IIT KHARAGPUR

AIFA Assignment: -1

DELIVERY SYSTEM OPTIMISATION

INDEX

Abstract

Ground vehicles such as trucks are typically used to deliver goods across the logistic networks but drones are gaining more attraction for delivery purposes in recent years. More recently, large companies such as Amazon, Flipkart have plans to utilize drones for delivery purposes. New-Delhi even allowed drone deliveries for more than 20 companies during the covid pandemic. The general idea is to deliver packages from a base location to pre-selected destinations by a given fleet of vehicles. UAVs, also known as drones, can deliver packages, like medicines, masks, home supplies, etc.

The scheduling model for the drone delivery problem is similar to the traveling salesman problem, but here, there are more than one vehicle and each group of destination locations is served by only one vehicle, and vehicles start and finish their path at the same location.

Problem Description

The delivery system optimization addresses the problem of delivering all the items to 'n' destinations via 'm' drones through the shortest path possible. It is one of the most realistic challenges in AI as it can be used for delivering almost anything ranging from home supplies to medicinal support or mails. Pathfinding strategies are usually employed in any AI movement system, but here we are trying to create a delivery network system that uses distance as a constraint rather than time.

Everyone is aware of the famous Djkstra's Algorithm, which finds the shortest path from the start node to all nodes. But in Djkstra's Algorithm has only one delivery agent. Our problem has many delivery agents and can be considered an advanced version of Djkstra. Out algorithm provides an optimal path for the delivery agents (UAV's). (Optimal path does not necessarily mean the shortest path, i.e., the combined distance of all the drones is minimised rather than the distance travelled by one drone)

Formal Problem Statement

We have 'm' number of drones that are scheduled to deliver items or services (be it anything which is deliverable via a drone) at 'n' places. We have to design an algorithm that provides the shortest path for each drone, such that all the products are delivered through the shortest possible path and a minimum number of available drones are used. Also the drones need to return to the starting point.

INPUT: - *The destinations are provided through the points of a Cartesian plane coordinate system and the origin is fixed at the starting place of all drones (which is the same and represented by 'S').*

OUTPUT: - *The output is given via an array of destinations (like [P_i, P_j, P_k]), which indicates that a drone travels from S to P_i to P_j to P_k to S. Also, it is assumed that the drones have a sufficient amount of battery to reach any 3 destinations.*

Quantity Optimised: - Distance is the quantity which is to be optimised.

AI Modelling

Our problem does not come under a specific category. However, this problem can be classified as a combination of SAT model ,CSP model and a planning model.

Our problem contains many possible paths for every drone. So the search space is vast, and standard search methods cannot be used to find the solution as the solution time and search space will be huge.

In order to solve this problem, we have used an SAT constraint to create a constraint table, based on which we reduce our search space of the possible paths of drones.

We then use the standard search method(BFS preferred), based on some constraints to traverse through the reduced search space and find an optimal path.

Here, we have preferred BFS over DFS as it is an optimization problem. Also, the level is not deep, so the complexity of BFS will not be an issue(complexity explained in detail on page 10). Also, we have not used A* or IDA* as the cost to travel to the next state does not necessarily decrease, and the heuristics are unclear.

Solution Approach

Imagine all the destinations are scattered through the cartesian plane about the origin, now the optimal path for a drone will occur if: -

i. The drone does not change much direction during its travel (like it should not try to reach a point whose direction is very much different than the path it is currently travelling).
ii. After satisfying the first condition, the drone should try to reach the longest path in the shortest way possible in that direction.

Below is the set of all variables used for this algorithm.

Drones (A): - $\{a_1, a_2,\ldots..,a_m\}$
Destinations (P): - $\{P_1, P_2,\ldots\ldots P_n\}$
"d_{PiPj}": -Distance between P_i and P_j
"S": - Starting position (origin)
P_i -> P_j: - drone travelling from P_i to P_j
$P_1 = \{x_1,y_1\}$

$P_2 = \{x_2,y_2\}$
:
:
:
$P_n = \{x_n,y_n\}$

Sorted in increasing order of distance from 'S'

path = $[path_1, path_2,path_3,\ldots\ldots,path_m]$
$path_z$:- refers to the path taken by the drone a_z, and is of format $[[P1,P2,P3],[d_{P3S}+d_{P3P2}+d_{P2P1}]]$
Initially , all path is set to empty. and each path is an array of points.
Destinations are also termed as nodes.

Constraint used:-

'd_{SPj}-d_{PiPj}':- This expression calculates the difference of distance of a point P_j from S and P_j from P_i. This is the constraint which tells us whether it is better to go to a node P_j from P_i or from S.

- If **d_{SPj}-d_{PiPj} > 0:-** It is profitable to go to node P_j from P_i, as drone will travel a shorter distance in that course.
- If **d_{SPj}-d_{PiPj} < 0:-** It is profitable to go to node P_j from S, as a drone will travel a shorter distance in that course.

Now, based on this constraint we create a constraint table(T) $\forall$ i, j $\in$ P

Constraint table:-

We will calculate 'd_{SPi}-d_{PiPj}' for all the combinations in the above table.This table is a 2D array represented by 'T'.

P_i/ P_j	P_1	P_2	P_3	P_4	P_5	.....	P_n
P_1	d_{sp1}	d_{Sp1}-d_{p1p2}	d_{sp1}-d_{p1p3}	.	.	. . . . .	d_{Spn}-d_{p1pn}
P_2	d_{Sp2}-d_{p2p1}	d_{Sp2}	.	.	.	. . . . .	.
P_3	d_{sp3}-d_{p3p1}	.	d_{Sp3}	.	.	. . . . .	.
P_4	d_{sp4}-d_{p4p1}	.	.	d_{Sp4}	.	. . . . .	.
P_5	.	.	.		d_{Sp5}	. . . . .	.
. . .	. . .	. . .	. . .	. . .	. . . .	. . .	. . . .
P_n	.	.	.	.	.	. . . . .	d_{Spn}

Explantion of Algorithm

Search Method:-

After creating the search table, we now try to reach the farthest point(last point of destinations array) through the shortest way. Since, all the points in P are sorted in the increasing order of their distance from origin, therefore the rightmost column of the table represents the constraint relation of the farthest point with all other points.
Now we follow some general steps to create a search tree: -

1. For a drone $a_z \in A$, $z \in [1,m]$
2. We choose the rightmost column of the table [lets say 'u'], we check the value $T[t][u]\ \forall\ t \in P-\{u\}$.
3. If $T[t][u] > 0$:- Then we add P_t as a child node of P_u. We then create all the child nodes of P_t. and substitute 't = u' and 'u = r' (for some $r \in P-\{u\}$), then repeat step 2 for all values of 'u'. A search tree is created in this manner.
4. If no value of 'u' exists, such that $T[t][u] > 0$, then we travel to P_t from 'S', so we add 'S' as a child of node P_u. i.e., we decide to reach P_u from 'S' rather than 'P_t', since it is profitable.
5. If length of possible path = 3 :- Then we stop the process of further adding children nodes and proceed to find the best path among the current search tree. Then we find the shortest distance travelled by drone among all the possible combinations.
6. Now, since a drone can travel three destinations at max, many points will exist for each 't' and each 'u' and in the worst case scenario the total number of

such combinations of ‘t’ and ‘u’ will be possible and less than ‘l^2’, where ‘l’ = length of P.

7. In all such possible combinations, we will choose the combination in which the drone travels the least distance and append that combination to $path_z$.
8. Thus, the final path travelled by drone a_z will be $path_z = [P_s, P_u, P_t]$.
9. Now, we will remove points P_s, P_u, P_t from P, according to which a new constraint table will be formed according to the new destination set P.
10. Then the same process will be repeated for the new ‘P’. This process will be repeated until P becomes empty.

(Code will be updated later)

An example of the above-mentioned process is shown below.

- We choose 6 random points:- [(26.7,56.9),(12.7,19.4),(47.45,-15.5),(-27,3),(-2,-60),(15,-15)].

```
Enter the number of destinations: 6
Enter the x co-ordinate of point 0 respectively: 26.7
Enter the y co-ordinate of point 0 respectively: 56.9
Enter the x co-ordinate of point 1 respectively: 12.7
Enter the y co-ordinate of point 1 respectively: 19.4
Enter the x co-ordinate of point 2 respectively: 47.45
Enter the y co-ordinate of point 2 respectively: -15.5
Enter the x co-ordinate of point 3 respectively: -27
Enter the y co-ordinate of point 3 respectively: 3
Enter the x co-ordinate of point 4 respectively: -2
Enter the y co-ordinate of point 4 respectively: -60
Enter the x co-ordinate of point 5 respectively: 15
Enter the y co-ordinate of point 5 respectively: -15
Destinations are: [[15.0, -15.0], [12.7, 19.4], [-27.0, 3.0], [47.45, -15.5], [-2.0, -60.0], [26.7, 56.9]]
```

- After giving the input in our program (will be updated later), the program sorts the point in increasing order of their distance and also returns the constraint table as follows: -

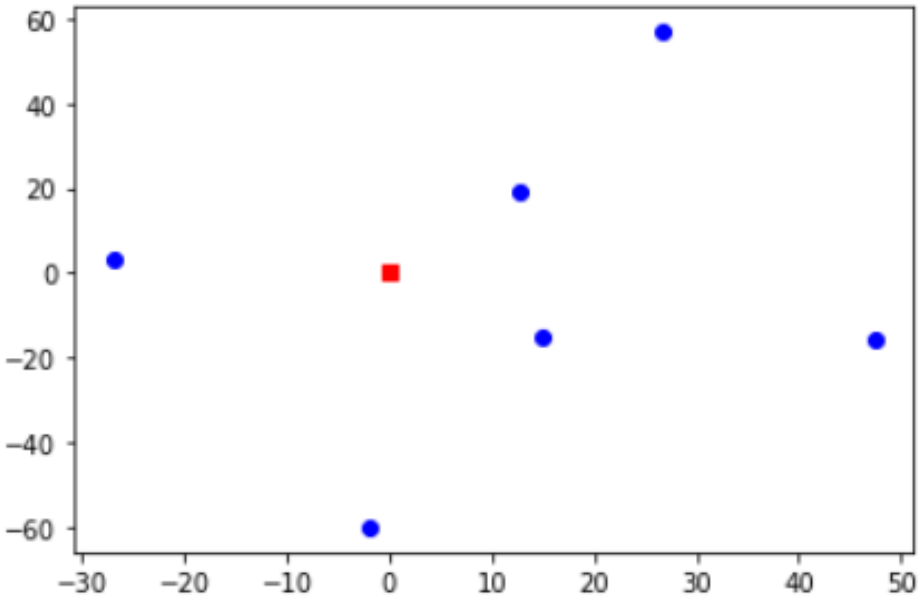
i) P_1 = (15.0, -15.0)
ii) P_2 = (12.7, 19.4)
iii) P_3 = (-27, 3)
iv) P_4 = (47.45, -15.5)
v) P_5 = (-2.0, -60)
vi) P_6 = (26.7, 56.9)

- Constraint Table: -

21.213	**-11.289**	**-18.528**	**17.463**	**11.929**	**-9.992**
-13.263	23.187	-15.787	0.667	-20.715	22.824
-24.481	-19.766	27.166	-26.796	-7.745	-13.231
-11.240	-26.062	-49.547	49.917	-6.491	-12.461
-26.890	-57.562	-40.612	-16.607	60.033	-57.518
-51.632	-16.840	-48.918	-25.397	-60.338	62.853

```
Constraint table:
[21.213203435596427, -11.28952279990542, -18.528483220771204, 17.46360501479709, 11.929270195991762, -9.992724286581293]
[-13.263600360579339, 23.187280996270346, -15.787889796720084, 0.667355343572801, -20.715979323353594, 22.824887991181328]
[-24.481435199587025, -19.766764214861986, 27.16615541441225, -26.7966389701129, -7.7457287321076365, -13.231817977614014]
[-11.240648415917292, -26.06282052646766, -49.547940422011465, 49.91745686631081, -6.49150211371888, -12.461819467862455]
[-26.89085044762635, -57.562022406297785, -40.61289739690993, -16.60736932662261, 60.03332407921454, -57.51850514629773]
[-51.632523961417064, -16.840834122980525, -48.91866567363397, -25.397365711983845, -60.33818417751539, 62.8530031104322]
```

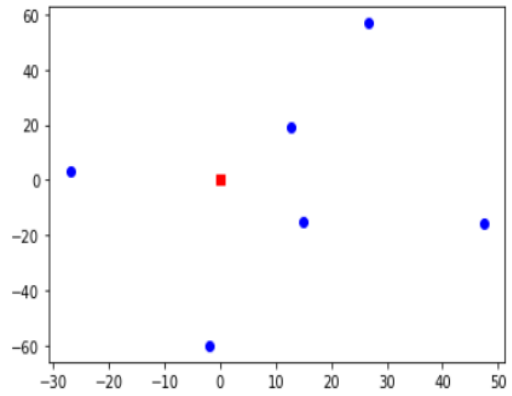
- Graph Plotted (red square represents origin, blue circle represent destination points): -



The whole program output screen looks something like:

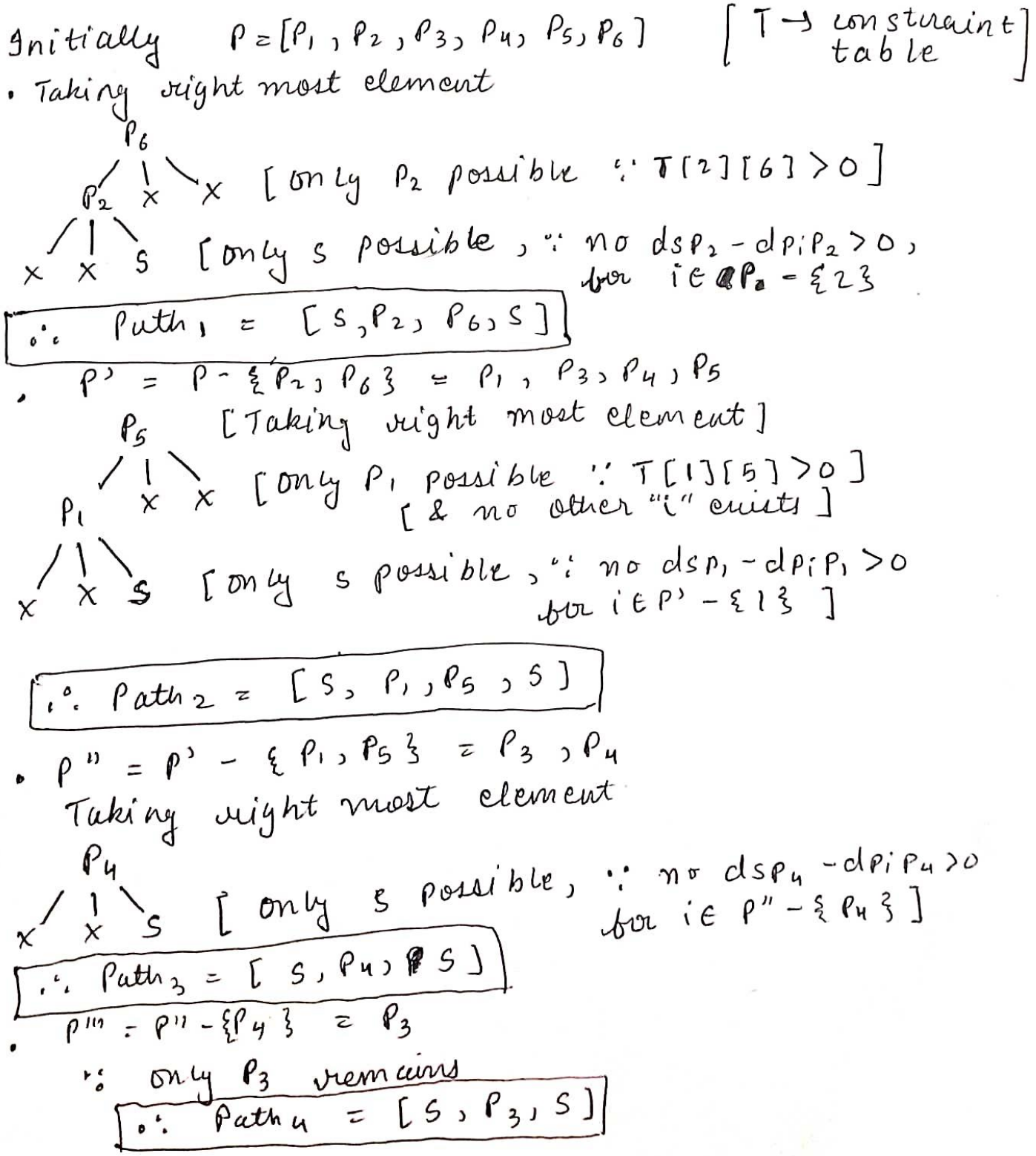
```
Enter the number of destinations: 6
Enter the x co-ordinate of point 0 respectively: 26.7
Enter the y co-ordinate of point 0 respectively: 56.9
Enter the x co-ordinate of point 1 respectively: 12.7
Enter the y co-ordinate of point 1 respectively: 19.4
Enter the x co-ordinate of point 2 respectively: 47.45
Enter the y co-ordinate of point 2 respectively: -15.5
Enter the x co-ordinate of point 3 respectively: -27
Enter the y co-ordinate of point 3 respectively: 3
Enter the x co-ordinate of point 4 respectively: -2
Enter the y co-ordinate of point 4 respectively: -60
Enter the x co-ordinate of point 5 respectively: 15
Enter the y co-ordinate of point 5 respectively: -15
Destinations are: [[15.0, -15.0], [12.7, 19.4], [-27.0, 3.0], [47.45, -15.5], [-2.0, -60.0], [26.7, 56.9]]

Constraint table:
[21.213203435596427, -11.28952279990542, -18.528483220771204, 17.46360501479709, 11.929270195991762, -9.992724286581293]
[-13.263600360579339, 23.187280996270346, -15.787889796720084, 0.667355343572801, -20.715979323353594, 22.824887991181328]
[-24.481435199587025, -19.766764214861986, 27.16615541441225, -26.7966389701129, -7.7457287321076365, -13.231817977614014]
[-11.240648415917292, -26.06282052646766, -49.547940422011465, 49.91745686631081, -6.49150211371888, -12.461819467862455]
[-26.89085044762635, -57.562022406297785, -40.61289739690993, -16.60736932662261, 60.03332407921454, -57.51850514629773]
[-51.632523961417064, -16.840834122980525, -48.91866567363397, -25.397365711983845, -60.33818417751539, 62.8530031104322]
```



- Points to note: - Since only 6 points are considered in this example, our algorithm finds an optimal path in the first round of every search.
- However, in real life, where there are hundreds and even thousands of points, the algorithm will have to search the best possible path by also comparing the total distance travelled.
- Also, it is evident that, before the constraint is used, there were many paths for a drone which were possible but not at all optimal, after using the constraint, our search space got reduced significantly (from ${}^{6}C_{3} * {}^{3}C_{2} = 60$ to just 4 possible ways) and our search method became efficient.
- Therefore, this method is also efficient for large numbers.

A visual representation of working of our algorithm for the above example is presented below: -



Code explanation

The code for the above-mentioned algorithm is attached below (Ctrl+Click to open). You can run or download the file from the drive.

AIFA-Group_Assignment_1

The basic functions of the code is explained in the following few steps: -

- dist_btw_two_points():- Takes two points as input and returns the distance between them.

```
import matplotlib.pyplot as plt       #importing tool for plotting

def dist_btw_two_points(p1,p2):       #This function takes two points as input and returns the distance between them rounded
    distance = (( ((p2[0]-p1[0])**2) + ((p2[1]-p1[1])**2) )**0.5)
    distance = round(distance*10000)
    distance /= 10000
    return distance
```

- point_array()- This function takes the input destinations from the user and returns a sorted array of destinations, sorted in the order of increasing distance from origin.

```
def sort(point):                      #The purpose of this function is to sort the points obtained from user i increasing order
    for i in range(1,len(point)):
        if dist_btw_two_points(point[i-1],[0,0]) > dist_btw_two_points(point[i],[0,0]):
            temp = point[i-1]
            point[i-1] = point[i]
            point[i] = temp
            sort(point)

def point_array():                    #The purpose of this function is to take input of points from the user and return a sorte
    n = int(input('Enter the number of destinations: '))
    destination_point = []
    sorted_point = []
    for i in range(n):
        p = []
        x = float(input(f"Enter the x co-ordinate of point {i+1} respectively: "))
        p.append(x)
        y = float(input(f"Enter the y co-ordinate of point {i+1} respectively: "))
        p.append(y)
        destination_point.append(p)
    sort(destination_point)
    return destination_point
```

- constraint_table():- This function takes a copy of the sorted array(“dt_cpy”) returned by the function point_array and creates the constraint table mentioned in our algorithm.

```
def constraint_table(point):          #This function takes input the sorted destinations and ret
    table = []
    for i in range(len(point)):
        rows = []
        for j in range(len(point)):
            dist  = dist_btw_two_points(point[j],[0,0])-dist_btw_two_points(point[j],point[i])
            rows.append(dist)
        table.append(rows)
    return table
```

- possible_paths_of_a_point():- It is one of the most important functions in our code. It takes the copied version of destinations(“dt_cpy”), and a temporary array “temp_array” as input, and based on the constraint table, searches all the possible profitable paths for the last point in “dt_cpy” (“dt_cpy” is sorted already) according to our algorithm, and stores it in temp array.

```
def possible_paths_of_a_point(dt_cpy,length,temp_array): #This function creates the search tree for the point which is farthe
    r = 3                                                #i.e. It creates the search tree of all possible paths along with to
    table = constraint_table(dt_cpy)
    anv1 = True
    for j in range(length):
        if table[j][length-1]>=0 and j!=length-1:
            anv1 = False
            anv2 = True
            for k in range(length):
                if table[k][j]>=0 and k!=length-1 and k!=j:
                    anv2 = False
                    if anv1 == False and anv2 == False:
                        temp_array.append([[dt_cpy[k],dt_cpy[j],dt_cpy[length-1]],[dist_btw_two_points(dt_cpy[j],dt_cpy[lengt
                        r = 2
                if anv2 == True and anv1 == False:
                    temp_array.append([[dt_cpy[j],dt_cpy[length-1]],[dist_btw_two_points(dt_cpy[j],dt_cpy[length-1])+dist_btw
                    r = 1
        if anv1 == True:
            temp_array.append([[dt_cpy[length-1]],[dist_btw_two_points(dt_cpy[length-1],[0,0])]])
            r = 0
    #for k in range(len(temp_array)):
        #print(temp_array[k])
    return r
```

- deciding_path():- This is the brain of our function. It basically calls all the function into one. First it takes the input from the user, then it calls the constraint table function to create the constraint table. Then it calls the function “possible_paths_of_a_point” in an orderly fashion and filters out the path for the last point of “dt_cpy” by chosing the path which has travelled the least distance. Then it appends the filtered path to the array named “path” and removes the points of the filtered path from “dt_cpy”. Then “possible_paths_of_a_point” is called again for the new “dt_cpy”. This process continues until “dt_cpy ” becomes empty.

```
path = []
def deciding_path():     #This is the main brain of the program. It calls the poss
    global s             #Then it appends that specific trajectory to the array na
    destinations = point_array()
    dt_cpy = destinations.copy()
    print(f"Destinations are: {destinations}")
    print("")
    print("Constraint table: ")
    table = constraint_table(dt_cpy)
    for i in range(len(dt_cpy)):
        print(table[i])
    while True:
        table = constraint_table(dt_cpy)
        if len(dt_cpy) == 0:
            break
        if len(dt_cpy) == 1:
            path.append([[dt_cpy[0]],[dist_btw_two_points(dt_cpy[0],[0,0])]])
            dt_cpy.remove(dt_cpy[0])
        if len(dt_cpy)!=0:
            temp_array = []
            if possible_paths_of_a_point(dt_cpy,len(dt_cpy),temp_array) == 2:
                d = float('inf')
                for t in range(len(temp_array)):
                    if temp_array[t][1][0] < d and len(temp_array[t][0]) == 3:
                        d = temp_array[t][1][0]
                        s = t
                path.append(temp_array[s])
                dt_cpy.remove(temp_array[s][0][0])
                dt_cpy.remove(temp_array[s][0][1])
                dt_cpy.remove(temp_array[s][0][2])
            if possible_paths_of_a_point(dt_cpy,len(dt_cpy),temp_array) == 1:
                d = float('inf')
                for t in range(len(temp_array)):
                    if temp_array[t][1][0] < d and len(temp_array[t][0]) == 2:
                        d = temp_array[t][1][0]
                        s = t
                path.append(temp_array[s])
                dt_cpy.remove(temp_array[s][0][0])
                dt_cpy.remove(temp_array[s][0][1])
            if possible_paths_of_a_point(dt_cpy,len(dt_cpy),temp_array) == 0:
                d = float('inf')
                for t in range(len(temp_array)):
                    if temp_array[t][1][0] < d and len(temp_array[t][0]) == 1:
                        d = temp_array[t][1][0]
                        s = t
                path.append(temp_array[s])
                dt_cpy.remove(temp_array[s][0][0])
    return destinations
```

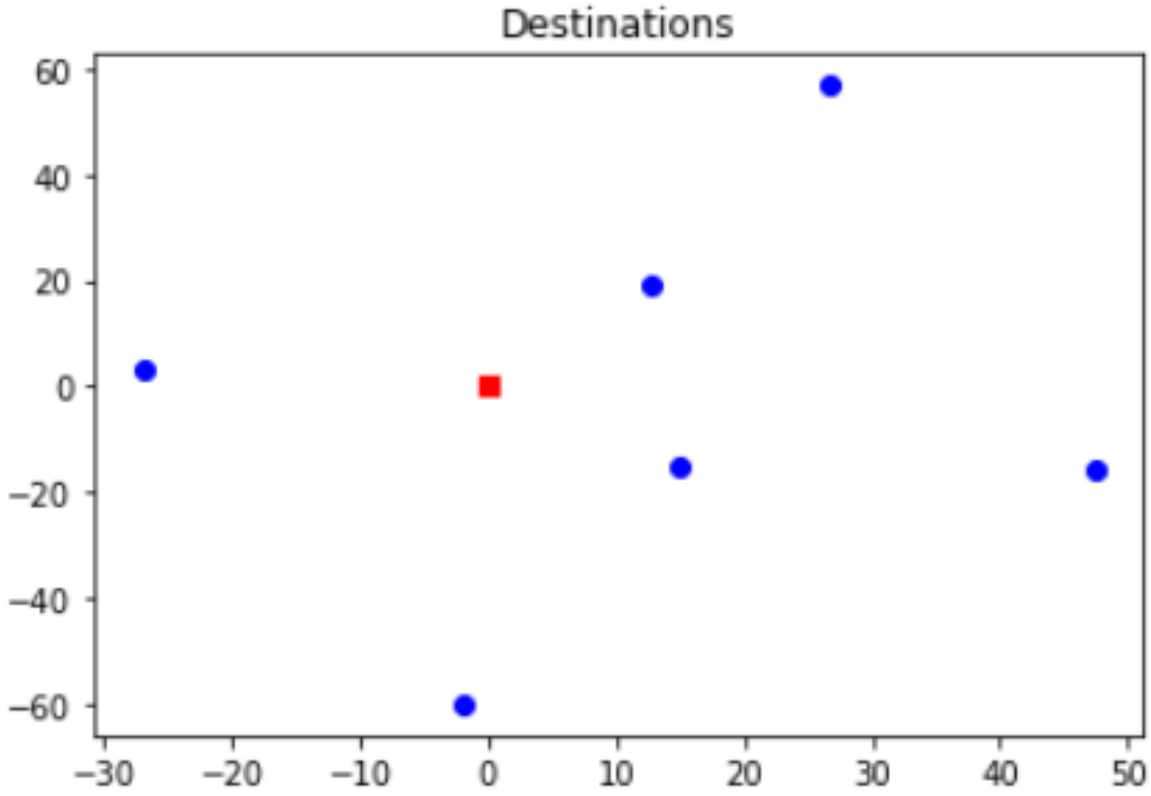
- Thus all the paths are appended to the “path array”.
- We the print the path.
- plot_point():- This function takes destinations as input and plots all the points, including the origin, on a 2D plane.
- plot_line():- This function takes “path” array as input and plots the path for the drones.
- If we consider the examples used above on page 11.
    - P_1 = (15.0, -15.0)
    - P_2 = (12.7, 19.4)
    - P_3 = (-27, 3)
    - P_4 = (47.45, -15.5)
    - P_5 = (-2.0, -60)
    - P_6 = (26.7, 56.9)

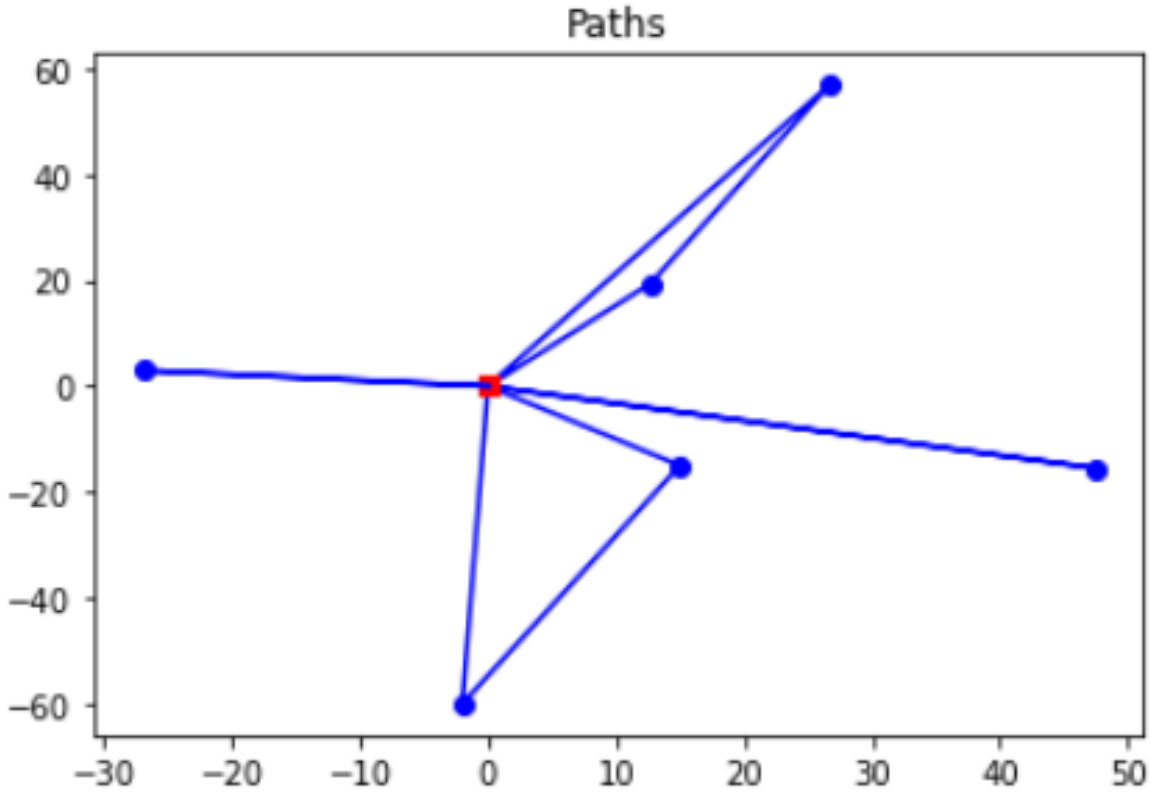
Output received is as follows: -

```
Enter the number of destinations: 6
Enter the x co-ordinate of point 1 respectively: 15
Enter the y co-ordinate of point 1 respectively: -15
Enter the x co-ordinate of point 2 respectively: 12.7
Enter the y co-ordinate of point 2 respectively: 19.4
Enter the x co-ordinate of point 3 respectively: -27
Enter the y co-ordinate of point 3 respectively: 3
Enter the x co-ordinate of point 4 respectively: 47.45
Enter the y co-ordinate of point 4 respectively: -15.5
Enter the x co-ordinate of point 5 respectively: -2
Enter the y co-ordinate of point 5 respectively: -60
Enter the x co-ordinate of point 6 respectively: 26.7
Enter the y co-ordinate of point 6 respectively: 56.9
Destinations are: [[15.0, -15.0], [12.7, 19.4], [-27.0, 3.0], [47.45, -15.5], [-2.0, -60.0], [26.7, 56.9]]


Constraint table:
[21.2132, -11.289499999999997, -18.5284, 17.4636, 11.929199999999994, -9.992699999999992]
[-13.263599999999997, 23.1873, -15.7878, 0.6673999999999936, -20.716000000000008, 22.8249]
[-24.4814, -19.7667, 27.1662, -26.796600000000005, -7.745800000000003, -13.2318]
[-11.240699999999997, -26.062800000000003, -49.5479, 49.9175, -6.491500000000002, -12.461800000000004]
[-26.890900000000002, -57.562000000000005, -40.612899999999996, -16.607300000000002, 60.0333, -57.518499999999996]
[-51.63249999999999, -16.8408, -48.9186, -25.39730000000001, -60.3382, 62.853]

Paths are:
[[[12.7, 19.4], [26.7, 56.9]], [63.2154]]
[[[15.0, -15.0], [-2.0, -60.0]], [69.3173]]
[[[47.45, -15.5]], [49.9175]]
[[[-27.0, 3.0]], [27.1662]]
```



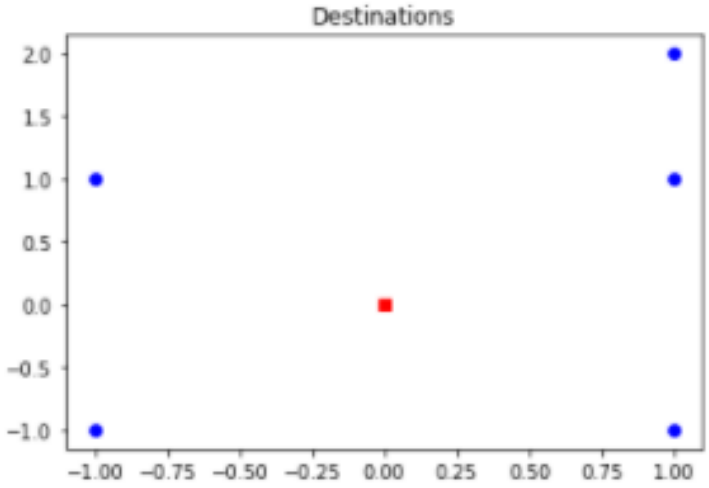


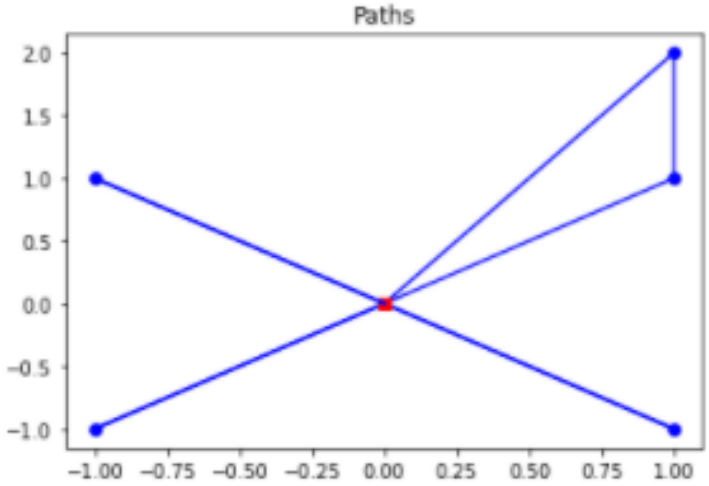
Some more examples of output are as follows:-

```
Enter the number of destinations: 5
Enter the x co-ordinate of point 1 respectively: 1
Enter the y co-ordinate of point 1 respectively: 1
Enter the x co-ordinate of point 2 respectively: 1
Enter the y co-ordinate of point 2 respectively: 2
Enter the x co-ordinate of point 3 respectively: -1
Enter the y co-ordinate of point 3 respectively: 1
Enter the x co-ordinate of point 4 respectively: 1
Enter the y co-ordinate of point 4 respectively: -1
Enter the x co-ordinate of point 5 respectively: -1
Enter the y co-ordinate of point 5 respectively: -1
Destinations are: [[1.0, 1.0], [-1.0, 1.0], [1.0, -1.0], [-1.0, -1.0], [1.0, 2.0]]

Constraint table:
[1.4142, -0.5858000000000001, -0.5858000000000001, -1.4142, 1.2361]
[-0.5858000000000001, 1.4142, -1.4142, -0.5858000000000001, 0.0]
[-0.5858000000000001, -1.4142, 1.4142, -0.5858000000000001, -0.7639]
[-1.4142, -0.5858000000000001, -0.5858000000000001, 1.4142, -1.3695]
[0.4141999999999999, -0.8219000000000001, -1.5858, -2.1914, 2.2361]

Paths are:
[[[1.0, 1.0], [1.0, 2.0]], [2.4142]]
[[[-1.0, -1.0]], [1.4142]]
[[[1.0, -1.0]], [1.4142]]
[[[-1.0, 1.0]], [1.4142]]
```

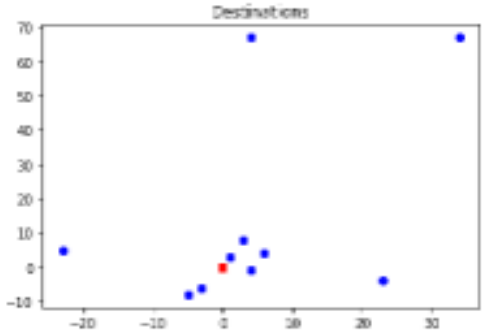


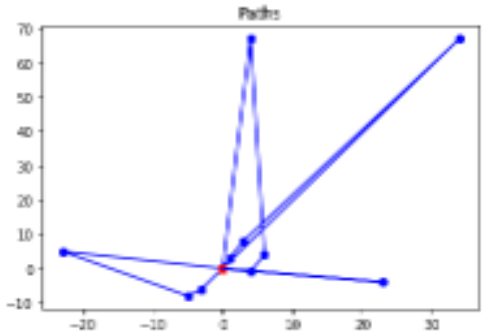


```
Enter the number of destinations: 10
Enter the x co-ordinate of point 1 respectively: -5
Enter the y co-ordinate of point 1 respectively: -8
Enter the x co-ordinate of point 2 respectively: -23
Enter the y co-ordinate of point 2 respectively: 5
Enter the x co-ordinate of point 3 respectively: 4
Enter the y co-ordinate of point 3 respectively: 67
Enter the x co-ordinate of point 4 respectively: 3
Enter the y co-ordinate of point 4 respectively: 8
Enter the x co-ordinate of point 5 respectively: 4
Enter the y co-ordinate of point 5 respectively: -1
Enter the x co-ordinate of point 6 respectively: 6
Enter the y co-ordinate of point 6 respectively: 4
Enter the x co-ordinate of point 7 respectively: 34
Enter the y co-ordinate of point 7 respectively: 67
Enter the x co-ordinate of point 8 respectively: 23
Enter the y co-ordinate of point 8 respectively: -4
Enter the x co-ordinate of point 9 respectively: -3
Enter the y co-ordinate of point 9 respectively: -6
Enter the x co-ordinate of point 10 respectively: 1
Enter the y co-ordinate of point 10 respectively: 3
Destinations are: [[1.0, 3.0], [4.0, -1.0], [-3.0, -6.0], [6.0, 4.0], [3.0, 8.0], [-5.0, -8.0], [23.0, -4.0], [-23.0, 5.0],
[4.0, 67.0], [34.0, 67.0]]

Constraint table:
[3.1623, -0.8769, -3.1407000000000007, 2.1121, 3.1588000000000003, -3.096, 0.2583999999999982, -0.5460000000000029, 3.04899
99999999924, 3.1263000000000005]
[-1.8377, 4.1231, -1.8941, 1.8258999999999999, -0.5114000000000001, -1.9678000000000004, 4.1098, -4.121400000000001, -0.880
7000000000045, 0.8896000000000032]
[-6.6866, -4.4792, 6.7082, -6.2425, -6.6875, 6.685599999999999, -2.7316000000000003, 0.7118000000000002, -6.21550000000000
6, -6.708100000000002]
[-1.9367, -1.2621000000000002, -6.7454, 7.2111, 3.5440000000000005, -6.844800000000001, 4.556899999999999, -5.48, 4.0875999
99999995, 6.191200000000009]
[-2.2229, -4.932300000000001, -8.5233, 2.2111, 8.544, -8.454500000000001, 0.021399999999999864, -2.635300000000001, 8.11079
9999999998, 8.484899999999996]
[-9.3677, -7.2787, 3.8798, -9.0677, -9.3445, 9.434, -4.939100000000003, 1.333599999999997, -8.418800000000005, -9.400800000
000004]
[-19.924500000000002, -15.112299999999998, -19.3686, -11.5772, -14.779799999999998, -18.850300000000004, 23.3452, -23.335,
-6.379000000000005, 3.2861000000000047]
[-20.920000000000003, -23.5355, -16.117199999999997, -21.8861, -17.6285, -12.769600000000002, -23.527, 23.5372, -0.50469999
99999997, -9.886799999999997]
[-60.908, -63.8769, -66.6266, -55.8286, -50.4645, -66.1041, -50.1531, -44.0868, 67.1193, 45.1332]
[-68.8446, -70.2005, -75.1331, -61.73089999999999, -58.10430000000001, -75.10000000000001, -48.5019, -60.6828, 37.119299999
999996, 75.1332]

Paths are:
[[[1.0, 3.0], [3.0, 8.0], [34.0, 67.0]], [75.1958]]
[[[4.0, -1.0], [6.0, 4.0], [4.0, 67.0]], [72.53999999999999]]
[[[-3.0, -6.0], [-5.0, -8.0], [-23.0, 5.0]], [31.7482]]
[[[23.0, -4.0]], [23.3452]]
```





We have tried this algorithm with many points and our algorithm gives accurate results in a short amount of time even for large number of points.

Practical Applications

Our algorithm also solves many real-life problems. Some of them are listed below: -

- To rescue people, stuck in their homes due to flood or landslide from rescue helicopters.
- To provide resources (food and supplies) to areas where it is not possible to reach manually.
- To provide us an optimised path between points where multiple delivery agents are considered.

Possible Improvements

i) Improvement in algorithm:

Even though we have tried our best to make our algorithm as general as possible, there is still some improvements which are possible: -

- Such as, we can modify our 2D co-ordinate system and make it a 3D co-ordinate system. The rest of the algorithm will remain the same as the constraint table and the search tree will not change.
- Here, we have assumed that our delivery agent travels to at max three points, and that there is only one supply station or depot(origin) in the entire system. A more general case would be that

there are multiple distribution centres and multiple destination along with multiple agents and 3D co-ordinates.

- In that case we will map or cluster different destinations to one supply centre based on the distance between the supply centre(depot) and the destination, the range of the drones based on the battery limit, recharging of battery in different depots.
- Then each cluster of depot and destinations can be solved individually for finding an optimised path.
- We can try to use heuristics for searching the optimal path in our algorithm after the creation of the constraint table (where we have preferentially used BFS).

ii)Improvements in code:

Many improvements in code are possible. As we are novices, our program works fine for the specified problem, but if we want to use it in real life problems, it has to be linked through all the tools which we use to gain the data, like satellites, recognition systems etc. Also, the overall structure of our code can be enhanced.

You-tube video link

https://youtu.be/6r8w1vXjy3A

Code link

AIFA-Group_Assignment_1

Challenges Faced

- Initially we were confused about the methods we will use to solve the problem, because there is no predefined method or algorithm specific to path finding problem in our case and each method which we came up with failed under certain circumstances.
- There were many ways to optimise the path among the points, and we had to decide the one which gave the best results.
- The creation of the part of the code which creates the search tree and stores it took immense hard work.
- This topic does not come under a specific category of AI, so we were doubtful about this topic.

Refferences

- https://www.youtube.com/watch?v=_uQrJ0TkZlc - For learning Python
- https://plaban.github.io/AIFA_21_Autumn.html - For general working and techniques of AI.
- https://data-flair.training/blogs/artificial-intelligence-project-ideas/ - To get some general ideas about AI, and to make sure that we create something innovative and not use something which is already developed.