ENS 491-492 – Graduation Project

Final Report

Project Title:

Application of Unmanned Aerial Vehicles in Operations Research: Monitoring Wildfire

Group Members:

Kıvılcım Eylül Tuyun

Gürkan Talha Soylu

Ahmet Enes Sılacı

Onur Mert Gürses

Benan Aygen

Selin Ağan

Supervisor(s): İhsan Sadati

Date: 4.06.2023

1. EXECUTIVE SUMMARY

The problem is the difficulties while detecting forest fires because of the far distances and being hard reachable points. To address these difficulties UAVs can be used to detect forest fires. Objective in this project is forming the best routes for UAVs, taking pictures of forest areas and forming these routes by spending less energy and having less cost. Because of that, we are planning to scan critical areas and do this as cheaply as possible while selecting our UAVs and forming these routes. So, forest fires can be detected more easily and quicker. Thanks to this project, forest areas will be checked on a regular basis with minimum cost, which includes fixed and variable costs. Fixed costs are purchasing the UAVs, building the charging stations and maintenance cost for UAVs. Variable costs that are related to charging the UAVs. The detected forest fire will be put out before it gets bigger. By the means of this project, we will ensure continuity of the ecosystem and its health by preventing forest areas from burning and annihilation of all living creatures living in the forests. In this project, a mathematical formulation for VRP has been formed and then it is converted into Gurobi/Python and then solved for given data set. After that, savings algorithm is used to solve it in a shorter time. At the end of Gurobi solution and savings algorithm solution, it has been seen that the solution of savings algorithm is far from optimal solution than the Gurobi solution.

2. PROBLEM STATEMENT

There has been an increase in the occurrence of forest fires recently. The most important reasons for this situation are global warming and increasing pollution, as indicated in Figure 1. Early detection of forest fires is very important because of their adverse effects on the environment and living things.

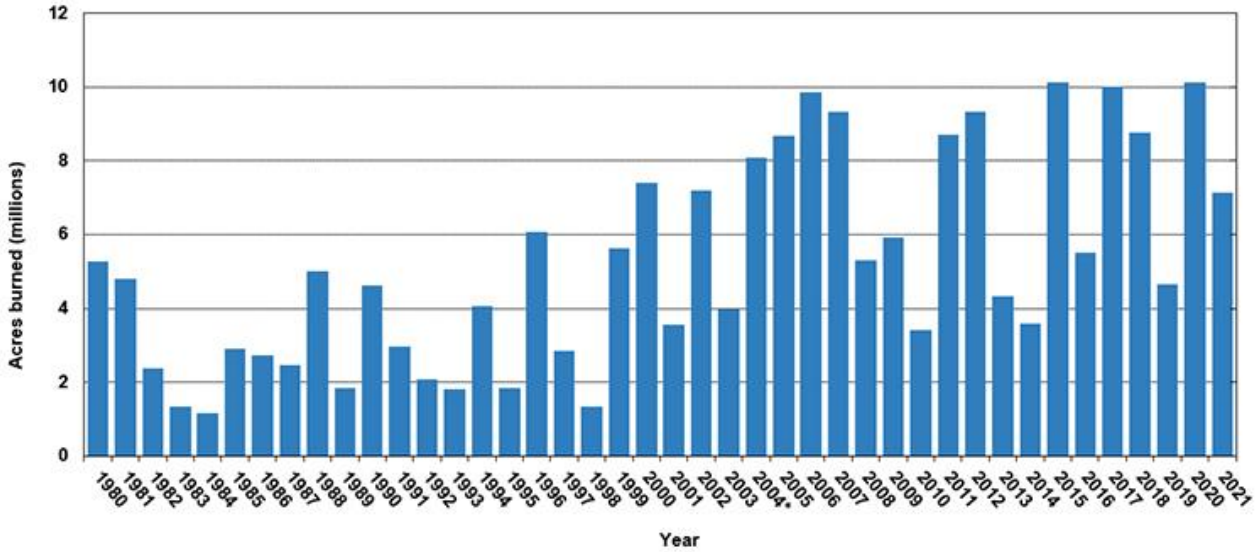

Figure 1 (Graph of annual number of acres burned in wildland fires between 1980-2021) https://www.iii.org/fact-statistic/facts-statistics-wildfires

Figure 2 states that CO2 emissions produced by forest fires started to increase after the 1850s and there is a significant increase after 1950. This situation endangers the lives of living things and the lungs of animals and people are damaged due to the toxic gas released, the resulting CO2 gas interacts with the humidity in the air and causes acid rain, which pollutes agricultural areas and clean water resources. However, because forest areas are typically located far from populated areas and are difficult to realize, early identification by people is not always possible. As a result of these problems, interfering with fire can be very difficult and it will spread too much until someone tries to put out the fire and until the fire is dealt with, a lot of damage can occur.

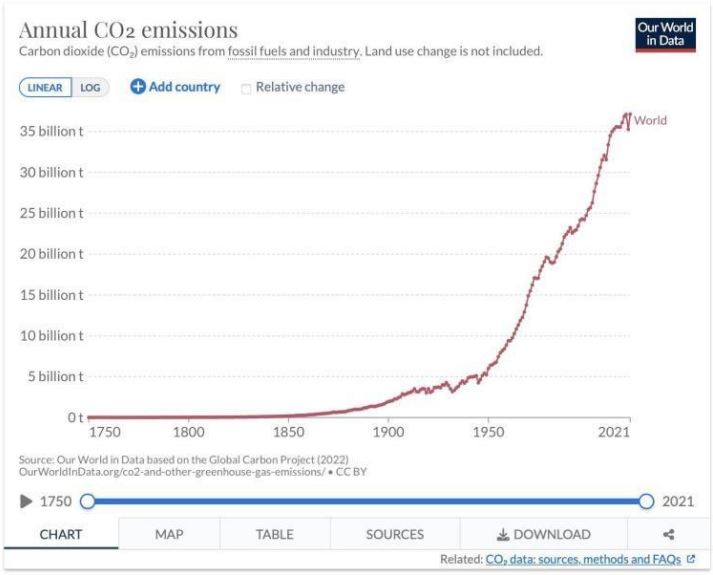

Figure 2 (Graph of annual CO2 emissions between 1750 2021)

https://ourworldindata.org/co2-emissions

Many early detection methodologies have been proposed to prevent forest fires as much as possible and to intervene early to minimize damage. UAV (Unmanned Aerial Vehicle) scanning of the most fire-prone areas is one of the finest approaches to identify forest fires. UAVs can be a much more effective solution in case of fire as they are a much faster method than detecting by smoke. In addition, Helicopters are sent to put out the fire, but sending helicopters can be more of a waste of time and money because there are UAVs specifically made for the job. A methodology based on UAVs was proposed by Kyuchukova (2019). According to this methodology, two UAVs are utilized for wildfire detection. First, a UAV which flies at high altitudes scans the forest from above to detect suspicious activity in the forest. If detection happens, the UAV sends a signal to the headquarters to verify the detection, then a UAV equipped with a thermal camera comes to the area. If the second UAV also verifies, the firefighting crew departs. However, it is not a wise economic strategy to use more UAVs than necessary and to go greater distances. Instead, by utilizing fewer UAVs and choosing routes that cover more ground, the project's cost can be decreased.

In addition to this methodology, another approach relies on the use of distant sensors (Bosch Global, 2022). Sensors which are hung on the trees are powered by solar power and constantly measure the environment. The sensors talking about in this methodology warn the emergency services when they notice a forest fire by sending out a signal. Since they are immobile, these sensors can only detect fire in very small, defined areas if it begins inside their detection range. If the fire is detected by UAVs, the fires could be realized and in addition to that more areas can be scanned in a shorter time.

2.1. Objectives/Tasks

Our objectives in this project are to create the most suitable routes for UAVs that take pictures of forest areas, and to create these routes with less energy and less cost, thus protecting the continuity and health of the ecosystem by preventing the burning of forest areas and the extinction of all living things living in the forests. In the plan we prepared and the criteria while doing this project, we aim to scan critical areas while choosing our UAVs and creating these routes and to do this as cheaply as possible. As a result of our project, forested areas will be checked regularly with minimum cost including fixed and variable costs. Fixed costs that we mentioned at this point are the costs of purchasing UAVs, installing charging stations and UAV maintenance and variable costs that are related to charging the UAVs. The budget allocated for fixed and variable costs and its details are examined in the scientific/technical developments section of the article. After determining the objectives and criteria, the UAVs that can be used were examined. Many different features of the UAVs, such as their costs, the widest reach, battery times, were compared. Then, the mathematical modeling that will create the optimal solution for the project was created and the Gurobi implementation to represent this mathematical modeling was created. Visible and invisible solutions can be obtained by testing the generated Gurobi implementation. As it is mentioned before, the sets, parameters and the decision variables can be seen in table 1. At first look, mathematical model seems like a VRP. However, in this problem there is battery consumption and time window constraints which differentiates it from VRP. Moreover, there are no such constraints as demand and vehicle capacity which are used in VRP. As a matter of effect, our problem can be considered as VRP- like problem at best.

Problem Formulation

In what follows, we provide the mathematical formulation of the problem. Table 1 summarizes the mathematical notion.

Sets:

V = Set of all locations

V0 = Set of all locations and the depot, V0 = V U {0}

Parameters:

tij = Travel time from node i to node j

dij = Distance from node i to node j

Si = Service time at node i

F = A sufficiently large constant

Dmax = Maximum tour duration

Q = Battery capacity

[ei, li] = Time window associated with node i

r = UAV battery consumption rate

Decision variables:

xij = 1 if arc (i,j) is traversed by an UAV ; 0 otherwise, ∀ j∈V0 and ∀ i∈V0

yi = Battery state of charge at node i, ∀ i∈V0

Ti = Service start time of node i by one of the UAVs, ∀ i∈V_0

Table 1

Model:

$\text{Min}(\sum_{j\in V_0} \sum_{i\in V_0}(d_{ij} * x_{ij}) + F * (\sum_{i\in V} x_{i0})$ (1)

Subject to:

$\sum_{j\in V0, j\neq i} x_{ij} = 1, \forall i \in V$ (2)

$\Sigma_{i\in v_0, i\neq j} x_{ij} = 1, \forall j \in V$ (3)

$e_i \leq T_i \leq l_i, \forall i \in V_0$ (4)

$T_i + (t_{ij} + S_i) - l_0(1 - x_{ij}) \leq T_j, i \in V_0, j\epsilon V$ (5)

$$y_i \leq y_i - (r.d_{ij})x_{ij} + Q(1 - x_{ij}) \forall i \in V_0, \forall j \in V, i \neq j \quad (6)$$

$$y_i \geq \Sigma_{i \in V, i \neq j}(r.d_{ij})x_{ij} \forall i \in V \quad (7)$$

$$y_0 = Q \quad (8)$$

$$T_i + S_i + t_{i_0} \leq D_{max}, \forall \text{i} \in \text{V} \quad (9)$$

$$x_{ij} \in \{0,1\}, \forall i \in V_0, , \forall j \in V_0 \quad (10)$$

$$y_i \geq 0, \forall i \in V_0 \quad (11)$$

$$T_i \geq 0, \forall i \in V_0 \quad (12)$$

The objective function (1) minimizes the total distance traveled by UAVs and minimizes total number of UAVs which are going to be used by multiplying them with a large constant F. Constraints (2) and (3) are the flow balance constraints that guarantee that each node is visited exactly once. Constraints (4) establish the service time window restrictions. The service start time of the node cannot be smaller than the earliest service time of the node and cannot be greater than the latest service time of the node. Constraints (5) keep track of the time at nodes (and the depot). Constraints (6) keep track of UAV charge level at each node while constraints (7) guarantee that the UAV never runs out of charge. Constraints (8) fill up the UAV when the UAV visits a recharging station. Constraints (9) requires that the total travel time, i.e., the time between the departure and return of each UAV be restricted by a maximum tour duration Dmax. Finally, constraints (10)– (12) define the decision variables. For the mathematical formulation in Python and Gurobi solver are used to solve problem for optimality. The Gurobi implementation can be seen in Appendix A.

2.2. Realistic Constraints

Economic: Available budget (if no budget is available existing resources is the constraint); maintenance cost; cost of current similar products on market etc.

Environmental: Pollution (air, noise, greenhouse gas, water etc.); global warming; energy saving; waste etc.

- The UAVs run out of battery when they go upper but in the territory that we are going to use, there will be trees so the UAVs cannot fly low.
- Distances between the points that should be visited by UAVs
- Weather conditions such as windy weather, foggy weather, rainy weather…
- There should be areas that UAVs could land and take off

Social:

- There could be personal privacy issues, because the UAVs should scan the areas that people present, some people get annoyed by this situation, and do not want to be seen by the cameras.
- There will be people who will charge these UAVs. They will have worker rights.

Health and safety:

- The UAVs could fall out, and could initiate the fire, we should not use UAVs which work with gasoline such that in a falling situation, it could initiate fire with higher probability.
- The UAVs could kill the animals while flying and could injure the plants.
- The UAVs could fall out due to some emergency circumstances and injure people.

Manufacturability:

- We should buy UAVs which are built with Carbon fiber-reinforced composites (CRFCs), Thermoplastics such as polyester, nylon, polystyrene, etc., Aluminum, and have Lithium-ion batteries.

Sustainability (social, economic, and environmental):

- The maintenance costs per month, and the operations costs of UAVs should not be too much, the UAVs could be built with recyclable materials

3. METHODOLOGY

We modified savings algorithm to solve our problem. Savings algorithm is a heuristic algorithm which gives a feasible enough solution but not the optimal solution. On the other hand, the time that algorithm requires is much smaller. That is why this algorithm is widely used in real life by cooperations. The steps of this algorithm can be seen in Table 2 and visualization of steps can be seen in Figure 3. We implement the mathematical formulation in Python. This implementation can be seen in Appendix B.

1) Take depot as starting point and index it as origin.

2) Form subtours as “depot – node – depot” for all nodes to be visited

3) Compute savings with $S_{ij} = c_{io} + c_{oj} – s_{ij}$ for all nodes to be visited

4) Order S_{ij} values from largest to smallest

5) Select the node pair i,j which gives highest savings value such that

- i and j are not in same subtour
- i is the last node in its subtour and j is the first node in its subtour (i.e., subtours contain (i,0) and (o,j) arcs

6) Form a larger subtour by connecting the subtours containing i and j

- Add the arc (i,j)
- Delete the arcs (i,0) and (o,j)

7) If the number of subtours is equal to 1, stop. Else, go to step 5 and continue.

Table 2

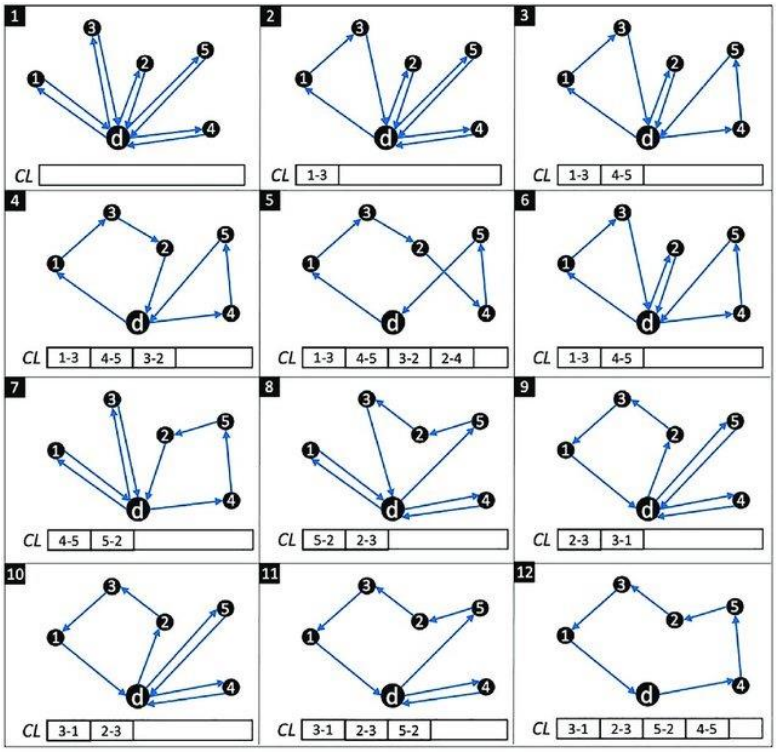

Figure 3 (Steps of Savings algorithm)

2-Opt is used as local search algorithm which aims for solving optimization problems in the framework of Travelling Salesman Problem. TSP finds the shortest path between the beginning city and a group of cities while stopping in each city and finally returns to the beginning city.

1. Initial Solution: Starting with a first solution which is taken random or defined route visits all cities.

2. Iterative Improvement: 2-Opt incrementally advances the solution by swapping pairs of edges for likely decreasing the total distance of the route.

3. Edge Swapping: In each iterative improvement, two edges are taken for swapping. In order to not to cause a self-intersecting route selected two edges should not have common endpoints.

4. Evaluation: In this step, until now total distance of taken route is calculated.

5. Swap Evaluation: The distance of two swapped edges is calculated. Swap will be made If the distance has shorter total distance.

6. Termination: Until last condition occurs or there are no more improvements are possible, the algorithm advances the iteration of improvement process.

The 2-Opt heuristic algorithm's goal is to find pairs of edges along the route, after swapping, it results in a reduced total distance. With the iteration of doing edge swapping, the algorithm purposes to get close to locally optimal solution.

Due to 2-Opt heuristic is a local search algorithm, it only recognizes small local changes at each step and does not precise of finding the global optimal solution. But it makes satisfactory results for many examples of optimization problems including TSP.

4. RESULTS & DISCUSSION

ID	Latitude	Longitude	Earliest	Latest	Service Time
0	41,19693575	28,92041069	0	300	30
1	41,18583399	28.988378163171728	0	150	30
2	41,18647977	28.988378397620913	0	210	30
3	41,1739541	28.956458917289407	0	210	30
4	41,17124044	28.94530653114603	120	300	30
5	41,19382879	28.904449453908022	0	210	30
6	41,20995893	28.880568349249245	120	270	30

7	41,23682961	28.887747577943788	0	210	30
8	41,22222203	28.870590981051123	60	210	30
9	41,20921018	28.991477080516997	0	90	30
10	41,18014102	29.01686307076339	0	135	30
11	41,16968545	29.0014124695151	60	225	30
12	41,22297041	28.972317621932433	60	210	30
13	41,19437101	28.90312436123621	0	150	30
14	41,23145654	28.908315156607046	0	210	30
15	41,22187472	28.945238724183334	0	210	30

Table 3

First of all, we present the Gurobi solution. To test the performance of the algorithm, we used a specific dataset which can be seen in Table 3. In the dataset, there are 6 columns which are, ID, latitude, longitude, earliest, latest and service time. ID represents nodes in the area, the first ID which is 0 corresponds to depot. Earliest and latest represents the time interval that the nodes need to be visited and the final column which is service time shows the time spent on that node that is 30 minutes.

As a group, we ran the Gurobi implementation in Python for an hour and at the end the solution which can be seen in Figure 4 is gathered. As it can be seen, the battery capacity decreases as the UAV travels. For example, when we examined the route X[0,5] -> X[5,13] -> X[13,0], the battery capacity decreases which is represented with "Y" is reduced. In addition to that, when the UAV returns to the depot, the battery capacity becomes full as intended.

In the Figure 4, the cost of the solution can be seen as 30.044. 44 of this cost corresponds to distance and the remaining 30.000 represent the total price of the UAV's to be used. Moreover, the remaining cost can be checked from the value of F * number of routes. In this solution, F is equal to 10.000 and number of routes is equal to 3.

In the solution, it is expected for each node to be visited with in the specified time interval. For instance, the node ID 10 needs to be visited within [0,135] minutes. In figure 5, it can be seen that T [10] is 135^{th} minute which corresponds to visit time of the node ID 10.

At the end of the Gurobi solution the gap value in Figure 4 is 66.5459%. The gap could be lowered to this number so far, this number would be 0 if the optimal solution is found at the problem, but this is impossible thing to do.

```
Best objective 3.004411190104e+04, best bound 1.005399092417e+04, gap 66.5359%
[41.2218747202323, 28.945238724183334] [41.2314565394876, 28.90831515660705]
x[0,5] 1.000000
x[0,9] 1.000000
x[0,12] 0.999993
x[1,10] 1.000000
x[2,1] 1.000000
x[3,4] 1.000000
x[4,0] 1.000000
x[5,13] 1.000000
x[6,0] 1.000000
x[7,8] 1.000000
x[8,6] 1.000000
x[9,2] 1.000000
x[10,11] 1.000000
x[11,3] 1.000000
x[12,15] 1.000000
x[13,0] 0.999993
x[14,7] 1.000000
x[15,14] 1.000000
y[0] 1.020000
y[1] 0.137661
y[2] 0.138540
y[3] 0.039768
y[4] 0.027761
y[5] 0.029306
y[6] 0.027761
y[7] 0.073856
y[8] 0.047329
y[9] 0.945335
y[10] 0.107473
y[11] 0.086188
y[12] 0.163885
y[13] 0.027761
y[14] 0.096141
y[15] 0.136124
T[1] 98.067308
T[2] 68.000820
T[3] 210.000000
T[4] 266.725818
T[5] 31.277268
T[6] 220.297511
T[7] 156.810544
T[8] 188.817227
T[9] 35.648258
T[10] 135.000000
T[11] 176.488458
T[12] 60.000052
T[13] 61.394126
T[14] 125.124709
T[15] 92.100095
```

Figure 4

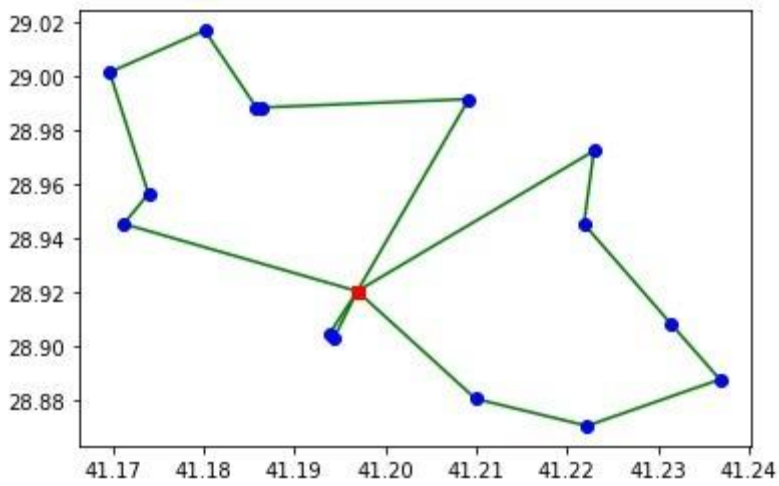

Figure 5 (Visualization of Resulting Route)

As it can be seen in Figure 5, there are three routes for visiting all nodes in the problem with respect to charge capacity of the UAVS, and time intervals of nodes. This indicates that at least three UAVs should be used to solve this problem because there are three routes in the solution.

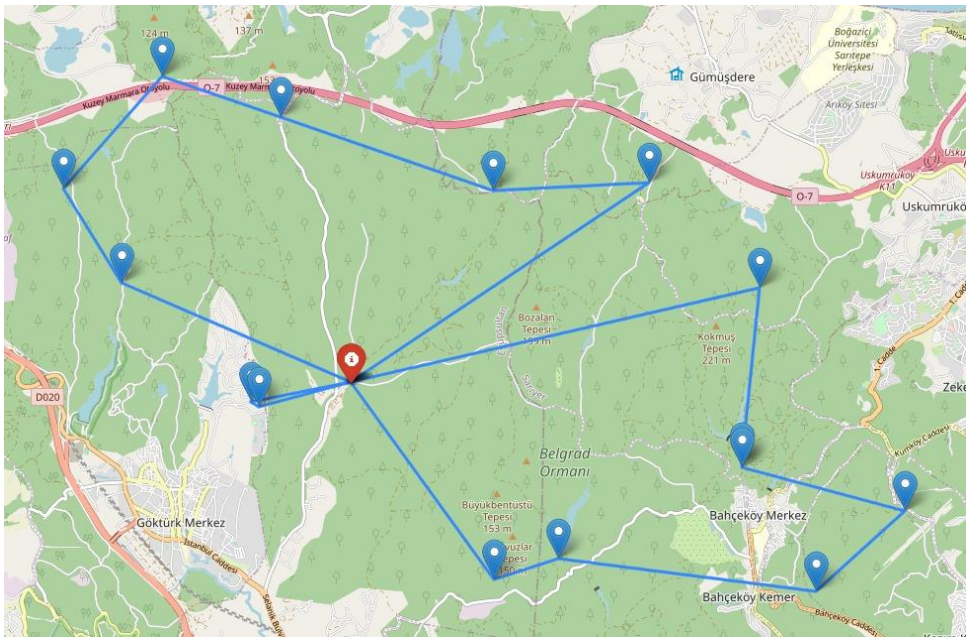

Figure 6 (Visualization of Resulting Route by map)

Figure 6 is another version of Figure 5, only difference is Google maps used in Figure 6 but not in Figure 5. To do so, Python's "folium" package is utilized.

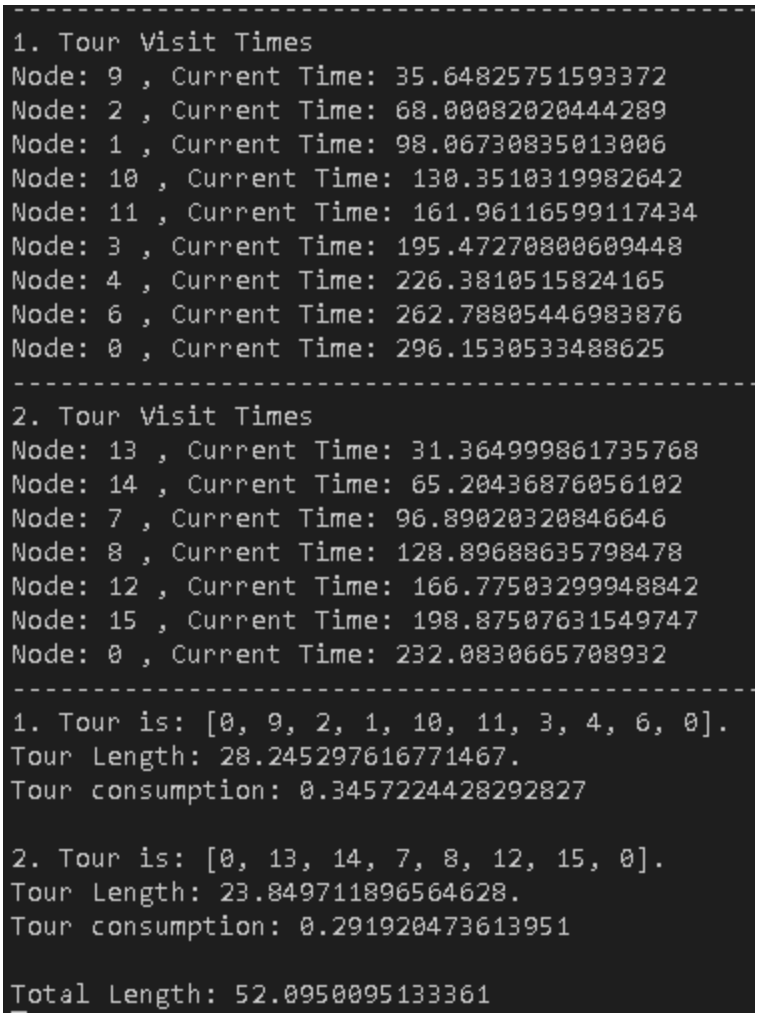

Figure 7 (Savings Output)

When it comes to Savings algorithm, the output can be seen in Figure 7. The solution indicates that the cost for the tours is 52.095, which is not the optimal solution. When it is compared with the Gurobi solution it is even worse than it, but on the other hand it gives the solution in seconds such that it is used in real life even if Gurobi solution is better than savings' solution. Another aspect of this solution is time windows, it also looks at time windows since the UAVs' delivery time cannot exceed specified time intervals. For example, node ID 15 needs to be visited within [0,210] minutes in the solution it is visited at 198.875^{th} minute which makes the solution valid.

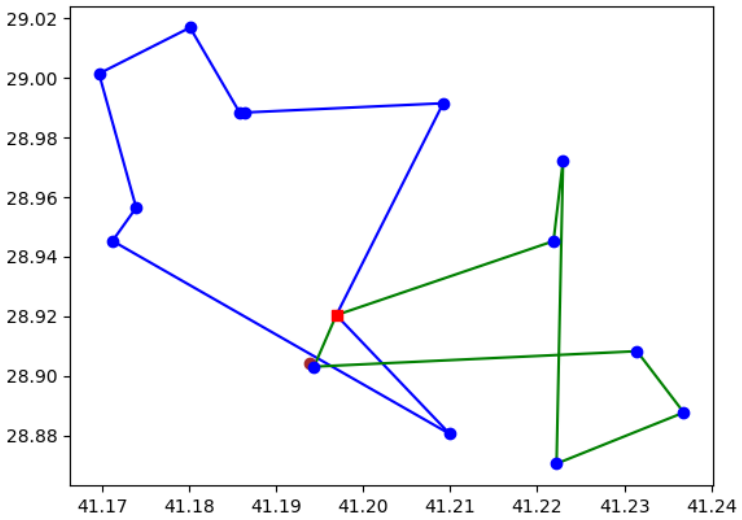

Figure 8 (Visualization of Saving Solution)

As it can be seen in Figure 8, there are 2 routes for this algorithm. When we compare the routes for the nodes are distributed more uniformly.

After utilizing savings algorithm on first test data, we performed 10 tests with 10 given instances. Each instance consists of 100 customers with their coordinates and time window values. In addition, 2-Opt algorithm is utilized on the resulting tours of each instance to see if we can further optimize tours. The results can be seen in Table 4.

Instance Name	Savings Solution **without** 2-Opt		Savings Solution **with** 2-Opt		GAP %
	# V	OFV	# V	OFV	
c101_21	2	148	2	148	0,00%
c109_21	2	154,55	2	154,55	0,00%
c201_21	0	0	0	0	0,00%
c208_21	0	0	0	0	0,00%
r101_21	0	0	0	0	0,00%
r112_21	1	63,76	1	63,76	0,00%
r201_21	2	168,57	1	168,57	0,00%
r211_21	1	265,05	1	265,05	0,00%
rc101_21	0	0	0	0	0,00%
rc108_21	2	156,82	2	156,82	0,00%
Average	1	95,64	0,9	95,64	0

Table 4

As it can be seen from Table 4, implementation of 2-opt algorithm did not result in improvement in tour lengths. The reason of this lies in the algorithm. Since we are dealing with time windows, swapping the edges are far more complicated. Each swapping needs to fit time window of both node of the edge and also the whole route needs to fit the time window restriction.

In the Table 4, it can be seen that for some of the instances, the number of vehicles utilized is less that one can expect and for some instances there are no routes neither. To make this clear, I will briefly explain the logic of algorithm. Algorithm controls every pair of nodes and their calculated savings value, in a descending order. While doing so, it also checks the time windows. For a route to be generated, the first node needs to fit into its time interval. So, basically, if you have 2 nodes that can be used as starting points, you have potential 2 routes. To check whether nodes are fit to be used as starting node, we basically calculated the exact time when they are visited by depot and check if fits into their time interval. For the instances "c208_21", "c208_21", "r101_21" and "rc101_21"; there were no candidate nodes to be used as starting point. Since we are not able to locate any starting point to build upon, the algorithm could no build any routes for these instances. The same logic applies in other instances. For the sake of example, let us examine result of instance "r211_21". When we run our code, the potential nodes to be used as beginning is printed as in Figure 9. For this instance, we are only able to use node with ID 2 as beginning node in our routes which means we are able to calculate only one route. The resulting route with each node visit time and time window can be seen in Figure 10. As it can be seen from Figure 10, each node is visited exactly between their corresponding time window and after a certain point we return to depot because in this instance, the capacity of UAV was 265.71 and the tour costs 265.05 in total.

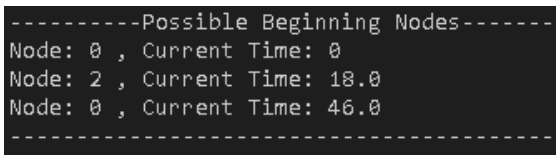

Figure 9

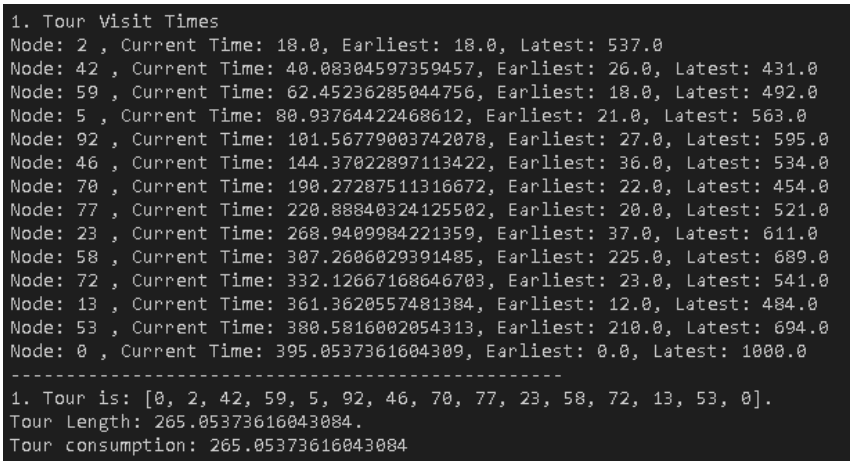

Figure 10

5. IMPACT

Scientific Impacts:

- VRP makes gathering data more efficiently possible for UAVs, which develops the management ability to cover forest fire zones and collect accurate and complete data.
- The optimized collected data provides scientific modelling and analysis which supplies better understanding the development of fire management actions.

Technological Impacts:

- VRP has a crucial role in advancing UAV routes and paths all along forest fire surveillance which enhances their efficiency and provides real time data communication.
- Using thermal imaging and infrared cameras can be useful as new technologies for UAV's to catch fire hotspots precisely and evaluate the intensity of the fire.

Socio-Economic Impacts:

- About resource allocation and evacuation management, fire management strategies can be developed with VRP-driven UAV surveillance
- VRP increases cost effectiveness by optimizing flight routes by decreasing the need for comprehensive operations.
- Using of UAV's enhances safety of firefighters and their protection against hazardous conditions by supplying real-time data.

Ecological Impacts:

- Through detection and suppression of forest fires, UAV surveillance helps to reduce environmental harm and aids in the preservation of forests and natural resources.

In General, VRP for UAV's in forest fire surveillance accomplishes in advanced technological capabilities, socio-economic results such as development in fire management reducing efficiency, enhanced safety for workers in fire area and sustainability in environmental protection.

The use of Vehicle Routing Problem (VRP) for Unmanned Aerial Vehicles (UAVs) in forest fire surveillance offers fascinating possibilities for inventiveness and business endeavors.

Innovative Aspects:

- By combining VRP algorithms with UAV technology, forest fire surveillance is renewed and improved data collecting and real-time monitoring are made possible.
- Modern remote technologies, for instance, thermal imaging and infrared cameras, improve the precision and dependability of detecting and evaluating fires.
- Advancements in technology about fire behavior predicts some evaluations where fire will be spread out and reduces risks.

Commercial/Entrepreneurial Aspects:

- The market for businesses that specialize in UAV platforms, sensor technologies, and data processing are created by the demand for UAV-based forest fire monitoring solutions.
- Businesses that provide VRP algorithms that are optimized and software solutions designed specifically for UAV-based fire monitoring can become major shareholder in the market.
- When UAV surveillance systems are commonly used firms that provides fire management services contains real-time data analysis risk evaluation and decision-making support.
- Initiatives can concentrate on optimizing resources, efficient solutions and valuable services such as data visualization and predictive analysis.

Innovations in VRP for UAVs in forest fire surveillance promotes technological advancements. Moreover, they generate creative solutions in fire management services and safety for both human and environment.
Some Freedom-to-Use (FTU) topics associated with the use of VRP for UAVs in the observation of forest fires. These problems may occur as a result of a number of things which can be described as legal and regulatory constraints, privacy concerns, and intellectual property rights.

Legal and Regulatory Constraints:

- The usage of UAVs for surveillance purposes is subject to specific regulations and permissions enforced by aviation authorities and local authorities. These regulations may limit the freedom to use UAVs in certain areas or impose limitations such as flight altitudes, paths, or operational hours.
- Sometimes forested areas can have more regulations governing UAV operations, especially during fire situations, to ensure the health and safety of firefighter and

the public. Compliance with these regulations can be very important to avoid legal issues and ensure responsible UAV usage.

Privacy Concerns:

- UAV surveillance, raises privacy concerns for capturing images or collecting data in residential or private areas even if it is for forest fire monitoring. The need for fire surveillance and privacy protection should be balanced for addressing potential FTU conflicts.
- Privacy regulations and guidelines must be considered to protect individuals' rights and prevent unauthorized use or spreading of collected data.

Intellectual Property Rights:

- In innovative algorithms, software solutions, or sensor technologies developed for UAV surveillance drones could deal with patent, trademark, copyright, trade secret for the companies that are involved.
- Acquiring and utilizing proprietary technologies should require license agreements or partnerships, especially the ones which has posing FTU challenges for entrepreneurs or the companies that wants to commercialize VRP solutions. Canalizing FTU issues requires a extensive understanding of relevant legal frameworks, obedience to regulations, don't get into peoples' private life, thus not to deal with privacy issues, and proper licensing with intellectual property owners. It is important to have agreement between the freedom to use UAVs for forest fire surveillance and respecting legal, regulatory and privacy considerations to ensure responsive and legal implementation of the VRP solutions.

6. ETHICAL ISSUES

We have planned to use the UAVs whose power comes from electricity but on the other hand the oil engine UAVs could be used in the project too and it would be more efficient in terms of covering more area. But on the other hand, there would be some ethical issues regarding the environment and global warming.

The UAVs' pilots could interrupt the privacy of their personal life with the choices that he made while using the UAVs, if that would be the situation there would be some legal issues.

In terms of batteries there could be some leaks which are harmful for the environment. These batteries could be toxic under specific conditions, if there is any problem. If the batteries have dark colors, they could heat too much and maybe burn, then it would be harmful for the environment.

Data security is another issue. The collection, storage, and transmission of data during UAV surveillance pose risks of data breaches or unauthorized access. It is crucial to implement robust data security measures to protect sensitive information and maintain the trust of stakeholders.
Accountability and responsibility is another issue. Clear accountability structures should be in place to address any malpractice or negligence in UAV operations. Organizations and individuals involved in UAV-based forest fire surveillance should be held responsible for their actions and adhere to ethical standards.

7. PROJECT MANAGEMENT

In Figure 11, the initial project schedule can be seen. In the beginning, we plan to collect data, write mathematical model of our proposed solution and the Gurobi implementation of mathematical model. The initial plans are not changed throughout the project until the implementation part. When we get to the implementation, time schedule was modified which can be seen in table 5. There were only small implementation changes in methodology but the main solution approach remains intact.

TASKS	MONTH 1	MONTH 2	MONTH 3	MONTH 4	MONTH 5	MONTH 6	MONTH 7	MONTH 8	MONTH 9
Literature Review									
Colleting Data 1									
Draft Proposal									
Project Proposal									
Progress Report-1									
Collecting detailed data									
Progress Report-2									
Mathematical Modelling									
Algorithm and Coding									
Final Report									
Draft Final Report									
Presentation									

Figure 11

Tasks	May	June
Savings Algorithm		
Draft Final Report		
2-Opt Algorithm		
Final Report		
Presentation		

Table 5 (Project Schedule Month-by-month)

Throughout the project, there were some issues regarding “mathematical modelling” and “algorithm and coding”. Briefly, there were faults and lacks in our mathematical model which resulted in issues in coding and solution. Because of these minor issues, we experienced approximately 1.5-week delay but after all, we corrected our faults and thus, managed to meet our initial planning expectations for consequent steps.

We have learned that managing a project requires continuous learning. At every step of the project, we have encountered many different methodologies. Even though you may be stuck at some point, there are always possible ways to move forward. In addition to that, we have learned how to be flexible. Projects often encounter unforeseen challenges or changes and being able to adapt those challenges reduces possible delays throughout the project.

8. CONCLUSION AND FUTURE WORK

As a conclusion, this project indicates that the early detection of forest fires via UAV surveillance can be performed. The results of the Gurobi implementation with respect to dataset shows that this problem is a feasible problem and it can be performed for multiple areas simultaneously.

The gap in the Gurobi solution shows that it is still possible to improve the algorithm. Current limitation in the algorithm is that it requires more time to decrease gap to more feasible values in the solution which can be cause some problems in real life applications since forest fires detection should be fast in real life. By improving mathematical model and thus, Gurobi implementation, this gap and time limitation can be decreased. In addition, it is shown that with help of Savings algorithm, time can be decreased to seconds but on the other hand it results in increase in cost. It may be possible to overcome “time – cost” trade-off with further improvements.

Afterwards, Two-opt algorithm could be used to decrease the cost of routes generated by Savings algorithm and Gurobi solution. When the improvement of algorithm is completed and the solutions become feasible enough, the real-life trials can be conducted with UAVs in remote, forest areas.

9. APPENDIX

Appendix A:

```
import numpy as np
import pandas as pd
from gurobipy import Model, GRB, quicksum
import sys
import math
import matplotlib.pyplot as plt
import ctypes
import folium

status_code = {1:'LOADED', 2:'OPTIMAL', 3:'INFEASIBLE', 4:'INF_OR_UNBD', 5:'UNBOUNDED'}
```

```
#Function for drawing routes on real map
def drawMap(active_arcs, positions):
    my_map = folium.Map(location = positions[0], zoom_start = 12, max_zoom = 20, min_zoom=10)

    for i,j in active_arcs:

        if i == 0:
            folium.Marker(location = [positions[i][0], positions[i][1]], icon=folium.Icon(color='red')).add_to(my_map)
        else:
            folium.Marker(location = [positions[i][0], positions[i][1]]).add_to(my_map)

        folium.PolyLine([[positions[i][0], positions[i][1]], [positions[j][0], positions[j][1]]]).add_to(my_map)

    #my_map.show_in_browser()
    my_map.save("Routes.html")

# Haversine distance function
def haversine(lat1, lon1, lat2, lon2):
    # Convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(math.radians, [lon1, lat1, lon2, lat2])
    # Haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = math.sin(dlat/2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon/2)**2
    c = 2 * math.asin(math.sqrt(a))
    # Earth's radius in kilometers
    r = 6371
    return c * r


df = pd.read_excel('UAV_Test_Data.xlsx')
row_index = 0
a = df.at[row_index, 'Latest']

nodes = list(range(1,df.shape[0]))   #list of positions to be visited
nodes_with_depot = [0]+ nodes        #list of positions with depot
only_depot = [0]                     #only depot

speed = 64.8                                #speed of UAV
energy_consumed = 0.01224                   #energy consumption rate per km
Q = 1.02                                    #battery capacity

service_time= list(df.iloc[0:,5])           #service time
latest_time = list(df.iloc[0:,4])           #list of latest times
earliest_time = list(df.iloc[0:,3])         #list of earliest times
max_tour_d = df.iloc[0,4]                   #max tour duration
latitude = list(df.iloc[0:,1])              #list of longitudes
longitude= list(df.iloc[0:,2])                      #list of latitudes
F = 10000                                   #Sufficiently large constant
```

```
positions = []
for i in range(len(df)):
  positions.append( (df['Latitude'][i] , df['Longitude'][i]) )

dist = np.array([[haversine(latitude[i], longitude[i], latitude[j], longitude[j])
                 for j in nodes_with_depot]
                 for i in nodes_with_depot]
                ) #Build distance matrix

travel_time = (dist/speed)*60 #travel time matrix calculated based on formula x = t*v

#Plot coordinates
positions = [(latitude[i], longitude[i]) for i in range(len(latitude))]
plt.plot(positions[0][0], positions[0][1], c='r', marker='s')          #Add depot
for i in positions[1:] : plt.scatter(i[0], i[1], c='b')                #Add nodes
plt.show()


#Create a new model
vrp = Model("vrp_model")

#Create variables
x = vrp.addVars(nodes_with_depot, nodes_with_depot, lb = 0,vtype = GRB.BINARY,name = 'x') #1 if arc(i,j) is
used
y = vrp.addVars(nodes_with_depot, lb = 0, ub = GRB.INFINITY ,vtype = GRB.CONTINUOUS, name = 'y')
             #battery state of charge at node i
T= vrp.addVars(nodes_with_depot, lb = 0, ub = GRB.INFINITY, vtype = GRB.CONTINUOUS, name = 'T')
                                    #service start time of node i vy one off the UAVs

#Flow balance constraints
vrp.addConstrs(((quicksum(x[i,j] for j in nodes_with_depot if j != i) == 1) for i in nodes), name = '2')

vrp.addConstrs(((quicksum(x[i,j] for i in nodes_with_depot if i != j) == 1) for j in nodes), name = '3')


#Time window constraints
vrp.addConstrs((earliest_time[i] <= T[i] for i in nodes_with_depot), name = '5.1')

vrp.addConstrs((T[i] <= latest_time[i] for i in nodes_with_depot), name = '5.2')


#Model Speed-up Constraint
vrp.addConstr(quicksum(x[0,i] for i in nodes) >= 1 )

#Time related constraints
for i in nodes_with_depot:
    for j in nodes:
        if i!=j:
```

```
            vrp.addConstr((T[i]+travel_time[i,j]+service_time[i]-a*(1-x[i,j]) <= T[j]), name ='6')

for i in nodes:
    vrp.addConstr(T[i] + service_time[i]+travel_time[i][0] <= max_tour_d, name='10')

#Capacity constraints
for i in nodes_with_depot:
    for j in nodes:
        if i!=j:
            vrp.addConstr(y[j] <= y[i]-(energy_consumed*haversine(latitude[i], longitude[i], latitude[j],
longitude[j])*x[i,j])+Q*(1-x[i,j]) , name='7')

vrp.addConstrs((y[i] >= quicksum((energy_consumed*haversine(latitude[i], longitude[i], latitude[j],
longitude[j]))*x[i,j] for i in nodes if i != j) for i in nodes), name = '8')

vrp.addConstr(y[0] == Q , name = '9')


#set objective function and time limit
vrp.setObjective( quicksum(dist[i][j]*x[i,j]
                           for i in nodes_with_depot
                           for j in nodes_with_depot if i != j) + F*quicksum(x[0,i] for i in nodes),
                GRB.MINIMIZE)

vrp.setParam("TimeLimit", 3600)
vrp.update()
vrp.optimize()

status = vrp.status

#Visualizing the result if it is optimal
A = [(i,j) for i in nodes_with_depot for j in nodes_with_depot]
if status !=3 and status != 4:

    active_arcs = [a for a in A if x[a].x > 0.99]

    for i, j in active_arcs:
        plt.plot([positions[i][0], positions[j][0]], [positions[i][1], positions[j][1]], c='g', zorder=0)


    print([positions[i][0], positions[i][1]], [positions[j][0], positions[j][1]])
    plt.plot(positions[0][0], positions[0][1], c='r', marker='s')
    for i in positions[1:] :
        plt.scatter(i[0], i[1], c='b')
    plt.show()

    drawMap(active_arcs, positions)

    for v in vrp.getVars():
```

```
        if vrp.objVal < 1e+99  and v.x!=0:
            print('%s %f'%(v.Varname,v.x))

else:
    print(status_code[status])
    ctypes.windll.user32.MessageBoxW(0, "Solution is INFEASIBLE", "OUTPUT", 1)
```

Appendix B:

```
COLORS = list(matplotlib.colors.cnames.values())
COLORS2 = mcolors.BASE_COLORS
mcolors.BASE_COLORS.pop('w')
COLORS2 = list(COLORS2.values())

SPEED = 1.0
ORIGIN = 0
CAP = 79.69
RATE = 1.0

# Haversine distance function
def haversine(lat1, lon1, lat2, lon2):

    # Convert decimal degrees to radians
    lon1, lat1, lon2, lat2 = map(math.radians, [lon1, lat1, lon2, lat2])

    # Haversine formula
    dlon = lon2 - lon1
    dlat = lat2 - lat1
    a = math.sin(dlat/2)**2 + math.cos(lat1) * math.cos(lat2) * math.sin(dlon/2)**2
    c = 2 * math.asin(math.sqrt(a))

    # Earth's radius in kilometers
    r = 6371
    return c * r

# Euclidean distance function
def euclidean(lat1, lon1, lat2, lon2):

    return math.sqrt( (lat1 -lat2)*(lat1 -lat2) + (lon1 - lon2)*(lon1 - lon2))

# Function for checking if route suits time windows
def checkTimeWindow(left, right, current_tour, earliest, latest, dis, service_time):

    tour = copy.copy(current_tour)
    if right == tour[1]:
        tour.insert(1, left)
    elif left == tour[-2]:
        tour.insert(-1, right)

    current_time = 0

    for i in range(1 , len(tour)):

        time = (dis[tour[i-1]][tour[i]] / SPEED) + service_time[tour[i-1]] + current_time
        if earliest[tour[i]] <= time <= latest[tour[i]]:
```

```
            current_time += (dis[tour[i-1]][tour[i]] / SPEED) + service_time[tour[i-1]]
        else:
            return False

    return True

# Function for printing time windows of route
def printTimeWindow(tour, dis, service_time, earliest, latest):
    current_time = 0

    for i in range(1 , len(tour)):
        current_time += (dis[tour[i-1]][tour[i]] / SPEED) + service_time[tour[i-1]]
        print(f"Node: {tour[i]} , Current Time: {current_time}, Earliest: {earliest[tour[i]]}, Latest: {latest[tour[i]]}")


# Another Function for checking if route suits time windows
def checkTimeWindow2(tour, dis, service_time, earliest, latest):

    current_time = 0
    for i in range(1 , len(tour)):

        time = current_time + service_time[ tour[i-1] ] + (dis[tour[i-1]][tour[i]] / SPEED)

        if earliest[tour[i]] <= time <= latest[tour[i]]:
            current_time += (dis[tour[i-1]][tour[i]] / SPEED) + service_time[tour[i-1]]
        else:
            return False

    return True

#Main function of Savings Algorithm
def savings(nodes, origin, dist, cap, earliest, latest, service_time):


    file = open("output.txt", "w")


    # Customer nodes except origin
    customers = {i for i in nodes if i != origin}

    # Savings computation
    savings = {(i, j): round(dist[i][origin] + dist[origin][j] - dist[i][j], 3)
               for i in customers for j in customers if j != i}

    # Nodes (i,j) savings decreasing order
    savings = dict(sorted(savings.items(), key=lambda item: item[1], reverse = True))
    visited_nodes = set()
    eliminated_nodes = set()
    tours = list()
    counter = 0

    #Check if nodes fits to capacity
    for customer in customers:
        if dist[origin][customer]*2*RATE > cap:
            visited_nodes.add(customer)
            eliminated_nodes.add(customer)
```

```
    # MODEL
    while len(visited_nodes) < len(customers) and counter < len(customers)*10:
        counter += 1
        current_tour: list = []
        filled_cap = 0

        if current_tour != []:
            print(current_tour)

        for (left, right) in savings:

            if current_tour != []:
                file.write(str(current_tour) + "\n")

            #If both left and right nodes are visited
            if left in visited_nodes and right in visited_nodes:
                pass

            #If right node is visited but left node is not visited
            elif right in visited_nodes and (left not in current_tour) and current_tour != []:

                indx = 0
                checking = True
                insertList = [1 ,-1]
                while indx < len(insertList) and checking:

                    tempTour = copy.copy(current_tour)
                    tempTour.insert(insertList[indx], left)

                    if checkTimeWindow2(tempTour,dist,service_time,earliest,latest) and insertList[indx] == 1:
                        if (dist[origin][left] + dist[tempTour[-2]][origin] + dist[left][tempTour[2]])* RATE + filled_cap <=
cap:
                            current_tour = tempTour
                            checking = False
                            filled_cap += dist[left][tempTour[2]] * RATE

                    elif checkTimeWindow2(tempTour,dist,service_time,earliest,latest) and insertList[indx] == -1:
                        if (dist[origin][tempTour[1]] + dist[left][origin] + dist[left][tempTour[-3]]) * RATE + filled_cap <=
cap:
                            current_tour = tempTour
                            checking = False
                            filled_cap += dist[left][tempTour[-3]] * RATE

                    indx += 1

            elif left in visited_nodes and (right not in current_tour) and current_tour != []:

                indx = 0
                checking = True
                insertList = [1 ,-1]
                while indx < len(insertList) and checking:

                    tempTour = copy.copy(current_tour)
                    tempTour.insert(insertList[indx], right)
```

```
                if checkTimeWindow2(tempTour,dist,service_time,earliest,latest) and insertList[indx] == 1:
                    if (dist[origin][right] + dist[tempTour[-2]][origin] + dist[right][tempTour[2]])* RATE + filled_cap
<= cap:
                        current_tour = tempTour
                        checking = False
                        filled_cap += dist[right][tempTour[2]] * RATE

                elif checkTimeWindow2(tempTour,dist,service_time,earliest,latest) and insertList[indx] == -1:
                    if (dist[origin][tempTour[1]] + dist[right][origin] + dist[right][tempTour[-3]]) * RATE + filled_cap
<= cap:
                        current_tour = tempTour
                        checking = False
                        filled_cap += dist[right][tempTour[-3]] * RATE

                indx += 1

            #If both left and right nodes are not visited
            elif left not in visited_nodes and right not in visited_nodes:

                #Check if current tour is empty and the pair suits for battery capacity and time window
                if current_tour == [] and (dist[origin][left] + dist[left][right] + dist[right][origin])*RATE <= cap and
checkTimeWindow2([origin, left, right, origin], dist, service_time, earliest, latest):

                    current_tour = [origin, left, right, origin]
                    filled_cap = dist[left][right] * RATE

                #Check if current tour is empty and pair do not fit time window, check for single node to fit in time
window
                elif current_tour == [] and checkTimeWindow2([origin, right, origin], dist, service_time, earliest, latest):

                    current_tour = [origin, right, origin]

                elif current_tour == [] and checkTimeWindow2([origin, left, origin], dist, service_time, earliest, latest):

                    current_tour = [origin, left, origin]

                #If current tour is not empty
                elif current_tour != []:

                    if left in current_tour and right in current_tour:
                        pass

                    elif right == current_tour[1] and (dist[origin][left] + dist[current_tour[-2]][origin] + dist[left][right]) *
RATE + filled_cap <= cap and checkTimeWindow(left, right, current_tour, earliest, latest, dist, service_time):
                        current_tour.insert(1, left)
                        filled_cap += dist[left][right] * RATE

                    elif left == current_tour[-2] and  (dist[origin][current_tour[1]] + dist[left][right] + dist[right][origin]) *
RATE + filled_cap <= cap and checkTimeWindow(left, right, current_tour, earliest, latest, dist, service_time):
                        current_tour.insert(-1, right)
                        filled_cap += dist[left][right] * RATE
                    else:
                        pass

        if current_tour != []:
```

```
            for node in current_tour:
                visited_nodes.add(node)

            tours.append(current_tour)

    # Tour length computations
    total_length = 0
    tours_length = []
    tour_demands = []

    for tour in tours:

        if tour != []:
            demand_tour = tour_length = 0

            print(f"{tours.index(tour)+1}. Tour Visit Times")
            printTimeWindow(tour, dist, service_time, earliest, latest)

            for i in range(len(tour)-1):

                tour_length += dist[tour[i]][tour[i+1]]
                total_length += dist[tour[i]][tour[i+1]]

                if i != 0:

                    demand_tour += (dist[tour[i-1]][tour[i]] * RATE)

            tour_demands.append(demand_tour)
            tours_length.append(tour_length)
            print("-"*50)

    file.close()
    return tours, tours_length, total_length, tour_demands, eliminated_nodes, visited_nodes

# Read excel
df = pd.read_excel('Data/rc108_21.xlsx')

# List of nodes
nodes = list(range(1,df.shape[0]))
nodes_with_depot = [0] + nodes
latitude = np.array(list(df.iloc[0:,1]))/10         #list of latitudes
longitude = np.array(list(df.iloc[0:,2]))/10        #list of longitudes
earliest = np.array(list(df.iloc[0:,3]))/10      #list of earliest times for visit
latest = np.array(list(df.iloc[0:,4]))/10    #list of latest times for visit
service_time = np.array(list(df.iloc[0:,5]))/10    #list of service times for nodes

# Distance matrix
dist = np.array([[euclidean(latitude[i], longitude[i], latitude[j], longitude[j])
                  for j in nodes_with_depot]
                  for i in nodes_with_depot]
                 )


#Check which nodes should be visited first
customers = {i for i in nodes if i != ORIGIN}
```

```
print("----------Possible Beginning Nodes----------")
for customer in customers:
    current_time = 0
    tour = [ORIGIN, customer, ORIGIN]

    if checkTimeWindow2(tour, dist, service_time, earliest, latest):

        print(f"Node: {tour[0]} , Current Time: {current_time}")
        for i in range(1 , len(tour)):
            current_time += (dist[tour[i-1]][tour[i]] / SPEED) + service_time[tour[i-1]]
            print(f"Node: {tour[i]} , Current Time: {current_time}")

        print("-"*50)



# Main Call
tours, tours_length, total_length, tour_demands, eliminated_nodes, visited_nodes = savings(nodes, ORIGIN, dist,
CAP, earliest, latest, service_time)

#Printing each tour length
for i in range(len(tours)):
    print(f"{i+1}. Tour is: {tours[i]}.\nTour Length: {tours_length[i]}.\nTour consumption:
{tours_length[i]*RATE}\n")

#Printing total tour lenght of all tours
print("Total Length:", total_length)

timely_visitedNodes = []
for tour in tours:
    timely_visitedNodes += tour
timely_visitedNodes = set(timely_visitedNodes)

for i in nodes:
    if i in eliminated_nodes:
        plt.scatter(latitude[i], longitude[i], c='k')
    elif (i in visited_nodes) and (i not in eliminated_nodes):
        plt.scatter(latitude[i], longitude[i], c='b')
    elif i not in timely_visitedNodes:
        plt.scatter(latitude[i], longitude[i], c='brown')

for indx, tour in enumerate(tours):
    for i in range(0, len(tour)-1):

        plt.plot([latitude[tour[i]], latitude[tour[i+1]]], [longitude[tour[i]], longitude[tour[i+1]]], c=COLORS2[ indx %
len(COLORS2) ], zorder=0)

plt.show()
```

Appendix C:

```
if twoOpt_calculate:
    def two_opt(tour, tour_length, d):
        current_tour, current_tour_length = tour, tour_length
        best_tour, best_tour_length = current_tour, current_tour_length
        solution_improved = True

        while solution_improved:

            solution_improved = False
            for i in range(1, len(current_tour)-2):
                for j in range(i+1, len(current_tour)-1):
                    difference = round((d[current_tour[i-1]][current_tour[j]]
                                + d[current_tour[i]][current_tour[j+1]]
                                - d[current_tour[i-1]][current_tour[i]]
                                - d[current_tour[j]][current_tour[j+1]]), 2)



                    Before_Reverse_length = 0
                    tourBefpreReverse=list(current_tour[i:j+1])
                    for q in range(0,len(tourBefpreReverse)-1):
                        Before_Reverse_length=Before_Reverse_length+d[tourBefpreReverse[q]][tourBefpreReverse[q+1]]

                    After_Reverse_length = 0
                    tourAfterReverse=list(reversed(current_tour[i:j+1]))
                    for q in range(0,len(tourAfterReverse)-1):
                        After_Reverse_length=After_Reverse_length+d[tourAfterReverse[q]][tourAfterReverse[q+1]]


                    while tourBefpreReverse:
                        tourBefpreReverse.pop()

                    while tourAfterReverse:
                        tourAfterReverse.pop()


                    difference=difference - Before_Reverse_length+ After_Reverse_length

                    if current_tour_length + difference < best_tour_length: #and checkTimeWindow2(current_tour[:i] +
list(reversed(current_tour[i:j+1])) + current_tour[j+1:], d, service_time, earliest, latest):
                        best_tour = current_tour[:i] + list(reversed(current_tour[i:j+1])) + current_tour[j+1:]
                        best_tour_length = round(current_tour_length + difference, 2)
                        solution_improved = True

            current_tour, current_tour_length = best_tour, best_tour_length

        # Return the resulting tour and its length as a tuple
        return best_tour, best_tour_length


    for i in range(0,len(nodes_with_depot)):
        for j in range(i+1, len(nodes)):
            dist[i][j] *= -1
```

```
    new_tours = list()
    new_total_tour_length = 0

    for tour, tour_length in zip(tours, tours_length):

        # Pass the elements to the `two_opt` function
        new_tour, new_tour_length = two_opt(tour, tour_length, dist)
        new_total_tour_length += new_tour_length

        new_tours.append(new_tour)

        # Print the best solution found as a result of the improvement phase
        print('Improved VRP tour is', new_tour, 'with total length', new_tour_length)

    #Printing total tour lenght after 2opts
    print("New Total Length:", new_total_tour_length)

    #Plotting tour routes
    plt.plot(latitude[0], longitude[0], c='r', marker='s')

    timely_visitedNodes = []
    for tour in tours:
        timely_visitedNodes += tour
    timely_visitedNodes = set(timely_visitedNodes)

    for i in nodes:
        if i in eliminated_nodes:
            plt.scatter(latitude[i], longitude[i], c='k')
        elif (i in visited_nodes) and (i not in eliminated_nodes):
            plt.scatter(latitude[i], longitude[i], c='b')
        elif i not in timely_visitedNodes:
            plt.scatter(latitude[i], longitude[i], c='brown')

    for indx, tour in enumerate(new_tours):
        for i in range(0, len(tour)-1):

            plt.plot([latitude[tour[i]], latitude[tour[i+1]]], [longitude[tour[i]], longitude[tour[i+1]]], c=COLORS2[ indx
% len(COLORS2) ], zorder=0)

    plt.show()
```

10. REFERENCES

Bosch Global. (2022, December 29). Early forest fire detection using sensors. Bosch Global. Retrieved December 29, 2022, from https://www.bosch.com/stories/early-forest- fire-detection-sensors/

Cerrone, C., Cerulli, R., & Sciomachen, A. (2021, June 7). Grocery distribution plans in urban networks with street crossing penalties. Networks, 78(3), 248–263. https://doi.org/10.1002/net.22061

Deltaquad Operation Manual. (n.d.). Retrieved December 26, 2022, from https://docs.deltaquad.com/deltaquad-operation-manual/

Facts+statistics: Wildfires. III. (n.d.). Retrieved December 7, 2022, from https://www.iii.org/fact-statistic/facts-statistics-wildfires

Haversine formula in Python (bearing and distance between two GPS points). (2011, February 6). Stack Overflow. Retrieved April 25, 2023, from https://stackoverflow.com/questions/4913349/haversine-formula-in-python-bearing-and-distance-between-two-gps-points

Kyuchukova, Diyana & Hristov, G.V. & Raychev, Jordan & Захариев, Пламен. (2019). EARLY FOREST FIRE DETECTION USING DRONES and ARTIFICIAL INTELLIGENCE. 1060-1065. 10.23919/MIPRO.2019.8756696

Matrice 600 PRO. (n.d.). Retrieved December 5, 2022, from https://www.dji.com/matrice600-pro?site=brandsite&from=landing_page

Ritchie, H., Roser, M., & Rosado, P. (2020, May 11). CO2 emissions. Our World in Data. Retrieved December 7, 2022, from https://ourworldindata.org/co2-emissions

Technical Datasheet. (n.d.). Retrieved December 12, 2022, from https://www.ssass.co.za/_files/ugd/f97229_98b4d7085616412da1ff395b592e484d.pdf

Technologies, V. (n.d.). Deltaquad Pro #VIEW VTOL surveillance UAV. DeltaQuad VTOL UAV. Retrieved April 16, 2023, from https://www.deltaquad.com/vtol- drones/view/