```
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

class Graph:

    def __init__(self):

        self.graph = {}
        self.weight = {}
        self.heuristic = {}

    def addEdge(self, o, d, w = 1):

        if o not in self.graph:
            self.graph[o] = []
            self.weight[o] = []
            self.heuristic[o] = 100
        if d not in self.graph:
            self.graph[d] = []
            self.weight[d] = []
            self.heuristic[d] = 100
        self.graph[o].append(d)
        self.weight[o].append(w)
        combined = sorted(zip(self.graph[o], self.weight[o]), key=lambda x:
x[0])
        self.graph[o], self.weight[o] = map(list, zip(*combined))
        self.graph[d].append(o)
        self.weight[d].append(w)
        combined = sorted(zip(self.graph[d], self.weight[d]), key=lambda x:
x[0])
        self.graph[d], self.weight[d] = map(list, zip(*combined))

    def addHeuristics(self, o, h):

        self.heuristic[o] = h

    def __str__(self):

        return f"{self.graph}\n{self.weight}\n{self.heuristic}"

class Algorithm:



    def DFS(self, g, o, d):

        visited = set()
        stack = [(o, [o])]  # Use a stack to store both the node and its path
        total_path = []
        while stack:
            node, path = stack.pop()
            total_path.append(path)
            if node == d:
                print(path)
                return total_path
            if node not in visited:
                visited.add(node)
                for neighbor in sorted(g.graph[node], reverse=True):
                    if neighbor not in visited:
                        stack.append((neighbor, path + [neighbor]))
        return None

    def BFS(self, g, o, d):

        visited = set()
        queue = [(o, [o])]  # Use a queue to store both the node and its
path
        total_path = []
        while queue:
            node, path = queue.pop(0)
            total_path.append(path)
            if node == d:
                print(path)
                return total_path
            if node not in visited:
                visited.add(node)
                for neighbor in g.graph[node]:
                    if neighbor not in visited:
                        queue.append((neighbor, path + [neighbor]))
        return None

    def BMS(self, g, o, d):

        paths = []
        stack = [(o, [o])]
        while stack:
            node, path = stack.pop()
            paths.append(path)
            for neighbor in g.graph[node]:
                if neighbor not in path:
                    stack.append((neighbor, path + [neighbor]))
        return paths

    def HC(self, g, o, d):

        path = []
        total_path = []
        visited = set()
        node = o
        while node != d:
            path.append(node)
            visited.add(node)
            neighbors = g.graph[node]
            neighbor_heuristics = [g.heuristic[neighbor] for neighbor in
neighbors]
            best_neighbor =
neighbors[neighbor_heuristics.index(min(neighbor_heuristics))]
            if best_neighbor in visited:
                return total_path
            node = best_neighbor
            total_path.append(list(path[:]))
        path.append(d)
        total_path.append(list(path[:]))
        print(path)
        return total_path

    def BS(self, g, o, d, bw=1):

        beam = [(g.heuristic[o], (o, [o]))]
        total_path = []
        while beam:
            beam.sort(key=lambda x: x[0])



            best_paths = beam[:bw]
            beam = []
            for misc, (node, path) in best_paths:
                total_path.append(path)
                if node == d:
                    print(path)
                    return total_path
                for neighbor in g.graph[node]:
                    if neighbor not in path:
                        heuristic_score = g.heuristic[neighbor]
                        new_path = path + [neighbor]
                        beam.append((heuristic_score, (neighbor, new_path)))
        return None

    def Oracle(self, g, o, d):

        all_paths = []
        total_path = []
        stack = [(o, [], 0)]  # (node, path, cost)
        while stack:
            current, path, cost = stack.pop()
            total_path.append(path+[current])
            if current == d:
                all_paths.append((path + [current], cost))
            else:
                for neighbor, weight in zip(g.graph[current],
g.weight[current]):
                    if neighbor not in path:
                        stack.append((neighbor, path + [current], cost + weight))
        print(all_paths)
        return total_path

    def BB(self, g, o, d):

        best_path = None
        best_cost = float('inf')
        priority_queue = [(0, o, [])]
        total_path = []

        while priority_queue:
            # Find the path with the lowest cost in the priority queue
            min_index = 0
            for i in range(1, len(priority_queue)):
                if priority_queue[i][0] < priority_queue[min_index][0]:
                    min_index = i
            cost, current, path = priority_queue.pop(min_index)
            total_path.append(path+[current])
            if current == d:
                if cost < best_cost:
                    best_path = path + [current]
                    best_cost = cost
            else:
                for neighbor, weight in zip(g.graph[current],
g.weight[current]):
                    if neighbor not in path:
                        if cost+weight<=best_cost:
                        # Add the neighbor to the priority queue with updated
cost
                            priority_queue.append((cost + weight, neighbor, path
+ [current]))

        print(best_path, best_cost)
        return total_path

    def EL(self, g, o, d):

        best_path = None
        best_cost = float('inf')


        priority_queue = [(0, o, [])]
        total_path = []


        extended_list = {node: False for node in g.graph}

        while priority_queue:

            min_index = 0
            for i in range(1, len(priority_queue)):
                if priority_queue[i][0] < priority_queue[min_index][0]:
                    min_index = i
            cost, current, path = priority_queue.pop(min_index)

            total_path.append(path+[current])

            if current == d:
                if cost < best_cost:
                    best_path = path + [current]
                    best_cost = cost
            else:
                for neighbor, weight in zip(g.graph[current],
g.weight[current]):
                    if not extended_list[current] and not
extended_list[neighbor]:
                        if cost+weight<=best_cost:

                            priority_queue.append((cost + weight, neighbor, path
+ [current]))
            extended_list[current] = True
        print(best_path, best_cost)
        return total_path

    def EH(self, g, o, d):

        best_path = None
        best_cost = float('inf')


        priority_queue = [(0, o, [])]
        total_path = []

        while priority_queue:

            min_index = 0
            for i in range(1, len(priority_queue)):
                if priority_queue[i][0] + g.heuristic[priority_queue[i][1]] <
priority_queue[min_index][0] + g.heuristic[priority_queue[min_index][1]]:
                    min_index = i
            cost, current, path = priority_queue.pop(min_index)

            total_path.append(path+[current])
            if current == d:
```

```
            if cost < best_cost:
                best_path = path + [current]
                best_cost = cost
        else:
            for neighbor, weight in zip(g.graph[current],
g.weight[current]):
                if neighbor not in path:
                    if cost+weight+g.heuristic[current]<=best_cost:

                        priority_queue.append((cost + weight, neighbor, path
+ [current]))

        print(best_path, best_cost)
        return total_path

    def Astar(self, g, o, d):

        best_path = None
        best_cost = float('inf')
        priority_queue = [(0, o, [])]
        total_path = []
        extended_list = {node: False for node in g.graph}

        while priority_queue:
            min_index = 0
            for i in range(1, len(priority_queue)):
                if priority_queue[i][0] + g.heuristic[priority_queue[i][1]] <
priority_queue[min_index][0] + g.heuristic[priority_queue[min_index][1]]:
                    min_index = i
            cost, current, path = priority_queue.pop(min_index)
            visited = set(path)
            total_path.append(path+[current])

            if current == d:
                if cost < best_cost:
                    best_path = path + [current]
                    best_cost = cost
            else:
                for neighbor, weight in zip(g.graph[current],
g.weight[current]):
                    if not extended_list[current] and not
extended_list[neighbor] and neighbor not in visited:
                        if cost+weight+g.heuristic[current]<=best_cost:
                            priority_queue.append((cost + weight, neighbor, path
+ [current]))
                extended_list[current] = True

        print(best_path, best_cost)
        return total_path



    def AOstar(self, g, o, d):
        open_list = [(g.heuristic[o], o, [])]
        closed_list = []
        total_path = []
        while open_list:
            open_list.sort(key=lambda x: x[0])
            h, current, path = open_list.pop(0)
            total_path.append(path+[current])
            print(total_path)
            if current == d:
                print("Optimal path:", path + [current])
                return total_path
            for neighbor, weight in zip(g.graph[current], g.weight[current]):
                if neighbor not in path and neighbor not in closed_list:
                    g_value = len(path) + weight
                    h_value = g.heuristic[neighbor]
                    f_value = g_value + h_value
                    new_path = path + [current]
                    open_list.append((f_value, neighbor, new_path))
            closed_list.append(current)
        print("No path found")
        return None


    def BestFirstSearch(self, g, o, d):

        best_path = None


        priority_queue = [(g.heuristic[o], o, [])]
        total_path = []

        while priority_queue:

            min_index = 0
            for i in range(1, len(priority_queue)):
                if priority_queue[i][0] < priority_queue[min_index][0]:
                    min_index = i
            heuristic, current, path = priority_queue.pop(min_index)

            total_path.append(path+[current])

            if current == d:

                best_path = path + [current]
                print(best_path)
                return total_path
            else:
                for neighbor in g.graph[current]:
                    if neighbor not in path:

                        priority_queue.append((g.heuristic[neighbor], neighbor,
path + [current]))

        print(best_path)
        return total_path

class GraphVisualization:

    def visualize_traversal(self, g, o, d, traversal_algorithm, bw = 1):
        G = nx.Graph()
        for node, neighbors in g.graph.items():
            for neighbor, weight in zip(neighbors, g.weight[node]):
                G.add_edge(node, neighbor, weight=weight)

        if traversal_algorithm.__name__ == "BS":
            paths = traversal_algorithm(g, o, d, bw)
        else:
            paths = traversal_algorithm(g, o, d)
        pos = nx.planar_layout(G)

        fig, ax = plt.subplots()

        def update(frame):
            ax.clear()
            node_labels = {node: f"{node}\nH:{g.heuristic[node]}" for node in
G.nodes()}

            nx.draw(G, pos, with_labels=True, node_size=700,
font_size=10, node_color='lightblue', font_color='black',
font_weight='bold',labels = node_labels, ax=ax)
            edge_labels = {(node, neighbor): G[node][neighbor]['weight'] for
node, neighbor in G.edges()}
            nx.draw_networkx_edge_labels(G, pos,
edge_labels=edge_labels, label_pos=0.5, font_size=8, ax=ax)

            # Highlight the path up to the current step
            if frame < len(paths):
                path = paths[frame]
                path_edges = [(path[i], path[i + 1]) for i in range(len(path) - 1)]
                nx.draw_networkx_edges(G, pos, edgelist=path_edges,
edge_color='red', width=2, ax=ax)

        ani = FuncAnimation(fig, update, frames=len(paths) + 1,
repeat=False, interval=1000)  # Adjust the interval to control animation
speed
        ani.save()
        plt.show()



choice = input("Click Enter to continue with default values, else enter
1")
g = Graph()
algo = Algorithm()

if choice == '':
    g.addEdge('S','A',3)
    g.addEdge('S','B',5)
    g.addEdge('A','B',4)
    g.addEdge('A','D',3)
    g.addEdge('D','G',5)
    g.addEdge('B','C',4)
    g.addEdge('C','E',6)
    g.addHeuristics('S',10)
    g.addHeuristics('A',7)
    g.addHeuristics('B',6)
    g.addHeuristics('C',7)
    g.addHeuristics('D',5)
    g.addHeuristics('E',4)
    g.addHeuristics('G',0)
else:
    pass

GraphVisualization().visualize_traversal(g, 'S', 'G', algo.AOstar)

import math,numpy as np

MAX=np.inf
MIN=-np.inf

arr=[8,7,3,9,9,8,2,4,1,8,8,9,9,9,3,4 ]

lim=math.log(len(arr),2)

def Alpha_Beta_Pruning(state_arr,index,depth,ismaxnode,alpha,beta):
    if(depth==lim):
        return state_arr[index]
    if(ismaxnode):
        best=MIN
        for i in range(2):

val=Alpha_Beta_Pruning(state_arr,index*2+i,depth+1,False,alpha,beta)
            best=max(best,val)
            alpha=max(alpha,best)
            if(alpha>=beta):
                print(f"Pruned at depth :{depth+1}")
                break
        return best
    else:
        best=MAX
        for i in range(2):

val=Alpha_Beta_Pruning(state_arr,index*2+i,depth+1,True,alpha,beta)
            best=min(best,val)
            beta=min(beta,best)
            if(alpha>=beta):
                print(f"Pruned at depth :{depth+1}")
                break
        return best

def minmax(state_arr,index,depth,ismaxnode):
    if(depth==lim):
        return state_arr[index]
    if(ismaxnode):
        best=MIN
        for i in range(2):
```

```
            val=minmax(state_arr,index*2+i,depth+1,False)
            best=max(best,val)
        return best
    else:
        best=MAX
        for i in range(2):
            val=minmax(state_arr,index*2+i,depth+1,True)
            best=min(best,val)
        return best


print("Alpha Beta
Pruning:",Alpha_Beta_Pruning(arr,0,0,True,MIN,MAX))
print("Min-Max Algorithm :",minmax(arr,0,0,True))
```