□

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
xxxx Words

Abstract

Write abstract

Acknowledgements

Write acknowledgements, must include openrouteservice and OpenStreetMap contributors.

Contents

1 Introduction

1.1 Motivation

Write about reasoning for this project, can copy a little bit over from presentation slides. Explain problem informally. Link into aims and objectives.

1.2 Aims and Objectives

Explin goals of the project, link in to methodology.

1.3 Methodology

Briefly explain how the project will be carried out. A more thorough description can be provided later on, i.e., when describing algorithms.

Mention code being available from github. Include a code segment and explain how figure captions denote filepath within the repository.

1.4 Summary

Explain what is in the rest of the report. "In this report I shall...", cover each section, etc.

2 Problem Formulation

2.1 Problem Description

Given a positive integer d and 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, 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

Update so durations aren't in the graph, instead something like D, mapping a duration to each location.

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. $d \in \mathbb{Z}, d > 0$
- $G = (V, E)$: 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 $y \in \mathbb{Q}, -90 \leq y \leq 90$.

- t: Duration, in minutes, indicating how much time to spend at this location.

  $t \in \mathbb{Z}, t > 0$.

- 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, our goal 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:

reword this

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

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:

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

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

Plan (and write) literature review

Remember to write about the strengths and weaknesses of existing work. At the end of this chapter you can then give a summary of the gaps that you'll be trying to improve with your work, and on the strengths that you will be maintaining in your work.

This literature review aims to explore existing research and approaches to other combinatorial optimisation problems. There is extensive previous research on various combinatorial problems, for example, the Travelling Salesman Problem, Vehicle Routing Problem and Tourist Trip Design Problem. It is important to understand how these problems are similar to the one presented in this report, as well as where those similarities end. By gaining an understanding of the strengths and limitations of existing approaches to similar problems, we can make better informed decisions regarding which approaches to investigate, how they may be adapted to suit our specific constraints and how they might be implemented in practice. While the approaches taken to these problems may not be directly applicable to our own, it is likely we can adapt their techniques to suit the specific constraints of this problem.

The Travelling Salesman Problem (TSP) is perhaps one of the most studied optimisation problems. This extensive research on the problem has acted as an 'engine of discovery for general-purpose techniques' offering 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 our own in their shared goal of minimising travel time, in fact, for inputs where $d = 1$, our 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 our project. Naturally, 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 certainly interesting, there isn't much discussion over 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 our own problem, in which our graph is asymmetric and based on travel time between locations, perhaps we could also use 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 our own problem, in which we are looking for several routes over multiple days, but without the optimisation objective of balancing routes.

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. TTDP shares the same motivation of tourist route planning as our own problem 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. Nevertheless, due to the similarities in the problem, it is worth investigating the approaches taken to solve TTDPs.

Gavalas et al. (2014)

3.1 Classical Traveling Salesman Problem (TSP)

- Definition and mathematical formulation
- Complexity analysis and NP-hardness
- Key solution approaches
- Relevance and limitations in relation to our specific problem

Recommended Literature:

- Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2006). *The Traveling Salesman Problem: A Computational Study.* Princeton University Press.
- Laporte, G. (1992). "The traveling salesman problem: An overview of exact and approximate algorithms." *European Journal of Operational Research*, 59(2), 231-247.

- Lin, S., & Kernighan, B. W. (1973). "An effective heuristic algorithm for the traveling-salesman problem." *Operations Research*, 21(2), 498-516.

- Helsgaun, K. (2000). "An effective implementation of the Lin–Kernighan heuristic." *European Journal of Operational Research*, 126(1), 106-130.

3.2 Multiple Traveling Salesman Problem (mTSP)

- Extension of the TSP with multiple agents
- Mathematical formulation differences from TSP
- Application to multi-day planning scenarios
- Connection to our requirement of visiting the starting point d times

Recommended Literature:

- Bektas, T. (2006). "The multiple traveling salesman problem: an overview of formulations and solution procedures." *Omega*, 34(3), 209-219.

- Kara, I., & Bektas, T. (2006). "Integer linear programming formulations of multiple salesman problems and its variations." *European Journal of Operational Research*, 174(3), 1449-1458.

- Gavish, B., & Srikanth, K. (1986). "An optimal solution method for large-scale multiple traveling salesmen problems." *Operations Research*, 34(5), 698-717.

- Carter, A. E., & Ragsdale, C. T. (2006). "A new approach to solving the multiple traveling salesperson problem using genetic algorithms." *European Journal of Operational Research*, 175(1), 246-257.

3.3 Vehicle Routing Problem (VRP) and Variants

- Basic VRP definition and formulation
- Vehicle Routing Problem with Multiple Trips (VRPMT)

- Capacitated VRP and other variants
- Relevance to our balanced route planning requirement

Recommended Literature:

- Toth, P., & Vigo, D. (Eds.). (2002). *The Vehicle Routing Problem.* SIAM Monographs on Discrete Mathematics and Applications.
- Cattaruzza, D., Absi, N., & Feillet, D. (2016). "Vehicle routing problems with multiple trips." *4OR*, 14(3), 223-259.
- Brandão, J., & Mercer, A. (1997). "A tabu search algorithm for the multi-trip vehicle routing and scheduling problem." *European Journal of Operational Research*, 100(1), 180-191.
- Olivera, A., & Viera, O. (2007). "Adaptive memory programming for the vehicle routing problem with multiple trips." *Computers & Operations Research*, 34(1), 28-47.

3.4 Tourist Trip Design Problem (TTDP)

- Problem definition focusing on tourist-specific constraints
- Time-dependent considerations and point-of-interest selection
- Personalization aspects in tourist routing
- Direct applicability to our problem's tourism context

Recommended Literature:

- Vansteenwegen, P., Souffriau, W., & Van Oudheusden, D. (2011). "The orienteering problem: A survey." *European Journal of Operational Research*, 209(1), 1-10.
- Gavalas, D., Konstantopoulos, C., Mastakas, K., & Pantziou, G. (2014). "A survey on algorithmic approaches for solving tourist trip design problems." *Journal of Heuristics*, 20(3), 291-328.

- Souffriau, W., Vansteenwegen, P., Vanden Berghe, G., & Van Oudheusden, D. (2013). "The planning of cycle trips in the province of East Flanders." *Omega*, 41(3), 522-531.

- Garcia, A., Vansteenwegen, P., Arbelaitz, O., Souffriau, W., & Linaza, M. T. (2013). "Integrating public transportation in personalised electronic tourist guides." *Computers & Operations Research*, 40(3), 758-774.

3.5 Multi-Objective Optimization in Routing Problems

- Balancing competing objectives (like total weight vs. variance)
- Pareto optimality concepts
- Solution approaches for multi-objective routing
- Applicability to our dual-objective function

Recommended Literature:

- Jozefowiez, N., Semet, F., & Talbi, E. G. (2008). "Multi-objective vehicle routing problems." *European Journal of Operational Research*, 189(2), 293-309.

- Paquete, L., & Stützle, T. (2006). "A study of stochastic local search algorithms for the biobjective QAP with correlated flow matrices." *European Journal of Operational Research*, 169(3), 943-959.

- Laporte, G., Semet, F., Matl, P., & Voß, S. (2018). "Multi-objective vehicle routing problem." *Operations Research Perspectives*, 5, 50-57.

- Coello, C. A. C., Lamont, G. B., & Van Veldhuizen, D. A. (2007). *Evolutionary Algorithms for Solving Multi-Objective Problems*. Springer.

3.6 Balance-Oriented Routing Problems

- Problems focusing on workload balancing

- Min-max objectives in routing
- Variance minimization approaches
- Connection to our goal of minimizing variance between trips

Recommended Literature:

- Jozefowiez, N., Semet, F., & Talbi, E. G. (2009). "An evolutionary algorithm for the vehicle routing problem with route balancing." *European Journal of Operational Research*, 195(3), 761-769.
- Dell'Amico, M., Monaci, M., Pagani, C., & Vigo, D. (2007). "Heuristic approaches for the fleet size and mix vehicle routing problem with time windows." *Transportation Science*, 41(4), 516-526.
- Lee, T. R., & Ueng, J. H. (1999). "A study of vehicle routing problems with load-balancing." *International Journal of Physical Distribution & Logistics Management*, 29(10), 646-657.
- Liu, R., Xie, X., Augusto, V., & Rodriguez, C. (2013). "Heuristic algorithms for a vehicle routing problem with simultaneous delivery and pickup and time windows in home health care." *European Journal of Operational Research*, 230(3), 475-486.

3.7 Synthesis and Research Gaps

- Comparison of problem characteristics across reviewed literature
- Key methodological approaches applicable to our problem
- Identification of research gaps in addressing our specific problem constraints
- Potential directions for adaptation of existing methodologies

Recommended Literature:

- Laporte, G. (2009). "Fifty years of vehicle routing." *Transportation Science*, 43(4), 408-416.

- Cordeau, J. F., Gendreau, M., Laporte, G., Potvin, J. Y., & Semet, F. (2002). "A guide to vehicle routing heuristics." *Journal of the Operational Research Society*, 53(5), 512-522.

- Eksioglu, B., Vural, A. V., & Reisman, A. (2009). "The vehicle routing problem: A taxonomic review." *Computers & Industrial Engineering*, 57(4), 1472-1483.

- Vidal, T., Crainic, T. G., Gendreau, M., & Prins, C. (2013). "Heuristics for multi-attribute vehicle routing problems: A survey and synthesis." *European Journal of Operational Research*, 231(1), 1-21.

3.8 Conclusion

- Summary of most relevant approaches

- Recommendation for methodological direction

- Justification for selected approach based on literature findings

4 Algorithms Investigated and Their Implementation

There should be a link either here or at end of literature which forms the basic for different methods (clustering, routing, trip generation).

Brief Paragraph describing different types of algorithm used (Routing then cluster, Cluster then Routing, Genetic, etc.)

Remember to justify the choice of algorithms. You may also need to explain how to adopt these algorithms in your work. A figure showing the ralationship between different components of your work may also help.

4.1 Implementation Pre-Requisites

There are two things need before we begin, a way of generating inputs to use our algorithms on, and an implementation of our objective function so we can evaluate our results.

4.1.1 Input Generation

For our input we will be using Openrouteservice's Points of Interest (PoI) and Distance Matrix API services. With the PoI service, we can input a location, such as a city centre, and retrieve a specified number of locations of interest around the area. The PoI service also allows us to limit the locations returned to certain categories allowing us to specifically find locations that one would likely be interested in visiting, for example points labelled 'tourism', 'historic' or 'arts and culture'.

After we have these locations, we can use the Distance Matrix to create a graph linking each location to each other with a weight associated with travelling between them. We can configure our API call to this service 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 we can test. Other alternative API's were looked into, such as the Google Maps API or Mapbox, but all the alternatives found were either too expensive or equally, if not more, restrictive.

With these, we can generate a list of durations associated with the found locations, as well as a number of days for the trip, and then our input is ready.

4.1.2 Objective Function

As described in the Problem Formulation, our objective function aims to calculate a cost for a given route. This function is needed such that we can compare our approaches and algorithms based on the quality of their results. Figure 1 shows how this was implemented in our project.

□

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

4.1.3 Project Code

All the code used in the project is available to find at: `https://git.cs.bham.ac.uk/projects-2024-25/jhl114`. Code snippets shown in this report, such as figure 1, are captioned detailing where in the git repository their code can be found.

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). In our 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. The goal of our 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 we can use a set of clusters alongside our graph and duration inputs to find a trip.

□

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

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). Our inputs will always contain only two dimensions, and we will be setting a maximum number of iterations, this makes both n and i constants allowing us to simplify the time complexity to $O(mk)$.

In our implementation of K-Means we will initialise our centroids by randomly selecting unique locations from our input, and placing our 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, we can be sure that all clusters have at least one location assigned to them. We will be using the coordinates of our locations to calculate the Euclidean distances between locations and centroids, these locations will be assigned to the cluster of the closest centroid. Figure 3 shows how this was accomplished.

□

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. Our implementation is shown in figure 4.

□

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. Our convergence criterion is that the centroids stop changing between iterations, i.e., the centroids are the same as the previous iteration. Our python implementation of this is shown in figure 5.

□

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 doesn't even 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).

For us to perform selection, and find the best solutions in a population, we need to assign some fitness to each individual. To do this we will combine the cluster assignments with a chosen routing algorithm, and then apply our cost function to the route found. This cost will be used to rank our population, helping us find 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 our 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. Genetic clustering has 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.

Figure out how to cite time complexity of genetic algorithms. Or go into explanation for our implementation.

For this genetic clustering, 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]

□

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

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

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

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

□

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). Once the population has been evaluated, the best two individuals are chosen as parents, as figure 9 shows.

□

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. We include some randomly generated individuals in an effort to increase genetic diversity and exploration of the search space. Figure 10 shows how this is decided in our implementation. For each individual a random number is generated between 0 and 1, if this number is lower than our crossover rate the individual is created via crossover, otherwise it will be randomly generated.

□

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

For our implementation of crossover, we will be performing a uniform crossover, 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 it's a good choice. With our 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 applying this method to our problem. 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 our python implementation.

□

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 our python implementation of this.

□

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

After crossover is complete, we randomly mutate the created offspring in the hopes of increasing genetic diversity and escaping local optima. We mutate our genome by iterating through each gene and randomly deciding if it will mutate or not, the likeliness of mutation is decided by our 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, we restart the process 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 genetic clustering using 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. The 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.

Discuss why chosen, no clue what to put here, I just did it bc it's cool.

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, we will have to reconsider our crossover and mutation methods. 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.

Figure out how to cite time complexity of genetic algorithms. Or go into explanation for our implementation.

Our new 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 our centroid-based genome.

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

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

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

□

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

The '_assign_nodes_to_centroid' method shown in figure 19 is the same method we used for K-Means, shown in figure 3. With the population evaluated we can select the best individuals for use in creating the next generation. To create the next generation we enact 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, we will be taking the values of both parents and merging them 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.

Create and insert drawing of merging parent centroids

Similarly to the previous implementation of crossover, we need to reorder the centroids to ensure that both parents are consistent with each other. This time, because are genome isn't formed of discrete values we can rename, we will instead take an approach of finding the most similar centroids from each parent and merging them. This is performed by calculating the distances between each of parent 1s centroids to each of parent 2s, 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.

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

For mutation, we will go through each centroid of each individual and randomly decide if it will mutate or not. If a centroid is chosen to mutate, we will randomly select a new location for it within the bound of the input coordinates. 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.

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, we will have a population of individuals that have been evolved to find good centroids for clustering the input. 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 our clustering to produce trips. Again, a line graph of the best evaluation found for each generation of the process, this 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.

Discuss why chosen, to investigate how a k-means like approach would work with a genetic algorithm and whether regular clustering is better.

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 as we introduce our greedy insertion 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

Brute Force 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 has a time complexity of $O(n!)$.

Maybe cite time complexity of brute force or permutations of a set?

It's worth noting that in our implementation, since all routes must return to the starting point, the final location of a route is fixed. Therefore, we only need to consider $n-1$ locations, 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 $L : \{0, 1, \ldots, n-1\}$ we can calculate k's corresponding permutation 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 our 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 our 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 $x_1 = 2, r_1 = 5$
We select $L_{x_1} = L_2 = c$ as the first element of our 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 $x_2 = 2, r_2 = 1$
We select $L_{x_2} = L_2 = d$ as the second element of our 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, $x_3 = 1, r_3 = 0$
We select $L_{x_3} = L_1 = b$ as the third element of our 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 us to check every possible route. Our python version of this is shown in figure 25.

□

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

With this method of generating routes in place, we just need to iterate through them all, evaluate them, and keep track of the best found. Our implementation of this is shown in figure 26.

□

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).

□

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 we were to use our maximum input size of 25 locations, over 620 sextillion routes would be considered —which, assuming the same rate of routes evaluated per second, would take over 800 billion years.

Brute force was included in this project for its use as a benchmark. Considering we will be evaluating algorithms based on their speed and the quality of their results, Brute Force offers both an upper bound for quality and a lower bound for speed.

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, we are checking the distance of every other location to find the closest, giving our 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. For our implementation we will simply make a copy of our graph input, then iteratively find the closest locations. When a location is visited, the edges leading to it are given an infinite cost to ensure that it is not visited again. Our python implementation of greedy routing is shown in figure 28.

□

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.

□

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 excell, 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 our 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 our route, we will be working through the set of unvisited locations and finding the best location to insert each one into our route. To find the best insertion point, we will evaluate the cost of the route if the location is inserted at every possible position. 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

□

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 Birmingam input is shown in figure31.

□

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 we have a route already, but want to add more locations for it, we can perform greedy insertion and find the best point for each location to insert into the pre-existing route. Furthermore, if for our new locations input, we provide a set consisting of a route's starting point d number of times, we can add d number of days to a trip. This allows us to create multi-day trips out of the routes found via our routing algorithms, by taking their routes and splitting them where greedy insertion believes is best.

4.3.4 Convex Hull

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 such as our own, 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, we can use the convex hull as a starting point, and then use greedy insertion to find a full route. The time complexity of finding a convex hull is $O(nh)$, where n is the number of locations, and h is the number of points on the hull. In the worst case, where all of the points are on the hull, this becomes $O(n^2)$.

Cite time complexity?

To find a convex hull we will be using a gift wrapping, or jarvis march, algorithm. 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 we can arbitrarily select a point as a candidate for the next point on the hull, then calculating the cross product $\vec{HC} \times \vec{HI}$, where H is the most recently added hull point, C is the candidate point and I is the point being evaluated. If the cross product is positive, then I is more counter-clockwise than C, and we can replace C with I as the new candidate to be added to the hull. 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.

□

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

□

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

Once our hull is found, we can use it as a starting point for greedy insertion. We will use the hull as our starting 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 our birmingham example.

□

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

Convex hull 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, we can also use genetic algorithms 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 us to use a routing algorithm on our genome and evaluate these routes. 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, we will be shuffling the order in which locations are visited to try and find a better route. Our 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. Our python implementation of this crossover method is shown in figure 37.

□

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

□

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

For mutation, we will randomly decide a number of individuals 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 reffig:GeneticRouting.mutation, the final location isn't considered for swapping, as all routes end at the starting point.

□

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

By repeatedly performing selection, crossover and mutation our 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.

□

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

□

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

□

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.

Discuss why chosen, seriously what do I put here? I just thought they were cool.

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 our permutation based routing algorithms, brute force and genetic routing, we can use them to generate full trips in one go. It was previously shown how, when given a set of locations, these algorithms attempt to find an optimal permutation of the set. If this set of locations is expanded, to include the starting point of the trip d times, we can use the same algorithms to find a multi-day trip.

5 Evaluation and Comparison

5.1 Methodology

5.1.1 Constraints

5.2 Results & Analysis

Gather data for London and Birmingham with POI

Write paragraph about experiment process. Comparison based on computation time and route evaluation. Describe how route is evaluated. Describe data being tested on.

Present comparison of different combinations of algorithms on different inputs.

Reflect about the questionns you are trying to answer with your evaluation. You can have onne subsection for each question that you are trying to answer. It's also important to justify your choices when it comes to the methodology used for evaluation.

6 Conclusion and Future Work

Write conclusion, discuss results, comparison of algorithms, etc.

6.1 Project Reflection

Reflect on the project, what went well, what didn't go well, what I would do differently. This should lead into future work.

6.2 Future work

Discuss further work, what I will be doing to improve the project

Discuss spectral clustering

Discuss Neural Networks

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.

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

Gavalas, Damianos et al. (2014). "Mobile recommender systems in tourism". In: *Journal of network and computer applications* 39, pp. 319–333.

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.

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

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