Experiment No. 4

Aim: To implement uniform cost search.

Requirements: Compatible version of python.

Theory:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

Uniform cost search algorithm:

1. Initialize visited, parent and Priority Queue data structure.

2. Enqueue source node into Priority Queue and add it in parent and set parent as Null.

3. While goal is not found repeat below steps.

4. Dequeue a node from Priority Queue and mark it as visited

5. Visit its neighbours and check if it is visited or not if it is visited continue else if it is already has some other parent check if current node has lower cost than current parent node if yes change the current parent node with current node.

6. If goal is found traverse parent data structure and reverse it.

Advantages:

o Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

o It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Implementation:

```
from queue import PriorityQueue

class Node:
    def __init__(self,Node_map,value = None, neighbours = {}):
        self.add_node(value = value, neighbours = neighbours,Node_map = Node_map)


    def add_node(self, Node_map, value = None ,neighbours = {}):
        assert value!=None, "Value cannot be none"
        if value in Node_map:
            Node_map[value].add_neighbours(Node_map = Node_map, neighbours =
neighbours)
            return

        self.value = value
        Node_map[value] = self
        self.neighbours = {}
        self.add_neighbours(neighbours = neighbours,Node_map = Node_map)

    def add_neighbours(self, Node_map, neighbours = {}):
        for neighbour, weight in neighbours.items():
            if neighbour not in Node_map:
                n = Node(value = neighbour, Node_map= Node_map)

            self.neighbours[neighbour] = weight

class Graph:
    def __init__(self):
        self.Node_map = {}

    def add_node(self, value = None, neighbours = {}):
        node = Node(value = value,neighbours = neighbours,Node_map = self.Node_map)

    def add_neighbours(self,value = None, neighbours = {}):
        assert value in self.Node_map, "Node doesn't exist"
```

```
        self.Node_map[value].add_neighbours(neighbours = neighbours,Node_map =
self.Node_map)

    def print_graph(self):
        for key, value in self.Node_map.items():
            print(f"key: {key}, value: {value.neighbours}")

    def uniform_search(self,source = None, goal = None):
        assert source != None and goal != None, "Source and goal node cannot be None"
        assert source in self.Node_map and goal in self.Node_map, "Source or Goal node
doesn't exist"

        parent = {source: (None,0)}
        visited = set()
        min_queue = PriorityQueue()
        min_queue.put(source)

        while(True):
            node = min_queue.get()
            if node == goal:
                break
            visited.add(min_queue)

            for neighbour, weight in self.Node_map[node].neighbours.items():
                if neighbour in visited:
                    continue
                min_queue.put(neighbour)
                if neighbour in parent:
                    if weight < self.Node_map[parent[neighbour][0]].neighbours[neighbour]:
                        parent[neighbour] = (node,weight)
                else:
                    parent[neighbour] = (node,weight)

        node  = goal
        ls = []

        total = 0
        while(parent[node][0]):
            total += parent[node][1]
            ls.append((node,parent[node][1]))
            node = parent[node][0]
        ls.append((source,0))
        ls.reverse()
        for x in ls:
            print(f"Node={x[0]}, Weight={x[1]}-->",end="")
        print("None")
        print(f"Total weight= {total}")

g = Graph()
g.add_node("S",{"A":1, "G": 12})
```

```
g.add_node("A",{"B":3, "C":1})
g.add_node("B",{"D":3})
g.add_node("C",{"D":1,"G":2})
g.add_node("D",{"G":3})
g.print_graph()
g.uniform_search(source="S", goal="G")
```

Output:

```
Node: S, neighbours: {'A': 1, 'G': 12}
Node: A, neighbours: {'B': 3, 'C': 1}
Node: G, neighbours: {}
Node: B, neighbours: {'D': 3}
Node: C, neighbours: {'D': 1, 'G': 2}
Node: D, neighbours: {'G': 3}
Node=S, Weight=0-->Node=A, Weight=1-->Node=C, Weight=1-->Node=G, Weight=2-->None
Total weight= 4
```

Conclusion: We have successfully implemented uniform cost search in python.