Imperial College London

Department of Computing

An Integrated London Journey Planner

Author:
Ryszard T. Kaleta

Supervisors:
Dr. Alessandra Russo
Dr. Luke Dickens

Second Marker:
Dr. Francesca Toni

19 June 2012

Submitted in part fulfilment of the requirements for the degree of Master of Engineering in Computing of Imperial College London

Abstract

Urban cycling is becoming increasingly popular. For many commuters and tourists alike it is the cheaper and more pleasant alternative to traditional modes of public transport. Urban cycling is supported by many cities worldwide through introduction of cycling lanes and, more importantly to those who do not own a pair of wheels themselves, also the creation of bicycle sharing schemes.

City public transportation networks are not easy to navigate. This is why most provide on-line journey planners that allow users to search for a desired mix of transport links to reach destination. We believe that such journey planners should also incorporate the bicycle sharing schemes. However, to build an effective journey planner one has to know the future arrival times of various modes of transport such that waiting time whilst connecting is minimised.

The latter is a non-trivial task when it comes to bicycle sharing schemes because there is no schedule of bicycle arrivals at various docking stations. This makes it hard to plan cycling journeys that are to occur in the future and is the reason no journey planner built thus far has fully catered for the needs of urban cyclists. We aim to change all this by designing and implementing a journey planner for London, UK that integrates a bicycle sharing scheme with other modes of public transport whilst minimizing wait times at docking stations through bicycle availability prediction.

Acknowledgements

I am grateful to Dr. Alessandra Russo and Dr. Luke Dickens for their continuous support and guidance throughout the course of this project.

Above all, I would like to thank my parents and closest family - without them I would not have been able to make it this far.

Contents

1 Introduction **3**

2 Background **7**
2.1 Terminology . . . 7
2.2 Bicycle Sharing Systems . . . 7
2.3 Transport for London . . . 8
2.4 Journey Planning Data Sets . . . 10
2.4.1 Past cycle journeys . . . 10
2.4.2 Live bicycle availability . . . 10
2.5 Probability Theory . . . 13
2.5.1 The Basics . . . 13
2.5.2 Density Models . . . 14
2.5.3 Density Estimation . . . 15
2.5.4 Maximum Likelihood Estimation . . . 16
2.6 Graph Theory . . . 17
2.6.1 The Basics . . . 17
2.6.2 Path finding . . . 17

3 System Architecture **20**

4 Predicting Bicycle Availability **23**
4.1 Model Definition . . . 25
4.2 Parameterizing the Model . . . 31
4.3 Making Predictions . . . 32
4.3.1 Using Cumulative Distribution Function . . . 33
4.3.2 By Sampling the Density Estimator . . . 35

5 Routing **41**
5.1 Graphs . . . 42
5.2 Modified `astar_path` . . . 42
5.3 Cost Models . . . 46
5.4 Complete Journey Planning . . . 52
5.5 Pathmax Optimisation . . . 54

6 Results and Evaluation 57
6.1 Bicycle Availability Model Performance 57
6.1.1 Functional Performance 57
6.1.2 Non-Functional Performance 61
6.2 Routing Algorithm Performance 67
6.2.1 Functional Performance 67
6.2.2 Non-Functional Performance 68

7 Conclusions and Future Work 73
7.1 Conclusion 73
7.2 Improving Bicycle Availability Predictions 74
7.3 Improving Router 75

A Journey Planner - In Action 77

Chapter 1

Introduction

Bicycle sharing systems are being introduced as the latest mode of public transport all over the world (see section 2.2). The providers are driven by their positive environmental impact to increase their popularity. The cyclists often see these systems as a cheap and cheerful alternative to more traditional modes of urban transport. Apart from introducing the systems themselves, the city planners attempt to make their streets more bicycle-friendly. Most journey planning software allows the user to set a number of parameters before the route is calculated, such that most desirable journey path can be found.

The amount of time an urban journey maker spends waiting whilst travelling on public transport has a significant influence on their choice of transport and the willingness to use it again. The more a passenger has to wait throughout their journey, the less *reliable* the transport mode in question will seem. Journey planners often consider current traffic and network conditions in their attempts to find routes that are most desirable to the user yet avoid any ongoing delays. This works well with modes of public transport that run according to a timetable as alternative routes that avoid these problems can be easily found.

The vast majority of journey planners that are capable of incorporating cycling into their routes assume the user owns a bicycle. Finding a cycling route is then relatively easy as all we have to do is to take into account users' preferences and find a path that satisfies them. This is done very successfully by a number of free route planning solutions. Apart from turn-by-turn navigation, *cyclestreets.net* [12] is able to provide a very impressive feedback on the proposed cycling journey, including the number of burnt calories, CO_2 avoided and even the number of traffic lights and crossings that are passed on the way. A number of *OpenTripPlanner* implementations [31] provide a similar service for a number of cities around the world. Created from data gathered by the cyclists themselves, they have the potential of containing information not found in other cycling route planners, such as picturesqueness.

However, trying to include cycling into routes, when no assumption about bicycle ownership can be made, is more difficult. This is because, while we are keen to utilise the bicycle sharing systems, these have a limit on the number of bicycles that are available at docking stations. The journey planning software is unable to guarantee that the user will be able to start and finish their cycling journey at the elected docking stations, since the docking stations of interest may either be out of bicycles or have no free parking space left. This is particularly problematic if the bicycle sharing system charges their users for bicycle hire - then, any delay that occurs because of inability to complete the cycling journey as planned by the routing software is not only putting the user off using that journey planning software and the bicycle sharing scheme again, but is now also costly.

The problem is tackled by both the bicycle sharing systems' providers as well as the cycling journey planners. Transport for London, who own a large bicycle sharing system in London, UK (called BCH and described in section 2.3), provide the following guidelines when problems with picking up or dropping off bicycles occur:

- if there are no bicycles at the docking station, the passenger can use the docking station's map to locate other docking stations nearby. There is no guarantee there will be a bicycle available at those stations
- if the docking station is full, the passenger can get up to 15 minutes extra time to cycle to another station before extra charges for late bicycle return start to apply. As above, there is no guarantee that there will be a parking space at the nearby stations

Often, this is not a good enough solution [24]. That is why we have seen a number of mobile phone applications being developed that can locate the nearest docking station and provide the latest available information on the number of working bicycles and free parking spaces at that station. However, with this solution the task of planning the journey is left to the user.

The most sophisticated solution to the problem is provided by journey planning software that bases its suggested cycle routes around the use of bicycle sharing schemes, but additionally considers the latest bicycle availability at all active docking stations. If a docking station is currently out of working bicycles or all of its docks are in use, the software seeks an alternative route that uses other docking stations. Transport for London is the best example of such a journey planner the author was able to find and we examine it in more detail in section 2.3.

All of the above routing software misses one important point - a user is rarely able to begin their cycling journey the very moment they ask for a route to be found. Normally, some time will pass between journey planning and the time the user arrives at a docking station to start their journey. As such, using *live* data on bicycle and parking space availabilities is not helpful as the *state of the world* is likely to change between now and journey start time. From the time

of planning the journey to actually reaching one of the docking stations the bicycles that were available when we planned our journey might have by now been taken away by other members of the public. The time a bicycle will be returned to this station such that we can continue on our journey is unknown. This is not the case for more traditional modes of public transport such as a train or a bus, where a timetable of arrivals exits.

The only true way of improving the reliability of journey planning software that includes cycle path routing based on bicycle sharing systems is to predict bicycle availability at journey origin/destination docking stations at the time the user is set to reach the docking station in question.

With this project, we aim to:

1. collect data on past BCH cycle journeys and current bicycle availability across all BCH stations
2. devise a model capable of predicting future availability of bicycles at BCH's docking stations based on above historical evidence
3. devise a route planner that will combine walking, cycling and the London Underground network to create a route that is most desirable to the user. It should be capable of calculating routes based on distance, time and route busyness
4. allow the user the control over the setting of those preferences such that they are able to define what the most desirable route would be (mention achieving this (maths wise) in cost models functions, and UI-functionality wise when evaluating user experience)
5. incorporate the bicycle availability prediction model into said route planner in the aim of creating a more accurate and satisfying journey planning experience

The end-product is an implementation of a journey planner capable of finding routes combining walking, cycling and travel on the London Underground across Greater London area. The journey planner tries to find a cycling route and when this is not possible given user-defined preferences, a mix of walking, cycling and London Underground routes is suggested as well. Our journey planner makes no assumption of bicycle ownership and instead utilises a large bicycle sharing scheme that exists in the city centre. It goes further than all other routing software has ever gone before by attempting to predict future bicycle availability within this system using density estimation techniques, such that the users feel cycling can be a reliable mode of public transport. The journey planner interacts with the users via map-based web interface that allows the users to specify their most desirable journey across a number of parameters.

Whilst working on this project, we have also been able to contribute to NetworkX, a Python language software package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks. We

have extended the functionality of NetworkX's `astar_path` to finding shortest paths in directed and undirected multigraphs, where only simple graphs were handled before [22].

The report is structured to describe our approach to each of the above aims in turn. Thus in Chapter 2 we describe the data we will use in developing bicycle availability models, themselves described in Chapter 4. In Chapter 5 we describe our routing methods that combine user's preferences as to the desired journey with the predicted bicycle availabilities at docking stations. Our journey planner is built as a number of components whose design is briefly described in Chapter 3. Chapter 6 shows our results and asses the suitability of our methods. To see the journey planner in action, investigate figures in Appendix A.

Chapter 2

Background

2.1 Terminology

The following definitions will be used frequently throughout this report. We clarify their indented meaning below:

- *BCH* is an acronym we will use when referring to Barclays Cycle Hire scheme
- *Bicycles* refers to bicycles that are part of the BCH
- *Docking station* refers to the London-wide BCH terminals where bicycles can be parked and picked up form
- *Bicycle dropoff* refers to the act of arriving at a docking station that is part of the BCH and parking the bicycle at an available dock
- *Bicycle pickup* refers to the act of departing from a docking station that is part of the BCH by taking an available and functional bicycle out of its dock and cycling away

2.2 Bicycle Sharing Systems

Bicycle sharing system is a service that provides affordable access to bicycles to individuals who do not own any themselves. Run mainly by local government agencies, the systems are an alternative to motorized public transport on short-distance trips. The authorities hope the systems will reduce traffic congestion, noise and air pollution. As of 2011, around 300 such schemes were operating worldwide [3]. Examples of successful implementation are manifold:

- *Dublinbikes*, setup in September 2009, reached 1 million uses in less than a year
- *Cyclocity* programs, launched by JCDecaux, spread out of France into Brisbane, Australia and Vienna, Austria
- New York City, USA plans to introduce its own *Citibike* system in July 2012. With 10,000 bicycles available from 600 stations spread throughout the city, this will be the largest system of its kind in North America

Operating the bicycle sharing schemes can be very profitable too - *Bixi* [6], a system developed by Public Bike System Company in Montreal, Canada, recorded net income of CAD1.5 million in the financial year 2011 [33]. Since most systems charge passengers on a per-trip basis, the providers are interested in increasing the popularity of their bicycle networks.

2.3 Transport for London

Transport for London (TfL) is the local government body responsible for most aspects of the transport system in Greater London. We are interested in TfL for two reasons:

1. they own and operate BCH, described next, on which we shall use for the cycling parts of the routes calculated by our journey planner
2. they provide data that we can use to build bicycle availability models. This data, described in sections 2.4.1 and 2.4.2, is provided free of charge and available to anyone who registers in TfL's Developer' Area [16].

Barclays Cycle Hire

BCH is a bicycle sharing system owned by Transport for London (TfL) that was launched on 30 July 2010. Available 24 hours a day, this self-service operates 8,000 bicycles across 570 docking stations spread around 65 km^2 of central London. By March 2012, the system has registered 10 million 'hires', making it one of the most successful in the world [2]. This also means we will have access to a substantial amount of historical data on which to build our availability model.

TfL already provides a cycle journey planner that incorporates BCH. Figure 2.1 shows the cycling journey planner following a request to calculate an exemplary cycling journey across central London. The start and finish points are entered manually by the user and we found that our home postcode was not recognised. The route is calculated by finding BCH docking stations nearest to user-defined start and finish locations. The route is then formed of three parts:

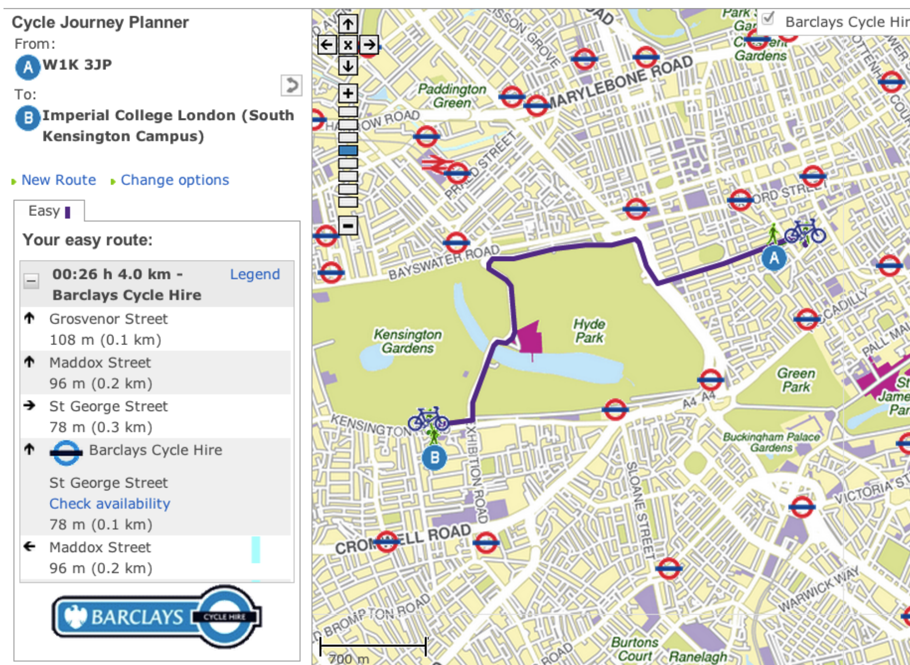

Figure 2.1: TfL's Cycle Journey Planner[17]

1. using the user-defined start location and the location of starting docking station, the *start-walk* part of route is found. This helps the passenger reach the nearest docking station
2. using the locations of starting and finishing docking stations, as well as preferences for route busyness (set in *options*), the cycling part of resulting route is found
3. finally, using the user-defined finish location and the location of the finishing docking station, the *finish-walk* part of the route is found

We can see in Figure 2.1 that the user can check live availability of a docking station to check if bicycles are available. This is an availability check made at the time the planner is used and no attempt is made to estimate the future availability.

2.4 Journey Planning Data Sets

In this section we describe the data that we were able to and needed to obtain as part of this project. We first describe the data we will need to build our model of bicycle availability. We then briefly mention other data that is needed to build our journey planner.

2.4.1 Past cycle journeys

We have obtained access to data listing all BCH journeys made from 30 July 2010 to 31 May 2011 [18]. Each journey record lists:

- bike ID
- journey start date and time
- start docking station
- end date and time
- end docking station

Methods described in sections 4.2 and 4.3 will use this data to estimate the number of pickups and dropoffs for each docking station at different time points of the day.

2.4.2 Live bicycle availability

We have also obtained access data listing the current status of every docking station. Unlike the past cycle journeys data described above, this is a live feed that comes directly from Serco Group's database and is updated in three-minute

intervals, 24 hours a day, seven days a week [19]. Serco Group are the service providers of BCH. Each update includes the following information on every operation docking station:

- update time stamp
- name, location and co-ordinates
- availability for usage
- total number of bicycles available at a docking station
- number of docking points available at a docking station, excluding any defective bike docks
- total number of docking points available at a docking station

Methods described in sections 4.2 and 4.3 will use this data to improve the estimated number of pickups and dropoffs for each docking station at different time points of the day, as calculated using past cycling journeys data described above.

London Underground Data

Our journey planner will be capable of mixing journeys on the London Underground into the routes it suggests to the user. For this, we need the following information on every London Underground station:

- station name
- station co-ordinates

We would also like to know how the stations are connected, such that we can find paths through the underground network. This means that for any two connected London Underground stations we would like to know:

- the London Underground lines that connect these stations
- the distance travelled by the underground train between these stations and the time this takes.

TfL does not provide a straight-forward access to above data. We have found alternative sources [26][27][11]. Later we find that the data is not always 100% accurate. Though we consider the accuracy good enough for a prototype application, we note in Chapter 3 that our journey planner has been designed with future improvements in mind - the underlying data can be easily swapped inside our database for a more accurate set without any code changes.

Greater London data

Finally, we need a data set from which a model of Greater London can be built. We need such model so that we can apply the techniques described in chapter 5 for finding street-level paths for walking and cycling. The data has to comprise a list of nodes (street level feature points such as junction) and edges (representing connections between pairs of nodes, such as a footpath, road or a bridge). An introduction to graph theory is provided in section 2.6. For now, we note that for this data we turned to OpenStreetMap - a collaborative project to create a free editable map of the world.

There are several reasons explaining our choice:

- our mapping needs require access to underlying data - the information, listed below, about every street, path and other street-level link that forms a network representing Greater London. If we were to collect data from Google Maps, for example, we would be creating *derived work*. The data Google uses in its maps service is either its own or licensed from mapping companies (for example NAVTEQ and Tele Atlas) or national mapping agencies, who made significant financial investment to obtain it and are understandably protective of their copyright. In practice, if our journey planner used the Google Maps API, we could be subject to licensing fees and contractual restrictions of these map providers. Use of OpenStreetMap for our purposes is completely free
- there exists a number of usage limits that apply to the Google Maps API
- we find that OpenStreetMap provides more information for built-up areas than Google Maps - house numbers are an example. There also exist a number of layers that can be applied on top of the underlying map tiles that show additional information, such as cycling routes or more points of interest

Of course, we are only interested in the area of Greater London. Having obtained an extract from OpenStreetMap that covers the city [10], we find it contains the following information:

- co-ordinates of nodes
- for every edge:
  - source and target nodes
  - edge length and geometry (an edge does not have to be a straight line)
  - car accessibility, which also tells us what type of road this edge is
  - bicycle accessibility, which also tells us how safe the edge is for cycling
  - foot accessibility

The accessibility information will help us calculate routes that suite our journey planner users' route busyness preference.

2.5 Probability Theory

Our approach to bicycle availability prediction will rely heavily on probability theory. Below we introduce the basics concepts that are required for understanding the topics discussed in later parts of this section.

2.5.1 The Basics

A *random variable* is a mapping from the sample space S to the real numbers, such that if X is a random variable, $X : S \to \mathbb{R}$. Each element of the sample space $s \in S$ is assigned by X a numerical value $X(s)$.

Probability distribution P is a function that describes the probability of X taking certain values in $\mathbb{R}$.

For a discrete random variable it holds that:

$$p(x) = \sum_{s} P(X = x) = 1, \forall s \in S \tag{2.1}$$

$p(x)$ is then called the *probability mass function* and it gives us the probability that a discrete random variable is exactly equal to some value [20].

The *cumulative distribution function* of random variable X tells us the probability that X takes a value less than or equal to x:

$$F(x) = P(X \leq x), \forall x \in \mathbb{R}$$

We can express the cumulative distribution function of a discrete random variable in terms of its probability mass function:

$$F(x_k) = \sum_{i=1}^{k} p(x_i) \tag{2.2}$$

Similarily

$$P(X < x_k) = \sum_{i=1}^{k-1} p(x_i) \tag{2.3}$$

2.5.2 Density Models

Chapter 4 describes bicycle availability models. These models need to estimate the number of bicycle pickups and dropoffs that occur at every bicycle docking station at different times of the day. We can think of these numbers as discrete random variables. They do this by estimating unobservable probability mass functions $p(X)$ that underlay these pickup/dropoff numbers. These models of the true distributions of random variables are otherwise known as density estimators or *density models*.

Density models can be parametric or non-parametric. The parametric density models are assumed to be of particular form that is characterised by a set of adjustable parameters θ, where $\theta \in \mathbb{R}$. In section 2.5.4 we introduce a method for calculating these parameters. First, however, we introduce two parametric forms of density models that will prove essential in our attempts to predict bicycle availability at docking stations.

Binomial Distribution

Binomial distribution is a discrete probability distribution defined as

$$P_p(k|N) = \binom{N}{k} p^k (1-p)^{N-k} \tag{2.4}$$

Since the above definition involves the *combination*

$$\binom{N}{k} = \frac{N!}{k!(N-k)!} \tag{2.5}$$

the binomial distribution can be thought of as describing the probabilities of obtaining k successes on N trials. In our case the k can be thought of as the number of dropoffs or pickups per some time interval in a day and N as the number of days for which we have sample data.

Poisson Distribution

For reasons listed in section 4.1 we are mainly interested in the *Poisson distribution*. Poisson distribution is another example of a parametric discrete probability distribution. It builds on the binomial distribution mentioned above to describe the probability of the number of events that are likely to occur within a fixed period of time. It is defined as the binomial distribution in the limiting case where $N \to \infty$, with p in (2.4) as the probability of a success.

If we set $\lambda = Np$, where λ can intuitively be thought of as the expected number of occurrences of an event in some time interval i, equation (2.4) can be rewritten as

$$P_{\lambda/N}(k|N) = \frac{N!}{k!(N-k)!}(\frac{\lambda}{N})^k(1-\frac{\lambda}{N})^{N-k} \tag{2.6}$$

Considering the mentioned limit, equation (2.6) becomes

$$\begin{aligned}
P_\lambda(k) &= \lim_{N\to\infty} P_p(k|N) \\
&= \lim_{N\to\infty} \left[\frac{N!}{N^k(N-k)!}\right](\frac{\lambda^k}{k!})(1-\frac{\lambda}{N})^N(1-\frac{\lambda}{N})^{-k} \\
&= \lim_{N\to\infty} \left[\frac{N(N-1)...(N-k+1)}{N^k}\right](\frac{\lambda^k}{k!})(1-\frac{\lambda}{N})^N(1-\frac{\lambda}{N})^{-k} \\
&= (1)(\frac{\lambda^k}{k!})(e^{-\lambda})(1) \\
&= \frac{\lambda^k e^{-\lambda}}{k!}
\end{aligned} \tag{2.7}$$

Formally, λ is a positive real number such that

$$\lambda = \mathbb{E}(X) = var(X) \tag{2.8}$$

2.5.3 Density Estimation

Density estimation helps us define the set of parameters θ that characterises a density model, such as a Poisson distribution, given observed data, such as that discussed in sections 2.4.1 and 2.4.2. Because we consider the observed data as having been drawn from the true distribution that we are trying to describe with our density model, we can make the assumption that such model inferred from such data is a good representation of this true distribution. In this context, the observed data can be referred to as the *sample data*.

Formally, density estimation is the problem of modelling a true, unobservable probability density (for continuous variables) or mass (for discrete variables) function $p(X)$ of a random variable X given a finite set of observations $\{x_i\}_{i=1}^N$ drawn from that true density function [9].

In section 2.5.2 we mentioned that assuming a parametric form of a density model is akin to limiting the hypothesis space of what the true distribution can possibly be. We note here that this means the parametric approach to density estimation introduces a number of assumptions that are made about the true distribution that we are attempting to estimate with our density models. These assumptions may or may not be true and they form a good basis for evaluating the density estimation methods described in section 2.5.4.

There exist a number of approaches to parametric density estimation [5]. In the next section we detail one of the methods.

2.5.4 Maximum Likelihood Estimation

As mentioned in sections 2.4.1 and 2.4.2, we have access to a number of observations about bicycle docking stations and some of the cycling journeys made in BCH's first year of operation. Considering this data as a sample of N random observations $\{x_i\}_{i=1}^N$, we wish to estimate the true value of a set of adjustable parameters θ of the probability distribution of the random variable X (representing the number of pickups or dropoff that occur) from which the sample was drawn. In other words, we assume the observed data is drawn from the true distribution and so we adjust the parameters that characterise our density model to make the observed data most likely, believing that this approximates our density model to the true distribution well.

Maximum likelihood estimation allows us to find $\hat{\theta}$, an estimator as close to the true value of θ as possible. The method works by building on the assumption that the probability of observing the sample data $\{x_i\}_{i=1}^N$, given θ, is a measure of the likelihood of θ given this data. By maximising the former we also effectively maximize the latter [32]. In other words, MLE will allow us to estimate the value of $\hat{\theta}$ by finding specific values for the parameters in θ that define a density model giving the random sample data the greatest probability.

It is easy to find $\hat{\theta}$ - this will be the set of density models parameters that maximises a likelihood function ℓ. A likelihood function describes the probability of obtaining exactly the observed data sample $\mathbf{x} = \{x_i\}_{i=1}^N$ given some values for the parameters in θ

$$\text{likelihood}(\theta, \mathbf{x}) = \ell(\mathbf{x}|\theta) \tag{2.9}$$

When we consider that the random observations $\{x_i\}_{i=1}^N$ are drawn independently from the same probability distribution, the above joint frequency function can be expressed as the product of the marginal frequency functions. This allows us to rewrite equation 2.9 as

$$\text{likelihood}(\theta, \mathbf{x}) = \prod_{i=1}^{n} \ell(x_i|\theta), \forall x_i \in \mathbf{x} \tag{2.10}$$

For convenience, we maximise a log of the likelihood function and not the likelihood function itself. Since a logarithm is a monotonically increasing function of its arguments, in an attempt to maximise the function all we have to do is maximise its log

$$\begin{aligned} \text{likelihood}(\theta, \mathbf{x}) &= \ln \prod_{i=1}^{n} \ell(x_i|\theta), \forall x_i \in \mathbf{x} \\ &= \sum_{i=1}^{n} \ln \ell(x_i|\theta), \forall x_i \in \mathbf{x}. \end{aligned} \tag{2.11}$$

Since the desired set of parameters θ is that which maximises the likelihood of sample data, we have that

$$\hat{\theta}_{MLE} = \arg\max_{\theta}(\text{likelihood}(\theta, \mathbf{x})) \tag{2.12}$$

2.6 Graph Theory

As well as attempting to predict future availability of BCH bicycles, we are also looking to develop our own router that will combine walking, cycling and London Underground paths into complete journeys suitable to users' requirements and preferences. Building the router requires an understanding of graph theory, which we introduce next.

2.6.1 The Basics

A *graph* G is a set of vertices V (also known as nodes) and a set of edges E (also know as arcs). An *edge* is a binary relationship between vertices (a, b) where $a, b \in G$. In this case a and b are known to be *adjacent*. If $a, b \in V$ and $a = b$ then the relationship (a, b) is called a *loop*. Edges can be *directed* or *undirected*. A directed edge distinguishes (a, b) from (b, a), whereas an undirected edge does not. A *cost* function $C(e)$ evaluates *weights* attached to an edge e, $\forall e \in E$, to return the expense of travelling along e.

A *simple graph* is one in which only a single edge can exist between any two vertices and no loops are allowed. A *multigraph* removes the first of these constraints. A *pseudograph* removes both. See Figure 2.2 for the illustration of each of these graphs.

2.6.2 Path finding

A *path* between a *source* vertex v_1 and a target vertex v_n, where $v_1, v_n \in V$, is a sequence of adjacent vertices $\{v_1, v_2, ..., v_n\}$. In a *connected* graph there exists a path between any two different vertices. If only a single path exists then this is the *optimal shortest path*. Otherwise, the optimal path is one of the lowest overall cost [13]. Methods for finding shortest paths in graphs have been studied extensively and a number of algorithms have been developed. The choice of an algorithm is influenced by the properties and types of graphs through which shortest paths will be looked for.

One of the properties governing the choice of an algorithm is its density D. For a simple undirected graph

$$D = \frac{2 \times \|E\|}{\|V\| \times (\|V\| - 1)} \tag{2.13}$$

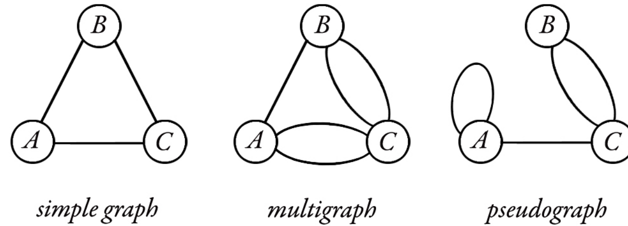

Figure 2.2: Graph types.

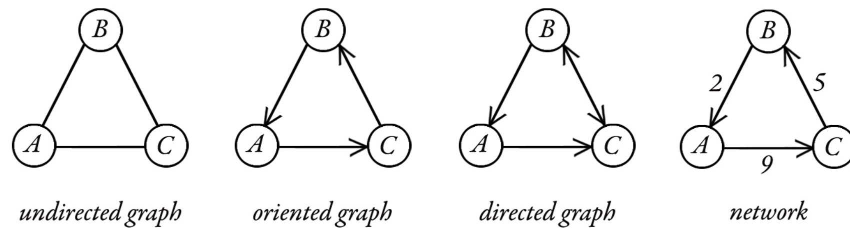

Figure 2.3: Graph edge types.

where $0 \leq D \leq 1$. $D = 1$ means every single vertex is connected to every single other vertex by an edge, in which case the graph is *maximal*. A *sparse* graph is one of low density.

Many shortest path algorithms have been developed, each one of varying time complexities that are normally governed by the challenges that different types of graphs present. In general, they can be divided into:

- non-informed search algorithms - so called *brute-force* searching - use no information about the likely 'direction' towards target vertex, instead only utilising the information already present in the problem description. Dijkstra's algorithm is an example [14]
- informed search algorithms - also know as *best-first* algorithms - attempt to establish some 'direction' to the search process using heuristics. Having to examine fewer vertices reduces the search space and as a result better running time performance is achieved

One of the most popular informed shortest path algorithms is the A* algorithm [13]. The algorithm improves on Dijkstra because it uses a *heuristic* function to estimate not just the cost of reaching the candidate node, but also the estimated

distance from the node to the target vertex. Formally, the cost associated with node k is given as a sum of two functions

$$f(k) = g(k) + h(k) \tag{2.14}$$

where $g(k)$ is the cost of reaching the node k from v_1 and $h(k)$ is a heuristic estimate of the cost from k to v_n. The A* algorithm finds the shortest path in a graph (if one exists) by expanding the lowest-cost node from among the candidate nodes - the successors to the latest nodes it was able to examine. To keep track of the vertices it visits, A* maintains a list of open nodes O, which is initialised with v_1. This list contains the candidate nodes and at each iteration a node in O with the lowest f cost is examined. As Algorithm 1 shows, A* terminates when the next node picked for examination is the target vertex v_n.

Algorithm 1 A* search algorithm for finding shortest path in a graph.

```
1: function FIND_SHORTEST_PATH(G, v1, vn, c, h)
2:     O = v1
3:     while O not empty do
4:         remove i ∈ O such that f(i) is least
5:         if i == vn then
6:             return path to i
7:         end if
8:         for all k ∈ children(i) do
9:             calculate h(k)
10:            calculate f(k)
11:            insert k into O ordered by f(k)
12:        end for
13:    end while
14:    fail
15: end function
```

In section 5.2 we will discuss our implementation of this algorithm in detail, including a small modification we hope will decrease the algorithm's search space further still. For now, we simply note that A* has been proven to be an optimal algorithm for finding a shortest path provided $h(k)$ is *admissible*, meaning it never overestimates the true cost of reaching target vertex v_n from node k, $\forall k \in V$ [13].

Chapter 3

System Architecture

Our cycling journey planner is written mostly in Python. We chose this language because of the relative ease with which it can manipulate large datasets. The author also had a personal interest in learning the language. Our journey planner is built of several components, which we now briefly describe.

Data feed handler

As mentioned in section 2.4.2 we have obtained access to a feed of updates about BCH docking station statuses. The `datafeed` package handles the functionality of listening for updates from TfL, downloading each one, processing its contents to update our database with the latest information and also restarting the update-downloading thread after system down time.

Database Manager

This journey planner relies heavily on information stored in databases. We wanted to make sure that our journey planner is:

- independent of the database type and version
- not overpopulated with strings representing SQL commands

We achieved this by utilising an object-relational mapper (ORM) provided by SQLAlchemy [34]. It provides the data mapper pattern, where classes can be mapped to the database tables. This decoupling of the object model from the database schema allowed us to almost completely avoid hand-written SQL. The disadvantage of any ORM in terms of slower database access and lack of support for complex queries did not outweigh the advantages of clearer code, database independence (in fact, we did have to shift from an SQLite3 database

to the departamental PostgreSQL database during the project and the switch was almost painless) and provision of database connection management (which we found useful as a number of data insertions lasting several hours had to be made and SQLAlchemy handled database connection recycling and others for us).

Data Loaders

Our bicycle prediction models and route calculators will need to frequently access various data held in the database. For example, the routing engine will require access to graphs of networks through which it is to find paths. It would be inefficient to build a new graph for every request so basic caching using module variable instantiation was implemented. Additionally, we cannot assume the underlying data is stored by ourselves - often, journey planners retrieve positional data from remote servers. This is why the methods for building such graphs are constructed with `data_loader` objects as parameters. Listing 3.1 shows how the graph building functionality combines caching of built graphs and independence of data source. A graph is built using a call similar to `tube_graph = build_graph(get_tube_data_loader())`, where `get_tube_data_loader()` is a method that returns an instance of a data loader that aggregates graph-related data from some source.

User Interface

Displays a map over which our journey suggestions are drawn. The modes of transport are color-coded. Additionally, this web-based user interface allows the users to specify the start time of their journey, its desired duration as well as their preferences towards being able to arrive at target on time, being certain about bicycles and free parking space availabilities at starting and finishing stations as well as preferred route busyness. The web-based interface sends a POST route request to our server which parses the route request parameters and initialises a route calculation.

Router

The router is responsible for calculating the single, overall journey that is most desirable to the user as per the received preferences. It fetches the required data using a number of different loaders that are designed similar to that in Listing 3.1. It uses NetworkX library for the manipulation of necessary networks, chosen for its Python language data structures for graphs, scalability (it is capable of handling graphs in excess of 10 million nodes and 100 million edges) and reasonable efficiency.

Listing 3.1: route_data_loader module used for building NetworkX graphs from nodes and edges data held in databse

```
1  class GraphLoader(object):
2      '''Abstract class for all graph loaders.
3      Child classes are expected to implement build_graph() method '''
4  
5      __metaclass__ = abc.ABCMeta
6  
7      def __init__(self, data_loader):
8          self.data_loader = data_loader
9          self.graph = None
10 
11     @abc.abstractmethod
12     def build_graph(self):
13         return NotImplementedError("Your child class should implement
               this method")
14 
15     def load_graph(self):
16         return NotImplementedError("Your child class should implement
               this method")
17 
18 _tube_graph = None
19 
20 class TubeGraphLoader(GraphLoader):
21 
22     def build_graph(self):
23         tube_graph = nx.Graph()
24         #steps for building the graph from data accessed through self.
               data_loader, omitted for readability
25         return tube_graph
26 
27     def load_graph(self):
28         global _tube_graph
29         if _tube_graph is None:
30             _tube_graph = self.build_graph()
31         return _tube_graph
32 
33 
34 def build_graph(graph_loader):
35     '''Common point of access for retrieving a networkx graph'''
36     return graph_loader.load_graph()
```

Chapter 4

Predicting Bicycle Availability

As described in chapter 2, we have access to two kinds of information about BCH

- the live bicycle availability data can tell us the current number of bicycles good for hire and the number of free docs into which bicycles can be parked
- the past cycle journeys data can tell us how many journeys were completed in and out of any docking station that was part of the system at the time of data collection, at various time intervals throughout the day

If were looking for current bicycle availability, we would simply have to look up the latest bicycle availability feed update the TfL have sent us for that station. Most of the time, however, we will instead be interested in predicting future bicycle availability. Even if our journey planner's users are wanting to immediately begin their journey, usually they will first have to reach, for example by walking, whichever docking station we suggest to them as the starting point of the cycling part of their overall journey - this will take some time. Similarly for the finishing docking station - we need to estimate the arrival time at that docking station and predict, for that future time point, the availability of a free docking space.

One of the approaches to predicting future bicycle availability at any given docking station is to estimate the number of people who will be picking up or dropping off bicycles at the docking stations between now and the future time point for which the availability prediction has been requested. Specifically, if we treat the number of pickups or dropoffs as discrete random variables and we heuristically divide the time between now and said future time point into a number of time intervals then, as outlined in section 2.5.3, we are interested in estimating the true, unobservable probability distribution of the number of

dropoffs and pickups that occur at the starting and finishing docking stations in each of those time intervals.

Existing Transport Models

If we compare a bicycle pickup to a passenger arrival at a public transport station and a bicycle dropoff to the arrival of the public transportation unit at that station, then there are a number of existing transport models we could apply to predict these numbers of dropoffs and pickups.

Normally, the presence of passengers at a public transport station at any given time point in the future is influenced by the knowledge of the arrival time of whatever mode of transport said passengers want to get onboard (a bus, for example, or a bicycle in our case). Thus past research [21] concentrated on clustering passengers into

- those who know the timetable
- those who do not know the timetable of arrivals

This clustering allowed for establishing the parametric form of the density models of arrivals of these two groups of passengers, since it was shown that passengers who do know the timetable arrive in a non-random pattern, whilst those who do not arrive at the stations in uniform distribution. A passenger arrival distribution curve for any station can then be calculated by combining these two groups of passengers.

Apart from passenger clustering, the existing transport models additionally rely on establishing public transport's headway [30]. Found to be the most important influence on passenger arrival distributions [25], it can be used to calculate the arrival median wait time at a public transport station - another factor influencing passenger arrivals. As with passenger clustering, these models depend on the existence of an arrival timetable for the transport mode in question.

However, there exists no schedule that would outline the presence of a bicycle at any given BCH docking station at different time points in the future. The presence of a bicycle at a docking station (equivalent to a bus arriving at a bus station) is instead influenced by the ratio of the number of drop-offs and pickups that occur between the latest time point when we had true data about the number of bicycles present at the docking station in question and the time in future for which we would like to estimate the bicycle availability. For example, if it is likely that there will be more pickups than drop-offs then it is less likely that a bicycle will be available.

4.1 Model Definition

Since we are unable to differentiate passengers based on their knowledge of the schedule of bicycle availability at different stations (a schedule does not exist), we could follow [21] in assuming that all passengers will arrive in uniform distribution. However, by investigating data described in section 2.4.1 we see that this is not true for bicycles. As an example consider Figure 4.1, which shows how the frequency of departures from four different stations varies throughout the day.

Since we cannot assume uniform distribution for our density estimator of the true distribution of the number of bicycle dropoffs and pickups, we look for a different parametric form for our density model.

Pickups and Dropoffs as Poisson Processes

Let us assume a typical scenario ω where there exists a docking station that contains several bicycles that can be picked up and a couple of free docks into which arriving bicycles can be dropped off. Since there are roughly 15,000 docking points across 570 docking stations and only 8,000 bicycles [2], this scenario is very common. Let us further define $N_t(\omega)$ as the number of pickups or dropoffs (generally, arrival events) that occur in the timer interval $[0, t]$ given the assumed scenario. Under certain assumptions, the following four conditions hold:

1. $N_0(\omega) = 0$
2. $N_t(\omega)$ increases by integer amounts, since it is impossible for two pickups or dropoffs to occur at exactly the same time. This is always true, since we can keep decreasing the time interval $[t, s]$ until only a single pickup or dropoff event occurs
3. $\forall t \geq 0, u > 0, N_{t+u} - N_t$ is independent of the history up to t, i.e. arrival events are independent of other such events that occurred in the past - the arrival of John at a docking station with the intention of picking up a bicycle is assumed to be unrelated to the arrival or Merry and Adam, who is instead terminating his journey at that docking station by dropping off a bicycle
4. $\forall t \geq 0, u > 0, N_{t+u} - N_t$ is independent of t, i.e. N, which we defined as the number of dropoffs or pickups (generally, arrival events) that occur in the future, is an independent random variable identically distributed over time

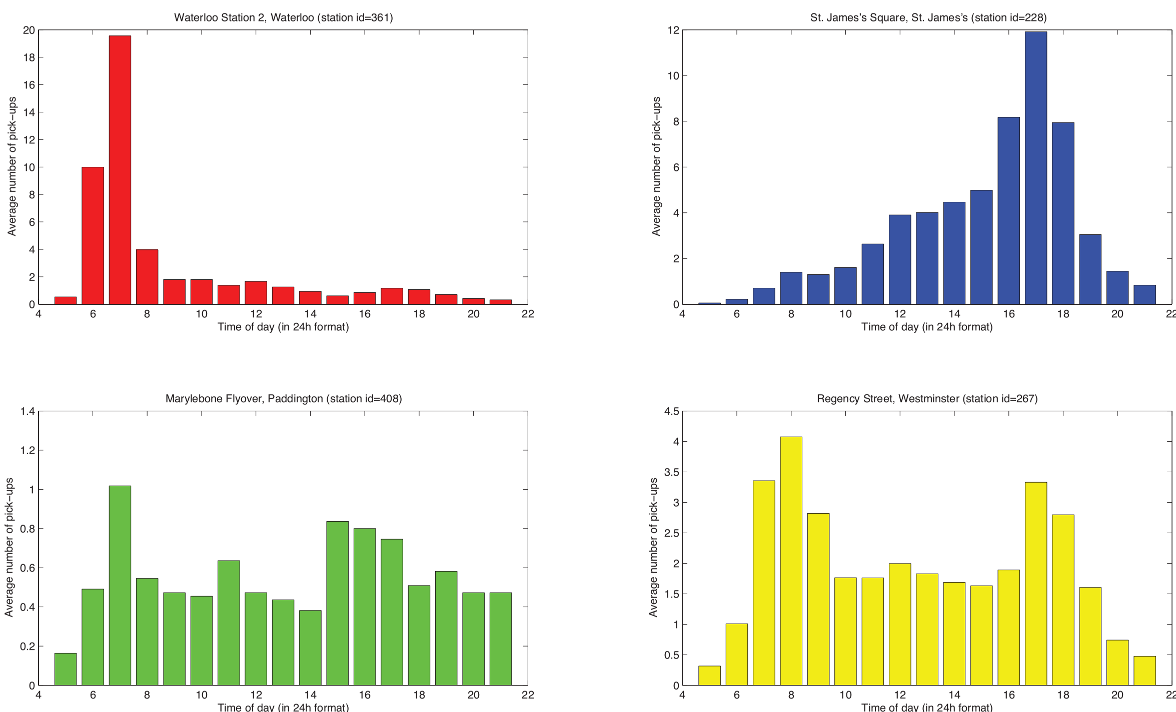

Figure 4.1: Average number of bicycle pick-ups at various docking station across the working hours of a weekday. We can see that the number of pick-ups varies differently throughout the day for different stations. At Waterloo, the morning rush hour passengers are most likely picking up the bicycles to connect to work. At St. James's Square, the evening rush hour passengers are most likely picking up the bicycles to connect to other modes of transport that will take them home. However, other stations, such as Marylebone Flyover, may have a more uniform distribution of pickups. It is also entirely possible for a station to have two intervals throughout the day when the pickup rate increases (see Regency Street station).

In this case we can refer to N as a *Poisson Process*. For non-negative integers k, the increments in N are found to follow the Poisson distribution we introduced in section 2.5.2 [35]

$$P(N_{t+u} - N_t = k) = \frac{(\lambda t)^k e^{-\lambda t}}{k!} \tag{4.1}$$

where λ is the expected number of pickups (equally, dropoffs) per period.

This result tells us that, under the assumptions outlined above, we can estimate the true, unobservable probability mass function of bicycle pickups and dropoffs using the Poisson distribution. We have therefore moved on from supposing the true, unobservable distribution of these is of uniform distribution and will now adopt exponential form for our density estimator. In section 4.2 we will show how the density estimation method described in section 2.5.4 can be used to find the parameter λ that characterises Poisson distributions.

Before we do this, we would like to discuss the implications of using Poisson distribution as our density estimator - do we think it is going to estimate the true distribution of the number of pickup and dropoff events at various times throughout the day well? This obviously depends on whether the assumptions of Poisson processes hold for these discrete random variables. The choice of Poisson distribution expresses our inductive bias about the true density of the number of pickups and dropoffs that occur in some time interval

- that there exists a single mode representing the most likely number of occurrences of an event
- that this density decays as we move away from the mode

This inductive bias motivates an important design decision in our approach to estimating the true density of the number of pickups and dropoffs that will occur in the future - rather than estimating the true density of pickups and dropoffs at docking stations throughout the entire day with just a single Poisson distribution, we instead consider the day to be split into a number of time intervals of smaller durations. It becomes our task to find a separate parameterization of the density estimator for each of the shorter intervals.

Estimating true, unobservable density of pickups and dropoffs that occur throughout the entire day with just a single Poisson estimator would be incorrect for two reasons: Firstly, consider the average number of pickups that occur at Regency Street station, shown in Figure 4.1. Clearly, the true distribution of pickups at this station is multi-modal. This goes against our inductive bias that the true probability mass function has a single (global and local) maxima and the fact that density of the number of pickups should decay in every direction away from the mean. Estimating the density of pickups for this station across an entire day with just a single distribution would require adopting a more sophisticated, multi-modal parametric form for our density estimator.

- However, the fact that a Poisson estimator is characterised by just one parameter λ and therefore of single degree of freedom is a big advantage to us, because it means we should be able to learn the value of λ from relatively small sample data set. This is important as the cycling journeys data has been collected in the first several months of BCH's operation, when the system was still gaining popularity and not all stations were active from the first day.
- Estimating the true density of the number of pickups and dropoffs for smaller intervals of the day solves this problem because in any sufficiently small time interval, the distribution of the number of pickups and dropoffs, from investigation, always seems to obey the two assumptions of our inductive bias

Secondly, consider the average number of pickups that occur at Waterloo station, shown in Figure 4.1. It tells us that the average number of pickups at this station throughout the entire day is roughly 48 (this is simply the sum of average number of arrivals in each 1 hour interval). As proved in the next section, this becomes the distribution parameter λ of the Poisson distribution estimating the number of pickups in that interval, shown in Figure 4.2. If we compare the predicted number of pickups that are likely to occur in the interval 5am-10pm of any day against the frequency density of the different number of pickups that we have on record for this station in our cycle journeys data (shown in Figure 4.3) we can see that the Poisson distribution does not estimate the true probability very well. In particular, the Poisson estimator gives low likelihood to the number of pickups being less than around 35 and more than 65, which by looking at Figure 4.3 we know is not entirely true.

However, the far bigger problem is that the most likely number of pickups to take place, as predicted by the Poisson estimator, is far higher than any average number of pickups we would expect in the time until the future time point for which we require a bicycle availability prediction. To explain, let us consider that a user has just put in a request for a journey they would like to start at their home near Waterloo in 1 hour. Using their house location and the location of the nearest docking station we can calculate the walking route to said docking station. Thus we know the exact time for which the bicycle availability prediction is to be made to be about 1/1.5 hours from now. Knowing that 48 pickups are likely to take place a day, we could divide this into the number of pickups likely to take place every hour and combine this similar reasoning about likely number of dropoffs and our knowledge of current bicycle availability, which we receive as updates from TfL every 3 minutes. However, since a single Poisson distribution is unable to describe the true density of the number of pickups and dropoffs that are likely to take place throughout the course of the day, our prediction is not likely to be accurate.

As before, the solution is to use the Poisson distribution as our estimator of choice but instead attempt to estimate the true number of pickups and dropoffs that will take place for much smaller time intervals. If our inductive bias is

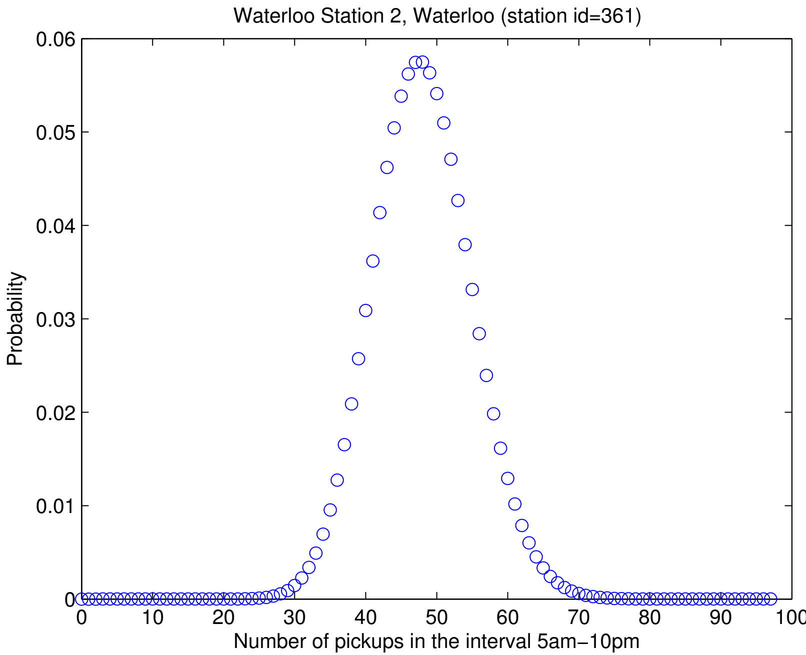

Figure 4.2: Per Figure 4.3 the average number of pickups in the interval 5am-10pm is 48.

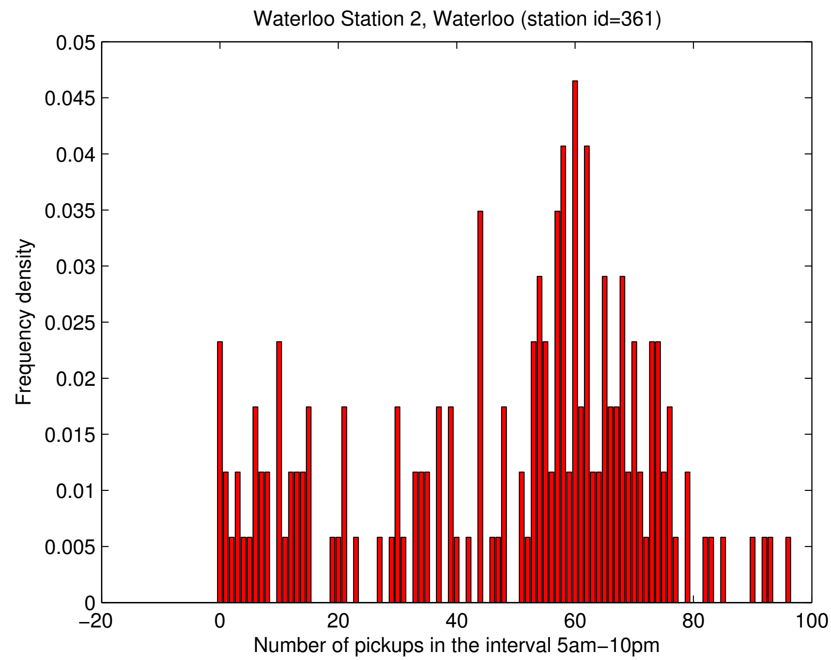

Figure 4.3: Frequency density of the number of pickups between 5am and 10pm. For example, of the 8261 journeys started at this station across 172 days, there were $0.035 \times 172 = 6$ days when the number of pickups was 44.

correct, the Poisson estimator should then perform well. However, if the true density within each interval does not have these properties then our estimator will perform very poorly. We hope that combining our estimations about true density in each smaller interval will guide us towards more accurate predictions for future time points. In section 4.3 we motivate the chosen duration for these intervals, as well as introduce two methods which take advantage of this approach to make predictions about future bicycle availability.

4.2 Parameterizing the Model

As mentioned in previous section, we are interested in estimating the true, unobservable probability mass function of the number of pickup and dropoff events that occur at every docking station at different time intervals throughout the day by fitting a Poisson distribution to the samples of the numbers of pickups and dropoffs that have previously occurred for those stations and time intervals. These numbers can be calculated from the historical cycle journeys data set described in section 2.4.1 and, as explained in section 2.5.3, we consider that they must be discrete random variables distributed according to the true probability mass function since they are real samples that have been drawn from it.

Density Estimation in Practice

For every docking station and every time interval throughout the day (discussed later), we need to establish two distribution parameters:

- λ_p parameter that characterises the Poisson distribution describing the probability of different number of pickups that occur for that docking station and time interval
- λ_d parameter that characterises the Poisson distribution describing the probability of different number of dropoff that occur for that docking station and time interval

One approach for finding these parameters is to find their value that will maximise the probability of sample data $\mathbf{x}$. This can be done with maximum likelihood estimation, introduced in section 2.5.4.

Formally, we can rewrite our result from (2.11) as

$$\hat{\lambda}_{MLE} = \arg\max_{\lambda}(\sum_{i=1}^{n} \ln \ell(x_i|\lambda)), \forall x_i \in \mathbf{x}$$

In our model, the likelihood of a single sample data point is given by the Poisson distribution. If we set k from 4.1 equal to 1, the above formula can be written

as

$$\begin{aligned}\hat{\lambda}_{MLE} &= \arg\max_{\lambda}(\sum_{i=1}^{n} \ln(\frac{\lambda^{x_i} e^{-\lambda}}{x_i!})), \forall x_i \in \mathbf{x} \\ &= \arg\max_{\lambda}(\text{-n}\lambda + (\sum_{i=1}^{\text{n}} x_i)\ln(\lambda) - \sum_{i=1}^{n} \ln(x_i!)), \forall x_i \in \mathbf{x} \qquad (4.2)\end{aligned}$$

Often, instead of maximising the log likelihood, we minimise the negative log likelihood, then referred to as an error function. Finding λ that minimises the error function can be done using gradient descend. However, here we can apply a more direct approach of solving for λ by taking the derivative of the error function with respect to λ and equating to zero. This gives us the maximum likelihood estimator for a Poisson distribution

$$\hat{\lambda}_{MLE} = \frac{1}{n} \sum_{i=1}^{n} x_i \qquad (4.3)$$

This result implies the estimate of true λ is in fact the sample mean, i.e. the mean of the observed number of pickups (dropoffs, similarly) for the station and time interval of interest. This is just what we would expect. By definition, the sample mean is also an unbiased estimate of true λ. The second order derivative of the log likelihood from (4.2) is always positive, thus we know we have found minimum of the error function.

We note that the BCH scheme has expanded since May 2011, when our historical cycle journeys data stopped being collected, and this means we will not be able to directly calculate the number of pickups and dropoffs for the stations that became active since. We solve this problem by assuming that the number of dropoffs and pickups that occur in any time interval of the day at a docking station which did not exist at the time the cycle journey data was collected are the same as those of the nearest docking station that did exist at the time.

4.3 Making Predictions

We have so far been able to establish the desired parametric form of the density model estimating the true, unobserved distributions of the number of pickups and dropoffs that occur for every station and each time interval of the day. We have then discussed a method for finding the parameters of each of these distributions using the sample data we have been able to obtain. Now we would like to discuss two methods that use these parametrised models to predict bicycle and parking space availability at any station at any time of the day.

In this section we will use the following notation:

- t represents time

- x_t is the number of bicycles present at a docking station at time t that are good for hire
- b_t is the number of empty docs present at a docking station at time t that are functional
- p_t^s is the number of pickups that occur at a docking station between times t and s
- d_t^s is the number of dropoffs that occur at a docking station between times t and s
- $pmf_{d_t^s}(x)$ is the probability mass function describing the probability of d_t^s being exactly equal to x
- $pmf_{p_t^s}(x)$ is the probability mass function describing the probability of p_t^s being exactly equal to x

Trivally, the following logical equality holds

$$x_s > 0 \iff p_t^s < d_t^s + x_t \tag{4.4}$$

Let us set t to be the time we receive the request for a route and s to be the time we estimate the person will reach a docking station (found by considering user-specified journey start time and the duration of any routes that are needed for the user to reach said docking station). The above equality tells us that to predict if there will be a bicycle available for hire at s, we need to know the current number of bicycles available at that station and additionally be able to estimate the number of pickups and dropoffs that will occur between t and s. We know the former from the updates TfL sends us every three minutes. Below, we describe two methods for establishing the latter using the Poisson estimator we have been discussing so far.

4.3.1 Using Cumulative Distribution Function

The simplest approach to estimating the number of pickups and dropoffs that will occur between t and s is to fit the Poisson density estimator to all samples of cycle journeys that begin and end, respectively, at that docking station in that time interval. As we have shown in previous section, the λ parameter of the Poisson distribution estimating the true probability mass function of the number of pickups and dropoffs that occur can be calculated as the mean number of each type of journeys.

We are thus looking to calculate $P(p_t^s < d_t^s + x_t)$. We can express it using the cumulative distribution function we have previously defined in (2.3) remembering that, since d_t^s is itself also a random variable, we need to consider its

probability too:

$$\begin{aligned} P(x_s > 0) &= P(p_t^s < d_t^s + x_t) \\ &= \sum_{d_t^s} pmf_{p_t^s}(d_t^s + x_t) \times pmf_{d_t^s}(d_t^s) \end{aligned} \tag{4.5}$$

Of course, d_t^s can take on any non-negative value - we therefore do not know the value of x_{k-1} from (2.3) and will instead terminate the calculation when the value of $pmf_{d_t^s}(d_t^s)$ becomes negligibly small. The resulting algorithm is shown as pseudo-code in Algorithm 2.

Algorithm 2 Predicting bicycle availability - single Poisson

```
 1: function PROB_BIKE_AVAILABLE(station_id, request_dt, journey_start_dt)
 2:     prob = 0
 3:     dropoffs = 1
 4:     acc_error = 0.00001
 5:     curr_num_bikes = get_curr_num_bikes(station_id)
 6:     mean_pickups = get_pickups_mean(station_id, request_dt,
 7:                                                  journey_start_dt)
 8:     mean_dropoffs = get_dropoffs_mean(station_id, request_dt,
 9:                                                  journey_start_dt)
10:     prob_dropoffs = poisson.pmf(dropoffs, mean_dropoffs)
11:     while prob_dropoffs > acc_error do
12:         cdf_pickups = poisson.cdf(dropoffs + curr_num_bikes, mean_pickups)
13:         prob += cdf_pickups × prob_dropoffs
14:         dropoffs += 1.0
15:         prob_dropoffs = poisson.pmf(dropoffs, mean_dropoffs)
16:     end while
17:     return prob
18: end function
```

The calculation of the probability of there being a free parking space at the finishing docking station follows a similar methodology

1. we wish to find $P(d_t^s < p_t^s + b_t)$
2. we calculate $\sum_{p_t^s} pmf_{d_t^s}(p_t^s + b_t) \times pmf_{p_t^s}(p_t^s)$

The *request_dt* and *journey_start_dt* in Algorithm 2 are, in reality, parameters to the model's constructor. The pseudo-code omits these and other implementation details for readability.

The model's predictive performance is evaluated in section 6.1. However, we note here the expense of this algorithm:

1. in terms of database accesses:

- the time interval for which we will be estimating the number of pickups and dropoffs is unknown - it is based on the user-defined journey_start_dt and the desired journey
- this means we have to calculate the sample mean of the number of cycle journeys that start and end at the docking station of interest for every request the user makes
- since the availability of a free parking space at the finishing docking station is dependant on route duration, we will have to perform a separate calculation of this for every route we wish to suggest to the user

2. in terms of algorithm complexity:
   - Basic implementations of get_pickups_mean(station_id, request_dt, journey_start_dt) and get_dropoffs_mean(station_id, request_dt, journey_start_dt) will run in $O(n)$ to find the cycle journeys that concern the docking station and time interval of interest. In section 7.3 we suggest a useful method for decreasing the complexity of this search, but now it is evident the method will run relatively slowly

For these reasons we have developed another model that uses sample means of the number of pickups and dropoffs that can be accessed $O(1)$.

4.3.2 By Sampling the Density Estimator

Previously, we have not been able to efficiently obtain the sample mean of the number of journeys beginning and finishing at the docking stations of interest, from which the expected values can be calculated, since the time period for which these were to be calculated was unknown. We now present a second model. The model is motivated by the fact that since the data set containing historical cycle journeys is static, we can divide the 24 hours of a day into a number of intervals of certain duration and pre-compute the sample means for every station and every time interval. Storing this information in a database and caching it at run-time allows us to look it up in constant time.

To motivate the chosen duration for these intervals (and thus effectively the number of them), we first introduce a further improvement to the model outlined in previous section. We noted at the end of section 4.2 that the BCH system has been extended a number of times since our historical cycle journeys data set was collected. As the system expands and becomes more popular, we would expect the true numbers of pickups and dropoffs to have changed since our data was collected.

The only data that we have access to which would characterise the BCH system as it functions today has been described in section 2.4.2. It cannot tell us anything about current number of pickups or dropoffs at docking stations at

different times of the day. However, since we keep being updated about the current number of working bicycles and empty docks at every station, we can track the average change in both of these for any interval of a day whose duration is a multiple of 3 minutes. We can therefore attempt to account for increased popularity and usage of the BCH system by scaling the sample means we were able to calculate from the data collected between 2010 and 2011 with these differences.

To be able to scale the sample means of the number of pickups and dropoffs we must introduce some new notation and a new assumption:

- $p^s_t('10)$ is p^s_t calculated from our historical cycle journeys. Similarily for $d^s_t('10)$
- $p^s_t('12)$ is p^s_t for cycle journeys that are being made under current size and popularity of BCH. This is unknown since we do not have any live updates on cycle journeys. But we would like to use them as input to our density estimation techniques instead of $p^s_t('10)$ so that we can estimate the true distribution of the number of pickups and dropoffs currently being experienced by the docking stations
- It follows that the average change in the number of bicycles present at a docking station in the time interval $[u, t]$ can be expressed as

  $$\hat{x^t_u} = d^t_u('12) - p^t_u('12) \tag{4.6}$$

  where $u < t$ and we can calculate $\hat{x^t_u}$ as $x_t - x_u$, where x_t and x_u are both known from the live updates TfL provides us
- We re-iterate the assumption about expected number of pickups and dropoffs for a station that did not exist when our cycle journeys data was collected, mentioned at the end of section 4.2
- We additionally assume that the following equality holds

  $$\frac{p^s_t('10)}{p^s_t('10) + d^s_t('10)} = \frac{p^s_t('12)}{p^s_t('12) + d^s_t('12)} \tag{4.7}$$

  that is the ratios of the number of pickups to all 'arrival events' remained constant throughout time, even if the absolute numbers might have increased

In the above we have two unknowns and two equations. Solving the simultaneous equations, we are able to calculate the estimated sample means of pickups and dropoffs currently being experienced at every docking station in the following manner

$$p^s_t('12) = \frac{\hat{x^t_u} \times p^s_t('10)}{d^s_t('10) - p^s_t('10)} \tag{4.8}$$

$$d^s_t('12) = \frac{\hat{x^t_u} \times d^s_t('10)}{d^s_t('10) - p^s_t('10)} \tag{4.9}$$

which is what we would expect as the formula simply scales the number of pickups/dropoffs as calculated from our historical cycle journeys data by the ratio between mean change in the number of bicycles available at the docking station that occurs now and the change mean change that was occurring when cycle journeys data was collected in 2010-2011.

What should be the duration of the intervals that we will split a day into? This decision is a trade-off between wanting to decrease their duration, so that we can estimate the local density well, and increasing their duration, such that we have a higher number of historical cycle journeys on which to train our model. To understand the second point, consider setting the duration of said interval to just three minutes - few cycle journeys will fall into this interval, which is not desirable. We settle for a duration of 15 minutes as, looking over historical cycle journeys data, at least a couple of pickups and dropoffs occur every 15 minutes. The time interval is still short enough for our density model to hopefully estimate the true density well.

Sample Mean Change in Bicycle Availability

We would like to scale the sample means as accurately as possible. By collecting updates about x_t every three minutes and from them calculating the average mean change for every 15 minute interval - every day - we are able to calculate one $\hat{x_u^t}$ for each of the 96 intervals in a day. As the days pass, rather than using whatever latest value of $\hat{x_u^t}$ we have for the interval $[u, t]$, we would instead like to learn from all the samples of $\hat{x_u^t}$ that we have observed so far.

We calculate $\hat{x_u^t}$ as a running mean - during every update from TfL, we work out the latest value of $d_u^t('12) - p_u^t('12)$, where t is the starting time of the interval within which the update time falls and u is the starting time of the previous 15-minute interval. We then calculate a mean of this latest evidence and all the samples we have witnessed for the interval $[u, t]$ in the previous days. To prevent having to consider (thus store) all historical values of $d_u^t('12) - p_u^t('12)$, we do this calculation using a stable one-pass algorithm.

Formally, assume we have observed $n - 1$ samples of $\hat{x_u^t}$ before examining this latest update. $\hat{x_u^t}[n-1]$ is the mean change in $\hat{x_u^t}$ across these samples. If we analyse the latest update and calculate its $\hat{x_u^t}$ as per (4.6), the new sample mean change in x_u^t after n samples is defined as

$$\begin{aligned}\hat{x_u^t}[n] &= \frac{(n-1) \times \hat{x_u^t}[n-1] + \hat{x_u^t}}{n} \\ &= \hat{x_u^t}[n-1] + \frac{\hat{x_u^t} - \hat{x_u^t}[n-1]}{n}\end{aligned} \tag{4.10}$$

Recalculating the average change in bicycle availability in this manner means our density model is using the *latest* data available to us, resulting in a continuously

improving density model.

We now have a time-efficient method for calculating sample means of pickups and dropoffs of every station for every 15 minute interval in a day, scaled to cater for increased size and popularity of BCH. As before, let us set t to be the time we receive the request for a route and s to be the time we estimate the person will reach a docking station (found by considering user-specified journey start time and any routes that are needed for the user to reach said docking station). In this approach, we will predict the availability of a bicycle at a docking station at time s by predicting the number of bicycles present at the end of each 15 minute interval that starts inside $[t, s]$. The latter is done in three steps:

1. We know the number of bicycles present at the station at t - this is the x_t we get in the latest update from TfL.
2. For every 15 minute time interval in $[t, s]$:
   (a) we draw from each Poisson estimator (estimating the number of pickups and dropoffs that will occur at that docking station in that interval) a random value for the number of pickups and dropoffs
   (b) using the drawn values and the number of bicycles from the previous interval, we calculate the new, predicted number of bicycles at the end of that interval
   (c) we set this as the new value of x_k, where k is the starting time of the next interval, to carry the predicted value through to next iteration of the algorithm
3. At s, x_s is our predicted number of bicycles

At each iteration of the algorithm we expect the points generated from an appropriate Poisson density model to fall at some positive distance from the sample mean of the number of pickups or dropoffs that have historically occurred in that interval. As we repeat steps 1-2-3 a large number of times, we can record how many times the predicted number of bicycles at s was strictly greater than zero. By dividing this value by the total number of runs, we obtain the probability that there will be a bicycle available at s.

The above method is summarised as Algorithm 3. Figures 6.1 and 6.2 show the first 10 traces of this method as it tries to predict the number of bicycles present at a station a number of time intervals into future.

As with the first model, the calculation of the probability of there being a free parking space at the finishing docking station follows a similar methodology:

1. we know the number of functioning free parking docks available at a station from the latest update from TfL.
2. for every 15 minute time interval in $[t, s]$:

Algorithm 3 Predicting bicycle availability - sampling the Poisson

```
1:  function PROB_BIKE_AVAILABLE(station_id, request_dt, journey_start_dt)
2:      counter = 0
3:      num_iterations = 1000
4:      timestep = 15 minutes
5:      next_interval_start_dt = request_dt
6:      curr_num_bikes = get_curr_num_bikes(station_id)
7:      for i = 1 → num_iterations do
8:          while next_interval_start_dt < journey_start_dt do
9:              num_pickups = get_scaled_pickups_mean(station_id, request_dt)
10:             num_dropoffs= get_scaled_dropoffs_mean(station_id, request_dt)
11:             drawn_pickups = poisson.rvs(num_pickups, size=1)
12:             drawn_dropoffs = poisson.rvs(num_dropoffs, size=1)
13:             curr_num_bikes = max(min(curr_num_bikes + drawn_dropoffs
                                - drawn_pickups, num_docks_all), 0)
14:             next_interval_start_dt += timestep
15:         end while
16:         if curr_num_bikes > 0 then
17:             counter += 1
18:         end if
19:         i += 1
20:     end for
21: end function
```

   (a) we draw from each Poisson estimator (estimating the number of pickups and dropoffs that will occur at that docking station in that interval) a random value for the number of pickups and dropoffs

   (b) using the drawn values and the number of empty docks from the previous interval, we calculate the new, predicted number of empty docks at the end of that interval

   (c) we carry this predicted value through to next iteration of the algorithm

3. we terminate when we reach an interval that contains s.

As with bicycle availability, we divide the number of times we predicted the number of free docks to be greater than zero by the total number of iterations we chose to perform to obtain the probability of there being an empty docking space at the station of interest at s.

Chapter 5

Routing

As mentioned in the introduction, the purpose of our journey planner is to construct suggestions of origin-destination trips in a multi-mode transport network. We are particularly interested in combining walking, cycling and travel on the London Underground. This is because our journey planner should try to find a cycling route that would satisfy user-defined preferences, such as trip duration, but when this is not possible a mix of walking, cycling and London Underground routes is to be suggested as well. The calculation of these trip suggestions has to be guided by preferences the users are able to specify and the availability of bicycles, the modelling of which we have already discussed in the previous chapter.

The problem of finding the most desirable journeys that combine walking, cycling and travel on the London Underground is two-fold:

1. firstly, we need to be able to separately calculate walking, cycling and tube routes between any two coordinate positions on the map of Greater London, taking into account a number of *hard-constraints* that we have no control over, such as road accessibility, and *soft-constraints*, which are user's preferences for the journey such as the desirable busyness of the suggested route

2. secondly, we need to be able to find a combination of the above such that the overall journey incorporates as much cycling as is possible considering the desired trip duration, and walking and travel on the London Underground in case the cycling on its own would take too long

The first three sections of this chapter will describe our solution to the first problem. In the fourth section we will describe how they can then be brought together to form an integrated London journey planner. In section 5.5 we examine a small optimisation we hope will improve the performance of our routing algorithm from section 5.2 further still.

graph	node attributes	edge attributes
tube_graph	station name, its coordinate position	source station, target station, edge length, time to travel by train, connecting lines
london_graph	node id, its coordinate position	source node, target node, edge length, car accessibility, bicycle accessibility, foot accessibility, geometry[1]

Table 5.1: Summary of nodes and edge attributes of *tube_graph*, *london_graph* and *bike_graph*.

5.1 Graphs

In order to be able to find a path that connects a starting point to the finishing point, we first need to be aware of the structure of the network through which this path is to be found. Taking the London Underground as an example, in order to be able to find a route from South Kensington station to Green Park station we need to know that South Kensington station is connected to Sloane Square station, itself connected to Victoria station, from which we can reach Green Park by Victoria Line. We would also like to know that there is an alternative route involving Piccadilly Line which allows us to reach Green Park without having to change at Victoria. Being able to examine alternative paths is useful because then we can compare these alternatives for their desirability to the user. Similarly, we need to know how streets, footpaths and other street-level features of the network representing a city are connected such that we can find sensible walking and cycling routes.

The most suitable way of encoding the above relationships is by representing the London Underground and Greater London networks as *graphs*, introduced in section 2.6.1. Table 5.1 lists the information held by the node and edge components of each of the resulting graphs. The data that is used for the attributes of both graphs has been previously described in sections 2.4.2 and 2.4.2.

5.2 Modified `astar_path`

Having defined the required graphs, we can now turn towards the problem of finding paths through them. Let us define C to be a *cost function* of the attributes of an edge in a graph that can tell us how undesirable travelling down that edge is under user-specified preferences (we will discuss how we can incorporate user preferences into our path finding using said cost function in

[1]Often, street-level links are not straight lines.

the next section) Our path-finding problem can then be seen as the problem of finding a path between the source and destination points that will minimise the total cost (sum of the individual costs of each edge in the route). This is then just a *shortest path* problem and there exists a number of algorithms designed to solve it. We have decided to use the A* algorithm, introduced in section 2.6.2, for the following reasons:

- we wanted an algorithm that will calculate the route efficiently - the time A* takes to find a path is proportional to the number of nodes in the resulting route and not the number of nodes in the graph being searched - it is a non-exhaustive search algorithm. This is desirable since the graph representing the Greater London area is of considerable size, as discussed later in this section, and this is the reason we were not interested in brute-force approaches to search
- the edge attributes we have listed in Table 5.1 will be the input to the cost function described in section 5.3. None of these attributes can ever be negative
- we have easy access to a heuristic that can inform our search - when examining nodes that could be part of the path (lines *8-12* in 2.6.2), we can calculate the great-circle distance of each node to the target vertex. We know that this would form an admissible heuristic since it is impossible to reach the target node any shorter way, particularly so in a city full of old, twisty roads and footpaths. Access to an admissible heuristic means we would like to (in the average case) improve on the time complexity of Dijkstra's algorithm
- both of our graphs are not dense, in the sense described in section 2.6.2. We find that with 267 stations and 309 connections, the density of *tube_graph* is 8.7×10^{-3}, whilst with 221,233 nodes and 285,798 links, the density of the *london_graph* is expectedly even smaller at 1.7×10^{-5}. Low density means we have little reason to be interested in shortest path algorithms for dense graphs such as the Floyd-Warshall algorithm

In section Chapter 3 we have motivated the use of NetworkX library. Its graph object stores information as dictionaries of dictionaries - this data structure allows the library to be very scalable and capable of handling graphs far larger than any we will be dealing with. It also means we can obtain fast direct access to the graph data using subscript notation, which is important since we will need to be frequently accessing edge attributes when evaluating edge costs, as described in section 5.3. A dictionary can hold any hashable object and thus NetworkX is very flexible when it comes to the definition of node objects - this allows us to store the node name/id together with its coordinate data as, for example, a tuple. Since our nodes will always be unique, NetworkX's data structuring allows us access to node and edge attributes in $O(1)$.

Our decision to use NetworkX was additionally motivated by the fact that it has built-in functionality for finding shortest paths in graphs using the A* algorithm,

which, as discussed above, we would like to employ for our journey planner. The `astar_path` function is presented in Listing 5.1. We like their implementation for a number of reasons:

- it uses the *heap queue* algorithm, also known as the priority queue algorithm, for storing the list of already-examined nodes (the O from section 2.6.2). Heaps are binary trees for which every parent node has a value less than or equal to any of its children. This gives us ability to lookup node of lowest f cost in $O(1)$. Using binary trees is also an improvement on storing nodes as an ordinary list as insertions and deletions are now of the order $O(log_2 n)$ instead of $O(n)$
- `astar_path` never actually removes from O the nodes it examines. Instead, *explored* is used to keep track of previously examined nodes. When a shorter path to some previously seen but not yet examined node has been found (see lines *35-38* in Listing 5.1), rather than deleting the old entry and inserting a new one, which is costly, the same node is added to the queue again, but with the lower cost. Lines *36-38* ensure that the old, higher-cost path to that node is never investigated again
- the heuristic function is a parameter to the algorithm so it can be customised

However, we are unable to use the source implementation of `astar_path` as-is for two reasons:

1. the native implementation of `astar_path` allows us to select only one of the edge attributes (it uses *weight* as the default attribute) for evaluation of an edge cost. Our cost function, as defined in section 5.3, will instead be interested in all the edge attributes in the given graph
2. the algorithm does not handle finding the shortest path in multigraphs. This is a big problem for us - whilst *tube_graph* is a simple, undirected graph, *london_graph* needs to be a directed multigraph

The first issue is specific to our problem domain, where we give the user an opportunity to prioritise the importance of edge attributes, as described in section 5.3. To enable `astar_path` to consider multiple attributes when evaluating the cost of travelling along an edge, we extract the cost-evaluation functionality from `astar_path` altogether (see lines *1,38,40* in Listing 5.2). The cost function is now a parameter to the algorithm just like the heuristic function h was before. It takes as input all the attributes of the edge being currently evaluated and returns a number. Because of the efficient edge attribute lookup, the operation maintains the constant time complexity in the case of a single edge (line *40*).

The second problem needs some explanation. The requirement for edge direction in *london_graph* stems from the fact that the attributes which apply to one direction of the *london_graph* may not necessarily apply in the opposite direction. An edge may be car or bicycle accessible one way and not the other

Listing 5.1: NetworkX's `astar_path` function for finding shortest path in graphs using A* algorithm

```
1  def astar_path(G, source, target, heuristic=None, weight='weight'):
2
3      if G.is_multigraph():
4          raise NetworkXError("astar_path() not implemented for Multi(Di)
               Graphs")
5
6      if heuristic is None:
7          def heuristic(u, v):
8              return 0
9
10     queue = [(0, hash(source), source, 0, None)]
11     enqueued = {}
12     explored = {}
13
14     while queue:
15         _, __, curnode, dist, parent = heappop(queue)
16
17         if curnode == target:
18             path = [curnode]
19             node = parent
20             while node is not None:
21                 path.append(node)
22                 node = explored[node]
23             path.reverse()
24             return path
25
26         if curnode in explored:
27             continue
28
29         explored[curnode] = parent
30
31         for neighbor, w in G[curnode].items():
32             if neighbor in explored:
33                 continue
34             ncost = dist + w.get(weight, 1)
35             if neighbor in enqueued:
36                 qcost, h = enqueued[neighbor]
37                 if qcost <= ncost:
38                     continue
39             else:
40                 h = heuristic(neighbor, target)
41             enqueued[neighbor] = ncost, h
42             heappush(queue, (ncost + h, hash(neighbor), neighbor,
43                              ncost, curnode))
44
45     raise nx.NetworkXNoPath("Node %s not reachable from %s" % (source,
           target))
```

and we need to consider this information when finding the path. This is not the case with the *tube_graph*, where the train has to travel the same distance and takes the same amount of time in both directions between any two stations. *london_graph* additionally needs to be of multigraph type as it is entirely possible for any two nodes to be connected by more than one edge in either direction. A simple example is a one-way road that splits around a pedestrian crossing - both lanes/edges connect the same node, but we have a choice which one to travel along and need to make an informed decision rather than pick one of the available edges at random. The simple solution would be to convert a multigraph to a simple directed graph before path-finding by investigating every single pair of nodes $\in$ *london_graph*, evaluating it using the injected cost function and deleting every edge other than that of minimal cost. However, consider the case when we want to find the path from vertex v_1 to one of its neighbours - in this case having to perform the conversion would be a significant overhead.

Our solution is more time efficient. Consider some node i and one of its successors k. The existence of multiple edges to k does not have to negatively impact our ability to find the shortest path if we are careful to check for it when we expand i in lines *31-43* of the original `astar_path`. The resulting algorithm is presented in Listing 5.2.

Because this is not an issue with our problem domain but instead a general deficiency of the current implementation of `astar_path`, we contacted the lead developer of NetworkX and will be submitting our solution to the second issue as a contribution to the future release of the library [22].

5.3 Cost Models

Using our journey planner's web interface, shown in Appendix A, the user, apart from picking the start and finish locations for their desired journey, is also able to express how important it is to them that:

1. they are able to complete their journey on time
2. they are able to pickup and dropoff their bicycle at the starting and finishing stations that they have been directed towards by our journey planner
3. that the calculated route is safe in terms of congestion level and road type

To find a journey that will aim to be most desirable to the user, we must take into account the relative importance of each of these factors when exploring the graph for a possible passage from start to finish point. Luckily, we have all the data we need - when finding shortest paths through *tube_graph*, we can investigate the edge's length and travel duration to satisfy user's settings of the first two preferences. *london_graph* additionally stores information on bicycle accessibility that will help us define how desirable is it to cycle along each edge

Listing 5.2: Our A* algorithm for finding shortest paths in NetworkX graphs

```
def astar_path(G, source, target, heuristic_func=None, cost_func=None
    ):

    if heuristic_func is None:
        def heuristic_func(s_node, t_node):
            return 0

    if cost_func is None:
        def cost_func(edge_attributes):
            return 0.5

    queue = [(0, hash(source), source, 0, None)]
    enqueued = {}
    explored = {}

    while queue:
        _, __, curnode, curr_cost, parent = heappop(queue)

        if curnode == target:
            path = [curnode]
            node = parent
            while node is not None:
                path.append(node)
                node = explored[node]
            path.reverse()
            return path

        if curnode in explored:
            continue

        explored[curnode] = parent
        curr_h = heuristic_func(G.node[curnode], G.node[target])

        for neighbor, edge_attributes in G[curnode].items():
            if neighbor in explored:
                continue

            if G.is_multigraph():
                cost_to_reach_neighbour = min(map(lambda edge_key:
                    cost_func(edge_attributes[edge_key]),
                    edge_attributes.keys()))
            else:
                cost_to_reach_neighbour = cost_func(edge_attributes)

            ncost = curr_cost + cost_to_reach_neighbour
            if neighbor in enqueued:
                qcost, h = enqueued[neighbor]
                if qcost <= ncost:
                    continue
            else:
                h = heuristic_func(G.node[neighbor], G.node[target])

            pathmax_h = max(h, curr_h-cost_to_reach_neighbour)
            enqueued[neighbor] = ncost, pathmax_h
            heappush(queue, (ncost + pathmax_h,
                             hash(neighbor),
                             neighbor,
                             ncost,
                             curnode))

    raise nx.NetworkXNoPath("Node %s not reachable from %s" % (source
        , target))
```

in the graph. Because *london_graph* provides no information on time taken to travel along each edge, we calculate it manually from edge length using the average walking and cycling speeds [23] [7] when examining walking and cycling routes respectively.

However, finding a path through a multiply constrained graph is a NP-complete problem. Our solution is to develop a cost function C that maps the multiple constraints in each graph edge attribute into a single cost. The single cost allows us to examine the attractiveness of travelling along the edge in the same way as we have been doing so far. We can thus optimise against user preferences towards multiple aspects of a route whilst maintaining the path searching as a P-complete problem that we can solve as described in the previous section. Formally, the cost of travelling an edge from some vertex a to its neighbour b is defined as

$$C(a,b) = \sum_{i=1}^{\#edgeattributes} w_i \times c_i(a,b) \tag{5.1}$$

where c_i returns the cost of travelling along the edge (a,b) in terms of ith attribute of that edge. The w_i are the weights which let us calculate the total cost of an edge in terms of weighted costs in each of that edge's attributes, where the weights, thought of as expressing relative importance of the cost in each attribute, are set by the user at request-time using the sliders shown in Appendix A. For example, sliding the top slider to the right increases w_i such that, from among other attributes, longer travel time makes travelling along that edge less attractive by a relatively bigger amount.

Before choosing a suitable cost function, we note that any cost function for travelling along the edge (a,b) should have the following properties [8]

1. If $\gamma_i = 0, \forall \gamma_i \in \psi$, where ψ is the set of attributes of edge (a,b), then $c_i(\psi) = 0$
2. $\frac{\partial c_i(\psi)}{\partial \gamma_i} > 0$ if $\gamma_{1i} > 0$, i.e. the cost of travelling along an edge of 'no resistance' should be zero
3. $\frac{\partial c_i(\psi)}{\partial \gamma_i} \geq 0$ if $\gamma_i = 0$, i.e. c_i should be increasing in each of the edge's attributes
4. $\frac{\partial^2 c_i(\psi)}{\partial \gamma_i^2} < 0$, i.e. the cost function c_i should be concave [1]

It follows by linearity that if each c_i has the above properties, so will C. We will thus concentrate on establishing the functional form of c_i. Our choice is the following

$$c_i(a,b) = 1 - e^{\frac{-x_i}{d_i}} \tag{5.2}$$

where x_i is that edge's value for i^{th} attribute and d_i is the average value of ith attribute across all edges in the considered graph. It fulfils all four requirements of a cost function and additionally bounds above the costs returned by c_i at 1.

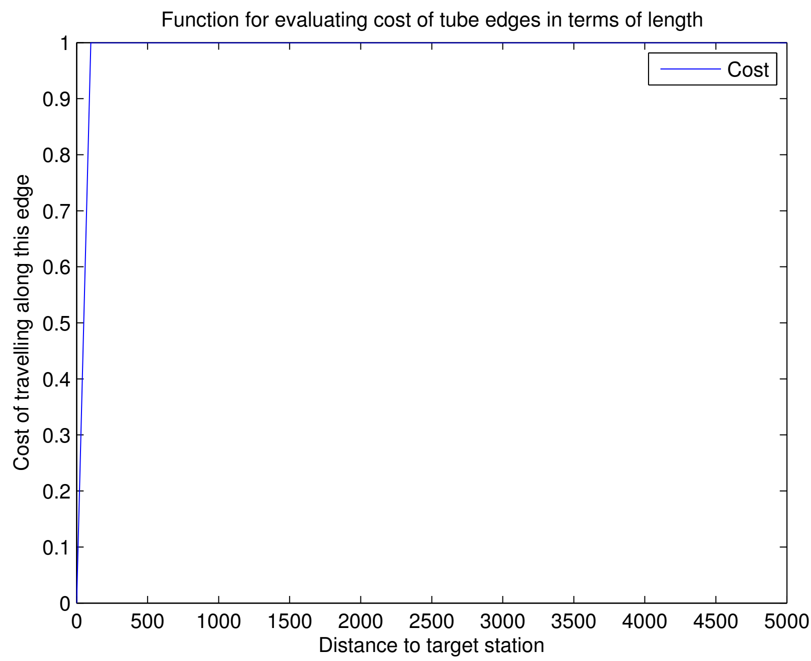

Figure 5.1: A cost function $1 - e^{-x}$ for the *tube_graph*, where x = edge length. 95% of edges in *tube_graph* are shorter than 5000m.

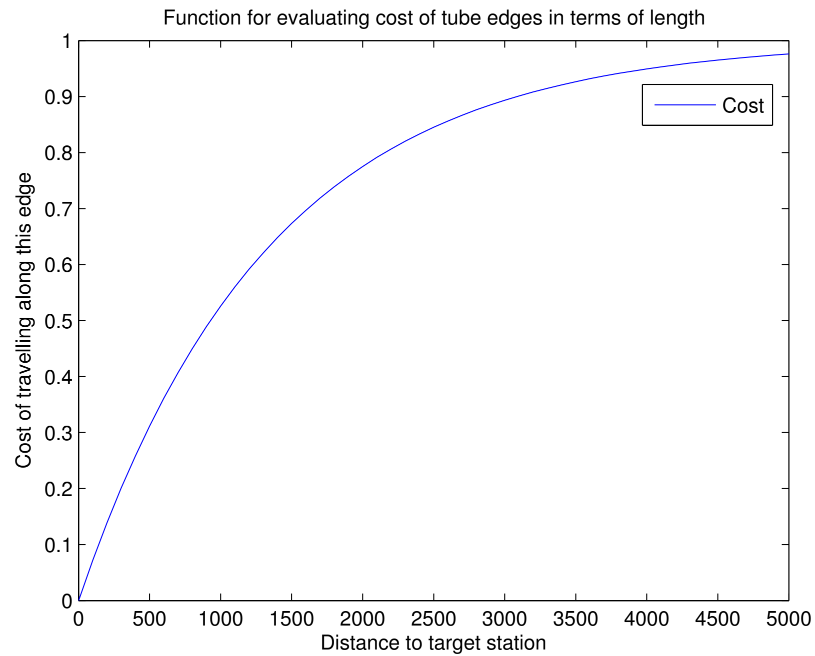

Figure 5.2: A cost function $1 - e^{\frac{-x}{d}}$ for the *tube_graph*, where $x =$ edge length and d is the average length across all edges, calculated to be 1340.2 meters. 95% of edges in *tube_graph* are shorter than 5000m and now they are discriminated by this cost function far more sensibly compared to the cost function shown in Figure 5.1.

The fact that the cost in each edge attribute is bounded above is important. A* express the overall desirability of travelling to any node k by evaluating its f cost, which we know from section 2.6.2 to be the sum of

- the cost of reaching the current node i
- the cost c of travelling along an edge to i's successor k
- the heuristic cost $h(k)$

If our edge cost function was, for example, a simple sum in edge attributes, weighted by user-specified w_i, the c in f could be overshadowed by a large $h(k)$ value, particularity if units of measure were not taken into account. $1 - e^{\frac{-x_i}{d_i}}$ gives the cost in each attribute and the heuristic cost an equal share in the overall f cost[2].

Lastly, we note that the reason (5.2) divides the edge's attribute value by the average value of that attribute across all edges is to stop the cost function from assigning high utility (i.e. low cost) to very short edges and uniformly penalising all other edges (strictly, all edges are assigned a different cost already since the cost function outlined is always increasing in its parameters - but only relatively short distances are discriminated meaningfully and for greater distances the differences in cost assignment are very small)[3]. As an example consider Figure 5.1, which shows the cost function that does not employ this trick. We can see that the function considers any edge longer than about 100m. as equally costly (in the non-strict sense we just described). This behaviour is not desirable, since the average value of the length attribute in *tube_graph* is 1340m, meaning the cost function will penalise most edges equally much - this is not the informative behaviour A* requires. Figure 5.2 shows the improved formula from (5.2), where the 95% of edges in *tube_graph* that are less than 5000 meters long are discriminated by our cost function far more sensibly.

As described in section 5.2, the cost function used for evaluating the attractiveness of every edge in a graph is not defined inside our A* algorithm. As with a heuristic function, the cost function is a parameter to our search algorithm. This gives us the ability to develop other, different cost functions in the future. As long as the new cost function will accept as input a list of edge attributes, will not try to evaluate non-existing attributes and, finally, return a number, it could alter the way user's preferences are considered any way it liked. This gives us the added flexibility of allowing the user to set more journey preferences in the future as we obtain more edge information from new data sources.

[2] So that the cost of travelling along the edge is not larger than the heuristic cost by the number of edge attributes, we first normalise c as defined in (5.2) before including it in f

[3] Our heuristic function, which will be calculating straight-line distance to target node, will take on this functional form also, with the exception that the calculated straight line distance x shall be divided by d where d = straight-line distance between that route's starting and finishing coordinates, i.e. the shortest possible route length.

5.4 Complete Journey Planning

In the previous section, we have described a heuristic-driven shortest path algorithm. It is based on A* and modified to search for paths in multigraphs. It enables us to find routes by successively selecting nodes the path to which meets a number of search-related criteria (for example if including it in the path will create a loop) and a number of other criteria, such as time taken to travel to that node or the busyness of said travel, whose relative impact on the decision to include the node in path is directly influenced by user-defined journey preferences. We are ready to tackle the second problem which we have mentioned at the beginning to this chapter - the problem of combining sub-routes of different transport modes to suggest single, overall journeys.

As mentioned in the introduction, this journey planner is to favour cycling among other modes of public transport. When receiving a journey-suggestion request, the routing engine should first consider if it is possible to find a journey that involves only cycling. Of course, walking sub-routes are added so that the user can reach the suggested starting and finishing docking stations from the journey starting position[4]. Whether searching for the main cycling route, or the walking sub-routes, we take into account user's preferences using methods described in section 5.3. The algorithm for finding the walking+cycling journey is straight-forward (see Algorithm 5). We explain two aspects of this algorithm:

- *bike_availability_model* is initialised with desired journey start time, hence we can find a docking station nearest to the starting coordinate position quite easily whilst taking into accounting the bicycle availability
- finding the finishing docking station is harder, because we don't know in advance the precise time for which we need to predict the availability of a docking space - this depends on the duration of the walking route to starting docking station and the duration of the cycling route. Our solution is to find a list of finishing stations, ordered by distance away from finishing coordinate position as selected by the user, and iterating through that list looking for the first station that would satisfy user's preference towards certainty of docking space availability.

If the resulting walking+cycling journey's duration does not exceed the user-specified desired trip duration, the journey planner achieved success and the found route is displayed on the journey planner's webpage interface as a color-encoded line on top of the map of local area. If, however, the returned walking+cycling journey is too long and the user specified earlier desired arrival time at the target, the journey planner attempts to find a shorter journey by including travel on the London Underground, assumed to provide a sort of 'short-cut'

[4]We make an implicit assumption that users want to begin their journeys in and around Central London, where the abundance of BCH docking stations means most can be reached on foot.

Algorithm 4 Handling journey plan requests

```
1: function CALCULATE_ROUTES(start_pos, finish_pos, journey_start_dt,
   user_preferences)
2:     london_graph = build_graph(get_london_data_loader())
3:     tube_graph = build_graph(get_london_data_loader())
4:     bike_availability_model = PoissonSamplingModel(request_dt=now,
5:                                    journey_start_dt)
6:     cost_model = DefaultCostModel
7:     cycling_route = get_cycling_route(start_pos, finish_pos,
8:                                    london_graph, bike_availability_model,
9:                                    user_preferences, cost_model)
10:    desired_journey_duration = user_preferences[trip_duration]
11:    if cycling_route.duration > desired_journey_duration then
12:        mixed_route = get_mixed_route(start_pos, finish_pos, london_graph,
13:                                    tube_graph, bike_availability_model,
14:                                    user_preferences, cost_model)
15:    end if
16:    return cycling_route, mixed_route
17: end function
```

Algorithm 5 Finding walking and cycling journeys

```
1: function GET_CYCLING_ROUTE(start_pos, finish_pos, london_graph,
   bike_availability_model, user_preferences, cost_model)
2:     start_dock = find_nearest_bch_doc(start_pos, bike_availability_model)
3:     start_walk,sw_duration = astar_path(london_graph, start_pos,
4:                                    start_dock, cost_model[cost_func])
5:     finish_docs = order_docs_by(finish_pos)
6:     for finish_dock ∈ finish_docs do
7:         route,c_duration = astar_path(london_graph, start_dock, finish_dock,
8:                                    cost_model[heuristic_func],cost_model[cost_func])
9:         get_dock_availability ∈ bike_availability_model
10:        if get_dock_availability(finish_dock, sw_duration + c_duration) ≥
   user_preferences[availability_certainty] then
11:            finish_walk,fw_duration = astar_path(london_graph, finish_dock,
12:                finish_pos, cost_model[heuristic_func],cost_model[cost_func])
13:            return start_walk+route+finish_walk
14:        end if
15:    end for
16: end function
```

whilst preserving at least parts of the cycling route.

The algorithm for finding this combined journey is harder to design, because it involves finding paths through *london_graph* and *tube_graph* and combining them into a chain trip. We solve this problem by first calculating a tube route as though the user wanted to travel on London Underground only. We then use this tube route as a 'guide line' along which we can investigate if adding a cycling sub-route will increase the overall journey duration beyond the user-specified limit. We begin the iteration at the end-points of the tube route. This is because, under assumption that London Underground is the fastest way to travel from among the modes of transport available to us, if a journey involving sole tube travel is already longer in duration than what the user requested than we should not attempt to find further combinations involving cycling as the resulting chain-journey's duration will only increase. Otherwise, we iterate through the stations of tube route, finding cycling sub-routes that would connect us to each of the stations if we decided to cycle to that station and not reach it by train. We do this until the duration of the next chain trip calculated this way would exceed the requested duration. The chain trip is returned as a 'mixed_route' since it's the journey suggestion that maximises the amount of cycling in a journey that nonetheless manages to get the user to desired target location on time.

The pseudo-code for the *route_request_handler* module that implements the above route chaining behaviour is outlined in Algorithm 4. It is called by our web server whenever the latter receives a POST request for a journey plan. Of interest are lines $2-4$ - as mentioned in section 3, *london_graph* and *tube_graph* are both cached at run_time, so that they can be accessed in $O(1)$ instead of having to be built from data held in database every time a new journey planning request is received.

5.5 Pathmax Optimisation

In this section we examine a small optimisation we hope will improve the performance of our routing algorithm further still. We mentioned in section 2.6.2 that for the A* algorithm to be optimal, the heuristic function $h(k)$ it uses to estimate the remaining cost of reaching target vertex v_n needs to be *admissible*. An admissible heuristic guarantees that our A* algorithm will find the shortest path if it exists.

Another property of a heuristic function is whether it is *consistent*. An A* algorithm that uses a consistent heuristic is known to be admissible, complete and optimally effective [13]. Formally, if k is a successor of some node i and

$$h(i) \leq c(i,k) + h(k) \tag{5.3}$$

$$h(v_n) = 0 \tag{5.4}$$

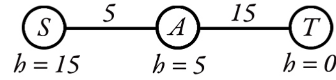

Figure 5.3: S is the source vertex and T is the target vertex. Edges are labelled with costs c. Nodes are labelled with h costs. f is not monotonically non-decreasing in depth. Since A* examines vertices in order of depth, in this example it fails to examine them in the f-order.

	Vertices		
Step	S	A	T
0	15		
1	15	10	
2	15	10	20

Table 5.2: f costs of nodes in Figure 5.3 as calculated at each step of A* without pathmax optimisation.

then the heuristic h is known to be consistent. Intuitively, as the search algorithm builds up its search tree by moving from some node i to its successor node k, the value returned by the heuristic function at vertex k cannot decrease by more than $c(i, k)$. This necessarily causes the f cost function to become monotonically non-decreasing in depth - as we examine the successors of node i in the 'direction' of the goal vertex, the f cost of these successor nodes is at least as large as that of i.

As an example, consider the network presented in Figure 5.3. Table 5.2 summarises the f costs A* assigns to each node as it searches for the shortest path to vertex $v_n = T$. We can see that the f costs of nodes on path SAT are not monotonically non-decreasing. Thus A* fails to visit the vertices in f order.

Examining nodes in f-order is important, because it means that once a vertex has been visited, the cost by which it was reached was the lowest possible (under assumption of no negative weights). An inconsistent heuristics may cause the A* algorithm to find shorter paths to nodes that were previously examined. If that case the re-visited node must be removed form the list of previously examined vertices, meaning it could be chosen for expansion again. This phenomenon is known as *node re-expansion*. This could be a problem when finding shortest paths through very large graphs since the A* requires memory linear in the number of visited vertices. The graph representing the area of Greater London contains a reasonable 221,233 nodes and 285,798 edges

Step	Vertices		
	S	A	T
0	15		
1	15	15	
2	15	15	20

Table 5.3: f costs of nodes in Figure 5.3 as calculated at each step of A* with pathmax optimisation.

but many cities worldwide that could use our journey planner are larger still - we are therefore interested in countering node re-expansions.

We described in previous section how our implementation of the A* algorithm allows for future development of other cost models. Whilst we expect the future contributors to recognise the importance of an admissible heuristic, we make no requirements for the consistency of the heuristic. To prevent node re-expansion, we adjust our A* algorithm by introducing the *pathmax* optimisation. Pathmax is a way of propagating inconsistent heuristic values in the search from a parent node to all of its successor vertices [28]. It causes the f-values of nodes to be monotonic non-decreasing along any path in the search tree by evaluating the heuristic cost at any node k which is a successor to some vertex i in the following manner:

$$\hat{h}(k) = max(h(k), h(i) - c(i,k)) \tag{5.5}$$

This alters the f costs seen in Table 5.2 to those shown in Table 5.3. We add it to the already-modified implementation of `astar_path` hoping it will decrease the search space of our algorithm (see lines $31, 50 - 56$ in Listing 5.2). In section 6.2 we will evaluate the effect this optimisation has on the performance of our A* algorithm.

Chapter 6

Results and Evaluation

6.1 Bicycle Availability Model Performance

The performance of our journey planner depends on the correctness of our bicycle availability prediction. As we have outlined in section 5.4, the choice of the starting docking station for a cycling route depends on matching the user's willingness to take risk of not being able to pick up a bicycle from nearest docking station at the benefit of not having to walk to a different station further away where bicycle presence is more likely. This risk preference is also taken into account when searching for a suitable finishing docking station. We therefore need our models to predict bicycle availability correctly, such that users of varying risk preferences are not guided to stations nearer of further away for the wrong reason. This section will present and evaluate the predictive capabilities of the two models, which we use to make the bicycle availability predictions as described in section 4.3.

6.1.1 Functional Performance

By default, our journey planner predicts the availability of a bicycle at a docking station at some point in the future using the sampling method described in section 4.3.2. There, we described how the method splits the difficult problem of estimating the true, unobservable density of the number of pickups and dropoffs that will occur between the time a request for the journey is received and the time a docking station is reached by splitting it into a number of smaller subproblems of estimating this very same density but for a number of short intervals that occur between the two times. We hope that in each of these intervals the number of pickups and dropoffs, as a discrete random variable, behave in a way that allows us to model their density more accurately.

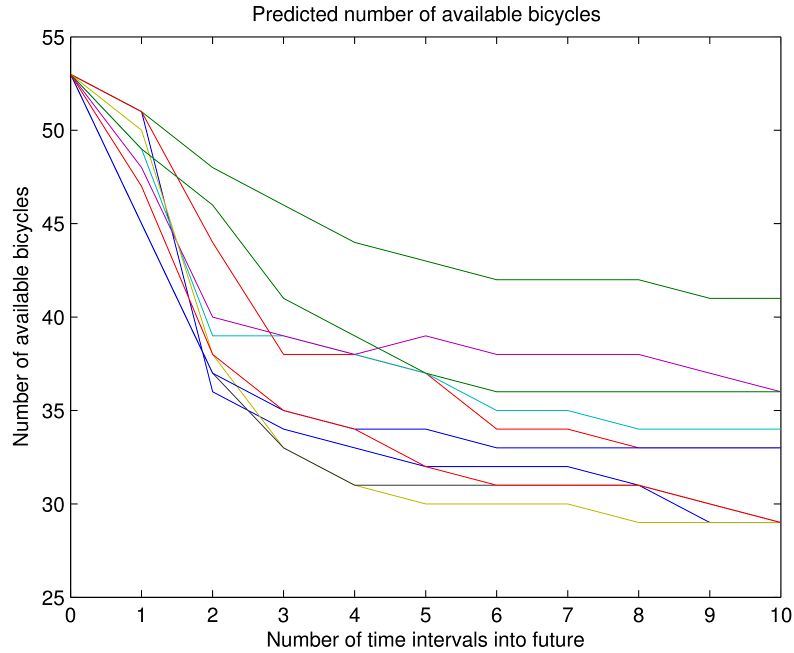

Figure 6.1: Showing 10 iterations of sampling method described in Algorithm 3 as run for Waterloo Station 2 docking station. The request time is 7:18am and the desired journey start time is 9:48am, ten 15-minute intervals later. At request time, the station was holding 53 bicycles in its 55 docking stations. At 9:48am, the station was estimated to hold between 29 and 41 bicycles. The algorithm returned $p(x_{9:48am} > 0) = 1$. This result is discussed further later in this section.

We will now examine three different scenarios to see if our model gives sensible predictions about bicycle availability. First, consider Figure 6.1. Plotted are 10 traces of the sampling method as it iterates through the time intervals that occur between the time a route request is received and the time the availability is to be checked for (see Algorithm 3 for details). In this case, we were looking to predict the number of bicycles available at Waterloo Station 2 (station_id=361). We made a route calculation request at 7:18am, when there were 53 bicycles at the station, and specified that we would like the journey to start at 9:48am, 2.5 hours later. The sampling method iterated through each of the ten 15-minute intervals that fit inside this timedelta, each time altering the number of available bicycles by the possible number of pickups and dropoffs in that interval (drawn from Poisson distribution of the expected number of pickups and dropoffs in that interval).

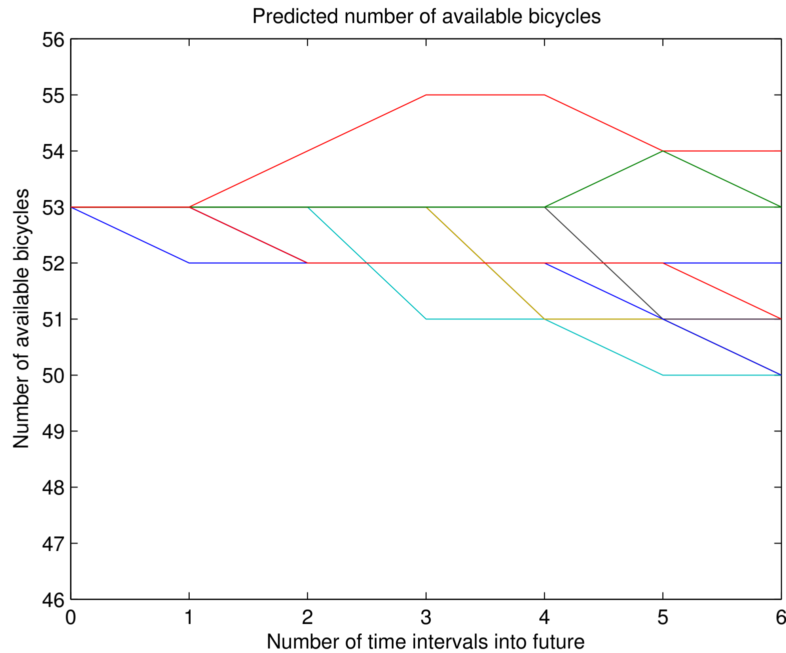

Figure 6.2: Showing 10 iterations of sampling method described in Algorithm 3 run for Waterloo Station 2 docking station. The request time is 11:18am and the desired journey start time is 12:48am, six 15-minute intervals later. At request time, the station was holding 53 bicycles in its 55 docking stations. At 12:48am, the station was estimated to hold between 50 and 54 bicycles. The algorithm returned $p(x_{12:48am} > 0) = 1$

Figure 6.5 shows that the expected number of pickups significantly exceeds the expected number of dropoffs at this station in the morning hours - possibly explained by the fact that Waterloo Station 2 is located near a major transportation hub that a lot of workers would have arrived at and who would be looking to cycle the final leg of their journey to the office. Thus the sampling method is correct in predicting a sharp decrease in the number of bicycles that will be present at this station by 9:48am. However, from the updates we receive from TfL we knew that at 7:18am there were 53 bicycles present at the station. Despite a number of bicycles being taken away by the morning commuters, we have never arrived at a conclusion that not a single bicycle will be available. As such, the sampling method predicts the user will indeed be able to find at least a single bicycle ready for pickup when they arrive at the station at 9:48am.

In Figure 6.1, to calculate the number of bicycles that will be available we had to consider intervals in which the expected number of pickups was significantly

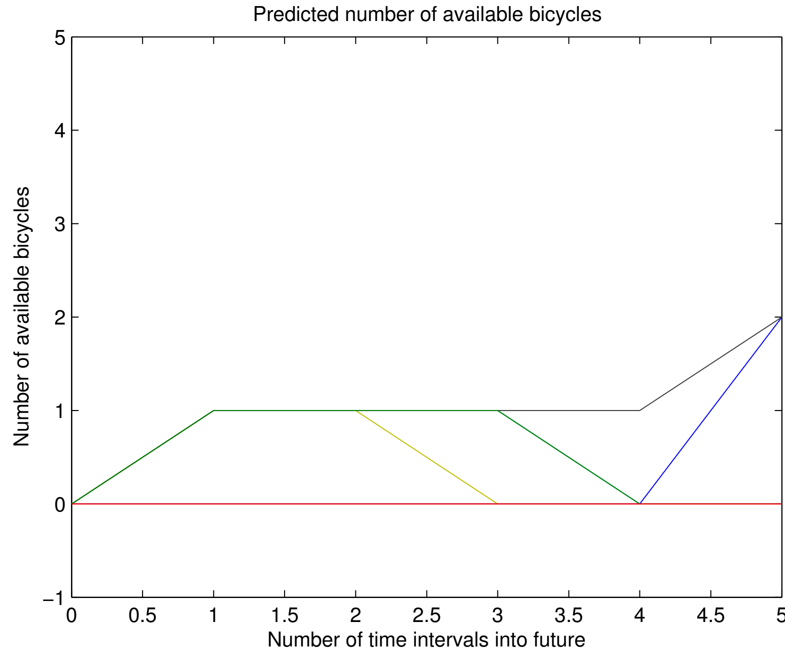

Figure 6.3: Showing 10 iterations of sampling method described in Algorithm 3 run for Waterloo Station 2 docking station. The request time is 10:45am and the desired journey start time is 11:48am, five 15-minute intervals later. At request time, the station was holding 0 bicycles in its 55 docking stations. At 11:48am, the station was estimated to hold between 0 and 2 bicycles, where the number of bicycles was estimated to be larger than 0 by only two of 10 iterations of the algorithm. The algorithm returned $p(x_{11:58am} > 0) = 0.2$

higher than the number of dropoffs. Figure 6.2 shows how the sampling method behaves when it iterates through intervals in which the expected number of pickups and dropoffs is similar. Again, we are trying to estimate the number of bicycles at Waterloo Station 2, but his time we made the request at 11:18am, when there were again 53 bicycles at the station, and specified that we would like the journey to start at 12:48am. From historical cycling journeys we know that the mean number of pickups and dropoffs at Waterloo Station 2 across the time intervals involved more or less match, and again we are pleased to see that our method has therefore estimated the number of bicycles at 12:48am to be roughly similar to the number of bicycles available at 11:18am.

So far we have examined how the sampling method behaves in regards to the estimated number of pickups and dropoffs across the time intervals of concern. Next, we would like to examine how our method will be influenced by a lack

```
rtk08=> select * from livestatuses where station_id=361;
 station_id | num_bikes_available | num_empty_docks | installed | locked |        update_dt        | status_age |         time_stamp
------------+---------------------+-----------------+-----------+--------+-------------------------+------------+----------------------------
        361 |                   0 |              54 | t         | f      | 2012-06-18 08:17:42.586 |         83 | 2012-06-18 10:45:40.508635
(1 row)
```

Figure 6.4: Database *bikestationrate* record for Waterloo Station 2 showing the station information following the latest update from TfL on 18/06/2012 at 10:45am. The update told us the number of bicycles available at this station is 0 and has been this way since 8:17am. This makes perfect sense when we consider that the station is located near a major hub of other modes of transport and passengers coming into work during the morning rush hour use the bicycles at this station to begin the final leg of their morning trips to work. Figure 6.5, which shows the historical mean number of pickups and dropoffs during morning hours shows that, indeed, the expected number of pickups prior to 10:45am is far greater than the expected number of dropoffs.

of available bicycles at route request time at the docking station of interest. Figure 6.3 shows the estimated number of bicycles at Waterloo Station 2 at 11:48am, five 15-minute time intervals after the route request was received at 10:45am. We specifically chose a similar time of the day to that of Figure 6.2 so that the expected number of pickups and dropoffs were similar. However, whilst the two previous tests where run on 08/06/2012, this test was instead run on 18/06/2012, when the number of bicycles present at 10:45am happened to be 0 (as seen in Figure 6.4). Figure 6.3 suggests that the distribution of the estimated number of available bicycles should have as its mean the number of bicycles available at route request time, i.e. 0. However, at every time interval we consider if the expected number of bicycles is less than 0 and adjust our estimate as shown in line 13 of Algorithm 3. As expected, a number of iterations were estimating the number of bicycles to be less than 0 at various time intervals of concern and have therefore ended up simply tracing the x-axis in Figure 6.3.

6.1.2 Non-Functional Performance

We have so far discussed the correctness of our sampling model for bicycle availability. We have seen that it is being correctly influenced by the expected numbers of pickups and dropoffs in intervals spanning the timedelta between route request time and journey start time, and also by the number of bicycles that are available at the station of interest at route request time. We would now like to examine its merits and limitations relative to the method outlined in Algorithm 2 as well as list of potential issues the choice of this model introduces.

rate_id	station_id	start_time	end_time	pickup_rate	dropoff_rate
50038	361	05:15:00	05:30:00	0.104651162790698	0.00581395348837209
50039	361	05:30:00	05:45:00	0.13953488372093	0.0232558139534884
50040	361	05:45:00	06:00:00	0.27906976744186	0.0174418604651163
50041	361	06:00:00	06:15:00	0.808139534883721	0.00581395348837209
50042	361	06:15:00	06:30:00	1.62209302325581	0.0232558139534884
50043	361	06:30:00	06:45:00	2.18604651162791	0.0465116279069767
50044	361	06:45:00	07:00:00	5.37209302325581	0.0523255813953488
50045	361	07:00:00	07:15:00	3.61627906976744	0.0697674418604651
50046	361	07:15:00	07:30:00	5.09302325581395	0.0581395348837209
50047	361	07:30:00	07:45:00	7.34883720930233	0.0523255813953488
50048	361	07:45:00	08:00:00	3.5	0.0697674418604651
50049	361	08:00:00	08:15:00	1.45348837209302	0.0755813953488372
50050	361	08:15:00	08:30:00	0.936046511627907	0.063953488372093
50051	361	08:30:00	08:45:00	0.959302325581395	0.0232558139534884
50052	361	08:45:00	09:00:00	0.616279069767442	0.0697674418604651
50053	361	09:00:00	09:15:00	0.534883720930233	0.063953488372093
50054	361	09:15:00	09:30:00	0.354651162790698	0.0581395348837209
50055	361	09:30:00	09:45:00	0.534883720930233	0.0523255813953488
50056	361	09:45:00	10:00:00	0.372093023255814	0.0523255813953488
50057	361	10:00:00	10:15:00	0.430232558139535	0.0988372093023256
50058	361	10:15:00	10:30:00	0.418604651162791	0.0988372093023256
50059	361	10:30:00	10:45:00	0.465116279069767	0.122093023255814

(22 rows)

Figure 6.5: The historical mean number of pickups and dropoffs at Waterloo Station 2, Waterloo, in the 15-minute intervals between 5:15am and 10:45am. As the expected number of pickups during morning rush hour intervals is far higher than the expected number of dropoffs, we are not surprised to find that by 10:45am, the number of available bicycles may be 0, as we did when the test underlying Figure 6.4 was run.

Sampling Model Merits

Estimating the probability of the number of events by drawing a large number of random variables from the corresponding density estimator introduces, by definition, uncertainty. In our case, the uncertainty in the number of pickups or dropoffs that occur at a station in a certain time interval is desired because it allows us to account for the different values these random variables may take. However, we want the uncertainty in the number of pickups or dropoffs in some time interval to apply only to that time interval.

The method shown in Algorithm 2 ignores this. In predicting bicycle availability at journey time, it models the density of the number of pickups and dropoffs that will occur across the entire timedelta using the expected number of pickups and dropoffs in just the first time interval that fits inside this timedelta multiplied by the number of time intervals that apply. This technique is motivated by (4.1), but it does not model well the uncertainty in the number of pickups and dropoffs across all time points between route request time and desired journey start time, since the uncertainty may be different at different time points. The sampling model was developed to counter this problem. If we estimate a different Poisson distribution for every time interval and only consider that interval's distribution when sampling for the possible number of pickups and dropoffs, then by combining the results from each interval we should obtain a more accurate estimate.

Sampling Model Limitations

However, the sampling method distorts our prediction about future bicycle availability in a different way. As mentioned in section 4.1, if we split the day into a number of shorter time intervals, the expected number of pickups and dropoffs in that interval is not very large. In particular, for the intervals of 15 minutes that we elected to use, the expected number of pickups and dropoffs in that period can sometimes be less than 0.5. This is particularly true for time intervals during the night and very early morning. Let us consider Waterloo Station 2 again, and in particular the expected number of pickups that occur at this station between 10:00am and 10:45am. As the sampling method iterates through these three intervals, it is most likely to estimate the number of bicycles available at 10:45am to be the same as the number of bicycles available at 10:00am. But looking at the expected number of pickups in those intervals, we would expect a single bicycle pickup by the time the second interval, from 10:15am to 10:30am, is over.

Consider now our other model - it will assume that the expected number of pickups that will occur in the interval 10:00am-10:45am will be 3×0.37209, the latter being the expected number of pickups in the interval 10:00am-10:15am. It will use the result as the parameter for the Poisson distribution estimating the number of pickups between 10:00am-10:45am. Thus, in a sense, the model

we have chosen not to use in our bicycle predictions works better when the expected number of arrival events (pickups or dropoffs) is very small. To avoid this problem as much as possible, we have made a heuristic decision to let our time intervals be of a 15 minute duration. This duration seems large enough to guarantee that we will have some samples of the number of pickups and drop-offs for that interval while at the same time it is short enough to allow us to consider the uncertainty in the number of pickups and dropoffs that occur during busy, 'rush-hour' times of the day more accurately.

Another surprising property of our bicycle availability model based on the sampling method is that the method, as shown in Figures 6.1 and 6.2, can return a result of 1 for the probability of there being a bicycle as some point in the future. That is to say we are absolutely certain that a bicycle will be present. This occurs when, throughout every iteration of the sampling algorithm, we have never ended up estimating the number of bicycles at the journey start time to be less than one. However, we agree that future is uncertain and it is incorrect to be guaranteeing bicycle (and, similarily, free docking space) presence. This is particularity true because of the random events the BCH network is subjected to which we are unable to predict. One such event is the relocation of bicycles - done ad-hoc by TfL to maintain the network load. As this does not constitute a cycle journey, we have no history of these events, using which we could try to account for them. This is most probably the reason behind the number of bicycles available at Waterloo Station 2 on 08/06/2012 being near that station's capacity, as seen in the results of an experiment we made that day shown in Figures 6.1 and 6.2, and zero when another experiment involving the same station and time of day (including the type of day, i.e. workday) was run 10 days later.

Density Model Selection Evaluation

In section 4.1 we noted that Poisson is a particularly useful distribution to us because, being parameterised only by λ, we should require little historical data to *train* our model. The parametric density models do not suffer from *course of dimensionality* in the same way that non-parametric methods do - the latter need exponential amount of data as the number of parameters that describe them grows.

However, this also means that our model adjusts to this data very well. If the calculated average numbers of arrival events are not truly representative of the true, unobservable probability mass function of the number of pickups and dropoffs that occur, then our model would be known to be *over-fitting* the sample (in this context also known as *training* data). Consider Figure 6.6 which shows the Poisson distribution fitted to the 7:30am-7:45am interval at Waterloo Station 2. The mean of this distribution is 7.35 - from our historical cycle journeys data set we know that to be the average number of pickups that occur at this station in this time interval. Now consider Figure 6.7 which shows the

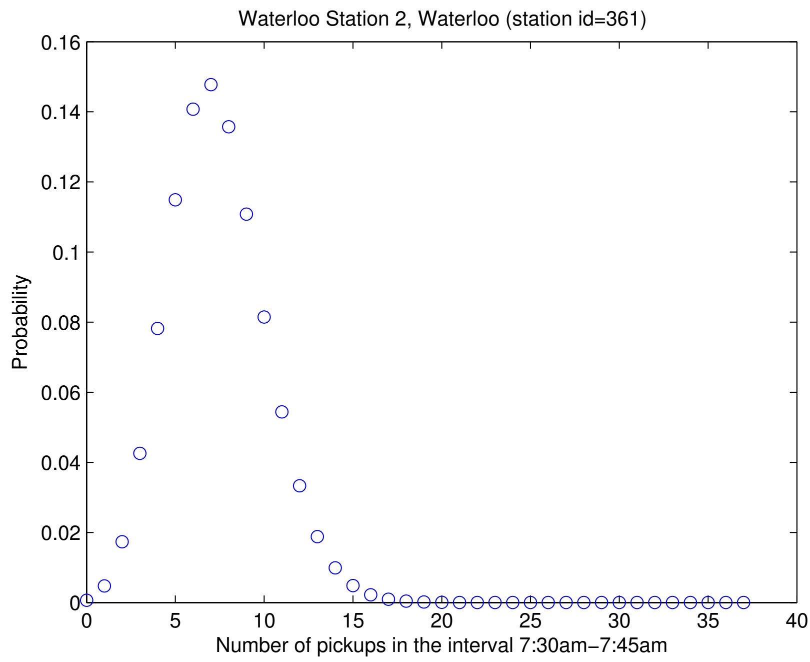

Figure 6.6: Showing the frequency density of the different number of pick-ups that occur at Waterloo Station 2 in the interval 7:30am-7:45am. Mean is 7.3488373 pickups.

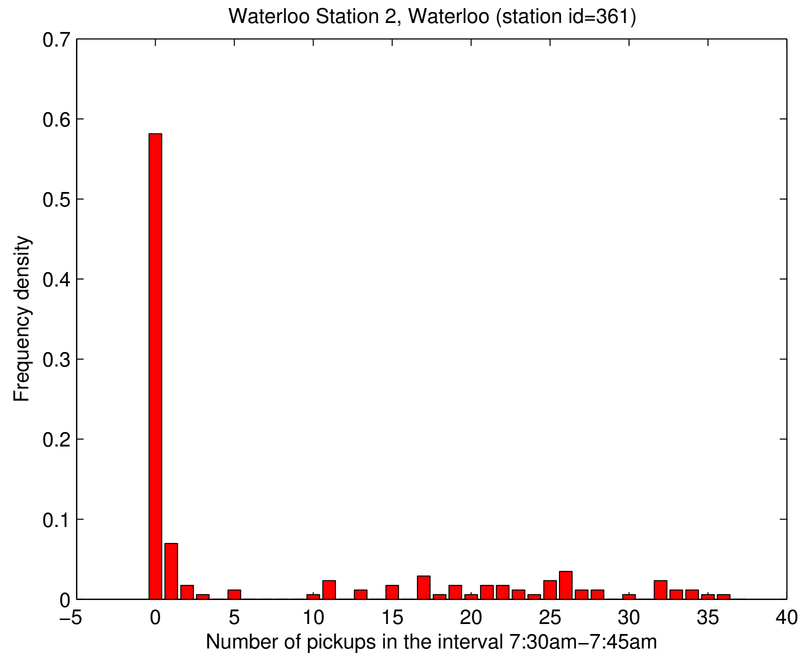

Figure 6.7: Showing the frequency density of different number of pickups that occur at Waterloo Station 2 in the interval 7:30am-7:45am.

frequency density of the different number of pickups that occur at Waterloo Station 2 in the same interval. This is the frequency of the occurrence of a particular number of pickups, divided by the total number of observations. We can view this as the empirical probability. Clearly, whilst the average number of pickups is indeed 7.35, this is because relatively rarely a really high number of pickups occurs. In contrast to what our Poisson model estimates, most frequently 0 pickups occurred.

If our historical cycle journeys data was not representative of the true distribution of cycle journeys that take place every day, the density models we build using this data would not allow us to predict future bicycle availability accurately. We suspect this is true because:

1. the cycle journeys data was collected only in the first few months of BCH's existence and the network has since increased in size and popularity.
2. it seems strange that on a random pattern of non-consecutive days an average of 22 pickups occur, followed by four weeks of no pickups.

We attempted to deal with the first problem by scaling the expected numbers

of pickups and dropoffs at docking stations (see section 4.3.2). To make this scaling reflect our latest view of the BCH network, we have employed a one-pass algorithm for calculating the average change in the number of bicycles and free docking stations present at docking stations in each of the intervals in a day. Recalculating the average change in bicycle availability in this manner means our prediction uses the *latest* data available to us - our bicycle prediction model self-improves as the time goes on.

Examining the second problem, we would suggest that the station might have been closed in that time. As with the network-load-related relocation of bicycles, we do not take these events into account in any way other than by decreasing our expected number of pickups for the time interval concerned. We note that a 'fresher' data set listing all cycling journeys from May 2011-February 2012, which TfL have recently made available, could help clear up the confusing results we see in Figures 6.6 and 6.7.

For now, we conclude that the best method of assessing the suitability of a density estimator is to test its predictive capabilities. A method called *n-cross-validation* can do this well - it involves splitting the historical observations about the number of pickups and dropoffs into n sets. Our density estimator is then *trained* on the $n-1$ sets and validated against the single set left out. This is done n times and every time the sets are shuffled, so that we don't cross-validate on the same data n times. At each *fold* the root mean squared error in prediction is calculated. The accuracy of the model can thereafter be expressed as the mean of these individual RMSEs, and in other forms like confusion matrices, from which useful statistics like recall and precision rates can be obtained. The author will evaluate our model using these methods in the coming days.

6.2 Routing Algorithm Performance

6.2.1 Functional Performance

As mentioned in the opening paragraphs of Chapter 5, the problem of finding the most desirable journeys that combine walking, cycling and travel on the London Underground is a two-fold problem. Before we investigate the performance of the algorithms underlying our journey planner, we would first like to see if the intended behaviour, as specified in Chapter 1, has been obtained. Appendix A contains screen shots of our journey planner in action. It shows the journey planner behaves as intended when the users alters their preferences, redefining the requirements of the 'most desirable journey' each time.

It is difficult to evaluate the correctness of the calculated sub-routes. Apart from user-defined preferences, the path of a route through *london_graph* or *tube_graph* is influenced by other factors such as accessibility. We have therefore tested the correctness of routes being found by asking a number of users to request routes

they know well - those users have the *expert knowledge* over these routes and can assess the quality of our suggestions. From this point of view, the quality of our journey planner will also depend on the accuracy of data we base our routing on. We mentioned in section 2.4.2 that the data describing Greater London as a network was obtained from OpenStreetMap community. It is generally very accurate for densely populated areas such as London, but we have found instances where our planner was suggesting to cycle along a currently closed bridge, for example. Additionally, whatever improvement the community will make to the map will not be incorporated into our journey planner as we are working off a local copy of the dataset representing Greater London. A similar case can be made for our London Underground data - we have 267 stations on record but this is expired data since the current number of stations is higher. Nonetheless, these problems can be easily fixed by obtaining more accurate datasets and are therefore not considered to be an issue to do with our routing functionality.

6.2.2 Non-Functional Performance

Path Finding Performance

As mentioned in section 5.2, we expected out modified implementation of `astar_path` to perform well in finding shortest paths, both through simple graphs like *tube_graph*, and directed multigraphs like *london_graph*. An easy way to test the performance of a search algorithm is to count the number of nodes it examines as part of finding the shortest path to target vertex. When the heuristic function is ill-defined, A* algorithm can behave like a breadth-first-search algorithm and then the number of vertices it needs to examine before finding the shortest path could be exponential in the number of nodes in said path. However, as argued in section 5.2 we have access to an admissible heuristic (that can be additionally made into a consistent one using pathmax optimisation, introduced in section 5.5) that we believed would result in a very reasonable performance of our algorithm. We were surprised to find that our modified implementation of `astar_path`, as shown in Listing 5.2, was examining a larger number of nodes with the pathmax optimisation than without it. After some analysis we concluded that, in contrast to author's original belief, the pathmax optimisation is not guaranteed to introduce monotonicity to A* algorithm's f cost function.

To explain why, let us consider the simple graph in Figure 6.8. Table 6.1 shows f costs of different vertices A* knows about as it searches its way towards target vertex T. We can see that the pathmax optimisation from (5.5) makes the f costs along the thus-far-explored paths towards T non-decreasing (consider how the f cost of C in step 2 is set to 9 instead of 3). However, what we failed to account for is that this monotonicity of f cost function holds only along the paths traversed thus far [29] - the f costs of nodes on paths that A* has not yet

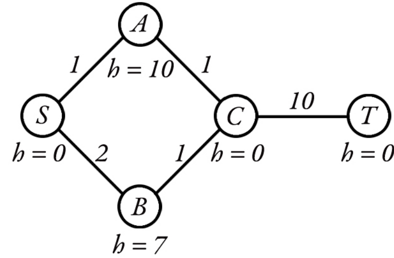

Figure 6.8: S is the source vertex and T is the target vertex. Edges are labelled with costs c. Nodes are labelled with h costs. f is not monotonically non-decreasing in depth despite pathmax optimisation.

	Vertices				
Current node	S	A	B	C	T
S	0	11	9		
B	0	11	9	9	
C	0	11	9	9	13
A	0	11	9	**11**	13
C	0	11	9	11	12
T	0	11	9	11	12

Table 6.1: f costs of nodes in Figure 6.8 as calculated at each step of A* with pathmax optimisation.

explored may be decreasing in the direction of the target vertex (consider the f costs of C and A after B is examined). As discussed in section 5.5, lack of f cost function monotonicity can lead to node re-examination, which we wanted to avoid by incorporatin pathmax. However, here we see that node C needs to be re-examined after, two iterations earlier, A* found the f cost of T to be higher than A, leading to examination of A and the re-adjustment of C's f cost that followed (shown in bold in Table 6.1).

The real problem here is as follows - suppose we could reach a number of other nodes from C whose f costs were less than that of A. The implementation shown in Listing 5.2 would examine each of those nodes before returning to A. This would explain the reason we were seeing a higher number of nodes being explored with the pathmax optimisation than without it. As a result, we have removed the pathmax optimisation from our version of the A* algorithm as presented in Listing 5.2. The resulting algorithm, which now powers our journey planner, is shown in Listing 6.1.

Listing 6.1: Version of our A* algorithm for finding shortest paths in NetworkX graphs that excludes the the malfunctioning pathmax optimisation

```
def astar_path(G, source, target, heuristic_func=None, cost_func=None
    ):

    if heuristic_func is None:
        def heuristic_func(s_node, t_node):
            return 0

    if cost_func is None:
        def cost_func(edge_attributes):
            return 0.5

    queue = [(0, hash(source), source, 0, None)]
    enqueued = {}
    explored = {}

    while queue:
        _, __, curnode, curr_cost, parent = heappop(queue)

        if curnode == target:
            path = [curnode]
            node = parent
            while node is not None:
                path.append(node)
                node = explored[node]
            path.reverse()
            return path

        if curnode in explored:
            continue

        explored[curnode] = parent

        for neighbor, edge_attributes in G[curnode].items():
            if neighbor in explored:
                continue

            if G.is_multigraph():
                cost_to_reach_neighbour = min(map(lambda edge_key:
                    cost_func(edge_attributes[edge_key]),
                    edge_attributes.keys()))
            else:
                cost_to_reach_neighbour = cost_func(edge_attributes)

            ncost = curr_cost + cost_to_reach_neighbour
            if neighbor in enqueued:
                qcost, h = enqueued[neighbor]
                if qcost <= ncost:
                    continue
            else:
                h = heuristic_func(G.node[neighbor], G.node[target])

            enqueued[neighbor] = ncost, h
            heappush(queue, (ncost + h,
                             hash(neighbor),
                             neighbor,
                             ncost,
                             curnode))

    raise nx.NetworkXNoPath("Node %s not reachable from %s" % (source
        , target))
```

Trip-Chaining Performance

Having introduced the fix described in previous section, our algorithm for finding shortest paths in simple graphs or multigraphs is of the expected complexity. Finding a cycling path from author's house to Imperial College London, a distance of around 8 miles across London's city centre, in in the order of hundreds of miliseconds. However, when more complicated trip chaining occurs and we need to find an optimal route that chains together walking, cycling and tube sub-routes, the performance of our journey planner slows down significantly, into the order of couple of seconds or even tens of seconds, depending on the length of request trip.

The bottleneck is in our trip-chaining method is the fact that we are trying to calculate the single, overall route by finding sub-routes in separate graphs representing different transport networks. To combine these routes, we need to know how the nodes in graphs are geographically related to each other. Since the nodes are in different graphs and as such no neighbouring relationships exist between them, whenever we try to switch from a sub-route of one mode of transport (e.g. cycling) to the other (e.g. London Underground), we must search the latter graph (*tube_graph*) for a node geographically closest to our current position. Unless we are able to intelligently partition our graphs according to their coordinate position, this search is of time complexity $O(n)$, where n is the number of nodes in the graph being searched.

As an example, in our partition-less approach, we need to consider all the nodes in *london_graph* every time we try to:

- find a node in *london_graph* nearest to the coordinate position specified by the user on map as the desired starting point for the overall journey - this node becomes the starting point of a walking sub-route that will take us to the nearest BCH docking station or the nearest London Underground station
- find a node in *london_graph* nearest to the coordinate position specified by the user on map as the desired finishing point for the overall journey - this node becomes the finishing point of a walking sub-route that allows us to arrive at the desired target location from the end-point of the cycling or tube route that brought us this far
- find a BCH docking station or a London Underground docking station nearest to the coordinate positions specified by the user on map as the desired starting and finishing points of the overall journey - these stations become the starting and finishing points for the cycling and London Underground sub-routes
- find a node in *london_graph* nearest to the coordinate position of a BCH docking station or a London Underground station - this node becomes the starting or finishing points of a walking sub-route that connects a cycling

Listing 6.2: Function for sorting nodes in a graph according to their distance from some coordinate position pos

```
1  def sort_nodes_by_loc(graph, pos):
2
3      node_iter = graph.nodes_iter(data=True)
4
5      def comparison_function(node):
6          nodeLatLon = (node[1]['latitude'],
7                        node[1]['longitude'])
8          return get_distance_in_m(pos, nodeLatLon)
9
10     return sorted(node_iter,key=comparison_function)
```

or a tube route to another sub-route in the trip chain.

Speedy route calculation was outside the aims of this project. This journey planner is a proof-of-concept, rather than an enterprise-quality solution. However, we attempted to mitigate some of the problem in two ways:

1. we improved the implementation of our sorting function. We use an iterator to traverse the list of nodes in a graph, which in Python is known to save both time and space for larger dataset compared to list-based approaches. The resulting function is shown in Listing 6.2

2. we precompute and cache a dictionary that maps every BCH docking station and London Underground station to its nearest node in *london_graph*. This is particularly useful inside the function for calculating routes combining all three modes of transport of interest to us which, as described in section 5.4, performs a number of sub-route-chaining iterations

We have briefly studied two approaches which could be used to further improve the complexity of our trip chaining functionality. The first of these approaches performs pre-computation to partition the nodes in a graph according to their geographic location. The second approach removes the need for linking sub-routes in different graphs all together. Both approaches are briefly discussed in section 7.3.

Chapter 7

Conclusions and Future Work

7.1 Conclusion

Through this project we wanted to develop a journey planner for the area of Greater London that would combine cycling, walking and London Underground journeys. We wanted it to promote cycling but combine it with the other two modes of transport if the trip duration as desired by the user was less than that of our cycling journey suggestion. We wanted the route calculation to take into account user-defined preferences such as its busyness and how important it was to arrive at the destination on time. However, the journey planner was to stand out because of its unique ability to predict future bicycle availability across the docking stations of a bicycle sharing scheme. We have so far shown that these aims were achieved. The trips suggestions are displayed across an attractive-looking web page and the user has a choice of displaying any of them over the map of Greater London.

As part of building the model of bicycle availabilities, we have developed two methods based on estimating the true, unobservable density of the number of pickups and arrivals, which we can then use to calculate the probability of there being a bicycle available at some station at some time point in the future. We needed to know this so that we could start the cycling sub-routes of our journey suggestions at BCH docking stations that suited the user's bicycle availability risk preferences. Similarly, we developed an equivalent method for predicting future free docking space availability. We found that these models train well on even small sample sets and have shown how the different expected numbers of pickups and dropoffs as well as the live bicycle availability affect our predictions.

We also noted that the data we use to build these models from is now fairly old and may not represent the true density of cycle arrival events well. We developed a method that accounts for the increased size and popularity of the BCH system by scaling the expected numbers of dropoffs and pickups by the average changes in bicycle and free docking space availabilities across some time intervals of a day. We have used a one-pass algorithm to keep the latest sample mean of these changes so that we can trace the latest state of the network. However, we have also noted that there are cases when our model fails to predict the number of pickups and dropoffs as evident from the frequency density of the different numbers of pickups and dropoffs that actually occurred and note that the proper investigation into predictive performance of our models is a subject of further work.

We have also developed a routing engine that uses the A* shortest path algorithm to find cycling, walking and tube sub-routes which are then combined into single, overall routes we can suggest to the user. We have modified and improved an A* shortest path algorithm of a popular network management library NetworkX and are currently in the process of contributing our source code to this library. Specifically, we have introduced a capability of finding shortest paths through multigraphs, and allowed the evaluation of the cost of edges to neighbouring nodes to be more sophisticated by considering multiple edge attributes. We made an attempt at applying the pathmax optimisation to our algorithm but have arrived at the conclusion that it will not introduce monotonicity to the f cost function in the expected manner.

Overall, we have arrived at an encouraging observation that the duration of cycling routes involving BCH tends to be less than the combinations of other modes of public transport.

These types of journey planners, we believe, will improve the quality and attractiveness of public transport that combines cycle schemes and the techniques that give us increasingly accurate predictions of bicycle availability should be investigated in future research.

This project involved many fields of computer science and there are many things we would like to improve in our journey planner. Listed below are topics which would be interesting to explore.

7.2 Improving Bicycle Availability Predictions

Our approach to predicting bicycle availability is known as the *frequentist approach* [5]. In this approach, we viewed the probabilities of events in terms of their relative frequency in a large number of repeatable events, which is to say that the maximum likelihood method allowed us to obtain point estimates for the adjustable parameter λ of Poisson distributions - our assumed density model - by calculating the sample mean of the number of pickups and dropoffs

that occurred within the corresponding time interval at the station of interest. However, a numerical estimate of λ does not indicate how good an estimate it is. It would be very interesting to compare our model to one that would employ the Bayesian approach to estimating

Instead of calculating a point estimate for λ, the Bayesian approach estimates the true value of λ using a probability distribution. It is generally viewed that such *generative* models will perform better. Whenever a new observation about the true number of pickups and dropoffs is made, we could apply the Bayes' Theorem to compute corresponding posterior distribution of λ. This would enable sequential learning of the parameters of Poisson distributions describing the expected number of pickups and dropoffs in a similar manner to our one-pass algorithm for calculating an up-to-date expected change in number of bicycles/free docks between time intervals, as opposed to the current solution where the estimates of λ parameters of all distributions are static and calculated using old and possibly unreliable data.

TfL has recently released another dataset of cycle journeys for a number of months covering late 2011 - early 2012. We would expect the incorporation of this data into our prediction models to improve its predictive performance, as the more recent data would account more truthfully for the increased size and popularity of BCH then we can with our method for scaling the sample means of number of pickups and dropoffs, as outlined in section 4.3.2.

In estimating the true density of the number of pickups and dropoffs that occur at a station throughout the day, it could be worth considering days of the week, since we suspect the true, unobservable density of the number of pickups and dropoffs is different throughout the weekend intervals compared to their weekday counterparts.

7.3 Improving Router

In evaluating the complexity of our trip-chaining function, we noted that the majority of the computation time is spent not calculating the routes themselves (our A* algorithm does this satisfactorily well) but in chaining the found subroutes together. Though we have been able to mitigate an issue to a certain extend, we are still forced to search the large *london_graph* a number of times. We have investigated two possible solutions to this problem:

- we could pre-compute a spatial decomposition of *london_graph*'s nodes (based on their coordinate position) using some tree-based structure. Of interest could be quadtrees [15], which are used to partition a two-dimensional space into four quadrants (so-called *buckets*). Each bucket has a maximum capacity and when this is reached it splits into four smaller buckets. When looking for a node nearest to some coordinate position, we would traverse

the tree, decreasing the area containing the nearest *london_graph* node by a factor of four for each level traversed.

- We could forgo trip-chaining by calculating sub-routes on separate graphs altogether and instead merge the graphs of all available modes of transport into one, multimodal graph [4]. This would rid us of the problem of chaining separately found sub-routes. By developing a smart multimodal model of the resulting network, the task of mode-chaining would be incorporated into path finding itself. A multimodal shortest path algorithm would do all the hard work for us.

We could also improve the sophistication of our implementation. We would look to introduce concurrency to our journey planner, so that a calculation of a trip suggestion for one user does not hold up a route request coming from another user. An enterprise-level caching system such as *memcache* should be employed instead of our current, primitive caching solution.

We can think of a number of ways in which the desirability of trips our journey planner suggest to users could be increased beyond the consideration of user-specified route preferences. If we allows users to rank the journeys we suggest to them, we could use this grading as feedback in later journey planning, effectively customising our journey planner to each user. We could also give the users an opportunity to provide feedback on journeys completed as per our suggestions, so that the information received could be used to reinforce our understanding 'of the world'. For example, a user could provide feedback on the busyness of various sections of the suggested cycling journey and we could combine this feedback with our current knowledge to obtain more accurate busyness information for each section of the suggested route.

Appendix A

Journey Planner - In Action

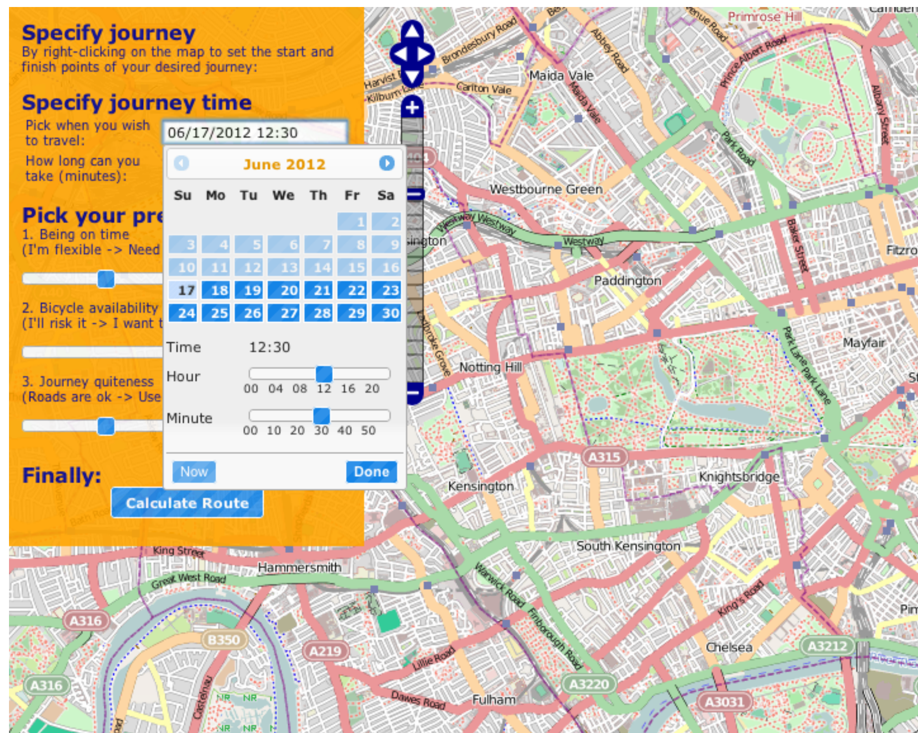

Figure A.1: The users can specify the start time of their journey, its desired duration as well as their preferences towards being able to arrive at target on time, being certain about bicycles and free parking space availabilities at starting and finishing stations as well as preferred route busyness. Journey start and finish points are set by right-clicking on the map at the desired location and choosing the point to set from a drop-down menu.

Figure A.2: The user elected to travel from outside Imperial College London towards Edgware Road. As the route involving just cycling (blue) and walking (green) is of a duration less than the desired trip duration, no alternative route combining other modes of transport was looked for.

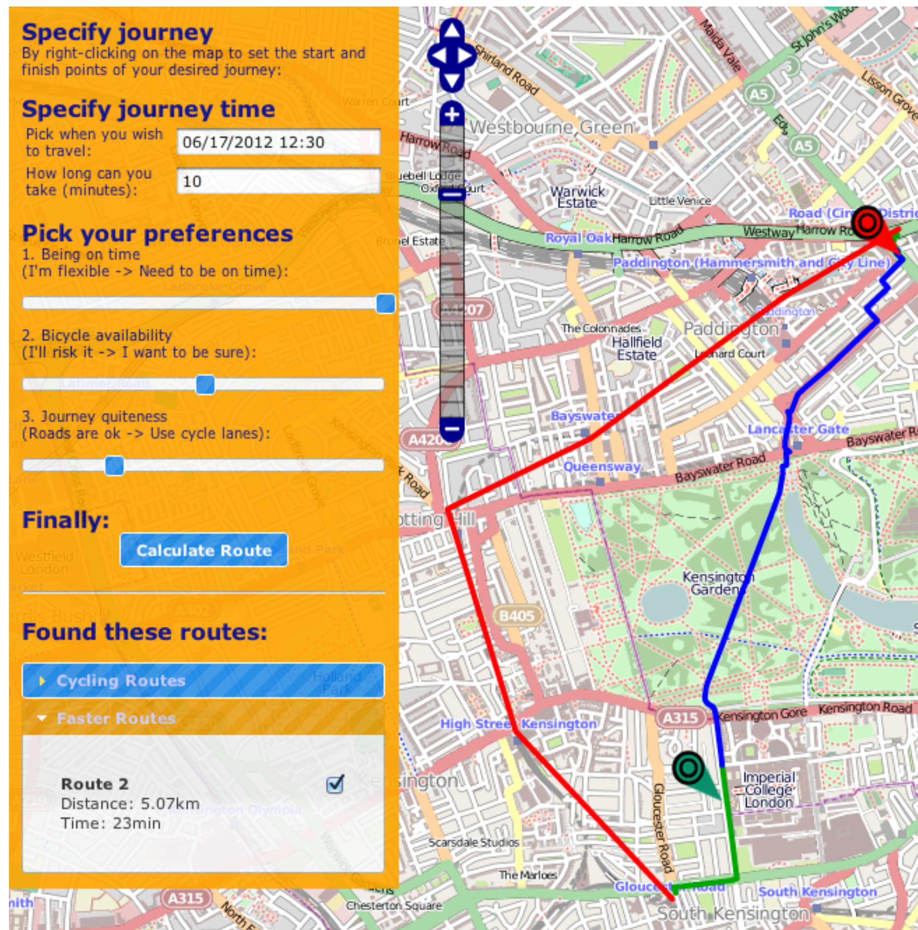

Figure A.3: The user elected to travel from outside Imperial College London towards Edgware Road. The user specified in preferences that they will not like to be late. As the route involving just cycling (blue) and walking (green) is of a longer duration (23 minutes) than the desired trip duration (10 minute), an alternative route combining other modes of transport was looked for. Both the cycling+walking route, and the alternative tube+walking routes are displayed. Interestingly, the faster route suggestion fails to beat the duration of the cycling+walking route. This is probably because the user has a longer walk to the starting London Underground station than to the nearest BCH docking station, and the faster tube travel is not enough to make up for the lost time.

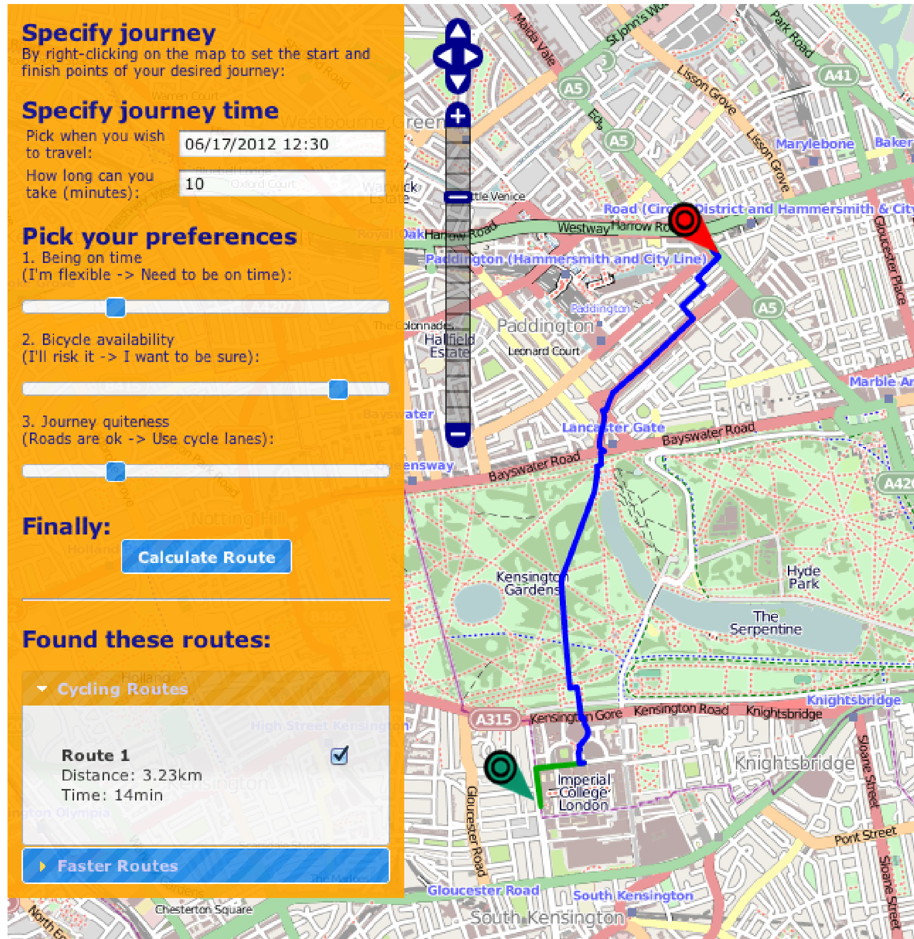

Figure A.4: The user elected to travel from outside Imperial College London towards Edgware Road. The user specified in preferences that they will not like to face uncertainty about the availability of a bicycle at the starting BCH docking station. The probability of there being a bicycle available at the station suggested as the starting point of the journeys seen earlier in this Appendix must have been less than required, hence a different starting BCH docking station was used.

Bibliography

[1] S. Anily and A. Federgruen. A class of euclidean routing problems with general route cost functions. 15(2):268–285, May 1990.

[2] BBC. London cycle hire scheme expands eastwards, 8 March 2012 and Accessed 10 March 2012. http://www.bbc.co.uk/news/uk-england-london-17296565.

[3] BBC. The Passport blog, 9 September 2011 and Accessed 17 September 2011. http://www.bbc.com/travel/blog/20110909-travelwise-bike-sharing-around-the-world.

[4] Maurizio Bielli, Azedine Boulmakoul, and Hicham Mouncif. Object modeling and path computation for multimodal travel systems. 2005.

[5] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 6th edition, 2007.

[6] Bixi. Bicycle sharing system, Accessed 11 June 2012. https://montreal.bixi.com.

[7] Nick Carey. Establishing Pedestrian Walking Speeds. 2005.

[8] Gand Cheng and Nirwan Ansari. A Theoretical Framework for Selecting the Cost Function for Source Routing. 2003.

[9] Seungjin Choi. EECE515 Machine Learning: Density Estimation. Pohang University of Science and Technology, Korea.

[10] OpenStreetMap Community. OpenStreetMap Extract covering Greater London Area, Accessed 18 March 2012. http://download.geofabrik.de/osm/europe/great_britain/england/greater_london.osm.bz2.

[11] OpenStreetMap Community. List of London Underground stations, Accessed March 2012. http://wiki.openstreetmap.org/wiki/London_Tube_Stations.

[12] cyclestreets. Journey planner, Accessed 17 December 2011. http://www.cyclestreets.net/.

[13] Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A*. 1985.

[14] E. W. Dijkstra. *A note on two problems in connexion with graphs*, volume 1. Numerische Mathematik (Historical Archive), Dec 1959.

[15] Raphael A. Finkel and Jon Louis Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Inf.*, 4:1–9, 1974.

[16] Transport for London. Developer' Area: Get Data, Accessed 11 June 2012. `http://www.tfl.gov.uk/businessandpartners/syndication/16492.aspx`.

[17] Transport for London. Cycle Journey Planner, Accessed 16 October 2011. `http://www.cyclejourneyplanner.tfl.gov.uk`.

[18] Transport for London. Barclays Cycle Hire Statistics, Accessed November 2011. `http://www.tfl.gov.uk/businessandpartners/syndication/16493.aspx`.

[19] Transport for London. Barclays Cycle Hire Availability Datafeed, Accessed since November 2011. `http://www.tfl.gov.uk/businessandpartners/syndication/16493.aspx`.

[20] Dr. N A Heard. Discrete Random Variables, Accessed 10 March 2012. `http://www2.imperial.ac.uk/~naheard/C245/discrete_random_variables_article.pdf`.

[21] J.K. Jolliffe and T.P. Hutchinson. A behavioural Explanation of the Association Between Bus and Passanger Arrivals at a Bus Stop. In *Transportation Science*, volume 9, pages 248–282, January 1970.

[22] Ryszard T. Kaleta. Adding multigraph handling to astar_path function, Accessed 14 June 2012. `https://networkx.lanl.gov/trac/ticket/731`.

[23] Richard L. Knoblauch, Martin T Pietrucha, and Marsha Nitzburg. Field Studies of Pedestrian Walking Speed and Start-Up Time. 1996.

[24] Tim Lewis. Has London's cycle hire scheme been a capital idea?, 10 July 2011 and Accessed 20 December 2011. `http://www.guardian.co.uk/uk/bike-blog/2011/jul/10/boris-bikes-hire-scheme-london`.

[25] Marco Luethi, Ulrich Weidmann, and Andrew Nash. Passenger Arrival Rates at Public Transportation Stations. 2006.

[26] Geoff Marshall. Distance Between Stations, Accessed March 2012. `http://ni.chol.as/media/geoff-files/sillymaps/milesdistances.gif`.

[27] Geoff Marshall. Travel Times Between Stations, Accessed March 2012. `http://ni.chol.as/media/geoff-files/sillymaps/travel_times.jpg`.

[28] Laszlo Mero. A heuristic search algorithm with modifiable estimate. *Artif. Intell.*, 23(1):13–27, May 1984.

[29] Nils J. Nilsson. *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann Publishers, San Mateo, CA, 1998.

[30] C.A. O'Flaherty and D.O. Mangan. Bus Passanger Waiting Times in Central Areas. In *Traffic Engineering and Control*, pages 419–421, 1975.

[31] OpenTripPlanner. Demos, Accessed 17 December 2011. `https://github.com/openplans/OpenTripPlanner/wiki/Demos`.

[32] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in FORTRAN, The Art of Scientific Computing*. Cambridge University, 2nd edition, 1992.

[33] PriceWaterhouseCoopers. Socit de vlo en libre-service: tats financiers. Montreal, Canada, 15 March 2011.

[34] SQLAlchemy. The Python SQL Toolkit and Object Relational Mapper, Accessed 11 December 2011. `http://www.sqlalchemy.org/`.

[35] Piet Van Mieghem. *Performance Analysis of Communications Networks and Systems*. Cambridge University Press, 2006.