VISUALIZATION

Program:

```
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 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)
            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)
        plt.show()
```

Ex. No: 01	BRITISH MUSEUM SEARCH
14.07.2023	

Algorithm:

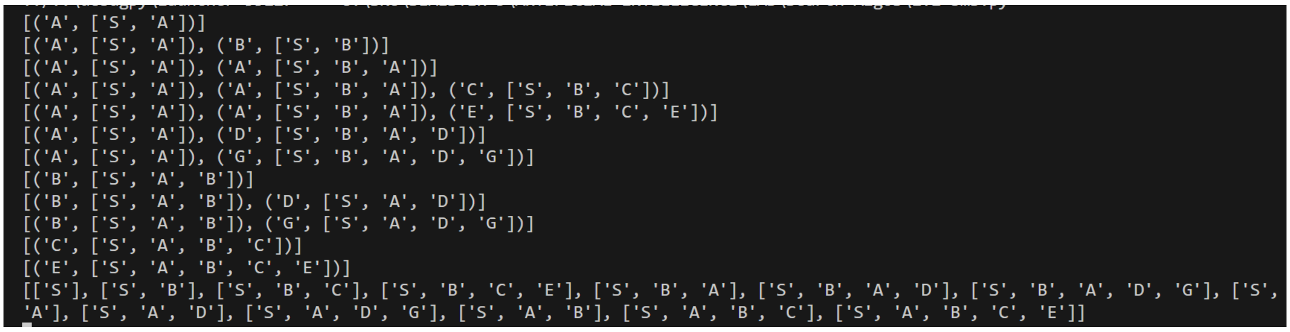
1. Initialize an empty list called paths to store all the paths found during the search.
2. Initialize a stack data structure with the starting node and a list containing only the starting node.
3. While the stack is not empty, do the following:
    a. Pop the top element (node, path) from the stack.
    b. Append the current path to the paths list.
    c. For each neighbor of the current node in the graph g:
        - If the neighbor is not already in the current path, add it to the stack with an updated path.
4. After the search is complete, print and return the list of all paths found during the search.

Program:

```
class Algorithm:
    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]))
                    print(stack)
        print(paths)
        return paths

g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
```

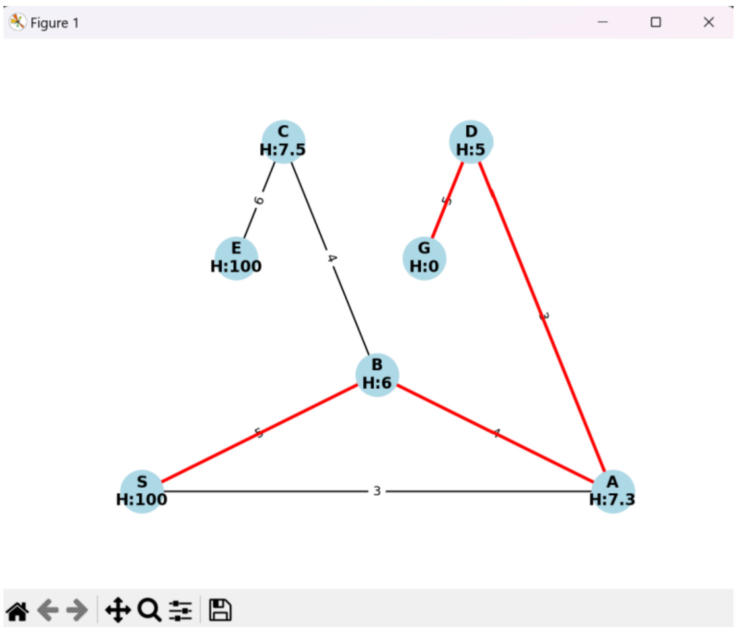
```
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.BMS)
```

Result:



```
[('A', ['S', 'A'])]
[('A', ['S', 'A']), ('B', ['S', 'B'])]
[('A', ['S', 'A']), ('A', ['S', 'B', 'A'])]
[('A', ['S', 'A']), ('A', ['S', 'B', 'A']), ('C', ['S', 'B', 'C'])]
[('A', ['S', 'A']), ('A', ['S', 'B', 'A']), ('E', ['S', 'B', 'C', 'E'])]
[('A', ['S', 'A']), ('D', ['S', 'B', 'A', 'D'])]
[('A', ['S', 'A']), ('G', ['S', 'B', 'A', 'D', 'G'])]
[('B', ['S', 'A', 'B'])]
[('B', ['S', 'A', 'B']), ('D', ['S', 'A', 'D'])]
[('B', ['S', 'A', 'B']), ('G', ['S', 'A', 'D', 'G'])]
[('C', ['S', 'A', 'B', 'C'])]
[('E', ['S', 'A', 'B', 'C', 'E'])]
[['S'], ['S', 'B'], ['S', 'B', 'C'], ['S', 'B', 'C', 'E'], ['S', 'B', 'A'], ['S', 'B', 'A', 'D'], ['S', 'B', 'A', 'D', 'G'], ['S',
'A'], ['S', 'A', 'D'], ['S', 'A', 'D', 'G'], ['S', 'A', 'B'], ['S', 'A', 'B', 'C'], ['S', 'A', 'B', 'C', 'E']]
```

Visualization:



Ex. No: 02	**DEPTH FIRST SEARCH**
21.07.2023	

Algorithm:

1. Initialize an empty set called visited to keep track of visited nodes.
2. Initialize a stack data structure with the starting node and a list containing only the starting node.
3. Initialize an empty list called total_path to store all the paths explored during the search.
4. While the stack is not empty, do the following:

   a. Pop the top element (node, path) from the stack.

   b. Append the current path to the total_path list.

   c. If the current node is the destination node (d), print the path and return the total_path.

   d. If the current node has not been visited yet, add it to the visited set.

   e. For each neighbor of the current node in the graph g (sorted in reverse order):

      - If the neighbor has not been visited, add it to the stack with an updated path.

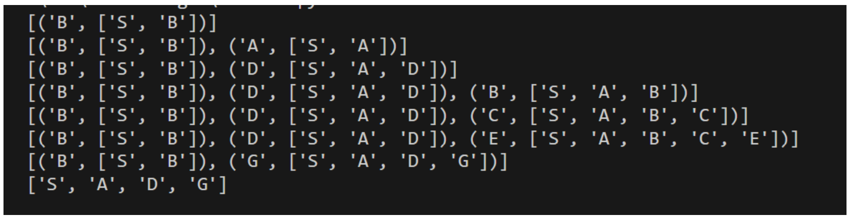
5. If the destination node is not found after the search, return None.

Program:

```
class Algorithm:
    def DFS(self, g, o, d):
        visited = set()
        stack = [(o, [o])]
        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]))
                        print(stack)
        return None
```

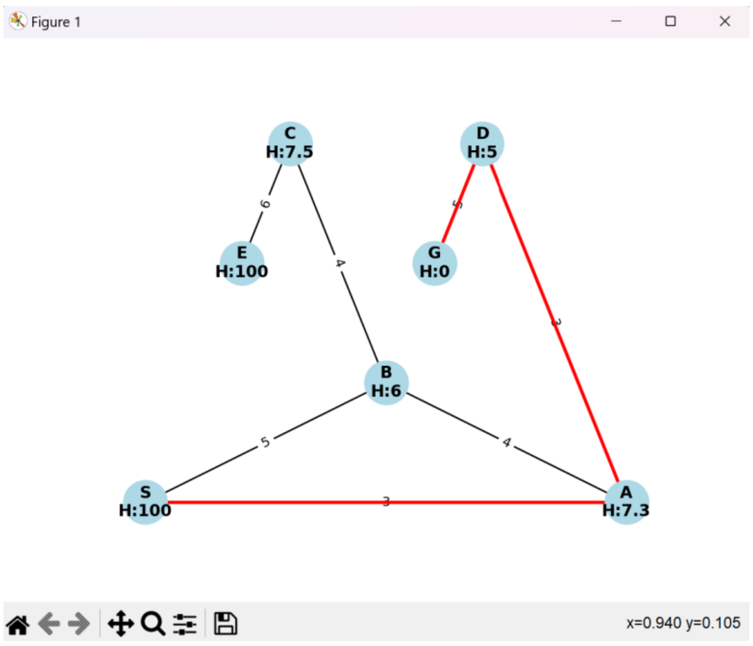
```
g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.DFS)
```

Result:



```
[('B', ['S', 'B'])]
[('B', ['S', 'B']), ('A', ['S', 'A'])]
[('B', ['S', 'B']), ('D', ['S', 'A', 'D'])]
[('B', ['S', 'B']), ('D', ['S', 'A', 'D']), ('B', ['S', 'A', 'B'])]
[('B', ['S', 'B']), ('D', ['S', 'A', 'D']), ('C', ['S', 'A', 'B', 'C'])]
[('B', ['S', 'B']), ('D', ['S', 'A', 'D']), ('E', ['S', 'A', 'B', 'C', 'E'])]
[('B', ['S', 'B']), ('G', ['S', 'A', 'D', 'G'])]
['S', 'A', 'D', 'G']
```

Visualization:



Ex. No: 03	BREADTH FIRST SEARCH
21.07.2023	

Algorithm:

1. Initialize an empty set called visited to keep track of visited nodes.
2. Initialize a queue data structure with the starting node and a list containing only the starting node.
3. Initialize an empty list called total_path to store all the paths explored during the search.
4. While the queue is not empty, do the following:

   a. Remove the first element (node, path) from the front of the queue.

   b. Append the current path to the total_path list.

   c. If the current node is the destination node (d), print the path and return the total_path.

   d. If the current node has not been visited yet, add it to the visited set.

   e. For each neighbor of the current node in the graph g:

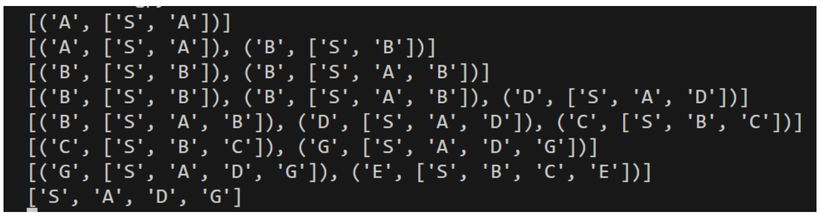
      - If the neighbor has not been visited, add it to the queue with an updated path.
5. If the destination node is not found after the search, return None.

Program:

```
class Algorithm:
    def BFS(self, g, o, d):
        visited = set()
        queue = [(o, [o])]
        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]))
                        print(queue)
        return None
```

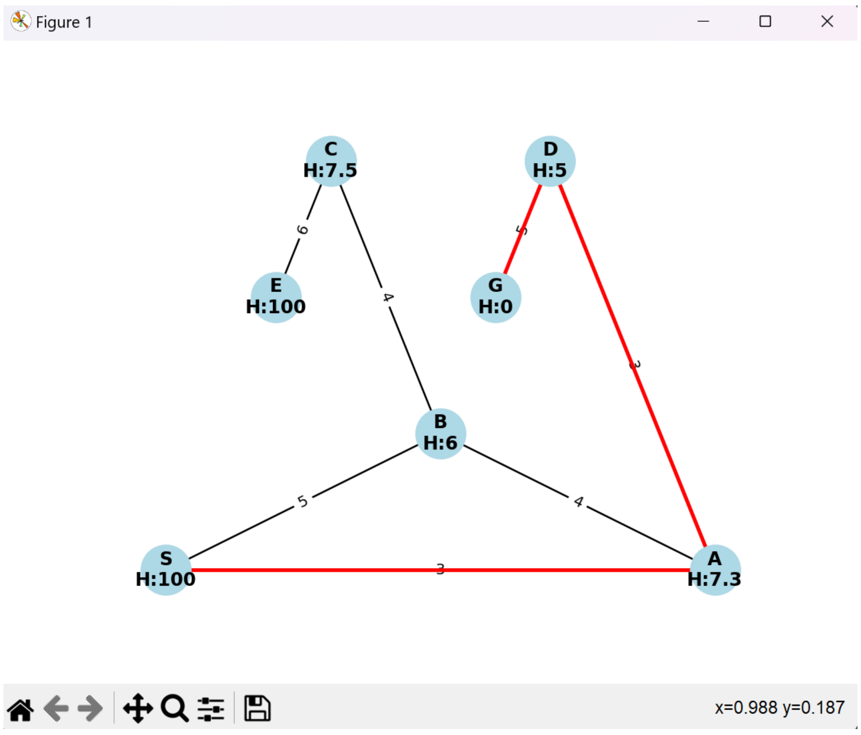
```
g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.BFS)
```

Result:



```
[('A', ['S', 'A'])]
[('A', ['S', 'A']), ('B', ['S', 'B'])]
[('B', ['S', 'B']), ('B', ['S', 'A', 'B'])]
[('B', ['S', 'B']), ('B', ['S', 'A', 'B']), ('D', ['S', 'A', 'D'])]
[('B', ['S', 'A', 'B']), ('D', ['S', 'A', 'D']), ('C', ['S', 'B', 'C'])]
[('C', ['S', 'B', 'C']), ('G', ['S', 'A', 'D', 'G'])]
[('G', ['S', 'A', 'D', 'G']), ('E', ['S', 'B', 'C', 'E'])]
['S', 'A', 'D', 'G']
```

Visualization:



Ex. No: 04	**BEAM SEARCH**
28.07.2023	

Algorithm:

1. Initialize a beam with a single element containing the heuristic value of the origin node and a tuple with the origin node and a list containing only the origin node.

2. Initialize an empty list called total_path to store all the paths explored during the search.

3. While the beam is not empty, do the following:

   a. Sort the beam based on the heuristic values and select the top paths up to the beam width (bw).

   b. Clear the beam for the next iteration.

   c. For each path in the selected best paths:

      - Append the current path to the total_path list.
      - If the current node is the destination node (d), print the path and return the total_path.
      - For each neighbor of the current node in the graph g, If the neighbor is not already in the current path, calculate its heuristic score and update the beam with the new path.

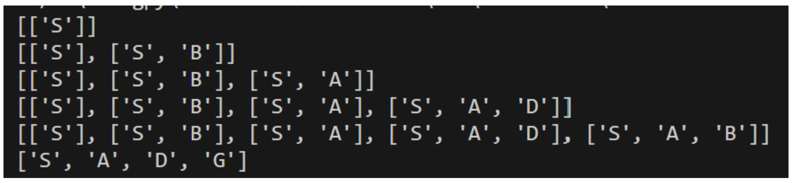
4. If the destination node is not found after the search, return None.

Program:

```
class Algorithm:
    def BS(self, g, o, d, bw=2):
        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)))
```

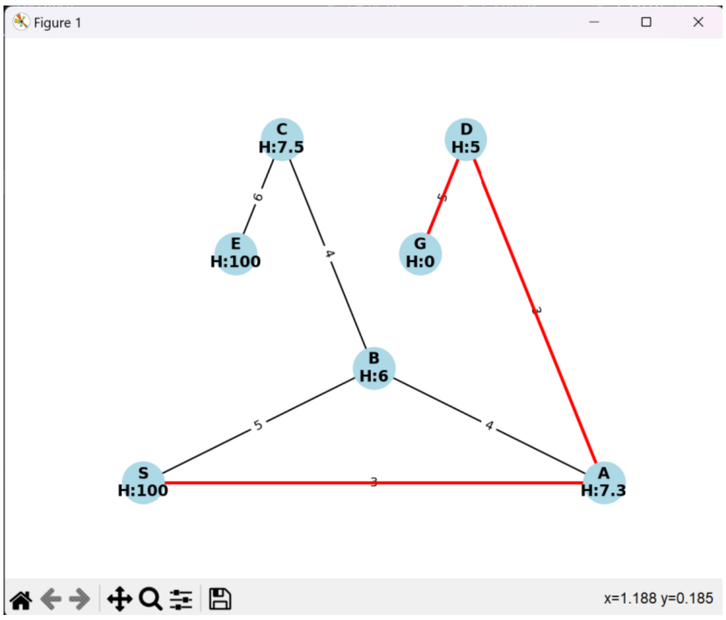
```
                print(total_path)
        return None

g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.BS)
```

Result:



```
[['S']]
[['S'], ['S', 'B']]
[['S'], ['S', 'B'], ['S', 'A']]
[['S'], ['S', 'B'], ['S', 'A'], ['S', 'A', 'D']]
[['S'], ['S', 'B'], ['S', 'A'], ['S', 'A', 'D'], ['S', 'A', 'B']]
['S', 'A', 'D', 'G']
```

Visualization:



Ex. No: 05	HILL CLIMBING
28.07.2023	

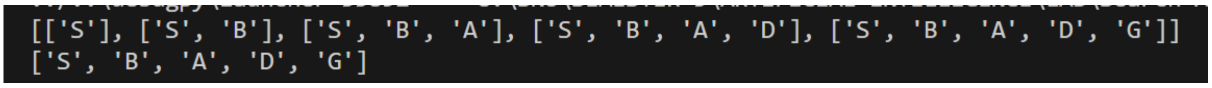
Algorithm:

1. Initialize an empty list called path to store the current path and an empty list called total_path to store all the paths explored during the search.
2. Initialize an empty set called visited to keep track of visited nodes.
3. Set the current node to the origin node (o).
4. While the current node is not the destination node (d), do the following:
   a. Append the current node to the path list.
   b. Add the current node to the visited set.
   c. Retrieve the neighbors of the current node from the graph.
   d. Calculate the heuristic values for each neighbor and select the neighbor with the minimum heuristic value.
   e. If the best neighbor is already visited, return the total_path.
   f. Update the current node to the best neighbor and append the current path to the total_path list.
5. If the destination node is reached, append it to the path and the total_path lists.
6. Print the total_path and path lists and return the total_path.

Program:

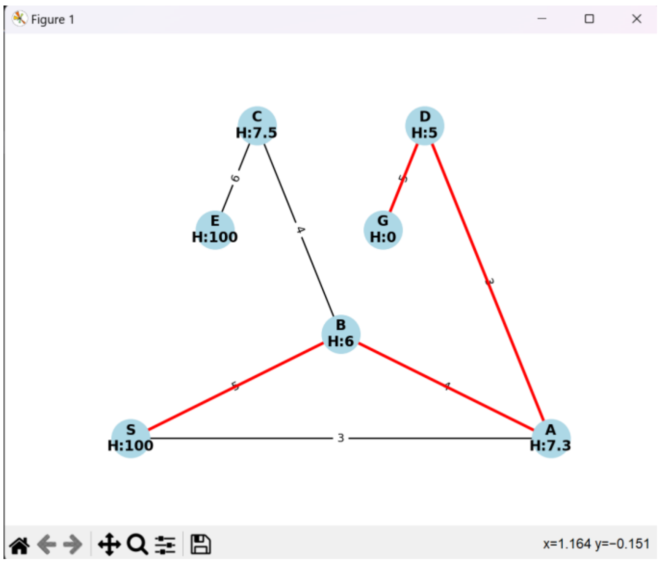
```
class Algorithm:
    def HC(self, g, o, d):
        path = []
        total_path = []
        visited = set()
        node = o
        while node != d:
            path.append(node)
            visited.add(node)
            neighbors = g.graph[node]
            neighbor_heuristics = [g.heuristic[neighbor] 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(total_path)
        print(path)
        return total_path

g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.HC)
```

Result:



```
[['S'], ['S', 'B'], ['S', 'B', 'A'], ['S', 'B', 'A', 'D'], ['S', 'B', 'A', 'D', 'G']]
['S', 'B', 'A', 'D', 'G']
```

Visualization:



Ex. No: 06	ORACLE
18.08.2023	

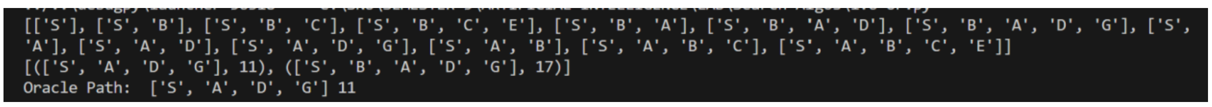
Algorithm:

1. Initialize an empty list called all_paths to store all the paths from the origin to the destination, along with their respective costs.
2. Initialize an empty list called total_path to store all the paths explored during the search.
3. Initialize a stack data structure with a tuple containing the origin node, an empty path, and a cost of 0.
4. While the stack is not empty, do the following:
   a. Pop the top element (current, path, cost) from the stack.
   b. Append the current path (including the current node) to the total_path list.
   c. If the current node is the destination node (d), append the current path and its cost to the all_paths list.
   d. Otherwise, for each neighbor of the current node, calculate the cumulative cost and add the neighbor, updated path, and cost to the stack.
5. Sort the all_paths list based on the costs of the paths.
6. Print the total_path and all_paths lists and return the total_path.

Program:

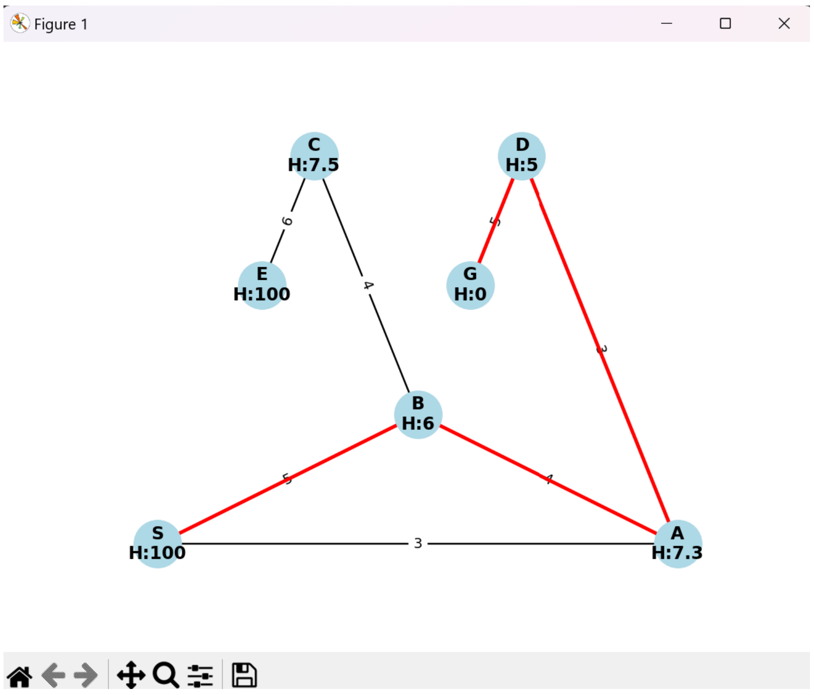
```
class Algorithm:
    def Oracle(self, g, o, d):
        all_paths = []
        total_path = []
        stack = [(o, [], 0)]  # (node, path, cost)
        while stack:
            current, path, cost = stack.pop()
            total_path.append(path+[current])
            if current == d:
                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))
        all_paths=sorted(all_paths)
        print(total_path)
        print(all_paths)
        return total_path

g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.Oracle)
```

Result:



```
[['S'], ['S', 'B'], ['S', 'B', 'C'], ['S', 'B', 'C', 'E'], ['S', 'B', 'A'], ['S', 'B', 'A', 'D'], ['S', 'B', 'A', 'D', 'G'], ['S',
'A'], ['S', 'A', 'D'], ['S', 'A', 'D', 'G'], ['S', 'A', 'B'], ['S', 'A', 'B', 'C'], ['S', 'A', 'B', 'C', 'E']]
[(['S', 'A', 'D', 'G'], 11), (['S', 'B', 'A', 'D', 'G'], 17)]
Oracle Path:  ['S', 'A', 'D', 'G'] 11
```

Visualization:



Ex. No: 07	**ORACLE WITH COST AND HEURISTICS**
18.08.2023	

Algorithm:

1. Initialize an empty list called all_paths to store all the paths from the origin to the destination, along with their respective costs.

2. Initialize an empty list called total_path to store all the paths explored during the search.

3. Initialize a stack data structure with a tuple containing the origin node, an empty path, and a cost of 0.

4. While the stack is not empty, do the following:

   a. Pop the top element (current, path, cost) from the stack.

   b. Append the current path (including the current node) to the total_path list.

   c. If the current node is the destination node (d), append the current path and its cost to the all_paths list.

   d. Otherwise, for each neighbor of the current node, calculate the cumulative cost, including the heuristic value, and add the neighbor, updated path, and cost to the stack.

5. Print the total_path and all_paths lists and return the total_path.

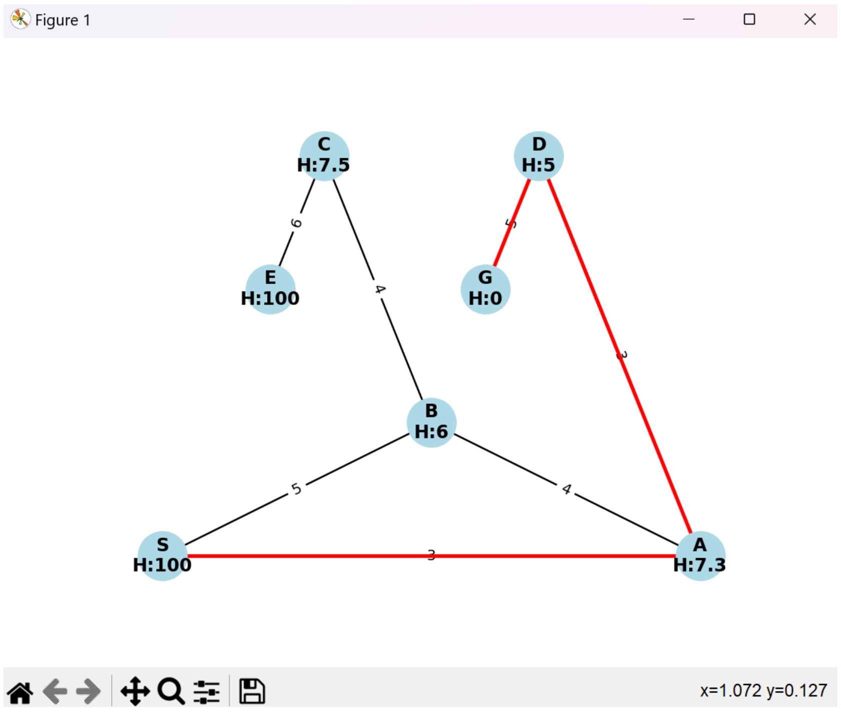
Program:

```
class Algorithm:
    def OHC(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 + g.heuristic[neighbor]))
        print(total_path)
        print(all_paths)
        return total_path

g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.OHC)
```

Result:

```
[['S'], ['S', 'B'], ['S', 'B', 'C'], ['S', 'B', 'C', 'E'], ['S', 'B', 'A'], ['S', 'B', 'A', 'D'], ['S', 'B', 'A', 'D', 'G'], ['S',
'A'], ['S', 'A', 'D'], ['S', 'A', 'D', 'G'], ['S', 'A', 'B'], ['S', 'A', 'B', 'C'], ['S', 'A', 'B', 'C', 'E']]
[(['S', 'A', 'D', 'G'], 11), (['S', 'B', 'A', 'D', 'G'], 17)]
```

Visualization:



Ex. No: 08	BRANCH AND BOUND
25.08.2023	

Algorithm:

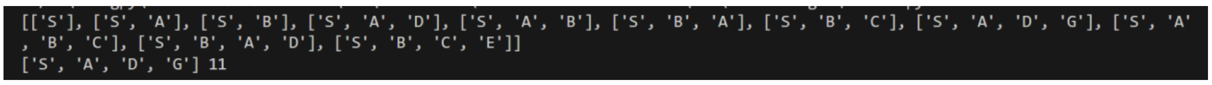
1. Initialize variables best_path and best_cost to track the best path and its corresponding cost, setting the initial cost to infinity.
2. Initialize a priority queue containing the cost, the current node, and an empty path.
3. Initialize an empty list called total_path to store all the paths explored during the search.
4. While the priority queue is not empty, do the following:
   a. Find the index of the element with the minimum cost in the priority queue and pop it.
   b. Append the current path (including the current node) to the total_path list.
   c. If the current node is the destination node (d), update the best_path and best_cost if the current cost is lower than the previous best cost.
   d. Otherwise, for each neighbor of the current node, check if adding the neighbor to the path would not exceed the best_cost. If it does not, add the neighbor, updated cost, and path to the priority queue.
5. Print the total_path, best_path, and best_cost and return the total_path.

Program:

```
class Algorithm:
    def BB(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] < 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]):
```

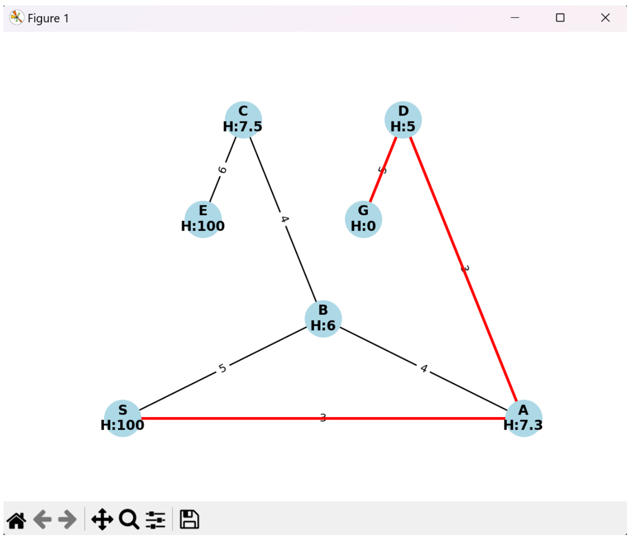
```
                    if neighbor not in path:
                        if cost+weight<=best_cost:
                            priority_queue.append((cost + weight,
neighbor, path + [current]))
        print(total_path)
        print(best_path, best_cost)
        return total_path
g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.BB)
```

Result:



```
[['S'], ['S', 'A'], ['S', 'B'], ['S', 'A', 'D'], ['S', 'A', 'B'], ['S', 'B', 'A'], ['S', 'B', 'C'], ['S', 'A', 'D', 'G'], ['S', 'A'
, 'B', 'C'], ['S', 'B', 'A', 'D'], ['S', 'B', 'C', 'E']]
['S', 'A', 'D', 'G'] 11
```

Visualization:



Ex. No: 09	BRANCH AND BOUND WITH HEURISTICS
22.09.2023	

Algorithm:

1. Initialize variables best_path and best_cost to track the best path and its corresponding cost, setting the initial cost to infinity. Initialize a priority queue (implemented as a list) with a tuple containing the cost, the current node, and an empty path.

2. Initialize an empty list called total_path to store all the paths explored during the search.

3. While the priority queue is not empty, do the following:

a. Find the index of the element with the minimum combined cost and heuristic value in the priority queue and pop it.

c. Append the current path (including the current node) to the total_path list.

d. If the current node is the destination node (d), update the best_path and best_cost if the current cost is lower than the previous best cost.

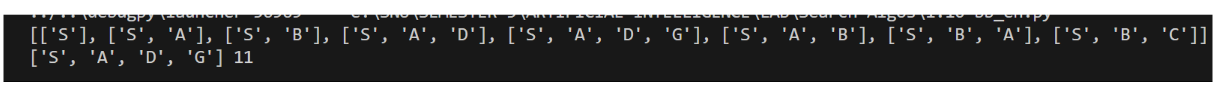
e. Otherwise, for each neighbor of the current node, check if adding the neighbor to the path along with its heuristic value would not exceed the best_cost. If it does not, add the neighbor, updated cost, and path to the priority queue.

4. Print the total_path, best_path, and best_cost and return the total_path.

Program:

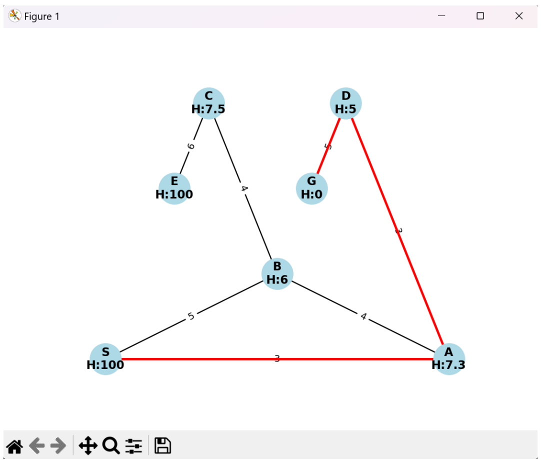
```
class Algorithm:
    def EH(self, g, o, d):
        best_path = None
        best_cost = float('inf')
        priority_queue = [(0, o, [])]
        total_path = []
        while priority_queue:
            min_index = 0
            for i in range(1, len(priority_queue)):
                if priority_queue[i][0] +
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(total_path)
        print(best_path, best_cost)
        return total_path
g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.EH)
```

Result:



```
[['S'], ['S', 'A'], ['S', 'B'], ['S', 'A', 'D'], ['S', 'A', 'D', 'G'], ['S', 'A', 'B'], ['S', 'B', 'A'], ['S', 'B', 'C']]
['S', 'A', 'D', 'G'] 11
```

Visualization:



Ex. No: 10	BRANCH AND BOUND WITH EXTENDED LIST
22.09.2023	

Algorithm:

1. Initialize variables best_path and best_cost to track the best path and its corresponding cost, setting the initial cost to infinity. Initialize a priority queue (implemented as a list) with a tuple containing the cost, the current node, and an empty path.

2. Initialize an empty list called total_path to store all the paths explored during the search.

3. Initialize an extended_list dictionary to keep track of visited nodes, setting all nodes initially to False.

4. While the priority queue is not empty, do the following:

   a. Find the index of the element with the minimum cost in the priority queue and pop it.

   b. Append the current path (including the current node) to the total_path list.

   c. If the current node is the destination node (d), update the best_path and best_cost if the current cost is lower than the previous best cost.

   d. Otherwise, for each neighbor of the current node, check if neither the current node nor the neighbor is in the extended list. If the condition is met and the cost is within the best_cost, add the neighbor, updated cost, and path to the priority queue.

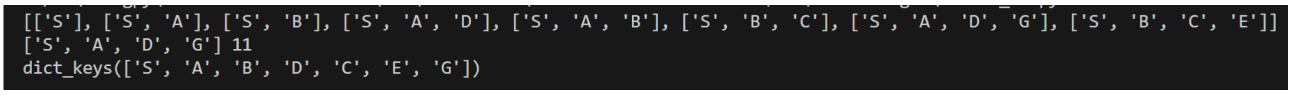
5. Mark the current node as visited in the extended_list.

6. Print the total_path, best_path, best_cost, and the keys of the extended_list. And return the total_path.

Program:

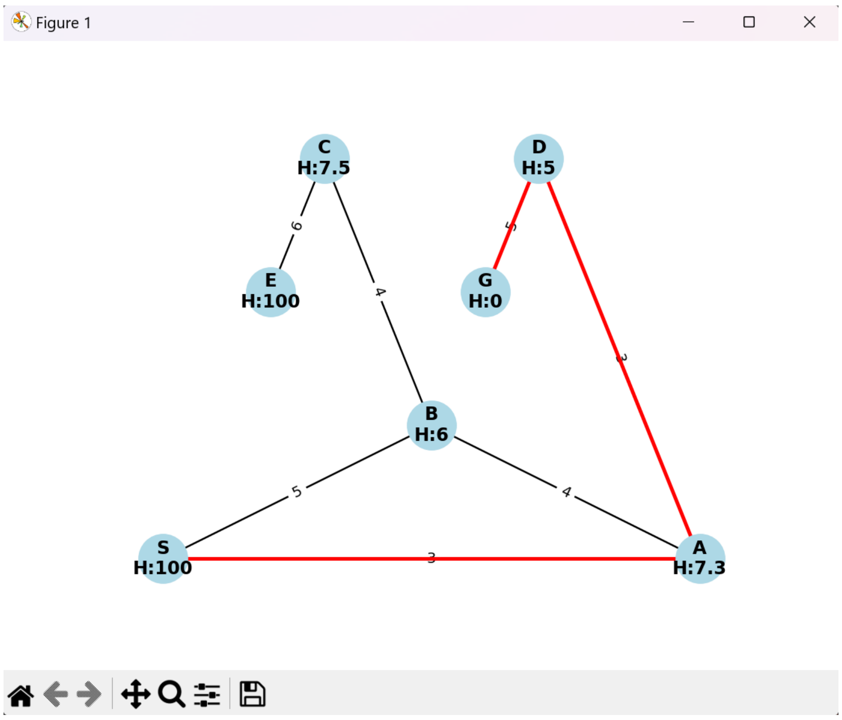
```
class Algorithm:
    def EL(self, g, o, d):
        best_path = None
        best_cost = float('inf')
        priority_queue = [(0, o, [])]
        total_path = []
        extended_list = {node: False for node in g.graph}
        while priority_queue:
            min_index = 0
            for i in 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(total_path)
        print(best_path, best_cost)
        print(extended_list.keys())
        return total_path

g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.EL)
```

Result:



```
[['S'], ['S', 'A'], ['S', 'B'], ['S', 'A', 'D'], ['S', 'A', 'B'], ['S', 'B', 'C'], ['S', 'A', 'D', 'G'], ['S', 'B', 'C', 'E']]
['S', 'A', 'D', 'G'] 11
dict_keys(['S', 'A', 'B', 'D', 'C', 'E', 'G'])
```

Visualization:



Ex. No: 11	A*
29.09.2023	

Algorithm:

The provided code represents an implementation of the A* search algorithm, a widely used pathfinding and graph traversal algorithm that combines the advantages of both Dijkstra's algorithm and a heuristic search. Here is the breakdown of the algorithm based on the given code:

1. Initialize variables best_path and best_cost to track the best path and its corresponding cost, setting the initial cost to infinity. Initialize a priority queue (implemented as a list) with a tuple containing the cost, the origin node, and an empty path.
2. Initialize an empty list called total_path to store all the paths explored during the search.
3. Initialize an extended_list dictionary to keep track of visited nodes, setting all nodes initially to False.
4. While the priority queue is not empty, do the following:

   a. Find the index of the element with the minimum combined cost and heuristic value in the priority queue and pop it.

   b. Create a set of visited nodes from the current path.

   c. Append the current path (including the current node) to the total_path list.

   d. If the current node is the destination node (d), update the best_path and best_cost if the current cost is lower than the previous best cost.

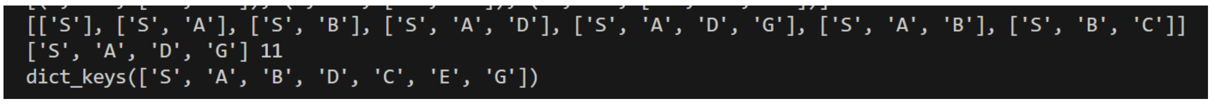
   e. Otherwise, for each neighbor of the current node, check if the current node and the neighbor are not in the extended list and the neighbor is not in the visited set. If the condition is met and the cost is within the best_cost, add the neighbor, updated cost, and path to the priority queue.
5. Mark the current node as visited in the extended_list.
6. Print the total_path, best_path, best_cost, extended_list and return the total_path.

Program:

```
class Algorithm:
    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]))
                            #print(priority_queue)
            extended_list[current] = True
        print(total_path)
        print(best_path, best_cost)
        print(extended_list.keys())
        return total_path

g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
```

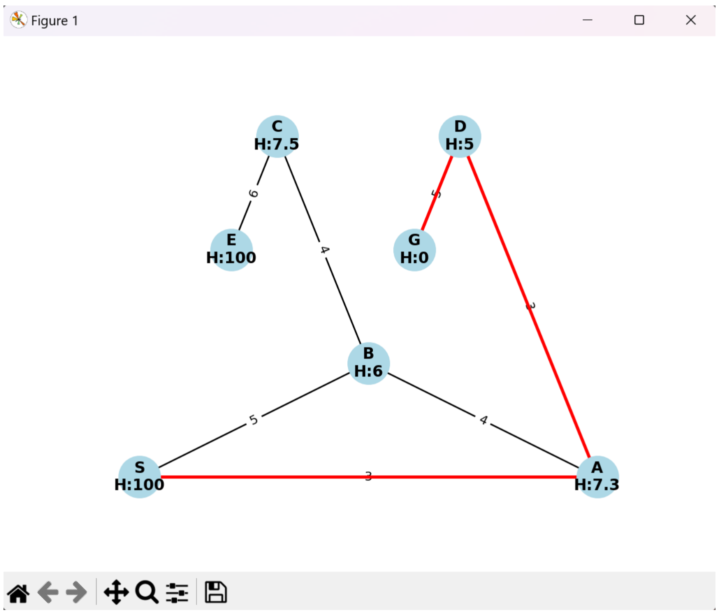
```
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.Astar)
```

Result:



```
[['S'], ['S', 'A'], ['S', 'B'], ['S', 'A', 'D'], ['S', 'A', 'D', 'G'], ['S', 'A', 'B'], ['S', 'B', 'C']]
['S', 'A', 'D', 'G'] 11
dict_keys(['S', 'A', 'B', 'D', 'C', 'E', 'G'])
```

Visualization:



Ex. No: 12	AO*
13.10.2023	

Algorithm:

1. Initialize an open list, a closed list, and a total_path list to manage the nodes, visited nodes, and paths explored during the search, respectively.

2. While the open list is not empty, do the following:

   a. Sort the open list based on the heuristic values and Pop the element with the lowest heuristic value from the open list.

   b. Append the current path (including the current node) to the total_path list.

   c. If the current node is the destination node (d), print the optimal path and return the total_path.

   d. For each neighbor of the current node, calculate the g, h, and f values and update the open list with the new values and path if the neighbor is not in the path or closed list.

3. Add the current node to the closed list to mark it as visited.

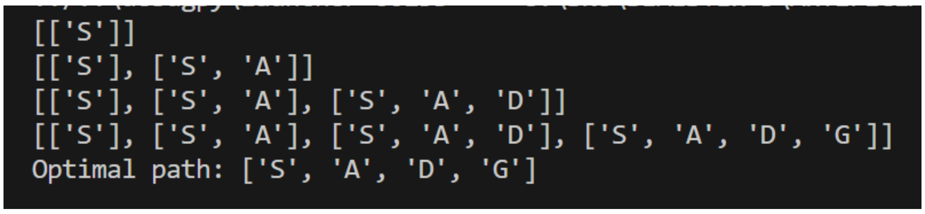
4. If no path is found, print "No path found" and return None.

Program:

```
class Algorithm:
    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]
```

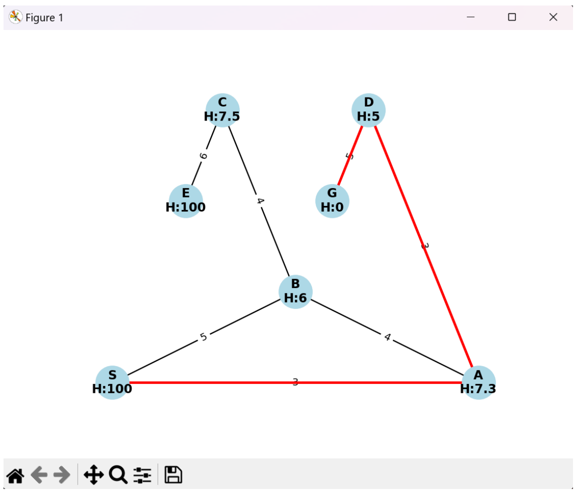
```
          open_list.append((f_value, neighbor, new_path))
      closed_list.append(current)
    print("No path found")
    return None

g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g, 'S', 'G', algo.AOstar)
```

Result:



```
[['S']]
[['S'], ['S', 'A']]
[['S'], ['S', 'A'], ['S', 'A', 'D']]
[['S'], ['S', 'A'], ['S', 'A', 'D'], ['S', 'A', 'D', 'G']]
Optimal path: ['S', 'A', 'D', 'G']
```

Visualization:



Ex. No: 13	BEST FIRST SEARCH
20.10.2023	

Algorithm:

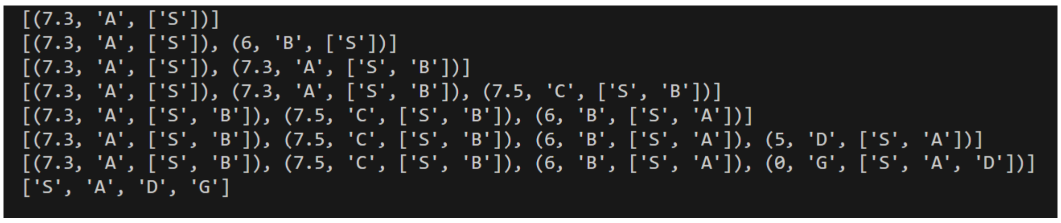
1. Initialize a variable best_path to keep track of the best path found during the search and a priority queue (implemented as a list) with a tuple containing the heuristic value, the origin node, and an empty path.
2. Initialize an empty list called total_path to store all the paths explored during the search.
3. While the priority queue is not empty, do the following:
   a. Find the index of the element with the minimum heuristic value in the priority queue.
   b. Pop the element with the minimum heuristic value from the priority queue.
   c. Append the current path (including the current node) to the total_path list.
   d. If the current node is the destination node (d), update the best_path with the current path, print it, and return the total_path.
   e. Otherwise, for each neighbor of the current node, check if the neighbor is not in the path. If it is not, add the neighbor, updated heuristic value, and path to the priority queue.
4. If the destination node is not found, print the best_path and return the total_path.

Program:

```
class Algorithm:
    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:
```

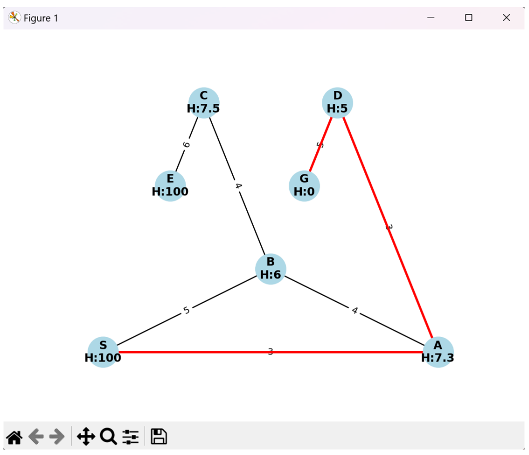
```
            priority_queue.append((g.heuristic[neighbor],
neighbor, path + [current]))
            print(priority_queue)
    print(best_path)
    return total_path

g = Graph()
algo = Algorithm()
g.addEdge('S','A',3)
g.addEdge('S','B',5)
g.addEdge('A','B',4)
g.addEdge('A','D',3)
g.addEdge('B','C',4)
g.addEdge('C','E',6)
g.addEdge('D','G',5)
g.addHeuristics('A',7.3)
g.addHeuristics('B',6)
g.addHeuristics('C',7.5)
g.addHeuristics('D',5)
g.addHeuristics('G',0)
GraphVisualization().visualize_traversal(g,'S','G', algo.BestFirstSearch)
```

Result:

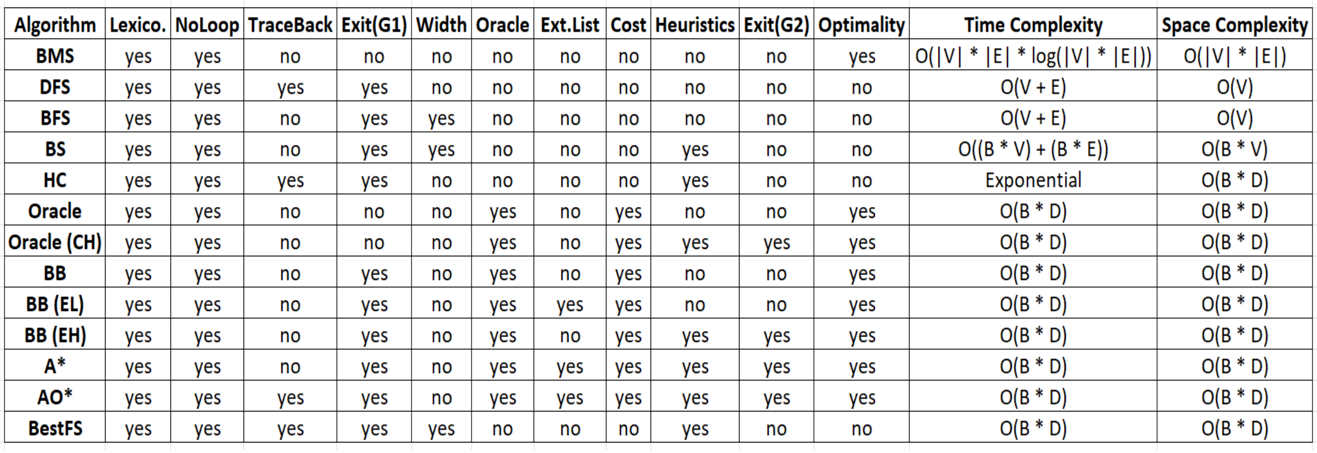


```
[(7.3, 'A', ['S'])]
[(7.3, 'A', ['S']), (6, 'B', ['S'])]
[(7.3, 'A', ['S']), (7.3, 'A', ['S', 'B'])]
[(7.3, 'A', ['S']), (7.3, 'A', ['S', 'B']), (7.5, 'C', ['S', 'B'])]
[(7.3, 'A', ['S', 'B']), (7.5, 'C', ['S', 'B']), (6, 'B', ['S', 'A'])]
[(7.3, 'A', ['S', 'B']), (7.5, 'C', ['S', 'B']), (6, 'B', ['S', 'A']), (5, 'D', ['S', 'A'])]
[(7.3, 'A', ['S', 'B']), (7.5, 'C', ['S', 'B']), (6, 'B', ['S', 'A']), (0, 'G', ['S', 'A', 'D'])]
['S', 'A', 'D', 'G']
```

Visualization:



PERFORMANCE COMPARISON:



Algorithm	Lexico.	NoLoop	TraceBack	Exit(G1)	Width	Oracle	Ext.List	Cost	Heuristics	Exit(G2)	Optimality	Time Complexity	Space Complexity
BMS	yes	yes	no	no	no	no	no	no	no	no	yes	O(|V| * |E| * log(|V| * |E|))	O(|V| * |E|)
DFS	yes	yes	yes	yes	no	no	no	no	no	no	no	O(V + E)	O(V)
BFS	yes	yes	no	yes	yes	no	no	no	no	no	no	O(V + E)	O(V)
BS	yes	yes	no	yes	yes	no	no	no	yes	no	no	O((B * V) + (B * E))	O(B * V)
HC	yes	yes	yes	yes	no	no	no	no	yes	no	no	Exponential	O(B * D)
Oracle	yes	yes	no	no	no	yes	no	yes	no	no	yes	O(B * D)	O(B * D)
Oracle (CH)	yes	yes	no	no	no	yes	no	yes	yes	yes	yes	O(B * D)	O(B * D)
BB	yes	yes	no	yes	no	yes	no	yes	no	no	yes	O(B * D)	O(B * D)
BB (EL)	yes	yes	no	yes	no	yes	yes	yes	no	no	yes	O(B * D)	O(B * D)
BB (EH)	yes	yes	no	yes	no	yes	no	yes	yes	yes	yes	O(B * D)	O(B * D)
A*	yes	yes	no	yes	no	yes	yes	yes	yes	yes	yes	O(B * D)	O(B * D)
AO*	yes	yes	yes	yes	no	yes	yes	yes	yes	yes	yes	O(B * D)	O(B * D)
BestFS	yes	yes	yes	yes	yes	no	no	no	yes	no	no	O(B * D)	O(B * D)

JUSTIFICATION FOR THE PERFORMANCE:

British Museum Search(BMS) is a simple algorithm that explores all possible paths from the origin to the destination, making it suitable for small graphs or problems with limited search spaces. BMS is effective for small-scale problems but can become inefficient for larger and more complex search spaces, as it does not incorporate any heuristics or optimizations to guide the search.

Depth-First Search (DFS) is a traversal algorithm that excels in scenarios where the goal is to explore deep into a graph, tree, or maze. DFS is particularly well-suited for problems that involve extensive exploration of deep search spaces, and it is often used in tasks like topological sorting or solving puzzles with significant depth. One of its key advantages is its efficient use of memory, making it a practical choice when memory constraints are a concern.

Breadth-First Search (BFS) is the algorithm of choice when the primary objective is to find the shortest path in an unweighted graph or to explore the shallowest levels in a graph or tree. BFS is highly effective for tasks such as network routing, web crawling, and situations where visiting neighbours before going deeper is a priority. Its ability to guarantee the shortest path makes it indispensable for such applications.

Hill Climbing is a local search algorithm that focuses on finding a solution by iteratively moving towards the direction of increasing value or minimizing the cost. It may get stuck in local optima, making it less effective for complex problems that involve multiple peaks or valleys in the search space.

Beam Search is a heuristic search algorithm that explores a limited set of the most promising paths at each iteration, making it more effective than plain BFS and DFS for certain problems. It balances between exploration and exploitation, ensuring that it doesn't get stuck in local optima and has the potential to find better solutions than Hill Climbing.

The Oracle algorithm is designed to find the optimal solution by considering the costs associated with each node in the graph. The Oracle algorithm is more sophisticated than BMS, effective in finding optimal solutions based solely on the cost information but may not consider additional insights into the problem domain.

The Oracle with Heuristics algorithm enhances the Oracle algorithm by incorporating heuristic information that guides the search process toward more promising paths in the graph. By leveraging heuristic information, the Oracle with Heuristics algorithm can make more informed decisions during the search, leading to a potentially faster convergence to the optimal solution.

Branch and Bound systematically explores the solution space, ensuring optimality for problems with a finite number of solutions. It's highly effective for smaller instances but can become computationally intensive as the problem size grows.

Branch and Bound with Heuristics leverages heuristics to improve efficiency while sacrificing the guarantee of optimality. It's a practical choice for complex problems where finding the exact optimal solution is infeasible, but high-quality near-optimal solutions are needed.

Branch and Bound with Extended List enhances traditional B&B by managing a larger list of potential solutions, reducing redundancy, and improving efficiency. It's a valuable tool for problems with extensive solution spaces, where careful management of candidate solutions is critical.

A* is a complete and optimal search algorithm. This means that it is guaranteed to find a solution if one exists, and it is guaranteed to find the optimal solution if one exists. A* works by using a heuristic function to estimate the cost of reaching the goal node from any given node. The algorithm then expands the node with the lowest estimated cost.

AO* is an extension of A* that is designed to be more efficient in dynamic environments. Dynamic environments are environments where the state of the world can change randomly. AO* works by maintaining a cache of heuristic estimates which allows the algorithm to avoid recalculating the heuristic for nodes that have already been expanded.

Best First Search is a simple search algorithm that expands the node with the best heuristic estimate. However, it is not complete or optimal. This means that it may not find a solution if one exists, and it may not find the optimal solution if one exists.

JUSTIFICATION FOR THE CHOICE OF PROGRAMMING MEDIUM:

Python is chosen for implementing and visualizing search algorithms due to its simplicity, readability, extensive libraries, easy integration of data structures, and effortless visualization using libraries like matplotlib and networkx.

Ex. No: 14	**MINIMAX ALGORITHM**
27.10.2023	

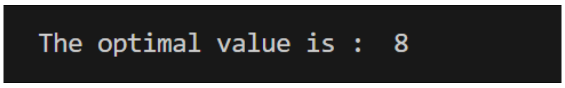
Algorithm:

1. Define a function, minimax, that takes parameters for the current depth, node index, turn flag (max or min), scores list, and the target depth.
2. Check if the current depth is equal to the target depth. If it is, return the score associated with the current node index.
3. If it is the max turn, return the maximum value from the two recursive calls of minimax with updated depth, node index, and turn for the left and right child nodes.
4. If it is not the max turn, return the minimum value from the two recursive calls of minimax with updated depth, node index, and turn for the left and right child nodes.
5. Define the scores list, representing the values associated with each node in the tree.
6. Calculate the tree depth based on the logarithm of the number of elements in the scores list.
7. Print the optimal value obtained by calling the minimax function with initial parameters.

Program:

```
import math

def minimax (curDepth, nodeIndex, maxTurn, scores, targetDepth):
    if (curDepth == targetDepth):
        return scores[nodeIndex]
    if (maxTurn):
        return max(minimax(curDepth + 1, nodeIndex * 2, False, scores, 
targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores, 
targetDepth))
    else:
        return min(minimax(curDepth + 1, nodeIndex * 2, True, scores, 
targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores, 
targetDepth))
scores = [4,8,9,3,2,-2,9,-1]
treeDepth = math.log(len(scores), 2)
print("\nThe optimal value is : ", minimax(0, 0, True, scores, treeDepth))
```

Result:



```
The optimal value is :  8
```

Ex. No: 15	**ALPHA BETA PRUNING**
27.10.2023	

Algorithm:

1. The `minimax` function now takes two additional parameters, `alpha` and `beta`, which represent the best values that the maximizing and minimizing players, respectively, can guarantee.

2. Within the `minimax` function, when the `maxTurn` is True, the function initializes the variable `best` as negative infinity and iterates through the child nodes to find the maximum value. It also updates the value of `alpha` and performs pruning if the beta value is less than or equal to the alpha value.

3. When the `maxTurn` is False, the function initializes the `best` variable as positive infinity and iterates through the child nodes to find the minimum value. It updates the value of `beta` and performs pruning if the beta value is less than or equal to the alpha value.

4. The code displays a message when a pruning operation occurs, indicating the depth, index, value, and the pruned child.

5. The `scores` list contains the values associated with each node in the tree.

6. The code calculates the tree depth based on the logarithm of the number of elements in the `scores` list.

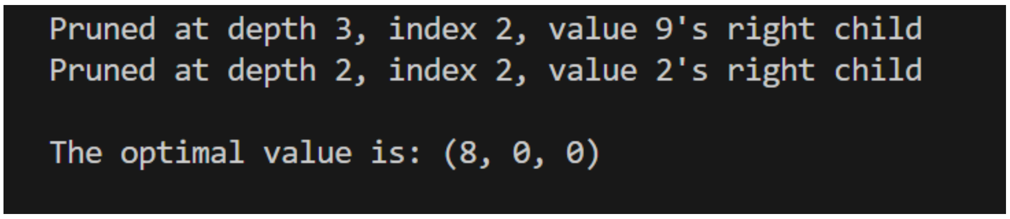
7. The `minimax` function returns the best value, along with the current depth and node index.

Program:

```
import math


def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth, alpha,
beta):
    if curDepth == targetDepth:
        return scores[nodeIndex], curDepth, nodeIndex
    if maxTurn:
        best = -math.inf
        for i in range(2):
            val, d, ind = minimax(curDepth + 1, nodeIndex * 2 + i, False,
scores, targetDepth, alpha, beta)
            if val > best:
                best = val
            alpha = max(alpha, best)
            if beta <= alpha:
                print(f"Pruned at depth {d}, index {ind}, value {val}'s
right child")
```

```
                break
        return best, curDepth, nodeIndex
    else:
        best = math.inf
        for i in range(2):
            val, d, ind = minimax(curDepth + 1, nodeIndex * 2 + i, True,
scores, targetDepth, alpha, beta)
            if val < best:
                best = val
            beta = min(beta, best)
            if beta <= alpha:
                print(f"Pruned at depth {d}, index {ind}, value {val}'s
right child")
                break
        return best, curDepth, nodeIndex


scores = [4, 8, 9, 3, 2, -2, 9, -1]
treeDepth = math.log(len(scores), 2)
print("\nThe optimal value is:", minimax(0, 0, True, scores, treeDepth, -
math.inf, math.inf))
```

Result:



```
Pruned at depth 3, index 2, value 9's right child
Pruned at depth 2, index 2, value 2's right child

The optimal value is: (8, 0, 0)
```

PERFORMANCE COMPARISON:

Algorithm	Time Complexity	Space Complexity
MINIMAX	$O(b^m)$ b is the branching factor m is the maximum depth of the tree	Dependent on the depth of the recursion.
ALPHA BETA PRUNING	$O(b^{m/2})$ (best case) $O(b^m)$ (worst case)	Similar to Minimax but with reduced memory consumption due to pruning.

JUSTIFICATION FOR THE PERFORMANCE:

The Minimax algorithm and its enhanced version, the Alpha-Beta Pruning algorithm, both serve the purpose of determining the optimal decision in a two-player game scenario.

While Minimax exhaustively explores the entire game tree to find the optimal move, Alpha-Beta Pruning enhances Minimax by pruning subtrees that do not affect the final decision, leading to a more efficient search process.

- Minimax guarantees the optimal solution in two-player games. However, it may be inefficient for large game trees, as it explores all possible moves without considering the potential irrelevance of some branches.

- Alpha-Beta Pruning enhances the Minimax algorithm by eliminating subtrees that do not affect the final decision, leading to a more efficient search process. It can significantly reduce the number of nodes evaluated, making it more practical for larger game trees.

JUSTIFICATION FOR THE CHOICE OF PROGRAMMING MEDIUM:

Python is a popular choice for implementing minimax and alpha-beta pruning algorithms due to its simplicity, readability, and extensive libraries for data structures. Its dynamic typing and ease of debugging make it a practical language for quickly prototyping and experimenting with complex algorithms, which is essential for AI and game theory implementations like minimax and alpha-beta pruning.

Ex. No: 16	**CARLA**
03.11.2023	

DESCRIPTION

CARLA (Car Learning to Act) is an open-source simulator for autonomous driving research. In CARLA, you can set up various scenarios to test and train autonomous driving algorithms.

Agent:

The agent in a CARLA setup represents the autonomous vehicle or the entity responsible for controlling the vehicle. It can be implemented using various control algorithms. When generating an NPC (Non-Player Character) in the CARLA simulator, the agent represents a simulated autonomous vehicle that is not directly controlled by a human.

NPCs are commonly used to populate the environment, adding realistic traffic and various driving behaviours to the simulation. These NPCs typically operate as rule-based agents, following predefined sets of rules and heuristics to simulate realistic driving behaviours and interactions.

1. Basic driving behaviour: NPCs adhere to traffic rules, maintain appropriate speeds, and follow lane discipline, contributing to a more realistic and dynamic driving environment.
2. Interaction with the environment: NPCs respond to traffic signals, yield to other vehicles, and navigate intersections, simulating real-world driving scenarios.
3. Collision avoidance: NPCs are programmed to avoid collisions with other vehicles, pedestrians, and obstacles, contributing to a safer and more realistic simulation environment.
4. Path planning: NPCs may have a basic path planning algorithm that enables them to navigate from one point to another within the simulated environment, considering factors such as traffic, road conditions, and the presence of other vehicles.

Environment:

The CARLA environment represents the virtual world in which the agent operates. This environment consists of various elements like:

- Roads and infrastructure: CARLA provides a realistic urban environment with roads, intersections, traffic lights, and various types of road infrastructure.
- Other traffic participants: Besides the agent's vehicle, CARLA simulates the presence of other vehicles, walking and running pedestrians, and cyclists that interact with the agent.
- Weather and lighting: CARLA supports dynamic weather conditions, enabling testing in various scenarios, including evening or nighttime and stormy or cloudy.

Sensor:

Sensors in CARLA are used to provide the agent with information about its surroundings, just like real-world sensors in an autonomous vehicle.

Common sensors used in the CARLA setups includes: Camera Manager, Lane Invasion Sensor, Collision sensor, GNSS, Radar, GPS, IMU (Inertial Measurement Unit)

Algorithmic Model:

In a CARLA setup with NPCs (Non-Player Characters) and various sensors, the algorithmic model refers to the control and decision-making algorithms that govern the behavior of the NPCs within the simulated environment.

1. Rule-based algorithms: Rule-based models define the behavior of NPCs based on predefined sets of rules and heuristics. These rules govern actions such as following traffic regulations, maintaining safe distances, obeying traffic signals, and reacting to other vehicles or pedestrians.

2. Path planning algorithms: Path planning algorithms enable NPCs to navigate from their current position to a specified destination while avoiding obstacles and other vehicles. These algorithms use information from sensors like GPS, radar, and cameras to map out safe and efficient routes and make decisions based on the surrounding environment and the desired destination.

3. Collision avoidance algorithms: Collision avoidance algorithms help NPCs detect and respond to potential collision risks with other vehicles, pedestrians, or obstacles. These algorithms utilize sensor data from cameras, radars, and collision sensors to assess the environment's dynamics and make real-time decisions to prevent accidents and ensure safe navigation.

4. Behavior modeling algorithms: Behavior modeling algorithms enable NPCs to exhibit realistic driving behaviors, such as lane changing, merging, yielding, and following appropriate driving etiquette. These models consider information from sensors like lane invasion sensors, GPS, and IMUs to mimic human-like driving behavior and interactions with other traffic participants.

By integrating these algorithmic models with a combination of different sensors, CARLA setups can simulate realistic driving scenarios and evaluate the performance of NPCs in various complex traffic environments.

PROGRAM:

```python
#Generating NPC in the simulator:
"""Script to generate traffic in the simulation"""
import glob
import os
import sys
import time

try:
    sys.path.append(glob.glob('../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

import carla
from carla import VehicleLightState as vls
import argparse
import logging
from numpy import random

def get_actor_blueprints(world, filter, generation):
    bps = world.get_blueprint_library().filter(filter)
    if generation.lower() == "all":
        return bps

    # If the filter returns only one bp, we assume that this one needed
    # and therefore, we ignore the generation
    if len(bps) == 1:
        return bps
    try:
        int_generation = int(generation)
        # Check if generation is in available generations
        if int_generation in [1, 2]:
            bps = [x for x in bps if int(x.get_attribute('generation')) ==
int_generation]
            return bps
        else:
            print("   Warning! Actor Generation is not valid. No actor will be
spawned.")
            return []
    except:
        print("   Warning! Actor Generation is not valid. No actor will be
spawned.")
        return []

def main():
    argparser = argparse.ArgumentParser(
        description=__doc__)
```

```python
    argparser.add_argument(
        '--host',
        metavar='H',
        default='127.0.0.1',
        help='IP of the host server (default: 127.0.0.1)')
    argparser.add_argument(
        '-p', '--port',
        metavar='P',
        default=2000,
        type=int,
        help='TCP port to listen to (default: 2000)')
    argparser.add_argument(
        '-n', '--number-of-vehicles',
        metavar='N',
        default=30,
        type=int,
        help='Number of vehicles (default: 30)')
    argparser.add_argument(
        '-w', '--number-of-walkers',
        metavar='W',
        default=10,
        type=int,
        help='Number of walkers (default: 10)')
    argparser.add_argument(
        '--safe',
        action='store_true',
        help='Avoid spawning vehicles prone to accidents')
    argparser.add_argument(
        '--filterv',
        metavar='PATTERN',
        default='vehicle.*',
        help='Filter vehicle model (default: "vehicle.*")')
    argparser.add_argument(
        '--generationv',
        metavar='G',
        default='All',
        help='restrict to certain vehicle generation (values: "1","2","All" -
default: "All")')
```

```
    argparser.add_argument(
        '--filterw',
        metavar='PATTERN',
        default='walker.pedestrian.*',
        help='Filter pedestrian type (default: "walker.pedestrian.*")')
    argparser.add_argument(
        '--generationw',
        metavar='G',
        default='2',
        help='restrict to certain pedestrian generation (values: "1","2","All"
- default: "2")')
    argparser.add_argument(
        '--tm-port',
        metavar='P',
        default=8000,
        type=int,
        help='Port to communicate with TM (default: 8000)')
    argparser.add_argument(
        '--asynch',
        action='store_true',
        help='Activate asynchronous mode execution')
    argparser.add_argument(
        '--hybrid',
        action='store_true',
        help='Activate hybrid mode for Traffic Manager')
    argparser.add_argument(
        '-s', '--seed',
        metavar='S',
        type=int,
        help='Set random device seed and deterministic mode for Traffic
Manager')
    argparser.add_argument(
        '--seedw',
        metavar='S',
        default=0,
        type=int,
        help='Set the seed for pedestrians module')
    argparser.add_argument(
        '--car-lights-on',
        action='store_true',
        default=False,
        help='Enable automatic car light management')
    argparser.add_argument(
        '--hero',
        action='store_true',
        default=False,
        help='Set one of the vehicles as hero')
    argparser.add_argument(
        '--respawn',
        action='store_true',
        default=False,
        help='Automatically respawn dormant vehicles (only in large maps)')
    argparser.add_argument(
        '--no-rendering',
        action='store_true',
        default=False,
        help='Activate no rendering mode')
    args = argparser.parse_args()
    logging.basicConfig(format='%(levelname)s: %(message)s',
level=logging.INFO)

    vehicles_list = []
    walkers_list = []
    all_id = []
    client = carla.Client(args.host, args.port)
    client.set_timeout(10.0)
    synchronous_master = False
    random.seed(args.seed if args.seed is not None else int(time.time()))

    try:
        world = client.get_world()
        traffic_manager = client.get_trafficmanager(args.tm_port)
        traffic_manager.set_global_distance_to_leading_vehicle(2.5)
        if args.respawn:
            traffic_manager.set_respawn_dormant_vehicles(True)
        if args.hybrid:
            traffic_manager.set_hybrid_physics_mode(True)
            traffic_manager.set_hybrid_physics_radius(70.0)
        if args.seed is not None:
            traffic_manager.set_random_device_seed(args.seed)
        settings = world.get_settings()
        if not args.asynch:
            traffic_manager.set_synchronous_mode(True)
            if not settings.synchronous_mode:
                synchronous_master = True
                settings.synchronous_mode = True
                settings.fixed_delta_seconds = 0.05
            else:
                synchronous_master = False
        else:
            print("You are currently in asynchronous mode. If this is a
traffic simulation, \
            you could experience some issues. If it's not working correctly,
switch to synchronous \
            mode by using traffic_manager.set_synchronous_mode(True)")
        if args.no_rendering:
            settings.no_rendering_mode = True
        world.apply_settings(settings)
        blueprints = get_actor_blueprints(world, args.filterv,
args.generationv)
        blueprintsWalkers = get_actor_blueprints(world, args.filterw,
args.generationw)

        if args.safe:
            blueprints = [x for x in blueprints if
x.get_attribute('base_type') == 'car']
        blueprints = sorted(blueprints, key=lambda bp: bp.id)
        spawn_points = world.get_map().get_spawn_points()
        number_of_spawn_points = len(spawn_points)
        if args.number_of_vehicles < number_of_spawn_points:
            random.shuffle(spawn_points)
        elif args.number_of_vehicles > number_of_spawn_points:
            msg = 'requested %d vehicles, but could only find %d spawn points'
            logging.warning(msg, args.number_of_vehicles,
number_of_spawn_points)
```

```
            args.number_of_vehicles = number_of_spawn_points

        # @todo cannot import these directly.
        SpawnActor = carla.command.SpawnActor
        SetAutopilot = carla.command.SetAutopilot
        FutureActor = carla.command.FutureActor

        # --------------
        # Spawn vehicles
        # --------------
        batch = []
        hero = args.hero
        for n, transform in enumerate(spawn_points):
            if n >= args.number_of_vehicles:
                break
            blueprint = random.choice(blueprints)
            if blueprint.has_attribute('color'):
                color =
random.choice(blueprint.get_attribute('color').recommended_values)
                blueprint.set_attribute('color', color)
            if blueprint.has_attribute('driver_id'):
                driver_id =
random.choice(blueprint.get_attribute('driver_id').recommended_values)
                blueprint.set_attribute('driver_id', driver_id)
            if hero:
                blueprint.set_attribute('role_name', 'hero')
                hero = False
            else:
                blueprint.set_attribute('role_name', 'autopilot')

            # spawn the cars and set their autopilot and light state all
together
            batch.append(SpawnActor(blueprint, transform)
                .then(SetAutopilot(FutureActor, True,
traffic_manager.get_port())))
        for response in client.apply_batch_sync(batch, synchronous_master):
            if response.error:
                logging.error(response.error)
            else:
                vehicles_list.append(response.actor_id)
        # Set automatic vehicle lights update if specified
        if args.car_lights_on:
            all_vehicle_actors = world.get_actors(vehicles_list)
            for actor in all_vehicle_actors:
                traffic_manager.update_vehicle_lights(actor, True)

        # -------------
        # Spawn Walkers
        # -------------
        # some settings
        percentagePedestriansRunning = 0.0      # how many pedestrians will
run
        percentagePedestriansCrossing = 0.0     # how many pedestrians will
walk through the road
        if args.seedw:
            world.set_pedestrians_seed(args.seedw)
            random.seed(args.seedw)
        # 1. take all the random locations to spawn
        spawn_points = []
        for i in range(args.number_of_walkers):
            spawn_point = carla.Transform()
            loc = world.get_random_location_from_navigation()
            if (loc != None):
                spawn_point.location = loc
                spawn_points.append(spawn_point)
        # 2. we spawn the walker object
        batch = []
        walker_speed = []
        for spawn_point in spawn_points:
            walker_bp = random.choice(blueprintsWalkers)
            # set as not invincible
            if walker_bp.has_attribute('is_invincible'):
                walker_bp.set_attribute('is_invincible', 'false')
            # set the max speed
            if walker_bp.has_attribute('speed'):
                if (random.random() > percentagePedestriansRunning):
                    # walking
```

```python
                    walker_speed.append(walker_bp.get_attribute('speed').recom
mended_values[1])
                else:
                    # running
                    walker_speed.append(walker_bp.get_attribute('speed').recom
mended_values[2])
            else:
                print("Walker has no speed")
                walker_speed.append(0.0)
            batch.append(SpawnActor(walker_bp, spawn_point))
        results = client.apply_batch_sync(batch, True)
        walker_speed2 = []
        for i in range(len(results)):
            if results[i].error:
                logging.error(results[i].error)
            else:
                walkers_list.append({"id": results[i].actor_id})
                walker_speed2.append(walker_speed[i])
        walker_speed = walker_speed2
        # 3. we spawn the walker controller
        batch = []
        walker_controller_bp =
world.get_blueprint_library().find('controller.ai.walker')
        for i in range(len(walkers_list)):
            batch.append(SpawnActor(walker_controller_bp, carla.Transform(),
walkers_list[i]["id"]))
        results = client.apply_batch_sync(batch, True)
        for i in range(len(results)):
            if results[i].error:
                logging.error(results[i].error)
            else:
                walkers_list[i]["con"] = results[i].actor_id
        # 4. we put together the walkers and controllers id to get the objects
from their id
        for i in range(len(walkers_list)):
            all_id.append(walkers_list[i]["con"])
            all_id.append(walkers_list[i]["id"])
        all_actors = world.get_actors(all_id)
        # wait for a tick to ensure client receives the last transform of the
walkers we have just created
        if args.asynch or not synchronous_master:
            world.wait_for_tick()
        else:
            world.tick()
        # 5. initialize each controller and set target to walk to (list is
[controler, actor, controller, actor ...])
        # set how many pedestrians can cross the road
        world.set_pedestrians_cross_factor(percentagePedestriansCrossing)
        for i in range(0, len(all_id), 2):
            # start walker
            all_actors[i].start()
            # set walk to random point
            all_actors[i].go_to_location(world.get_random_location_from_naviga
tion())
            # max speed
            all_actors[i].set_max_speed(float(walker_speed[int(i/2)]))
        print('spawned %d vehicles and %d walkers, press Ctrl+C to exit.' %
(len(vehicles_list), len(walkers_list)))
        # Example of how to use Traffic Manager parameters
        traffic_manager.global_percentage_speed_difference(30.0)
        while True:
            if not args.asynch and synchronous_master:
                world.tick()
            else:
                world.wait_for_tick()
    finally:
        if not args.asynch and synchronous_master:
            settings = world.get_settings()
            settings.synchronous_mode = False
            settings.no_rendering_mode = False
            settings.fixed_delta_seconds = None
            world.apply_settings(settings)
        print('\ndestroying %d vehicles' % len(vehicles_list))
        client.apply_batch([carla.command.DestroyActor(x) for x in
vehicles_list])
        # stop walker controllers (list is [controller, actor, controller,
actor ...])
```

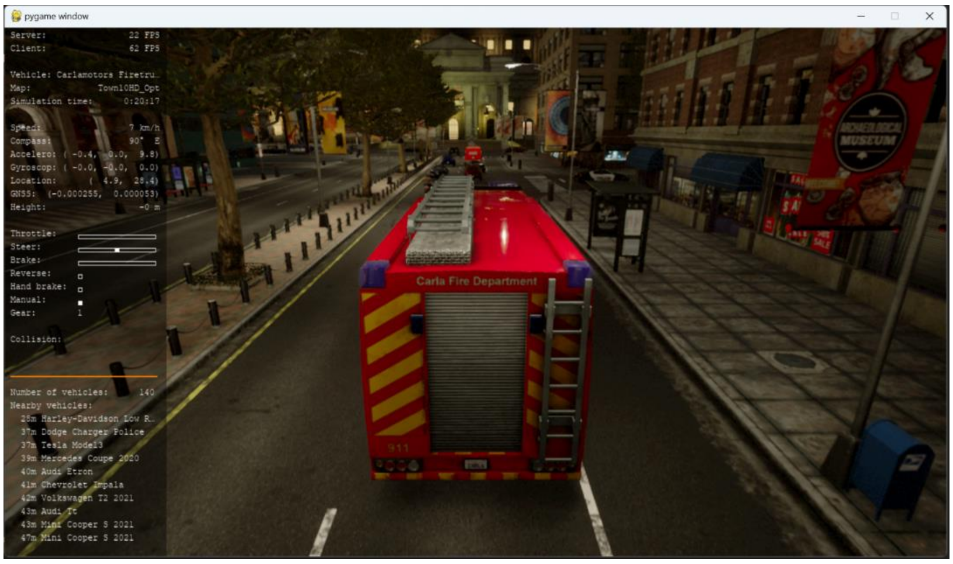
```
        for i in range(0, len(all_id), 2):
            all_actors[i].stop()
        print('\ndestroying %d walkers' % len(walkers_list))
        client.apply_batch([carla.command.DestroyActor(x) for x in all_id])
        time.sleep(0.5)
if __name__ == '__main__':
    try:
        main()
    except KeyboardInterrupt:
        pass
    finally:
        print('\ndone.')
```

SIMULATIONS

Default Spectator View:

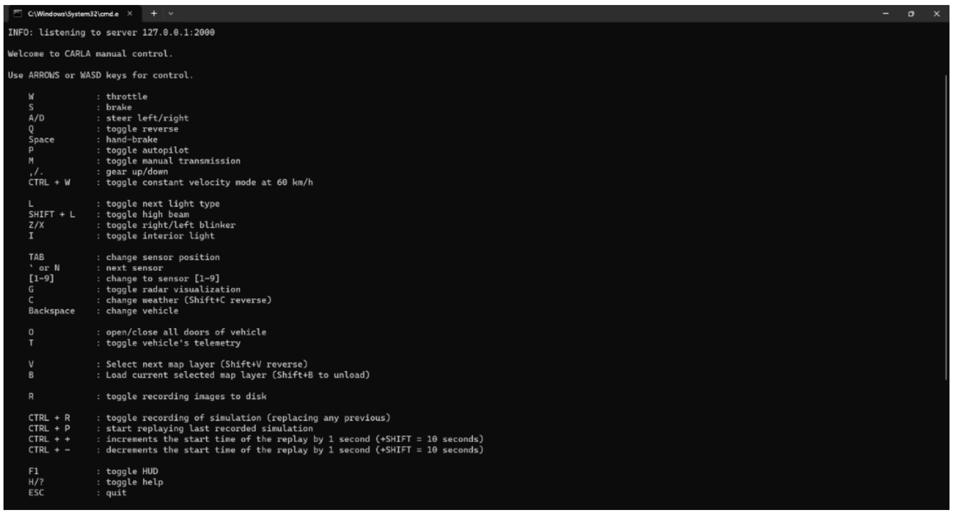
Spectator view after importing NPC in the simulator:

Manual Control by the User:



```
Server:                22 FPS
Client:                62 FPS

Vehicle: Carlamotors Firetru…
Map:              Town10HD_Opt
Simulation time:       0:20:17

Speed:                 7 km/h
Compass:                90° E
Accelero: ( -0.4,  0.0,  9.8)
Gyroscop: ( -0.0, -0.0,  0.0)
Location:      (  4.9,  28.4)
GNSS:   (-0.000255,  0.000053)
Height:                  -0 m

Throttle:
Steer:
Brake:
Reverse:
Hand brake:
Manual:
Gear:        1

Collision:

Number of vehicles:       140
Nearby vehicles:
  28m Harley-Davidson Low R…
  37m Dodge Charger Police
  37m Tesla Model3
  39m Mercedes Coupe 2020
  40m Audi Etron
  41m Chevrolet Impala
  42m Volkswagen T2 2021
  43m Audi Tt
  43m Mini Cooper S 2021
  47m Mini Cooper S 2021
```

Instruction for the manual controls:



```
INFO: listening to server 127.0.0.1:2000

Welcome to CARLA manual control.

Use ARROWS or WASD keys for control.

    W            : throttle
    S            : brake
    A/D          : steer left/right
    Q            : toggle reverse
    Space        : hand-brake
    P            : toggle autopilot
    M            : toggle manual transmission
    ,/.          : gear up/down
    CTRL + W     : toggle constant velocity mode at 60 km/h

    L            : toggle next light type
    SHIFT + L    : toggle high beam
    Z/X          : toggle right/left blinker
    I            : toggle interior light

    TAB          : change sensor position
    ` or N       : next sensor
    [1-9]        : change to sensor [1-9]
    G            : toggle radar visualization
    C            : change weather (Shift+C reverse)
    Backspace    : change vehicle

    O            : open/close all doors of vehicle
    T            : toggle vehicle's telemetry

    V            : Select next map layer (Shift+V reverse)
    B            : Load current selected map layer (Shift+B to unload)

    R            : toggle recording images to disk

    CTRL + R     : toggle recording of simulation (replacing any previous)
    CTRL + P     : start replaying last recorded simulation
    CTRL + +     : increments the start time of the replay by 1 second (+SHIFT = 10 seconds)
    CTRL + -     : decrements the start time of the replay by 1 second (+SHIFT = 10 seconds)

    F1           : toggle HUD
    H/?          : toggle help
    ESC          : quit
```