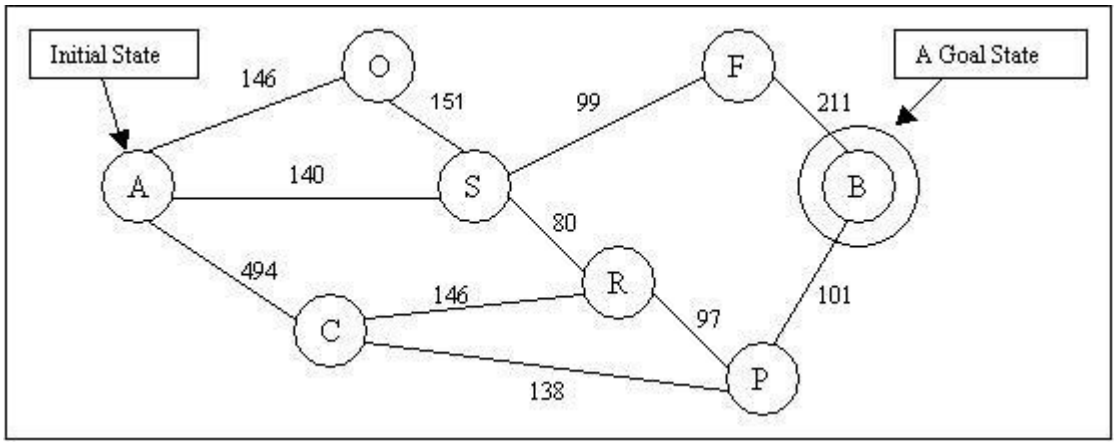
Lab: 05

Task 01: Solve Greedy Best-First Search of the following graph:



Solution:

```
from queue import PriorityQueue
v = 12
graph = [[] for i in range(v)]
def best_first_search(source, target, n,node_target):
    visited = [0] * n
    p = PriorityQueue()
    p.put((0, source))
    while p.empty() == False:
        vertex = p.get()[1]
        print(vertex, end=" ")
        node_target.append(vertex)
        if vertex == target:
            break
        for v, c in graph[vertex]:
            if visited[v] == False:
                visited[v] = True
                p.put((c, v))
    print()
def addedge(x, y, cost):
    graph[x].append((y, cost))
    graph[y].append((x, cost))
addedge(0,1,146) #A TO O
addedge(0,3,140) #A TO S
addedge(0,2,494) #A TO C
addedge(1,3,151) #O TO S
addedge(2,4,146) #C TO R
addedge(2,6,138) #C TO P
addedge(3,4,80) #S TO R
addedge(3,5,99) #S TO F
addedge(4,6,97) #R TO P
addedge(5,7,211) #F TO B
addedge(6,7,101) #P TO B

source = 0 #A Start state
target = 7 #B Goal state
node_target=[]
best_first_search(source, target, v,node_target)
nodes=['A','O','C','S','R','F','P','B']
path=[]
for i in range(len(node_target)):
    path.append(nodes[node_target[i]])
print("Path Taken = {}".format(path))
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL                Python +
0 3 4 6 5 7
Path Taken = ['A', 'S', 'R', 'P', 'F', 'B']
PS C:\Users\Saad>
```

Task 02: Develop code to implement the A* algorithm in order to find the optimal path in the Travel in Romania problem. Use the heuristic given in the text above. Furthermore, propose an admissible heuristic of your own and compare the two heuristics utilized. Suggest which of the two heuristics is a better choice for the travel in Romania problem and why?

```python
class Node():
    """A node class for A* Pathfinding"""

    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
        self.g = 0
        self.h = 0
        self.f = 0

    def __eq__(self, other):
        return self.position == other.position


def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start to the given end in the given maze"""

    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0

    # Initialize both open and closed list
    open_list = []
    closed_list = []

    # Add the start node
    open_list.append(start_node)

    # Loop until you find the end
    while len(open_list) > 0:

        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index

        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)

        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path

        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]: # Adjacent squares

            # Get node position
            node_position = (current_node.position[0] + new_position[0], current_node.position[1] + new_position[1])

            # Make sure within range
            if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or node_position[1] > (len(maze[len(maze)-1]) -1) or node_position[1] < 0:
                continue

            # Make sure walkable terrain
            if maze[node_position[0]][node_position[1]] != 0:
                continue

            # Create new node
            new_node = Node(current_node, node_position)
```

```
            # Append
            children.append(new_node)

        # Loop through children
        for child in children:

            # Child is on the closed list
            for closed_child in closed_list:
                if child == closed_child:
                    continue

            # Create the f, g, and h values
            child.g = current_node.g + 1
            child.h = ((child.position[0] - end_node.position[0]) ** 2) + ((child.position[1] - end_node.position[1]) ** 2)
            child.f = child.g + child.h

            # Child is already in the open list
            for open_node in open_list:
                if child == open_node and child.g > open_node.g:
                    continue

            # Add the child to the open list
            open_list.append(child)


def main():

    maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]

    start = (0, 0)
    end = (7, 6)

    path = astar(maze, start, end)
    print(path)


if __name__ == '__main__':
    main()
```

```
PS C:\Users\Saad> & C:/Users/Saad/AppData/Local/Programs/Python/Python39/python.exe c:/Users/Saad/Downloads/Task2_Lab5.py
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (6, 5), (7, 6)]
PS C:\Users\Saad>
```