{*}(s,a) = \max_{\pi} Q^{\pi}(s,a) \tag{3.13}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$. These optimal value functions must also satisfy the recursive condition of the Bellman equations 3.9 and 3.10. These are called the Bellman optimality equations:

$$V^{*}(s) = \max_{a} \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma V^{*}(s') \right] \tag{3.14}$$

$$Q^{*}(s,a) = \sum_{s' \in \mathcal{S}} \mathcal{P}^{a}_{ss'} \left[\mathcal{R}^{a}_{ss'} + \gamma \max_{a'} Q^{*}(s',a') \right] \tag{3.15}$$

Once we have $V^*(s)$ it is relatively straight forward to arrive at an optimal policy. For every state, there is at least one action that, if taken, will attain the maximum in the Bellman optimality equation. In other words, the action(s) that appear best after only considering the immediate actions and their successor states will be optimal actions. Or, any policy that is greedy with respect to V^* is an optimal policy. We know this because V^* already takes into account all possible future behaviour at every state. This line of analysis can also be extended to Q^*. However, in this case it becomes even easier to arrive at an optimal policy. One need only consider immediate actions that maximise the Bellman optimality equation. Optimal actions may be selected without having to know the value of being in any successor states.

The Bellman optimality equations 3.14 and 3.15 are systems of $|\mathcal{S}|$ equations, one for each state, with $|\mathcal{S}|$ unknowns. Explicitly solving these equations is often prohibitively computationally expensive. Furthermore, we may not accurately know the dynamics of the environment (transition probabilities $\mathcal{P}$), as is the case for the environment considered in this project. One often has to be satisfied with approximate solutions, several of which will be presented in the next section.

3.3.2 Solution Methods

Dynamic Programming

Requiring knowledge of the environment dynamics, dynamic programming (DP) will not be used to solve the RL problem that this report is centred around. However, it is presented here because of its theoretical importance. With all solution methods, we need a learning mechanism to successively evaluate and improve policies. Equation 3.16 shows successive policy evaluations ($\xrightarrow{\text{E}}$) and policy improvements ($\xrightarrow{\text{I}}$) that converge on the optimal policy π^* and value function V^*.

$$\pi_0 \xrightarrow{\text{E}} V^{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} V^{\pi_1} \xrightarrow{\text{I}} \pi_2 \xrightarrow{\text{E}} \dots \xrightarrow{\text{I}} \pi^* \xrightarrow{\text{E}} V^* \tag{3.16}$$

The first step is to take an arbitrary initial policy π_0 and evaluate it to get V^{π_0}. We already have a recursive definition for V^π in the form of the Bellman equation (equation 3.9), which a system of $|\mathcal{S}|$ simultaneous linear equations with $|\mathcal{S}|$ unknowns. We follow an iterative solution method whereby the sequence of approximate value functions $V_0, V_1, V_3, ...$ are generated using the Bellman equation. The update rule is as follows:

$$V_{k+1}(s) = \sum_{a \in \mathcal{A}(s)} \pi(s,a) \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V_k(s')\right] \tag{3.17}$$

As $k \to \infty$, $V_k = V^\pi$. This update rule is known as a *full backup* operation because it replaces the current value at s with a new value obtained from the values of all successor states s'.

Once V^{π_0} is established, we may now improve it to recover π_1 (as in equation 3.16). We already know the value of being in s under a policy π, but what if now we decided to take an action that is not dictated by the current π? Would the value of that action be greater than the current policy action? We can answer this question by considering the action-value function $Q^\pi(s,a)$ where the action

taken is $a = \pi'(s) \neq \pi(s)$. The criterion we want to evaluate is thus:

$$Q^{\pi}(s, \pi'(s)) \geq V^{\pi}(s) \tag{3.18}$$

We now have a mechanism to evaluate a change in policy at a single state. We want to, however, extend this to *all* possible states and actions. This means we select the best immediate action at each state according to $Q^{\pi}(s, a)$. Therefore, the new policy can be expressed as:

$$\begin{aligned} \pi'(s) &= \arg\max_a Q^{\pi}(s, a) \\ &= \arg\max_a \sum_{s' \in \mathcal{S}} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^{\pi}(s')\right] \end{aligned} \tag{3.19}$$

Monte-Carlo

While DP required complete knowledge of the transition probabilities $\mathcal{P}^a_{ss'}$, we do not assume such knowledge when applying Monte-Carlo methods (MC). All we need is sample experience of state transitions through the state space. This method learns the value of a particular state by averaging the return over a full episode. In this sense, we are not updating $V^{\pi}(s)$ on a step-by-step basis, but rather an episode-by-episode basis.

Evaluating $V^{\pi}(s)$ is pleasingly intuitive. We already know from equation 3.7 that the value function is the expected cumulative reward while following the policy π. So, we generate an episode using π, and after the first visit to each state s calculate the return that follows (remembering that the return is the sum of the discounted rewards, see equation 3.6). A number of episodes are generated in this manner, and the return for each s calculated. The value of a state $V^{\pi}(s)$ is the average of the returns after n episodes. Of course, as $n \to \infty$, the approximate calculation of $V^{\pi}(s)$ becomes an equality[9]. In practice, however, policy evaluation is completed after only one episode. This practice is introduced to avoid generating an infinite number of episodes, with the acceptance of not achieving complete policy evaluation[10].

Once $V^{\pi}(s)$ is established, recovering an improved policy π' follows the same method of generalised policy iteration (equation 3.16) described in the previous section. We act greedily with respect to the calculated value of each state to generate the new, improved, policy. That is, at each state, choose the action that maximises the action-value function (see equation 3.19).

MC has a number of advantages over DP. We have already mentioned that MC does not require knowledge of the dynamics of the environment. A further advantage is that, since the estimate of the value of a state is independent of the estimates of the other states (does not "bootstrap" as in DP), the price of computation is independent of $|\mathcal{S}|$. The upshot of this is that one could calculate the value of a subset of the state space, only generating episodes for the particular states of interest.

[9]The error on the average returns is $1/\sqrt{n}$, so by the law of large numbers we can see MC converges to $V^{\pi}(s)$ in the limit $n \to \infty$.

[10]This is justified since we do not expect to achieve complete policy evaluation, only to move the value function *toward* V^{π_k}.

Temporal Difference

Temporal difference (TD) learning mixes elements from DP and MC. Like MC, we don't require any knowledge of the environment dynamics, this is because the solution method involves learning from experience. But like DP, updates are made on the consideration of learned estimates. We shall explore how the TC method works by comparing it to MC in the previous section. The update rule for MC is as follows:

$$V(s_t) \leftarrow V(s_t) + \eta[R_t - V(s_t)] \tag{3.20}$$

In words, we wait until we know the return after a state (at time t) is visited, and then use this as the new target for $V(s_t)$. The value η is used as a learning rate with values typically close to, but less than, 1 to restrict the speed with which a solution is found. In TD we need not wait until the return is known, but simply update using the reward at the next time step $t+1$, and the estimated value $V(s_{t+1})$. The update rule for the simplest TD method known as TD(0) is as follows:

$$V(s_t) \leftarrow V(s_t) + \eta[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \tag{3.21}$$

This TD(0) update rule has a number of advantages over DP and MC methods. As as ready stated, there is no requirement that the transition probabilities $\mathcal{P}^a_{ss'}$ be known. This is especially advantageous in our case since we have no access to these probabilities in the state space of the London road network. A second advantage is that updates of the value of a state can occur in an incremental, on-line fashion, rather than having to wait until an episode has finished. Episodes through the London state space are very unlikely to visit every state and action. Though we do not expect every state to be visited, episodes could still take an intractably long time to execute, making learning slow.

3.3.3 Q-Learning

In the previous section we explored three solution methods for the forward RL problem of finding an optimal policy π^* to move episodically through a state space while accumulating the highest reward. We now move to the method that is used to solve the problem set out in section 1.2.3; applying an RL algorithm to generate optimum routes through London. We will for the time being assume that a reward function has already been set. The issue of setting a reward function will be dealt with in section 3.5 when detailing the theory behind inverse reinforcement learning.

The technique described in this section was first introduced by C. Watkins in 1989 [40] and is called Q-Learning. As the name may suggest, it is not the value function that is being optimised but the action-value function $Q(s,a)$. This is a TD method because it learns from experience (like MC), and from previous estimates of the action-value function (like DP). The dominant feature of Q-Learning is that $Q(s,a)$ is updated using delayed rewards, the impact of which propagates through the state space from possible future states to the current state. The update rule is as follows:

$$Q(s,a) \leftarrow Q(s,a) + \eta[r + \gamma \max_{a'} Q(s',a') - Q(s,a)] \tag{3.22}$$

Let us examine the terms inside the brackets. The first thing to note is the suspension of the t subscripts introduced in update rules 3.20 and 3.21, replaced with s for the current state, and s' for the subsequent state. The reward r in this case is the reward that is obtained when action a is taken in state s. The next term in the brackets, $\max_{a'} Q(s', a')$, can be interpreted as looking at the next state s' after action a and returning the maximum possible Q value in the next state. Or, return the maximum Q value after having checked for the best possible action in state s'. This foresight is restricted by the discount factor γ that typically takes a value slightly less than one. The final term $-Q(s, a)$ acts to limit the growth of the Q values as the learning algorithm iterates toward a solution.

It is important to note that, while at each state s the agent only examines the best possible action a' at the next state s', the discounted rewards from future states propagate down to the current state-action decision. They propagate down as a result of their being discovered by the agent in previous episodes of the Q-Learning algorithm, and being used to update Q values ahead of the current state-action pair. The degree to which future rewards have an influence in the Q value of the current $Q(s, a)$ is dictated by the discount γ, and the speed with which these rewards are felt is dictated by the learning rate η. In psudo-code, the Q-Learning algorithm is shown in algorithm 2.

Algorithm 2 Q-Learning control algorithm

Require: Learning rate $\eta \in [0, 1]$
Require: Discount $\gamma \in [0, 1]$
Require: Source and target states s_σ, s_τ
Ensure: Optimum action-value function $Q^*(s, a)$
1: Initialise $Q(s, a) \leftarrow \mathcal{R}$
2: **for** $i = 1$ to `num_episodes` **do**
3: $s \leftarrow s_\sigma$
4: **while** $s \neq s_\tau$ **do**
5: $a \leftarrow \arg\max_a Q(s, a)$
6: Take action a, observe r and s'
7: $Q(s, a) \leftarrow Q(s, a) + \eta[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
8: $s \leftarrow s'$
9: **end while**
10: **end for**
11: **return** $Q^*(s, a)$

The algorithm begins by initialising the action-value function with the same values as the reward function. This is different to most standard incarnations of this algorithm, which initialise $Q(s, a)$ arbitrarily, or with all zeros[11]. However, in this project the initial action-value function is set to equal the reward function $\mathcal{R}$. The reward function is direction independent, meaning the sequence $s \rightarrow a \rightarrow s'$ yields the same reward as $s' \rightarrow a \rightarrow s$. This will initially be true for Q values, but over the course of the algorithm, direction-specificity will manifest. The decision to use the reward function $\mathcal{R}$ to initialise $Q(s, a)$ will be explored

[11] If the action-value function is initialised as 0 for all state-action pairs, there would need to be an `if-else` junction at line 5 where: if the state has not yet been visited (value of 0), choose an action at random.

further in section 4.9. For now, we simply state that doing this helps drive the agent toward the terminal state s_τ from the very first episode, rather than explore randomly.

At the beginning of every new episode, the initial state is set as s_σ; the user defined starting point. From there, a is set to be the action that yields the maximum Q value. The agent then takes this action, recording the reward r and the next state s'. The update rule from equation 3.22 is then applied to $Q(s, a)$. Finally, the current state s is replaced by s', before returning to line 5. Lines 5 - 8 repeat until the target state s_τ is reached, where the agent returns to the source state s_σ. The whole process is repeated for a number of episodes until we are satisfied of the final $Q(s, a)$.

Once an optimal action-value function $Q^*(s, a)$ is found, we proceed as we have done with the other solution methods discussed in this section; act greedy with respect to $Q^*(s, a)$ to recover an optimal policy π^* that tells our agent which action to take at a given state. Moving through the state space from s_σ to s_τ under π^* would generate a path that maximises the agents return. Such a path is sought from a user given their unique value function and specified s_σ and s_τ, and is one of the ultimate goal of this project.

3.3.4 ϵ-Greedy Action Selection

With algorithm 2, the current process for finding an optimal path is as follows. The agent starts at s_σ, selects the action that maximises the current Q value, then updates it combining the reward r yielded from taking that action, and the maximum Q value from the next available actions a'. The agent moves ever closer to the terminal state s_τ, while the rewards propagate down through the explored paths, informing the agent about the rewards of future states.

The criterion for selecting an action is greedy, meaning only the maximum Q value at any given state is considered. We trust, however, that because of the propagated rewards from future states, the immediate best action is also the best action in the long run. At the beginning of the algorithm this behaviour might cause a problem. Because the agent has no knowledge of future rewards, it is conceivable that a "wrong" decision be made in the early episodes. These wrong decisions would then propagate as episodes elapse, and because of the greedy property of the algorithm, effectively lock in a path from the beginning that may not yield the highest return.

This is known as *exploitation*. The agent makes decisions based on the knowledge it already has of the environment, irrespective of whether that knowledge leads to more optimal paths. The opposite behaviour is called *exploration*. This is where the agent periodically decides to take a random action from those available, instead of the action that yields the greatest Q value. The probability that such a random action is taken is given by the parameter ϵ, leading to the name for this adaptation; "ϵ-greedy". The purpose of taking random actions is to allow exploration of states outside what would be chosen under the current best policy. Although, due to the size of the state space[12], we do not expect the agent to explore every state-action pair, some exploration is desirable to avoid committing to a sub-optimal path early on in the algorithm.

At the other extreme, where the agent acts completely with the aim to

[12] $|\mathcal{S}| = 18110$ and $|\mathcal{A}| = 23531$.

explore the state space, no stable solution if found. Ideal behaviour would be to have early episodes focus more on exploration, and then as potential optimal paths are identified, shift to a more exploitative policy. This would mean starting with a high value for ϵ, making it more likely the agent take a random action, then after each episode reduce ϵ until a purely exploitative policy is implemented (when $\epsilon = 0$). Algorithm 3 outlines the ϵ-greedy described above.

Algorithm 3 Q-Learning algorithm with ϵ-greedy action selection

Require: Initial $\eta \in [0, 1]$
Require: Initial $\epsilon \in [0, 1]$
Require: Decay rate $\lambda \lesssim 1$
Require: Discount $\gamma \in [0, 1]$
Require: Source and target states s_σ, s_τ
Ensure: Optimum action-value function $Q^*(s, a)$
1: Initialise $Q(s, a) \leftarrow \mathcal{R}$
2: **for** $i = 1$ to `num_episodes` **do**
3: $s \leftarrow s_\sigma$
4: $\epsilon \leftarrow \lambda * \epsilon$
5: **while** $s \neq s_\tau$ **do**
6: **if** rand_num $< \epsilon$ **then**
7: $a \leftarrow$ random action from $\mathcal{A}(s)$
8: **else**
9: $a \leftarrow \arg\max_a Q(s, a)$
10: **end if**
11: Take action a, observe r and s'
12: $Q(s, a) \leftarrow Q(s, a) + \eta[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
13: $s \leftarrow s'$
14: **end while**
15: **end for**
16: **return** $Q^*(s, a)$

Algorithm 3 is almost identical to algorithm 2. Where they differ most significantly is the inclusion of an `if` statement starting on line 6 of the former. A random number between 0 and 1 is generated, and if this number is smaller than the current value of ϵ then a random action is chosen from all those available at the current state. Otherwise, the agent is greedy and selects the action with the highest Q value. After every episode ϵ is multiplied by the decay factor λ, reducing its magnitude, and therefore lowering the threshold for a random number to be below before a random action is taken. The effect of this is that action selection becomes ever more exploitative, taking the greedy option, episode after episode.

3.4 London as a State Space

We now take some time out of the formal theory presentation to address an issue that has until now been ignored. That is the task of selecting an appropriate state and action representation of the London road network. Intuition might suggest that, because the agent (someone commuting or exercising) transitions

form one road to the next, states be represented as road segments, and actions as intersections. This is a common representation for many IRL applications concerning routing problems, following similar work by H. Wu *et al.* [42] and B. Ziebart *et al.* [47]. However, we adopt the opposite stance and choose to represent states as nodes, and actions as edges. This has the cleaner visual properties associated with conceptualising the state space (see Fig. 3.2), but will also help when implementing the Q-Learning algorithm described above.

3.5 Inverse Reinforcement Learning

In the RL frameworks described above, the agent is provided with a reward function. Whenever an action is executed in some state, feedback about the agent's performance is provided. This reward function is used to obtain an optimal policy; where the expected discounted return is maximised. IRL is the case where the optimal policy (or historic samples from that policy) is given, and the task becomes that of recovering the reward function for which the given behaviour is optimal. A summary of the differences between RL and IRL problems is shown in Fig. 3.3.

	RL	IRL
Input	(partially observabed) $\mathcal{R}$	π^* (or traces ζ_{π^*})
Output	π^*	$\mathcal{R}$

Figure 3.3: Summary of inputs and outputs of RL and IRL problems.

In this research, the reward function is unknown to the agent. Some attempts have been made so far to artificially construct reward functions based on road segment length and average air pollution exposure. For example, a forward RL algorithm could be applied to the London road network to generate an optimal policy using air pollution as a reward function[13]. The policy would then be able to generate a route matching that from Dijkstra's algorithm. This is theoretically sound, but an unnecessary extension.

What we would prefer is a reward function that reflects the desirability of each road having taken into account multiple factors. What is more, we want to be able to calculate how important each of those factors are in the selection of that particular road. IRL offers the ability to generate a reward function based on the consideration of observed optimal behaviour through the state space. In other words, given historic exercise and commuter data in London, we can calculate the desirability of every road, even when a road has not been visited, and generate optimal routes through London based on this desirability.

3.5.1 Ng & Russel paper

The first documentation of solution methods for IRL was in a 2000 paper by A. Ng and S. Russell titled *"Algorithms for Inverse Reinforcement Learning"* [26]. They introduce three algorithms to be implemented in the following cases:

[13]This would actually be a *cost* function with the polarity of the γ_{avg} values reversed. This is to ensure any optimal policy would "seek" areas of low air pollution, rather than avoid them.

- π^* is known and the state space is small and finite.
- π^* is known and the state space is large or possibly infinite.
- π^* is unknown but trajectories can be obtained under π^*.

These three algorithms are explored briefly here, before moving on to the specifics of the solution method used in this project.

Linear Programming for Small State Spaces

The first assumption we must make for this solution method is that we have complete knowledge of the expert's[14] optimal policy π^*, which is also deterministic, meaning states map to a single action $\pi(s) = a$. Knowledge of state transition probabilities $\boldsymbol{P}^{\pi}$ for always choosing $\pi(s) = a$ in state s. Note the difference in notation with $\mathcal{P}^{a}_{ss'}$, meaning the probability of transitioning $s \to s'$ after having taken action a. Note also that $\boldsymbol{P}^{\pi}$ is a vector of dimension $|\mathcal{S}| \times |\mathcal{S}|$.

For each state, there are $k-1$ actions that are not selected by the policy, i.e. $a \neq \pi(s)$. This leads to there being a set of transition matrices for the probabilities of non-policy transitions: $\boldsymbol{P}^{\neg\pi} = \{\boldsymbol{P}^1, \boldsymbol{P}^2, ..., \boldsymbol{P}^{k-1}\}$. The main result from Ng & Russel's paper defines the solution set of reward function as follows:

$$\forall \boldsymbol{P}^i \in \boldsymbol{P}^{\neg\pi} : (\boldsymbol{P}^i - \boldsymbol{P}^{\pi})(\boldsymbol{I} - \gamma \boldsymbol{P}^{\pi})^{-1}\mathcal{R} \succeq 0 \tag{3.23}$$

where $\succeq$ means every element in a vector is greater than or equal to the equivalent element of another vector. Here, $\mathcal{R}$ represents a column vector with the reward associated with every state in the MDP. Equation 3.23 represents $|\mathcal{S}|$ linear equations with the same number of unknowns, so the exact rewards for each state can be found using a linear programming method. The reward set, however, contains many degenerate solutions. For example, $\mathcal{R} = 0$ is a solution. A number of heuristics are suggested to alleviate this problem. The first is to maximise the interval between the policy's Q values and Q values associated with other actions:

$$\text{maximise} \sum_{s \in \mathcal{S}} \left[Q^{\pi}(s, \pi(s)) - \max_{a \in \mathcal{A} \setminus \pi(s)} Q^{\pi}(s, a) \right] \tag{3.24}$$

Applying this heuristic will select the reward function from the solution set that forces the difference as large as possible between the policy action and the second best action in all states. Requiring this condition successfully eliminates degenerate solutions, but leads to reward functions that are very different to the real reward function, with an increased likelihood of more extreme values. The second heuristic is the introduction of a penalty term in the maximisation above, which leads to a minimisation of the reward function's L1-norm $||\mathcal{R}||_1$. Further to this, a maximum limit to the size of individual rewards is set $\forall s \in \mathcal{S} : |\mathcal{R}(s)| \leq R_{\max}$.

[14] The term "agent" is replaced by "expert" in IRL problems to reflect the fact that we want to learn the reward function from behaviour that is assumed to be optimal.

Linearisation of Rewards

While equation 3.23 with heuristics allows for exact reward functions to be found, in some situations it may be necessary to linearise individual rewards. This is done for three reasons: the first is that the state space may be very large (or infinite). The second is that observations of the optimal policy may not take place over the entirety of the state space. The third reason is that by linearly combining multiple reward functions, we can explicitly know which features provide the largest contribution to the overall reward. All three conditions are met with this project (large state space of the London road network, and small number of trajectories from exercise data), so we proceed by reformulating the reward function as a linear combination of d fixed, known, and bounded basis functions:

$$\hat{\mathcal{R}}(a) = \alpha_1\phi_1(a) + \alpha_2\phi_2(a) + ... = \sum_{i=1}^{d} \alpha_i\phi_i(a) \tag{3.25}$$

It should be highlighted here that $\hat{\mathcal{R}}$ is an approximation of the reward because of the finite number of basis vectors. Also, the rewards are action dependant and state-independent, meaning the same reward is yielded through the transition $s \to a \to s'$ as well as the transition $s' \to a \to s$. This reflects the nature of the environment we have chosen for the London road network (see section 3.4); both directions of traversal for a road segment yields the same reward.

The basis vectors $\phi_i(a)$ are also action dependant, and contain information about the features of the road segments. They are column vectors of length $|\mathcal{A}|$ populated with 0s and 1s indicating the presence of a particular feature. For example, the second feature could be "road type = A-road", so for action a_3 the feature vector is $\phi_2(a_3) = (0, 0, 1, 0, ...)^\top$, plus the any other features that might be present.

The α_i coefficients are the values we want to fit. Their magnitudes indicate the *importance* of each feature to a particular user. In principle, these coefficients could take any real-number value, negative and positive. However, the range of α_i in this project will be restricted to $0 \leq \alpha \leq 1$. This reflects the fact that features always increase the overall reward of an action (negative values would reflect a cost), and it also helps compare the difference in importance between features by defining identical domains.

Linear Programming with Trajectories

The final algorithm in Ng & Russel's paper is applicable when the optimal policy is not fully known, but rather suggested via a number of trajectories through the state space. This exactly matches the scenario encountered in this report; sample exercise routes through London. These trajectories are ordered lists of state-action pairs under the assumed optimal policy π^* of the expert. Trajectories may be of different length:

$$\zeta_{\pi^*} = \{(s_0, a_0), (s_1, a_1), ..., (s_T, a_T)\} \tag{3.26}$$

We want to calculate the empirical value of a trajectory $\hat{V}(\zeta)$. Referencing equation 3.7, the value of being in a state is the cumulative discounted rewards

yielded under the optimal policy. Considering the ith basis function, the value of an entire trajectory can therefore be written as:

$$\begin{aligned}\hat{V}_i(\zeta) &= \phi_i(a_0) + \gamma\phi_i(a_1) + \gamma^2\phi_i(a_2) + ... \\ &= \sum_{a_j \in \zeta} \gamma^j \phi_i(a_j)\end{aligned} \tag{3.27}$$

Summing over the d basis functions, the overall empirical value of a single trajectory becomes:

$$\begin{aligned}\hat{V}(\zeta) &= \alpha_1\hat{V}_1(\zeta) + \alpha_2\hat{V}_2(\zeta) + ... \\ &= \sum_{i=1}^{d} \alpha_i \hat{V}_i(\zeta)\end{aligned} \tag{3.28}$$

As stated before, we want to find the parameters α_i such that expert trajectories ζ_{π^*} yield a higher value than any other non-policy trajectory. To start the algorithm, generate an arbitrary trajectory under a random policy π^1. Comparing this trajectory to the expert trajectory, clearly we want:

$$\hat{V}(\zeta_{\pi^*}) \geq \hat{V}(\zeta_{\pi^1}) \tag{3.29}$$

Using linear programming we can find the best fitting parameters α_i and generate a new reward function. With this new reward function, a policy π^2 can be generated that is optimal for that given reward function. A trajectory from this new policy can be compared to the expert trajectory using equation 3.29. This inductive process continues for m iterations, growing the linear programming constraints to:

$$\forall \zeta_{\pi^{\neg *}} \in \{\zeta_{\pi^1}, \zeta_{\pi^2}, ..., \zeta_{\pi^m}\} : \hat{V}(\zeta_{\pi^*}) \geq \hat{V}(\zeta_{\pi^{\neg *}}) \tag{3.30}$$

The final linear programming formulation becomes:

$$\begin{aligned}\text{maximise} \quad & \sum_{i=1}^{m} p(\hat{V}(\zeta_{\pi^*}) - \hat{V}(\zeta_{\pi^i})), \\ \text{such that} \quad & |\alpha_i| \leq 1, i = 1, 2, ..., d\end{aligned} \tag{3.31}$$

Because we have modelled the reward function as a linear function operator, there is the possibility of not being able to express any reward function for which π is optimal. To circumvent this, the constraints in equation 3.30 may be relaxed, paying a penalty when they are violated. The penalty function $p(x)$ is given by $p(x) = x$ if $x \geq 0$, and $p(x) = c \cdot x$ otherwise[15].

3.5.2 Setting α with Heuristics

A simpler approach to those described above will be implemented in this research project. It follows method that has been adapted from a method introduced

[15] Ng and Russell select $c = 2$ in [26].

by J. Choi and K. Kim in their paper on predicting routes from taxi driver GPS data [7]. As a benchmark for testing the efficacy of their Bayesian nonparametric approach to feature construction, they set the feature weights α_i heuristically. They are set by counting the feature visitations in the trace data, and then using the normalised counts of feature visitations offset by the maximum count value to set the feature weights accordingly. Although this method was introduced to establish a benchmark for the other algorithms that were the focus of their paper, the method achieved greater accuracy in turn prediction and route prediction that of MaxEntIRL.

The intuition behind this method is simple: the higher the feature count in the trace data, the more desirable that feature is. There are problems with this reasoning, however. By restricting what we define as "desirable" by an expert in the London state space to a finite number of features per road segment, we do not include all the (near countless) other relevant features that may possibly influence determining the rewards in some unknown way. The true nature of the reward function will always be an approximation. This problem can be mitigated somewhat by domain expertise, ensuring that relevant features are selected that may distinguish between desirable and undesirable road segments.

A further problem with heuristically setting the feature weights in this manner is that it does not account for the ubiquity of a particular feature in the state space as a whole. A high feature count in the trace data may simply indicate that that feature is more common in the state space, not that it is more desirable from the point of view of the expert. The final α_i values must take into account this fact by diminishing the weight of features that occur very frequently in the state space, and increases the weight of terms that occur rarely.

Dropping the i subscript temporarily, the feature weights are given by the product of two terms: the feature expectation μ and the adjustment term κ responsible for increasing or decreasing the overall magnitude of α.

$$\alpha = \kappa\mu \tag{3.32}$$

The feature expectation is given by the number of times a feature is encountered in all of the trajectories of the trace data, normalised by the number of trajectories:

$$\mu = \frac{\text{count}(\phi)}{|\mathcal{Z}|} \tag{3.33}$$

The adjustment term is derived in the following way by borrowing from the field of information theory. First, we define the probability that a given action a contains the feature ϕ with the following expression:

$$P(\phi|a) = \frac{|\{a \in \mathcal{A} : \phi \in a\}|}{|\mathcal{A}|} \tag{3.34}$$

The numerator is the number of elements of the set that contains all the actions (road segments) with the feature ϕ. The denominator is the total number of road segments in the network. Therefore, equation 3.34 defines a relative frequency. To get our adjustment term κ we invert the probability and take the natural logarithm:

$$\begin{aligned}\kappa &= \ln\left(\frac{1}{P(\phi|a)}\right) \\ &= \ln\left(\frac{|\mathcal{A}|}{1+|\{a \in \mathcal{A} : \phi \in a\}|}\right)\end{aligned} \tag{3.35}$$

The 1 is added in the denominator as a smoothing technique to avoid division by zero if the feature does not exist in the state space. What we have defined above is in fact the self-information, or the degree of "surprise" (surprisal) one experiences when a random variable is sampled. Treating the feature set as a random variable, let's use some example numbers to explore the behaviour of the self-information term κ. At the extreme, if it is certain that a feature exists in all actions (i.e. $P(\phi|a) = 1$) then the surprisal is, unsurprisingly, 0. The surprisal remains less that 1 until $P(\phi|a) = 1/e \approx 0.368$, meaning that a feature's presence in the state space must be lower than $1/e$ for the self-information to have an augmenting effect on the overall feature weight α (i.e. $\kappa > 1$). At the other extreme, perhaps a feature is very uncommon in the action set, let's say only one occurrence. From equation 3.35 this would give $\kappa \approx 9.373$.

	small κ	large κ
small μ	small α	mid α
large μ	mid α	large α

Figure 3.4: Summary of possible feature weight magnitudes.

Figure 3.4 summarises the various regimes α can take. The feature weights are only large if the feature is common in the trace data, but uncommon in the action set as a whole. The opposite is true for very common features in the action set that do not appear frequently in the trace data. Where either high action-set presence or high trace-data presence is true, the two act to cancel the effect of the other, leaving a mid-range value for the feature weight.

3.6 Current Landscape

The preceding sections in this chapter have explored the theoretical groundwork necessary to achieve the three goals set out in chapter 1. A summary of the different approaches explored in this report is shown in Fig. 3.5.

Starting with the air pollution data provided by Ollie Hamelijnck of Green *et al.*, a value for the average air pollution γ_{avg} is associated with road segments in London. The total air pollution is this average value multiplied by the length of the road. Using this as an edge weight, and providing source and target nodes, an optimal path can be calculated by applying Dijkstra's algorithm with respect to the pollution penalty edge weights. These steps can be followed in the flow chart from the air pollution data through to the optimum paths in the bottom left of the figure. "ED" in this case means "edge data", and we drop the avg subscript for clarity.

Following on from this, we may incorporate historic user data in the form of weights to artificially lower or raise the pollution penalty. This strikes a balance

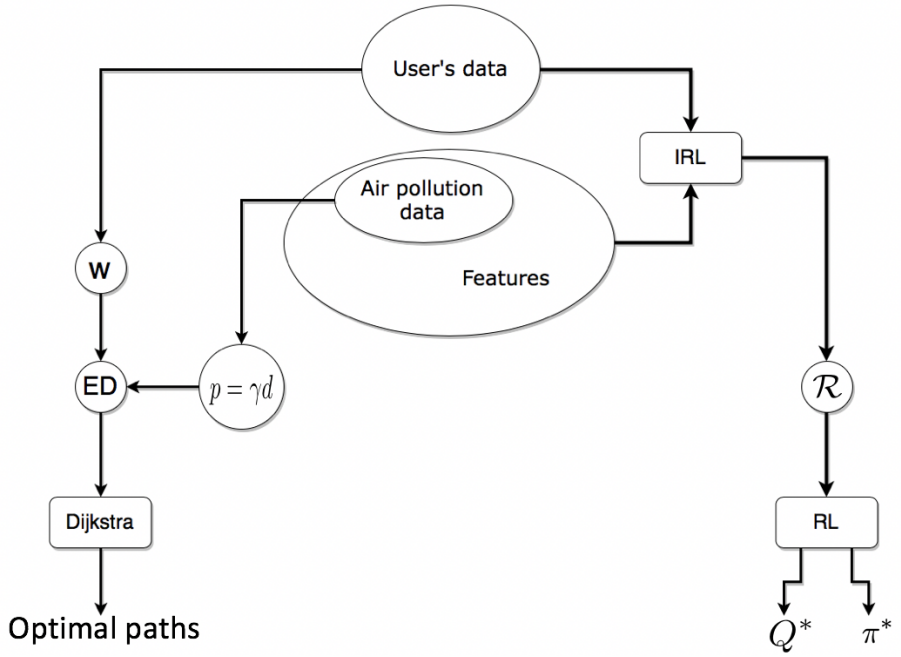


Figure 3.5: Flow chart outlining the various approaches to solve the problems addressed in this project.

between paths that minimise air pollution, but also follow paths that a user is more likely to habitually take.

Chapter 4

Implementation

4.1 Database Storage of Edges and Nodes

Both the London road network as provided by Ordnance Survey [35], and the synthetic pollution data as provided by Ollie Hamelijnck of Green *et al.* were originally in shape file format (.shp). Detailed in Patrick O'Hara's bachelor's thesis [27] is the method he employed to convert and combine the two shape files to a PostgreSQL database. It is this database that we refer to throughout this chapter. The database consists of two tables; one for edges (or road segments), and one for nodes (or intersections). Fig. 4.1 and 4.2 detail the attributes that exist in these tables.

Attribute	Description
nid	Node ID
lat	Latitude coordinate
lon	Longitude coordinate

Figure 4.1: Description of the attributes in the nodes table.

Attribute	Description
eid	Node ID
gamma	Average NO_x along edge (μg m^{-3})
distance	Length of road segment (m)
startnode	Node ID of start node
endnode	Node ID of end node

Figure 4.2: Description of the attributes in the edges table.

From herein, the two tables will be called `model2.nodes` and `model2.edges` to reflect the naming convention of the database. The shapes of `model2.nodes` `model2.edges` are (18110, 3) and (23531, 5), respectively. Access to these data tables is achieved by executing PostgreSQL queries in either the terminal, or in the code of the Python implementations of the algorithms.

4.2 Visualising with QGIS

QGIS (Quantum Geographic Information Systems) is a piece of open-source software for the purpose of editing, visualising, and analysing geospatial data. It is the application with which all of the visualisations for this project were produced. QGIS also integrates with other open-source technologies such as PostGIS (and therefore by extension PostgreSQL), making it a very useful implementation tool for this project. Two examples of the road network used in this study are shown in Fig. 4.3 and 4.4.

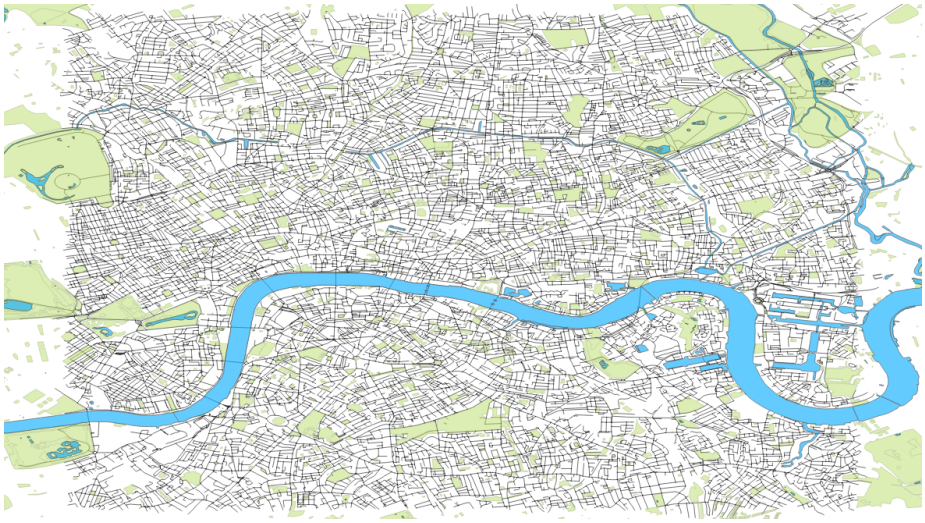


Figure 4.3: An example of the road network visualised in QGIS. Inclusion of water and green spaces for reference.

Visualisations such as those in Fig. 4.3 and 4.4 are made possible in QGIS by creating a connection to the PostgreSQL database where the geospatial data of the road network and its attributes are stored. Any output from the algorithms described in chapter 3 that alter the content of the database can be reflected instantaneously to the visualisations in QGIS with PostgreSQL query updates hard-coded into the Python executables.

4.3 Graph Optimisation with NetworkX

Central to the implementation of the algorithms described in the previous chapter is that of the graph data structure of the London road network and its attributes. The Python library NetworkX provides many methods and functions that make it suited to build such a graph structure. Converting attribute data from the database is achieved by leveraging a number of other Python libraries such as Psycopg2 to create a connection object to connect to the database, and GeoPandas to load the `model2.edges` table into data frame, and then to a NetworkX graph object.

NetworkX graph objects follow a *"dictionary of dictionaries"* system of storing edge and node attributes, as well as connections between adjacent nodes.

Figure 4.4: Close-up of the same rendering in Fig. 4.3. Nodes are shown in red.

This allows for efficient access to attribute data and potential scalability to a large number of nodes and edges. NetworkX also comes with built-in functions for common analytics of networks. These include a number of shortest path algorithm implementations. We are interested in a single source shortest path algorithm that truncates when a target node is found. For this, the `networkx.dijstra_path` function satisfies this requirement. The function takes as input a graph object, source node, target node, and a weight parameter. The function returns an ordered list of nodes from source to target from the shortest path (or in our case, since the weight parameter is set to $\gamma_{avg} \cdot d(u, v)$, the least polluted path).

The final stage of processing is to convert the list of nodes to a list of edges. This is so that the edges in the path can be visualised in QGIS. To do this, we look up the edge ID contained in the sub dictionary of the graph object when using two adjacent nodes as keys. A code snippet of how this was implemented is shown in Fig 4.5.

To visualise the path, an extra column was added to the `model2.edges` table called `inpath`, which takes the boolean values *t* or *f*. Looping through the ID list and using the PostgreSQL update query below the `optimisePath` function in Fig. 4.5, it is possible to change the value of the `inpath` value for each edge.

4.4 Historical Data

In order to implement adapted routes and RL as introduced in sections 1.2.2 and 1.2.3, it is necessary to have access to trajectory data of the graph representation of London. In this project, this is provided in the form of recorded exercise and commuting routes of users of the mobile fitness tracking application Strava [34]. A total of $\sim$ 1500 trajectories has been collected from a single user, and a further $\sim$ 100 from two other users.

```
def optimisePath(graph, start_v, end_v, minimise_param="distance"):

    # nodes in minimum cost path
    mincost_nodes = nx.dijkstra_path(graph, start_v, end_v,
                                     minimise_param)

    # convert to edge IDs
    optimum_path_edges = []
    for i in range(len(mincost_nodes)-1):
        edgeID = graph[mincost_nodes[i]][mincost_nodes[i+1]]["eid"]
        optimum_path_edges.append(edgeID)

    return(optimum_path_edges)


# query to update inpath edge attribute
update_query = "UPDATE model2.edges SET inpath=true WHERE eid=%s;"

# set inpath to true for all edges in optimal path
for eid in optimal_path:
    cur.execute(update_query, eid)

connection.commit()
```

Figure 4.5: Code snippet of function that generates a list of edges in the optimum path between two nodes. Lower portion details the update rule used to visulaise the path.

4.4.1 Processing Strava Data

Individual trajectories as recorded by the Strava app are saved as `.gpx` files. These files contain the longitude and latitude (lon/lat) coordinates, elevation, and time stamp of each coordinate. A quick examination of these files reveal that a coordinate is recorded every 2 - 3 seconds. A typical trajectory therefore contains several thousand coordinates, depending on the length of recording time. Parsing the `.gpx` files was handled by the `gpxpy.geo` library.

Since many trajectory files need to be analysed, two custom classes were defined. The first for an activity file containing a user's complete trajectories, and a second for a single trajectory. Included in each class are a number of methods to help standardise analysis of the files, for example returning the number of files as an iterator object, or the file names. For the *activity* class, custom methods include functions that automate the parsing of the raw `.gpx` file into a data frame of lon/lat values for each recorded coordinate. With the trajectories in this format, it is possible to perform map-matching to generate a sequence of visited edges for each trajectory.

4.4.2 Improved Map-Matching Algorithm

Algorithm 1 outlines a naïve approach to map-matching. Two issues were found with this algorithm. The first is its low efficiency, with time complexity $O(nl)$. The second is that assuming proximity of a trajectory coordinate is the best

indicator of current road segment. The risk of this assumption is illustrated in Fig. 4.6. The sequence of trajectory points clearly show a vehicle moving from left to right following the main road "Yesler way". However, Algorithm 1 would select edges that are not part of the true trajectory, namely p_b would be associated with "E Fir St". We need a map-matching algorithm that more closely matches the trace data, producing a smoothly connected sequence of edges.

Figure 4.6: Due to the spatial and temporal uncertainty in the trajectory coordinates (blue points and line), the selected edge for p_b is the vertical main road. This is clearly false when considering the points p_a and p_c [21].

The broad approach that we take is to first reduce the number of trajectory coordinates. The trajectories recorded using Strava are often of high resolution, measuring thousands of lon/lat points for a single route. Representing these trajectories with fewer points while still maintaining fidelity may be achieved with a curve simplifying algorithm. This would reduce the problem size when translating trajectories to edge sequences. The Ramer-Douglas-Peucker (RDM) algorithm is a curve simplifying algorithm. It takes a δ parameter which is the distance a point must be from the line connecting other points it is rejected. Initially, the two end points are chosen, and a line is drawn between them. For perpendicular distances less than δ, any unmarked points are discarded. Otherwise, points must be marked and the algorithm is repeated for segments between marked points. Fig. 4.7 shows an example of how adjusting δ reduces the number of points in a trajectory.

δ	Number of points	% of original number of points
10^{-1}	2	0.22%
10^{-2}	3	0.33%
10^{-3}	13	1.4%
10^{-4}	54	6.0%
10^{-5}	398	44%
10^{-6}	819	91%
$10^{-\infty}$	897	100%

Figure 4.7: Number of points increasing with smaller values of δ. Original number of points when $\delta = 10^{-\infty} = 0$.

The second step is to find the nearest node to each of the coordinate points in the reduced trajectory. This is done using a similar method described in section 3.2.2, but this time limiting the search to a window centred on the trajectory coordinate. The selected nodes are unlikely to be adjacent to each other, but close enough to follow the broad shape of the original trajectory. To render the trajectory as a sequence of edges, the nodes are then joined together, in their sequence, with paths generated by Dijkstra's algorithm. The outline to this improved map-matching algorithm is detailed in algorithm 4. Its major improvement over algorithm 1 is that, because Dijkstra's algorithm is used to connect nodes, a single path is generated. This is a benefit when considering the RL formulation where we consider traces of the state space as a series of state-action pairs. It also more closely reflects the paths that a human user would naturally take.

Algorithm 4 Improved Map-Matching Algorithm

Require: Trajectory coordinates $c_i \in \{c_1, c_2, ..., c_l\}$
Require: Node coordinates of a graph $v \in V$
Ensure: Set of nodes T that match the trajectory
1: $\mathcal{C}_r \leftarrow \text{RDP}(\{c_1, c_2, ..., c_l\})$
2: $T \leftarrow \text{NaïveMM}(V, \mathcal{C}_r)$
3: **for all** $v \in T_r$ **do**
4: $\quad s \leftarrow \text{Dijkstra}(v, v', \text{“distance”})$
5: $\quad T \leftarrow T \cup \{s\}$
6: **end for**
7: **return** T

Walking through algorithm 4, the original trajectory coordinate set is reduced and closest nodes found with algorithm 1 (lines 1 and 2). Then, Dijkstra's algorithm is applied to the nodes to connect them with the shortest path. This new map-matching algorithm offers no theoretical improvements on run-time performance since the original algorithm is still applied. However, due to the significant reduction of the trajectory coordinates, the practical run-time is significantly improved.

There is, however, a compromise that needs to be made, which concerns the parameter δ. A high value for δ means the reduced number of coordinates becomes too low to faithfully reconstruct the trajectory. Take for example the extreme case of two coordinates. Algorithm 4 would generate an edge sequence identical to the shortest path rather than the true trajectory. A low δ would then suffer from the same problems as algorithm 1, as the number of coordinates would only be slightly reduced. Through experimentation with various δ, it was decided that the best balance between fidelity and smoothness is at a value of $\delta = 0.001$. An example of how robustly trajectories are matched to edges is shown in Fig. 4.8. Note how that, even though the trajectory veers towards nodes and edges that are clearly not part of the path, the algorithm does not include them.

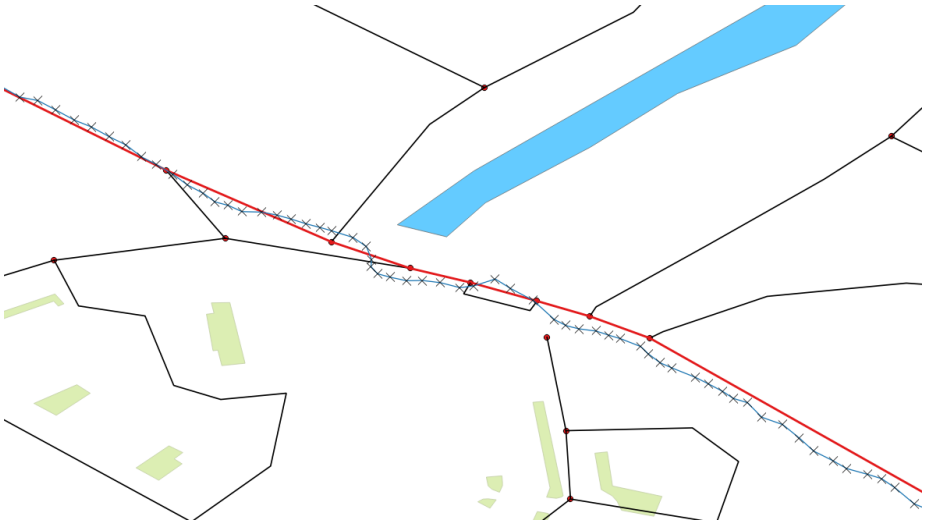


Figure 4.8: Map-matching with algorithm 4. The original trajectory, with a full compliment of coordinates (before application of RDM), is shown by the blue line, and the matched edges shown in red.

4.5 Adding Popularity Scaling Factor

With the trajectories converted to lists of edge-node pairs, it is trivial to count the visitations of each edge over a user's complete history. These edge counts are stored as a dictionary with its *key:value* pairs set to *edgeID:count*. Using either equation 3.4 or 3.5 converts the raw edge count to a scaling factor. In order to incorporate the popularity scaling factor into the overall edge weight, we must create a new column in the `model2.edges` table stored in the database. This can be achieved with the following PostgreSQL update query in the terminal window (after having navigated to the database):

```
1 ALTER TABLE model2.edges
2 ADD COLUMN weight_scaling NUMERIC;
```

This new `weight_scaling` column will now appear as edge data when the graph object is created. Calculating the optimum path is conducted in the same manner shown in Fig. 4.5, only this time the optimising parameter is the product $w_e = d(u, v)\gamma_{\text{avg}} f_e$.

4.6 Feature Engineering

As part of the IRL formulation detailed in section 3.5, we are required to associate various features with the state-action pairs of the environment. These features should reflect some aspect of the state-action pair that makes it more or less desirable for an agent to choose. We already have two such features; from `model2.edges` in Fig. 4.2 we could used distance and γ as features. There are many more we could include, some of which will be detailed in this section.

Furthermore, the features must be discretised, limiting the values a feature can take to either 1 or 0, meaning the feature is present or not, respectively.

4.6.1 Feature Discretisation

The gamma attribute will be taken as an example to demonstrate how discretisation is achieved. Firstly, multiple new columns need to be added to `model2.edges` to accommodate the binary values. The number of new columns is equal to the number of bins we decide to split the data into. 10 bins were chosen at equal quantiles, meaning each bin has the same frequency, but do not have equal width. This number of bins strikes a balance between being numerous enough to describe the distribution of gamma values, but without being too cumbersome to add into the table. Indeed, adding these extra columns follows the same method as described for adding the weight_scaling column; execute an appropriate PostgreSQL query to update the table.

With the new columns added, we must populate them with either a 1 or 0, depending on what the value of gamma is in a particular row. To do this, further PostgreSQL queries need to be executed, one for each new column. An example of such an update query is shown below. The naming convention for the column headings is that g006_010 means a gamma value of $6 < \gamma \leq 10$. If the condition is true, the column is set to 1, and 0 otherwise. The same process can be applied to the distance column, generating a further 10 columns representing the atomic features of road distance.

```
1 UPDATE model2.edges
2 SET g006_010 = CASE
3 WHEN gamma > 6 AND gamma <= 10
4 THEN 1
5 ELSE 0
6 END;
```

4.6.2 Proximity to Water

When considering appropriate features to include in the edge data, one must think about which features of a road would be the deciding factor between its selection in an exercise route, or commuting route, and it being disregarded. From domain knowledge, the proximity one is to bodies of water could be such a deciding factor. It is intuitive to think that users would prefer to be closer to water while travelling, and therefore attempt to include such high water-proximity roads in their routes.

The shape files for bodies of water in London were obtained from the London Data Store [33]. The distance measure we will be using is the shortest distance from the midpoint of a road segment to the perimeter of a water polygon. Achieving this requires the creation of a data layer "mask" were each pixel is assigned a value for the shortest distance to a water polygon edge measured in metres. Then, using the point sampling tool, it is possible to lift the pixel values from the mask were the midpoint of the road segments are. Doing this leaves each road segment with a new water proximity attribute. The proximity to water of each road midpoint is shown in Fig. 4.9. As is the same with all features, water proximity is then discretised in the same fashion as γ and

distance; binning the data into 10 equal quantiles using PostgreSQL update queries.

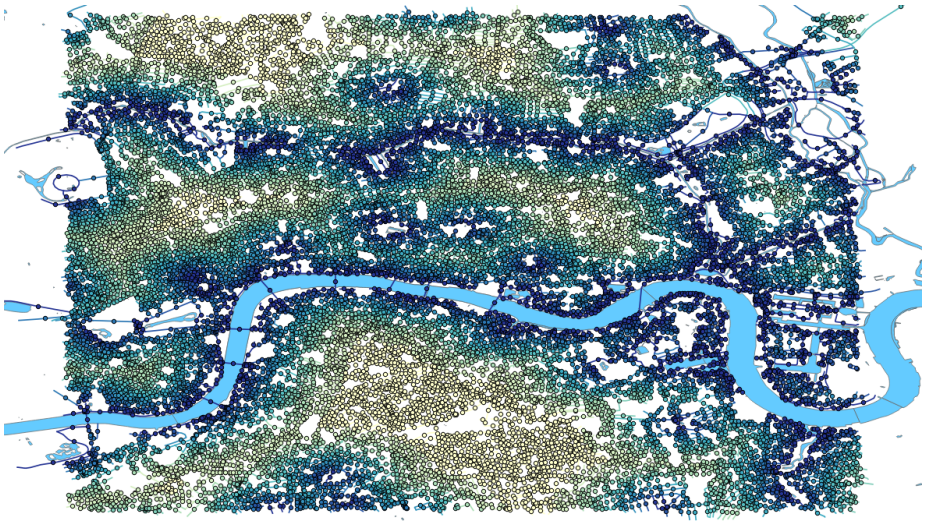


Figure 4.9: Map showing proximity of each road segment's midpoint to the edge of water polygons. Deeper blue colours indicate high proximity, while light yellow colours indicate low proximity.

4.6.3 Proximity to Green Spaces

Further to the consideration of water proximity, we also include proximity to green spaces as edge features. As with water proximity, it is intuitive to think that green space proximity would be a deciding factor when selecting a route through London, wanting to travel nearby a park or green areas, or even travelling within them.

The shape files for all public and privately owned green spaces in the London area were obtained by extracting the layer from Open Street Map of London [28]. The process by which we attribute the distance from a green space polygon edge to the midpoint of a road segment follows identically the process applied to the water polygons. The proximity to green spaces of each road midpoint is shown in Fig. 4.10. It is clear to see that, compared to water, green spaces are distributed more evenly over the map of London, indicated by the higher spatial frequency of colours.

A further feature regarding green space is also considered, that is the single binary feature indicating whether a road lies within a green space or outside it. This feature essentially labels a new road type; paths in parks. Its inclusion is to account for the fact that roads deep in the interior of a green space may have lower proximity to the green space than roads running parallel to its perimeter, despite being as close as one could possibly be to the green space. QGIS makes this possible by highlighting roads that fall within a green space polygon using the "intersect" tool.

Figure 4.10: Map showing proximity of each road segment's midpoint to the edge of green space polygons. Deeper green colours indicate high proximity, while light yellow colours indicate low proximity.

4.6.4 Road Type

The type of road may be a strong factor to someone travelling through London. For example, commuters on bicycles may prefer roads with dedicated bicycle lanes, or roads with lighter traffic. Perhaps the opposite is true; A-roads, while carrying heavier traffic, present the most direct path to a destination. Fig. 4.11 shows a map of the colour-coded road types of London.

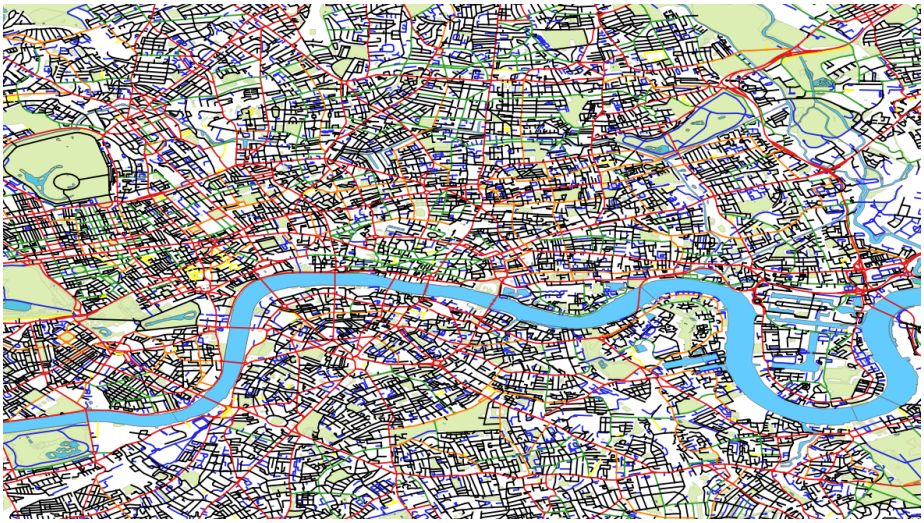


Figure 4.11: Distribution of road types in London.

The data for the different road types were present in the original shape file from the Ordnance Survey download [35]. In total, 15 road types are included, for example A-road, duel carriageway etc. A road is also permitted to be more than one type, for example a road could be both a roundabout and a B-road.

4.7 Calculating Feature Weights α_i

Engineering the features of every road segment in the London road network involved added an extra column to the `model2.edges` data table. When brought into the main algorithm from the data base, this table is used to create the atomic basis vectors ϕ first introduced in section 3.5.1. These basis vectors are effectively selectors for the feature weights α_i. Summing the basis vectors, with their corresponding weights, gives each action (road segment) an associated reward. The formulation used for the calculation α_i is given in equation 3.32 as $\alpha = \kappa\mu$. However, for the implementation in Python, it is more natural to think of this in vector form:

$$\boldsymbol{\alpha} = \boldsymbol{\kappa} \circ \boldsymbol{\mu} \tag{4.1}$$

Each component of the above equation is a column vector of shape (d, 1), where $d = 57$ is the number of features. We first focus on the calculation of the feature expectation vector $\boldsymbol{\mu}$. Using the map-matching algorithm described in section 4.4.2, the trace data from Strava is converted to lists of state-action pairs, referenced by their IDs. Looping through these trajectories, the IDs of actions are used to select rows from the feature matrix, a matrix of shape (23531, 57) with each row representing an action, and each column a feature. The selected rows are then transposed, summed, and normalised to give $\boldsymbol{\mu}$.

The self-information adjustment term $\boldsymbol{\kappa}$ is calculated independently from the trace data. Equation 3.35 requires us to calculate the term $|\{a \in \mathcal{A} : \phi \in a\}|$, which is a count of the number of actions in the state space that has a particular feature. This can be easily determined by summing the columns of the feature matrix, which produces a vector of feature counts over the whole state space. The elements from this vector can be used to determine the elements of $\boldsymbol{\kappa}$. For optimal execution times, all vectors and matrices are implemented as Python NumPy arrays.

4.8 The Environment and Q-Table

There are two central data structures used for the RL treatment; the environment (or `env` for short) and the Q-table. Both are matrices of shape (18110, 23531) with rows representing states and columns representing actions. The difference between them is, while the Q-table is essentially the action-value function $Q(s, a)$ where each element is the Q value of taking action a in state s, the `env` matrix's elements are tuples. The tuples are (s', r); the subsequent state s' after having taken action a in state s, and the reward r associated with that sequence. The Q-table is dynamic, with its elements being updated through every iteration of the Q-Learning algorithm, while `env` is static, used primarily to look up rewards and subsequent states.

4.9 Setting a Reward Gradient

So far, we have explored the reward associated with taking a particular action in a particular state, that is to say, selecting a particular road at an intersection. This is borne of the combination of different features of the road, and their relative importance. But, there is a further reward layer that has yet to be considered. That is the reward associated with being closer to the target state s_τ. We would expect the rewards to increase as an agent approaches s_τ. Indeed, this reward *gradient* is necessary to drive the agent toward the target to complete an episode. Applying no reward gradient would condemn the agent to continuously explore the environment, making greedy decisions at each state, only arriving at s_τ by happenstance. We do allow some exploration at the beginning of the ϵ-greedy algorithm. But, whenever an action is chosen greedily, it should move the agent closer to its goal.

The requirements for a reward gradient is that it is steep enough to drive the agent toward the target state, but not too strong that it overrides the effect of the original rewards defined from the dot product of the basis vectors and feature weights $\mathcal{R}(a) = \boldsymbol{\alpha} \cdot \boldsymbol{\phi}(a)$. Careful experimentation will be carried out to determine the optimal gradient function that achieves this behaviour.

It was mentioned in section 3.3.3 that the Q-table would be initialised with the reward function $\mathcal{R}$. We can now see why this is necessary in order to help encourage the agent toward the terminal state s_τ from the very first episode, rather than exploring randomly. The large state space means that this random walk may carry on for a prohibitively long time, rendering the algorithm inefficient.

Chapter 5

Results & Analysis

We have explored the theoretical basis of three approaches the problems set out in chapter 1. We have also seen how these approaches can be implemented. We now wish to present the results from these implementations. That is the subject of this chapter.

5.1 Least Polluted Routes

The first problem we set out to solve was to be able to generate routes through the London road network while minimising exposure to air pollution. To analyse the effect of changing the optimisation parameter, an example of the shortest path between two nodes is shown in Fig. 5.1. For reference, the air pollution grid is superimposed over the road network. Darker colours indicate high concentrations of air pollution, whereas lighter colours indicate low concentrations. The lightest coloured grid cells have $\gamma = 0$.

The path starts from the bottom of the figure, moves north following the shortest and most direct path, and terminates near the top. There is clearly no consideration for the level of pollution of the road segments; there are two areas of high air pollution just before and after the bridge section that the shortest path intersects.

Fig. 5.2 shows how the path is altered when it is not the distance that is used as the optimisation parameter, but the pollution penalty $p = \gamma_{\text{avg}} d$, instead. The source and target nodes are exactly the same as in Fig. 5.1; moving from bottom to top[1]. It is clear to see that the optimum path avoids areas of high air pollution. The two areas highlighted earlier either side of the bridge are conspicuously avoided by the optimum path.

[1] Although, the reverse would produce exactly the same path as the pollution penalty is direction independent.

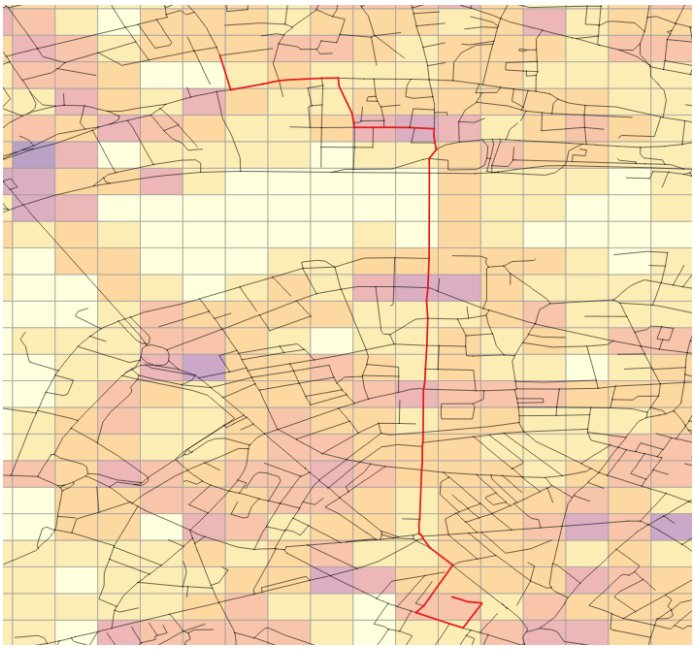


Figure 5.1: Shortest path generated using Dijkstra's algorithm.

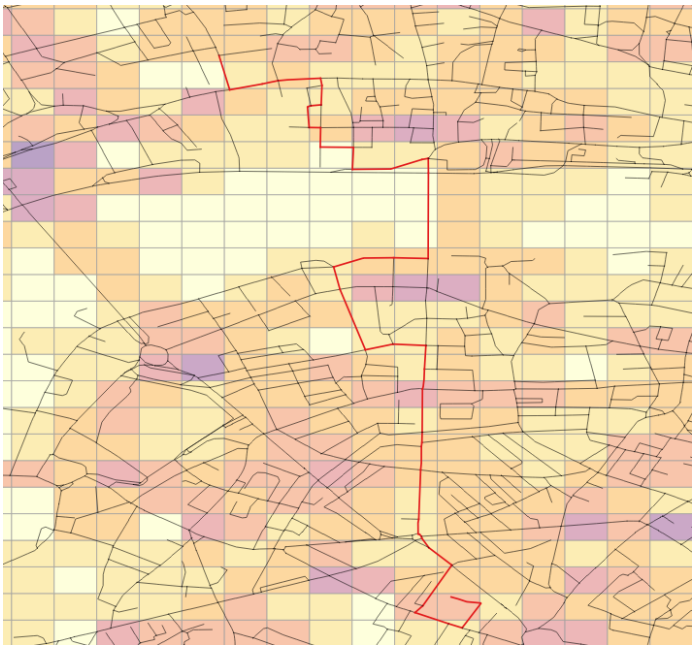


Figure 5.2: Least polluted path generated using Dijkstra's algorithm.

The optimum path will not avoid high pollution areas *at all costs*. Because the pollution penalty is a function of γ and distance, it may be advantageous to pass through a high-pollution area if it is for a short distance only. The converse is also true; a path will not take an inordinately long detour to intersect grid cells of lower air pollution if it means travelling for a long enough distance to be exposed to more pollution than a short, high pollution route.

5.2 Road Popularity Adaptation

One noticeable effect from the routes generated in the previous section, an effect that was anticipated, is that to avoid areas of high air pollution, the route takes large detours. For an individual using this system to plan, for example, a commuting route from home to work, the extra distance is highly undesirable. Artificially lowering the edge weights with road-popularity scores would divert the optimum route to roads that are more preferable.

First, the historic data collected by the mobile exercise application Strava was processed using algorithm 4 to produce counts for the number of times a road segment is traversed in the trace data. The results from implementing this is shown in Fig. 5.3. A single individual's exercise data can be seen, with thicker lines indicating a higher traversal count.

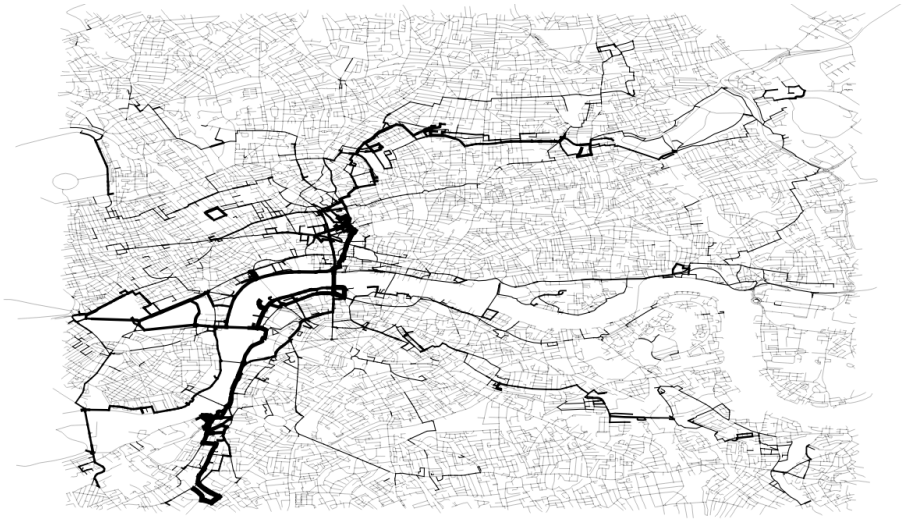


Figure 5.3: Road map showing the most popular routes from a single user's ~ 1500 trajectories. Thicker lines indicate a higher traversal count.

The map in Fig. 5.3 gives an indication of which roads are more popular geographically, but doesn't help us understand the distribution of that popularity. The thickness of the roads goes someway to revealing this distribution, but is difficult to interpret and may be misleading. To understand how popularity is distributed throughout the road network a log-log plot of traversal counts and the *rank* of the road[2]. This plot if shown in Fig. 5.4.

[2]i.e. the road with the highest counts is given a rank $R = 1$, second highest counts given

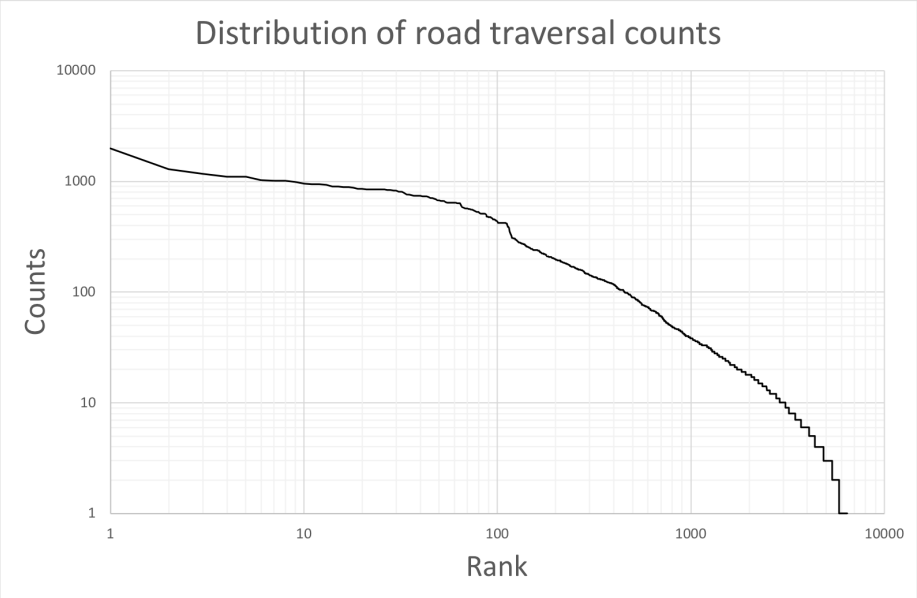


Figure 5.4: Distribution of road traversal count as a function of road rank.

We can see that the distribution of traversal counts follows approximately a power law of the form $f(r) \propto r^{-m}$ where $\alpha \approx 1$ is a scale factor. We know this because of the (approximate) negatively sloped straight line on the log-log plot. The gradient of the slope is given by $-m$. The interpretation of a power law in this case is thus: very few roads are traversed a many times, and many roads are traversed only a few times.

Once the counts for each road is established, the scaling factor can then be calculated using either equation 3.4 or 3.5. To demonstrate the effect of including a popularity weight, 50, 100, and 200 trajectories from the trace data (of the same user whose data is shown in Fig. 5.4) were used to calculate popularity weights. Then, two points, a source node and target node, were chosen and an optimal path generated between the two using this newly weighted edges. The results of this experiment can be see in Fig. 5.5, 5.6, and 5.7, referring to the 50, 100, and 200 trajectory cases, respectively. All three cases use the exponential formula in equation 3.4 to set the weight scaling factor.

For the 50 trajectory case, the optimum route follows closely that which would have been generated with no popularity weighting. Starting south of the river Themes moving north, the route moves through areas with the lowest air pollution values, as indicated by the lighter colour of the grid cells. There is a slight deviation as the route moves along the south bank that we wouldn't expect if the route were purely following the least polluted roads. This can therefore be attributed to the effect of the popularity weight of the road lowering the pollution penalty, making its selection more advantageous.

rank $R = 2$ etc.

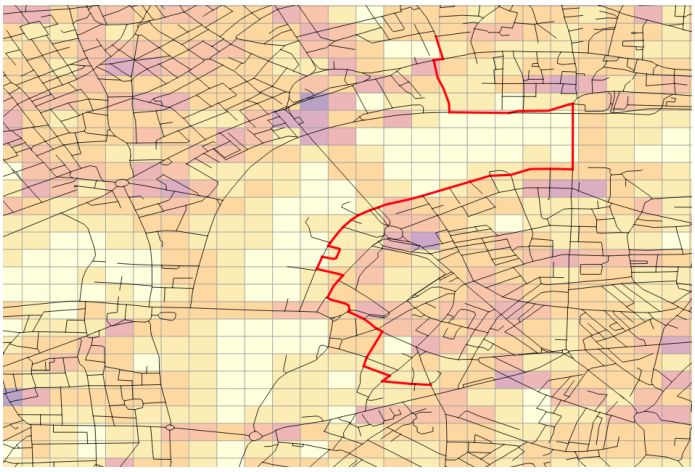


Figure 5.5: Optimum path with adapted edge weights from 50 trajectories.

As the number of trajectories increase, the weight scaling factor decreases exponentially. Eventually, this overrides the effect of the initial pollution penalties, leading the optimum path to more closely resemble a route that the user would likely take, ignoring the need to minimise air pollution exposure. This effect can be seen to take place as early as 100 trajectories in Fig. 5.6. The optimum path passes through many grid cells that have high air pollution, due to them being included in many previous trajectories from the user's historic data.

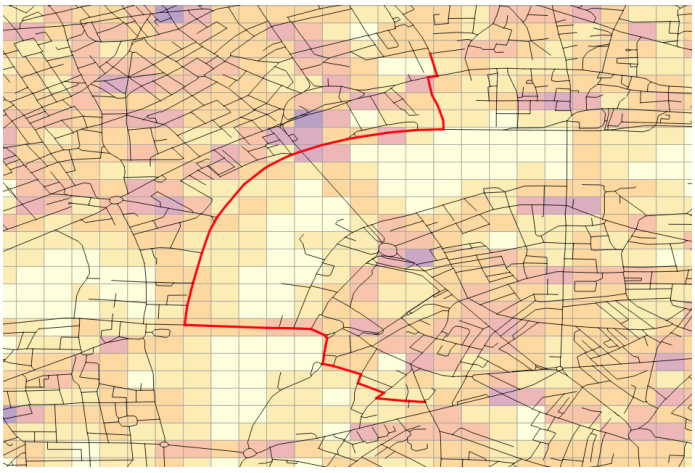


Figure 5.6: Optimum path with adapted edge weights from 100 trajectories.

Adding a further 100 trajectories to reach 200 overall, the optimum path further diverts from the lowest polluted path. So much so in fact that initially the calculated route moves *away* from the target node in order to reach edges that have lower weights. This route would be very costly if only distance or air pollution were taken into account. But, because of the overriding effect of

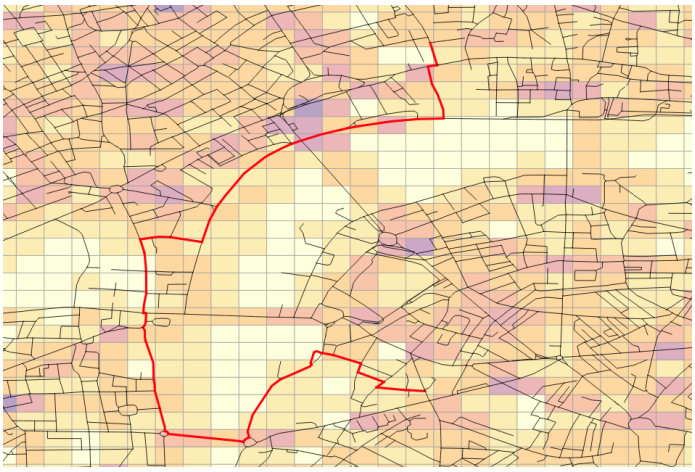


Figure 5.7: Optimum path with adapted edge weights from 200 trajectories.

the popularity weights, Dijkstra's algorithm selects these in the optimum path between source and target nodes.

5.3 Reinforcement Learning Approach

The section explores the results following the RL approach described in section 3.3 and 3.5. In order to do this we introduce two further sets of trace data. Already, the trace data from $\sim$ 1500 trajectories has been presented and analysed. We shall call this user "User A". After careful examination of the trajectories, their length, speed, frequency etc. User A can be described as an avid amateur mid-distance runner, if not verging on semi-professional. User B is also a runner, but trains much less seriously than User A. User C is a cyclist who principally commutes to and from work, taking an identical (or nearly identical) route each time. While we have access to $\sim$ 1500 trajectories for User A, there are only 28 and 79 trajectories available for User B and C, respectively.

5.3.1 Comparing Feature Weights $\boldsymbol{\alpha}$

There feature weights $\boldsymbol{\alpha}$ were calculated for each user following the method described in section 4.7. A total of 57 features are included in the edge data of the London road network; 10 bins each for road length, pollution level γ_{avg}, water proximity, and green-space proximity, and 17 road types. The continuous variables of the first 4 feature sets are binned by 10 quantiles, such that the first quantile represents values of the variable between which represents 10% of the total data, the second quantile represents the next 10% of the data, and so on. In this sense, the bins are actually representing 10th, 20th etc. *percentiles* of the data.

Distance and Pollution Levels

The first two feature sets to explore are those that were in the original graph structure. Fig. 5.8 and 5.9 show the feature weights α of all three users for varying road lengths and pollution levels. α has been normalised to $0 \leq \alpha \leq 1$ so that easy comparisons can be made between not only the three users, but between different features and feature sets. The interpretation that follows from normalising the rewards weight vector $\boldsymbol{\alpha}$ is that the most *important* feature for a particular user is given the value 1, and every other feature is scaled to this most important one. For example, a feature with $\alpha = 0.5$ is half as important as the most important feature, and will yield half the reward.

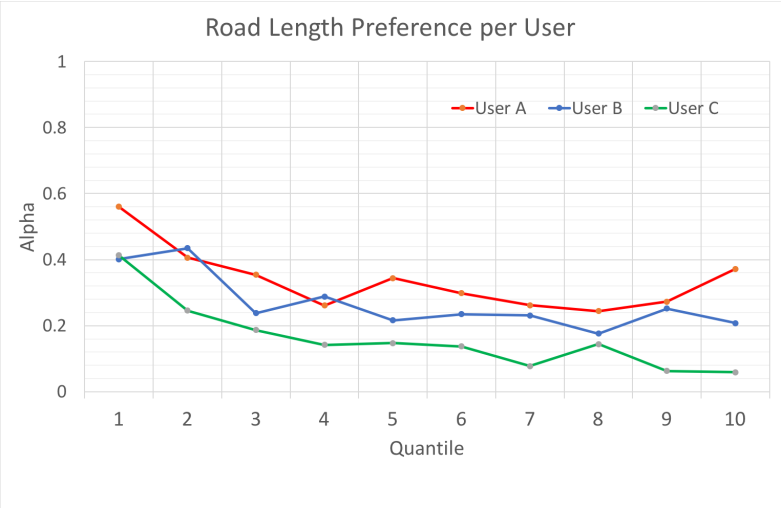


Figure 5.8: Changing α as road segments become longer.

Two general patterns emerge from examination of the graphs in Fig. 5.8 and 5.9. All users have a weak preference for shorter roads, and a weak preference for highly polluted roads. Discrepancies to this trend are User A's tendency to also prefer less polluted roads and shorter roads, too. Another general trend is that User A has the highest overall preference with respect to his or her most preferred feature, User B second, and User C third.

Given the nature of the feature sets, it is difficult to conclude that users would be continuously selecting exercise routes or commuting routes based on the length of individual roads. The most probable explanation of a slight preference for shorter roads is that where the historic routes were recorded there is an abundance of short roads (like city centres). Likewise for air pollution; few users actively try to limit their air pollution exposure, it is more a consequence of following routes they prefer which happen to be in highly polluted areas. For instance, User A has a higher preference for roads in low polluted areas than the other two users. When considering Fig. 5.3 much of the trajectories follow the riverside where air pollution is lower, in general. User C's lower preference overall is a reflection of his or her higher preference for other features such as road type. For a user commuting on bicycle with a fixed route, desirable features are those that allow the user to reach the destination in the least time.

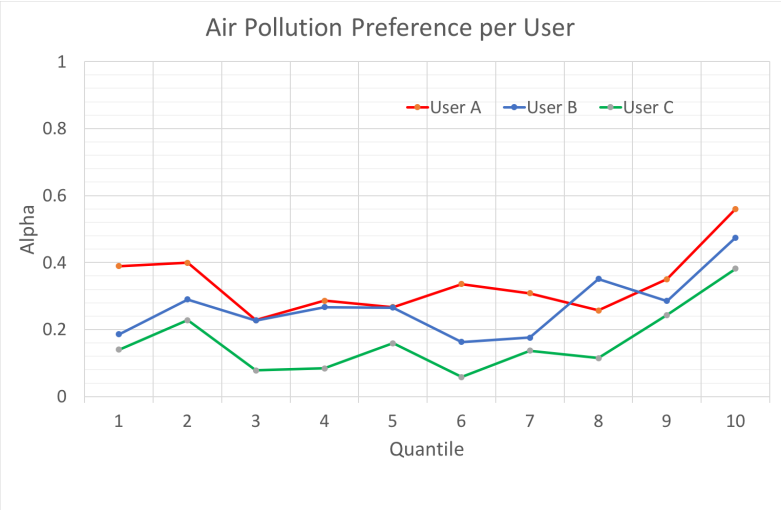


Figure 5.9: Changing α as road segments have greater level of air pollution.

Water and Green Proximity

The second feature set concerns the distance a road is from either bodies of water or green spaces. As with road length and pollution level, the continuous variable of proximity is discretised into 10 quantiles. Fig. 5.10 and 5.11 show the weightings for each user for these features.

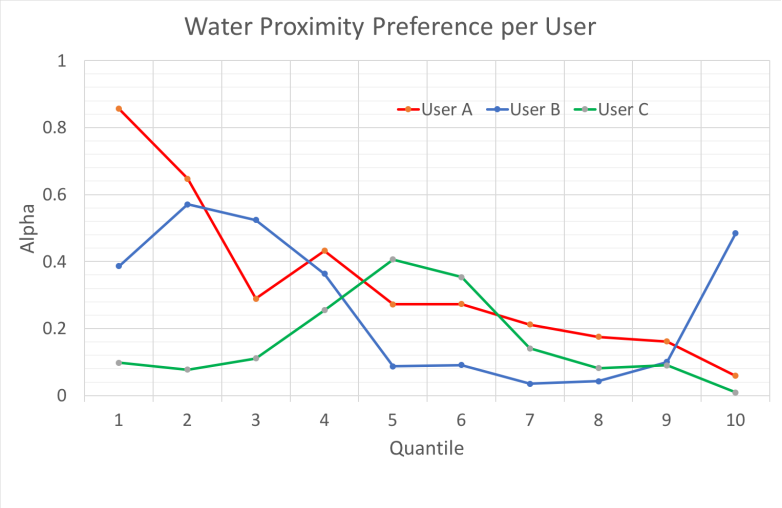


Figure 5.10: Changing α as road segments move further away from bodies of water.

There is a lot of variation between the users for the water proximity feature weight. User A has a clear preference for roads that are close to water, starting

at $\alpha > 0.8$ for the first quantile falling to near 0 for the last quantile. User B has a slightly lower preference for roads close to water, and also for roads furthest from water. Roads that are a median-average distance from water are most desirable for User C.

Since User A and B are both runners, it is unsurprising that their preference for roads close to water is high. Dominated by the river Themes, London's waterways are attractive areas for leisure and exercise, as is evidenced from the thicker lines near the water's edge in Fig. 5.3 for User A. It is unusual to see a high α for roads in the 10th quantile for water proximity for User B. This could be because of a number of trajectories being limited to start in areas away from water, before moving closer to water. This limitation of geography could also explain why User C prefers median distances from water. It is not that these roads are the desired distance from water, but rather that the fixed commuting route happens to move through roads with that particular feature.

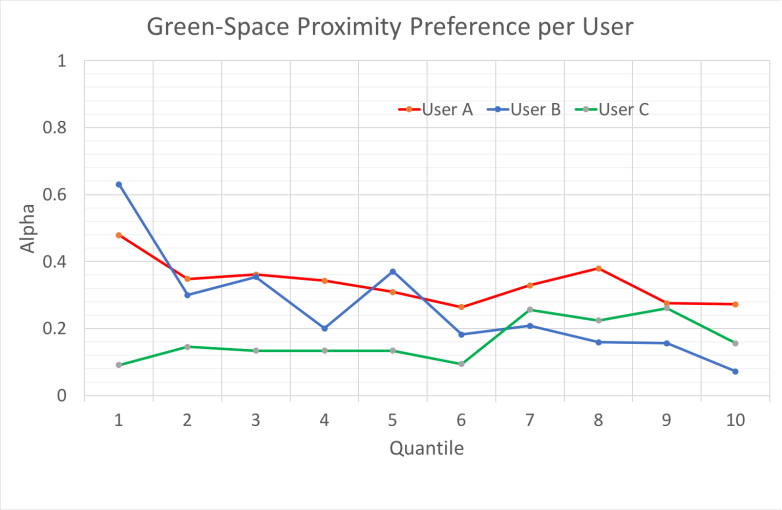


Figure 5.11: Changing α as road segments move further away from green spaces.

The variation between users for green-space proximity is also noticeable, but not so pronounced. The two runners have a preference for roads close to green spaces, which gradually declines as roads move further away. The decline is more noticeable for User B than User A, but the variation in User B's preference is greater. User C has a lower preference overall, and even a slight increase in preference as roads move away from green spaces.

It is reasonable to think that runners would prefer to run in green areas. While this preference is shown in the high α values at low quartiles, the degree to which this is true remains low. This could be due to the fairly even distribution of green spaces spread over London (see Fig. 4.10), so by moving only a short distance from a green space the runner might be in the 7th quantile already. Similarly, without intending, proximity to green spaces is achieved quite easily due to their ubiquity in London. This would lead to a more even preference through the 10 quantiles.

Road Type

The final set of features is the classification of each road. Unlike the previous two features sets, road type is not a continuous variable that has been binned into 10 quantiles, but rather a categorical variable. For this reason, the feature weights are displayed as a bar chart in Fig. 5.12.

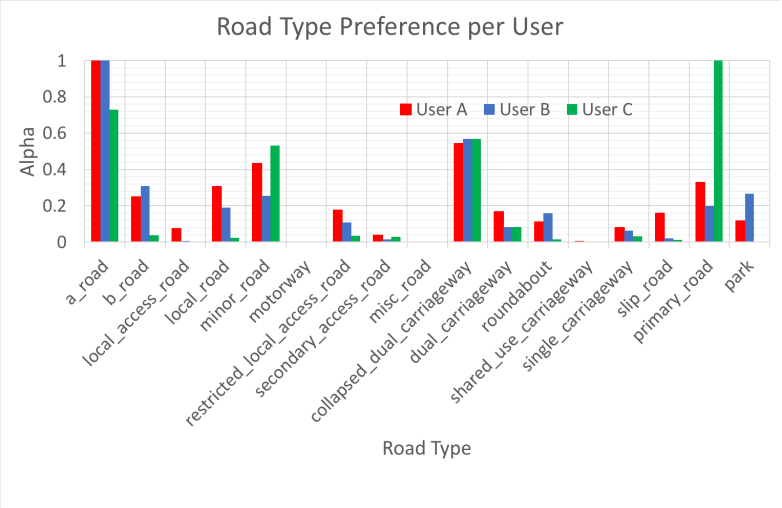


Figure 5.12: Value of α for different road types.

For the two runners the most preferred road type, and the most preferred feature, is A road. User C has a high preference for A roads too, but it is not the highest. The preferred road type is primary road. This is in stark contrast to the other two users, who attribute only a low importance to primary roads. Presumably this is because, as a commuter travelling by bicycle, following roads with this particular feature is more direct than only following A roads. All users have $\alpha = 0$ for both motorways and miscellaneous roads. This is simply because these features did not appear at all in any of the trajectories from the trace data. Though they are subtle differences, User C is less likely to take B roads, local roads, roads through parks.

5.3.2 Comparing Reward Functions $\mathcal{R}$

It is now possible to calculate the total reward for each road segment in the London road network. This is simply the linear combination of feature weights multiplied their basis vectors (see equation 3.25). Knowing the rewards for every road is the same as knowing the reward function $\mathcal{R}$ for each user. We can visualise this by colouring each road segment based on the magnitude of the reward associated with a road. Fig. 5.13, 5.14, and 5.15 show the reward function for User A, B, and C, respectively. The colour scheme was selected such that roads with larger rewards are more red, and roads with lower rewards are more blue.

User A

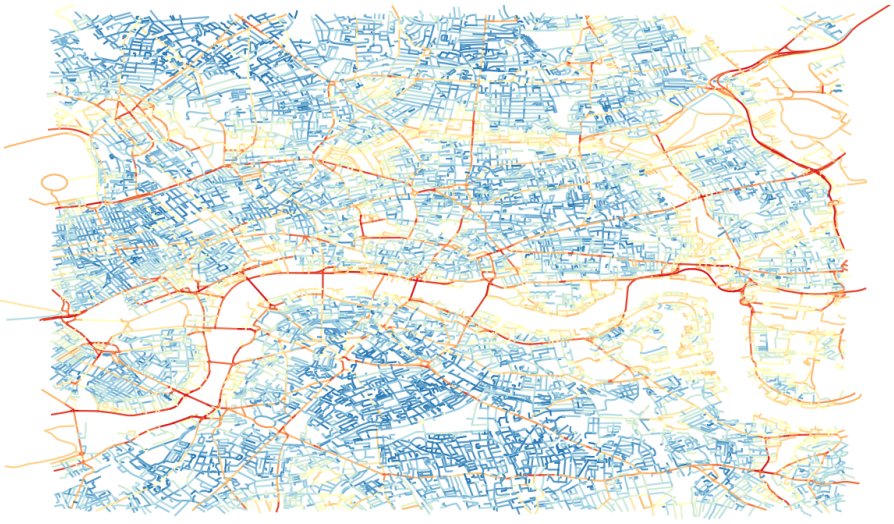


Figure 5.13: Reward function $\mathcal{R}$ for User A.

For User A, we can see the roads with the highest rewards are focused around bodies of water, particularly the river Themes running through the middle of the road map. It is also noticeable that A roads are amongst roads that receive the highest rewards. The utility of such a map is that we can understand the importance, or desirability, of a road even if it has not been included in previous trajectories. Referring to the trace data for User A in Fig. 5.3 and 5.4, only $\sim 25\%$ of all edges have been traversed.

User B

User B, being a runner also, shows much the same preference as User A; the streaks of red highlighting A roads are present, as well as reds and yellows surrounding water. The difference with Fig. 5.13 is that this preference for waterways is less pronounced, with colours noticeably closer to yellow than strong red. In addition, where there are "cold spots" for User A, User B's reward function fills these in with higher rewards. This is most conspicuous in the top left corner and lower middle of the road map.

User C

The reward function for User C is the most dissimilar to the other two, owing to the fact that the trajectories from which the feature weights were calculated were very homogeneous. Much of the roads in the network receive a small reward, reflected in the abundance of blue in Fig. 5.15. Roads that are close to green spaces and waterways both have low rewards. The only roads with detectably high rewards are primary roads, the skeleton of which can be seen in red spanning the road network.

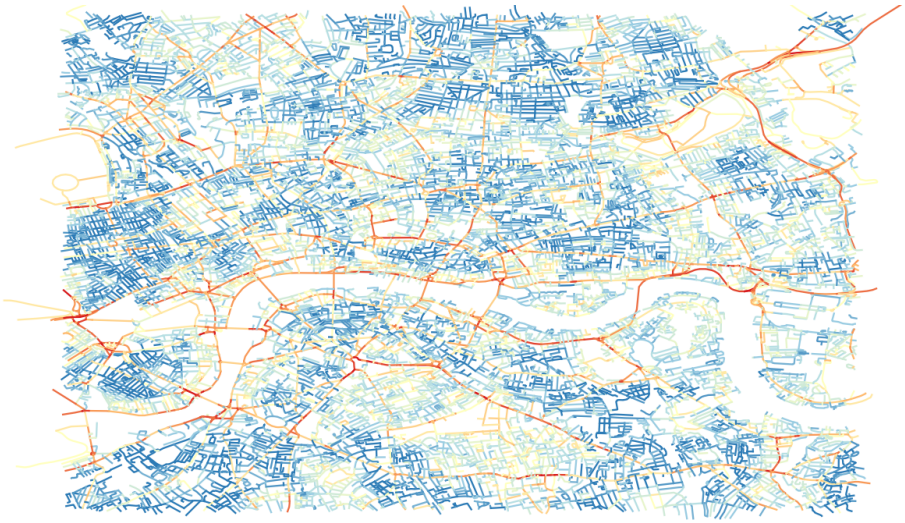


Figure 5.14: Reward function $\mathcal{R}$ for User B.

Figure 5.15: Reward function $\mathcal{R}$ for User C.

5.3.3 Applying a Reward Gradient

The final aim of this project is to generate routes using the bespoke reward functions described in the previous section. This is done by applying algorithm 3 with a start and end node. It was mentioned in section 4.9 that a reward gradient would be applied to the reward function to provide a driver for the agent as it conducts exploratory searches through the episodes of the algorithm. The gradient takes the form of a multiplicative factor applied to each reward, the magnitude of which is determined by a road's distance from the target node.

The gradient factor is defined as:

$$r_g = \begin{cases} 1 - \frac{d}{t} & \text{for } d \leq t \\ 0 & \text{for } d > t \end{cases} \tag{5.1}$$

where d is the distance a road is from s_τ and t is a threshold. The threshold is set as the shortest distance between s_σ and s_τ. The effect of this is that roads outside a radius $< t$ are given a negative reward, deterring the agent from selecting these roads. As roads draw closer to the target node, the gradient factor linearly increases to 1. The reward gradient must be steep enough to drive the agent toward the target state, but not too strong that it overrides the effect of the original reward function. The effect of applying a gradient to User A's reward function can be seen in Fig. 5.16. Notice that the underlying reward function is visible within a radius of about 3000m from the target node with a gentle gradient towards it. Outside this radius the rewards for each road become negative (shown in blue for clarity).

Figure 5.16: Combination of User A's reward function $\mathcal{R}$ and a reward gradient from equation 5.2 with a threshold $t = 3000$.

5.3.4 Troubleshooting

The performance of the algorithm depends on a number of parameters. Initially, parameters were set as:

- Learning rate $\eta = 0.85$. This represents the percentage of the *next best action-state's* Q value.
- Discount $\gamma = 0.95$. This controls the *foresight* of the agent; how far ahead of the current action-state do future action states influence the current Q value.
- Initial $\epsilon = 0.5$. Starting with 50% exploration and 50% exploitation.

- Decay rate $\lambda = 0.999$. After each episode, ϵ is multiplied by λ to achieve a gradual shift to 100% exploitation.
- `num_episodes` = 5000. We terminate the algorithm when satisfied a suitably optimal solution is found.

Using the above parameters and the reward gradient defined by equation 5.2, a number of difficulties were found during testing.

Shallow Gradient

The time taken to complete each episode was very long, typically several minutes. When considering the algorithm terminates in 5000 episode this represented a critical weakness. Reducing the λ from $0.999 \rightarrow 0.7$ did not ameliorate the problem. This suggests that although the agent is acting greedily, and gets greedy quicker, the next most advantageous state-action pair does not drive it towards the goal. To remedy this issue, a steeper reward gradient was applied in an additive fashion rather than a multiplicative one. The form of the gradient is defined by the equation:

$$r_g = \begin{cases} r_\tau \left(1 - \exp\left(-\frac{t}{d}\right)\right) & \text{for } d > 0 \\ r_\tau & \text{for } d = 0 \end{cases} \tag{5.2}$$

where r_τ is the maximum reward associated with traversing a road that terminates at s_τ. The purpose of this form of reward gradient is to give a large reward to the final state-actions (or very close to it), and control the gradient with the t parameter. The exact value of t can be adjusted based on the distance between s_σ and s_τ. It was found that $t = 100$ and $r_\tau = 500$ with $d(s_\sigma, s_\tau) \approx 6000$m allowed the algorithm to execute 5000 episodes.

Greedy Cycles

Investigating the progress of the algorithm, the average reward of an episode and the number of actions it takes to reach s_τ were interrogated. A composite plot is shown in Fig. 5.17.

As one would expect, the average reward per episode steadily increases. In the early episodes the agent is exploring the state space, and so spends a long time in areas that do not necessarily yield high rewards. As episodes elapse, the agent switches to exploitative behaviour, selecting actions based on previously known high-reward-yeilding decisions. In the initial stages, it is also expected that the number of actions start high, then decrease as an optimum route is established. This is what is observed from the algorithm in Fig. 5.17 with a steep decline in actions reaching a plateau that lasts for ~ 2000 episodes. What is counterintuitive however is that after this plateau, the number of actions begin to increase.

This effect can be understood by considering the ever more exploitative behaviour in the latter stages of the algorithm. It might happen that the agent get stuck in a *greedy cycle* where the next best action-state is the one from whence it came. Reverting back to the previous action-state would therefore mean transitioning again to the same action-state as one time step ago. This would continue ad infinitum until a random action is taken that is not the

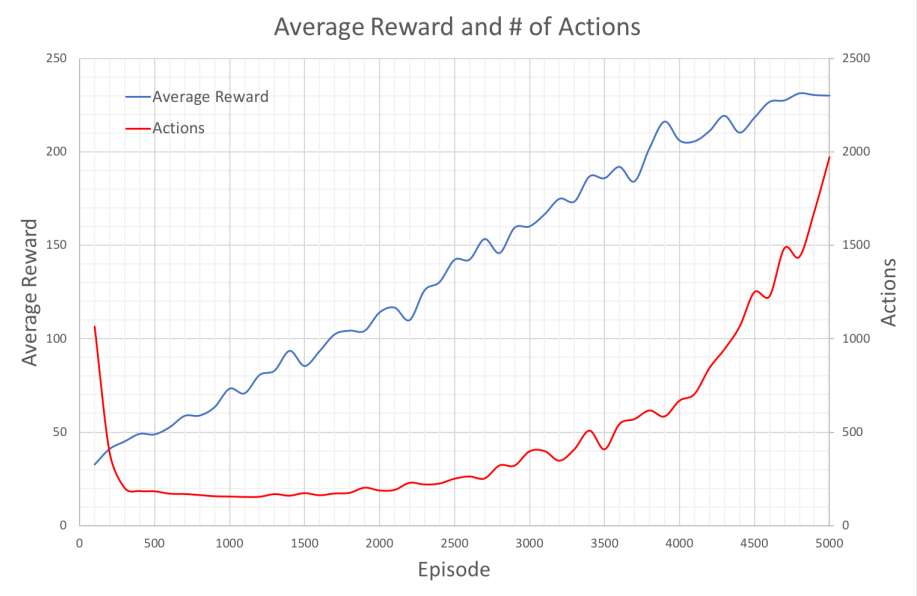


Figure 5.17: Composite plot of average reward and number of action to completion as the algorithm progresses. Both data series are 100-point averages of the raw outputs from the algorithm.

immediate best one. Being in the latter stages of the algorithm, the chances of a random action being taken gets ever more unlikely, hence the increasing actions required to complete an episode.

While the agent is caught in this cycle it accrues rewards, leading the average reward to increase in the process. Therefore, the average reward data series in Fig. 5.17 may not represent rewards gathered from ever more optimal routes, but rather routes with cycles that give the illusion of increasing optimality. A truer representation of an optimal route would be one where the average reward from uniquely selected action is highest. To show this, instead of counting all rewards from all transitions in an episode, rewards are counted only once and then averaged upon completion. If the agent has already selected an action, its reward does not contribute to the final average calculation. A plot of this over the course of 5000 episodes is shown in Fig. 5.18.

Removing greedy cycles has a noticeable effect. The average reward initially increases as greedy decisions are increasingly being made. As behaviour shifts to exploitation, rewards do not continue to increase but rather flatten off. This suggests that an optimal route is being reached.

5.3.5 Generating Optimum Routes

Optimal routes can be generating by acting greedily with respect to the best estimation of the action-value function Q. In the context of this project, theoretically the best estimation of Q is achieved after the 5000th episode. However, from Fig. 5.18 the optimal route still does not stabilise sufficiently for us to be sure the route from the final episode is the optimal. For this project, we take the

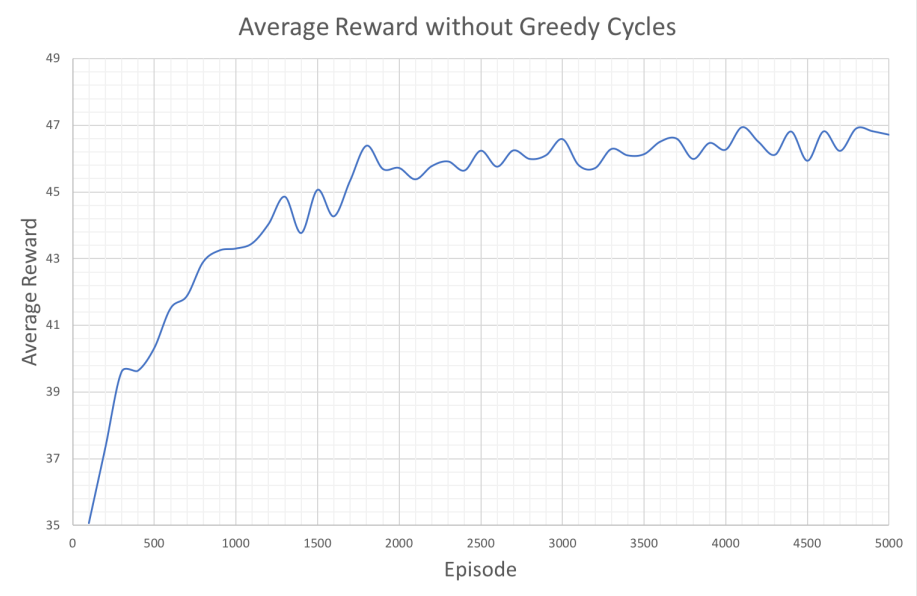


Figure 5.18: Average reward as algorithm progresses after having removed greedy cycles. Data series calculated from 100-point average of the raw outputs from algorithm.

optimal route as that which generates the highest average reward after removal of greedy cycles. Since this could occur just by chance during the exploration phase, only the final 10% of episodes are considered when selecting a route that yields the highest average reward, ensuring the agent is well in its exploitation phase. For comparison, three routes are shown in Fig. 5.19, two generated from Dijkstra's algorithm with respect to distance and air pollution, and another from algorithm 3 with the above parameters.

Starting from the lower-middle portion of the map, the shortest path in black naturally follows the most direct route to the target node in the upper-left corner. When the edge weights are changed to the pollution penalty the route takes a drastically different direction, seeking areas of lowest pollution. The RL-generated route takes a more direct route than that of the purple, but does not follow exactly the shortest path. Referring back to the reward function of User A in Fig. 5.13, we can see the route follows approximately the roads that yield the highest reward. What we do not see is the effect the reward gradient has on which roads are selected. As a further comparison, routes were generated using the trajectory data for all three users. This can be seen in Fig. 5.20.

The route for User A remains unchanged. The route for User B is almost identical to User C's route save for the initial diversion south west before moving north to rejoin. Taking into account the individual reward functions or each user, one could be convinced that optimal paths have been found; it appears the generated paths follow roads with high reward. In practice it is difficult to establish.

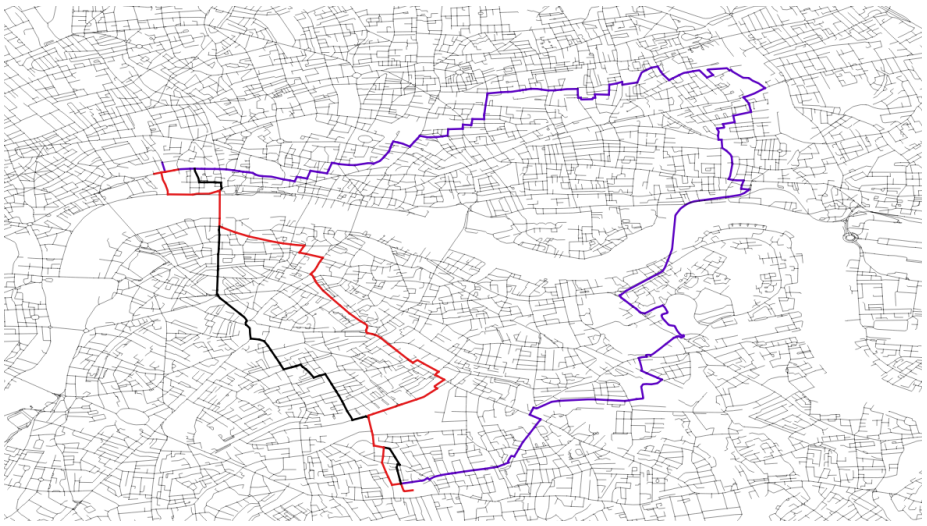


Figure 5.19: Three routes: black - Dijkstra's w.r.t. distance, purple - Dijkstra's w.r.t. pollution, red - algorithm 3 with User A's trajectory data.

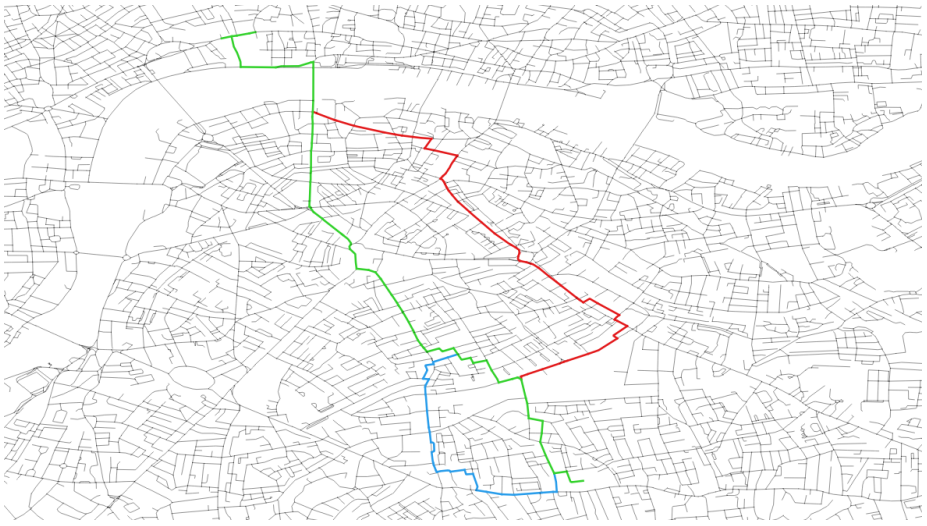


Figure 5.20: Three routes: red - algorithm 3 with User A's trajectory data, blue - User B, green - User C.

Chapter 6

Project Management

6.1 Initial Aims

The broad aims set out at the beginning of this project in February 2018 were largely kept intact throughout. The aims were to compute running and cycling routes in London that minimise exposure to air pollution. This was expressed as a graph optimisation problem where we sought a connected set of edges that minimise the total pollution penalty incurred upon traversal of these edges. In addition, these routes needed to be similar to routes previously completed by the user by employing techniques from the field of reinforcement learning. Further to this broad aim, further objectives of this project were listed:

1. Develop and improve the current algorithm that minimises air pollution along a path between two user-defined locations on a road map in London.
2. Develop and improve the current algorithm that minimises air pollution over a cycle of fixed length starting and ending at a user-defined location on a road map of London.
3. Prove that the unbound knapsack problem is NP-hard.
4. Develop a reinforcement learning model for generating trajectories through London that minimises air pollution and preserves similarity to previous trajectories.
5. Prove that iterative improvements of the value function and policy of a reinforcement learning model converge to optimality.
6. A web application with the above functionality will be launched with the help of developers at the Alan Turning Institute.

Two of the six objectives have been achieved; 1 and 4. Two other objectives (5 and 6) were dropped because, as the project progressed, it became apparent that these objectives would fall outside the main thrust of research. Although worthy pursuits of research, they would be better served with dedicated treatment. Objectives 2 and 3 were tackled in the undergraduate project of Patrick O'Hara.

6.2 Time Management

The initial schedule for this project can be seen in the Gantt chart in Appendix B. This project was broken down into three main phases, separated by a presentation in mid-April, and submission of an interim report at the start of July.

6.2.1 Phase One: 22nd February - 23rd April

This phase took a relatively relaxed approach to further research and understand existing code related to this project. In hindsight, this was a misjudgment. More time should have been spent on producing original code, procuring historic data, and developing original research interests. Instead, a large portion of time was spent on understanding theoretical concepts that did not directly drive research efforts. For example, much of this early stage was dedicated to tackling a related problem of finding least polluted cycles; the main thrust of the bachelor's thesis by Patrick O'Hara. This unnecessary pursuit was the result of a plan lacking in detail and direction.

Despite this poor start, at the end of this phase some original results were presented. Previously developed code was adapted to generate least polluted paths from start node to end node. The first incarnation of the map-matching algorithm (algorithm 1) was developed. In addition, the need to first calculate a reward function before applying and forward RL algorithm was proposed.

6.2.2 Phase Two: 23rd April - 25th June

Realising the need to make more significant progress, after delivering the presentation, the original action plan was changed. The amended Gantt chart is shown in Appendix C. From the beginning of May, it was decided to pursue two lines of research. The first concentrating on the simpler problem of incorporating historic exercise data as adaptive edge weights to the original pollution penalties. The second being the development of an IRL/RL method of generating optimal routes. The first approach was completed on time (end of task 7), culminating in a presentation at the Alan Turing Institute delivered to other members of the Clean Air In London team. Progress on the IRL/RL front had until that point still been slow. The improved map-matching algorithm (algorithm 4) had been successfully developed.

6.2.3 Phase Three: 25th June - 13th September

Initially, through reading other similar papers on IRL-based route calculation, an algorithm that utilises a MaxEntIRL approach was proposed. However, as task 6 began in mid-July (see Appendix D), it became apparent that more time was needed. It was therefore decided to move forward with the heuristically set feature weights. A forward RL algorithm was developed and tested throughout August, culminating i the results see in chapter 5.

6.3 Risk Assessment

The potential risks of the project have been identified and evaluated as either high, medium, or low risk. Proposals to limit or mitigate the risks have also

been suggested.

- **Slow progress**
  Likelihood: Medium
  Impact: Medium
  Solution: Asses progress at regular intervals and meet with Dr. Theo Damoulas weekly.

- **Lost data**
  Likelihood: Low
  Impact: High
  Solution: Back up all data on an external hard drive at regular intervals. Use private Github account to store project code off-site.

- **Burnout**
  Likelihood: High
  Impact: Medium
  Solution: The effects of this risk can be avoided by adding slack to the timetable to allow for deadline extensions if necessary.

- **Lost focus**
  Likelihood: Medium
  Impact: Low
  Solution: Meet regularly with Dr. Theo Damoulas.

- **Low quality of results**
  Likelihood: Low
  Impact: Medium
  Solution: Substantiate all results with solid theory.

6.4 Resources

6.4.1 Python

The main programming language that was used is Python. This was due to Python's well documented libraries geared towards solving complex computational problems. Much of the existing code had already been written in Python, so it was also the natural choice to continue developing the system.

6.4.2 NetworkX

The road network of London and average pollution values were represented as a NetworkX graph. NetworkX is a Python package for the analysis of complex networks. It comes with many useful methods for interrogating graph topologies and some basic shortest path functions.

6.4.3 QGIS

QGIS is a geographic information systems (GIS) platform that was used primarily for viewing and editing shape files. QGIS is also able to calculate summary statistics. All of the routes generated in this project were visualised using QGIS.

6.4.4 PostGreSQL

A PostGreSQL database had already been set up to store data for the road network and air pollution. When data for road segment features became available, they were also saved in the database.

6.4.5 PostGIS

PostGIS is an extension to PostGreSQL that supports geographical geometry types such as lines, points, and polygons. It was used widely with other GIS packages such as QGIS.

6.4.6 Excel

Excel supports easy-to-use spreadsheet manipulation for data analysis and visualisation. Important results were first saved as `CSV` files and imported into Excel to generate graphs.

6.4.7 Github

Github is an online version control repository. This allowed collaborative coding and easy retrieval of code to a previous state. In addition, Github served as a backup for code if it is accidentally deleted or lost.

6.5 Ethics

Since GPS data of real users of the Strava application has been used in this research, legal and ethical issues need to be considered. The trajectories of User A were provided voluntarily. Trajectories of User B ad C were collected using a feature accessible to paying Strava subscribers, whereby routes of any other user can be downloaded. Users are able to opt-out of this feature. The London road network data is free to use under the Open Government License [**?**].

Chapter 7

Conclusion

This project has successfully demonstrated the feasibility of calculating routes through London that minimise air pollution exposure. Also, it has shown that by considering historic trajectory data, routes can be adapted to follow more popular roads. Generalising to areas of London that have not been traversed, road preference maps were generated and utilised with a Q-Learning algorithm to generate optimal routes specific to the historic data being considered. There are some limitations to the work presented in this report, the improvements to which are briefly outlined in this section.

7.1 Improvements

7.1.1 Full IRL Treatment

The manner in which the feature weights $\boldsymbol{\alpha}$ were calculated did not follow standard methods of IRL, but rather a scaled-back approach concerning only feature visitation frequency. Despite producing results that are easy to interpret and nicely behaved (as in, the results match our expectation), the method suffers from a number of weak theoretical guarantees.

The idea for the calculation of $\boldsymbol{\alpha}$ was borrowed from the field of information retrieval. Devised by K. Jones in 1972, the numerical statistic $tf\text{-}idf$ (short for "term frequency - inverse document frequency") is a measure how important a word is in a document, which is itself part of a larger corpus [18]. To arrive at the definition of κ in equation 3.35, the logic from the formulation of $tf\text{-}idf$ was followed, replacing "term in a document in a corpus" for "feature of an action in an MDP". At first glance, this appears to be a valid extrapolation of the original statistic's purpose. Indeed, the idea of surprisal has been used in other probability based formulations. Also, the distribution of action visitations in the trace data follows very closely a power law (see Fig. 5.4), the same law that governs the Zipf distribution of word frequency in a corpus [49]. However, a full and strict derivation has not been carried out, putting any the formulation's legitimacy in this case under question.

The $tf\text{-}idf$ statistic in information retrieval is widely used as a heuristic, with much of today's search engines employing it in their systems. Despite this, its theoretical foundations are still not fully accepted more than four decades after its introduction. S. Robertson has attempted to put the statistic on a

firmer theoretical grounding [30]. To avoid this theoretical uncertainty, the full IRL formulation as presented in section 3.5.1 could be made. Adapting for the specifics of this project, following a MaxEntIRL approach has been shown to work well with route modelling problems. The principle of maximum entropy states that: the probability of a trajectory demonstrated by the expert is exponentially higher for higher rewards than lower rewards [17], i.e.:

$$P(\zeta) \propto e^{\mathcal{R}(\zeta)} \tag{7.1}$$

The feature weights α_i of the optimum reward function can be computed by maximising the likelihood of the demonstrated trajectories:

$$\begin{aligned} \boldsymbol{\alpha}^* &= \arg\max_{\boldsymbol{\alpha}} L(\boldsymbol{\alpha}) \\ &= \arg\max_{\boldsymbol{\alpha}} \frac{1}{|\mathcal{Z}|} \sum_{\zeta \in \mathcal{Z}} \log P(\zeta|\boldsymbol{\alpha}) \end{aligned} \tag{7.2}$$

Obtaining feature weights involves using gradient-based optimisation. The gradient is defined as the learner's expected feature counts $\widetilde{\boldsymbol{f}} = \frac{1}{|\mathcal{Z}|} \sum_{\zeta} \boldsymbol{f}_{\zeta}$ and the expected empirical feature counts, the latter of which can be expressed in terms of expected state visitation frequencies, $P(s|\boldsymbol{\alpha})$.

7.1.2 Larger Feature Set

By considering a finite number of road features we accept that it is only an approximation for the true features that are under consideration by a individual. The 57 features used in this project could certainly be augmented, perhaps allowing users to select which features are more important for them, to both strive for and avoid. An interesting extension to feature selection is explored by J. Choi and K. Kim in their paper *Bayesian Nonparametric Feature Construction for Inverse Reinforcement Learning* [7]. They construct new features based on features that co-occur frequently. For example, the presence of a cycle lane and close proximity to water would be given its own feature attribute with (presumably) a high feature weight. Feature conjunctions like this were learned using a Bayesian nonparametric approach. S. Levine *et al.* propose a similar method, presenting a Feature construction for Inverse Reinforcement Learning (FIRL) algorithm, which constructs features as logical conjunctions of the components that are most relevant for the observed examples [20].

7.1.3 Intrinsic Reward Gradient

One of the motivations for IRL is to avoid the need to manually set a reward function. Unfortunately it was necessary to apply a reward gradient in order to drive the agent towards the user defined target state. A better way would be to be able to generate a reward gradient intrinsically. One method to avoid this is proposed by J. Zheng and L. Ni [45]. Instead of considering historic data as unconnected edge visitations, they model drivers' routing decisions by exploiting entire trajectories while isolating the influence of heterogeneous destinations.

7.2 Future Work

7.2.1 Different Modes of Travel

Further segmenting the historic data into clusters with each cluster of trajectories sharing similar attributes would offer an interesting extension to this project. Thus far the trajectories were segmented by user and implicitly by mode of travel (cycling or running). But trajectories may not have been carried for the same purpose. For example, we could segment the data based on whether the trajectory is for training or commuting purposes. There may be interesting differences in road selection based on the time of day, or whether the user is going for a record time. Further attributes of each trajectory would need to be recorded from the user, data that was not available for this project.

7.2.2 Sub-Goals

Further work on IRL being carried out by Eduardo Barp in his master's thesis titled *"Bayesian Inverse Reinforcement Learning for Path-Based Reward Inference"*. Based on the work by B. Michini and J. How [23], he his working on inferring sub-goals from historic trajectory data. This would be an interesting an beneficial extension to this project as it would eliminate the need to apply a reward gradient.

7.3 Final Remarks

In conclusion, this master's project met the three aims set out in chapter 1. Weaknesses that exist in the methodology have been examined and solutions proposed to be tackled in further work. It is hoped that this research will inform future research opportunities in the field of adaptive routing systems.

Bibliography

[1] P. Abbeel and A. Y. Ng. Apprenticeship learning via inverse reinforcement learning. In *Proceedings of the Twenty-first International Conference on Machine Learning*, ICML '04, pages 1–, New York, NY, USA, 2004. ACM.

[2] P. Apparicio, M. Carrier, J. Gelb, A.-M. Séguin, and S. Kingham. Cyclists' exposure to air pollution and road traffic noise in central city neighbourhoods of montreal. *Journal of Transport Geography*, 57:63 – 69, 2016.

[3] I. E. Authority. Energy and air pollution: World energy outlook special report. https://www.iea.org/publications/was freepublications/publication/WorldEnergyOutlookSpecialReport2016EnergyandAirPollution.pdf, 2016.

[4] B. Bakker and J. Schmidhuber. Hierarchical reinforcement learning with subpolicies specializing for learned subgoals. In *Neural Networks and Computational Intelligence*, pages 125–130. Citeseer, 2004.

[5] R. Beelen, G. Hoek, D. Vienneau, M. Eeftens, K. Dimakopoulou, X. Pedeli, M. Tsai, N. Künzli, T. Schikowski, A. Marcon, K. Eriksen, O. Raaschou-Nielsen, E. Stephanou, E. Patelarou, T. Lanki, T. Yli-Tuomi, C. Declercq, G. Falq, M. Stempfelet, M. Birk, J. Cyrys, S. von Klot, G. Nádor, M. Varró, A. Dedele, R. Gražulevičiene, A. Molter, S. Lindley, C. Madsen, G. Cesaroni, A. Ranzi, C. Badaloni, B. Hoffmann, M. Nonnemacher, U. Krämer, T. Kuhlbusch, M. Cirach, A. de Nazelle, M. Nieuwenhuijsen, T. Bellander, M. Korek, D. Olsson, M. Strömgren, E. Dons, M. Jerrett, P. Fischer, M. Wang, B. Brunekreef, and K. de Hoogh. Development of no2 and nox land use regression models for estimating air pollution exposure in 36 study areas in europe - the escape project. *Atmospheric Environment*, 72:10–23, 6 2013.

[6] R. Bellman. A markovian decision process. *Journal of Mathematics and Mechanics*, 6(5):679–684, 1957.

[7] J. Choi and K.-E. Kim. Bayesian nonparametric feature construction for inverse reinforcement learning. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, pages 1287–1293. AAAI Press, 2013.

[8] R. Collins, Z. Huang, S. Perera, B. Sheridan, and O. Styles. Cycle hire and air pollution in london. *Group Project for IM917 - Spatial Methods and Practice in Urban Science*, 2017.

[9] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[10] W. R. O. for Europe. Health risk assessment of air pollution, 2016.

[11] C. Gatti. *Design of Experimenmts for Reinforcement Learning*, chapter 2, pages 7 – 16. 2190-5053. Springer International Publishing, 1st edition, 2015.

[12] S. Gelly and D. Silver. Monte-carlo tree search and rapid action value estimation in computer go. *Artificial Intelligence*, 175(11):1856 – 1875, 2011.

[13] J. Green, J. Grosso, L. Ingram, O. Hamelijnck, and S. Barr. To breathe or not to breathe: Modelling air pollution. Master's thesis, University of Warwick, 2017.

[14] T. Guardian. Tipping point: revealing the cities where exercise does more harm than good. https://www.theguardian.com/cities/2017/feb/13/tipping-point-cities-exercise-more-harm-than-good, February 2018.

[15] M. Habermann, M. Billger, and M. Haeger-Eugensson. Land use regression as method to model air pollution. previous results for gothenburg/sweden. *Procedia Engineering*, 115:21 – 28, 2015. Toward integrated modelling of urban systems.

[16] T. A. T. Institute. Clean air in london. https://www.turing.ac.uk/research-projects/clean-air-london/, February 2018.

[17] E. T. Jaynes. Information theory and statistical mechanics. *Phys. Rev.*, 106:620–630, May 1957.

[18] K. S. JONES. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[19] A. Kara and I. Dogan. Reinforcement learning approaches for specifying ordering policies of perishable inventory systems. *Expert Systems with Applications*, 91:150 – 158, 2018.

[20] S. Levine, Z. Popovic, and V. Koltun. Feature construction for inverse reinforcement learning. In J. D. Lafferty, C. K. I. Williams, J. Shawe-Taylor, R. S. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems 23*, pages 1342–1350. Curran Associates, Inc., 2010.

[21] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang. Map-matching for low-sampling-rate gps trajectories. In *Proceedings of the 17th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, GIS '09, pages 352–361, New York, NY, USA, 2009. ACM.

[22] L. Mercer, A. A Szpiro, L. Sheppard, J. Lindström, S. Adar, R. W Allen, E. Avol, A. P Oron, T. Larson, L.-J. Sally Liu, and J. Kaufman. Comparing universal kriging and land-use regression for predicting concentrations of

gaseous oxides of nitrogen (nox) for the multi-ethnic study of atherosclerosis and air pollution (mesa air). *Atmospheric Environment*, 45:4412–4420, 08 2011.

[23] B. Michini and J. P. How. Bayesian nonparametric inverse reinforcement learning. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 148–163. Springer, 2012.

[24] L. A. Q. Network. Is pollution worse in london? https://www.londonair.org.uk/LondonAir/ guide/London.aspx, 2016., 2016.

[25] B. News. Artificial intelligence: Google's alphago beats go master lee se-dol. http://www.bbc.co.uk/news/technology-35785875, March 2016.

[26] A. Y. Ng and S. J. Russell. Algorithms for inverse reinforcement learning. In *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, pages 663–670, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[27] P. O'Hara. Running from air pollution. Master's thesis, University of Warwick, 2018.

[28] OSM. Open street map. https://www.openstreetmap.org/#map=7/52.446/-1.154, August 2018.

[29] S. M. Piers MacNaughton1, J. Vallarino, G. Adamkiewicz, and J. D. Spengler. Impact of bicycle route type on exposure to traffic-related air pollution. *The Science of the total environment*, 490:37–43, 2014.

[30] S. Robertson. Understanding inverse document frequency: on theoretical arguments for idf. *Journal of Documentation*, 60(5):503–520, 2004.

[31] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484 EP –, 01 2016.

[32] L. D. Store. London atmospheric emissions for 2013 of nox. https://data.london.gov.uk/dataset/london-atmospheric-emissions-inventory-2013, February 2018.

[33] L. D. Store. Shape files concerning the bodies of water in london. https://data.london.gov.uk/dataset/water-quality-london-rivers-other-waterbodies, August 2018.

[34] Strava. Strava website. https://www.strava.com, August 2019.

[35] O. Survey. Ordnance survey. https://www.ordnancesurvey.co.uk, August 2018.

[36] R. S. Sutton and A. G. Barto. *Reinforcement Learning, An Introduction*. The MIT Press, 1st edition, 1998.

[37] M. Tainio, A. J. de Nazelle, T. Götschi, S. Kahlmeier, D. Rojas-Rueda, M. J. Nieuwenhuijsen, T. H. de Sá, P. Kelly, and J. Woodcock. Can air pollution negate the health benefits of cycling and walking? *Preventive Medicine*, 87:233 – 236, 2016.

[38] M. Thorup. Undirected single-source shortest paths with positive integer weights in linear time. *J. ACM*, 46(3):362–394, May 1999.

[39] J. F. G. Tromp. Combinatorics of go. https://tromp.github.io/go/gostate.pdf, 2016.

[40] C. J. C. H. Watkins. *Learning from delayed rewards*. PhD thesis, King's College, Cambridge, 1989.

[41] A. William. Moore. *Efficient Memory-Based Learning for Robot Control*. PhD thesis, University of Cambridge, 06 1990.

[42] H. Wu, Z. Chen, W. Sun, B. Zheng, and W. Wang. Modeling trajectories with recurrent neural networks. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 3083–3090, 2017.

[43] P. Zannetti. *Air Pollution Modeling - Theories, Computational Methods and Available Software*. Springer US, 1st edition, 1990.

[44] F. B. Zhan and C. E. Noon. Shortest path algorithms: An evaluation using real road networks. *Transportation Science*, 32(1):65–73, 1998.

[45] J. Zheng and L. Ni. Modeling heterogeneous routing decisions in trajectories for driving experience learning. *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, pages 951–961, 09 2014.

[46] Y. Zheng, F. Liu, and H.-P. Hsieh. U-air: When urban air quality inference meets big data. In *Proceedings of the 19th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '13, pages 1436–1444, New York, NY, USA, 2013. ACM.

[47] B. D. Ziebart, A. Maas, J. A. Bagnell, and A. K. Dey. Maximum entropy inverse reinforcement learning. In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, pages 1433–1438. AAAI Press, 2008.

[48] B. D. Ziebart, A. L. Maas, A. K. Dey, and J. A. Bagnell. Navigate like a cabbie: Probabilistic reasoning from observed context-aware behavior. In *Proceedings of the 10th International Conference on Ubiquitous Computing*, UbiComp '08, pages 322–331, New York, NY, USA, 2008. ACM.

[49] G. Zipf. *The Psychobiology of Language: An Introduction to Dynamic Philology*. M.I.T. Press, Cambridge, Mass., 1935.

Appendix A

Algorithm 5 Dijkstra's Algorithm

Require: A source node s
Ensure: The shortest paths from s to all other nodes in G

```
1:  for all v ∈ V[G] do
2:      d[v] ← +∞
3:      previous[v] ← undefined
4:  end for
5:  d[s] ← 0
6:  S ← empty set
7:  Q ← V[G]
8:  loop
9:      Q is not an empty set
10:     u ← Extract_Min(Q)
11:     S ← S ∪ {u}
12:     for all edge (u, v) outgoing from u do
13:         if d[u] + w(u, v) < d[v] then
14:             d[v] ← d[u] + w(u, v)
15:             previous[v] ← u
16:         end if
17:     end for
18: end loop
```

Appendix B

Gantt chart on next page.

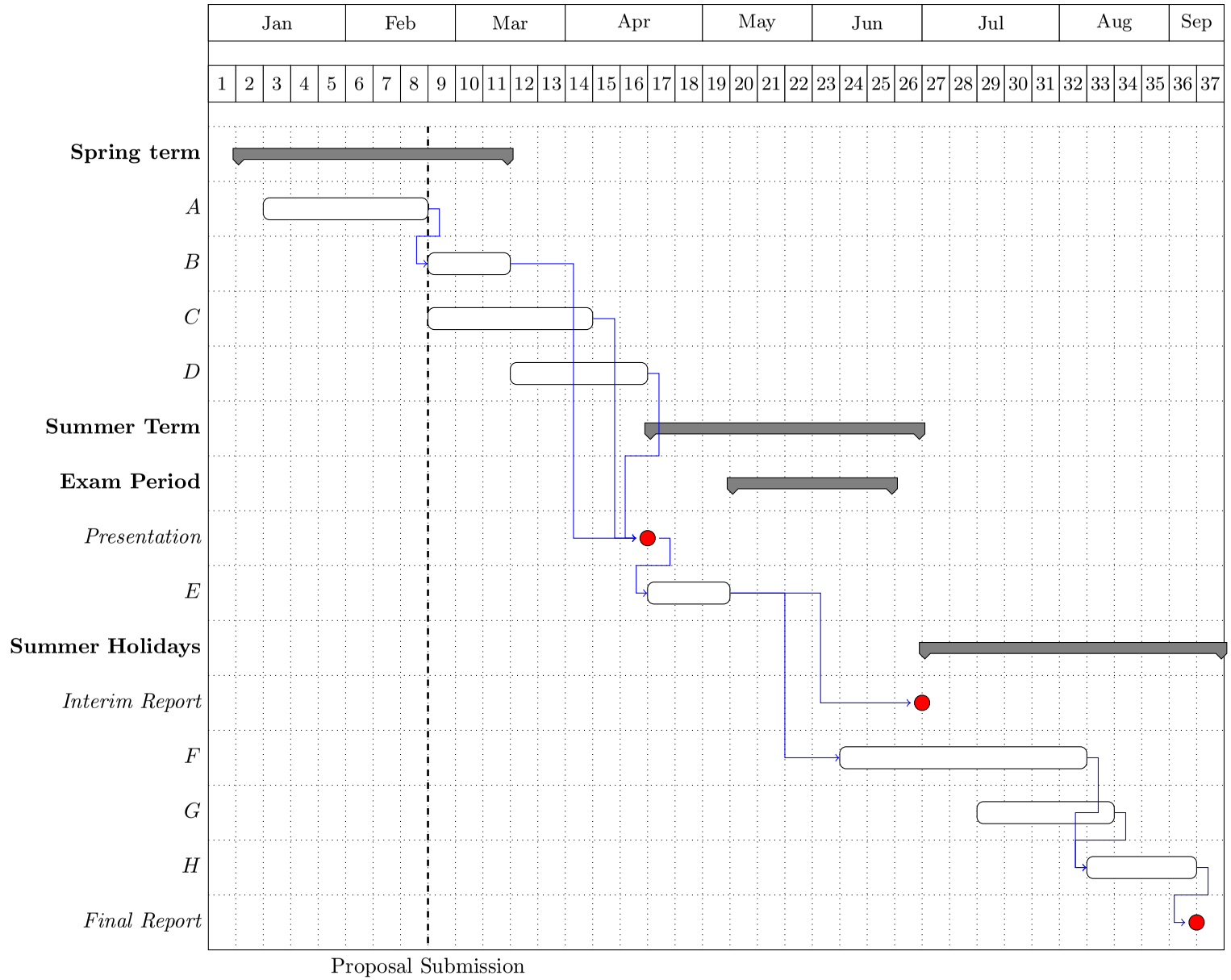


Figure 7.1: Numbers represent weeks. Key: *A* - Literature review and strategy development; *B* - Continue strengthening knowledge base in key theory and practice; *C* - Familiarise self with existing code; *D* - Begin running initial algorithms with air pollution and road data; *E* - Appraisal and reflection on project so far, details of which will form part of the interim report; *F* - Research and development of novel techniques based on progress of project to this point; *G* - Implementation period; *H* - Writing up final report.

Appendix C

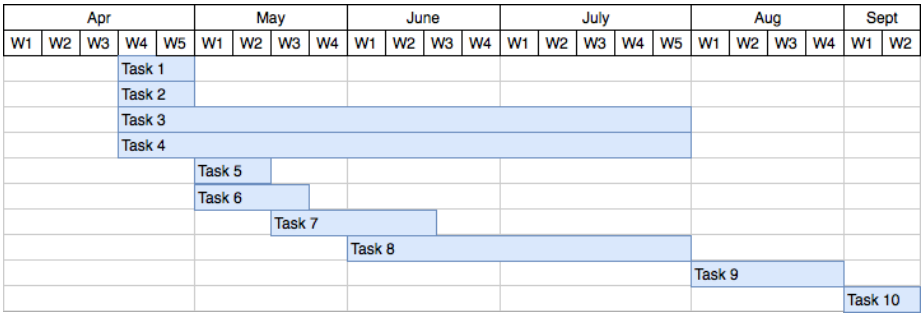


Figure 7.2: Key: Task 1 - Convert users' trajectories to lists of nodes and edges; Task 2 - Download at least 3 more user example data; Task 3 - Start coding RL model; Task 4 - Start coding IRL model; Task 5 - Download data from a further 12 users; Task 6 - Build data layer expressing users' route preference; Task 7 - Build a working model incorporating pollution minimisation and user preference; Task 8 - Improve first model from task 7 incorporating RL and IRL; Task 9 - Write up thesis; Task 10 - Check and submit.

Appendix D

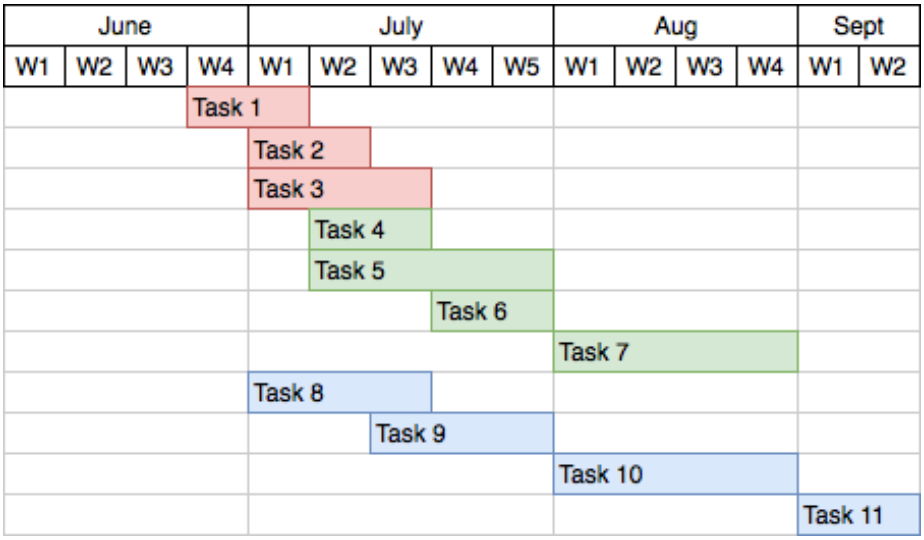


Figure 7.3: Key: Task 1 - Discretise continuous features such as distance, pollution etc.; Task 2 - Calculate proximity to green spaces and discretise; Task 3 - Include road type and water proximity in the feature set; Task 4 - Develop subroutine to calculate the feature expectation $\boldsymbol{\mu}$; Task 5 - Set benchmark with heuristically set reward weights; Task 6 - Develop full IRL system based on MaxEntIRL; Task 7 - Testing phase used to generate optimum paths; Task 8 - Write introduction, literature, and theory of dissertation; Task 9 - Write implementation; Task 10 - Write results and conclusion; Task 11 - Check and submit dissertation.