CONTENTS

Energy Awareness in Flying Base Stations Trajectory Optimization

Abhishek Mahendran Yadav
Department of Engineering
King's College London

Abstract

Unmanned Aerial Vehicles, also known as UAVs, are drones which have many uses such as cargo delivery, surveillance, and agriculture. However, they also have another use which is extremely helpful, and it is that they can extend the range of cell towers and/or increase the speed of the network. This study investigates the factors that affect the energy consumption of UAV-BS, and how different TSP algorithms affect the energy consumption of the UAV, for different models of UAVs. The problem to be solved is finding the optimum path trajectory of the base station depending on the number of nodes and the distance between the nodes. UAVs which do not have enough energy to complete the full tour, the TSP tour will be split into clusters. where UAVs will return to the starting node to recharge and thereafter complete the remaining clusters.

I. Introduction

UAVs have many use cases. A cell tower that is malfunctioning and needs to be repaired is one such instance where UAVs can be used. UAVs are also used when climate catastrophes such as earthquakes or landslides occur, where building infrastructures are not physically possible nor are they economically viable. UAVs can be rapidly deployed, and provide a wide coverage area, and can be easily repositioned to provide coverage where it is needed most. They can operate in harsh and inaccessible environments, providing communication services where other systems are unavailable. In densely populated areas, such as concerts, sport events, or any other densely populated area where there will be more demand for data usage, UAVs can also increase the network speed and cater to the high demand in those areas. The problem is that these UAV-BSs can only be in the air for a limited amount of time before they return to their depot to charge again. The typical flying time for a flying base station is less than an hour.

Since the key challenge to the deployment of Unmanned Aerial Vehicles (UAVs) for communication is the limited energy of the drone. To illustrate the case, energy consumption of various TSP algorithms for different models of drones is performed in order to minimise the energy consumed.

II. Background

NP-hard problems are a class of computational problems that are regarded as some of the most challenging problems to solve. The abbreviation NP stands for non-deterministic polynomial time. Examples of NP-hard

problems include the traveling salesman problem, the knapsack problem, and the satisfiability problem. One thing that all these problems have in common is that the best solution is among to set of a large number of possible solutions. The number of these possible solutions grow rapidly as the size of the problem increases.

To illustrate the difficulty of NP-hard problems, the Knapsack problem can be considered. The Knapsack problem involves selecting a subset of items, each with its own weight and value (or profit), to be put into a knapsack of limited capacity, so that the total value of the selected items is maximized. The problem becomes harder to solve as the number of items and their weights and values increase, and as the capacity of the knapsack decreases [1].

The problem that is required to be solved for energy optimisation for UAVs is essentially the travelling salesman problem, also mathematically known as the Hamiltonian Cycle, Previous studies have shown the effectiveness of multiple TSP heuristics. This study will apply the knowledge of TSP heuristics in the UAV industry to minimise its energy consumption.

In the paper *The Traveling Salesman Problem: A Case Study in Local Optimization*, by David S. Jhonson et al. [2], the researchers concluded local search algorithms such as 2-Opt, 3-Opt, and Lin-Kernighan were the best approaches to solving the TSP. While simulated annealing and genetic algorithms can find better solutions in more time, they do not match the efficiency of local search for this problem. Implementation choices and careful attention to algorithmic tradeoffs can significantly reduce running times. These insights may be valuable for improving performance in UAVs.

III. Methodology

Four well-known algorithms for the TSP will be implemented and their comparisons in performance, i.e. time complexity and its comparison to the lower bound tree. They are the nearest neighbour heuristic, greedy heuristic, christofides algorithm, and simulated annealing. This study will also compare the 2-Opt optimal variations of the algorithms.

All the nodes will be generated randomly on a square plane with sides of 1 kilometer. In order to produce fair results, the Monte Carlo method will be used to compare the distance covered by each of the algorithms; this consists of repeatedly generating nodes with different seeds for multiple trials. This study uses trials in the range of 25-50 trials. Performing more trials will considered to be unnecessary as the data would have already reach convergence. The best TSP heuristic will then be used to compare 3 drones with different specifications.

IV. Candidate TSP Heuristics

A. Brute Force:

The brute force algorithm is by far the most trivial approach to the TSP. It involves iterating through every possible solution and compare the cost for each candidate solution. Although this will find the global optimal solution, this technique is not viable for a large number of nodes. For 30 nodes, the number of total

possible solutions exceed 4.42×10^{23}. This algorithm will not be tested due to the unreasonable amount of time it will take to find the solution after 20 nodes.

Algorithm 1 Brute Force Algorithm

$N \leftarrow$ number of nodes
$min\_path \leftarrow$ None
$min\_distance \leftarrow \infty$
for each permutation of paths p of the cities **do**
  $current\_distance \leftarrow 0$
  for $i \leftarrow 1$ to $N-1$ **do**
    $current\_distance \leftarrow current\_distance + distance(p[i], p[i+1])$
  end for
  $current\_distance \leftarrow current\_distance + distance(p[1], p[N])$
  if $current\_distance < min\_distance$ **then**
    $shortest\_distance \leftarrow current\_distance$
    $min\_path \leftarrow p$
  end if
end for
return min_path,current_distance =0

For any graph G with N nodes, the number of possible Hamiltonian cycles is $\frac{(N-1)!}{2}$. Therefore the time complexity of this algorithm is $O(N!)$.

B. Nearest Neighbour Heuristic:

Nearest Neighbour algorithm is a naive algorithm used to provide a candidate path for the TSP problem. In this problem, the nearest neighbour heuristic starts the UAV at the depot, and moving to the closest, unvisited node as the next destination. This continues until all the nodes have been visited, and the UAV returns to the depot [12].

Since the algorithm only requires comparing distances between adjacent nodes, which can be stored in the distance matrix, this makes it easy to implement and efficient for small and medium-sized instances of the problem.

Algorithm 2 Nearest Neighbour Heuristic

Require: $distance\_matrix[][] \leftarrow$ symmetrical distance matrix

Require: $degree[] \leftarrow$ array which stores degree of nodes

Require: $nodes[][]$ position of nodes

1: $path \leftarrow [1]$, starting from first node

2: $cost \leftarrow 0$

3: **for** $i \leftarrow 2$ to N **do**

4: $\quad j \leftarrow find(min(distance\_matrix(:, i)))$

5: $\quad path(i) \leftarrow j$, finds nearest unvisited node

6: $\quad distance\_matrix(i, j) = distance\_matrix(j, i) = \infty$, setting nodes i and j as visited

7: $\quad cost = cost + distance(nodes(i), nodes(j))$

8: **end for**

9: **return** path, cost =0

Step of Algorithm	Operation	Runtime
1	comparing N nodes to N-1 nodes	$O(N) * O(N-1)) = O(N^2)$
2	updating visited nodes	$O(N)$
3	Total	$O(N^2) + O(N) = O(N^2)$

TABLE I

TIME COMPLEXITY CALCULATION OF NNH.

C. Greedy Heuristic:

Another greedy heuristic has been used to find a sub-optimal solution to the problem. The heuristic works by comparing all edges in the adjacency matrix and connects the nodes which are closest together. If a cycle is formed by joining the nodes, then the next nearest set of nodes will be connected, resulting in a valid TSP tour [2].

Algorithm 3 Greedy Heuristic

Require: $distance\_matrix[][] \leftarrow$ symmetrical distance matrix

Require: $degree[] \leftarrow$ array which stores degree of nodes

Require: $nodes[][]$ position of nodes

1:

2: $cost \leftarrow 0$

3: **for** $i \leftarrow 2$ to N **do**

4: $(j, k) \leftarrow find(min(distance\_matrix(:, :)))$ (connect j and k)

5: $distance\_matrix(j, k) = \infty$, setting edge (j,k) as visited

6: $cost = cost + distance(nodes(i), nodes(j))$

7: **end for**

8: **return** cost, distance_matrix =0

Step	Operation	Runtime
1	search for least distance edge	$O(log(N))$
2	for each edge, check if does not create a cycle	$O(N^2)$
5	Total	$O(log(N)) + O(N) = O(N^2 log(N))$

TABLE II

TIME COMPLEXITY CALCULATION OF GREEDY HEURISTIC. [2]

D. Minimum Spanning Tree

In graph theory, a minimum spanning tree is a tree that covers all the vertices in such a way that the sum of the length of edges (edge weights) are. The problem of finding a minimum spanning tree is a classic optimisation problem and is solved mostly with Prim's or Kruskal's algorithm. The MST has many real-world applications, such as in network design and in transportation engineering [7].

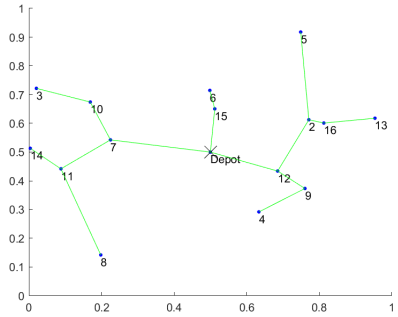

Fig. 1. MST of a graph with 16 nodes

E. Christofides Algorithm

The Christofides Algorithm uses the minimum spanning tree of the graph to result in a TSP tour. The minimum spanning tree can be obtained using Prim's or Kruskal's algorithm. Prim's algorithm [7] works

by connecting neighbours of a current node to the nearest unvisited node, given the adjacency matrix. Then, the number nodes with an odd number must be connected such that when paired, the total distance should be minimised. For the bipartite minimum weight matching step, the branch and bound algorithm [8] is used. One of the amazing properties of the Christofides Algorithm is that the ratio to a christofides solution to the optimal solution for a particular number of nodes will never be greater than 1.5 [10].

Algorithm 4 - Christofides Algorithm

$T \leftarrow$ minimum spanning tree of V

$degrees \leftarrow$ degree of each node in T

$odd\_nodes \leftarrow$ list of nodes in T with odd degree

$distmat\_odd \leftarrow$ distance matrix of the odd degree nodes

$matching \leftarrow$ minimum weight perfect matching edges on $distmat\_odd$

$eulerian\_graph \leftarrow$ list of edges from T

Add edges from $matching$ to $eulerian\_graph$, incrementing the weight if the edge already exists

$path\_eg \leftarrow$ Eulerian path on $distmat\_eg$

Remove any duplicate nodes from $path\_eg$

$path\_eg \leftarrow$ add the first node to the end of $path\_eg$

return $path\_eg$ =0

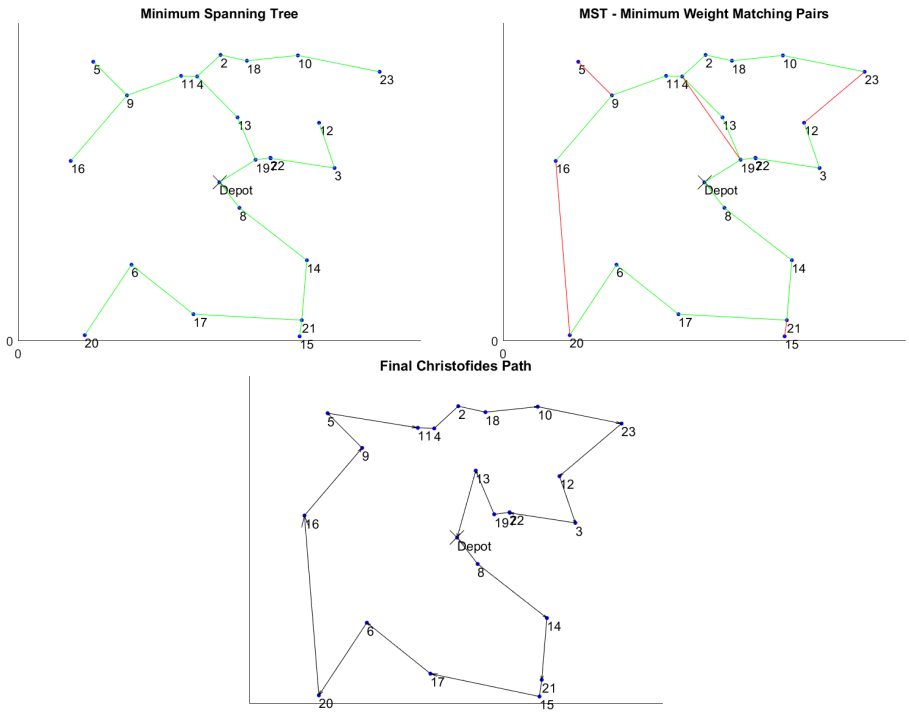

Fig. 2. Visual demonstration of three major steps of the Christofides Algorithm (23 nodes)

Step	Operation	Runtime
1	MST (Prim's Algorithm)	$O(N^2 * log(N))$
2	Finding odd-degree vertices	$O(N)$
3	minimum weight perfect matching	$exponential$
4	Total	$O(N^2 * log(N))$

TABLE III

TIME COMPLEXITY CALCULATION OF CHRISTOFIDES ALGORITHM [10].

2-Opt Algorithm

Unlike the previous algorithms, the 2-Opt Algorithm takes an existing path and improves it. The algorithm works by swapping all combinations of edges that are not consecutive [3]. 3-Opt is similar to 2-Opt except that all combinations of 3 edges are swapped. 3-Opt and k-Opt will not be compared because they were expensive to implement.

Algorithm 5 - 2-Opt

$mincost \leftarrow$ getCost($nodes$, T)
$minpath \leftarrow T$
for $i \leftarrow 1$ to $N - 3$ **do**
  for $j \leftarrow i + 2$ to $N - 1$ **do**
    $P' \leftarrow$ swapEdges(T, i, j)
    $c \leftarrow$ getCost($nodes$, P')
    if $c < mincost$ **then**
      $minpath \leftarrow P'$
      $mincost \leftarrow c$
    else
      $T \leftarrow$ swapEdges(T, i, j)
    end if
  end for
end for
return minpath, mincost =0

F. Simulated Annealing

The Simulated Annealing problem for the TSP is a stochastic algorithm which works by exploring the search space to find the best local optima. This makes the SA algorithm a meta-heuristic [17]. A meta-heuristic efficiently explores the search space to find good solutions. The algorithm starts from a randomly chosen path, or an already existing solution, from another heuristic, and attempts to make improvements to it. This could be done by random swaps, 2-Opt, 3-Opt [17].

The beauty of the algorithm lies in the temperature factor. In fact the term annealing is analogous to the metallurgical annealing, where a material is very prone to shapeshift at a high temperature but solidifies as it cools down. Therefore, the cooling rate acts as a trade-off, with a high cooling rate makes the algorithm pinpoint a local optima faster, but a slower cooling rate would escape local optima more frequently and explore the search space.

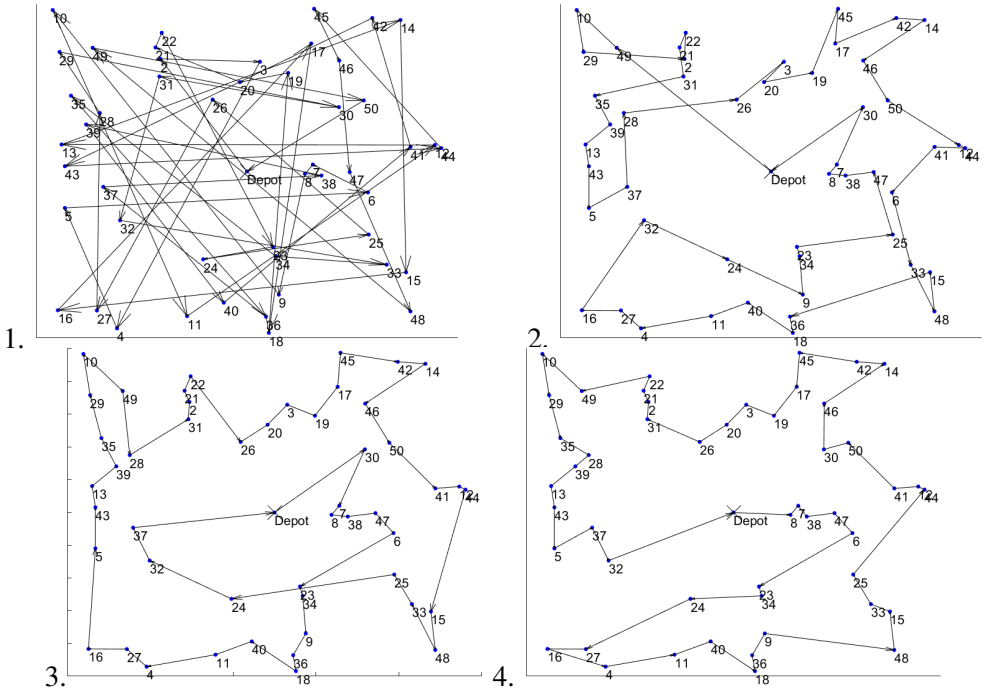

Fig. 3. Repeatedly applying 2-Opt from a random path until local optimum

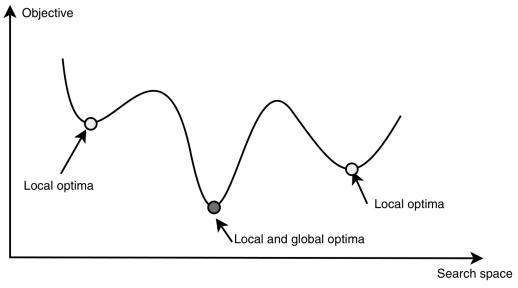

Fig. 4. Graphical Representation of local and global optima

1) Choice of Cooling Rate: Choosing an appropriate cooling rate that balances the trade-off between exploration and exploitation is essential. The optimal cooling rate depends on the specific problem and the characteristics of the search space, and it may require some experimentation to find the best value. The SA algorithm will choose a worse path more likely in the beginning of the algorithm towards the end. Because of this, if the cooling rate is too slow, the algorithm may get stuck in local optima and not explore the search space efficiently. On the other hand, if the cooling rate is too fast, the algorithm may need more time to explore untouched regions of the search space before converging to an acceptable solution.

Algorithm 6 Simulated Annealing Algorithm for TSP

```
min_cost_sa ← inf
T0 ← 1
coolingRate ← 0.90
numIterations ← 10
numAttempts ← 10
for i ← 1 to numIterations do
    T ← T0 * (coolingRate^i)
    currentPath ← randperm(length(nodes))'
    currentPath(currentPath == 1) ← []
    currentPath ← [1; currentPath; 1]
    currentDist ← getCost(nodes, currentPath)
    for j ← 1 to numAttempts do
        path_sa ← two_Opt(nodes, currentPath)
        newDist ← getCost(nodes, path_sa)
        deltaE ← newDist - currentDist
        acceptanceProbability ← exp(-deltaE/T)
        if rand() > acceptanceProbability then
            path_sa ← random_swap(path_sa), (could be 2-Opt or 3-Opt)
        end if
        if deltaE ≤ 0 then
            currentPath ← path_sa, currentDist ← newDist
        else
            acceptanceProbability ← exp(-deltaE/T)
            if rand() < acceptanceProbability then
                currentPath ← path_sa, currentDist ← newDist
            end if
        end if
    end for
    if getCost(nodes, path_sa) < min_cost_sa then
        min_cost_sa ← getCost(nodes, path_sa), min_path ← path_sa
    end if
end for
path_sa ← min_path =0
```

The time complexity of the simulated annealing algorithm largely depends on the number of iterations and number of attempts. As a safe assumption, the number of attempts and iterations can be N/10 respectively (this assumption was determined from personal experimentation and trial and error). Where N is the number of nodes. When calculating the time complexity this factor will not make any difference regardless.

Step	Operation	Runtime
1	Number of Iterations	$O(N)$
2	Number of Attempts	$O(N)$
3	Applying 2-Opt	$O(N^2)$
4	Total	$O(N^4)$

TABLE IV

TIME COMPLEXITY CALCULATION OF SIMULATED ANNEALING [20].

As the number of nodes increases, the differences in the final path distance will be larger. In fact, major changes in the TSP tours start from as soon as 13 nodes.

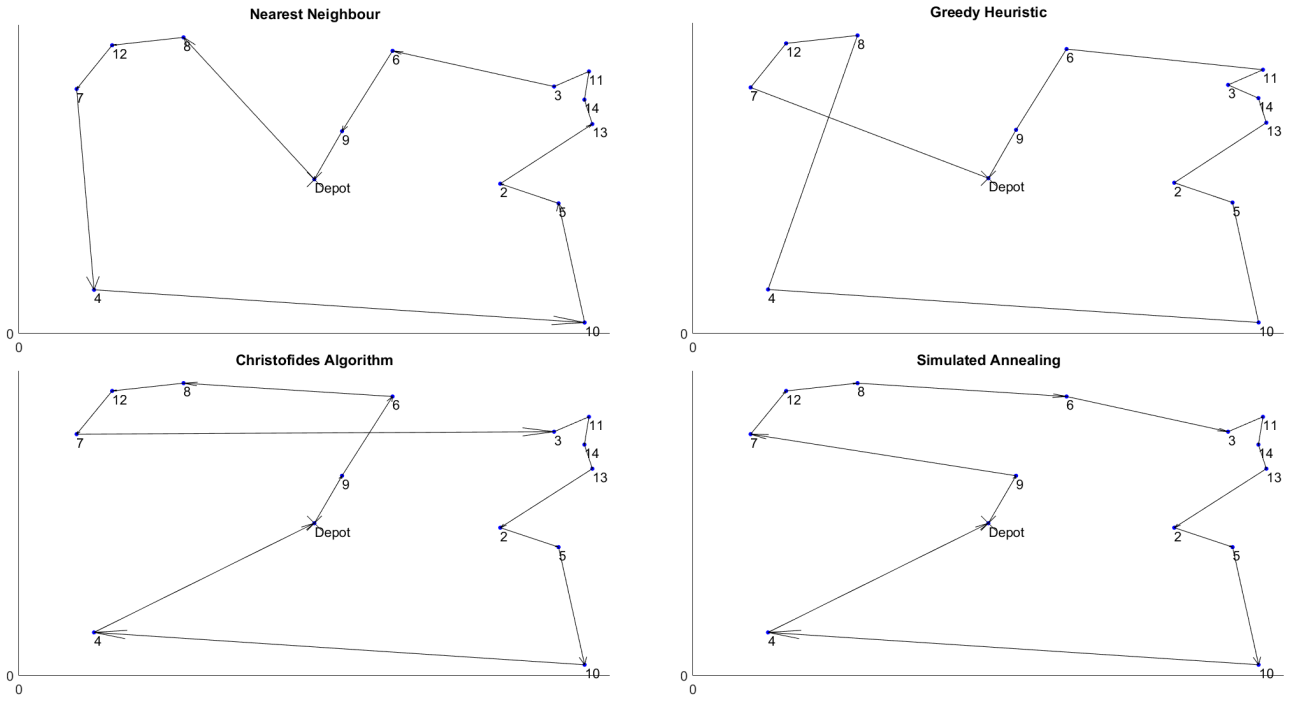

Fig. 5. Path Outputs of the TSP algorithms at 13 nodes

G. 1-Tree Lower Bound

The one tree lower bound is a lower bound of the minimum spanning tree. It is based on the idea that any spanning tree in a graph can be expressed as a combination of an MST and a set of edges that form a tree with exactly one cycle. A spanning tree is an edge removed from a valid TSP tour [5].

The one tree is formed by removing an edge from the graph, finding the MST, and connecting that edge to the 2 closest nodes, to form a cycle. The sum of weights of this tree will always be less than or equal

to the cost of the TSP tour. Now, the one tree with the maximum cost is called the one-tree lower bound. This bound is preferred to the MST because it is tighter.

Heuristic	Average Cost/1-Tree Lower Bound
NNH	1.25
GH	1.17
CHR	1.11
SA	NA (Varies)

TABLE V

AVERAGE COST RATIO OF HEURISTICS COMPARED TO THE 1-TREE LOWER BOUND [6]

V. UAV MODELS

As mentioned before, UAVs can be used for various reasons such as surveillance, delivery, search and rescue operations, especially in remote areas. Regarding temporary cell coverage, the UAV must meet several requirements i.e., the regulations, sufficient battery life, signal range, and payload capacity. Therefore, 3 UAV drones with different weight classes have been simulated and compared to see which drone consumed the least energy. before assuming that the larger drone will consume more energy, the smaller drone would need more recharges to complete the tour, if it is not able to complete the tour in a single charge, thereby increasing the overall energy consumed by the lighter drone.

All the models were chosen from DJI as the comparison would be more accurate. Since all the models use similar components and design features, this makes it somewhat easier to isolate the features that cause the difference in energy consumption. In this case, they are the weight and the total energy of the battery.

The DJI Inspire 2 is a versatile drone that is used in various industries. However, regarding communication services, it is on the lighter end of the UAV spectrum, and therefore should consume less energy to complete the TSP tour. The second model is the DJI Matrice 210 which is slightly heavier at 4.53kg, containing 2 batteries of 174.6Wh energy [12]. The third model is the DJI Matrice 600 at 9.6kg including the batteries; which are 6 129.96Wh batteries, having a total of 779.76Wh battery capacity[13].

Specification\Aircraft	DJI Inspire 2	DJI Matrice 210	DJI Matrice 600
Weight	3.290kg	4.53kg	9.6kg
Max Takeoff Weight	4.000kg	6.14kg	15.1kg
Max Ascent Speed	5 m/s	4 m/s	5 m/s
Max Descent Speed	4 m/s	3 m/s	3 m/s
Max Speed	94 kph	82.8 kph	65 kph
Max Wind Speed Resistance	10 m/s	10 m/s	8 m/s
Battery			
Battery Model	TB50	TB55	TB47S/TB48S
Capacity	4280 mAh	4920 mAh	7800 mAh
Battery Energy	2×97.58 Wh	2×174.6 Wh	6×129.96 Wh

Relevant Specifications of the DJI UAVs [11]-[13]

VI. Power Consumption Model

The power consumption of the drones are calculated as follows [14],

$$P_T = P_0(1 + \frac{3V^2}{U\_tip^2}) + P_i(\sqrt{1 + \frac{V^4}{4v_0^4}} - \frac{V^2}{2v_0^2})^{\frac{1}{2}} + \frac{1}{2}d_0 psAV^3 \tag{1}$$

where P_0 represents the blade profile power is expresses as $P_0 = \frac{\delta}{8}\rho sA\Omega^3 R^3$ and P_i represents the induced power in hovering status and is calculated by $P_i = (1+k)\frac{mg^{\frac{3}{2}}}{\sqrt{2\rho A}}$.

When considering typical communication applications, it is acceptable to overlook the extra energy used by a UAV during acceleration since the time it takes for the UAV to maneuver is typically a small fraction of the overall operation time, for the purpose of simplifying calculations. All the constants used in the value are standard values (Appendix of 14).

Notation	Description	Simulation Value
W	Weight kg	(varies)
ρ	Air density in kg/m^3	1.225
R	Rotor radius m	0.5
A	Rotor disc area m^2	0.79
Ω	Blade angular velocity in radians/second	50
U_tip	Tip speed of the rotor blade	200
s	Rotor solidity	0.05
d_0	Fuselage drag ratio	0.3
k	Thrust-to-weight ratio	2
Ω	Profile drag coefficient	0.012
v_0	Mean rotor induced velocity in hover	7.2

Notations and Terminologies for rotary-wing aircraft (Table 1 of[14])

A. *Deciding An Optimum Speed*

The maximum endurance (ME) and maximum range (MR) speed of the drones are important qualities of the drone that can be calculated with the power consumption model above [14]. However since the use of the UAVs in this case involves stopping and starting at many intervals, as well as ascending and descending since this is a real-life problem. It will be assumed that the UAV will travel at a constant speed between nodes.

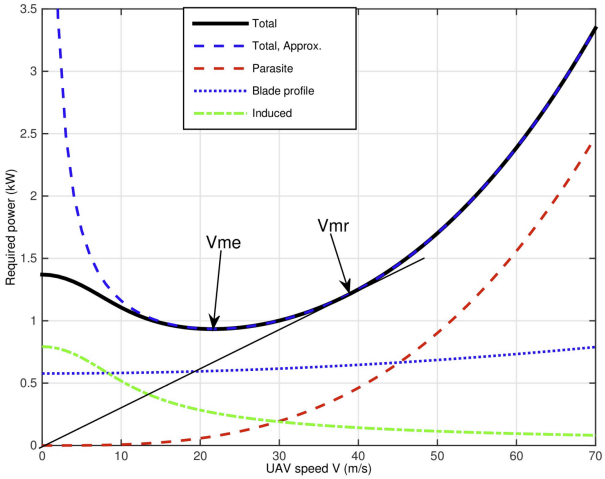

Fig. 6. Propulsion power consumption versus speed V for rotary-wing UAV. Figure 1. of [14]

Figure 4 shows a plot of P(V) against the speed of the UAV. When operating the UAV, it will face many changes in trajectory, including steep rises and falls. Since the smallest ascent/descent speed is 3m/s and $\arg\min(V) = 3m/s$, for $V \in (0, 3]$, 3m/s will be chosen as the final constant speed for all drone models, while traversing the nodes. Given the distance of the TSP tour, the total energy can be calculated by,

$$E = P(V) \cdot T, \qquad [T = D/V, \qquad E = P(V) \times (D/V)] \tag{2}$$

VII. Constraints and Clustering

In real-life situations, factors such as wind, rain, and other temperatures were ignored. This is due to the fact these factors are applied to the algorithms in the same manner and do not favour any algorithm over the other. Due to similar reasons, factors such as turning angle between nodes, and gravity can be neglected. Since the energy required for communications is the same regardless of the model of the drone or TSP heuristic used, this factor can be neglected as well.

The main problem that arises when implementing the TSP algorithms to the UAV is that the UAV may not have enough energy to complete the tour in one full charge. A simple fix to this problem is to split the nodes into clusters. This is done by k-means i.e., MATLAB's in-built k-means function with its default settings. Therefore, if there is more than one cluster, the UAV will return to the depot after serving one cluster and recharge fully before serving the other remaining clusters.

The k-means algorithm works by selecting k random points on a plane which contains the data points, k here being the number of desired clusters that the data points want to be grouped into. All points are assigned to clusters depending on how close the data point is to the cluster center. Cluster centers are now recalculated by finding the average position of all data points within each cluster. This process repeats until there is a convergence i.e., there is no significant change in cluster centers or the maximum number of iterations are completed. [22]

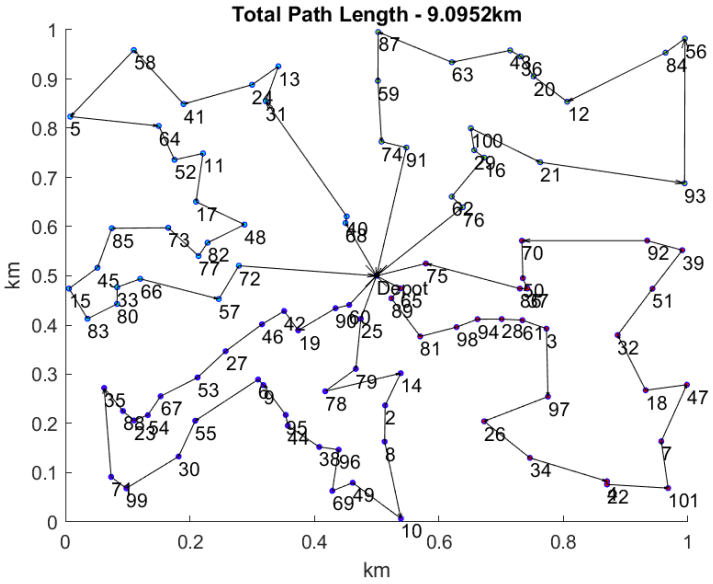

Fig. 7. Splitting 100 nodes into four clusters using k-means and plotting the path direction of each cluster using SA

VIII. Numerical Results

A. Time Complexity Comparison

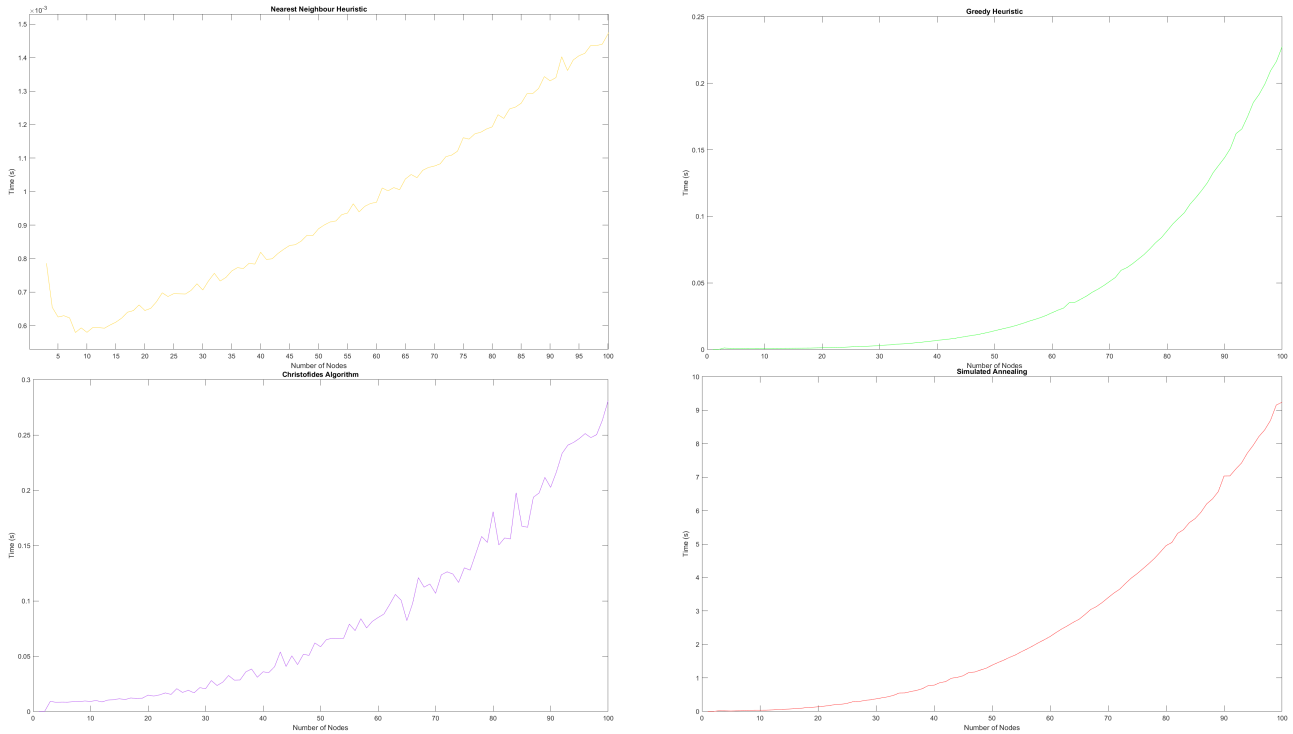

Fig. 8. Time taken to perform the algorithm at each number of nodes

The reason for the erratic results for the Christofides algorithm is because The behavior of MATLAB can be unpredictable regarding the time it takes to execute functions. The duration of MATLAB function executions can fluctuate drastically, especially when calling functions from external files, or libraries. No number of trials gave a smooth result for the Christofides algorithm.

Besides, there are no noticeable outliers in the figures and all the algorithms, and the time complexity shown by the remaining graphs of the other algorithms justify the time complexity, as solved earlier in the paper.

B. Minimum Spanning Tree Results

The Prim's algorithm used to find the minimum spanning tree was compared to MATLAB's in-built minspantree(G) function and the results were indentical.

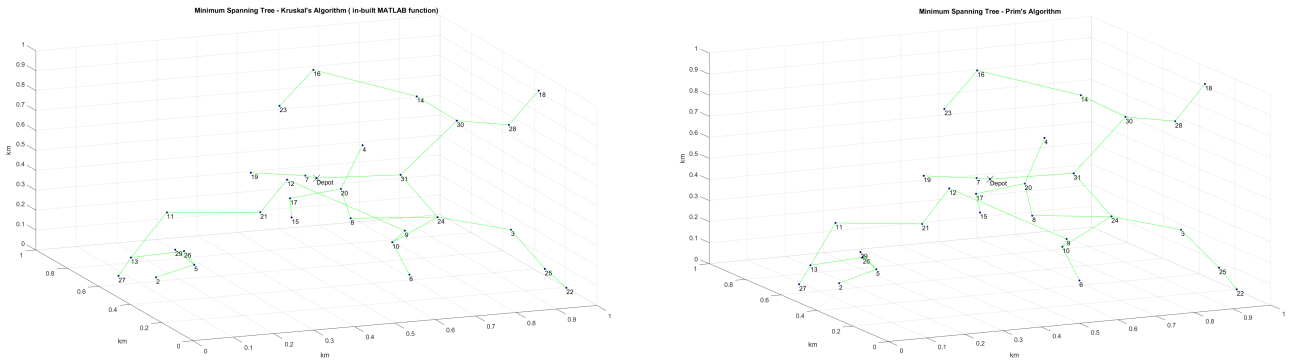

Fig. 9. MST comparison of Prim's and Kruskal's algorithm

C. Cost Comparison

1) Monte Carlo Method: A Monte Carlo Method was conducted where the average cost of the path for each algorithm from 2 nodes up to 100. The position of the nodes was randomised using the MATLAB seed function for multiple trials.

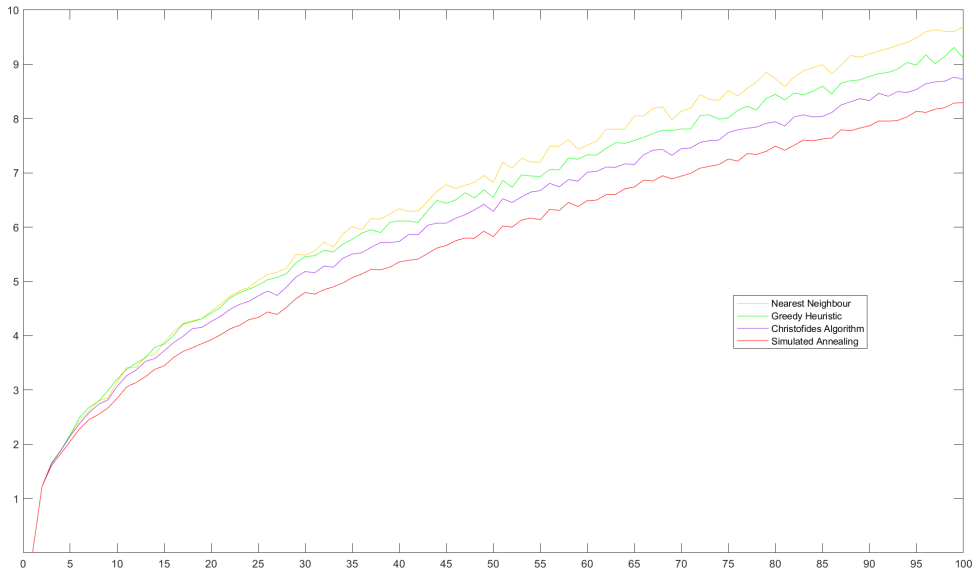

Fig. 10. Cost Comparison of TSP Algorithms - Monte Carlo Method (50 Trials)

The results are as expected, with simulated annealing giving the lowest cost on average for all nodes, followed by christofides, greedy heuristic, and nearest neighbour. The reason for the logarithmic rise in path cost is due to the fact that the size of the plane remained constant and only the number of nodes increased. This results in a lot of nodes being generated close to each other and thereby affecting the overall distance insignificantly. Only the time taken to implement the heuristics increase, since there are exponentially more possible TSP tours for each node added to the plane.

Algorithm\Nodes	25	50	75	100	125	150
Nearest Neighbour	5.0397	6.872	8.556	9.763	10.685	11.835
Greedy	4.965	6.627	8.112	9.146	10.142	11.207
Christofides	4.772	6.319	7.770	8.766	9.699	10.641
Simulated Annealing	4.365	5.852	7.276	8.305	9.2141	10.1324

TABLE VI

COST OF PATH IN KM OF TSP ALGORITHMS FOR A VARIOUS NUMBER OF NODES (50 TRIALS)

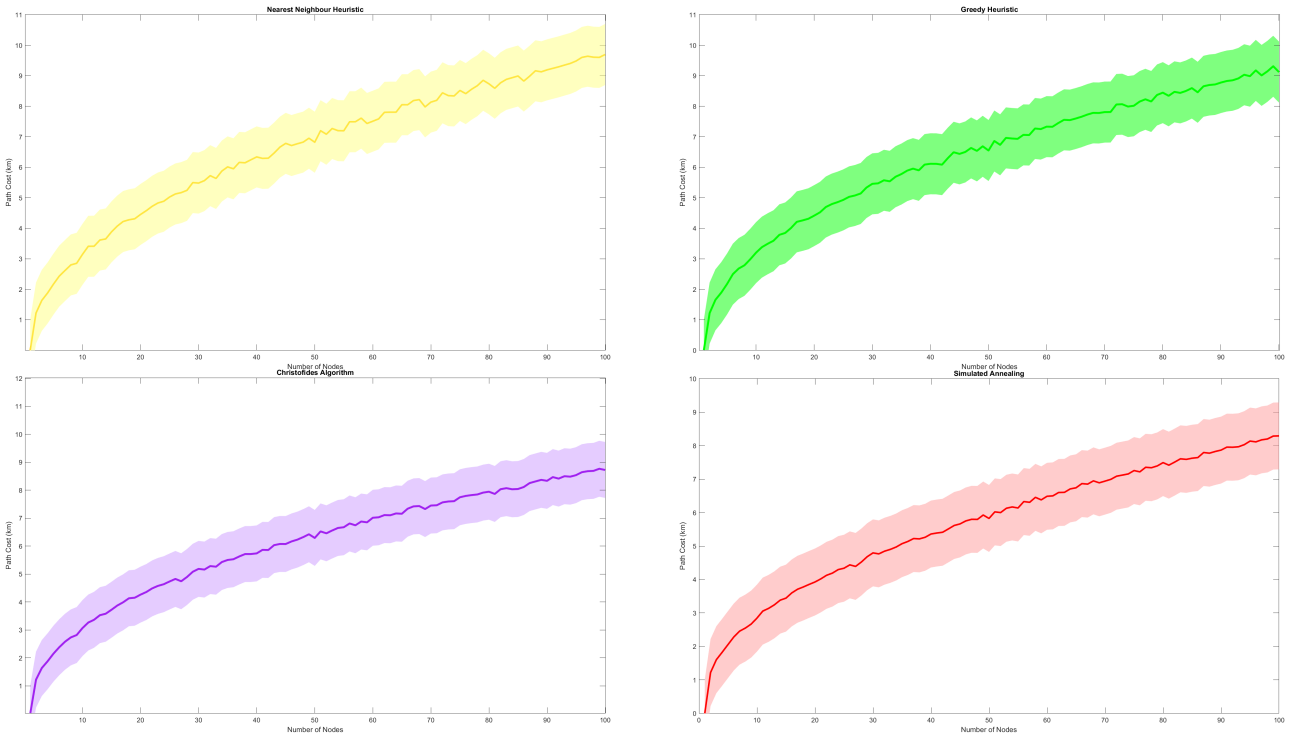

Fig. 11. Standard Deviation as a shaded region of the TSP Heuristics (50 trials)

2) Standard Deviation: All the heuristics operate within 1 standard deviation, with SA having the least deviation by a fair amount across all nodes. The same order is displayed as the mean figure. This is expected because a better performing algorithm should generally have a lower standard deviation since a quality algorithm would have more values around the mean, whereas an inefficient algorithm would more likely have sparsely dispersed points skewing the standard deviation, as displayed by the nearest neighbour heuristic, making it unreliable and inconsistent.

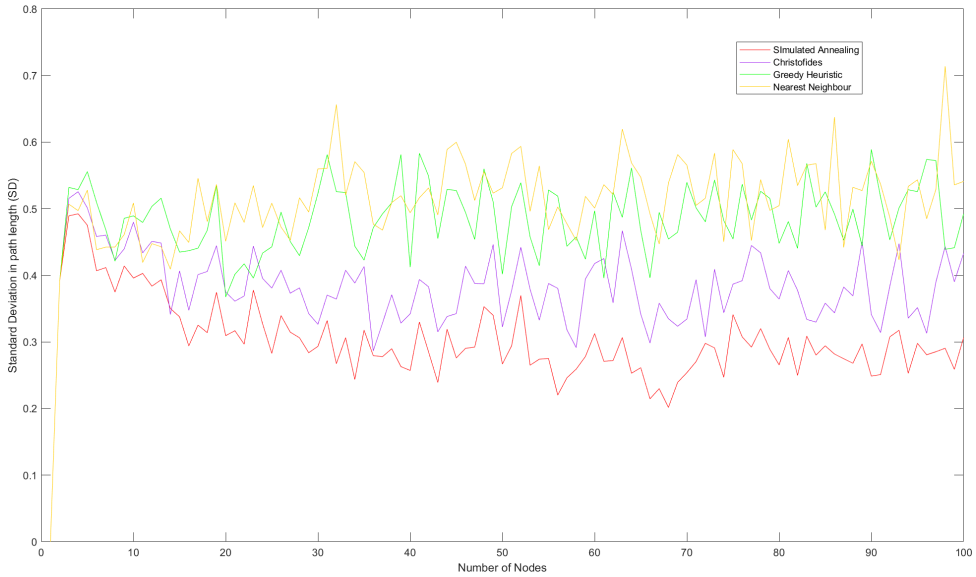

Fig. 12. Standard Deviation (SD) in Path length across all nodes (50 Samples)

In the case of comparing 2-Opt optimal solutions, the order of heuristics still remain the same, however the algorithms display a very similar level of performance. A 2-Opt optimal path is a path which has already reached its local optima by repeatedly applying 2-Opt. Despite SA already being a 2-Opt optimal solution, it is still a higher quality algorithm than the other three. However, after around 80 nodes, the Christofides algorithm performs as well as simulated annealing, if not marginally better. This is because there were not as many iterations for a larger number of nodes in the SA algorithm. Increasing the number of iterations i.e. the number of attempts at finding the local optima is expensive. Considering the time complexity of simulated annealing, It is best that Christofides (2-Opt Optimal) algorithm should be used, when the number of nodes is above 80.

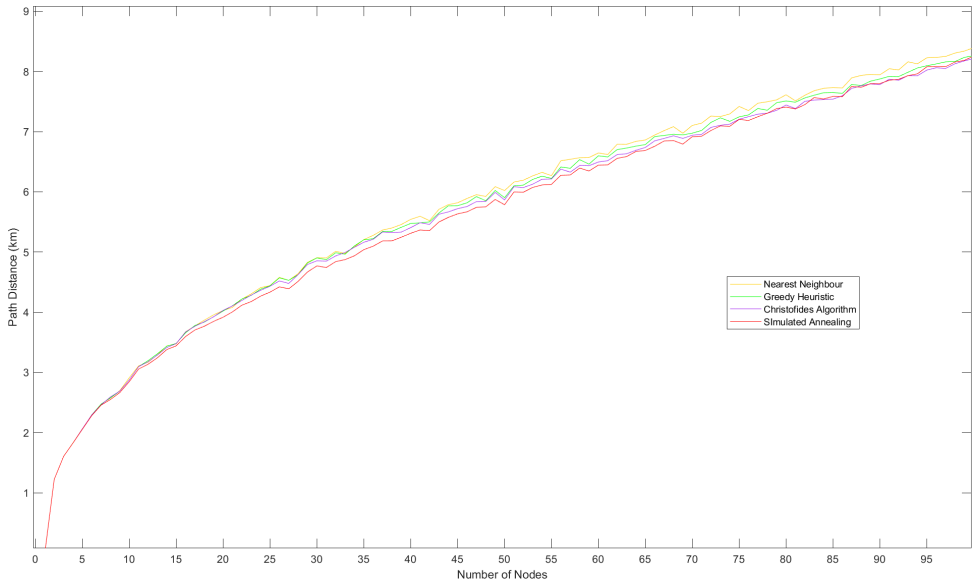

Fig. 13. Cost Comparison of TSP Algorithms (2-Opt Optimal)- Monte Carlo Method (50 Trials)

3) Simulated Annealing: A Comprehensive Overview: Despite the temperature analogy and the beauty of the algorithm. The new solution after applying 2 Opt is either better or the same as the current solution. Therefore accepting the new solution does not make any difference. Using random swaps instead of 2-Opt as a means of attempting to improve the path did not yield acceptable results. This rendered the temperature and cooling rate parameters insignificant. In addition, 3-Opt and k-Opt were too expensive and excessively time consuming.

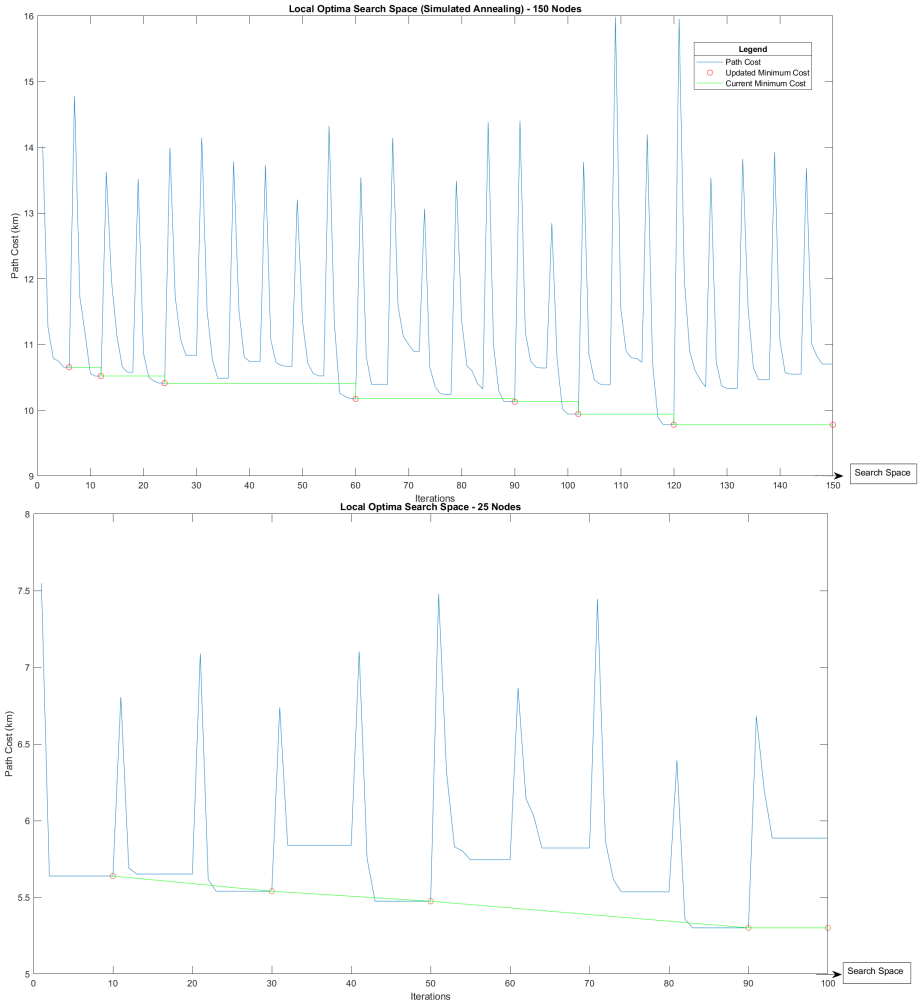

Fig. 14. Visual representation of the Simulated Annealing algorithm showcasing the working of the SA algorithm

This shows that an adequate solution can be obtained using simulated annealing just within a few iterations. but to arrive at a near optimal solutions, deciding the number of iterations can be difficult, as there is a trade-off. Having more iterations is extremely expensive, but is guaranteed to find a better solution than the current one. After trial and error. it was found that the local optima is reached after 6-7 executions of 2-Opt for under 150 nodes. Performing this multiple times will give multiple local optima. The major disadvantage of this algorithm is that we do not know if we have arrived at a near optimal solution or if there is a better solution in the search space.

D. Clustering

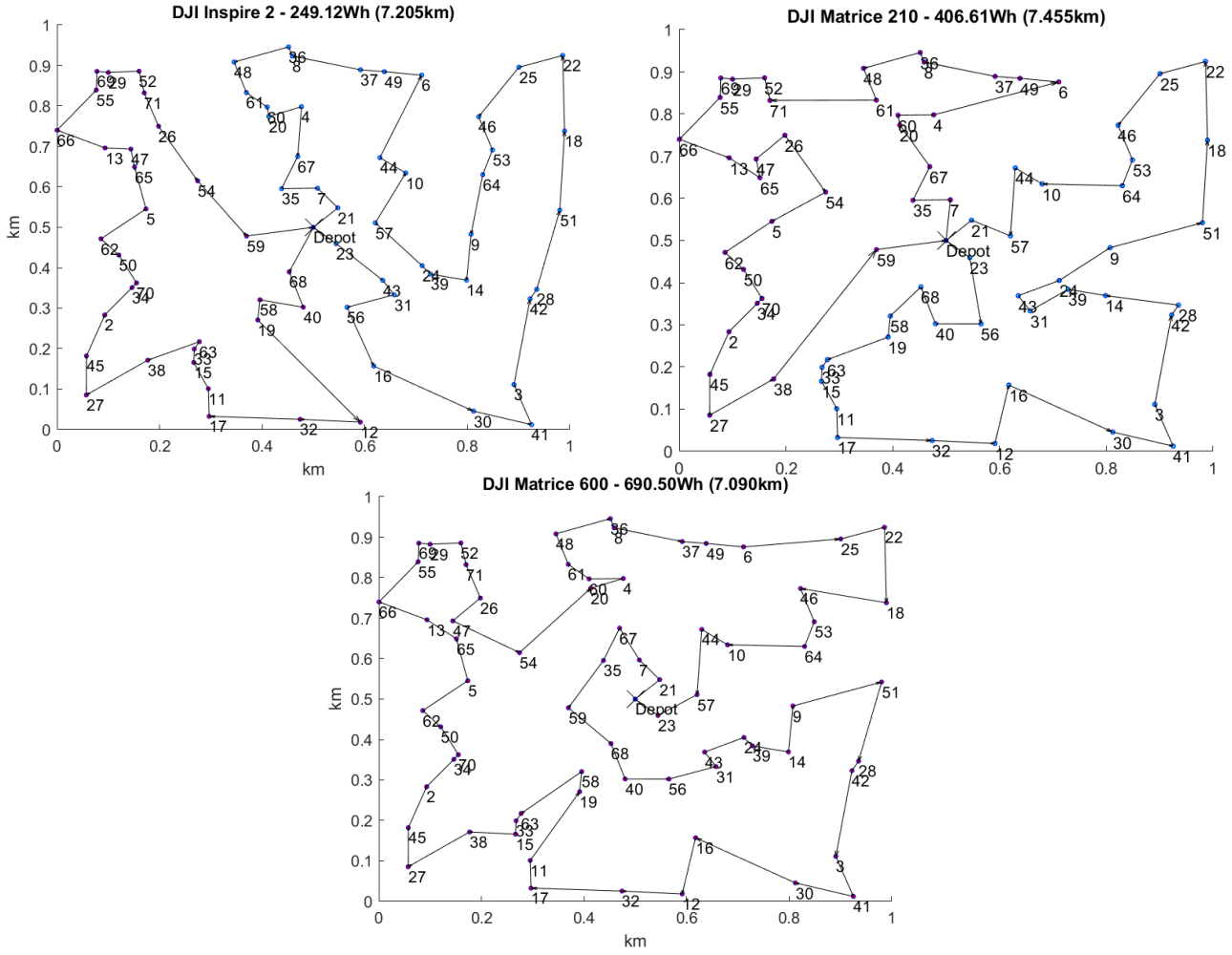

Fig. 15. Visual representation of valid tours of the three UAV models for 70 nodes

Simulated annealing was used to optimise each individual cluster, since the number of nodes are relatively small. For example, at 70 nodes, the results are as expected, with the DJI Matrice 600 requiring only 1 cluster, i.e. being able to serve all nodes in a single charge, while the DJI Inspire 2 and DJI Matrice 210 require 2 full charges. This results in an overall longer path distance.

Despite the longer path, for the lighter drones, the weight of the DJI Matrice 600 requires more energy due to its weight. This trend was analysed across a range of a different number of nodes, Validating the fact that the decrease in overall path length for the heavier drone does not compensate for the energy consumed due to its weight.

Nodes	10	20	30	40	50	60	70	80	90	100	110
DJI Inspire 2	125.04	125.13	152.11	168.89	213.77 (2)	232.09 (2)	224.59 (2)	264.76 (2)	271.78 (2)	291.67 (2)	309.68 (3)
DJI Matrice 210	197.26	197.41	239.96	261.91	310.88	371.40	341.07 (2)	419.17 (2)	437.08 (2)	454.19 (2)	481.74 (2)
DJI Matrice 600	352.25	352.51	428.51	462.11	568.43	582.52	598.73	745.64	777.07	817.17 (2)	864.75

TABLE VII

ENERGY CONSUMED BY EACH DRONE FOR PATHS WITH A CERTAIN NUMBER OF NODES IN WH

the number in the brackets indicates the number of clusters that were required for the drone to complete the path without running out of energy.

IX. Conclusion

Regarding mean value and standard deviation, this study has achieved to prove that simulated annealing outperforms the other algorithms by a clear margin, followed by Christofides, greedy heuristic, and nearest neighbour heuristic, including 2-Opt optimal variations. However, it is recommended that Christofides algorithm should be used for a larger number of nodes if time taken to compute the path is a concern. Time complexities of the heuristics were also verified. It can also be concluded that the clustering factor does not hold greater significance than the aircraft model in determining the outcome. This suggests that while clustering is an important aspect to consider, it does not have be prioritized over the model of the aircraft when making decisions to minimise energy consumption

Moving forward. This study will explore more algorithms, such as ant colony optimisation and the genetic algorithm. A study called *Travelling salesman problem for UAV path planning with two parallel optimization algorithms*, by Jie Chen et al. [23], concluded based on their experimental results and analysis, that the improved genetic algorithm (IGA) and the particle-swarm-optimization-based (PSO-ACO) algorithms outperformed the contrast approach, i.e. just the ACO algorithm for TSP and produced more reasonable and effective solutions for UAV path planning, and that their algorithms for optimal UAV path planning are recommended. Therefore, we can expect that ACO and GA algorithms will perform better than the heuristics in this study.

X. Ethics and Health Safety

Drone laws are generally strict all over the world. There are many laws that have restricted drone usage. In fact, almost all countries in the MENA (Middle East/North Africa) region have either banned UAV usage or have made it close to impossible to fly drones without getting the required licence [18]. There are multiple reasons to this, such as:

i) Instability of the political climate of the country. Drones are seen as a threat in such countries. ii) Security and surveillance reasons iii) Wildlife protection

Drone laws in the UK are strict as well. You must have a licence to operate a drone in London and this can only be obtained from the Civil Aviation Authority (CAA). Drones can't be flown over crowded areas. Unfortunately, this happens to be the primary use case for the UAV. The drone must not be over 500g. Most ABSs weigh a minimum of 4kg [19]. The CAA also mentions that when operating, the drone must exceed the height of 120m, whereas most UAVs performing communication service operate higher than this. Regarding weather conditions, the CAA vaguely mentions not to fly the UAV if the weather conditions do not permit but does not give the exact speed of the wind or amount of rainfall that is safe. All these factors restrict the usage of UAVs not just in the UK but in the EU region as well. Therefore, this simulation cannot be conducted in real life UK unless the UAV operator has a special permit to use the UAV.

Drones should be operated safely and under proper supervision, if required. Only licenced individuals must operate heavy UAVs. Improper training could easily result in injury, or fatalities of civilians. Drones can also be a nuisance to people. Therefore, prolonged use of drones should be avoided as it can cause hearing impairment easily. [20]

Using UAVs as flying base stations can lead to more sustainable cities and communities. By eliminating the need for cell towers, drones can sometimes optimize their paths, significantly reducing energy consumption and minimizing energy waste, which is a step towards reducing climate change. In addition, using drones as base stations can have potential health benefits, as it may reduce constant exposure to electromagnetic waves. Instead, the waves will only be present when there is a high demand in mobile data usage. Using drones as base stations can contribute to sustainability by improving energy efficiency and reducing potential health risks.

REFERENCES

[1] Martello, Silvano; Toth, Paolo. (1990). *Knapsack problems: Algorithms and computer implementations.*

[2] Johnson, David S.; McGeoch, Lyle A. (1997). *The Traveling Salesman Problem: A Case Study in Local Optimization.*

[3] "Solving Travelling Salesman (TSP) using 2-Opt Algorithm in Python." Learn with Panda, 19 Dec. 2022, https://learnwithpanda.com/2022/12/19/solving-travelling-salesman-tsp-using-2-opt-algorithm-in-python/.

[4] Goodrich, Michael T.; Tamassia, Roberto, "18.1.2 The Christofides Approximation Algorithm", *Algorithm Design and Applications*, Wiley, 2015

[5] Volgenant, T., and Jonker, R. , "A branch and bound algorithm for the symmetric traveling salesman problem based on the 1-tree relaxation", 1982

[6] The traveling salesman problem: When Good Enough Beats Perfect, YouTube. 2022, https://www.youtube.com/watch?v=GiDsjIBOVoA

[7] "Applications of Minimum Spanning Trees," University of Texas at Dallas, Richardson, TX, USA, 2004.

[8] Sherif Tawfik (2023). mixed-integer LP (https://www.mathworks.com/matlabcentral/fileexchange/6990-mixed-integer-lp), MATLAB Central File Exchange. Retrieved March 10, 2023.

[9] Austin Buchanan Christofides Algorithm Visualisation (https://github.com/AustinLBuchanan/TSP_VRP/blob/main/Christofides_algorithm.ipynb GitHub, 2022

[10] Rico Zenklusen, A 1.5-Approximation Path for TSP, 2018

[11] DJI, "Inspire 2 User Manual, V1.0" DJI, Shenzhen, China, 2016 Decemeber.

[12] DJI, "Matrice 210 Series User Manual, V1.4" DJI, Shenzhen, China, 2018 November.

[13] DJI, "Matrice 600 User Manual, V1.0" DJI, Shenzhen, China, 2017 July.

[14] Y. Zeng, J. Xu and R. Zhang, "Energy minimization for wireless communication with rotary-wing UAV, " *IEEE Transactions on Wireless Communications*, Vol.18.

[15] A. Filippone, *Flight performance of fixed and rotary wing aircraft*. American Institute of Aeronautics Ast (AIAA), 2006.

[16] G. Laporte, "The traveling salesman problem: an overview of exact and approximate algorithms," 1992.

[17] ZIcheng Wang, Xiutang Geng, Zehui Shao, An Effective Simulated Annealing Algorithm for Solving the Traveling Salesman Problem,

[18] Drone Laws in the UAE, "https://uavcoach.com/drone-laws-in-saudi-arabia/"

[19] Flying drones and model aircraft, "https://register-drones.caa.co.uk/"

[20] Beat Schäffer ,Reto Pieren, Kurt Heutschi, Jean Marc Wunderli Drone Noise Emission Characteristics and Noise Effects on Humans—A Systematic Review , 2021

[21] Sasaki, Galen H. Optimization by Simulated Annealing: A Time-Complexity Analysis., 1987

[22] kmeans - k-means clustering , MathWorks. https://uk.mathworks.com/help/stats/kmeans

[23] Jie Chen, Fang Ye, Yibing Li, Travelling salesman problem for UAV path planning with two parallel optimization algorithms, 2017

APPENDIX

Source Code

```
clc;clear;close all;
%

N=30;

nodes=generateNodes(N,144);
```

```
origin=[0.5 0.5 0.5];

nodes=[origin;nodes];



% Original Path
tic
figure('Name','Random Order of Nodes','NumberTitle','off');
plotNodes(nodes);
path_rnd=1:N+1;
path_rnd(end+1)=path_rnd(1);
cost_rnd1=getCost(nodes,path_rnd);
fprintf('Cost of Random Path: %f\n',cost_rnd1);
plotPathDirection(nodes,path_rnd);
% Original Path - 2-Opt
toc;
figure('Name','Random Order - 2-Opt Optimal','NumberTitle','off');

path_rnd=two_Opt(nodes,path_rnd);
cost_rnd2=getCost(nodes,path_rnd);
while(~(cost_rnd2==cost_rnd1))
    cost_rnd2=getCost(nodes,path_rnd);
    path_rnd=two_Opt(nodes,path_rnd);
    cost_rnd1=getCost(nodes,path_rnd);
end



plotNodes(nodes);

plotPathDirection(nodes,path_rnd);
fprintf('Cost After 2-Opt Optimization: %f\n',getCost(nodes,path_rnd));

%Nearest Neighbour Heuristic
tic;
figure('Name','Nearest Neighbour Heuristic','NumberTitle','off');
```

```
% nodes(end,:) = [];
distmat_nnh=getDistmat(nodes);
plotNodes(nodes);
p1=origin;
current=1;
for i=1:N
    x=min_nz(distmat_nnh(current,:));
    x(1)=current;
    p2=nodes(x(2),:);
    dp=p2-p1;
    quiver3(p1(1),p1(2),p1(3),dp(1),dp(2),dp(3),0,'black');

    p=distmat_nnh(current,:);
    p(p>0)=0;
    distmat_nnh(current,:)=p;
    p=distmat_nnh(:,current);
    p(p>0)=0;
    distmat_nnh(:,current)=p;
    distmat_nnh(current,x(2))=-1;
    distmat_nnh(x(2),current)=-1;

    p1=p2;
    current=x(2);

end
current=1;
p=distmat_nnh(current,:);
p(p>0)=0;
distmat_nnh(current,:)=p;
p=distmat_nnh(:,current);
p(p>0)=0;
distmat_nnh(:,current)=p;
distmat_nnh(current,x(2))=-1;
distmat_nnh(x(2),current)=-1;
```

```
p1=p2;p2=origin; dp=p2-p1;quiver3(p1(1),p1(2),p1(3),dp(1),dp(2),dp(3),0,'black');
path_nnh=getPath(distmat_nnh,1);

fprintf('Cost of NNH: %f\n',getCost(nodes,path_nnh));



toc;


% 2-Opt NNH
figure('Name','Nearest Neighbour Heuristic - 2-Opt Optimal','NumberTitle','off');
plotNodes(nodes);
cost_nnh1=getCost(nodes,path_nnh);
path_nnh=two_Opt(nodes,path_nnh);
cost_nnh2=getCost(nodes,path_nnh);
while(~(cost_nnh2==cost_nnh1))
    cost_nnh2=getCost(nodes,path_nnh);
    path_nnh=two_Opt(nodes,path_nnh);
    cost_nnh1=getCost(nodes,path_nnh);
end
plotNodes(nodes);

plotPathDirection(nodes,path_nnh);


fprintf('Cost After 2-Opt Optimization: %f\n',getCost(nodes,path_nnh));
plotPathDirection(nodes,path_nnh);




% Greedy Heuristic Approach
tic;
```

```
figure('Name','Greedy Heuristic','NumberTitle','off');


plotNodes(nodes);


distmat_gh=getDistmat(nodes);



traversed=zeros(N+1,1);
path_gh=0;

for j=1:(N+1)^2
    x=min_nz(distmat_gh);
    if(x(3)==0)
        break;
    end

    if(traversed(x(1))<2 && traversed(x(2))<2 && ~areIndirectlyConnected(distmat_g




        p1=nodes(x(1),:);p2=nodes(x(2),:);

        line(p1,p2);
        path_gh=path_gh+dist(p1,p2);

        traversed(x(1))=traversed(x(1))+1;
        traversed(x(2))=traversed(x(2))+1;

        distmat_gh(x(1),x(2))=-1;
        distmat_gh(x(2),x(1))=-1;

    else
```

```
        distmat_gh(x(1),x(2))=0;
        distmat_gh(x(2),x(1))=0;
    end




end


a=1; % Adding last edge manually
for a=1:length(traversed)
    if(traversed(a)==1)
        p1=nodes(a,:);
        break;
    end
end
for i=(a+1):length(traversed)
    if(traversed(i)==1)
        p2=nodes(i,:);
        distmat_gh(a,i)=-1;
        distmat_gh(i,a)=-1;
    end
end
line(p1,p2);
path_gh=path_gh+dist(p1,p2);

fprintf('Cost of GH: %f\n',path_gh);

hold off



toc;
figure('Name','Greedy Heuristic - 2-Opt Optimal','NumberTitle','off');
```

```
path_gh_2opt=getPath(distmat_gh,1); %  greedy heuristic


path_gh_2opt=two_Opt(nodes,path_gh_2opt); path_gh_2opt=two_Opt(nodes,path_gh_2opt)



plotNodes(nodes);

plotPathDirection(nodes,path_gh_2opt);
fprintf('Cost After 2-Opt Optimization: %f\n',getCost(nodes,path_gh_2opt));

%CHRISTOFIDES ALGORITHM
figure('Name','Minimum Spanning Tree - Prim''s Algorithm','NumberTitle','off');
plotNodes(nodes);
distmat_mst=getDistmat(nodes);
prims_algorithm(nodes,distmat_mst);



%MST
figure('Name','Minimum Spanning Tree','NumberTitle','off');


G=graph(distmat_mst);
T=minspantree(G,'Method','sparse'); %Kruskal
plotNodes(nodes);
for i=1:length(T.Edges.EndNodes)
    greenLine(nodes(T.Edges.EndNodes(i,1),:),nodes(T.Edges.EndNodes(i,2),:))
end

degrees=degree(T);
odd_nodes=find(mod(degrees, 2) == 1); %finds all the nodes which have an odd numbe
distmat_match=getDistmat(nodes(odd_nodes,:));

distmat_match(distmat_match==0)=inf;
```

```
matching = min_perfect_matching(distmat_match);

matching=[odd_nodes(1:length(matching)),odd_nodes(matching)];

for i=1:length(matching)
    for j=1:length(matching)
        if (matching(i,1)==matching(j,2) && matching(i,2)==matching(j,1))
            matching(j,:)=[0 0];
        end
    end
end
matching(all(~matching,2), : ) = [];


eulerian_graph=[T.Edges.EndNodes];
eulerian_graph=[eulerian_graph,ones(length(eulerian_graph),1)];


matching=[matching,ones(length(matching),1)];

for i=1:size(matching,1)

    for j=1:length(eulerian_graph)
        if (matching(i,1)==eulerian_graph(j,1) && matching(i,2)==eulerian_graph(j,
        eulerian_graph(j,3)=eulerian_graph(j,3)+1;
        matching(i,3)=0;
        end
    end
    redLine(nodes(matching(i,1),:),nodes(matching(i,2),:));
end

for i=1:size(matching,1)
    if(matching(i,3)==1)
        eulerian_graph=[eulerian_graph;matching(i,:)];
    end
end
```

```
== eulerian_graph(j, 2)) || (n1 == eulerian_graph(j, 2) && n2 == eulerian_graph(j,
```

```
eulerian_path=zeros(sum(eulerian_graph(:,3)),1);eulerian_path(1)=current_node;p=2;
||eulerian_graph(j,2)==current_node)
%                 if dist(nodes(eulerian_graph(j,1),:),nodes(eulerian_graph(j,2),:
%                     min_dist=dist(nodes(eulerian_graph(j,1),:),nodes(eulerian_gr
min_edge=j;
```

```
distmat_eg=zeros(N+1);

for i=1:length(eulerian_graph)
    distmat_eg(eulerian_graph(i,1),eulerian_graph(i,2))=eulerian_graph(i,3);
    distmat_eg(eulerian_graph(i,2),eulerian_graph(i,1))=eulerian_graph(i,3);
end

path_eg=get_eulerian_path(distmat_eg,nodes);
path_eg=unique(path_eg,"stable");
path_eg=[path_eg,path_eg(1)];
path_eg=path_eg';

figure('Name','Christofides Algorithm','NumberTitle','off');
plotNodes(nodes);
plotPathDirection(nodes,path_eg);
fprintf('Cost of CA: %f\n',getCost(nodes,path_eg));
path_eg=two_Opt(nodes,path_eg);
figure('Name','Christofides Algorithm - 2-Opt Optimal','NumberTitle','off');
plotNodes(nodes);
```

```
plotPathDirection(nodes,path_eg);
fprintf('Cost after 2-Opt: %f\n',getCost(nodes,path_eg));

toc;

%Simulated Annealing
figure('Name','Simulated Annealing','NumberTitle','off');
distmat_sa=getDistmat(nodes); % randomly generate a distance matrix

min_cost_sa=inf;

T0=1;
coolingRate = 0.95;
numIterations = 10;
numAttempts = 10; %

% Generate initial solution
currentPath = path_gh_2opt;

currentDist = getCost(nodes,currentPath); % calculate the total tour distance

for i = 1:numIterations

    T = T0 * (coolingRate ^ i);


    for j = 1:numAttempts
        path_sa = two_Opt(nodes,currentPath);
        newDist = getCost(nodes, path_sa);
        deltaE = newDist - currentDist;

        if deltaE < 0
            currentPath = path_sa;
            currentDist = newDist;
        % If new solution is worse, accept it with probability e^(-deltaE/T)
        else
```

```
            acceptanceProbability = exp(-deltaE/T);
            if rand() < acceptanceProbability
                currentPath = path_sa;
                currentDist = newDist;
            end
        end
    end
end

% find(path_sa==1);
% path_sa=circshift(path_sa,N+2-(find(path_sa==1)));
if(getCost(nodes,path_sa)<min_cost_sa)
    min_cost_sa=getCost(nodes,path_sa);

end
path_sa=[path_sa;1];
plotNodes(nodes);plotPathDirection(nodes,path_sa);
fprintf('Cost of SA: %f\n',getCost(nodes,path_sa));


% OPTIMISATION FUNCTIONS
function path = get_eulerian_path(adj_matrix,nodes)

        start_vertex = 1;

    % Initialize stack and path
    stack = start_vertex;
    path = [];

    while ~isempty(stack)
        % Take the last vertex on the stack
        current_vertex = stack(end);


        unvisited_edges = find(adj_matrix(current_vertex, :) > 0);
        visited_edges = find(adj_matrix(current_vertex, :) == 0);
```

```
        if ~isempty(unvisited_edges)
            % Visit the first unvisited edge
            next_vertex = nearest_unvisited_node(current_vertex,unvisited_edges,no

            adj_matrix(current_vertex, next_vertex) = adj_matrix(current_vertex, n
            adj_matrix(next_vertex, current_vertex) =adj_matrix(next_vertex, curre


            stack(end+1) = next_vertex;
        else

            stack(end) = [];

            % Add the current vertex to the path
            path(end+1) = current_vertex;
        end
    end


    path = fliplr(path);
end


function [MST] = prims_algorithm(nodes, distmat)

N=length(nodes);
nodes=nodes;
visited = false(1, N); % initialize all nodes as unvisited
MST = zeros(N-1, 2); % initialize the minimum spanning tree
total_cost = 0; % initialize the total cost

% choose the first node as the starting point
current_node = 1;
visited(current_node) = true;
```

```
for i = 1:N-1 % repeat N-1 times
    % find the minimum edge that connects the current node to an unvisited node
    min_dist = inf;
    min_node = 0;
    for j = 1:N
        if visited(j)
            % check all neighbors of the current node
            for k = 1:N
                if ~visited(k) && distmat(j,k) < min_dist
                    min_dist = distmat(j,k);
                    current_node=j;
                    min_node = k;
                end
            end
        end
    end

    % add the minimum edge to the minimum spanning tree
    MST(i,:) = [current_node, min_node];
    greenLine(nodes(current_node,:),nodes(min_node,:));
    % total_cost = total_cost + min_dist;
    visited(min_node) = true;
    current_node = min_node;
end

end

function min_path=two_Opt(nodes,path)
cost_2opt=getCost(nodes,path);
min_path=path;

for i=1:length(path)-3 % first edge
    for j=i+2:length(path)-1    % second edge will always be after the first edge

        path=swapEdges(path,i,j);
```

```
        c=getCost(nodes,path);
        if(c<cost_2opt)
            min_path=path;
            cost_2opt=c;


        else

            path=swapEdges(path,i,j);
        end
    end
end

end
% FUNCTIONS
function min_node=nearest_unvisited_node(current_node,unvisited_edges,nodes)
    min_dist=inf;
    min_node=1;
    for i=1:length(unvisited_edges)

            if(dist(nodes(current_node,:),nodes(unvisited_edges(i),:))<min_dist)
                min_dist=dist(nodes(current_node,:),nodes(unvisited_edges(i),:));
                min_node=unvisited_edges(i);
            end
    end
end

function distmat=MST(nodes)

distmat=zeros(length(nodes),length(nodes));
end

function plotPathDirection(nodes,path)

for i=1:length(path)-1
    p1=nodes(path(i),:);
```

```
    p2=nodes(path(i+1),:);  dp=p2-p1;
    %dp=p2-p1;quiver(p1(1),p1(2),dp(1),dp(2),"off",'black','filled','ShowArrowHead
    quiver3(p1(1),p1(2),p1(3),dp(1),dp(2),dp(3),"off","black");

end
end

function distmat=getDistmat(nodes)
distmat=zeros(length(nodes),length(nodes));
for i=1:length(nodes) % matrix of distances between two nodes
    p1=nodes(i,:);
    for j=1:i
        p2=nodes(j,:);
        distmat(i,j)=dist(p1,p2);
        distmat(j,i)=dist(p2,p1);
        if(dist(p1,p2)==0)
            distmat(i,j)=inf;
        end
    end
end



end

function path=swapEdges(path,p1,p2)
path=path(1:end-1);



path(p1+1:p2)=flip(path(p1+1:p2));
path(end+1)=path(1);
end

function cost=getCost(nodes,path)
cost=0;
for i=1:length(path)-1
    cost=cost+dist(nodes(path(i),:),nodes(path(i+1),:));
```

```
end
end

function out=isInArray(arr,a)
out=false
for i=1:length(arr)
    if(arr(i)==a)
        out=true;
    end
end
end

function out=areIndirectlyConnected(distmat,p1,p2)
out=false;
path=getPath(distmat,p1);
if(path(end)==p2)
    out=true;
end
end

function path=getPath(distmat,start)
distmat1=distmat;
p1=start;
path=p1;

for k=1:length(distmat1(1,:))
    for i=1:length(distmat1(:,1))

        if(distmat1(p1,i)==-1)
            if(length(path)>1 && i==path(end-1))

            else
                p1=i;
                distmat1(p1,i)=0;
                distmat1(i,p1)=0;
                path=[path;i];
```

```
                if(i==start)

                    return;
                end
                break;

            end
        end
    end
end

if(~path(end)==path(1))

    path(end+1)=path(1);
end
end

function out=isIncomplete(traversed,x)
out=true;
temp=traversed;


if(temp(x(1))==1 && temp(x(2))==1)
    temp(x(1))=temp(x(1))+1;
    temp(x(2))=temp(x(2))+1;

    for i=1:length(temp)
        if(temp(i)~=2)
            out=false;

            break;
        else

        end
    end
end
```

```
end

function greenLine(p1,p2,c)
deleteLine(p1,p2)
X = [p1(:,1) p2(:,1)] ;
Y = [p1(:,2) p2(:,2)] ;
Z = [p1(:,3) p2(:,3)] ;
plot3(X',Y',Z','green');
hold on
plot3(X',Y',Z','.',Color='black');
end

function redLine(p1,p2,c)
deleteLine(p1,p2)
X = [p1(:,1) p2(:,1)] ;
Y = [p1(:,2) p2(:,2)] ;
Z = [p1(:,3) p2(:,3)] ;
plot3(X',Y',Z','red');
hold on
plot3(X',Y',Z','.',Color='black');
end
function line(p1,p2)
X = [p1(:,1) p2(:,1)] ;
Y = [p1(:,2) p2(:,2)] ;
Z = [p1(:,3) p2(:,3)] ;
plot3(X',Y',Z','black');
hold on
plot3(X',Y',Z','.',Color='black');
end

function deleteLine(p1,p2)
plot([ p1(1) p2(1)], [p1(2) p2(2)],'LineStyle','none');
end
function f=min_nz(arr) % Return the index and value minimum non-zero value in a ma
m=inf; f=[1 1 0]; % [1 1] is the default value
for i=1:length(arr(:,1))
```

```
    for j=1:length(arr(1,:))
        if(arr(i,j)<m && arr(i,j)>0)
            m=arr(i,j);
            f=[i,j,m];

        end
    end
end
if (m==inf)
    f=[i,j,0];
end

end



function plotNodes(nodes)
clf;
scatter3(nodes(:,1),nodes(:,2),nodes(:,3),10,'blue','filled');
hold on
xlabel('km');ylabel('km');zlabel('km');
scatter3(nodes(1,1),nodes(1,2),nodes(1,3),250,'black','x');
text(nodes(1,1),nodes(1,2),nodes(1,3),'Depot','HorizontalAlignment','left','FontSi
for i=2:(length(nodes(:,1)))
    text(nodes(i,1),nodes(i,2),nodes(i,3),num2str(i),'HorizontalAlignment','left',

end
xlim([0 1]); ylim([0,1]); zlim([0,1]);
end


function out=isTraversed(traversed,a)% Checks if a node has already been traversed
out = false;
for i=1:length(traversed(:,1))
    if(a==traversed(i,:))
        out = true;
```

```
    end
end
end

function nodes=generateNodes(N,r)
rng(r); % Could be any random natural number
seed=rng;
nodes=rand(N,3);
rng(seed);
end
function dis=dist(a,b) % calculates the distance between the two points a and b on


dis=sqrt((b(1)-a(1))^2+(b(2)-a(2))^2+(b(3)-a(3))^2);



end

function path_sa = generatepath_sa(path)
% Generates a new solution by swapping two random cities
N = length(path);
i = randi(N);
j = randi(N);
while i == j
    j = randi(N);
end
path_sa = path;
path_sa([i j]) = path_sa([j i]);
end
function P_T=power_uav(V)
U_tip=200;%m/s
phi=1.225; %kg/m^3
A=0.79; %rotor disk area m^2
d_o=0.3;%Fuselage drag ratio
```

```
s_hat=0.05;%rotor solidity
% Aircraft forward sped in m/s
small_sigma=0.012;%profile drag coefficient
s=0.05;%rotor solidity
omega=400;%Blade angular velocity in radians/second
R=0.5;%Rotor radius in m
m=3.29;%mass kg
g=9.81; %gravity m/s^2
k=2;%Thrust to wight ratio T/W
v_o=sqrt((m*g)/(2*phi*A));%motor induced velocity m/s
P_o=(small_sigma/8)*phi*s*A*(omega^3)*(R^3);

P_i=(1+k)*(m*g)^1.5/sqrt(2*phi*A);
P_T=P_o*(1+3*(V^2)/U_tip^2)+ P_i*sqrt((sqrt(1+V^4/(4*v_o^4)))-V^2/(2*v_o^2))+0.5*d
end
```

Similar variations of the code has been used to calculate the time complexity, cost comparison, clustering, etc.