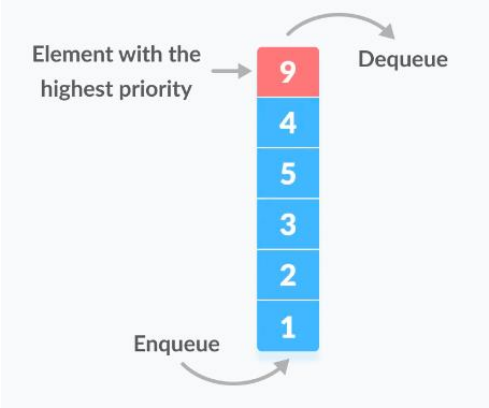
Lab 7

CLO: 01, 02

Priority Queue:

A priority queue is a special type of queue in which each element is associated with a priority value. And, elements are served on the basis of their priority. That is, higher priority elements are served first. However, if elements with the same priority occur, they are served according to their order in the queue.



Question 2:

Implement **Priority Queue** using Python for weighted graph:

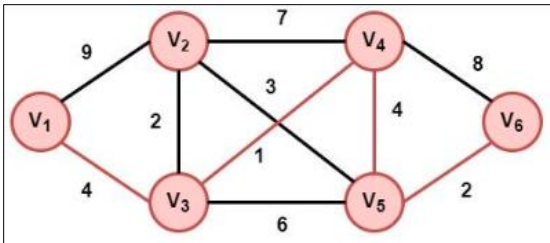
```
class WeightedGraph:
    def __init__(self):
        self.graph = {}

    def add_edge(self, start, end, weight):
        if start not in self.graph:
            self.graph[start] = []
        self.graph[start].append((end, weight))

    def get_neighbors(self, node):
        return self.graph.get(node, [])

def priority_queue_for_graph(graph):
    pq = queue.PriorityQueue()
    pq.put(('A', 0))  # Starting node and its priority

    while not pq.empty():
        current_node, current_priority = pq.get()
        print(f"Processing {current_node} with priority {current_priority}")

        for neighbor, weight in graph.get_neighbors(current_node):
            next_priority = current_priority + weight
            pq.put((neighbor, next_priority))
```

```python
# Create the weighted graph
g = WeightedGraph()
g.add_edge('A', 'B', 5)
g.add_edge('A', 'C', 3)
g.add_edge('B', 'C', 2)
g.add_edge('B', 'D', 4)
g.add_edge('C', 'D', 6)

# Use a priority queue to process the nodes
priority_queue_for_graph(g)
```

```
Processing A with priority 0
Processing B with priority 5
Processing C with priority 3
Processing C with priority 7
Processing D with priority 9
Processing D with priority 9
Processing D with priority 13
```

Question 3:

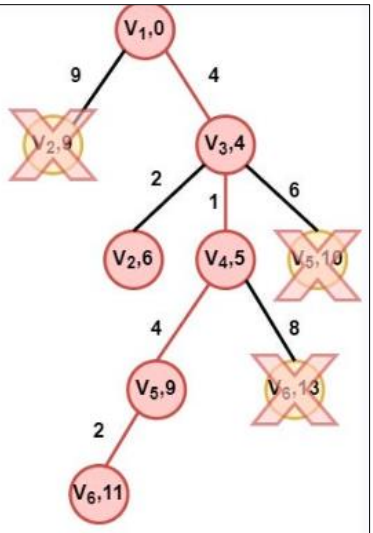
Uniform Cost Search (UCS) Algorithm: Uniform-cost search is an uninformed search algorithm that uses the lowest cumulative cost to find a path from the source to the destination. Nodes



Pace	Opened	Closed
0	{(V_1,0)}	{-}
1	{(V_2,9),(V_3,4)}	{(V_1,0)}
2	{(V_2,6),(V_4,5),(V_5,10)}	{(V_1,0),(V_3,4)}
3	{(V_2,6),(V_6,13),(V_5,9)}	{(V_1,0),(V_3,4),(V_4,5)}
4	{(V_6,13),(V_5,9)}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6)}
5	{(V_6,11)}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6),(V_5,9)}
6	{-}	{(V_1,0),(V_3,4),(V_4,5),(V_2,6),(V_5,9)}

Path: V_1 - V_3 - V_4 - V_5 - V_6

Total Cost: 11



are expanded, starting from the root, according to the minimum cumulative cost. The uniform-cost search is then implemented using a Priority Queue.

Question 4:

Implement Uniform Cost Search (UCS) algorithm, using Python

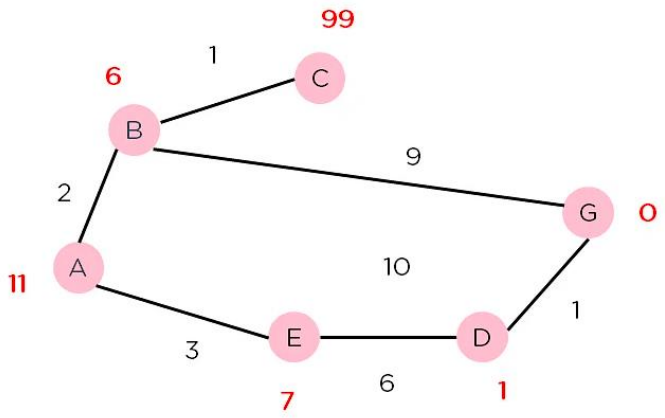
```
# Example usage
g = WeightedGraph()
g.add_edge('A', 'B', 1)
g.add_edge('A', 'C', 5)
g.add_edge('B', 'C', 2)
g.add_edge('B', 'D', 3)
g.add_edge('C', 'D', 1)
g.add_edge('C', 'E', 7)
g.add_edge('D', 'E', 2)

start_node = 'A'
goal_node = 'E'
```

Question 3:

A* Search:

A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes in a graph. It will be used for the shortest path finding. It is an extension of Dijkstra's shortest path algorithm (Dijkstra's Algorithm). The A* Search Algorithm also uses a heuristic function that provides additional information regarding how far away from the goal node we are. This function is used in conjunction with the f-heap data structure in order to make searching more efficient.

Example:



A* Search Algorithm

1. Initialize the open list
2. Initialize the closed list
   put the starting node on the open
   list (you can leave its f at zero)

3. while the open list is not empty
   a) find the node with the least f on
   the open list, call it "q"

   b) pop q off the open list

   c) generate q's 8 successors and set their
   parents to q

   d) for each successor
      i) if successor is the goal, stop search

      ii) else, compute both g and h for successor
      successor.g = q.g + distance between
      successor and q
      successor.h = distance from goal to
      successor (This can be done using many
      ways, we will discuss three heuristics-
      Manhattan, Diagonal and Euclidean
      Heuristics)

      successor.f = successor.g + successor.h

      iii) if a node with the same position as
      successor is in the OPEN list which has a
      lower f than successor, skip this successor

      iV) if a node with the same position as
      successor is in the CLOSED list which has
      a lower f than successor, skip this successor
      otherwise, add the node to the open list
   end (for loop)

   e) push q on the closed list
   end (while loop)

```python
#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
        H_dist = {
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0,

        }

        return H_dist[n]
```

```python
#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],

}
aStarAlgo('A', 'G')
```