Heuristic and Approximation Algorithms and Applications in Robotics and AI

Professor Eugene Levner

Students : Nurlan Aghazada, Ali Hasanov

Home Works

1) Advantages and Dangers of Artificial Intelligence Algorithms

Advantages:

1. Increase work efficiency -> Unlike people, these machines can work continuously. For example, AI-powered chat assistants can answer consumer queries and provide help to visitors every minute of the day and enhance the sales of a company.
2. High accuracy -> AI remove human's errors from their tasks to achieve accurate results every time they do that specific task. Artificial intelligence powered machines can solve complex equations and perform critical tasks on their own so that the results obtained have higher accuracy as compared to their human counterparts.
3. Reduce cost of training and operation -> AI machines that optimize their machine learning abilities so that they learn much faster about new processes. This way the cost of training robots would become much lesser than that of humans.
4. Helping in Repetitive Jobs -> In our day-to-day work, we will be performing many repetitive works like sending a thanking mail, verifying certain documents for errors and many more things. We may use artificial intelligence to efficiently automate these daily tasks and even to eliminate "boring" tasks for people, allowing them to focus on being more creative.

Dangers:

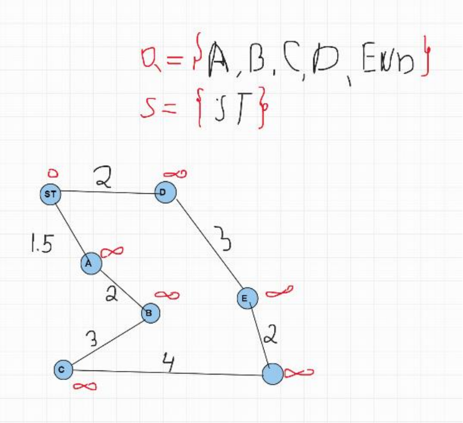
1. Loss of workplaces -> Machines conduct routine and repeating activities far better than humans. Many firms would prefer machines instead of

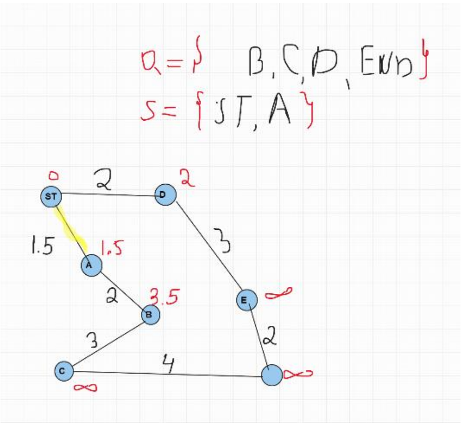
humans to boost their profitability, therefore diminishing the jobs that are available for the human worker.

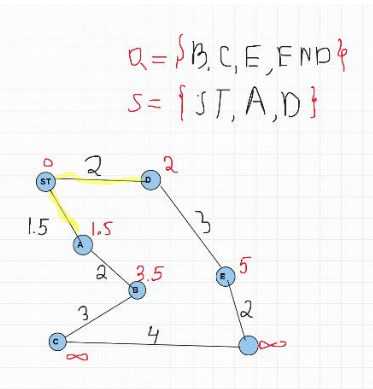
2. High cost of Maintenance -> As AI is updating every day the hardware and software need to get updated with time to meet the latest requirements. Machines need repairing and maintenance which need plenty of costs
3. Unpredictable Future-> One of the smartest people now involved in AI research is Elon Musk. He has also stated openly that AI is the biggest threat to human civilization in the future. Thus, the dystopian future depicted in sci-fi films is not implausible.

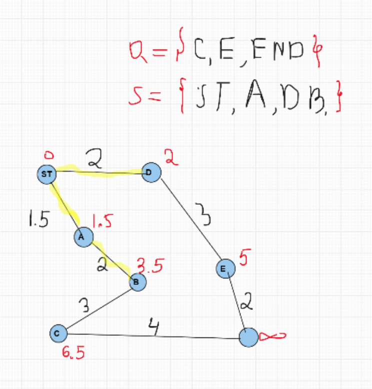
2) Solution of 3 graphs in L1-L3 by Dijkstra Algorithm manually, step-by-step.

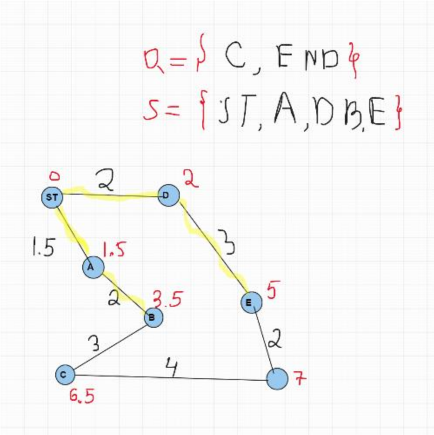
1.Graph

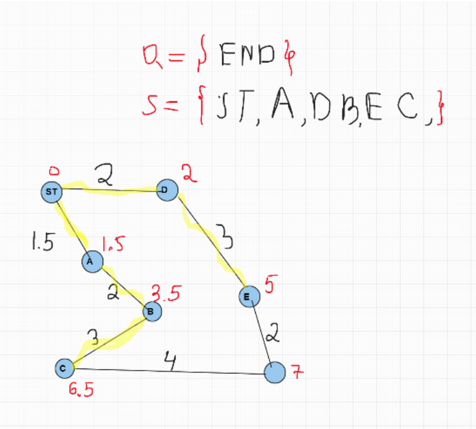


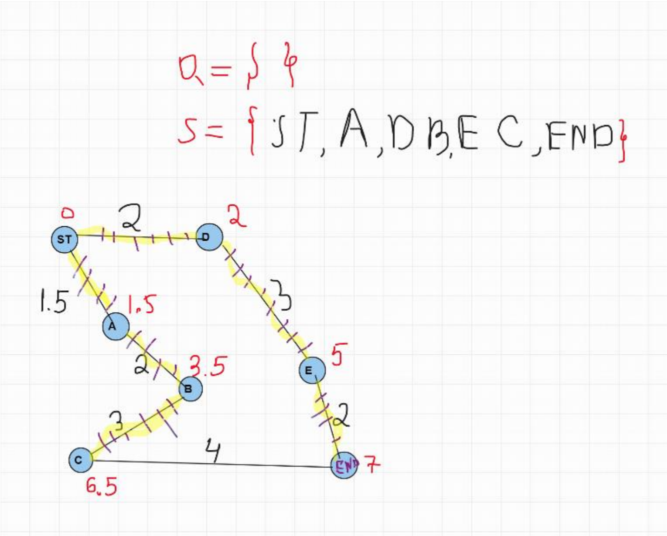






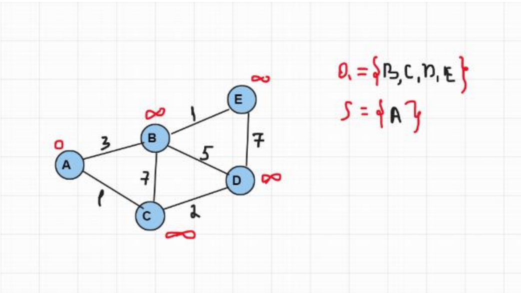


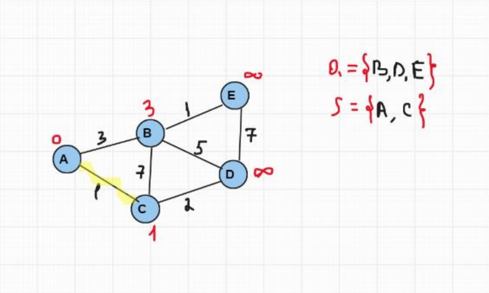


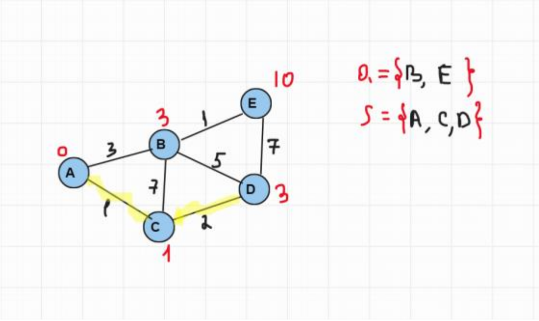


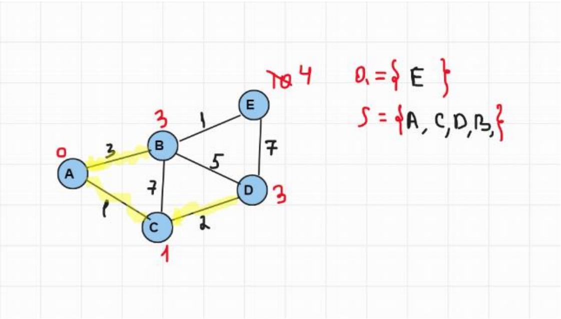
Shortest Path will be: ST→D→E→END

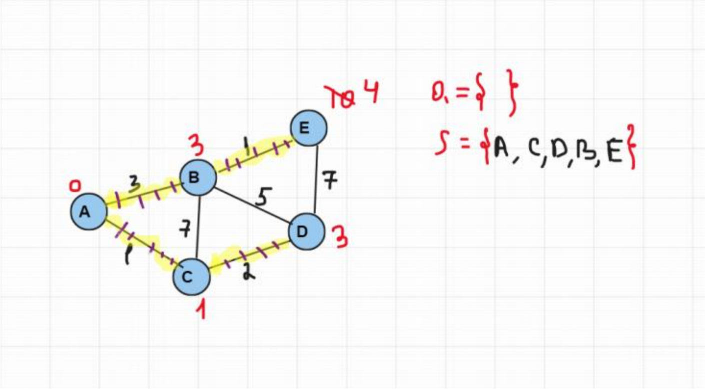
2.Graph







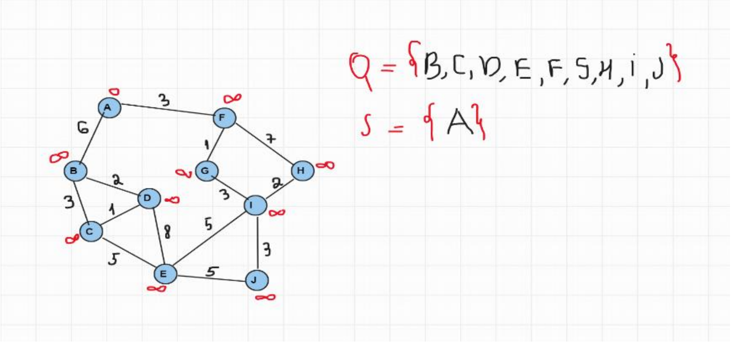


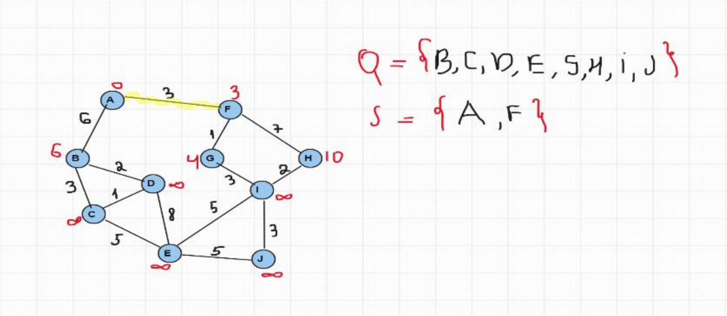


Shortest Path will be:

A→B→E

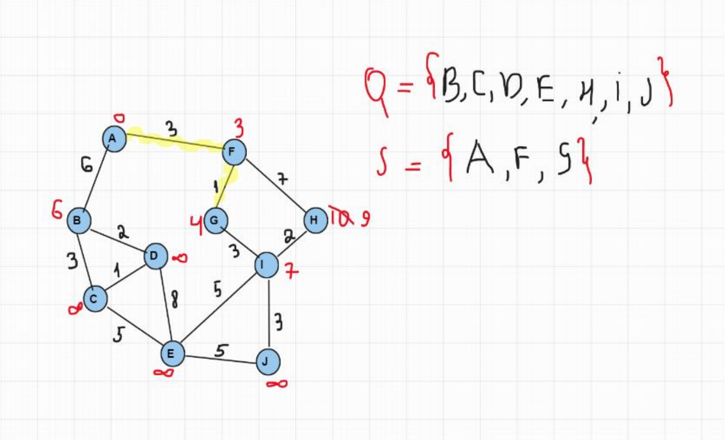
3.Graph





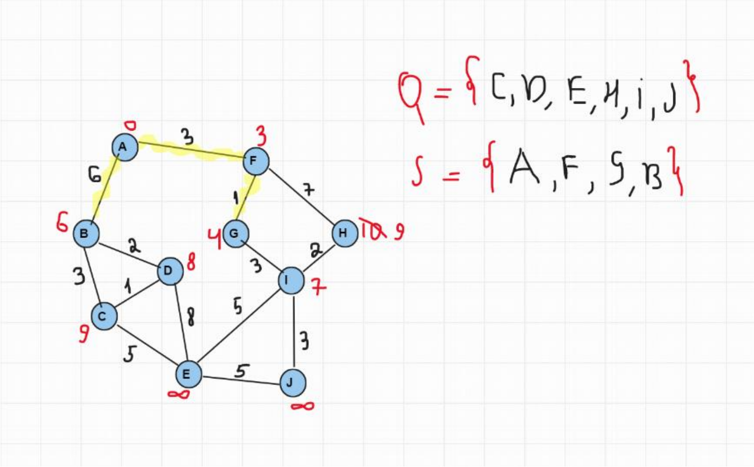
Q = {B, C, D, E, G, H, i, J}

S = {A, F}



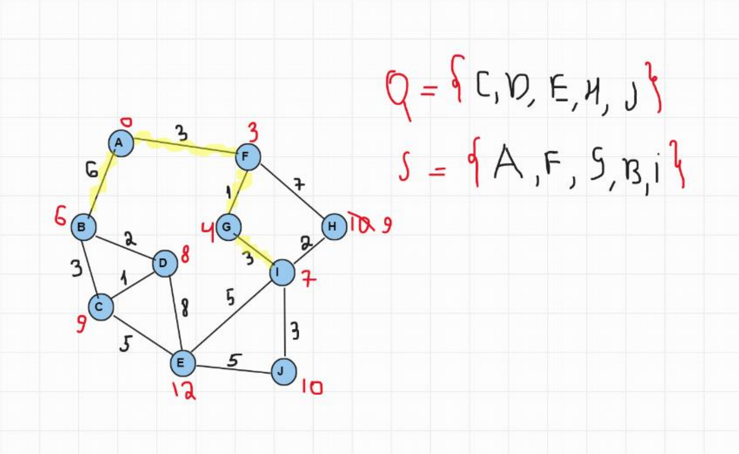
Q = {B, C, D, E, H, i, J}

S = {A, F, G}



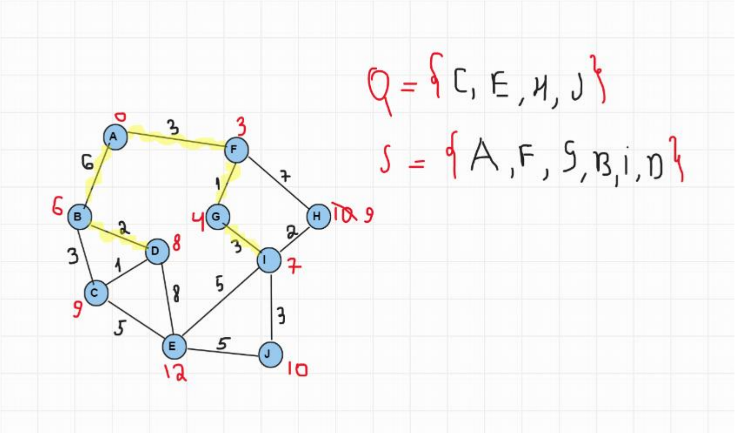
Q = {C, D, E, H, i, J}

S = {A, F, G, B}



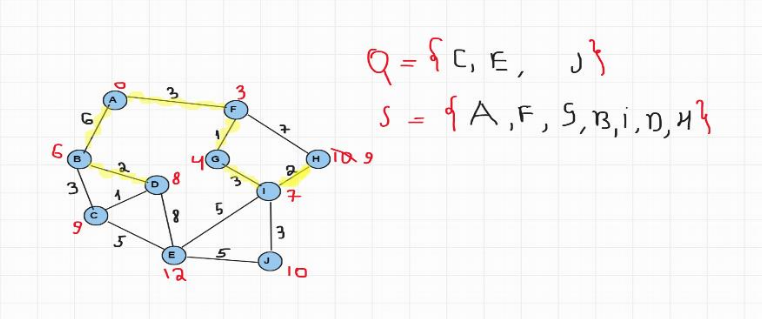
Q = {C, D, E, H, J}

S = {A, F, G, B, I}



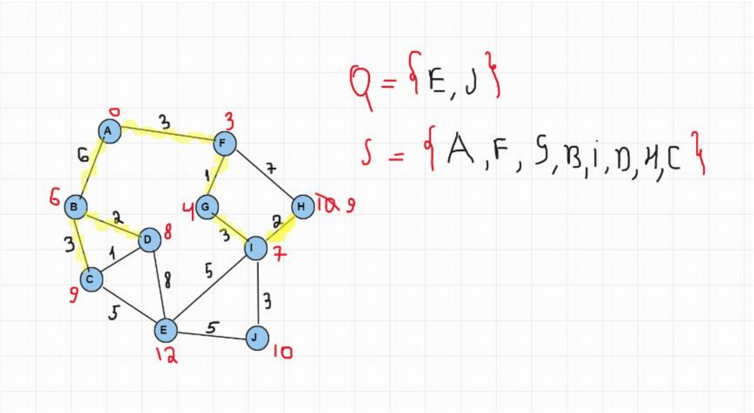
Q = {C, E, H, J}

S = {A, F, G, B, I, D}



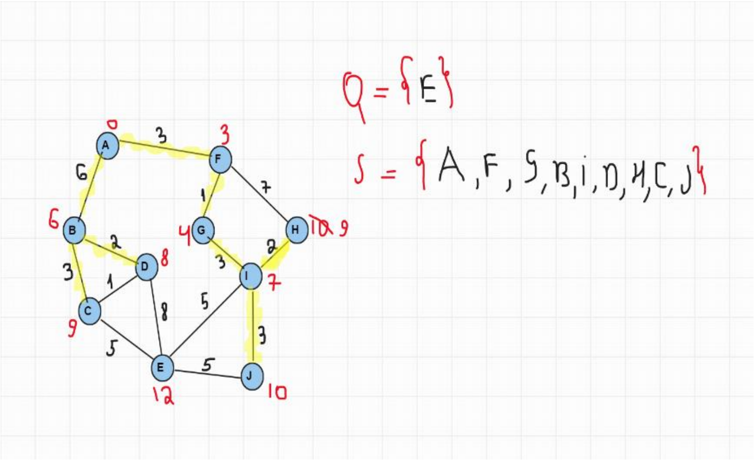
Q = {C, E, J}

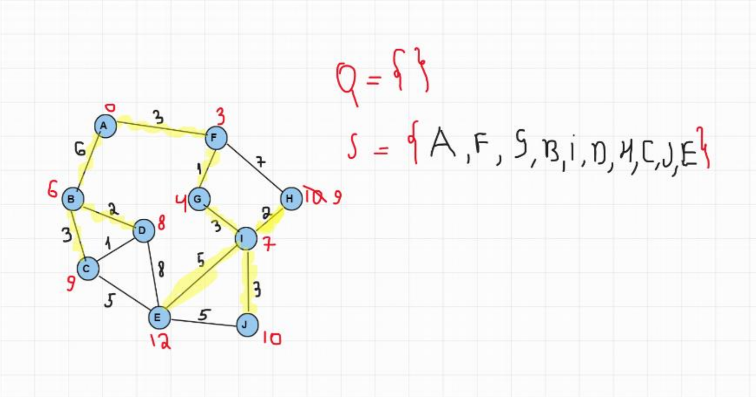
S = {A, F, G, B, I, D, H}

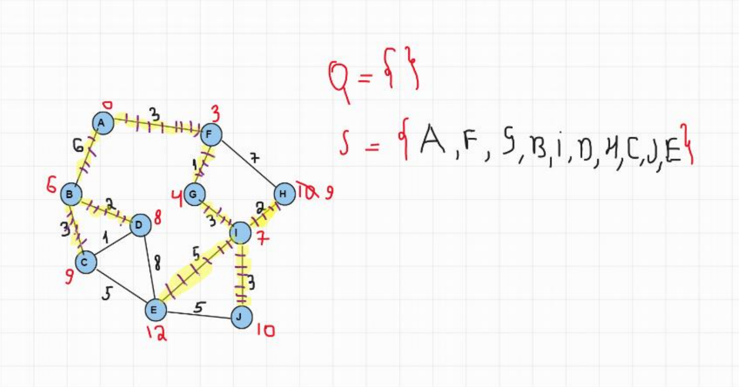


Q = {E, J}

S = {A, F, G, B, I, D, H, C}



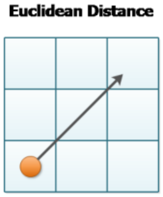




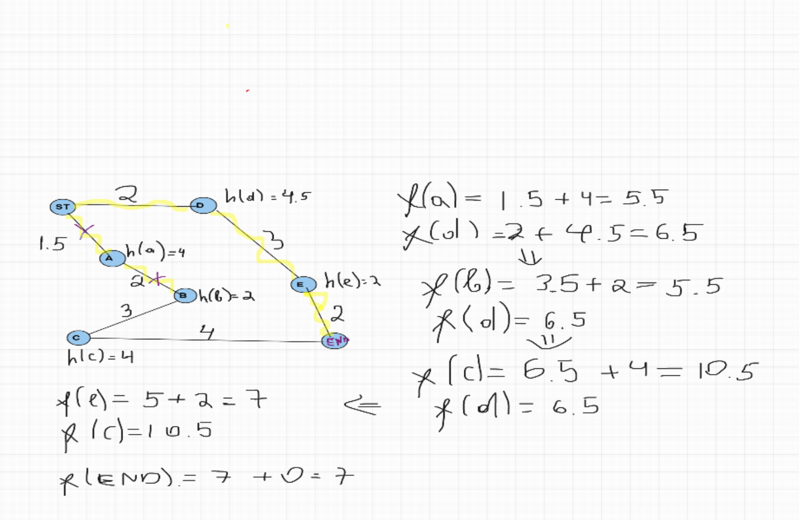
Shortest Path Will be :A→F→G→I→J

3)Solution of 3 graphs in L1-L3 by A* Algorithm manually, step-by-step, for Eucledean metrics and Manhattan metrics.

1.Graph With Euclidean Metric:

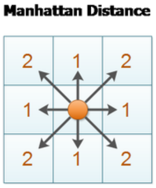


$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

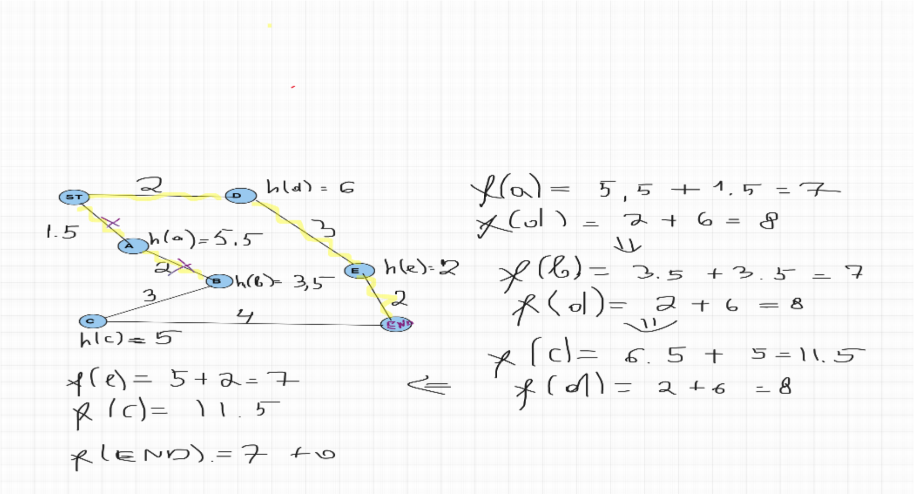


$f(a) = 1.5 + 4 = 5.5$
$f(d) = 2 + 4.5 = 6.5$
$\Downarrow$
$f(b) = 3.5 + 2 = 5.5$
$f(d) = 6.5$
$\Downarrow$
$f(c) = 6.5 + 4 = 10.5$
$f(d) = 6.5$
$\Leftarrow$
$f(e) = 5 + 2 = 7$
$f(c) = 10.5$
$f(END) = 7 + 0 = 7$

With Manhattan Metric

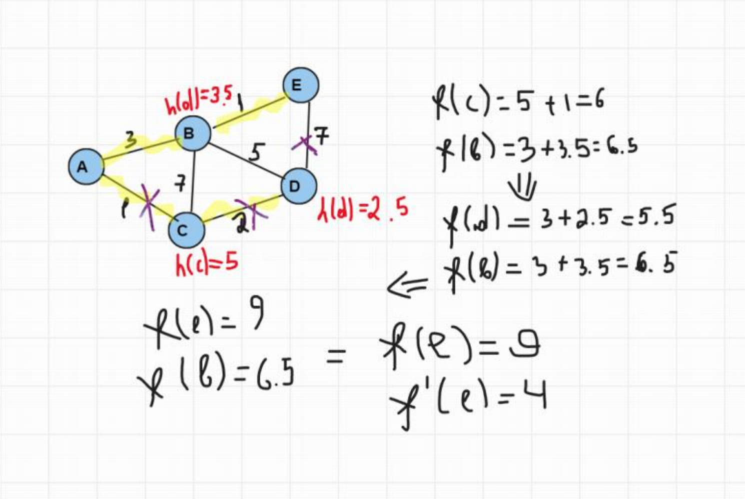


: $|x_1 - x_2| + |y_1 - y_2|$

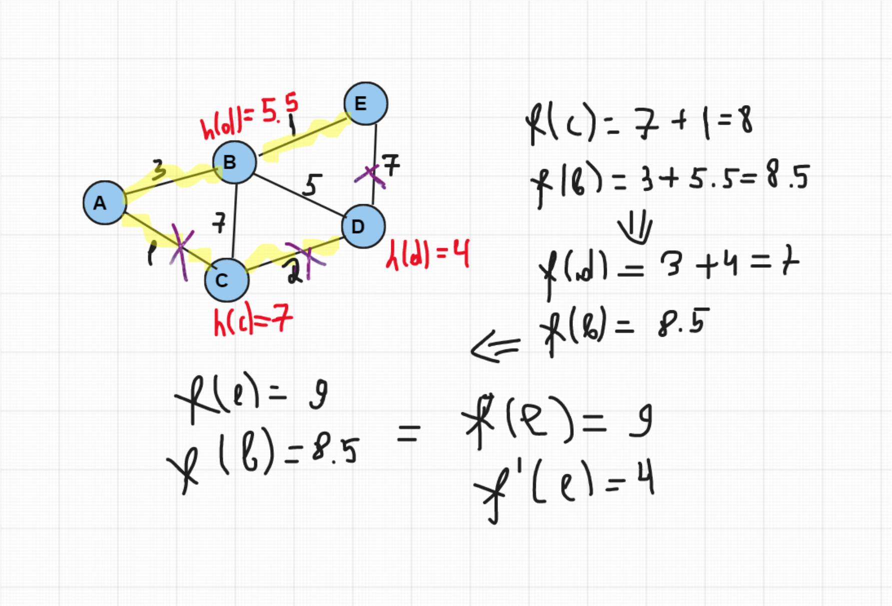


$f(a) = 5,5 + 1.5 = 7$
$f(d) = 2 + 6 = 8$
$\Downarrow$
$f(b) = 3.5 + 3.5 = 7$
$f(d) = 2 + 6 = 8$
$\Downarrow$
$f(c) = 6.5 + 5 = 11.5$
$f(d) = 2 + 6 = 8$
$\Leftarrow$
$f(e) = 5 + 2 = 7$
$f(c) = 11.5$
$f(END) = 7 + 0$

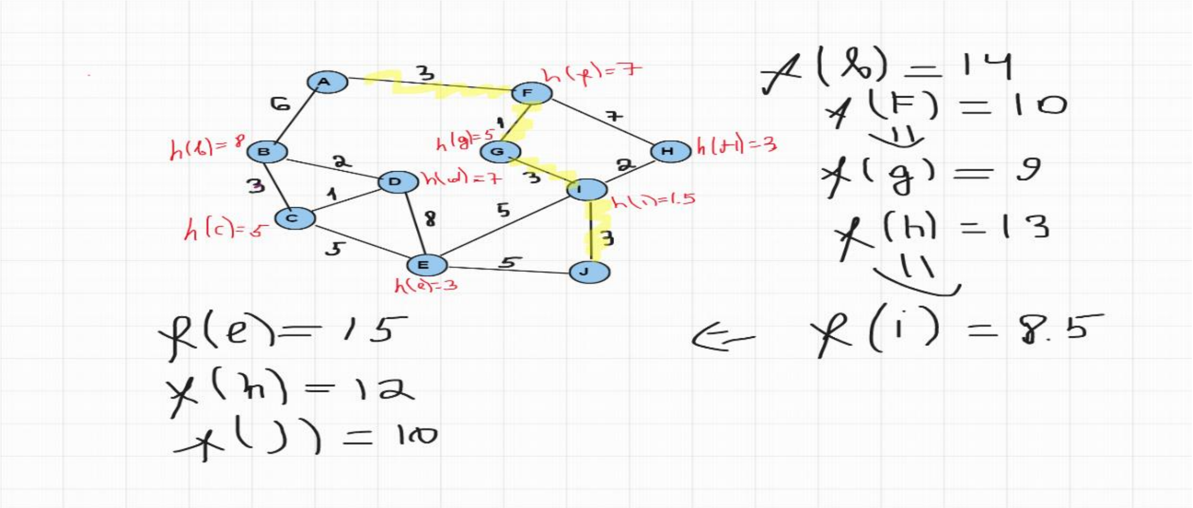
2.Graph With Euclidean Metric:



With Manhattan Metric :



3.Graph With Euclidean Metric:



$f(b) = 14$

$f(F) = 10$

⇓

$f(g) = 9$

$f(h) = 13$

⇓

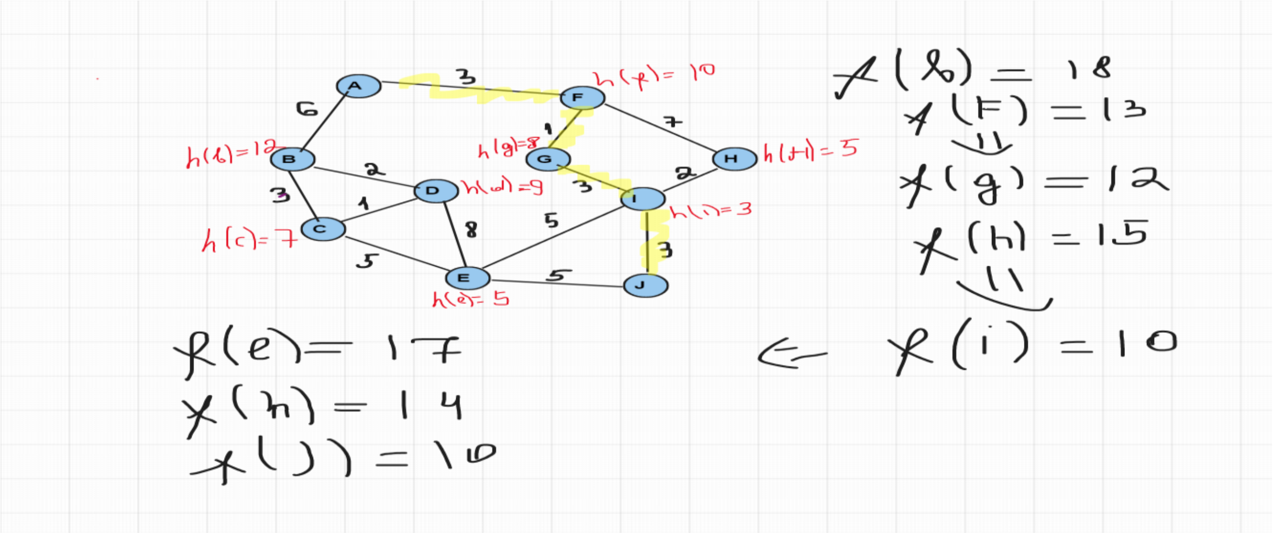
⇐ $f(i) = 8.5$

$f(e) = 15$

$f(h) = 12$

$f(j) = 10$

With Manhattan Metric :



$f(b) = 18$

$f(F) = 13$

⇓

$f(g) = 12$

$f(h) = 15$

⇓

⇐ $f(i) = 10$

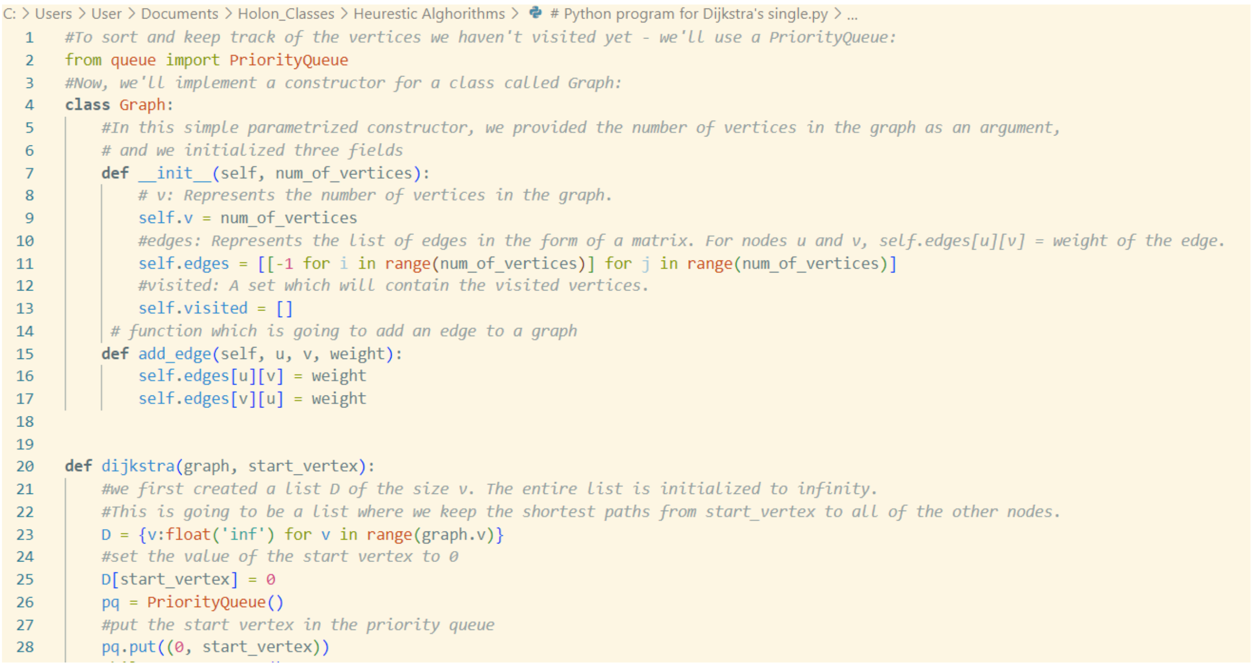
$f(e) = 17$

$f(h) = 14$

$f(j) = 10$

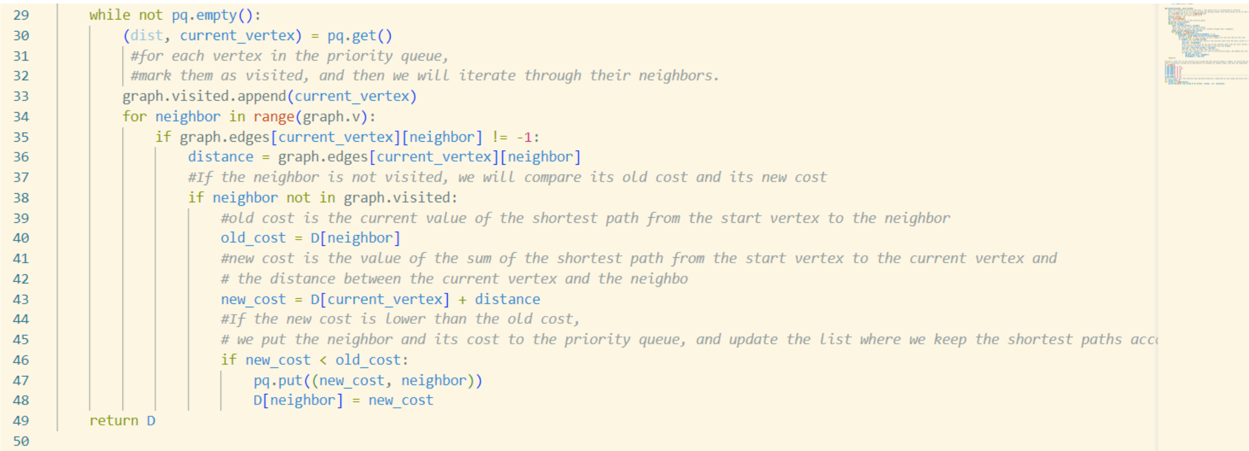
4-5) HW4-HW5. Solution of 3 graphs in L1-L3 by Dijkstra and A* Algorithm by PYTHON , step-by-step, with Eucledean metrics and Manhattan metrics.

1.Graph solution with Phyton using Djikstra Alghortihm:

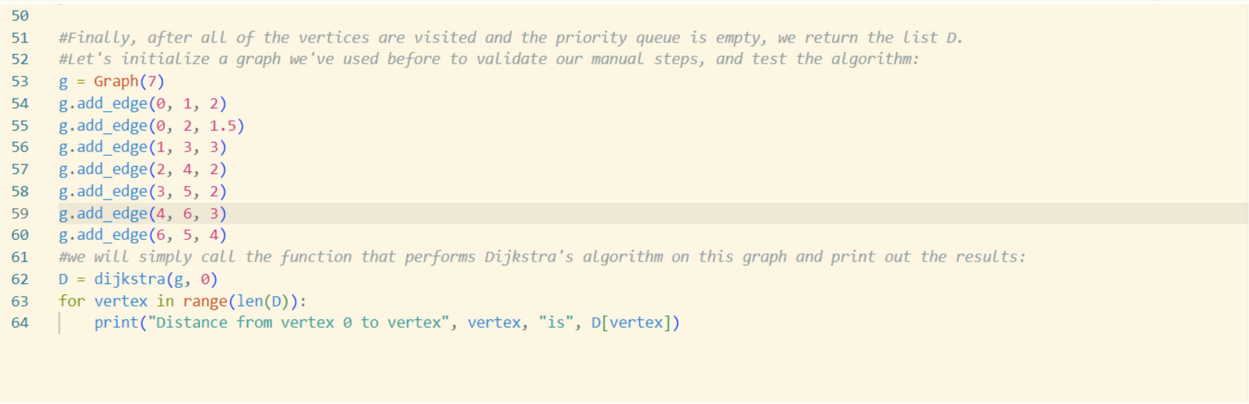


C: > Users > User > Documents > Holon_Classes > Heurestic Alghorithms > # Python program for Dijkstra's single.py > ...

```python
#To sort and keep track of the vertices we haven't visited yet - we'll use a PriorityQueue:
from queue import PriorityQueue
#Now, we'll implement a constructor for a class called Graph:
class Graph:
    #In this simple parametrized constructor, we provided the number of vertices in the graph as an argument,
    # and we initialized three fields
    def __init__(self, num_of_vertices):
        # v: Represents the number of vertices in the graph.
        self.v = num_of_vertices
        #edges: Represents the list of edges in the form of a matrix. For nodes u and v, self.edges[u][v] = weight of the edge.
        self.edges = [[-1 for i in range(num_of_vertices)] for j in range(num_of_vertices)]
        #visited: A set which will contain the visited vertices.
        self.visited = []
     # function which is going to add an edge to a graph
    def add_edge(self, u, v, weight):
        self.edges[u][v] = weight
        self.edges[v][u] = weight


def dijkstra(graph, start_vertex):
    #we first created a list D of the size v. The entire list is initialized to infinity.
    #This is going to be a list where we keep the shortest paths from start_vertex to all of the other nodes.
    D = {v:float('inf') for v in range(graph.v)}
    #set the value of the start vertex to 0
    D[start_vertex] = 0
    pq = PriorityQueue()
    #put the start vertex in the priority queue
    pq.put((0, start_vertex))
```

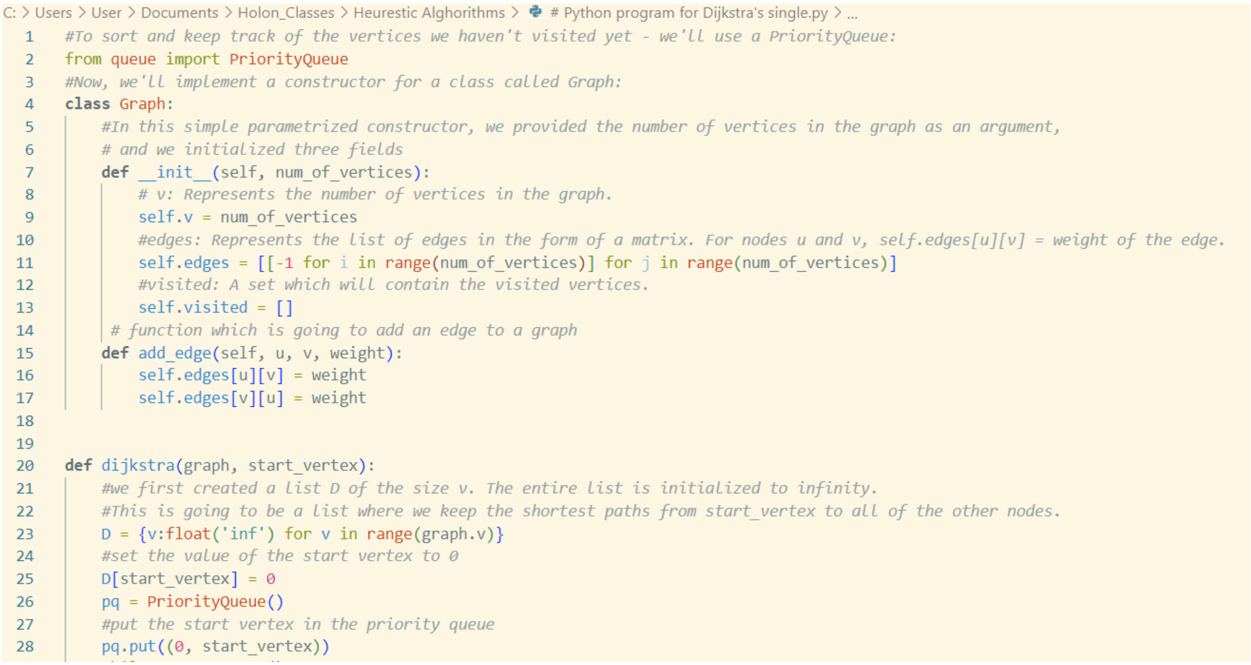


```python
    while not pq.empty():
        (dist, current_vertex) = pq.get()
        #for each vertex in the priority queue,
        #mark them as visited, and then we will iterate through their neighbors.
        graph.visited.append(current_vertex)
        for neighbor in range(graph.v):
            if graph.edges[current_vertex][neighbor] != -1:
                distance = graph.edges[current_vertex][neighbor]
                #If the neighbor is not visited, we will compare its old cost and its new cost
                if neighbor not in graph.visited:
                    #old cost is the current value of the shortest path from the start vertex to the neighbor
                    old_cost = D[neighbor]
                    #new cost is the value of the sum of the shortest path from the start vertex to the current vertex and
                    # the distance between the current vertex and the neighbo
                    new_cost = D[current_vertex] + distance
                    #If the new cost is lower than the old cost,
                    # we put the neighbor and its cost to the priority queue, and update the list where we keep the shortest paths acc
                    if new_cost < old_cost:
                        pq.put((new_cost, neighbor))
                        D[neighbor] = new_cost
    return D

```



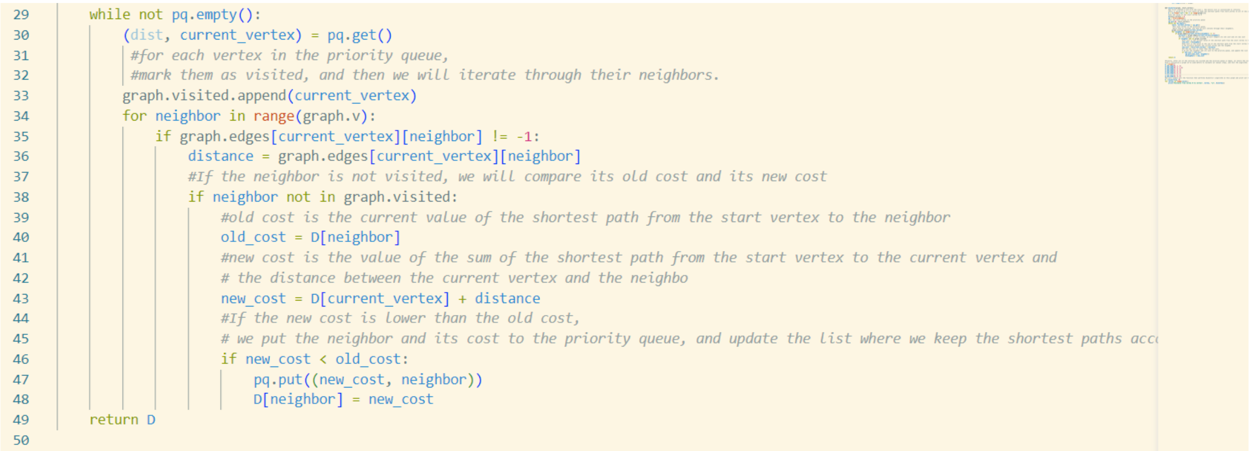
```python

#Finally, after all of the vertices are visited and the priority queue is empty, we return the list D.
#Let's initialize a graph we've used before to validate our manual steps, and test the algorithm:
g = Graph(7)
g.add_edge(0, 1, 2)
g.add_edge(0, 2, 1.5)
g.add_edge(1, 3, 3)
g.add_edge(2, 4, 2)
g.add_edge(3, 5, 2)
g.add_edge(4, 6, 3)
g.add_edge(6, 5, 4)
#we will simply call the function that performs Dijkstra's algorithm on this graph and print out the results:
D = dijkstra(g, 0)
for vertex in range(len(D)):
    print("Distance from vertex 0 to vertex", vertex, "is", D[vertex])
```

2.Graph solution with Phyton using Djikstra Alghortihm:

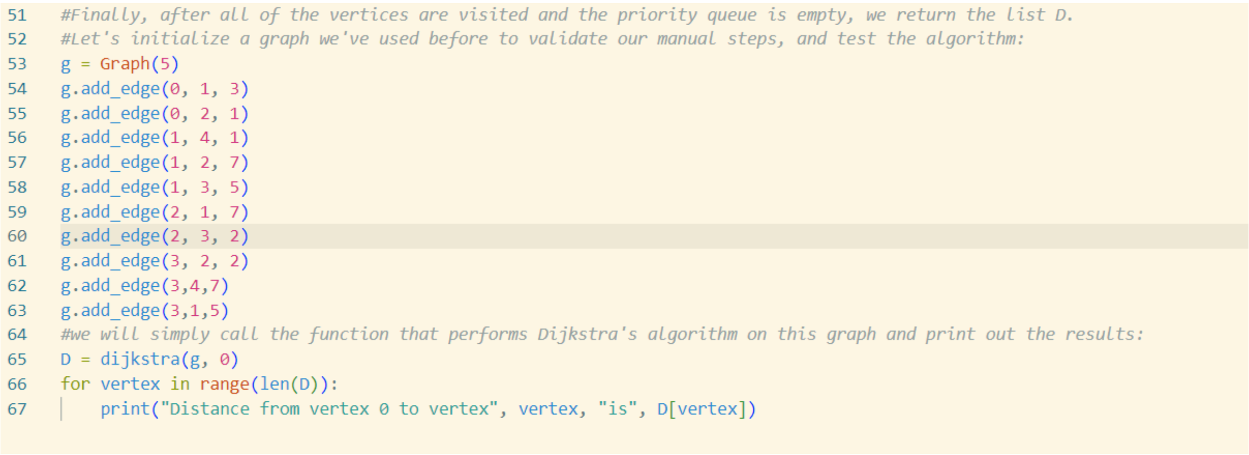


C: > Users > User > Documents > Holon_Classes > Heurestic Alghorithms > # Python program for Dijkstra's single.py > ...

```python
#To sort and keep track of the vertices we haven't visited yet - we'll use a PriorityQueue:
from queue import PriorityQueue
#Now, we'll implement a constructor for a class called Graph:
class Graph:
    #In this simple parametrized constructor, we provided the number of vertices in the graph as an argument,
    # and we initialized three fields
    def __init__(self, num_of_vertices):
        # v: Represents the number of vertices in the graph.
        self.v = num_of_vertices
        #edges: Represents the list of edges in the form of a matrix. For nodes u and v, self.edges[u][v] = weight of the edge.
        self.edges = [[-1 for i in range(num_of_vertices)] for j in range(num_of_vertices)]
        #visited: A set which will contain the visited vertices.
        self.visited = []
     # function which is going to add an edge to a graph
    def add_edge(self, u, v, weight):
        self.edges[u][v] = weight
        self.edges[v][u] = weight


def dijkstra(graph, start_vertex):
    #we first created a list D of the size v. The entire list is initialized to infinity.
    #This is going to be a list where we keep the shortest paths from start_vertex to all of the other nodes.
    D = {v:float('inf') for v in range(graph.v)}
    #set the value of the start vertex to 0
    D[start_vertex] = 0
    pq = PriorityQueue()
    #put the start vertex in the priority queue
    pq.put((0, start_vertex))
```

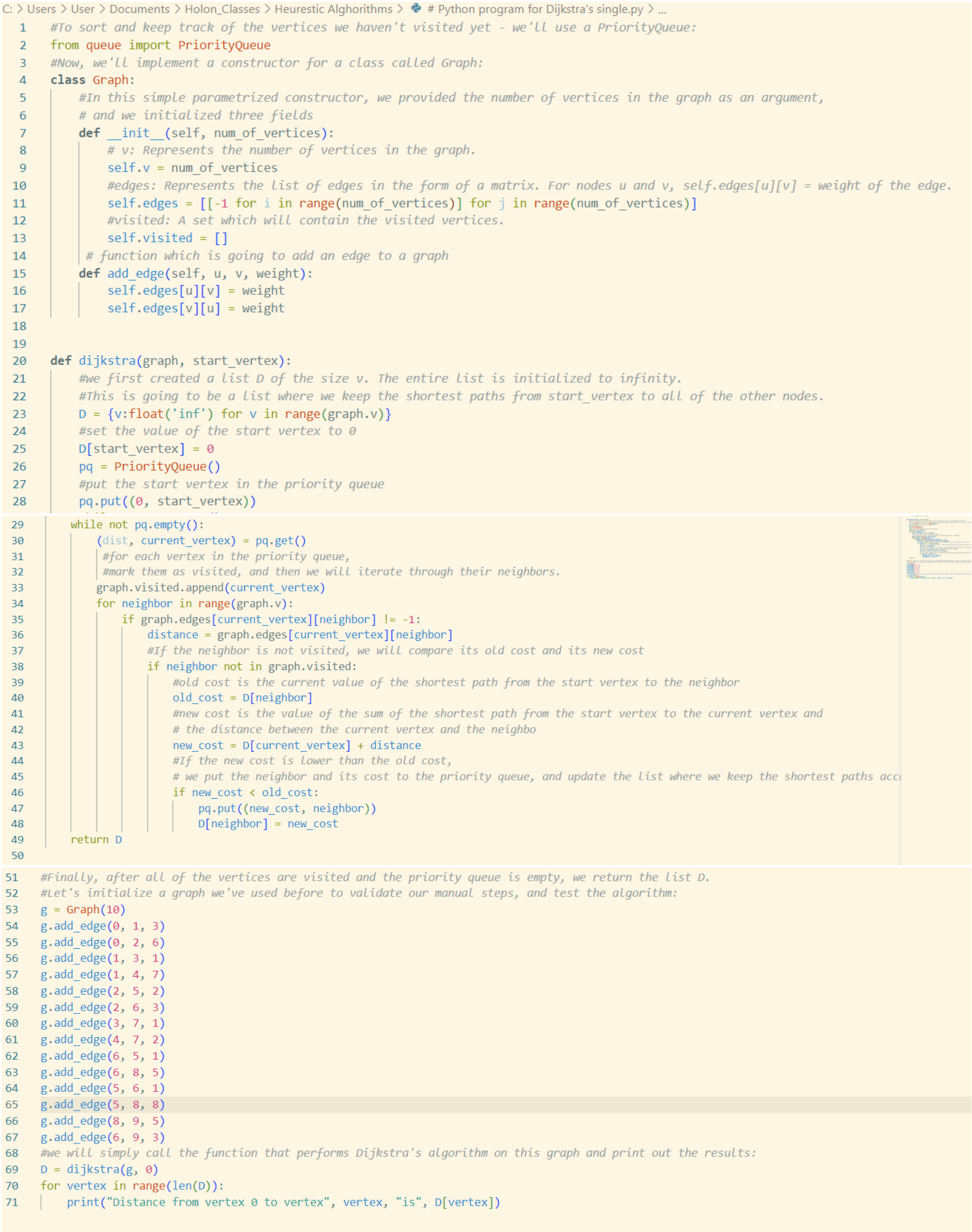


```python
    while not pq.empty():
        (dist, current_vertex) = pq.get()
        #for each vertex in the priority queue,
        #mark them as visited, and then we will iterate through their neighbors.
        graph.visited.append(current_vertex)
        for neighbor in range(graph.v):
            if graph.edges[current_vertex][neighbor] != -1:
                distance = graph.edges[current_vertex][neighbor]
                #If the neighbor is not visited, we will compare its old cost and its new cost
                if neighbor not in graph.visited:
                    #old cost is the current value of the shortest path from the start vertex to the neighbor
                    old_cost = D[neighbor]
                    #new cost is the value of the sum of the shortest path from the start vertex to the current vertex and
                    # the distance between the current vertex and the neighbo
                    new_cost = D[current_vertex] + distance
                    #If the new cost is lower than the old cost,
                    # we put the neighbor and its cost to the priority queue, and update the list where we keep the shortest paths acc
                    if new_cost < old_cost:
                        pq.put((new_cost, neighbor))
                        D[neighbor] = new_cost
    return D

```



```python
#Finally, after all of the vertices are visited and the priority queue is empty, we return the list D.
#Let's initialize a graph we've used before to validate our manual steps, and test the algorithm:
g = Graph(5)
g.add_edge(0, 1, 3)
g.add_edge(0, 2, 1)
g.add_edge(1, 4, 1)
g.add_edge(1, 2, 7)
g.add_edge(1, 3, 5)
g.add_edge(2, 1, 7)
g.add_edge(2, 3, 2)
g.add_edge(3, 2, 2)
g.add_edge(3,4,7)
g.add_edge(3,1,5)
#we will simply call the function that performs Dijkstra's algorithm on this graph and print out the results:
D = dijkstra(g, 0)
for vertex in range(len(D)):
    print("Distance from vertex 0 to vertex", vertex, "is", D[vertex])
```

3. Graph solution with Phyton using Djikstra Alghortihm:

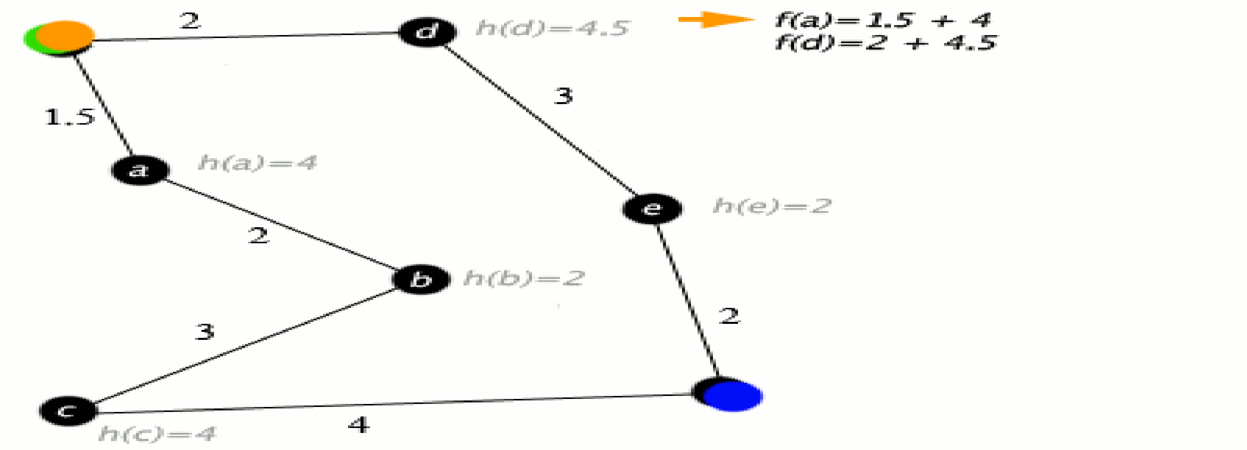


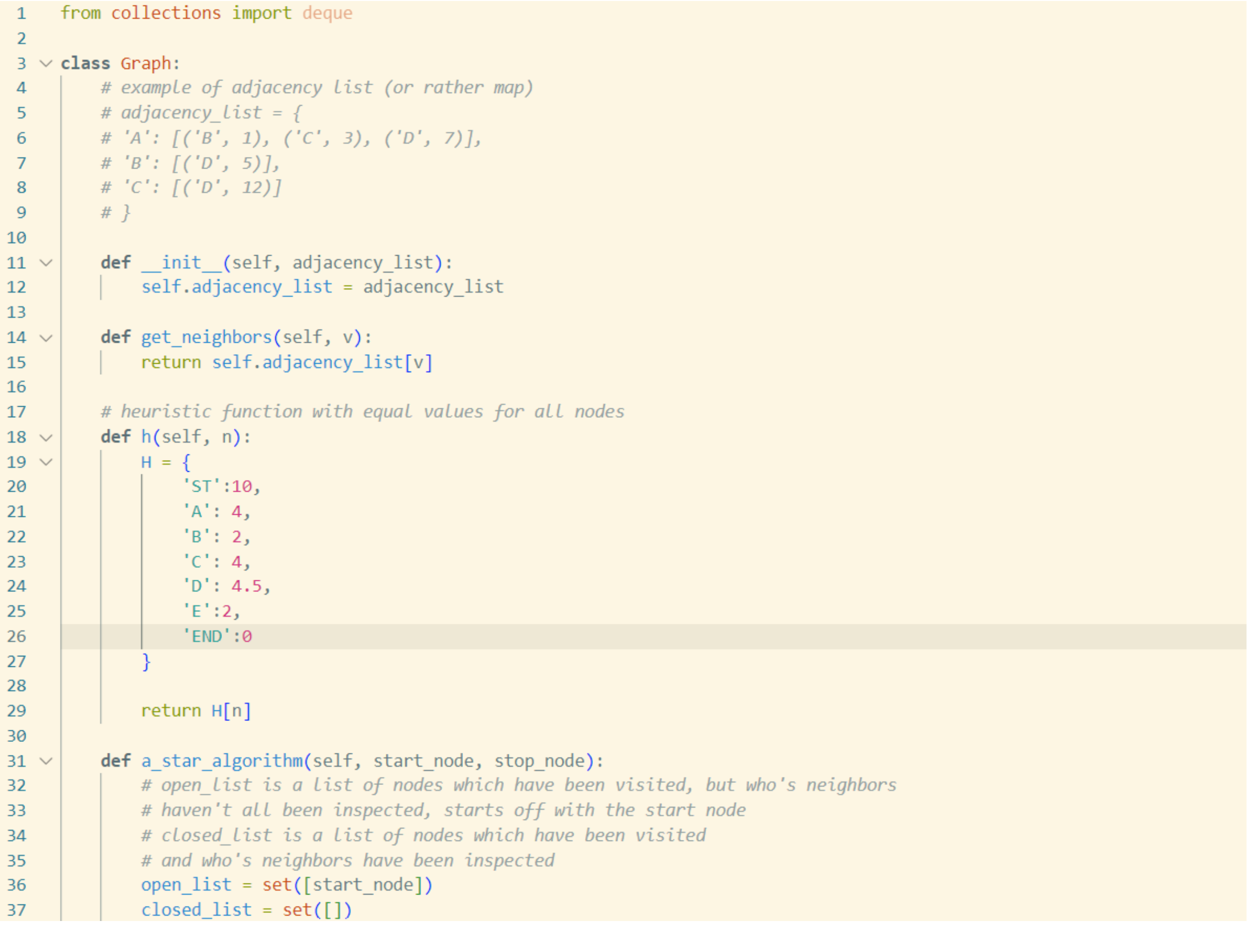
C: > Users > User > Documents > Holon_Classes > Heurestic Alghorithms > # Python program for Dijkstra's single.py > ...

```python
#To sort and keep track of the vertices we haven't visited yet - we'll use a PriorityQueue:
from queue import PriorityQueue
#Now, we'll implement a constructor for a class called Graph:
class Graph:
    #In this simple parametrized constructor, we provided the number of vertices in the graph as an argument,
    # and we initialized three fields
    def __init__(self, num_of_vertices):
        # v: Represents the number of vertices in the graph.
        self.v = num_of_vertices
        #edges: Represents the list of edges in the form of a matrix. For nodes u and v, self.edges[u][v] = weight of the edge.
        self.edges = [[-1 for i in range(num_of_vertices)] for j in range(num_of_vertices)]
        #visited: A set which will contain the visited vertices.
        self.visited = []
    # function which is going to add an edge to a graph
    def add_edge(self, u, v, weight):
        self.edges[u][v] = weight
        self.edges[v][u] = weight


def dijkstra(graph, start_vertex):
    #we first created a list D of the size v. The entire list is initialized to infinity.
    #This is going to be a list where we keep the shortest paths from start_vertex to all of the other nodes.
    D = {v:float('inf') for v in range(graph.v)}
    #set the value of the start vertex to 0
    D[start_vertex] = 0
    pq = PriorityQueue()
    #put the start vertex in the priority queue
    pq.put((0, start_vertex))
    while not pq.empty():
        (dist, current_vertex) = pq.get()
        #for each vertex in the priority queue,
        #mark them as visited, and then we will iterate through their neighbors.
        graph.visited.append(current_vertex)
        for neighbor in range(graph.v):
            if graph.edges[current_vertex][neighbor] != -1:
                distance = graph.edges[current_vertex][neighbor]
                #If the neighbor is not visited, we will compare its old cost and its new cost
                if neighbor not in graph.visited:
                    #old cost is the current value of the shortest path from the start vertex to the neighbor
                    old_cost = D[neighbor]
                    #new cost is the value of the sum of the shortest path from the start vertex to the current vertex and
                    # the distance between the current vertex and the neighbo
                    new_cost = D[current_vertex] + distance
                    #If the new cost is lower than the old cost,
                    # we put the neighbor and its cost to the priority queue, and update the list where we keep the shortest paths acc
                    if new_cost < old_cost:
                        pq.put((new_cost, neighbor))
                        D[neighbor] = new_cost
    return D

#Finally, after all of the vertices are visited and the priority queue is empty, we return the list D.
#Let's initialize a graph we've used before to validate our manual steps, and test the algorithm:
g = Graph(10)
g.add_edge(0, 1, 3)
g.add_edge(0, 2, 6)
g.add_edge(1, 3, 1)
g.add_edge(1, 4, 7)
g.add_edge(2, 5, 2)
g.add_edge(2, 6, 3)
g.add_edge(3, 7, 1)
g.add_edge(4, 7, 2)
g.add_edge(6, 5, 1)
g.add_edge(6, 8, 5)
g.add_edge(5, 6, 1)
g.add_edge(5, 8, 8)
g.add_edge(8, 9, 5)
g.add_edge(6, 9, 3)
#we will simply call the function that performs Dijkstra's algorithm on this graph and print out the results:
D = dijkstra(g, 0)
for vertex in range(len(D)):
    print("Distance from vertex 0 to vertex", vertex, "is", D[vertex])
```

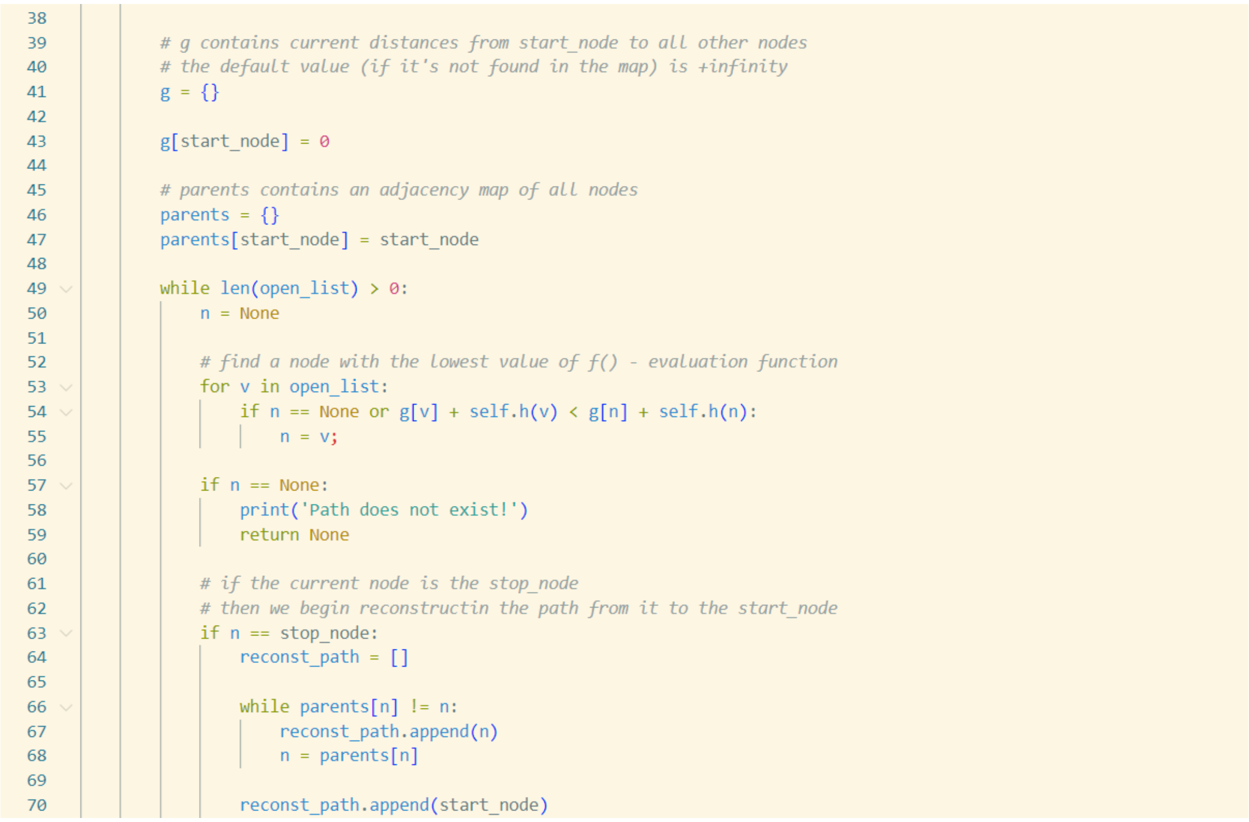
Credit: https://stackabuse.com/courses/graphs-in-python-theory-and-implementation/lessons/dijkstras-algorithm/

1.Graph solution with Phyton using A* Algorithm using Euclidean distance:

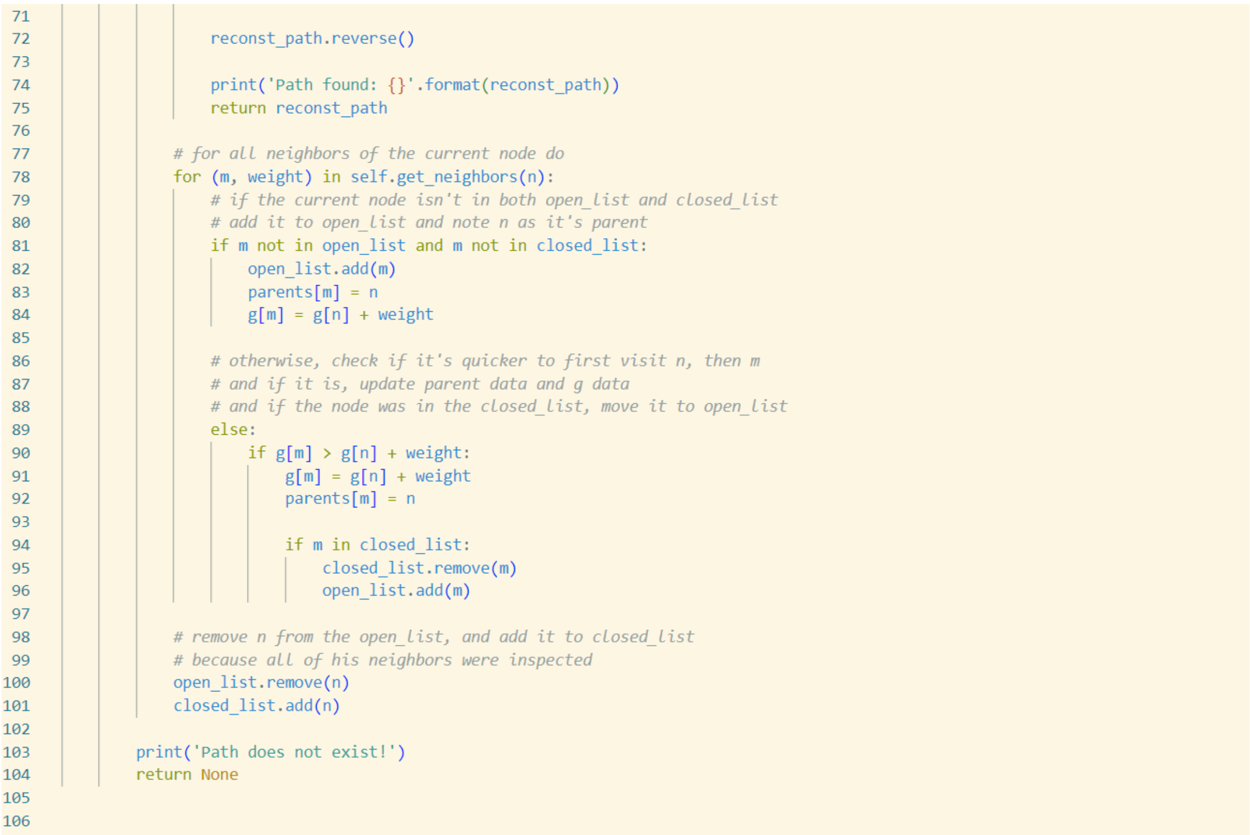




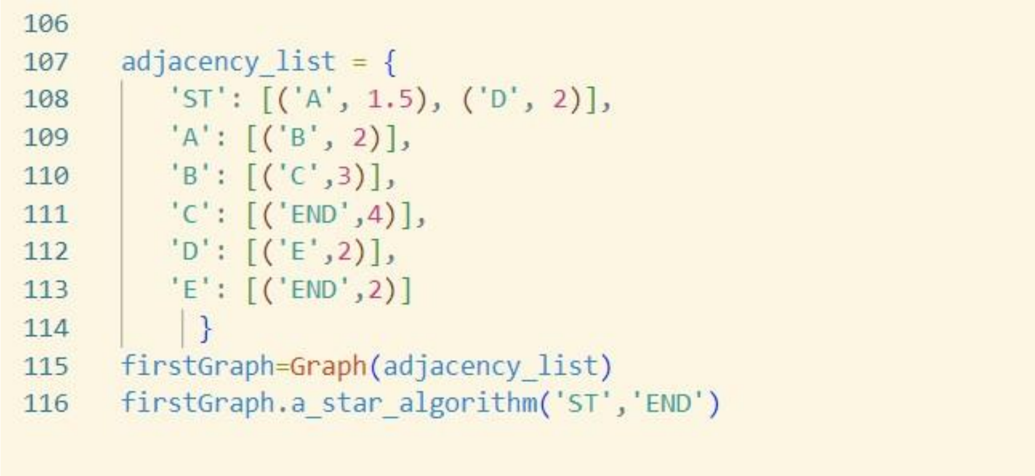
```python
from collections import deque

class Graph:
    # example of adjacency list (or rather map)
    # adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)],
    # 'B': [('D', 5)],
    # 'C': [('D', 12)]
    # }

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'ST':10,
            'A': 4,
            'B': 2,
            'C': 4,
            'D': 4.5,
            'E':2,
            'END':0
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's neighbors
        # haven't all been inspected, starts off with the start node
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])
```



```python

        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            if n == stop_node:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)
```

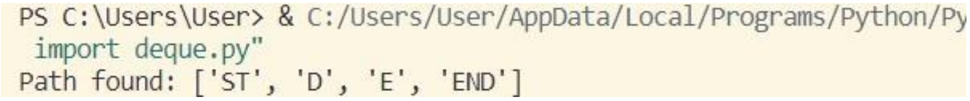


```python

                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and closed_list
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None


```



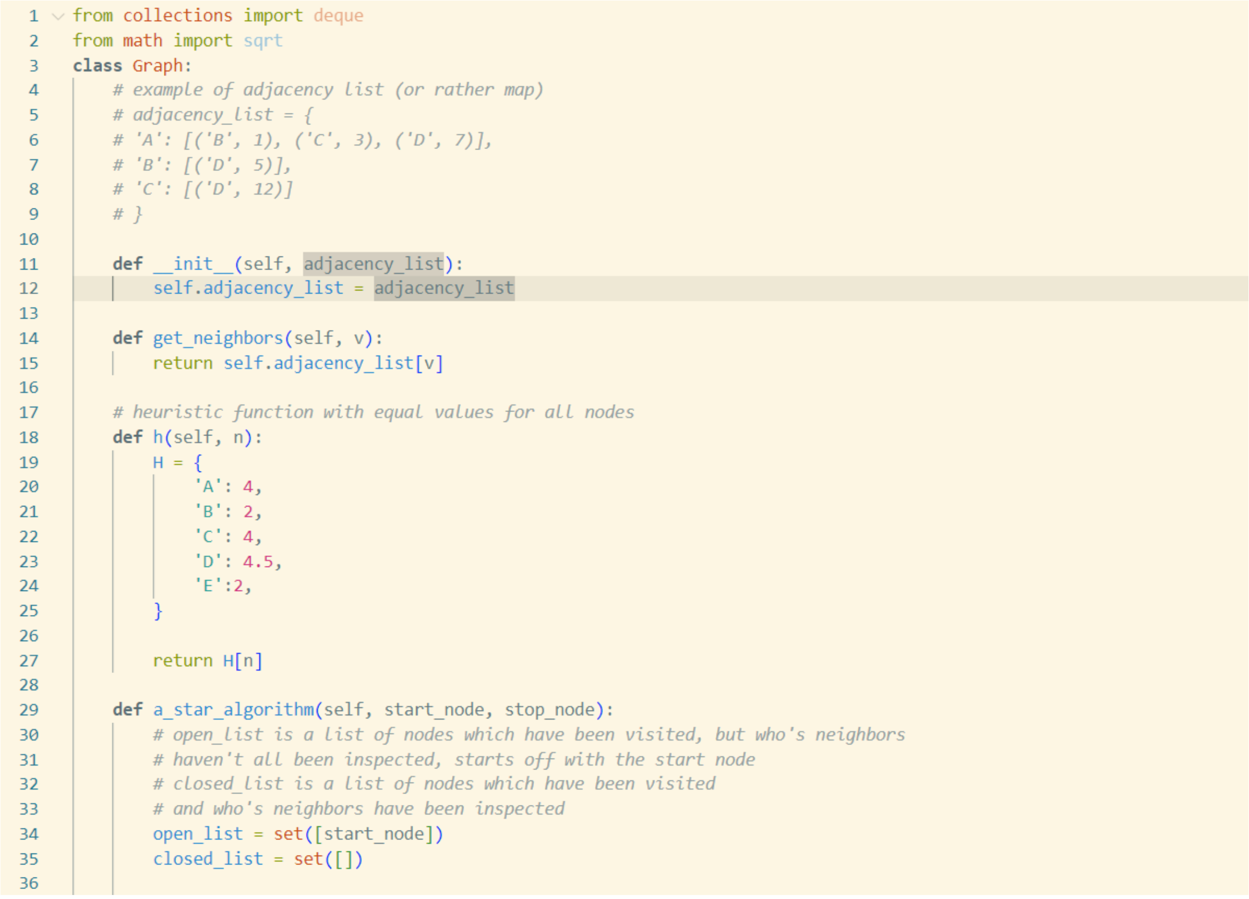
```
adjacency_list = {
    'ST': [('A', 1.5), ('D', 2)],
    'A': [('B', 2)],
    'B': [('C',3)],
    'C': [('END',4)],
    'D': [('E',2)],
    'E': [('END',2)]
     }
firstGraph=Graph(adjacency_list)
firstGraph.a_star_algorithm('ST','END')
```

Result:

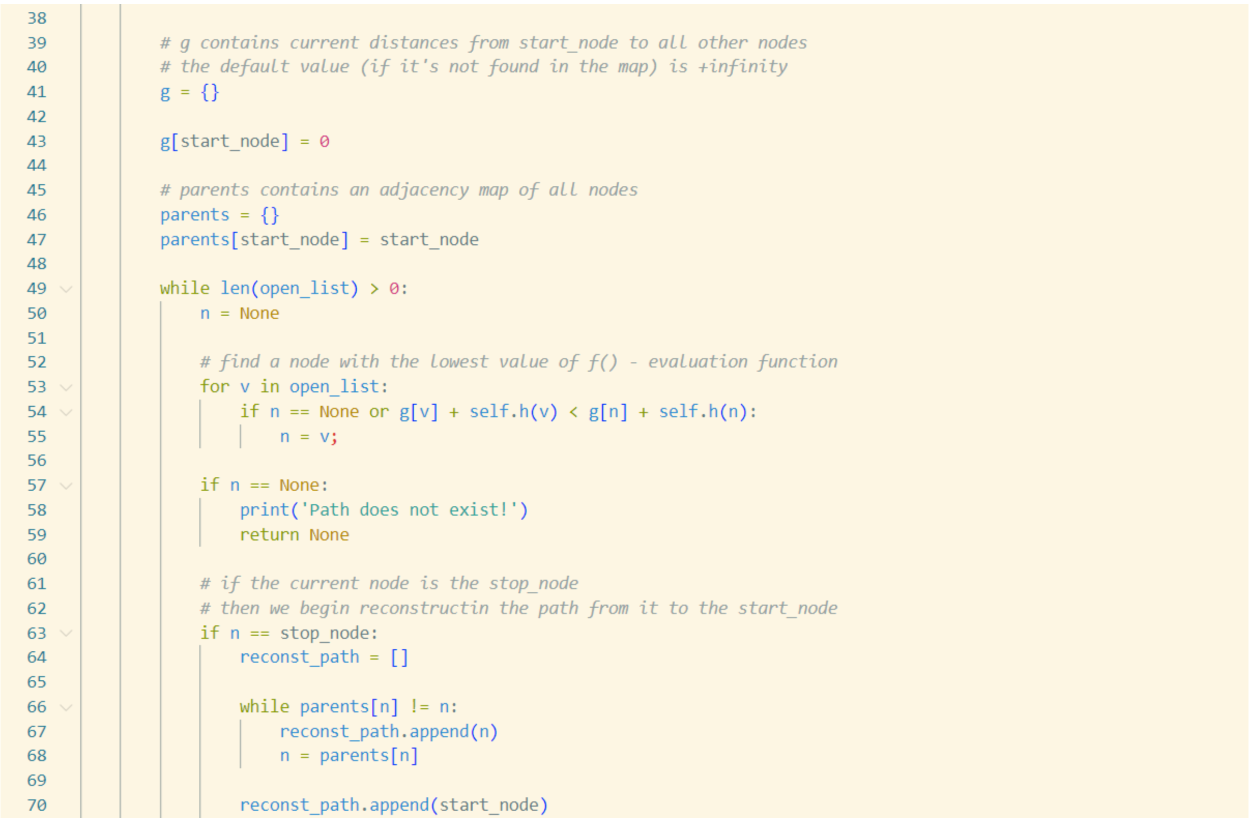


```
PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Py
 import deque.py"
Path found: ['ST', 'D', 'E', 'END']
```

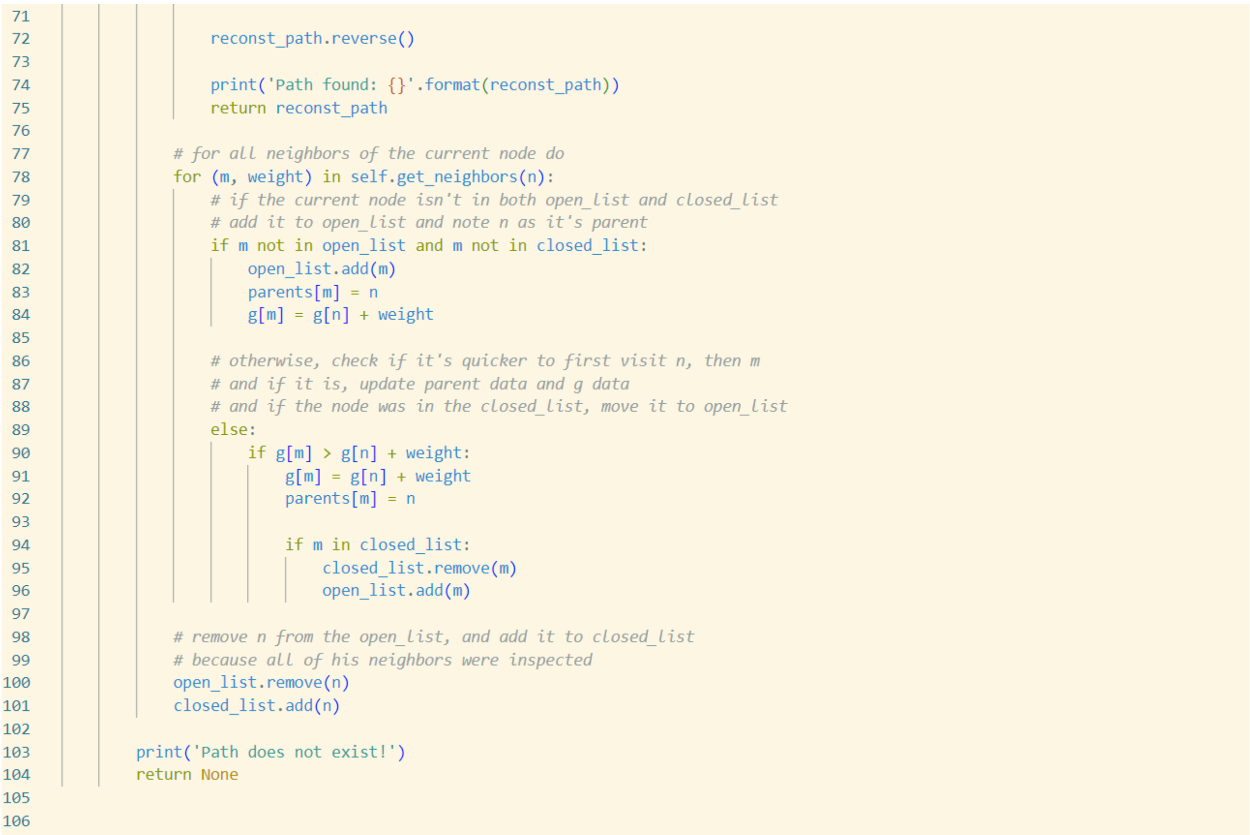
For Manhattan distance:



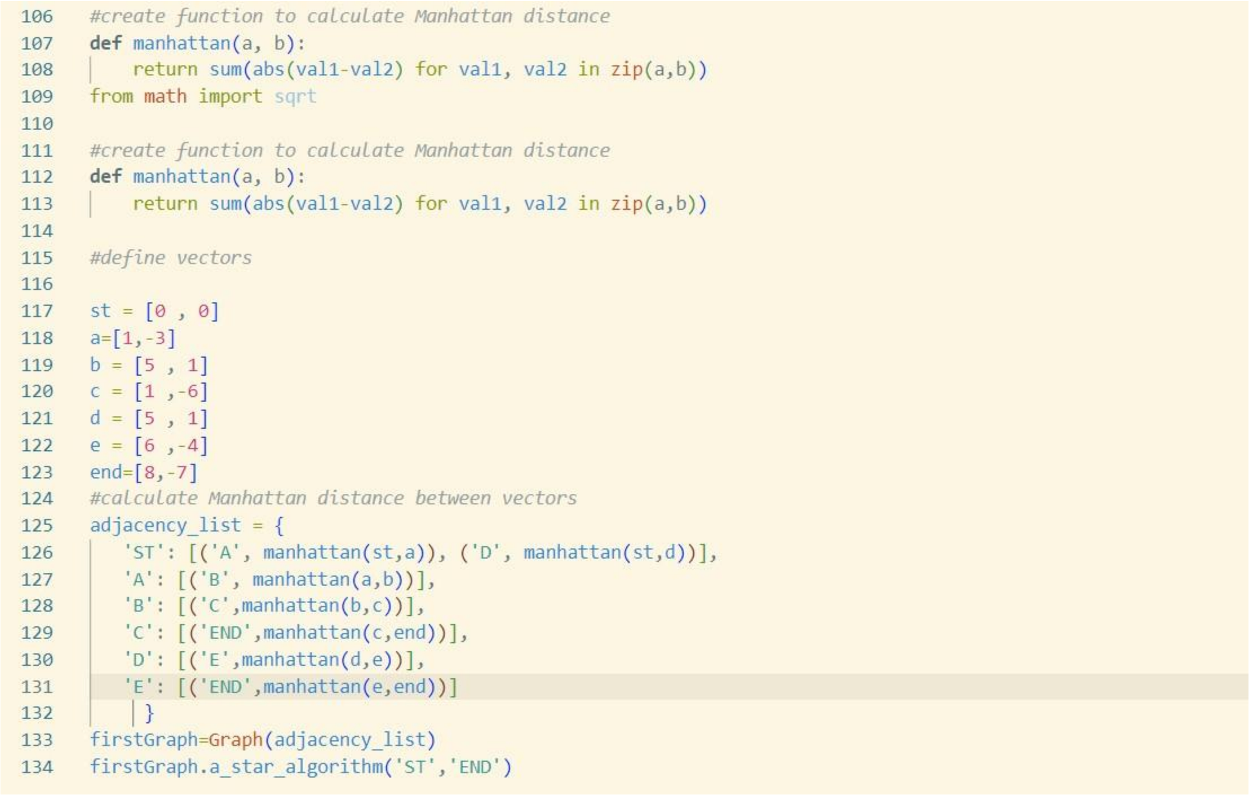
```
from collections import deque
from math import sqrt
class Graph:
    # example of adjacency list (or rather map)
    # adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)],
    # 'B': [('D', 5)],
    # 'C': [('D', 12)]
    # }

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'A': 4,
            'B': 2,
            'C': 4,
            'D': 4.5,
            'E':2,
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's neighbors
        # haven't all been inspected, starts off with the start node
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])

```



```python

        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            if n == stop_node:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)
```

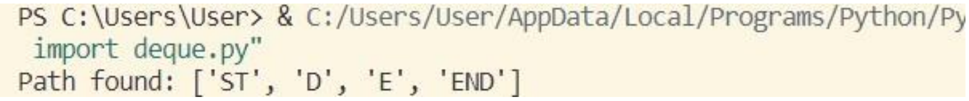


```python

                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and closed_list
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None


```



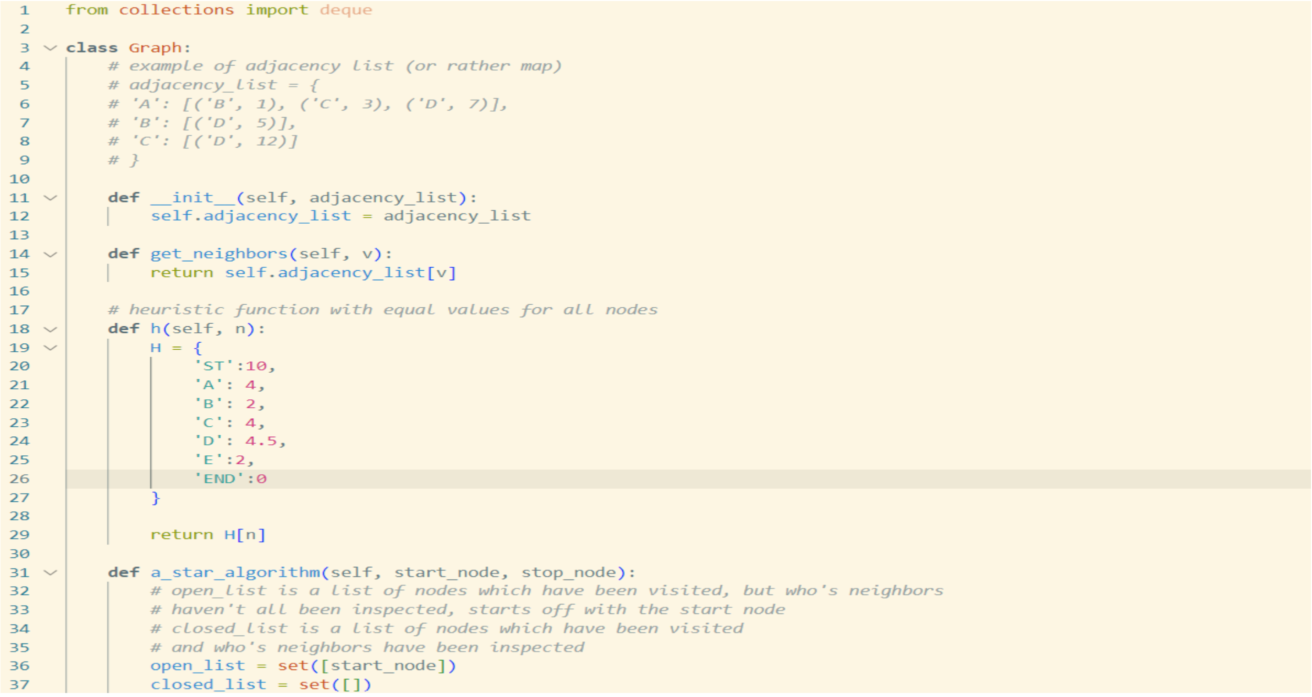
```
#create function to calculate Manhattan distance
def manhattan(a, b):
    return sum(abs(val1-val2) for val1, val2 in zip(a,b))
from math import sqrt

#create function to calculate Manhattan distance
def manhattan(a, b):
    return sum(abs(val1-val2) for val1, val2 in zip(a,b))

#define vectors

st = [0 , 0]
a=[1,-3]
b = [5 , 1]
c = [1 ,-6]
d = [5 , 1]
e = [6 ,-4]
end=[8,-7]
#calculate Manhattan distance between vectors
adjacency_list = {
    'ST': [('A', manhattan(st,a)), ('D', manhattan(st,d))],
    'A': [('B', manhattan(a,b))],
    'B': [('C',manhattan(b,c))],
    'C': [('END',manhattan(c,end))],
    'D': [('E',manhattan(d,e))],
    'E': [('END',manhattan(e,end))]
     }
firstGraph=Graph(adjacency_list)
firstGraph.a_star_algorithm('ST','END')
```

Result:

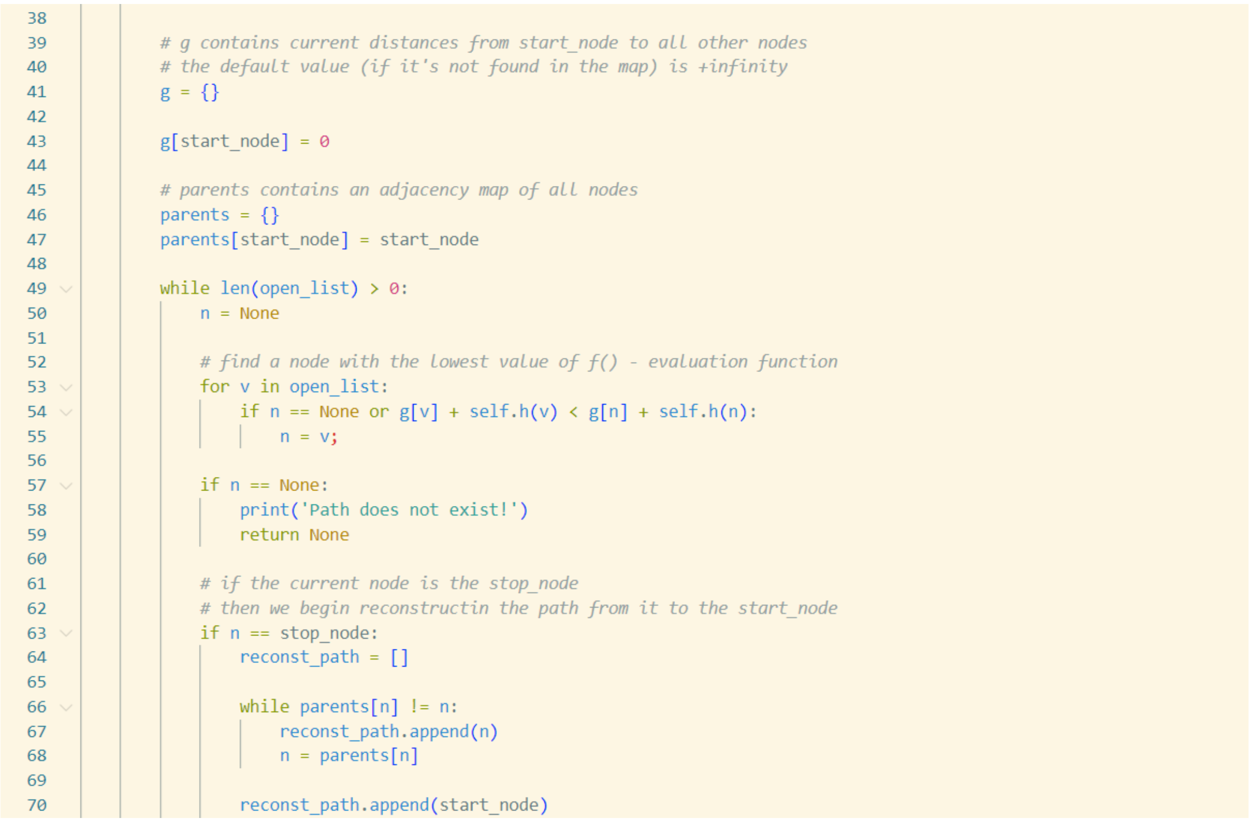


```
PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Py
 import deque.py"
Path found: ['ST', 'D', 'E', 'END']
```

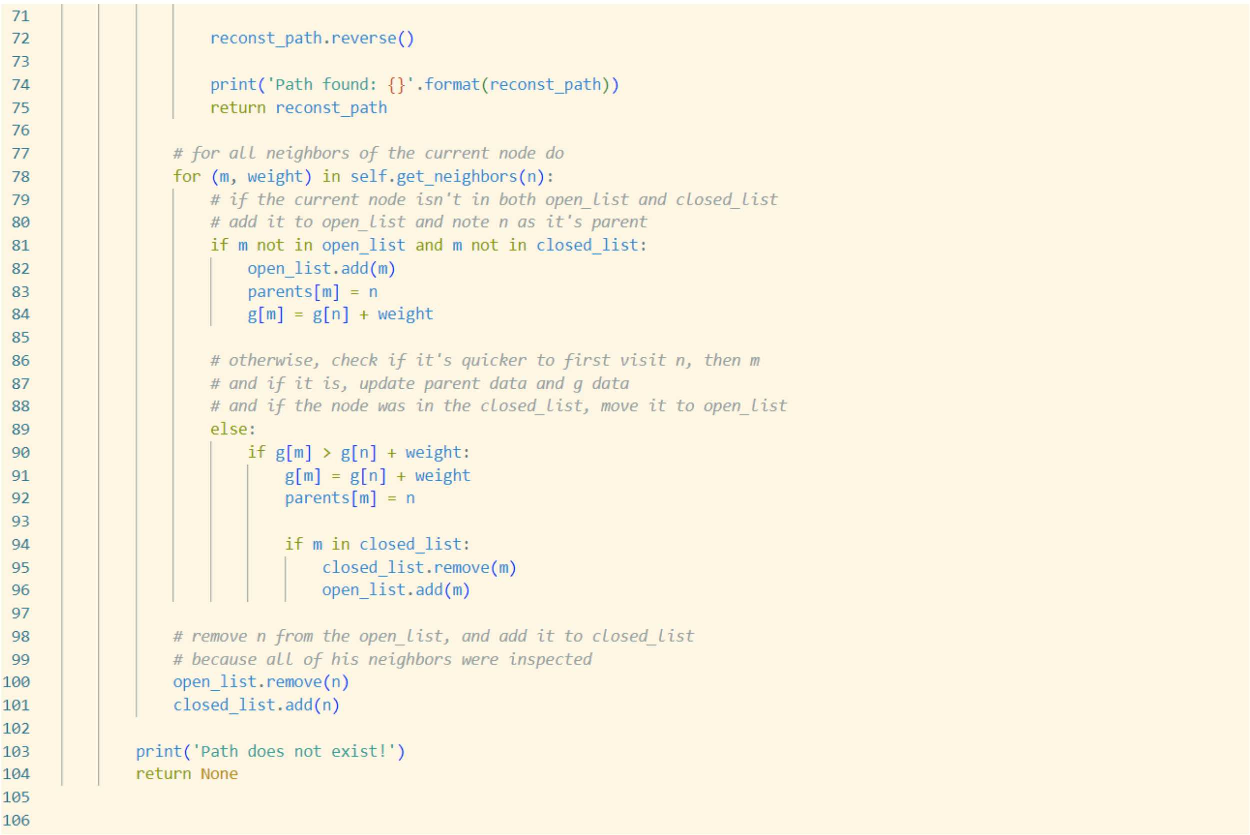
2.Graph solution with Phyton using A* Algorithm using Euclidean distance:



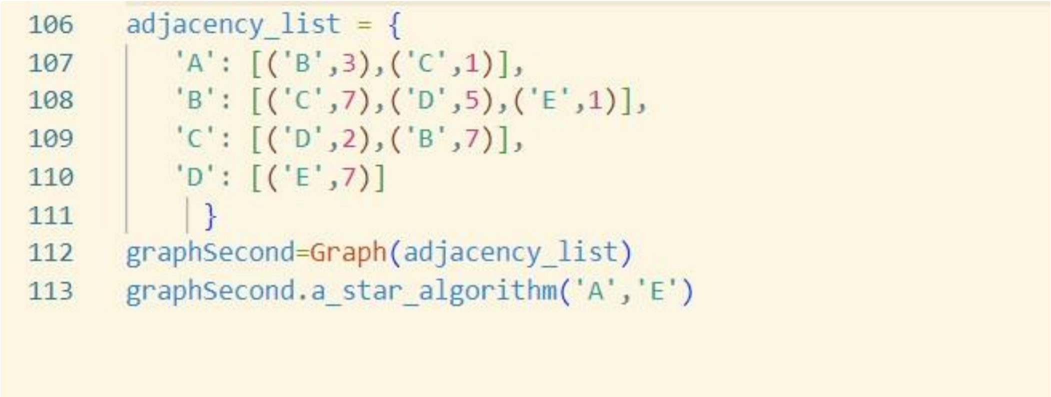
```
from collections import deque

class Graph:
    # example of adjacency list (or rather map)
    # adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)],
    # 'B': [('D', 5)],
    # 'C': [('D', 12)]
    # }

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'ST':10,
            'A': 4,
            'B': 2,
            'C': 4,
            'D': 4.5,
            'E':2,
            'END':0
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's neighbors
        # haven't all been inspected, starts off with the start node
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])
```



```python

        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            if n == stop_node:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)
```

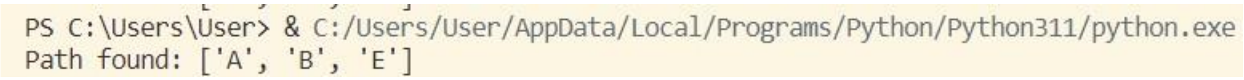


```python

                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and closed_list
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None

```



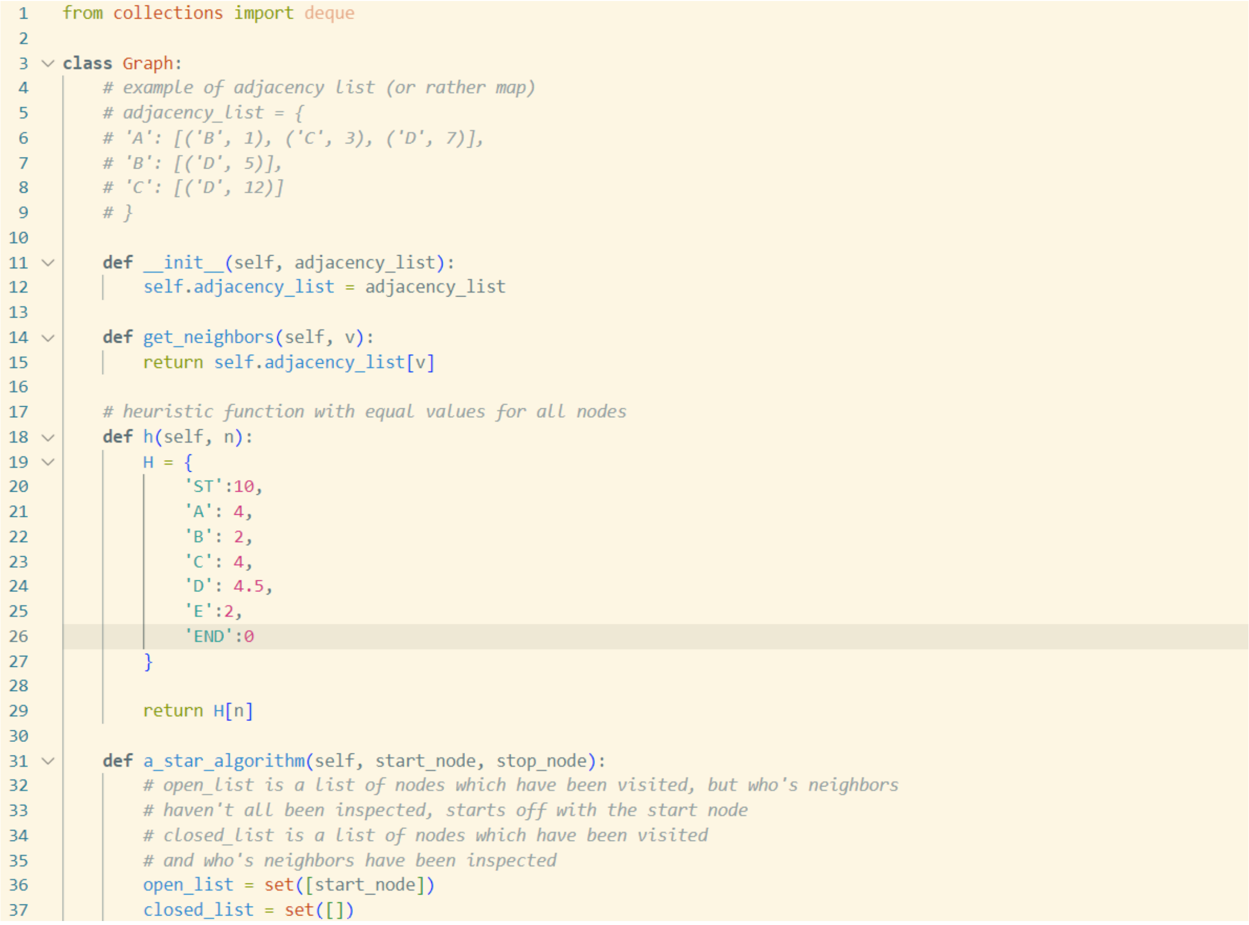
```python
adjacency_list = {
    'A': [('B',3),('C',1)],
    'B': [('C',7),('D',5),('E',1)],
    'C': [('D',2),('B',7)],
    'D': [('E',7)]
     }
graphSecond=Graph(adjacency_list)
graphSecond.a_star_algorithm('A','E')
```

Result:

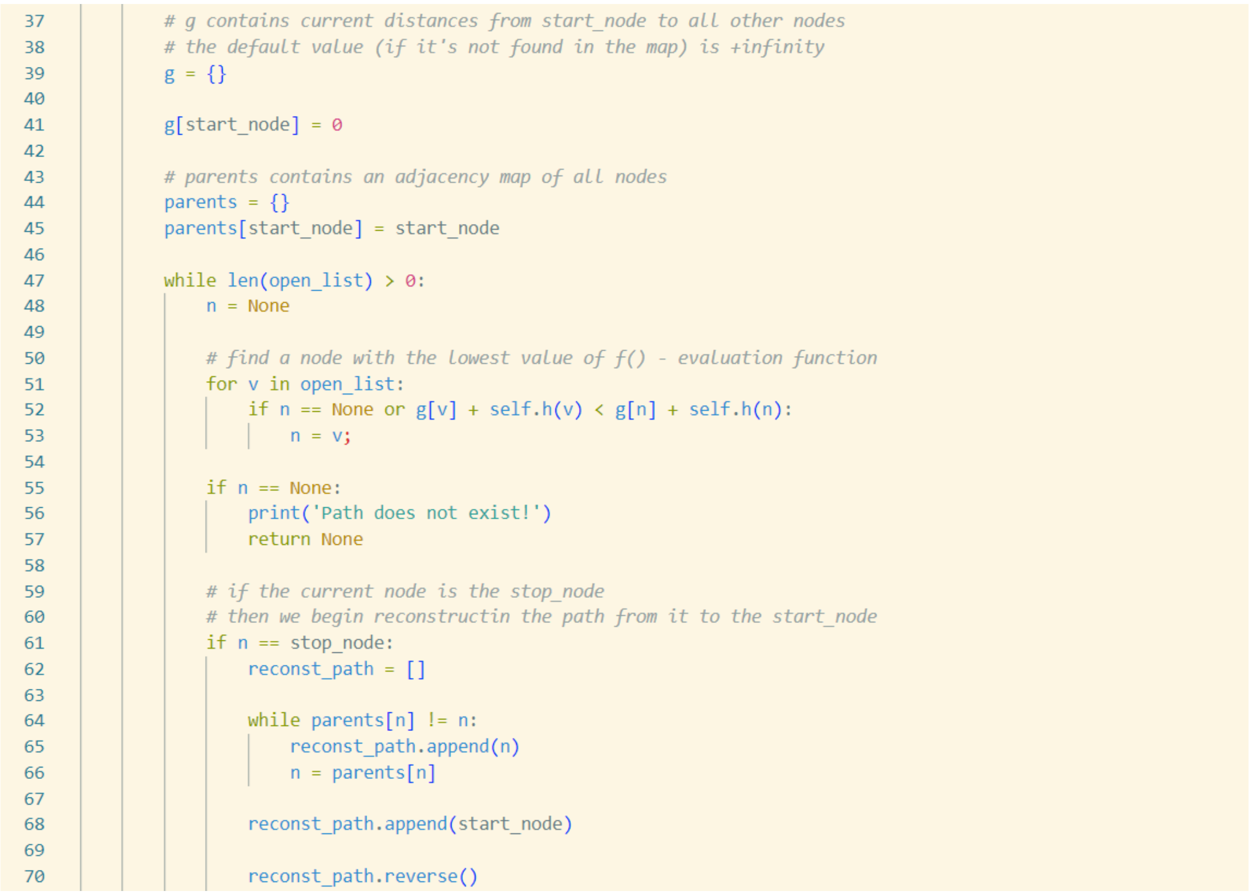


```
PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python311/python.exe
Path found: ['A', 'B', 'E']
```

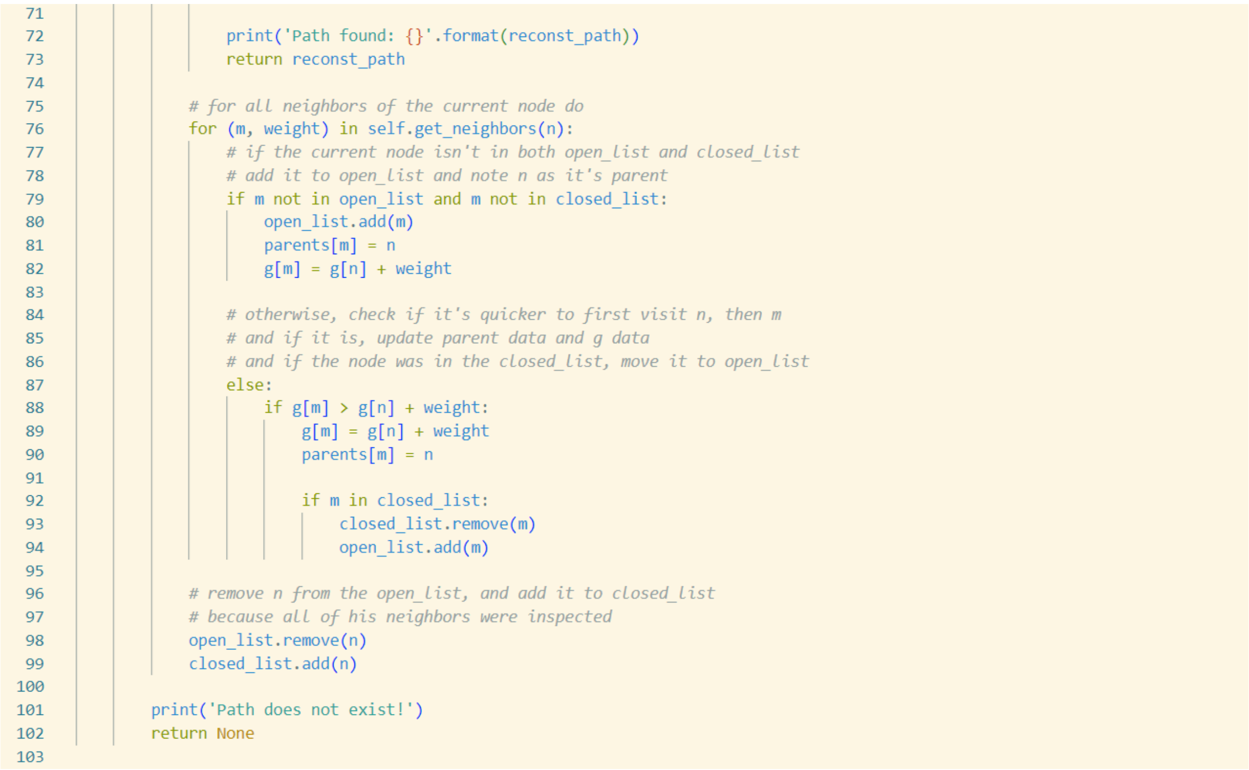
For Manhattan distance:



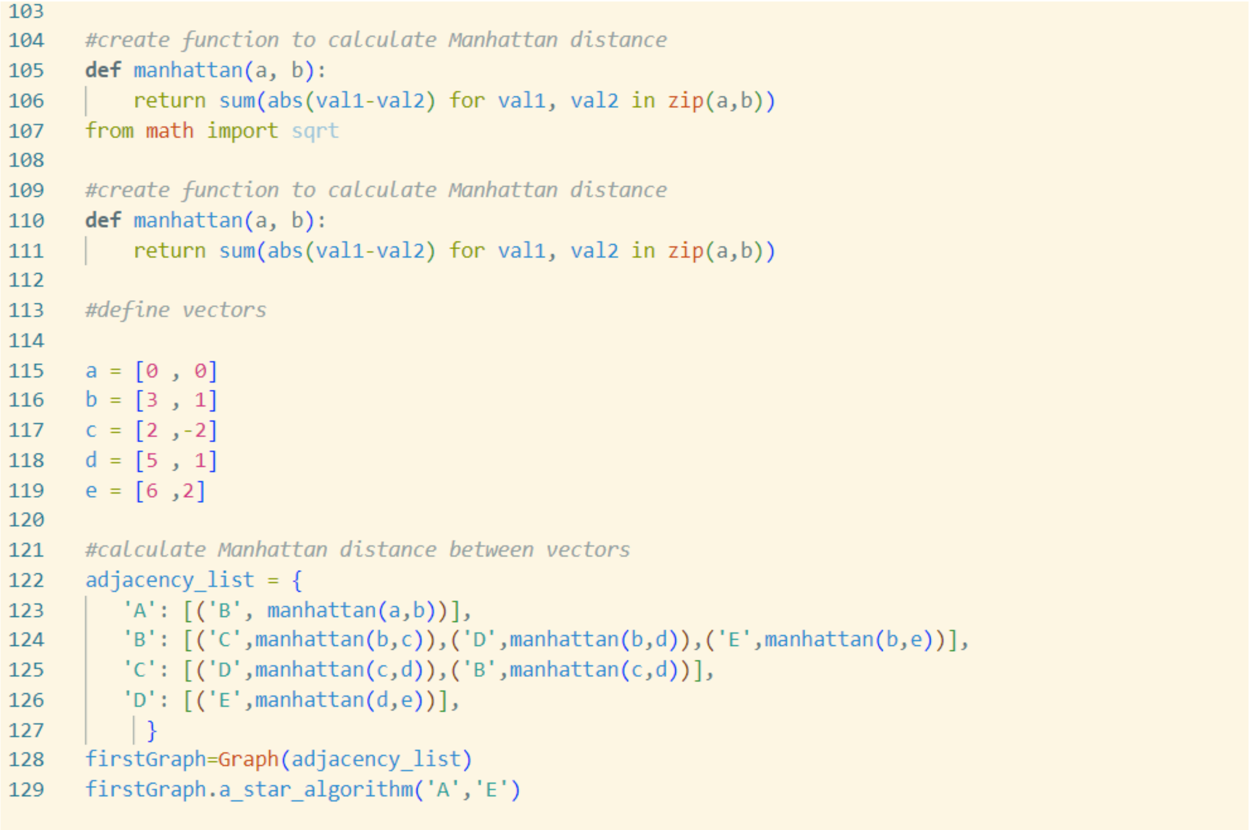
```python
from collections import deque

class Graph:
    # example of adjacency list (or rather map)
    # adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)],
    # 'B': [('D', 5)],
    # 'C': [('D', 12)]
    # }

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'ST':10,
            'A': 4,
            'B': 2,
            'C': 4,
            'D': 4.5,
            'E':2,
            'END':0
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's neighbors
        # haven't all been inspected, starts off with the start node
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])
```



```python
        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            if n == stop_node:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)

                reconst_path.reverse()
```

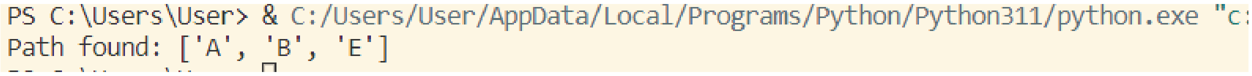


```python

                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and closed_list
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None

```



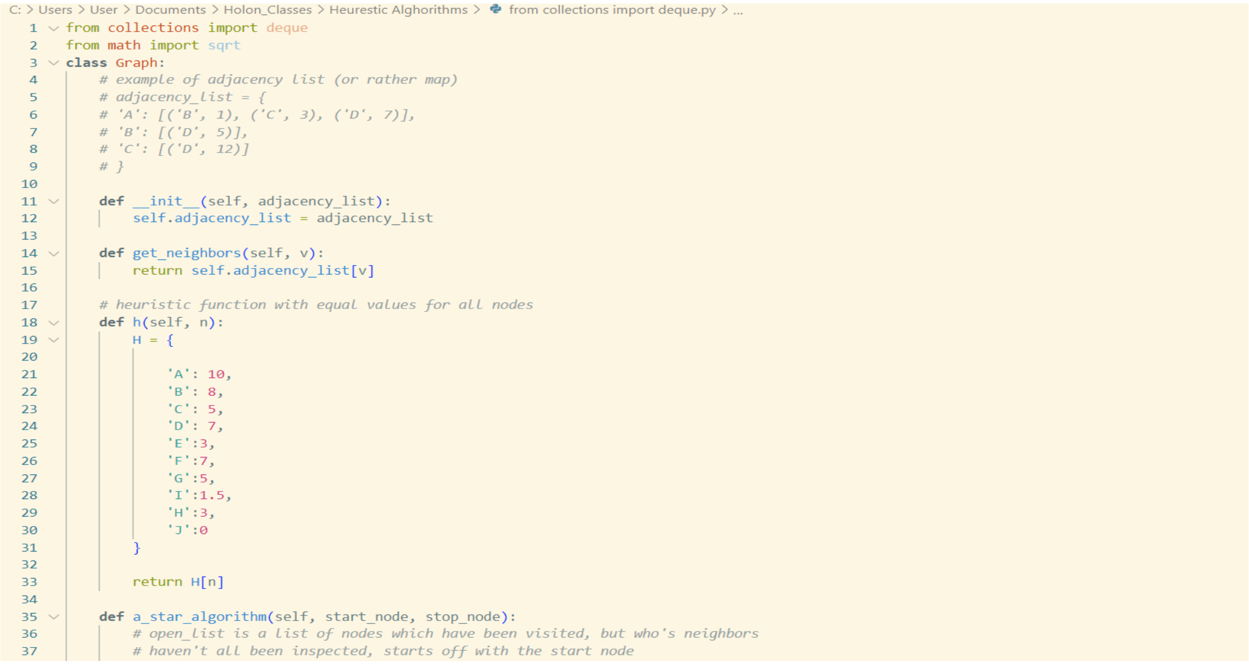
```python

#create function to calculate Manhattan distance
def manhattan(a, b):
    return sum(abs(val1-val2) for val1, val2 in zip(a,b))
from math import sqrt

#create function to calculate Manhattan distance
def manhattan(a, b):
    return sum(abs(val1-val2) for val1, val2 in zip(a,b))

#define vectors

a = [0 , 0]
b = [3 , 1]
c = [2 ,-2]
d = [5 , 1]
e = [6 ,2]

#calculate Manhattan distance between vectors
adjacency_list = {
    'A': [('B', manhattan(a,b))],
    'B': [('C',manhattan(b,c)),('D',manhattan(b,d)),('E',manhattan(b,e))],
    'C': [('D',manhattan(c,d)),('B',manhattan(c,d))],
    'D': [('E',manhattan(d,e))],
     }
firstGraph=Graph(adjacency_list)
firstGraph.a_star_algorithm('A','E')
```

Result:



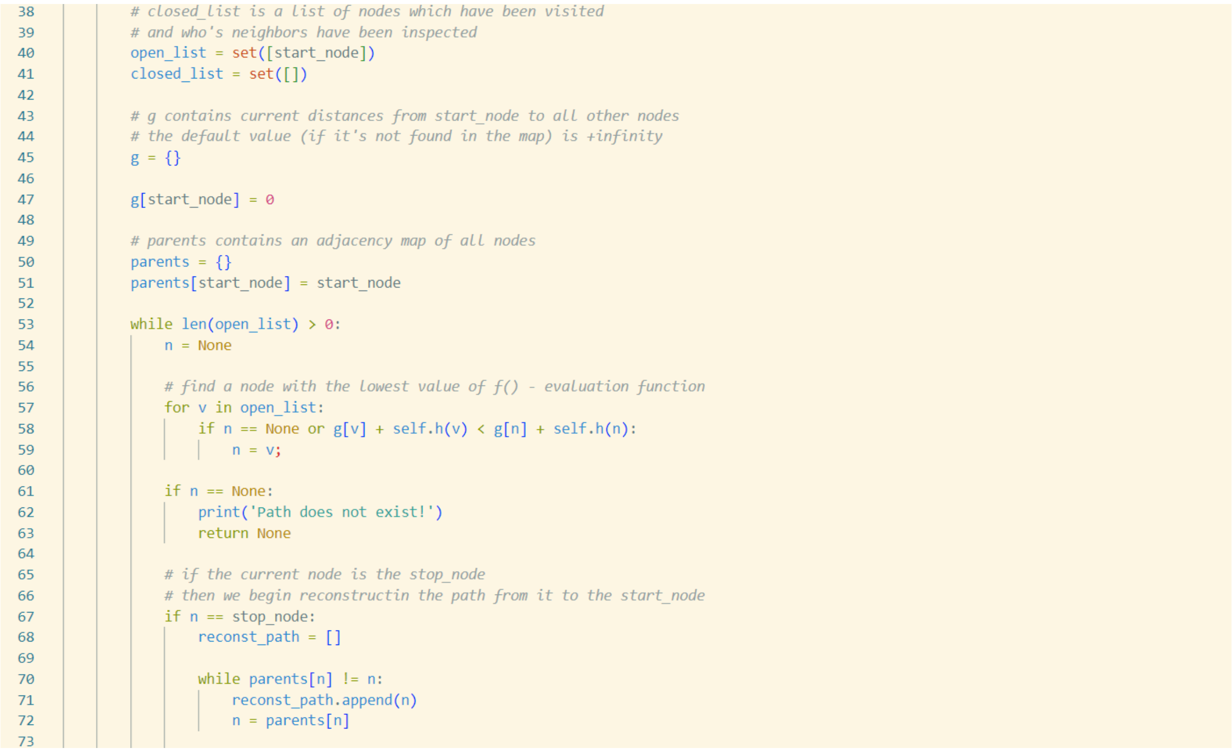
```
PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python311/python.exe "c:
Path found: ['A', 'B', 'E']
```

3.Graph solution with Phyton using A* Algorithm using Euclidean distance:

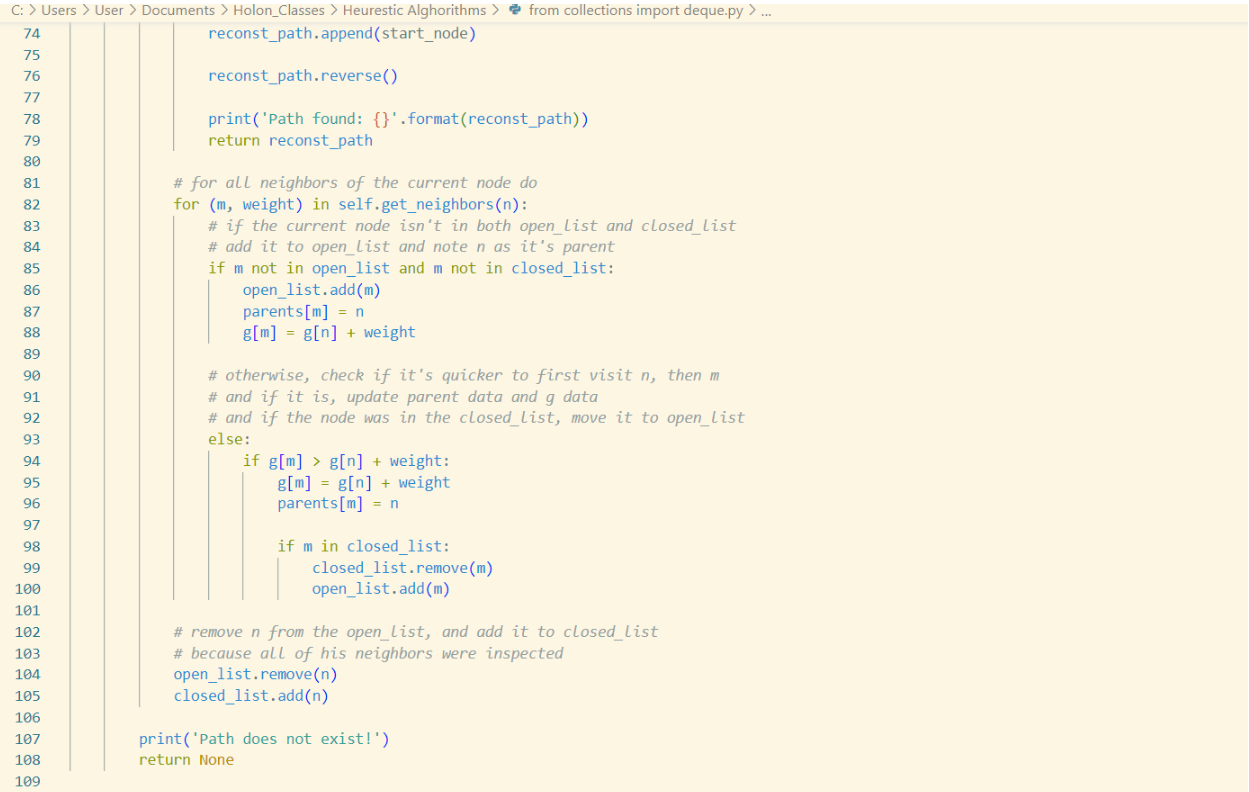


C: > Users > User > Documents > Holon_Classes > Heurestic Alghorithms > from collections import deque.py > ...

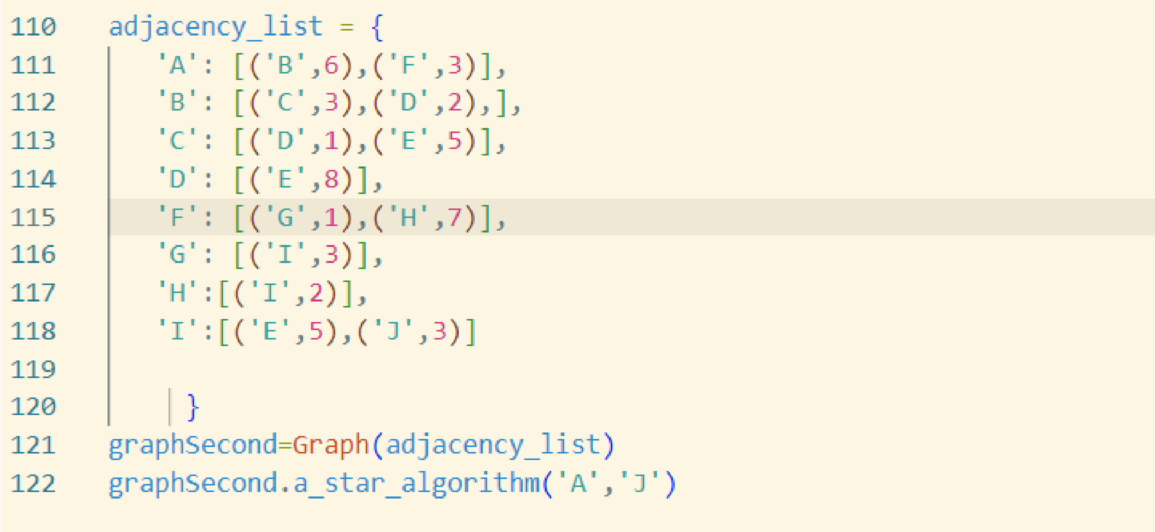
```python
from collections import deque
from math import sqrt
class Graph:
    # example of adjacency list (or rather map)
    # adjacency_list = {
    # 'A': [('B', 1), ('C', 3), ('D', 7)],
    # 'B': [('D', 5)],
    # 'C': [('D', 12)]
    # }

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {

            'A': 10,
            'B': 8,
            'C': 5,
            'D': 7,
            'E':3,
            'F':7,
            'G':5,
            'I':1.5,
            'H':3,
            'J':0
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's neighbors
        # haven't all been inspected, starts off with the start node
```



```python
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])

        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            if n == stop_node:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

```

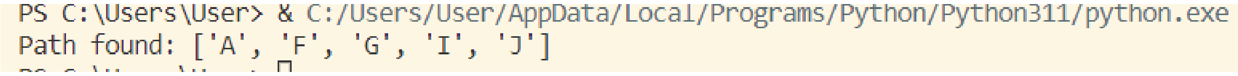

C: > Users > User > Documents > Holon_Classes > Heurestic Alghorithms > from collections import deque.py > ...

```python
                reconst_path.append(start_node)

                reconst_path.reverse()

                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and closed_list
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None

```



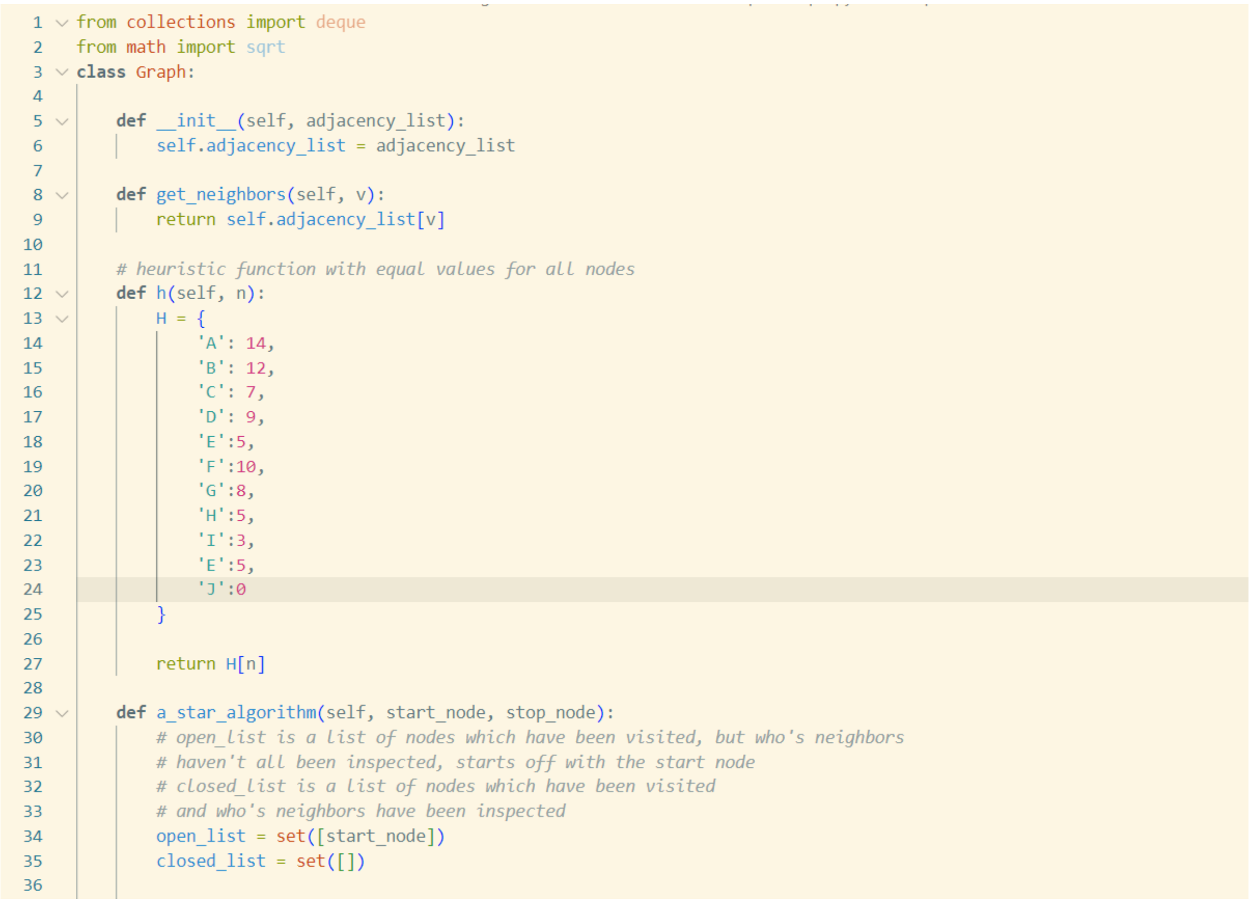
```python
adjacency_list = {
    'A': [('B',6),('F',3)],
    'B': [('C',3),('D',2),],
    'C': [('D',1),('E',5)],
    'D': [('E',8)],
    'F': [('G',1),('H',7)],
    'G': [('I',3)],
    'H':[('I',2)],
    'I':[('E',5),('J',3)]

     }
graphSecond=Graph(adjacency_list)
graphSecond.a_star_algorithm('A','J')
```

Result:

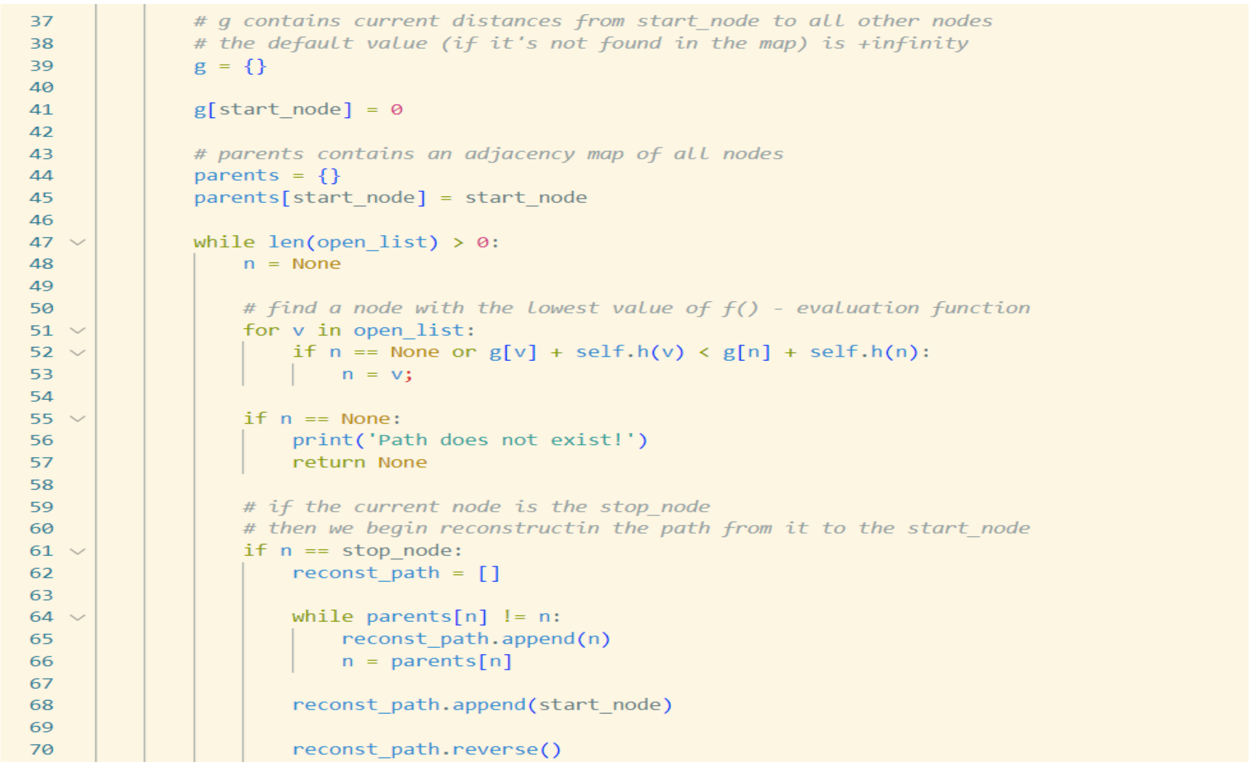


```
PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python311/python.exe
Path found: ['A', 'F', 'G', 'I', 'J']
PS C:\Users\User> 
```

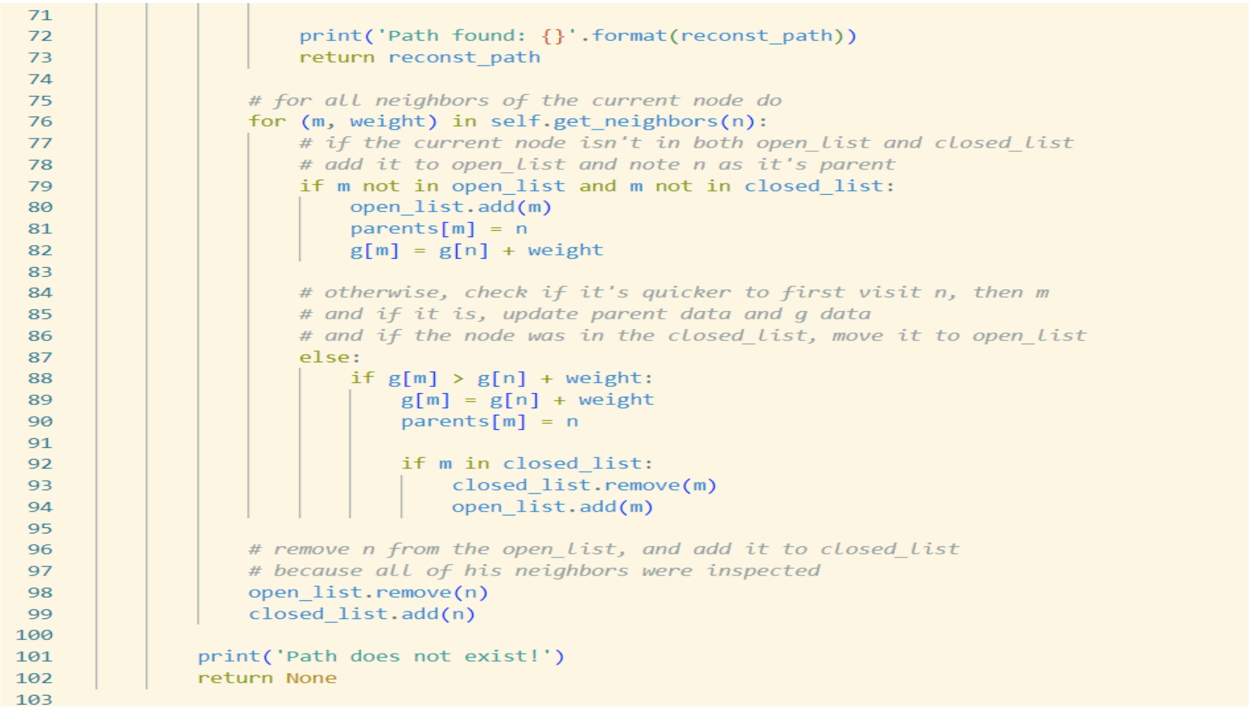
With Manhattan distance :



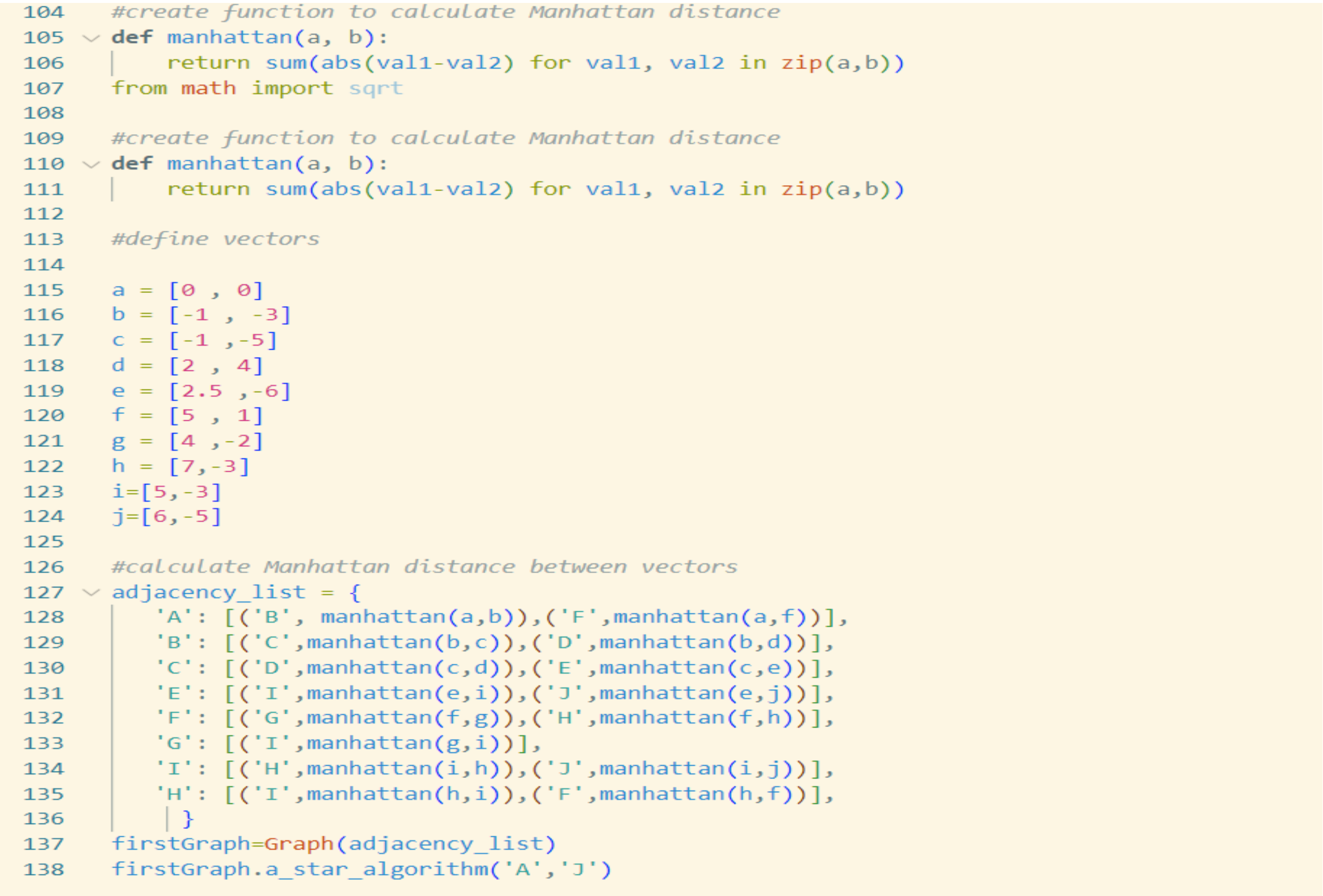
```python
from collections import deque
from math import sqrt
class Graph:

    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list

    def get_neighbors(self, v):
        return self.adjacency_list[v]

    # heuristic function with equal values for all nodes
    def h(self, n):
        H = {
            'A': 14,
            'B': 12,
            'C': 7,
            'D': 9,
            'E':5,
            'F':10,
            'G':8,
            'H':5,
            'I':3,
            'E':5,
            'J':0
        }

        return H[n]

    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but who's neighbors
        # haven't all been inspected, starts off with the start node
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected
        open_list = set([start_node])
        closed_list = set([])

```



```
        # g contains current distances from start_node to all other nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
        parents[start_node] = start_node

        while len(open_list) > 0:
            n = None

            # find a node with the lowest value of f() - evaluation function
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the start_node
            if n == stop_node:
                reconst_path = []

                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]

                reconst_path.append(start_node)

                reconst_path.reverse()
```

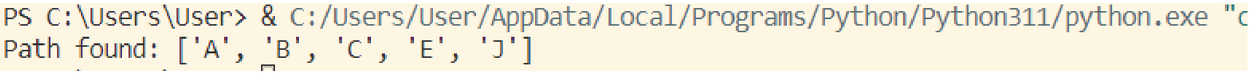


```

                print('Path found: {}'.format(reconst_path))
                return reconst_path

            # for all neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
                # if the current node isn't in both open_list and closed_list
                # add it to open_list and note n as it's parent
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight

                # otherwise, check if it's quicker to first visit n, then m
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None

```



```
#create function to calculate Manhattan distance
def manhattan(a, b):
    return sum(abs(val1-val2) for val1, val2 in zip(a,b))
from math import sqrt

#create function to calculate Manhattan distance
def manhattan(a, b):
    return sum(abs(val1-val2) for val1, val2 in zip(a,b))

#define vectors

a = [0 , 0]
b = [-1 , -3]
c = [-1 ,-5]
d = [2 , 4]
e = [2.5 ,-6]
f = [5 , 1]
g = [4 ,-2]
h = [7,-3]
i=[5,-3]
j=[6,-5]

#calculate Manhattan distance between vectors
adjacency_list = {
    'A': [('B', manhattan(a,b)),('F',manhattan(a,f))],
    'B': [('C',manhattan(b,c)),('D',manhattan(b,d))],
    'C': [('D',manhattan(c,d)),('E',manhattan(c,e))],
    'E': [('I',manhattan(e,i)),('J',manhattan(e,j))],
    'F': [('G',manhattan(f,g)),('H',manhattan(f,h))],
    'G': [('I',manhattan(g,i))],
    'I': [('H',manhattan(i,h)),('J',manhattan(i,j))],
    'H': [('I',manhattan(h,i)),('F',manhattan(h,f))],
    }
firstGraph=Graph(adjacency_list)
firstGraph.a_star_algorithm('A','J')
```

Result :



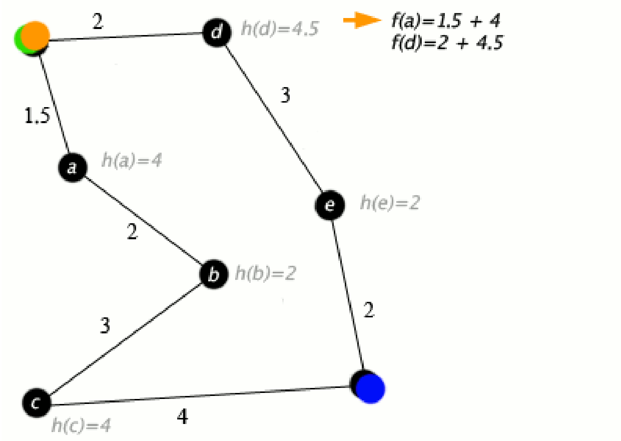
```
PS C:\Users\User> & C:/Users/User/AppData/Local/Programs/Python/Python311/python.exe "c
Path found: ['A', 'B', 'C', 'E', 'J']
```

6)Preparation to Test of Half-Semester on 16.12.2022

1. What are the dangers of the AI Algorithms in our modern life?

In the present world, there are a number of possible risks associated with AI algorithms. One risk is that AI systems could decide in ways that are bad for people or society. For instance, a loan decision-making AI program may unintentionally bias against particular groups of people. Another risk is that AI algorithms could evolve to the point where they outperform human intelligence and endanger humanity. The usage of AI algorithms may also result in job losses because they may be able to complete some tasks faster than people can. Overall, it's critical to carefully weigh the hazards that could result from utilizing AI algorithms and to take action to reduce those risks.

2. Given a graph in which a shortest path is to be found. What are the first three steps of the Algorithm Dijkstra.



1) Initialize the distances of all vertices in the graph to infinity, except for the starting vertex, which has a distance of 0.

2) Choose the vertex with the smallest distance from the starting vertex as the current vertex. In our case it is a.

3) For each neighbor of the current vertex, calculate the distance from the starting vertex to the neighbor by adding the weight of the edge connecting the current vertex and the neighbor. If this distance is smaller than the current distance of the neighbor, update the distance of the neighbor to the new distance. Repeat this step for all neighbors of the current vertex

3. What is the main advantage of the A* in comparison with the Dijkstra Algorithm?

The A* algorithm's key benefit over the Dijkstra method is that it is quicker and more effective. This is so that it can explore more plausible paths and avoid paths that are unlikely to result in a solution. The A* algorithm uses a heuristic function to guide the search. By drastically reducing the number of nodes that must be searched, this can shorten the running time. The Dijkstra method, in contrast, analyzes every potential path without the use of a heuristic approach, which might result in slower processing times, particularly for big or complex tasks.

4.What is the main advantage of the PRM algorithm in comparison with the Dijkstra Algorithm?

The key benefit of the Probabilistic Roadmap (PRM) algorithm over the Dijkstra algorithm is that it is quicker and more effective at locating a path across a large amount of data. This is so that the PRM algorithm can swiftly determine a path between two points by creating a roadmap of the open area in the environment.

As a result, the search space and the quantity of nodes that must be investigated can be greatly reduced, which can shorten the running time. In contrast, the Dijkstra algorithm, which might be ineffective for high-dimensional spaces, examines the whole search space without employing a roadmap. Additionally, the PRM algorithm delivers a solution with a specific likelihood of success, whereas the Dijkstra algorithm provides the optimal solution with certainty.

5.Give three examples of the problems in Computer science, for which the greedy algorithm quickly gives the exact solution

1) **Dijkstra's algorithm** is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks.

2) The Minimum Spanning Tree Problem: The objective of this issue is to identify a subset of the edges that forms a tree that contains all the vertices and has the least overall weight. You are given a connected, undirected graph with weighted edges.

3) The Huffman Coding Problem: You are given a set of symbols and their related frequencies in this task, and your objective is to create a prefix code that encrypts the symbols with the shortest average length possible.

6. Give an example of the problem in Computer science, for which the greedy algorithm quickly gives arbitrarily bad solution.

The **travelling salesman problem:** (also called the **travelling salesperson problem** or **TSP**) asks the following question: "Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

7. What is the Digital Medicine?

In the subject of "digital medicine," diseases are better identified, treated, and prevented using digital technologies like smartphones, wearable electronics, and sensors. Using digital technologies to gather data about a person's health, such as their heart rate or blood pressure, and using that information to make better educated decisions about their treatment, are two examples of how this might be done. The use of AI algorithms to analyze vast volumes of health data and find patterns that can aid in the early detection and treatment of diseases is another aspect of digital medicine. Digital medicine's overarching objective is to employ technology to raise the quality, effectiveness, and accessibility of healthcare.

HWT-7. Knapsack. GREEDY Perf. Guarantee 2. DP

Each item has a value (bi) and a weight (wi)

• Objective: maximize value

• Constraint: knapsack has a weight limitation (W)

Given: Knapsack has weight limit W A set S of n items, items labeled 1, 2, …, n (arbitrarily), each item i has - bi - a

positive benefit value - wi - a positive weight (assume all weights are integers)

Goal: Choose items with maximum total benefit but with weight at most W.

Let T denote the set of items we take –

Objective: maximize – i∈ T ∑ bi

Constraint: i∈T ∑ wi ≤W

Problem, in other words, is to find

max i=n ∑ xibi subject to i ∑ xiwi ≤ W; where $xi\epsilon$ {0,1}

- bi - a benefit value

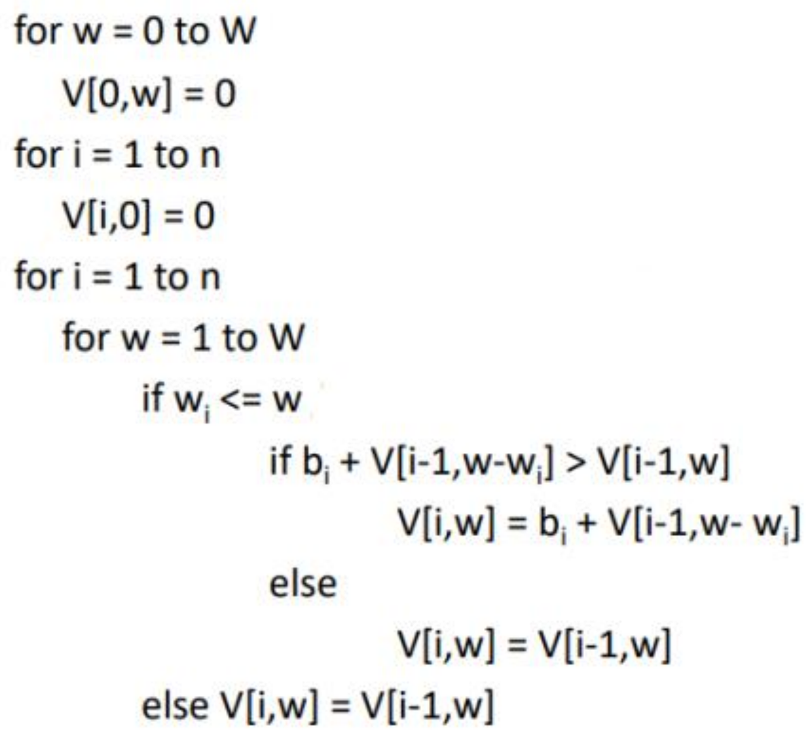
- wi - a weight

Recursive Formula:

$$V[k,w] = \begin{cases} V[k-1,w] & \text{if } w_k > w \\ \max\{V[k-1,w], V[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of Sk that has the total weight ⍰ w, either contains item k or not.
- First case: wk>w. Item k can’t be part of the solution, since if it was, the total weight would be > w, which is unacceptable.
- Second case: wk ⍰ w. Then the item k can be in the solution, and we choose the case with greater value

The Algorithm of 0/1 Knapsack - Calculation of complexity



```
for w = 0 to W
    V[0,w] = 0
for i = 1 to n
    V[i,0] = 0
for i = 1 to n
    for w = 1 to W
        if $w_i$ <= w
            if $b_i$ + V[i-1,w-$w_i$] > V[i-1,w]
                V[i,w] = $b_i$ + V[i-1,w- $w_i$]
            else
                V[i,w] = V[i-1,w]
        else V[i,w] = V[i-1,w]
```

Here is an instance of the knapsack challenge where the limit is 70 and a collection of items is present:

Greedy Algorithm Solution:

Items:

1. Food (weight: 20, value: 100)
2. Water bottle (weight: 30, value: 120)
3. Sleeping bag (weight: 40, value: 200)
4. Tent (weight: 50, value: 300)
5. Book (weight: 10, value: 60)

A greedy algorithm solution to this problem involves sorting the items based on their value-to-weight ratio in descending order, like so:

1. Tent (weight: 50, value: 300, ratio: 6)
2. Sleeping bag (weight: 40, value: 200, ratio: 5)
3. Water bottle (weight: 30, value: 120, ratio: 4)
4. Food (weight: 20, value: 100, ratio: 5)
5. Book (weight: 10, value: 60, ratio: 6)

The greedy algorithm solution proceeds as follows:

Take the most valuable item (the tent) which has a weight of 50 and a value of 300. Since its weight (50) is less than the capacity of the knapsack (70), it is added to the knapsack and the weight is removed from the remaining capacity, resulting in a remaining capacity of 20.

The next item to be considered is the sleeping bag with a weight of 40 and a value of 200. Since its weight is less than the remaining capacity (20), it is added to the knapsack and its weight is also removed from the remaining capacity, leaving a remaining capacity of 0.

The knapsack is now full and contains the tent and sleeping bag with a total weight of 90 and a total value of 500.

It is important to note that this solution may not be the optimal one as the greedy algorithm only takes into consideration the most valuable item at each step, not the overall impact of its choices. For instance, choosing the book and food instead of the tent and sleeping bag would result in a lower weight but the same total value.

The steps to solve the knapsack problem using dynamic programming are as follows:

1. Define a two-dimensional array "M" with rows representing the items and columns representing the weights.
2. Initialize the array with base cases: M[0][j]=0 for all j and M[i][0]=0 for all i.
3. Fill in the values of the array using the formula: M[i][j] = max(M[i-1][j], M[i-1][j-w[i]] + v[i]), where w[i] is the weight of the i-th item and v[i] is its value.
4. The maximum value that can be achieved with a knapsack of capacity j is M[n][j], where n is the number of items.

Solution for Advanced Greedy Algorithms:

The steps listed below can be used to resolve the knapsack problem with a performance guarantee of -approximation, where is the desired approximation factor:

1. Using the elements' value-to-weight ratios (value/weight), arrange them in descending order. This will guarantee that the most priceless objects are taken into account initially.

2. Set the variables "total value" and "total weight" to 0 as initial values.

3. Repeat the steps for each item as you go down the list. Add the item to the knapsack and update the total value and total weight as necessary if the item's weight combined with the current total weight is less than or equal to the knapsack's capacity (c). a. If the combined weight of the object and the whole situation

Objects: 1 Book (weight: 10, value: 60)

2. Food (weight: 20, value: 100) (weight: 20, value: 100)

3. Bottle of water (weight: 30, value: 120)

4. A bag to sleep in (weight: 40, value: 200)

5. Tent (weight: 50, value: 300) (weight: 50, value: 300)

Based on the elements' value to weight ratio, arrange them in descending order:

Items:

1. Tent (weight: 50, value: 300, ratio: 6) (weight: 50, value: 300, ratio: 6)

a slumbering sack (weight: 40, value: 200, ratio: 5)

3. Bottle of water (weight: 30, value: 120, ratio: 4)

4. Food (weight: 20, value: 100, ratio: 5) (weight: 20, value: 100, ratio: 5)

5. Book (weight: 10, value: 60, ratio: 6) (weight: 10, value: 60, ratio: 6)

6. Set the total value and total weight to 0 in the beginning.

7. Go through each item again and take the following action for each one: a. If the combined weight of the item and the current total is less than or equal to the capacity of the knapsack (70), add the item to the knapsack and update

the total value and total weight accordingly. b. If the item's weight plus the current total weight is greater

than the capacity of the knapsack (70), compute the fraction of the item's value that can be added to the

total value, based on the remaining capacity and the item's weight: value fraction = 2 * (70 - total weight) /

weight * value c. Add the value fraction to the total value and set the total weight to the capacity of the

knapsack (70).

8. Return the total value as the solution to the knapsack problem.

אור אטיאס 207953308 פתרון ש"ב מס' 7 + 8 בקורס בינה מלאכותית

Advanced Greedy Algorithm Solution:

Items:

1. Tent (weight: 50, value: 300)

2. Sleeping bag (weight: 40, value: 200)

3. Water bottle (weight: 30, value: 120)

4. Food (weight: 20, value: 100)

5. Book (weight: 10, value: 60)

Initialize total_value to 0 and remaining_capacity to 70.

Iterate through the sorted list of items and do the following for each item:

• For the tent (weight: 50, value: 300): Since the tent fits in the remaining capacity (70), add its value (300) to total_value and remove its weight (50) from remaining_capacity, setting total_value to 300 and remaining_capacity to 20.

• For the sleeping bag (weight: 40, value: 200): Since the sleeping bag fits in the remaining capacity (20), add its value (200) to total_value and remove its weight (40) from remaining_capacity, setting total_value to 500 and remaining_capacity to 0.

• For the water bottle (weight: 30, value: 120): Since the water bottle does not fit in the remaining capacity (0), but its value (120) is at least 2 times the remaining capacity (0), add 2 times the remaining capacity (0) to total_value and set the remaining_capacity to 0, setting total_value to 500 and remaining_capacity to 0.

• For the food (weight: 20, value: 100): Since the food does not fit in the remaining capacity (0) and its value (100) is less than 2 times the remaining capacity (0), skip it and move on to the next item.

• For the book (weight: 10, value: 60): Since the book does not fit in the remaining capacity (0) and its value (60) is less than 2 times the remaining capacity (0), skip it and move on to the next item.

At this point, we have iterated through all the items and our solution is to include the tent and the sleeping bag, with a total value of 500 and a total weight of 90.

Note that this solution is a 2-approximation of the optimal solution, as it guarantees that the value of the items included in the knapsack is at least half of the optimal value. However, it may not be the optimal solution itself, as it only considers the most valuable items and does not consider the overall weight of the items.