40.018 Heuristics and Systems Theory Project Report

Su Yi Heng 1007800, Dylan Gay 1007831, Kenneth Koh 1007782

July 2025

1 Introduction

Inspired by the use of electric vehicles (EVs) to support parcel delivery efforts, our project aims to find the optimal routes for a fleet of EVs to drop-off their parcels at their destination from the origin station. With *FedEx Singapore* as our chosen courier for this project, we seek to minimise the distance travelled for all EVs utilised. We aim to account for factors such as delivery locations, location of nearest charging stations, and number of delivery vehicles used.

2 Exact Method

Let:

- N: set of all nodes (including depot D, customer nodes C, and charging stations S)
- K: set of available vehicles
- $B_{\max}$: battery capacity
- d_{ij}, e_{ij}: the distance and energy required to travel from node i to node j respectively

$$\text{minimise} \quad \sum_{k\in K}\sum_{i\neq j\in N} d_{ij}\cdot x_{ijk} \tag{1}$$

$$\text{s.t.} \quad \sum_{k\in K}\sum_{i\in N, i\neq j} x_{ijk} = 1 \quad \forall j \in C \tag{2}$$

$$\sum_{i\in N} x_{ijk} = \sum_{i\in N} x_{jik} \quad \forall j \in C,\, k \in K \tag{3}$$

$$\sum_{j\in N\setminus D} x_{0jk} = u_k \quad \forall k \in K \tag{4}$$

$$\sum_{i\in N\setminus D} x_{i0k} = u_k \quad \forall k \in K \tag{5}$$

$$q_{jk} = q_{ik} - e_{ij}\cdot x_{ijk} + \delta_{ik} \quad \forall i \neq j, j \neq 0,\, k \in K \tag{6}$$

$$\delta_{ik} \leq B_{\max} - q_{ik} \quad \forall i \in S,\, k \in K \tag{7}$$

$$\delta_{ik} = 0 \quad \forall i \notin S,\, k \in K \tag{8}$$

$$q_{ik} \leq B_{max} \quad \forall k \in K \tag{9}$$

$$q_{0k} = B_{max} \quad \forall k \in K \tag{10}$$

$$z_{ik} - z_{jk} + (n-1)\times x_{ijk} \leq n-2 \quad \forall i \neq j, i, j \in N \setminus D, k \in K \tag{11}$$

$$z_{0k} = 0 \quad \forall k \in K \tag{12}$$

$$\sum_{i,j\in N, i\neq j} t_{ijk} \leq 540 \quad k \in K \tag{13}$$

Firstly, we ensure that each customer (delivery location) is visited exactly once by a single vehicle (2). Additionally, vehicles that enter a given location must leave that location as well (3) for flow conservation, while all utilised vehicle(s) must enter and exit through depot (4), (5).

The battery dynamics consist of the current charge q_{ik}, the energy consumed $e_{ij} \cdot x_{ijk}$ and the charging amount (if destination is a charger) δ_{ik} (6). Meanwhile overcharging an EV's battery at charger nodes is prevented using (7), and we ensure that the battery does not charge at non-charger nodes (8). We also ensure battery level must not exceed design capacity (9), and all electric vehicle start with full battery capacity at the depot (10)

We also have the subtour elimination constraints (11). Since depots are first to be visited, we initialise their order of visit to be 0. (12)

Lastly, delivery hours are limited to 9 hours $\approx$ 540 minutes a day, specifically between 0900 - 1800. This ensures that all vehicles return to the depot on time. (13)

2.1 Decision Variables

Name	**Definition of Decision Variable**
Vehicle Usage Indicator	$u_k \in \{0,1\},\ \forall k \in K$. If vehicle k is used, $u_k = 1$
Routing Decision	$x_{ijk} \in \{0,1\},\ \forall i,j \in N,\ i \neq j,\ k \in K$ $x_{ijk} = 1$ if vehicle k travels from node i to node j
Battery Level	$q_{ik} \in [0, B_{\max}],\ \forall i \in N,\ k \in K$ Records the current battery level at node i for vehicle k
Charging Amount	$\delta_{ik} \in [0, B_{\max}],\ \forall i \in N,\ k \in K$ Amount of energy vehicle k charges at node i. If $i \notin S$, then $\delta_{ik} = 0$
Visit Order (MTZ variable)	$z_{ik} \in [0, \lvert N \rvert - 1],\ \forall i \in N,\ k \in K$ Ordering of visits to eliminate subtours for vehicle k
Time Taken	$t_{ijk} \geq 0,\ \forall k \in K$ Length of time to travel from node i to node j for vehicle k

Table 1: Decision Variables

2.2 Objective Function

While our model focuses on **minimising total distance travelled**—which we assume to be closely correlated with energy consumption—there are several other objectives that may be relevant depending on operational priorities. These include:

- **Minimising the maximum delivery time** across all routes (to ensure timely service),
- **Maximising the number of customers served** within a time window (especially when total demand exceeds vehicle capacity),
- **Minimising the number of late deliveries**, and
- **Minimising the number of vehicles deployed** to reduce operational costs.

Each objective can lead to different trade-offs and solution structures, depending on the context and constraints of the routing scenario.

2.3 Exact solution

In order to accommodate to the complexity of the problem, and to achieve an optimal solution using Gurobi, we had to scale down our problem size to a reasonable one and adjusted some parameters. Specifically, we reduced the battery capacity of each vehicle (to encourage the use of charging nodes within a smaller instance) and tightened the time constraint per vehicle (to promote the use of multiple vehicles, as the solver would otherwise default to using a single vehicle for efficiency) We are able to obtain an optimal solution for a problem size consisting of 10 vehicles, 2 chargers, and 2 vehicles.

However depending on the time constraint set on vehicles, the complexity of the problem varies significantly, and optimal solution changes as well.

1. 300 mins/vehicle time constraint. Only 1 vehicle utilised (239 mins), optimality can be achieved in 20 seconds.
2. 225 mins/vehicle time constraint. The tightening of time constraint is intentional, as we want to utilise more than one vehicle to better model a fleet routing problem rather than just a single vehicle. The optimal solution obtained is 2 vehicles utilised, optimality achieved in 518.9 seconds.

Increasing the problem size any further, even just slightly makes the problem much more complex. For example, with 15 customers, 5 charging points and 3 vehicles, the solver is unable to find optimality even after 3 hours, achieving a gap of 10% at best.

3 Data and Processing Steps

There are three main components for our data. We have our client locations, our charging stations as well as our depots.

3.1 Obtaining Charging Stations

We obtained the network of 7811 charging stations via (Datamall Feb 2025). The dataset provides the latitude, longitude, charger type (charging speed and connector type), operator name amongst others. However, for simplicity sake, we simply assume that the chargers are all equal in capability.

Additionally, since each charging location has multiple chargers, we simply group the chargers of the same location together using R. This gives us a total of 2036 unique charging locations. This is shown in the image below.

3.2 Obtaining Depot Location

Based on Fedex's website, we have identified the depot location as the sorting facility located at 90 ALPS Avenue, near Changi Airport.

3.3 Obtaining Client Locations

Using data obtained from Github (DataGov 2017), we have the locations of HDBs in Singapore. We will then randomly sample c customer nodes from these locations. There are over 82000 locations in this dataset.

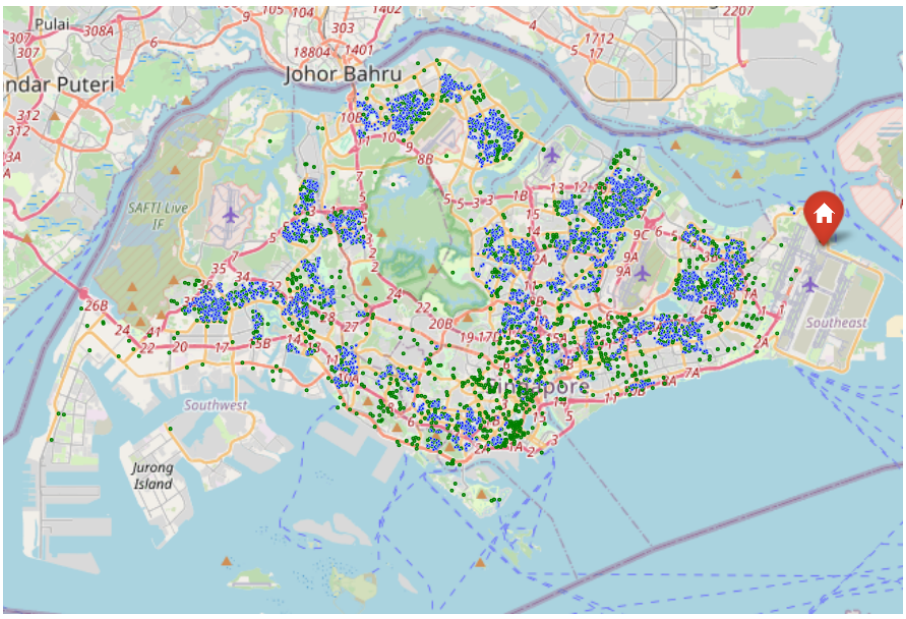


Figure 1: Delivery, Charging and Depot Locations

Delivery Locations: Blue markers indicate locations for deliveries.
Charging Stations: Green markers represent available EV charging points.
Depot Location: Location marker represents the location of the depot.

However, for the actual routing problem, we will sample a subsection of these locations to reduce computational complexity. The combination of our delivery, charging and depot location(s) forms our list of locations, which we then use to calculate the euclidean distances between each points to form a 2D-matrix.

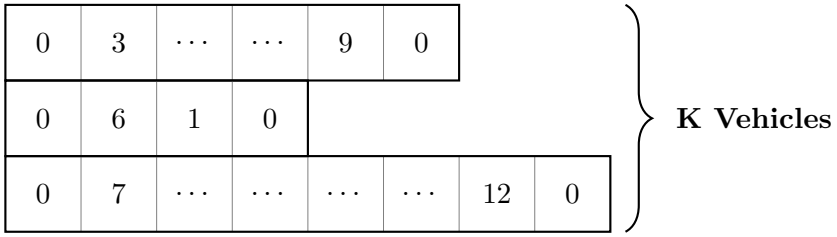
3.4 Additional Info

Our delivery vehicles each have a battery capacity of 60kWh (approximate), which equates to a battery discharge rate of 0.187kWh/km. We multiply this to the distance matrix to obtain an energy matrix which informs us on the energy required to travel between locations.

4 Metaheuristic - Simulated Annealing

4.1 Solution Encoding

Our solution is encoded using a discrete permutation encoding, and implemented in Python as a nested array.



Each client location can only be visited once (unless it is a charging station). Additionally, each location must start and end with 0, which is our depot node.

4.2 Standalone Neighbourhood Search Operators

4.2.1 Swap Between Vehicle Operator

Listing 4: The swap procedure can be described as follows:

- We randomly choose two vehicles from the K vehicles in our current solution.
- Swap operation: We choose two random client locations from each vehicle and perform a swap. The chosen locations must represent a client delivery location, and not a charger or the depot.

This ensures that the total number of clients each vehicle has remains the same.

4.2.2 Insertion Operator

Listing 5: The insertion procedure can be described as follows:

- Similar to swap, we first randomly choose two vehicles from the K vehicles in our current solution.
- We then randomly select up to 3 customers, from vehicle 1.
- For each customer, randomly insert them into a position in vehicle 2, such that the positions are after the starting and ending depot.

In this case, the number of clients remaining in each vehicle is **not the same**. An example to illustrate this insertion operator is shown below:

$$\text{Vehicle 1: } [0, 2, 1, 5, 0] \quad \text{Vehicle 2: } [0, 3, 4, 7, 0]$$

An insertion operator could choose locations 2 and 1 from vehicle 1, which are then randomly inserted into vehicle 2.

$$\text{Vehicle 1: } [0, 5, 0] \quad \text{Vehicle 2: } [0, 3, 2, 4, 7, 1, 0]$$

The feasibility of the new solution might thus be affected. With more locations, our vehicle could either violate the time or battery constraints, which will be addressed below.

4.2.3 K-Opt Operator

In classic K-Opt applications for TSP, it is applied within one vehicle. However, it is possible for our scenario to use K-Opt between multiple vehicles.

K-Opt within a vehicle:
Listing 6: Firstly, we randomly select a vehicle. If the value of K is 2, we select two *customer* positions, and reverse the route trapped between these two segments. However, if the value of K is 3, we select 3 customer positions, and partition the route into 3 segments. We either reverse one of the segments or swap two adjacent segments. This is mostly similar to the lecture notes example.

The time complexity of 2-opt is $O(n^2)$, while the time-complexity of 3-opt is $O(n^3)$, for n customers, if we wanted to find the best solution in the neighbourhood, across all iterations. One thing to note is that our charging stations are not included in this time-complexity calculation.

2-Opt between vehicle(s):
Listing 7: An alternate implementation of K-Opt is to swap segments between vehicles. We randomly select two vehicles who plan to visit more than 2 customers. Thereafter, for each customer (avoid the depot nodes), we perform the K-Opt operation.

To illustrate how this works, suppose that we have two vehicles. We have randomly chosen two short segments in green to cut for each of the vehicles. Each segment s_i is chosen such that $1 < s_i \leq |route|$.

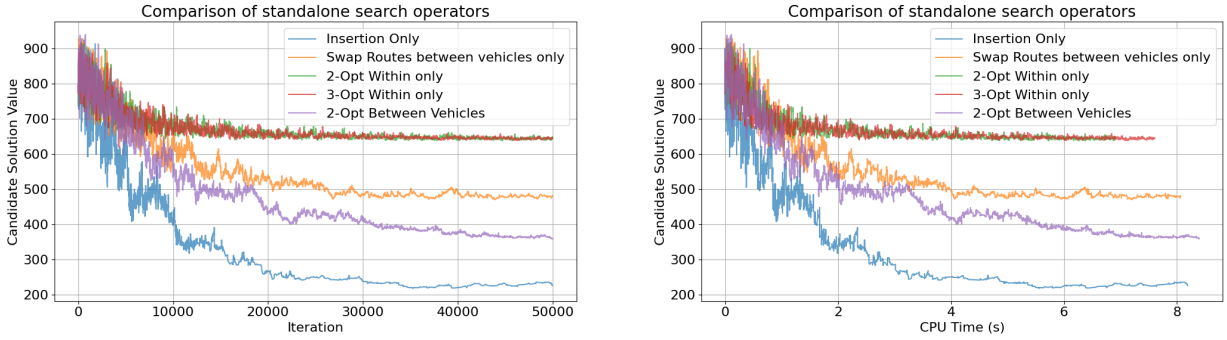
$$\text{Vehicle 1: } [0, 2, 10, 5, 6, 9, 0] \quad \text{Vehicle 2: } [0, 3, 4, 8, 1, 0]$$

Thereafter, we swap the segments as such:

$$\text{Vehicle 1: } [0, 2, 3, 4, 9, 0] \quad \text{Vehicle 2: } [0, 10, 5, 6, 8, 1, 0]$$

If the cost of selecting two routes can be neglected, its time complexity is the same as 2-opt. This method preserves the relative ordering of customer routes while, in theory, allocating routes such that it is more convenient for another vehicle. Toward the end of the project, we discovered that there are many other metaheuristic operators such as CROSS (which generalises the K-Opt) which we did not implement due to time constraints. (Fei et al. 2025)

4.3 Comparing the performance of standalone search operators



(a) Solution Value against Iterations

(b) Solution Value against CPU Time (s)

Figure 2: Performance of standalone search operators compared

We can visualize the performance of all the search operators evaluated so far. The k-opt operators (k=2 and k=3) show comparable behavior, with the poorest performance in terms of objective value achieved but with a faster computational time. The 2-Opt Between Vehicle operator performs the 2nd best, indicating that shifting routes between vehicles gives significant routing gains. However, among

all standalone operators, insertion is the most effective, reaching a minimum candidate solution value of 218. However, it requires a longer time of 8.1s to complete.

Search Operator	**Best Objective Value**	**Time to Complete 50,000 Iterations (s)**
K-opt (k=2, k=3) Within Vehicle	∼638	6.9–7.6
2-Opt Between Vehicles	359	8.39
Swap Between Vehicles	470	8.06
Insertion	218	8.18

Table 2: Performance Comparison of Standalone Search Operators

4.4 Composite Neighbourhood Search Operators

4.4.1 Insertion Operator + Swap Within

The swap-within operator is deliberately implemented to build upon the insertion-only approach to refine vehicle routing solutions.
Consider an optimization process using the insertion operator, resulting in the following candidate solution:

Vehicle 1: $[0, 0] \approx 0$ mins Vehicle 2: $[0, 3, 2, 4, 6, 1, 0] \approx 300$ mins Vehicle 3: $[0, 5, 0] \approx 20$ mins

Suppose that the optimal solution is actually:

Vehicle 1: $[0, 0] \approx 0$ mins Vehicle 2: $[0, 1, 6, 4, 2, 3, 0] \approx 235$ mins Vehicle 3: $[0, 5, 0] \approx 20$ mins

which is simply a reordering of the client routes of vehicle 2.

This discrepancy causes the metaheuristic approach to become trapped in a local optimum. A possible next step for the optimizer would be to transfer all the client locations to vehicle 2, which will reduce the distance travelled (since vehicle 2 need not travel at all after such an allocation). This is because our primary objective, as formulated in (1), is to *minimize the total distance traveled*. However, direct relocation of client location 5 from Vehicle 3 to Vehicle 2 could be infeasible due to Vehicle 2's operational time constraints.

Similarly, transferring locations from Vehicle 2 to Vehicle 1 as an intermediary to achieve the desired swap in locations is unlikely since it this will lead to a longer distance overall (Vehicle 1 which is previously not deployed now has to travel twice the distance, from the depot to the client and back to the depot). This is true especially in later stages of simulated annealing, where lower temperatures restrict acceptance of suboptimal solutions.

To escape local optima, we decided to combine insertion with a swap within operator. Rather than redistributing clients between vehicles, this operator reorders locations within a single chosen vehicle. This allows a larger neighbourhood of solutions to be explored beyond the insertion operator. The time complexity of this composite operator is likely to be the same as 2-Opt, assuming for each insertion we have (n-1) possible places to insert and for each swap (even in worse case where a vehicle has all n clients), there are n-1 possible locations to swap with. Hence:

$$\underbrace{O(n^2)}_{\text{Insertion}} + \underbrace{O(n^2)}_{\text{Swap Within}} = O(n^2)$$

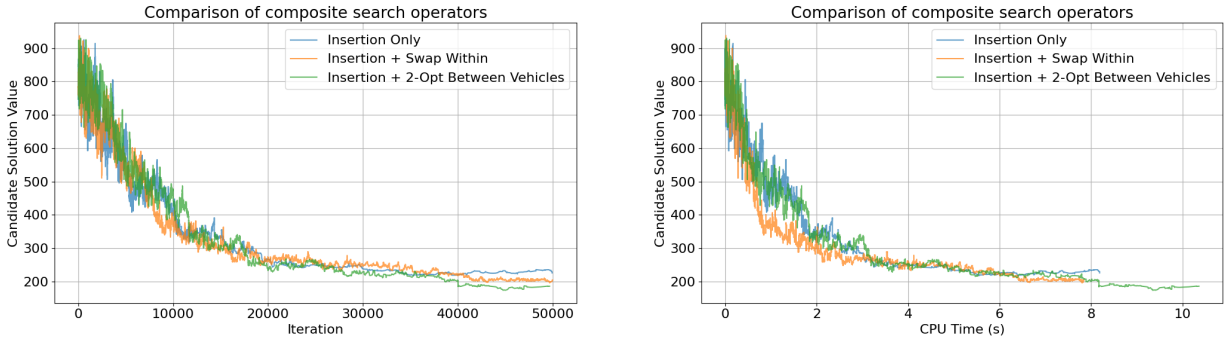
Since this method is effectively combining two neighbourhoods, a neighbourhood selection procedure will be elaborated on in 4.6

4.4.2 Insertion Operator + 2-Opt Between

Additionally, as the 2-Opt Between has previously shown promising performance in the standalone benchmark (Figure 2), we can also include it as a possible combination of neighbourhood search operators.

4.5 Comparing the performance of Composite Neighbourhood Search Operators

Next, we consider composite search operators that uses a neighborhood selection strategy to apply one of several operators at each step. We keep the standalone **insertion-only** operator as a control for comparison.



(a) Solution Value against Iterations

(b) Solution Value against CPU Time (s)

Figure 3: Performance Comparison of Standalone Search Operators

Among these, the combination of Insertion + 2-Opt between vehicles achieved our best result, reaching a candidate solution value of **174** in **10.3s**. The variant using Insertion + Swap Within Routes reached a solution value of **197** in a faster **7.8s**.

Hence depending on the regarded tradeoff between solution time and objective value, we can consider either of these composite neighbourhood search operators, as they are better in either objective value or speed compared to the standalone search operators, though the difference is marginal for small problem sizes.

Search Operator	**Best Objective Value**	**Time to Complete 50,000 Iterations (s)**
Insertion + Swap Within	197	7.8
Insertion + 2-Opt between vehicles	174	10.35

Table 3: Performance Comparison of Composite Neighbourhood Search Operators

4.6 Neighborhood Operator Selection Procedure

4.6.1 Exhaustive Selection

We did not go with exhaustive selection as the larger neighbourhood would mean that we would need to first repair more solutions in the neighbourhood before selecting one, which is computationally expensive.

4.6.2 Probabilistic Selection

Suppose we have a few neighbourhood (search operator) functions, $f_1, f_2, \ldots, f_k$.
We define the probability of selecting any neighborhood function f_i as:

$$\mathbb{P}(f_i) = \frac{1}{k}, \quad \forall i = 1, 2, \ldots, k$$

At each iteration t, a neighborhood is selected uniformly at random, which we then apply as a *small perturbation* to the current solution to form our new candidate solution. This is somewhat similar to idea of first descent.

4.7 Basic Feasible Solution (BFS)

4.7.1 Trivial method

Based on the number of vehicles available, randomly allocate a roughly equal number of clients to each vehicle. If any of the vehicles have an infeasible route, we repair the solution by inserting chargers along the route, using the repairing strategy.

An example of what such an allocation might look like is shown in Figure 4:

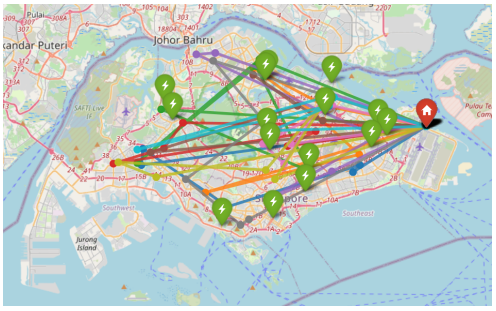


Figure 4: Basic Feasible Solution

4.7.2 K-Means Clustering Initialization

To improve the quality of our initial solution, we used the K-Means clustering algorithm to group client nodes based on their geographical proximity. The number of clusters k was set equal to the number of vehicles available.

Each vehicle was then assigned to serve one cluster, and a feasible route was generated by visiting the clients within that cluster. If a route violated any of the battery or time constraints, it was subsequently repaired by inserting charging stations, similar to the trivial method.

The idea behind this method is to assign each vehicle to serve nearby clients which results in more compact routes. It aims to reduce overall travel distance and inter-region traversal.

Comparison of Initial Solutions:

Initialization Method	**Total Distance (km)**
Random Assignment	∼900
K-Means Clustering	**409.23**

Table 4: Comparison of Initial Objective Function Values

As shown in the table, the K-Means initialization resulted in more than a 50% reduction in total distance compared to the randomly generated solution. This demonstrates the effectiveness of spatial clustering in producing a strong initial solution, which can serve as a better starting point for metaheuristic optimization.

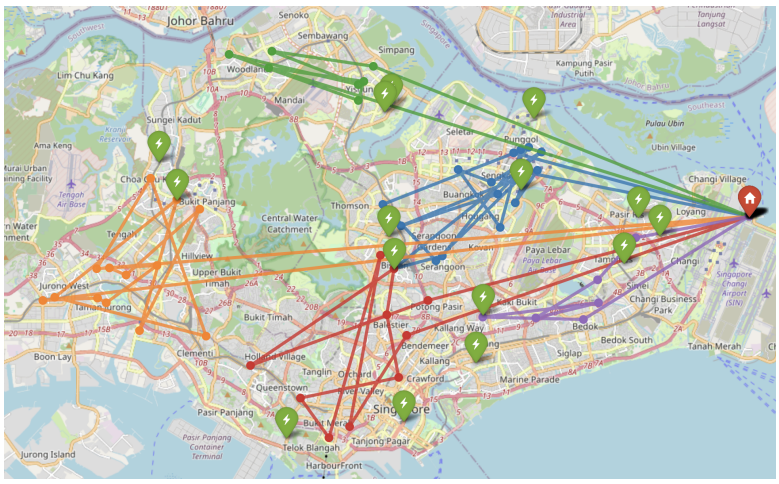


Figure 5: Initial Solution Using K-Means Clustering

However we had an issue when we tried to integrate the K-Means Clustering algorithm into our simulated annealing algorithm. Initially, the solution was good because of the K-means-based BFS, yet the algorithm quickly transitioned to a much poorer solution, suggesting that it did not make use of the good starting solution at all. To fix this, we came up with 3 methods to address the issue.

1. Moving the cooling schedule ahead by about 15%. The rationale for this is that the K-Means BFS behaves as if we've run the standard algorithm using a trivial BFS for around 15,000 iterations. Hence, by adjusting the cooling schedule, we can keep the good starting solution. Let T_0 be the initial temperature, N be the maximum number of iterations in our SA, and C be our cooling rate. Our cooling schedule thus becomes:

$$T = \frac{T_0}{1 + (1 - C) \cdot (i + 0.15N)}$$

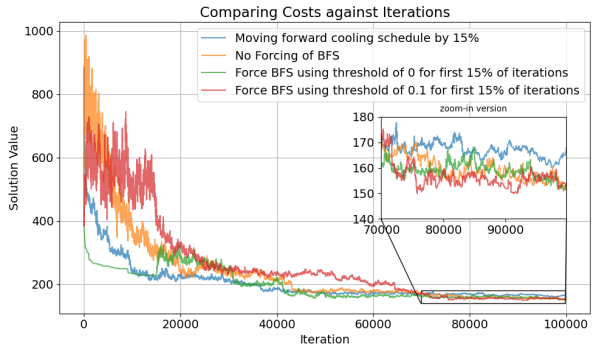
2. Setting the probability for accepting non-improving solutions for the first 15% of iterations to be 0. This causes the algorithm to behave like a local search for the first 15% of iterations. Mathematically, if our threshold is expressed as $g(i)$ where i is the iteration number, we have:

$$g(i) = \begin{cases} 0 & \text{if } i < 0.15N \\ e^{\frac{-\Delta E}{T}}, & \text{otherwise} \end{cases}$$

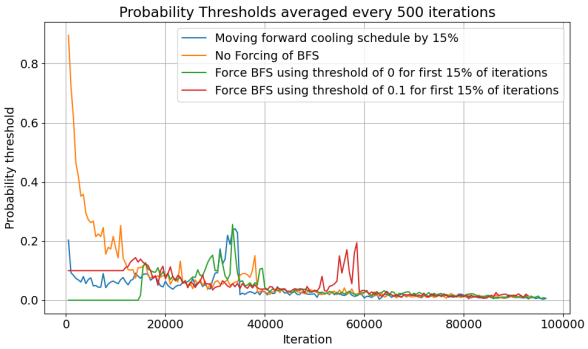
3. Setting the probability for accepting non-improving solutions for the first 15% of iterations to be around 0.1. This causes the algorithm to behave in a more conservative manner initially, akin to having less exploration and more exploitations around the good starting solution.

$$g(i) = \begin{cases} 0.1, & \text{if } i < 0.15N \\ e^{\frac{-\Delta E}{T}}, & \text{otherwise} \end{cases}$$

To verify each of the methods, we could plot the threshold of the acceptance probabilities of non-improving solution $e^{\frac{-\Delta E}{T}}$, as shown in the Figure 6 below.



(a) Solution Values



(b) Acceptance Probabilities

Figure 6: Comparing Methods for preserving K-Mean BFS

From Figure 6, we can infer that the method "Moving forward cooling schedule by 15%", is fairly effective as it achieves the lowest objective value of 210 within 20000 iterations. However, as the number of iterations increases, the performance gap narrows, with the "No Forcing of BFS" method (which uses a trivial starting solution) catching up to methods that begin with a better initial solution. That being said, for larger problem sizes, the K-Means method to achieve a good starting solution might be effective in reducing computational time.

4.8 Constraint Handling Techniques

One of the main constraints which we are concerned about is (6), ensuring that our delivery vehicle does not run out of battery along the route.

4.8.1 Repairing Strategy

We repair the infeasible solutions by inserting chargers. However, it is difficult to determine exactly when the charger should be inserted. One approach is for a vehicle to look for a charger once a critical threshold B_{min} is reached:

$$f_{\text{charge}}(b_i) = \begin{cases} 1 & \text{if } b_i \leq B_{\min} \\ 0 & \text{otherwise} \end{cases}$$

- where b_i is the current battery level for a vehicle i

This approach is implemented in Listing 2, as part of our feasibility function. However, the drawback with this approach is that there are cases that a car might reach a destination, only to realise no nearby chargers are within reach. One way is to simply hard-code a safety buffer that is above 0, as we have done.
However, these two approaches might cause a vehicle to miss a charger that is nearby *(and en-route)* and instead find a charger far away. Instead, another strategy is to charge probabilistically where the parameter α can be tuned to change the probability of charging to decay more or less rapidly based on battery level.

$$\mathbb{P}_{\text{charge}}(b_i) = \begin{cases} 1 & \text{if } b_i \leq B_{\min} \\ \exp\left(-\alpha \cdot \dfrac{b_i - B_{\min}}{B_{\max} - B_{\min}}\right) & \text{if } B_{\min} < b_i < B_{\max} \\ 0 & \text{if } b_i = B_{max} \end{cases}$$

One issue we discovered is that over time, introducing chargers into the solution tends to pollute the route with excess chargers. In subsequent iterations, a neighborhood operator might swap client locations between vehicles, which could result in those chargers no longer being necessary. Therefore, we need an additional piece of code to remove any redundant chargers.

We first keep track of the current charge q_k for a vehicle k. If the next location is a charger C, then we calculate the distance from current location to location after the charger. If the energy required to cover the distance is possible with the current battery level, the charger is removed. The code is shown in Listing 3 in Appendix.

Having mentioned these strategies, there is also another potential strategy that we could implement; the Look Ahead strategy which we detailed when solving ACO in Section 5.1.

4.8.2 Preserving Strategy

This might involve having search operators which swap locations in such a way that the constraints are feasible. However, we felt that this was too computationally expensive. Hence, we left it out.

4.8.3 Penalising Strategy

Not all constraints are of equal importance, hence a penalising strategy could be considered. Specifically, a weighted penalty function can be applied. To illustrate, slightly exceeding the time constraint given to each vehicle is less critical than violating the battery constraint.
Therefore, the penalty for time violations should be lower than that for battery constraint violations. This might involve picking the optimal λ value such that good but infeasible solutions are not discarded.

Choosing the right λ value is therefore crucial. If λ is too high, the algorithm may reject infeasible solutions that are close to being feasible with minor adjustments. Conversely, if λ is too low, the algorithm may become too lenient, resulting in poor-quality solutions.

There are numerous ways to select an appropriate λ value, one of them being trial-and-error. In particular, tuning using grid search can be employed.
This method involves manually trying a range of λ values (e.g., 10, 100, 500) and evaluating the resulting solution quality, feasibility rate, and convergence. Results can be visualized through a heatmap showing objective values versus constraint violations across λ values.
However, since grid search is exhaustive, it can become computationally expensive, which is why we ultimately chose not to pursue the penalising strategy in our implementation.

Alternatively, adaptive penalty adjustment could be used. In this approach:

- If a constraint is violated too often, increase its λ value slightly.
- If a constraint is rarely violated, reduce its λ gradually.

This ensures that λ dynamically reflects the difficulty of satisfying each constraint as the solution space evolves. Adaptive adjustment has several advantages over grid search: it is non-exhaustive, scalable, and robust. While it may not yield the globally optimal λ values, it helps the algorithm converge on feasible solutions more effectively. Its adaptability is particularly advantageous in dynamic contexts such as ours, where delivery routes and constraints are constantly changing.

4.8.4 Progressive Constraint Tightening

An issue arose when we realized that having fairly tight constraints at the start of the problem could cause our program to get stuck, with the initial BFS. At times, the number of iterations per second which normally hovered around ($1000 - 3000$ it/s) dropped to (< 500 it/s), as the algorithm was stuck in attempting to repair the solution. Suppose a hypothetical delivery operator sets the initial conditions for the problem as such:

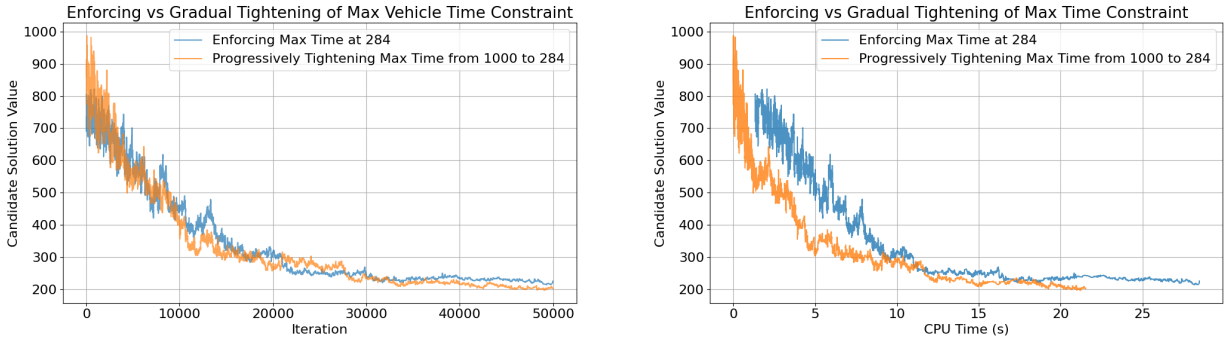
- CARS DEPLOYED = 5
- MAX Time Taken for a vehicle = 284

- CLIENT LOCATIONS = 50

Unfortunately, the delivery operator would not know if these constraints are too tight or too loose for the problem, and this could cause the algorithm to become stuck. Thus our solution to this problem was to tighten the constraints gradually as the number of iterations increased. To illustrate this, suppose we would like to enforce the "Max Time Taken" constraint - $h(i)$. However, as we are unsure of its degree of tightness for the problem, we can then set an arbitrary upper $H_{\max}$ and lower bound $H_{\min}$ for the right hand side of this constraint. The lower bound forms the ideal value that we want the constraint to eventually achieve. Thereafter, we tighten $h(i)$ as such:

$$h(i) = \begin{cases} H_{\max} - \left(\dfrac{H_{\max} - H_{\min}}{0.1 \cdot N}\right) \cdot i, & \text{if } i < 0.1N \\ H_{\min} & \text{otherwise} \end{cases}$$

One thing of note here, is that we have specified the constraint to tighten within the first 10% of iterations, however, this choice is arbitrary. We can verify the effectiveness of this method in the plots shown in Figure 7.



(a) Solution Value against Iterations

(b) Solution Value against CPU Time (s)

Figure 7: Performance of Progressive Constraint Handling

From Figure 7, we can see that progressively enforcing constraints is shown to reduce the CPU Time Taken. For instance to complete 50000 iterations, the enforcing constraint method takes 28s to achieve a solution value (distance) of 214, compared to the much shorter 21s to reach a lower minimum distance of 197 for the Progressive method. Notably, the enforcing method takes up to 1s to find a feasible neighbour, indicating the tightness of the constraint.

4.9 Simulated Annealing

4.9.1 Temperature Cooling Function T(i) - Geometric Series

To reduce unnecessary computations, we update the temperature once every 100 iterations. The temperature function we have chosen follows a geometric series:

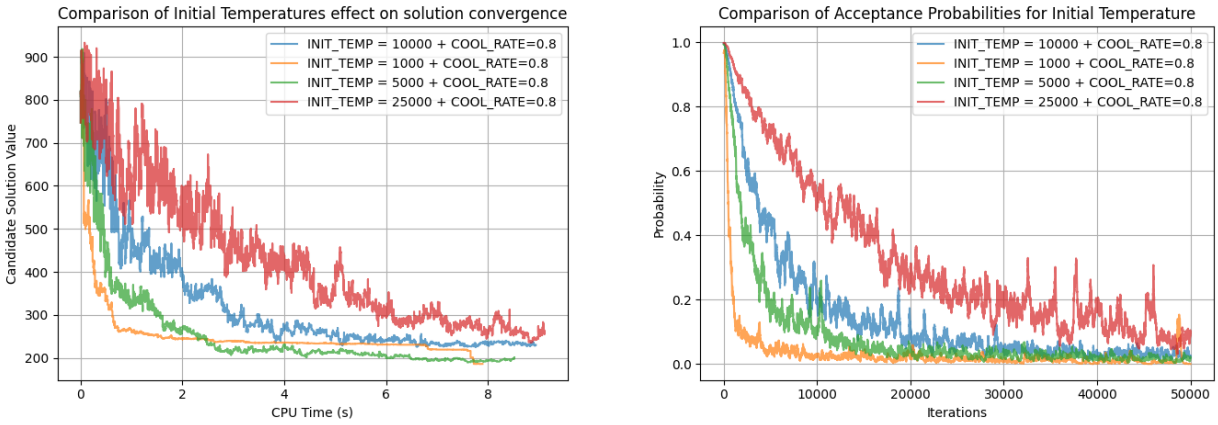
$$T(i) = \frac{T_0}{1 + (1 - C) \cdot i}$$

where

- C - is the cooling rate such that $0 < C < 1$. A example rate used is $C = 0.9$
- i - is the current iteration number
- T_0 - the initial temperature, set to 10,000°C

4.10 Tuning of the Hyperparameters

We first kept the problem size constant at 50 customers, 5 vehicles, 10 charging stations.

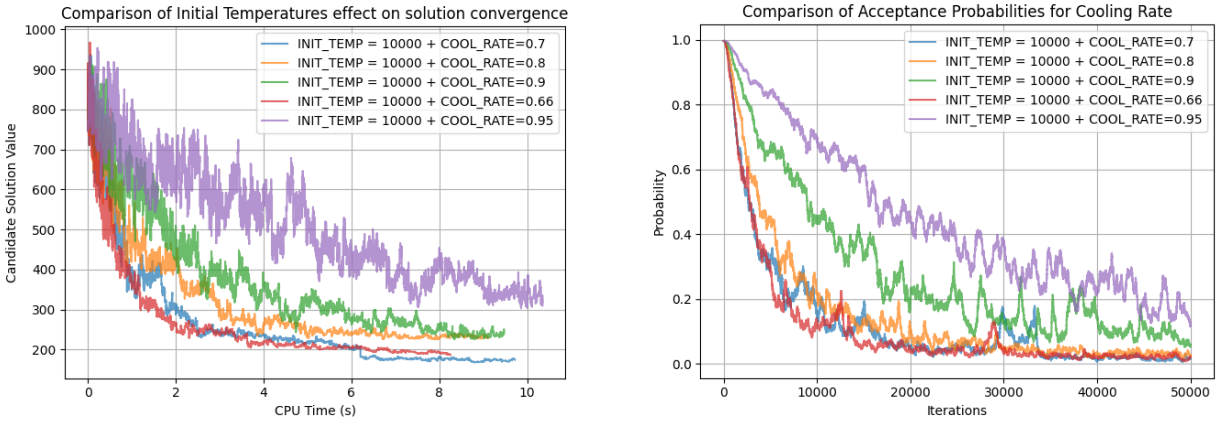


(a) Solution Values (b) Acceptance Probabilities

Figure 8: Comparison of different Initial Temperature T_0

We keep all other parameters constant, including random seed. When initial temperature is set to 10,000, there is a trade-off between exploration and exploitation: the acceptance probability of worse solutions drops below 0.2 after approximately 10,000 iterations as shown in Figure 8:, which allows for convergence while still allowing for escaping local optima.

On the other hand, while reducing the initial temperature T_0 to 5,000 or 1,000 yields a slightly lower candidate solution value, it may also increase risk of premature convergence and limit exploration of the search space.



(a) Solution Values (b) Acceptance Probabilities

Figure 9: Comparison of different Cooling Rates C

Similarly, the cooling rate affects the exploration and exploitation amounts for the SA algorithm. We decided to settle on the cooling rate of 0.7 as it gives the lowest candidate solution value. However, if our size of the initial problem changes, this cooling rate will likely vary.

5 Metaheuristic – Ant Colony Optimisation

Ant Colony Optimisation (ACO) presents an appealing metaheuristic approach due to its independence from traditional search operators and its ability to initiate without requiring a feasible starting solution.

Unlike algorithms such as Simulated Annealing or Local Search, ACO constructs solutions from scratch by probabilistically selecting paths based on two key factors:

- **Pheromone trails**, which encode learned desirability from previous iterations;
- **Heuristic visibility**, defined as the inverse of distance for our problem, guiding exploration.

The probability p_{ij} of an ant moving from node i to node j is given by:

$$p_{ij} = \frac{[\tau_{ij}]^\alpha \cdot [\eta_{ij}]^\beta}{\sum_{k \in \mathcal{N}_i} [\tau_{ik}]^\alpha \cdot [\eta_{ik}]^\beta}$$

where:

- τ_{ij} is the pheromone level on edge (i, j);
- η_{ij} is the visibility (e.g., $1/\text{distance}_{ij}$);
- α controls the influence of pheromone;
- β controls the influence of visibility;
- $\mathcal{N}_i$ is the set of feasible next nodes from node i.

This structure allows ACO to explore the solution space in a decentralized and adaptive manner, reducing sensitivity to the quality of any initial configuration. Furthermore, due to its semi-greedy nature—favouring edges with high visibility—ACO tends to produce reasonably good solutions even in the early iterations, before pheromone information has been reinforced.

5.1 Alternative Method for Battery Constraint Handling

During the implementation of the ACO, we encountered the same challenge as our Simulated Annealing (SA) approach: battery constraint violations. Our initial method checked whether the vehicle had sufficient battery to move from the current node to the next node. If not, the vehicle would attempt to go to the nearest charger before continuing its route. However, this was flawed — there exist the possibility that there is insufficient battery to even reach a charging station.
To address this, we implemented a **look-ahead rule**. Before deciding to move to the next customer node, the algorithm checks whether, after reaching that node, the vehicle would have enough battery to travel to any charging point or return to the depot. If this condition is not met, the vehicle detours to a charger *from the current node* instead. This look-ahead mechanism prevents vehicles from becoming stranded and significantly improves feasibility in our solutions.

5.2 Advantages of ACO in Our Problem Context

1. **Fast Early Convergence.** ACO consistently produced "good-enough" solutions faster than both SA and Gurobi across different problem scales. This is particularly valuable in time-sensitive real-world scenarios. Figure 11 illustrates ACO's early convergence advantage compared to the other methods.

2. **Interpretability.** The behavior of ACO is guided by interpretable hyperparameters:

   - α: influences reliance on pheromone trails (exploitation).
   - β: determines the weight of heuristic visibility (e.g., 1/distance).
   - ρ: evaporation rate controlling memory retention.

   These parameters provide clear rationale for how and why certain paths are favored in the solution, unlike SA where randomness and search operators often dominate decision-making.

5.3 Limitations of ACO

1. **Parameter Sensitivity.** ACO performance is highly sensitive to hyperparameter settings. Fine-tuning parameters like α, β, Q, and ρ is essential but computationally expensive. In contrast, SA is generally less sensitive to exact parameter values and allows flexibility in exploration strategies via different move operators and cooling schedules.

2. **Risk of Premature Convergence.** If the pheromone update mechanism is not well-balanced (e.g., overly strong reinforcement or low evaporation), the algorithm may converge too early to suboptimal paths.

5.4 Parameter Tuning for ACO

To maintain simplicity and interpretability, we avoided exhaustive grid search, which becomes computationally expensive as the problem scales. Instead, we adopted a **random search** approach for hyperparameter tuning.
While random search does not guarantee the discovery of globally optimal parameters, it offers a lightweight and scalable alternative that enables us to:

- Identify **decent parameter configurations** that yield good solutions within a short time,
- **Reuse robust parameter sets** across different problem sizes with minimal adjustments,
- Avoid overfitting parameters to a specific instance.

This pragmatic approach strikes a balance between tuning quality and computational efficiency.
We ran the ACO optimisation for 20 random trials, with each trial limited to a maximum of 60 seconds. The best parameter set was selected based on the lowest total distance found. Each trial sampled parameters as shown in the simplified Python snippet below:

```python
def sample_hyperparams():
    num_cust = len(customer_nodes)
    return {
        "ALPHA":      np.random.uniform(0.1, 2.0),
        "BETA":       np.random.uniform(1.0, 5.0),
        "RHO":        np.random.uniform(0.05, 0.4),
        "Q":          np.random.uniform(0.1, 3.0),
        "NUM_ANTS":   np.random.randint(max(10, num_cust // 2), num_cust + 1),
        "ELITE_RANK": np.random.randint(1, 4)
    }
```

Listing 1: Sampling random ACO parameters for tuning

This randomized sampling enabled us to discover a good balance between exploration and exploitation without the need for exhaustive search.

6 Performance Benchmark

For the performance benchmark, we set the settings as shown below for both metaheuristics algorithms. Because it is not possible to have the exact hyperparameters for both SA and ACO in terms of the tuning (exploration vs exploitation), we instead aimed for similar problem parameters, as shown in Table 5 below.

6.1 Small Scale Problem

Param	Val
Customers	10
Chargers	2
Vehicles	2
Batt Cap	10
Energy/km	0.187
Time/km	3
Time Limit/veh	300
Seed	5

Table 5: Simulation Parameters

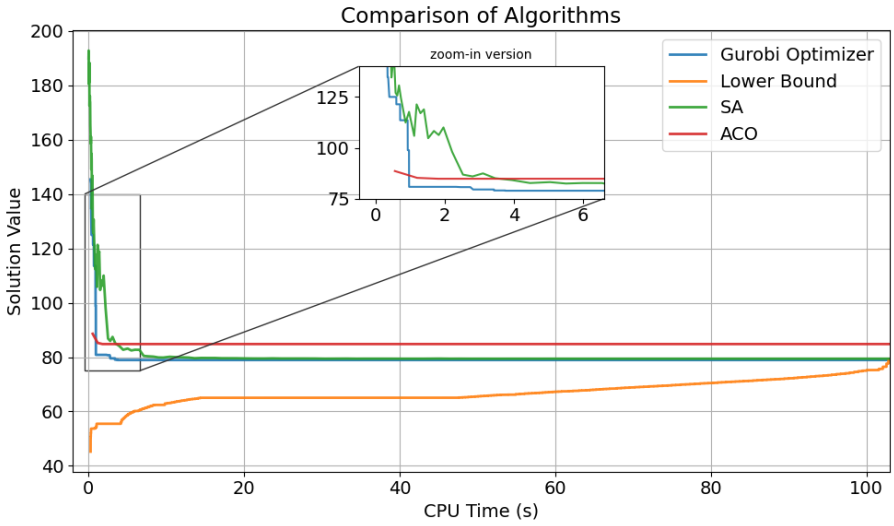


Figure 10: Comparison of Metaheuristic Algorithms [a]

[a] CPU Time(s) may differ from 2.3 due to PC differences in benchmarking

For this problem size, the exact method is fast, while SA and ACO produce comparable results—with SA achieving a minimum distance of 79.40km which is close to the optimal distance of 78.92km.

6.2 Medium Scale Problem

Param	Val
Customers	15
Chargers	5
Vehicles	3
Batt Cap	10
Energy/km	0.187
Time/km	3
Time Limit/veh	300
Seed	5

Table 6: Simulation Parameters

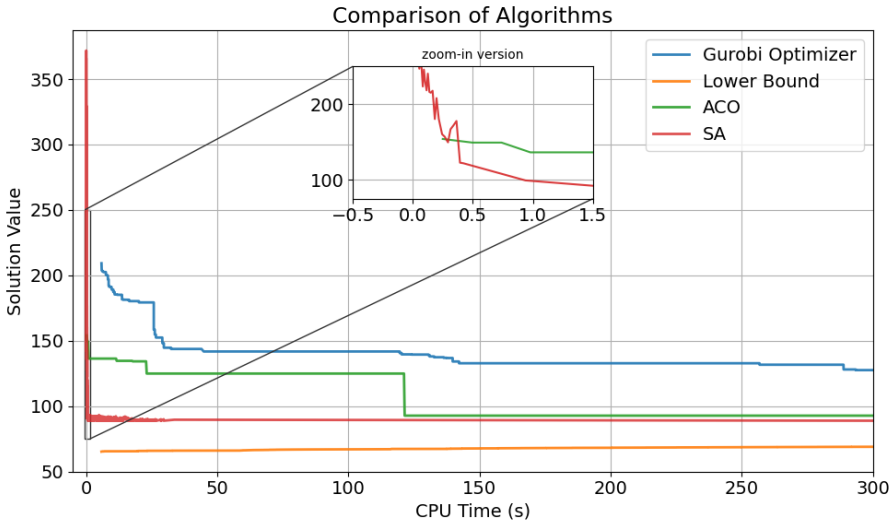


Figure 11: Comparison of Metaheuristic Algorithms

From the graph, it is evident that both ACO and SA achieve a better result in less CPU Time compared to the Gurobi Optimizer, even for a moderately sized problem.

6.3 Larger Scale Problem

Param	Val
Customers	**50**
Chargers	10
Vehicles	5
Batt Cap	10
Energy/km	0.187
Time/km	3
Time Limit/veh	600
Seed	5

Table 7: Simulation Parameters

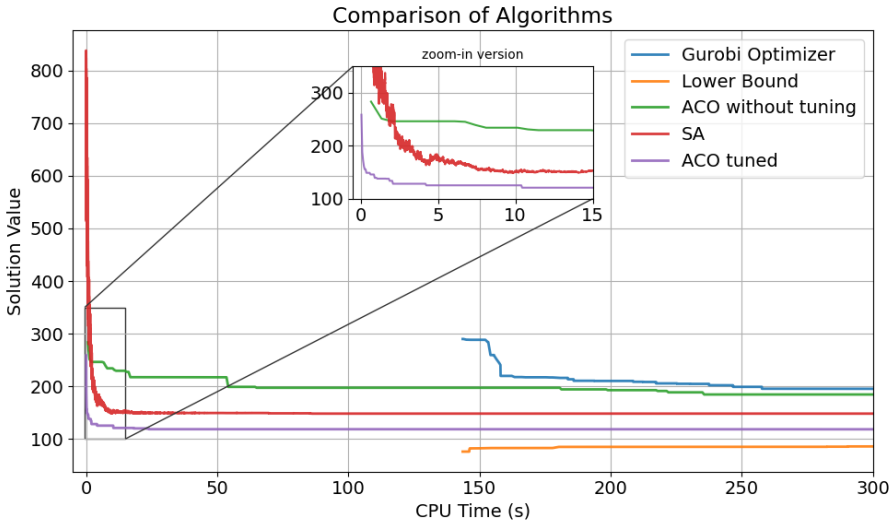


Figure 12: Comparison of Metaheuristic Algorithms

Tuned ACO converges significantly faster than SA, reaching high-quality solutions within the first few seconds. Although SA shows gradual improvement over time and performs better than the untuned version of ACO, it still lags behind the fine-tuned ACO in both speed and final solution quality.

This results emphasises the importance of hyperparameter tuning for ACO. Without proper tuning, ACO does not perform to its full potential. The tuning process used to obtain these parameters is detailed in Appendix (Figure 15).

6.3.1 Forcing Time Constraint of 300min/vehicle

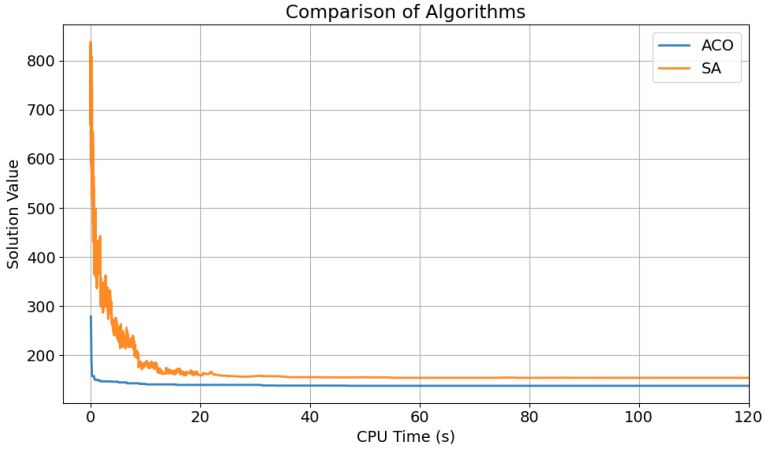


Figure 13: SA and ACO peformance for 300min/vehicle

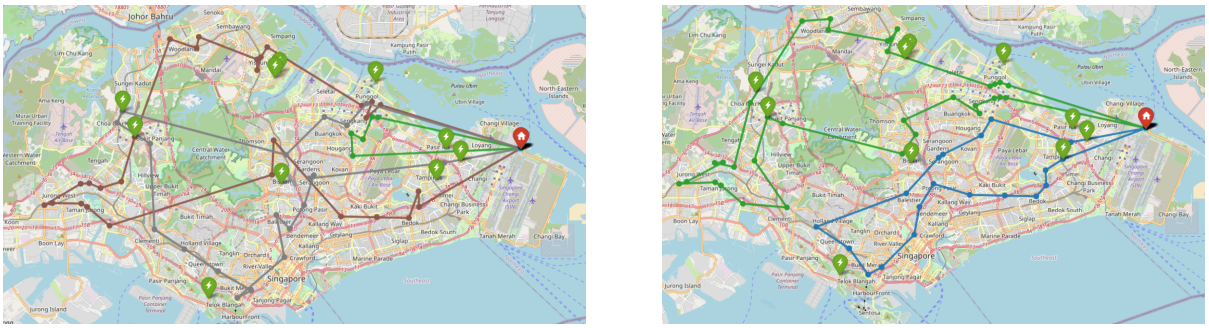
Previously, our solution used only a single vehicle, which does not reflect a realistic fleet routing scenario. To better simulate practical conditions, we imposed a time constraint of 300 minutes per vehicle. Under this constraint, a single vehicle can no longer serve all customers feasibly, requiring multiple vehicles.

When solved using Gurobi, the model required 12,815 seconds (3.5 hours) just to identify an integer feasible solution of objective value 471.87, with a large optimality gap of 81.3% (refer to Appendix, Listing 8). Even after 21,600 seconds (6 hours), the best objective value achieved was 157.7 with a gap of 44%.

In contrast, our metaheuristic approaches were significantly more efficient. ACO achieved a solution value of 150.24 within the first second, while SA reached an objective value of 160 in roughly 20 seconds. This demonstrates that as problem complexity increases, the advantage of metaheuristic algorithms becomes increasingly pronounced.

7 Insights from the Optimal Solution

Here is a routing solution found using metaheuristics, with 50 customers, and 5 electric delivery vehicles. The near-Optimal solution in Figure 14 suggests that only 2 vehicles are needed to match all the constraints, to minimise electricity running costs, supposing time limit of 600 mins.



(a) Time Limit: 280

(b) Time Limit: 600

Figure 14: Comparison of two near-optimal solutions

An insight that we have gained is that the problem is greedy in nature. Most of the time, the optimal solution obtained is such that the problem is solved trivially using just one vehicle. If we allow a smaller time frame of 280 minutes instead of 600 minutes, we will start to utilise more vehicles, showcasing the greedy nature of the problem. To model a real world scenario where there might be possible risks associated with only using 1 or 2 vehicle, other constraints might be needed to reflect this.

Furthermore, we observe that vehicles seldom visit charging stations more than once over their journey. This could be due to a lack of distinction between complete or partial charging and the associated costs that comes with charging. Moreover, the current distance matrix assumes straight-line travel between locations, yielding infeasible paths through inaccessible areas (i.e Bukit Timah Reserve) as shown above. For feasibility, our energy and distance matrix should ideally be adjusted with realistic topologies.

8 Assumptions

- Battery discharge rate is directly proportional to distance travelled.
- Time taken is directly proportional to distance travelled (assuming constant average speed).
- Euclidean distance is used to simplify distance calculations between locations.
- Every vehicle is operational.
- Charging time is negligible.
- There is no restriction on the delivery capacity of an electric delivery vehicle; this can be added to increase complexity.
- Customers do not need to receive their parcels quickly (there is no time constraint for each customer)

- No traffic delays or road closures are considered.

References

DataGov (2017). *HDB Locations*. URL: https://github.com/JackFongNew/Singapore-HDB-Resale-Price-Prediction/blob/main/dataset/resale_hdb_price_2017_onward_updated9.csv (visited on 06/01/2025).

Datamall, LTA (Feb 2025). *Electric Vehicle Charging Network*. URL: https://datamall.lta.gov.sg/content/dam/datamall/datasets/Facts_Figures/Electric-Vehicle-Charging-Network/Electric_Vehicle_Charging_Points_Feb%202025.xlsx (visited on 06/01/2025).

Fei, Liu et al. (2025). *Heuristics for Vehicle Routing Problem: A Survey and Recent Advances*. URL: https://arxiv.org/pdf/2303.04147 (visited on 07/01/2025).

Appendix

```python
def is_route_feasible(route, locations):
    """
    (Does not use our solution encoding)
    Instead, it checks if a single route is feasible by inserting charging stations if necessary.

    function returns:
        - New route with chargers inserted if feasible.
        - False if no feasible charger insertion exists.
    """
    battery = BATTERY_CAPACITY
    new_route = [route[0]]
    total_time_taken = 0

    for i in range(1, len(route)):
        prev, curr = new_route[-1], route[i]
        to_discharge = energy_matrix[prev][curr]

        # if desired destination uses too much battery (with some safety buffer)
        if to_discharge > battery - random.uniform(1, 5):
            # Find nearest charger that is reachable from prev and can reach curr
            inserted = False
            # find nearest charger using distance matrix
            charger = np.argmin(distance_matrix[prev][1+NUM_CUSTOMERS:], axis=0) + 1 + NUM_CUSTOMERS

            if curr < 1+NUM_CUSTOMERS: # next destination is not a charger
                d1 = energy_matrix[prev][charger]
                d2 = energy_matrix[charger][curr]

                if d1 <= battery and d2 <= BATTERY_CAPACITY:
                    new_route.append(charger)
                    new_route.append(curr)
                    total_time_taken += time_matrix[prev, charger] + time_matrix[charger, curr]
                    battery = BATTERY_CAPACITY-d2
                    inserted = True

            else: # next destination is a charger
                # find a nearer charger
                d1 = energy_matrix[prev][charger]

                if d1 <= battery:
                    new_route.append(charger)
                    total_time_taken += time_matrix[prev, charger]
                    battery = BATTERY_CAPACITY
                    inserted = True

            if not inserted:
                return False, "battery"  # infeasible
        else:
            battery -= to_discharge
            new_route.append(curr)

            # add time taken to travel
            total_time_taken += time_matrix[prev, curr]

            if curr >= 1 + NUM_CUSTOMERS:
                battery = BATTERY_CAPACITY

    if total_time_taken > TOTAL_TIME_ALLOWED: # exceeded total time to complete
        return False, "time"
```

Listing 2: Repairing solution with EV Charging

The code below is actually integrated as part of our `is_route_feasible()` function above.

```
optimized_route = [new_route[0]]
battery = BATTERY_CAPACITY
for i in range(1, len(new_route)):
    prev, curr = optimized_route[-1], new_route[i]
    to_discharge = energy_matrix[prev][curr]

    # If curr is a charger, check if we can skip it
    if curr >= 1 + NUM_CUSTOMERS and i + 1 < len(new_route):
        next_node = new_route[i + 1]
        to_next = energy_matrix[prev][next_node]
        if to_next < battery:
            # Skip this charger
            continue

    battery -= to_discharge
    if curr >= 1 + NUM_CUSTOMERS:
        battery = BATTERY_CAPACITY
    optimized_route.append(curr)

return optimized_route
```

Listing 3: Removing Excess Chargers

```
def swap_routes(routes):
    """
    Swap client locations between two vehicles.
    Takes in a nested array (our solution encoding)
    """
    new_routes = [r[:] for r in routes]

    not_charger_or_depot = False

    # prevent swapping of chargers, only client locations
    while not_charger_or_depot == False:
        # choose 2 vehicles to swap routes
        veh_1, veh_2 = random.sample(range(len(new_routes)), 2)

        # randomly choose two client locations of the vehicles to swap
        i = random.randint(0, len(new_routes[veh_1])-1)
        j = random.randint(0, len(new_routes[veh_2])-1)

        if 0 < new_routes[veh_1][i] < 1 + NUM_CUSTOMERS and 0 < new_routes[veh_2][j]
    < 1 + NUM_CUSTOMERS and new_routes[veh_1][i] != new_routes[veh_2][j]:
            not_charger_or_depot = True

    new_routes[veh_1][i], new_routes[veh_2][j] = new_routes[veh_2][j], new_routes[
    veh_1][i]
    return new_routes
```

Listing 4: Swap Operator

```
def tweak_insert(routes):
    """
    Remove customer from one vehicle and insert to another vehicle
    Takes in a nested array (our solution encoding)
    """
    new_routes = [r[:] for r in routes]
    veh_1, veh_2 = random.sample(range(len(new_routes)), 2)

    # Get all valid customer indices in veh_1 (not depot or charger)
    customer_indices = [i for i in range(len(new_routes[veh_1]))
                        if 0 < new_routes[veh_1][i] < 1 + NUM_CUSTOMERS]

    # Retry until we find a vehicle with at least 1 customer
    while not customer_indices:
```

```
        veh_1, veh_2 = random.sample(range(len(new_routes)), 2)
        customer_indices = [i for i in range(len(new_routes[veh_1]))
                            if 0 < new_routes[veh_1][i] < 1 + NUM_CUSTOMERS]

    # choose up to 3 customers
    num_to_move = min(3, len(customer_indices))
    from_indices = sorted(random.sample(customer_indices, num_to_move), reverse=True)

    customers = []
    for idx in from_indices:
        customers.append(new_routes[veh_1].pop(idx))

    # ensure veh_2 route has basic structure if no client locations left
    # aka vehicle becomes inactive
    if len(new_routes[veh_2]) < 2:
        new_routes[veh_2] = [0, 0]

    # insert each customer at a random position in veh_2
    for customer in customers:
        insert_pos = random.randint(1, len(new_routes[veh_2]) - 1)
        new_routes[veh_2].insert(insert_pos, customer)

    return new_routes
```

Listing 5: Insert Operator

```
def k_opt_within(routes, k=2): # check again
    """
    Performs K-Opt within a single vehicle's route
    """
    assert k in (2,3) # function oni allows 2-opt or 3-opt

    new_routes = [r[:] for r in routes]

    # exclude depot at start and end
    # vehicle route is long enough to k-opt on
    vehicle_choices = [i for i, route in enumerate(new_routes) if len(route)-2 >= k]

    veh = random.choice(vehicle_choices)
    route = new_routes[veh]

    # find customers indices to k-opt on
    customer_indices = [i for i in range(1, len(route) - 1)
                        if 0 < route[i] < 1 + NUM_CUSTOMERS]

    if len(customer_indices) < k:
        return new_routes

    # List of points for cutting
    selected = sorted(random.sample(customer_indices, k))

    if k == 2:
        i, j = selected # Choose 2 points to cut so that i < j
        route[i:j+1] = reversed(route[i:j+1])
    elif k == 3:
        i, j, l = selected # Choose 3 points to cut so that i < j < k

        option = random.choice([1, 2, 3])
        if option == 1:
            route[i:j+1] = reversed(route[i:j+1])
        elif option == 2:
            seg1, seg2 = route[i:j+1], route[j+1:l+1]
            route[i:l+1] = seg2 + seg1
        elif option == 3:
            route[i:l+1] = reversed(route[i:l+1])

```

```
    new_routes[veh] = route
    return new_routes
```

Listing 6: K Opt Within

```
def k_opt_between_vehicles(routes, k=2):
    """
    Performs K-Opt between 2 vehicles
    """
    assert k==2 # function oni allows 2-opt

    new_routes = [r[:] for r in routes]

    eligible_vehicles = [i for i, r in enumerate(new_routes) if len(r) > 2]

    if len(eligible_vehicles) < 2:
        return new_routes

    veh1, veh2 = random.sample(eligible_vehicles, 2)
    route1, route2 = new_routes[veh1], new_routes[veh2]

    cust1 = [i for i in range(1, len(route1) - 1)]
    cust2 = [i for i in range(1, len(route2) - 1)]

    if not cust1 or not cust2:
        return new_routes

    i = random.choice(cust1)
    j = random.choice(cust2)

    if k == 2:
        len1 = min(2, len(route1) - 2)
        len2 = min(2, len(route2) - 2)
        seg1_start = max(1, i - len1 + 1)
        seg2_start = max(1, j - len2 + 1)
        seg1 = route1[seg1_start:i+1]
        seg2 = route2[seg2_start:j+1]
        route1[seg1_start:i+1] = seg2
        route2[seg2_start:j+1] = seg1

    new_routes[veh1] = route1
    new_routes[veh2] = route2
    return new_routes
```

Listing 7: K-Opt Between Vehicles

A ACO Hyperparameter Tuning Results

```
[ 1/20] dist =   218.36 km  α=1.67  β=2.2  ρ=0.07  Q=2.11  ants=41   elite=2  time= 60.0s
[ 2/20] dist =   200.22 km  α=1.18  β=1.2  ρ=0.09  Q=0.10  ants=31   elite=2  time= 60.0s
[ 3/20] dist =   185.56 km  α=0.70  β=1.2  ρ=0.08  Q=0.25  ants=45   elite=2  time= 60.0s
[ 4/20] dist =   187.54 km  α=0.84  β=1.0  ρ=0.08  Q=0.93  ants=33   elite=1  time= 60.0s
[ 5/20] dist =   189.96 km  α=0.24  β=4.3  ρ=0.09  Q=2.10  ants=48   elite=2  time= 60.0s
[ 6/20] dist =   188.36 km  α=1.46  β=4.0  ρ=0.05  Q=1.41  ants=35   elite=3  time= 60.0s
[ 7/20] dist =   207.68 km  α=0.31  β=1.6  ρ=0.05  Q=2.14  ants=33   elite=1  time= 60.0s
[ 8/20] dist =   219.27 km  α=1.53  β=3.4  ρ=0.06  Q=2.72  ants=29   elite=1  time= 60.0s
[ 9/20] dist =   197.40 km  α=1.50  β=1.9  ρ=0.06  Q=2.79  ants=25   elite=3  time= 60.0s
[10/20] dist =   194.43 km  α=1.15  β=1.8  ρ=0.07  Q=2.06  ants=34   elite=1  time= 60.0s
[11/20] dist =   202.03 km  α=0.96  β=3.8  ρ=0.08  Q=1.76  ants=28   elite=1  time= 60.0s
[12/20] dist =   184.56 km  α=0.51  β=3.0  ρ=0.10  Q=2.85  ants=30   elite=2  time= 60.0s
[13/20] dist =   194.24 km  α=0.16  β=3.0  ρ=0.08  Q=1.28  ants=45   elite=2  time= 60.0s
[14/20] dist =   191.94 km  α=0.72  β=2.9  ρ=0.07  Q=0.89  ants=43   elite=2  time= 60.0s
[15/20] dist =   204.50 km  α=0.81  β=4.9  ρ=0.07  Q=0.71  ants=29   elite=1  time= 60.0s
[16/20] dist =   181.75 km  α=0.69  β=1.7  ρ=0.05  Q=0.45  ants=38   elite=2  time= 60.0s
[17/20] dist =   199.10 km  α=0.21  β=2.0  ρ=0.10  Q=2.46  ants=30   elite=2  time= 60.0s
[18/20] dist =   185.07 km  α=0.37  β=2.5  ρ=0.07  Q=0.16  ants=42   elite=1  time= 60.0s
[19/20] dist =   204.21 km  α=1.90  β=4.2  ρ=0.07  Q=0.82  ants=38   elite=2  time= 60.0s
[20/20] dist =   198.29 km  α=0.77  β=3.9  ρ=0.08  Q=2.32  ants=50   elite=2  time= 60.1s

✅ All results saved to C:\Users\Dylan Gay\Desktop\Work SUTD\HST Project\tuning_results.csv

🏆 Top 5 parameter sets
1. dist=181.75 km | α=0.69, β=1.7, ρ=0.05, Q=0.45, ants=38, elite=2 | time=60.0s
2. dist=184.56 km | α=0.51, β=3.0, ρ=0.10, Q=2.85, ants=30, elite=2 | time=60.0s
3. dist=185.07 km | α=0.37, β=2.5, ρ=0.07, Q=0.16, ants=42, elite=1 | time=60.0s
4. dist=185.56 km | α=0.70, β=1.2, ρ=0.08, Q=0.25, ants=45, elite=2 | time=60.0s
5. dist=187.54 km | α=0.84, β=1.0, ρ=0.08, Q=0.93, ants=33, elite=1 | time=60.0s

⏱️ Total tuning time: 1200.5 s for 20 trials
```

Figure 15: Tuning results of ACO for EV routing across 20 trials.

B Gurobi Log Output for 300min/vehicle Constraint

```
H866099 795329 471.8716997 88.30030 81.3% 41.6 12815s
H866176 795320 453.8763473 88.30030 80.5% 41.6 12911s
H866187 795495 430.8680780 88.30030 79.5% 41.6 12965s
H866239 795488 423.7948237 88.30030 79.2% 41.6 12965s
H866467 795620 401.1214217 88.30030 78.0% 41.6 13130s
H866623 795425 374.1081031 88.30101 76.4% 41.6 13130s
H866723 795136 359.5375815 88.30137 75.4% 41.6 13331s
H867101 792359 309.9985858 88.30137 71.5% 41.6 13735s
H867429 783587 274.3446576 88.30137 67.8% 41.6 13979s
H867462 762197 238.7115451 88.30137 63.0% 41.6 14121s
H868135 742218 221.7433425 88.30190 60.2% 41.6 14523s
H868423 722350 208.2345008 88.30190 57.6% 41.6 15047s
H868772 684880 191.3935821 88.30190 53.9% 41.6 16083s
H869290 671226 185.6935381 88.30190 52.4% 41.6 16567s
H869807 611861 168.8410068 88.30190 47.7% 41.6 18128s
H869990 561412 157.8830548 88.30190 44.1% 41.6 18573s
H880372 570071 157.7108198 88.32330 44.0% 41.8 21300s

Cutting planes:
  Gomory: 75
  Lift-and-project: 22
  Cover: 32
  Implied bound: 185
  Projected implied bound: 9
  MIR: 222
  Mixing: 7
  Flow cover: 660
  GUB cover: 18
  Inf proof: 17
```

```
  Zero half: 51
  RLT: 254
  Relax-and-lift: 418
  BQP: 1

Explored 882000 nodes (36928566 simplex iterations) in 18001.56 seconds (3100.59 work
    units)
Thread count was 16 (of 16 available processors)

Solution count 10: 157.711 157.883 158.171 ... 184.748

Time limit reached
Best objective 1.577108198020e+02, best bound 8.832433004109e+01, gap 43.9960%
Solution found!
Objective value: 157.710819802008
```

Listing 8: Gurobi Solver Log