Imperial College London

Department of Computing

Simulating and Optimising a Delivery Network of Autonomous Courier Agents

Author:
Jake Cracknell

Supervisor:
Dr. Krysia Broda
Second marker:
Professor Marek Sergot

A thesis submitted in fulfilment of the requirements for the degree of Bachelor of Engineering in Computing

June 2015

Abstract

Businesses and consumers use same-day city couriers to transport time-sensitive goods. Companies like CitySprint want to optimise the allocation of delivery jobs across its fleet of vehicles, so as to minimise cost. An inexpensive rival service is proposed using unmanned, autonomous cars, which also form a distributed computing platform for solving the allocation problem. A number of algorithms and routing strategies were developed for allocation, multi-waypoint route planning and fault tolerance. Several strategies that implement the contract net protocol are presented and local optimisation is performed using both a greedy and a genetic algorithm. Using mapping data from OpenStreetMap and live traffic data from HERE Maps, an interactive multi-agent simulation was built to test this network.

Using maps large and small, an extensive evaluation of the network is performed. The sensitivity of many key parameters is analysed and the optimal network size is found. Using sensible parameters and twenty cars, a courier network in Greater London was simulated and shown to be profitable charging only £2 + £2.37 per hour – a fraction of the current market price.

Acknowledgements

I would like to thank Dr Krysia Broda for her unparalleled support and interest throughout this project as my supervisor, and throughout my time at Imperial, as my personal tutor. I would also like to thank Professor Marek Sergot for agreeing to be my second marker, for his excellent suggestions during the project review and for his motivational comments on the interim version of this report.

Lastly, I wish to thank my extended family for believing in me. Without them, this project, nor I would exist.

Contents

Abstract i

Acknowledgements ii

Contents ii

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Objectives . . . 2
- 1.3 Report outline . . . 3

2 Background 5
- 2.1 Autonomous Agents . . . 5
  - 2.1.1 Contract Net Protocol . . . 6
- 2.2 Autonomous Cars . . . 7
- 2.3 Route Finding and Graph Theory . . . 7
  - 2.3.1 Haversine Formula . . . 7
  - 2.3.2 Strongly Connected Components . . . 8
  - 2.3.3 A* Search Algorithm . . . 9
- 2.4 Travelling Salesman Problem . . . 11
  - 2.4.1 Nearest Neighbour Algorithm . . . 12
  - 2.4.2 Genetic Algorithms . . . 12
- 2.5 Related Work . . . 13
  - 2.5.1 AORTA . . . 13
  - 2.5.2 Event-Driven Multi-Agent Simulation . . . 14
  - 2.5.3 An Agent-Based Simulation Tool for Modelling Sustainable Logistics Systems . . . 15

3 Design Specification 17
- 3.1 Customer Experience . . . 17
  - 3.1.1 Booking a Collection . . . 17
  - 3.1.2 Receiving a Delivery . . . 18
- 3.2 Business Operations . . . 18
  - 3.2.1 Pricing . . . 18
  - 3.2.2 Routing . . . 19
  - 3.2.3 Security . . . 20
- 3.3 Summary . . . 20

4 Data Sources **22**
4.1 OpenStreetMap . . . 22
4.1.1 Nodes, Ways and Relations . . . 22
4.1.2 Highways . . . 24
4.1.3 Road Delays . . . 25
4.1.4 Businesses . . . 25
4.1.5 Fuel Stations . . . 26
4.1.6 Faults . . . 26
4.2 HERE Traffic API . . . 27
4.2.1 Integration Challenge . . . 28
4.2.2 Discussion . . . 29

5 Implementation **30**
5.1 Implementation Language . . . 30
5.2 Discrete Event Simulation . . . 30
5.2.1 Event Logging and Statistics Collection . . . 31
5.2.2 Simulation Parameters . . . 33
5.3 Geography . . . 33
5.3.1 Data Structures and Objects . . . 33
5.3.2 Map Pruning . . . 35
5.3.3 Route Finding . . . 37
5.4 Traffic . . . 41
5.4.1 Traffic Flow . . . 41
5.4.2 Road Delays . . . 42
5.5 Courier Jobs . . . 44
5.5.1 The Dispatcher . . . 45
5.5.2 The Broadcaster . . . 47
5.5.3 Modelling Failure . . . 48
5.6 Autonomous Agents . . . 49
5.6.1 Movement and Positioning . . . 49
5.6.2 Job Fulfilment . . . 49
5.6.3 Fuel . . . 50
5.7 Graphics . . . 52
5.7.1 Map . . . 52
5.7.2 Graphical User Interface . . . 56

6 Optimisation **60**
6.1 Planning . . . 60
6.1.1 Problem Specification . . . 61
6.1.2 Preliminary Computation . . . 63
6.1.3 Nearest Neighbour Search . . . 64
6.1.4 Genetic Algorithm . . . 65
6.1.5 Other Attempts . . . 67
6.2 Routing and Job Allocation Strategies . . . 73
6.2.1 Contract Net Protocol . . . 73
6.2.2 Round-robin . . . 78
6.2.3 Free-for-all . . . 78

6.3 Idle Strategies . . . 79
6.3.1 Premature Refuelling . . . 80
6.4 Strategic Placement . . . 80
6.4.1 Unimplemented Strategies . . . 81

7 Evaluation **82**
7.1 Optimality of Algorithms . . . 82
7.1.1 Route Finding . . . 83
7.1.2 Planning . . . 86
7.1.3 Allocation . . . 89
7.2 Routing and Job Allocation Strategies . . . 89
7.2.1 Efficiency Challenge . . . 91
7.2.2 Reliability Challenge . . . 91
7.2.3 Availability Challenge . . . 92
7.2.4 Summary . . . 92
7.3 Idle Strategies . . . 94
7.3.1 Performance . . . 95
7.3.2 Refuelling . . . 96
7.3.3 Summary . . . 96
7.4 Dispatch Rate . . . 97
7.5 Dispatchers . . . 99
7.6 Job Difficulty . . . 101
7.6.1 Package Size . . . 102
7.6.2 Failed Deliveries . . . 103
7.6.3 Deadlines . . . 104
7.7 Vehicles . . . 106
7.7.1 Vehicle Type . . . 106
7.7.2 Population . . . 108
7.8 Economics . . . 113
7.9 Simulator and Software Design . . . 117
7.9.1 Running Time . . . 117

8 Conclusion **119**
8.1 Limitations . . . 120
8.1.1 Illegal Road Turns . . . 120
8.1.2 Agent Collision . . . 120
8.1.3 Job Stream . . . 120
8.2 Extensions . . . 121
8.2.1 Easy Additions . . . 121
8.2.2 Progressive Route Finding . . . 121
8.2.3 Handoff . . . 121
8.2.4 Traditional Couriering . . . 122
8.2.5 Map-Specific Strategies . . . 122
8.3 Closing Remarks . . . 122

A Table of Parameters **124**

A.1 Fulfilment . . . 124
A.2 Speed Limits . . . 125
A.3 Road Delays . . . 125
A.4 Dispatchers . . . 126
A.4.1 Dispatch Rate . . . 126
A.5 Vehicles . . . 127

B User Guide **129**
B.1 Installation . . . 129
B.2 Importing Maps . . . 129
B.3 Usage . . . 130
B.3.1 Loading a Map . . . 130
B.3.2 Configuring the Simulation . . . 130
B.3.3 Running the Simulation . . . 130
B.4 Stopping a Simulation . . . 131

C Playground **132**

D Extra Material **134**
D.1 Temporal Traffic Distributions . . . 134
D.2 Tables . . . 135
D.3 Maps . . . 136

Bibliography **138**

Chapter 1

Introduction

1.1 Motivation

Whilst communication is shifting away from physical letters, the growth of the Internet has brought about ever-increasing demand for parcel delivery. The express parcel delivery market is worth £5.8 billion a year and comprises 1.1 billion items, according to a 2010 report by Datamonitor [1]. Consumers and businesses are increasingly demanding speedier and time-slotted deliveries, which better fit in with their busy lives and business deadlines. In many metropolitan areas, same-day delivery services are available. Whilst they represent a tiny proportion of deliveries, this is mainly due to the high cost. Many small and mid-size businesses will use their own staff to make small deliveries, as this will often be the most economical option for them.

Efficiently running a same-day courier service is a logistical nightmare. Large courier companies use centralised computer systems to delegate delivery jobs to drivers. The exact protocols used remain trade secrets, however humans are ultimately in control. Those that solicit jobs at the companies' offices cannot know the true marginal cost per parcel, as they do not have an accurate state of the road network. Many companies use a computer system to generate quotes. These typically include a high 'call-out charge' plus a function of the straight-line distance from pick-up to delivery. It does not take into account the whereabouts of the couriers, which has a significant impact on the actual cost to the courier. If delivery men must carry out multiple jobs concurrently, their chosen routes may be suboptimal. Whilst some advanced satellite navigation devices support navigation to multiple waypoints, they cannot account for all of the constraints. The high cost for the customer can be attributed to this inefficiency and lack of real-time knowledge.

To illustrate the current state of the market, the following quotes were collected on Monday 1st June 2015 at 9 a.m. and were for the transport of a 50x50x50 cm, 10 kg box from Imperial College London locations, Charing Cross Hospital (W6 8RP) to South Kensington (SW7 2AZ). Google Maps estimate the journey to be 12 minutes and 4.2 km [2], which indicates a labour cost of £1.30 (assuming a National Minimum Wage of £6.50/hour) and a petrol cost of less than £1.00. Prices exclude VAT:

Company	**Quote**	**Guaranteed pick-up**	**Guaranteed delivery**
Royal Mail [3]	£33.00	10:00 (1h)	direct route time
CitySprint [4]	£13.95	09:46 (46m)	11:16 (+1h30)
Shutl.it (1) [5]	£10.83	10:15–12:00 (1h15–3h)	12:00–13:00 (+0h–2h45)
Shutl.it (2) [5]	£9.17	20:15–21:00 (11h15–12h)	21:00–22:00 (+0h–1h45)
Anywhere Sameday Couriers [6]	£75.00	10:00 (1h)	direct route time

These prices seem shockingly expensive. The motivation for this project is simply to answer the question, by eliminating human inefficiencies and carefully designing protocols, can a same-day courier service be made affordable for all?

1.2 Objectives

The project will investigate whether the use of *autonomous, self-driving road vehicles* can bring low cost same-day delivery. This entails:

- Accurately simulating autonomous vehicles driving on any large road network. For meaningful results, traffic must be modelled.
- Efficient and optimal multi-hop route finding with respect to distance and/or time.
- Efficient and optimal allocation of tasks to agents.
- Agents act autonomously in that all routing decisions are made locally, not centrally. This allows any computation to be done in parallel. It also means signal loss or individual failure of any component can be tolerated.
- Simulating a busy same-day courier service.
  - Jobs are spawned randomly throughout the day at times that generally reflect consumer demand. Their properties (locations, deadlines and physical size) conform to appropriate probability distributions, which can be adjusted for experimentation.

- Implementing and testing multiple protocols for the agents. Agents are designed to maximise profit by minimising distance, time and fuel consumption. Agents may cooperate or compete.
- Failure is modelled and remedied. Notably, the inability to pick up, deliver or meet deadlines.

1.3 Report outline

Chapter 2 presents background information relevant to this project and a critical overview of similar works. Later sections will assume an understanding of this. Among the material is an overview of the travelling salesman problem and two algorithms to find a near-optimal solution. A variant of this problem is that which must be solved to achieve efficient routes between waypoints.

Chapter 3 outlines the same-day courier service that is being simulated in non-technical terms, such that the reader understands the context and so as to guide the implementation of the simulator.

Chapter 4 introduces OpenStreetMap and HERE Maps – the data sources for this simulation's environment. This section discusses the decisions made and challenges faced in interpreting and unifying both datasets.

Chapter 5 fully details the implementation of the simulator written for this project: how the street map is represented internally; how route finding is performed; how traffic flow and obstacles such as pedestrian crossings are modelled; how the autonomous agents and their fuel supply are represented; and lastly how the state of the simulation is visualised.

With this platform, we then describe in detail in chapter 6 the algorithms and protocols developed to plan routes and allocate jobs to agents. After much research and many discarded attempts, the final solution to the planning problem (a variant of the travelling salesman problem), was to use a combination of a greedy and a genetic algorithm to find an efficient route that minimises any deadline violations. The allocation problem is solved using the contract net protocol – a kind-of reverse auction where agents bid the marginal cost for themselves to fulfil the job. An extension is detailed where agents are allowed to trade off jobs to each other in efforts to avoid delivering any late. A range of 'idle strategies' are proposed to control the behaviour of an agent when it has no jobs.

Chapter 7 is the evaluation wherein hundreds of simulations are performed and their results, plotted on graphs. It begins with a critical evaluation of the route finding, planning and allocation algorithms. We then determine the optimal routing and idle

strategy and how different factors, like package size and probability of failure affect performance. Using several different cities, from the small and quaint Scottish island of Stronsay to the sprawling behemoth that is Greater London, the ideal agent population is deduced. Using sensible parameters and these optimal populations, the break-even point is found and we arrive at an answer to our original question. Without modelling fixed costs, a London-based courier service with a fleet of twenty cars can be created and will break even, charging only £2 per job, plus £2.37 per hour of extra driving. At this price, the mean price paid by customers was only £3.75!

The report concludes with chapter 8 where the contributions of this project are critically evaluated, the limitations to the simulation are identified and a number of extensions to the project are proposed to the reader.

In the appendix, one will find a table of parameters in section A, listing the hundreds of parameters in the simulation and the default values that were used in the evaluation. To anyone who wants to install and run the simulator for themselves, a quick start guide is provided in section B. Section C contains a brief description of the 'playground' mode. This was a simulation environment designed to test and benchmark the routing algorithms and the stability of the simulator, by simulating the movements of thousands of agents. Section D contains some extra material that is peripheral to the main body of the report.

Chapter 2

Background

2.1 Autonomous Agents

This project endeavours to build and study a multi-agent system. Such a system must have autonomy: the agents act autonomously and independently, they are self-aware and they have no full global view of the system [7]. A MAS is decentralised in that there is no designated controlling agent – otherwise it would be deemed a monolithic system [8].

According to a paper entitled "A Taxonomy for Autonomous Agents" , an autonomous agent is a "system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future." [9]. An agent is goal driven and fundamentally, reacts to stimuli without the interference of the owner.

An autonomous courier vehicle can be characterised by this definition. Its environment is the road network – represented as a graph of nodes and vertices. Its stimuli are incoming, unallocated courier jobs, which are broadcast over the air to all agents, as well as (potentially) the positions and status of other agents. Its overarching goal is to allocate themselves to, pick up and deliver as many packages as possible, whilst conforming to each associated deadline. There are also secondary goals, such as refuelling whenever possible and minimising fuel consumption in various ways. These goals will be programmed in to all agents in the hopes that the multi-agent system, as a whole, will satisfy the owner's goal: profit.

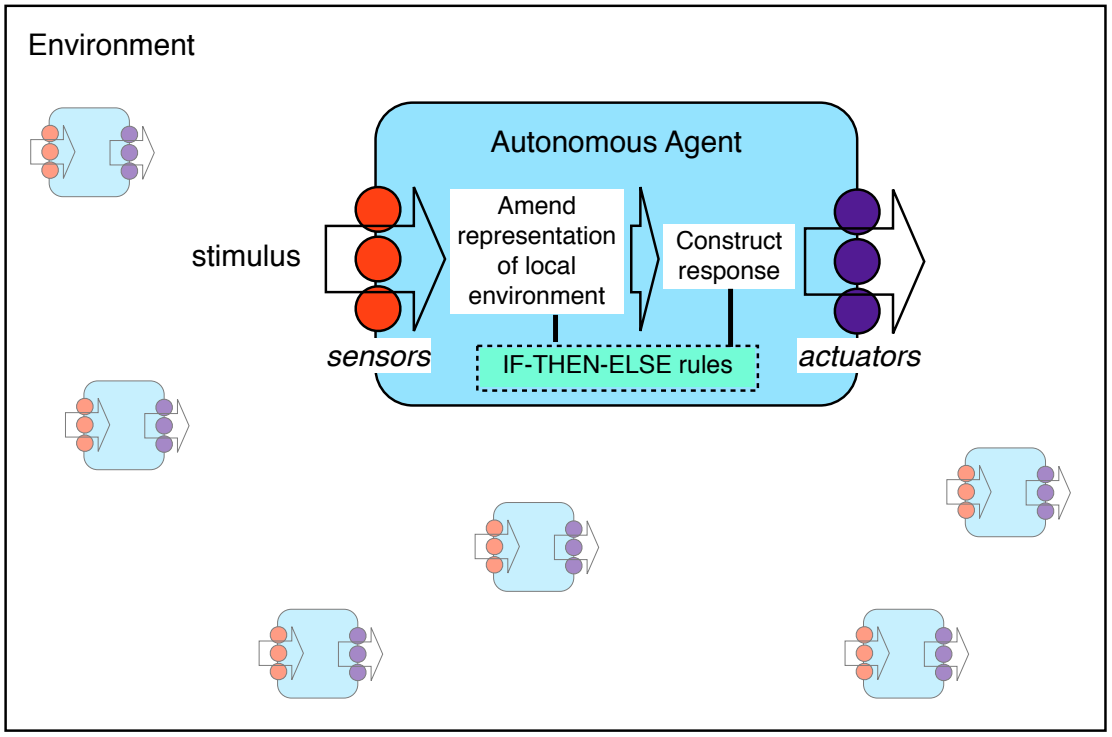


FIGURE 2.1: A multi-agent system in its most general form.

2.1.1 Contract Net Protocol

In a multi-agent system, the Contract Net Protocol provides a method for task-sharing through competitive negotiation. It is specified in [10]. Each agent of the 'contract net' is self-interested, so the solution may not be the most optimal for the network as a whole. When an agent hears a task, it breaks it down into independent sub-tasks and acting as 'manager', it broadcasts these to the net of 'contractors'. The negotiation happens in five stages:

1. **Recognition**. The manager receives a task, which it breaks down where possible.
2. **Announcement**. Specifications for each task, including constraints and meta-task information are serialised and broadcast to some or all contractors.
3. **Bidding**. Contractors that are capable of completing the task place a bid, which is transmitted to the manager.
4. **Awarding**. The manager decides on which contractor to award the contract to and notifies all those who bid.
5. **Expediting**. The winning contractor either performs the task, or assumes the role of manager and subcontracts to further contract nets.

In some problem domains, bids are associated with cost values and the negotiation resembles a traditional auction. For example, if there exist two capable contractors, but one poses a greater interest in the task, it will be awarded the contract.

2.2 Autonomous Cars

An autonomous, or self-driving car is one capable of sensing the environment and navigating without human input. Its vision and local mapping capabilities are realised using technologies such as radar, lidar, GPS and cameras [11]. When provided also with a wider representation of the environment, they are able to route autonomously through an urban road network. In many cases, it is possible to modify existing cars to incorporate the artificial intelligence. One such example is pictured in Figure 2.2. The research and development in this field is enormous; however this project innovates on a higher level – on the protocols for running a courier service, rather than the low-level technologies. The project abstracts the concept of self-driving cars to autonomous agents that are given coordinates to drive to and commands to stop or drive.

FIGURE 2.2: A Toyota Prius modified by Google to operate as a driverless car [12].

2.3 Route Finding and Graph Theory

2.3.1 Haversine Formula

As the simulation takes place on the oblate spheroid that is our Earth, calculating the surface distance between two points requires special consideration. The Pythagorean Theorem is for two-dimensional planes and can only provide a rough estimate of orthodromic distance, which is too inaccurate for route finding. It may be acceptable in some use cases, such as if one has a point P and needs to quickly determine which point from a large set of well-distributed points is *one of the closest* to P. If one adjusts the latitudes

in accordance with the Mercator projection (instead of ϕ, use $ln(tan(\phi) + sec(\phi))$, the accuracy improves, but there is still substantial error. Figure 2.3 illustrates this with an example.

For distance measurements that need to be accurate, the Haversine Formula for great-circle/orthodromic distances is the most appropriate. We define R as the radius of the sphere and for the Earth, we assume this to be 6,371 kilometres. In actual fact, the Earth is not a perfect sphere and the radius of the Earth varies from 6,353 km to 6,384 km [13]. However, a spherical approximation will be appropriate, as it will only be used for short distances and time efficiency will be crucial. Given two points, (ϕ_1, λ_1) and (ϕ_2, λ_2), the distance d is calculated as such:

$$a = \sin^2(\frac{\Delta\phi}{2}) + \cos(\phi_1)\cos(\phi_2)\sin^2(\frac{\Delta\lambda}{2})$$

$$c = 2\arcsin(\sqrt{a})$$

$$d = Rc$$

where between the two points, $\Delta\phi$ is the difference in latitude and $\Delta\lambda$ is the difference in longitude [14].

2.3.2 Strongly Connected Components

A strongly connected graph is one in which every node is reachable from every other node. As later discussed in 4.1.6, it will be necessary to find and remove disconnected components of the graph, or rather, prune all but the largest component of the graph. Furthermore, inescapable nodes must be pruned. This is so that agents will never be asked to route to somewhere inaccessible, or get 'stuck' having completed a route.

There are a number of algorithms that solve this problem by enumerating the strongly connected components. Whilst most utilise depth-first search, they differ in whether to perform several rounds of DFS or to do bookkeeping, trading off spatial efficiency for time efficiency. The former approach is that used in Kosaraju's algorithm [15], which must perform the second round of DFS on the transpose graph. As later documented in 5.3.2, a bespoke algorithm, somewhat similar to Kosaraju's, was devised to solve this problem.

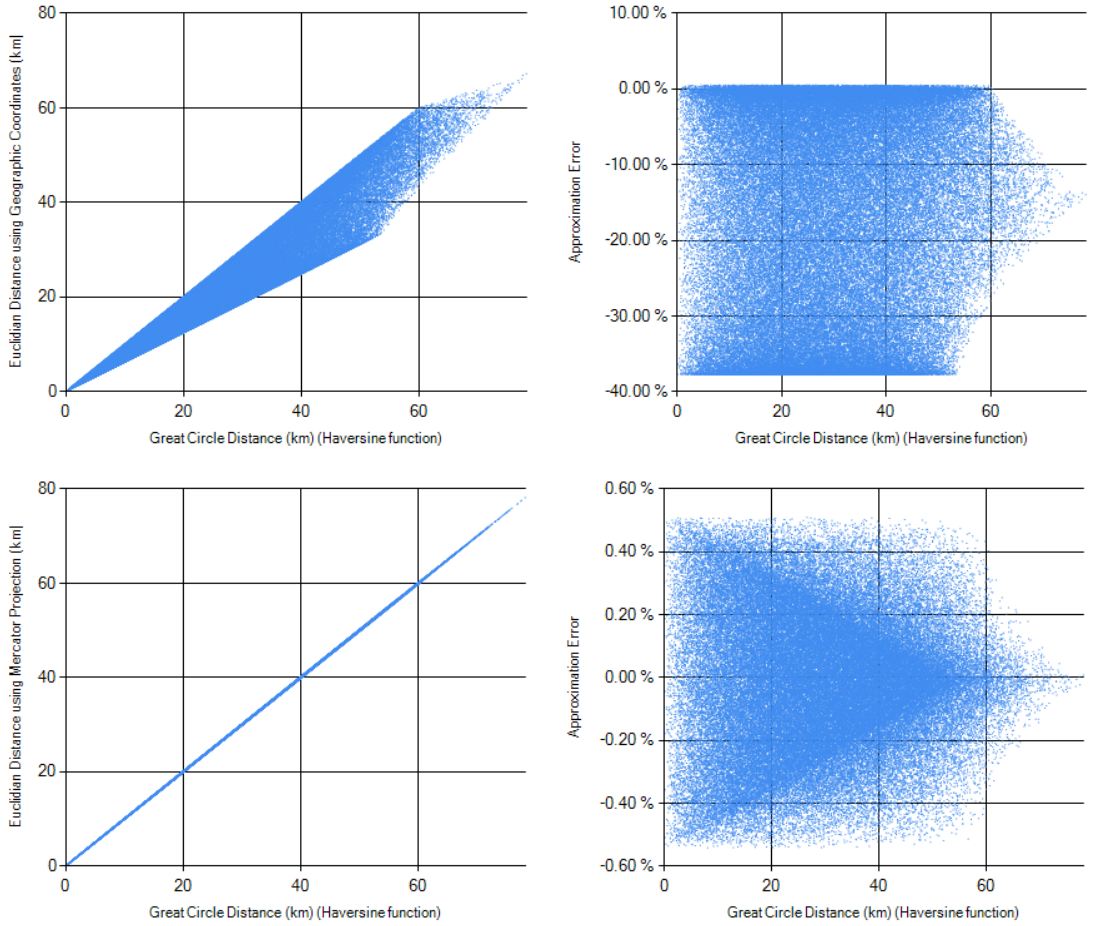


FIGURE 2.3: 10000 pairs of coordinates were randomly selected in Greater London (a 60 x 54 km map). Their distances were measured using the accurate Haversine formula, which is on all graphs' x-axes. In the first row of graphs, the Pythagorean theorem is used to approximate distance. In the second row, the Pythagorean theorem on a Mercator projection is used. For both approximators, the values on the y-axis are multiplied by the length of one degree of latitude at the map's midpoint. Next to each graph is a plot of the error as a percentage. In London, comparing with the true orthodromic distance, the Euclidean distance will be up to 37% less. Using the Mercator projection, Euclidean approximations will be ±0.5% the true value.

2.3.3 A* Search Algorithm

For a weighted graph, the shortest path problem is solved using Dijkstra's Algorithm. This will find the shortest path from one node to every other node in the graph. Though this will find an optimal solution, it is wasteful if one is only interested in a single path.

The A* search algorithm uses a best-first search, expanding nodes that have the lowest score and hence are the most promising. This section references the original formal specification [18]. It uses a heuristic/evaluation function $f(n)$ that combines the full cost of the path to a node $g(n)$, with the estimated remaining cost $h(n)$ into an estimated total cost. A* will always find the optimal solution if one exists, as long as there are

finitely many nodes n with $f(n) \leq f(goal)$ and the heuristic is admissible. $h(n)$ is admissible iff $\forall n.h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost. In other words, it must never overestimate, otherwise the path that is returned by the algorithm may not be the most optimal. An underestimation or correct guess is allowable.

Theorem 2.1. *Path p_1 found by A* is guaranteed to have the lowest path cost.*

Proof. Assume a shorter-cost path, p_2 exists (i.e. $g(p_2) < g(p_1)$). Some partial path of p_2, named p_3 exists also. $p_1, p_2, p_3 \in S$, the open set. Because p_1 was expanded before p_3, $f(p_1) \leq f(p_3)$ and because p_1 reaches the goal, $h(p_1) = 0$, so $g(p_1) \leq g(p_3) + h(p_3)$. Because h is admissible and p_2 extends p_3 towards the goal node, $g(p_3) + h(p_3) \leq g(p_2)$. Therefore $g(p_1) \leq g(p_2)$. Contradiction! □

Algorithm 1 A* Search Algorithm

```
1:  function FIND-SHORTEST-PATH(GRAPH, START, GOAL)
2:      openset = [start]                          ▷ discovered, unexplored nodes
3:      closedset = []                             ▷ discovered, fully explored nodes
4:      predecessors = {}          ▷ map of node edges, forms tree rooted at startnode
5:      gscores = {(start, 0)}          ▷ map of best known path costs for each node
6:      while openset not empty do
7:          choose n ∈ openset such that f(n) is minimal
8:          if n = goal then
9:              return path reconstructed from predecessors[n]
10:         move n from openset to closedset
11:         for all (n, neighbour, cost) ∈ graph do          ▷ test all of n's neighbours
12:             if neighbour ∈ closedset then
13:                 skip                                  ▷ assumes consistent heuristic
14:             g = gscores[n] + cost                  ▷ a tentative g(neighbour)
15:             if neighbour ∉ openset or g < gscores[neighbour] then
16:                 predecessors[neighbour] = n
17:                 gscores[neighbour] = g
18:                 if neighbour ∉ openset then
19:                     h = costestimate(neighbour, goal)      ▷ an admissible estimate
20:                     f = g + h
21:                     add (neighbour, f) to openset
22:     return null                                       ▷ no path exists
```

$h(n)$ is monotonic if the total estimated path cost $f(n)$ does not decrease as the heuristic goes down the node tree. A heuristic that is admissible and monotonic is also termed consistent. For some problem domains, this non-monotonic heuristic will demand a modified algorithm. Line 13 exists for efficiency – there is no point recomputing $h(n)$ for an already explored neighbour, as it is expected to always be greater. Using a non-monotonic heuristic will mean this sometimes may not be the case. The line would be

replaced with code to update the neighbour with the new, lower, g value and update the neighbour's parent in *predecessors*, representing a different, more optimal path.

For an automotive navigation system, these costs can be specified as distances, travel time or even fuel usage [19]. If we wish to minimise distance, the heuristic estimate will be the orthodromic, *as-the-crow-flies* distance to the destination and the total cost will be the sum of every orthodromic distance across each arc used. Any heuristic function considers the possibility of a straight-line highway H from current node n to the destination d, permitting any speed. Trivially, no path can be shorter than the straight-line distance, so the heuristic is admissible. To find the fastest path, one must somehow calculate the time it takes to traverse each arc. $h(n)$, the time to drive at full speed along H is admissible, as it would be impossible to drive from n to the destination node in a shorter time. Hence $h(n)$ will be a function of the orthodromic distance and the maximum legal speed of the vehicle. Likewise for fuel usage, each arc cost will be a function of the distance and the best fuel economy achievable (measured in miles per gallon) given a road's speed restrictions. If the car is most efficient at 30 mph, $h(n)$ would be equal to the fuel usage driving down H at 30 mph. All of these heuristics are consistent, although they assume the arc costs do not change during the journey.

2.3.3.1 Bounded Relaxation

It is possible to speed up an A* search by increasing the effect of the heuristic function. If $h(n)$ is admissible, replacing the equation on line 20 to instead calculate $f(n) = g(n) + (1 + \varepsilon)h(n)$ may lead to a non-optimal solution, but it will not be more than $(1 + \varepsilon)$ times worse than the best solution [20]. Look ahead to sections 5.3.3.1 and 7.1.1 to see how this technique is used in the simulation and the real world trade-off between speed and optimality.

2.4 Travelling Salesman Problem

Given a set of connected nodes in a graph, to find the shortest possible path that visits each node and returns to the starting position is an $\mathcal{NP}$-hard problem [21]. The same is true for finding the optimal Hamiltonian path, which starts at some chosen node and ends at a different node. This project will be heavily interested in this class of problem, of which there exist many variants and methods of finding optimal or near-optimal solutions. This section presents two heuristic methods to find near-optimal solutions. Later on in section 6.1, it is explained how these algorithms were adapted for the courier simulation, where there are additional constraints and a different definition of optimality.

2.4.1 Nearest Neighbour Algorithm

The nearest neighbour algorithm is a simple, greedy heuristic for solving the travelling salesman problem [21]. It begins with a single node n and constructs a path by repeatedly visiting the nearest unvisited node. Once there exist no further unvisited nodes, an arc from the last node to n is added and the algorithm terminates. If only a Hamiltonian path is required, this last step is omitted. It runs in quadratic time, specifically, requiring $\frac{1}{2}(n-1)n$ distance calculations.

2.4.2 Genetic Algorithms

As with many other optimisation problems, a genetic algorithm that mimics aspects of natural evolution – inheritance, mutation, selection and crossover – can be used as a search heuristic [21]. The initial population may be a randomly generated set of solutions or it may be seeded with near optimal solutions. A fitness function must be defined to assess the quality of a solution and in the case of the travelling salesman problem, it will simply be the length of the path or cycle. In brief, a genetic algorithm would maintain a large population of solutions (chromosomes), adding and subtracting from this pool over the course of many loop iterations. There may be a stopping criterion or it could simply run for a set number of iterations. Such a decision depends on the time constraints for the system; the latter guarantees termination in a known amount of time, whereas the former could find a more optimal solution. In each iteration, it selects the top k solutions as ordered by their fitness. Depending on the implementation, it will perform one or both genetic operators:

- **Crossover**. Creating a new chromosome, constructed using aspects of two of the selected chromosomes. There are many methods for doing this, outlined in [22].
- **Mutation**. Slightly modifying the chromosome, such as by swapping two randomly picked nodes of the sequence. This can be useful on its own, mimicking natural selection of asexual organisms. It can also be used in conjunction with crossover to maintain genetic diversity and avoid local minima.

The child chromosomes are added back into the population. Through the process of natural selection, it is expected that after many generations, the best solution in the population is suitably optimal.

2.5 Related Work

This chapter will conclude by showcasing similar research projects that are publicly available. It is very surprising how much related work there is. There seem to be many similar papers that document the simulation of autonomous vehicles, city courier networks and other transportation domains. Despite this, it appears that this project will be the first to simulate a same-day courier network that uses dynamic routing and planning algorithms to maximise efficiency. Having read many papers, it seems that a common pitfall was that the routing strategies were rarely compared or the results were inconclusive. Nonetheless, there will be important insights to be gained from analysing all these related works, even those that don't directly relate to this project's investigation. This section will showcase three papers that contributed the most to this problem domain, delivered a concrete implementation and crucially provided some results, so that this project can go above and beyond the state of the art.

2.5.1 AORTA

Approximately Orchestrated Routing and Transportation Analyzer was a software simulation developed to model and optimise traffic flow of autonomous vehicles in metropolitan areas. It was initially an undergraduate project and was later published as an academic paper [23] [24]. Though this project will focus mainly on multi-hop routing as a means to transport parcels, it will simulate traffic to a degree and this work provided some valuable insight to the routing challenges I would face.

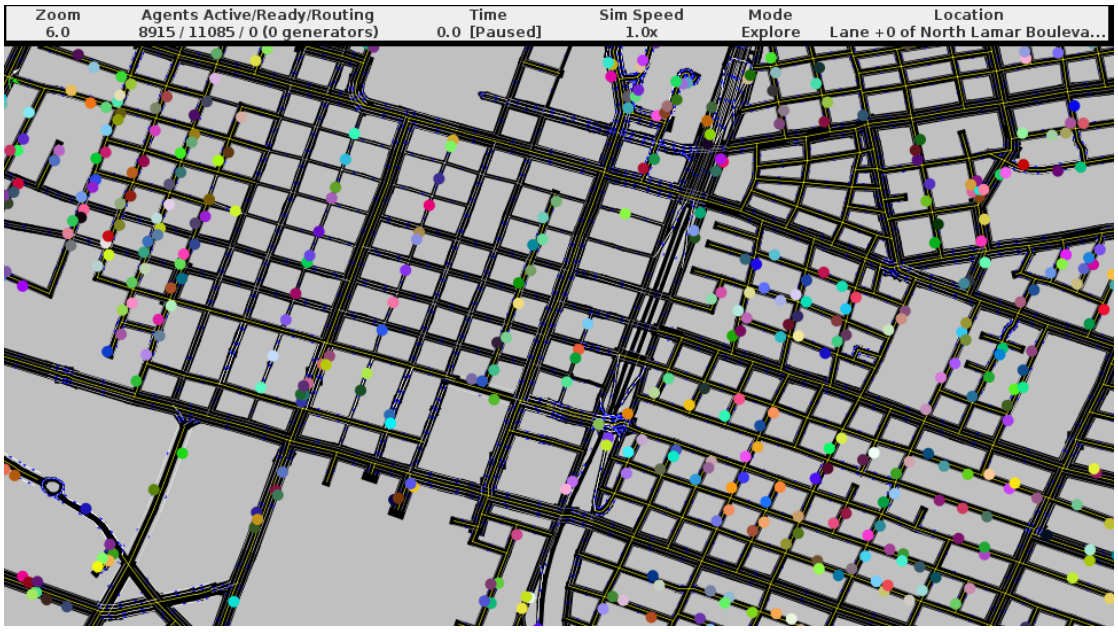


FIGURE 2.4: AORTA: visualising autonomous agents in downtown Austin, Texas.

I judged their design to be somewhat flawed in certain areas, though I understand their reasoning. Their map loader attempts to simplify certain map artefacts that were correctly drawn by OpenStreetMap contributors, but do not serve their functional purpose if interpreted literally. Namely, two three-way intersections that are closely linked (less than 50 metres apart) are combined into a single four-way intersection. This eliminates unintended bottlenecks in their simulator [26]. In my opinion, hard-coding map alterations like these makes the loader less flexible. Were it applied to other maps, errors are likely to crop up, ranging from the benign (e.g. slightly suboptimal routes) to the fatal (e.g. roads becoming inaccessible or disappearing, maps becoming altered to the state that they are unrecognisable).

Many of the novel concepts proposed are transferable. In a follow-up paper, the authors detail a new auction-based system for intersection management [25]. When a number of vehicles would like to use an intersection to continue on their route, agents competitively place bids at a price proportional to: (1) the utility of being able to use the resource and (2) how many auctions it intends to bid on. The 'winner' is then free to use it and a new round of bidding can take place if needed. From this, the paper investigates different 'wallet agents' to place bids on behalf of the car: *free-rider* (never bids anything), *static* (unlimited funds, always bids a fixed amount, useful for emergency vehicles) and *fair*. This latter wallet is initialised with funds proportional to the journey length and at any given intersection, it will bid $\frac{funds\_left}{num\_of\_intersections\_left}$.

This, among other papers and self-initiated research prompted me to use OpenStreetMap as my data source and internal data model.

2.5.2 Event-Driven Multi-Agent Simulation

A paper, originating from the University of Manchester and presented at AAMAS'14 (the 13th International Conference on Autonomous Agents and Multiagent Systems) explored the use of a framework called FAMOS for multi-agent simulations [27]. These were performed using an event-driven approach, which differs from the more common discrete time-driven approach. The difficulty lies in that future events must be predetermined – scheduled at the time of previous, causal events.

It is particularly relevant as it uses a "City Courier Service" as an example model. The model presented was complex in that it used a real-world street map, agents were either bikes or cars and the speeds were determined by the type of road. Though it does not go into the exact detail, the agents decide amongst themselves who fulfils the delivery using an adaptation of the contract net protocol, as described earlier in section 2.1.1.

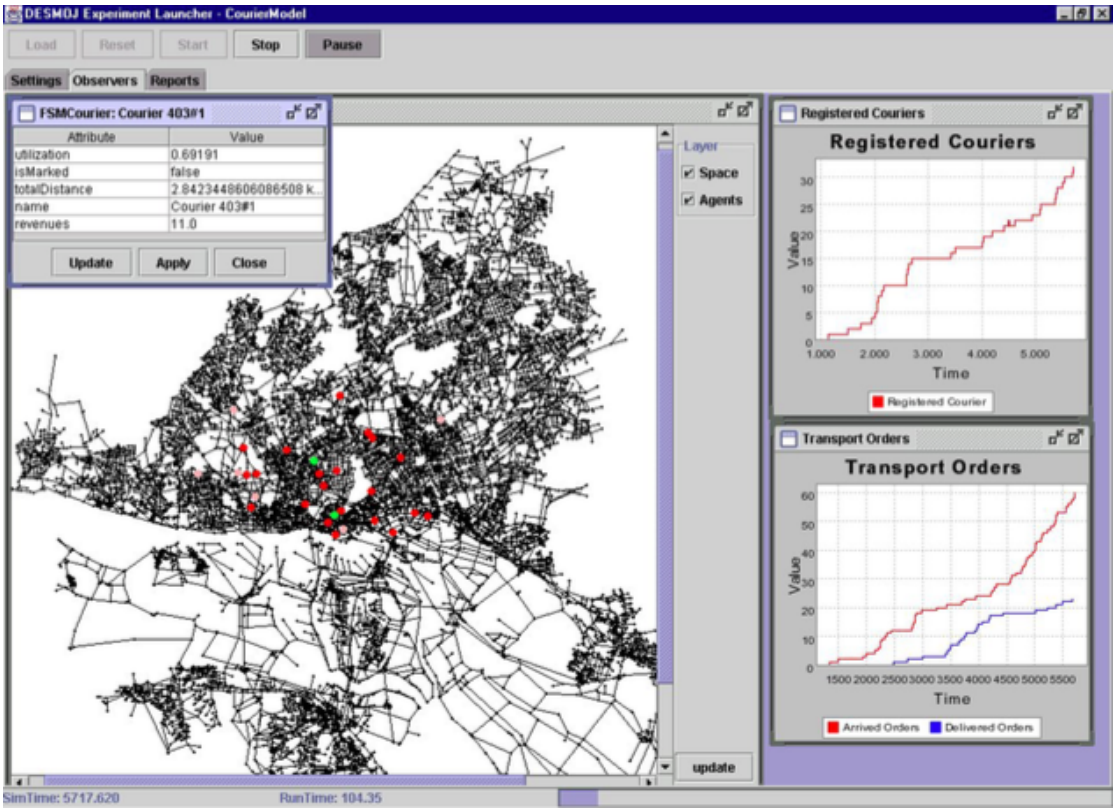


FIGURE 2.5: FAMOS simulation. City courier service in Hamburg, Germany.

Though the paper advocates writing an event-driven simulation, it was not convincing enough that this approach will be more appropriate, despite the simulation model being very similar. If one is to add in randomness and unknown time delays, particularly those arising from the uncertainty of traffic, there will be too much overhead (i.e. too many events) to implement a fully event-based simulation. The state of the world is dynamic and so too will be the behaviour of the agents. The agents in the FAMOS simulation did not respond to new jobs when they are non-idle, however this is essential for the courier service that this project proposes. Despite this, the paper was informative as it demonstrated a working CNP implementation. This provided some reassurance that CNP (once modified to account for constraints like vehicle capacity that the author had abstracted) would be appropriate for this project.

2.5.3 An Agent-Based Simulation Tool for Modelling Sustainable Logistics Systems

Finally, a 2003 joint study between the University of Hamburg and two local couriers investigated optimal strategies to solve the city courier problem. They too used FAMOS and the city of Hamburg for their investigation [28]. It compared the 'status quo' method of door-to-door delivery with the use of hubs. In particular, the **hub and**

shuttle strategy would require hubs to be placed in lucrative areas. Hub-to-hub deliveries would take place periodically and as the packages can be bundled together, efficiency gains were promised. For lengthy deliveries, local couriers would only travel between customers and their local hub. The **inside/outside** strategy requires a single central hub and the classification of an inside zone of high order volume and many outside zones. *Intra*-zone deliveries would take place as usual, but all other deliveries must pass through the central hub. At the hub, inside-zone packages would be speedily delivered by bike, whilst for each outside zone, packages would be bundled together and sent out in batches. Surprisingly, neither strategy brought consistent benefit when evaluating on total distance travelled.

A follow-up research paper proposed a new strategy, using **fixed exchange points** [29]. A complex preprocessing step is taken – the city is divided up into many clusters that best reflect the flow of orders previously observed. As they come through, cross-cluster deliveries are split up into several orders using the Floyd-Warshall algorithm, which minimises the number of exchanges to take place at cluster boundaries. Preliminary results showed this strategy to be more efficient, but very sensitive to suboptimal cluster creation.

The studies used a mediation protocol that closely resembles CNP: “When an order is placed, the office announces the request to all potential couriers in order of their priority. A consignment with bicycle preference is e.g. announced to inactive bicycle-couriers first, then to active bikers and finally to motorised couriers. The couriers rate orders according to a quantitative rating function".

The use of one or many central hubs goes against the spirit of this project, which is to evaluate the profitability of a decentralised autonomous courier network. However, the methodical way in which these papers presented and evaluated ‘strategies’ will be mirrored in this report. Furthermore, it provides some useful insight on the expected efficiency gains that will come from ‘bundling’ deliveries together. This principle is expected to be one of the major factors in this project that will drive costs down.

Chapter 3

Design Specification

This chapter will provide a non-technical specification of the same-day courier service that will be simulated – first from the customer's perspective and then from a business perspective. At the end of this chapter, one will find figure 3.2 – a flow chart that summarises the process. This chapter has been written so as to guide the development of the simulator. Some details are largely irrelevant to the simulation, however they are provided to convince the reader that such a service is feasible (or not).

3.1 Customer Experience

As customers will not be interacting with a human courier, the user experience is vastly different. Therefore, both sender and recipient will need to be guided through the process to minimise the chance of failure.

3.1.1 Booking a Collection

The customer who books and pays for the delivery will need to use the company's website or mobile application. A user account, consisting of the customer's contact and payment details will need to be set up. When ordering, they will need to provide the source and destination addresses of the delivery, a rough estimate of the size of the shipment, a phone number for the recipient and the deadline for the package to be delivered. A price is displayed or in rare cases, an apology, with a hyperlink to another local courier who may be able to take the job. If the sender agrees to the price, they are asked to wait up to a few minutes for confirmation. During this time, the recipient receives a pre-recorded phone call, asking them to confirm that they will be able to physically take in a delivery order to themselves up to the deadline. Additional instructions are spoken

on request. If and when it is confirmed, both are presented with a hyperlink they can use to track the delivery vehicle in real-time. If the recipient is unavailable, the sender is unable to book the job.

The customer cannot expect an immediate pick-up, but they can expect it to take place early enough such that the delivery deadline is met. They are notified by a pre-recorded phone call five minutes prior to its arrival and told a pin code. Upon arrival, unless specifically requested, the vehicle will sound its horn or play a pre-recorded message. The sender is expected to walk up to the vehicle, enter a pin code, open the doors and place their package into the vehicle. They are given two minutes to do this. Their payment method is charged when the van departs and if the customer does not load the package, the price is partially reduced based on any cost savings.

3.1.2 Receiving a Delivery

The recipient is informed by a pre-recorded phone call when the van is five minutes away. Upon arrival, unless specifically requested in this call, the vehicle will sound its horn or play a pre-recorded message. The recipient is expected to walk up to the vehicle, enter a pin code, open the doors and retrieve their package into the vehicle. A spotlight shone onto the package is used to help the user identify theirs, given there may be many to choose from. They are given two minutes to do this. If they fail to retrieve their package, the vehicle will continue with its later deliveries and at some point deliver the package to one or one of many depot/collection points. The recipient and sender are notified of their options: have the package disposed of, collect the parcel themselves or to arrange another autonomous delivery, charged at the standard rate.

3.2 Business Operations

3.2.1 Pricing

As customers ask for quotes, they are quoted a price proportional to the amount of additional driving time it will take to pick up and drop off the delivery, plus a base price. Existing contracts must not be broken, however, so some demands must be refused if no vehicle can confidently replan their route to incorporate the new job, without missing deadlines. An example is shown in figure 3.1. The replanning must also ensure that at no point the vehicle would become too full or run out of fuel. By providing a link to an alternative, likely more expensive courier, additional revenue can be earned as an affiliate.

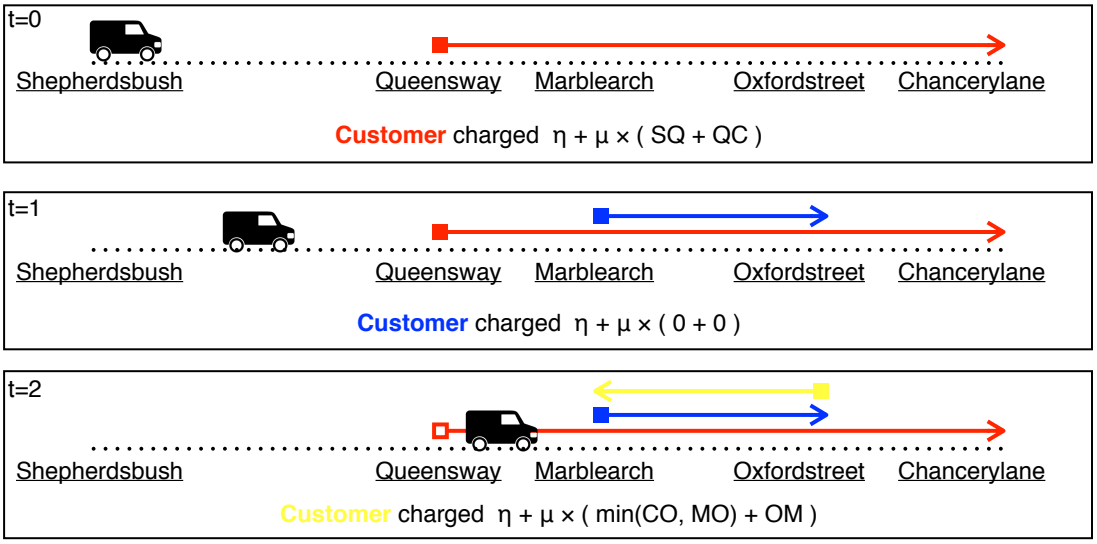


FIGURE 3.1: An illustrative example of courier pricing in proportion to driving time. Coloured arrows represent unfulfilled jobs that are booked in at different points in time, t. They span from the pick-up to the drop-off location and the points used as examples lie along a straight road. A route cost is the driving time in current traffic conditions (e.g. for Shepherd's Bush to Queensway, SQ is typically 0.2 hours). At time 0, the red job is booked and the customer is charged for the driving time from Shepherd's Bush to pick-up location, Queensway, plus the additional time thereon to Chancery Lane. Shortly after at time 1, the blue job from Marble Arch to Oxford Street is booked and the customer only pays the base price, because they are 'piggybacking' on a route already paid for by the red customer. At time 2, the agent has now picked up the red job. The yellow customer books a more difficult job from Oxford Street to Marble Arch. Here, there is no piggybacking possible and the cost to drive from O to M must be paid. However, the agent can choose to either complete the red job first and double back or complete the blue job first and return back to O. Its choice will be whatever route is shorter and cheaper for the yellow customer, Chancery Lane to Oxford Street or Marble Arch to Oxford Street. Base price η and unit price μ are constants that can be set after running the simulation and performing break-even analysis (in 7.8).

3.2.2 Routing

When jobs are assigned to the vehicles, they incorporate them into their routes in the most optimal way. For this reason, it is not possible to give customers an estimated pick-up and drop-off time until five minutes prior to the event. Vehicles must not divert their routes in this window, however seemingly arbitrary rerouting will be commonplace at other times. There is a rare edge case when this promise must be revoked, when a delivery is followed immediately by a pick-up, but the delivery failed. The vehicle may not have enough room to carry both and so must notify the customer.

A 'failed' waypoint refers to the situation when the customer does not put in or take out their package from the vehicle. This would often be due to the customer not being in. Failed pick-ups may be partially refunded, but the base price will always be payable. The exact amount will be proportional to the amount of driving time saved after removing

the now unneeded delivery waypoint and replanning. Due to the triangle inequality, this will never be zero. Failed deliveries are rerouted to one or one of many depot/collection points. The deadline for this revised drop-off is twelve hours, so it is unlikely to pose a substantial fuel or opportunity cost, unless it is a physically large package. Low failure rates are expected in practice due to the real-time tracking and frequent notifications.

3.2.3 Security

Delivery vehicles are fitted with high resolution cameras and the floor of the vehicle is a set of scales. A cellular network connection and GPS is fitted, as is standard with autonomous vehicles. If theft, vandalism or other mischief is detected, it can alert the company and the police. Faults with the vehicle can also be reported to the company or its insurers, so that an engineer can repair it.

When the sender is invited to place their package into the vehicle, it will politely notify them by pre-recorded message if the scales and cameras detect a package to be removed. The same occurs if the recipient begins to remove packages that are not theirs. If a harsher message is also ignored, the vehicle will send a distress signal and drive off to complete further deliveries, whilst a company representative can inform the customers whose packages were stolen.

It should be noted that this service we propose is not designed to carry high-value consignments and compensation offered to customers in the event of loss would be capped at a low amount. The simulator will not model these security measures, but the protocols developed should be able to accommodate them as extensions when they are installed on actual vehicles.

3.3 Summary

A summary of the courier business is presented as a flow chart overleaf, in figure 3.2.

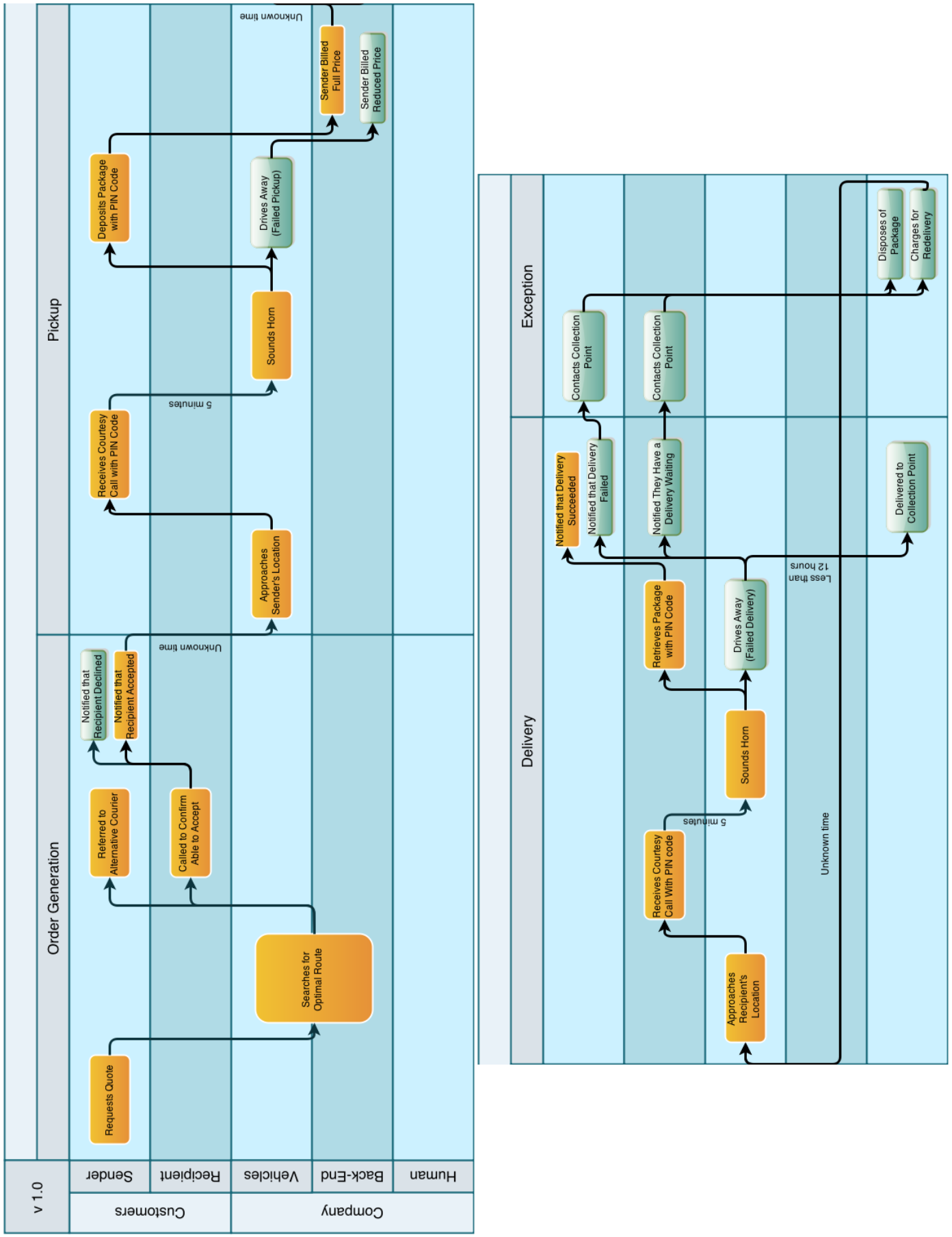


FIGURE 3.2: Flow chart showing the operation of the proposed business. It is split in two and is to be read left-to-right, then top-to-bottom.

Chapter 4

Data Sources

4.1 OpenStreetMap

The simulator uses real-world road map data from OpenStreetMap – a UK-based non-profit, Wikipedia-inspired, collaborative project that strives to map the whole world. Its data is crowdsourced from the general public and made available under the Open Database License, so it is free to use for research purposes, among others [30].

OSM data for a given region can be exported as an XML file, which lists the data primitives – nodes, ways and relations. The simulator enumerates such files in its working directory and upon the user selecting one, it will parse the XML into its internal data structures. Theoretically, any sized region can be loaded, however dense cities occupy a large amount of disk space. Instead of using the standard .NET $XMLDocument$ loader, to improve on speed and memory utilisation, the program developed uses a stream-based reader to navigate the file and extract only the data required.

Greater London, as pictured, is 904 MB as an uncompressed XML file and utilises 1.27 GB of memory for 3,856,726 nodes and 167,414 ways (716,148 arcs).

4.1.1 Nodes, Ways and Relations

Minimally, a **node** is a point that consists of a unique ID and a pair of latitude and longitude coordinates [31]. They can be used to identify standalone features, such as trees (`natural=tree`) and telephone boxes (`amenity=telephone`). These *tags* are key-value pairs associated with nodes, ways and relations. Both the key and value are free format text fields, however in practice there are agreed conventions of how tags are used.

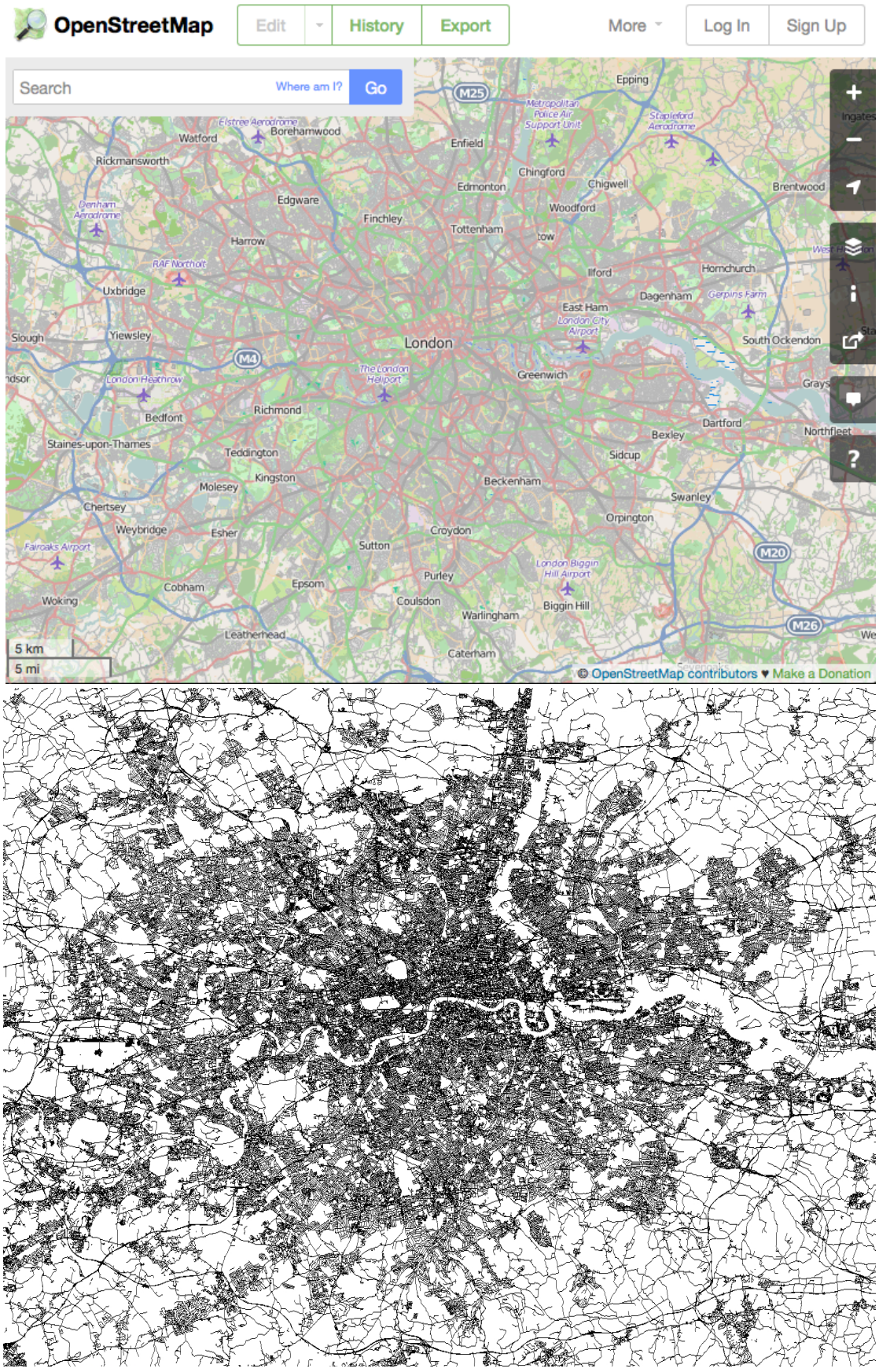


FIGURE 4.1: Greater London, as pictured on OpenStreetMap.org and in the simulator.

A **way** is a linear feature, defined in the XML structure as an ordered list of 2–2000 node IDs, together with some tags [31]. In terms of graph theory, they are *directed paths*. As the arcs connecting two nodes are straight lines, ways will often consist of many short arcs, so as to represent a curve in the real world. An *open way* can take many forms, such as `waterway`, `railway`, `aeroway`, `coastline` and indeed, `highway` [32]. There are also *closed ways* and *areas*. These are polygons – i.e. the last node of the way is also the first node. Examples include roundabouts and areas tagged as parks (`leisure=park`). The module that parses this data is indifferent to the particular classification of way.

Relations are multi-purpose data structures that document a relationship between two or more nodes, ways and/or other elements [31]. A common use is to label bus and cycle routes, which will typically consist of an ordered list of ways. Another use is to accurately identify turning restrictions across intersecting ways. This highlights one abstraction that the simulator must make: irregular turning restrictions are ignored, as to properly model them would either add a large overhead to the routing algorithms or require a complete, messy redesign of the data structures that represent nodes and edges. Such a redesign would degrade the time and spatial complexity of the simulation, making large city simulations infeasible.

4.1.2 Highways

This application is mostly concerned with ways that are tagged as `highways`, specifically those that can be traversed by a privately-owned autonomous car. OSM define this key as "any kind of road, street or path" [33]. The `highway` tag will often have such values as `track` (agricultural roads), `path` (footways), `cycleway`, `bus_stop` and other such things that are more suited to pedestrian and public transit applications [34]. These ways are discarded and the remaining highways are filtered further on the tag `access` to exclude those that are tagged as private, only for public service vehicles (e.g. routes through bus stations) or no throughfare.

Ways can be bidirectional or one-way and these are parsed as such. There exist some very rare instances of `oneway=reversible`, which identifies a road (typically a lane of a main road) that's direction is scheduled to switch at certain times of day to accommodate traffic flow. Such cases are assumed to be always one-way in an arbitrary direction. Note that no such roads exist in any of the maps used for evaluation.

The tag, `maxspeed`, identifies the maximum lawful speed for vehicles on a given highway. This is parsed and converted from miles per hour to kilometres per hour where needed. If no tag is present, the simulation will naively assume it to be the national speed limit for that type of road and vehicle class.

4.1.3 Road Delays

When navigating the public road network, much of the journey time is spent waiting at traffic lights and pedestrian crossings. These are mapped in OpenStreetMap as tagged nodes of the road network. The parser labels the nodes as they are iterated through, filtering for the following particular key-value combinations. As tagging standards for crossings have changed over time and some contributors provide more detail than others, nodes often have multiple tags, such as `highway=crossing; crossing_ref=zebra`, which may appear in any order. For this reason, all tags are parsed and the classification of the greatest magnitude is used:

1. `railway=level_crossing` marks a level crossing. *Highest magnitude.*
2. `highway=traffic_signals` marks a set of traffic lights, typically used at an intersection to regulate traffic circulation.
3. `crossing=traffic_signals` and `crossing_ref=pelican/toucan/pegasus/puffin` mark pedestrian crossings that use traffic lights.
4. `crossing=uncontrolled/zebra`, `crossing_ref=zebra` and `highway=crossing` mark pedestrian crossings that do not use traffic lights and hence are likely to have fewer users. These crossings are not counted at all if they lie on a residential or service road. *Lowest magnitude.*

Some potential road delays are ignored, as they are of very low abundance. These include movable bridges such as Tower Bridge in London and toll booths.

4.1.4 Businesses

Same-day city couriers are primarily used by businesses, hence the simulation spawns a high proportion of jobs with pick-up or drop-off locations that correspond to real businesses. A typical city contains thousands of nodes and ways that are tagged as being one of the following:

- `shop=*`. Possible values include `supermarket`, `convenience`, `fashion` and `florist`.
- `office=*`. Possible values include `company`, `lawyer`, `accountant` and `government`.
- `craft=*`. Possible values include `carpenter`, `shoemaker`, `brewery` and `electrician`.
- `amenity`. Only the values `school`, `restaurant`, `bank`, `fast_food`, `cafe`, `kindergarten`, `pharmacy`, `hospital`, `pub`, `bar`, `fire_station`, `police` and `townhall` are included.

These nodes would not lie directly on the road network, but in most cases they will be defined within a close proximity. In some cases, such as with shops that are within a shopping mall, this would not be the case. In any event, the parser identifies and names (by parsing a tag like `name=HSBC`) these nodes and adds them to a list, such that they can be highlighted in the map display. Many businesses are not represented as nodes, but instead as closed ways (polygons that resemble the building shape). In these cases, the parser adds only the first node of the way to the list. Visiting a business node involves routing to the geographically nearest node that is part of the road network.

In the simulator's GUI, where the name of the business is provided, it is shown verbatim. Otherwise, for unnamed shops and offices, it is transformed for readability like so: `shop=mobile_phone` becomes "Mobile phone shop". For businesses of keys `amenity` and `craft`, just the value is shown (e.g. "Fast food" or "Optician").

4.1.5 Fuel Stations

The vehicles need designated points where they can refuel. All petrol stations are mapped in OpenStreetMap as nodes or ways, under the tag `amenity=fuel` [35]. This can be parsed alongside businesses and was initially done so to get a list of node IDs. However for greater flexibility, it was decided to store fuel points as editable plain-text node IDs in an *.aa* file to accompany the *.osm* (XML) file. This meant it was easy to add missing stations and remove incorrect entries. In our model, depots serve as fuel stations as well. After parsing the map, the *.aa* file is read to reveal depot node IDs on the first line and fuel point node IDs on the second. These nodes are stored in separate lists for fast lookup. The simulation makes the assumption that all fuel points and depots operate 24 hours a day. As depots are frequent delivery waypoints, they are uniquely labelled as 'DEPOT A' to 'DEPOT Z'. Figure 4.2 shows all of the fuel stations (of which some have been labelled depots also) mapped in Greater London. It also shows all yellow business nodes and red road delay nodes.

4.1.6 Faults

As OpenStreetMap crowdsources its data, it is inevitable there will be inaccuracies in the properties and tags of nodes and ways. Incorrect or missing information will, to some extent, limit the realism of any simulation. For example, speed limits of roads are sourced mainly from users and the property is often unspecified. In these cases, the parser must estimate the speed limit based on the laws of the country the simulation takes place in. The UK government clarify the speed limits that usually apply, but local councils specify different speeds, which are signposted [36], but not always mapped. Furthermore, this

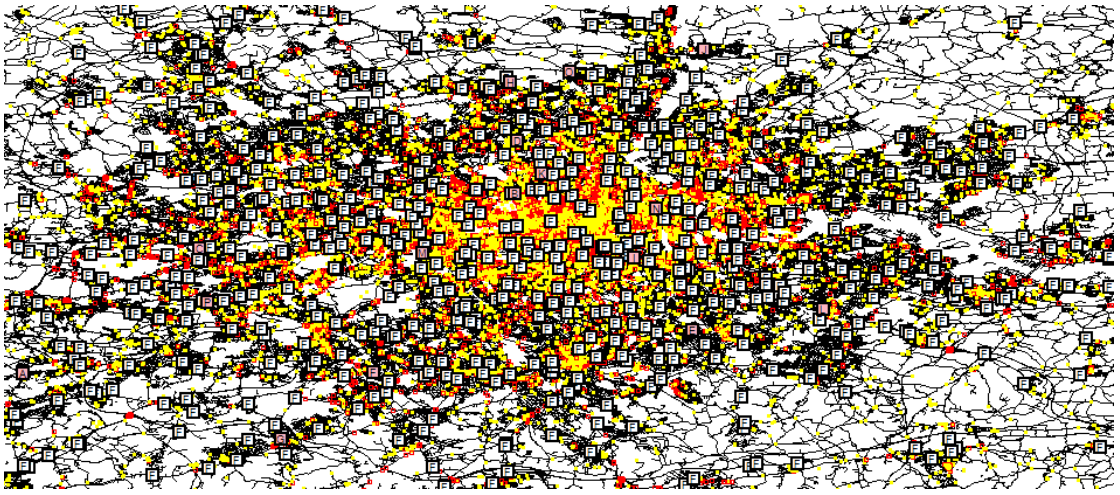


FIGURE 4.2: Fuel stations, businesses and road delays, modelled in Greater London.

data can fall out of date. OSM do not display speed limit data on its visual maps (only in the 'edit mode' and in the XML export). Such invisible discrepancies in the metadata are more likely to go unnoticed than topological errors.

Another major issue that must be overcome is the presence of inaccessible parts of the map, as without careful pruning, these will cause the routing algorithms to fail.

Figure 4.3 shows two examples from Jersey. In the first, a way, 'Mont Misere', has been labelled a `track`, causing a road leading off it to be inaccessible. If a job spawned here, no agent would be able to fulfil it. In the second, two one-way service roads off 'Route de la Marette' link the entrance and exit to a private car park. Often in OpenStreetMap, car parks are mapped out, however in this case, only an area has been formed and tagged `amenity=parking`. Were a job to spawn on the entrance to this car park, the agent would be unable to escape. Were a job to spawn on the exit, no agent would be able to reach it. The method used to identify these as disconnected graph components and prune them is in section 5.3.2.

4.2 HERE Traffic API

HERE is an online mapping service owned by Nokia. They provide a traffic API for developers, accessible over HTTP. One of their endpoints is traffic flow data. When provided a set of coordinates that define a bounding box, the API can return a list of main roads. Each main road has properties about its *current flow*: *free flow speed* (a motorist's average speed given ideal conditions), *jam factor*, current estimated *speed* capped or uncapped by the speed limit and the *confidence* to which it states these figures [37]. By default, HERE identifies sections of roadways using Traffic Message Channel

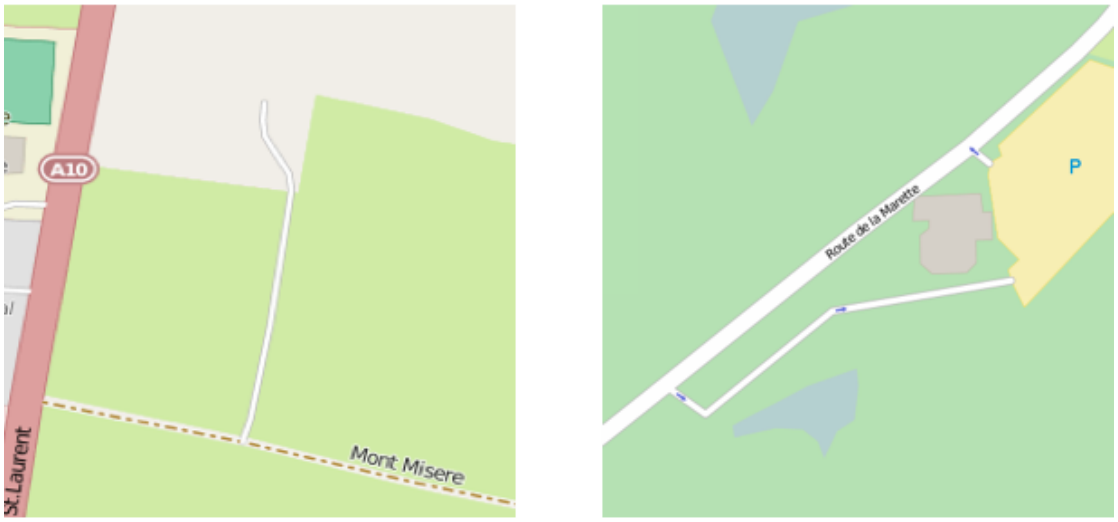


FIGURE 4.3: Examples of inaccessible, inescapable ways from OpenStreetMap.

point codes and lengths. A mapping from TMC codes to OpenStreetMap nodes does not exist in the UK [38]. However, HERE also provide an alternative representation in the form of unordered lists of coordinates that form the shape of the roadway segment. Using a script that ran every five minutes, 20 GB of data was scraped for the Isle of Wight, Las Vegas Valley and Greater London over the course of one typical week (Tuesday 14th April 2015, 00:00 – Monday 20th April 2015, 23:59 GMT).

4.2.1 Integration Challenge

It was hugely challenging to integrate this data with OpenStreetMap. The complete lack of common identifiers and the fact that roadway and OSM highway lengths can vary significantly meant that an approximate solution had to be developed using the lists of points. Some of the coordinates were observed to be inaccurate to such an extent that the corresponding ways could be ambiguously construed from the data. Hence, the solution had to be robust to noise. Realistic traffic data was of the utmost importance to the credibility of the simulation results, so a lot of effort was put into this integration.

In the simulator, the *Node* class was augmented with *SpeedAtTime* tables to store speeds. Each one of the 2016 slots corresponded to time intervals of the week, with Monday 00:00–00:05 being at index 0. For each of the 2016 XML dumps, a loop ran over each 'flow item' (a road segment). For each pair of coordinates used to draw the road shape, a mapping was found from the pair to the geographically nearest node that was part of the map's connected road network. This demanded a new, spatially indexed data structure to store nodes – see section 5.3.1.2. The flow item's estimated speed was placed into the tables of each of these nodes at the corresponding time slot. So that the

simulation time can start on a Monday at 00:00:00 in *any time zone*, the snapshots had to be enumerated with the help of an offset – 0 for UK maps, 1920 for Las Vegas.

The next step was to iterate over all ways in the map, inspecting their nodes. If at least 20% of its composite nodes had a full set of speeds, the mean average was taken at each time step. This data was then exported to the '.aa' files in the format $WayID : speed0, speed1, ...speed2015$. Upon restarting the program, this is loaded following the OSM map data. If it doesn't find a complete set of traffic data snapshots for any particular way, it will assume no traffic and use the roads' speed limits exclusively. This exporting step was not required, but considering the disk footprint is almost 100 times reduced, it is highly preferable.

4.2.2 Discussion

Despite the challenges faced, the final result appears to be a very accurate translation. On the three maps used, no falsely identified ways are apparent, nor are there any main roads missing traffic data. There are a few short gaps in long stretches of highway, caused by disjoint ways from the OpenStreetMap data. Not wanting to risk introducing false positive flow data, the parser does not attempt to correct these. The peaks caused by rush hour traffic appear to perfectly match up to time and day of week. Although certain main roads are almost always moving at below free-flow speed, others vary day by day. This gives more interesting consequences, however it may have been more appropriate to use a data source that provides smoother data that has been averaged over time.

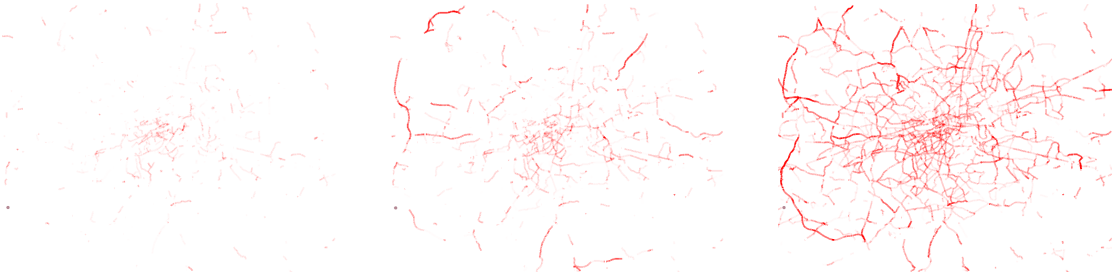


FIGURE 4.4: HERE Maps traffic data in Greater London, as parsed by the simulator. Traffic intensity is shown on a Monday at 00:30, 06:30 and 08:30. Darker shades of red indicate a high difference between the current speed and the fastest speed observed throughout the week.

Chapter 5

Implementation

5.1 Implementation Language

Despite advice from previous students and the desire to build upon the related works, it was decided very early on that the simulator would be developed from scratch. Simulation packages such as MASON, FAMOS and SWARM were investigated, however it was decided that for full control, clean design and performance, it would be better to develop a bespoke solution from the ground up. The lack of unnecessary overhead allows for massive environments and lengthy simulations. Further, implementing complex interactions between agents, such as the Contract Net Protocol, would have been unnecessarily challenging. The implementation runs as a 64-bit Common Language Runtime (CLR) binary on the Microsoft .NET 4.5 framework and is written in Visual Basic.NET. This decision was mainly grounded in the ability to quickly prototype GUI applications. .NET also offers a good balance of computational performance to development/debugging time. The program makes extensive use of .NET's Language-Integrated Query (LINQ) for powerful, yet succinct operations on lists and other collections. In the final release, no additional libraries or frameworks were used. It can be run on Windows or using Mono, Linux and Mac OS X.

5.2 Discrete Event Simulation

The actors within the simulation are the autonomous agents and the centralised component that generates and broadcasts jobs to the agents. An *AACourierSimulation* object holds the state of the simulation. This is a sub class of *AASimulation* and it has a sibling class *AAPlayground*, which is discussed in the appendix, section C. The class contains a list of *Agent*s and the centralised component is accessed via a module named

NoticeBoard. This was necessary, as with some routing strategies (namely, CNP5, as described in section 6.2.1.5) it is required that agents are able to interact directly with the centralised component, in addition to the other way around. The noticeboard contains a global *Timespan* object to represent the time that has elapsed. This begins at 00:00:00 and the simulation begins on a Monday, which is significant because job frequency and traffic varies by the time and day of the week. The simulation object provides a *Tick*() method, which in turn calls *NoticeBoard.Tick*() and then for each agent calls *Agent.Move*(). When the noticeboard is ticked, the current time is incremented by one second, which is the smallest unit of time represented in the simulation. Every tick, the noticeboard 'ticks' the dispatcher (see 5.5.1), which may or may not generate a new job to be immediately broadcast to the agents by the broadcaster (see 5.5.2). As would the central server of a real-world courier firm, the noticeboard also keeps records of past and current jobs.

Agents are dynamically added to the simulation by the user via the main GUI form. They, along with jobs that are generated, are assigned unique identifiers from a *UIDAssigner* module. This is immaterial for the simulation, but it does allow the user to observe and reason about particular jobs and agents.

5.2.1 Event Logging and Statistics Collection

Another module, named *SimulationState*, logs events and caches the current state of agents and their jobs, which can then be displayed in the GUI. This approach was favoured over direct access between the GUI code and the *AASimulation* object. As the GUI must be kept asynchronous with the simulation, race conditions can occur unless the simulation object or objects therein are locked, which would make the simulation code cluttered and may hurt performance. The *SimulationState* module acts as an intermediate data store. In addition to caching state variables, such as job counts, it provides methods to log events to a queue, which is dequeued to the GUI form. Events are one of fourteen neatly formatted strings that are built using the factory pattern in an *LogMessages* module. For example, *Agent has refuelled: {0} L at a cost of £{1}*. Events may have an associated agent ID, unless they originate from the *NoticeBoard*. Figure 5.1 shows an example of a simulation state.

A separate statistics module, named *StatisticsLogger*, is a wrapper for a large *DataTable* that logs current and cumulative aspects of the simulation, as it runs. In the loop that calls *AASimulation.Tick*(), every n ticks, a call to *StatisticsLogger.Log*(*AgentsList*) is made to log the simulation state. For testing, a value of $n = 20$ was used, because long simulations generate too much data to store and graph. The module also provides

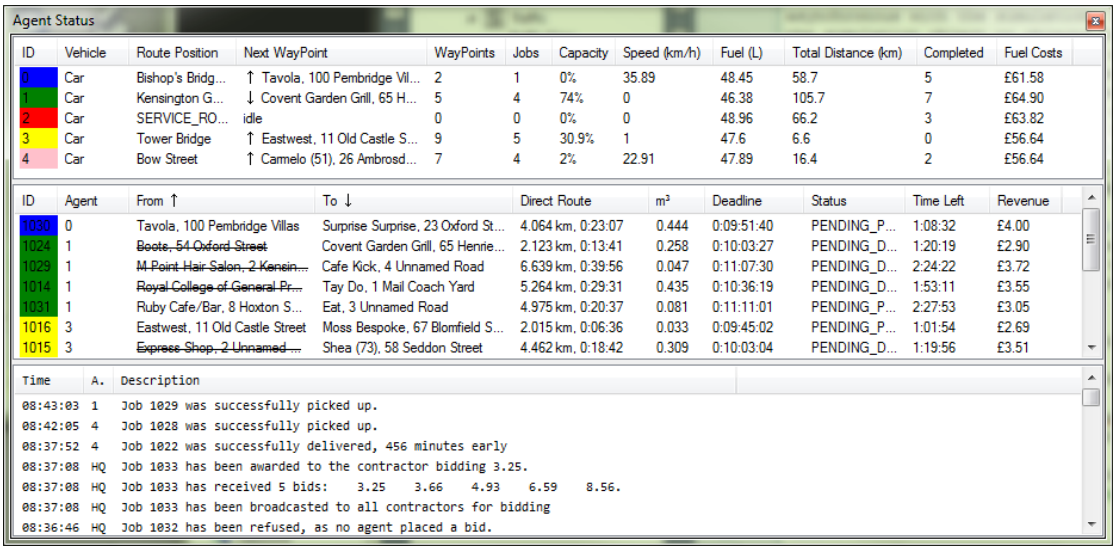


FIGURE 5.1: The simulator GUI, showing lists of agents, their jobs and simulation events.

functionality to export the data to XML, during or after the simulation, which was used in the evaluation to produce the final graphs.

The data table is used by a statistics GUI form. It is bound as a data source to a .NET *Chart* control and the values from the most current row are displayed in the sidebar. The chart plots simulation state variables as a line graph, with time on the x-axis. Tick marks are drawn on the x-axis every hour (3600 ticks), and grid lines every day. Selecting one or more column names from the sidebar plots them as series on the chart. A checkbox, if ticked, refreshes the chart periodically for a live display. A button allows the user to save the graph as a PNG file. This GUI form is shown in figure 5.2.

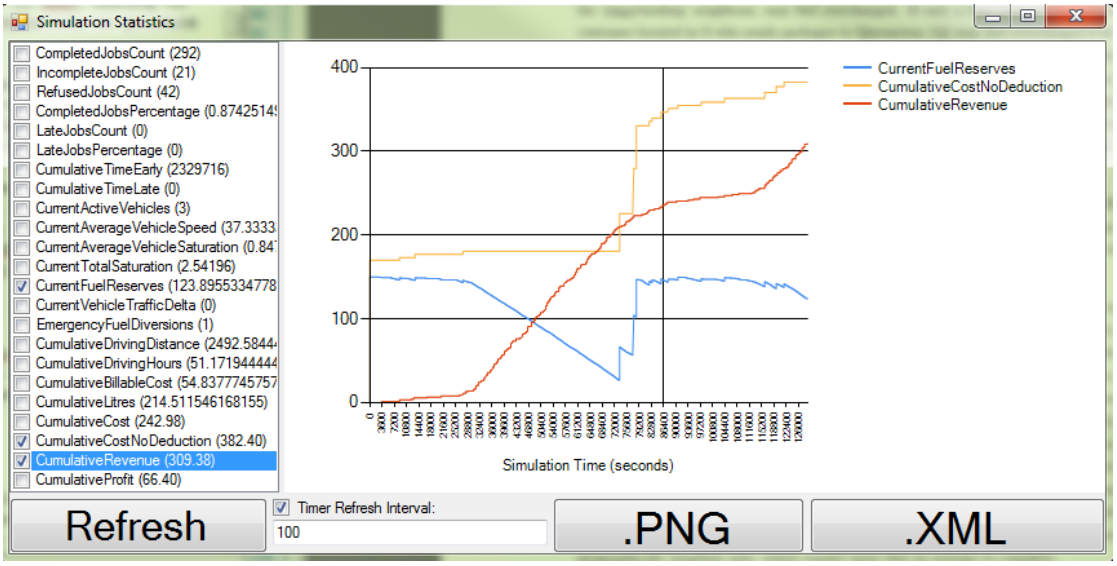


FIGURE 5.2: The statistics GUI, plotting over 1.5 days, the current fuel supply, with cost and revenues. Cost and fuel reserves spike upwards when agents refuel.

5.2.2 Simulation Parameters

All important parameters and constants reside in one particular module, *SimulationParameters*. A GUI form allows the variables to be changed before or during the simulation. As well as the speed of the simulation, the graphics refresh rate and the opacity of the traffic graphics overlay, these include:

- Dispatch rate coefficient, to control the frequency of jobs being spawned. See 5.5.1.
- Package size. Specifically the λ value used for the exponential distribution. See 5.5.1.
- Deadline excess. Specifically the θ value used for the gamma distribution. See 5.5.1.
- Probability of failed pick-ups, and also of deliveries. See 5.5.3.
- Delivery fee base and unit price. For example, £2 + £5.00 per marginal hour of driving.
- The value of ε to use in A* searches. See 5.3.3.1.

5.3 Geography

5.3.1 Data Structures and Objects

The data structures used to reason about the OpenStreetMap data were chosen for speed and space efficiency. All of these are held within a single *StreetMap* object, which is instantiated when parsing the OSM data and owned by the *AASimulation* object. The classes that make up the geography of the simulation are described below and shown pictorally in figure 5.3.

- *Node*. Represents an OSM node. *StreetMap* owns a list of all of these, as well as a list of depots, fuel points and businesses, which are all just *Node*s.
- *Way*. Represents an OSM way, but may also contain an array of traffic speeds from HERE maps. All ways contain an ordered array of *Node*s, which make up the shape of the highway. *StreetMap* owns a *SortedList* of *Way*, so that they can be quickly accessed by their OSM ID, when the parser is iterating through traffic data.

- *Bounds*. A rectangle with coordinates for the top-left and bottom-right points. This is parsed directly from the OSM XML file.
- *Hop*. An arc between two *IPoints*, with an associated *Way* and a precomputed distance.
- *HopPosition*. An exact position along a *Hop*, represented as a floating point percentage.
- *NodesAdjacencyList*. See 5.3.1.1.
- *NodesGrid*. See 5.3.1.2.

Node and *HopPosition* implement *IPoint*, which is used throughout the program in cases where only *GetLatitude* and *GetLongitude* are relevant methods. When a *HopPosition* is instantiated from a *Hop* and a percentage, the end points of the *Hop* need to be downcast. Its constructor ensures that there can never be a *HopPosition* defined as a point between two *HopPositions*, as this could start a long descending chain. Instead, it coalesces them based on the product of their percentages.

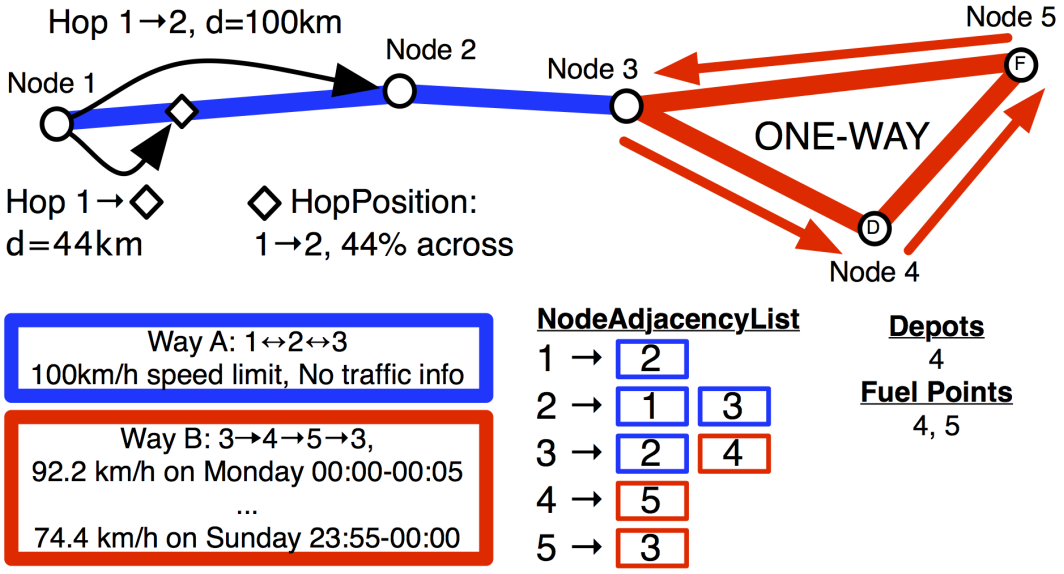


FIGURE 5.3: A visual example of a *StreetMap*, showing five *Nodes*, two *Ways* (one of which is directionally one-way and also has traffic information), *Hops* (one between a pair of *Nodes* and one between a *Node* and a *HopPosition*), a *HopPosition* between *Nodes* 1 and 2 and a *NodeAdjacencyList*.

5.3.1.1 Node Adjacency List

StreetMap owns an adjacency list for the nodes that make up the connected road network. It contains a *Dictionary* (hash table) of *NodesAdjacencyListRows*, that are

indexed by their node IDs. Each row has a list of $NodesAdjacencyListCell$ to represent outgoing arcs, each of which has a $Node$ (the outgoing node), the corresponding Way and the distance, calculated using the Haversine Formula (see 2.3.1). This is an ideal data structure for fast exploration of the road network, as all lookups occur in constant time. The structure is built up iteratively, using its $AddWay$ method which creates new cells for each arc of the Way and new rows for each newly encountered $Node$. Unless the Way's one-way flag is set, it is traversed in both directions. After processing each way, the non-trivial task of pruning any disconnected components remains. See 5.3.2.

5.3.1.2 Nodes Grid

Initialised using the $Bounds$ of the $StreetMap$, this object contains a 100 by 100 array of $Node$ lists. Each list is a quadrant of the map. $StreetMap$ owns a grid of the nodes of the connected road network. This constitutes a spatially indexed data structure. For tasks like mapping business $Node$s and HERE Maps coordinates to their nearest road network $Node$s, the speed improvement is around 1111x compared to iterating through each node in the adjacency list. Computing distances using the Haversine formula is expensive and large maps have millions of nodes, so this data structure was absolutely necessary in order to run long simulations. The $GetNearestNode(lat, lon, max\_radius)$ function determines the quadrant that the point lies in and starting with a radius of $r = 1$, checks that quadrant and all neighbouring quadrants for nodes. After each scan, if no nodes are found, r is incremented and the search space increases to $(2r + 1)^2$. An example of how an OSM map translates to a grid is shown in figure 5.4. The algorithm implemented in $GetNearestNode$ is shown diagrammatically in figure 5.5.

5.3.2 Map Pruning

The first round of pruning occurs in the parsing step (see 4.1.2). It consists of simply ignoring the ways that are not accessible highways – for example, footpaths, dirt tracks, private driveways and others. Having done this, the $NodeAdjacencyList$ that remains will represent a directed graph, which is most likely not strongly connected. Though a number of formally documented and verified algorithms exist to enumerate strongly connected components, of the three tried (Kosaraju's [15], Tarjan's [16] and path-based strong component algorithm [17]), none of them were functional for the input data at hand – massive graphs where 99% of the nodes are present in a single, strongly connected component, C. A simpler, more efficient algorithm was devised to find C and is most similar to Kosaraju's Algorithm. It is described in pseudocode below and demonstrated diagrammatically in Figure 5.6. It requires as input, a node *start*, which the user knows

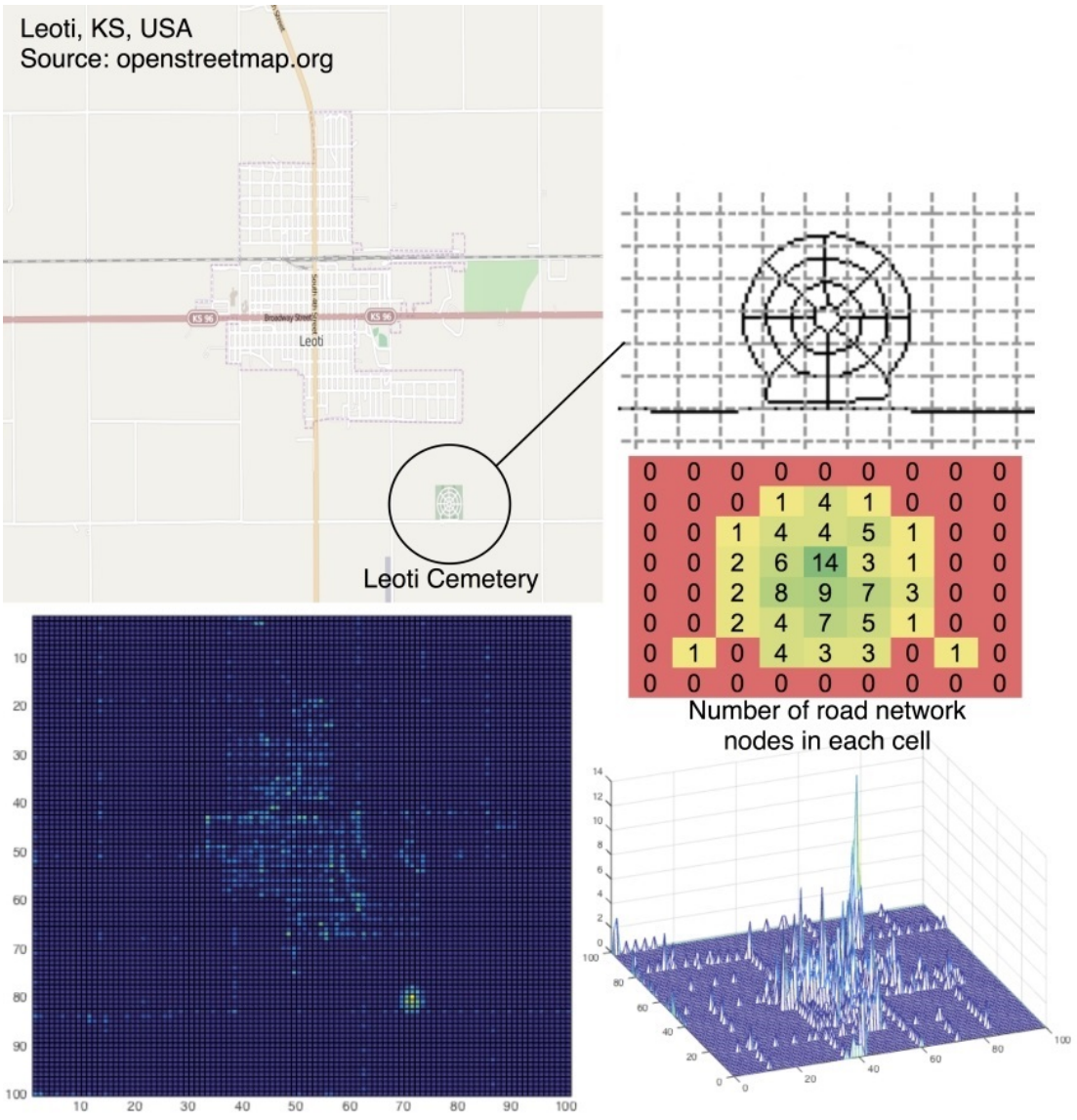


FIGURE 5.4: The 'nodes grid' for Leoti, KS, USA and its highly detailed map feature, Leoti Cemetery, therein.

is present on C. The parser uses the first, user-specified *depot node*, or if one is not available, the node that is geographically closest to the centre point of the map. In simplest terms, the algorithm performs a non-recursive depth-first search from *start*, which forms a tree of nodes that are labelled as explored. These nodes are *accessible*, but not necessarily inescapable. At this point, only *start* is *connected*. Iteratively, discovered nodes call a helper function to try and run a depth-first search to any other connected node. If this succeeds, the node and all nodes of that path are labelled as connected. All but the connected nodes are pruned from the graph.

On an AMD Phenom II X4 955, the time needed to prune 13346 nodes from Greater London's total of 706117 ranged from 4.2–4.6 seconds. The final loop in lines 15–18 takes up most of the computational time. Replacing it with a *Parallel.ForEach* loop

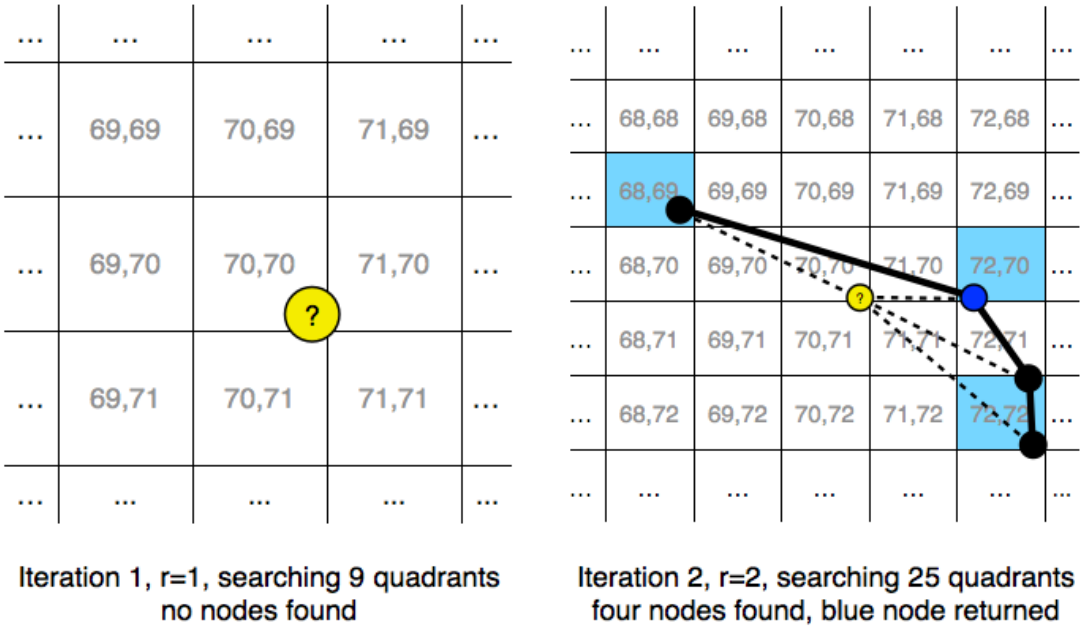


FIGURE 5.5: An example of finding the nearest road network node to a point in a sparse area of a 'nodes grid'.

and locking the *HashSet* objects widened the range to 2.0–6.4 seconds and cluttered the code, so the single-threaded version was used.

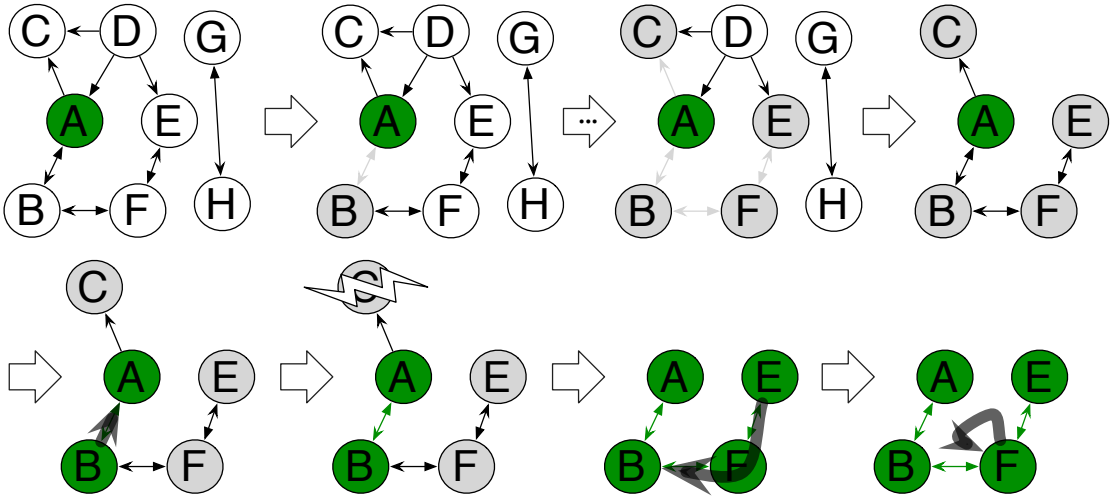


FIGURE 5.6: Algorithm to prune components that are not strongly connected to node A.

5.3.3 Route Finding

A *Route* object is simply an ordered list of Hops, wherein all intermediate points are *Nodes*, apart from the start and/or end position, which may be a *HopPosition*. A

Algorithm 2 Strongly Connected Components Algorithm

```
1:  procedure REMOVE-DISCONNECTED-COMPONENTS(GRAPH, START)
2:      explored = {start}
3:      dfsStack = {start}
4:      while dfsStack not empty do                    ▷ non-recursive DFS from start
5:          n = dfsStack.peek
6:          for all neighbour to n do
7:              if neighbour ∉ dfsStack and neighbour ∉ explored then
8:                  push neighbour to dfsStack
9:                  break while
10:         move n from dfsStack to explored
11:     for all n ∉ explored do                        ▷ prune unreachable nodes
12:         prune n from graph
13:     verified = {start}
14:     unverified = explored \ verified                          ▷ set difference
15:     for all n ∈ unverified do                      ▷ prune inescapable nodes
16:         if n ∉ verified then          ▷ verified may be modified by this function
17:             if not VERIFY-CONNECTED-NODES(n, verified, graph) then
18:                 prune n from graph
19: function VERIFY-CONNECTED-NODES(START, CONNECTED, GRAPH)
20:     explored = {start}
21:     dfsStack = {start}
22:     while dfsStack not empty do          ▷ DFS to any node in the connected set
23:         n = dfsStack.peek
24:         if current ∈ connected then
25:             add all dfsStack to connected
26:             return true
27:         for all neighbour to n do
28:             if neighbour ∉ dfsStack and neighbour ∉ explored then
29:                 push neighbour to dfsStack
30:                 break while
31:         move n from dfsStack to explored
32:     return false                    ▷ start cannot be verified, as no path exists
```

Route provides methods that sum the costs of each hop to provide a total cost. These are stored as fields, but evaluated lazily, to avoid having a heavyweight constructor.

Depending on how the agent is configured, the cost it is interested in minimising is either distance, driving time or fuel usage. The route finding class, *AStarSearch*, takes a *RouteFindingMinimiser* – an enum with values `DISTANCE`, `TIME_NO_TRAFFIC`, `TIME_WITH_TRAFFIC`, `FUEL_NO_TRAFFIC` and `FUEL_WITH_TRAFFIC`. Optimising for fuel efficiency is likely to yield similar results to time, however it will not prefer to use highways that allow driving at over 86 km/h (section 5.6.3.1 explains why). By default, agents attempt to minimise time with traffic.

The implementation of the A* algorithm as described formally in section 2.3.3 is augmented slightly to account for *HopPositions* in place of source and destination points. It begins by checking for edge cases: where the two points reside on the same *Hop* or are equal to each other, in which case the solution is trivial. If not, it runs the algorithm as specified in the background, using a priority queue of *AStarNode* objects in place of *openset* and *predecessors*. An *AStarNode* contains a reference to its predecessor/parent node, or null if it is the source node. Upon completion, the successful *AStarNode* is unravelled all the way to the root to create and return a *Route*. Expansion of the lowest cost node in the priority queue is done by enumerating the *NodeAdjacencyListRow*. If the first expansion is a *HopPosition* $A \overset{x\%}{\Rightarrow} B$, it adds B to the queue and also A if its *Way* is not one-way. If the A* search's destination is a *HopPosition* like denoted above, upon exploring A (or B if the way is not one-way), it will add an *AStarNode* with a heuristic cost of 0. In both cases, the costs are calculated in the same way as if the source and destination HopPosition was a node itself. The adaptation is quite intricate, but it is necessary to allow routing between positions that lie on arcs, not just nodes. The costs are all calculated using the great circle distance, which is found using the Haversine formula, as described in section 2.3.1. Where $d = distance(n, dest)$, the computation depends on the minimiser:

Minimiser	**Total cost, g(n)**	**Heuristic, h(n)**
Distance	$\sum_{h \in hops} distance(h)$	d
Time, no traffic	$\sum_{h \in hops} distance(h)/speed\_limit(h)$	d/max_speed
Time with traffic	$\sum_{h \in hops} distance(h)/speed\_at\_time(h, t_{h-1})$ where $t_h = \sum_{i=0..h-1} distance(i)/speed\_at\_time(i, t_i)$	d/max_speed
Fuel, no traffic	$\sum_{h \in hops} fuel\_usage(distance(h), min(speed\_limit(h), 86))$	$fuel\_usage(d, 86)$
Fuel with traffic	$\sum_{h \in hops} fuel\_usage(distance(h), min(speed\_at\_time(h, t_{h-1}), 86))$ where $t_h = \sum_{i=0..h-1} distance(i)/min(speed\_at\_time(i, t_i), 86)$	$fuel\_usage(d, 86)$

max_speed is a constant set to 112 km/h – the national speed limit in the UK and USA. For minimising time and fuel and accounting for traffic, it is important to keep track of the total time elapsed as the tree of nodes is explored. This makes $g(n)$ a recursive function with a base case cost of 0. In practice, to save computation time, the parent node n' cost $g(n')$ and its current time $t_{n'}$ is stored in the parent *AStarNode* object. Expanding the path from n' to n involves incrementing the cost and if applicable, the current time.

5.3.3.1 Bounded Relaxation

As discussed in the background (2.3.3.1), it is possible to speed up an A* search if one makes the heuristic non-admissible. As evaluated in some detail in section 7.1.1, computation time is substantially reduced and the optimality trade-off is justified. The heuristic cost is multiplied by $(1 + \varepsilon)$, where $\varepsilon = 0$ by default, but can be increased by the user, pre- or mid-simulation.

An admissible heuristic is one that never overestimates the cost. However, the estimate may be so much lower than the true cost that the search is almost as inefficient as an uninformed breadth-first search. For minimising time in particular, the admissible heuristic suggests that any node may be the start of a motorway going straight to the destination. In Central London, where the average traffic speed is 14.5 km/h [39], this leads to A* searches that evaluate nearly every node. Setting a value of $\varepsilon = 4$, is equivalent to adjusting the 'speed limit' from 112 km/h to 22.4 km/h. This reduces the search space and is unlikely to produce a different solution.

5.3.3.2 Route Caching

As routes between waypoints need to be evaluated many times in successive runs of the planning algorithm (see 6.1), the timing of the simulation can be improved by caching precomputed routes that are known to be optimal. If any part of the simulation needs a route that it suspects may be re-evaluated by itself or by another agent, instead of instantiating a new *AStarSearch*, it uses the method $GetRoute(src, dest, time)$ in the *RouteCache* module. This essentially performs a constant time lookup in a table with composite key $\langle src, dest \rangle$. It then compares the absolute difference in the starting time of the route with any routes available – typically three or fewer. The optimal route will vary by time of day, but using the assumption that the optimal route will be more or less the same within thirty minutes, *RouteCache* may return a cached *Route* that was scheduled to begin up to ±30 minutes away.

If a recent route is not present in the cache, a new one is computed and indexed. Typically, the *AStarSearch* class alone accounts for 85% of the simulation's CPU time, even after extensive profiling and micro-optimisation. Given that in the simulation, the cache hit rate is typically around 80% [1], this design decision more than triples the simulation's performance. The nested lookup table is implemented using a $Dictionary\langle Dictionary\langle IPoint, List\langle Route\rangle\rangle\rangle$. A standard *List* was appropriate to store routes between the same points, as *Route*s have a *StartingTime* field and this can be accessed using a simple LINQ query.

Every twenty-four hours of simulation time, the outer *Dictionary* is reinstantiated and its memory footprint reclaimed by the garbage collector. Without this, the simulator would eventually run out of memory.

[1] If the agents' *RouteFindingMinimiser* is not one that accounts for traffic, the time check can be omitted. An 80–90% hit rate is achievable in this case.

5.3.3.3 Other Attempts

Alternative optimal and near-optimal algorithms were implemented, however they are no longer used as the final A* implementation described above was the most time efficient and provides full control with regards to the speed/optimality trade-off:

- In the early stages of the project, a simple **breadth-first-search** was used. Long routes would cause $OutOfMemoryException$s, due to the fact that the search is *uninformed*.
- When experimenting with large maps with long, windy roads, **a modified A* search that iteratively expanded nodes until meeting an intersection** was written. This was still optimal and performance was marginally improved in rural maps (such as Alaska, USA), but the increased overhead made it slower for dense cities.[2]
- **Bidirectional A* search** was attempted, but it was unclear when to stop the search. Stopping when a common node was reached gave solutions that were far from optimal.

5.4 Traffic

A city road network simulation needs to simulate vehicle and pedestrian traffic to provide meaningful results. With large maps, it is infeasible to simulate tens of millions of non-courier agents. Instead, the simulation uses approximated traffic flow speeds for major roads and probabilistic activation of crossings and traffic lights, which block parts of the road for a number of seconds.

5.4.1 Traffic Flow

As discussed in section 4.2.1, the main challenge in implementing traffic was to union the two sources of data, OpenStreetMap and HERE. As the simulation runs, the agents have $RoutePosition$s, which consist of an ordered list of Hops. Each Hop has an associated Way. When the agent moves, the Way's $GetSpeedAtTime(t)$ method returns the speed in kilometres per hour at which the agent is allowed to move along that Hop. If its current Way has traffic data, this is likely to rise and fall throughout the week. Otherwise,

[2] This is similar to the 'reach algorithm', which was used by Microsoft Research to improve route finding along freeways in the USA [40].

t is ignored and the way's speed limit is returned – either the one specified in the OpenStreetMap data, or if unspecified, the national speed limit for that type of road.

The traffic traces were sampled every five minutes for the course of a week, so if the data for a way exists, this function first transforms t into a number corresponding to the sample index, $i \in [0, 2016)$. An internal array of floating point numbers holds the data. Some micro-optimisation is performed in the Way class because the $GetSpeedAtTime(t)$ method is called very often by the route finder. It is also called in the *Route* class method $GetEstimatedHours(StartTime)$, which is used by the routing strategies to determine if routes will be completed on time. As later detailed in 5.7.1.3, a separate function is used to visualise congestion. $GetSpeedDifferenceAtTime(t)$ returns the difference between the highest speed of the week and the current speed: $max(SpeedAtTime) - GetSpeedAtTime(t)$.

5.4.2 Road Delays

As discussed in section 4.1.3, nodes that represent level crossings, traffic lights and pedestrian crossings are marked as potential delays. The main challenge was to devise a probabilistic activation scheme, which determines the time points at which they will require the vehicle to halt. The frequency and lengths of such delays should be a reasonably accurate approximation of the real world.

For traffic lights and level crossings, the constants have been defined:

Road Delay	**Frequency (s)**	**Delay Length (s)**	**Time Range**
Traffic Lights	60	30	all day
Level Crossing	600	120	all day, excluding 01:00–04:00

As an example, this means that an arbitrary level crossing node will block all agent traffic for fixed two minute intervals between 5 a.m. and 1 a.m. Every two minute road block is followed by an eight minute free flow at the speed dictated by the way's traffic flow data or in its absence, the speed limit. Traffic lights oscillate between red and green, toggling every thirty seconds, all day. Unfortunately, more detailed schedules are not available to the public. Given the OpenStreetMap node n, the type of road delay d (e.g. level crossing) and the current simulation time in seconds t, the following function is used to determine whether the node is delayed/blocked:

$$IsBlocked(n, d, t) = \begin{cases} 1, & \text{if } t \in d.range \wedge ((t + n.id) \bmod d.freq) < d.length. \\ 0, & \text{otherwise.} \end{cases}$$

The function ensures that this oscillation sequence is offset at different, well-distributed times for each node, yet the result will be the same for each agent. Another function is defined to determine the average, or expected delay at a certain time. This is used by the A* route finder and in the *Route* class method $GetEstimatedHours(StartTime)$, which is used by the routing strategies to determine if routes will be completed on time. The function is the product of the chance of being delayed $\frac{d.length}{d.freq}$ and the average length of being delayed $\frac{d.length}{2}$:

$$WaitTime(n, d, t) = \begin{cases} 0, & \text{if } t \notin d.range. \\ \dfrac{d.length^2}{2 \times d.range} & \text{otherwise.} \end{cases}$$

Pedestrian crossings differ in that their usage varies by day. There don't seem to be hour-by-hour statistics for pedestrian traffic in urban areas, however there is for road traffic. A yearly statistical data set from the UK Department of Transport [41] gives a distribution of traffic flow by hour. The simulation assumes road traffic is proportional to pedestrian traffic. In addition, all other nodes that make up the connected road network have the propensity to delay a vehicle and the simulation models frequent minor delays and infrequent major delays. In the real world, vehicles may need to stop momentarily, such as for jaywalkers (minor) or double parked cars (major). Such obstacles would normally scale with traffic density.

Letting f be the traffic flow value at time t according to the table in appendix section D.1 and $max = 212$ (the peak value in that table), the constants have been defined:

Road Delay	**Period (s)**	**Delay Length (s)**	**Probability p**
Traffic Light Crossing	45	15	$1.0 \times \frac{f}{max}$
Zebra Crossing	30	10	$1.0 \times \frac{f}{max}$
Unexpected (minor)	2	2	$0.00222 \times \frac{f}{max}$
Unexpected (major)	30	30	$0.00417 \times \frac{f}{max}$

p is *the probability that a road block of a given length occurs within the current period.* At peak time, this is 100% for pedestrian crossings and as low as 3.7% at 2–3 a.m. on a Monday. For the unexpected minor and major delays, this equates to 4 and 0.5 times per hour respectively. This may seem low for the inner city, but recall this is per node and there are over 30,000 in London's zone 1 alone. For pedestrian crossings, the table ensures that vehicles are not delayed for longer than the expecting delay length. Traffic signals ensure fairness. This is not the case with the unexpected delays as the period and lengths are equal – it's just really unlikely.

A uniformly distributed random variable determines whether a delay is to occur in a given period. It is a *Bernoulli* distribution with probability parameter p. To ensure

all agents in the simulation get the same value in each period, the random number generator uses the same seed: $(t \text{ div } d.period) + n.id$. If a 1 is sampled, a second, deterministic, uniformly distributed random variable is sampled to determine the discrete offset o at which the delay occurs within the period. The $IsBlocked(n, d, t)$ function, as mentioned before, returns true iff a delay occurs in the given period and $o \leq t \bmod d.period \leq o + d.length$. For nodes that exhibit only unexpected delays, the agents query $IsBlocked(n, u\_minor, t) \vee IsBlocked(n, u\_major, t)$. We can derive the average wait time in a similar way as before:

$$WaitTime(n, d, t) = d.probability(t) \times \frac{d.length^2}{2 \times d.period}$$

5.5 Courier Jobs

A *CourierJob* object is comprised of a pick-up position, a drop-off position, a volumetric size and a deadline. In the real world, they would be created when a customer places an order, however in the simulator, they are spawned by a centralised 'dispatcher' using a seeded random number generator, such that the simulation results are reproducible. A centralised 'broadcaster' receives these and, depending on the planning and allocation strategy, transmits the job specifics to one or all agents. A *CourierJob* has a status field, which is transformed by its methods. Figure 5.7 shows the state transition of a job, with each box representing a member of the *JobStatus* enumeration. The simulation has been designed per the specification in chapter 3 and flow chart 3.2.

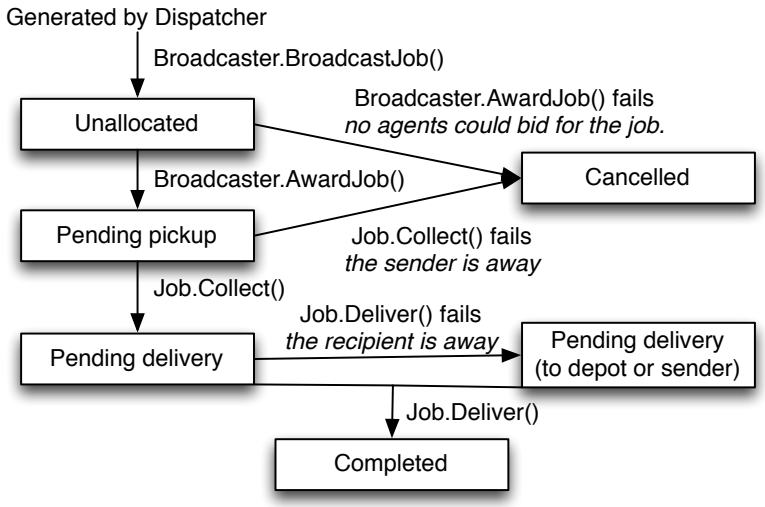


FIGURE 5.7: State transition diagram of the *JobStatus* field of a *CourierJob*.

5.5.1 The Dispatcher

The Dispatcher is a centralised singular component of the simulation. All dispatchers written implement *IDispatcher*. They provide a single *Tick*() method, which is to be called every chronon. *CityDispatcher* is predominantly used in the evaluation and is described below. Within the *Tick*() method, a sample from a $D \sim Bernoulli(p)$ distribution determines whether a job should be spawned. The p parameter is chosen based on the hour of the day and whether it is a weekday. This probability is then multiplied by a dispatch rate coefficient, which can be set by the user before or during the simulation. Set to 1.0, the probability of dispatch is shown in figure 5.8. The figure also shows the probability distribution of different types of job at different times: B2B, B2C, C2B and C2C (where B is short for business and C is short for consumer). This value is selected using another random number generator. These parameters can always be altered, however sensible values have been chosen to reflect expected business and consumer behaviour.

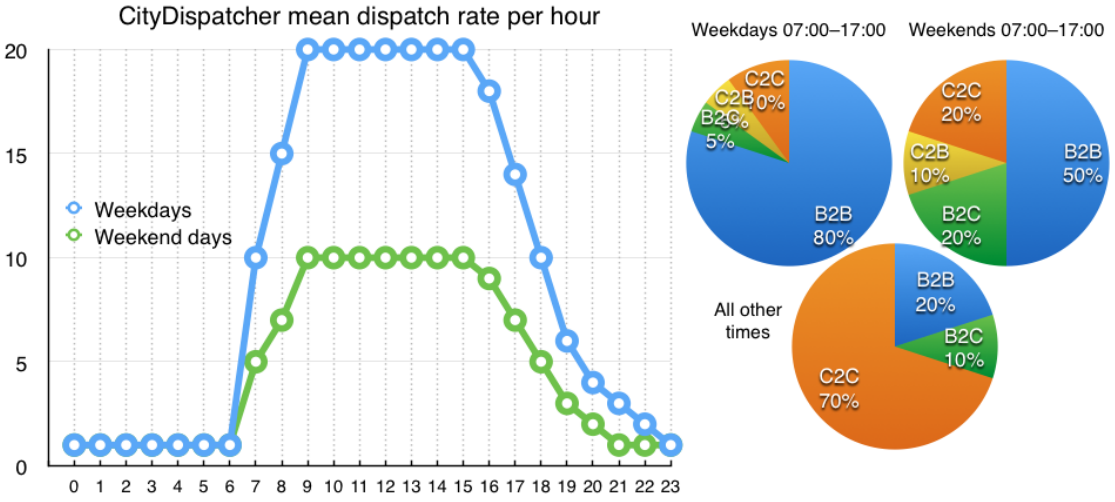


FIGURE 5.8: The distribution of jobs dispatched per hour and the types of jobs they are, as used in *CityDispatcher*.

Depending on the job type, an RNG is used to pick a random business node (see 4.1.4 for a list of those included) and/or a random position on the road network to represent the consumer. The deadline is generated by first performing an A* search that minimises time with traffic and then calculating the time it would take to drive directly from sender to receiver. To this, a number of hours is added, sampled from a Gamma distribution: $X \sim \Gamma(k, \theta)$. The decision to use this distribution is based solely on its shape, which it is assumed most accurately resembles customers' time demands. For the evaluation, the shape parameter was set, $k = 2$ and the scale parameter, $\theta = 1$. This makes the mean excess time $k\theta = 2$ hours and the mode, $(k-1)\theta = 1$ hour. If a certain flag is set, all B2B and C2B deliveries will be assigned a deadline that is uniformly distributed around the

end of the that business day. If the direct route could possibly be completed before 17:15, the deadline is set as 17:15 + $minutes(Uniform(0, 30))$. Otherwise, it is calculated as before. This was implemented to investigate the impacts of scattered versus concentrated deadlines and because many real-world services like UPS promise EOB delivery instead of arbitrary deadlines. Finally, the volumetric size of the package, simply specified in cubic metres, is sampled from an exponential distribution: $X \sim Exp(\lambda)$. The rate parameter is set to $\lambda = 3$ and the result is then normalised to lie within a valid range: $0.0002 \leq x \leq 0.999$. These distributions are shown in figure 5.9.

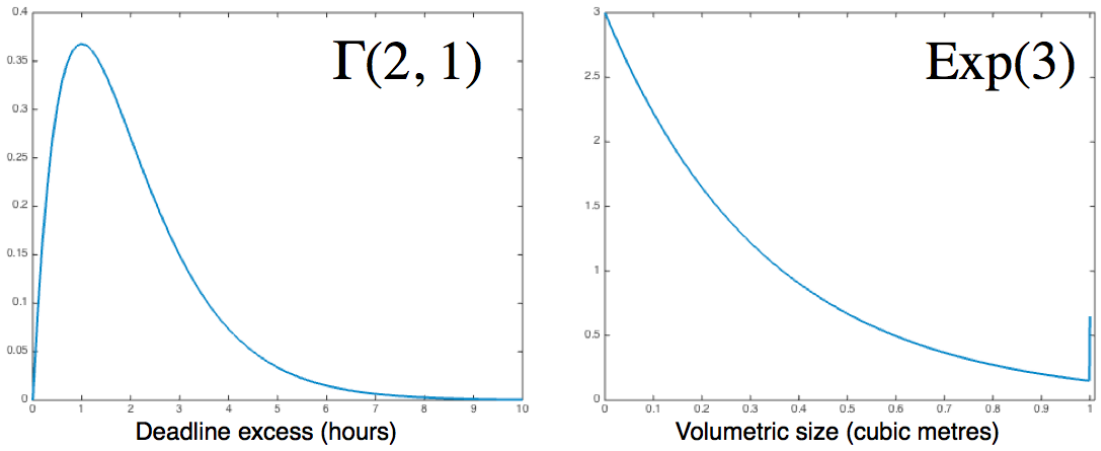


FIGURE 5.9: The probability density functions for *CourierJob* properties.

5.5.1.1 Other Dispatchers

- *SingleBusinessDispatcher*. For the purposes of evaluation, an additional implementation of *IDispatcher* was written wherein one central depot acts as the hub of all deliveries. This could model an online shop that offers same-day delivery to local customers. The probability of dispatch scales with time in the same way as before (figure 5.8), however a $Bernoulli(p)$ distribution determines whether a job is B2C (i.e. orders) or C2B (i.e. customer returns). Based on this, either the pick-up or drop-off position is the first depot in the *StreetMap*'s list of depots (the business) and the other is a randomly chosen point (the customer) on the map.

- *HubAndSpokeDispatcher*. An extension to the SBD, this dispatcher follows the hub and spoke topology. Most large distributors use this model to reduce 'last mile' costs [42], which according to parcel delivery company, Parcel2Go, comprises 28% of the total cost [43]. Many delivery companies offer reduced parcel delivery costs if pick-up and/or collection takes place at 'parcel shops'. Online retailer, Amazon, have installed lockers in select cities, which some customers use for receiving and returning items. This dispatcher models the hub and spoke topology on a small

scale. Because the locations of these 'spokes' are not publicly available in machine-readable form, it uses fuel points instead. Incidentally, many fuel stations actually do serve as parcel shops in the UK. The dispatcher works identically to the SBD, with the key distinction being the randomly chosen point becomes a randomly chosen fuel point (spoke). Unless there are hundreds of fuel points in a single map, there are likely to be great efficiency gains because many *CourierJob*s will overlap and marginal costs will be extremely minimal.

- *RuralDispatcher*. A variant of *CityDispatcher* was created and is chosen by default if the map has fewer than 50 business nodes. The dispatch frequency is the same as before (figure 5.8), but all jobs are C2C. This allows for proper testing of very small or rural areas.
- *DepotDispatcher*. See section 5.5.3.1.

5.5.2 The Broadcaster

Two broadcasters that implement *IBroadcaster* are presented. They both rely on a list of agents (contractors). *ContractNetBroadcaster* auctions jobs to all contractors and in line with the contract net protocol (as described in section 2.1.1) will award the job to the winning bidder – the agent whose extra cost is lowest. This simple process is shown in figure 5.10. Computing how much to bid is a computationally expensive operation. Since the planning problem is independent for each agent, it is performed in parallel. If no bids are received, the job is cancelled. Otherwise, the winning bid determines the fee, as per the design specification (refer back to figure 3.1):

$$Fee = BasePrice + EstExtraDrivingCost \times UnitPrice$$

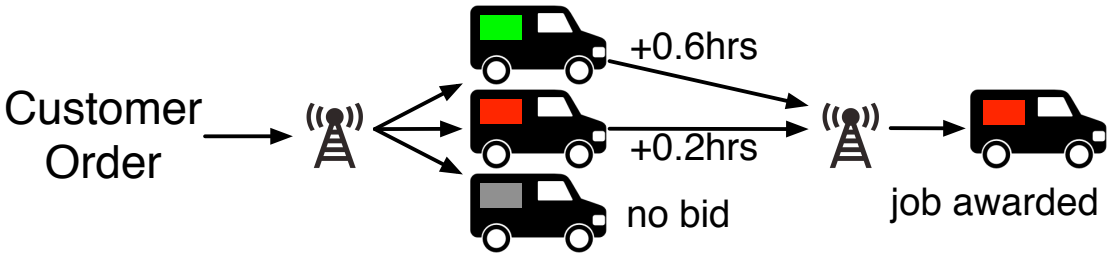


FIGURE 5.10: An example of the Contract Net Protocol, where agents report cost as additional driving time.

RoundRobinBroadcaster is fairer and less computationally demanding, but most likely less optimal. Instead of broadcasting jobs to all agents, it simply transmits it to one. In

a circular sequential fashion, it allocates the job to an agent from its list of contractors. This ensures that jobs are distributed fairly, although it does not discriminate on the difficulty of the job. The fee is calculated as above, but using the direct route as the estimated extra cost. Agents are unable to refuse to be allocated a job, so a prerequisite of using this strategy is that the capacity of the smallest vehicle is greater than the maximum size of a package. More information on the implementation of the Contract Net Protocol and the Round Robin allocation scheme is available in section 6.2.

5.5.3 Modelling Failure

A very common occurrence in real-world couriering is customers being absent when the vehicle arrives. As the vehicles are autonomous, it is not possible to offload an undelivered package without a trusted human. Hence, depending on a user-specified parameter, failed deliveries can either be routed to the nearest depot/collection point or to the pick-up position, where delivery is guaranteed to take place[3]. A non-technical overview can be seen at the tail end of the flow chart (figure 3.2) in the design specification chapter.

Failed pick-ups simply cause the job to become cancelled and a partial refund to be applied. This refund is equal to the cost saving after rerunning the planning algorithm with the delivery waypoint removed. A sample from a $Bernoulli(p)$ distribution determines waypoint success or failure. As the sender pays for the order, it is expected that failure will be more common at the receiving end. Hence, by default, for pick-ups, $p = 0.99$ and for drop-offs, $p = 0.9$. These can be changed by the user pre- or mid-simulation for fault tolerance experiments.

5.5.3.1 Rebooking Failed Deliveries

If the simulation is configured to reroute failed deliveries to the nearest depot, the receiving customer may rebook the package to be sent to them from their local depot. When a job is delivered to a depot, a *DepotDispatcher* object is notified of the job. It takes a sample from an exponential distribution, $X \sim Exp(\frac{1}{48})$. If $x > 48ln2$ (the median), the job is discarded (i.e. the customer picks it up themselves or allows for it to be disposed of). If $x \leq 48$, the job is scheduled to be reordered in x hours with the same delivery position and size, but a new deadline. Using these default parameters, this means that the probability of rebooking a failed delivery is 0.5. The scheduling is done using an internal priority queue. In $NoticeBoard.Tick()$, the *CityDispatcher* or

[3] 'Collect at depot' and 'return to sender' are common options with established couriers, who will often call the customer so that they can make the decision.

a variant thereof is only 'ticked' if the *DepotDispatcher* has nothing to dispatch at that second.

5.6 Autonomous Agents

In this simulation, the autonomous agents are self-driving vehicles. Three such vehicles have been modelled and are shown in table 5.1.

Mid-size hatchback car	Small commercial van	7.5 tonne truck
		
$1m^3$ capacity 50 L petrol tank 34.17 mpg peak	$2m^3$ capacity 80 L diesel tank 31.43 mpg peak	$8m^3$ capacity 92 L diesel tank 18.79 mpg peak

TABLE 5.1: Courier vehicles modelled in the simulation.

5.6.1 Movement and Positioning

An agent has a *CourierPlan*, which contains an ordered list of waypoints and *Route*s thereto. It also has a *RoutePosition*, which holds the exact position of the agent along a certain *Route*. As a *Route* is a list of *Hop*s, the exact position can be represented as the index of the current hop, plus the distance travelled across that hop, as a percentage.

RoutePosition.*Move*() is called by *Agent*.*Move*() and it adjusts these variables to represent one second of driving at the current *Hop*'s *Way* speed. If the agent is minimising fuel consumption, the speed is capped. The method calculates the distance it should travel and iterates through the hop sequence until this distance has been reached. Every time the hop index is incremented, a call to *IsBlocked* is made to see if a road delay is active, in which case the iteration stops (see 5.4.2). The *Move* method is ineffective for as long as the the road delay persists.

5.6.2 Job Fulfilment

When a route is completed, this often means that a waypoint has been reached, in which case the relevant *CourierJob*'s status is transitioned (e.g. from 'pending pickup'

to 'pending delivery'). This was summarised earlier in figure 5.7. The *RoutePosition* is recreated with the next waypoint's *Route* in the *CourierPlan*, or a null route if there are none left. To model the time it takes for the customer to load or unload the van, a *Delayer* object is used to block calls to $RoutePosition.Move()$. This is a timer that is set to the maximum wait time of 120 seconds if the customer is away or a uniformly distributed random variable between 20 and 120 seconds. Other actions may be performed, depending on the routing strategy (described in section 6.2).

CourierPlan provides a number of methods for modifying and querying the plan, including extracting fulfilled and cancelled waypoints, assessing whether any jobs will arrive late (so that action can be taken by the routing strategy) and replanning in response to traffic changes.

5.6.3 Fuel

The vehicles have fuel levels, a floating point number representing the litres remaining, which deplete based on the distance travelled in $RoutePosition.Move()$. For example, a *car* moving at 80 km/h uses 1.5 millilitres of petrol every one second tick. At times when the vehicles are stationary, the fuel level does not deplete at all. As the simulation starts with the agents stationed at a depot, they begin with a full tank of fuel and the cost of this is accounted for. At the end of simulations, the value of the remaining fuel reserves is subtracted from the total costs. Average UK fuel prices from 9th April 2015 were used: £1.1327/L for unleaded regular petrol and £1.1882/L for diesel [44].

The *StreetMap* object has a list of nodes that represent fuel points (see 4.1.5 for how this was built). Agents route to and refuel themselves at these points before their fuel level reaches zero. The same as with visiting pick-up and delivery waypoints, a *Delayer* object is set upon arrival. This models the time it takes to refuel. The agent is ordered to remain stationary for one minute.

5.6.3.1 Fuel Economy

For increased realism and accuracy to a real-world operation, the amount of fuel used does not scale linearly with speed x. Based on data from a number of sources [45][46][47][48], a polynomial function was approximated using $k = 1, 0.92, 0.55$ for car, van and truck respectively:

$$MilesPerGallon = k \times (-0.0119x^2 + 1.2754x)$$

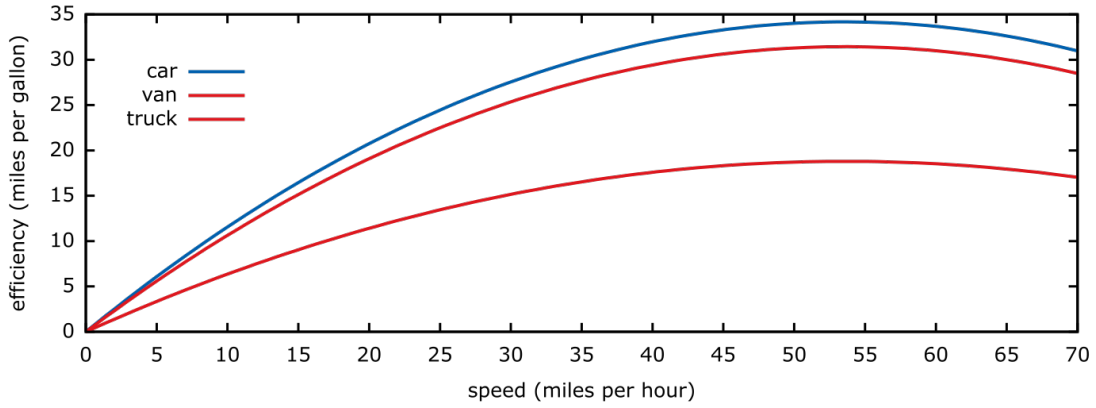


FIGURE 5.11: Fuel economy model used in simulation.

The peak fuel efficiency, 34.17 mpg is reached at 53.5882 mph $\approx$ 86.242 kph. As such, if the measured speed of a road is above this value and the vehicle is aiming to minimise fuel usage, it will restrain its speed to this value. The simulation could be made more accurate by modelling specific models of cars that have been rigorously tested. In addition, it could account for the type of road surface, weather and weight of cargo and fuel.

5.6.3.2 Emergency Refuelling

Normally, agents will refuel once every twenty-four hours, when they have no more jobs to fulfil (see 6.3). However, this cannot be relied upon and in some conditions the vehicle would run out of fuel whilst fulfilling jobs. Therefore, the agents continuously monitor their fuel supply and if their fuel tank is below a certain percentage threshold (5%), a 'fuel diversion' is set. This suspends all jobs in its *CourierPlan* and routes the agent to the geographically nearest fuel point. Upon arrival it refuels and resumes its *CourierPlan*. There is a very slim chance that the vehicle will run out of fuel before arriving at this point, however this never occurred in any simulation. If the time complexity of each simulation tick was not an issue, this could be avoided by recomputing using A* the shortest route to any fuel point. Performed every tick and adding redundancy for traffic changes, this would alert the agent the moment the fuel cost of this route is too close to the vehicles fuel level. If the simulation was to be run on rural areas with greater distances between fuel points, this or a variant thereof would be necessary.

5.7 Graphics

5.7.1 Map

.NET has excellent built-in 2D graphics capabilities. The street map and agents are drawn onto a *Bitmap* using GDI+ (Graphics Device Interface) and this image is placed into a *PictureBox* control of the main GUI form. The rate at which the simulation state is redrawn is controllable using a slider or can be disabled entirely. For better performance, instead of redrawing the map periodically, a background image is drawn and the periodic refreshes only draw an overlay onto a clone of the original image. As a flattened bitmap is provided, this means that the display does not suffer from a flickering effect at high refresh rates.

5.7.1.1 Projection

A *CoordinateConverter* object was made to translate geographic latitude and longitude pairs into (X,Y) points to draw. It is initialised with the *Bounds* object of the *StreetMap* and the size of the canvas. The maps in the evaluation are arguably small enough to assume a flat earth, however as large maps such as Iceland and Alaska were used in informal testing, a Mercator projection was implemented. Where λ is the longitude, ϕ is the latitude and λ_0 is the centre longitude in the map bounds, the formulae used are as follows [52]:

$$x = \lambda - \lambda_0 \qquad\qquad y = ln(tan(\frac{\pi}{4} + \frac{\phi}{2}))$$

x and y are then scaled to fit inside the canvas and converted to an integer. The canvas is allowed to be stretched in any direction. y is inverted, as the origin of the drawing panel is in the top left corner. This operation uses the minimum and maximum values found in the *Bounds* object.

$$x = PanelWidth \times \frac{x}{\lambda_{max} - \lambda_{min}} \qquad y = PanelHeight \times (1 - \frac{y - \phi_{min}}{\phi_{max} - \phi_{min}})$$

5.7.1.2 Background Image

By using a *CoordinateConverter*, map features at arbitrary coordinates within *Bounds* can be drawn. The road network is drawn by iterating through the *StreetMap*'s list of *Ways*, enumerating each *Node* and drawing a line between each pair of projected node coordinates. As the parser ignores any ways that stretch outside the map's bounds, the road network appears as a self-contained web. Disconnected ways are still shown, but

are neither selectable nor traversable by the agents. For large maps, it is preferable to choose thin black lines. For small maps, a thick outlined line is used, allowing one-way roads to be distinctly represented. The drawing mode for $Ways$ can be changed by the user during the simulation.

After iterating over the $Ways$, the $StreetMap$'s nodes are iterated over. Depending on the user's preferences, special nodes may be drawn. These include filled yellow rectangles for each business node; road delay nodes, as small transparent red rectangles, that may be filled in by the overlay when activated; fuel points, as white boxes with an 'F' inside; and depots as pink boxes with the depot letter (A-Z) inside.

This background image can take time to render due to having to draw thousands of arcs. Hence, a clean copy with no overlay is stored. It is redrawn only when the user toggles the road draw mode option, toggles showing special nodes, resizes the window or loads a new map. Figure 5.12 shows a projection of Mayfair, London with no simulation overlay.

FIGURE 5.12: Map of Mayfair, using thin and thick roads, with all map features shown.

5.7.1.3 Simulation Overlay

The simulation overlay conveys 'live' traffic and road delay information, as well as the position and plans of the agents.

Traffic intensity is drawn as thick, semi-transparent red lines. The alpha value of the red colour varies by minute with the speed of the road, as compared to the best recorded speed that week – typically equal to the speed limit. Set to 3.0 by default, a coefficient α can be set by the user to adjust the darkness of the traffic display. The ARGB color of the line is $(A = min(\alpha \times Way.GetSpeedDifferenceAtTime(now), 255), R = 255, B = 0, G = 0)$. This is drawn for each Way that has HERE traffic data associated with it.

If road delays are shown and a road delay is 'active' (i.e. traffic flow at this point is currently blocked), the red rectangle drawn on the background image is 'filled in' using

a filled red rectangle on the overlay. During the simulation, a large city with many traffic lights and crossings will have flashing red boxes, which flash more often during busy times. If the agent is on a road delay node, the rectangle is shown on top, but as the inverse/complimentary colour of the agent, so as to be distinguishable. This is found by XOR-ing the ARGB integer with 0xFFFFFF. Figure 5.13 shows traffic flow and road delays at 7 AM in Greater London. 100 agents are spawned to highlight the aforementioned feature. To see how the traffic flow display varies by day, refer back to figure 4.4.

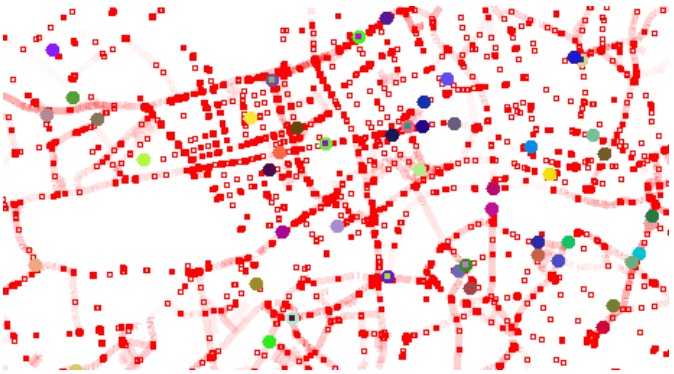


FIGURE 5.13: Map of London Zone 1, showing the traffic overlay (red-shaded roads), active road delay nodes (filled red squares), inactive road delay nodes (empty red squares) and agents (coloured circles).

Agents are represented as filled circles with diameters of 10px. Each agent has a *Color* field, which is selected from a set of distinct colours, or randomly after there are more than ten agents. When they have a *Delayer* timer object that is not yet elapsed, they are drawn as a pie chart sector, with the 'sweep angle' corresponding to the amount of waiting time left. See figure 5.14 for an example. Optionally, an agent-coloured line can be drawn between the agent and their destination, if one exists.

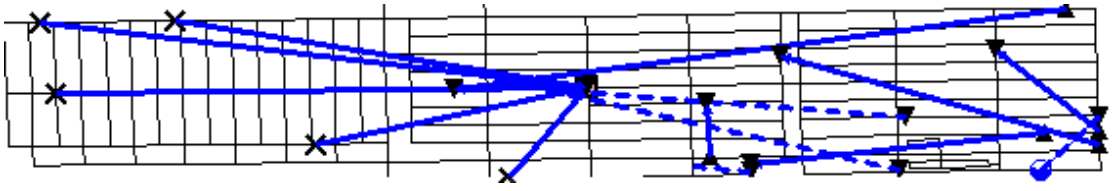


FIGURE 5.14: Job view for one agent, showing 5 drop-offs on the left side of the map being rerouted to the depot in the centre. The agent is shown as a two-thirds-filled pie circle in the bottom-right, waiting for the customer at a pick-up point. A thin line is drawn from the agent to its next destination, a drop-off, on the far right.

The overlay displays all agents' jobs. Three ways of drawing this are presented and switching between them mid-simulation is a quick and effective way of assessing the optimality of the plan, as generated by the algorithms in the next chapter. Examples of the three are shown in figure 5.15.

- **Job view** draws agent-coloured arrowed lines between the pick-up and drop-off waypoints for each job. A dashed line is used if the pick-up has already taken place. Each unvisited waypoint is marked with a black triangle, pointing upwards for pick-ups and downwards for drop-offs. Failed drop-off waypoints, once visited, are marked with a cross and a line from the cross to either the nearest depot or the pick-up position is drawn – see figure 5.14 for an example.
- **Line view** simply draws a straight agent-coloured line between each waypoint in the order that the plan dictates it will visit them in. For this reason, it is expected for the path to mutate periodically, as waypoints are added/removed or traffic conditions dictate a different route. An optimal plan, like an optimal solution to the Hamiltonian Path problem, will usually have no lines crossing over.
- **Route view** does the same as line view, however the lines are drawn onto the road network as exact routes. This can be more insightful, as seemingly long, suboptimal arcs between waypoints may be explained away by the choice of road.

5.7.1.4 Route Testing

Because optimal route finding is key to efficient couriering, some effort was put into creating the facility to find the optimal path between two points selected by the user. From the route testing menu, the user can select a $RouteFindingMinimiser$ (see section 5.3.3). Selecting the 'Route from...' item allows the user to select any node on the graph with the mouse. Doing so labels the node 'FROM'. Selecting a second node will draw the route and label the node 'TO (n hops, y km, t_0 - t_1 min, f_0 - f_1 L)'. The times and fuel usages shown correspond to the times without traffic and with traffic on a Monday at 8 a.m. The labels are drawn atop a white rectangle for readability. As the $AStarSearch$ class provides methods to get a list of the nodes in the order in which they were explored, these are plotted and given colours of the visible light spectrum, relative to their position in this list. The result plainly shows the order in which nodes were evaluated, as well as the beam of the search.

Selecting a different minimiser at this point will recompute and draw a new route. If the 'Keep refreshing route' menu item is checked, it will redraw the route upon the user adjusting the A* epsilon value in the parameters GUI form. Figure 5.16 shows an example of a route query in London, minimising time with traffic (the slowest minimiser). Increasing ε gives a higher weight to the heuristic cost, making the search more prone to minimise straight-line distance. This results in a suboptimal route through London, almost twice as costly, being chosen, rather than using the M25 motorway. However,

many fewer nodes were evaluated. Notice how, despite the shorter distance, the fuel usage is greater for the inner city route. This is due to the slower driving speed.

5.7.2 Graphical User Interface

The simulator was built with ease of use in mind. As mentioned earlier in this chapter, in addition to the main GUI form which provides the map display and access to commands, there are GUIs for graphing simulation variables, viewing the statuses of agents and jobs, viewing the event log and controlling parameters. Figure 5.17 shows a sample session with all GUI forms open.

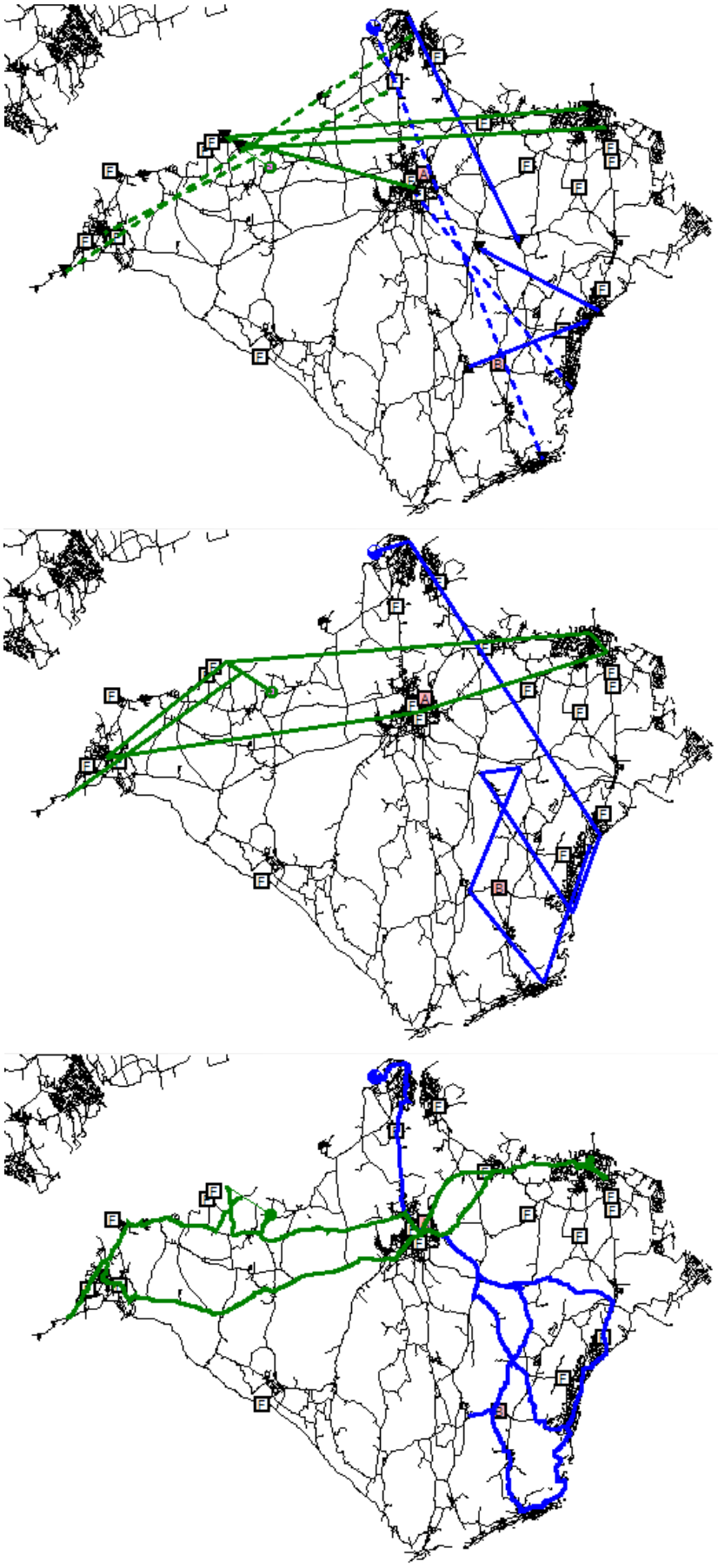


FIGURE 5.15: An example simulation in the Isle Of Wight with two agents. The agents' *CourierPlans* are displayed in (1) job view, (2) line view and (3) route view.

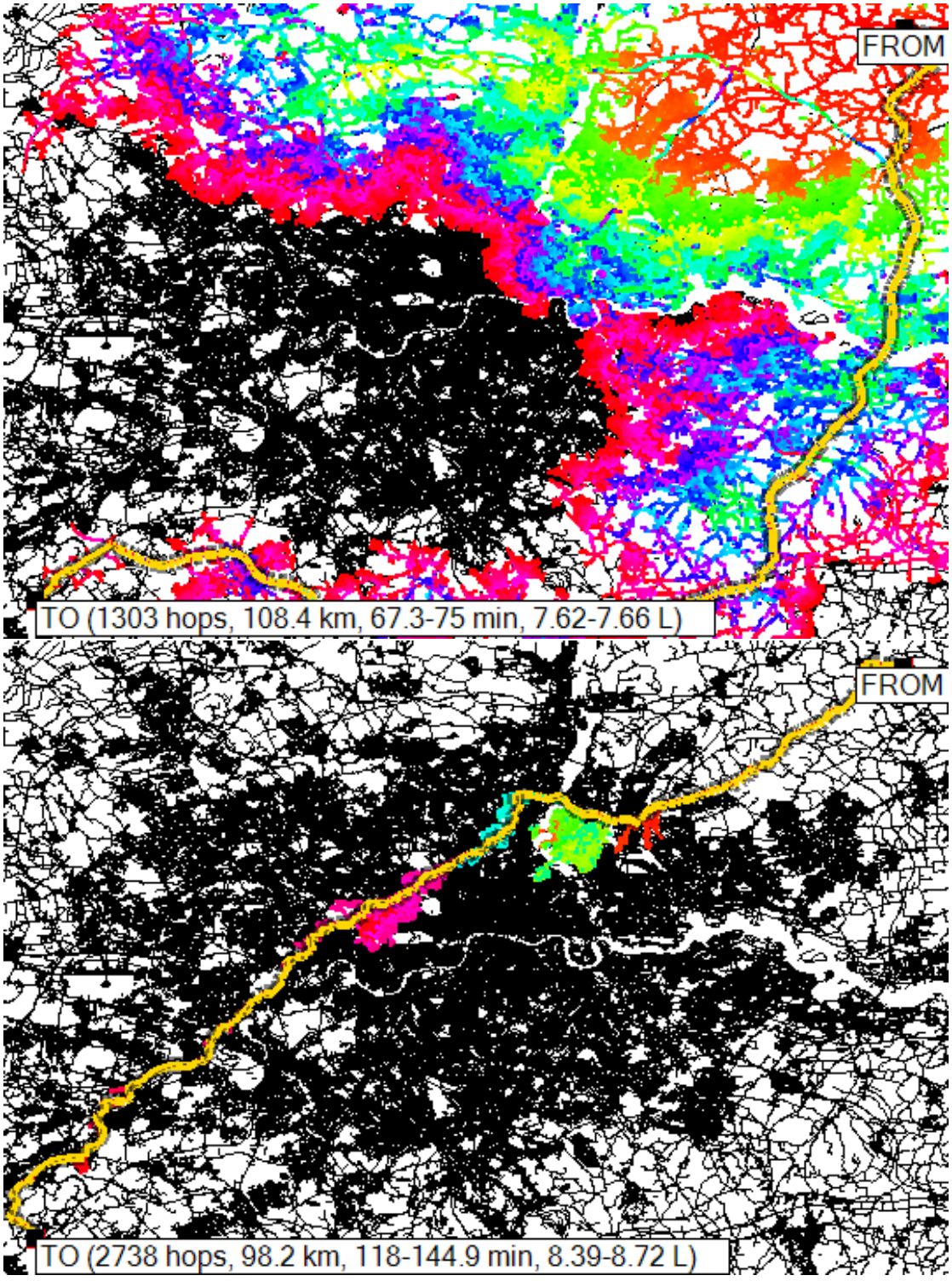


FIGURE 5.16: Route testing in Greater London, minimising time with 8 a.m. traffic. Epsilon values of (1) $\varepsilon = 0$ and (2) $\varepsilon = 4$ were used on the same start and end points. This shows how degrading the admissibility of the heuristic reduces the search space (nodes that are either coloured or lie directly under the route drawn), but can result in a suboptimal route being selected. Black portions of the map are unexplored nodes.

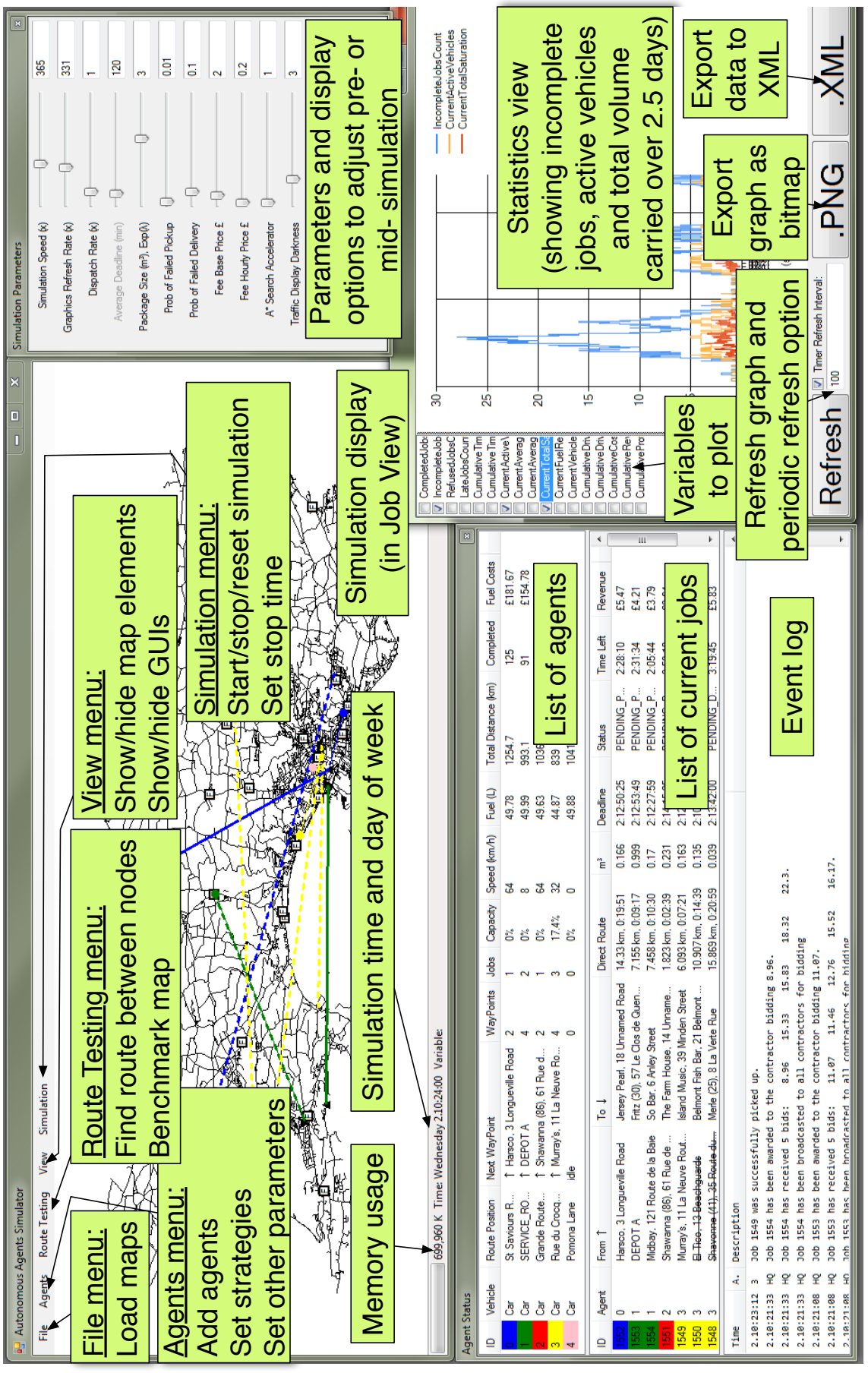


FIGURE 5.17: A sample session in the simulator, with GUI elements and other functionality annotated.

Chapter 6

Optimisation

In the previous chapter, we described in detail the simulator that was built. Using this as a foundation, we proceed to design and implement algorithms to locally optimise routes through multiple waypoints, to allocate jobs to agents and to control the agents when they are idle.

6.1 Planning

Given a starting position and a set of jobs allocated to a single agent, it should complete these jobs by taking an optimal or near-optimal multi-hop route, in regard to driving time or fuel usage. The main goal is minimising the number of late deliveries and as later discussed in section 6.2, the routing and allocation strategy may also assist to this regard. The planning algorithm is fundamental to the performance of the courier agents. The problem at hand is a specialisation of the Hamiltonian Path problem, which differs from the travelling salesman problem only in that the solution is a path, not a cycle. These problems are NP-hard and are impractical to solve by brute force because given n points to visit, there are $n!$ different routes.

Implementing the planner was a particular challenge, as no existing solutions to this particular problem could be found. Heuristic algorithms to more general problems like the travelling salesman problem, the Hamiltonian path problem and even the vehicle routing problem generally cannot be adapted such that the constraints are respected.

Many classes of algorithm were attempted, however the one presented here uses an inexpensive greedy algorithm, followed by an expensive genetic algorithm if the latter fails. Both of these are self-contained in a class named $NNGAPlanner$, which takes an $Agent$ and outputs a $CourierPlan$. In some cases, it will output null, as one can specify

whether or not to use the genetic algorithm. More information on how this is used can be found in section 6.2.

6.1.1 Problem Specification

To decompose the inputs of the planning problem, we define a set of waypoints W in place of jobs. A job can be represented as two waypoints – one for the pick-up and one for the drop-off. A waypoint has:

- a geographic position on the road network.
- a volume delta in cubic metres, corresponding to the amount of volumetric space lost after visiting the waypoint. For drop-offs this is negative and for pick-ups this is positive.
- (for drop-offs only) a predecessor waypoint
- (for drop-offs only) a deadline

Note that if a delivery fails and the job needs to be rerouted to the nearest depot, we merely modify the position and deadline of the drop-off waypoint. If a pick-up fails, the drop-off waypoint is removed.

Three additional inputs to the problem, representing the starting state of the agent, are required:

- $\mathcal{S}$ – the geographic starting position of the agent.
- $\mathcal{C}$ – the current remaining capacity of the agent vehicle's storage.
- $\mathcal{F}$ – the current remaining fuel supply of the agent.
- $\mathcal{T}$ – the current time

The happened-before relation, denoted $\rightarrow$, is used to show that an event takes place before another. It is a strict partial order that is transitive, irreflexive and antisymmetric. The sequence of n waypoints $w_1 \rightarrow w_2 \rightarrow w_3 \rightarrow ... \rightarrow w_n$ correspond to the order in which they will be visited and fulfilled. The optimal routes between pairs of waypoints are denoted $w_i w_j$. The chain of routes, $\mathcal{S}w_1, w_1w_2, ..., w_{n-1}w_n$, corresponds to the agent's complete journey. Each route has an estimated time and fuel cost.

A sequence is **valid** iff:

- $\nexists w \in W. \quad w \to predecessor(w)$. *Pick-ups occur before drop-offs.*
- Starting from the start state, no subsequence of any length m has a negative cumulative sum of volume deltas. $\forall m \leq n. \quad \mathcal{C} - \sum_{i=1}^{m} volume(w_i) \geq 0$. *The vehicle is never over-capacity.* [1]
- $\mathcal{F} - fuel(\mathcal{S}w_1) - \sum_{i=1}^{n-1} fuel(w_i w_{i+1}) \geq 0$. *The vehicle never runs out of fuel.*

Let ω be the maximum waiting time, common to all waypoints, and define a function for arbitrary pick-up waypoints w_p and delivery waypoints w_d:

$$OnTime(w_p) = 1$$

$$OnTime(w_d) = \begin{cases} 1, & \text{if } deadline(w_x) \geq \mathcal{T} + time(\mathcal{S}w_1) + \sum_{i=1}^{x-1} (time(w_i w_{i+1}) + \omega). \\ 0, & \text{otherwise.} \end{cases}$$

To improve the robustness of the solution to the planning problem, $\mathcal{T}$ can be substituted for $\mathcal{T} + n\tau$, where τ is an arbitrary redundancy time that is added for each route to account for road delays. A sequence is **on-time** iff:

$$\sum_{i=1}^{n} OnTime(w_i) = n$$

The goal is to find a sequence of waypoints from W that is optimal. A sequence $Q = w_1 \to w_2 \to ... \to w_n$ is optimal if it is a valid and on-time permutation of W and the cost function is minimised:

$$Cost(Q) = cost(\mathcal{S}w_1) + \sum_{i=1}^{n-1} cost(w_i w_{i+1})$$

See 5.3.3 for a description of the cost functions, which may be distance, driving time or fuel usage, with or without traffic. If there does not exist an on-time sequence, the goal is to find valid solutions that maximise the number of on-time deliveries[2] and from this solution space, find the one that is of minimal cost.

$$OnTimeWayPoints = \sum_{i=1}^{n} OnTime(w_i)$$

[1] It is also true that $C + \sum_{i=1}^{n} capacity(w_i)$ will equal the vehicle's capacity

[2] An alternative representation could be to minimise the sum of minutes late over all jobs. However, minimising the number is more profitable if, as is the case here, late jobs are fully refunded.

6.1.1.1 Additional Constraints

As required in section 6.2.1.3, an alternate formulation of this problem exists. It modifies the first condition for sequence validity by requiring that drop-offs occur immediately after pick-ups:

$$\forall w_i \in W, i > 1. \quad w_{i-1} = predecessor(w_i) \vee w_i = predecessor(w_{i+1})$$

Such a constraint greatly reduces the cost-saving opportunities for the courier agent, however it is more fault-tolerant in practice and would be more appropriate for high value cargo that must be delivered without any intermediate stops.

6.1.1.2 Simplifying Assumptions

To simplify the problem and reduce the search space, the following assumptions were made:

- The deadline for a pick-up waypoint w_p can be set to the deadline of their associated drop-off waypoint, w_d minus the minimum duration of the direct route (i.e. assume no traffic or delays) and the maximum wait time. $deadline(w_p) = deadline(w_d) - time(w_p w_d) - \omega$.
- It is almost guaranteed that an agent will be idle at least once every twenty-four hours. The main idle strategies (see 6.3) begin by refuelling at a local fuel point. Furthermore, in the event of a fuel shortage, the agent will suspend all jobs to refuel and resume thereafter (see 5.6.3.2). The result is that the fuel constraint can be ignored.

6.1.2 Preliminary Computation

The design specification in section 3.2.2 states that a customer will receive a phone call notifying them five minutes before their courier will arrive. When the planner is initialised with an existing *CourierPlan*, it first iterates through the existing list of routes, locking any waypoints that would be reached within five minutes, as presumably, these customers have already received their phone call. As mentioned before, there exists a rare edge case where the agent may have locked in a pick-up, but is unable to perform it as a delivery failure occurred in the last five minutes. For this reason, the iteration short circuits if the capacity constraints are violated and the pick-up must

be rescheduled whenever optimal. A *CourierPlanState* is a struct which is used by the greedy algorithm to checkpoint progress. It consists of: wall time, current map position, capacity left and a list of the remaining waypoints. *NNGAPlanner* first builds a starting state. The planner ignores the deadlines for the locked waypoints when it runs the following one or two algorithms.

6.1.3 Nearest Neighbour Search

The greedy nearest neighbour algorithm for the travelling salesman problem is described in the background (section 2.4.1). For the purposes of this planning problem, it can be modified such that the next neighbour chosen must maintain the validity and punctuality of the sequence of waypoints. In addition, the arcs between waypoints are in fact paths – *Route*s found by an A* search. Hence, evaluating the cost of travelling between two waypoints requires an expensive A* search and an aggregate cost function over each *Hop*. Previous attempts to solve the planning problem showed that this adaptation was too simple. Because the algorithm picks arcs to nearby waypoints over farther ones that are approaching their deadline, it frequently halts due to the sequence not being on-time. Nonetheless, the algorithm is promising and with an ability to *backtrack*, it can be very effective.

The nearest neighbour best-first search algorithm we developed performs a greedy search using a *Stack* of *NNSearchNode*s. These nodes correspond to partial solutions. They have associated parent nodes, to allow an ordered list of waypoints (a complete solution) to be generated from the final node if the algorithm succeeds. In each loop of the algorithm, a node is popped from the stack. From this, a set of child nodes are evaluated for each remaining waypoint whose position here would be valid, given the predecessor and volume delta rules. Nodes that have a wall time exceeding any of the deadlines of its current waypoint and the remaining waypoints are filtered out. This gives a list of valid, on-time next node selection candidates. They are ordered by route cost and pushed onto the stack, such that the least costly, 'nearest' neighbour is popped in the next iteration. If no nodes are valid and on-time, in the next iteration the algorithm will backtrack to the next node on the stack.

With no backtracking, this algorithm runs in quadratic time, meaning that for n cities, up to $\frac{1}{2}(n^2 - n)$ A* searches will be performed. With backtracking, the algorithm could theoretically run in factorial time. For this reason, a counter is used to keep track of the number of failed branches – where the node popped from the stack yielded no on-time child nodes. The stopping criteria are: a solution is found, the stack is empty or 720 branches had failed. 720 is used because it allows an exhaustive search of six waypoints

(6! = 720). Informal testing showed that increasing it further made little difference to failure rate, but did negatively impact computation time.

It is made less expensive still by bounding the number of waypoints that are actually considered at each node. The algorithm is summarised in 3 and a step-by-step working of an example problem is given in figures 6.1–6.4. The LINQ query below is used to build an array of subsequent waypoints. It first filters the invalid ones. It then orders the waypoints by their orthodromic distance (a far less expensive computation than A*) and selects the shortest three candidates. This query corresponds to lines 9-10 in the pseudocode. If the speed of the planner was less important, `Take`-ing more than three or omitting `Take` entirely would result in marginally improved solutions.

```
Node.State.WayPointsLeft.Where( _
            Function(W) Node.State.CapacityLeft - W.VolumeDelta >= 0 AndAlso _
                Not Node.State.WayPointsLeft.Contains(W.Predecessor)). _
                    OrderBy(Function(W) HaversineDistance(Node.State.Point, _
                                                          W.Position)).Take(3)
```

Algorithm 3 Nearest Neighbour Search (Planning)

```
 1: function BUILD-PLAN(START, LOCKEDWAYPOINTS)
 2:     nodeStack = {start}
 3:     failedBranches = 0
 4:     while nodeStack not empty ∧ failedBranches < 720 do
 5:         n = nodeStack.pop                          ▷ the current node to expand
 6:         if waypointsLeft(n) is empty then
 7:             return lockedWaypoints ++ n.visited                ▷ solution found
 8:         nextNodes = {}                                          ▷ a sorted list
 9:         nextWaypoints = waypointsLeft(n) where valid
10:         select bottom 3 from nextWaypoints, ordered by distance to position(n)
11:         for all w ∈ nextWaypoints do
12:             next = CREATE-CHILD-NODE(n, w)
13:             if ∃w ∈ waypointsLeft(n) | deadline(w) − τ < time(next) then
14:                 failedBranches = failedBranches + 1        ▷ a deadline was missed
15:                 continue
16:             cost = GET-COST-OF-SHORTEST-PATH(position(n), position(next))
17:             add ⟨cost, next⟩ to nextNodes
18:         for all next ∈ nextNodes do          ▷ in reverse order, greatest cost first
19:             push next to nodeStack
20:     return null                                              ▷ no solution found
```

6.1.4 Genetic Algorithm

If nearest neighbour search fails to produce a solution and one is required, the genetic algorithm will be invoked. It requires an initial solution to seed the solution pool. This solution is found using another variant of the nearest neighbour algorithm that ignores

deadline violations. Computing this is less expensive, as the routes between waypoints were most likely computed and cached in the nearest neighbour search phase.

The algorithm makes use of a fitness function for solutions that are valid, but potentially not on-time. This is designed to minimise lateness first and then minimise driving cost – i.e. a route that is twice as long is more preferable to one where any job is delivered late. If the pick-up point is late as well, this does not affect the score. Both a real and fuzzy/approximate score function are implemented.

For a valid sequence of waypoints, $Q = w_1 \rightarrow w_2 \rightarrow ... \rightarrow w_n$ and using the definitions in section 6.1.1:

$$TrueScore(Q) = 1000 * \sum_{i=1}^{n}(1 - OnTime(w_i)) + Cost(Q)$$

$$ApproximateScore(Q) = 1000 * \sum_{i=1}^{n}(1 - OnTime(w_i)) + ApproximateCost(Q)$$

$$ApproximateCost(Q) = haversine(\mathcal{S}, w_1) + \sum_{i=1}^{n-1} haversine(w_i, w_{i+1})$$

The latter is an approximation of the true score, which requires expensive route finding. The $OnTime$ function requires a route time, which can be approximated by multiplying the orthodromic distance (found using the Haversine function) by a coefficient μ. μ is the average amount of time it takes to travel to a point 1 km away 'as-the-crow-flies' in a city. It will vary significantly, especially for cities with slow traffic speeds or road networks that are difficult to navigate. An approximation of μ is found as the seed solution is constructed. That is, if the seed solution is named Q:

$$\mu = \frac{time(\mathcal{S}w_1) + \sum_{i=1}^{n-1} time(w_i w_{i+1})}{ApproximateCost(Q)}$$

A mutator is also defined to swap two waypoints, without breaking the solution's validity. It should be noted that mutating a courier plan is not as trivial as mutating a travelling salesman problem solution, due to the constraints. Each waypoint in the solution is highly dependent on the previous waypoints, especially when the packages are large. For this reason, it makes sense to only perform adjacent, pairwise swaps. The function $GetAllMutations$ takes a valid solution, iterates over each neighbouring pair of waypoints that are not locked in place and returns a list of mutated solutions. The function first defines an array CR to represent the vehicle's remaining capacity immediately before fulfilling the waypoint at each index. Two adjacent waypoints $w_i \rightarrow w_j$ can always be swapped if:

- Both are pick-ups. We know $CR_{j+1} = CR_i - volume(w_i) - volume(w_j) \geq 0$, so we also know $CR_{j+1} = CR_i - volume(w_j) - volume(w_i) \geq 0$.
- Both are drop-offs (trivially).
- w_i is a pick-up and w_j is a drop-off, so long as they do not correspond to the same job – i.e. $w_i \neq predecessor(w_j)$.
- w_i is a drop-off and w_j is a pick-up, only if at position i, there is room for w_j to come first – i.e. $volume(w_j) \leq CR_i$.

Having defined the fitness and mutator function, and given the nearest neighbour solution as the seed, the structure of the genetic algorithm is summarised as pseudocode in algorithm 4. Figure 6.5 uses the same example as used in figure 6.1 and shows one generation of mutations.

Algorithm 4 Genetic Algorithm (Planning)

```
1:  function BUILD-PLAN(SEED)
2:      solutionPool = {⟨SCORE-SOLUTION(seed), seed⟩}        ▷ a priority queue
3:      topSolutions = {}                                     ▷ a sorted list
4:      for generation = 1 to 500 do
5:          ⟨fitness, solutionToMutate⟩ = remove-min(solutionPool)
6:          add ⟨fitness, solutionToMutate⟩ to topSolutions
7:          for all mutation ∈ GET-ALL-MUTATIONS(solutionToMutate) do
8:              if mutation ∉ solutionPool ∪ topSolutions then
9:                  mfitness = SCORE-SOLUTION(mutation)
10:                 add ⟨mfitness, mutation⟩ to solutionPool
11:         if solutionPool is empty then
12:             break                  ▷ all solutions evaluated and in topSolutions
13:     ⟨bestFitness, bestSolution⟩ = remove-min(topSolutions)
14:     return bestSolution
```

6.1.5 Other Attempts

In early stages of the project, an insertion heuristic was used to solve the planning problem. It required as input, a *CourierPlan* and one additional job. It worked by taking the existing sequence of waypoints, $w_1 \rightarrow w_2 \rightarrow ... \rightarrow w_n$, and testing each possible interleaving of this with the new job's waypoints: $w_p \rightarrow w_d$. This ensured the original ordering is maintained, so there would be no precedence violations. For n waypoints, there would be $\frac{1}{2}(n+2)(n+1)$ permutations to test. If only a single waypoint was being inserted (after a failed delivery), the time complexity was much less – only $n + 1$ permutations.

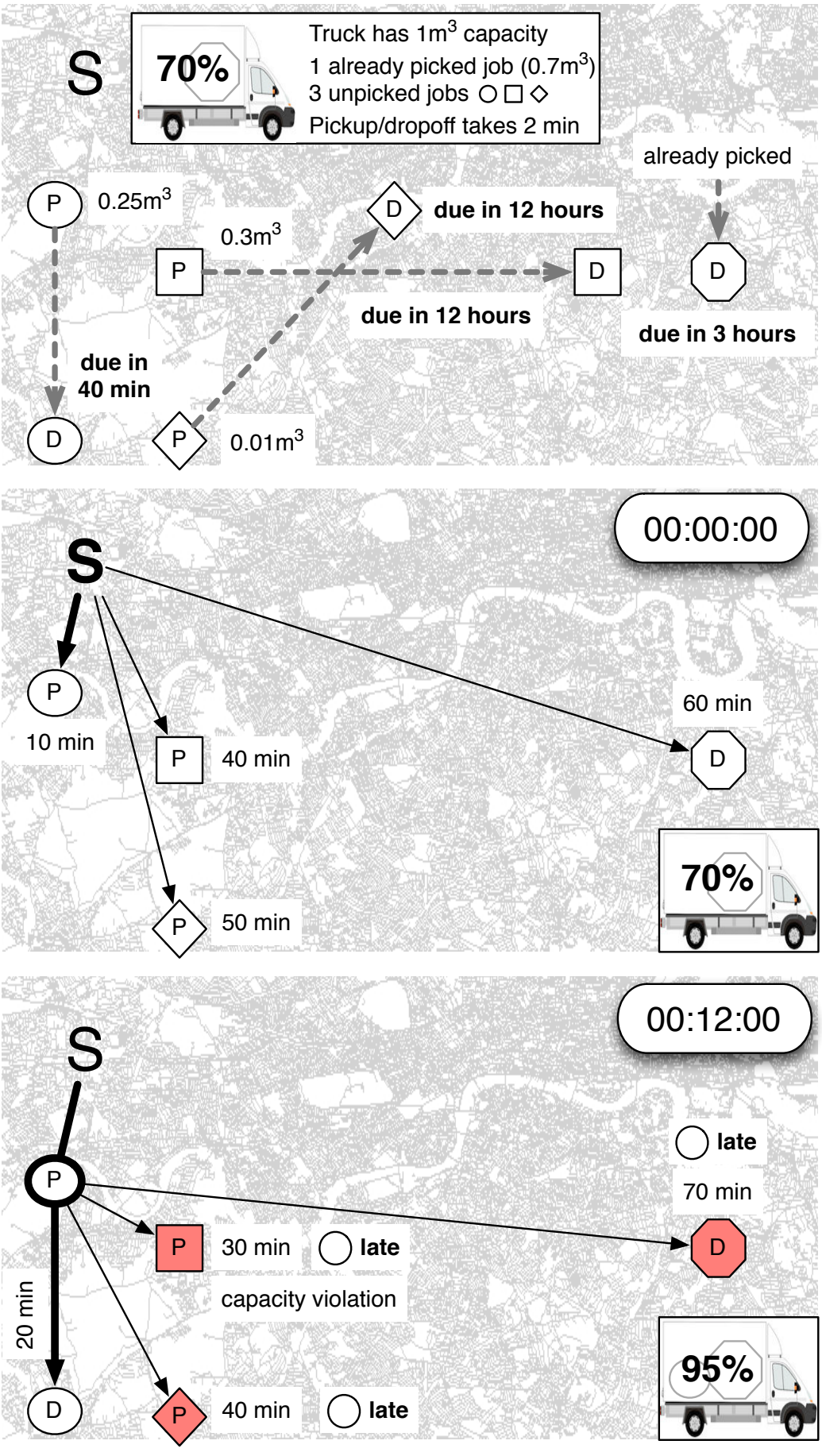


FIGURE 6.1: Nearest Neighbour Search example. Inputs and steps 1–2.

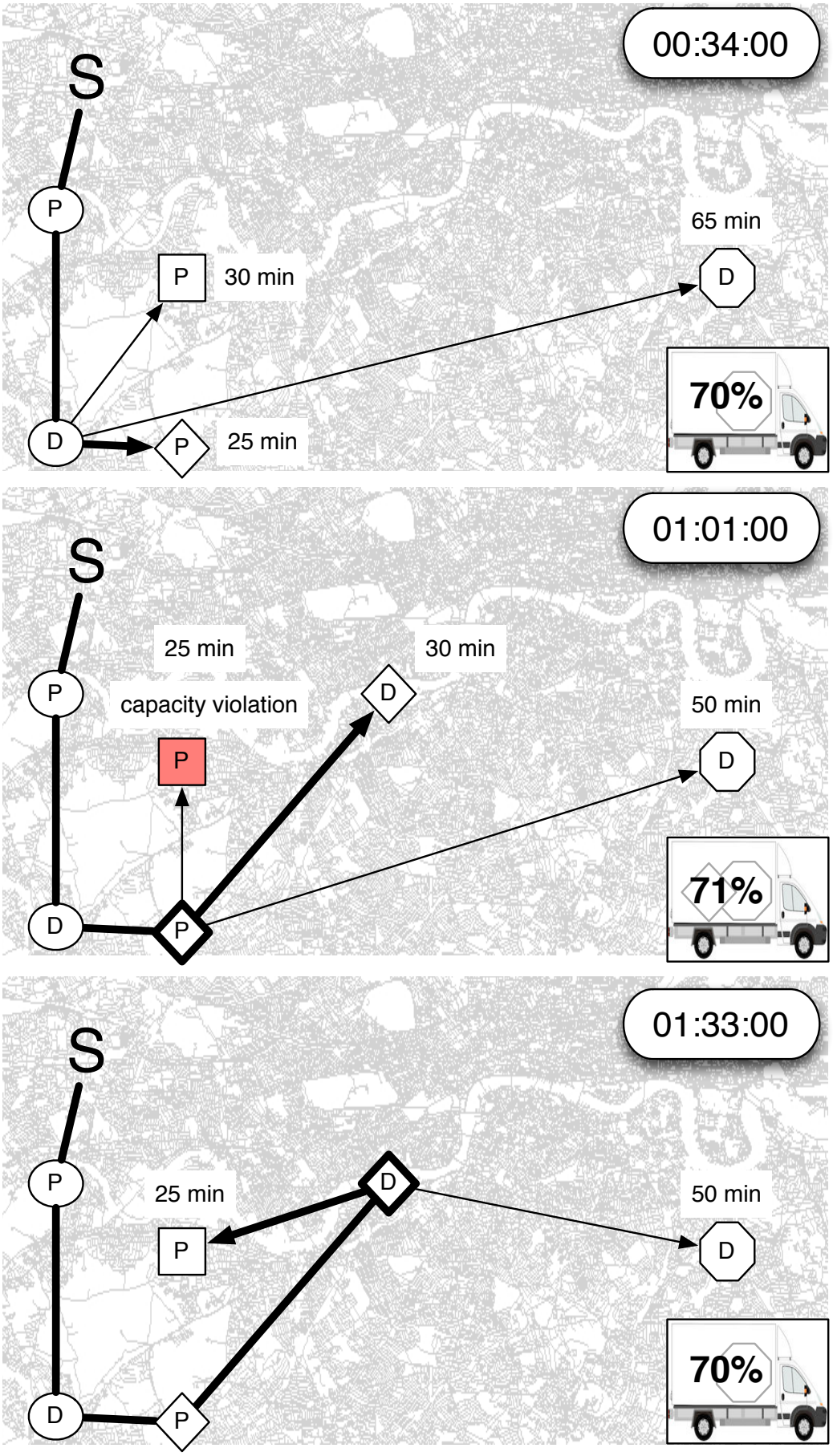


FIGURE 6.2: Nearest Neighbour Search example. Steps 3–5.

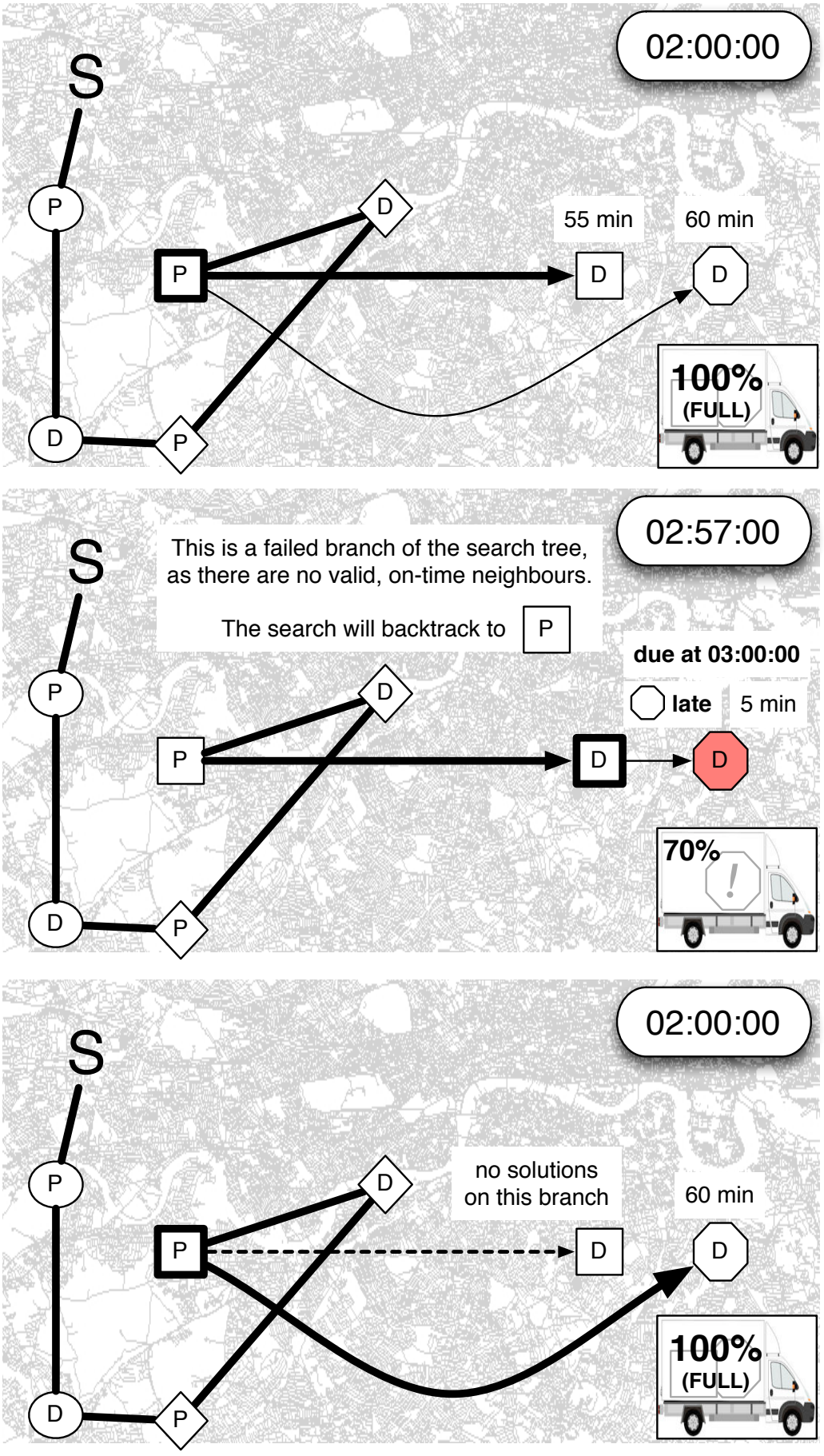


FIGURE 6.3: Nearest Neighbour Search example. Steps 6–8, including backtracking.

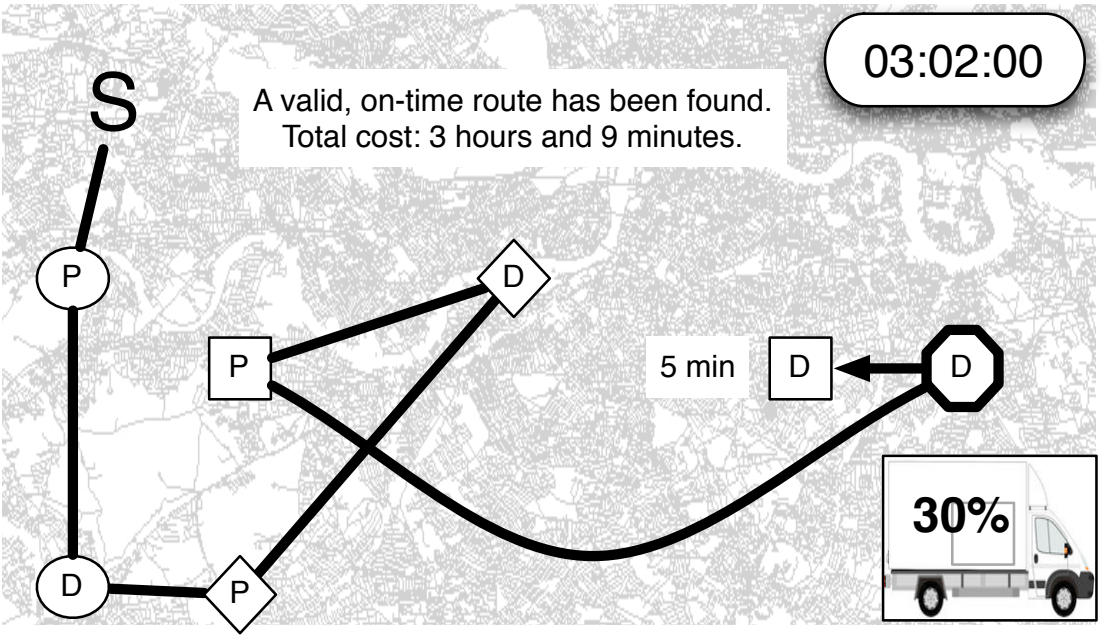


FIGURE 6.4: Nearest Neighbour Search example. Final step 10, showing solution.

For each permutation, it would first perform a validity check with regard to capacity constraints. The cost, found using a function similar to the exact fitness function of the genetic algorithm, was calculated for each new valid route and the best route was selected. Though the time complexity of this solution was within bounds, the optimality could be very poor depending on the order of new jobs being inserted. An attempt was made to mitigate this effect by using a brute force solver, $ExhaustiveTSPPlanner$, to test every permutation up to six waypoints (720 permutations) and then use the insertion algorithm thereafter. Unfortunately, this made little difference to the optimality of longer solutions and was not feasible for large maps due to the large number of A* searches being performed. This brute force solver, however, was useful in evaluating the above algorithms in 7.1.2.

The algorithm critically does not have the potential to replan at arbitrary times. Namely, it was not clear how to efficiently handle failed pick-ups and drop-offs, apart from to simply remove the unneeded delivery waypoint or immediately insert a depot waypoint after $\mathcal{S}$. When traffic was added to the simulation, replanning (i.e. completely reconstructing the route from scratch) was required and the nearest neighbour search algorithm replaced the insertion.

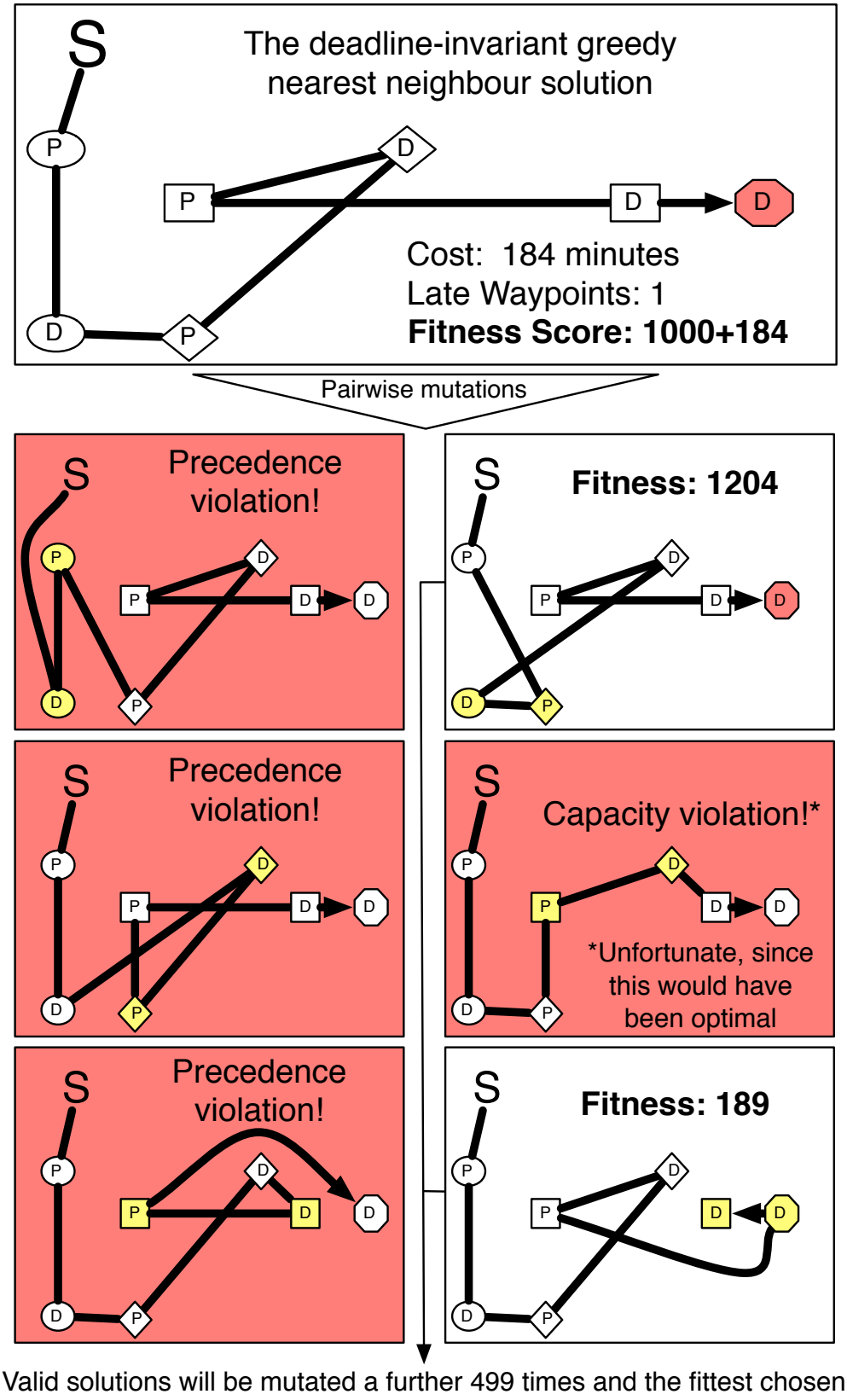


FIGURE 6.5: Mutations performed by the genetic planning algorithm on the example problem in figure 6.1.

6.2 Routing and Job Allocation Strategies

The strategy pattern is used in the simulator to facilitate the development and evaluation of multiple schemes of routing around the environment and fulfilling jobs. Agents 'have' one of the three subclasses of *AgentStrategy* that were developed and are presented below. They also have an *IIdleStrategy*, which dictates an agent's behaviour when there are no jobs in its *CourierPlan*. Any two routing and idle strategies can be selected by the user, however agents will always assume that they share the same type of routing strategy.

6.2.1 Contract Net Protocol

The Contract Net Protocol is the most effective method for job allocation. As described in the background section 2.1.1, the protocol entails running an auction where each agent (members of the 'contract net') can bid for jobs. The implementation of this from the perspective of the centralised *ContractNetBroadcaster* is in section 5.5.2. The behaviour of the agents is realised in *ContractNetStrategy* and *ContractNetContractor* and described here. Five routing strategies for the contract net protocol were written and they are numbered CNP1–5, in the order of their complexity and flexibility.

Following the broadcast of a job, the agent recalculates the cost of their existing list of routes in their *CourierPlan*. If the agent is on a fuel diversion (see 5.6.3.2), the starting point $\mathcal{S}$ is assumed to be that of the fuel point. For predicting whether deadlines are met, the starting time $\mathcal{T}$ is set to be the fuel point ETA plus the refuelling time. Otherwise, the current *HopPosition* and time is used. It then attempts to create a new *CourierPlan* that incorporates the new job and any such fuel diversion. The specifics of this step depends on the CNP variant. If it finds an on-time solution, it calculates the total cost of this new plan and 'bids' the extra cost. Depending on the *RouteFindingMinimiser* that all agents share, this is either the extra distance (kilometres), driving time (hours) or fuel required (litres) (see 5.3.3). If no on-time solution was found, the agent replies with a null bid. The agent that bid the lowest cost is awarded the job and notified. When the strategy is next run the *CourierPlan* is updated to include the newly allocated job.

6.2.1.1 CNP1

Agents bid for jobs *only if they are idle*, hence they will only fulfil one job at a time. Let w_1 and w_2 be the pick-up and delivery waypoints of the new job respectively and

using the notation defined in 6.1.1, the condition must hold:

$$deadline(job) \geq \mathcal{T} + time(\mathcal{S}w_1) + \omega + time(w_1w_2) + \tau$$

Provided that the deadline would be met, an agent will always bid:

$$cost(route(\mathcal{S}, job.pickup)) + cost(route(job.pickup, job.dropoff))$$

6.2.1.2 CNP2

Agents can bid for jobs if they are non-idle, however new jobs can only be appended to the tail end of their existing route, even if it would be less costly to insert the new job at the start or mid-route. A CNP2 agent's waypoint list is a FIFO queue. With this variant, the agent needs to iterate through its routes, and calculate an ETA at which it would be arriving at the drop-off position of this new job. Using the notation defined in 6.1.1, and denoting the new jobs' waypoints as w_{n-1} and w_n, a bid can only be made if:

$$deadline(job) \geq \mathcal{T} + n\tau + time(\mathcal{S}w_1) + \sum_{i=1}^{n-1}(time(w_iw_{i+1}) + \omega)$$

or if the agent is currently idle, the deadline condition from CNP1 must hold. Let L be the drop-off point of the agent's last job, or if the agent is idle, its current position. Provided that the deadline for this new job would be met, an agent will always bid:

$$cost(route(w_n, job.pickup)) + cost(route(job.pickup, job.dropoff))$$

6.2.1.3 CNP3

CNP3 extends the specification further to allow jobs to be inserted into the plan wherever it is most optimal. However, a job's drop-off must always occur immediately after its pick-up. Hence, an agent is only fulfilling one job at a time and at no time will the vehicle's storage contain more than one package. Using this strategy allows one to model and evaluate an autonomous courier service for high value cargo, where packages never come into contact with other customers, as is the case with CNP4 and CNP5.

A simpler, modified version of the nearest neighbour algorithm (as described in section 6.1.3) is used, which reasons about jobs rather than waypoints. When a new job arrives, each agent runs the nearest neighbour algorithm to compute a near-optimal route. If the algorithm is successful at finding a valid, on-time plan, the agent bids the difference between the new total cost and the old total cost. Otherwise it bids nothing.

With CNP3–5, the agent will periodically attempt to replan. Due to road delays and uncertainty in pick-up and drop-off times, the optimality of the current *CourierPlan* might be improved by constructing a new plan from scratch. Every five minutes (corresponding to the interval between traffic flow data samples), the agent will construct a new plan using the planning algorithm and compare the 'true score' (see 6.1.4) to the old plan. If the score is an improvement it will use the new plan. Given the resolution of this simulation and also the fact that the agents have good knowledge about future traffic conditions, the differences were marginal and often nil. However in the real world where there is more uncertainty in traffic flow and job fulfilment, this would unquestionably be required.

6.2.1.4 CNP4

With CNP4, Agents are now permitted to interleave pick-ups and drop-offs of all jobs, provided the sequence of waypoints it visits is valid (as defined formally in 6.1.1). A *NNGAPlanner* is used by the agent, who will only bid for a job if it can produce a plan that is also on-time in a computationally short time frame. When bidding for jobs, the agent only makes use of the nearest neighbour search algorithm (see 6.1.3) and not the genetic algorithm for performance reasons. This behaviour can be changed by a boolean constant in the agent's code. In the real world, where a replanning time of 1000 ms is just as acceptable as 100 ms, there is no reason to exclude the GA. As before, the agent bids the difference in total cost before and after inserting the new job, however because waypoints can be interleaved, this value is much less and at peak times is not really proportional to the length of the job.

Unlike CNP1–3, if a failed delivery takes place, the delivery to the depot or pick-up position can be postponed to whenever it is is most optimal. It reruns the planner, specifying to use the genetic algorithm if and only if NNS fails to produce an on-time solution. Diversions caused by failed deliveries are the most frequent cause of lateness – particularly when the failed package is large enough that it could not simply remain in the vehicle until the end of the day. For this reason, the genetic algorithm is essential in minimising the number of late deliveries.

Failed pick-ups are handled in a similar way, however the genetic algorithm would only need to be run if the existing plan is not on-time. Because the planning algorithm is not optimal, in some circumstances, the act of replanning will result in an inferior plan, especially when the old plan was one generated using the genetic algorithm. For this reason, two candidate plans are evaluated. To begin, the existing plan is modified: the drop-off waypoint is removed and a new *Route* to stitch together its preceding and

subsequent waypoints is found using an A* search. This will always be less than or equal to the old cost due to the triangle inequality. In addition, the agent constructs a new plan with this set of waypoints. The $CourierPlan$'s 'true score' (see 6.1.4) is calculated for each and the best plan is used. The customer is partially refunded using the same cost coefficient, as applied to the difference in total cost of the old plan and the new selected plan.

6.2.1.5 CNP5

The final and most advanced CNP-based strategy uses an alternative form of allocating jobs, wherein the agent who fulfils the job is not fully known at the point of broadcast and auction. Simply, jobs are not allocated to agents - they are 'tentatively owned' by them. As with the other strategies, the agents who bid for a job only bid their expected additional cost. The strategy is identical to CNP4, except that the agent now has the ability to offload some of its jobs to the other agents. It only does this when *circumstances change for the worse* and replanning using $NNGAPlanner$ could not produce an on-time $CourierPlan$. Circumstances include:

- A failed delivery takes place and the agent has to factor in an additional waypoint for the depot or pick-up position.
- Fuel runs very low and an emergency fuel diversion is required.
- Traffic conditions change. This is detected when the agent periodically replans.

The reallocation procedure is described below and an example is shown in figure 6.6:

1. The agent separates the jobs that it must fulfil – those where the pick-up has already taken place – from those that could potentially be carried out by other agents. Name the reallocatable list, R and the picked job list P.
2. Order $r \in R$ in ascending order by the deadline urgency:

$$urgency(r) = deadline(r) - time(route(r.pickup, r.dropoff))$$

3. Send a message to the broadcaster, requesting to reallocate all of R.
4. The broadcaster performs a series of consecutive auctions with the other agents, closely resembling the standard CNP bidding procedure. In each auction, the agents replan to fit in the broadcast job, bid their estimated marginal cost (or null if no on-time solution was found) and are awarded the job if they bid the

lowest. The winning agent's *CourierPlan* is updated after each job is awarded. Like before, each auction is performed in parallel for improved speed.

5. The agent is returned a list, N, of the jobs that could not be reallocated, because all other agents responded with null bids. N and P are combined and the agent uses $NNGAPlanner$ to create a new optimal $CourierPlan$.

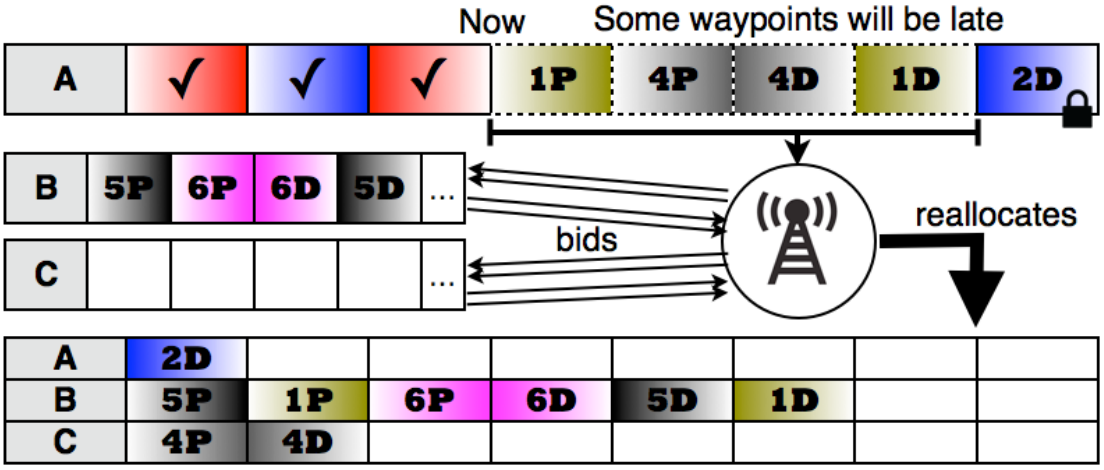


FIGURE 6.6: An example of job reallocation for CNP5 agents. Agent A anticipates late deliveries and successfully reallocates its two unpicked jobs to agents B and C.

If the simulation was enriched or this strategy implemented in the real world, many other phenomena, such as mechanical faults, could be programmed to trigger a reallocation request. The downside to this strategy is that it achieves the least autonomy, as the central broadcaster, as well as assuming the role of an auctioneer for the tasks coming on the job feed, must also be able to perform auctions, award jobs and report back the results at the request of an agent. With previous variants, only the broadcaster can initiate a conversation with agents. Due to the frequent replanning, it is also the most demanding computationally.

This strategy addresses the problem where a few dominant agents emerge and are awarded most of the jobs from the job feed, as their better coverage allows them to bid less. Even accounting for the redundancy time τ, it can fall behind schedule as circumstances change for the worse. Meanwhile, the other agents may be mostly idle. With CNP5, this lateness will be minimised or eradicated as the other agents will assume some of the workload. In practice, the number of jobs sent to the central broadcaster for a secondary auction tends to be about a third of the jobs in the plan of the oversubscribed agent. The winning bids, as before, tend to be made by the moderately busy agents due to their coverage. This may perpetuate the problem to some extent and it may seem more intuitive to allocate the unwanted jobs evenly, to the least busy agents.

However, informal testing showed that this did not affect the percentage of jobs arriving late, though, expectedly, it led to more costly routes.

A summary of the five CNP allocation strategies is shown in figure 6.7.

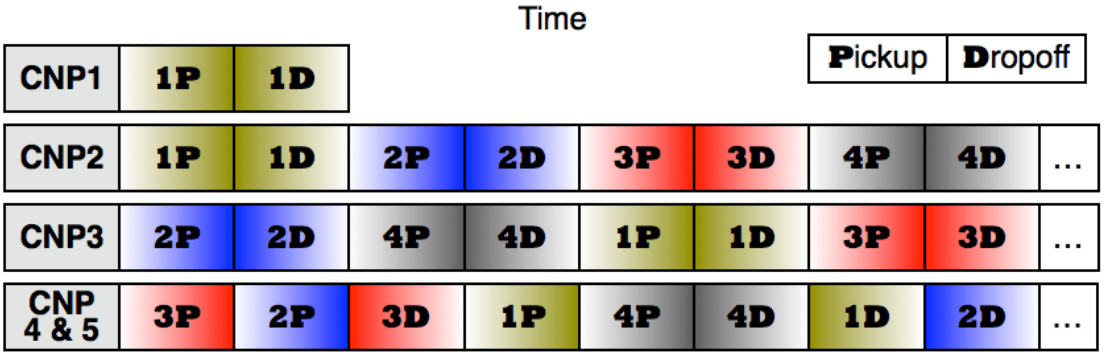


FIGURE 6.7: Valid *CourierPlans* for each variant of the Contract Net Protocol, showing the variance in the flexibility of planning.

6.2.2 Round-robin

Section 5.5.2 describes how jobs are allocated to agents via the broadcaster itself. The counterpart to the *RoundRobinBroadcaster* is the *RoundRobinStrategy*, which behaves in a similar way to CNP4 (6.2.1.4). If it has been allocated a new job, it fits it in optimally to its current *CourierPlan*. As jobs are allocated in a less optimal way and agents are unable to refuse jobs that cannot be delivered on time, the planning algorithm is more likely to attempt to minimise the number of late deliveries rather than cost.

6.2.3 Free-for-all

Free-for-all, as defined by Merriam-Webster is "a competition, dispute, or fight open to all comers and usually with no rules" [49]. This strategy is named such in that there is no cooperation and little, if any, communication between the agents. The result is disorder and it performs worse than the other strategies on all counts, however it does showcase the benefits of effective planning and cooperation over competition. The strategy requires only a one-way communication channel between the dispatcher and the agent. Thus, out of all of the strategies presented, it gives the agent the most autonomy. It could be used to model a competitive market of freelance couriers that have access to the same job feed, but do not share information amongst themselves.

It is implemented in a similar way to the other strategies, however the *FreeForAllStrategy* has access to the *NoticeBoard*'s list of unallocated jobs, which, in the real world would be realised by using a periodic wireless broadcast to all agents. From this list, and using

a *cost* and *route* function in line with the agent's $RouteFindingMinimiser$ (see 5.3.3), it selects the job with the highest 'value', which is computed like so:

Define $\mathcal{S}$ as the agent's current position, $\mathcal{T}$ as the current time, ω as the maximum waiting time per waypoint and τ as some redundancy time to account for delays. Let $r_1 = route(\mathcal{S}, job.pickup)$ and $r_2 = route(job.pickup, job.dropoff)$:

$$value(job) = \begin{cases} 0, & \text{if } job.deadline < \mathcal{T} + time(r_1) + \omega + time(r_2) + \tau. \\ \dfrac{ost(r_2)}{cost(r_1)}, & \text{otherwise.} \end{cases}$$

Using the above equation makes agents prefer long (expensive) jobs that can be picked up nearby. Having selected a job, the agent routes towards it. Often, multiple agents will compete for the same job. An agent is not allocated the job until it has arrived at the pick-up position. By default, agents do not know when this has occurred; they need to arrive at the pick-up point first and replan from there. If a boolean flag is set, this aspect of the model can be relaxed.

6.3 Idle Strategies

In this section, we discuss the strategies that an agent may execute when the agent has completed all of its allocated jobs. Such strategies could be paramount to the efficiency of the service, in certain circumstances:

- Jobs are infrequent, so agents are usually idle.
- Deadlines tend to be very slim, so placement is critical.
- The environment/map is large or otherwise slow to navigate.
- Pick-up locations are somewhat predictable.

Four implementations of $IIdleStrategy$ have been created. The first is named $NoIdleStrategy$, which does absolutely nothing when run – the agent simply rests at its last drop-off waypoint until it is awarded additional jobs. It is unique in that it has no cost overhead and is used as a control when evaluating the effectiveness of the other, more complex strategies. The other strategies are named $SleepingIdleStrategy$, $PredictiveIdleStrategy$ and $ScatterIdleStrategy$. They attempt to do one or both of the following: premature refuelling and strategic placement.

6.3.1 Premature Refuelling

Emergency refuelling, as described in section 5.6.3.2, involves postponing waypoints, thus risking late deliveries and lost revenue. It seems more reasonable to refuel when the agent would otherwise be idle, in efforts to minimise these risky fuel diversions. Some strategies make use of the utility function, $GetOptimalFuelRoute(agent, endpoint)$, if the agent's fuel supply $\mathcal{F}$ is below a certain threshold (95% full). This selects a fuel point P and returns a *Route* there, so that it can 'top up' (even if it is 90% full). P is ideally a point that requires the smallest diversion when travelling from start point $\mathcal{S}$ to some desired end point E. Specifically, the route minimises:

$$IdleStrategyCost = cost(route(\mathcal{S}, P)) + cost(route(P, E))$$

subject to

$$OptimalFuelUsage(route(\mathcal{S}, P)) < \mathcal{F}$$

The *cost* function depends on the agent's $RouteFindingMinimiser$ (see section 5.3.3). If the agent's fuel supply is low, it may not take the optimal route, but rather one where P is nearer to $\mathcal{S}$. In fact, in very rare edge cases, it is possible to be stranded. As A* searches are computationally expensive, instead of evaluating each fuel point, the function first uses LINQ to order the fuel points by orthodromic distance: $distance(\mathcal{S}, P) + distance(P, E)$. It then properly evaluates the first three in this list and picks the best of the three. If none of the three are reachable given $\mathcal{F}$, it iterates through the rest of the list, computing the optimal A* route and testing each to assess whether they are reachable given $\mathcal{F}$. This continues until a suitable P is found.

With all idle strategies, the route from $\mathcal{S}$ to P to E is aborted when a job is awarded.

6.4 Strategic Placement

Idle strategies exist to 'prepare the agent for the day ahead'. In addition to refuelling, this may include routing to an ideal position (E) on the map.

The specific idle strategies implemented determine E in different ways:

- $SleepingIdleStrategy$. After refuelling at the *nearest fuel point*, the agent does nothing until new jobs come in. Hence, agents will end up being stationed next to fuel points and depots overnight. This has a low overhead (fuel costs that are not paid for directly by customers), however agents will be in suboptimal positions when new jobs come in.

- *PredictiveIdleStrategy*. The agent stores all of the pick-up positions of past jobs it has fulfilled. When idle, it will calculate the mean position of the last 100 pick-ups and route to the node closest to that point. E is found using the $NodesGrid$ (see 5.3.1.2). This aims to reduce the time to route to the next job. Typically, agents will park in areas with many business nodes or the centre of the map if these are well dispersed. If needed, it will use $GetOptimalFuelRoute$ to refuel along the way to this centre point.

- *ScatterIdleStrategy*. This strategy requires that agents are able to know each other's positions. It attempts to spread the agents out across the map by, for each agent, routing to the point that maximises the distance between all neighbours and map bounds. Trading off accuracy for reduced computation time, it uses the Pythagorean theorem instead of the Haversine formula for the distance function. For each connected node, it computes the minimum of: the distance to any edge of the map and the distances to other agents. It finds the node with the highest minimum distance. If needed, it will use $GetOptimalFuelRoute$ to refuel along the way to this most isolated node, E. It has the highest cost overhead in practice, because this node is often on the other side of the map.

6.4.1 Unimplemented Strategies

Two other strategies are proposed, however they are impractical to implement given the data at hand. These best suit the operation of a city courier that is expected to be idle overnight.

Parking in the city centre is often prohibited, expensive or unavailable. One solution would be for vehicles to slowly drive around aimlessly or encircling their optimal position to prevent incurring parking charges. Were parking charges modelled, this may be more economical, despite the unnecessary fuel consumption. Some roads have minimum speed limits, which would need to be avoided, such as 10 mph on the Dartford Crossing in the Greater London map used [50]. In reality, local councils are likely to object to this, as slow-moving vehicles can be disruptive and dangerous. Hence, implementing this would be a waste of time.

A more efficient strategy would be for vehicles to set route to the nearest area that offers free parking (for example, residential areas). Unfortunately, this strategy cannot be simulated properly as OpenStreetMap contributors have provided very little data on whether highways have parking. As of February 2015, The key `parking:lane` has less than 100,000 instances world-wide and almost none in Greater London [51].

Chapter 7

Evaluation

In this chapter, we critically assess the algorithms developed for the autonomous courier network, compare the performance of the many different strategies developed in action and explore to what extent varying the difficulty of the job stream impacts performance. We then proceed to determine the optimal sizes of the network for different cities and with that, produce an achievable pricing model.

These simulations use a number of assumptions. The parameters used are those named earlier in chapter 5 as 'the default values', which for the reader's convenience have been summarised in appendix A. Where a certain parameter has been changed for experimentation, this is clearly noted. Where computation time is assessed, note that the PC used to run these tests has an AMD Phenom II X4 955 3.2 GHz processor and 6 gigabytes of RAM.

7.1 Optimality of Algorithms

A hierarchy of three algorithms is employed to bring about efficient fulfilment of courier jobs. The first is route finding using the A* algorithm. The second is local optimisation and planning (given a set of waypoints, find the optimal route to visit all of them). The third is allocation of jobs to agents, to which we propose the contract net protocol as the most fit solution. Optimality-wise, the third is dependent on the results of the second, which is dependent on the results of the first. However, as time is a scarce resource, especially when one wants to run month-long simulations, each algorithm can be configured to execute faster, in exchange for some loss in optimality.

7.1.1 Route Finding

As mentioned in section 5.3.3, a standard A* search has been implemented. However, using an admissible heuristic can lead to very slow searches, wherein too much of the search space is explored. Using the concept of bounded relaxation, a value of ε is used to boost the effect of the heuristic when scoring newly discovered nodes. Here we evaluate how this will affect the optimality of searches.

A benchmark tool is built into the simulator program, which is used to provide the user with some guidance on which value of ε (or rather, which value of $(1 + \varepsilon)$) to use for a particular map and $RouteFindingMinimiser$. The benchmark is conducted as follows:

1. Generate 1000 pairs of random points on the map.
2. Using the given $RouteFindingMinimiser$, find the most optimal route ($\varepsilon = 0$) between these pairs and record the cost according to the $RouteFindingMinimiser$ (in kilometres, seconds or litres of petrol, were a car to drive the route). Also record the execution time.
3. For each heuristic weight $(1 + \varepsilon)$ from a selection of values ranging from 1.05 to 20, find the most optimal route and then the cost. For each cost, divide it by the optimal cost found in step 2. Name this ratio r. $r \geq 1$. Also record the execution time.
4. For each heuristic weight, find the mean r over all routes and the mean execution time over all routes.
5. Show this as a table to the user.

For each map used in this evaluation and then for each $RouteFindingMinimiser$, this test was performed and the results are plotted on the graphs in figure 7.1. Images of the maps are given in the appendix – see section D.3.

It is immediately obvious that increasing the ε value has a real effect on the optimality of the routes found. For minimising distance, this rise happens quite early. If one can permit a route that is about 5% less optimal, for most maps, the value of ε could not be increased far past 0.4, which in theory would never lead to a route that is 40% more costly. With regular maps, this is far from the typical case and this sacrifice brings a speed improvement of between 2–17x. Note that the scale on the y-axis, representing execution time is logarithmic. Larger maps with a dense, non-uniform road layout (i.e. Greater London, but not Las Vegas) exhibit the best speed improvements.

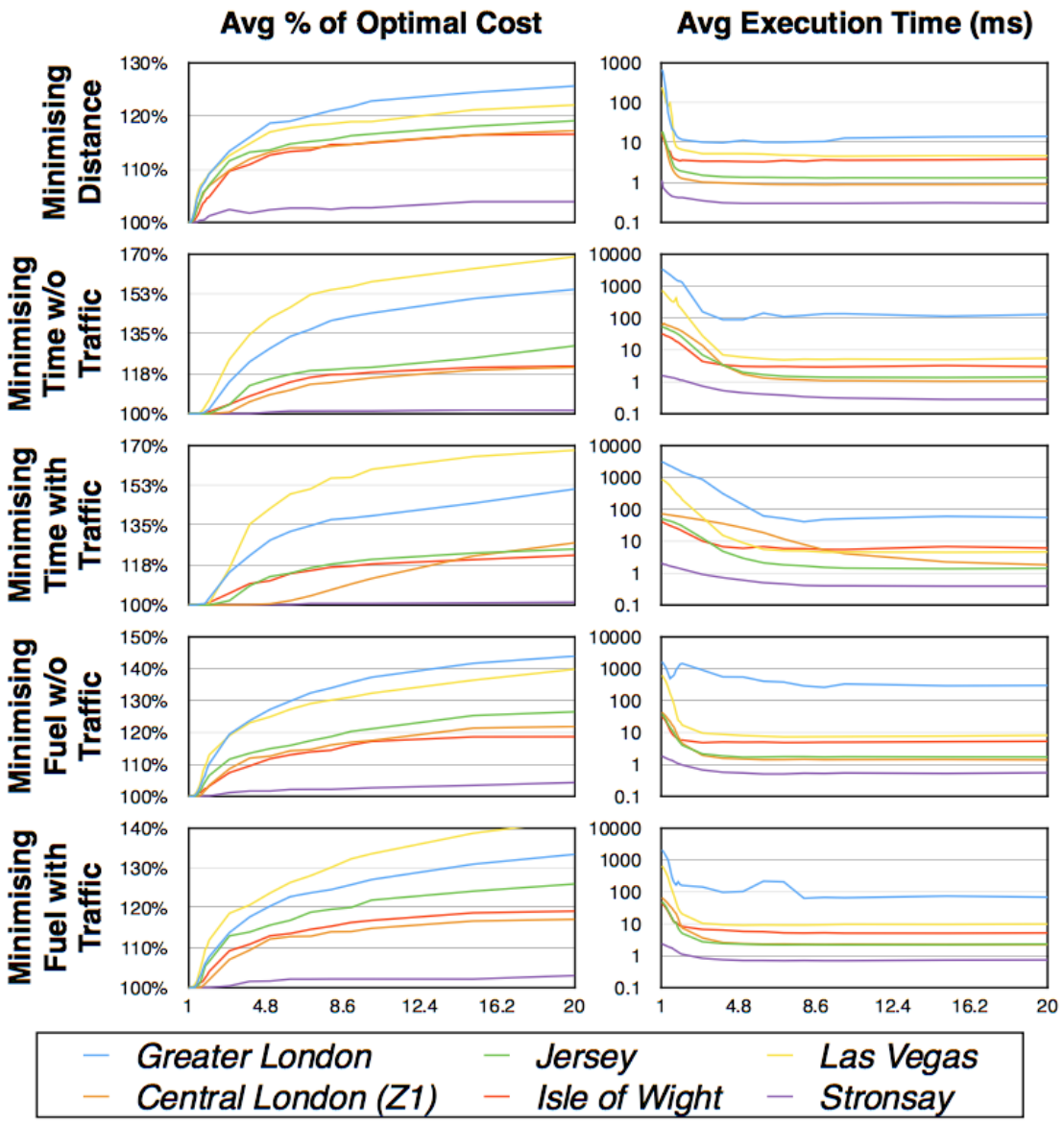


FIGURE 7.1: The effect on optimality and execution time for using bounded relaxation in A* route finding. The effects are plotted for six different maps.

Minimising time, with or without traffic, is the worst, both for optimality for high values of ε and for execution time. The average execution time for an admissible search in Greater London was 3.082 seconds, which is too high to run a simulation where thousands of routes are calculated per day. It is high as road speeds are comparatively slow, which causes the search to be very undirected, almost like Dijkstra's algorithm. In some cases, the majority of nodes in the map are expanded and given the map has 706,117 connected nodes, the long runtime is unsurprising. This time can be reduced to 41 milliseconds if an epsilon value of 7 is used; however this comes at the expense of having routes that are on average, 37% longer in time. This is because of the simple fact that travelling between two randomly picked points in Greater London often can be done much faster by using the M25 motorway, which surrounds the county, rather

than taking a route that is more or less a straight line. Earlier, in figure 5.16, there is an illustration of this. In this evaluation, we actually do accept and test using this trade-off, because long simulations would be completely infeasible otherwise. A totally admissible search could be used if the algorithms needed to run in real-time. However, this evaluation chapter required hundreds of lengthy simulations with tens of thousands of A* searches performed in each.

In any event, informal testing in Greater London simulations showed that increasing ε from 0 to 7 does not make a substantial difference to the refusal rate of jobs, nor the total driving cost – only about a 5% decline to both. The reason is most likely due to the fact that long routes are rarely queried, except in off-peak times where agents will rarely have more than one job to carry out. As jobs build up and the average distance between waypoints decreases, the fastest route tends to be through slow city streets. Also, most B2B deliveries are in Central London anyway and those that are not are good candidates for rejection. If businesses were mostly located around the edges of the map, this compromise would be less wise. The M25 could massively reduce journey times, so long as the A* search is admissible, thus permitting routes that are fast, but geographically divergent.

The runtime issue is less of a problem with the Las Vegas Valley map, however the route cost rises much quicker and much higher than Greater London, because of the many main roads that run straight through and around the city. Travelling between two randomly picked points in Las Vegas will very often benefit from interstate 15 and the main roads connected to it. If these are missed, the likely result will be travelling through city blocks, which are a lot slower and have traffic lights at every intersection. The heuristic cost, when amplified even using a small value of $\varepsilon = 3$ gives routes that are typically 35% longer. This reduces computation time from 868 to 15 milliseconds. In testing, ε was set to 2 for Las Vegas, as this yielded routes only 16% suboptimal.

Jersey, Central London and Stronsay are smaller maps, whilst the Isle Of Wight has far fewer roads. For these reasons, execution time was less of an issue and so very little optimality was traded off for speed. ε was set to the maximum value to which routes in these maps would be no worse than 5% suboptimal on average. It is very apparent that execution time scales with map size and node density.

Minimising fuel with or without traffic is less difficult, as it is mostly dependent on distance travelled rather than speed. Unfortunately, it would not make sense to minimise anything other than time without traffic for this business, as deadlines are expected to be quite tight. It would be more profitable to fulfil more jobs than to fulfil fewer at a lower fuel cost, especially as having more waypoints reduces the average amount of driving per job. This is discussed further when we evaluate dispatch rate in 7.4.

For Stronsay, optimality does not suffer much, nor does execution time improve significantly. This is because the road map of the island is almost like a spanning tree, with only a few cycles. This means there are very few routes available.

7.1.2 Planning

With a near-optimal path finding algorithm at hand, the next search task is to find a sequence of routes that connect up an agent's allocated waypoints, in a valid, maximally on-time and minimally costly way. For large sets of waypoints, it is difficult to fairly evaluate the greedy and genetic algorithms developed to solve this problem. This is because finding the actual optimal solution for n waypoints would require testing $n!$ permutations. However, a partial evaluation for smaller problem sizes is provided and the test is conducted like so:

1. A small planning problem consists of 4 randomly generated jobs at 10 a.m. in the Central London map. Generate 2500 such problems where there exist no deadlines and 2500 such where the deadline excesses are sampled from a relatively difficult $\Gamma(2, 0.5)$ distribution[1].

2. For each problem, find the actual optimum through brute-force testing of every single solution. Then find a solution using the greedy planner and the genetic planner. Make note of occasions where the greedy planner failed to find an on-time solution when one existed. Where both solutions were found and on-time, divide each solution's cost by the cost of the optimal solution to get a percentage – for example, 120% if the solution was 120 minutes and the optimal was 100 minutes. For the genetic solution, make note of the number of late deliveries compared to that in the optimal.

3. Having produced a table of costs, the mean cost degradations can be found and the results plotted to scatter graphs.

The results of the test without deadlines are plotted in figure 7.2. We can also report the average degradation of optimality over the 2500 problems. The greedy planner produced solutions that were on average 143.7% the cost of the optimal, which is more than the genetic algorithm, whose average was 134.1%. If the simulation were to run both algorithms in parallel and pick the best of the two, the average would be 133.2%. Only in 2.0% of cases did it find the actual optimum, however given there exist 40320 solutions, this is an unrealistic expectation. The figure also shows a scatter plot of the

[1] With $\Gamma(2, 0.5)$, the mean deadline excess is only one hour and the mode, 30 minutes.

scores from the greedy versus the genetic solutions. Their close similarity is because the lack of deadlines mean the nearest neighbour search does not perform any backtracking and its output will exactly match that of the deadline-invariant greedy solution, which is used as the seed in the genetic algorithm. Often the seed solution or a very closely related solution will be selected as the most fit. Nonetheless, it is clear that the genetic algorithm usually outperforms the search. The greedy search was best 17.2% of the time; the genetic algorithm was best 66.2% of the time; and on 16.6% of occasions, they output the same solution.

The results of the test with deadlines are summarised in table 7.1. It shows that in most cases, if the optimal solution contains k late deliveries, the solution found by the genetic algorithm will too have k late deliveries. Though there were some extreme cases, such as the planner giving a solution with three late waypoints when an on-time one existed, these were a rarity. It should be noted that in this test, the greedy algorithm succeeded in finding an on-time solution (either as its first solution or by way of its backtracking), in 7.0% of cases where one actually did exist. Out of these 148 cases, in 73 (49.3%) of them, the solution was less costly than that found by the genetic algorithm. In most simulations, this figure is likely to be much higher, because the problems are less demanding, deadline-wise.

Genetic Planner Score	4000–5000	0	0	0	0	0
	3000–4000	14	7	3	0	
	2000–3000	107	103	12		
	1000–2000	715	262			
	0–1000	1277				
		0–1000	1000–2000	2000–3000	3000–4000	4000–5000
		Actual Optimum Score				

TABLE 7.1: For 2500 test problems, this shows the score of the output of the genetic planner, as compared to the optimum route. The number of 1000s in a score correspond to the number of late deliveries.

These results at first glance give a bad impression of these algorithms. One would think that increased costs of 33% imply that the algorithms are not very effective.

Although many, many solutions do exist for any particular planning problem, there only exist a tiny number with the minimum number of late deliveries. Finding such a solution can be colloquially compared to 'finding a needle in a haystack'. We attempt to illustrate this in figure 7.3, where for one typical problem, we enumerate and plot the cost of every single valid permutation. This figure shows that only 0.5% of solutions were on-time. The genetic algorithm found an on-time solution that was 133.7% the cost of the optimal. Though this figure appears to be overly costly, it ranked 79th out of 40320. It was a local minimum and was quite distant to the global minimum, in terms

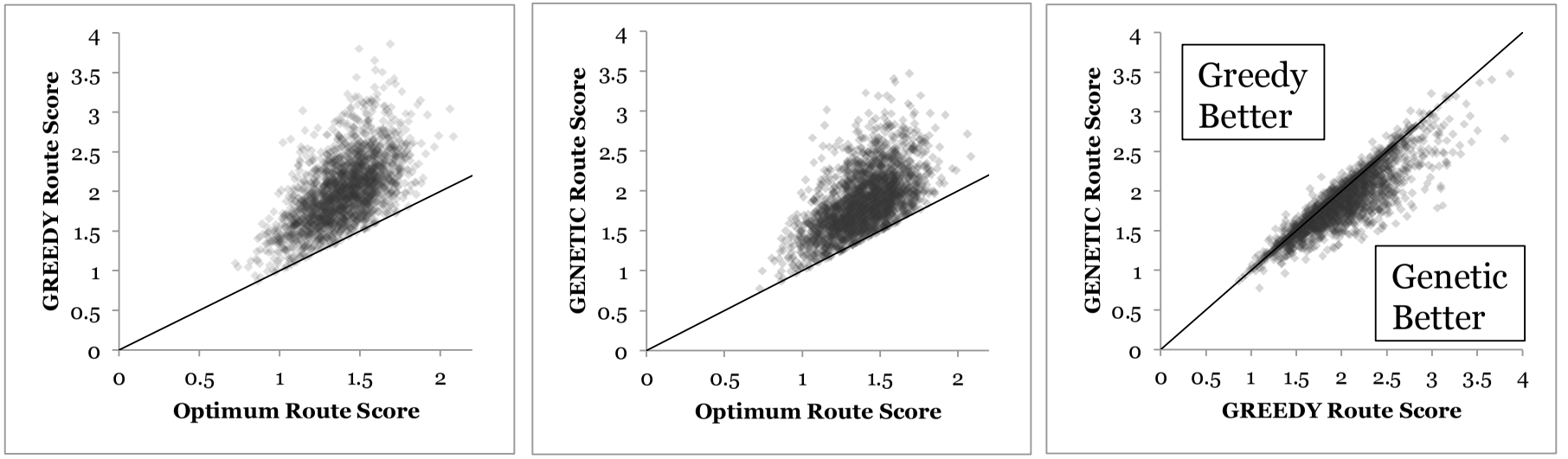
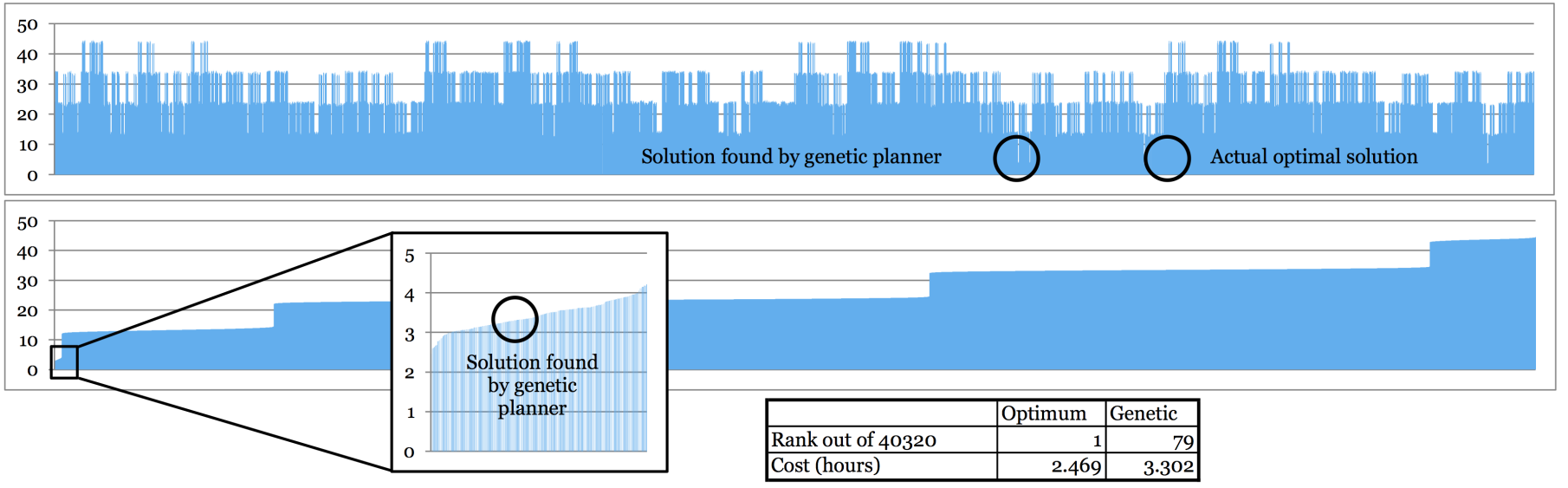


FIGURE 7.2: For 2500 small problems, the solutions of the greedy and genetic planner are compared against the optimal solution as well as each other.



	Optimum	Genetic
Rank out of 40320	1	79
Cost (hours)	2.469	3.302

FIGURE 7.3: All solutions of a typical small planning problem, as they appear in sequence and once sorted by their score. The score penalty for lateness is 10. A solution scoring 32.5 has 3 late deliveries and is 2h30 long. The solution found by the genetic algorithm is labelled.

of mutations. This figure may also persuade the reader that a hill-climbing algorithm would not be appropriate to solve this problem, as local minima are spread rather evenly.

7.1.3 Allocation

Evaluating the optimality of the contract net protocol for task sharing is equivalent to a utilitarian evaluating the use of an auction as a method of dispensing resources to people with a finite amount of money. For individual auctions, the method is guaranteed to allocate a resource or a task to the most appropriate bidder, merely by the definition of what it means to bid the highest (or the lowest for CNP). However, for *a sequence of auctions*, wherein at no time do the bidders know how many auctions are left, this is not guaranteed to give an optimal allocation. There does not appear to be a better method of static allocation in this problem domain and as one can see in the next section (7.2), CNP undeniably outperforms the 'fairer' round-robin strategy in practice.

One method of evaluating CNP would be to compare it to the global optimum – calculating the cost of every possible allocation. However it is unclear when one would do this, as jobs do not arrive in batches, but rather in an uneven stream. Furthermore, optimality will not degrade until agents become saturated, which would require too many jobs to make a brute force evaluation feasible. In the related work studied (see section 2.5), some authors used CNP, however they do not offer a method of evaluation. Therefore, in this report, a qualitative evaluation is provided to convince the reader that CNP is a reasonably efficient allocation strategy. Figure 7.4 gives a typical scenario of when CNP best allocates two jobs. Figure 7.5 gives a scenario where the two jobs could have been allocated better. It is claimed that the first scenario is more common than the second, as the second requires agents' schedules to be reasonably saturated.

7.2 Routing and Job Allocation Strategies

Several strategies have been developed and are described in detail in section 6.2. The prospective courier company wants to know which strategy they should use in their courier network. In this series of tests, we hope to determine which is the best and by what margin. A series of experiments were conducted in different environments and with different parameters. This is so that the full range of behaviours and responses can be shown.

At this point in time, we will not measure performance in terms of cost and revenue, as such things are arbitrary. Instead, we use raw variables from the simulation. A 'good' strategy is one that maximises:

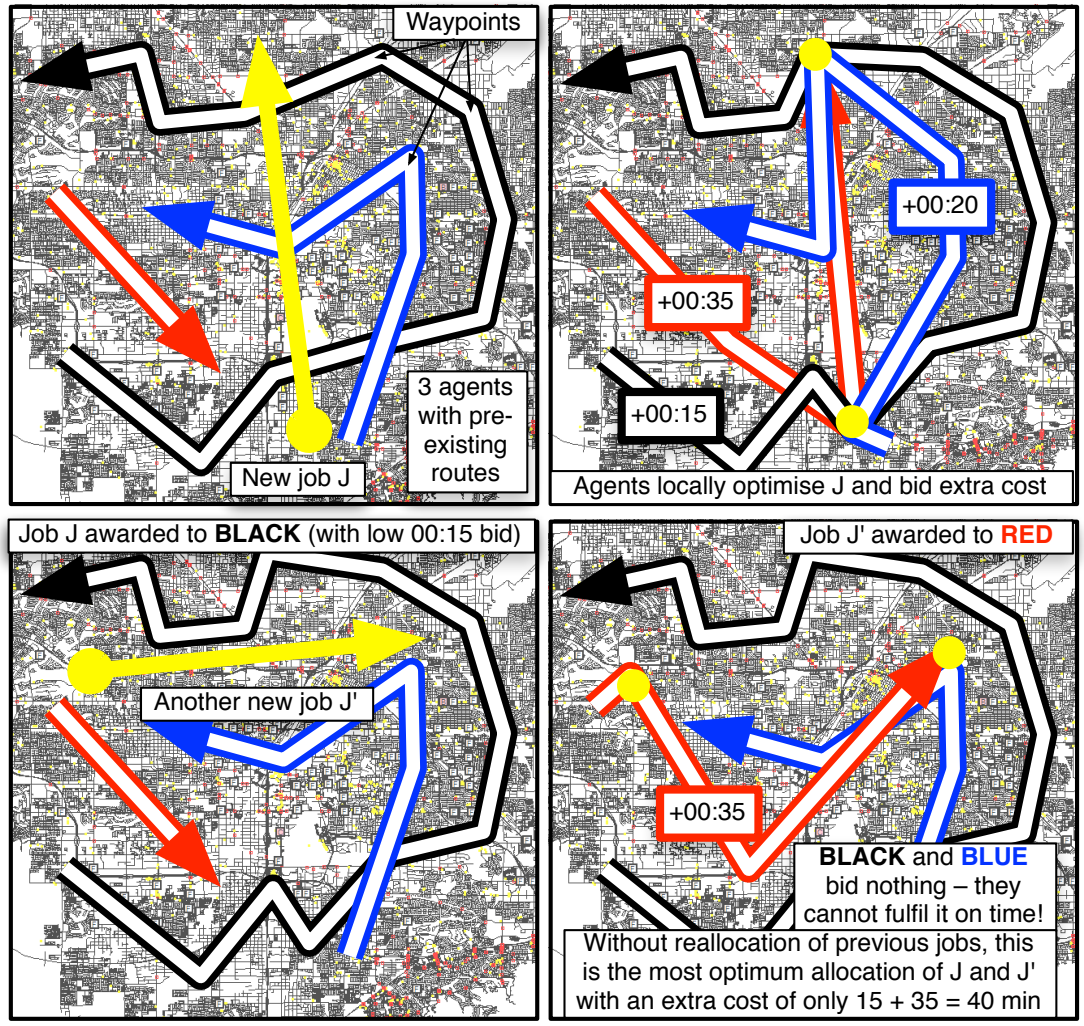


FIGURE 7.4: An example of CNP optimally allocating two jobs, J and J', that arrive in sequence. The figure is to be read left-to-right, top-down.

- Availability. It completes as many jobs as possible and refuses as few jobs as possible.
- Reliability. Of those jobs completed, very few or none of them arrive late.
- Efficiency. The amount of driving is minimised, leading to reduced operating costs for the business.

To measure these aspects, the variables of interest are: the number of completed, refused and late jobs, as well as the total distance covered, the amount of time driving and the fuel usage. The following simulations all took place over the course of one week. The predictive idle strategy was used.

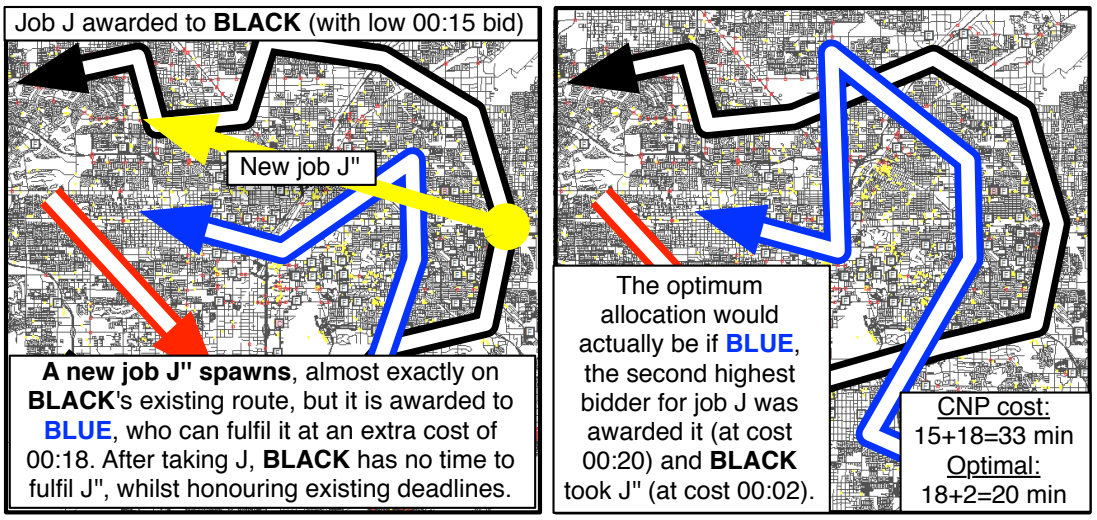


FIGURE 7.5: Extending from the top two squares in the previous example in figure 7.4, this is an example where CNP did not result in the best allocation, because the second job that arrives, J'', would have been far more suited to the now saturated agent that took the first job, J. The actual optimum is displayed in the second panel.

7.2.1 Efficiency Challenge

In this experiment, five cars were used in Central London (Zone 1). Given the map is 39.9 km^2, this gives an agent to area ratio of 8.0. This is reasonably low, so strategies should not struggle to allocate most jobs, however they will vary in efficiency.

Strategy	Job Counts			Cost (whilst driving)		
	Completed	Late	Refused	Distance (km)	Time (h)	Fuel (l)
Free-for-all	1199	0	323	18455	509	1924
Round-robin	1532	179	0	15409	431	1620
CNP1	1089	0	428	14247	395	1489
CNP2	1281	0	242	14341	401	1509
CNP3	1344	3	181	13948	391	1469
CNP4	1473	8	58	12186	344	1290
CNP5	1477	5	54	12116	341	1280

TABLE 7.2: Results of the efficiency challenge.

7.2.2 Reliability Challenge

The same parameters and Central London map as in the efficiency challenge are used, however now we alter the simulation such that failed deliveries must go back to the pick-up position, the packages are larger (an Exp(1.5) distribution is used instead of Exp(3) – see 5.5.1) and the probability of failed deliveries is 50%, not 10%. These parameters are designed to bring about a lot of late deliveries, as agents will be forced to make long, unplanned diversions. A good strategy should nonetheless minimise the late job count.

Strategy	Job Counts			Cost (whilst driving)		
	Completed	Late	Refused	Distance (km)	Time (h)	Fuel (l)
Free-for-all	1053	0	421	18565	512	1939
Round-robin	1474	219	0	17628	490	1850
CNP1	948	0	526	14600	406	1534
CNP2	1099	4	375	15294	427	1613
CNP3	1159	31	315	15416	429	1618
CNP4	1267	37	207	14682	410	1547
CNP5	1273	24	201	14563	408	1537

TABLE 7.3: Results of the reliability challenge.

7.2.3 Availability Challenge

This experiment takes place over the entirety of Greater London. Only 10 cars are employed and given the map is 3221.36 km^2, the agent to area ratio is 322.1. This is very high given the dispatch rate, so it is expected that even the best strategies will have a high number of refused jobs.

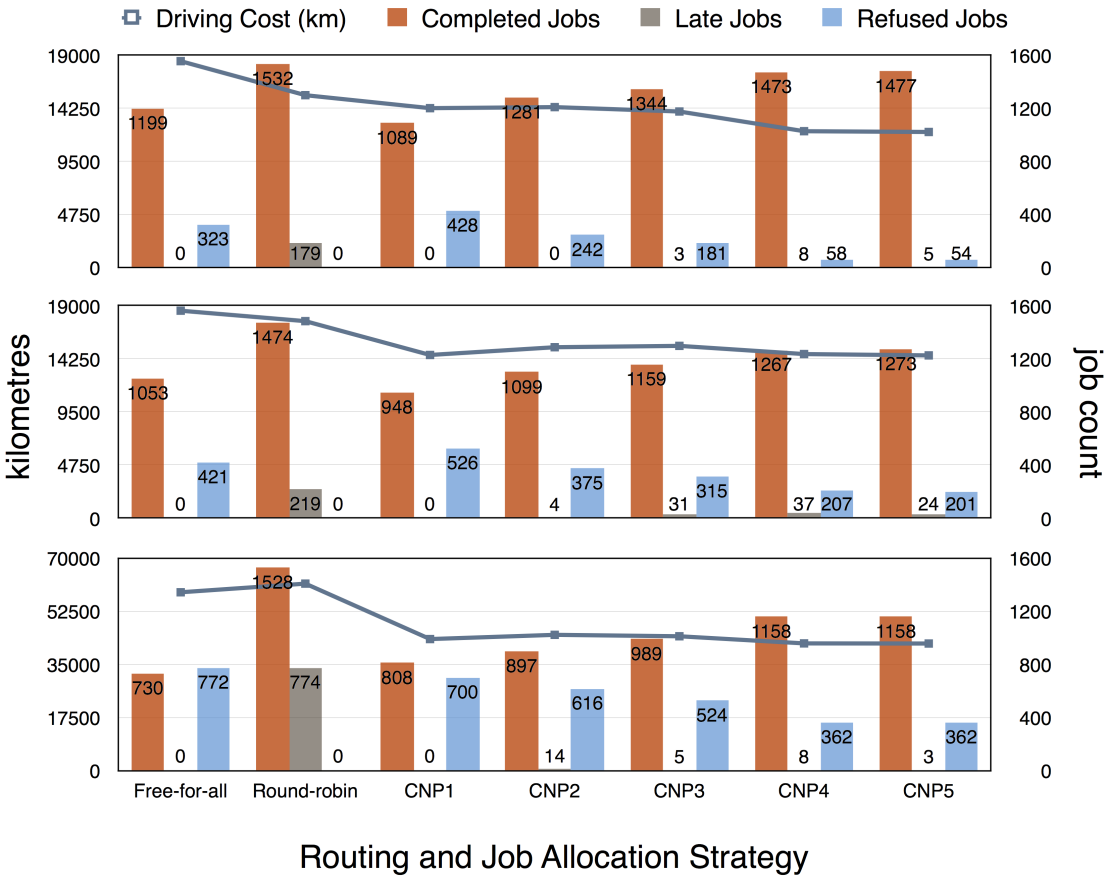
Strategy	Job Counts			Cost (whilst driving)		
	Completed	Late	Refused	Distance (km)	Time (h)	Fuel (l)
Free-for-all	730	0	772	58687	1393	5578
Round-robin	1528	774	0	61543	1440	5798
CNP1	808	0	700	43259	1053	4168
CNP2	897	14	616	44656	1076	4279
CNP3	989	5	524	44184	1057	4226
CNP4	1158	8	362	41833	984	3963
CNP5	1158	3	362	41791	984	3964

TABLE 7.4: Results of the availability challenge.

7.2.4 Summary

These tests were designed to stress test the strategies in different scenarios: to minimise cost in desirable conditions, to minimise the number of late deliveries in tough conditions and accepting the largest number of jobs despite being outnumbered. The first observation one can make from the graphs is that the free-for-all and round-robin strategies exhibit very different behaviour to the variants of the contract net protocol.

Round-robin allocates jobs to agents without checking that they are capable of delivering them on time. Hence, there are no refused jobs, but a very high number of late jobs – especially so in the Greater London test, where waypoints are more scattered around the map. Given that the courier service being proposed refunds customers whose deliveries are late, this strategy is not suitable. However, with the top, efficiency test, one can observe that the number of on time deliveries ($completed - late$) is beaten only by



CNP4 and CNP5. Round-robin uses the standard *NNGAPlanner* and can carry several packages at once. Since it outperforms CNP3 in this respect, this flexibility seems to outweigh the benefits of the contract net protocol, at least for easy environments.

Free-for-all is similar to CNP1 in that it only carries one package at any one time. FFA is the worst performing amongst them all when comparing the driving cost per completed job. This is because many agents will often route to the same pick-up points simultaneously, thus wasting resources. The bigger the map, the greater the waste. As FFA and CNP1 only attempt jobs that can be done on time, neither delivered any jobs late. However, the refused job count varies. For Central London, it was about 20-24% lower for FFA than CNP1. This is most likely because CNP1 immediately refuses jobs when no agents are available. FFA allocates in a much different way: the broadcaster keeps rebroadcasting jobs that are available. They only become 'refused' when their deadline expires. For greater London, the wasted time outweighs this factor – fewer jobs are even considered, because agents are too often busy routing to jobs that they will ultimately not reach in time.

The contract net protocol, however, is the focus of this project and our intuition tells us that it will bring the best performance. The CNP variants, numbered 1–5 are ordered by their level of flexibility, with CNP5 allowing agents to carry out many jobs at once and

reallocate jobs to other agents when in trouble. This last factor is the only difference between CNP4 and CNP5 and due to the redundancy time built into the planning algorithm, reallocations were rare. However, the effects can still be seen, because of the length of the simulation. In all three tests, CNP5 achieved a lower rate of late jobs. Even in the second test (where half of the jobs had to be returned to the sender and the packages themselves were usually too big to allow this detour to be postponed), CNP5 delivered only 1.8% late. In tests 1 and 3, this rate was 0.3%. CNPs 2 and 3 had varied rates compared to 4 and 5.

If fulfilling multiple jobs at once cannot be allowed by the business for whatever reason, CNP3 appears to be preferable over CNP2, as the late job counts for both are relatively low, but CNP3 is able to fit in more jobs into its schedule. This comes at a cost in the second test, as a longer route increases the chance of any one job to be delivered late. Comparing cost in the efficiency test, we see that for CNP 1–3, the total driving cost was more or less the same. CNP5 marginally beats CNP4, which substantially beats CNP1–3. For the other tests, all variants are more or less the same. However, the efficiency gains are very apparent when one compares this to the completed jobs count, which rises steadily over each CNP variant. During the development of CNP4 and CNP5, it quickly became apparent that it is almost impossible to have zero refused jobs. The probability distribution used allows jobs to be spawned with deadlines so tight that an agent needs to be stationed very close to the pick-up point, in order for it to confidently take the job.

To conclude, whether availability, efficiency or reliability is the priority, CNP4 and CNP5 outperform the other strategies. If the communication model cannot allow job reallocation, CNP4 still performs well in all areas. CNP5 is used for the remainder of this chapter.

7.3 Idle Strategies

Idle strategies are particularly important where the map is difficult to navigate, when pick-up locations are predictable and when refuelling is a concern. It is difficult to design a test that will effectively assess these, as it is accepted that in a normal environment, the idle strategy is unlikely to make a significant difference. In the following tests, the simulation has been altered such that:

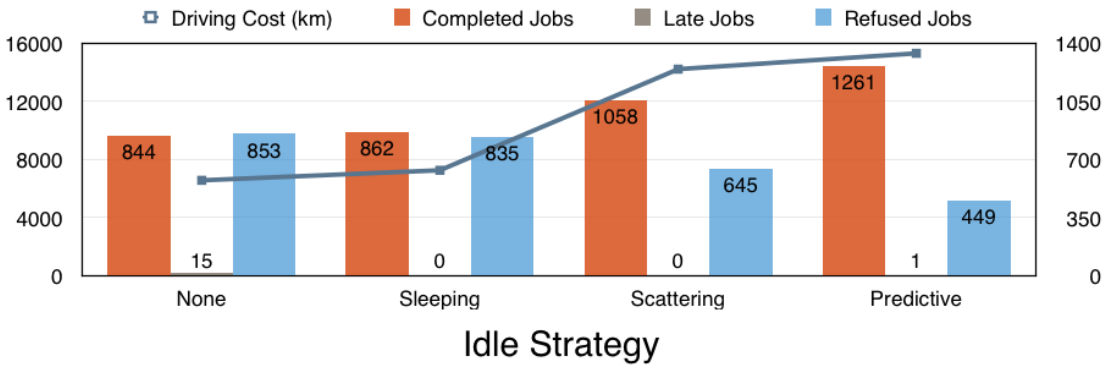
- Jobs are dispatched less often. The dispatch rate coefficient is 0.1. This means at peak times, only two jobs are dispatched per hour.

- The deadline excess will be exactly twenty minutes, instead of being sampled from a $\Gamma(2, 1)$ distribution. Hence, a job whose direct route involves an hour of driving will have a deadline of exactly 01:20. If no agent is within twenty minutes of the pick-up point, the job will be refused.
- Five trucks are used instead of cars, as they have poorer fuel economy. Their fuel tanks have been reduced from 92 litres to 10 litres, further limiting their range.
- The simulation begins with the trucks scattered randomly (not uniformly) across the map, in contrast to all starting in the main depot. Otherwise, the first set of jobs will mostly be refused.
- The Central London map is used and all road speeds have been halved to simulate especially high traffic.

A 12-week long simulation will be used to test each of the four idle strategies. The routing and allocation strategy used will be CNP5, as it performed best in the previous test. The variables of interest are the completed/refused job counts, the total driving cost and the number of emergency fuel diversions.

Idle Strategy	Job Counts			Cost (whilst driving)			Fuel Diversions
	Completed	Late	Refused	Distance (km)	Time (h)	Fuel (l)	
None	844	15	853	6546	372	2232	338
Sleeping	862	0	835	7238	417	2494	0
Scattering	1058	0	645	14208	785	4724	10
Predictive	1261	1	449	15298	864	5190	9

7.3.1 Performance



These results clearly show the trade-off that exists between minimising driving cost and optimising agent placement. The additional cost of running each idle strategy can be interpreted approximately as the difference between it and the costs of the

NoIdleStrategy, which directs the agent to do absolutely nothing when idle[2]. About 57% of the driving cost for the predictive strategy, *PredictiveIdleStrategy*, is due to the behaviour of the idle strategy. Note however that with normal dispatch rates, this overhead is comparatively small. It is high here because the transition from being active to idle occurs many times throughout the day, compared to once or twice with greater levels of demand.

The fact that the predictive strategy has the highest number of completed jobs implies that it gives the most optimal agent placement. Scattering is second best, however as it fails to account for high density areas, it performs marginally worse. Sleeping addresses the refuelling problem, however it results in all agents converging to fuel stations. If a job spawns that is far away from any fuel point, it is likely to be refused.

7.3.2 Refuelling

The idle strategies address the issue of refuelling in different ways. *NoIdleStrategy*, in efforts to minimise overhead costs, relies on the emergency refuelling protocol (detailed in 5.6.3.2), which activates when an agent's fuel tank is less than 5% full. This might normally be an acceptable strategy, however in this scenario, where deadlines are slim, such diversions are likely to cause late deliveries – 15, to be precise. *SleepingIdleStrategy* is overly cautious, but is rewarded with no fuel diversions nor late jobs. In a larger map, fuel diversions may be inevitable if a particular job is so long that it couldn't be performed on one full tank alone. The predictive and scattering idle strategies work by selecting an optimal sleeping position and then, if needed, refuelling on the way there. As the criteria for refuelling is being less than 95% full and the routes to the sleeping position will be substantially longer than the route to the nearest fuel point, this serves as a middle ground. 9–10 diversions over the course of twelve weeks is exceptionally low and as a result, it made almost no impact on meeting deadlines. Figure 7.6 shows a plot of the sum of all agents' fuel supplies over the course of the 84 days. This reinforces the previous commentary.

7.3.3 Summary

In conclusion, the idle strategy only makes a noticeable effect in scenarios that require very quick responses. Regular refuelling is a concern with agents that are busy all day. Therefore, it is advisable to do this when one does not have any jobs, rather than waiting

[2]This is not absolutely the case, as the standard parameters were used for failed deliveries and a *DepotDispatcher* probabilistically regenerates jobs that were sent to the depot. This means a better performing strategy has marginally more jobs available to fulfil.

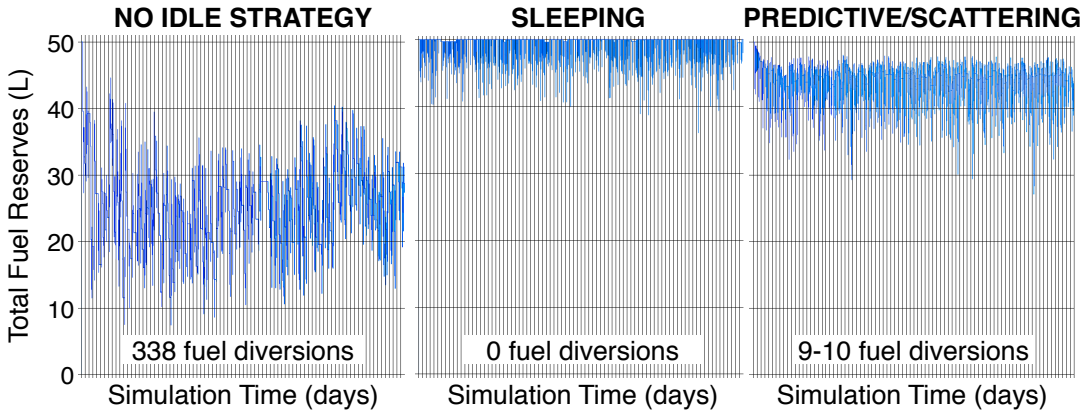


FIGURE 7.6: Fuel reserves over the course of three 84 day simulations, using different idle strategies.

until absolutely necessary. It is clear that the latter causes jobs to become late. Idle strategies that optimise agent placement will increase the total driving costs. However, if the agent transitions between being idle and active irregularly, such costs are less of a concern. Based on the results, *PredictiveIdleStrategy* is used by default for the rest of the evaluation.

7.4 Dispatch Rate

The efficiency of the service is likely to be largely dependent on the number of jobs. To evaluate this, five sets of simulations using different seeds for the dispatcher were performed, using different coefficients for the dispatch rate. As described in 5.5.1, the dispatch rate coefficient is used in conjunction with a Bernoulli distribution and an expected distribution of customer demand over the day. If set to 1.0, at peak time an average of 20 jobs are generated per hour. Set to 0.5 and an average of 10 jobs are generated per hour.

These tests were performed on the London Z1 map, using five car agents running the CNP5 and predictive strategies. As 35 simulations were run, the simulations took place on a weekday and were halted after 24 hours. Table 7.5 shows the mean results over the five runs. In the appendix, table D.1 shows the raw results of each simulation. Figure 7.7 plots these results on a graph with error bars to reflect standard error. The raw results are also shown on scatter plots.

From these results, it is very clear that average cost falls as the dispatch rate rises. This is not due to a simple economies of scale principle, although this would apply if fixed costs were modelled. This is due to the ability to construct more efficient routes.

Dispatch Rate	Completed Jobs	Completed Jobs (%)	Late Jobs (%)	Distance (km)	Distance per Job (km)
0.1	24.0	99.16%	0.00%	328.0	13.59
0.25	56.8	96.68%	0.00%	671.8	11.86
0.5	102.6	96.17%	0.17%	1077.6	10.62
1	226.6	92.81%	0.03%	1834.6	8.10
2	324.8	70.42%	1.05%	2259.4	6.97
3	349.2	50.95%	2.29%	2422.0	6.95
5	399.8	34.64%	2.77%	2711.0	6.79

TABLE 7.5: Mean results for varying the dispatch rate in a 24-hour simulation.

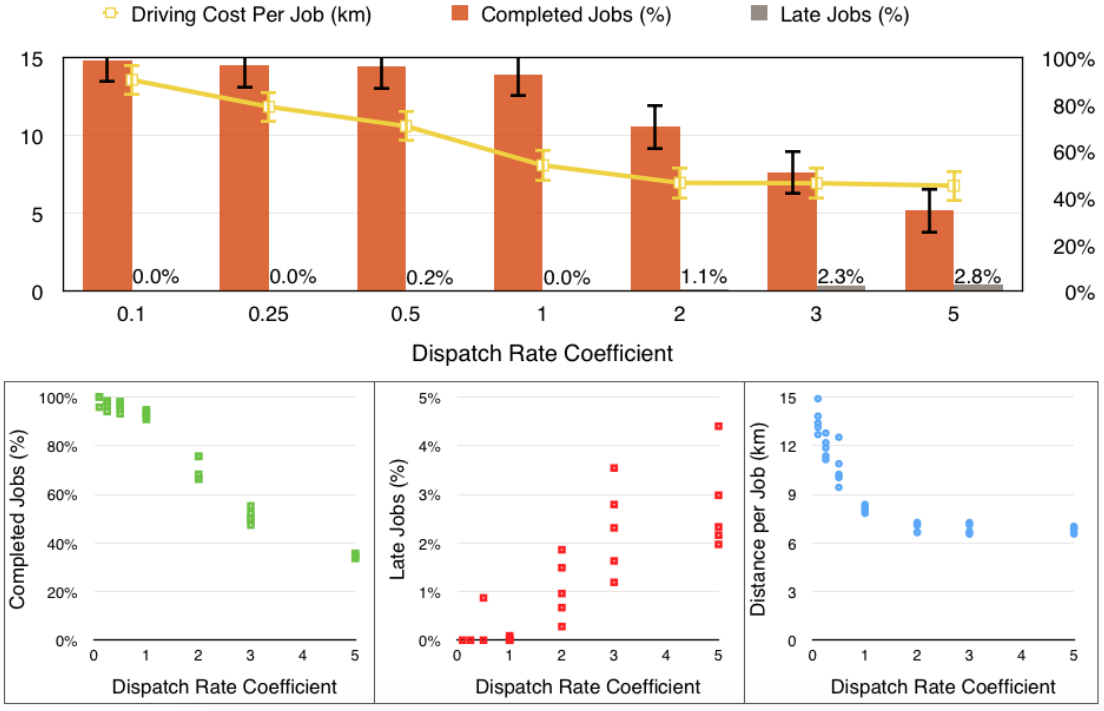


FIGURE 7.7: The effects of varying the dispatch rate on job acceptance rate, lateness and driving cost.

The average number of kilometres that an agent drives per job falls when its schedule is allowed to grow larger, making the distance between each waypoint smaller. If the dispatcher generates 10 packages from one cluster of businesses C_1, which are to be delivered to another cluster of businesses C_2, which is 10 km away from C_1, the average cost for an agent already near C_1 is only about $\frac{10km}{10} = 1.0km$. If the dispatch rate is slower and only 2 such jobs exist, each will have a cost of about $\frac{10km}{2} = 5.0km$. In reality, such a pattern of dispatches is very unlikely, but the principle applies in the same way at a smaller scale. The contract net protocol ensures that such optimisations are allowed to occur. Performance does not scale like this for non-CNP strategies, as shown earlier.

Comparing the number of completed jobs and the percentage of completed jobs (as opposed to refused jobs) shows that with increased demand, availability falls steadily, following a linear trend. It is difficult to compare simulations with small dispatch rates, as the few number of jobs mean that one particularly difficult job could skew results

negatively. Hence, the need for repeated testing. At higher dispatch rates, where more jobs are refused than are accepted, the performance in all three areas may be unfairly skewed positively, because the agents may refuse difficult jobs, whilst accepting easy ones. With the largest dispatch rate, vehicles maintained around 20–25 jobs each in their schedules during 09:00–18:00. When an agent's *CourierPlan* is at almost full capacity (i.e. it is almost impossible to insert any waypoint into its schedule without making subsequent ones late), the only available slots are those that come at the tail end of the schedule. As these schedules tend to stretch around four hours in this scenario, only those jobs with the furthest away deadlines are accepted during peak times. This back-to-back ordering of waypoints is to blame for the higher number of late jobs. Not only is the agent more susceptible to unexpected delays and diversions causing subsequent waypoints to be delivered late, the fact that its neighbours are also saturated means the CNP5 reallocation procedure fails in most cases. Hence, agents are only able to run the genetic planning algorithm in hopes to find a plan that minimises the number of late deliveries.

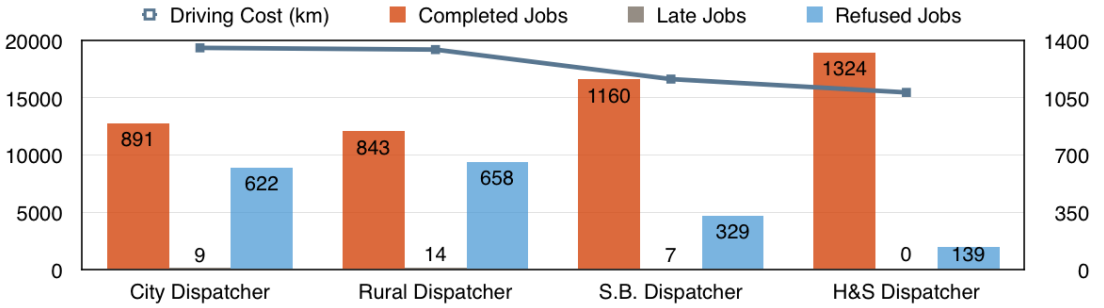
7.5 Dispatchers

The focus of this project is to design a general purpose, same-day courier service. However, the simulator has been designed to easily support the simulation of other delivery models. Several 'dispatchers' have been implemented and are all described at length in section 5.5.1. The general purpose one, used in the rest of the evaluation is named *CityDispatcher*. A *RuralDispatcher* (C2C deliveries only) is usually only used if there are very few businesses. Lastly, hub-based delivery models, which could be used by online retailers are implemented – *SingleBusinessDispatcher* and *HubAndSpokeDispatcher*. Using the same parameters and varying only the dispatcher used, a week-long simulation of three agents (cars) in the Isle Of Wight will show how the choice of pick-up and delivery locations affects performance.

Dispatcher	**Job Counts**			**Cost (whilst driving)**		
	Completed	**Late**	**Refused**	**Distance (km)**	**Time (h)**	**Fuel (l)**
CityDispatcher	891	9	622	19324	327	1548
RuralDispatcher	843	14	658	19172	333	1557
SingleBusinessDispatcher	1160	7	329	16601	293	1350
HubAndSpokeDispatcher	1324	0	139	15437	262	1230

Figure 7.8 shows the state of the simulations at peak time. This may give the reader a clue as to why performance differs so greatly in some cases.

Compare the rural dispatcher to the city dispatcher and one sees only a marginal difference in the job acceptance rate and the average driving cost per job. As the city



dispatcher mainly picks business nodes as pick-up and drop-off locations, whereas the rural dispatcher simply uses random points on the map, the distribution of pick-up and drop-off points are more scattered with the latter. For this reason, distances between waypoints are generally, slightly greater. However, one can also imagine that if the open countryside is connected by fast roads, but roads linking central business districts are very congested, the rural dispatcher might outperform the city dispatcher.

The main observation however is the difference between the general purpose dispatchers and the hub-based models. The hub based models have a higher acceptance rate, much lower driving costs and fewer late jobs (mostly due to fewer failed deliveries). This is because either the pick-up or the drop-off point generated by these dispatcher is the centrally located depot, which results in much less driving and as a result, more availability. Essentially, only one waypoint exists – that of the customer. With the hub and spoke model, this gain is further amplified, as this outer waypoint is just one out of a small set of points. Many jobs were charged only the base price, as an identical/overlapping job had already been planned, hence no additional cost. The Isle of Wight was chosen as it has an even distribution of fuel points, which we model as the spokes. If a company like Amazon wanted to offer a same-day delivery and returns service in the Isle Of Wight, these results serve as an indication of the cost savings achievable by using collection points. In some cases, the route efficiency was impeded by the capacity of the cars, so it would be preferable to use larger vehicles or to offer the service only for small items.

It should be noted that some (but certainly not all) of the efficiency gain with the hub and spoke dispatcher is due to the improbability of failed pick-ups and drop-offs, which is programmed in. As with all dispatchers, depots never fail a pick-up or a drop-off. The SBD therefore results in a comparatively low rate of failure and the H&SD has none. The negative effects of failed deliveries are evaluated later on in section 7.6.2, which shows negligible differences between a 0% and 10% failure rate (as used by default and in this experiment).

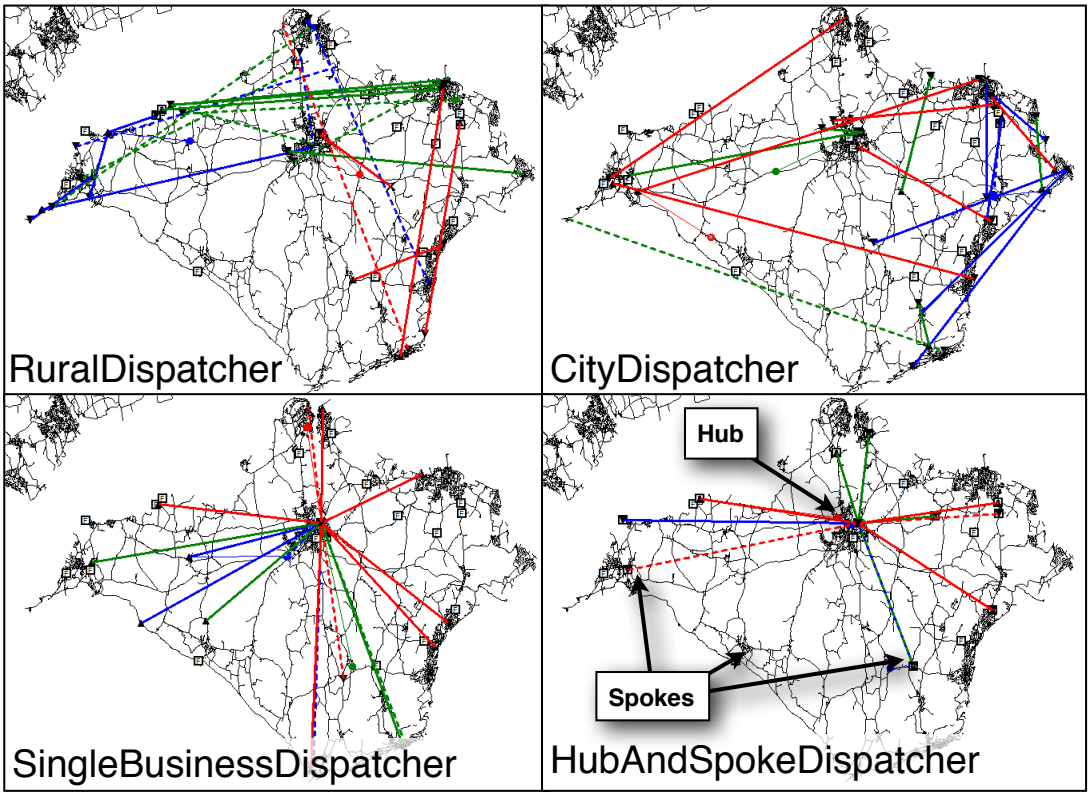


FIGURE 7.8: 'Job view' on a Monday at 10 a.m. in four Isle Of Wight simulations using different dispatchers. The spatial distribution of the pick-up and drop-off points (as marked by black triangles) impacts overall efficiency. The hub and spoke model is the most efficient of the four, as many of the jobs overlap.

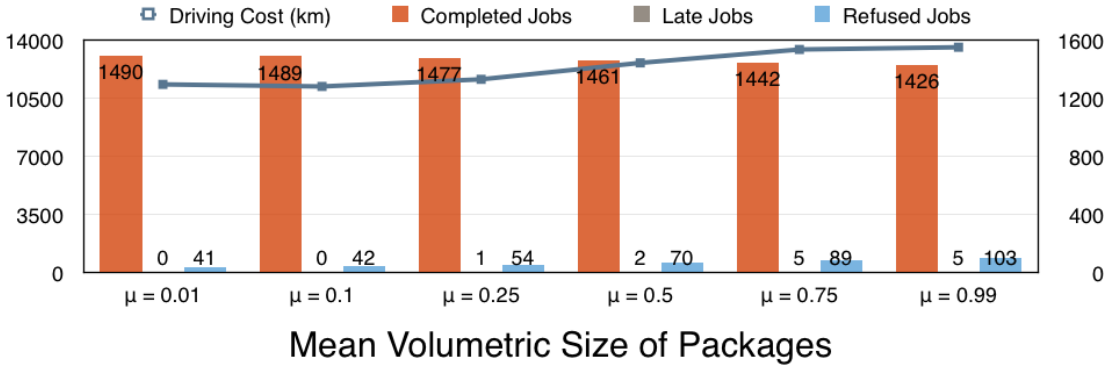
7.6 Job Difficulty

For general-purpose evaluations of the courier network, we try to use sensible values for parameters. It is useful to determine how sensitive the performance is to changing different properties of the *CourierJob*s. A volumetrically large job with a tight deadline, succeeding other jobs that are likely to require redelivery, can be deemed 'difficult'. Couriering letters with a lengthy deadline in an environment where failure never occurs is comparatively easy. An investor of the proposed courier business would need some degree of confidence in the typical properties of jobs, to trust the results of simulations. Some properties will be more important to get right than others. In this section, many simulations are run on the same configuration (Central London (Zone 1) with five car agents), but with some parameters altered.

7.6.1 Package Size

As seen by the performance of CNP4–5 in section 7.2, there are great efficiency gains to be made with the ability to bundle deliveries together. However, such an ability would be of lesser consequence if the deliveries are very large and fewer will fit in the vehicles. As explained in section 5.5.1, samples from an exponential distribution are used by the dispatcher to generate a package size, in cubic metres. This section analyses the effects of changing the λ parameter in the Exp(λ) distribution. By default, $\lambda = 3$, which makes the mean package size $\lambda = \frac{1}{3}$. No package over 0.999 m^3 will ever be generated and this test uses cars, which have a 1 m^3 capacity.

Package Sizes	**Job Counts**			**Cost (whilst driving)**		
Exp(λ), $\mu = 1/\lambda$	**Completed**	**Late**	**Refused**	**Distance (km)**	**Time (h)**	**Fuel (l)**
$\mu = 0.01$	1490	0	41	11335	320	1205
$\mu = 0.1$	1489	0	42	11210	318	1193
$\mu = 0.25$	1477	1	54	11639	330	1241
$\mu = 0.5$	1461	2	70	12629	356	1338
$\mu = 0.75$	1442	5	89	13436	377	1420
$\mu = 0.99$	1426	5	103	13571	380	1432



The results of this experiment show that larger packages do make a negative impact on performance. There was no real difference between the first two simulations. This implies that if the average package is 0.1 m^3 or less[3], capacity constraints will rarely ever impact the agent's efficiency. However there is a noticeable difference as this rises. The number of refused jobs more than doubles for very large packages. The average cost per job is shown to increase, however the range 7.61–9.52 km per job is not that great. This is surprising, as large or heavy parcels typically incur large price premiums in the delivery market.

With the large jobs, there are a handful of late deliveries. These often occur when a vehicle is carrying large items, has little capacity left and has planned pick-ups that are nearing their deadlines. If a delivery fails, the agent must replan with less flexibility. It will have to postpone many planned, unpicked jobs, as they wouldn't fit. Necessary

[3]The reader may prefer to instead picture a cube that's length, width and height are less than 46 cm.

diversions to the depot cause some jobs to be delivered late. As only 10% of deliveries are expected to fail, there were only five late jobs.

7.6.2 Failed Deliveries

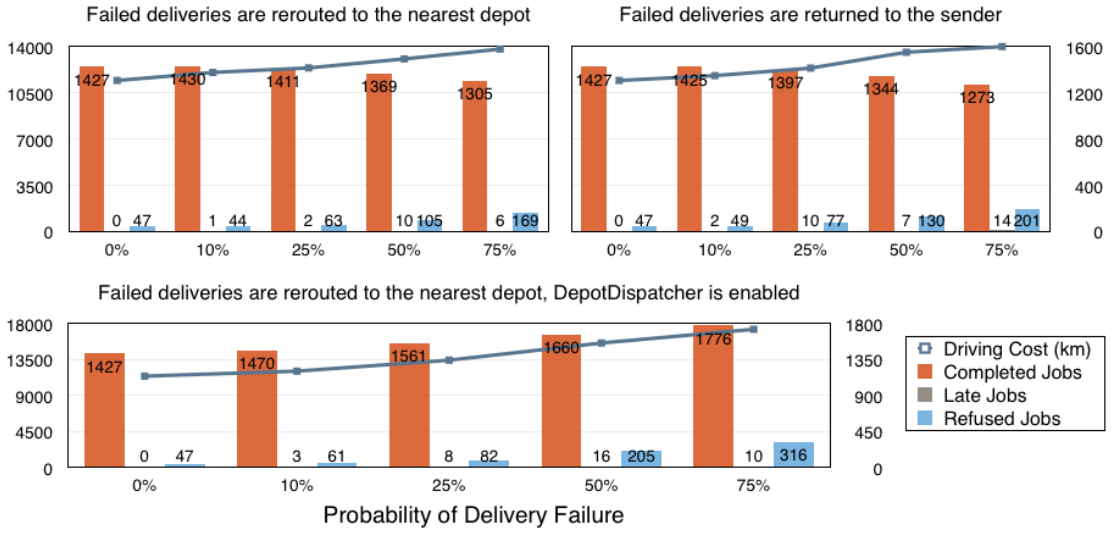
The probability of failed deliveries is expected to have a huge impact on performance, but the default value of 10% is very arbitrary and not based on any official statistics. For this reason, it is crucial to know the extent to which this impacts performance. By default, failed deliveries are rerouted to the nearest depot and the *DepotDispatcher*, which models half of customers replacing their order, is usually enabled (see section 5.5.3). In the simulation statistics, a regenerated order counts as an additional 'completed job'. This can be misleading when assessing performance – the number of completed jobs rises with more failure, merely for the fact they are so often regenerated. Two sets of simulations were run with *DepotDispatcher* enabled and disabled. Another option is that the packages are returned to the sender. The results of these three sets of simulations are detailed in the tables and graphs below.

Pr(Failed Delivery) to nearest depot	**Job Counts (NO reorders)**			**Cost (whilst driving)**		
	Completed	**Late**	**Refused**	**Distance (km)**	**Time (h)**	**Fuel (l)**
0%	1427	0	47	11430	324	1216
10%	1430	1	44	12037	339	1277
25%	1411	2	63	12379	349	1313
50%	1369	10	105	13060	367	1381
75%	1305	6	169	13803	385	1453

Pr(Failed Delivery) to pick-up position	**Job Counts (NO reorders)**			**Cost (whilst driving)**		
	Completed	**Late**	**Refused**	**Distance (km)**	**Time (h)**	**Fuel (l)**
0%	1427	0	47	11430	324	1216
10%	1425	2	49	11797	335	1257
25%	1397	10	77	12365	350	1316
50%	1344	7	130	13565	382	1437
75%	1273	14	201	13988	394	1482

Pr(Failed Delivery) to nearest depot	**Job Counts (incl. reorders)**			**Cost (whilst driving)**		
	Completed	**Late**	**Refused**	**Distance (km)**	**Time (h)**	**Fuel (l)**
0%	1427	0	47	11430	324	1216
10%	1470	3	61	12056	339	1277
25%	1561	8	82	13432	375	1413
50%	1660	16	205	15580	431	1630
75%	1776	10	316	17297	475	1797

Fulfilling a job that ultimately fails essentially consists of visiting three waypoints, with the third being the depot or the pick-up position, revisited. With this in mind, it is hardly surprising that the amount of driving cost rises as more deliveries fail. As fulfilling these extra waypoints takes up valuable time, more jobs must be refused because they cannot be fulfilled by any of the agents on time.



It is curious that between 25% and 75%, the number of late deliveries for each simulation is inconsistent. One would expect them to rise, as unexpected diversions to depots and pick-up points often must postpone other deliveries, making them late. However, this trend is not observed. It is suggested that the addition of these extra waypoints, plus the resulting change in the jobs that are allocated (rather than refused), causes the workload to be significantly different between these simulations. Diversions may, in some fortunate cases, lead the agent into suboptimal positions that prevent them taking on certain risky jobs with slim deadlines.

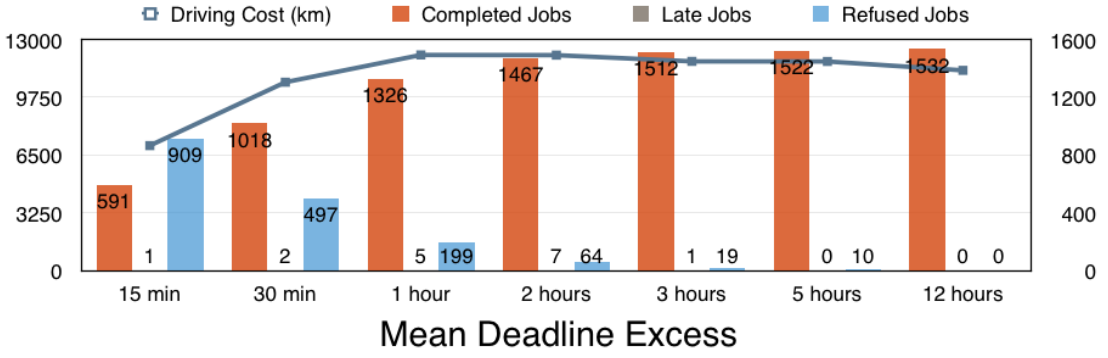
Lastly, compare the difference in driving cost between the policy of 'return to depot' versus 'return to sender'. This, of course, will be very much dependent on the distribution of depots in the map and how far away the sender usually is from the recipient. With the two depots modelled in the Central London map, performance did not vary significantly and of course, not at all with a 0% failure rate. If there were more depots scattered around the city, the 'return to depot' option would surely be more efficient.

7.6.3 Deadlines

The focus of this project is to model a courier service that primarily deals with same-day deliveries, but not those that require an instant response. However, many customers would expect something similar to a taxi service – i.e. a driver immediately takes their item and delivers it in as soon a time as possible without any diversions. Other customers may be content with delivery by the end of the day. As detailed in section 5.5.1, job deadlines are generated as the sum of the direct route time, plus a sample from a gamma distribution, $\Gamma(k, \theta)$ – the excess time. By default, the excess is a sample from $\Gamma(2, 1)$ and the mean excess is 2×1 hours. In this experiment, the mean excess is varied by

adjusting the scale parameter θ. Another simulation was ran, where a flag was set to adjust all same-day C2B and B2B deadlines to a time around end-of-business (17:00–17:30). Other deadlines come with an excess from a $\Gamma(2, 6)$ distribution. This allows us to investigate whether the burstiness of delivery deadlines has a noticeable effect.

Mean Deadline Excess	Job Counts			Cost (whilst driving)		
	Completed	Late	Refused	Distance (km)	Time (h)	Fuel (l)
15 min	591	1	909	7026	198	745
30 min	1018	2	497	10588	298	1121
1 hour	1326	5	199	12119	340	1282
2 hours	1467	7	64	12109	341	1283
3 hours	1512	1	19	11759	331	1246
5 hours	1522	0	10	11754	331	1244
12 hours	1532	0	0	11254	317	1193
EOB or 12 hours	1455	0	61	10987	310	1165



Considering values of an hour or less, mean deadline excess has a massive impact on performance. 61% of jobs were refused at 15 minutes, compared to almost half at 30 minutes, 33%. Surprisingly, there is also a sizeable difference between 2 hours and 3 hours. Over the course of a day, jobs begin to be refused around noon. This is when the agent's incrementally constructed plan becomes saturated and spans a few hours. At this point, only jobs that lie almost directly on their existing path can be bid for. It seems that only a small extension to a deadline can massively increase the chance of a job being accepted.

There is a clear trend showing looser deadlines give more efficient routes. This is because the fewer constraints give the greedy planner more freedom in selecting 'nearest neighbours'. The genetic algorithm never needed to be invoked for the 12-hour test.

It is interesting that the number of late jobs rises and falls as deadlines become longer. This is likely due to the fact that when more jobs are taken on, the chances of any one of them falling behind schedule increases. However at some point the number of jobs arriving late will fall, as deadlines become so far ahead in the future that with a limited supply of jobs, it would be almost impossible to deliver any late.

Comparing the 12 hour test to the EOB test, there is a very noticeable increase in the refused job count. Figure 7.9 illustrates why this occurred. Also of note is the fact that no jobs arrived late. Typically, jobs that arrive late are those that are both created at, and due during peak times. Planning decisions often involve choosing which job to deliver last, such that the others can be on time. However, with these bursty deadlines, there is only ever one time of day when many jobs are competing to be fulfilled on time and that is at EOB.

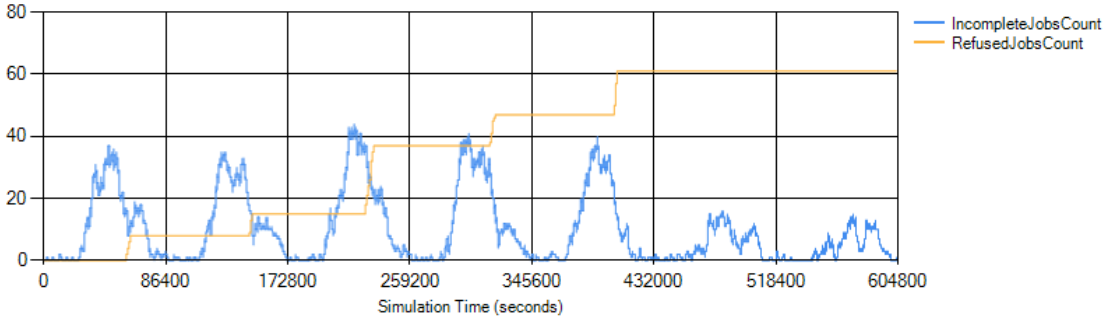


FIGURE 7.9: A graph of the simulation where business deliveries all had EOB deadlines. The blue plot shows jobs that are currently being fulfilled and the green plot shows the cumulative count of refused jobs over the seven days of simulation time. The burstiness of the deadlines causes any jobs generated just before or during EOB to be refused, as the agents' plans are saturated with immovable waypoints at this time. The green plot does not jump during the weekend, as fewer jobs are generated on these days.

7.7 Vehicles

A courier business launching in a new city will need an idea of the number of vehicles it should employ. In this section, we simulate courier networks of varying population in a variety of maps. After establishing a minimum acceptable threshold for availability and reliability, it should be apparent how many vehicles are needed.

7.7.1 Vehicle Type

The first choice is to decide which types of vehicle to use. Whilst cars will have better fuel economy than vans and trucks, the routes may be less efficient due to their smaller capacity. The properties of the vehicles modelled can be found in section 5.6. Also, autonomous vans and trucks will likely be more expensive than cars. Any improvements in performance might be outweighed by the higher upfront cost.

This first experiment performed consisted of a set of week-long simulations in Central London, designed to compare performance between varying numbers and types of vehicles. As the vehicles use different types of fuel, for comparison, the fuel costs are recorded in monetary form, rather than in litres.

Cars	Job Counts			Cost (whilst driving)		
	Completed	Late	Refused	Distance (km)	Time (h)	Fuel (£)
1	520	10	973	4216	116	497.40
3	1213	8	309	9416	265	1127.84
5	1473	3	58	11986	337	1437.73
10	1507	1	24	13293	375	1597.62
20	1508	1	23	13460	379	1619.02
50	1514	1	17	13624	383	1639.04
100	1510	0	21	13783	388	1666.99

Vans	Job Counts			Cost (whilst driving)		
	Completed	Late	Refused	Distance (km)	Time (h)	Fuel (£)
1	581	11	917	4068	113	548.64
3	1280	4	243	9120	257	1243.62
5	1492	0	39	11373	321	1551.07
10	1508	0	23	12106	341	1647.96
20	1511	0	20	12712	359	1732.10
50	1511	0	20	12741	360	1725.76
100	1511	0	20	12741	359	1730.88

Trucks	Job Counts			Cost (whilst driving)		
	Completed	Late	Refused	Distance (km)	Time (h)	Fuel (£)
1	617	1	883	4147	115	933.04
3	1311	2	213	9092	257	2078.64
5	1481	0	50	11056	313	2526.48
10	1503	0	28	11857	335	2708.99
20	1511	1	20	12342	348	2813.81
50	1511	0	20	12484	353	2844.60
100	1511	0	20	12533	353	2842.88

There are noticeable improvements as larger vehicles are used, however these are diluted as the size of the fleet increases. The main assumption here is the inter-arrival time of courier jobs (see section 5.5.1). The package size is also a very important factor. With the chosen parameters, cars are only marginally less productive than vans, which are marginally less productive than 7.5 tonne trucks. This can only be said with confidence for agent populations of 1 and 3, however. It has been demonstrated earlier that package size does make a difference to cost efficiency (and hence, availability), as capacity constraints will lead to suboptimal plans. This is certainly the case with employing a single vehicle. However, as load will usually be shared amongst tens of vehicles, it is less important that a single agent can concurrently fulfil several large jobs. The fuel

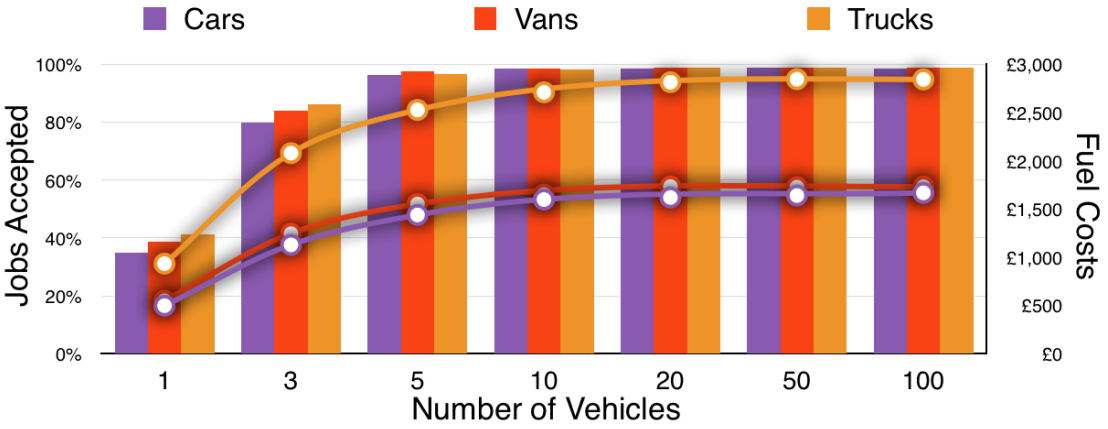


FIGURE 7.10: Performance of the courier network in Central London, using different numbers and types of vehicles. Coloured bars show the percentage of jobs that were fulfilled (rather than refused) and the plotted lines show the total fuel cost across the entire fleet.

economy model used requires that vans and trucks refuel more often, which may be to blame for the inconsistent results regarding availability for the large populations.

It is also notable that vans and trucks had far fewer late jobs than cars. This is because the impact of a failed delivery is much less. A 1 m^3 package can remain at the back of a truck for the rest of the day before it is dropped off at the depot. With a car and to a lesser extent, a van, capacity constraints may dictate that it is dropped off earlier to make way for other large packages.

In conclusion, we can say that in Central London, a courier company that expects the simulated level of demand (20 jobs per hour at peak time) and aims for a 90%+ acceptance rate, would **ideally want to purchase around five cars**. Buying any more or switching to vans or trucks hardly makes any difference – although due to the nature of the idle strategies, doing so will increase operating costs (fuel). If the company can only afford 1–3 vehicles and expected productivity is a more important factor than capital expenditure and fuel costs, it would be more preferable to purchase vans or trucks. In this project's evaluation, the car is the preferred vehicle. With ever-rising fuel costs, higher tax rates for inefficient vehicles [53] and responsibility to minimise one's carbon footprint, fuel economy is an important factor. The reduction in performance is marginal when employing a sensible number of agents.

7.7.2 Population

In the following simulations, we assume that the courier company is to use cars. For compactness, the acceptance and lateness rate are recorded instead of the individual job counts. For example, consider a simulation where 1000 jobs were spawned, of which, 100

were refused and 900 accepted and of those accepted, 9 were delivered late. Here, the acceptance rate is 90% and the lateness rate is 1%. To allow direct comparison with the Central London test, cost is measured as the monetary value of the fuel used. Note that the value of the total fuel reserves at the end of the week is deducted from this total.

In addition to Central London (above), five additional maps with different properties were used. See section D.3 in the appendix for images and properties of these maps. Across all maps and aside from agent population, all parameters remain the same. No traffic flow data could be sourced for the two small maps Stronsay and Jersey, however neither island has a large population and the signposted road speeds are expected to be relatively equal to the real-world road speeds.

Cars	Jersey	Stronsay	C. London	Isle Of Wight	Las Vegas	G. London
1	39.59%	35.16%	34.83%	24.92%	16.85%	12.93%
3	88.13%	82.47%	79.70%	58.90%	42.70%	33.00%
5	98.56%	98.22%	96.21%	82.07%	65.23%	49.05%
10	98.89%	98.88%	98.43%	97.52%	94.18%	77.32%
20	98.89%	98.95%	98.50%	97.98%	96.28%	92.28%
50	98.95%	99.01%	98.89%	98.30%	95.89%	93.59%
100	98.82%	99.01%	98.63%	98.37%	97.39%	94.77%

TABLE 7.6: Percentage of jobs that were accepted and delivered

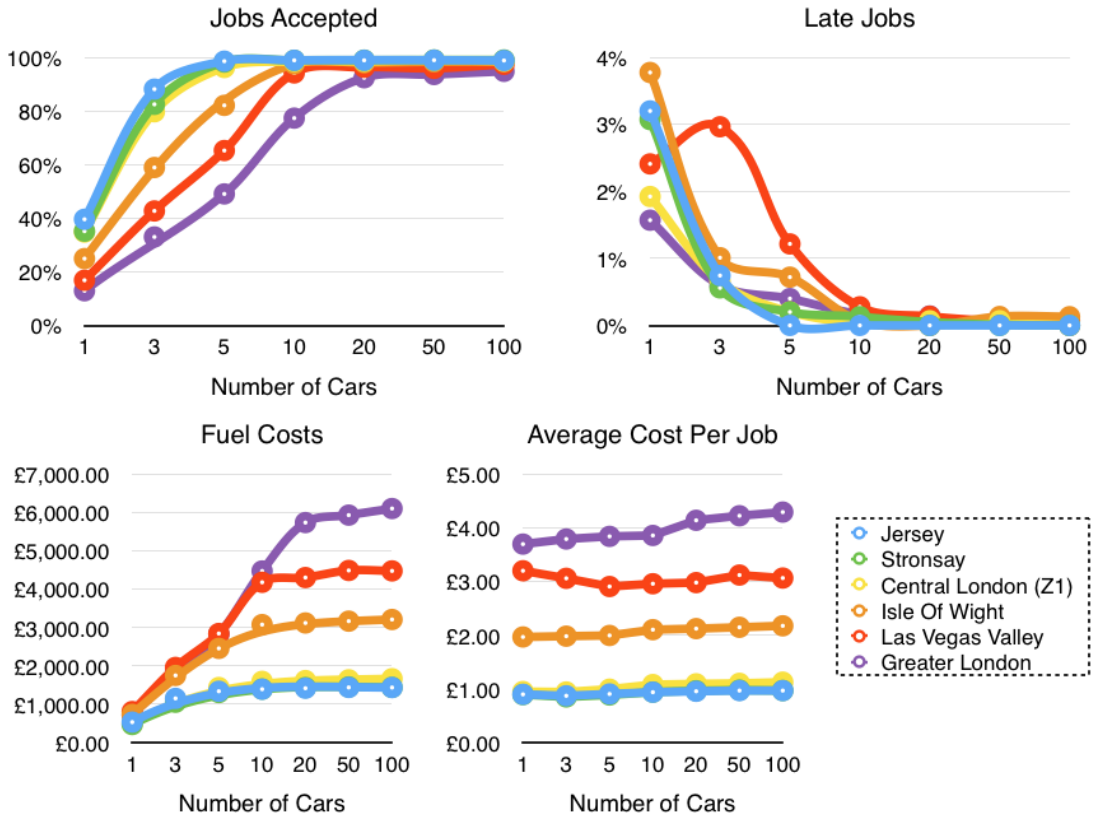
Cars	Jersey	Stronsay	C. London	Isle Of Wight	Las Vegas	G. London
1	3.20%	3.07%	1.92%	3.77%	2.41%	1.57%
3	0.74%	0.56%	0.66%	1.01%	2.96%	0.60%
5	0.00%	0.20%	0.20%	0.72%	1.22%	0.40%
10	0.00%	0.13%	0.07%	0.07%	0.28%	0.17%
20	0.00%	0.00%	0.07%	0.00%	0.14%	0.14%
50	0.00%	0.00%	0.07%	0.13%	0.07%	0.07%
100	0.00%	0.00%	0.00%	0.13%	0.07%	0.00%

TABLE 7.7: Percentage of accepted jobs that were delivered late

Cars	Jersey	Stronsay	C. London	Isle Of Wight	Las Vegas	G. London
1	£534.16	£471.91	£497.40	£736.33	£809.98	£717.98
3	£1154.22	£1053.41	£1127.84	£1753.31	£1962.27	£1879.32
5	£1340.16	£1307.69	£1437.73	£2459.38	£2848.06	£2829.60
10	£1394.33	£1396.51	£1597.62	£3080.36	£4185.45	£4480.13
20	£1419.41	£1435.78	£1619.02	£3123.64	£4308.92	£5744.66
50	£1440.06	£1453.60	£1639.04	£3170.57	£4494.09	£5939.01
100	£1437.70	£1441.87	£1666.99	£3213.88	£4482.96	£6110.87

TABLE 7.8: Total fuel costs

As one would expect, increasing the agent population increases the job acceptance rate. This rate plateaus at around 97–99%, with the remaining few being refused due to time constraints. As the dispatcher samples the deadline excess from an exponential distribution, some absurdly short deadlines are inevitable. Even with very small maps, over



long periods a 100% acceptance rate is impossible to achieve unless a deadline-invariant allocation strategy like *RoundRobinStrategy* is used. The rate at which the courier network reaches this plateau varies depending on the size, speed and connectedness of the road network. The courier company would experience diminishing returns were they to continually invest in additional vehicles.

The number of vehicles appears to be the most influential, controllable factor on the performance of the courier network. Acceptance rates rise, not only due to there being more capacity at peak times, but also because the agents are spread out more, allowing lower-cost insertions. Even in the toughest, largest map, Greater London, with the level of demand used, in no simulation did the number of vehicles being employed ever exceed 20 at any point in time. However, there is a noticeable difference in acceptance rate between the simulations of 20, 50 and 100 agents.

It was surprising that despite Stronsay being only 32.75 km^2, it required more agents than Jersey (119.5 km^2) to achieve certain acceptance rate thresholds. Both islands' performance also closely resembles that of Central London (39.9 km^2), which performed only slightly worse than Stronsay. This could be due to several factors:

- Jersey has a large number of businesses in its small capital city, Saint Helier, whereas Stronsay has none at all. This means many of the jobs generated by the

CityDispatcher will be of quite low cost. The businesses in Central London are in comparison, quite spread out.

- The topology of Jersey's road network is a lot more connected than Stronsay's, which is skeletal.
- Both islands mostly consist of relatively small and slow roads, however Jersey does have a few trunk roads spanning the island. Central London has some main roads, however during peak times for the courier, they tend to suffer heavy traffic.

Nonetheless with both islands, five cars are required to meet 95% of demand. Remember however, that to produce comparable results, the dispatch rate coefficient is kept the same over all maps, irrespective of their human population!

The Isle Of Wight, at 380 km^2, requires many more vehicles to meet demand. The largest town therein, Newport is located in the centre. The other towns are dotted around the island's coastline, separated by long trunk roads. This means most jobs have quite a distance to travel. Hence, more vehicles are required and even with 100 vehicles, 1.6% of jobs were refused.

Las Vegas Valley and Greater London were by far the largest maps with areas of 1947 km^2 and 3221 km^2 respectively. In addition to larger map size, speeds within Greater London tended to be slower than Las Vegas, which has major, multi-lane highways surrounding and dissecting the city. Greater London, on the other hand, has far fewer high-speed roads, especially in the city's centre.

Between each map, the mean fuel expenditure per job is well-defined. For example, it is distinctly higher for Greater London than it is Las Vegas. However, with most maps, this rises steadily as the population increases. Between 1 and 100 agents, the increase was around 10% for all maps except Las Vegas. It may seem counter-intuitive that with greater numbers of vehicles and the same number of completed jobs, total driving cost increases. The contract net protocol should prevent this from occurring, as only the best placed agent should be awarded a job. In fact, this phenomenon is due to the overhead brought on by the predictive idle strategy. Because it instructs unemployed agents to route to a certain location upon completing a job, it follows that more agents mean more additional routes and more occasions when an agent switches from being active to being idle. Also, with smaller populations, the agents may avoid difficult jobs that have far away waypoints or are otherwise costly to insert into their plan. This can skew the cost-per-job figure downwards, even if no idle strategy is used.

It is very apparent that reliability improves with higher agent populations. Note that this is not because of the effects of agents not taking 'risks' if they can be avoided. CNP

does not instruct the agents to bid higher in the reverse auction if jobs are more likely to arrive late. On the contrary, agents with busy schedules are more likely to be awarded jobs, as their map coverage is greater. The sole reason that greater populations lead to fewer late jobs is that CNP5 allows an agent's awarded jobs to be reallocated before they or other jobs in their plan arrive late. The more agents there are, the more jobs can be reallocated at critical times. However, as can be seen in the data, this is not a strong correlation. The reallocation procedure is used sparingly because, deadlines aside, it offsets the benefits of the initial CNP auction, thereby increasing total cost.

In some rare and unfortunate cases, an agent will be unable to avoid certain waypoints arriving late. A typical case is when an unpicked job that is nearing its deadline is far away from the other agents, yet the nearby allocated agent must unexpectedly divert to unload at the depot or refuel at a fuel point. The number of emergency refuelling diversions tends to be 1–2 per day, per agent for the small populations. This is due to the fact that they are all non-idle for very long stretches of the day. To demonstrate this, figure 7.11 shows the fuel level in the single agent Greater London simulation.

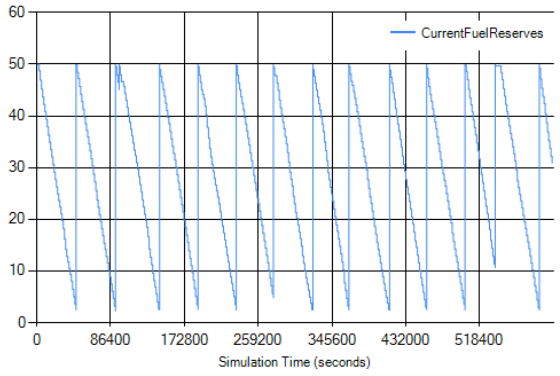


FIGURE 7.11: The fuel level of a single, independent agent in Greater London over seven days. The fuel level often drops below the 5% threshold, forcing the agent to divert to the nearest fuel point and sometimes causing late deliveries. From the graph, it is clear that the agent was overworked and rarely idle, even during the night. Only on two occasions (the early mornings of day 2 and day 7) was it idle for long enough to refuel, as per its idle strategy.

Finally, it should be noted that CNP5 is ill-suited towards single-agent populations. Typically, as the agent is unable to perform reallocations, it will take on several jobs; then it will inevitably fall behind schedule and thereafter refuse all jobs until its *CourierPlan* is free of any late waypoints. For a multi-agent network, this is sensible as it places a cap on the number of late deliveries and avoids getting a single agent into deeper trouble. For this pathological case, this approach is arguably too cautious for a single-agent population. Also, the behaviour of the predictive idle strategy is not tuned for very large populations, as many will idle around the *central* business district. Few, if any, will park themselves in the outer zones of the city, where jobs may still spawn. This

doesn't make a massive difference with regard to acceptance rates, but a hybrid of the predictive and the scattering idle strategy could potentially bump up the plateau level.

7.8 Economics

Using sensible parameters from earlier simulations, this evaluation will conclude by answering the ultimate question posed in the beginning: can an autonomous courier service be provided at a lower cost? To answer this question, we will perform break-even analysis. Investopedia defines this as "an analysis to determine the point at which revenue received equals the costs associated with receiving the revenue"[54]. The simulation does not model fixed costs, nor depreciation of capital nor staff costs. However, given the lack of drivers and the sparsity of depots, such costs are unlikely to be substantial. The major cost – fuel – is modelled in fine detail. As zero jobs equals zero expenditure with this configuration, traditional break-even analysis would conclude immediately by saying that 0 units is required to break-even. For more interesting results, we are instead concerned with the minimum price that can be charged, given an expected level of demand.

The sole source of revenue is fulfilling jobs and the exact cost of a job is determined by a base price and the number of additional cost units (time in hours, fuel in litres or distance in kilometres) to complete the job. To produce the results below, an arbitrary base and unit price is set. The simulation tracks the number of billable cost units that are used to reach these prices. As in this evaluation, we choose to minimise driving time with traffic, this number is equal to the sum of the lowest bids in the contract net and it is around 90–97% of the total number of hours driven. For example, in the Greater London simulation, customers were billed for 1091 hours, but the agents were driving for 1169 hours. This 6% overhead is due to having to carry failed deliveries to the depot, as well as carrying out the predictive idle strategy. Such overhead must be accounted for in the customer's price. Essentially, Londoners must pay a 6% tax to pay for the vehicles positioning themselves before receiving the job and also to pay for the 10% chance that the recipient won't be in. This cost could be reduced in the simulation by having more depots, however in reality this would push up the fixed costs.

For Greater London, twenty agents were used and 1429 jobs were fulfilled over a week. The total fuel costs amounted to £5447.62 and customers were charged for 1090.55 hours of driving. Late deliveries result in both the base and unit price to be refunded and this is accounted for. If the courier is often late, this pushes prices up for all customers, as they are effectively subsidising these refunds. Partial refunds due to failed pick-ups are already deducted from the 1090.55 figure, which would be a few percentage points

higher otherwise. The figures can be substituted into this equation:

$$Profit = (Units - LateUnits) \times UnitPrice + (JobCount - LateJobCount) \times BasePrice - Cost$$

Setting $Profit$ to 0 allows us to find the break-even point.

$$0 = (1090.55 - 0) \times u + (1429 - 0) \times b - 5447.62$$

This can be plotted as a line on a Cartesian plane, showing us the minimum prices that could be charged to customers, whilst still breaking even. See figure 7.12. For comparison, the same experiments were ran on other maps:

- Jersey with 5 agents. $0 = (266.26 - 0) \times u + (1511 - 0) \times b - 1372.69$
- Isle Of Wight with 10 agents. $0 = (518.07 - 0.27) \times u + (1488 - 1) \times b - 3040.82$

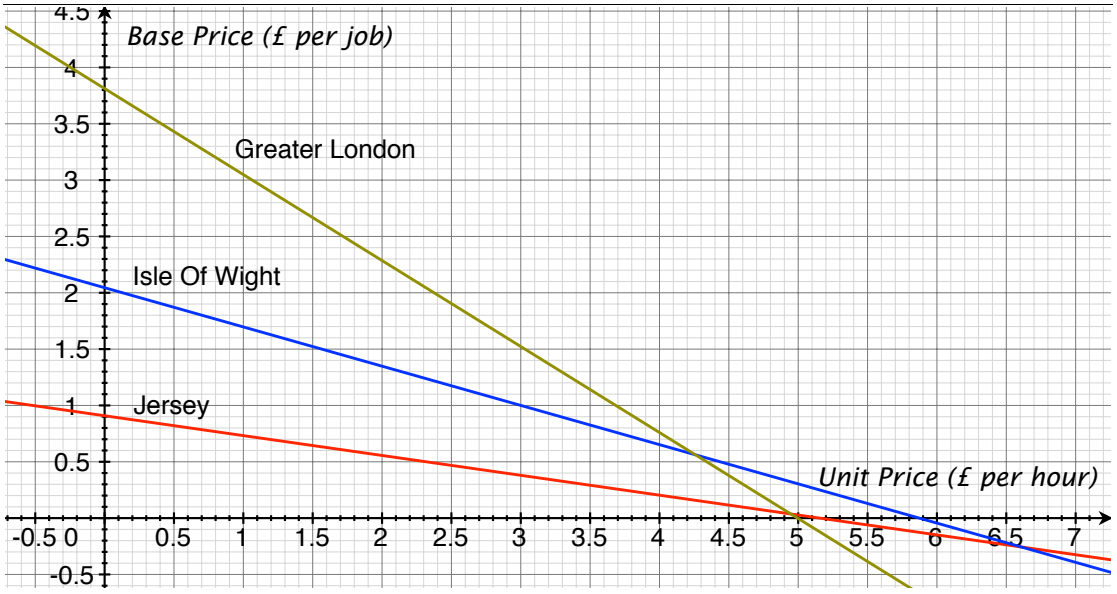


FIGURE 7.12: Pricing models for Greater London, the Isle Of Wight and Jersey. Prices are specified in terms of a fixed base price and a unit cost which scales for more difficult jobs. Regions under the break-even lines contain loss-making pricing models, whereas regions above the lines contain profit-making models. In the simulations ran, if any such model lying exactly on the line were used, the total profit at the end of the week would be exactly £0.00.

The equations and graphs illustrate the fact that the courier company could simply charge only a base price or only a unit price (e.g. either £3.81/job or £5.00/hour in Greater London). However, this may be unwise given customer's pricing expectations. Shifting some of the unit cost into the base price 'smoothens' the cost curve. With

reference to the map in figure 3.1, consider an example of two customers in Shepherd's Bush (S) who place orders to Chancery Lane (C) at similar times. As the pairs of waypoints are very similar, the marginal unit cost of the second job that comes in will be almost zero, regardless of the length of the journey. This is because the first customer has already paid for the slow journey from S to C and also the agent's journey from somewhere nearby to S to S. If there is no base price, the first customer, having talked to his 'piggybacking' neighbour, may feel overcharged. If only a base price is charged, a customer located in S who sends packages to Queensway (Q) may feel overcharged if he is paying the same price as his neighbours who send items to C. Some customers might seize the opportunity to send very large packages across the city with slim deadlines, whilst using competitors for easier jobs. This would require the base price to change often with shifting demand, or the company would make a loss. For these reasons, it is fairer and less risky from a business perspective to charge a combination of base price and unit cost.

For London, an example pricing scheme could be £2 + £2.37/hour; for the Isle Of Wight, a round £1.00 + £3.00/hour would cover costs; and for Jersey, we could charge as little as £0.50 + £2.31/hour. These examples roughly correspond to the midpoint of the lines in figure 7.12. It should be well-noted that these prices exclude VAT, which at the time of writing (June 2015), was 20% in the UK [55]. In Jersey, there exists only a Goods and Services Tax, which would increase the price for customers by 5% [56].

Comparing these three curves raises a few questions as to why they differ so much. It is clear that higher costs appear to be charged for larger maps, however if you look closely, if the courier charges only a unit price, it would be more expensive in the Isle Of Wight than in London. This is mostly due to the relationship between time and fuel economy. A typical hour of driving between waypoints will use more litres of petrol than in London or Jersey, because of the higher road speeds. If the unit price was defined in terms of fuel, it would be slightly above the modelled fuel cost of £1.1882/L. However, both base price and unit price will be affected by the driving overhead as mentioned earlier. A Greater London map with only one depot would have a much higher curve. The intersection between the y-axis and the curves – the base-price-only model – varies significantly and expectedly so. It is effectively the average cost per job and with larger maps come geographically lengthier jobs, which require more fuel on average to complete.

Figure 7.13 shows a plot of the profit levels over a week in the Jersey simulation with the example pricing model given earlier. Due to being unable to charge customers a fraction of a penny, the week concludes with a tiny loss of £5.50. If customers were charged an extra penny on the base price (£0.51), the profit would be £9.61. Changing the balance between base price and unit cost would change this plot, but only due to these

rounding errors. As revenue is received after each delivery, the rises and falls on the weekdays loosely correspond to pick-ups per hour minus deliveries per hour. This can be interpreted by the incomplete jobs count, which is plotted alongside profit. One can see that it results in a two hour lag between surges in demand and profit. Net losses are common on weekends because demand is halved, but overhead remains. Failed deliveries have more of an impact – the packages must still go to the depot and fewer deliveries can be bundled together. On a weekday, depots can more often expect several drop-offs at a time. Also, the idle strategies are run more often. These overheads are very small, but the plot magnifies subtle changes in efficiency.

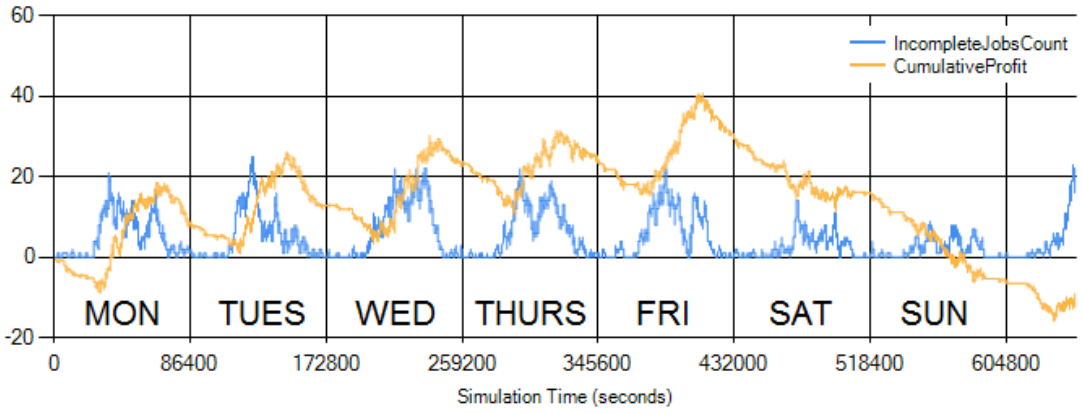


FIGURE 7.13: Total, cumulative profit in the Jersey simulation, as measured by total revenue minus total costs. The revenue model is £0.50 + £2.31/hour.

Of course, in reality, investors would expect to see profit. Were an autonomous courier to launch in one such area, the prices would likely be a lot higher. Not just to recoup any fixed and operating costs, but also because massively undercutting competitors would damage the market as a whole and could be seen by competition authorities as anti-competitive. Having said that, the courier market is small because of price elasticity of demand. The market is elastic for some because there are much cheaper substitutes – customers will respond a lot to price changes, because they may be willing to sacrifice on speed and use a cheaper, next-day courier or sacrifice on convenience and deliver the package themselves. For others who demand speed, PED may be inelastic as the service is a necessity. Since the average cost per job drops as demand increases (as shown in section 7.4), it would be wise to price the service low enough to attract many new customers to the market, but high enough so that profit can be made. For example, if the London service charged £4 + £5/hour, the simulation reports a profit of £5721.13. For the hypothetical 12-minute journey example in the introduction (chapter 1), we can definitely undercut Shutl's £10.83 (ex. VAT) quote, even if the most appropriate agent has no jobs and is an hour away. As the job is quite short and within zone 1-2, it is unlikely to be rejected unless the deadline is very slim. Shutl and CitySprint quoted pick-up and delivery within 3–4 hours, which would be an easy deadline for our network.

7.8.0.1 Premium Pricing Models

An alternative, unexplored pricing model would charge premiums for difficult aspects of the job, in addition to the base and unit price. This may be an absolute charge (e.g. +£0.50 extra) or a percentage of some sort (e.g. +5%). Such premiums might include:

- Oversized packages, particularly if the network employs a variety of vehicles.[4]
- Very short deadlines.
- Failed delivery 'insurance', which dictates that failed drop-offs are to be returned to the sender.
- The requirement that from pick-up to drop-off, no intermediate stops are made, meaning the fastest, direct route must be taken.

It may not seem fair to charge a customer extra for such things. The pricing model we use should naturally charge more for more difficult jobs, because the customer pays for the extra cost incurred by the courier agent. If their job has a slim deadline or is oversized (thus requiring a direct route), the solution to the planning problem with the additional waypoints will have a much greater cost. However, if their job is objectively difficult, but the agent is idle or otherwise happens to be in a prime position to fulfil it, the customer is not charged more simply because of the size or deadline. This however may be the wrong approach. Consider a case when, in the early morning, an idle agent is awarded a large job that has a long direct route from pick-up and drop-off points P and D. The agent picks it and begins the route. Because the agent was idle anyway, the size would not have affected the price. At the time of bidding, the size of the job does not have an impact on the time it takes to travel between P and D. However, future jobs that come in will be more difficult to efficiently slot into its plan – most likely, they will have to be appended to the end, even if they lie along the route from P to D. Perhaps, charging for premiums would act as a disincentive for customers to place jobs that *may later affect the efficiency* of the courier network.

7.9 Simulator and Software Design

7.9.1 Running Time

To allow for very long multi-agent simulations, a lot of effort was put into reducing the running time of the simulator. This was done by designing the planning algorithms to

[4] Both CitySprint and Shutl charge different rates for deliveries, depending on their size and weight. Specifically, they ask the customer to select a vehicle that is large enough to carry their package [4] [5].

run quickly and short circuit when an acceptable solution is found (i.e. not running the genetic algorithm unless the greedy search fails). Thereafter, the Visual Studio profiler was used to identify 'hot' lines of code that could potentially be optimised or performed in parallel. The A* algorithm typically makes up 95% of the profiler's samples and it is mainly invoked in parallel by the agents via their *NNGAPlanner* when they need to bid for jobs. Micro-optimisation was performed on functions that were called very frequently. For example, a function that finds the great circle distance between two points using the Haversine formula is called thousands of times a second. At present, profiling a typical simulation says that it accounts for 17% of CPU time. Several other implementations of the function were benchmarked, but they were slower or too inaccurate. In the end, a performance gain of around 30% was achieved by replacing instances of the exponentiation operator (^) with multiplication. Doing so made no difference to accuracy.

Table 7.9 shows some examples of running times for various experiments performed earlier, for 24 simulated hours. The values of ε are those used for said experiments and are based on the findings from section 7.1.1. 2D graphics were disabled for these simulations, although it only makes a noticeable difference to the two fastest simulations. For the Isle Of Wight, runtimes for the three different dispatchers, *CityDispatcher*, *SingleBusinessDispatcher* and *HubAndSpokeDispatcher* are given to illustrate the performance benefits of being able to use cached routes. St. Agnes is a tiny island off the coast of England that is 1.48 km^2 and has only a few roads. Its tiny runtime of 2 seconds per simulation day mainly consists of the overhead of adding rows with simulation state data to the table used by the chart GUI.

Map	**Agent Population**	**A* Epsilon**	**Running Time**
St. Agnes	1	0	0h 0m 2s
Stronsay	5	3	0h 0m 8s
Jersey	5	2	0h 0m 46s
Central London (Z1)	5	7	0h 1m 39s
Las Vegas Valley	10	1	0h 22m 54s
Greater London	20	7	0h 24m 15s
Isle of Wight (CD)	10	2	0h 2m 1s
Isle of Wight (SBD)	10	2	0h 0m 47s
Isle of Wight (H&SD)	10	2	0h 0m 26s

TABLE 7.9: Runtimes for simulating a number of environments.

Chapter 8

Conclusion

This report concludes with a summary of the project's achievements and its contributions to the field of logistics optimisation.

A number of algorithms were developed to enable efficient fulfilment of deliveries. Bottom-up, these are:

- An implementation of the A* algorithm for point-to-point route finding. Depending on the user's requirements, this can attempt to minimise distance, time or fuel. This was designed such that it can be sped up in exchange for optimality.
- The planning algorithm, which consists of two phases: a greedy algorithm with bounded backtracking, followed by a genetic algorithm. These solve an $\mathcal{NP}$-hard problem and as demonstrated in section 7.1.2, can be combined to find a solution typically only a third more costly than the optimal.
- The routing and allocation strategy, which extends the Contract Net Protocol and allows for dynamic and immediate allocation of tasks to agents. By performing optimisation locally and in parallel over a distributed system, agents are able to bid for jobs. If the agent with the winning bid later suffers delays, it is able to reperform the auction in efforts to spread its workload and reduce any deadline violations.
- A selection of idle strategies, which maximise an agent's performance through premature refuelling and strategic positioning.

To make journey times in the simulation more accurate, the OpenStreetMap data was unified with minute-by-minute traffic data from Here Maps. The simulator also models pedestrian crossings, traffic lights, level crossings and random unexpected obstacles, which as a whole can massively influence journey times in real-world cities.

8.1 Limitations

8.1.1 Illegal Road Turns

Due to the lack of OpenStreetMap metadata and the sheer difficulty in making accurate estimates to this regard, turning restrictions are not modelled. On two-way streets, vehicles are allowed to double back, which may in the real world constitute an illegal U-turn. Some turning restrictions are marked as relations in OpenStreetMap, however these are not read by the OSM parser. Whilst this does limit the accuracy of the simulation to some extent, it is unlikely that such restrictions will make a substantial difference to the results in the evaluation. Remedying this would add some additional overhead on the *AStarSearch* class, necessitating a validity check before adding a node to the open set.

8.1.2 Agent Collision

Because the number of agents modelled typically far outweigh the number of nodes in the road network, physical agent interaction is not modelled. If a road delay node is active, agents do not queue behind it, but rather occupy the same space. To this effect, the simulation could be made more accurate by implementing collision avoidance; however it is unlikely to make a substantial difference to the results gathered. In large maps, agents will rarely come into contact with one another, apart from at depots.

8.1.3 Job Stream

Much care was put into implementing a dispatcher that would best emulate the hourly demands of businesses and consumers. However, in the real world, demand is unlikely to be predictable. There are so many unmodellable factors that could affect the frequency and types of jobs ordered. A more extensive evaluation could be carried out with actual, long-term data from a city courier such as CitySprint. Were this available, say in the form of a CSV file, a new implementation of *IDispatcher* could be made. Upon instantiation, it would parse all of this data into a queue of *CourierJob*s, ordered by date and time. Like the *DepotDispatcher*, this queue forms a schedule and it would broadcast a job upon dequeuing it.

8.2 Extensions

8.2.1 Easy Additions

The simulator's software design allows certain functionality to be developed with ease, against interfaces. We are pleased that despite the complexity of the simulation, good software design principles could be followed.

- Dispatchers implement *IDispatcher*. To control the properties, whereabouts and frequency of the jobs in the job feed, a new class can be easily written. For example, if one wanted to simulate a courier network for blood, this could be implemented easily by randomly selecting waypoints from nodes tagged `amenity=hospital` and perhaps using fixed deadlines (e.g. the current time plus thirty minutes).
- More autonomous vehicles can be modelled, such as a motorcycle [57] or even a bipedal robot. The *Vehicles* module defines an Enum. One could add another vehicle and define its fuel economy, using either a coefficient on the existing parabolic formula, or a new formula altogether.
- Routing and idle strategies have been designed to be interchangeable. Writing a new one or another CNP variant would prove to be an easy extension.

8.2.2 Progressive Route Finding

As can be seen in the results in section 7.1.1, if one wants to run a long simulation, they must compromise on the optimality of the route finding algorithm. Computation time is less of an issue if the algorithms are allowed to execute in real time, however customers would still expect reasonably fast responses when they request a quote on a job. A modified solution is proposed wherein the vehicles make quotes based on suboptimal route finding, but then in the time between job requests coming in, they rerun the A* algorithm between each pair of waypoints in the plan, using an admissible heuristic. In fact, running Dijkstra's algorithm and caching the results may be more appropriate in this case, as there is more time available. The agent could also keep running the genetic algorithm over thousands of generations, changing the plan upon seeing a less costly one.

8.2.3 Handoff

Like in the simulations developed in [28] and [29], it would be interesting to extend the abilities of the agents, such that they are able to hand off packages at fixed exchange

points. The papers do not demonstrate a real improvement in efficiency over taking the direct route, at least in terms of driving cost. However, it assumes an overly large network of local couriers. This contrasts to this project wherein the agent population tends to be no larger than what is necessary, which leads to agents' schedules becoming saturated. For us, job acceptance rate is the key performance indicator. The CNP5 extension is somewhat limited by the inability to auction off jobs that have already been picked up. If this can be performed, perhaps with a human mediator at a depot, the number of late deliveries may be minimised further.

8.2.4 Traditional Couriering

This project has demonstrated the efficiencies that can be made in running an *unmanned* same-day courier service. It would easily be possible to modify certain agent behaviours to emulate a traditional courier service that uses human drivers. Though it would require changes to the functionality of each routing strategy, it would not be too difficult to have the 'agent' reattempt failed deliveries at neighbouring addresses. A courier company's fleet tend to be parked on their own property, rather than scattered around the city. This could be implemented by modifying the sleeping idle strategy, such that the vehicles route towards depots, rather than fuel points.

8.2.5 Map-Specific Strategies

Certain maps would benefit from custom strategies. For example, in the Isle Of Wight, the predictive idle strategy tends to position all the agents somewhere in the centrally located town of Newport, as it will inevitably be the geographical 'mean' pick-up location. However, this increases cost if a job spawns on one of the coastal cities. A better strategy would be to position an agent in each town, so that the distance to the first waypoint of the day is better minimised. This could easily be implemented as an idle strategy, although of course, it would not be portable. If the model can be loosened, there could be a human controller who is able to remotely send routing commands to the agents at off-peak times. This would couple nicely with the fact that in the real world, the cars will need occasional maintenance.

8.3 Closing Remarks

The initial objectives of this project were to design, implement and test a same-day courier network. On reflection, we believe that due to careful time management and

swift recovery following unexpected setbacks, these objectives have largely been exceeded in scope and this project has gone much further than initially thought. A fully functional product has been output and with minor adjustments could be rolled out to a fleet of vehicles. The algorithms could be eternally iterated upon and the reality is, custom per-city strategies will always outperform general purpose implementations (i.e. standard CNP). However, the results of this project's evaluation show that what has been developed is beyond fit for purpose.

A large-scale, affordable, same-day courier service has been shown to be viable once unmanned autonomous cars become street-legal. However, there do exist some unanswered questions that we leave to the reader. The solution proposed has made it possible for people to cheaply transport packages across a city in a matter of hours. However, is this what people really want, given the increasing usage of digital delivery for documents and the advancements in 3D printing technology? Will high upfront costs of autonomous cars largely outweigh the efficiencies made? Would customers be comfortable with their mail being accessible to other customers? Will police forces even allow such a service to be rolled out, given the scope for anonymous transportation of illegal or dangerous goods? In any event, the author of this project is keen to see what the future holds for the same-day courier industry.

Appendix A

Table of Parameters

All default values for the parameters used in the simulation are given in the Implementation (chapter 5). For the reader's convenience, they are also summarised below. Note that time spans and frequencies/periods are usually specified in seconds, as these correspond to the smallest unit of time in the simulation. Speeds are specified in kilometres per hour. Volumetric sizes and capacities are specified in cubic metres. Fuel amounts and tank sizes are specified in litres.

A.1 Fulfilment

Parameter	**Range**	**Default value**
Minimum time for a customer to load a package	1–∞	20
Maximum time for a customer to load a package	1–∞	120
Minimum time for a customer to retrieve a package	1–∞	20
Maximum time for a customer to retrieve a package	1–∞	120
Time for a depot employee to load a package	1–∞	20
Time for a depot employee to retrieve a package	1–∞	20
Time before arrival that the customer is notified, meaning the waypoint is 'locked' in place and cannot be rescheduled	0–∞	300
Revised deadline for a job to reach a depot or the sender, after delivery failure	0–∞	12 hours
Probability of a pick-up being successful	0–1	0.99
Probability of a drop-off being successful	0–1	0.9
Planner redundancy time per waypoint, τ	0–∞	240

Note, in most cases, the A* algorithm is configured to minimise `TIME_WITH_TRAFFIC`, rather than distance or fuel. Unless otherwise specified, the routing strategy is CNP5 and the idle strategy, *PredictiveIdleStrategy*. The main dispatcher is the *CityDispatcher*, failed deliveries are rerouted to the nearest depot and the *DepotDispatcher* is enabled.

A.2 Speed Limits

For each OpenStreetMap way, speed limits are sometimes tagged. Where it is not or the tag's value cannot be parsed, we assume the following speeds for each road type:

Highway Classification	**Range**	**Default value**
Motorways	0–1.08×10^9	112
Trunk roads	0–1.08×10^9	80
Primary roads	0–1.08×10^9	64
Residential roads	0–1.08×10^9	32
Service roads	0–1.08×10^9	8
Other roads	0–1.08×10^9	48
Global speed limit (for route finding)	0–1.08×10^9	112

A.3 Road Delays

Note that all probabilities are multiplied by the corresponding value in the traffic distribution table (section D.1), divided by the maximum number in that table, which is 212. By this measure, the probability of a road delay as listed below is equal to the probability of it being activated within a period at peak time. Delay lengths must be less than or equal to the frequency or period.

Parameter	**Range**	**Default value**
Traffic lights, frequency	1–∞	60
Traffic lights, delay length	1–∞	30
Traffic lights, activation times	any range	all day
Level crossings, frequency	1–∞	600
Level crossings, delay length	1–∞	120
Level crossings, activation times	any range	all day, except 01:00–04:00
Traffic light pedestrian crossings, period	1–∞	45
Traffic light pedestrian crossings, delay length	1–∞	15
Traffic light pedestrian crossings, probability	0–1	1
Zebra pedestrian crossings, period	1–∞	30
Zebra pedestrian crossings, delay length	1–∞	10
Zebra pedestrian crossings, probability	0–1	1
Unexpected minor delays, period	1–∞	2
Unexpected minor delays, delay length	1–∞	2
Unexpected minor delays, probability	0–1	0.00222
Unexpected major delays, period	1–∞	30
Unexpected major delays, delay length	1–∞	30
Unexpected major delays, probability	0–1	0.00417

A.4 Dispatchers

For the table below, note that the maximum size of a package should be set in accordance to the capacity of the vehicles, which will immediately refuse any job that is too large. The mean sample from an exponential distribution is λ^{-1}, so the mean of the default distribution is 0.333 m^3. The sum of the probabilities of jobs being B2B (business-to-business), B2C (business-to-consumer), C2B (consumer-to-business) or C2C (consumer-to-consumer) at any one time must equal 1.

Parameter	**Range**	**Default value**
Mean dispatch rate per hour	0–∞	see A.4.1
Probability of job being B2B	0–1	see A.4.1
Probability of job being B2C	0–1	see A.4.1
Probability of job being C2B	0–1	see A.4.1
Probability of job being C2C	0–1	see A.4.1
Deadline excess time in hours, $X \sim \Gamma(k, \theta)$	$k > 0$, $\theta > 0$	$k = 2$, $\theta = 1$
End-of-business time range	any, except all day	17:15–17:45
Volumetric size of packages, $X \sim Exp(\lambda)$	$\lambda > 0$	$\lambda = 3$
Volumetric size of packages, minimum	> 0	0.0002
Volumetric size of packages, maximum	> 0	0.999
DepotDispatcher rebooking time in hours, $X \sim Exp(\lambda)$	$\lambda > 0$	$\lambda = \frac{1}{48}$
DepotDispatcher rebooking probability	0–1	0.5
SingleBusinessDispatcher, probability of B2C	0–1	0.5
HubAndSpokeDispatcher, probability of B2C	0–1	0.5

A.4.1 Dispatch Rate

Dispatch rates and proportions of job types vary by hour and whether it is a weekday or a weekend. In the tables below, 'Rate' refers to the average number of jobs that are generated per hour.

Monday–Friday

Hour	Rate	B2B	B2C	C2B	C2C
0	1	20%	10%	0%	70%
1	1	20%	10%	0%	70%
2	1	20%	10%	0%	70%
3	1	20%	10%	0%	70%
4	1	20%	10%	0%	70%
5	1	20%	10%	0%	70%
6	1	20%	10%	0%	70%
7	10	80%	5%	5%	10%
8	15	80%	5%	5%	10%
9	20	80%	5%	5%	10%
10	20	80%	5%	5%	10%
11	20	80%	5%	5%	10%
12	20	80%	5%	5%	10%
13	20	80%	5%	5%	10%
14	20	80%	5%	5%	10%
15	20	80%	5%	5%	10%
16	18	80%	5%	5%	10%
17	14	80%	5%	5%	10%
18	10	20%	10%	0%	70%
19	6	20%	10%	0%	70%
20	4	20%	10%	0%	70%
21	3	20%	10%	0%	70%
22	2	20%	10%	0%	70%
23	1	20%	10%	0%	70%

Saturday and Sunday

Hour	Rate	B2B	B2C	C2B	C2C
0	1	20%	10%	0%	70%
1	1	20%	10%	0%	70%
2	1	20%	10%	0%	70%
3	1	20%	10%	0%	70%
4	1	20%	10%	0%	70%
5	1	20%	10%	0%	70%
6	1	20%	10%	0%	70%
7	5	50%	20%	10%	20%
8	7	50%	20%	10%	20%
9	10	50%	20%	10%	20%
10	10	50%	20%	10%	20%
11	10	50%	20%	10%	20%
12	10	50%	20%	10%	20%
13	10	50%	20%	10%	20%
14	10	50%	20%	10%	20%
15	10	50%	20%	10%	20%
16	8	50%	20%	10%	20%
17	7	50%	20%	10%	20%
18	5	20%	10%	0%	70%
19	3	20%	10%	0%	70%
20	2	20%	10%	0%	70%
21	1	20%	10%	0%	70%
22	1	20%	10%	0%	70%
23	1	20%	10%	0%	70%

A.5 Vehicles

Fuel economy is specified in miles per gallon. At speed x and using the vehicle's fuel economy coefficient k, the vehicle achieves $k \times (-0.0119x^2 + 1.2754x)$ mpg. The refuelling thresholds are based on how full the fuel tank is, as a percentage.

Parameter	**Range**	**Default value**
Car, capacity	0–∞	1
Car, tank size	0–∞	50
Car, fuel economy coefficient k	0–∞	1
Car, fuel type	petrol, diesel	petrol
Van, capacity	0–∞	2
Van, tank size	0–∞	80
Van, fuel economy coefficient k	0–∞	0.92
Van, fuel type	petrol, diesel	diesel
Truck, capacity	0–∞	8
Truck, tank size	0–∞	80
Truck, fuel economy coefficient k	0–∞	0.55
Truck, fuel type	petrol, diesel	diesel
Petrol cost per litre (£)	0–∞	1.1327
Diesel cost per litre (£)	0–∞	1.1882
Emergency refuelling threshold	0–1	0.05
Idle strategy refuelling threshold	0–1	0.95

Appendix B

User Guide

B.1 Installation

The courier simulator requires an installation of the .NET framework, version 4.5.2. A portable, standalone executable is provided, which can be run as is.

B.2 Importing Maps

Maps can be downloaded from www.openstreetmap.org. Simply navigate to the desired area, click the 'Export' button at the top and download the map using the 'Overpass API' link. See figure B.1. Save the file with an .osm extension to the `maps/` directory.

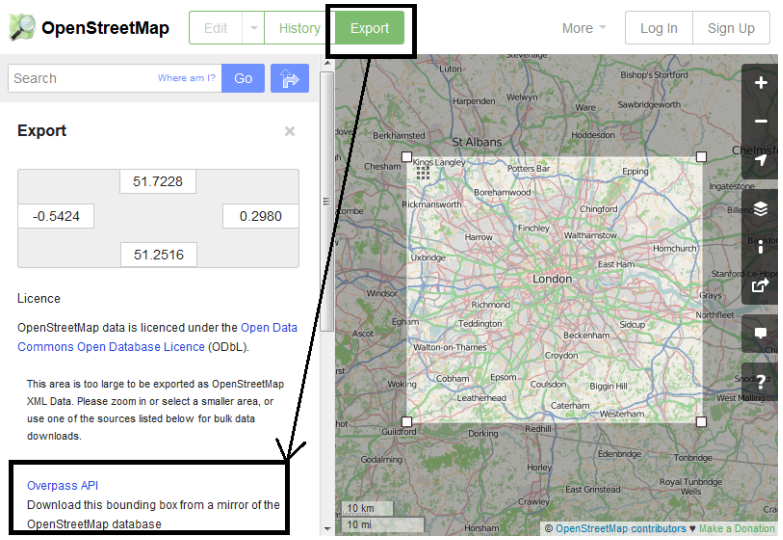


FIGURE B.1: Downloading a map from www.openstreetmap.org.

B.3 Usage

B.3.1 Loading a Map

After opening the program, one first needs to choose a map from the `File` menu. Note that loading very large or dense maps may take some time and will require more memory. When it is loaded, the map will be drawn in the window, which one might want to resize to match the proportions of the map area.

B.3.2 Configuring the Simulation

From the `Agents` menu, one can change any of the following preferences: routing and allocation strategy, idle strategy, vehicle type, dispatcher and whether failed deliveries should be returned to the nearest depot or the sender. These must be set before the simulation is started. The simulation parameters window will be visible on start-up, however if it has been closed, it can be reopened via the menus, `Simulation > Set Parameters`. Here one can change parameters used in probability distributions that correspond to dispatch rate, deadline excess, package size and pick-up/delivery failure. In addition, one can change the base and unit price charged to customers and the weight that is applied to the heuristic cost in A* route finding (equal to $1+\varepsilon$). All such variables can be changed pre- or mid-simulation by adjusting sliders or typing a value into the adjacent text box.

B.3.3 Running the Simulation

Having configured the simulation, select `Simulation > Start Simulation`. Then, add the desired number of agents from the `Agents` menu. The simulation begins in real-time. To speed it up, look to the top two variables, `Simulation Speed` and `Graphics Refresh Rate` in the parameters window. Adjust these to be higher. Turning off the map display will allow the simulation to run faster. This can be done by selecting select `View > Pause Display`.

As the simulation runs, one can observe the status of the agents and their plans by looking at the 'agent status console'. This window is opened at launch and can be reopened via the menus, `Simulation > View Console`. One can also see certain simulation variables plotted on a graph by accessing `Simulation > View Statistics`. Here, one needs to click the refresh button or check the timer refresh checkbox for the values to be updated. The variables are presented as a list of checkboxes. Selecting one or more will

plot them on the graph adjacent. Do note however, that due to the very large number of data points, frequent refreshing of the chart can slow down the simulation.

B.4 Stopping a Simulation

To pause a simulation, go to `Simulation > Pause` and to resume, go to `Simulation > Start Simulation`. To reperform a simulation on the same map, perhaps with different parameters or preferences, first do `Simulation > Reset`. Then, after making any adjustments to the configuration (as explained earlier), select `Simulation > Start Simulation`. Selecting a new map from the `File` menu will also reset the current simulation, giving you the chance to configure the new one, before starting the simulation like before.

Appendix C

Playground

In the early stages of the project, we worked on building a platform that could be used to simulate a courier network. At this point, there existed autonomous agents, however their routing strategy was pointless. They would simply choose a random point on the map to route towards, calculate the A* route in a background thread and begin driving there. Upon arrival, the strategy would repeat. If the user clicked anywhere on the map, all agents would navigate to that point and recommence the original strategy thereafter. Once courier jobs were modelled and proper routing and allocation strategies were written, this code was moved into an *AAPlayground* class, a subclass of *AASimulation* and a sibling class of *AACourierSimulation*, to which this report is based upon. It can still be run as before, by selecting `Simulation > Start Playground`.

Though this mode provides no insight, it did have a useful purpose in allowing us to benchmark the performance of the route finding class (at this early point, several were in development) and test the stability of the simulator. Due to the simple nature of the strategy, it is feasible to spawn thousands of agents on a single map and have them drive around concurrently. Figure C.1 shows one such example. Testing such things as the strongly connected components algorithm successfully pruning all inaccessible and inescapable nodes and road delay nodes delaying all passing agents at the same time (see figure C.2), were reduced into simple observation tasks.

FIGURE C.1: An *AAPlayground* in Jersey, with 5000 autonomous agents!

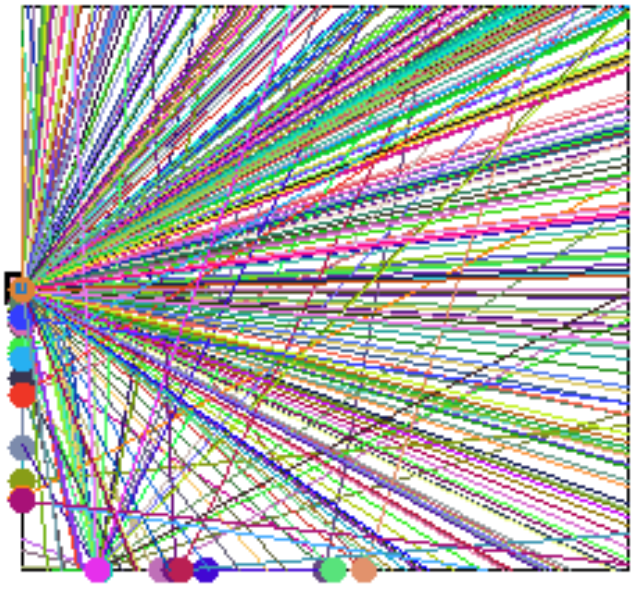


FIGURE C.2: An *AAPlayground* on a fictional map with a single one-way road that forms a square. Almost all of the 500 agents are concurrently stuck waiting for a level crossing on the middle-left node. A line is drawn from each agent to its destination.

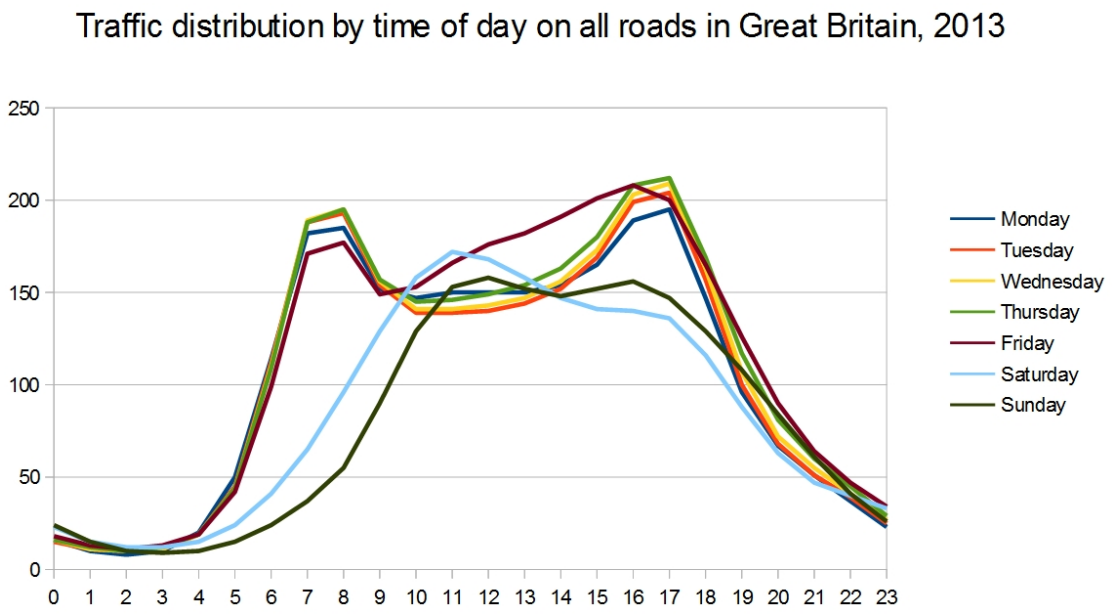
Appendix D

Extra Material

D.1 Temporal Traffic Distributions

The following data set was used for computing the likelihood of pedestrian-related road delays occurring. It is table TRA0307, published by the UK Department of Transport in June 2014 [41].

Time of day	Mon	Tue	Wed	Thu	Fri	Sat	Sun
00:00-01:00	16	15	16	16	18	23	24
01:00-02:00	10	11	11	12	13	15	15
02:00-03:00	8	10	10	10	11	12	10
03:00-04:00	10	11	11	12	13	12	9
04:00-05:00	20	19	19	19	19	15	10
05:00-06:00	50	45	44	44	42	24	15
06:00-07:00	114	112	110	109	99	41	24
07:00-08:00	182	188	189	188	171	65	37
08:00-09:00	185	193	195	195	177	96	55
09:00-10:00	152	154	156	157	149	129	90
10:00-11:00	147	139	141	145	153	158	129
11:00-12:00	150	139	141	146	166	172	153
12:00-13:00	150	140	143	149	176	168	158
13:00-14:00	150	144	147	154	182	158	152
14:00-15:00	154	152	156	163	191	147	148
15:00-16:00	165	169	173	180	201	141	152
16:00-17:00	189	199	203	208	208	140	156
17:00-18:00	195	204	209	212	200	136	147
18:00-19:00	147	157	163	169	165	116	129
19:00-20:00	96	100	107	117	126	88	108
20:00-21:00	67	68	72	81	90	63	84
21:00-22:00	51	51	55	60	64	47	61
22:00-23:00	37	39	41	45	47	40	41
23:00-00:00	23	25	27	29	34	33	26



D.2 Tables

Dispatch Rate	Completed Jobs	Completed Jobs (%)	Late Jobs (%)	Distance (km)	Distance per Job (km)
0.1	19	100.00%	0.00%	241	12.68
0.1	23	95.80%	0.00%	308	13.39
0.1	27	100.00%	0.00%	355	13.15
0.1	29	100.00%	0.00%	432	14.90
0.1	22	100.00%	0.00%	304	13.82
0.25	63	94.02%	0.00%	716	11.37
0.25	49	98.00%	0.00%	581	11.86
0.25	55	96.49%	0.00%	670	12.18
0.25	54	96.43%	0.00%	690	12.78
0.25	63	98.44%	0.00%	702	11.14
0.5	115	96.64%	0.87%	1154	10.03
0.5	102	98.08%	0.00%	1043	10.23
0.5	81	93.10%	0.00%	1014	12.52
0.5	103	98.10%	0.00%	1121	10.88
0.5	112	94.91%	0.00%	1056	9.43
1	229	93.85%	0.04%	1918	8.38
1	216	90.76%	0.00%	1781	8.25
1	217	94.76%	0.00%	1756	8.09
1	231	91.67%	0.09%	1840	7.97
1	240	93.02%	0.00%	1878	7.83
2	303	68.24%	0.67%	2200	7.26
2	349	75.54%	0.28%	2319	6.64
2	323	66.19%	1.86%	2287	7.08
2	335	75.62%	1.49%	2232	6.66
2	314	66.53%	0.96%	2259	7.19
3	339	49.78%	3.54%	2409	7.11
3	345	50.15%	2.31%	2455	7.12
3	358	52.26%	2.79%	2400	6.70
3	369	55.24%	1.63%	2416	6.55
3	335	47.32%	1.19%	2430	7.25
5	403	35.35%	2.98%	2765	6.86
5	406	34.06%	1.97%	2696	6.64
5	386	35.70%	4.40%	2643	6.85
5	417	34.49%	2.16%	2733	6.55
5	387	33.59%	2.33%	2718	7.02

TABLE D.1: Varying the dispatch rate for a 24-hour London Z1 simulation

D.3 Maps

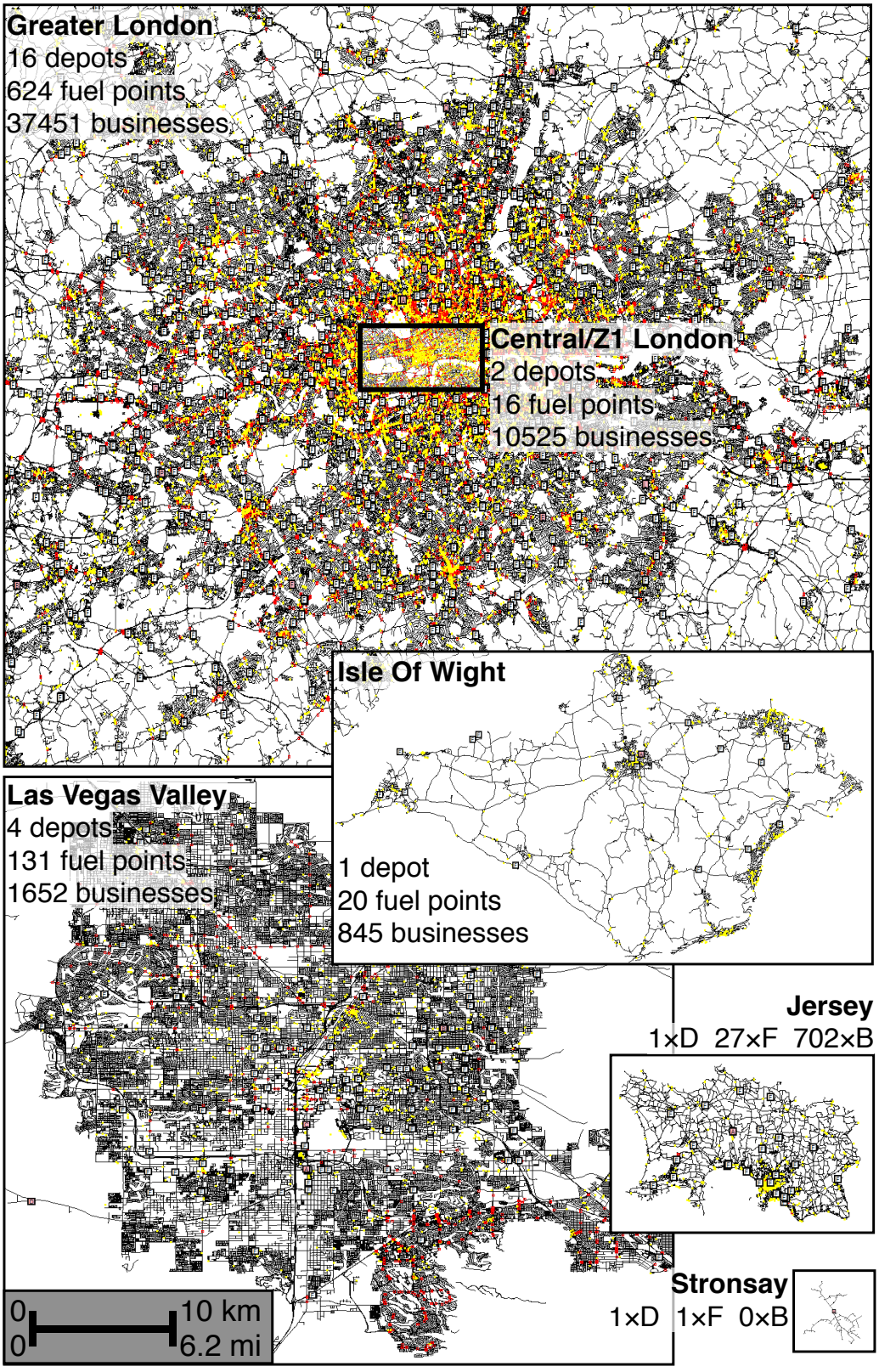


FIGURE D.1: Renderings of all of the maps used in the evaluation and their properties. The maps are drawn to scale.

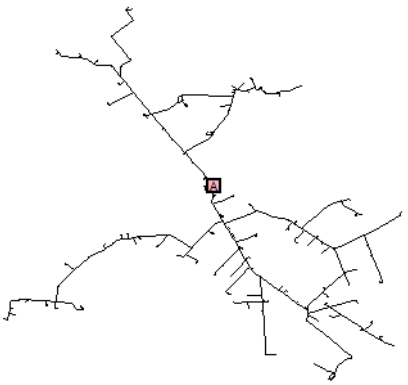


FIGURE D.2: Rendering of Stronsay (not to scale).

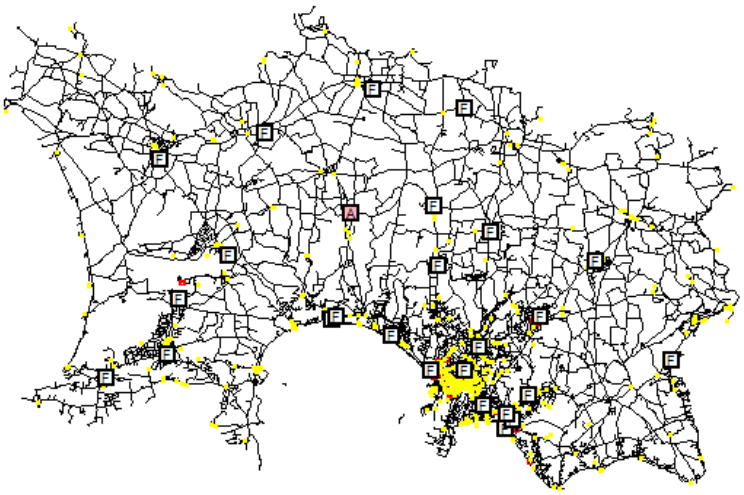


FIGURE D.3: Rendering of Jersey (not to scale).

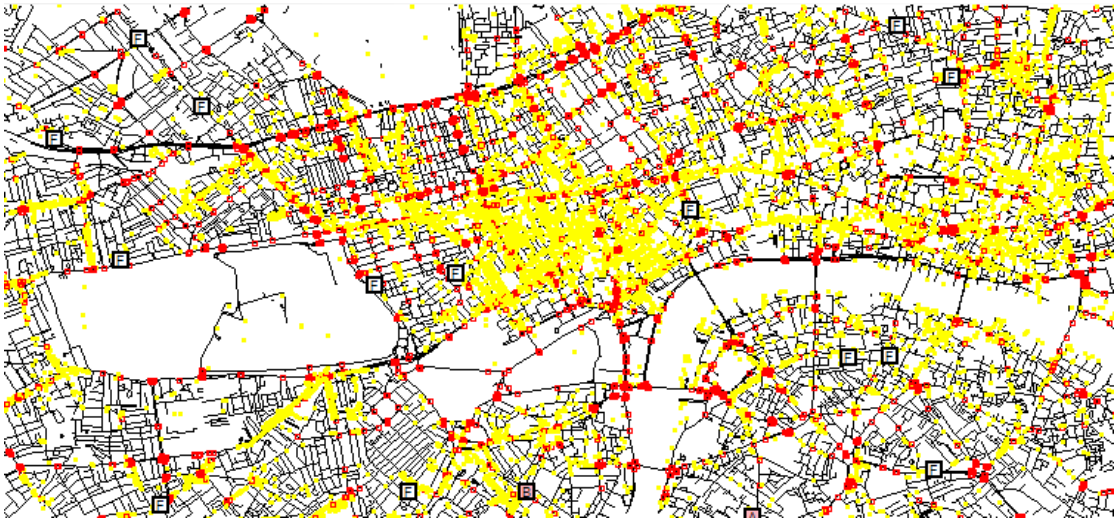


FIGURE D.4: Rendering of Central London (TFL Zone 1) (not to scale).

Bibliography

[1] Royal Mail Group, Datamonitor, 2012. *Royal Mail Group parcels expansion to create 1,000 new UK jobs.* Available online at http://www.royalmailgroup.com/cy/node/4315. Accessed on January 23, 2015.

[2] Google Maps. *W6 8RP, United Kingdom to London SW7 2AZ, UK.* Available online at https://www.google.co.uk/maps/dir/W6+8RP/SW7+2AZ/. Accessed on June 1, 2015.

[3] Royal Mail. *Sameday parcel delivery and courier services.* Available online at http://www.royalmail.com/parcel-despatch-low/uk-delivery/sameday. Accessed on June 1, 2015.

[4] CitySprint. *Sameday Courier QuikQuote.* Available online at https://onlinecc.citysprint.co.uk/QuoteAndBooking/Quote.aspx. Accessed on June 1, 2015.

[5] Shutl Ltd. *Shutl.it.* Available online at https://uk.shutl.it/. Accessed on June 1, 2015.

[6] Anywhere Sameday Couriers. *Instant Online Courier Prices.* Available online at http://www.anywherecouriers.co.uk/same-day-courier/. Accessed on June 1, 2015.

[7] M. Wooldridge. *An Introduction to MultiAgent Systems*, p 366. John Wiley & Sons, 2002.

[8] L. Panait, S. Luke. *Cooperative Multi-Agent Learning: The State of the Art.* In *Autonomous Agents and Multi-Agent Systems* Vol. 11, No. 3, 2005, pp 387–434. Available online at http://cs.gmu.edu/~eclab/papers/panait05cooperative.pdf.

[9] S. Franklin, A. Graesser. *Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents.* In proceedings of the *Third International Workshop on Agent Theories, Architectures, and Languages*, Springer-Verlag, 1996. Available online at http://www.inf.ufrgs.br/~alvares/CMP124SMA/IsItAnAgentOrJustAProgram.pdf.

[10] R. G. Smith. *The Contract Net Protocol: High-Level Communication and Control in a Distributed Problem Solver*. In *IEEE Transactions on Computers*, Vol. 29, No. 12, 1980. Available online at http://www.reidgsmith.com/The_Contract_Net_Protocol_Dec-1980.pdf.

[11] M. Horauer, B. Chen, P. Zingaretti. *Mechatronic and Embedded Systems Pave the Way for Autonomous Driving*. Available online at http://sites.ieee.org/itss/2013/08/22/y13n1/. Accessed on February 7, 2015.

[12] S. Jurvetson. *Jurvetson Google driverless car [image]*. Available online at http://commons.wikimedia.org/wiki/File:Jurvetson_Google_driverless_car.jpg. Accessed on February 7, 2015.

[13] M. M. Deza, E. Deza. *Encyclopedia of Distances*, p 521. Springer-Verlag Berlin Heidelberg, 2014. Available online at https://books.google.co.uk/books?id=q_7FBAAAQBAJ&pg=PA521&lpg=PA521#v=onepage&q&f=false.

[14] R. W. Sinnott. *Virtues of the Haversine*. In *Sky and Telescope*, Vol. 68, No. 2, 1984, p 159. Available online at http://daimi.au.dk/~dam/thesis/Sky_and_Telescope_1984.pdf.

[15] M. Sharir. *A strong-connectivity algorithm and its applications in data flow analysis*. In *Computers & Mathematics with Applications*, Vol. 7, No. 1, 1981. Available online at http://www.sciencedirect.com/science/article/pii/0898122181900080.

[16] R. Tarjan. *Depth-First Search and Linear Graph Algorithms*. In *SIAM Journal on Computing*, Vol. 1, No. 2, 1971, pp 146–160. Available online at http://epubs.siam.org/doi/abs/10.1137/0201010.

[17] E. W. Dijkstra. *A Discipline of Programming*, Ch. 25. Prentice Hall, 1976.

[18] P. E. Hart, N. J. Nilsson, B. Raphael. *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. In *IEEE Transactions on Systems Science and Cybernetics*, Vol. 4, No. 2, 1968, pp 100–107. Available online at http://ai.stanford.edu/~nilsson/OnlinePubs-Nils/PublishedPapers/astar.pdf.

[19] R. Kraujutis. *Optimal Route Search in Geographical Information Systems*. In *Journal of Young Scientists*, Vol. 26, No. 1, 2010, p 30. Available online at http://www.su.lt/bylos/mokslo_leidiniai/jmd/10_01_26_priedas/kraujutis.pdf.

[20] J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[21] C. Nilsson. *Heuristics for the Traveling Salesman Problem*, Linkoping University, 2003, p 1. Available online at http://web.tuke.sk/fei-cit/butka/hop/htsp.pdf.

[22] K Deep, H. Mebrahtu. *New Variations of Order Crossover for Travelling Salesman Problem.* In *International Journal of Combinatorial Optimization Problems and Informatics*, Vol. 2, No. 1, 2011, pp 2-13. Available online at http://www.researchgate.net/publication/50923713_New_Variations_of_Order_Crossover_for_Travelling_Salesman_Problem.

[23] D. Carlino. *Approximately Orchestrated Routing and Transportation Analyzer: City-scale traffic simulation and control schemes*, University of Texas, 2013. Available online at http://apps.cs.utexas.edu/tech_reports/reports/tr/TR-2157.pdf.

[24] D. Carlino, M. Depinet, P. Khandelwal, P. Stone. *Approximately Orchestrated Routing and Transportation Analyzer: Large-scale Traffic Simulation for Autonomous Vehicles.* In *Proceedings of the 15th IEEE Intelligent Transportation Systems Conference (ITSC 2012)*, 2012. Available online at http://www.cs.utexas.edu/~aim/papers/ITSC2012-dcarlino.pdf.

[25] D. Carlino, M. Depinet, P. Khandelwal, P. Stone. *Auction-based autonomous intersection management.* In *Proceedings of the 16th IEEE Intelligent Transportation Systems Conference*, 2013. Available online at http://www.cs.utexas.edu/~aim/papers/ITSC13-dcarlino.pdf.

[26] D. Carlino. *Spliced roads now visualized better.* Available online at http://www.aorta-traffic.org/2014/01/28/spliced-roads-now-visualized-better/. Accessed on January 25, 2015.

[27] R. Meyer. *Event-Driven Multi-Agent Simulation.* Multi-Agent-Based Simulation XV (MABS 2014). Springer International Publishing, 2015. Available online at http://cfpm.org/discussionpapers/147/event-driven-multi-agent-simulation.

[28] N. Knaak et al. *Agentenbasierte Simulation nachhaltiger Logistikkonzepte für Stadtkurierdienste.* 2004 http://edoc.sub.uni-hamburg.de/informatik/volltexte/2009/61/pdf/B_260.pdf.

[29] N. Knaak, S. Kruse, B. Page. *An agent-based simulation tool for modelling sustainable logistics systems.* In *Proceedings of the iEMSs Third Biennial Meeting: Summit on Environmental Modelling and Software*, 2006. Available online at http://www.iemss.org/iemss2006/papers/s7/225_Page_2.pdf.

[30] R. Weait. *OpenStreetMap data license is ODbL*. Available online at https://blog.openstreetmap.org/2012/09/12/openstreetmap-data-license-is-odbl/. Accessed on February 5, 2015.

[31] OpenStreetMap Wiki. *Elements* Available online at http://wiki.openstreetmap.org/wiki/Elements. Accessed on February 7, 2015.

[32] OpenStreetMap Wiki. *Map Features* Available online at http://wiki.openstreetmap.org/wiki/Map_Features. Accessed on February 7, 2015.

[33] OpenStreetMap Wiki. *Key:highway* Available online at http://wiki.openstreetmap.org/wiki/Key:highway. Accessed on February 7, 2015.

[34] OpenStreetMap Taginfo. *highway* Available online at https://taginfo.openstreetmap.org/keys/highway#values. Accessed on February 7, 2015.

[35] OpenStreetMap Wiki. *Tag:amenity=fuel* Available online at http://wiki.openstreetmap.org/wiki/Tag:amenity%3Dfuel. Accessed on May 3, 2015.

[36] GOV.UK – Government Digital Service. *Speed limits*. Available online at https://www.gov.uk/speed-limits. Accessed on January 30, 2015.

[37] Nokia. *TrafficML – XML Schema Definition*. Available online at http://traffic.cit.api.here.com/traffic/6.0/xsd/flow.xsd?app_id=DemoAppId01082013GAL%20&app_code=AJKnXv84fjrb0KIHawS0Tg. Accessed on May 3, 2015.

[38] OpenStreetMap Wiki. *Traffic Message Channel*. Available online at http://wiki.openstreetmap.org/wiki/TMC. Accessed on May 3, 2015.

[39] Transport for London. *London Streets Performance Report Quarter 1 2012/13*, p 14, 2012. Available online at http://www.tfl.gov.uk/cdn/static/cms/documents/london-streets-performance-report-q1-2012-13.pdf.

[40] J. Chang. *Making the Shortest Path Even Quicker*. Available online at http://research.microsoft.com/en-us/news/features/shortestpath-070709.aspx. Accessed on April 27, 2015.

[41] Department for Transport. *Table TRA0307, Traffic distribution on all roads by time of day in Great Britain*, 2014. Dataset available online at https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/316562/tra0307.xls [XLS]. Summarised at https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/317454/annual-road-traffic-estimates-2013.pdf.

[42] S. Abdinnour-Helm. *Network design in supply chain management.* In *International Journal of Agile Management Systems*, 1999 Vol. 1, No. 2, pp 99–106. Available online at http://www.emeraldinsight.com/doi/abs/10.1108/14654659910280929.

[43] E. Coupland. *The 'last mile' problem, by Parcel2Go.* Available online at http://www.supplychaindigital.com/logistics/3355/The-039last-mile039-problem-by-Parcel2Go. Accessed on May 19, 2015.

[44] Fubra Limited. *UK Petrol Prices for Thursday 9th April 2015.* Available online at http://www.petrolprices.com/. Accessed on April 9, 2015.

[45] S. W. Diegel, S. C. Davis, Oak Ridge National Laboratory. *Transportation Energy Databook: Edition 22*, pp 7–28. DIANE Publishing, 2002. Available online at https://books.google.co.uk/books?id=VyzXPizRoksC.

[46] L. Reich. *The Cost of Speeding: Save a Little Time, Spend a Lot of Money.* Available online at http://blog.automatic.com/cost-speeding-save-little-time-spend-lot-money/. Accessed on April 27, 2015.

[47] Department for Transport. *Table ENV0104, Average heavy goods vehicle fuel consumption: Great Britain, 1999–2010.* Available online at https://www.gov.uk/government/uploads/system/uploads/attachment_data/file/384241/env0104.xls [XLS]. Accessed on April 27, 2015.

[48] Quadrant Vehices Ltd. *Van Fuel Consumption Figures.* Available online at http://www.quadrantvehicles.com/van-fuel-consumption-figures.php. Accessed on April 27, 2015.

[49] Merriam-Webster. *Definition of free-for-all.* Available online at http://www.merriam-webster.com/dictionary/free-for-all. Accessed on April 30, 2015.

[50] Google Maps. *Dartford Crossing Tunnel Entrance Street View.* Available online at https://www.google.co.uk/maps/@51.458427,0.250475,3a,41.6y,43.63h,87.07t/data=!3m4!1e1!3m2!1sCf_WTh6RVS-fAT2Cv0rN5w!2e0. Accessed on February 7, 2015.

[51] OpenStreetMap Wiki. *Key:parking:lane.* Available online at http://wiki.openstreetmap.org/wiki/Key:parking:lane. Accessed on February 7, 2015.

[52] J. P. Snyder. *Map Projections: A Working Manual*, p 41. No. 1395. USGPO, 1987. Available online at http://pubs.er.usgs.gov/publication/pp1395.

[53] GOV.UK – Government Digital Service. *Vehicle tax rate tables*. Available online at https://www.gov.uk/vehicle-tax-rate-tables. Accessed on May 19, 2015.

[54] Investopedia. *Break-Even Analysis Definition*. Available online at http://www.investopedia.com/terms/b/breakevenanalysis.asp. Accessed on May 24, 2015.

[55] GOV.UK – Government Digital Service. *VAT rates*. Available online at https://www.gov.uk/vat-rates. Accessed on June 11, 2015.

[56] gov.je – Information and public services for the Island of Jersey. *GST liability of goods and services*. Available online at http://www.gov.je/TaxesMoney/GST/GSTCustomers/Pages/PayingGSTJersey.aspx. Accessed on June 11, 2015.

[57] A. Levandowski, A. Schultz, C. Smart, A. Krasnov, D. Song, H. Lee, H. Chau, B. Majusiak, F. Wang. *Autonomous Motorcycle Platform and Navigation – Blue Team DARPA Grand Challenge 2005*, DARPA, 2010. Available online at http://www.cs.duke.edu/courses/spring06/cps296.1/handouts/BlueTeam.pdf.