UoB$_{L}ogo$

A Comparison of Approaches to Combinatorial Optimisation for Multi-Day Route Planning

Jacob Luck
2338114

B.Sc. Computer Science with a Year in Industry
Project Supervisor - Leandro Minku

April 2025
9814 Words

Acknowledgements

I would like to thank openrouteservice and OpenStreetMap contributors, without whose efforts this project may not have been possible.

I would also like to thank my project supervisor, Leandro Minku, for his guidance and support throughout the development of this project and report.

Abstract

The Multi-Day Trip Planning problem presented in this report is a combinatorial optimisation problem that aims to plan a trip across multiple days, visiting a specified set of locations. The problem is optimised according to the amount of time spent travelling in the trip, and the balance of time spent across each day. An effective solution to this problem would allow a user to easily plan trips that efficiently visit the locations they wish to see.

In this report a number of different approaches to solving this problem are investigated, including algorithms shown to be effective in similar optimisation problems. These approaches are evaluated against each other to try and determine the most practical solutions to the problem. By the end of this report it is shown that a combination of genetic clustering algorithms with approximate and exact routing methods seem to be the most promising approaches.

Contents

1 Introduction

The Multi-Day Trip Planning problem investigated in this project can be simply described like so: given several locations, and a number of days over which the trip will take place, find a route that visits all locations while minimising the time spent travelling and distributing time spent evenly across each day. The motivation behind this problem is centred around tourism and travel, where a user may want to visit several points of interest in a city or region, and would like to find an efficient route for doing so across a number of days.

This project aims to investigate different potential approaches to solving this problem, and to evaluate the performance of these approaches against each other. The structure of this report will be as follows:

- In section 2 a mathematical formulation of the Multi-Day Trip Planning problem will be introduced.
- Section 3 will examine and analyse existing research into problems similar to that covered in this project, and the approaches taken to solve them.
- Section 4 discusses the algorithms that have been investigated and implemented in this project.
- Section 5 will present the methodology used to evaluate these algorithms, and present the results of the experiments ran.
- Finally, section 6 will conclude this report, reviewing the project and any future work that could be done.

All the code used in the project is available at: `https://git.cs.bham.ac.uk/projects-2024-25/jhl114`. Code snippets shown in this report are captioned detailing where in the gitlab repository their code can be found.

2 Problem Formulation

2.1 Problem Description

Given a positive integer: d, the number of days in the trip; a graph $G = (V, E)$, where V is set of locations including a designated starting point s, and E is a set of weighted edges linking every location to every other location; and a duration function $D(v), v \in V$, which assigns a visit duration to each node v, find a route that:

1. Visits all nodes $V \setminus s$ once.
2. Starts and finishes at s, having visited it d times, without ever visiting consecutively.
3. Minimises both the cumulative edge weights in the route and the variance in cumulative weight between each visit to s.

2.2 Inputs, Outputs and Design Variables

Inputs:

- d: The number of times s should be visited in a route. Contextually, d represents the number of days a tourist will spend on their trip. $d \in \mathbb{Z}, d > 0$
- $D(v)$: A duration functions that assigns a duration to each location. Contextually this represents the amount of time a tourist would like to spend visiting each point of location.
- $G = (V, E)$: A pair comprising:
    - V: A set of nodes representing locations the tourist would like to visit. $v \in V, v = (x, y, t)$, a triple comprising:
        - x: Longitude, indicating the location's geographic east-west position on the earth $x \in \mathbb{Q}, -180 \leq x \leq 180$.
        - y: Latitude, indicating the location's geographic north-south position on the earth $y \in \mathbb{Q}, -90 \leq y \leq 90$.

- E: A set of edges $e \in E$ that connects every node to every other node, bidirectionally. $e = (v_1, v_2, w)$, a triple comprising:
  - v_1: A location representing the origin of the edge.
    $v_1 \in V$.
  - v_2: A location representing the destination of the edge.
    $v_2 \in V$.
  - w: A weight indicating the sum of the time it takes to travel from v_1 to v_2 and the time the tourist wishes to spent at v_2.
    $w \in Z, w > 0$.
- s: Starting point that should be visited d times. Contextually, s represents where the tourist is staying and will return to at the end of each day.
  $s \in V$.

Outputs:

- R: A valid route satisfying all constraints, represented as an ordered sequence of locations.
  $R = [r_1, r_2, \ldots, r_n], r_i \in V$.

2.3 Objective Function

As previously mentioned in the Problem Description, the goal of this problem is to find a route that minimises the cumulative weight and the variance in route weight between each visit to s. To accomplish this the following cost function is applied to each route:

$$Cost(R) = W/d \times (1 + \sigma^2) \tag{1}$$

Where W is the sum of the weights of all edges traversed in the route and σ^2 is the variance of the sum of weights between each visit to s:

$$W = \sum_{i=0}^{n-1} w(r_i, r_{i+1}) + D(v_i), r_i \in R, v_i \in V \tag{2}$$

Where $w(r_i, r_{i+1})$ is the weight of the edge between r_i and r_{i+1}.

$$\sigma^2 = \frac{\sum_{i=0}^{d}(x_i - \mu)}{d}, x_i \in X \tag{3}$$

Where R is divided into sections between each visit to s and X is a list of the sum of weights within these sections.

μ is the mean cumulative weight of each x_i.

2.4 Constraints

A valid solution must satisfy the following constraints:

- The route must visit every node $v \in \{V \setminus s\}$ exactly once:

  $\forall_{v \in \{V \setminus s\}}, |\{i \in \{1, \ldots, n\} : r_i = v\}| = 1$

- The route must visit s exactly d times:

  $|\{i \in \{1, \ldots, n\} : r_i = s\}| = d$

- The route must not visit s consecutively:

  $\forall_{i \in \{1, \ldots, n-1\}}, r_i \neq r_{i+1}$

- The route must end at s:

  $r_n = s$

3 Literature Review

This literature review will explore existing research and approaches to other combinatorial optimisation problems. In this review existing research for the Travelling Salesman Problem, Multiple Travelling Salesman Problem, Vehicle Routing Problem and Tourist Trip Design Problem, will be covered. By gaining an understanding of the strengths and limitations of existing approaches to similar problems, better informed decisions can be made regarding which approaches to investigate, how they may be adapted to suit specific constraints of the Multi-Day Trip Planning Problem, and how these approaches might be implemented in practice. While the approaches taken to these problems may not be directly applicable to this study, it is likely their techniques can be adapted to better suit the problem investigated in this report.

The Travelling Salesman Problem (TSP) is perhaps one of the most studied combinatorial optimisation problems in computer science. The extensive research on the problem has acted as an 'engine of discovery for general-purpose techniques' offering large contributions across a wide range of mathematics(Applegate 2006, p. 40–41). The TSP can be described simply as: Given a set of locations and the cost of travel between them, find the shortest route that visits each location and returns to the start (Applegate 2006, p. 1). The problem relates directly to this goal of minimising travel time considered in this study, in fact, for inputs where $d = 1$, the Multi-Day Trip Planning problem becomes the TSP, with greater values of d introducing additional complexity.

The TSP is proven to be NP-hard (Cormen et al. 2022, p. 1096–1097), meaning that there are no known algorithms capable of solving the problem in polynomial time. Exact methods, such as brute force, branch-and-bound and dynamic programming are capable of finding optimal solutions, they just take an impractically long time to do so. Applegate (2006, p. 489–530) discusses how the best solvers of the time had solved problems of thousands of locations, but took many CPU years to do so. Even with

advances in computer processing, finding exact solutions appear too impractical in the context of this study. This leads towards the investigation of heuristic and approximation algorithms, which aim to find a near optimal solution within a reasonable time frame.

Laporte (1992) categorises TSP heuristics as either constructive or improvement heuristics, and claims that the best approaches to the problem are a combination of both. The paper presents a number of heuristic algorithms across these three approaches including: nearest neighbour routing, which iteratively adds the nearest location to a route; r-opt swapping, which takes an existing route, removes r connections and rebuilds the tour optimally; and the CCAO algorithm, which uses a convex hull and cheapest insertion procedure to build a route, before improving it via angle maximisation and r-opt swapping. While the various algorithms presented in the paper are worthy of note, there is a lack of significant discussion regarding the performance of these algorithms, and the comparisons they do make, lack supporting evidence.

A recent study by Goutham et al. (2023) investigated performance of the Convex Hull and Cheapest Insertion (CHCI) steps of the CCAO algorithm for travelling salesman problems in non-Euclidean space. The study used existing TSP datasets and modified them to add separators to the graph, or by using the $\mathcal{L}_1$ norm to calculate distances, however the convex hull was still formed according to the initial Euclidean graph. Despite this, CHCI was shown to outperform other heuristic and meta-heuristic approaches, such as Nearest Insertion or Ant Colony Optimisation, the majority of the time. While the non-Euclidean spaces considered differ to the input space used in this study, in which graphs are asymmetric and based on travel time between locations, there is potential to utilise Euclidean heuristics as a starting point for finding routes.

The multiple travelling salesman problem (mTSP) is an extension of the TSP, this time aiming to find a set of m routes that together visit every location once, with the goal of minimising the total cost of all routes (Bektas 2006, p. 209). This comes closer to the focus of this study, in which several routes are sought over multiple days, but without the optimisation objective of balancing routes. A literature review by Bektas (2006) provides

a comprehensive overview of procedures for solving the mTSP, of particular interest are Bellmore and Hong (1974)'s transformation of the mTSP into the TSP, and Tang et al. (2000)'s Modified Genetic Algorithm solution.

By transforming the mTSP into a TSP, one is able to use existing TSP algorithms to find mTSP routes. If similar transformations could be applied within this study, the possibility exists to take advantage of existing TSP research to implement algorithms that are already known to be effective. Bellmore and Hong (1974) propose a method of modifying a TSP graph to include artificial nodes and edges, with the nodes indicating the end of one salesman's route and the start of another and the edges including the cost of including additional salesmen in the problem.

Tang et al. (2000) modelled the schedulling of steel production as an mTSP before performing a similar transformation to Bellmore and Hong (1974)'s, to convert their mTSP into a TSP and solve the problem using their modified genetic algorithm (MGA). Typically at the time, genetic algorithms used a selection procedure in which both parents were chosen semi-randomly from the population, with each individuals fitness increasing their chance to be chosen. Tang et al. (2000)'s MGA altered this selection so one of the parents would always be the best individual, helping to maintain good solutions and reach convergence quicker. Not only did this MGA approach prove effective in computational testing, but when employed for scheduling in an iron and steel complex in Shanghai, showed an average improvement of 20% (Tang et al. 2000, p. 278–281).

Unfortunately, considering how the objective function of this study differs from most other combinatorial optimisation problems, it is likely that performing such a transformation could overly complicate the resultant problem. Nevertheless, the effectiveness of applying TSP solvers to other problems, related or not is certainly interesting. It would be worth considering scenarios that would allow the utilisation of pre-existing algorithms without too much modification.

First introduced by Dantzig and Ramser (1959), the Vehicle Routing Problem (VRP) is similar to the mTSP, considering multiple vehicles departing from, and returning to,

a central depot while visiting a number of locations. The VRP typically extends the problem to include a number of constraints related to the context of servicing customers, which does bring it to differ from the problem being studied.

One approach of particular interest is the generalised assignment heuristic considered by Fisher and Jaikumar (1981), which divided the input space geometrically, assigning each vehicle to one of these divisions. After these locations were assigned to a vehicle a route visiting them was found and optimised to reduce travel time. The generalised heuristic was shown to outperform the methods it was compared against (Fisher and Jaikumar 1981, p. 123).

Further investigation into this approach has been carried out by those such as Nallusamy et al. (2010), who combined the use of both k-means clustering and genetic algorithms to approximate solutions to the mVRP. K-Means was used to group nearby locations and have them visited by the same vehicle, then genetic algorithms were used to find the routes of each vehicle independently. These approaches seem particularly relevant to this study's problem, locations could be assigned to given days of the overall trip.

The Tourist Trip Design Problem (TTDP) takes potential points of interest (PoIs) and attempts to find the most interesting route based on a number of criteria. The TTDP shares the same motivation of tourist route planning as the problem investigated in this study and often considers similar parameters, such as travel time between PoIs and the desired time spent at each one (Vansteenwegen and Van Oudheusden 2007). A large difference though, is that it does not require every location to be visited, prioritising the visitation of certain PoIs according to user preference.

Vansteenwegen and Van Oudheusden (2007) discusses modelling multiple day trip planning through the Team Orienteering Problem (TOP). The TOP aims to find a set of routes that visit a number of locations, with the goal of maximising the total value of the locations visited within some time limit. Vansteenwegen, Souffriau, and Van Oudheusden (2011)'s survey provides a summary of the best TOP algorithms

according to benchmarks on their speed and solution quality. Among the best performing algorithms were meta-heuristics such as Genetic Based Tabu Search, Ant Colony Optimisation and Greedy Randomised Adaptive Search Procedure. The paper also discusses how different local search algorithms apply different 'local search moves' to improve the quality of a solution. While a wide range of moves exist, all effective search algorithms used greedy insertion as part of their solution, perhaps indicating its effectiveness.

The literature reviewed in this section provides valuable insights into how the problem may be tackled. With the expansive pre-existing research into the TSP, mTSP, VRP and TTDP, existing approaches and how they may be applicable for this project, may be evaluated. A diverse range of approximate solutions are considered in the reviewed literature, from simple construction heuristics such as nearest neighbour routing or greedy insertion, to complex meta-heuristics such as genetic algorithms or ant colony optimisation.

Among the most promising approaches are those that combine both clustering and routing algorithms to split up locations and find multiple routes from these divisions. It is unfortunate that unlike this study's problem, the literature covering these techniques are only concerned with minimising overall route length, hopefully these methods will still prove useful despite these differences.

Much of the research reviewed does not consider the optimisation of multiple objectives, with none of it sharing the goals of this study to minimise both travel and route variance, this would suggest that more research is needed in this area.

Considering the lack of investigation into solving problems with the constraints and optimisation objectives applied in this study, this project will compare a swathe of approaches and algorithms, aiming to identify which are most appropriate for the problem at hand.

The majority of approaches will be those that split routing and assignment into two

steps, which will be performed in one of two ways:

- Assigning locations to a given day in the trip and then finding routes for each group, as previously described by Fisher and Jaikumar (1981) and Nallusamy et al. (2010).
- Finding a route between all locations and then using insertion algorithms, such as those compared by Vansteenwegen, Souffriau, and Van Oudheusden (2011), to break the route up into a multiple day trip.

Multiple routing and clustering algorithms will be implemented for use in these approaches, as will be described in the next section of this report.

4 Algorithms Investigated and Their Implementation

Based on what was learned during the literature review, the decision was made to investigate three distinct approaches to the Multi Day Route Planning problem:

- Cluster then Route, where a clustering algorithm is used to group together locations in the input, before a routing algorithm is applied to each group.
- Route then Insert, where a route is found that visits all locations, before the end of each day is inserted somewhere appropriate in the route.
- Trip Generation, which aims to find a whole multi day trip in one step.

To form these approaches a number of clustering, routing and trip generation algorithms were implemented, with each of these listed below:

Clustering

- K-Means Clustering,
- Genetic Clustering,
- Centroid-Based Genetic Clustering.

Routing

- Brute Force Routing,
- Greedy Routing,
- Greedy Insertion Routing[1],
- Convex Hull Routing,
- Genetic Routing.

[1]Greedy Insertion Routing will also be extended to allow its usage for splitting routes into multiple days.

Trip Generation

- Brute Force Trip Generation,
- Genetic Trip Generation,
- Greedy Insertion Trip Generation.

After describing the pre-requisites to implementing these algorithms, this section of the report will include a description for all the algorithms listed above, as well as explain how each was implemented within python.

4.1 Implementation Pre-Requisites

There are two things needed before beginning to implement the chosen algorithms, a method for generating inputs on which to use these algorithms, and an implementation of an objective function to enable the evaluation of routes.

4.1.1 Input Generation

For generating inputs Openrouteservice's Points of Interest (PoI) and Distance Matrix API services will be used. With the PoI service, locations such as a city centres can be input, and a specified number of locations of interest around the area will be returned. The PoI service also allows the limitation of locations returned to certain categories and the identification of specific that a user would likely be interested in visiting, for example points labelled 'tourism', 'historic' or 'arts and culture'.

After these locations have been identified, the Distance Matrix can be used to create a graph linking each location to one another with a weight associated with travelling between them. The API call to this service can be configured such that, instead of distance, the weights in the matrix represent the time taken to travel between locations via public transport. Unfortunately, the Distance Matrix service limits the number of locations in a call to 25, severely limiting the size of input that can be test. Other alternative API's were considered, such as the Google Maps API or Mapbox, but all the alternatives were either too expensive or equally, if not more, restrictive.

With these, a list of durations associated with the found locations can be generated, as well as a number of days for the trip, to finally have the input ready.

4.1.2 Objective Function

As described in the Problem Formulation, the Multi-Day Planning Problem's objective function aims to calculate a cost for a given trip. This function is needed such that different approaches and algorithms can be compared based on the quality of their results. Figure 1 shows how this was implemented in this project.

$$\boxed{\text{Algorithm.evaluate}_{r}oute}$$

Figure 1: Code from Algorithm.evaluate_route in algorithms\algorithm.py

4.2 Clustering

Clustering as a concept can be described as the 'organization of a collection of patterns (usually represented as a vector of measurements, or a point in a multidimensional space) into clusters based on similarity' (Jain, Murty, and Flynn 1999, p. 265). For this problem, clustering will be used to group locations together to form an itinerary for each day of the trip. These clusters, or days, will then be used as an input for some routing algorithm, which will try and find an optimal route for each day, which can then be combined to form a complete trip. Here, the goal of clustering algorithms is to find a set of clusters that, when combined with some routing algorithm, will produce a trip with minimal cost. The code in figure 2 shows how a set of clusters can be used alongside graph and duration inputs to find a trip.

$$\boxed{\text{Clustering.find}_{r}oute_{f}rom_{c}lusters}$$

Figure 2: Code from Clustering.find_route_from_clusters in algorithms\clustering.py

The clustering algorithms implemented in this project are: K-Means, Genetic Clustering and Genetic Centroid-based Clustering.

4.2.1 K-Means Clustering

K-Means is an iterative clustering algorithm that defines its clusters using a set of centroids (means) which are given a location in the input space, a given location is assigned to the cluster of the 'closest' centroid. The algorithm starts by initialising random centroids and iteratively improving the clustering from there, continuing until the algorithm converges (on a local optimum) or an iteration limit is reached. K-Means runs in linear time with a time complexity of $O(mnki)$, where m is the number of locations, n is the dimensionality of the input, k is the number of clusters, and i is the number of iterations (Hartigan and Wong 1979, p. 102). The inputs will always contain only two dimensions, and a maximum number of iterations will be set, making both n and i constants, allowing the simplification of the time complexity to $O(mk)$.

In this implementation of K-Means centroids will be initialised by randomly selecting unique locations from the input, and placing the initial centroids at their coordinates. A different approach was considered, which involved initialising the centroids with random coordinates in a similar area to the input, however this had the potential to create clusters with zero locations assigned to them resulting in invalid trips. By starting with locations from the input, there is certainty that all clusters have at least one location assigned to them. We will be the coordinates of inputted locations will be used to calculate the Euclidean distances between locations and centroids, these locations are then assigned to the cluster of the closest centroid. Figure 3 shows how this was accomplished.

Clustering.$_a ssign_n odes_t o_c entroid$

Figure 3: Code from Clustering._assign_nodes_to_centroid in algorithms\clustering.py

After assignment, the centroids are recalculated such that their coordinates are the average of all locations assigned to their cluster. This is done by iterating through each cluster and calculating the mean of all locations assigned to it. The implementation of this is shown in figure 4.

KMeans._compute_means

Figure 4: Code from KMeans._compute_means in algorithms\clustering.py

These steps of cluster assignment and centroid recalculation are repeated until either a maximum allowed number of iterations is reached, or until the algorithm converges on an optimum solution. The convergence criterion is that the centroids stop changing between iterations, i.e., the centroids are the same as the previous iteration. The python implementation of this is shown in figure 5.

KMeans.find_clusters

Figure 5: Code from KMeans.find_clusters in algorithms\clustering.py

Figure 6 shows an example of the iterations of a K-Means algorithm run on an input with 12 points of interest around London to be visited over 3 days.

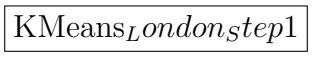

Figure 6a: K-Means example Step 1, Initial centroid positions and cluster assignments.

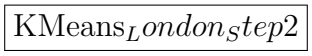

Figure 6b: K-Means example Step2, Centroids have been updated, locations are reassigned to their closest centroid.

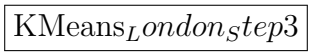

Figure 6c: K-Means example Step3, Centroids have updated and no locations have changed cluster, a solution has been found.

It is worth noting that K-Means only forms these clusters based on Euclidean distances, grouping together locations that are close geographically. However, as formally described in the Objective Function section of the Problem Formulation, a good trip will minimise both the route length of the trip and the variance between time spent each day. K-Means does not aim to optimise for the variance between days, it fails to consider the time spent at each location. Furthermore, while each cluster might be optimised for distance, how

close two locations are may not reflect the travel time between them. While K-Means does not intentionally optimise for variance between days or consider travel time between locations, it was still chosen for this project out of curiosity as to how effective a heuristic it might provide. Hopefully it will offer a simple and computationally efficient baseline for comparison with more complex algorithms.

4.2.2 Genetic Clustering

Genetic clustering applies genetic algorithms to attempt and find the best assignment of locations to clusters. Genetic algorithms are a type of evolutionary algorithm that aims to mimic biological evolution to find an optimal solution. They involve creating a population of potential solutions (individuals) and iteratively improving the population through selection (keeping the best individuals in the population, akin to natural selection), crossover (combining individuals to create offspring, akin to sexual reproduction), and mutation (randomly altering the genomes of individuals in the population, akin to biological mutation).

To perform selection, and find the best solutions in a population, each individual requires some fitness assigned to it. To calculate this fitness cluster assignments will be combined with a chosen routing algorithm, and a cost function applied to the route found. This cost will be used to rank a population, assisting in the identification of clusters that can produce a good trip.

The performance of Genetic algorithms is highly dependent on its hyperparameters: mutation rate, determining how common mutation is; crossover rate, determining how often offspring are created via crossover, as opposed to new additions to the population; population size, determining how many individuals there are per generation; number of generations, determining how many generations will be evolved to reach a solution; crossover method used, determining how crossover is performed to create offspring; and in this case, the routing algorithm used, which may impact how clusters are used to form routes. These hyperparameters impact both the runtime of the algorithm and the exploration of the search space, indirectly impacting the quality of the solution. For

each generation, Genetic Clustering has to route and evaluate each individual in the population, yielding a time complexity of $O(gprn)$, with g being the number of generations, p being the population size, r being the time complexity of the chosen routing algorithm and n being the number of locations and days.

For this Genetic Clustering implementation, an individual is represented by a genome, which will provide a mapping that assigns each location to a cluster. Figure 7 shows an example of this.

Genome: [0, 0, 0, 1, 1, 1, 2, 2, 2]

Barcelona$_G$*enome*$_E$*xample*

Figure 7: Example of how an individual's genome corresponds to cluster assignments.

These genomes are the target for performing crossover and mutations. We begin this evolution process by randomly generating an initial population of individuals. From there, following steps are repeated until reach a maximum number of generations is reached:

1. Evaluation of the fitness of each individual in the population.
2. Selection of the best individuals from the population, these will be carried over into the next generation, as well as be used to create offspring.
3. Creation of the new population using crossover and mutation.

As previously discussed, the fitness of each individual is evaluated by applying a routing algorithms to the clusters defined by the genome, and then applying the cost function to the resulting route. Figure 8 shows this route finding and evaluation.

GeneticClustering.$_e$*valuate*$_p$*opulation*

Figure 8: Code from GeneticClustering._evaluate_population in algorithms\clustering.py

The methods called in figure 8 are those previously shown in figure 2 (find_route_from_clusters) and figure 1 (evaluate_route). Tang et al. (2000)'s genetic

algorithm featured a selection process involving choosing the best individual for one parent, with the other parent being selected via a semi-random process with bias based on the performance of each individual. For the genetic algorithms implemented in this project, the two best individuals of a population will be chosen as parents, without any random selection. Figure 9 shows this simple selection process in python.

GeneticClustering.find$_c lusters.select_p arents$

Figure 9: Code from GeneticClustering.find_clusters in algorithms\clustering.py

Excluding the two parents, who will be copied over, the next generation will be created via crossover and mutation, or through random generation. By taking the two best individuals as parents, good solutions are likely to be maintained, but will trade off genetic diversity, perhaps leading the algorithm to get stuck in local optima and not sufficiently exploring the search space. To counter-act this, and boost the genetic diversity of each population some randomly generated individuals in an effort to increase genetic diversity and exploration of the search space. Figure 10 shows how this decision was implemented. For each individual a random number is generated between 0 and 1, if this number is lower than the crossover rate the individual is created via crossover, otherwise it will be randomly generated.

GeneticClustering.find$_c lusters.crossover$

Figure 10: Code from GeneticClustering.find_clusters in algorithms\clustering.py

For this implementation of crossover, a uniform crossover will be performed, using crossover masks. A crossover mask is a bit array the same length as the genome, with the parity of each bit indicating which parent to choose from for the corresponding bit in the created genome. In a uniform crossover mask, each bit has an 50% chance of being a 0 or 1, meaning that each bit in the offspring genome has an equal chance of being from either parent (Syswerda et al. 1989). If the two parents, being the best individuals in a population, agree on a bit it then it would appear likely to be a good choice. With this implementation of crossover, the offspring will always copy over the bits that parents agree on. Figure 11 shows an example of how a crossover mask can be

used to create offspring from two parents, the example uses a hypothetical input including 6 locations and 3 clusters.

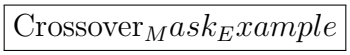

Figure 11: Example of using a crossover mask to create offspring from two parents.

There is however, one slight issue with directly applying this method. In the example shown in figure 11, parent 1's 0th cluster only includes the first location, similarly, parent 2's 2nd cluster also only includes the first location. Both parents are forming a cluster using these locations, however due to them being labelled differently, the offspring produced did not continue this clustering. To solve this problem a genome's clusters will be relabelled in order of their appearance in the genome, ensuring consistency between parents. Figure 12 shows an example of this applied to the parents in figure 11, and figure 13 shows how this is accomplished in python.

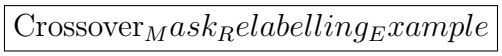

Figure 12: Example of relabelling a parent's clusters and the resultant offspring

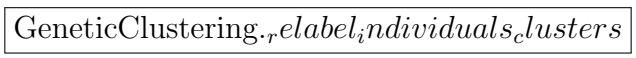

Figure 13: Code from GeneticClustering._relabel_individuals_clusters in algorithms\clusterin.py

After this relabelling, crossover can be performed without issue. Figure 14 shows the python implementation of this.

Figure 14: Code from GeneticClustering._crossover in algorithms\clustering.py

After crossover is complete, created offspring are randomly mutated in the hopes of increasing genetic diversity and escaping local optima. The genome is mutated by iterating through each gene and randomly deciding if it will mutate or not, the likeliness of mutation is decided by the mutation rate. If a gene is chosen to mutate, it will assign itself to a random cluster. Figure 15 shows how this is implemented in python.

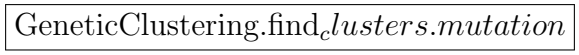

Figure 15: Code from GeneticClustering.find_clusters in algorithms\clustering.py

After the new population has been created, the process is restarted for the next generation, repeating these steps until a maximum number of generations is reached. By time evolution is complete the population will hopefully have converged on a good solution. Figure 16 shows an example of using Genetic Clustering the same input as for the K-Means example in figure 6. In the example, Genetic Clustering is used alongside Greedy Routing (to be covered in section 4.3.2) to form a complete trip, the best routes found within their respective generations are shown. Genetic Clustering was run for 30 generations with a population size of 12, a crossover rate of 0.9 was used alongside a mutation rate of 0.1. Figure 17 shows a line graph of the best evaluation found for each generation.

Figure 16a: Genetic Clustering example, best route found in Generation 1.

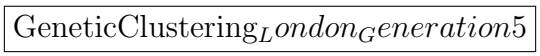

Figure 16b: Genetic Clustering example, best route found in Generation 5.

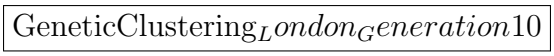

Figure 16c: Genetic Clustering example, best route found in Generation 10.

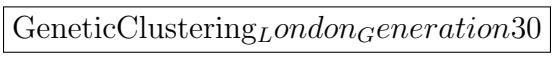

Figure 16d: Genetic Clustering example, best route found in Generation 30, evolution is now complete.

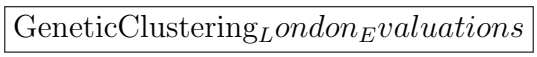

Figure 17: Line graph showing the best evaluation for each generation of clusters evolution process.

Unlike K-Means, which finds optimal routes according to geographic base, Genetic Clustering is capable of optimising across both optimisation objectives, finding short

routes and minimising daily variance. By evaluating each set of clusters according to the objective function, some assignment of locations to each day can be found that will produce a good overall trip. It is likely Genetic Clustering will run slower than traditional heuristic approaches such as K-Means, it will be of interest whether the genetic approach can produce significantly better results as a trade off.

Genetic algorithms are also highly versatile and by modifying aspects of the algorithm can find solutions across a number of problems. This adaptability allows the implementation of genetic algorithms for route finding and trip generation approaches, as well as performing a genetic centroid-based clustering similar to K-Means. By comparing the performance of genetic algorithms against their counterparts across different approaches where they perform best can be investigated.

4.2.3 Centroid-Based Genetic Clustering

Inspired by K-Means, Centroid-Based Genetic Clustering uses a genetic algorithm to find the best set of centroids to cluster the data. The same process of evolution is used, as described previously, except this time the genome will specify the centroids to use for clustering. With a different genome structure, the crossover and mutation methods will need be reconsidered. Centroid Based Genetic Clustering keeps the same time complexity as Genetic Clustering, $O(gprn)$, with g being the number of generations, p being the population size, r being the time complexity of the chosen routing algorithm and n being number of locations and days.

This genetic algorithms genome will be a list of coordinate pairs, representing the latitude and longitude of each centroid. Figure 18 shows an example of how locations are clustered using a centroid-based genome.

$$\text{Genome: } \begin{bmatrix} 0.00 & 0.01 & -0.01 & 0.00 & 0.00 \\ 51.477 & 51.477 & 51.477 & 51.482 & 51.472 \end{bmatrix}$$

$$\boxed{\text{GeneticCentroids}_{Greenwich}Example}$$

Figure 18: Example of how an individual's genome corresponds to cluster centroids.

Once again, the population will be iteratively evolved through the process of selection, crossover and mutation. The selection procedure here is largely the same in that each individual is evaluated by generating a trip with the help of its genome and calculating the associated cost of these trips. The only difference is that, because this genome is no longer a direct assignment of clusters, each location will first need to be assigned to its nearest centroid. This is shown in figure 19.

GeneticCentroids.$_evaluate_population$

Figure 19: From GeneticCentroidClustering._evaluate_population in algorithms\clustering.py

The '_assign_nodes_to_centroid' method shown in figure 19 is the same method used for K-Means, shown in figure 3. With the population evaluated the best individuals can be selected for use in creating the next generation. New generations are created via the same stages of copying over the best individuals, generating some new individual's randomly and creating the rest via crossover and mutation.

For this implementation of crossover, instead of selecting random genes from each parent, the values of both parents will be taken and merged together. In practice, this means taking the coordinates of each centroid from both parents and finding a point between them to generate a new centroid. How close this new centroid is to each parent is determined by generating a random weight, indicating how much influence each parent has on the offspring.

Similarly to the previous implementation of crossover, the centroids will need to be reordered to ensure that both parents are consistent with each other. This time, because the genome isn't formed of discrete values that can be renamed, instead an approach of finding the most similar centroids from each parent and merging them is taken. This is performed by calculating the distances between each of parent 1's centroids to each of parent 2's, and reordering the genome to place the closest centroids together. Figure 20 shows this implementation of crossover, including how parents are reordered and merged together to produce offspring.

GeneticCentroids._crossover

Figure 20: Code from GeneticCentroidClustering._crossover in algorithms\clustering.py

For mutation, each centroid of every individual will be considered, and a random decision taken as to whether the centroid will mutate or not. If a centroid is chosen to mutate, a new location will be within the bound of the input coordinates will be selected. The minimum and maximum latitude and longitude of the input are used to form a region within which new centroids can be generated, ensuring that the mutation is still relevant to the input. Figure 21 shows this process of mutation.

GeneticCentroids.find_clusters.mutation

Figure 21: Code from GeneticCentroidClustering.find_clusters in algorithms\clustering.py

Once again, this process of selection, crossover and mutation is repeated until a maximum number of generations is reached. By the end of this process, a population of individuals that have been evolved to find good centroids for clustering the input will have been identified. Figure 22 shows an example of genetic centroid-based clustering using the same input as for previous clustering examples. This example of genetic centroid-based clustering used the same hyperparameters as the previous genetic clustering example, evolving 30 generations with a population size of 12, a crossover rate of 0.9, a mutation rate of 0.1 and once again using Greedy Routing alongside found clusters to produce trips. Again, a line graph of the best evaluation found for each generation of the process is shown in figure 23.

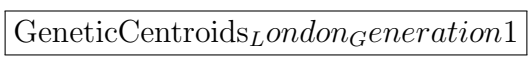

Figure 22a: Genetic Centroid Clustering example, best route found in Generation 1.

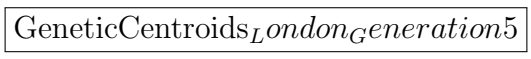

Figure 22b: Genetic Centroids Clustering example, best route found in Generation 5.

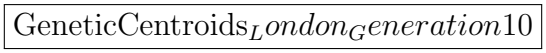

Figure 22c: Genetic Centroids Clustering example, best route found in Generation 10.

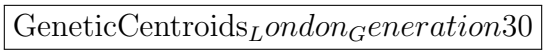

Figure 22d: Genetic Centroids Clustering example, best route found in Generation 30, evolution is now complete.

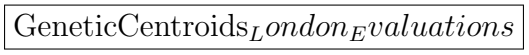

Figure 23: Line graph showing the best evaluation for each generation of centroids evolution process.

Genetic Centroid Clustering was included in this project to explore how a centroid-based approach, inspired by K-Means, would perform when combined with a genetic algorithm. Of particular interest is seeing how this approach compares to both standard K-Means as well as the other genetic clustering approach. Hopefully, by optimising chosen centroids according to the objective function, and exploring a wider range of potential centroids, the quality of solution will be greater than that of regular K-Means.

4.3 Routing

The routing algorithms implemented in this project are TSP solvers, aiming to find a route with minimal travel time that visits all given locations. There are two ways in which routing will be used to form a trip either by using a clustering algorithm and then finding a route for each cluster (as previously described in section 4.2), or by finding a route that visits every location, before splitting the route into multiple days. This approach of splitting a route into multiple days will be covered Greedy Insertion is introduced. algorithm which, on top of finding routes of its own, can also be used to build upon existing routes. The routing algorithms implemented in this project are: Brute Force Routing, Greedy Routing, Greedy Insertion, Convex Hull-Based Routing and Genetic Routing.

4.3.1 Brute Force Routing

Brute Force Routing is an exhaustive algorithm that tries every possible route to find one with the least cost. We find every route by generating all permutations of the locations to be visited, with the order of the locations in each permutation representing the order in which they will be visited. Each permutation is evaluated based on the amount of time it takes to travel the route, with the shortest route being returned. By checking every route Brute Force is guaranteed to find the best possible route, but due to checking every permutation has a time complexity of $O(n!)$, where n is the number of locations.
It is worth noting that in this implementation, since all routes must return to the starting point, the final location of a route is fixed. Therefore, only $n-1$ locations need considering, resulting in $(n-1)!$ possible permutations. To check every route, we'll be creating a mapping between every integer $\{0, 1, \ldots, (n-1!)-1\}$ to every permutation. For a given integer k, and a list of possible locations $L : \{0, 1, \ldots, n-1\}$ k's corresponding permutation can be obtained through the following steps:

1. Divide k by $(n-2)$ to find $x_1 : \{0 \leq x < n-1\}$, and the remainder r_1.

2. x_1 is used to select a location for L, which will be the first location in the permutation. L_{x_1} is removed from L.

3. Divide r_1 by $(n-3)!$ to find $x_2 : \{0 \leq x < n-2\}$, and the remainder r_2.

4. x_2 is used to select another location from L, which will be the second location in the permutation. Again, L_{x_2} is removed from L.

5. Repeat this process until all locations are assigned a position.

To help explain this, an example is shown in figure 24, generating a permutation P of the set $L = \{a, b, c, d\}$ that maps to $k = 17$.

Step 1: Divide $k = 17$ by $(n-1)! = 3! = 6$
$17 \div 6 = 2$ with remainder 5. Thus $x_1 = 2, r_1 = 5$
$L_{x_1} = L_2 = c$ is selected as the first element of the permutation.
Updated sets: $L = \{a, b, d\}, P = \{c\}$

Step 2: Divide $r_1 = 5$ by $(n-2)! = 2! = 2$
$5 \div 2 = 2$ with remainder 1. Thus $x_2 = 2, r_2 = 1$
$L_{x_2} = L_2 = d$ is selected as the second element of the permutation.
Updated sets: $L = \{a, b\}, P = \{c, d\}$

Step 3: Divide $r_2 = 1$ by $(n-3)! = 1! = 1$
$1 \div 1 = 1$ with remainder 0. Thus, $x_3 = 1, r_3 = 0$
$L_{x_3} = L_1 = b$ is selected as the third element of the permutation.
Updated sets: $L = \{a\}, P = \{c, d, b\}$

Step 4: There's only one element left, so a becomes the last element.
Updated sets: $P = \{c, d, b, a\}$

Figure 24: Example of generating a permutation of a set using an integer mapping

This method ensures that each integer maps to a unique permutation, allowing the checking of every possible route. The python version of this is shown in figure 25.

$\boxed{\text{Algorithm.generate}_{r}oute}$

Figure 25: Code from Algorithm.generate_route in algorithms\algorithm.py

With this method of generating routes in place, all that is required is to iterate through them all, evaluate them, and keep track of the best found. The implementation of this is shown in figure 26.

$\boxed{\text{Routing.brute}_{f}orce}$

Figure 26: Code from Routing.brute_force in algorithms\routing.py

Figure 27 shows an example of the route find by performing Brute Force on an input of 10 locations around Birmingham. The location marked in green represents the starting point, and the blue marker indicates the first location visited (showing the direction of the route).

$\boxed{\text{Bruteforce}_{B}irmingham_{E}xample}$

Figure 27: Example of a route found visiting 10 locations around Birmingham using Brute Force routing

Being able to find a perfect route is certainly useful, but the computational cost of Brute Force becomes impractical as the input size grows. In calculating the 10 location route shown in figure 27, 362,880 possible routes were considered, taking around 15 seconds. If the maximum input size of 25 locations were to be used, over 620 sextillion routes would be considered —which, assuming the same rate of routes evaluated per second, would take over 800 billion years.

While Brute Force is impractical for routing between a large number of locations, is it still useful for smaller inputs. This may prove useful when combined with clustering algorithms, providing clusters are not too large, Brute Force may be able to effectively find intra-cluster routes. Furthermore, for small input sizes, Brute Force can be useful as a benchmark against which to compare other routing algorithms.

4.3.2 Greedy Routing

Greedy Routing is a heuristic algorithm that builds a route iteratively, always choosing the next location according to the shortest available path. This is done by starting at the first location and then repeatedly selecting the next closest location until all locations have been visited, then returning home. At every location, the distance to every other location is checked to find the closest, giving this greedy algorithm a time complexity of $O(n^2)$.

Greedy Routing was by far the simplest algorithm to implement, and does not require much explanation. This project's implementation will simply make a copy of the graph input, then iteratively find the closest location to add to the route. When a location is visited, the edges leading to it are given an infinite cost to ensure that it is not visited again. The python implementation of Greedy Routing is shown in figure 28.

Routing.greedy$_r$*outing*

Figure 28: Code from Routing.greedy_routing in algorithms\routing.py

Figure 29 shows an example of a route find by Greedy Routing, using the same

Birmingham input as that shown for Brute Force in figure 27.

GreedyRouting$_{B}irmingham_{E}xample$

Figure 29: Example of a route found visiting 10 locations around Birmingham using Greedy Routing

As is perhaps evident from its example, Greedy Routing isn't always a great fit, with the effectiveness of the algorithm being highly dependent on the input. It may just happen that for some inputs the greedy algorithm will find a good route, but as input size grows and routes become more complex, this becomes increasingly unlikely.

Where Greedy Routing does excel, and the reason for its inclusion in this project, is in its speed. The algorithm found a route for the 10 location Birmingham input in less than a thousandth of a second, over 500,000 times faster than Brute Force. Some fast routing algorithms are needed in order to use alongside the genetic clustering methods which, for evaluation, require a route to be found for every individual in a population.

4.3.3 Greedy Insertion

Similar to Greedy Routing, Greedy Insertion is another heuristic algorithm that iteratively builds routes. This time, instead of selecting the next location to add into the route, the set of unvisited locations will be iterated through, finding the best location to insert each one into the route. To find the best insertion point, cost of the route produced by each possible insertion point will be calculated position, the cheapest insertion point will be chosen. Compared to Greedy Routing, this adds an extra $O(n)$ step to each iteration, increasing the time complexity to $O(n^3)$.

While Greedy Insertion is a more complex algorithm than Greedy Routing, it is still relatively simple to implement. All that's required is to iterate through the list of locations, evaluate the route created by inserting them at each position, and keeping the best route each time. The code used for Greedy Insertion can be seen in figure 30

Routing.greedy$_{i}nsertion$

Figure 30: Code from Routing.greedy_insertion in algorithms\routing.py

An example of a route found by applying Greedy Insertion to the 10 location Birmingham input is shown in figure 31.

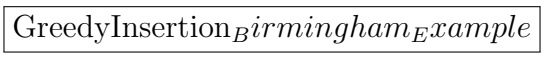

Figure 31: Example of a route found visiting 10 locations around Birmingham using Greedy Insertion

For this input, Greedy Insertion was able to find the optimal solution, as indicated by it returning the same route as Brute Force. Finding said route took Greedy Insertion a few thousandths of a second, which while several times slower than greedy routing, is still a huge improvement on Brute Force. Hopefully Greedy Insertion can provide better results than Greedy Routing, without too much of a cost in terms of time improvement.

Finding routes isn't where insertion's usefulness ends though, it can also be used to build upon existing routes, or to split a route up into multiple days. Perhaps a route has already been found, but there is a want to add more locations to it, Greedy Insertion can be performed to find the best point for each location to insert into the pre-existing route. Furthermore, if for Greedy Insertion's new locations input, a set consisting of a route's starting point d number of times is provided, d number of days can be added to form a complete trip. This allows the creation of multi-day trips out of the routes found via the implemented routing algorithms, by taking their routes and splitting them where Greedy Insertion believes is best.

4.3.4 Convex Hull Routing

Given a set of points, its convex hull is the smallest border one can draw between them, such that all points are contained inside. For two-dimensional problems, it can be described as the shape a rubber band would take if stretched around the points. Figure 32 shows a drawing of a convex hull surrounding a set of points

Figure 32: Drawing of a convex hull around a set of points

If all locations happen to fall on the border of the convex hull, then the convex hull

is itself an optimal route; although, this is rather unlikely to happen. Regardless, the convex hull does still indicate that an optimal route will visit the border cities in the order they appear on the hull (Applegate 2006, p. 46). With this in mind, the convex hull can be used as a starting point, before Greedy Insertion finds a complete route. The time complexity of finding a convex hull is $O(nh)$(Cormen et al. 2022, p. 1029), where n is the number of locations, and h is the number of points on the hull. In the worst case, where all the points are on the hull, this becomes $O(n^2)$.

To find a convex hull a Gift Wrapping, or Jarvis March, algorithm will be used. This algorithm works by finding the leftmost point in the set and then progressively finding the most counter-clockwise point to add to the hull. To discover the most counter-clockwise point a candidate for the next point on the hull is arbitrarily selected. Then, for every other location not yet in the hull, the cross product is calculated: $\vec{HC} \times \vec{HI}$, where H is the most recently added hull point, C is the candidate point and I is the location being evaluated. If the cross product is positive, then I is more counter-clockwise than C, and C can be replaced with I as the new candidate to be added to the hull. When all possible values for I produce a negative cross product, the candidate point is added to the hull and a new arbitrarily selected candidate is chosen. This process is repeated until the next hull candidate is the original starting point of the hull.

Figures 33 and 34 show how this was accomplished in python, and the hull formed around the Birmingham example, respectively.

$\boxed{\text{Routing.gift}_w rapping}$

Figure 33: Code from Routing.gift_wrapping in algorithms\routing.py

$\boxed{\text{ConvexHull}_B irmingham_H ull}$

Figure 34: Example of the convex hull found around 10 locations in Birmingham

Once the hull is found, it can be used as a starting point for Greedy Insertion. This hull will be used as a starting point for a route, and all the interior points as locations for insertion. Figure 35 shows the route found by applying Greedy Insertion to the hull found around the Birmingham example.

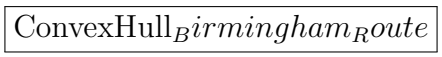

Figure 35: Example of a route found using Greedy Insertion and the convex hull of 10 locations in Birmingham

Convex Hull Routing was included in this project to investigate how well it would work as a starting point for greedy insertion. It will be interesting to see whether starting from a convex hull could improve the results and/or the speed of Greedy Insertion, as opposed to running Greedy Insertion on the whole set of locations.

4.3.5 Genetic Routing

As well as clustering, genetic algorithms can also be used for routing. We will evolve a population of potential routes similarly to before, this time with each genome representing a potential route, and with new crossover and mutation techniques to account for this. Without the need to use another routing algorithm in order to evaluate its population, genetic routing has a slightly faster time complexity of $O(gpn)$, with g being the number of generations, p being the population size and n being number of locations in the route.

By finding a route directly, selection is simpler, no longer requiring the use a routing algorithm with each genome to find a route capable of evaluation. This time, a genome will represent a permutation of the locations to be visited, with the order of the locations in the genome representing the order in which they will be visited, similar to the Brute Force algorithm. We will be selecting the best individuals in a population by evaluating the route formed in their genome. Via crossover and mutation, the order in which locations are visited will be shuffled to try and find a better route. The crossover algorithm will be a simple single-point crossover, a random point in the genome will be chosen and all locations up to that point will be taken from one parent. Then, all the remaining locations will be added in the order they appear in the second parent's genome. A simple example of single-point crossover is shown in figure 36, using two hypothetical genomes with 6 locations. The python implementation of this crossover

method is shown in figure 37.

Point$_C rossover_E xample$

Figure 36: Example of using single point crossover to create offspring from two parents.

GeneticRouting.$_c rossover$

Figure 37: Code from GeneticRouting._crossover in algorithms\routing.py

For mutation, a number of individuals will be randomly chosen to mutate inline with the algorithms mutation probability. Chosen individuals will have two locations in their route randomly chosen and swapped. This implementation is shown in figure 38, the final location isn't considered for swapping, as all routes end at the starting point.

GeneticRouting.mutation

Figure 38: Code from GeneticRouting.find_route in algorithms\routing.py

By repeatedly performing selection, crossover and mutation the population will hopefully converge on a better route. Figure 39 shows an example of genetic routing performed on the 10 location Birmingham input. For this example, 30 generations were evolved with a population size of 12, a crossover rate of 0.9 and a mutation rate of 0.4. A line graph of the best evaluation found for each generation is shown in figure 40.

GeneticRouting$_B irmingham_G eneration1$

Figure 39a: Genetic Routing example, best route found in Generation 1.

GeneticRouting$_B irmingham_G eneration5$

Figure 39b: Genetic Routing example, best route found in Generation 5.

GeneticRouting$_B irmingham_G eneration10$

Figure 39c: Genetic routing example, best route found in Generation 10.

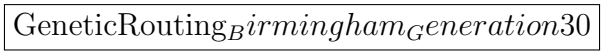

Figure 39d: Genetic Routing example, best route found in Generation 30, evolution is now complete.

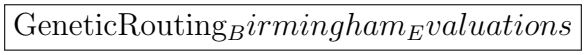

Figure 40: Line graph showing the best evaluation for each generation of routing evolution process.

Having implemented a number of heuristic approaches to finding routes, as well as the exact Brute Force method, genetic routing was included to see how a more meta-heuristic algorithm might perform. Likely to take longer than the other approximate routing methods, we'll have to see if genetic routing can make up for this time with the quality of its results. Hopefully genetic routing can strike a balance between fast approximate methods and the slow, exact Brute Force method.

Additionally, the suitability of genetic algorithms for routing purposes can be evaluated and compared with the implemented genetic clustering approaches. It is possible that genetic algorithms are only practically applicable to a certain approaches, being unable to form good solutions for others.

4.4 Trip Generation

We have now discussed how routing can be combined with both clustering and insertion to form complete trips, but there is still one more approach that will be considered in this project. By modifying a few existing routing algorithms namely Brute Force, Greedy Insertion and genetic routing, they can be used to generate full trips in one go. If the locations input for these algorithms is expanded, to include the starting point of the trip d times, the same algorithms can be used to find a multi-day trip. For Brute Force and genetic routing, this means the permutations being considered is expanded to include returning to the starting point d times. For Greedy Insertion, this means that all locations on the trip will first be inserted into the route, before inserting each return to the start.

Trip generation methods directly produce whole trips instead of needing some

compound approach of clustering and routing or routing and insertion. These algorithms were extended to include trip generation to investigate how the approach compared, by forming a route in one stage these algorithms may more effectively minimise cost than by combining two algorithms which take different approaches to finding an optimal solution.

5 Algorithm Evaluation and Results

This section will contain an explanation of the methodology taken to evaluate different approaches to Multi Day Trip Planning. This explanation will include a description of the approaches to be evaluated, alongside the experiment procedure used and any constraints that were placed on the problem. Later in this section, the results of these experiments will be presented and discussed.

5.1 Methodology

The purpose of this evaluation is to compare the performance of different approaches to Multi Day Trip Planning. The experimental process aimed to answer the following questions:

- How do the different approaches compare in terms of computation time?
- How do the different approaches compare in terms of the quality of trips produced?
- How does the performance of each approach scale with input complexity?

To answer these questions, each proposed approach will be applied to a range of different inputs, recording the computation time to produce a trip and the quality of the trip produced according to the cost function described in section 2.3.
For use in these experiments 25 testing data sets were created, each representing inputs for trips to major cities around the world. Each data set includes:

- A set of coordinates, representing different points of interest around the city.
- A list of durations, representing the time to be spent at each point of interest.
- A complete graph, containing the time taken to travel between each point of interest.

These were created using openrouteservice, the process of which was previously described in section 4.1.1 titled “Input Generation”. Due to previously discussed limitations of the openrouteservice API, each dataset was limited to 25 points of

interest.

To investigate how the performance of these algorithms scales with input complexity, experiments with a range of inputs for the number of locations and number of days in each trip were conducted. For these experiments, a subset of locations and their corresponding durations and graphs were selected from the 25 datasets, each approach was run on the same subset. Every approach was run on every dataset for every combination of locations and days. A full list of the combinations of the number of locations and number of days is available in table 1.

Table 1: Lists all combinations of the number of locations and number of days used in evaluation.

Number of Locations	Number of Days
25	7, 6, 5, 4
20	6, 5, 4, 3
15	5, 4, 3, 2
10	4, 3, 2
8	3, 2
5	2

Using various combinations of the implemented clustering, routing and trip generation algorithms, a total of 16 approaches were evaluated. The full list of these approaches and an explanation of each one, as well as the shorthand used to refer to each approach, is available in table 5.1. Most of these approaches are either trip generation methods, or different combinations of the clustering and routing algorithms previously described. The only exceptions to this are the 'Genetic Algorithm Clustering + Greedy Routing + Brute Force' (GAC+GR+BF) and 'Genetic Algorithm Centroid Clustering + Greedy Routing + Brute Force' (GACC+GR+BF) approaches. These approaches use greedy routing to evaluate the clusters found during the genetic algorithm process. Then, once evolution has completed, the resultant clusters are passed to Brute Force to find optimal routes within the clusters found. The hope is that greedy routing will allow a fast genetic algorithm, while using Brute Force will allow for an optimal final trip.

Approach	Description
Genetic Algorithm Trip Generation (GATG)	Generates trip using Genetic Algorithm
Greedy Insertion Trip Generation (GITG)	Generates trip using Greedy Insertion
K-Means Clustering + Greedy Routing (KM+GR)	Finds clusters using K-Means then finds intra-cluster routes Greedy Routing
K-Means Clustering + Greedy Insertion (KM+GI)	Finds clusters using K-Means then finds intra-cluster routes using Greedy Insertion Routing
K-Means Clustering + Brute Force (KM+BF)	Finds clusters using K-Means then finds intra-cluster routes using Brute Force Routing
K-Means Clustering + Convex Hull (KM+CH)	Finds clusters using K-Means then finds intra-cluster routes using Convex Hull Routing
K-Means Clustering + Genetic Algorithm Routing (KM+GAR)	Finds clusters using K-Means then finds intra-cluster routes using Genetic Algorithm Routing
Genetic Algorithm Clustering + Greedy Routing (GAC+GR)	Finds clusters using Genetic Clustering then finds intra-cluster routes using Greedy Routing
Genetic Algorithm Clustering + Greedy Routing + Brute Force (GAC+GR+BF)	Finds clusters using Genetic Clustering then finds intra-cluster routes using Greedy Routing - Once final clusters are obtained, Brute Force is used to find final routes
Genetic Algorithm Clustering + Greedy Insertion (GAC+GI)	Finds clusters using Genetic Clustering then finds intra-cluster routes using Greedy Insertion
Genetic Algorithm Centroid Clustering + Greedy Routing (GACC+GR)	Finds clusters using Genetic Centroid Clustering then finds intra-cluster routes using Greedy Routing
Genetic Algorithm Centroid Clustering + Greedy Routing + Brute Force (GACC+GR+BF)	Finds clusters using Genetic Centroid Clustering then finds intra-cluster routes using Greedy Routing - Once final clusters are obtained, Brute Force is used to find final routes
Genetic Algorithm Centroid Clustering + Greedy Insertion (GACC+GI)	Finds clusters using Genetic Centroid Clustering then finds intra-cluster routes using Greedy Insertion Routing
Greedy Routing + Greedy Insertion (GR+GI)	Finds route using Greedy Routing then splits route into a multi-day trip using Greedy Insertion
Convex Hull + Greedy Insertion (CH+GI)	Finds route using Convex Hull Routing then splits route into a multi-day trip using Greedy Insertion

Genetic Algorithm Routing + Greedy Insertion (GAR+GI)	Finds route using Genetic Algorithm Routing then splits route into a multi-day trip using Greedy Insertion

During these experiments, genetic algorithms all shared the following hyperparameters:

- Number of Generations: 150,
- Population Size: 50,
- Crossover Rate: 0.9.

Genetic Clustering approaches used a mutation rate of 0.1, while Genetic Routing approaches used a mutation rate of 0.4.

5.2 Results & Analysis

Through this experimentation each of the 25 datasets were tested using each combination of locations and days, resulting in a total of 450 inputs given to each approach. The average results of tests ran for several different combinations of locations and days are shown in tables 3 (8 locations and 3 days), 4 (15 locations and 4 days) and 5 (25 locations and 7 days). The computation time and evaluation shown for each approach is the average of values obtained from all datasets ran with these location and day inputs. Also included are the standard deviation (σ), which represents the deviation in results between different location data sets, and the coefficient of variation ($\frac{\sigma}{\mu}$), which represents the ratio between mean values and their standard deviation. The results are sorted by evaluation, and the best and worst values in each row are highlighted in bold and italic respectively. Table 6 shows average results across all data sets and input sizes.

Table 3: Lists average computation time and evaluation for each approach across tests using 8 locations and 3 days. This table also includes evaluations for Brute Force Trip Generation (BFTG) and Brute Force + Greedy Insertion (BF+GI)

Approach	Computation Time - μ	Computation Time - σ	Computation Time - $\frac{\sigma}{\mu}$	Evaluation - μ	Evaluation - σ	Evaluation - $\frac{\sigma}{\mu}$
BFTG	*10.963727*	*1.373076*	0.125238083	**580.843119**	**145.607101**	0.25068232
GAC+GR+BF	2.652661	0.924027	0.348339523	581.771109	147.287947	0.253171643
GATG	0.828975	0.470336	0.56737038	582.483376	146.081298	0.250790502
GAC+GR	2.652392	0.924045	0.348381659	583.209066	148.632858	0.254853476
GACC+GR+BF	1.901471	0.515371	0.271037872	592.071590	146.909233	0.248127482
GACC+GR	1.901226	0.515358	0.271065947	593.390534	148.322398	0.249957472
GAR+GI	0.691986	0.224216	0.324017778	650.987103	158.635774	0.243684972
BF+GI	0.300547	0.055220	0.183732569	651.094458	160.621126	0.246694046
CH+GI	0.003120	0.004787	*1.534394689*	657.864674	157.007055	0.238661629
GITG	0.004070	0.004507	1.107453904	661.532642	173.934502	0.26292656
GR+GI	**0.000024**	**0.000003**	**0.121703435**	666.328613	176.682224	*0.265157792*
KM+GAR	1.287709	0.576568	0.447747083	670.487638	156.321777	0.233146398
KM+BF	0.013186	0.002541	0.192707762	670.653112	156.392637	0.233194529
KM+GR	0.011910	0.001995	0.167545233	672.931430	159.437076	0.236929156
KM+CH	0.013184	0.002048	0.155312695	674.160886	157.178385	0.233146699
GAC+GI	4.552093	1.227828	0.269728177	969.678679	214.733237	0.221447828
GACC+GI	3.908192	0.733851	0.187772562	970.224031	213.070291	**0.219609373**
KM+GI	0.012178	0.002108	0.173063036	*996.233150*	*235.638250*	0.23652922

Table 4: Lists average computation time and evaluation for each approach across tests using 15 locations and 4 days

Approach	Computation Time - μ	Computation Time - σ	Computation Time - $\frac{\sigma}{\mu}$	Evaluation - μ	Evaluation - σ	Evaluation - $\frac{\sigma}{\mu}$
GAC+GR+BF	2.390620	0.457114	0.191211464	**929.924224**	129.695911	0.139469333
GATG	0.999130	0.155427	0.155562286	931.678599	131.730899	0.141390925
GAC+GR	2.387176	0.456494	0.191227802	932.052914	129.772740	0.139233233
GACC+GR+BF	2.160390	0.434917	0.201313923	941.841538	130.008533	0.138036524
GACC+GR	2.143543	0.434804	0.202843494	944.666649	**129.641948**	0.137235657
GITG	0.012276	0.003757	0.306096212	952.964413	132.694472	0.13924389
CH+GI	0.008618	0.002095	0.243065556	959.980725	144.641551	*0.150671308*
GR+GI	**0.000039**	**0.000004**	**0.104597412**	961.133817	135.196293	0.14066334
GAR+GI	0.939494	0.159025	0.169266722	963.440913	143.976282	0.14943966
KM+BF	*7.332659*	*31.810776*	*4.338231859*	1113.188018	159.945485	0.143682363
KM+GAR	1.684152	0.334373	0.198540614	1113.690387	160.192210	0.143839088
KM+GR	0.013239	0.004664	0.352271037	1117.856057	159.415227	0.142608009
KM+CH	0.016252	0.004993	0.307194959	1121.260628	161.795389	0.144297753
GACC+GI	6.847682	0.904047	0.132022276	1647.327226	225.550233	**0.1369189**
GAC+GI	6.735855	0.892049	0.132432856	1654.194200	230.241480	0.139186487
KM+GI	0.013876	0.004755	0.342677914	*1680.691761*	*232.201145*	0.138158079

Table 5: Lists average computation time and evaluation for each approach across tests using 25 locations and 7 days

Approach	Computation Time - μ	Computation Time - σ	Computation Time - $\frac{\sigma}{\mu}$	Evaluation - μ	Evaluation - σ	Evaluation - $\frac{\sigma}{\mu}$
GAC+GR+BF	2.757364	0.070118	0.025429523	**1224.921655**	218.453597	0.178340873
GAC+GR	2.752229	0.069379	**0.025208386**	1227.974831	219.077690	0.178405684
GATG	1.174132	0.065157	0.055493493	1249.211698	232.957427	0.186483546
GAR+GI	1.145115	0.042931	0.037490438	1298.771994	217.955721	0.16781677
GITG	0.037008	0.004152	0.112203647	1299.978163	229.402565	0.176466476
GACC+GR+BF	2.322438	0.354714	0.152733636	1300.039380	256.373114	0.197204114
GR+GI	**0.000055**	**0.000010**	0.174405584	1303.278191	224.211835	0.172036819
GACC+GR	2.171517	0.188289	0.086708524	1303.379124	256.455549	0.196762051
CH+GI	0.023478	0.003493	0.148782618	1303.547054	**216.694963**	0.166234861
KM+GAR	2.473922	0.134556	0.054389867	1423.636393	271.047180	0.190390735
KM+BF	1.697500	*4.332882*	*2.55250715*	1427.268910	282.173377	*0.197701621*
KM+GR	0.011821	0.001488	0.125869635	1428.105419	272.554303	0.190850269
KM+CH	0.016291	0.001898	0.116514653	1434.496055	272.688511	0.190093594
GACC+GI	*11.164482*	1.662562	0.148915268	2200.445359	321.745265	**0.146218248**
GAC+GI	9.161882	0.358560	0.039136028	2233.113133	344.559348	0.154295518
KM+GI	0.012839	0.001507	0.117347705	*2261.413558*	*383.340894*	0.169513839

Table 6: Lists average computation time and evaluation for each approach across all tests.

Approach	Computation Time - μ	Computation Time - σ	Computation Time - $\frac{\sigma}{\mu}$	Evaluation - μ	Evaluation - σ	Evaluation - $\frac{\sigma}{\mu}$
GAC+GR+BF	2.979977	7.530829	2.527143137	**1020.368065**	**375.291204**	0.367799833
GAC+GR	2.449077	0.573792	**0.234288951**	1022.932742	376.517470	0.368076467
GATG	1.056100	0.350635	0.332009669	1025.823776	377.272585	0.367775239
GACC+GR+BF	3.283957	11.213699	3.414691397	1026.018469	378.328927	0.368735007
GACC+GR	2.058841	0.552098	0.268159607	1035.318311	384.760096	0.37163459
GITG	0.016816	0.013973	0.830951736	1097.052085	404.128161	0.368376458
GAR+GI	0.979447	0.291409	0.297524026	1098.753750	405.460659	0.36901868
GR+GI	**0.000053**	**0.000078**	1.472593046	1101.631357	403.964025	0.366696193
CH+GI	0.012669	0.010609	0.837414977	1101.948744	405.566210	0.368044532
KM+BF	*9.931990*	*35.391964*	*3.563431295*	1142.732633	418.983039	**0.366650104**
KM+GAR	1.691711	0.608345	0.359603359	1232.529265	461.950473	0.374798787
KM+GR	0.013553	0.005196	0.383363287	1236.909841	463.293563	0.374557261
KM+CH	0.017589	0.006696	0.380662466	1243.418740	466.350523	0.375055087
GACC+GI	8.228977	3.850029	0.467862454	1792.357060	674.690213	0.376426231
GAC+GI	7.738760	3.107706	0.401576755	1802.357683	686.223849	0.380736773
KM+GI	0.014386	0.005420	0.376779259	*1830.032473*	*701.160152*	*0.383140825*

From analysing these results, there are a few approaches that stand out as being particularly effective. GAC+GR+BF consistently produces some of the best evaluations across all datasets, while also being relatively fast. In fact, approaches that utilise genetic algorithms for clustering or trip generation overwhelmingly outperform the conventional heuristic approaches; on every input with locations greater than 10, genetic algorithms held the top four spots for route evaluation. Other approaches of note are GITG, GR+GI and CH+GI, which all find trips within a few hundredths of a second, while remaining middle of the pack in terms of trip quality.

By far the worst performing approaches are GACC+GI, GAC+GI and KM+GI. The quality of routes produced by the approaches that perform Greedy Insertion Routing are

significant outliers amongst the other approaches, which begs the question as to why this is so. Logically, one would expect greedy insertion routing to produce a route at least as good and greedy routing, if not better. It is suggested that further investigation is needed to understand why this is the case, and whether this could be caused by a flawed implementation of the greedy insertion algorithm.

Amongst algorithms that start by clustering locations, K-Means Clustering appears to produce significantly worse routes than those found with genetic approaches. K-Means approaches are much faster than their genetic counterparts, but this trade off doesn't appear to be worth it considering that Routing+Insertion approaches are even faster and produce better routes. Even the K-Means inspired genetic algorithms are outperformed by the standard genetic algorithm approaches, indicating that these geographic heuristics for clustering do not translate well to producing a good trip.

Further evidence of this is how Convex Hull Routing produces the worst results out of Routing+Insertion approaches. Convex Hull Routing was included in this project to investigate whether a convex hull could provide a good starting point for greedy insertion, hopefully being able to speed up the process or produce better results. While convex hull routing does produce slightly worse results though, it did find routes around 25% faster than Greedy Insertion Trip Generation.

It is also worth noting that these geometric based heuristics often produced large coefficients of variation, indicating that compared to other approaches, different location inputs resulted in a larger difference in the quality of the routes produced.

The full set of experiment results is available in this project's related gitlab repository, located in 'data\results.csv'.

6 Conclusion and Future Work

The goal of this project was to investigate and compare different approaches to solving the proposed multi-day route planning problem. Throughout this project, a number of algorithms were investigated, implemented and evaluated to determine their effectiveness. From the results shown in section 5.2, a conclusion can be drawn that the most promising approaches are those that first cluster locations, before finding routes between them. Genetic clustering methods proved particularly exemplary, often finding the best trips out of the approaches tested, while still maintaining a relatively fast computation speed.

While the aims of this project have largely been accomplished, further study is required to resolve gaps in the potential solutions considered. Certain categories of approach are somewhat lacking in the diversity of algorithms implemented; clustering for example, only includes one conventional heuristic approach in the form of k-means, with other clustering algorithms being the meta-heuristic genetic algorithm approaches. Most notable is greedy insertion being the only implemented insertion algorithm. While Greedy Insertion proved effective when combined with a routing algorithm, it would have been interesting to have other insertion algorithms to compare it to.

Future work on this problem should investigate additional clustering algorithms, of particular recommendation would be researching spectral clustering, a clustering technique that aims to group together graph nodes that are closely connected. It would also likely be of value to examine further local search techniques such as 2-opt that, similar to greedy insertion, could potentially be implemented to create multi-day trips from pre-existing TSP routes. Finally, considering the success of genetic algorithms in this project, it could be beneficial to consider other meta-heuristic approaches such as Ant Colony Optimisation or Neural Network approaches.

7 Bibliography

References

Applegate, David L (2006). *The traveling salesman problem: a computational study*. Vol. 17. Princeton university press.

Bektas, Tolga (2006). "The multiple traveling salesman problem: an overview of formulations and solution procedures". In: *omega* 34.3, pp. 209–219.

Bellmore, Mandell and Saman Hong (1974). "Transformation of multisalesman problem to the standard traveling salesman problem". In: *Journal of the ACM (JACM)* 21.3, pp. 500–504.

Cormen, Thomas H et al. (2022). *Introduction to algorithms*. MIT press.

Dantzig, George B and John H Ramser (1959). "The truck dispatching problem". In: *Management science* 6.1, pp. 80–91.

Fisher, Marshall L and Ramchandran Jaikumar (1981). "A generalized assignment heuristic for vehicle routing". In: *Networks* 11.2, pp. 109–124.

Goutham, Mithun et al. (2023). "A convex hull cheapest insertion heuristic for the non-euclidean tsp". In: *arXiv preprint arXiv:2302.06582*.

Hartigan, John A and Manchek A Wong (1979). "Algorithm AS 136: A k-means clustering algorithm". In: *Journal of the royal statistical society. series c (applied statistics)* 28.1, pp. 100–108.

Jain, Anil K, M Narasimha Murty, and Patrick J Flynn (1999). "Data clustering: a review". In: *ACM computing surveys (CSUR)* 31.3, pp. 264–323.

Laporte, Gilbert (1992). "The traveling salesman problem: An overview of exact and approximate algorithms". In: *European Journal of Operational Research* 59.2, pp. 231–247.

Nallusamy, R et al. (2010). "Optimization of multiple vehicle routing problems using approximation algorithms". In: *arXiv preprint arXiv:1001.4197*.

Syswerda, Gilbert et al. (1989). “Uniform crossover in genetic algorithms.” In: *ICGA*. Vol. 3. 2–9.

Tang, Lixin et al. (2000). “A multiple traveling salesman problem model for hot rolling scheduling in Shanghai Baoshan Iron & Steel Complex”. In: *European Journal of Operational Research* 124.2, pp. 267–282.

Vansteenwegen, Pieter, Wouter Souffriau, and Dirk Van Oudheusden (2011). “The orienteering problem: A survey”. In: *European Journal of Operational Research* 209.1, pp. 1–10.

Vansteenwegen, Pieter and Dirk Van Oudheusden (2007). “The mobile tourist guide: an OR opportunity”. In: *OR insight* 20.3, pp. 21–27.