1. Write a program to implement Breadth First Search Traversal.

```
# BFS algorithm in Python
import collections

# BFS algorithm
def bfs(graph, root):

    visited, queue = set(), collections.deque([root])
    visited.add(root)

    while queue:

        # Dequeue a vertex from queue
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")

        # If not visited, mark it as visited, and
        # enqueue it
        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)



if __name__ == '__main__':
    graph = {0: [1, 2], 1: [2], 2: [3], 3: [1, 2]}
    print("Following is Breadth First Traversal: ")
    bfs(graph, 0)
```

2. Write a program to implement Depth First Search Traversal.

```python
# Python3 program to print DFS traversal
# from a given  graph
from collections import defaultdict


# This class represents a directed graph using
# adjacency list representation
class Graph:

    # Constructor
    def __init__(self):

        # Default dictionary to store graph
        self.graph = defaultdict(list)


    # Function to add an edge to graph
    def addEdge(self, u, v):
        self.graph[u].append(v)


    # A function used by DFS
    def DFSUtil(self, v, visited):

        # Mark the current node as visited
        # and print it
        visited.add(v)
        print(v, end=' ')

        # Recur for all the vertices
        # adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)


    # The function to do DFS traversal. It uses
    # recursive DFSUtil()
    def DFS(self, v):

        # Create a set to store visited vertices
        visited = set()

        # Call the recursive helper function
        # to print DFS traversal
```

```
        self.DFSUtil(v, visited)


# Driver's code
if __name__ == "__main__":
    g = Graph()
    g.addEdge(0, 1)
    g.addEdge(0, 2)
    g.addEdge(1, 2)
    g.addEdge(2, 0)
    g.addEdge(2, 3)
    g.addEdge(3, 3)

    print("Following is Depth First Traversal (starting from vertex 2)")

    # Function call
    g.DFS(2)
```

3. Write a program to implement A* .

```
from collections import deque

class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    # This is heuristic function which is having equal values for
all nodes
    def h(self, n):
        H = {
            'A': 1,
            'B': 1,
            'C': 1,
            'D': 1
        }

        return H[n]

    def a_star_algorithm(self, start, stop):
        # In this open_lst is a lisy of nodes which have been
visited, but who's
        # neighbours haven't all been always inspected, It starts
off with the start
  #node
```

```
        # And closed_lst is a list of nodes which have been visited
        # and who's neighbors have been always inspected
        open_lst = set([start])
        closed_lst = set([])

        # poo has present distances from start to all other nodes
        # the default value is +infinity
        poo = {}
        poo[start] = 0

        # par contains an adjac mapping of all nodes
        par = {}
        par[start] = start

        while len(open_lst) > 0:
            n = None

            # it will find a node with the lowest value of f() -
            for v in open_lst:
                if n == None or poo[v] + self.h(v) < poo[n] +
self.h(n):
                    n = v;

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop
            # then we start again from start
            if n == stop:
                reconst_path = []

                while par[n] != n:
                    reconst_path.append(n)
                    n = par[n]

                reconst_path.append(start)

                reconst_path.reverse()

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

            # for all the neighbors of the current node do
            for (m, weight) in self.get_neighbors(n):
              # if the current node is not presentin both open_lst
and closed_lst
                # add it to open_lst and note n as it's par
```

```
                if m not in open_lst and m not in closed_lst:
                    open_lst.add(m)
                    par[m] = n
                    poo[m] = poo[n] + weight

                # otherwise, check if it's quicker to first visit
n, then m
                # and if it is, update par data and poo data
                # and if the node was in the closed_lst, move it to
open_lst
                else:
                    if poo[m] > poo[n] + weight:
                        poo[m] = poo[n] + weight
                        par[m] = n

                        if m in closed_lst:
                            closed_lst.remove(m)
                            open_lst.add(m)

            # remove n from the open_lst, and add it to closed_lst
            # because all of his neighbors were inspected
            open_lst.remove(n)
            closed_lst.add(n)

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

Give Input as:

```
adjac_lis = {
    'A': [('B', 1), ('C', 3), ('D', 7)],
    'B': [('D', 5)],
    'C': [('D', 12)]
}
graph1 = Graph(adjac_lis)
graph1.a_star_algorithm('A', 'D')
```

4. Write a program to count the total number of goal states.

```
# Function to count the number of nodes
# with maximum connections
def get(graph):

    # Stores the number of connections
    # of each node
    v = [];
```

```
    # Stores the maximum connections
    mx = -1;
    for arr in graph.values():
        v.append(len(arr));
        mx = max(mx, (len(arr)));

    # Resultant count
    cnt = 0;
    for i in v:
        if (i == mx):
            cnt += 1

    print(cnt)

# Drive Code
graph = {}

nodes = 10
edges = 13;
for i in range(1, nodes + 1):
    graph[i] = []

# 1
graph[1].append(4);
graph[4].append(1);

# 2
graph[2].append(3);
graph[3].append(2);

# 3
graph[4].append(5);
graph[5].append(4);

# 4
graph[3].append(9);
graph[9].append(3);

# 5
graph[6].append(9);
graph[9].append(6);

# 6
graph[3].append(8);
graph[8].append(3);

# 7
```

```
graph[10].append(4);
graph[4].append(10);

# 8
graph[2].append(7);
graph[7].append(2);

# 9
graph[3].append(6);
graph[6].append(3);

# 10
graph[2].append(8);
graph[8].append(2);

# 11
graph[9].append(2);
graph[2].append(9);

# 12
graph[1].append(10);
graph[10].append(1);

# 13
graph[9].append(10);
graph[10].append(9);

get(graph);
```

5. Write a program to implement uniform cost search

```
def  uniform_cost_search(goal, start):

    # minimum cost upto
    # goal state from starting
    global graph,cost
    answer = []
```

```
# create a priority queue
queue = []

# set the answer vector to max value
for i in range(len(goal)):
    answer.append(10**8)

# insert the starting index
queue.append([0, start])

# map to store visited node
visited = {}

# count
count = 0

# while the queue is not empty
while (len(queue) > 0):

    # get the top element of the
    queue = sorted(queue)
    p = queue[-1]

    # pop the element
    del queue[-1]

    # get the original value
    p[0] *= -1

    # check if the element is part of
    # the goal list
    if (p[1] in goal):

        # get the position
        index = goal.index(p[1])

        # if a new goal is reached
        if (answer[index] == 10**8):
            count += 1

        # if the cost is less
        if (answer[index] > p[0]):
            answer[index] = p[0]

        # pop the element
        del queue[-1]
```

```
            queue = sorted(queue)
            if (count == len(goal)):
                return answer

        # check for the non visited nodes
        # which are adjacent to present node
        if (p[1] not in visited):
            for i in range(len(graph[p[1]])):

                # value is multiplied by -1 so that
                # least priority is at the top
                queue.append( [(p[0] + cost[(p[1],
graph[p[1]][i])])* -1, graph[p[1]][i]])

        # mark as visited
        visited[p[1]] = 1

    return answer

# main function
if __name__ == '__main__':

    # create the graph
    graph,cost = [[] for i in range(8)],{}

    # add edge
    graph[0].append(1)
    graph[0].append(3)
    graph[3].append(1)
    graph[3].append(6)
    graph[3].append(4)
    graph[1].append(6)
    graph[4].append(2)
    graph[4].append(5)
    graph[2].append(1)
    graph[5].append(2)
    graph[5].append(6)
    graph[6].append(4)

    # add the cost
    cost[(0, 1)] = 2
    cost[(0, 3)] = 5
    cost[(1, 6)] = 1
    cost[(3, 1)] = 5
    cost[(3, 6)] = 6
```

```
    cost[(3, 4)] = 2
    cost[(2, 1)] = 4
    cost[(4, 2)] = 4
    cost[(4, 5)] = 3
    cost[(5, 2)] = 6
    cost[(5, 6)] = 3
    cost[(6, 4)] = 7

    # goal state
    goal = []

    # set the goal
    # there can be multiple goal states
    goal.append(6)

    # get the answer
    answer = uniform_cost_search(goal, 0)

    # print the answer
    print("Minimum cost from 0 to 6 is = ",answer[0])
```

6. Write a program to implement a Water Jug Problem.

```
from collections import defaultdict

# jug1 and jug2 contain the value
# for max capacity in respective jugs
# and aim is the amount of water to be measured.
jug1, jug2, aim = 4, 3, 2

# Initialize dictionary with
# default value as false.
visited = defaultdict(lambda: False)

# Recursive function which prints the
# intermediate steps to reach the final
# solution and return boolean value
# (True if solution is possible, otherwise False).
# amt1 and amt2 are the amount of water present
# in both jugs at a certain point of time.
def waterJugSolver(amt1, amt2):

    # Checks for our goal and
    # returns true if achieved.
    if (amt1 == aim and amt2 == 0) or (amt2 == aim and amt1 == 0):
```

```
        print(amt1, amt2)
        return True

    # Checks if we have already visited the
    # combination or not. If not, then it proceeds further.
    if visited[(amt1, amt2)] == False:
        print(amt1, amt2)

        # Changes the boolean value of
        # the combination as it is visited.
        visited[(amt1, amt2)] = True

        # Check for all the 6 possibilities and
        # see if a solution is found in any one of them.
        return (waterJugSolver(0, amt2) or
                waterJugSolver(amt1, 0) or
                waterJugSolver(jug1, amt2) or
                waterJugSolver(amt1, jug2) or
                waterJugSolver(amt1 + min(amt2, (jug1-amt1)),
                amt2 - min(amt2, (jug1-amt1))) or
                waterJugSolver(amt1 - min(amt1, (jug2-amt2)),
                amt2 + min(amt1, (jug2-amt2))))

    # Return False if the combination is
    # already visited to avoid repetition otherwise
    # recursion will enter an infinite loop.
    else:
        return False

print("Steps: ")

# Call the function and pass the
# initial amount of water present in both jugs.
waterJugSolver(0, 0)
```