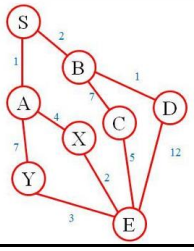
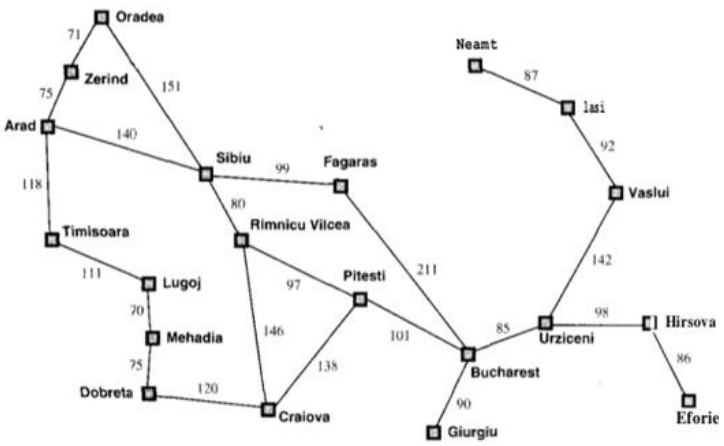
Bahria University,
Karachi Campus

LAB EXPERIMENT NO.
02

LIST OF TASKS

TASK NO	OBJECTIVE
1.	Implement A* Search and find the optimal path for finding the goal Start = S Goal = E 
2.	Arad To Neamt Using A* Algorithm  Straight-line distance to Bucharest Arad 366 Bucharest 0 Craiova 160 Dobreta 242 Eforie 161 Fagaras 178 Giurgiu 77 Hirsova 151 Iasi 226 Lugoj 244 Mehadia 241 Neamt 234 Oradea 380 Pitesti 98 Rimnicu Vilcea 193 Sibiu 253 Timisoara 329 Urziceni 80 Vaslui 199 Zerind 374

Submitted On:
4/5/2024

TASK NO 1:Develop a Python application to generate data visualizations Scenario:

```
from collections import deque
class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list
    def get_neighbors(self, v):
        return self.adjacency_list[v]
    def h(self, n):
        H = {'A': 5,'B': 6,'C': 4,'D': 15,'E': 0,'X': 5,'Y': 8}
        return H[n]
    def a_star_algorithm(self, start_node, stop_node):
        open_list = set([start_node])
        closed_list = set([])
        g = {}
        g[start_node] = 0
        parents = {}
        parents[start_node] = start_node
        while len(open_list) > 0:
            n = None
            for v in open_list:
                if n == None or g[v] + self.h(v) < g[n] + self.h(n):
                    n = v;
            if n == None:
                print('Path does not exist!')
                return None
            if n == stop_node:
                reconst_path = []
                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]
                reconst_path.append(start_node)
                reconst_path.reverse()
                print('Path found: {}'.format(reconst_path))
                return reconst_path
            for (m, weight) in self.get_neighbors(n):
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n
                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)
            open_list.remove(n)
            closed_list.add(n)
        print('Path does not exist!')
        return None
adjacency_list = {
    'S': [('A', 1), ('B', 2)],
    'A': [('Y', 7), ('B', 2)],
    'B': [('C', 7),('D', 1), ('S', 2)],
    'C': [('E', 5),('B', 7)],'E': [('D', 12),('C',5),('X',2),('Y',3)],X': [('A',
4),('E',2)],'Y': [('A', 7),('E',3)],}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('S', 'E')
print("Cost = 14 ")
```

```
Path found: ['S', 'B', 'C', 'E']
Cost = 14
```

TASK NO 2:Implement a text summarization model using Transformers Scenario:

```
from collections import deque
class Graph:
    def __init__(self, adjacency_list):
        self.adjacency_list = adjacency_list
    def get_neighbors(self, v):
        return self.adjacency_list[v]
    def h(self, n):
        H = {'Arad': 366, 'Zerind': 374, 'Oradea': 380, 'Sibiu': 253, 'Timisoara': 329,
'Lugoj': 244, 'Mehadia': 241, 'Drobeta': 242, 'Craiova': 160, 'Rimnicu Vilcea':
193, 'Fagaras': 178, 'Pitesti': 98, 'Bucharest':226, 'Neamt': 234}
        return H[n]
    def a_star_algorithm(self, start_node, stop_node):
        open_list = set([start_node])
        closed_list = set([])
        g = {}
        g[start_node] = 0
        parents = {}
        parents[start_node] = start_node
        while len(open_list) > 0:
            n = None
            for v in open_list:
            if n == stop_node:
                reconst_path = []
                while parents[n] != n:
                    reconst_path.append(n)
                    n = parents[n]
                reconst_path.append(start_node)
                reconst_path.reverse()
                print('Path found: {}'.format(reconst_path))
                return reconst_path
            for (m, weight) in self.get_neighbors(n):
                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    parents[m] = n
                    g[m] = g[n] + weight
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n
                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)
            open_list.remove(n)
            closed_list.add(n)
        print('Path does not exist!')
        return None
adjacency_list = {
    'Arad': {'Zerind': 75, 'Sibiu': 140, 'Timisoara': 118},
    'Zerind': {'Arad': 75, 'Oradea': 71},
    'Oradea': {'Zerind': 71, 'Sibiu': 151},
    'Sibiu': {'Arad': 140, 'Oradea': 151, 'Fagaras': 99, 'Rimnicu Vilcea':
80},
    'Timisoara': {'Arad': 118, 'Lugoj': 111},
    'Lugoj': {'Timisoara': 111, 'Mehadia': 70},
    'Mehadia': {'Lugoj': 70, 'Drobeta': 75},
    'Drobeta': {'Mehadia': 75, 'Craiova': 120},
    'Craiova': {'Drobeta': 120, 'Rimnicu Vilcea': 146, 'Pitesti': 138},
    'Rimnicu Vilcea': {'Sibiu': 80, 'Craiova': 146, 'Pitesti': 97},
    'Fagaras': {'Sibiu': 99, 'Bucharest': 211},
    'Pitesti': {'Rimnicu Vilcea': 97, 'Craiova': 138, 'Bucharest': 101},
    'Bucharest': 'Iasi': {'Vaslui': 92, 'Neamt': 87},'Neamt': {'Iasi': 87}
graph1 = Graph(adjacency_list)
graph1.a_star_algorithm('Arad', 'Neamt')
```

```
A* Path: ['Arad', 'Sibiu', 'Rimnicu_Vilcea', 'Pitesti', 'Bucharest','Urziceni','Vaslui','Iasi','Neamt']
Total Cost: 824
```