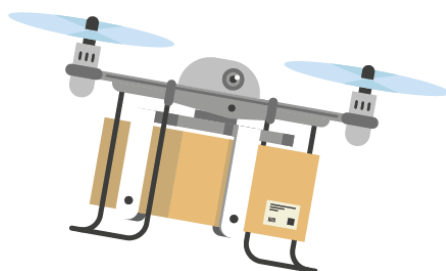




Al Imam Mohammad Ibn Saud Islamic University
College of Computer and Information Sciences
Computer Science Department

CS 361 – Artificial Intelligence
Fall 2021 - Semester Project
Developing Interactive AI Tutorials



Drone Delivery

Student's Name	Student's E-mail	Student's ID
Areej Turkey Alotaibi	atsalotaibi03@sm.imamu.edu.sa	440023303
Raghad Adel Alshabana	raaalshabana@sm.imamu.edu.sa	440021235
Slima Mohammed bnous	smbjlal@sm.imamu.edu.sa	439049380
Shoroog Saad Alarifi	sssalarifi@sm.imamu.edu.sa	440022128

Supervisor: Amjad Alamr



Table of Contents

1. Theoretical part.....	3
1.1 Introduction.....	3
1.1.1 Objectives and main concepts	3
1.2 Our problem	4
1.2.1 Explain the problem	4
1.2.2 problem formulation.....	4
1.2.3 A* Algorithm	5
1.2.4 Heuristic Function	5
1.3 Traveling salesman problem implementation Using A* algorithm.....	7
2. Practical part.....	8
2.1 The algorithm	8
2.1.1 Explanation.....	8
2.1.2 Implementation	9
2.1.4 Testing on several samples	15
3. Real Sample.....	18
4.futuerwork	19
4.1 Ant Colony Algorithm :	19
4.2 Partial Swarm Optimization Algorithm:	20
5. Table of Figures.....	21
6. References and resources.....	22



1. Theoretical part

1.1 Introduction

1.1.1 Objectives and main concepts

In recent years, drones have been used to aid delivery, according to COVID -19 pandemic that has greatly affected people around the world. Drones have helped people adhere to the rules of social distance by providing non-contact delivery services.

In addition, drones help with transportation in many ways, such as avoiding traffic congestion, making faster deliveries, and lowering transportation costs. Unlike delivery by truck, which has a longer range as it is fuel-based and trucks can carry heavy and large loads, which drones cannot. Nevertheless, traditional delivery by truck is slow and involves high transportation costs. As a team, we opted for drone delivery because it minimizes total cost of ownership and time.

However, we intend to cover the abstraction of this problem by ignoring the effects of the drone such as limit for the shipment weight, as drones cannot carry heavy wights, drones are battery-powered, thereby limiting their delivery ranges and we are not going to consider the charging process. In this project we will discuss a problem of Drone Delivery and try to solve it using the A* algorithm.



1.2 Our problem

1.2.1 Explain the problem

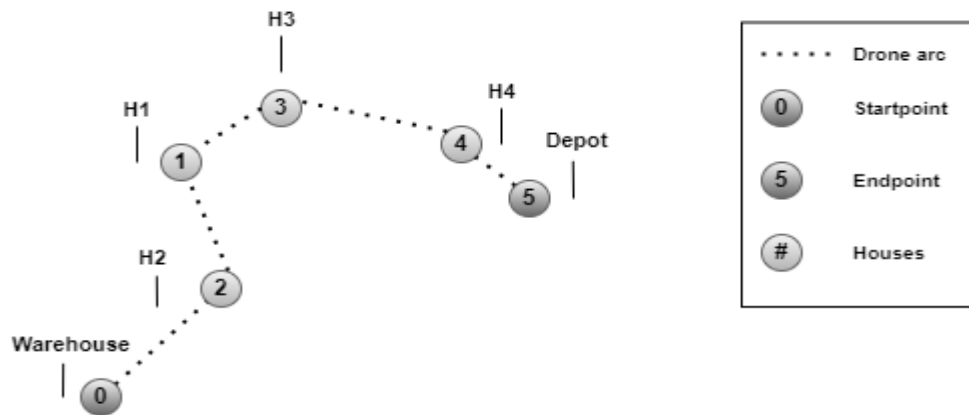


Figure 1-The Figure above shows the logical view of Drone Delivery graph

The drone delivery is a graph problem implemented with the A*-algorithm. It is defined on the graph $G=(V, A)$, where A is a set of arcs, each of which connects two nodes in V , as $V = \{0, 1, 2, 3, 4, 5\}$ where node 0 represent the starting point, which is the warehouse of shipments where the drone starts delivering, 5 represent the ending point, which is the depot of the drone after it has delivered all the shipments to the houses, nodes $\{1, 2, 3, 4\}$ are the set of houses.

The goal of this implementation is to find the tour with the shortest delivery path so that all houses are served by the drone.

There are constraints that need to be considered:

- The drone must start from the warehouse and end at the drone depot.
- Houses can only be served once.
- The drone can only fly to the depot if it has served all 4 houses.
- The drone searches for the overall cost to get shortest path that covered all houses.
- The drone has no rendezvous node before the drone depot node.

1.2.2 problem formulation

In order to understand the problem and simplify it, we must strip it of other details as we mentioned above such as the speed of the drone, delivery path congestion, weather, stopping to fill up battery ... etc.

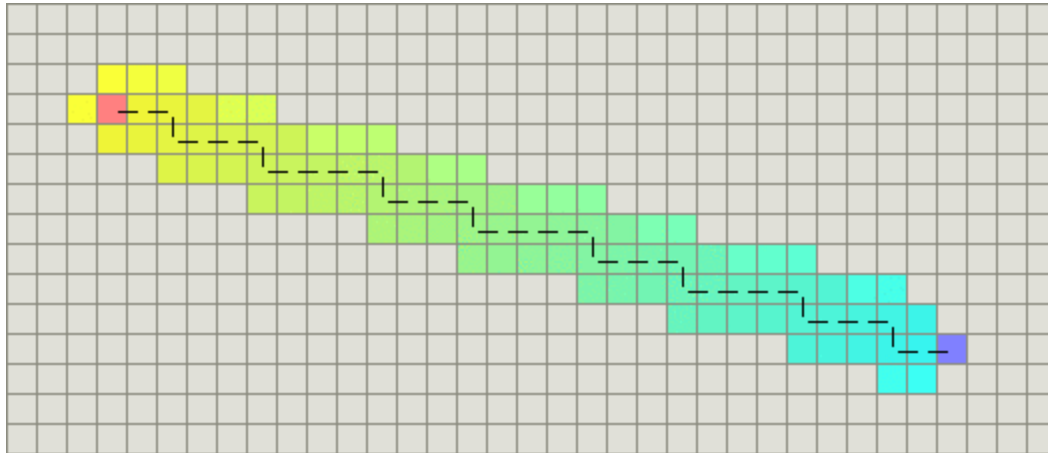
- Problem formulation:
 - o Initial state: Warehouse
 - o Actions: Go from warehouse/houses
 - o State space: each house that will be delivered, Warehouse and depot
 - o Goal test: reach the depot in min Time and all target houses are covered.



It's an Artificial Intelligence Algorithm and Informed Search type used to find the shortest possible path from start to end state. In our problem, we are trying to apply this algorithm with multiple heuristic functions to achieve the goal with other preferences. The comparison between heuristics will be discussed later.

We use 3 different heuristics, Manhattan, Euclidian, and Diagonal.

The Manhattan distance is a standard heuristic for a square grid. Look to your cost function and identify the lowest cost for going from one node to an adjacent node. It's also an application of the mathematical rule that the straight-line distance is the shortest.



The function formula as bellow:

```
function heuristic(node) =  
    dx = abs(node.x - goal.x)  
    dy = abs(node.y - goal.y)  
    return (dx + dy)
```



Diagonal Distance

It's a method where (4 northeast) allowed instead of 8 directions. Heuristic function should be $4 * D$ where D is the cost of moving diagonally, we have fixed value in the code as $D=5$

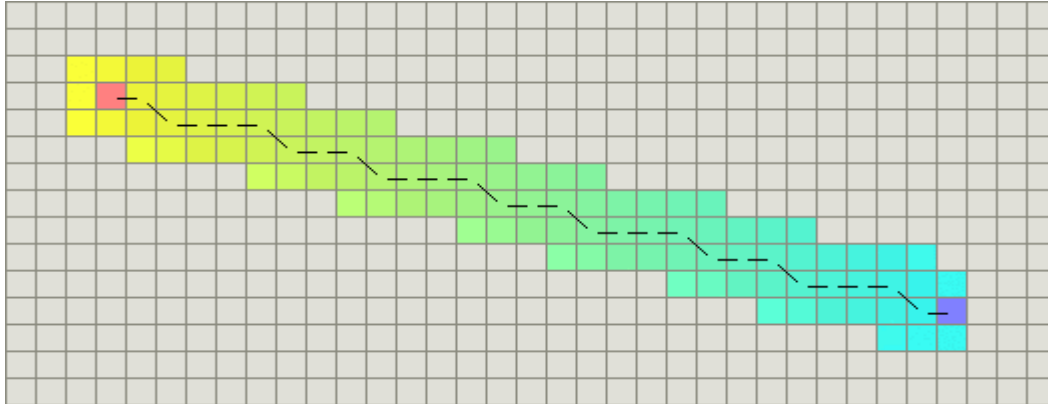


Figure 3-The figure above shows Diagonal flow and seems no difference from Manhattan function

The function formula as below: [1].

```
function heuristic(node) =  
    dx = abs(node.x - goal.x)  
    dy = abs(node.y - goal.y)  
    return D * max(dx, dy)
```

Euclidean Distance

The heuristic function where all directions are allowed to go with.

where D is the cost of moving diagonally, we have fixed value in the code as $D=1$

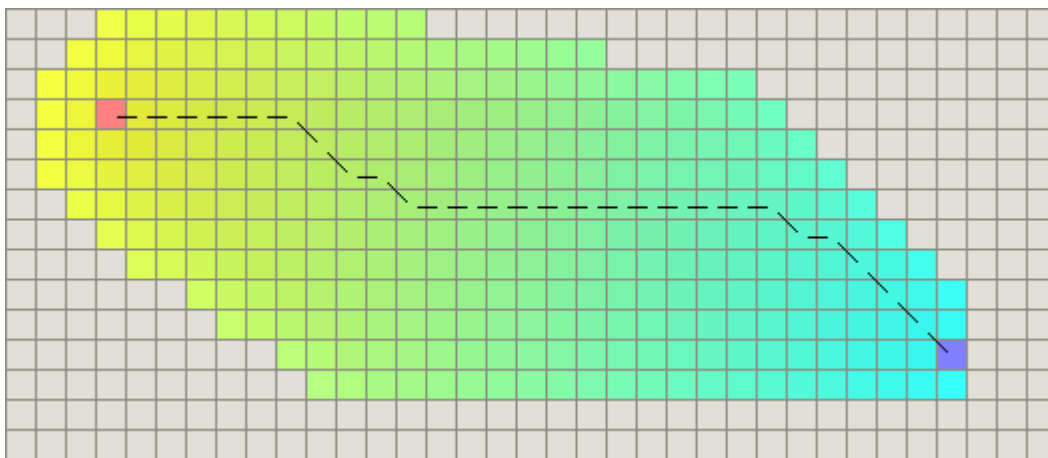


Figure 4-The figure above shows Euclidean flow and it seems lesser than the previous two functions

The function formula as below:

```
function heuristic(node) =  
    dx = abs(node.x - goal.x)  
    dy = abs(node.y - goal.y)  
    return D * sqrt(dx * dx + dy * dy)
```



After applying it with our problem we discover that each heuristic displays different result , but all of them try to give the best result. The admissible heuristic was Euclidean distance is shorter than Manhattan or diagonal distance and it gives an optimal path. , then the second heuristic after Euclidean was Manhattan heuristic , and the last one is diagonal .

1.3 Traveling salesman problem implementation Using A* algorithm

The traveling salesman problem, TSP for short, has model character in many branches of mathematics, computer science, and operations research. the problem is what is known as NP-hard. Is problem for which no polynomial time algorithm is known. (The time it takes is a polynomial function of the size of the input.) As the number of houses that the salesman needs to visit increases, the difficulty of solving the problem grows exceptionally quickly.

Our thought of TSP problem as a graph problem is that the houses being the vertices and the connections between them being the arcs.

The main idea of TSP says that " *salesman must visit all the vertices on a map exactly once, returning to the start vertex at the end of the journey. There is a direct connection from every vertex to every other vertex, and the salesman may visit the vertices in any order*" [2].

Important note we designed the code to match our constraints, the cyclic concept in TSP is not needed in our code and does not match our constraints so we get rid of the cycles and commit with one path without return.

TSP has many approach the very known approach is the Greedy but the one we use in our code is the naive approach .in contrast, both give order of $O(n!)$ time complexity for large scale of input, it sounds risky, but we don't have to worry about that since we have a countable scale of input "number of houses" which it is four.

The naive approach simply in version of TSP says the salesman is interested in visiting all four houses. It will try all the combination in this graph, for our graph it will try house#1 with all other houses, house#2 with all other houses, house#3 with all other houses, house#4 with all other houses. Moreover, the explanation of the time complexity is that for each n "house" will try $(n * n-1 * n-2 * n-3 * n-4 \dots)$ until there are no more houses which is $n!$ However, the Traveling salesman problem implementation Using A* algorithm is quite the same except that the TSP estimation now is the heuristic rather than the actual cost.

In conclusion, what we did is combine these two algorithms to match our constraint achieving that all houses are visited beside the estimation is the heuristic.



2. Practical part

2.1 The algorithm

2.1.1 Explanation

A* Algorithm goes as follow:

1. Take a list of goals (including the warehouse)
2. Send it to TSP to do its work
 - a. TSP as mentioned above, will calculate the permutation of all possible solution for this list. The warehouse SHOULD be the 1st node of all permutations since it's the start node.
 - b. The results path will be compared then by their total function, which's the addition of the heuristics and actual distance.
 - c. We consider all houses to be covered as sub goal; therefore the calculation will be based on the overall cost that included all houses NOT just to the final goal (Depot).
 - d. In a reversed way, pathDistance(shortestPath) will return the path from the Final node to the root of the path which is known by "Last" in the method.
3. The main method will initialize our sample graph (as shown in the figure 14) as Node Object. Including the Warehouse and Depot.
4. The user (or whom concern to know the path) should identify the distance between each pair of node. Including the Warehouse and Depot.
5. Node object automatically will calculate the neighbor information by addBranch methode; which is the edge between the main node and the neighbor and it will be known as actual cost. The heuristic will be also calculated automatically from the row and column information. The heuristic distance function will be calculated based on the user's choice, e.g., Manhattan, Diagonal..Etc. As well as the final function F(n) it's required data will be ready to added.
6. After add neighbors for each node and know all its information, we need to link them together to show us the connection, so we use dictionary to link each node with neighbor's function
7. Finally we get the path from findShortestPath method , and the distance from pathDistance method .

findShortestPath: this method will compute all possible paths then compare its distance , after get the minimum distance will return the shortest path.

pathDistance: this method will receive the shortest path and compute its distance as cumulative cost.

The shortest path will contain path from Warehouse to all houses. then we have dependent step by add 2 additional method to let drone go to the depot

GoToDepot: this method will get the last house in the shortest path then return its number to compute the distance from depot to the last house and add it to the total distance

NewPath: this method easily add the depot at end of the path

At the end will print the path with its distance by specific heuristic .



2.1.2 Implementation

```
class TSP:
    distances={ }
    initalNode=""
    def __init__(self,distances,initalNode):
        self.distances=distances
        self.initalNode=initalNode
    def swap(self,arr,first,second):
        temp=arr[first]
        arr[first]=arr[second]
        arr[second]=temp
    def allPermutationsHelper(self,permutation,permutations,n):
        if(n<=0):
            if(permutation[0] == self.initalNode):
                permutations.append(permutation)
                return
            tempPermutation=permutation+[]

            for i in range(n):
                self.swap(tempPermutation, i , n-1)

                self.allPermutationsHelper(tempPermutation,permutations,n-1)

                self.swap(tempPermutation,i,n-1)
    def permutations(self, original):
        permutations=[]

        self.allPermutationsHelper(original,permutations, len(original))

        return permutations

    def pathDistance(self, path):
        last=path[0]
        distance = 0

        for i in path[1:]:

            distance+=self.distances[last][i]
            last = i
        return distance
    def findShortestPath(self):
        cities=list(self.distances.keys())
        paths=self.permutations(cities)
        shortestpath=None
```



```
miniDistance=float ('inf')
```

```
for path in paths:
    distance=self.pathDistance(path)
    distance+=self.distances[path[len(path)-1]][path[0]]
    if(distance<miniDistance):
        miniDistance=distance
        shortestpath=path
    return shortestpath
def GoToDepote(self,path):
    LastHouse=path[len(path)-1] #path[len(path)-1:len(path)]
    if(LastHouse == "H1"):
        return 1
    if(LastHouse == "H2"):
        return 2
    if(LastHouse == "H3"):
        return 3
    if(LastHouse == "H4"):
        return 4

def NewPath(self,path):
    path.append("D")
    return path
```

```
import math
class Node:
    def __init__(self, row, col):
        self.row = row
        self.col = col
        self.neighbors = []

    def addBranch(self, weight, node):
        newEdge = Edge(self,node, weight)
        self.neighbors.append(newEdge)
    def toString(self):
        return "Node{ " + "row=" + str(self.row) + ", col=" + str(self.col) + " neighbors: " +
str(self.neighbors)
    def getRow(self):
        return self.row
    def setRow(self, row):
        self.row = row
    def getCol(self):
```



```
        return self.col
    def setCol(self, col):
        self.col = col
    def getNeighbor(self):
        return self.neighbors
    #####
class Edge(Node):
    DigonalCost = 2
    EculedianCost = 1
    def __init__(self, node1,node, g):
        self.node = node
        self.g = g
        self.node1=node1
        self.Dh = self.calculate_Digonal_Heuristic(node1)
        self.setDf(self.Dh)
        self.Eh = self.calculate_Euclidean_Heuristic(node1)
        self.setEf(self.Eh)
        self.Mh = self.calculate_Manhattan_Heuristic(node1)
        self.setMf(self.Mh)
    def setMf(self, Mh):
        self.Mf = self.Mh + self.g
    def setEf(self, Eh):
        self.Ef = self.Eh + self.g
    def setDf(self, Dh):
        self.Df = self.Dh + self.g
    def calculate_Manhattan_Heuristic(self,node1):
        self.Mh = abs(self.node.getRow() - self.node1.getRow()) + abs(self.node.getCol() -
self.node1.getCol())
        return self.Mh
    def calculate_Euclidean_Heuristic(self,node1):
        x = math.pow(abs(self.node1.getRow() - self.node.getRow()), 2)
        y = math.pow(abs(self.node1.getCol() - self.node.getCol()), 2)
        self.Eh = self.EculedianCost * math.sqrt(x + y)
        return self.Eh
    def calculate_Digonal_Heuristic(self,node1):
        dx = abs(self.node1.getRow() - self.node.getRow())
        dy = abs(self.node1.getCol() - self.node.getCol())
        self.Eh = self.DigonalCost * max(dx, dy)
        return self.Eh
```

Demo:



```
W=Node(0,0)
H1=Node(2,6)
H2=Node(1,3)
H3=Node(3,5)
H4=Node(4,2)
D=Node(5,6)
```

```
W.addBranch(110, H1);
W.addBranch(80, H2);
W.addBranch(130, H3);
W.addBranch(160, H4);
```

```
H1.addBranch(110, W);
H1.addBranch(40, H2);
H1.addBranch(30, H3);
H1.addBranch(80, H4);
```

```
H2.addBranch(80, W);
H2.addBranch(40, H1);
H2.addBranch(60, H3);
H2.addBranch(90, H4);
```

```
H3.addBranch(130, W);
H3.addBranch(30, H1);
H3.addBranch(60, H2);
H3.addBranch(60, H4);
```

```
H4.addBranch(160, W);
H4.addBranch(80, H1);
H4.addBranch(90, H2);
H4.addBranch(60, H3);
```

```
D.addBranch(170, W);
D.addBranch(100, H1);
D.addBranch(110, H2);
D.addBranch(80, H3);
D.addBranch(20, H4);
```

```
ManhattanDistances = {"W":
    {"H1": W.getNeighbor()[0].Mf,
     "H2": W.getNeighbor()[1].Mf,
     "H3": W.getNeighbor()[2].Mf,
     "H4": W.getNeighbor()[3].Mf
    },
    "H1":
```



```
{ "W": H1.getNeighbor()[0].Mf,
  "H2": H1.getNeighbor()[1].Mf,
  "H3": H1.getNeighbor()[2].Mf,
  "H4": H1.getNeighbor()[3].Mf
},
"H2":
{ "W": H2.getNeighbor()[0].Mf,
  "H1": H2.getNeighbor()[1].Mf,
  "H3": H2.getNeighbor()[2].Mf,
  "H4": H2.getNeighbor()[3].Mf
},
"H3":
{ "W": H3.getNeighbor()[0].Mf,
  "H1": H3.getNeighbor()[1].Mf,
  "H2": H3.getNeighbor()[2].Mf,
  "H4": H3.getNeighbor()[3].Mf
}, "H4":
{ "W": H4.getNeighbor()[0].Mf,
  "H1": H4.getNeighbor()[1].Mf,
  "H2": H4.getNeighbor()[2].Mf,
  "H3": H4.getNeighbor()[3].Mf
}
}
```

```
EculideanDistances = { "W":
  { "H1": W.getNeighbor()[0].Ef,
    "H2": W.getNeighbor()[1].Ef,
    "H3": W.getNeighbor()[2].Ef,
    "H4": W.getNeighbor()[3].Ef
  },
  "H1":
  { "W": H1.getNeighbor()[0].Ef,
    "H2": H1.getNeighbor()[1].Ef,
    "H3": H1.getNeighbor()[2].Ef,
    "H4": H1.getNeighbor()[3].Ef
  },
  "H2":
  { "W": H2.getNeighbor()[0].Ef,
    "H1": H2.getNeighbor()[1].Ef,
    "H3": H2.getNeighbor()[2].Ef,
    "H4": H2.getNeighbor()[3].Ef
  },
  "H3":
  { "W": H3.getNeighbor()[0].Ef,
    "H1": H3.getNeighbor()[1].Ef,
    "H2": H3.getNeighbor()[2].Ef,
```



```
"H4": H3.getNeighbor()[3].Ef
}, "H4":
{"W": H4.getNeighbor()[0].Ef,
"H1": H4.getNeighbor()[1].Ef,
"H2": H4.getNeighbor()[2].Ef,
"H3": H4.getNeighbor()[3].Ef
}
}
DiagonalDistances = {"W":
{"H1": W.getNeighbor()[0].Df,
"H2": W.getNeighbor()[1].Df,
"H3": W.getNeighbor()[2].Df,
"H4": W.getNeighbor()[3].Df
},
"H1":
{"W": H1.getNeighbor()[0].Df,
"H2": H1.getNeighbor()[1].Df,
"H3": H1.getNeighbor()[2].Df,
"H4": H1.getNeighbor()[3].Df
},
"H2":
{"W": H2.getNeighbor()[0].Df,
"H1": H2.getNeighbor()[1].Df,
"H3": H2.getNeighbor()[2].Df,
"H4": H2.getNeighbor()[3].Df
},
"H3":
{"W": H3.getNeighbor()[0].Df,
"H1": H3.getNeighbor()[1].Df,
"H2": H3.getNeighbor()[2].Df,
"H4": H3.getNeighbor()[3].Df
}, "H4":
{"W": H4.getNeighbor()[0].Df,
"H1": H4.getNeighbor()[1].Df,
"H2": H4.getNeighbor()[2].Df,
"H3": H4.getNeighbor()[3].Df
}
}

tsp= TSP(ManhattanDistances , "W")
shortestPath = tsp.findShortestPath()
LastHouse=tsp.GoToDepote(shortestPath)
distance = tsp.pathDistance(shortestPath)+int(D.getNeighbor()[(LastHouse)].g)
print("The path is " + str(tsp.NewPath(shortestPath)) + " in " +str(distance) + " miles." + " With
Manhattan Heuristic")
```



```
tsp2= TSP(EculideanDistances , "W")
shortestPath2 = tsp2.findShortestPath()
LastHouse2=tsp2.GoToDepote(shortestPath2)
distance2 =int(tsp2.pathDistance(shortestPath2)+int(D.getNeighbor()[(LastHouse2)].g) )
print("The path is " + str(tsp2.NewPath(shortestPath2)) + " in " +str(distance2) + " miles." + "
With Eculidean Heuristic (optimal resut)")
```

```
tsp3= TSP(DiagonalDistances , "W")
shortestPath3 = tsp3.findShortestPath()
LastHouse3=tsp3.GoToDepote(shortestPath3)
distance3 =int(tsp3.pathDistance(shortestPath3)+int(D.getNeighbor()[(LastHouse3)].g) )
print("The path is " + str(tsp3.NewPath(shortestPath3)) + " in " +str(distance3) + " miles." + "
With Diagonal Heuristic")
```

2.1.4 Testing on several samples

Sample1:

	0	1	2	3	4	5	6
0	W	-	-	-	-	-	-
1	-	-	-	H2	-	-	-
2	-	-	-	-	-	-	H1
3	-	-	-	-	-	H3	-
4	-	-	H4	-	-	-	-
5	-	-	-	-	-	-	D

Figure 5-SAMPLE:1 "basic sample"

	0	1	2	3	4	5	6
0	W	-	-	-	-	-	-
1	-	-	-	H2	-	-	-
2	-	-	-	-	-	-	H1
3	-	-	-	-	-	-	H3
4	-	-	H4	-	-	-	-
5	-	-	-	-	-	-	D

Figure 6- SAMPLE:1-path "basic path"

Current Path:

The path is ['W', 'H4', 'H3', 'H1', 'H2', 'D'] in 416 miles. With Manhattan Heuristic
The path is ['W', 'H2', 'H1', 'H3', 'H4', 'D'] in 240 miles. With Eculidean Heuristic (optimal result)
The path is ['W', 'H4', 'H3', 'H1', 'H2', 'D'] in 455 miles. With Diagonal Heuristic

Figure 7- SAMPLE:1-path "output"



Sample 2:

	0	1	2	3	4	5	6
0	W	-	H2	-	-	-	-
1	-	-	-	-	-	-	H3
2	-	-	-	H4	-	-	-
3	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	H1	-	D

Figure 8-SAMPLE2:

	0	1	2	3	4	5	6
0	W	→	H2	-	-	-	-
1	-	-	-	↘	-	-	→ H3
2	-	-	-	-	H4	→	-
3	-	-	-	-	-	-	-
4	-	-	-	-	-	-	-
5	-	-	-	-	-	H1	↘ D

Figure 9 -SAMPLE:2, "path"

Path :

The path is ['W', 'H2', 'H4', 'H3', 'H1', 'D'] in 375 miles. With Manhattan Heuristic
The path is ['W', 'H2', 'H4', 'H3', 'H1', 'D'] in 371 miles. With Eculidean Heuristic (optimal resut)
The path is ['W', 'H2', 'H4', 'H3', 'H1', 'D'] in 382 miles. With Diagonal Heuristic

Figure 10-SAMPLE:2 path "output"



Sample 3:

	0	1	2	3	4	5	6
0	W	-	-	-	-	-	-
1	-	-	-	-	H3	-	-
2	-	-	-	-	-	-	-
3	-	H2	-	-	H1	-	-
4	-	-	-	-	-	-	H4
5	-	-	-	-	-	-	D

Figure 11-SAMPLE3

	0	1	2	3	4	5	6
0	W	-	-	-	-	-	-
1	-	-	-	-	H3	-	-
2	-	-	-	-	-	-	-
3	-	H2	-	-	H1	-	-
4	-	-	-	-	-	-	H4
5	-	-	-	-	-	-	D

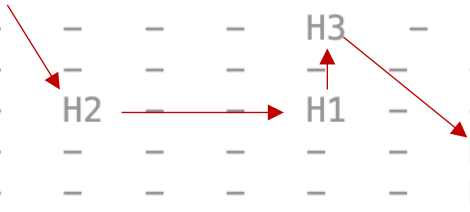


Figure 12 -SAMPLE:3,"path"

Path :

The path is ['W', 'H2', 'H4', 'H3', 'H1', 'D'] in 377 miles. With Manhattan Heuristic
The path is ['W', 'H2', 'H4', 'H3', 'H1', 'D'] in 373 miles. With Eculidean Heuristic (optimal resut)
The path is ['W', 'H2', 'H4', 'H3', 'H1', 'D'] in 386 miles. With Diagonal Heuristic

Figure 13- SAMPLE:3-path "output"

Note: Arcs does not mean path for specific heuristic, just for directions



3. Real Sample

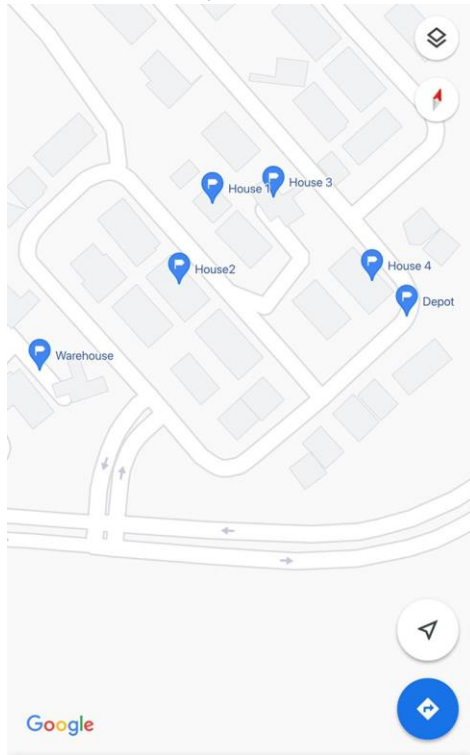


Figure 14: the figure above shows the real view of drone delivery map.



Figure 15: GIF.



4.futuerwork

With research and investigation, we have come up with algorithms called metaheuristic, which are a variety of algorithms inspired by nature that help in improving the work of tsp, to reach a better solution and a more efficient result .Some of the Meta-Hypothesis Methods for Improving the Efficiency of TSP like PSO and Ant colony

4.1 Ant Colony Algorithm :

Ant colony algorithm is a tool for solving TSP which was proposed by Durrigo et al in 1992. This algorithm which is an example of multi-operator systems is inspired from food finding behavior of real ants, where each operator is an artificial ant . In general, in order to improve TSP using ant colony

4 steps are required including:

- 1) initializing parameters including number of ants m , pheromone α , heuristic function operator β , pheromone evaporation ρ , amount of pheromone, maximum iteration.
- 2) construction solution space; that is each ant is located at several starting points, cities to be visited are computed and it continues until all cities are visited.
- 3) Updating pheromone; length of each path through which ants pass and record of the optimal solution in the current counter is calculated.

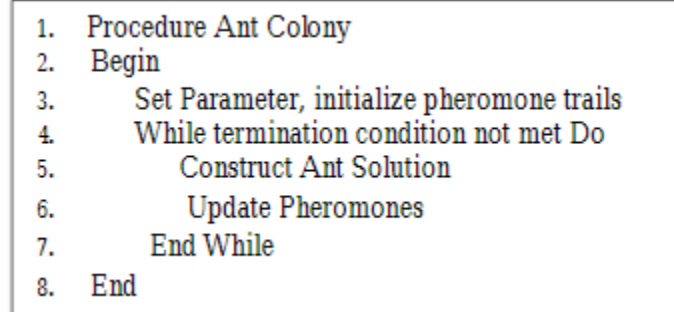


Figure 16 :Ant Colony Algorithm

- 4) Determining stopping condition; determining whether number of iterations has reached its maximum or not. If it has not, one unit is added to counter and associated record is erased and returns to constructing the solution space. Otherwise, it is stopped.



4.2 Partial Swarm Optimization Algorithm:

PSO is one of the most intelligent algorithms in swarm intelligence area. This algorithm is inspired from social behavior of animals like fishes and birds (which live together in small or large groups) and was introduced by James Kennedy and Russel Aberhurt in 1995. In PSO, members of the population are connected directly and interact through exchanging information and remembering memories. PSO is suitable for solving a wide range of continuous and discrete problems and has presented a lot of suitable solution is wide range of optimization problems.

```
1. Procedure Particle Swarm
2. Begin
3.   For each particle
4.     Initialize particle
5.   For each particle
6.     Calculate cost
7.     If the cost is better than the best cost (pbest)
        in history
8.       Set current value as the new pbest
9.   Choose particle with best cost of all the
        particles as gbest
10.  Update particle position
11.  While maximum iterations or optimum result
12.  End
```

Figure 17: PSO algorithm

In the future, the TSP algorithm can be studied with metaheuristic algorithms for make better results.



5. Table of Figures

Figure 1-The Figure above shows the logical view of Drone Delivery graph.....	4
Figure 2-The Figure above shows how Manhattan flow	5
Figure 3-The figure above shows Diagonal flow and seems no difference from Manhattan function	6
Figure 4-The figure above shows Euclidean flow and it seems lesser than the previous two functions.....	6
Figure 5- SAMPLE:1 "basic sample"	10
Figure 6- SAMPLE:1-path1 "basic path"	10
Figure 7- SAMPLE:1-path1 "output"	10
Figure 8- SAMPLE:2.....	11
Figure 9- SAMPLE:2-path "output"	11
Figure 10- SAMPLE:3.....	12
Figure 11- SAMPLE:3.....	12
Figure 12- SAMPLE:3 path.....	12
Figure 13- SAMPLE:3path "output"	12
Figure 14- the figure above shows the real view of drone delivery map.....	13
Figure 15- GIF	13
Figure 16- Ant Colony Algorithm	14
Figure 17- PSO algorithm.....	15



6. References and resources

- [1] <http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html>
- [2] <https://blogs.oracle.com/javamagazine/post/how-to-solve-the-classic-traveling-salesman-problem-in-java>
- [3] https://www.researchgate.net/publication/320961227_Meta-Heuristic_Approaches_for_Solving_Travelling_Salesman_Problem
- [4] <https://earth.google.com/web/>