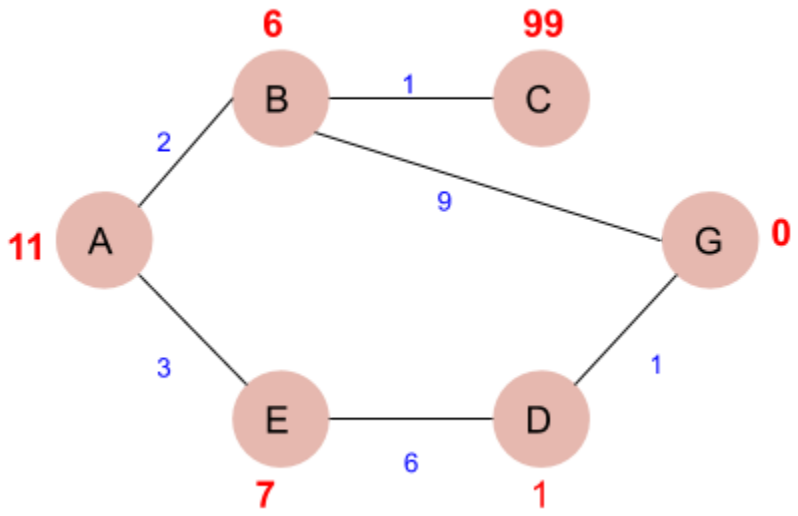
LAB 4

we are going to find out how A* algorithm can be used to find the most cost-effective path in a graph. Consider the following graph below



The numbers written on edges represent the distance between the nodes while the numbers written on nodes represent the heuristic values. Let us find the most cost-effective path to reach from start state A to final state G using A* Algorithm.

Let's start with node A.Since A is a starting node, therefore, the value of g(x) for A is zero and from the graph, we get the heuristic value of A is 11, therefore

- g(x) + h(x) = f(x)
- 0+ 11 =11
- Thus for A, we can write
- A=11

Now from A, we can go to point B or point E, so we compute f(x) for each of them

- A → B = 2 + 6 = 8

- A → E = 3 + 6 = 9

Since the cost for A → B is less, we move forward with this path and compute the f(x) for the children nodes of B

Since there is no path between C and G, the heuristic cost is set infinity or a very high value

- A → B → C = (2 + 1) + 99= 102
- A → B → G = (2 + 9) + 0 = 11

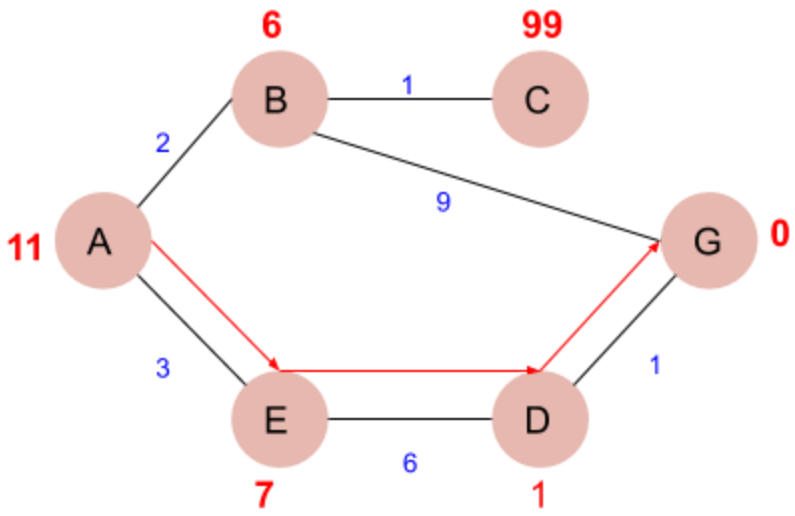
Here the path A → B → G has the least cost but it is still more than the cost of A → E, thus we explore this path further

A → E → D = (3 + 6) + 1 = 10

Comparing the cost of A → E → D with all the paths we got so far and as this cost is least of all we move forward with this path. And compute the f(x) for the children of D

A → E → D → G = (3 + 6 + 1) +0 =10

Now comparing all the paths that lead us to the goal, we conclude that A → E → D → G is the most cost-effective path to get from A to G.




```
def aStarAlgo(start_node, stop_node):


        open_set = set(start_node)

        closed_set = set()

        g = {} #store distance from starting node

        parents = {}# parents contains an adjacency map of all nodes



        #ditance of starting node from itself is zero

        g[start_node] = 0

        #start_node is root node i.e it has no parent nodes

        #so start_node is set to its own parent node

        parents[start_node] = start_node
```

```
        while len(open_set) > 0:
            n = None

            #node with lowest f() is found
            for v in open_set:
                if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                    n = v


            if n == stop_node or Graph_nodes[n] == None:
                pass
            else:
                for (m, weight) in get_neighbors(n):
                    #nodes 'm' not in first and last set are added to
first
                    #n is set its parent
                    if m not in open_set and m not in closed_set:
                        open_set.add(m)
                        parents[m] = n
```

```
                    g[m] = g[n] + weight



                #for each node m,compare its distance from start i.e
g(m) to the

                #from start through n node

                else:

                    if g[m] > g[n] + weight:

                        #update g(m)

                        g[m] = g[n] + weight

                        #change parent of m to n

                        parents[m] = n



                        #if m in closed set,remove and add to open

                        if m in closed_set:

                            closed_set.remove(m)

                            open_set.add(m)



        if n == None:

            print('Path does not exist!')

            return None
```

```
            # if the current node is the stop_node
            # then we begin reconstructin the path from it to the
start_node
            if n == stop_node:
                path = []

                while parents[n] != n:
                    path.append(n)
                    n = parents[n]

                path.append(start_node)

                path.reverse()

                print('Path found: {}'.format(path))
                return path


            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
```

```
            open_set.remove(n)
            closed_set.add(n)

        print('Path does not exist!')
        return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
#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]

#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')
```

.