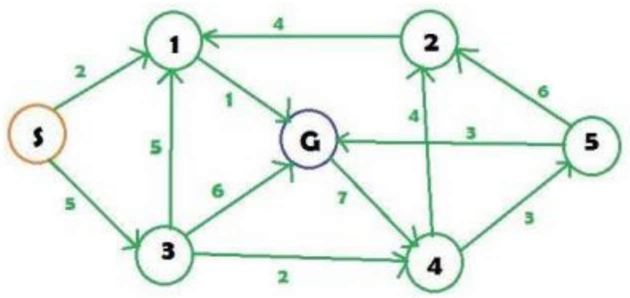
1. Apply the Uniform-Cost search approach on a given graph to find whether there exists a path between starting node 'S' and goal node 'G. Show the path along with the path cost.



Code:

```
# uniforn cost search
from queue import PriorityQueue


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

    def add_edge(self, from_node, to_node, weight):
        if from_node not in self.edges:
            self.edges[from_node] = []
        self.edges[from_node].append(to_node)
        self.weights[(from_node, to_node)] = weight

    def neighbors(self, node):
        return self.edges[node]

    def cost(self, from_node, to_node):
        return self.weights[(from_node, to_node)]


def uniform_cost_search(graph, start, goal):
    priorQueue = PriorityQueue()
    priorQueue.put(start, 0)
    open= {}
    open[start] = True
    came_from = {}
    cost_so_far = {}
    closed= {}

    came_from[start] = None
```

```
    cost_so_far[start] = 0

    while not priorQueue.empty():
        current = priorQueue.get()
        closed[current] = True
        if current in open:
            del open[current]
        if current == goal:
            break
        for next in graph.neighbors(current):
            new_cost = cost_so_far[current] + graph.cost(current, next)
            if next not in open and next not in closed:
                cost_so_far[next] = new_cost
                priorQueue.put(next, new_cost)
                open[next] = True
                came_from[next] = current
            elif new_cost < cost_so_far[next]:
                    cost_so_far[next] = new_cost
                    priorQueue.put(next, new_cost)
                    open[next] = True
                    came_from[next] = current

                    if next in closed:
                        del closed[next]


    return came_from, cost_so_far

def get_path(came_from, start, goal):
    current = goal
    path = []
    while current != start:
        if current==6:
            path.append('G')
        else:
            path.append(current)
        current = came_from[current]

    path.append('S')
    path.reverse()
    return path


def main():

    graph = Graph()
    graph.add_edge(0,1,2)
    graph.add_edge(0,3,5)
    graph.add_edge(1,6,1)
```

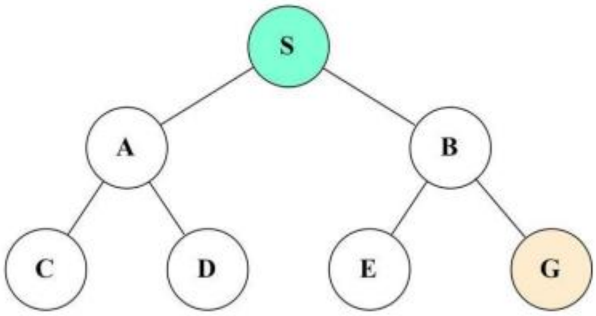
```python
    graph.add_edge(3, 4, 2)
    graph.add_edge(3,6,6)
    graph.add_edge(3, 1, 5)
    graph.add_edge(6,4,7)
    graph.add_edge(2, 1, 4)
    graph.add_edge(4, 5, 3)
    graph.add_edge(4, 2, 4)
    graph.add_edge(5, 2, 6)
    graph.add_edge(5,6,3)
    #  0 is the start node(S) and 6 is the goal node(G)
    came_from, cost_so_far = uniform_cost_search(graph, 0, 6)
    path = get_path(came_from, 0, 6)
    print(path)
    print("cost to reach G from S is: ", cost_so_far[6])


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

Output:

```
['S', 1, 'G']
cost to reach G from S is:  3
```

2. Apply the Iterative Deepening Depth First Search approach on a given graph to find whether there exists a path between starting node ‘S’ and goal node ‘G’.



Code:

```
# Iterative Deepening Depth First search
from queue import PriorityQueue
class Graph:
    def __init__(self):
        self.edges = {}

    def add_edge(self, from_node, to_node):
        if from_node not in self.edges:
            self.edges[from_node] = []
        self.edges[from_node].append(to_node)


    def neighbors(self, node):
        return self.edges[node]
def IDDFS(graph,src,goal,maxDepth):
    for i in range(maxDepth):
        visited = {}
        if DLS(graph,src,goal,i,visited):
            print("iteration ",i+1)
            return True
    return False

def DLS(graph,src,goal,maxDepth,visited):

    if src==goal:
        return True
    if maxDepth<=0:
        return False

    visited[src] = True
    for i in graph.neighbors(src):
        if i not in visited:
            if DLS(graph,i,goal,maxDepth-1,visited):
```

```python
                return True
    return False

def get_maxDepth():
    return 3
def get_path(came_from, start, goal):
    current = goal
    path = []
    while current != start:
        path.append(current)
        current = came_from[current]

    path.append(start)
    path.reverse()
    return path
def main():

    graph = Graph()
    graph.add_edge('S','B')
    graph.add_edge('A','C')
    graph.add_edge('A','D')
    graph.add_edge('S','A')
    graph.add_edge('B','E')
    graph.add_edge('B','G')
    # # and reverse edges are
    # graph.add_edge('B','S',0)
    # graph.add_edge('C','A',0)
    # graph.add_edge('D','A',0)
    # graph.add_edge('A','S',0)
    # graph.add_edge('E','B',0)
    # graph.add_edge('G','B',0)
    #  start node(S) and the goal node(G)
    start = 'S'
    goal = 'B'
    maxDepth = get_maxDepth()
    if IDDFS(graph,start,goal,maxDepth):
        print("Goal Found")
    else:
        print("Goal Not Found")

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

Output:

```
iteration  2
Goal Found
```