LONDON'S GLOBAL UNIVERSITY

Demand Responsive Transport Algorithms

Implementing practical route planning algorithms

for demand responsive bus services

Candidate code: XGTG5 [1]

BSc Computer Science

Supervisor: Robin Hirsch

Submission date: 23rd April 2023

[1]**Disclaimer:** This report is submitted as part requirement for the BSc Computer Science at UCL. It is substantially the result of my own work except where explicitly indicated in the text. The report will be distributed to the internal and external examiners, but thereafter may not be copied or distributed except with permission from the author.

Abstract

Demand responsive transport (DRT) provides a modern solution to rural mobility. User travel requests can be received in real time, via an Uber-style interface, and a route for a fleet of vehicles can be calculated in real time, which serves all users' needs.

This project aims to create the basis for a system to calculate these routes.

OpenStreetMap road data is imported and modelled as a graph. Inputs, in the form of a series of coordinates of locations at which travellers will board or alight, are processed into an "abstracted" graph, with the edge weights calculated using A* search. A hill-climbing based Travelling Salesman Problem (TSP) solver finds a (close to) optimal ordering of the locations to be served. The resulting route is then be returned.

This algorithm is made available as a JSON API, to enable it to be incorporated into a wide variety of route planning and resource allocation systems.

Contents

Chapter 1

Introduction

1.1 Chapter outline

This chapter explains what Demand Responsive Transport (DRT) is, and the real-world political explanation of its purpose.

It then outlines a high-level plan for what the project will entail.

1.2 The problem

Public transport in rural areas is slow, irregular, and unreliable. Why?

Gavin Booth, in "What Bus Passengers Want" (from the Chartered Institute of Logistics and Transport) gave the salient and pithy statement that "The needs of bus passengers could be simply expressed as excellent service at low prices." [Cil].

Only 50% of rural households are within a 13 minute walk of a bus stop with at least an hourly service, compared to 96% of urban households [Gre]. It is hard to get an absolute definition of what constitutes "excellent" service, but these figures seem to show that people in rural areas would not consider their level of service as excellent, simply by comparison with that received by urban resident.

This less-than-excellent service will likely cause less patronage, causing less funding, requiring increased pricing. Poor service at higher prices causes even less patronage, causing a downward spiral of diminishing public transport in rural areas.

1.3 Potential solutions

1.3.1 Increased funding

The Transport Act 1985 recognised the fact that there would be areas that commercial operators would not serve due to a lack of expected profits, so it gave local authorities

the statutory duty to provide funding to these "socially necessary services" [Par85].

The subsidy for a given service could be increased to have a service than runs every 1 hour, as opposed to every 2. This would be an improvement, but still not as good as more densely populated areas. How frequent would be "frequent enough"?

In 2021, the government announced an additional £3bn of funding for public transport, under the "Bus Back Better" scheme [Tra21]. This money has already been distributed and, by the time it reaches the level of individual services, hadn't caused the startling improvement one might have hoped for from a lump sum that could cover $\frac{1}{3}$ of the cost of the pay-rise that UK nurses are requesting at the time of writing [Gua22].

Additional funding is something that would benefit any bus service operated by any entity, but it isn't a good solution to the problem of infrequent rural services.

1.3.2 Replan routes

Lots of these infrequent bus services are tendered by local councils. This means that the routes being operated are not planned by the operators (the people actually on the ground, interacting with passengers) but by politicians who may never have visited the communities that these routes are serving.

In addition, the effects of the financial crisis have caused the total budget for Highways and Transport services to fall from £6.6B in 2007 [Hou10] to £4.3B in 2018 [Hou18]. This is a fall of 58% in real terms [Eng]. This lack of funding, post financial crisis, has meant far fewer transport planners being employed by local authorities. This in turn has caused "route rot" where demand has shifted and services haven't pivoted to meet it.

To address this, bus operators could be given more autonomy over the routes they are operating. This would put the power in the hands of people *close* to the demand, who will be better placed to make adjustments to meet changing needs.

However, bus operators don't have to act in the interest of the public and, in the case of privately owned companies, must act in the interest of their shareholders, which means maximising profit. This is usually closely aligned with customer needs, as happy customers purchase more services, but not always.

If there was a package of routes put out for tender, with 1 route out in the middle of nowhere that was tricky for the operator to serve, they could suggest (or make the decision solo, depending on the level of autonomy handed out) that this route be cut. They could make arguments that the resource wasn't needed there, and that there was a potential route with much more demand, that conveniently finished right by their depot. And the small community "in the middle of nowhere" would have lost their only low-cost

powered transport option.

This is not to say that bus operators are evil, or in the business of disappointing their customers, quite the opposite, but they are in the business of making money. And rural public transport is subsidised to the point of it being almost entirely a public service, which sometimes means taking slightly counterintuitive decisions when viewed through a purely financial lens.

1.3.3 Demand-responsive transport

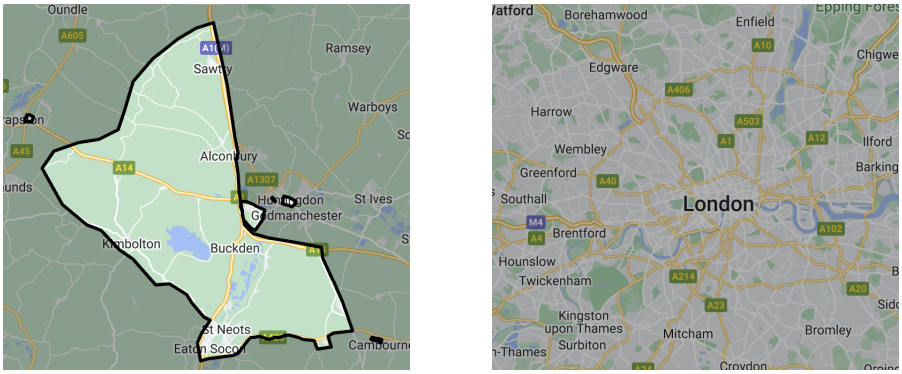
What if we could take a small fleet of vehicles, currently operating infrequent services along rigid routes, and have them serve the exact stops that passengers want to travel from and to. Not using historic travel data to host a monthly planning session. Not even taking pre-bookings up to the night before travel. In real-time, minutes before travel occurs.

This would represent no increase in PVR (Peak vehicle requirement), no increase in admin during the tender process, and provide a vastly superior offering to customers, who would pay bus prices for taxi service.

This is not a new concept. In the early 2000's Lincolnshire CallConnect had one of the first "dial-a-ride" services. They had a call centre with people taking bookings, and a team of transport planners rerouting buses manually on massive paper maps. This was, however, very costly, and not a solution available to all local authorities in all areas, even at the time. Post financial crisis, this is even more unattainable (although CallConnect has continued existing in various forms up to the time of writing).

In 2023, however, everyone carries a device that can send an operator their desired route with a far higher precision than a phone conversation could have managed, with zero staff involved. Advances in hardware and open data have transformed the task of routing vehicles from an error-prone manual process, to a very well understood automatic one. And drivers can have navigation on a smartphone, updating in real-time, without them ever needing to know what their old route was going to be and what has changed.

Figure 1.1 shows an 4-vehicle DRT operation near Huntingdon, in comparison with Greater London at the same scale.



(a) Ting zone (4 vehicles) (b) Comparable area in London

Figure 1.1: Comparing a 4-vehicle DRT operation with a comparable area in London

Creating a useful set of fixed routes to serve this area would be an almost impossible task, however the Ting DRT service has been successfully taking people exactly where they want to go for several years.

1.4 Technical solution

There are existing players in the DRT space, most notably Via, who have a lot of deployments in the USA. However, every existing solution is a closed platform, requiring operators and drivers to deal with separate apps and management tools for each different technology provider.

Creating a routing API which can be used by existing operations management systems represents the "last mile" of DRT technology, making it possible for drivers and operations controllers to seamlessly switch between operating pre-planned routes, and demand-responsive ones.

An Uber-style interface for passengers, an admin interface for operations staff, and a driver app are all required for a full end-to-end solution to be enacted. However, app and web design and development are not the focus of this dissertation, so I will focus on the algorithmic core of the project that takes on the job of the transport planners.

1.5 Aims and objectives

This project aims to investigate and implement algorithms for planning bus routes, in response to demand.

This project aims to fulfil the following objectives:

1. Research shortest path algorithms, potential optimisations, and heuristics.

2. Research solutions to the travelling salesman problem (TSP), their performance as the size of network scales, and metaheuristics that can be employed to produce real-world results.

3. Research the vehicle routing problem (VRP), how it is different to the TSP, and how it can be solved or approximated.

4. Investigate the availability of, and interaction with, OpenStreetMap data to use as the basis for a graph representation of the UK road network.

5. Develop a system of algorithms that can give approximate answers to the VRP on a reasonably-sized (UK road network) graph.

6. Develop an API/web service that accepts real-world (longitude and latitude) VRP inputs as requests, and returns vehicle routes for the UK road network, using the algorithms laid out in objective 5, and the OpenStreetMap data referenced in objective. 4

7. Evaluate the outputs produced by the VRP algorithms, against vehicle routes produced by human schedulers.

1.6 Working methodology

The objectives lend themselves to an iterative approach. I will develop the core of algorithms on test data, before bringing in real OpenStreetMap data, and finally building the web service.

Chapter 2

Background

2.1 Chapter outline

This chapter gives the theoretical basis for, and history of, real-world routing problems and graph theory.

It begins with a background on algorithms and graphs, then goes in depth on the two main problems solved in this project: the shortest path problem and the travelling salesman problem.

2.2 Algorithms

2.2.1 Algorithms vs Heuristics

Algorithms are step-by-step procedures for solving a problem. They are (supposed to be) guaranteed to produce a correct result, and usually have proofs accompanying them.

Heuristics sacrifice some guarantees for the sake of some other efficiency (usually time or space complexity). For example you might accept a 5% chance of an inaccurate result for a heuristic that is 90% faster than an equivalent algorithm.

2.2.2 Complexity Classes

A **complexity class** is a way of grouping problems by how "complex" they are to solve. There are lots of types of measures of complexity which are used to group problems, some which are very often discussed in computer science are P, NP, and their related groups.

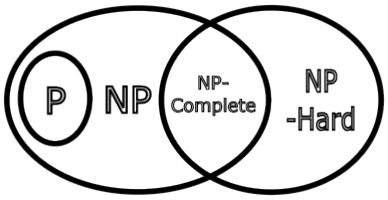


Figure 2.1: Venn diagram of complexity classes

P, also known as **PTIME**, contains all **decision problems** (problems with Yes/No answers) that can be solved in polynomial time. If the relationship between the size of the input to a problem and the time taken to solve it can be described as a polynomial function, then the problem can be solved in **polynomial time**. For example, performing insertion sort on a list with n elements requires at most n^2 steps (of arbitrary but fixed size).

NP (nondeterministic polynomial time) contains all decision problems that can have positive answers (Yes's) verified in polynomial time. P is contained within NP.

A problem, $p1$, is a member of **NP-hard** if every problem in NP can be transformed into $p1$ in polynomial time.

If a problem is a member of both NP and NP-hard, then it is a member of **NP-complete**. Their membership of NP means that solutions can be verified quickly, but not found found quickly. Their membership in NP-hard means that every problem in NP can be transformed into them in polynomial time. And their membership in both means that every problem in NP-complete can be transformed into any other problem in NP-complete.

It is not known whether every member of NP is also a member of P. This is an open problem in computer science, referred to as P = NP, and is 1 of the 7 Millennium Prize Problems posed by the Clay Mathematics Institute [Coo01], with $1 million of prize money attached.

2.3 Graphs

2.3.1 Origin

A **graph**, in the mathematical sense, is a way of representing interconnected entities. They were first conceived of by Euler, in order to solve the Königsberg bridge problem.

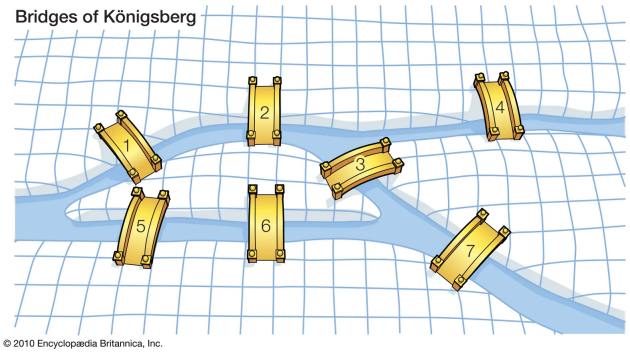


Figure 2.2: The Königsberg bridges [Britannica]

Königsberg was a Prussian city that had 7 bridges crossing it, and the people of the city had a long-standing challenge to cross all seven bridges exactly once whilst walking through the town.

Euler, in his typical manner of founding fields of mathematics, noted that the bridges could be abstracted to a series of "nodes" and "edges" whilst maintaining the relationships key to the problem.

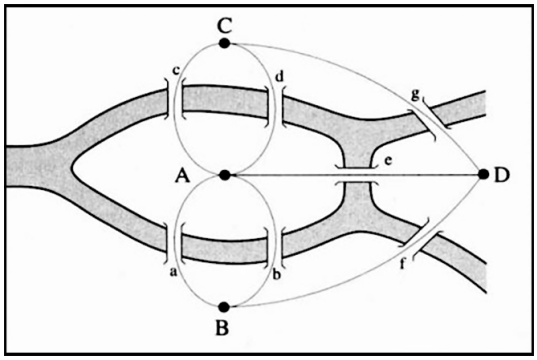


Figure 2.3: The Königsberg bridges (abstracted) [Weber State University]

He then observed that in any walk between distinct points (visiting the same point multiple times is allowed), the start and end vertices have an odd number of crossings, and the rest have even. This is a requirement, because if I enter a node I must exit it (unless it is the start or the end), giving an even number of edge intersections. If you have a walk that starts and ends at the same node then the 2 odd numbers of edges add to an even number. So a walk can either have all even vertices, or exactly 2 odd vertices and the rest even.

All 4 nodes in the Königsberg bridge graph have an odd number of edges, so Euler proved that the challenge was unfortunately not possible. He presented this finding in a paper to the Academy of Sciences [Eul41].

A **trail** (a walk without repeated edges) in a finite graph that visits every edge exactly once is now referred to as an **Eulerian trail** or **Eulerian path**, in reference to this paper.

And an Eulerian trail that starts and ends on the same node is termed an **Eulerian circuit** or **Eulerian cycle**.

The field of graph theory has grown into a very vital field in mathematics and computer science, and is at the core of a litany of innovations, including packet routing for the Internet, scheduling complex systems, search engine ranking, and of course routing journeys [Iñi23].

This project will represent real-world geographies, and passenger journeys, as graphs, and use a number of algorithmic techniques to implement the solution laid out in the introduction.

2.3.2 Formalism

Undirected graph

An **undirected graph** can be defined as

$$G = (V, E)$$

Where

- V is a set of vertices, and
- E is a set of edges, $e = (v_1, v_2)$, where $v_1, v_2 \in V$

If a pair of vertices, (v_1, v_2), is in E, then there is a *bidirectional* edge between v_1, and v_2.

Directed graphs

A **directed graph** has the same structure, $G = (V, E)$, however a pair of vertices, (v_1, v_2), only denotes an edge in one direction, from v_1 to v_2.

Weighted graphs

A **weighted graph** can be directed or undirected. It is defined as

$$G = (V, E, w)$$

Where

- V, and E are defined as previously
- w is a function $w : E \to \mathbb{R}^{\infty}$, where the output of the function is a rational number giving the weight of the edge

2.3.3 Representing Road Networks As Graphs

A road network maps onto a graph very neatly; as can be seen in their very first application above.

The graph used will be **weighted**, as the distance between two nodes on the road network is not consistent, and is relevant to the calculation. In future the weights could be replaced with the time taken to get between nodes, if this information was able to be collected.

The graph will be **directed** because of roundabouts, one way roads, priorities, and other oddities of reality that make travelling the same stretch of road in opposite directions take more or less distance/time.

There is also the question of "resolution". This is really a consideration for implementation, but it's useful to be aware of the vague shape of the graphs at play when considering the algorithms.

In the Königsberg problem Euler was able to abstract away all the parts of the graph not relevant to solving the problem.

In the datasets used for this project, roads are represented with twists and turns in, so that they look accurate to the ground. The fact that the input graph will be "high resolution" enough to accurately geographically represent the road network (as opposed to abstractly representing links between a smaller number of roads, e.g. intersections) means that the weights for the edges of the graph can be suitably accurately calculated by taking the "as the crow flies" distance.

This does however lead to lots of nodes with only 2 edges. These can cause a lot of unnecessary calculation exploring them, when they are equivalent (in terms of routing) to a line between the first and last node in a series of 2-edge nodes. There are techniques to deal with this, discussed in 2.4.8 and 2.4.9.

2.4 Shortest Path Algorithm

2.4.1 Motivation

Before we can determine the "best" order in which to visit our passenger's origins and destinations, we need to know how far apart they are.

In the source data, all of the nodes will have longitudes and latitudes attached to them, making it possible to determine the crow's distance between any two points, but we want to follow the road network.

This is an instance of the **shortest path problem**, to which there are a number of approaches.

2.4.2 Comparing Shortest Path Algorithms

The first thing to consider when considering shortest path algorithms (or heuristics) is what type of graph we're working with. As discussed earlier, the road network will be represented by a weighted, directed graph.

It's also important to note that the graph will have **cycles**, there will be trails that take you from a node, back to that same node, without backtracking.

The algorithms each solve subtly different issues, so it's important to consider exactly what problem needs solving.

And lastly speed of execution (and space used, to some extent) is important.

2.4.3 Bellman-Ford

Bellman-Ford is useful if the graph has negative weights, but it is much slower than Dijkstra [Abu+20]. The road network doesn't have negative weights, so this is doesn't seem worth pursuing.

2.4.4 Floyd-Warshall (all pairs)

This algorithm solves the "all pairs shortest paths" problem, i.e. the distance between every pair of nodes on the network. This could be extremely useful in some applications, but would be very impractical on the scale of the millions of nodes of the road network.

In order to achieve it's increased scope, it has a much higher time complexity than Dijkstra: $O(V^3)$ vs $O(E \log v)$ [Gee19].

2.4.5 Goldberg-Radzik

The Goldberg-Radzik algorithm was published in a paper, Shortest paths algorithms: Theory and experimental evaluation [CGR96], that empirically compared various shortest path algorithms, including a novel one.

The conclusion of the paper highlighted the fact that there is no single best shortest path algorithm, and that even their empirical analysis was done on "on restricted classes of graphs and small problem sizes". They did, however, suggest two algorithms: one for networks with negative arcs, and one for networks without.

For networks with negative arcs, they recommended the Goldberg-Radzik method, created by the authors of the paper.

For networks without, they recommended Dijkstra.

2.4.6 Dijkstra's Algorithm

This is probably the most famous shortest path algorithm, the most famous graph algorithm, and one of the most famous algorithms full stop [Icp].

It was conceived of in 20 minutes over a coffee, and the proof of its correctness fits on 2 sides of A4 (minus title and references) [Dij59].

The process involves using a breadth first search, using a priority queue prioritising nodes closest to a defined source point [Bae22].

The time complexity depends on the data structures used to store the current state of the algorithm, but it is commonly quoted as $O(E + VlogV)$.

2.4.7 A* Search

Whilst Dijkstra (and variations and optimisations thereof) seems to be the front-runner in the world of *algorithms*, we can achieve much higher speeds with a **heuristic**.

The variety of different implementations of Dijkstra, A*, and other shortest path heuristics muddies the water about which algorithm a piece of code is implementing. As such, this section essentially refers to any shortest path heuristic that is a version of A*, or builds on Dijkstra.

A* is very similar to Dijkstra, with a slight alteration to the priority queue. Instead of selecting the next node to visit solely based on which is closest to the origin, it will also try to take into account which is heading in the correct direction for the destination [HNR68].

Take towns/cities UK as an example, routing from London to Edinburgh, say that we have determined that it is 60 miles to Cambridge. The next closest node on our graph may be Norwich. Looking at this on a map, however, it seems very unlikely that Norwich will be part of our shortest path to Edinburgh. A human intuition might suggest trying Peterborough next. A* tries to mimic this intuition by prioritising nodes which are closer (by some heuristic) to the destination node, even if they may be farther from the source.

This "reprioritisation" means that A* can potentially be an order of magnitude or two faster, but it's heavily dependent on the specific structure of the data in the use case.

2.4.8 Highway Hierarchies

This is not a shortest path algorithm in itself, rather a preprocessing technique that can dramatically speed up methods like Dijkstra and A*, particularly when the base data is a road network.

Road networks have millions of nodes, which aren't able to be practically visited, but they're also not likely to be useful.

As humans we relatively intuitively know to head from smaller roads to larger roads

in order to reach our destination faster. We can achieve a similar behaviour in a shortest path algorithm by defining multiple layers of graphs [SS06],

The lowest layer is the complete road network, which we very quickly try to exit by getting to the closest node in the layer above (e.g. from a residential street to a dual carriageway). Then, depending on the distance of the journey, we may want to reach a motorway.

At the end of the motorway portion of the route, it is then unclear where to go, so we simply do the same process in reverse. Starting from the desired origin, seek the "entrance point" of the "neighbourhood" with the next highest level until both the origin and destination searches meet.

In order to achieve this, there is a preprocessing step that has to be done, to determine the levels and neighbourhoods of the input graph.

N.B. it is interesting to note that the algorithm doesn't have to be informed about the priority, capacity, or even speed limit of the roads it is analysing, in order to (almost always) correctly determine that a road we refer to as a motorway is of the highest level of importance.

2.4.9 Contraction hierarchies

As mentioned in 2.3.3, roads represented in graphs tend to have lots of nodes all in a line, connected only to the previous node and the next node. To a human, it is obvious that they form a line, but Dijkstra (or A*) will independently calculate the path for each node one by one.

Contraction hierarchies extend the concept of highway hierarchies by giving all nodes priorities (as opposed to a few groups of nodes). They then add edges to the graph, bypassing lower importance nodes (unless those nodes are the origin/destination) [Gei+08].

Counterintuitively, adding edges to the graph makes the algorithm faster. Similarly to highway hierarchies, contraction hierarchies require a preprocessing time.

2.5 Travelling salesman problem

2.5.1 History

In 1857 William Rowan Hamilton presented his Icosian Game at a meeting of the British Association in Dublin. The aim of the game was to find a path around a dodecahedron which visits each corner exactly once [Dar23].

Figure 2.4: The Icosian Game [David Darling]

One year prior, Reverend Thomas Kirkman posed a more general problem about visiting every vertex in a polyhedron [Kir56]. However, Hamilton's work ended up being better known, causing his stating of the problem to be known as a **Hamiltonian cycle**.

Karl Menger (son of the economist *Carl* Menger) studied what he referred to as the "Messenger problem" [Men32] in the 1920s, which was very close to what the modern understanding of the TSP is. He noted that *"The rule, that one should first go from the starting point to the point nearest this, etc., does not in general result in the shortest path."* I.e. that a greedy algorithm is not optimal.

It was first referred to as the Travelling Salesman Problem in a 1949 paper [Rob49] by Julia Robinson who was working for RAND corporation. Notably for this project, the problem was introduced to RAND corporation by Merrill Flood in the 1940's, who had first started working on it to solve a school bus routing problem.

In 1962 Proctor & Gamble published a competition asking for the shortest possible route through 33 US cities, starting and finishing in Chicago, offering a $10,000 prize. This was the first of several contests based on the TSP offering large cash prizes, highlighting how commercially valuable this piece of applied mathematics is.

Figure 2.5: Procter & Gamble contest, 33-city road TSP, 1962 [University of Waterloo]

Several people tied for first place in this competition, including George Dantzig, Ray Fulkerson, and Selmer Johnson, whose work on the problem proved invaluable to future researchers. David Applegate, a Google researcher who helped solve an 85,900 city tour in 2009 [DLA09], said *"Dantzig, Fulkerson, and Johnson showed a way to solve large instances of the TSP; all that came afterward is just icing on the cake".*

Their groundbreaking approach used the simplex algorithm developed by Dantzig,

which took a series of linear rules and optimised a given variable, by encoding instances of the TSP as a series of linear rules. However, the 33 city tour would have produced over 2 trillion rules, which would have had to be calculated by hand. To resolve this, they took an unrestricted model of the problem, then only added rules based on human intuition (and excellent comprehension of the problem). This resulted in a stunning 9 rules that produced an optimal solution [DFJ54]. This method was formally known as the "cutting-plane method", because each rule added cuts away a portion of the possible answer space that is being optimised.

As several people found the same best result, the winner of the $10,000 prize came down to an essay competition about the benefits of one of P&G's products, which was won by mathematicians Robert Karg and Gerald Thompson, who implemented one of the first electronic TSP solvers. It turned out that, whilst their method didn't guarantee an optimised tour, they had in fact found one. Karg and Thompson later decided to create a larger test graph with 57 cities (with which they came within 30 miles of the optimal route [Wat16]), which became a common test instance for future researchers.

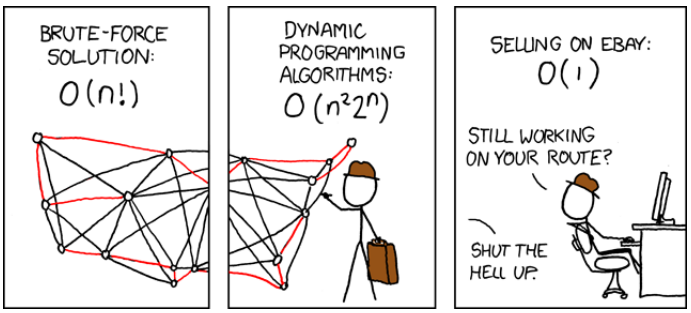
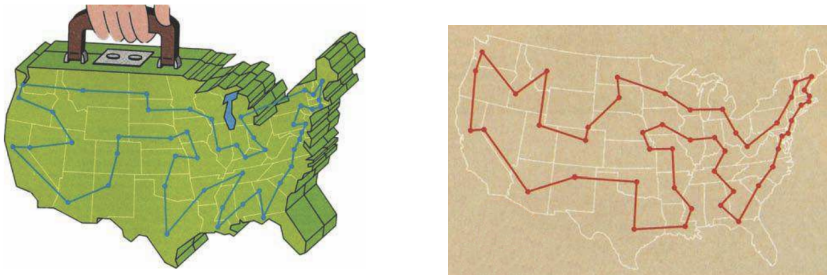


Figure 2.6: XKCD 399 Travelling Salesman Problem [XKCD]

Very little happened in TSP research until 1971 when the Held-Karp algorithm was published. It used a dynamic programming approach to quickly solve the original P&G problem, found the optimal solution to Karg and Thompson's 57 cities, and a random 64-city instance. This algorithm had a time complexity of $O(n^2 2^n)$ (a dramatic improvement on the brute-force $O(n!)$ but still exponential). Karp would go on to receive the Turing award in 1985 for contributions to the theory of algorithms.

In 1985 an article was published about the TSP in Discover, accompanied by figure 2.7(a), which was stated to be the shortest such tour. Unfortunately the ordering used was from Dantzig et al., which was not a tour through the state capitals but rather a (specific) city in every state. To rectify his embarrassing editorial error, Martin Gardner (who wrote the article) tried contacting various TSP experts until he got into contact with Shen Lin.



(a) Incorrect tour (April 1985) (b) Correct tour (July 1985)

Figure 2.7: USA state capital tours [Discover magazine]

Lin's approaches to the problem were iterative in nature, taking an initial tour (either random, or an intuitive best guess) and then finding rules to improve that tour. He went on to work with Brian Kernighan to produce the Lin-Kernighan method. These rules have proved invaluable in modern approaches to the TSP, and are expanded on in section 2.5.5.

Most further developments used modifications and combinations of cutting-planes, and dynamic programming (Held-Karp), or metaheuristics.

In 1987 Grötschel and Holland worked on 666 cities (the number of the beast, representing a "beastly challenge for TSP computations") using a hybrid of cutting-plane and general integer programming [GH91].

In 1991 Gerhard Reinelt published TSPLIB, a collection of 100+ TSP instances from various sources, to give a benchmark against which researchers could compare their work.

In 2000 the DIMACS TSP Challenge was announced [DIM01], calling for the global academic community to collate and compare their best algorithms and implementations against the TSPLIB, as well as randomly generated instances (in the TSPLIB format). This is still an invaluable resource when developing new TSP solvers today.

In the late 90's the Concorde solver [Wat13a] began to be developed by David Applegate, Bob Bixby, Vašek Chvátal and William Cook. It would go on to solve all the remaining TSPLIB instances, including an 85,900 city tour [Wat13b] for optimising the route of a laser used in circuit manufacturing (see fig 2.8).

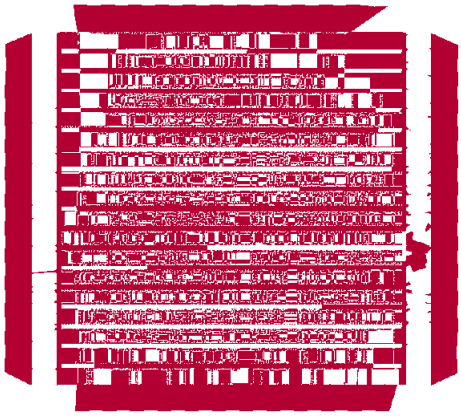


Figure 2.8: Optimal path for the TSPLIB pla85900 instance [University of Waterloo]

The current challenge for the TSP community is the World TSP, an instance with 1,904,711 cities from across the globe. The current (at time of writing) shortest tour is 7,515,755,956.

Also, Robert Bosch has produced 6 TSP instances, from 100K-200K cities, which produce line drawings of famous pieces of art [Wat14]. The results are rather beautiful (see fig 2.9)

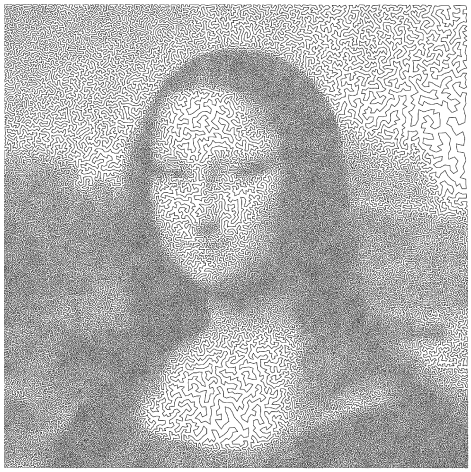


Figure 2.9: Mona Lisa 100K, solution by Yuichi Nagata [University of Waterloo]

2.5.2 TSP and variations

In graph theory a **walk** is a sequence of edges which join a sequence of vertices. A **trail** is a walk with all edges being distinct. A **path** is a trail with all vertices being distinct. A **cycle** is a trail with all vertices being distinct, except for the first and last which must be equal.

An **Eulerian trail** (sometimes incorrectly termed an Eulerian path) is a trail in graph

which visits every edge exactly once. An **Eulerian cycle** has the equivalent definition for a cycle.

A **Hamiltonian path** is a path in a graph which visits each vertex exactly once. A **Hamiltonian cycle** has the equivalent definition for a cycle.

The **Travelling Salesman Problem** (TSP) is less cleanly defined, as the problem being solved varies from implementation to implementation, however it essentially asks for the Hamiltonian path/cycle on a weighted graph with the *shortest* total weight (typically distance). The vertices on a graph in the TSP are commonly referred to as **cities**, due to early research focussing on specific test data sets. The resulting path/cycle/route is commonly referred to as a **tour**.

In the **Symmetric TSP**, the distance between two cities is the same in each direction, i.e. the graph is undirected. In the **Asymmetric TSP**, the distance between two cities can be different in each direction, i.e. the graph is directed.

In the **Metric TSP**, the distances between cities satisfy the triangle inequality. I.e., the distance AB must be shorter than (or equal to) the sum of the distances AC and CB. This restriction naturally maps onto the real world, and is a requirement for various algorithms/heuristics/methods.

The **Euclidean TSP** is a special case of the Metric TSP where the distances between the cities are calculated using the Euclidean distance.

The **Vehicle Routing Problem** (VRP) is a generalisation of the TSP, where multiple "salesman" (typically vehicles) can be used, to parallelise the process of visiting the cities. This problem is essentially what shipping companies have to solve in order to efficiently deliver packages.

Beyond the VRP there are numerous variations which try to take into account additional factors, such as **Vehicle Routing Problem with Pickup and Delivery** (where visiting city A necessitates the same vehicle visits city B), **Vehicle Routing Problem with Time Windows** (where specific pickup/drop off times are imported, such as a scheduled taxi), **Capacitated Vehicle Routing Problem** (where a vehicle can only carry certain quantity of items), and many more.

In practice, individual needs of specific businesses mean that a solution to a cross-breed of the varieties of problems studied academically is typically required. In this project I aim to produce a solution that can take into account common restrictions, and easily include more in the future if required.

2.5.3 Complexity of the problem

In 1972 Richard Karp showed that the Hamiltonian cycle problem (determining whether or not a Hamiltonian cycle exists for a given graph) was NP-complete, implying that the TSP is NP-hard [Kar72].

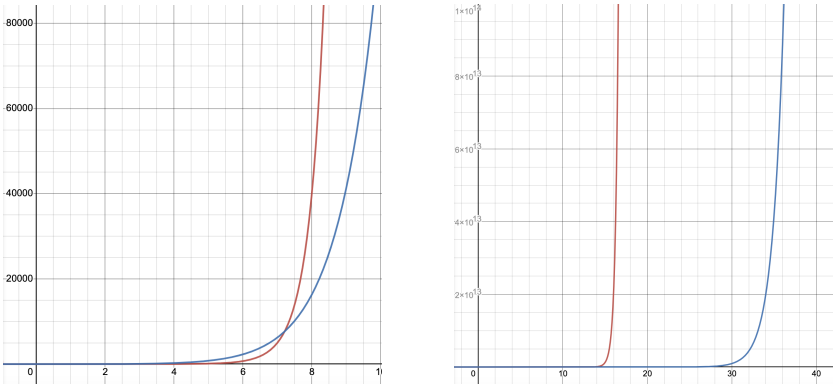
The decision version of the TSP (determining whether a tour exists shorter than a certain length) is also NP-complete.

If we want to find the shortest path visiting 4 cities, then we have 4 choices for the first city. There are then 3 cities left from which to choose the second city, then 2 for the third, and only 1 for the fourth. Multiplying these gives 24 possible combinations of cities, therefore 24 possible tours to check to find the shortest.

Generalising this we see that for n cities, the number of possible tours is $n \times (n-1) \times (n-2) \times \cdots \times 1$ or $n!$. Assuming the time taken to check a possible tour is roughly constant, the time complexity of checking every combination (i.e. a **brute-force** approach) is on the order of $n!$, or $O(n!)$.

A factorial time complexity is typically regarded as very poor and, wherever possible, necessitates a different approach: preferably one which achieves at least a polynomial time complexity. Unfortunately, we know that the TSP is NP-hard, meaning that a polynomial time solution is something that has eluded the mathematics community so far.

Nevertheless, we have solutions better than factorial time. The Held-Karp algorithm produced a time complexity of $O(n^2 2^n)$, offering a massive improvement (see fig 2.10).



(a) $n! < n^2 2^n,\ 0.7408 \lessapprox n \lessapprox 7.2074$

(b) $n! \gg n^2 2^n,\ n \to \infty$

Figure 2.10: Brute force (red) vs Held-Karp bound (blue)

The cutting-plane method pioneered by Dantzig et al., and taken to its limits by Concorde, is the most effective algorithm currently published, however determining its time complexity has proven very tricky. Chvátal, Cook, and Hartmann proved in 1989 that a strong variant of branch-and-cut requires at least $\frac{2^{\frac{n}{72}}}{n^2}$ operations to solve a deliberately

adversarial instance of the Hamiltonian-circuit problem [Coo12]. Other TSP instances may require even more steps, however there haven't been large amounts of work done to determine this.

2.5.4 Heuristics

(Almost) all of the incredible work done by mathematicians from Reverend Thomas Kirkman, to Dantzig et al., to Applegate et al. has been focussed on finding the absolute best solution, to specific test datasets, sometimes with incredible amounts of computing power, and relatively relaxed time constraints.

This project needs to handle random sets of test data, 100s (or 1000s or 10,000s) of times a day, in as close to real time as possible, on affordable enough hardware that the DRT services have a chance of turning a profit. But it doesn't need a perfect solution: it needs a good solution.

These requirements mean that a heuristic is better suited for this project than an algorithm.

The Christofides Algorithm [sic] is a heuristic which uses Euler's original observations about the bridges of Königsberg. You add edges that meet every odd vertex exactly once (a perfect matching), producing a graph with all even degrees, making an Eulerian path possible. This path can then be shortcutted into a tour.

The most time-consuming process is producing the perfect matching, for which there are a few different implementations. Two common ones give time complexities of $O(n^3)$, and $O(n^2 \log n)$. These are a staggering improvement over any algorithm guaranteeing a perfectly shortest tour [OT22].

A tour produced by the Christofides Algorithm is guaranteed to have a total distance no more than one and half times the optimal tour [Chr22]. This is known as an **α-approximation** with $\alpha = 1.5$. Since its publication in 1976 the first potential improvement, giving $\alpha = 1.5 - 10^{-36}$, was released in 2020 [KKG21], however this has not yet been peer-reviewed.

2.5.5 Local search heuristics

Another heuristic approach tries to take an existing tour and gradually improve it. At the core of these approaches are "opts".

First floated by Merrill Flood in 1956 [Flo56], and formalised by G. A. Croes in 1958 [Cro58], a **2-opt** is an operation on a TSP tour which takes 2 edges and swaps their connections, as seen in fig 2.11.

The **2-opt algorithm** is a relatively naive local search algorithm which consists of looking for 2-opt moves that improve a tour until no more can be found. In worst cases

finding an optimal it could have $O(2^n)$, however this is highly implementation dependent and a close-to-optimal tour can usually be found relatively quicker.

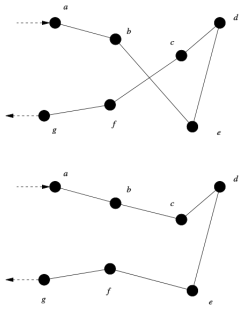


Figure 2.11: Example 2-opt move [Wikipedia]

A **3-opt** operation involves deleting 3 edges, producing 3 sub-tours; there are 7 ways that these can be reconnected (see fig 2.12). The length of each of the possible 7 tours is found, and the shortest one selected. Similarly, the **3-opt algorithm** involves iteratively applying the 3-opt until no better paths can be found. A single iteration of the 3-opt algorithm has a time complexity of $O(n^3)$.

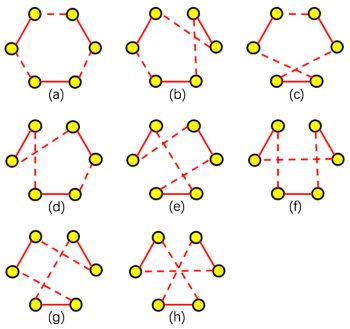


Figure 2.12: Possible 3-opt moves [Cho+21]

A k**-opt** operation extends the ideas above, by deleting k edges from a graph and iterating through all the possible ways to reconnect them, finding the best path [Mah].

2.5.6 Metaheuristics

Motivation

The Christofides Algorithm (and it's variations) is a very powerful tool, however it specifically optimises for length of tour. Almost no work has been done adding in additional constraints (such as vehicle capacity, or time windows).

Some attempts have been made at adding these types of constraints to the dynamic programming [Dum+95] and cutting planes [Coo99] algorithms. Looking at the ap-

proaches taken, each different constraint requires a restating of the problem, and a re-working of the maths behind the solution.

In a transport company, the requirements desired to be optimised will change and grow as the company does, as well as when regulation changes. Some requirements are quite foreseeable, such as vehicles requiring fuel after a certain distance, or drivers have restrictions on how long they can work without breaks. However some considerations may not always be important, and then be required at a later date, such as trying to make sure that a uniform set of vehicles have roughly the same number of miles on the clock, to make them easier to sell.

To achieve this we can use **metaheuristics**. These are heuristic methods that can be applied to most problems. They typically borrow metaphors from other fields.

Hill climbing

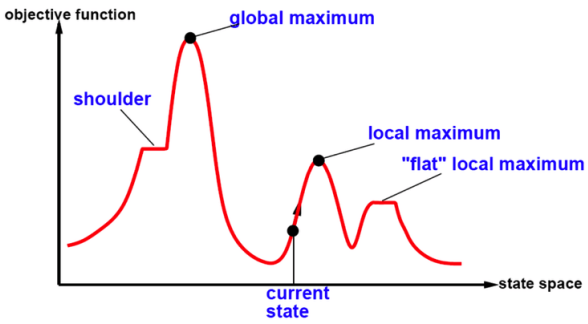


Figure 2.13: Example hill climbing graph [Bat19]

A very common metaheuristic is **hill climbing**, where one imagines a graph with all possible tours enumerated along the x-axis, and the lengths of those tours on the y-axis (typically with shorter tours at the top, so that the metaphor of reaching the peak lines up with the goal of this problem). The aim is to find the highest point on that graph.

To do this, you take a starting tour (often random) and you mutate it in some way to produce a **neighbour**. This mutation can be swapping 2 nodes (i.e. a 2-opt), reversing a subset of nodes, or any other definable operation. If the neighbour has a shorter total distance then we have ascended a small amount, and this tour is the new tour being mutated.

This process is repeated until no shorter neighbouring tour is found.

The Lin-Kernighan algorithm is a very well-known hill climbing heuristics. It uses k-opts, and produces excellent efficient results by varying k on the fly, depending on how successfully an individual opt is progressing [LK73]. Each iteration involves removing

more and more edges (simultaneously planning how to reconnect them) until a positive move cannot be found. At this point, the graph is reconstructed and a new iteration begins on the next node in the tour. Every node will end up having at least 2-opt run on it, up to an unknowable (instance specific) k-opt, different for each node. As such, an exact time complexity cannot be known, however it has been practically estimated at $O(n^{2.2})$.

Repeated 2-opts (which is what many applications of Lin-Kernighans end up being) are only guaranteed to produce a solution less than $4\sqrt{n}$ times longer than the optimum tour (on the Metric TSP), however in practice the results tend to be much better than this.

LKH (Lin-Kernighan-Helsgaun) is an implementation of the Lin-Kernighan heuristic, however the work put into it merits its independent fame. Where the classical Lin-Kernighan heuristic is typically based on the 2-opt, LKH is based on the 5-opt. This seems like a simple and obvious improvement, however it involved Helsgaun considering and optimally dealing with 148 separate cases for rejoining 5 edges. Since the release of LKH several attempts to use 6-, 7-, 8-, and 9-opt, however LKH still holds the record for all DIMACS instances with unknown optima [Lkh].

The main problem with hill climbing is that you can get stuck in **local maxima**. This is a region of the graph which is higher than any of its neighbours, but not the highest point overall.

Lin-Kernighan tries to avoid this by repeating the process several times with random starting points and taking the best resulting tour, in order to have a higher chance of finding a global maxima.

Simulated annealing

Another way to deal with local maxima is an approach called **simulated annealing**. This borrows a metaphor from metal-working where, when metal is hot its crystal structure is very malleable, then as it cools it becomes more solid.

Simulated annealing is a variation of hill climbing where we keep track of a global "temperature". When a neighbour is generated, if it is shorter than the current best tour, then it is accepted. If it is longer, there is still a chance that it will be accepted, with the probability of this chance being proportional to the temperature.

When the temperature is high, the heuristic often makes moves that give longer tours. Gradually the temperature decreases over time, heading towards a local maxima, that is hopefully also the global maxima, thanks to the random movements at the start.

Chained local optimisation

This approach also models the solution space as a landscape to be navigated. The first step is running a single iteration of Lin-Kernighan on a random tour. Then, instead of picking another random starting tour, we "kick" the current best tour.

This "kick" is designed to jump the current tour over low points in the landscape to reach a neighbouring (potentially higher) peak. The way to achieve such a kick is by performing a move that Lin-Kernighan cannot easily undo. It was found that a special type of 4-opt (see fig 2.13) achieves this very successfully [Mar91].

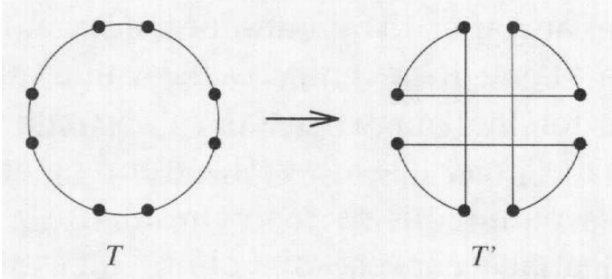


Figure 2.14: "Double bridge" 4-opt "kick" [ACR03]

A version of this approach was used in Concorde, and was able to find solutions for 100,000 city instances in 1–2 seconds which were within 1% of the optimal solution.

Genetic algorithms

This approach models the tour as living being that can evolve. An initial "generation" of tours is randomly generated, ideally with lots of diversity. Then, pairs of tours in the generation "mate" to produce a new generation of (hopefully better) tours.

There are several ways to take this loose metaphor and implement it. Ideally several chunks of each "parent" will end up intact in the "child". One very successful approach is known as edge-assembly crossover (EAX) which was developed by Yuichi Nagata.

His approach currently holds the best known tour for the 100,000-city Mona Lisa TSP (as seen in fig 2.9).

Ant colonies

This approach uses the observation that ants typically move in a single straight line. This is because they leave pheremone trails as they walk, to signify a good route. If an ant discovers an obstacle free path to food, then it makes sense for the rest of the colony to use that same route, as opposed to try to blaze their own trail.

Taking this concept, a number of agents perform separate walks of a graph, making decisions at each node, which node to go to next. This decision is informed by a pheremone

value associated with each edge. After each walk, the pheremone values are updated according with how short the walk was relative to others.

This approach hasn't proved competitive with Lin-Kernighan-based methods, however it has been applied in other areas of optimization.

Chapter 3

Requirements and analysis

3.1 Chapter outline

This chapter ties together the introduction and background chapters, in order to synthesise an actionable set of requirements from which to work.

3.2 Requirement Reasoning

This project aims to create a route optimisation tool specifically for the public transport sector. It should facilitate the production of close to optimal (by various metrics) routes through user defined pick-ups and drop-offs.

To achieve this, the road network needs to be represented as a graph, so that the graph algorithms discussed in 2.4 and 2.5 can be ran on it.

The needs of a transport company are constantly changing, and individual use cases of this tool within the same company could have different requirements. As such, the tool should be able to be easily extended to optimise for additional variables, such as passenger onboard time, or fuel usage. It should also be able to be easily extended to add restrictions on routes, for example keeping within driving hours legislation, or not leaving a certain geographic area.

Speed of execution is very important, particularly for the travelling salesman problem as the time complexity grows very quickly with the size of graph, and the road network is very large. A low-level compiled language such as C or Rust is the best fit for this.

This project will need to be accessed from web/mobile applications which act as a UI for DRT services, which are typically not written in low-level languages. In addition, while it is possible to produce these systems in C or Rust, this tool could be useful to an array of customers who will each have a different tech stack.

To facilitate this, a hosted web API will allow users to interact with the system cleanly without worrying about the implementation.

3.3 Requirements

These requirements are collated from the introduction, background, and reasoning above.

They use the MoSCoW analysis technique to prioritise development within the timeline of this project.

Each has an ID with 'F'/'N' for functional/non-functional, 'B' or 'W' for backend/web interface, and a sequential number.

3.3.1 Functional requirements

ID	Requirement
FB1	The tool must have the ability to find the shortest path between two points in a graph
FB2	The tool must have the ability to find the ordering of nodes in the graph close to the shortest length (travelling salesman problem)
FB3	The tool could have the ability to find n separate orderings of the nodes in a graph with close to the shortest total length (vehicle routing problem)
FB4	The tool must have the ability to read OpenStreetMap data so that the above algorithms can operate on it
FB5	The tool must be able to combine the shortest path and travelling salesman solvers on OpenStreetMap data, to accept passenger journey information as a pair of longitude and latitudes (pickup/drop-off) and produce a close to optimal route as a series of longitude and latitudes following the road network
FB6	The tool could have the ability to accept requested pickup/drop-off times along with journey location information, and attempt to order the pickup/drop-offs as close to the requested times as possible
FB7	The tool should have the ability to read TSPLIB data so that it can be compared against existing tools
FB8	The tool could have the ability to produce a visual representation of a graph for ease of debugging
FW1	The tool should have a web API interface, which accepts and producing JSON data representing the problem and solution (shortest path/travelling salesman/vehicle routing)

Non-functional requirements

ID	Requirement
NB1	The tool should be implemented in a language chosen primarily for speed, whilst taking into account ease-of-use, ecosystem, and community.
NB2	The tool must have a data structure to represent weighted directed graphs in memory
NB3	The travelling salesman solver should be extensible, to be able to add multiple parameters to optimise for, and add hard restrictions on outputs
NB4	The tool should have $> 90\%$ test coverage
NB5	The tool should have good documentation in the form of inline comments
NB6	The tool should have benchmarks for each problem, to check for unexpected regressions in performance
NW1	The API should have a response time of < 1s
NW2	The API should have documentation for its endpoints for ease of use by future integrators

Chapter 4

Design and implementation

4.1 Chapter outline

This chapter details the full development process of the specific tool created in this project.

It details the reasoning behind the choices made about what tools and data to use for the project. It then goes over the iterative development processes for the two main problems (shortest path and travelling salesman).

It shows how the different components fit together to create a full usable routing system, and a few important odds and ends.

4.2 Language & tools

4.2.1 Programming language

According to requirement NB1, the primary consideration should be speed of execution. A single language will never be the fastest at everything across the board, however The Computer Language Benchmarks Game attempts to give some data for comparison.

Figure 4.1 shows that C, C++, and Rust are far faster than the other compared languages [Ben], as their medians are outside the interquartile ranges of almost all others.

These 3 are close enough together that it seems reasonable to consider the other requirements of NB1: ease-of-use, ecosystem, and community.

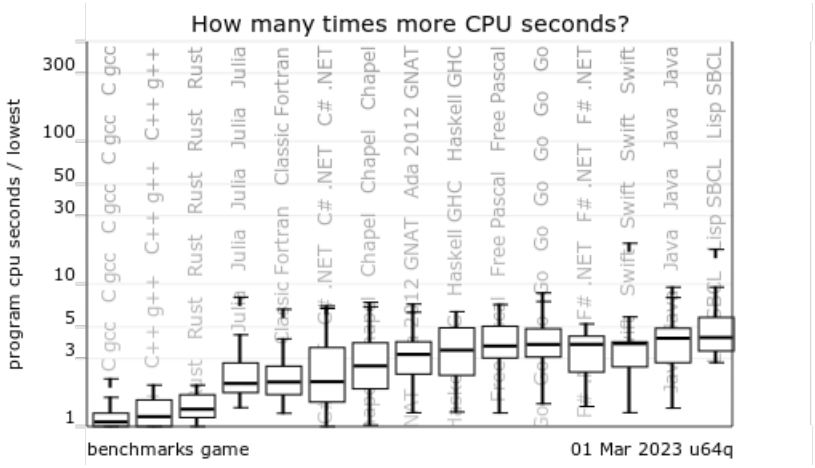


Figure 4.1: "Fastest" contributed programs [The Computer Language Benchmarks Game]

C is a very powerful language, underpinning many mature performant projects including the Linux Kernel. This power allows experienced developers to make optimisations that a compiler couldn't. However, it also provides many potential pitfalls.

C++ is an attempt to add some niceties to C that you might come to expect after working with newer languages, such as types, exceptions, and standard implementations of common things. However, if you search "programming footgun" into Google, the only language explicitly mentioned is C++. The attempts to modify C to add these features has resulted in multiple ways to do the same thing, which can cause unexpected results.

Rust, in comparison, has been designed from the start with many of these considerations in mind. Not only does it have a strict type system, like Haskell, it also has its "borrow checker". This is a concept (relatively) unique to Rust, which tracks ownership over objects, allowing for a whole new class of errors to be spotted at compile time. As a result, memory errors, segfaults, and null pointers are almost entirely removed.

As well as null pointers, the entire concept of null is not present in the language. The "null" value was introduced by Tony Hoare in 1965 in ALGOL W "simply because it was so easy to implement", and the concept was replicated in almost every language from then on (including C, Java, Python, Ruby, and many more). Unfortunately, not considering null values is the cause of many software errors, and Hoare now refers to it as his "Billion Dollar Mistake" [Hoa09]. To fill this role, the Rust stdlib provides enums such as `Option`, which either contains `Some(T)` or `None`. `None` may sound like null, but, as it is a value of the `Option` type, the compiler requires that you deal with the potential of the `None` value wherever an `Option` is returned.

The `Box` struct effectively replaces `malloc` and `free`. It allocates memory on the heap, which can be accessed exactly as you would expect without `Box` being there. And when the variable holding the `Box` goes out of scope, the heap space is automatically freed.

These features, along with many other elegant design decisions, make Rust much easier to use (purely from a code style perspective) in my opinion.

The Rust ecosystem is neatly encapsulated within the `cargo` command line tool included with your Rust installation. You use `cargo` to build and run your code. It also runs your tests, automatically discovering them from various locations, including central test folders, tests alongside implementations, and even doctests (a fantastic functionality, inspired by Python [Pyt], where code snippets including in inline documentation are ran as part of testing). `cargo` also includes a built-in code formatter (`rustfmt`) and style linter (`clippy`).

Also, `cargo` manages your dependencies. Users enter a list of package names and versions (like `osmpbf = "0.3"`) into the `Cargo.toml` file (which also includes other metadata about your project, like rust version). Then, whenever the project is run/tested/benchmarked, as part of the build process `cargo` checks that the currently installed dependencies match those specified in `Cargo.toml` and fixes any differences.

All of this functionality built into 1 tool which is bundled with the language has prevented the sprawl of duplication that many other languages suffer from. For example, the various testing frameworks for JavaScript (Jest/Jasmine/Karma/Cypress etc.), or litany of Python linters (pycodestyle/Pylint/PyFlakes etc.).

Taking into account the above, the Rust programming language meets the requirements set out in requirement NB1.

4.2.2 Version Control

Version control is a very valuable tool for software engineering, as it allows developers to "Code Fearlessly" [Cam]. Once you have a stable version, you can commit your code and then freely break things until you have something stable again.

The landscape of version control has changed dramatically over the last 20 years, going from multiple choices including Subversion, Mercurial, and Perforce, to essentially just Git [Atl]. It's branching workflow attracted enough of a market share that hosting companies started to be built around it, and this symbiosis has caused it to be completely dominant. As such I am going to use Git.

Leading on from this, as well as the version control system used, the host is also an important decision. Hosts like GitHub, GitLab, and Bitbucket now offer much more than just git hosting, including project planning and CI/CD. These offerings, particularly for open-source software (where all services are typically provided for free) are relatively undifferentiated, so I have chosen to use GitHub because I am already very familiar with it.

4.2.3 Code Editor/IDE

Many developers use command line tools like vim and emacs, and strictly fight for their choice. A lot of newer developers (particularly in the web development space) have chosen Visual Studio Code, with almost 75% market share in the Stack Overflow 2022 Developer Survey [Sta].

Personally, I have found the suite of IDE's developed by JetBrains to be invaluable. In comparison to VS Code, each editor (whilst based on the same core) comes already setup for the language(s) it is designed for, with subtleties already taken care of. And, in comparison to command line editors, they are very easy to learn and often more powerful, particularly when it comes to refactoring.

In addition, partially thanks to the centrality of `cargo`, lots of Rust tooling integrates very nicely with the editor. For example, test runs are incorporated into the GUI, with individual test run outputs separated out, and the ability to easily rerun specific tests (as opposed to a long command line output, and having to type another command to specify a test).

Rust doesn't have a specific JetBrains IDE (like CLion for C-like languages), so it is wrapped into in IntelliJ IDEA.

The reason JetBrains editors are able to be so feature rich and finely tuned out of the box is that they are typically paid products, however they offer free licenses to students, so I will be using IntelliJ IDEA for this project.

4.2.4 Continuous integration

As mentioned in 4.2.2, I am using GitHub to host my git repository. One of the services they offer is GitHub Actions, a Continuous integration/Continuous deployment (CI/CD) service, which allows you to trigger services, and arbitrary code, to be run when certain events occur. They give 2,000 free minutes for personal accounts, which was more than enough for my purposes.

I've created a "workflow" which builds my project, runs my tests, and runs Rust's built-in linter Clippy, every time new code is pushed to a branch.

See Appendix D for the workflow file.

I'm also using a service called DeepSource. Their primary product is a set of linters for various different languages, which I enabled for Rust (as their checks cover different things to Clippy). However, they also offer a service called Transformers, which runs auto-formatting tools on code bases.

Cargo (the rust CLI) has a built in formatter called rustfmt, which is set up to run on my code every time I push. This means I don't have to worry about minor style issues like spacing and quotes, whilst producing a clean-looking code-base. This will make any

future collaboration much easier.

I will be using several Cargo crates (Rust libraries) for this project, for reading file formats, creating graphs, creating API's etc. In order to keep these up-to-date I used a service called Renovate. This runs periodically, checking for updates for the packages I depend on, and creates Pull Requests for me. I can easily tell whether these package updates will break my code, because the testing workflow runs against the created PR's.

4.3 Mapping data

4.3.1 Data source

There are several entities with high quality mapping data. Google Maps, Apple Maps, and Mapbox are all large players in the space, however they only give access to derivatives of their data. They will give you "map tiles" which are rasterised versions of the underlying network data. This is what a user sees when looking at a map on their phone/browser.

They will also give API access to routing algorithms, however these are very expensive per request (up to $0.005 per A to B route [Goo]), and they aren't able to take into account bus lanes/bus only roads.

For this project, I need access to the underlying network (or graph) which these services are doing routing on.

In the UK, the Ordnance Survey publishes their (very high quality) road network data, however this is a premium service, and also doesn't include bus specific data.

OpenStreetMap (OSM) is a project founded in 2004 in response to the Ordnance Survey not releasing its mapping data publicly. It is an open source dataset of the geography of the world, collated from various open data sources and the continued input of its community of users. (Since the creation of OSM, OS has released many of its datasets as open data, however their full road network is still a premium product).

Notably for this project, OSM roads (or "ways") have ways of denoting whether they have bus only lanes, or are bus only roads. However, there are actually several different fields for this purpose (`busway`, `lanes:psv`, `\verb`bus:lanes|, etc.) and the data is not very accurate because most users don't care about these attributes. As such, for the implementation of this project I will just use the full OSM road network, and recommend that any deployment of this tool should be accompanied with manual verification of the OSM bus-lane data of the area being operated in.

4.3.2 OpenStreetMap data handling

Format

OSM data is represented as XML, in files typically ending with `.osm`. It is designed to be able to represent almost any real world mapping feature.

It is commonly distributed in "Protocol Buffer Format" (PBF) in files ending with `.osm.pbf`. These files are about half the size of the raw XML [Osm].

Geofabrik

The central OSM dataset is stored as a PostgreSQL database, being updated in real-time. Exporting this data as PBF files, and cutting out specific regions, requires keeping a synced copy of this database maintained. Luckily, a German company called Geofabrik, maintains daily-updated downloads of regions of varying specificity, from continents to counties.

Osmium

Whilst useful, the PBF files produced by Geofabrik are not always exactly what is required. Sometimes you want a smaller region, or you may want to convert between PBF and OSM formats, as some tools only accept one or the other.

To do this, there is a command line tool called Osmium which allows you to do both of those things and more.

JOSM

It's also useful to be able to directly edit OSM data, either to fix an error, or to create custom test instances in the OSM/PBF format. JOSM is a graphical OSM editor, which facilitates just that.

4.3.3 Finding distances between points

In this project I mention the phrase "as the crow flies", or "crow-flies distances" a lot. Finding the distance between two longitudes and latitudes is not as simple as simply using Pythagoras, as this doesn't take into account the curvature of the earth.

The great circle distance, finds the distance between two points on a sphere. The

formula for this is commonly called the **Haversine distance**:

$$
\begin{aligned}
R_{km} &= 6371 \\
\Delta\text{Lat} &= \frac{\text{Lat}_2 - \text{Lat}_1}{180}\pi \\
\Delta\text{Lon} &= \frac{\text{Lon}_2 - \text{Lon}_1}{180}\pi \\
a &= \sin^2\left(\frac{1}{2}\Delta\text{Lat}\right) + \cos\left(\frac{\text{Lat}_1}{180}\pi\right) * \cos\left(\frac{\text{Lat}_2}{180}\pi\right) * \sin^2\left(\frac{1}{2}\Delta\text{Lon}\right) \\
d &= 2\arcsin\sqrt{a} * R_{km}
\end{aligned}
$$

Technically the earth is not spherical so this approach is slightly inaccurate. There is another approach, called Vincenty's formulae, however this is iterative and therefore much slower. Finding the distance between two points will have to happen for each edge that is imported, which could be millions. The Haversine formula is known to be accurate to 0.5%. In addition, it is much more consistent with points close by. As the algorithms in this project are all about finding the best result in comparison to others, if the tool thinks a distance is 0.5% different from reality, but it finds this about all the distances nearby, the resulting path will still be correct.

As such, I will use the Haversine formula to determine the distance between two longitudes and latitudes, and this will be very easy to change in the future if needs be. All references to "crows-distance" or similar, is referring to the great circle (or Haversine) distance.

4.4 Graph implementation

4.4.1 Types of graph implementation

There are 2 common representations of a graph data structure: adjacency matrices, and adjacency lists.

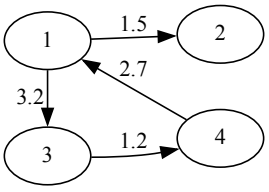


Figure 4.2: Example graph

An adjacency matrix has a column for every node, and a row for every node. For a weighted directed graph, the a value in a cell is the weight of the edge from the column node to the row node. For the graph in figure 4.2, an adjacency matrix would look like:

	1	2	3	4
1				2.7
2	1.5			
3	3.2			
4			1.2	

An adjacency list has a new record in the list for each edge on the graph. The records hold which node the edge is to and from, and the weight of the edge. For the graph in figure 4.2, an adjacency list would look like:

- $1 \rightarrow 2 : 1.5$
- $1 \rightarrow 3 : 3.2$
- $3 \rightarrow 4 : 1.2$
- $4 \rightarrow 1 : 2.7$

The two representations have different use cases, due to how their storage requirements scale.

An adjacency matrix requires storage proportional to the number of nodes in a graph. If you have 10 nodes in your graph, even if you only have 1 edge, the adjacency matrix needs 100 memory locations, 99 of which will be unused.

An adjacency list requires storage proportional to the number of edges in your graph. You could have a graph with 1,000,000 nodes, but if there is just 1 edge between node 34,985 and node 903,400 then you only need space for 1 record in memory. However, the space used to represent 1 edge for an adjacency list is not the same as the space used to represent 1 edge for an adjacency matrix because the from and to nodes have to be saved in memory alongside the weight, whereas in an adjacency matrix the from and to nodes are implied by the structure of the data.

As a result, adjacency lists are more space-efficient on **sparse graphs** where there are lots more nodes than edges (this is not a concrete definition as it depends on the exact space usage for an implementation), whereas adjacency lists are more space-efficient on **connected graphs**, where every node is connected to every other node, or graphs which are close to connected.

In addition, adjacency matrices are more time efficient for looking up data. If you want to know whether there is an edge between two points, and what its weight is, in an adjacency list you could have go through every adjacency in the list before finding it

($O(E)$). Whereas with an adjacency list you can calculate the exact memory location of the information with `from_node_id + to_node_id * num_nodes`, which is a constant time operation ($O(1)$).

4.4.2 Selecting a graph implementation

The final TSP solver can't run on the whole OSM network because even the smallest regions have 10's of thousands of nodes. As such, there will be a two-step process, where the inputted locations have the shortest path between every pair calculated and a new "abstracted" graph is formed with as many nodes as there were inputted locations (see fig 4.3).

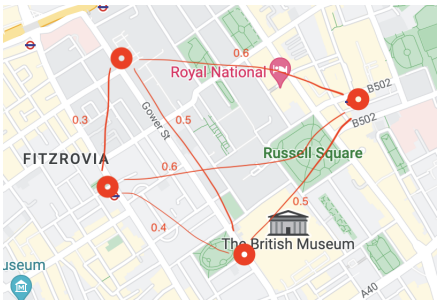


Figure 4.3: Forming abstracted graph for 4 nodes [base map from Google Maps]

The full OSM data for a region will have potentially millions of nodes, and each node will probably have at most 4 or 5 edges connected to it (at a complex intersection) so the graph would be sparse, meaning an adjacency list would be the most space-efficient representation.

The abstracted graph will then be a fully connected graph, so an adjacency matrix would be most space efficient. In addition, the weights in the abstracted graph will be being accessed repeatedly, meaning the faster lookup times will be very useful.

As such, I have implemented both data representation methods in this project.

4.4.3 Implementing graphs in Rust

Having two different graph implementations in the same project opens up the potential for duplicated code. This is not always a bad thing, but it can lead to confusion and unnecessary additional work. For this project I wanted a consistent API for my graph data structures, so that input, output and processing code could run the same, regardless of the underlying storage format.

Rust doesn't have the concept of classes, objects, and inheritance that object-oriented languages such as Java do. Instead, in Rust, you define and implement **traits**. A trait

defines a set of methods that a struct which conforms to that trait should implement. They can also include default implementations of these traits.

For example, the `Debug` trait requires your struct to implement the `fmt` function, which should return a string representing the struct. This trait is used when printing an instance of your struct to the terminal. This is similar to `__str__` in Python, except Rust gives you access to a generic version of this system to allow creation of user-defined traits.

To give my 2 graph implementations a consistent API, I created the `Digraph` trait. This requires defining certain methods such as `num_vertices`, `add_edge_by_index`, and `dist`. The `ALDigraph` and `AMDigraph` structs both implement this trait, and the various methods which operate on graphs (such as `dijkstra`) accept arguments of the type `impl Digraph`, as opposed to specifically `ALDigraph` or `AMDigraph`.

4.4.4 Storing additional node data

The adjacency list/matrix only store the structure of the graph itself. For real-world routing, there is additional data that is useful to store against each node.

Firstly it is important to store the longitude and latitude of each node. This data is used to create the initial graph representation, by finding the distance between two nodes, but it is also needed at runtime so that a user-inputted longitude and latitude can be matched to the closest node on the graph.

Also, the abstracted graph will have different indexes for each node than the original graph. To map the solution on the abstracted graph back onto the original graph, in order to get the road-accurate path, some form of ID needs to be stored with the nodes to link them back together. All OSM nodes are allocated an ID so this should be able to be stored.

To do this, I created another struct called `NodesData`. This contains a hash map linking node ID's to node indexes, node indexes to node ID's, and node indexes to instance of the `NodeData` struct, which currently just holds longitudes and latitudes.

The code for managing node data is the same across adjacency lists and matrices, so the two implementations can have a field holding an instance of `NodesData`, centralising the implementation of this identical functionality.

4.5 Shortest path problem

4.5.1 Dijkstra's algorithm

As discussed in 2.4.6 there are various possible algorithms to find the shortest path between two points on a graph.

As the graphs in this project won't have negative weights, and the specific problem being solved is between specific pairs of nodes (as opposed to all pairs), Dijkstra's algorithm is the best choice.

In addition, Dijkstra is the basis for the heuristics/optimisations discussed: A*, highway hierarchies, and contraction hierarchies.

The core idea of Dijkstra is a breadth-first-search, but choosing to priorities exploring nodes which are closer to the source node.

Pseudocode

```
// Input: g, from_node, to_node

// Initialise data structures
from_node_to_current_node = [inf; g.num_vertices()]
queue = PriorityQueue()

// Distance to from_node is 0
from_node_to_current_node[from_node] = 0
// Start exploring graph at from_node
queue.push(from_node, 0)

while v:= queue.pop()
    for edge in v.adj()
        distance_to_edge_node = from_node_to_current_node[v] + edge.weight
        if distance_to_edge_node < from_node_to_current_node[edge.to_node]
            queue.push(edge.to_node, distance_to_edge_node)

return from_node_to_current_node[to_node]
```

4.5.2 Implementing Dijkstra's algorithm

Instead of implementing a function, I created a struct `Dijkstra` which holds the current state of the algorithm. This makes it much neater to access multiple different outputs from a single run of the algorithms, for example the distance between two points, A and B, and also the list of points between A and B. I used this pattern for all the graph algorithms implemented in this project.

My initial implementation of `queue`, used a `VecDeque` (a double ended queue based on the `Vec` type) as the underlying data structure. To get the next item with the smallest distance from the start (`queue.pop()`), it iterated through every item in the queue. This approach was functional, and allowed me to write a test on a small (8-node) network, but

it was unnecessarily inefficient as it looped through every item in the queue with each iteration.

To resolve this, I needed the queue to store its values in order. I could have used the same `VecDeque` and found the correct location for each item to be inserted, but this would still potentially require running through the whole queue on each iteration. Instead, I used a binary tree, which is a tree that guarantees that every parent node is greater than all of its children. This fact allows insert operations to have a worst-case time complexity of $O(\log n)$, and an average case of $O(1)$, and pop operations to have a worst- and average-case time complexity of $O(\log n)$ [Rus].

To implement it I created a struct called `PriorityQueue`, which was a wrapper around the `BinaryHeap` struct from the Rust stdlib. Thanks to Rust's trait system, I was able to make it generic over the queue's `priority`s, requiring that they satisfy the trait `PartialOrder`. This means that in the future I could use more complex objects to rank where to go next, by just implementing that trait.

This change decreased runtime of my benchmarks by 24–99%. On the OSM Monaco instance (with 9936 nodes), the average runtime went from 133.58 ms, to 981.10 µs.

4.5.3 Benchmarking shortest path algorithms

To have a measure of how much optimisations improved efficiency, and try and prevent regressions in speed wherever possible, I used several benchmark instances.

Firstly, a randomly generated graph, where the probability of nodes being linked is based on a normal distribution function. This is a very unpredictable instance, as it doesn't model anything like the real-world, and was only useful before I could load standard test data into my graph representations.

The DIMACS d1291 is a TSP test instance. Whilst the first stage of TSP for this project will start by running shortest path, in this case the data is not properly geographical, but it is a stable and useful test instance nonetheless.

OSM Monaco is the closest to the real use-case of the shortest path algorithm, as this is real road data taken from the OpenStreetMap with real longitudes and latitudes.

Lastly, DIMACS ran a separate challenge for shortest path algorithms in 2005 [Dim], and I started benchmarking against the USA-road-d.NY instance after the first 2 implementations of Dijkstra.

After using the binary heap and before switching to A*, the ability to trace what the shortest path is (as opposed to just knowing how long it is) was added, which necessarily increased runtime.

See 4.6 for more information on Benchmarking. See A.1 for the full benchmarking data on the iterations of the shortest path algorithm.

4.5.4 A* Heuristic

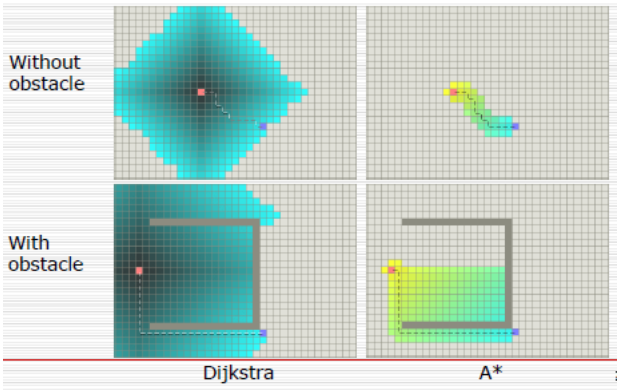


Figure 4.4: Dijkstra vs A* [Stanford]

The A* algorithm is an extension of Dijkstra's. The primary difference is the priority value used for the binary heap.

We define $g(n)$ to be the current best distance from the start node to n, $h(n)$ to be the estimated distance from n to the end, and $f(n) = g(n) + h(n)$ to be the estimated total distance from the start to the end, via the node n.

In Dijkstra's algorithm, the priority used for the binary heap is $g(n)$, meaning that the node closest to the start (which hasn't yet been explored) is explored next. This is guaranteed to give an optimal result, however it does cause nodes which are almost certainly not useful for the final path to be explored. For example, if you're travelling from UCL main campus to UCL East, the quickest route starts by taking the Metropolitan line eastbound to King's Cross St. Pancras, but the closest station is Great Portland Street via the Circle line westbound. See figure 4.5.

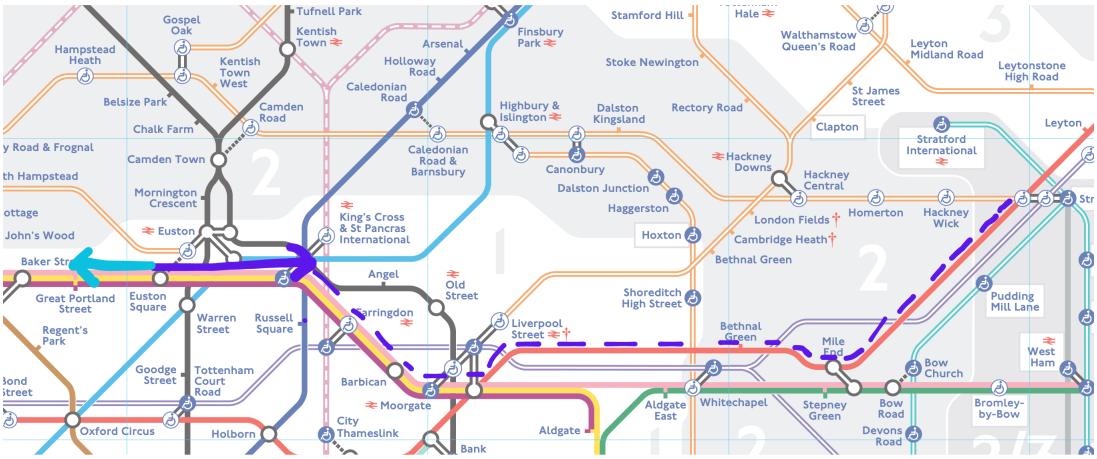


Figure 4.5: Next closest station (light blue) vs Best next station (dark blue) [base map from TFL]

To try and explore nodes which are heading in the right direction, A* search uses $f(n)$ as the priority for the binary heap. This prioritises nodes which are heading roughly towards the target. In the example above, while $g(GPS) < g(KXSP)$, $h(GPS) \gg h(KXSP)$ to the point that $f(GPS) > f(KXSP)$ so King's Cross St Pancras would be explored next.

For each explored node, we know $g(n)$, but we don't know $h(n)$, otherwise we could use $h(start)$ to solve the whole problem, so we need to estimate it. If the estimated value of $h(n)$ is smaller than or equal to the true value, then A* is still guaranteed to give an optimal output [HNR68]. When working with real-world graph data, an easy heuristic therefore is the "as the crow flies" distance between n and the target node, as this is guaranteed to be a lower bound on the route taken between the two nodes.

Switching from Dijkstra to A* sped up USA-road-d.NY by 59%. OSM Monaco, which is from the real dataset that these algorithms will be run on, only improved by 14%. This is mainly because the effects of A* get more powerful, the larger the graph being worked on (9936 vs 264346 nodes). On larger OSM test instances the performance benefits will be much larger.

The random graph benchmark never have correct longitude and latitude values (they were both set to 0). This meant that the algorithm that ended up being run was simply Dijkstra's, because $h(n)$ always evaluated to 0. But, with the added overhead of unnecessary calculations, this cause a large slowdown comparing A* to Dijkstra. The random graph benchmark had been quite unpredictable up to this point, so I removed it before continuing working on the shortest path algorithm.

An additional complication specific to this project is that, while forming the abstracted

graph, we need to find the distances between all combinations of locations we are trying to order. Currently, a single run of Dijkstra can give all the distances between one source node and every other, meaning Dijkstra is run n times for n nodes inputted to the TSP. The A* $h(n)$ heuristic described above targets a specific node for each run of the algorithm, meaning that the distances for any other node are not guaranteed to be optimal, and they may not have even been explored.

To resolve this, while maintaining the guarantee that the estimate is less than the true value, we take $h(n)$ to be the crow-flies distance to the closest target node.

Each node on the graph can be visited multiple times, as parts of paths from different parent nodes. This means that $h(n)$ will be calculated multiple times, causing unnecessary memory accessing and calculations.

I precomputed the shortest distance between each node and the closest `to_node` and stored it in a `HashMap`, to be read when adding to the queue, as opposed to calculating on the fly. This was a case where benchmarking was critical, as something which (I felt) intuitively should have sped up the algorithm instead made it much slower, particularly on large instances. The overhead of reading values from a large `HashMap` caused the largest test instance to slow down by 247%.

Some caching was still possible however, as each calculation of $h(n)$ required getting the locations of all of the `to_nodes`, which there is only likely to be on the order of 100's of at an absolute maximum.

The algorithm only needs to know the location of the closest node, not which node it is, so they could be stored in a `Vec` of `LatLng`'s and iterated through sequentially (instead of accessing specific keys from a `HashMap`). Loading them once from the graph's `NodesData` halved the number of `NodesData` accesses when calculating $h(n)$, and saved 15–29% of runtime.

4.5.5 Routing on real world roads

Importing PBF files

Once I had a stable graph representation, and a functioning (if slow) implementation of Dijkstra, I expected to be able to import a PBF file and get a route from A to B on real roads.

The smallest country in the world is Vatican City, but it doesn't have public roads, and Geofabrik doesn't offer it as a download; as such, I used Monaco as my test area, and I've found this to be very common in the OSM community.

I loaded `monaco-latest.osm.pbf` into the `ALGraph` struct (adjacency list graph implementation) successfully, however running Dijkstra on it produced the following error.

```
thread 'main' panicked at 'Couldn't get node_with_min_distance'
```

The `node_with_min_distance` variable should contain the next node to be "relaxed".

The Monaco graph had 28,092 nodes at time of writing, making it frustrating to debug with, so I created a small test PBF file using JOSM.

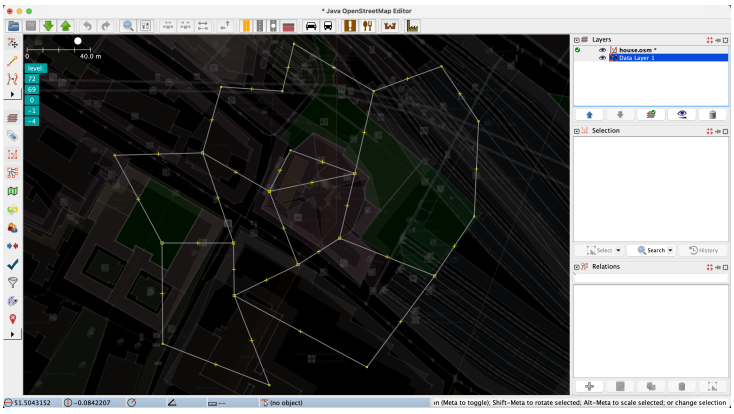


Figure 4.6: Screenshot showing creation of my custom "house" instance in JOSM

I ran Dijkstra and successfully found the shortest path between two points

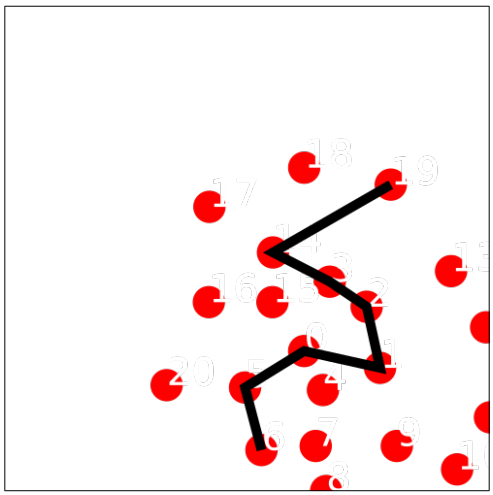


Figure 4.7: Shortest path between two points on the "house" instance

N.B. The path image doesn't exactly match the view in JOSM as my SVG exporter ignored the projection of longitude and latitude at this stage.

Connected components

Dijkstra worked on the "house" graph imported from a PBF file, implying that the issue was not a fundamental issue with the PBF importer. Some debugging on the Monaco

file showed that, just before the error occurred, the distance from the start point to every unexplored node is infinite.

This could mean that the start point must be on an "island", disconnected from the rest of the graph. It could also mean that a single point wasn't linked to the rest of the graph; as Dijkstra explores from the start node to every single node, just 1 disconnected node could cause this error.

To test this I created another test graph, with 2 groups of disconnected nodes.

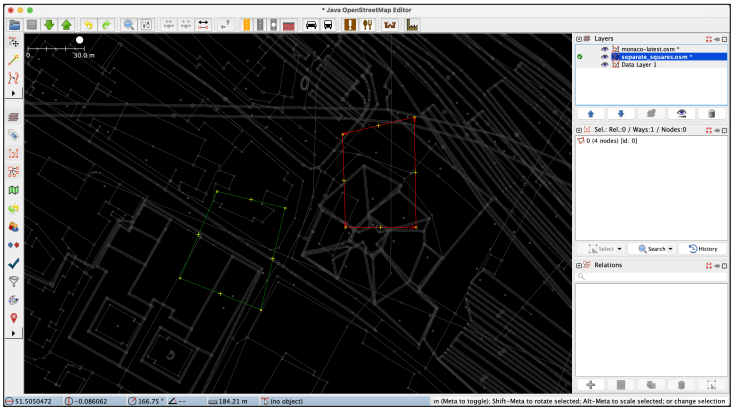


Figure 4.8: Screenshot showing creation of my custom "separate squares" instance in JOSM

This produced the same error as Monaco, providing a small enough test area to debug with.

Strongly connected components

In an undirected graph, a **connected component** is a set of vertices that have paths (finite sequence of edges) between all vertices.

In a directed graph, a **strongly connected component** is a set of vertices where every vertex is "reachable" from every other vertex.

My plan was to find the strongly connected components of the graph, and find the largest to be the "main" one.

There are several algorithms for strongly connected components (all significantly more complex than connected components on an undirected graph); 2 very common ones are Kosaraju's, and Tarjan's. Kosaraju's algorithm requires 2 passes of depth-first search, whereas Tarjan's only requires 1.

This algorithm will need to run on graphs the size of countries, with 100's of thousands of nodes, so I decided to use Tarjan's.

Tarjan's strongly connected components algorithm pseudocode

My original implementation was based on pseudocode from Wikipedia. I implemented it in Rust, and modified it for my specific use case. The below pseudocode is based on my Rust implementation, however the general structure is similar to the original.

```
// Input: g

// Initialise variables
index = 0
node_stack = []
indexes = [-1; g.num_vertices]
low_links = [-1; g.num_vertices]
components = []

// Main loop
for i in 0..g.num_vertices
    // If we haven't grouped this node yet
    if indexes[i] == -1
        strong_connect(i)

// Recursive function
fn strong_connect(v)
    indexes[v] = index
    low_links[v] = index
    index += 1
    node_stack.push(v)

    for w in g.adj(v)
        if indexes[w] == -1
            strong_connect(w)
            low_links[v] = min(low_links[v], low_links[w])
        elif node_stack.contains(w)
            low_links[v] = min(low_links[v], indexes[w])

    if low_links[v] == indexes[v]
        w = -1
        node_indexes_this_component = []
        while w != v
            w = node_stack.pop();
            node_indexes_this_component.push(w)

        components.push(node_indexes_this_component)

return components
```

Success

After running Tarjan's on the OSM Monaco graph, I took the largest connected component (18797/28092 nodes) and successfully ran Dijkstra on real world data.

Figure 4.9: Example route produced by my Dijkstra implementation on the OSM Monaco dataset [base map from OpenStreetMap]

The demo in figure 4.9 uses the web API, discussed in 4.9

The explanation and implication of using only $\frac{2}{3}$ of the nodes in the input data are discussed in 6.3.1.

4.6 Benchmarking

For a performance-sensitive system like this, it was import that each improvement to functionality didn't (unnecessarily) detriment performance.

Rust has a very handy benchmarking tool called Criterion, which made it easy to define benchmarks, and it would compare a benchmark run to the previous one.

I'm using GitHub Pull Requests to manage my development process, and I already have a GitHub Actions workflow (4.2.4) set up, so I added benchmarking.

This wasn't quite as simple as simple to set up as building, testing, and linting, because I wanted to compare the latest commit's benchmark against the master branch. To achieve this on each CI run: it checks out the master branch, runs benchmarks on it, then checks out the current branch again, and runs benchmarks again relative to master.

The data from benchmarking was very useful for developing both the shortest path and travelling salesman problem algorithms, and is discussed in both sections. Below are

a couple of examples not strictly relevant to the development of either algorithm, but notable nonetheless.

4.6.1 Pitfalls avoided by benchmarking

Unnecessary data in commonly created structs

When I added the capability for graph nodes to have data associated with them (e.g. longitudes and latitudes), both the shortest path and travelling salesman benchmarks became 500–800% slower.

To diagnose the issue, I split up the new functionality into several small (fully functional) steps:

1. Add the new functions to the interface
2. Add fields to the graph structs to store this new node data
3. Update benchmarks/tests/main code to start reading/writing node data

The first step shouldn't have had much of an effect, however it was already causing an 50% decrease in performance.

The `adj(node_id: usize)` function which should give a vector of nodes linked to `node_id` actually returned a vector of a struct called `DigraphAdjacency`, in order to include additional data (e.g. the weight of the edge linking the nodes). It felt neat to include the node data for the linked nodes, so I updated the struct:

```
// Old struct
pub struct DigraphAdjacency {
    pub to: usize,
    pub weight: f64,
}
// New struct(s)
pub struct NodeData {
    pub node_index: usize,
    pub longitude: f64,
    pub latitude: f64,
}
pub struct DigraphAdjacency {
    pub node_data: NodeData,
    pub weight: f64,
}
```

I first thought that having to allocate 2 pieces of memory per `DigraphAdjacency` was causing the slow down, so I flattened `NodeData` back into `DigraphAdjacency`:

```rust
pub struct DigraphAdjacency {
    pub node_index: usize,
    pub node_longitude: f64,
    pub node_latitude: f64,
    pub weight: f64,
}
```

However, Rust is able to tell at compile-time how large of a space to allocated for `NodeData` in the same way as a primitive type, so this had minimal effect.

I realised the actual issue was just that more memory was being allocated, because I was including the longitudes and latitudes in this struct that will be initialised on the order of whichever algorithm is being run, e.g. `O(E log V)` for Dijkstra.

This data wasn't actually needed anywhere I was currently using `adj`, and if it's needed in the future I can either look it up each time using the `node_index`, or create a different version of `DigraphAdjacency` and `adj` which includes longitudes and latitudes, so I ended up with

```rust
pub struct DigraphAdjacency {
    pub node_id: usize,
    pub weight: f64,
}
```

Adding the new functions to the interface (step 1) then had no impact on performance, as you would expect.

Unnecessary function calls in commonly called functions

OpenStreetMap nodes all have an ID associated with them, which are represented by 64-bit numbers. My initial graph representations used the "node ID" as the index to the adjacency list/matrix (as they were sequential), which would require allocating vectors $1.8e+19$ items long to accommodate every possible OSM node ID.

To avoid this, I wanted to accept arbitrary 64-bit node IDs, but internally map them to sequential "node indexes", keeping the size of the adjacency data structure as small as possible.

I created a `HashMap` to store this relationship, and updated the `Digraph` trait so that all the functions would accept node ids, and convert them to node indexes on the fly.

This increased runtime by more than 800%!

I quickly realised that the shortest path/travelling salesman algorithms were calling `adj` and `dist` thousands of times, and each call could end up with at least 1 if not 2, or even $O(n)$ memory accesses to convert IDs to indexes.

As a compromise between ease-of-use and functionality, I had the graph creation (`add_node_data` and `add_edge`) and graph data lookup (`get_node_data`) functions still accept node IDs, because these would only be called at the start and end of

a run (whilst population or interpreting a graph). And the high-throughput functions would still use node indexes.

The shortest path/travelling salesman algorithms never needed to create or read data from the graphs so they cleanly interact with node indexes, and the user is left to process the node indexes returned back into IDs.

The addition of node IDs ended up only adding 0–15% of runtime.

4.7 Travelling salesman problem

4.7.1 Benchmarking

Different approaches work better at different scales of TSP solver, and ≈ 1000 nodes is a good upper bound for the size of instance needing solving for DRT applications.

As such, all TSP implementations were tested on the smallest DIMACS instance, d1291. This instance has 1,291 nodes, and an optimal tour length of 50,801 [Rut].

4.7.2 Simulated annealing (naive)

The first approach I tried was a custom simulated annealing algorithm.

With each iteration it would perform a mutation. This mutation had a 50/50 chance of being either a swap between two adjacent nodes, or a transposition of a node to another point in the tour. These are both equivalent to cyclic permutations.

It had a starting temperature (T) of 100, an α value (the multiplier used to reduce the temperature with each iteration) of 0.999, and would run until the temperature was 10^{-9} (T_{min}). This essentially mutated the graph 25,316 times:

$$\begin{aligned} T \times \alpha^i &> T_{min} \\ 100 \times 0.999^i &> 10^{-9} \\ 0.999^i &> 10^{-11} \\ i \log 0.999 &> -11 \log 10 \\ i &> \frac{-11 \log 10}{\log 0.999} \\ i &> 25315.76969 \\ i_{max} &= 25,316 \end{aligned}$$

The number of iterations (i) can be calculated quickly, and using float multiplication over integer addition is a more expensive way to keep track of a loop. However, the value T provides an exponentially decreasing variable which can be used to take "risks" early

on in the optimisation process.

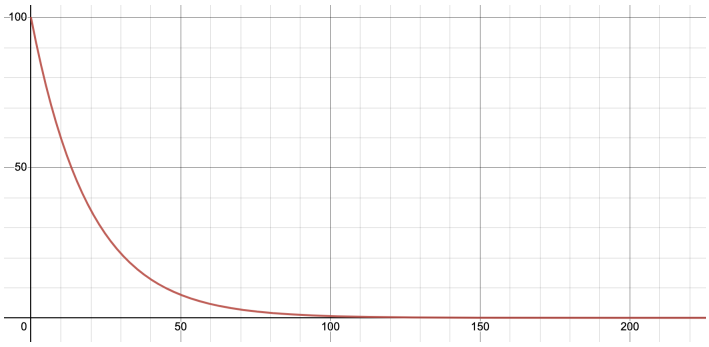
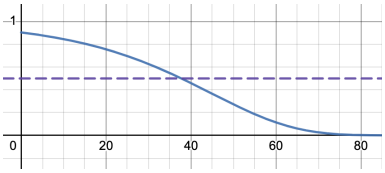


Figure 4.10: $T = 100 \times 0.95^i$

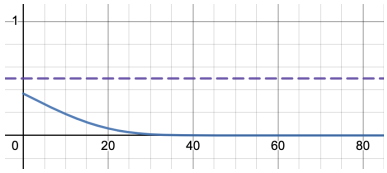
If the result of mutation (the neighbour path) was shorter than the current best path, it would be accepted. If it was longer, the probability of accepting it was calculated with the following formula:

$$P(\text{accepting longer tour}) = e^{-\frac{\Delta\text{path length}}{T}}$$

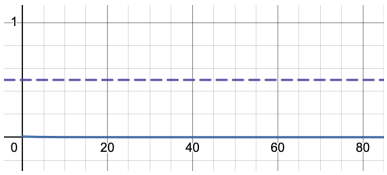
Δpath length is guaranteed to be positive because a negative delta would have been automatically accepted. T is guaranteed to be positive because it starts off positive, and is only altered by being multiplied by another positive value. Therefore $-\frac{\Delta\text{path length}}{T}$ is guaranteed to be negative, and so $e^{-\frac{\Delta\text{path length}}{T}}$ is guaranteed to be between 0 and 1. This makes it a suitable formula to use as a probability function.



(a) Δpath length = 10



(b) Δpath length = 100



(c) Δpath length = 500

Figure 4.11: # iterations vs probability of accepting a worse tour (blue), $P = 0.5$ (purple)

As can be seen in figure 4.10, T quickly drops off to a (visually) indistinguishable value at around 110 iterations. This is even more clear in figure 4.11 where the likelihood of taking a risk is almost 0 at 30 iterations, for a Δ of just 100.

For an instance on the scale of a room, with values in metres (e.g. a robot solving a maze) a *Delta* of 100 is very significant. Whereas on the scale of the UK with values in millimetres (for example) it would be very insignificant. Further development of this approach would require tuning this probability function.

Running this 10 times on the d1291 instance achieved a minimum tour length of 705099.4863 (13.88× optimum) a mean of 724126.1288 (14.25× optimum). It took 2–3 seconds to run each iteration, which is more than double the requirement set out in 3.3.1, excluding all the overhead outside of TSP solving.

See A.2 for full benchmarks for all TSP solvers.

4.7.3 Simulated annealing (2-opt)

As noted above, the mutation function was simply picking between two different types of cyclic permutation. This was a good starting point, however a more common approach is k-opts. As discussed in 2.5.5, a k-opt is a mutation that involves disconnecting k edges, and reconnecting them in various combinations. The simplest form of k-opt is the 2-opt, which only has 1 possible method of recombination.

The practical way do this, with a tour represented as a list of nodes, is reversing a subset of the list, e.g. a, b, c, d, e, f, g, h could be mutated to a, b, c, g, f, e, d, h.

This change improved the minimum tour length of by 34% over the original mutation function.

I expected the simpler function to be faster, however it still consistently took 2–3 seconds to run. My potential theory for this is that both mutation functions required 2 calls to a random number generator, which was likely the time bottleneck.

4.7.4 Simulated annealing (multiple iterations per temp)

As noted in 4.7.2, the period of time in which "worse" mutations are accepted is relatively short (the first ~ 200 out of $\sim 25,000$). To increase this ability I introduced a new variable N, the number of iterations performed at each temperature.

With $N = 2$, the minimum tour length decreased by 27%, and with $N = 5$ by 50%! These did however increase runtime to 4.89s and 9.80s respectively.

4.7.5 Simulated annealing (tuning parameters)

There are several interconnected parameters which needed tuning:

- T_0: the initial temperature value (T)

- N: the number of inner iterations for each external loop
- α: the multiplier used to decrease the value of T with each external loop
- T_{min}: the smallest allowed value of T

The values used in the previous section were $T_0 = 100; N = 1, 2, 5; \alpha = 0.999, T_{min} = 10^{-9}$.

Tuning T_0 and T_{min}

These values are very tightly linked, as the combination of the two determine the number of iterations. Setting $T_0 = 100, T_{min} = 10^{-5}$ will give the same number of iterations as $T_0 = 10, T_{min} = 10^{-6}$, because the ratio between both values is 10^7, and the loop variable runs on multiplication.

To try to optimise them in tandem, I tried all possible combinations of the following values:

- $\alpha = 0.95, 0.99, 0.999$
- $N = 1, 5, 10$
- $T_0 = 1, 10, 100, 1000, 10000$
- $T_{min} = 10^{-1}, 10^{-3}, 10^{-5}, 10^{-7}, 10^{-9}$

This yielded 225 combinations. I loaded them into excel, and ran a Pivot Table, grouping by the same values of α and N, and grouping by the same ratio between T_0 and T_{min}.

Unfortunately, there was no clear visual pattern that I could see. As shown in figure 4.12, there are seemingly random better and worse cells in the middle of almost promising gradients. The full data is in table A.3.1

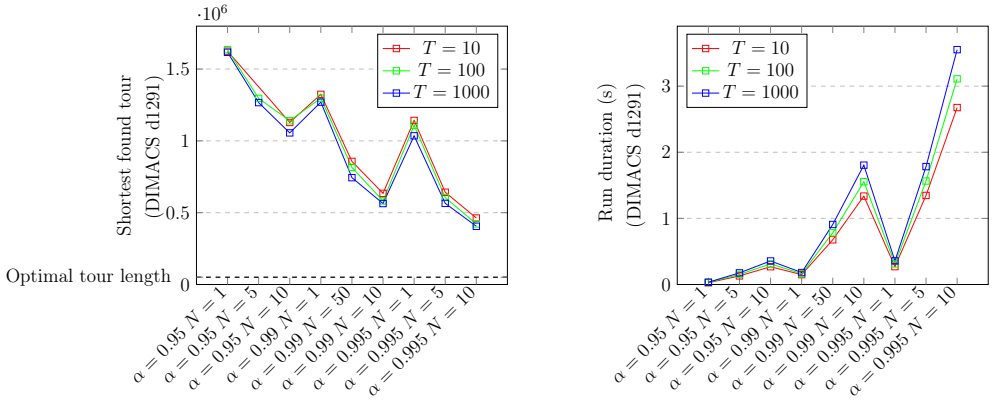
Min. of length	Column Labels																
Row Labels	10	100	1000	10000	100000	100000	1000000	1000000	10000000	100000000	100000000	1000000000	1000000000	10000000000	1E+11	1E+12	1E+13	Grand Total
010.95	1702418.127	1557834.839	1567376.166	1302954.127	1020478.231	947017.1622	1463827.374	1498086.694	1068595.129	868184.1562	839469.4796	468829.4342	1115056.187	1039854.606	476525.4159	467850.7058	478643.5404	467850.7058
010.99	1620120.569	1633905.574	1529780.244	1211215.941	1116372.035	902930.3187	1193612.442	1172354.808	678977.6572	526808.0546	524485.1917	1373812.464	1292249.139	584640.0592	614538.8428	651928.8884	319717.7961	319717.7961
010.995	1664087.032	1656183.626	1466229.588	1146552.802	1373473.89	1497870.679	1331938.599	1360614.944	629255.9068	636637.6731	640294.1254	1152937.206	965607.7854	494925.2811	493988.6363	383526.5488	403256.7315	383526.5488
050.95	1699247.98	1659612.541	1634492.93	1190197.386	1460568.779	1184123.577	1359546.214	1394914.164	724525.5689	683754.9073	665925.6752	536711.5172	1014437.472	916623.2064	558742.0472	389446.188	405151.5398	389446.188
050.99	1582331.27	1574209.407	1391213.528	1158435.585	1074921.682	1064281.344	1530614.81	1156218.496	717629.3581	982731.0785	470722.376	1088100.119	1130752.342	519580.4239	501643.326	493460.6944	573579.8224	470722.376
050.995	1656221.853	1612221.393	1302011.787	1344456.742	1214577.767	1336106.232	1219821.443	1237663.296	744172.6252	556621.6998	559002.1096	967899.7889	901305.3492	692114.9308	380870.2143	331932.2097	365227.9818	331932.2097
100.95	1637037.039	1656473.6	1568125.187	1358874.429	1302519.402	979445.5765	1246785.078	1269464.859	854058.5537	600125.6007	598126.2247	465599.8754	936462.1447	832722.8246	481022.7035	343892.4693	363117.3928	343892.4693
100.99	1698903.094	1573290.822	1247282.504	1225416.276	979509.1719	913252.0699	1430733.271	1415568.917	671854.8639	769755.6716	753817.9881	1005708.058	1035680.719	632338.6547	477943.6117	433260.34	466323.1321	433260.34
100.995	1604113.136	1600805.521	1176856.577	1093857.663	1136996.684	1247617.091	1152690.848	1190101.855	625523.6117	489973.4563	506775.8872	1186967.939	1238768.921	539598.0335	567889.2916	557805.8959	320216.2985	320216.2985
Grand Total	**1582331.27**	**1557834.839**	**1176856.577**	**1093857.663**	**979509.1719**	**902930.3187**	**1152690.848**	**1156218.496**	**625523.6117**	**489973.4563**	**470722.376**	**465599.8754**	**901305.3492**	**494925.2811**	**380870.2143**	**331932.2097**	**319717.7961**	**319717.7961**

Figure 4.12: $T = 100 \times 0.95^i$

Optimising 2 parameters (shortest distance, and duration) with 4 inputs is quite tricky (particularly when the decision making is partly qualitative). Without a clear pattern, I decided to fix T_{min} at 10^{-5}, and tune T_0.

Tuning T_0

I kept the same set of combinations for α and N, and restricted to $T = 10, 100, 1000$, keeping T_{min} fixed. This yielded 27 combinations.



(a) α, N vs Shortest tour

(b) α, N vs Run duration

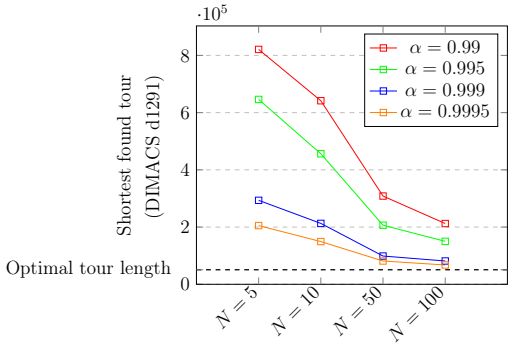
Figure 4.13: Charts of data for tuning T_0

Figure 4.13 shows the effects of different values of T on various value of α and N. In almost all cases, a larger T gives a shorter tour. However this improvement is roughly linear, whereas the effect on runtime is increasing in a non-linear fashion. The full data is in table A.3.2.

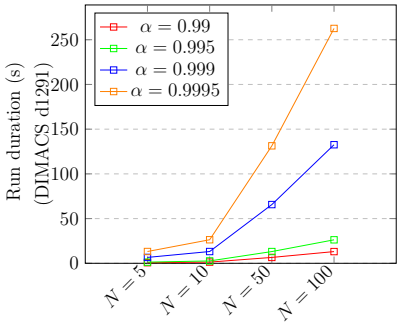
As such, I have chosen to use $T = 10$, in the hopes that a better distance vs run time trade off is possible for α or N.

Tuning α and n

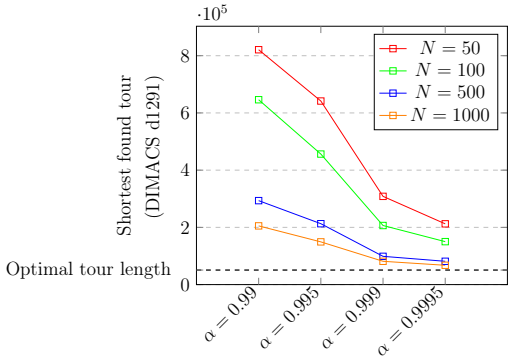
Using $T_0 = 10$, $T_{min} = 10^{-5}$, $\alpha = 0.99, 0.995, 0.999, 0.9995$, and $N = 50, 100, 500, 1000$, giving 16 combinations.



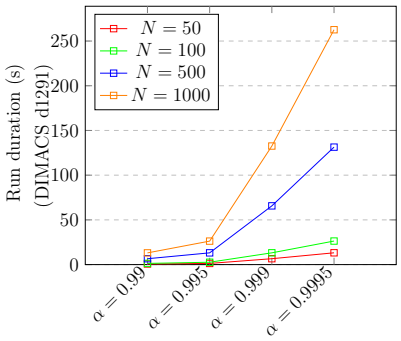
(a) N vs Shortest tour (grouped by α)



(b) N vs Run duration (grouped by α)



(c) α vs Shortest tour (grouped by N)



(d) α vs Run duration (grouped by N)

Figure 4.14: Charts of data for tuning α and N

The charts in figure 4.14 show that the approach is viable, as the best result achieves a tour 30% more than the optimum. In the context of the TSP this is an acceptable result, for example the Christofides Algorithm only guarantees values within 50% of the optimum. However, the runtime for this result was 262.64s, which is far too long for a useable web API. The full data is in table A.3.3.

Selecting values here is quite subjective. To have some rationale for the decision, I have taken the path length with the shortest duration which has a shorter path length than the previous record. This is $\alpha = 0.999, N = 10$.

With the tuned parameters ($T_{min} = 10^{-5}, T_0 = 10, \alpha = 0.999, N = 10$), the minimum tour length achieved was 200,453.14 (n=10).

4.7.6 Simulated annealing vs Hill climbing

The two pairs of graphs in figure 4.14. This was surprising, as higher values of n mean more iterations at higher temperatures than higher values of α. Whether this had a positive or negative impact, it should have had some impact.

This caused me to question whether the simulated annealing was having a meaningful

improvement over simple hill climbing. For the parameters listed above, I calculated the equivalent number of iterations of simple hill climbing using the following formula:

$$i_{max} = N \times \frac{\log\left(\frac{T_{min}}{T_0}\right)}{\log(a)}$$

Figure 4.15 shows that there is very little difference between the two approaches (with equivalent numbers of iterations) and in most cases tested here, hill climbing actually performs better.

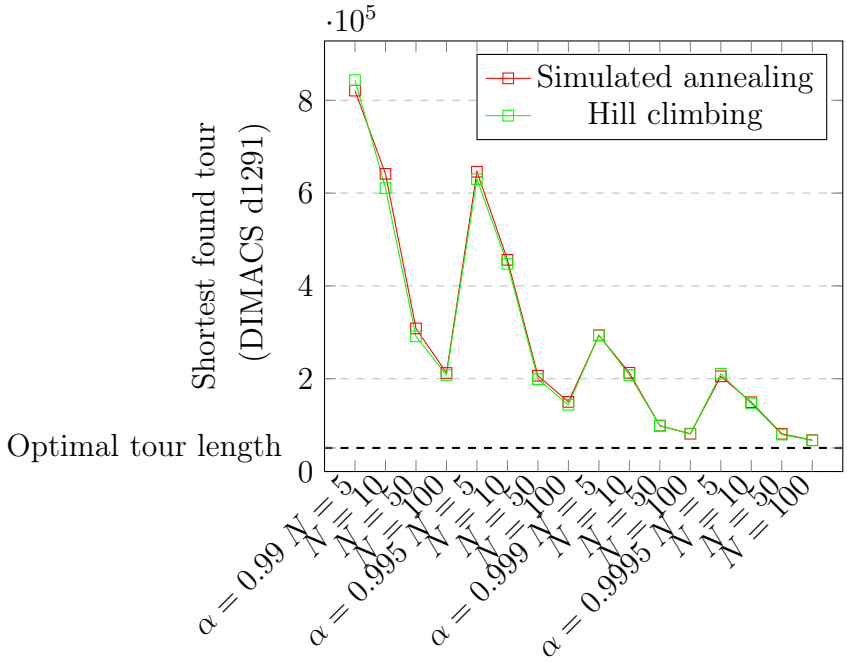


Figure 4.15: Simulated annealing vs Hill climbing (Shortest tour)

I was surprised by figure 4.16, as it seems the run duration overhead from simulated annealing was completely negligible.

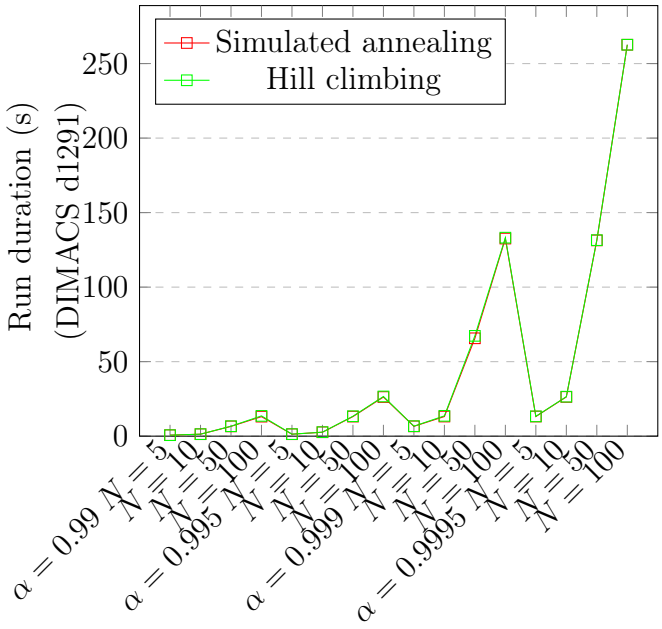


Figure 4.16: Simulated annealing vs Hill climbing (Run duration)

Either way, it seems the simulated annealing was having no effect, and therefore was an unnecessary complication.

4.7.7 Hill climbing

Despite the simulated annealing having no runtime overhead, it was still adding a lot of indirection to tuning the number of iterations of the algorithm. Replacing 4 parameters with 1, the number of iterations i, allows much easier cost-benefit analysis to be performed.

As discussed in 4.7.2, ideally the parameters to the solver would be dependent on input size. With simulated annealing it was relatively unclear how to do this, however with hill climbing it is a lot simpler.

I took 1.5 times the optimal tour length as a rough goal (the best guarantee for the current best non-meta heuristic: Christofides). Using several different sizes of TSPLIB instances from the National TSP dataset, I found the median number of iterations (with 10 attempts) required to reach 1.5 times the optimal tour length. I used the median instead of the mean to remove the effect outliers, as most values should be close together.

The number of iterations required to hit this goal happens to be very close to the number of cities in the instance squared.

Instance	**No. cities**	**Median iterations for 1.5× optimum**	**(No. cities)^2**
Djibouti	38 cities	1,277 iterations	1,444
Qatar	194 cities	33,167 iterations	37,636
Uruguay	734 cities	561,049 iterations	538,756
Oman	1,979 cities	399,8070 iterations	3,916,441

For very small instances (< 10 cities) the number of iterations will be < 100. With the random nature of the approach, a large proportion of those iterations could end up not going anywhere useful. Iterations on instances this small will be very quick, so I will set the minimum number of iterations to be 1000.

On very large instances, the runtime will be completely impractical; to obtain the data for Oman in the table above, it took about 11 minutes per iteration. However, I don't think there should be a cap placed on the number of iterations here, as the result given will likely not be very close to the optimal solution. Instead, users will realise that instances of more than ~ 100 cities will take more than a few seconds.

Using this method of determining the number of iterations to be ran, the minimum tour length reduced dramatically to 77402.36, however it took 160.19 s on average to run.

4.7.8 Hill climbing (path length delta)

With a distance target ($1.5\times$ optimal) as well as a time target (< 1 s), there was clearly further optimisation required. To determine where to focus effort, I used a profiling tool called `pprof-rs`, which measures which areas of the code are being used the most.

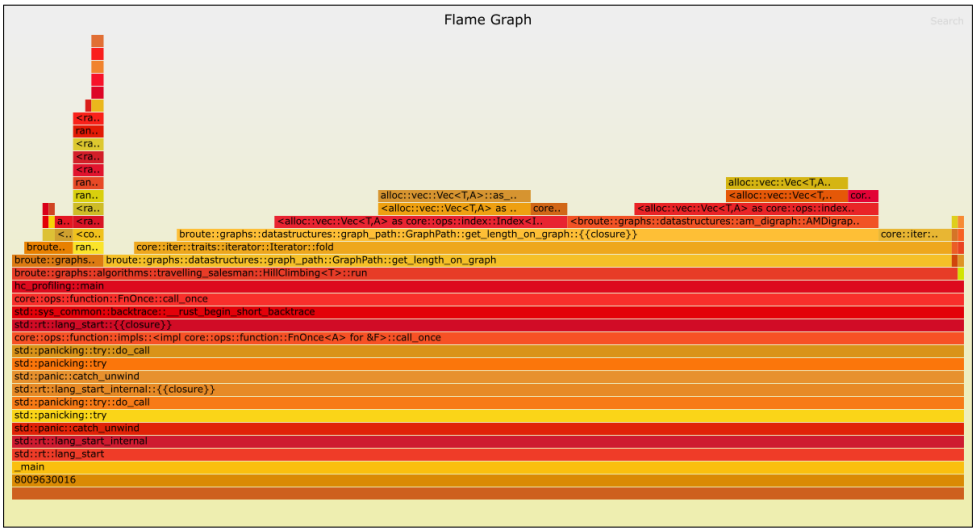


Figure 4.17: Initial profiling flamegraph of hill climbing algorithm

My assumption was that the random number generation would be the slowest factor, however the "flamegraph" generated (fig 4.17), shows that it is the `get_length_on_graph` function. This is a 3-line function that iterates over a `GraphPath` and finds how long it is by summing all the weights for the edges in the path. This requires a number of array access equal to the size of the graph every single iteration.

Unlike the original mutation method, the 2-opt is a consistent operation, which happens to have a consistent effect on the length of the path. Removing two edges and adding

two others, means that the new length of the path is equal to the old length minus the lengths of the removed edges, plus the lengths of the new edges.

Keeping track of the current path length in this incremental method produced paths with the (roughly) the same lengths, but taking an average of 7.32 s: a 95% runtime improvement!

The ability to calculate the path length delta separately from performing the mutation, means that the delta can be found first. If the delta would be positive, the mutation doesn't need to be run, potentially saving hundreds of array read and writes per iteration.

This improvement decreased runtime a further 40%, with roughly consistent tour lengths.

4.7.9 Hill climbing (better random numbers)

Running profiling again revealed that the bottleneck was now in the random number generation function, as expected.

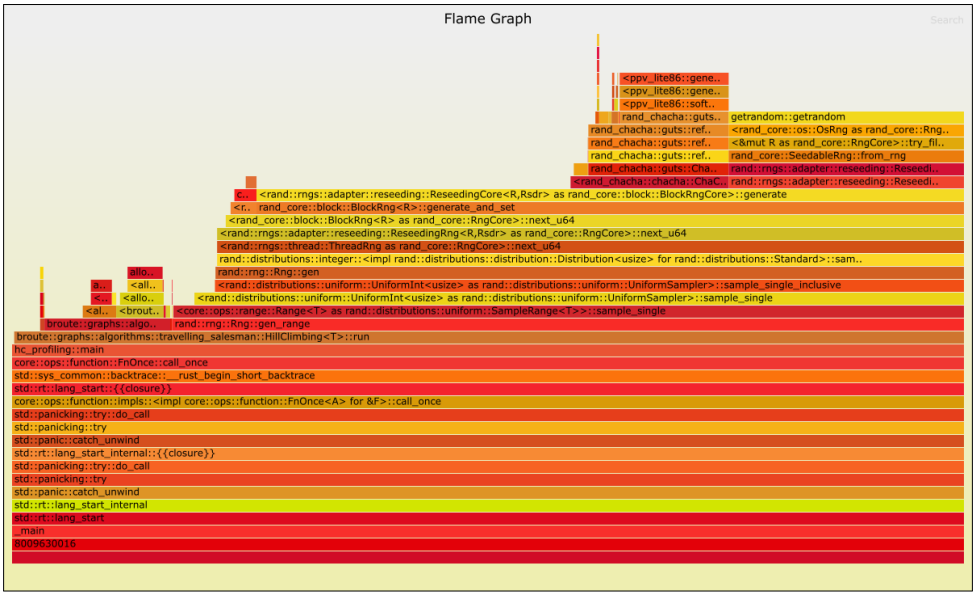


Figure 4.18: Profiling flamegraph of hill climbing algorithm, after incremental path lengths and conditional mutations

Random generation is not built into the Rust standard library, however the `rand` package is so popular (almost 190M downloads) that it is the de-facto standard. The documentation uses the `thread_rng` function in most examples, so I used that with little examination.

A large consideration for random number generation is cryptographic security, meaning that the results of the random number generation shouldn't be possible to be predicted. This is vital for applications where the random numbers are being used as encryption

keys. It makes sense for a library to push a cryptographically secure random number generator (RNG) as the default, to try and prevent people making mistakes, however it is not necessary for this application.

Comparing the available RNGs I chose `Pcg64Mcg`, because it has a high throughput (7 GB/s), and still relatively good statistical quality [Pcg].

Using the new RNG produced tours with comparable distances, with a mean duration of 848.1 ms, the first value under the 1s requirement set out in 3.3.1!

4.7.10 Hill climbing (micro-optimisations)

Looking at the new flamegraph (fig 4.19), the `two_opt_cost` function is now taking the majority of the runtime.

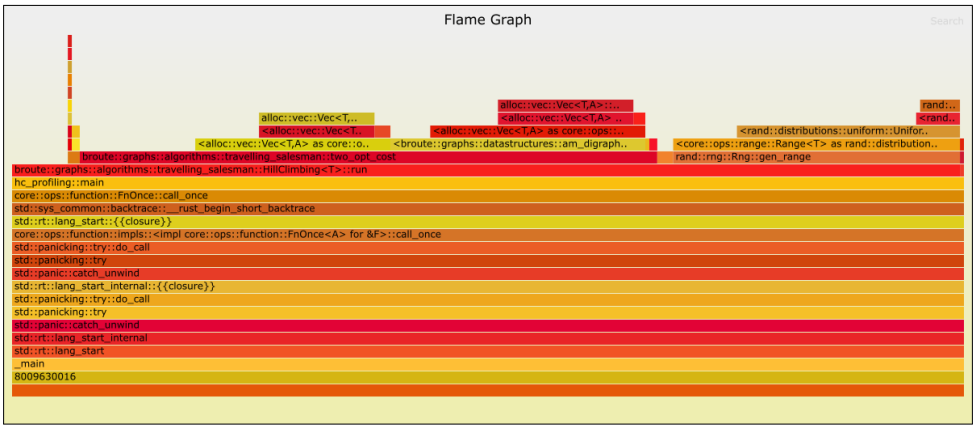


Figure 4.19: Profiling flamegraph of hill climbing algorithm, after new random number generator

It currently looks like this (Rust-based pseudocode)

```
fn two_opt_cost(g: AMDigraph, path: GraphPath, i: i32, j: i32) -> f64 {
    first = min(i, j);
    second = max(i, j);
    length_delta = - g.dist(
      path[first],
      path[(first + 1) % path.length]
    ) + g.dist(
      path[first],
      path[second]
  );
    if second < path.length - 1 {
        length_delta -= g.dist(
            path[second],
            path[(second + 1) % p.path.length]
        )
        length_delta += g.dist(
```

```
            path[(first + 1) % path.length],
            path[(second + 1) % path.length],
        )
    }
    return length_delta;
}
```

Each of the 4 nodes is accessed from `p.path` twice, causing unnecessary memory accesses.

```
pub fn two_opt_cost(g: &AMDigraph, p: &GraphPath, i: usize, j: usize) -> f64 {
  first = min(i, j);
    second = max(i, j);
    a = path[first];
    b = path[(first + 1) % path.length];
    c = path[second];
    d = path[(second + 1) % path.length];
    length_delta = - g.dist(a, b) + g.dist(a, c);
    if second < p.path.length - 1 {
        length_delta += -g.dist(c, d) + g.dist(b, d)
    }
    return length_delta;
}
```

Accessing them once (as above), reduces mean runtime to 731.32 ms.

In addition, the function currently accepts values of `i` and `j` in any order, and then finds which is first and second. This is also separately carried out by the `two_opt` function. If both functions are defined to require `i` $<$ `j`, this reduces mean runtime to 714.6 ms.

```
pub fn two_opt_cost(g: &AMDigraph, p: &GraphPath, i: usize, j: usize) -> f64 {
    a = path[first];
    b = path[(first + 1) % path.length];
    c = path[second];
    d = path[(second + 1) % path.length];
    length_delta = - g.dist(a, b) + g.dist(a, c);
    if second < p.path.length - 1 {
        length_delta += -g.dist(c, d) + g.dist(b, d)
    }
    return length_delta;
}
```

The flamegraph now shows that the majority of time in `two_opt_cost` is spent in `Index` which is the built-in code for accessing values from vectors. Typically when most time is being spent in core or standard library code, the function is close to fully

optimised.

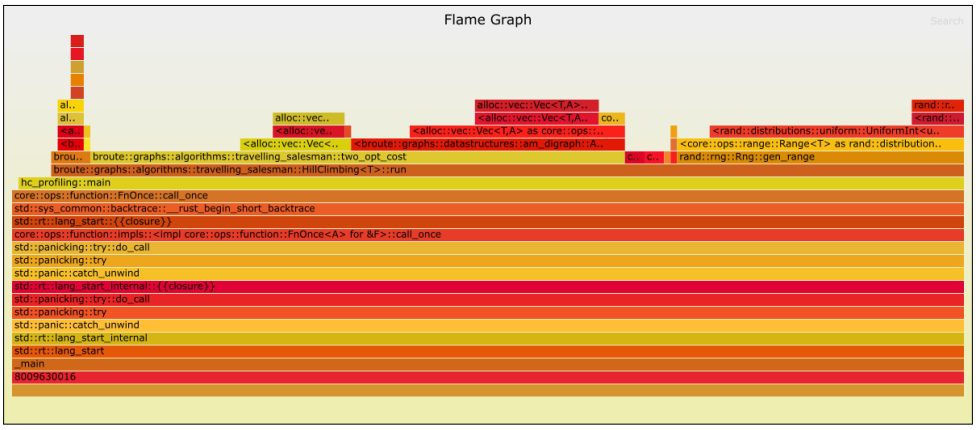


Figure 4.20: Profiling flamegraph of hill climbing algorithm, after micro-optimisations

4.7.11 Hill climbing (Rust optimisations)

All of the timing values given in this section (4.7) have been produced running Rust in debug mode. This collects information like line numbers for errors, and also does less optimisation, reducing build times.

When running the TSP solver on the d1291 instance with the `release` profile enabled, the average runtime is just 37.62 ms.

4.8 DRT routing

4.8.1 Approach structure

Whilst the TSP is the crux of the DRT routing problem (DRP), the DRP is a larger problem due to having to work with real-world data.

The OSM data for Greater London contains over 8 million nodes, and, ignoring the fact that industry leading TSP solvers would take hours to find a close to optimal path, it's not actually necessary. We don't care what the optimal path through every street corner in London is, we care what the optimal order to serve certain points is.

As discussed in 4.4.2, the tool will instead create an "abstracted" graph, containing only the nodes being served. The abstracted graph will be created by running A* between all pairs of inputted nodes, and creating a fully connected graph which the TSP can then run on.

This means that the number of "cities" the TSP will need to run on is simply the number of pick up and drop off locations, which is unlikely to ever go above 1000 nodes in a single DRT area in a single day.

The random nature of the TSP solver means that a single run is not guaranteed to give an optimal result. As the runtime is just 37.62 ms, it's possible to run the solver 5 times and select the best result, and still expect to achieve $<$ 1s response times on the API@.

Once a (close to) optimal tour has been found on the abstracted graph, the route on the full graph can then be worked out, again using A*, to get a list of longitudes and latitudes which can be plotted into an accurate line on a map.

The full process is laid out in figure 4.21

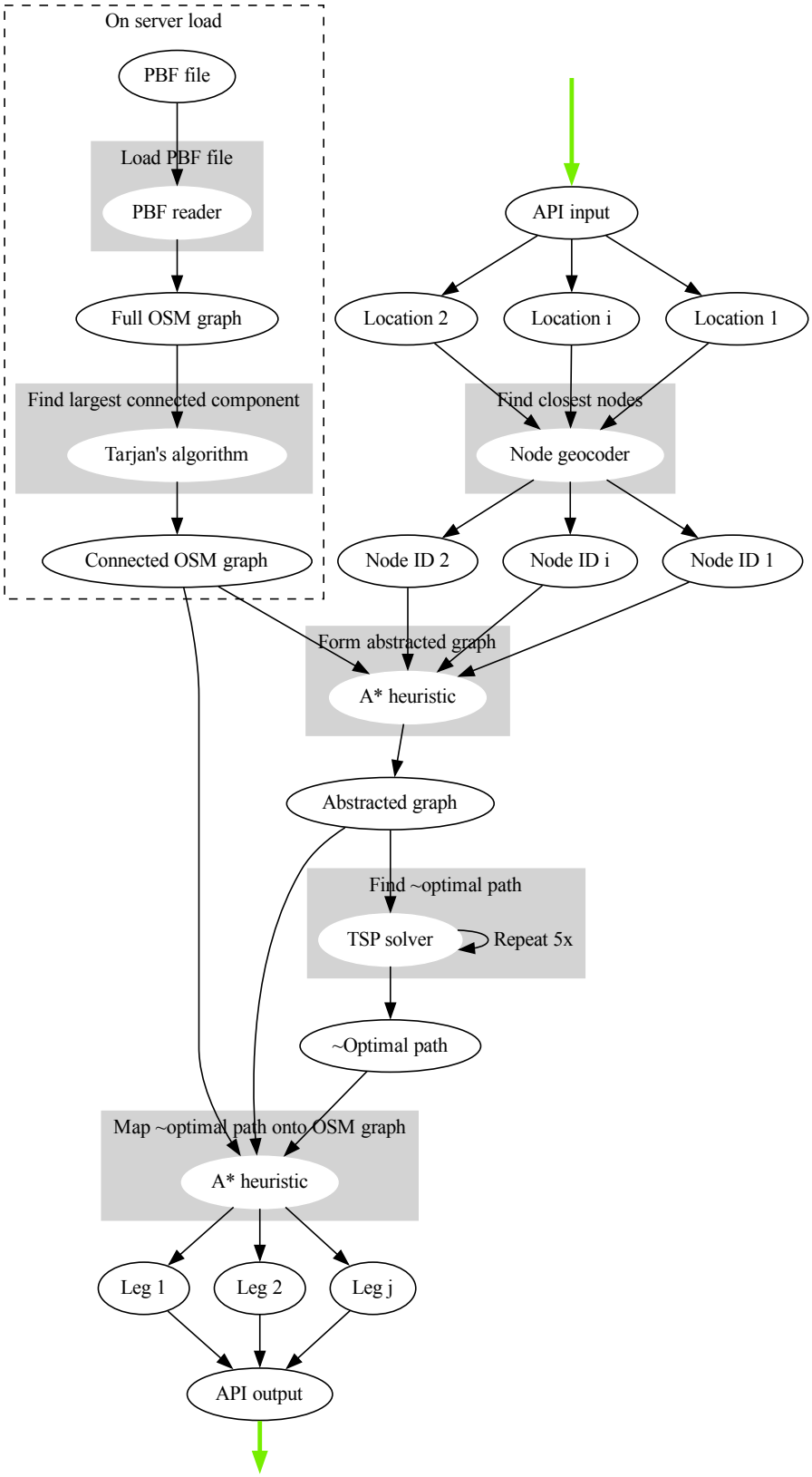


Figure 4.21: Full TSP API pipeline

For the API input and output, see 4.9

4.8.2 Node geocoding

Users (both end users accessing a product, and developers using the API) (almost certainly) don't have any information about OSM or where its nodes are. Instead, they want to work in WGS84 coordinates (AKA longitudes and latitudes).

To deal with this, the tool takes the list of coordinates, iterates through all nodes in memory, and finds the closest node by crow's distance. This produces a list of OSM Node ID's.

4.8.3 Forming abstracted graph

To form the abstracted graph, we need the distance between each pair of points (see 4.4.2). Using the multi-goal functionality in the A* implementation, it's possible to get this data with 1 run of A* per node.

For speed of lookup the A* algorithm works on node indexes, not Node ID's. This is tricky when forming the abstracted graph, because the node indexes will not be consistent between the original full-size graph, and the new abstracted graph.

Pseudocode

```
// Input: g, node_ids

abstracted_graph = AMGraph()

node_indexes = node_ids.map(
    node_id -> g.get_node_index_by_id(node_id)
)

for from_node_id in node_ids
    from_node_index = g.get_node_index_by_id(from_node_id)
    astar = AStar(g, from_node_index, node_indexes)
    for to_node_id in node_ids
        if to_node_id != from_node_id
            to_node_index = g.get_node_index_by_id(to_node_id)
            abstracted_graph.add_edge_by_id(
                from_node_id,
                to_node_id,
                astar.dist(to_node_index)
            )

return abstracted_graph
```

4.8.4 Finding the optimal path

The abstracted graph is setup just how TSP solver expects, as a fully connected (relatively) small graph, similar to the TSPLIB test instances.

The solver is run 5 times, in case the randomness of the solver gives an unexpectedly poor result. The shortest of the 5 paths found is returned.

4.8.5 Mapping optimal path onto OSM

The result returned from the TSP solver is a list of node indexes for the abstracted graph. These first need to be turned into Node ID's, to be mapped onto the main OSM graph. Then, the distance from each Node ID to the next is calculated using A*

Pseudocode

```
// Input: g, abstracted_graph, path

p_node_ids = path.map(
    node_index -> g.get_node_id_by_index(node_index)
)

legs = []

for i in 0..(p_node_ids.length - 1)
    from_node_index = g.get_node_index_by_id(p_node_ids.length[i])
    to_node_index = g.get_node_index_by_id(p_node_ids.length[i + 1])
    astar = AStar(g, from_node_index, [to_node_index])

    leg_node_indexes = astar.get_graph_path(to_node_index)

    leg = leg_node_indexes.map(
        node_index -> g.get_node_data_by_index(node_index).latlng
    )
    legs.push(leg)

return legs
```

4.9 Web interface

In order for the tool to be accessible to users, without them worrying about interfacing with rust, I created a JSON-based REST API@.

Rust is not currently a common language to write website backends in, however there is a library (still under development) called Rocket which makes this task a lot easier.

There are 2 API endpoints: 1 to directly access the A*-based shortest path problem solver, and 1 to access the travelling salesman solver.

Creating an API endpoint in Rocket is as simple as defining a function. There is a macro which allows you to define the URL for the endpoint, along with any parameters; the parameters in the URL should then be listed as parameters to the function (with expected types), and when the URL is accessed the function will be called, with the desired parameters passed in. And, if you define a struct with the desired return values, Rocket will convert the data to JSON for you. This is all possible thanks to Rust's incredible macro system, which allows complete introspection of the types of the function's input parameters, and the output struct's fields.

In order to efficiently serve multiple user's requests at once, Rocket highly encourages using async functions. This means that multiple requests could be being served in parallel. Both API endpoints need access to the OSM graph, which takes some time to import, however multiple threads accessing the same piece of memory is treacherous ground.

Luckily, Rust has excellent built-in primitives for dealing with this. Firstly `Arc`, short for atomic reference counter, which can hold any value while keeping track of how many places are still using it (or could still be using it) even across threads. This allows multiple threads to access the same object at once, without worrying about Rust losing track of whether memory can be freed or not.

In most languages, this would be sufficient, however one of Rust's (very valuable) safety features is it's move checker. This means that as soon as 1 thread reads the value in the `Arc`, it will have been "moved" and therefore not available to any other threads. To get around this, we can copy the data to each thread, but this seems an unnecessary memory (and time) overhead.

Luckily, Rust provides another useful primitive: `RwLock`. This allows any number of threads to read the data inside it, while only allowing 1 to edit it at a given time. Placing the OSM graph inside an `RwLock` inside the `Arc`, gives a thread-safe, borrow-safe, efficient way of providing access to the data to as many user requests as is needed (subject to system resources).

4.10 Automated test coverage

The rust compiler has the ability to collect code coverage data at a source-code level, using LLVM tooling.

Mozilla maintains a tool called grcov which can take this code coverage data and process it into a human-readable HTML format.

LINES

93.93 %

BRANCHES

100 %

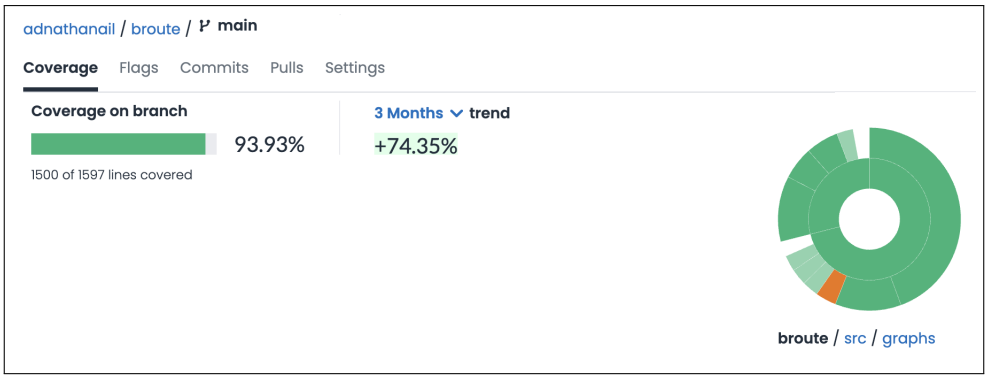
Directory	Line Coverage		Branches	
src	98%	245 / 250	100%	0 / 0
src/algorithms	63.64%	14 / 22	100%	0 / 0
src/geography/algorithms	100%	15 / 15	100%	0 / 0
src/geography/datastructures	60%	6 / 10	100%	0 / 0
src/graphs/algorithms	98.61%	283 / 287	100%	0 / 0
src/graphs/datastructures	87.58%	134 / 153	100%	0 / 0
src/graphs/input	85.06%	131 / 154	100%	0 / 0
src/graphs/output	89.21%	281 / 315	100%	0 / 0
tests	100%	1 / 1	100%	0 / 0
tests/algorithms	100%	18 / 18	100%	0 / 0
tests/geography	100%	16 / 16	100%	0 / 0
tests/graphs	100%	356 / 356	100%	0 / 0

Figure 4.22: grcov HTML report

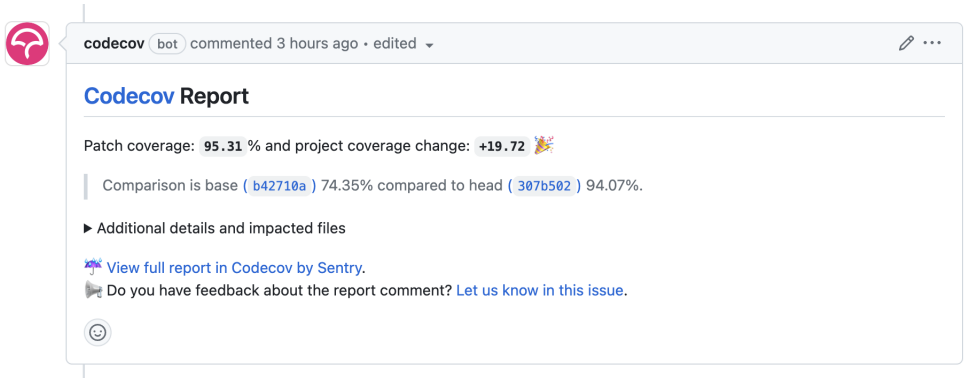
Figure 4.22 shows a lot of red, however the line coverage is still 93.93%. Some lines are not covered by tests because they are not currently used, but they still make sense to be there. For example `LatLng.latitude_radians()` is used in the `haversine` function but `LatLng.longitude_radians()` is not used anywhere. Nevertheless, for parity it seems sensible to keep it.

It can also process the data into `lcov` format. This format was originally created for the Linux kernel, but is now a standard code coverage format which lots of tools produce and interpret.

Codecov is one such tool: a SaaS which provides a nice interface for viewing and managing code coverage. It provides a visual representation of which parts of a GitHub repository are covered (Fig 4.23a), and comments on Pull Requests with the effect that merging the PR will have on coverage (Fig 4.23b).



(a) Coverage graph



(b) Coverage PR comment

Figure 4.23: Coverall features

The final code coverage for the project was 93.93%.

4.11 Documentation

Another feature wrapped into Rust's `cargo` CLI is `rustdoc`. This searches for triple-slash comments: `///` (normal rust comments are double-slashes: `//`), and associates them with a module/function/struct/etc. It then processes them into HTML to be viewed as a documentation, with stdlib and dependency code all linked in.

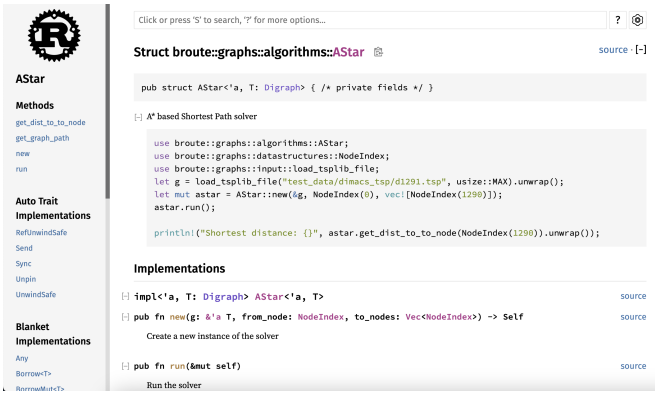


Figure 4.24: Example documentation page

You can include code samples in the documentation comments (see Fig 4.24). These code samples are then included in automated tests, potentially adding additional code coverage, and guaranteeing that documentation examples are correct.

Rust has a number of built-in lints in the form of macros, such as `#[warn(dead_code)]` which checks for any unused code. This is enabled by default.

`rustdoc` provides a lint `#![warn(missing_docs)]`, which checks every place it is possible to have documentation and generates a warning if there is no documentation.

Thanks to this lint, this project is fully documented.

GitHub provides a free static file hosting feature called Pages. I created a GitHub Action to generate the documentation and copy it to the hosting each time a PR is merged.

The documentation is available here.

4.12 Libraries

The main functionality of the tool, described above, is all my own work. However there are a number of common tasks which I used libraries for, as they were not the focus of this project.

For version numbers, see `cargo.toml` in Appendix D

Testing

- criterion (benchmarking)
- float-cmp (comparing floating-point values)
- pprof (profiling)

- reqwest (HTTP requests for statistics)

Random number generation

- rand (generating random numbers)
- rand_pcg (additional random number generators for `rand`)
- rand_distr (generating random numbers according to specific statistical distributions)

File parsing/writing

- osmpbf (reading OSM files)
- tsplib (reading TSPLIB files)
- graphviz-rust (producing GraphViz graphs)
- plotlib (producing data plots)
- svg (writing SVG files)

Web API

- rocket (producing an API interface from Rust functions)
- serde ((de)serialising Rust structs to JSON)

4.13 Name

I have named the tool **Broute**. This is a portmanteau of "bus route". It is also a play on "brute force", the naive approach to optimisation problems.

Chapter 5

Testing

5.1 Chapter outline

This chapter details the empirical testing I performed on the various components of the tool, with graphs of data gathered, and conclusions drawn.

5.2 Hardware

All tests were run on a 2021 MacBook Pro with an M1 Max chip and 32 GB of RAM.

5.3 Shortest path problem solver

As mentioned in 4.5.3, the Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) ran a challenge for the global community to submit their best shortest path solvers.

To facilitate this, they provided 12 test instances. Unlike the travelling salesman problem, there are lots of questions that can be asked of the same graph. In order to fairly compare solvers, they produced a system of randomisation to try lots of different routes on each instance, to produce comparable results.

This was hard to get working, as the challenge was from 2006, and macOS is not knowing for its backwards compatibility. As this project is more focussing on the TSP, I decided to do use their data but create my own simplified testing methodology.

I ran my A*-based algorithm on each instance 100 times, with a random pair of nodes each time. The sizes of the instances ranged from 250,000 to 24,000,000 nodes.

Figure 5.1 shows that the run duration scales roughly exponentially with the size of the test graph. The full data is in table A.4.

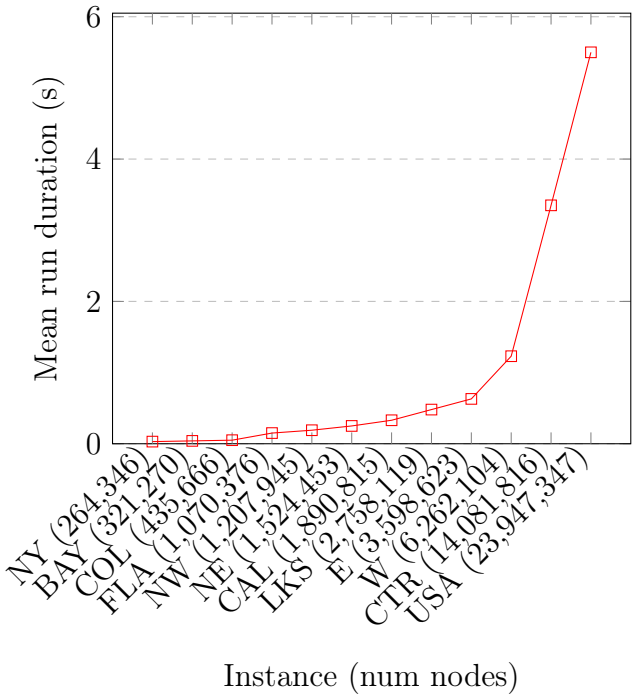


Figure 5.1: Shortest Path: Instance vs Run duration (DIMACS 9 instances)

The final results of the DIMACS challenge were only centrally published for the `USA` instance, with 6 different preprocessing techniques. The slowest average query time was just 2.78ms [DGJ16], however this required 88 minutes of preprocessing time. The quickest query time was 0.019ms, with 104 minutes of preprocessing. In comparison, my average runtime for `USA` was 5.5s.

My implementation, for an OSM graph like Greater London with 8 million nodes, would take over a second per A* run. This will create a large performance bottleneck for the TSP pipeline, when creating the abstracted graph.

The results of all the preprocessing algorithms were then run on A* based solvers. This means that the current solver is still very useful as a core, and can be built on to achieve much better results.

5.4 Travelling salesman problem solver

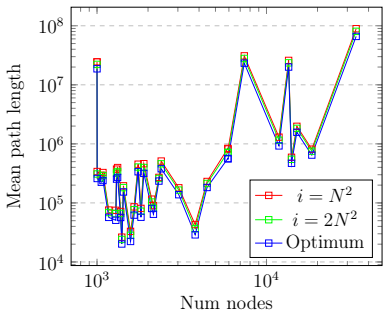
5.4.1 DIMACS data

DIMACS also ran a challenge for the TSP; the `d1291` instance was used as a reference during development. The full DIMACS 8 TSP instance set contains 33 instances with 1000 to 85,900 nodes.

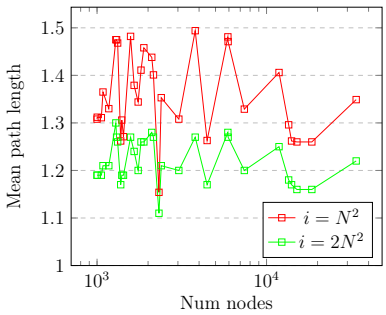
I ran the hill climbing algorithm on each instance 100 times, recording the path lengths

found, and the run duration. I ran this once with the standard number of iterations, determined in 4.7.7, and once with double this number.

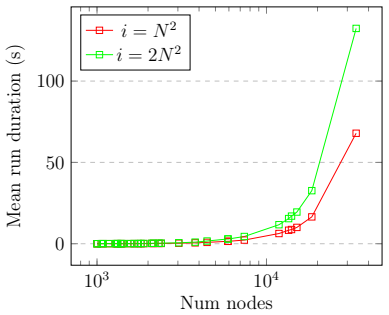
The graphs below show the results of this testing. The full data is available in A.5.1 and A.5.2.



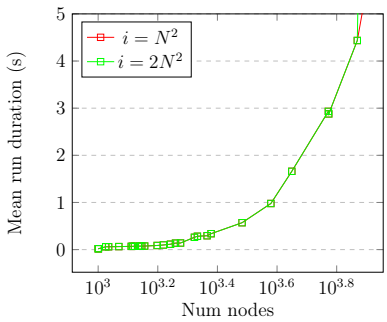
(a) Num nodes vs Path length



(b) Size of graph vs Path length relative to optimum



(c) Size of graph vs Run duration



(d) Size of graph vs Run duration - Zoomed

Figure 5.2: TSP: Testing graphs (DIMACS 8 instances)

I was unable to test the largest instance, `pla85900`, due to the memory available on my laptop. The next smallest instance, `pla33810`, with 33,810 nodes, was using 8.79GB, whereas `pla85900` took 55GB. This makes sense, because the TSPLIB importer adds edges between every pair of edges. This means that the number of edges scales with the square of the number of nodes.

My laptop has 32GB of RAM, meaning `pla85900` needed to use the permanent storage as memory (swap), which is much slower than RAM. Whilst `pla33810` took an average of 67.9s per iteration, I calculated that a single iteration of `pla85900` would take 78 days.

Figure 5.2a shows the two sets of test data against the optimal result on a log graph. The results are a very consistent distance apart, but the range of values of the graph makes the data tricky to interpret.

Figure 5.2b shows the two sets of data relative to the optimal path. Both numbers

of iterations are consistently below the target of 1.5× optimum. However, doubling the number of iterations does consistently improve the output.

Figure 5.2d shows the effect of doubling the number of iterations has relatively little effect on runtime with less than 10,000 nodes, however figure 5.2c shows an already significant increase (almost double the runtime) at 33,000 nodes. This is likely due to more constant or linear factors taking up a lot of runtime for smaller instances.

Finding up-to-date benchmarks for these results was surprisingly frustrating given the purpose of the challenge. These results from 2003 are the best I can find, however they list 4 instances as open problems, which they are no longer. Also, 8 of the values in the benchmarks table have complications in their data, such as being run on parallel diverse hardware.

Taking the top "clean" benchmark value, they found the optimum solution for `fl3795` in 69886.48 seconds (on 2003 hardware). My solver found a solution with a length of 1.27× the optimum, in 0.9780 seconds.

Every clean value in the table which I also tested my solver on, my solver was faster, often by several orders of magnitude. In all cases, my result was within 1.5× the optimum.

5.4.2 OSM data

Graphs are very diverse beasts. While having standardised testing is very useful for comparing solvers, it's also very insightful to test the solver on the specific data is will be working with.

Running full end-to-end API requests, on the OSM Monaco instance, 100 times, with 5, 10, 25, 50, and 100 nodes per request, produced the following graph (full data in A.5.3):

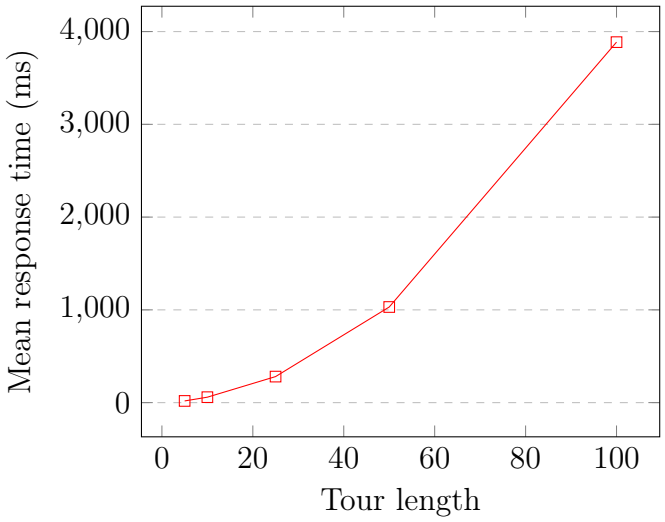


Figure 5.3: TSP: Tour length vs Response time (OSM Monaco)

This shows that the targeted 1 ms response time is breached at around 50 nodes.

Removing the indirection of the API allows breaking down the response times into the different stages of requests, shown in the graph below (full data in A.5.4).

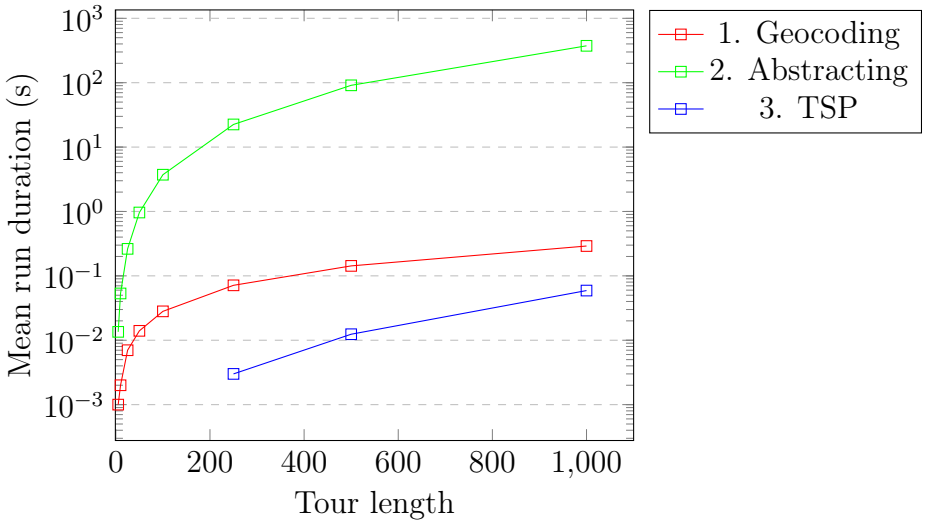


Figure 5.4: TSP: Tour length vs Run duration broken down (OSM Monaco)

Missing values from the graph indicate a runtime too small to measure

5.5 Conclusions

5.4.1 shows that the speed of the TSP solver is very competitive. Achieving tours within 1.5× optimum, in under 1 second, for tours with over 5000 nodes.

Figure 5.3 seems to indicate that the solver performs drastically worse on OSM data, taking almost 4 seconds for a tour with 100 nodes. However, figure 5.4 shows that the TSP takes close to no time at all, even up to 100 nodes. In fact, the forming of the abstracted graph is the primary bottleneck.

The primary operation when forming the abstracted graph is the A*-based shortest path solver. Figure 5.1 shows that it can take over a second for a single run of A* on a graph the size of London. A* needs to be run once per node in the tour, so a tour of 1000 nodes would take over 15 minutes to form the abstracted graph.

Chapter 6

Conclusions and evaluation

6.1 Chapter outline

This chapter compares the finished tool with the requirements laid out in 4, and outlines other potential improvements that could be made to the tool.

It then compares the tool to the current market, lays out what has been produced by the project, and gives a plan for future development.

6.2 Requirements

6.2.1 Functional requirements

FB1 - Must - Achieved

The tool must have the ability to find the shortest path between two points in a graph

The A*-based shortest path solver can find the shortest path between one source node and any number of destination nodes.

FB2 - Must - Achieved

The tool must have the ability to find the ordering of nodes in the graph close to the shortest length (travelling salesman problem)

The hill climbing-based TSP solver, with random 2-opt mutations, is typically able to find the shortest ordering of nodes to within 1.5% of the optimum (see 5.2b).

FB3 - Could - Not achieved

The tool could have the ability to find n separate orderings of the nodes in a graph with close to the shortest total length (vehicle routing problem)

This was not achieved, however the metaheuristic based approach means that this functionality can be added as an extension to the TSP solver, as opposed to a whole new implementation.

Given 4 vehicles, I propose adding 2 values to the path which represent the break between different vehicle's tasks.

VRP state:

$$[12, 1, 8, V, 10, 3, 5, 6, V, 9, 4, 11, 2, V, 7, 13, 14, 0]$$

Resulting vehicle paths:

$$V_1 = [12, 1, 8]$$
$$V_2 = [10, 3, 5, 6]$$
$$V_3 = [9, 4, 11, 2]$$
$$V_4 = [7, 13, 14, 0]$$

VRP state after 2-opt (2,5)

$$[12, 1, 8, 3, 10, V, 5, 6, V, 9, 4, 11, 2, V, 7, 13, 14, 0]$$

Resulting vehicle paths

$$V_1 = [12, 1, 8, 3, 10]$$
$$V_2 = [5, 6]$$
$$V_3 = [9, 4, 11, 2]$$
$$V_4 = [7, 13, 14, 0]$$

Assuming the vehicles start and end at their first and last locations (as the TSP solver currently does) the cost function would be very similar to the TSP, with just the edges between each vehicle's paths removed (e.g. $10 \to 5, 6 \to 9, 2 \to 7$ above).

FB4 - Must - Achieved

The tool must have the ability to read OpenStreetMap data so that the above algorithms can operate on it

The PBF importer is able to read OpenStreetMap files for anywhere on earth into a graph representation that both the shortest path and TSP solvers can operate on.

FB5 - Must - Achieved

The tool must be able to combine the shortest path and travelling salesman solvers on OpenStreetMap data, to accept passenger journey information as a pair of longitude and latitudes (pickup/drop-off) and produce a close to optimal route as a series of longitude and latitudes following the road network

As laid out in figure 4.21, the tool is able to receive longitude and latitude based input, match it to OpenStreetMap data, and produce a close to optimal route between the locations following the road network.

FB6 - Could - Not achieved

The tool could have the ability to accept requested pickup/drop-off times along with journey location information, and attempt to order the pickup/drop-offs as close to the requested times as possible

Similarly to FB3, while this was not achieved within the scope of the project, the overall design of the tool means that significant changes to the functionality of the tool simply require modifying the cost function.

For example, if accepting timed pickup/drop-off requests, I would propose adding the square of the difference between the requested and actual times at each location. This idea, based on linear regression, causes a 20 minute delay to one passenger, to be weighted much worse than 10 minute delays to two passengers.

As discussed in 1.2, it's hard to quantify excellent service, but research has shown that a waiting time of 10 minutes is considered "neutral", whereas more than that is unfavourable [Oli16].

FB7 - Should - Achieved

The tool should have the ability to read TSPLIB data so that it can be compared against existing tools

The TSPLIB reader is able to import the wealth of TSP test instances made available, in order to compare functionality with existing tools (see 5.4.1).

FB8 - Could - Achieved

The tool could have the ability to produce a visual representation of a graph for ease of debugging

The SVG graph exporter (used in figure 4.7), and GraphViz graph exporter (example below in figure 6.1) were very useful for debugging the various graph algorithms.

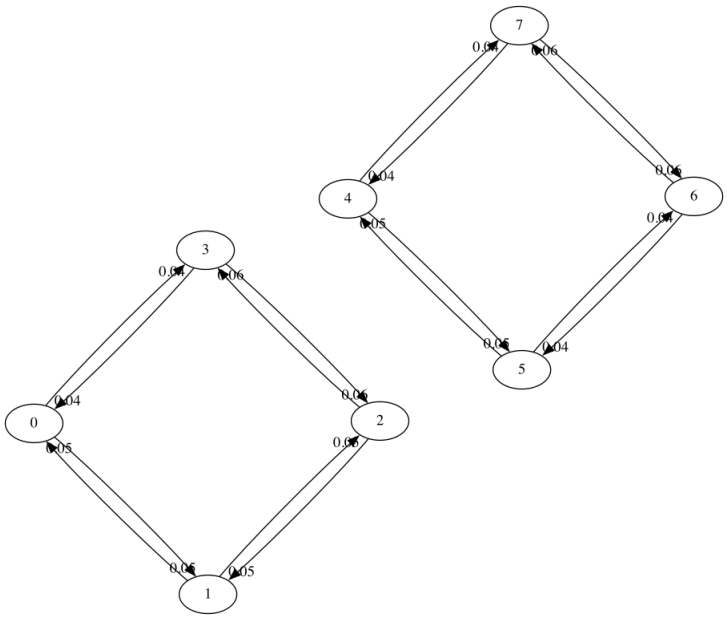


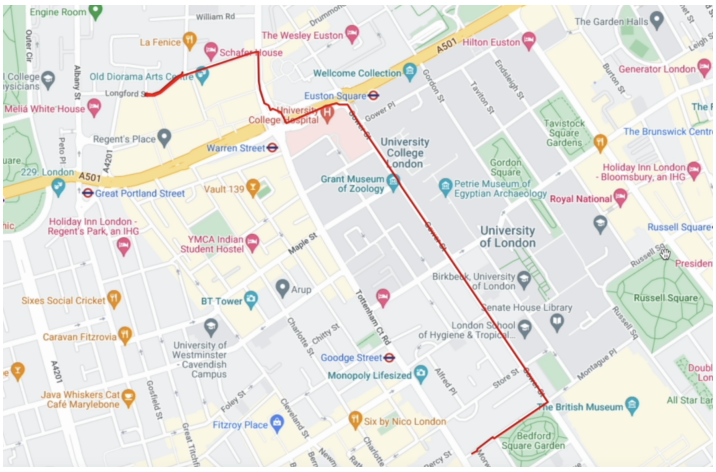
Figure 6.1: Example GraphViz export (separate squares testing instance)

FW1 - Should - Achieved

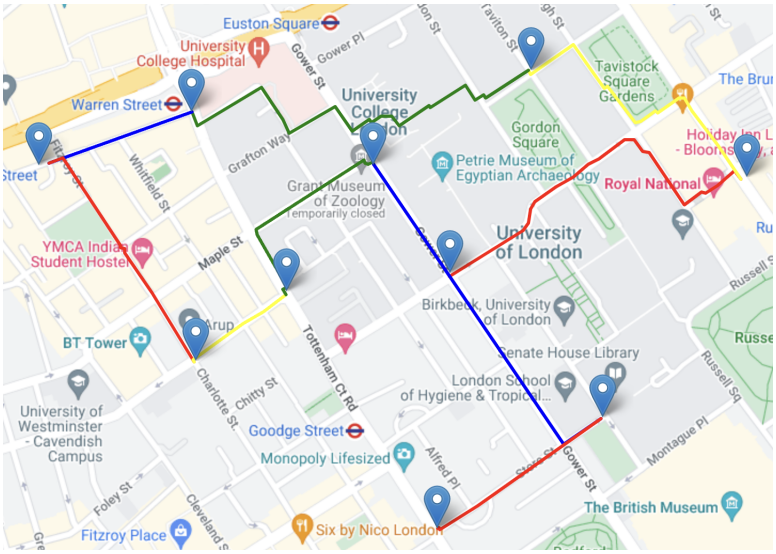
The tool should have a web API interface, which accepts and producing JSON data representing the problem and solution (shortest path/travelling salesman/vehicle routing)

The Rocket-based web interface, described in 4.9, provides a JSON API for both the shortest path and travelling salesman solvers.

This allowed me to produce an interactive demonstration of the tools in Javascript, which can be seen in my Demonstration video. Figure 6.2 shows screenshots of the results, with the only link between the browser-based map and the tool being the JSON API@.



(a) Shortest path solver



(b) TSP solver

Figure 6.2: JSON API web demos [base map from Google Maps]

Non-functional requirements

NB1 - Should - Achieved

The tool should be implemented in a language chosen primarily for speed, whilst taking into account ease-of-use, ecosystem, and community.

The language chosen was Rust, which I believe meets these requirements, as discussed in 4.2.1.

NB2 - Must - Achieved

The tool must have a data structure to represent weighted directed graphs in memory

The tool has two data structures to represent weighted directed graphs in memory, one using an adjacency matrix, and one using an adjacency list. They both conform to a standard API (trait), meaning all the functions that accept graphs can accept graphs of either type (see 4.4 for full details of this).

This has allowed using the different storage formats whenever appropriate, saving space and run time.

NB3 - Should - Achieved

The travelling salesman solver should be extensible, to be able to add multiple parameters to optimise for, and add hard restrictions on outputs

As discussed in FB3 and FB6 above, the hill-climbing design of the TSP solver means that adding new constraints and optimisation variables doesn't require any changes to the fundamentals of the tool.

NB4 - Should - Achieved

The tool should have $> 90\%$ *test coverage*

As discussed in 4.10, the tool has 93.93% test coverage.

NB5 - Should - Achieved

The tool should have good documentation in the form of inline comments

Every module, function, struct, struct field, enum, and enum value has an associated documentation comment. Thanks to `rustdoc` this has been checked by the `missing_docs` lint, and, as discussed in sec:documentation, these comments have been processed into an interactive documentation site, available at https://adnathanail.github.io/broute/brou

NB6 - Should - Achieved

The tool should have benchmarks for each problem, to check for unexpected regressions in performance

The tool has benchmarks in the same repository, which were run on each pull request. These checked the shortest path, connected components, and travelling salesman problem solvers against problem-specific test instances. See 4.6.

NW1 - Should - Achieved

The API should have a response time of $< 1s$

The response time varies according to the size of the requested tour. Figure 5.3 shows that up to a tour length of 50 nodes, a response time of < 1 second is achieved.

Whilst this is smaller than the figure of 1000 I have quoted throughout the project, 50 is still a reasonably sized request. As no size was specified in the requirement, it has been achieved.

NW2 - Should - Achieved

The API should have documentation for its endpoints for ease of use by future integrators

The API documentation is available in the main documentation for the project, here. A version is also available in Appendix B.

6.2.2 Conclusion

	Functional requirements		**Non-functional requirements**	
	Backend	**Web**	**Backend**	**Web**
Must	4/4		1/1	
Should	1/1	1/1	5/5	2/2
Could	1/3			

Every requirement except for FB3 and FB6 were achieved. These requirements were both "could" requirements, and the purpose of the "should" requirement NB3, was to provide the ability to implement these in the future.

6.3 Feature improvements

This section focusses on "critical" features to the project: features which are employed as part of the end-to-end API user experience. Features such as the various graph import/export features (TSPLIB import, SVG export etc.), whilst useful to the development of the project, are not core functionality.

6.3.1 PBF importing

Way accuracy

OpenStreetMap data is very rich with details. Currently the PBF importer simply extracts all "ways" to produce the resulting graph.

The directionality of way segments (whether a road is one way or two way) is currently ignored, and all nodes are assumed to be two-directional.

Whether a way segment is for cars, cyclists, or pedestrians is currently ignore.

Critical for this project: whether a way segment is a bus only lane/road is currently ignored.

Whilst any user generated data is likely to have imperfections, the fact that the connected components algorithm produced a graph with 18,797 nodes, from an original 28,092 (when processing OSM Monaco) is quite notable. This is likely due to various types of way not meshing together nicely.

These are all things which should be considered and deal with before deploying this tool to be used with real vehicles.

Real world distances

As discussed in 4.3.3, the Haversine formula is not a completely accurate method of determining the distance between two coordinates.

Consideration should be given to whether a more accurate method is needed, however this will have an effect on import time.

6.3.2 Connected components algorithm

When trying to load the full API server on OSM Greater London, the PBF importer loads all 8M nodes fine, but my implementation of Tarjan's connected components algorithm reaches the maximum Rust recursion depth. This is due to the recursive nature of the algorithm.

Refactoring the implementation to be iterative instead of recursive should resolve this problem, and the speed implications are not too worrying as this algorithm is only run once, on server loading.

6.3.3 Node geocoding

When users provide coordinates, they need to be matched to their closest OSM node. Currently this is done by simply iterating through all of the nodes, and finding which has the shortest distance.

At large numbers of nodes this will likely become too slow, particularly when handling a TSP request with potentially hundreds of coordinates.

A common approach to deal with this is a quadtree. This is a type of tree datastructure, specially designed for geographic/spatial data, where each node has 4 children, each representing a "quadrant" of the parent. Starting from the top node, the coordinate being searched with is compared with the 4 child quadrants, to see which one it fits in. This is repeated for the child's 4 child quadrants over and over until the matching node is found.

This is analogous to a binary tree, where a node's left children are always less than the value it holds, and the right children are always greater.

If all the nodes were processed into a quadtree on import, this would make locating a matching node much faster. It could take time to do the initial formation of the quadtree, but this moves time complexity from the critical path onto server load times.

6.3.4 Shortest path problem solver

As shown in figure 5.4, the shortest path solver is by far the bottleneck on the size of tour able to be solved by the TSP solver.

As such, the highway hierarchies and contraction hierarchies, discussed in 2.4.8 and 2.4.9, will be critical to be implemented before the tool is deployed.

6.3.5 Travelling salesman problem solver

Surprisingly (to me), this has not ended up being the bottleneck of the project. If the connected components and shortest path algorithms were developed to deal with larger networks, I believe the current iteration of the TSP solver could keep up.

The other two algorithms have a scale of work proportional to the geographic area they are working on, which could reach enormous scales; whereas, the TSP solver has the luxury of a scale of work proportional the number of locations requested, which is always likely to be under 1000.

Nevertheless, improved efficiency would have a number of benefits, including the ability to add lots more repeats (to potentially achieved a better result), and less processing power used (and therefore a lower cost) per request.

The current solver is based on random applications of the 2-opt operation. Attempting a similar approach with the 3-opt operation could achieve better results.

Also, trying a systematic application (trying each pair of nodes sequentially) of 2-opt, 3-opt, or k-opt could yield better results. This is the approach employed by Lin-Kernighan, who preferred to limit their randomisation to the starting state, and repeated

application of the algorithm several times.

Combining Lin-Kernighan (or even LKH) with the 4-opt kick described in 2.5.6 could yield even better results.

6.4 Market research

6.4.1 Comparison with other products

	A to B routing (shortest path)	Parcel delivery route optimisation (Travelling salesman)	Demand responsive travel route optimsiation (Travelling salesman)	Geocoding (matching places to coordinates)
Google Maps	Y	N	N	Y
Graphhopper	Y	Y	N	Y
Amazon location service	Y	N	N	Y
Here maps	Y	Y	N	Y
Tomtom	Y	Y	N	Y
Mapbox	Y	Y	N	Y
Broute (This project)	Y	Y*	Y*†	Y
Via	Y‡	N	Y‡	Y‡

** With the addition of pickup and drop-off location ordering to the cost function (i.e. a drop-off cannot be put before its matching pickup)*

†With OSM data filtered for buses

‡Only available inside proprietary software (no API)

6.4.2 Potential client comment

While not developed for, or in partnership with, any other entity, this product was created with the knowledge it could potentially be of interest to the transport-technology startup Vectare.

Peter Nathanail, the Commercial and Operations Director, had this to say:

"DRT routing is something that only a few companies do, and all the technology is currently baked into proprietary products. Having an API based tool which we can integrate into our existing systems will give a massive improvement in quality of life to our drivers

and service controllers, who will only have to use one piece of software. This in turn will allow us to give a better service to our customers."

6.5 Deliverables

- A Rust-based mapping tool, running on real-world geographic data [GitHub]
- A suite of testing, linting, and benchmarking tools to assist its future development [GitHub]
- Documentation for the Rust code, to assist its future development [Docs]
- Documentation for the JSON API, to assist developers integrating with the tool [Docs]
- This report, detailing the process of research, design, and development

6.6 Conclusion

With continued development, I would prioritise the following improvements to the tool, to make it a minimally usable product:

1. Refactor the connected components algorithm to be iterative as opposed to recursive, allowing much larger OSM graphs to be imported.
2. Read metadata about OSM ways being imported, allowing users to specify routing on foot, on a bike, in a car, or in a bus.
3. Add highway/contraction hierarchies to the shortest path problem solver, to minimise (or remove) the bottleneck on instance size.

Beyond this, there are a few additional features that would likely be required for a full deployment:

1. Extend the TSP solver to accept pairs of pick-up/drop-off locations, with desired pick-up/drop-off times.
2. Extend the TSP solver to a VRP solver, allowing the tool to return paths for multiple vehicles, allowing operators to use it on arbitrary deployment areas.

When taken to scale, there are lots of things that users may require, I think the following would likely be most important:

1. Implement node geocoding with a quad-tree, to speed up individual request times (moving time requirements to start-up)

2. Experiment with other TSP solving approaches, to get closer to optimum paths within reasonable timeframes

3. Add very specific (potentially user-arbitrary) constraints to the TSP solver, such as driving hours, or vehicle fuel requirements.

While there are several potential improvements to be made, they are all improvements of speed and scale. The underlying tool is fully functional end-to-end.

Of the 17 requirements laid out at the start of development, both being incremental improvements to the existing tool.

No DRT routing API currently exists on the market, and Broute (this project) successfully fulfils that role.

Bibliography

[Abu+20] Samah W.G. AbuSalim et al. "Comparative Analysis between Dijkstra and Bellman-Ford Algorithms in Shortest Path Optimization". In: *IOP Conference Series: Materials Science and Engineering* 917.1 (Sept. 2020), p. 012077. DOI: 10.1088/1757-899x/917/1/012077.

[ACR03] David Applegate, William Cook, and André Rohe. "Chained Lin-Kernighan for Large Traveling Salesman Problems". In: *INFORMS Journal on Computing* 15.1 (2003), pp. 82–92. DOI: 10.1287/ijoc.15.1.82.15157.

[Atl] *Why Use Git for Your Organization*. URL: https://www.atlassian.com/git/tutorials/why-git#git-for-developers (visited on 03/31/2023).

[Bae22] Baeldung. *Understanding Time Complexity Calculation for Dijkstra Algorithm*. Nov. 9, 2022. URL: https://www.baeldung.com/cs/dijkstra-time-complexity.

[Bat19] Jean-Pierre Batault. "A Local Search Approach To The Student Sectioning Problem Using Hill Climbing". In: (2019). DOI: 10.13140/RG.2.2.11838.02887.

[Ben] *Charts showing the fastest programs by language (Benchmarks Game)*. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/box-plot-summary-charts.html (visited on 03/10/2023).

[Cam] *Code Fearlessly*. URL: http://cam.ly/blog/2010/12/code-fearlessly/ (visited on 03/31/2023).

[CGR96] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. "Shortest paths algorithms: Theory and experimental evaluation". In: *Mathematical Programming* 73.2 (May 1996), pp. 129–174. DOI: 10.1007/bf02592101.

[Cho+21] Xiaochen Chou et al. "3-opt Metaheuristics for the Probabilistic Orienteering Problem". In: *2021 The 8th International Conference on Industrial Engineering and Applications(Europe)*. ACM, 2021. DOI: 10.1145/3463858.3463867.

[Chr22] Nicos Christofides. "Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem". In: *Operations Research Forum* 3.1 (2022), p. 20. ISSN: 2662-2556. DOI: 10.1007/s43069-021-00101-z.

[Cil] *What Bus Passengers Want*. The Chartered Institute of Logistics and Transport, 2010. URL: http://www.ciltscotland.com/arcmr230310.php.

[Coo01] Stephen Cook. "The P versus NP problem". In: *The millenium prize problems* (Feb. 2001).

[Coo12] William J. Cook. *In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation*. Princeton University Press, 2012.

[Coo99] Jennifer L. Cook William; Rich. "A Parallel Cutting-Plane Algorithm for the Vehicle Routing Problem with Time Windows". In: *Computation and Applied Mathematics, Rice University* (1999).

[Cro58] G. A. Croes. "A Method for Solving Traveling-Salesman Problems". In: *Operations Research* 6.6 (1958), pp. 791–812. ISSN: 0030364X, 15265463. URL: http://www.jstor.org/stable/167074 (visited on 03/05/2023).

[Dar23] David Darling. "Icosian Game". In: *daviddarling.info* (2023). URL: https://www.daviddarling.info/encyclopedia/I/Icosian_Game.html (visited on 02/27/2023).

[DFJ54] G. Dantzig, R. Fulkerson, and S. Johnson. "Solution of a Large-Scale Traveling-Salesman Problem". In: *Journal of the Operations Research Society of America* 2.4 (1954), pp. 393–410.

[DGJ16] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. "Implementation Challenge for Shortest Paths". In: *Encyclopedia of Algorithms*. Springer New York, 2016, pp. 947–951. DOI: 10.1007/978-1-4939-2864-4_181.

[Dij59] E. W. Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. DOI: 10.1007/bf01386390.

[Dim] *9th DIMACS Implementation Challenge - Shortest Paths*. URL: http://www.diag.uniroma1.it/challenge9/ (visited on 03/31/2023).

[DIM01] DIMACS. *8th DIMACS Implementation Challenge: The Traveling Salesman Problem*. May 2001. URL: http://dimacs.rutgers.edu/archive/Challenges/TSP/index.html (visited on 02/28/2023).

[DLA09] Vašek Chvátal William Cook Daniel G. Espinoza Marcos Goycoolea Keld Helsgaun David L. Applegate Robert E. Bixby. "Certification of an optimal TSP tour through 85,900 cities". In: *Operations Research Letters* 37.1 (2009), pp. 11–15.

[Dum+95] Yvan Dumas et al. "An Optimal Algorithm for the Traveling Salesman Problem with Time Windows". In: *Operations Research* 43.2 (1995), pp. 367–371. DOI: 10.1287/opre.43.2.367.

[Eng] Bank of England. *Inflation calculator*. URL: https://www.bankofengland.co.uk/monetary-policy/inflation/inflation-calculator.

[Eul41] Leonhard Euler. "Solutio problematis ad geometriam situs pertinentis". In: *Commentarii academiae scientiarum Petropolitanae, Volume 8, pp. 128-140* (1741). URL: https://scholarlycommons.pacific.edu/euler-works/53/.

[Flo56] Merrill M. Flood. "The Traveling-Salesman Problem". In: *Operations Research* 4.1 (1956), pp. 61–75. ISSN: 0030364X, 15265463. URL: http://www.jstor.org/stable/167517 (visited on 03/05/2023).

[Gee19] GeeksforGeeks. *Comparison of Dijkstra's and Floyd–Warshall algorithms*. May 21, 2019. URL: https://www.geeksforgeeks.org/comparison-dijkstras-floyd-warshall-algorithms/.

[Gei+08] Robert Geisberger et al. "Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks". In: *Experimental Algorithms*. Springer Berlin Heidelberg, 2008, pp. 319–333. DOI: 10.1007/978-3-540-68552-4_24.

[GH91] Martin Grötschel and Olaf Holland. "Solution of large-scale symmetric travelling salesman problems". In: *Mathematical Programming* 51.1-3 (1991), pp. 141–202. DOI: 10.1007/bf01586932.

[Goo] *Directions API Usage and Billing*. URL: https://developers.google.com/maps/documentation/directions/usage-and-billing#pricing-for-product (visited on 03/31/2023).

[Gre] *7 ways the UK's transport system is unfair*. Jan. 2021. URL: https://www.greenpeace.org.uk/news/uk-transport-unfair-car-dependence-social-justice/ (visited on 04/19/2023).

[Gua22] The Guardian. *UK nurses' pay rise demand 'remarkably high', says minister as strike looms*. 2022. URL: https://www.theguardian.com/society/2022/nov/10/uk-nurses-pay-rise-strike-minister-chris-heaton-harris-nhs.

[HNR68] Peter Hart, Nils Nilsson, and Bertram Raphael. "A Formal Basis for the Heuristic Determination of Minimum Cost Paths". In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (1968), pp. 100–107. DOI: 10.1109/tssc.1968.300136.

[Hoa09] Tony Hoare. *Null References: The Billion Dollar Mistake*. QCon conference. Aug. 2009.

[Hou10] Communities & Local Government Ministry of Housing. *Local authority revenue expenditure and financing England: 2007 to 2008 individual local authority data*. Sept. 2010. URL: https://www.gov.uk/government/statistics/local-authority-revenue-expenditure-and-financing-england-2007-to-2008-individual-local-authority-data.

[Hou18] Communities & Local Government Ministry of Housing. *Local authority revenue expenditure and financing England: 2018 to 2019 budget individual local authority data*. June 2018. URL: https://www.gov.uk/government/statistics/local-authority-revenue-expenditure-and-financing-england-2018-to-2019-budget-individual-local-authority-data.

[Icp] *The real 10 algorithms that dominate our world*. URL: https://www.icesi.edu.co/blogs/icpc/2019/01/21/real10thatruletheworld/ (visited on 03/31/2023).

[Iñi23] Agustín Iñiguez. *Graph theory and its uses with 5 examples of real life problems*. Jan. 13, 2023. URL: https://www.xomnia.com/post/graph-theory-and-its-uses-with-examples-of-real-life-problems/.

[Kar72] Richard M. Karp. "Reducibility among Combinatorial Problems". In: *Complexity of Computer Computations*. Springer US, 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[Kir56] Thomas P. Kirkman. "On the Representation of Polyedra". In: *Philosophical Transactions of the Royal Society of London* 146 (1856), pp. 413–418.

[KKG21] Anna R. Karlin, Nathan Klein, and Shayan Oveis Gharan. "A (slightly) improved approximation algorithm for metric TSP". In: *Proceedings of the 53rd Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 2021. DOI: 10.1145/3406325.3451009.

[LK73] S. Lin and B. W. Kernighan. "An Effective Heuristic Algorithm for the Traveling-Salesman Problem". In: *Operations Research* 21.2 (1973), pp. 498–516. ISSN: 0030364X, 15265463. URL: http://www.jstor.org/stable/169020 (visited on 03/05/2023).

[Lkh] *LKH*. Nov. 2022. URL: http://webhotel4.ruc.dk/~keld/research/LKH/ (visited on 03/05/2023).

[Mah] Arthur Mahéo. *Local TSP Heuristics in Python*. URL: https://arthur.maheo.net/python-local-tsp-heuristics/.

[Mar91] Steve; Felten Edward Martin Olivier; Otto. "Large-Step Markov Chains for the Traveling Salesman Problem". In: *Complex Systems* 5.3 (1991).

[Men32] Karl Menger. "Das botenproblem". In: *Ergebnisse eines Mathematischen Kolloquiums 2* (1932).

[Oli16] Ivana Olivková. "Evaluation of Quality Public Transport Criteria in Terms of Passenger Satisfaction". In: *Transport and Telecommunication Journal* 17.1 (2016), pp. 18–27. DOI: 10.1515/ttj-2016-0003. URL: https://doi.org/10.1515%2Fttj-2016-0003.

[Osm] *PBF Format - OpenStreetMap Wiki*. URL: https://wiki.openstreetmap.org/wiki/PBF_Format (visited on 03/21/2023).

[OT22] Google OR-Tools. *C++ Reference: christofides*. Sept. 2022. URL: https://developers.google.com/optimization/reference/graph/christofides (visited on 03/05/2023).

[Par85] UK Parliament. *Transport Act 1985*. 1985.

[Pcg] *PCG, A Family of Better Random Number Generators*. URL: https://www.pcg-random.org/index.html (visited on 03/31/2023).

[Pyt] *doctest — Test interactive Python examples*. Mar. 2023. URL: https://docs.python.org/3/library/doctest.html (visited on 03/10/2023).

[Rob49] Julia Robinson. "On the Hamiltonian Game (A Traveling Salesman Problem)". In: *RAND PROJECT AIR FORCE ARLINGTON VA* (1949).

[Rus] URL: https://doc.rust-lang.org/stable/std/collections/struct.BinaryHeap.html (visited on 03/31/2023).

[Rut] *TSP Challenge - Optimum lengths*. URL: http://archive.dimacs.rutgers.edu/Challenges/TSP/opts.html.

[SS06] Peter Sanders and Dominik Schultes. "Engineering Highway Hierarchies". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 804–816. DOI: 10.1007/11841036_71.

[Sta] *2022 Developer Survey - Integrated development environment*. URL: https://survey.stackoverflow.co/2022/#section-most-popular-technologies-integrated-development-environment (visited on 03/31/2023).

[Tra21] Department for Transport. *National Bus Strategy For England: Bus Back Better*. 2021.

[Wat13a] University of Waterloo. *Concorde TSP Solver*. July 2013. URL: http://www.math.uwaterloo.ca/tsp/concorde.html.

[Wat13b] University of Waterloo. *Optimal 85,900-Point Tour*. July 2013. URL: https://www.math.uwaterloo.ca/tsp/pla85900/index.html.

[Wat14] University of Waterloo. *TSP Art Instances*. July 2014. URL: https://www.math.uwaterloo.ca/tsp/data/art/.

[Wat16] University of Waterloo. *History of the TSP with road distances*. Dec. 2016. URL: https://www.math.uwaterloo.ca/tsp/us/history.html (visited on 02/28/2023).

Appendix A

Data tables

A.1 Shortest path problem benchmarks

	DIMACS d1291 (1291 nodes)	Random graph (3000 nodes)	OSM Monaco (9936 nodes)	DIMACS USA-road-d.NY (264346 nodes)
Naive Dijkstra (`VecDeque`)	4.4886 ms	28.762 ms	133.58 ms	-
Better Dijkstra (`BinaryHeap`)	3.1279 ms *-30%*	21.770 ms *-24%*	981.10 µs *-99%*	-
Dijkstra + path-tracing + node data	6.9048 ms *+121%*	76.816 ms *+253%*	896.73 µs *-9%*	58.103 ms
A* $h(n)$ on-the-fly	336.53 µs *-95%*	133.84 ms *+1642%*	772.91 µs *-14%*	23.864 ms *-59%*
A* precomputing $h(n)$	406.60 µs *+21%*	-	2.0569 ms *+166%*	82.730 ms *+247%*
A* caching `to_nodes`	266.47 µs *-21%*	-	658.07 µs *-15%*	16.996 ms *-29%*

A.2 Travelling salesman problem benchmarks

	Min tour length	**Mean tour length**	**Mean run time**
10 iterations			
Simulated annealing (naive)	709,598.77	729,309.28	2.50 s
Simulated annealing (new mutation function)	464,836.28 *-34%*	479,964.65 *-34%*	2.43 s *-3%*
Simulated annealing ($N = 2$)	337,397.17 *-27%*	344,141.30 *-28%*	4.89 s *+101%*
Simulated annealing ($N = 5$)	212,834.24 *-37%*	241,372.85 *-30%*	13.16 s *+169%*
Simulated annealing (tuned)	200,453.14 *-6%*	208,318.76 *-14%*	13.19 s *0%*
Hill climbing (tuned)	77,261.05 *-61%*	77,402.36 *-63%*	160.19 s *+1114%*
Hill climbing (incremental path lengths)	72,813.92 *-6%*	74,773.02 *-3%*	7.32 s *-95%*
Hill climbing (conditional mutation)	73,005.33 *0%*	74,536.47 *0%*	4.35 s *-41%*
Hill climbing (faster RNG)	73,011.77 *0%*	74,817.98 *0%*	848.1 ms *-81%*
25 iterations			
Hill climbing (duplicate array accesses)	72,589.60 *-1%*	74,817.98 *0%*	848.1 ms *0%*
Hill climbing (required ordering)	75,307.12 *+4%*	74,817.98 *0%*	742.8 ms *-12%*
Hill climbing (Rust optimised)	72,178.97 *-4%*	74,863.69 *0%*	39.68 ms *-95%*

A.3 Travelling salesman problem - Simulated annealing tuning

A.3.1 Tuning T_0 and T_{min}

α	N	T_0	T_{min}	**Path length**	**Duration (s)**
0.95	1	1	0.1	1702418	0.01
0.95	1	1	0.001	1699248	0.01
0.95	1	1	0.00001	1637037	0.02
0.95	1	1	0.0000001	1620121	0.03
0.95	1	1	0.000000001	1582331	0.04
0.95	1	10	0.1	1698903	0.01
0.95	1	10	0.001	1664087	0.02
0.95	1	10	0.00001	1656222	0.03
0.95	1	10	0.0000001	1604113	0.04
0.95	1	10	0.000000001	1557835	0.04
0.95	1	100	0.1	1659613	0.01
0.95	1	100	0.001	1656474	0.02
0.95	1	100	0.00001	1633906	0.03
0.95	1	100	0.0000001	1574209	0.04
0.95	1	100	0.000000001	1573291	0.05
0.95	1	1000	0.1	1656184	0.02
0.95	1	1000	0.001	1612221	0.03
0.95	1	1000	0.00001	1600806	0.04
0.95	1	1000	0.0000001	1593135	0.04
0.95	1	1000	0.000000001	1567376	0.05
0.95	1	10000	0.1	1650048	0.02
0.95	1	10000	0.001	1634493	0.03
0.95	1	10000	0.00001	1618106	0.04
0.95	1	10000	0.0000001	1568125	0.05
0.95	1	10000	0.000000001	1529780	0.06
0.95	5	1	0.1	1624407	0.02
0.95	5	1	0.001	1503687	0.07
0.95	5	1	0.00001	1391214	0.11
0.95	5	1	0.0000001	1314860	0.16
0.95	5	1	0.000000001	1247283	0.19
0.95	5	10	0.1	1588405	0.04
0.95	5	10	0.001	1466230	0.09

α	N	T_0	T_{min}	**Path length**	**Duration (s)**
0.95	5	10	0.00001	1331988	0.13
0.95	5	10	0.0000001	1302012	0.17
0.95	5	10	0.000000001	1176857	0.22
0.95	5	100	0.1	1498482	0.07
0.95	5	100	0.001	1417651	0.11
0.95	5	100	0.00001	1302954	0.15
0.95	5	100	0.0000001	1235253	0.2
0.95	5	100	0.000000001	1190197	0.24
0.95	5	1000	0.1	1488506	0.09
0.95	5	1000	0.001	1358874	0.13
0.95	5	1000	0.00001	1260190	0.18
0.95	5	1000	0.0000001	1211216	0.22
0.95	5	1000	0.000000001	1158436	0.26
0.95	5	10000	0.1	1428047	0.11
0.95	5	10000	0.001	1350283	0.15
0.95	5	10000	0.00001	1225416	0.2
0.95	5	10000	0.0000001	1238889	0.24
0.95	5	10000	0.000000001	1146553	0.28
0.95	10	1	0.1	1542441	0.04
0.95	10	1	0.001	1344457	0.13
0.95	10	1	0.00001	1208714	0.22
0.95	10	1	0.0000001	1093858	0.3
0.95	10	1	0.000000001	1020478	0.39
0.95	10	10	0.1	1460569	0.09
0.95	10	10	0.001	1302519	0.18
0.95	10	10	0.00001	1116372	0.27
0.95	10	10	0.0000001	1074922	0.35
0.95	10	10	0.000000001	979509	0.43
0.95	10	100	0.1	1373474	0.13
0.95	10	100	0.001	1214578	0.22
0.95	10	100	0.00001	1136997	0.31
0.95	10	100	0.0000001	1035210	0.39
0.95	10	100	0.000000001	947017	0.48
0.95	10	1000	0.1	1270346	0.17
0.95	10	1000	0.001	1184124	0.26
0.95	10	1000	0.00001	1062924	0.35
0.95	10	1000	0.0000001	979446	0.44
0.95	10	1000	0.000000001	902930	0.52

α	N	T_0	T_{min}	**Path length**	**Duration (s)**
0.95	10	10000	0.1	1236897	0.22
0.95	10	10000	0.001	1175580	0.31
0.95	10	10000	0.00001	1064281	0.39
0.95	10	10000	0.0000001	965047	0.48
0.95	10	10000	0.000000001	913252	0.57
0.99	1	1	0.1	1656938	0.02
0.99	1	1	0.001	1497871	0.07
0.99	1	1	0.00001	1398634	0.11
0.99	1	1	0.0000001	1336106	0.16
0.99	1	1	0.000000001	1247617	0.2
0.99	1	10	0.1	1567075	0.05
0.99	1	10	0.001	1463827	0.09
0.99	1	10	0.00001	1359546	0.13
0.99	1	10	0.0000001	1246785	0.18
0.99	1	10	0.000000001	1193612	0.22
0.99	1	100	0.1	1530615	0.07
0.99	1	100	0.001	1430733	0.11
0.99	1	100	0.00001	1331939	0.16
0.99	1	100	0.0000001	1219821	0.2
0.99	1	100	0.000000001	1152691	0.25
0.99	1	1000	0.1	1498087	0.09
0.99	1	1000	0.001	1394914	0.14
0.99	1	1000	0.00001	1269465	0.18
0.99	1	1000	0.0000001	1172355	0.22
0.99	1	1000	0.000000001	1156218	0.27
0.99	1	10000	0.1	1415569	0.11
0.99	1	10000	0.001	1360615	0.16
0.99	1	10000	0.00001	1237663	0.2
0.99	1	10000	0.0000001	1190102	0.25
0.99	1	10000	0.000000001	1123574	0.29
0.99	5	1	0.1	1409819	0.11
0.99	5	1	0.001	1068595	0.33
0.99	5	1	0.00001	893435	0.55
0.99	5	1	0.0000001	790036	0.77
0.99	5	1	0.000000001	724526	0.99
0.99	5	10	0.1	1200680	0.23
0.99	5	10	0.001	967019	0.45
0.99	5	10	0.00001	854059	0.67

α	N	T_0	T_{min}	**Path length**	**Duration (s)**
0.99	5	10	0.0000001	751920	0.89
0.99	5	10	0.000000001	678978	1.11
0.99	5	100	0.1	1081222	0.34
0.99	5	100	0.001	905415	0.55
0.99	5	100	0.00001	783864	0.78
0.99	5	100	0.0000001	717629	1
0.99	5	100	0.000000001	671855	1.23
0.99	5	1000	0.1	984973	0.44
0.99	5	1000	0.001	880867	0.66
0.99	5	1000	0.00001	773136	0.88
0.99	5	1000	0.0000001	663821	1.11
0.99	5	1000	0.000000001	629256	1.33
0.99	5	10000	0.1	952776	0.56
0.99	5	10000	0.001	838980	0.78
0.99	5	10000	0.00001	744173	0.99
0.99	5	10000	0.0000001	687515	1.21
0.99	5	10000	0.000000001	625524	1.42
0.99	10	1	0.1	1193773	0.22
0.99	10	1	0.001	868184	0.67
0.99	10	1	0.00001	683755	1.11
0.99	10	1	0.0000001	600126	1.58
0.99	10	1	0.000000001	526808	2.05
0.99	10	10	0.1	982731	0.46
0.99	10	10	0.001	769756	0.92
0.99	10	10	0.00001	636638	1.38
0.99	10	10	0.0000001	556622	1.83
0.99	10	10	0.000000001	489973	2.27
0.99	10	100	0.1	839469	0.68
0.99	10	100	0.001	665926	1.14
0.99	10	100	0.00001	598126	1.59
0.99	10	100	0.0000001	524485	2.05
0.99	10	100	0.000000001	470722	2.5
0.99	10	1000	0.1	753818	0.91
0.99	10	1000	0.001	640294	1.37
0.99	10	1000	0.00001	559002	1.83
0.99	10	1000	0.0000001	506776	2.28
0.99	10	1000	0.000000001	468829	2.73
0.99	10	10000	0.1	729237	1.14

α	N	T_0	T_{min}	**Path length**	**Duration (s)**
0.99	10	10000	0.001	617854	1.6
0.99	10	10000	0.00001	536712	2.01
0.99	10	10000	0.0000001	503985	2.45
0.99	10	10000	0.000000001	465600	2.89
0.995	1	1	0.1	1576516	0.05
0.995	1	1	0.001	1373812	0.13
0.995	1	1	0.00001	1179026	0.22
0.995	1	1	0.0000001	1088100	0.31
0.995	1	1	0.000000001	1005708	0.4
0.995	1	10	0.1	1437011	0.09
0.995	1	10	0.001	1284351	0.18
0.995	1	10	0.00001	1152937	0.27
0.995	1	10	0.0000001	1048122	0.36
0.995	1	10	0.000000001	967900	0.45
0.995	1	100	0.1	1341738	0.13
0.995	1	100	0.001	1186968	0.22
0.995	1	100	0.00001	1115056	0.31
0.995	1	100	0.0000001	1014437	0.4
0.995	1	100	0.000000001	936462	0.49
0.995	1	1000	0.1	1292249	0.18
0.995	1	1000	0.001	1130752	0.27
0.995	1	1000	0.00001	1035681	0.36
0.995	1	1000	0.0000001	965608	0.45
0.995	1	1000	0.000000001	901305	0.53
0.995	1	10000	0.1	1238769	0.22
0.995	1	10000	0.001	1143649	0.31
0.995	1	10000	0.00001	1039855	0.4
0.995	1	10000	0.0000001	987172	0.49
0.995	1	10000	0.000000001	916623	0.58
0.995	5	1	0.1	1209180	0.22
0.995	5	1	0.001	832723	0.67
0.995	5	1	0.00001	697534	1.12
0.995	5	1	0.0000001	584640	1.56
0.995	5	1	0.000000001	519580	2
0.995	5	10	0.1	984966	0.45
0.995	5	10	0.001	745866	0.89
0.995	5	10	0.00001	632339	1.33
0.995	5	10	0.0000001	560044	1.79

α	N	T_0	T_{min}	**Path length**	**Duration (s)**
0.995	5	10	0.000000001	494925	2.23
0.995	5	100	0.1	864236	0.67
0.995	5	100	0.001	692115	1.11
0.995	5	100	0.00001	591141	1.55
0.995	5	100	0.0000001	539598	2.01
0.995	5	100	0.000000001	476525	2.45
0.995	5	1000	0.1	764684	0.9
0.995	5	1000	0.001	646644	1.34
0.995	5	1000	0.00001	558742	1.78
0.995	5	1000	0.0000001	511562	2.23
0.995	5	1000	0.000000001	481023	2.65
0.995	5	10000	0.1	749470	1.11
0.995	5	10000	0.001	614539	1.55
0.995	5	10000	0.00001	542257	2
0.995	5	10000	0.0000001	501643	2.42
0.995	5	10000	0.000000001	477944	2.89
0.995	10	1	0.1	988643	0.45
0.995	10	1	0.001	648692	1.33
0.995	10	1	0.00001	493989	2.22
0.995	10	1	0.0000001	418680	3.09
0.995	10	1	0.000000001	380870	4.01
0.995	10	10	0.1	744783	0.89
0.995	10	10	0.001	567889	1.78
0.995	10	10	0.00001	467851	2.65
0.995	10	10	0.0000001	389446	3.55
0.995	10	10	0.000000001	343892	4.43
0.995	10	100	0.1	651929	1.33
0.995	10	100	0.001	493461	2.22
0.995	10	100	0.00001	433260	3.12
0.995	10	100	0.0000001	383527	4.01
0.995	10	100	0.000000001	331932	4.87
0.995	10	1000	0.1	557806	1.78
0.995	10	1000	0.001	478644	2.66
0.995	10	1000	0.00001	405152	3.56
0.995	10	1000	0.0000001	363117	4.45
0.995	10	1000	0.000000001	319718	5.31
0.995	10	10000	0.1	573580	2.2
0.995	10	10000	0.001	466323	3.11

α	N	T_0	T_{min}	**Path length**	**Duration (s)**
0.995	10	10000	0.00001	403257	3.95
0.995	10	10000	0.0000001	365228	4.83
0.995	10	10000	0.000000001	320216	5.72

A.3.2 Tuning T_0

α	N	T_0	**Path length**	**Duration (s)**
0.95	1	10	1624182	0.03
0.95	1	100	1635077	0.03
0.95	1	1000	1616434	0.04
0.95	5	10	1351481	0.13
0.95	5	100	1295731	0.15
0.95	5	1000	1266449	0.18
0.95	10	10	1128244	0.27
0.95	10	100	1140197	0.31
0.95	10	1000	1055590	0.36
0.99	1	10	1323807	0.15
0.99	1	100	1297810	0.16
0.99	1	1000	1269612	0.18
0.99	5	10	857148	0.68
0.99	5	100	814935	0.78
0.99	5	1000	743849	0.91
0.99	10	10	634672	1.34
0.99	10	100	582301	1.56
0.99	10	1000	563702	1.81
0.995	1	10	1141954	0.27
0.995	1	100	1108929	0.31
0.995	1	1000	1035169	0.36
0.995	5	10	642391	1.35
0.995	5	100	605658	1.57
0.995	5	1000	565015	1.78
0.995	10	10	462902	2.68
0.995	10	100	421904	3.11
0.995	10	1000	405543	3.55

A.3.3 Tuning α and N

α	N	**Path length**	**Duration (s)**
0.99	5	820871	0.67
0.99	10	641577	1.33
0.99	50	308423	6.56
0.99	100	212357	13.09
0.995	5	646107	1.31
0.995	10	456231	2.64
0.995	50	206505	13.13
0.995	100	150122	26.25
0.999	5	293462	6.59
0.999	10	212834	13.16
0.999	50	98767	65.67
0.999	100	81459	132.58
0.9995	5	205197	13.18
0.9995	10	149298	26.33
0.9995	50	81414	131.36
0.9995	100	67503	262.64

A.3.4 Comparing simulated annealing to hill climbing

α	N	Simulated annealing		i	Hill climbing	
		Path length	**Duration (s)**		**Path length**	**Duration (s)**
0.99	5	820871	0.67	6875	843581	0.66
0.99	10	641577	1.33	13750	610891	1.33
0.99	50	308423	6.56	68750	291165	6.67
0.99	100	212357	13.09	137500	208157	13.60
0.995	5	646107	1.31	13785	630172	1.35
0.995	10	456231	2.64	27570	447412	2.68
0.995	50	206505	13.13	137850	199020	13.37
0.995	100	150122	26.25	275700	143848	26.63
0.999	5	293462	6.59	69045	294065	6.69
0.999	10	212834	13.16	138090	207945	13.63
0.999	50	98767	65.67	690450	98966	67.31
0.999	100	81459	132.58	1380900	80687	133.18
0.9995	5	205197	13.18	138125	210560	13.28
0.9995	10	149298	26.33	276250	146624	26.49
0.9995	50	81414	131.36	1381250	79646	131.51
0.9995	100	67503	262.64	2762500	67010	262.65

A.4 Shortest path problem testing

Instance	Loading time (s)	# nodes	Path length (nodes)		Runtime (s)	
			Mean	Std Dev.	Mean	Std Dev.
NY	0.152	264346	375.39	193.72	0.0270	0.0178
BAY	0.156	321270	486.30	236.30	0.0403	0.0232
COL	0.206	435666	725.98	376.05	0.0493	0.0287
FLA	0.582	1070376	990.00	593.69	0.1522	0.0912
NW	0.641	1207945	1136.33	538.59	0.1902	0.1066
NE	0.836	1524453	1333.56	770.31	0.2466	0.1306
CAL	1.115	1890815	1268.06	621.33	0.3256	0.1624
LKS	1.602	2758119	2525.77	1556.78	0.4799	0.2486
E	2.041	3598623	2156.32	1231.24	0.6319	0.3478
W	3.912	6262104	2646.50	1315.53	1.2334	0.7015
CTR	9.930	14081816	3665.81	1741.26	3.3520	1.8783
USA	18.884	23947347	5093.53	2708.35	5.4988	3.1800

A.5 Travelling salesman problem testing

A.5.1 DIMACS instances (N^2 iterations)

Instance	Loading time (s)	num nodes	Optimum path length	Path length		Runtime (s)	
				Mean	Std Dev.	Mean	Std Dev.
dsj1000	0.02	1000	18660188	24409670.71	434777.96	0.0123	0.00227
pr1002	0.02	1002	259045	339914.02	5037.48	0.0103	0.00122
u1060	0.03	1060	224094	293705.59	4915.37	0.031	0.00141
vm1084	0.03	1084	239297	326639.24	6118.28	0.0312	0.00145
pcb1173	0.03	1173	56892	75659.59	932.15	0.0333	0.00166
d1291	0.04	1291	50801	74924.14	1301.51	0.0359	0.00144
rl1304	0.04	1304	252948	373044.26	6665.93	0.0375	0.00244
rl1323	0.04	1323	270199	396575.64	6403.11	0.0389	0.00243
nrw1379	0.05	1379	56638	71465.65	883.23	0.04	0.0024
fl1400	0.05	1400	20127	26291.45	1109.72	0.0399	0.00249
u1432	0.05	1432	152970	194500.89	2004.8	0.0402	0.00187
fl1577	0.06	1577	22249	32981.73	958.41	0.0468	0.0021
d1655	0.07	1655	62128	85650.56	1063.71	0.0528	0.00282
vm1748	0.08	1748	336556	452359.69	6189.65	0.0591	0.00311
u1817	0.08	1817	57201	80727.71	961.2	0.0623	0.00378

Instance	Loading time (s)	num nodes	Optimum path length	Path length		Runtime (s)	
				Mean	Std Dev.	Mean	Std Dev.
rl1889	0.08	1889	316536	461480.55	7284.46	0.0709	0.00447
d2103	0.1	2103	80450	115706.54	1476.33	0.1329	0.00832
u2152	0.11	2152	64253	90005.07	1002.96	0.1435	0.01211
u2319	0.12	2319	234256	270389.63	1860.73	0.1613	0.01504
pr2392	0.13	2392	378032	511329.88	5571.27	0.1806	0.01685
pcb3038	0.23	3038	137694	180090.66	1322.5	0.316	0.02295
fl3795	0.34	3795	28772	42990.65	1040.49	0.5287	0.02687
fnl4461	0.48	4461	182566	230523.33	1454.47	0.8283	0.02802
rl5915	0.84	5915	565530	837299.28	7163.87	1.489	0.04107
rl5934	0.84	5934	556045	817771.91	6529.96	1.4571	0.04903
pla7397	1.28	7397	23260728	30921567.63	318201.17	2.2442	0.02373
rl11849	3.54	11849	923288	1298452.01	6578.28	6.2732	0.05878
usa13509	4.67	13509	19982859	25893118.71	148488.37	8.2872	0.08709
brd14051	5.16	14051	469385	592526.96	2611.57	8.7317	0.03576
d15112	5.49	15112	1573084	1981875.03	6703.51	10.0371	0.07632
d18512	8.5	18512	645238	812718.84	2297.09	16.4992	0.11946
pla33810	33.3	33810	66048945	89110150.44	319205.14	67.9235	0.74189

A.5.2 DIMACS instances ($2N^2$ iterations)

Instance	Loading time (s)	num nodes	Optimum path length	Path length		Runtime (s)	
				Mean	Std Dev.	Mean	Std Dev.
dsj1000	0.02	1000	18660188	22176626.040	412580.57	0.0182	0.00075
pr1002	0.02	1002	259045	308796.860	4344.56	0.0185	0.00142
u1060	0.03	1060	224094	267405.480	4414.75	0.0602	0.00192
vm1084	0.03	1084	239297	290592.550	4933.07	0.0604	0.00112
pcb1173	0.03	1173	56892	68685.070	833.75	0.0638	0.00081
d1291	0.04	1291	50801	65866.410	1182	0.0687	0.00212
rl1304	0.04	1304	252948	321356.290	6558.57	0.0693	0.00115
rl1323	0.04	1323	270199	340339.590	5896.42	0.0706	0.00157
nrw1379	0.05	1379	56638	66300.040	686.08	0.0739	0.00236
fl1400	0.05	1400	20127	24038.330	883.5	0.0737	0.00226
u1432	0.05	1432	152970	181701.830	1802.45	0.0754	0.00191
fl1577	0.06	1577	22249	28304.240	876.41	0.0875	0.00268
d1655	0.07	1655	62128	76732.830	1112.8	0.0981	0.00448
vm1748	0.08	1748	336556	404650.230	6574.35	0.1146	0.00396

Instance	Loading time (s)	num nodes	Optimum path length	Path length		Runtime (s)	
				Mean	Std Dev.	Mean	Std Dev.
u1817	0.08	1817	57201	72353.710	999.84	0.1379	0.01767
rl1889	0.08	1889	316536	397732.730	6900.71	0.14	0.01248
d2103	0.1	2103	80450	103362.450	1469.67	0.261	0.02004
u2152	0.11	2152	64253	81280.330	874.85	0.2833	0.03905
u2319	0.12	2319	234256	259277.230	1640.18	0.2939	0.0071
pr2392	0.13	2392	378032	458999.530	4621.53	0.3377	0.04068
pcb3038	0.23	3038	137694	164756.100	1207.48	0.5703	0.02185
fl3795	0.34	3795	28772	36634.500	1076.39	0.978	0.03522
fnl4461	0.48	4461	182566	213287.440	1292.31	1.6609	0.04168
rl5915	0.84	5915	565530	721134.050	6353.58	2.9326	0.07175
rl5934	0.84	5934	556045	708540.550	6366.87	2.8785	0.07392
pla7397	1.28	7397	23260728	27957940.150	288992.49	4.4357	0.08869
rl11849	3.54	11849	923288	1152074.150	6995.77	11.6806	0.02925
usa13509	4.67	13509	19982859	23559018.480	188862.26	15.4471	0.0455
brd14051	5.16	14051	469385	547412.020	2031.7	16.9731	0.11166
d15112	5.49	15112	1573084	1829804.820	5996.17	19.4756	0.08794
d18512	8.5	18512	645238	751325.460	2227.95	32.6251	0.48955
pla33810	33.3	33810	66048945	80518468.180	271829.25	132.3775	1.94843

A.5.3 OSM Monaco (API end-to-end)

Num nodes	Duration mean	Duration std. dev.
5	18.15	0.004486368
10	58.74	0.00596426
25	280.64	0.012237255
50	1030.79	0.059125848
100	3886.09	0.083088278

A.5.4 OSM Monaco (Step-by-step breakdown)

Num nodes	Geocoding (s)		Abstracting (s)		TSP (s)	
	Mean	Std dev.	Mean	Std dev.	Mean	Std dev.
5	0.0010	0.00000	0.0135	0.003	0	0
10	0.0020	0.00000	0.0533	0.005	0	0
25	0.0070	0.00000	0.2627	0.014	0	0
50	0.0140	0.00000	0.9636	0.019	0	0

Num nodes	Geocoding (s)		Abstracting (s)		TSP (s)	
	Mean	Std dev.	Mean	Std dev.	Mean	Std dev.
100	0.0281	0.00030	3.7220	0.025	0	0
250	0.0713	0.00064	22.4684	0.028	0.003	0
500	0.1428	0.00108	91.0716	8.236	0.0124	0.000489898
1000	0.2907	0.00200	373.9748	52.926	0.0591	0.007175653

Appendix B

API documentation

B.1 Endpoints

B.1.1 Shortest path

URL:

`/shortest_path/<start_lat>/<start_lon>/<end_lat>/<end_lon>`

Example response:

```
{
  from_point: (52.00323, -1.4345),
  to_point: (52.02323, -1.4365),
  path: [(52.546734, -1.5436), ...],
  path_length: 15.43,
}
```

B.1.2 Route optimisation

URL:

`/route_optimisation/<points_str>`

Where `points_str` is a list of longitude and latitude pairs (`<lat>,<lon>`) separated by `|` characters, e.g. `52.432,-1.543|57.356,-3.425`

Example response:

```
{
  legs: [
    [(52.00323, -1.4345), ...],
    ...
  ],
}
```

Appendix C

Project plan

Implementing practical route planning algorithms for demand responsive bus services

Alex Nathanail
Supervised by Robin Hirsch

1 Aims and objectives

Aim:

> To investigate and implement algorithms for planning and live re-planning bus routes, in response to demand

Objectives:

1. Research shortest path algorithms, potential optimisations, and heuristics
2. Research solutions to the travelling salesman problem (TSP), their performance as the size of network scales, and metaheuristics that can be employed to produce real-world results
3. Research the vehicle allocation problem (VAP), how it is different to the TSP, and how it can be solved or approximated
4. Investigate the availability of, and interaction with, OpenStreeMap data to use as the basis for a graph representation of the UK road network
5. Develop a system of algorithms that can give approximate answers to the VAP on a reasonably-sized (UK road network) graph
6. Develop an API/web service that accepts real-world (longitude and latitude) VAP inputs as requests, and returns vehicle routes for the UK road network, using the algorithms laid out in 5, and the OpenStreetMap data referenced in 4
7. Evaluate the outputs produced by the VAP algorithms, against vehicle routes produced by human schedulers

2 Expected outcomes/deliverables

- A documented and functional piece of open-source software that serves a web process that can provide (usefully) approximate answers to the vehicle allocation problem on the UK road network
- A set of automated tests to verify the functionality of the software
- API documentation for potential users to interact with the web service
- Project report detailing processes and outcomes

3 Work plan

- 7th Oct - 4th Nov (4 weeks) Initial research and project plan (due Wed 9th Nov)
- 5th Nov - 25th Nov (3 weeks) Develop initial TSP solver, and start writing background section of report
- 26th Nov - 16th Dec (3 weeks) Apply TSP solver to OpenStreetMap data (with web server and API), and complete background section of report
- 17th Dec - 13th Jan (4 weeks) Extend TSP solver to VAP solver, and prepare interim report (due Wed 18th Jan)
- 14th Jan - 10th Feb (4 weeks) Evaluate VAP solver against humans, and make improvements
- 11th Feb - 25th Feb (2 weeks) Produce first draft of report for supervisor review
- 26th Feb - 17th Mar (2 weeks) Work on final report and produce video preview (due Wed 22nd Mar)
- 18th Mar - 7th Apr (3 weeks) Produce release candidate for final report, ready for supervisor review (due Wed 12th Apr)
- 7th Apr - 21st Apr (2 weeks) Produce final version of final report, ready for submission (due Wed 26th Apr)

4 Ethics review

I don't believe this project will require ethical approval, having considered the following factors:

- This project won't be collecting any data directly from individuals
- The data used (OpenStreetMap) is licensed under the Open Data Commons Open Database License, which only requires that I credit OpenStreetMap and its contributors in my use of their data
- None of the OpenStreetMap data contains information that could be used to identify an individual
- My work doesn't have any moral implications, its application is will strictly be useful for bus operators

Appendix D

Code listing