Lab Program 1a:

Aim:

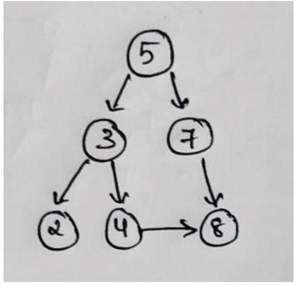
Implement Exhaustive search techniques using

a. BFS

Program:

```
def bfs(graph, start, goal):
    explored = []
    frontier = []
    frontier.append([start, None])
    parents = {start: None}
    while frontier:
        print("Frontier ", frontier)
        print("Explored ", explored)  # ,"\n")
        node, parent = frontier.pop(0)
        if node == goal:
            print("\nGoal node found")
            print("Path:", print_path(start, node, parents))
            break
        if node not in explored:
            explored.append(node)
            for neighbor in graph[node]:
                if neighbor not in explored:
                    frontier.append([neighbor, node])
                    parents[neighbor] = node
    if node != goal:
        print("Goal node not found")
def print_path(start, goal, parents):
    path = []
    while goal != start:
        path.append(goal)
        goal = parents[goal]
    path.append(start)
    return path[::-1]
if __name__ == '__main__':
    graph = eval(input("Enter Graph:"))
    s = eval(input("Enter Start node: "))
    g = eval(input("Enter goal node: "))
    print("\n")
    bfs(graph,s,g)
```

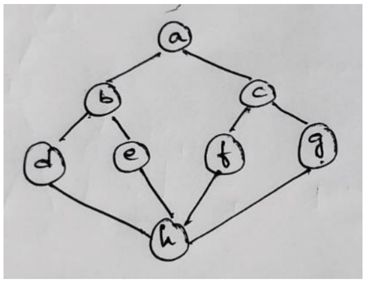
Graph 1:



Output 1:

Enter Graph:{'5': ['3','7'], '3': ['2', '4'], '7': ['8'], '2': [], '4': ['8'], '8': []}
Enter Start node: '5'
Enter goal node: '8'
F: [['5', None]] E: []
F: [['3', '5'], ['7', '5']]
E: ['5']
F: [['7', '5'], ['2', '3'], ['4', '3']]
E: ['5', '3']
F: [['2', '3'], ['4', '3'], ['8', '7']]
E: ['5', '3', '7']
F: [['4', '3'], ['8', '7']]
E: ['5', '3', '7', '2']
F: [['8', '7'], ['8', '4']]
E: ['5', '3', '7', '2', '4']
Goal node found
Path: ['5', '3', '4', '8']

Graph 2:



Output 2:

Enter Graph:{"a": ["b", "c"], "b": ["d", "e"], "c": ["f", "g"], "d": ["h"], "e": ["h"], "f": ["h"], "g": ["h"], "h": []}
Enter Start node: 'b'
Enter goal node: 'g'
F: [['b', None]] E: []
F: [['d', 'b'], ['e', 'b']]
E: ['b']
F: [['e', 'b'], ['h', 'd']]
E: ['b', 'd']
F: [['h', 'd'], ['h', 'e']]
E: ['b', 'd', 'e']
F: [['h', 'e']]
E: ['b', 'd', 'e', 'h'] Goal node not found

Lab Program 1b:

Aim:

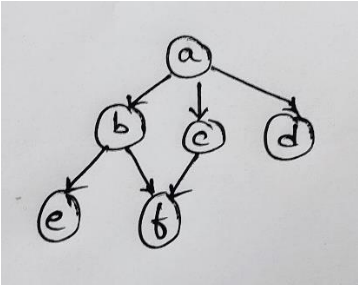
Implement Exhaustive search techniques using

b. DFS

Program:

```
def dfs(graph, start, goal):
    explored = []
    frontier = []
    frontier.append([start, None])
    parents = {start: None}
    while frontier:
        print("F: ", frontier)
        print("E: ", explored)  # ,"\n")
        node, parent = frontier.pop(0)
        print("Node: ", node, "  Parent: ", parent, end="\n\n")
        if node == goal:
            print("\nGoal node found")
            print("Path:", print_path(start, node, parents))
            break
        if node not in explored:
            explored.append(node)
            for neighbor in graph[node]:
                if neighbor not in explored:
                    frontier.append([neighbor, node])
                    parents[neighbor] = node
    if node != goal:
        print("Goal node not found")
def print_path(start, goal, parents):
    path = []
    while goal != start:
        path.append(goal)
        goal = parents[goal]
    path.append(start)
    return path[::-1]
if __name__ == '__main__':
    graph = eval(input())
    s = eval(input("Enter Start node: "))
    g = eval(input("Enter goal node: "))
    print("\n")
    dfs(graph, s, g)
```

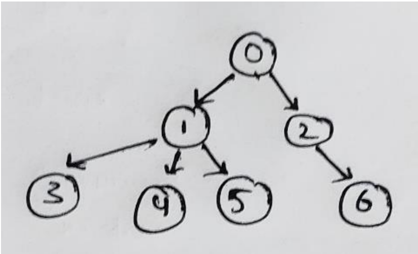
Graph 1:



Output 1:

{ "a" : ["b","c","d"],"b" : ["e", "f"],"c" : ["f"],"d" : [],"e":[]}
Enter Start node: "a"
Enter goal node: "f"
F: [['a', None]]
E: []
F: [['b', 'a'], ['c', 'a'], ['d', 'a']]
E: ['a']
F: [['c', 'a'], ['d', 'a'], ['e', 'b'], ['f', 'b']]
E: ['a', 'b']
F: [['d', 'a'], ['e', 'b'], ['f', 'b'], ['f', 'c']]
E: ['a', 'b', 'c']
F: [['e', 'b'], ['f', 'b'], ['f', 'c']]
E: ['a', 'b', 'c', 'd']
F: [['f', 'b'], ['f', 'c']]
E: ['a', 'b', 'c', 'd', 'e']
Goal node found
Path: ['a', 'c', 'f']

Graph 2:



Output 2:

{0:[1,2],1:[3,4,5],2:[5],3:[],4:[],5:[],6:[]}
Enter Start node: 1
Enter goal node: 6
F: [[1, None]]
E: []
F: [[3, 1], [4, 1], [5, 1]]
E: [1]
F: [[4, 1], [5, 1]]
E: [1, 3]
F: [[5, 1]]
E: [1, 3, 4]
Goal node not found

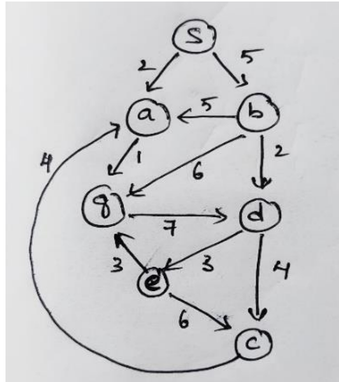
Lab Program 1c:
Aim:

Implement Exhaustive search techniques using
c.UCS

Program:

```
from queue import PriorityQueue

def ucs(graph, start, goal):
    explored = []
    frontier = PriorityQueue()
    frontier.put((0, start, None))
    while not frontier.empty():
        cost, node, parent = frontier.get()
        print("\nE:", explored)
        print("F:", [(item[1], item[0]) for item in frontier.queue])
        if node == goal:
            explored.append(node)
            print("\nE:", explored)
            print("F:", [(item[1], item[0]) for item in frontier.queue])
            print("\nGoal node found")
            break
        if node not in explored:
            explored.append(node)
            for neighbor, neighbor_cost in graph[node].items():
                if neighbor not in explored:
                    new_cost = cost + neighbor_cost
                    frontier.put((new_cost, neighbor, node))
                else:
                    # Check if the node is in frontier with higher cost
                    for item in list(frontier.queue):
                        if item[1] == neighbor and item[0] > new_cost:
                            frontier.queue.remove(item)
                            frontier.put((new_cost, neighbor, node))

    else:
        print("Goal node not found")

    print("\nMinimum cost:",cost)

if __name__ == '__main__':
    graph = eval(input("Enter graph:"))
    s=eval(input("Enter start node:"))
    g=eval(input("Enter goal node:"))
    ucs(graph, s, g)
```

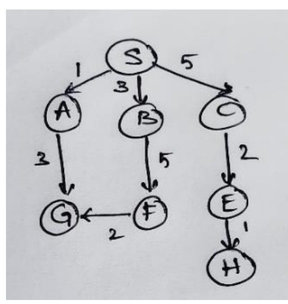
Graph 1:



Output 1:

Enter graph:{'s':{'a':2,'b':5}, 'a':{'g':1}, 'b':{'a':5,'g':6,'d':2}, 'g':{'d':7}, 'd':{'c':4,'e':3}, 'c':{'a':4}, 'e':{'c':6,'g':3}}
Enter start node:'s'
Enter goal node:'g'
E: ['s'] F: [('b', 5)]
E: ['s', 'a']
F: [('b', 5)]
E: ['s', 'a', 'g'] F: [('b', 5)]
Goal node found
Minimum cost: 3

Graph 2:



Output 2:

Enter graph:{'S': {'A': 1, 'B': 3, 'C': 5}, 'A': {'G': 3}, 'G': {}, 'B': {'F': 5}, 'F': {'G':2}, 'C': {'E': 2}, 'E': {'H': 1}, 'H': {}}
Enter start node:'S'
Enter goal node:'F'
E: ['S'] F: [('B', 3), ('C', 5)]
E: ['S', 'A']
F: [('G', 4), ('C', 5)]
E: ['S', 'A', 'B']
F: [('C', 5), ('F', 8)]
E: ['S', 'A', 'B', 'G']
F: [('F', 8)]
E: ['S', 'A', 'B', 'G', 'C'] F: [('F', 8)]
E: ['S', 'A', 'B', 'G', 'C', 'E'] F: [('H', 8)]
E: ['S', 'A', 'B', 'G', 'C', 'E', 'F'] F: [('H', 8)]
Goal node found
Minimum cost: 8

Lab Program 1d:

Aim:

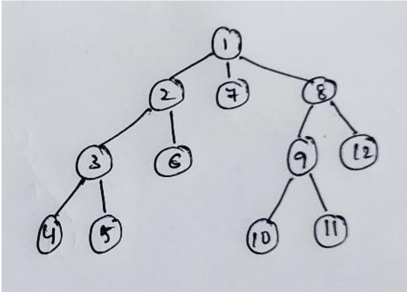
Implement Exhaustive search techniques using

d. IDDFS

Program:

```
class Node:
    def __init__(self, state, parent, depth):
        self.state = state
        self.parent = parent
        self.depth = depth
class DFIDS:
    def __init__(self, graph, max_depth):
        self.graph = graph
        self.max_depth = max_depth
    def search(self, start, goal):
        for depth in range(self.max_depth + 1):
            explored = []
            frontier = []
            frontier.append(Node(start, None, 0))
            while frontier:
                print('f:', [node.state for node in frontier], '\ne:', explored, '\n\n')
                cur_node = frontier.pop(0)
                print("Node:", cur_node.state, " Parent:", cur_node.parent, " Depth:", cur_node.depth)
                if cur_node.state == goal:
                    print("\nGoal node found")
                    return
                if cur_node.depth < self.max_depth and cur_node.state not in explored:
                    explored.append(cur_node.state)
                    for neighbor in self.graph[cur_node.state]:
                        if neighbor not in explored:
                            frontier.append(Node(neighbor, cur_node.state, cur_node.depth + 1))
        print("Goal node not found within depth limit")
if __name__ == '__main__':
    graph = eval(input("Enter the graph: "))
    start = input("Enter Start node: ")
    goal = input("Enter goal node: ")
    depth = int(input("Enter max Depth: "))
    print("\n")
    dfids = DFIDS(graph, depth)
    dfids.search(start, goal)
```

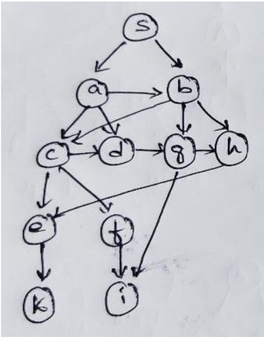
Graph 1:



Output 1:

Enter the graph: {'1':['2','7','8'], '2':['3','6'], '3':['4','5'], '4':[], '5':[], '6':[], '7':[], '8':['9','12'], '9':['10','11'], '10':[], '11':[], '12':[]}
Enter Start node: 1
Enter goal node: 9
Enter max Depth: 2
f: ['1'] e: []
f: ['2', '7', '8'] e: ['1']
f: ['7', '8', '3', '6'] e: ['1', '2']
f: ['8', '3', '6'] e: ['1', '2', '7']
f: ['3', '6', '9', '12'] e: ['1', '2', '7', '8']
f: ['6', '9', '12'] e: ['1', '2', '7', '8']
f: ['9', '12'] e: ['1', '2', '7', '8']
Goal node found

Graph 2:



Output 2:

Enter the graph: {'s': ['a', 'b'],'a': ['b', 'c', 'd'],'b': ['c', 'g', 'h'],'c': ['d', 'e', 'f'],'d':[],'e': ['k'],'f': ['i'],'g': ['h', 'i'],'h': ['e'],'i': ['k'],'k': []}
Enter Start node: s
Enter goal node: i
Enter max Depth: 3
f: ['s'] e: []
f: ['a', 'b'] e: ['s']
f: ['a', 'c', 'g', 'h']
e: ['s', 'b']
f: ['a', 'c', 'g', 'e']
e: ['s', 'b', 'h']
f: ['a', 'c', 'g']
e: ['s', 'b', 'h']
f: ['a', 'c', 'i']
e: ['s', 'b', 'h', 'g']
Goal node found

Lab Program 1e:

Aim:

Implement Exhaustive search techniques using

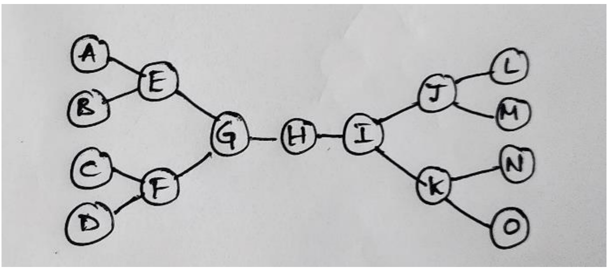
e. Bidirectional Search

Program:

```
def reverse_graph(graph):
    reversed_graph = {}
    for node in graph:
        reversed_graph[node] = []
    for node, neighbors in graph.items():
        for neighbor in neighbors:
            reversed_graph[neighbor].append(node)
    return reversed_graph
def bds(graph, rev_graph, start, goal):
    explored1 = []; explored2 = []
    frontier1 = [];  frontier2 = []
    frontier1.append([start, None])
    frontier2.append([goal, None])
    path1=[]
    path2=[]
    intersect=None
    while frontier1 and frontier2:
        # Exploration from start node
        node, parent = frontier1.pop(0)
        path1.append([node,parent])
        if node == intersect:
            break
        if node not in explored1:
            explored1.append(node)
            intersect=node
            for neighbor in graph[node]:
                if neighbor not in explored1:
                    frontier1.append([neighbor, node])
        # Exploration from goal node
        node2, parent2 = frontier2.pop(0)
        path2.append([node2,parent2])
        if parent2!=None and parent2 == intersect:
            print("\nGoal node found")
            break
        if node2 not in explored2:
            explored2.append(node2)
            intersect=node2
            for neighbor in rev_graph[node2]:
                if neighbor not in explored2:
                    frontier2.append([neighbor, node2])
    path2 = [[node2, node1] for node1, node2 in path2]
    print("Explored 1:", explored1)
    print("Frontier 1:", frontier1)
    print("Explored 2:", explored2)
    print("Frontier 2:", frontier2)
```

```
        print("Path 1:", path1)
        print("Path 2:", path2[::-1])
        print("The search intersected at: ",intersect)
    if __name__ == '__main__':
        graph = eval(input())
        rev_graph = reverse_graph(graph)
        s = input("Enter Start node: ")
        g = input("Enter goal node: ");print("\n")
        bds(graph, rev_graph, s, g)
```

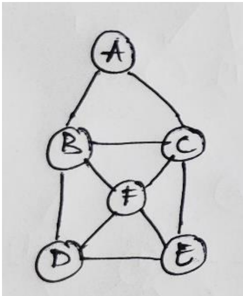
Graph 1:



Output 1:

```
{'A':['E'], 'E':['B','G'], 'B':[], 'G':['F','H'], 'F':['C','D'], 'C':[], 'D':[], 'H':['I'], 'I':['J','K'], 'J':['L','M'], 'L':[],
'M':[], 'K':['N','O'], 'N':[], 'O':[]}
Enter Start node: A
Enter goal node: E
Explored 1: ['A']
Frontier 1: []
Explored 2: ['E']
Frontier 2: [['A', 'E']]
Path 1: [['A', None], ['E', 'A']] Path 2: [[None, 'E']]
The search intersected at:  E
```

Graph 2:



Output 2:

```
{'A':['B','C'], 'B':['A','D','F','C'], 'C':['A','F','E','B'], 'F':['D','B','C','E'], 'D':['F','B','E'], 'E':['D','C','E']}
Enter Start node: A
Enter goal node: C
Goal node found
Explored 1: ['A', 'B', 'C']
Frontier 1: [['D', 'B'], ['F', 'B'], ['C', 'B'], ['F', 'C'], ['E', 'C']]
Explored 2: ['C', 'A']
Frontier 2: [['F', 'C'], ['E', 'C'], ['B', 'A']]
Path 1: [['A', None], ['B', 'A'], ['C', 'A']] Path 2: [['C', 'B'], ['C', 'A'], [None, 'C']]
The search intersected at:  C
```

Lab Program 2a:

Aim:

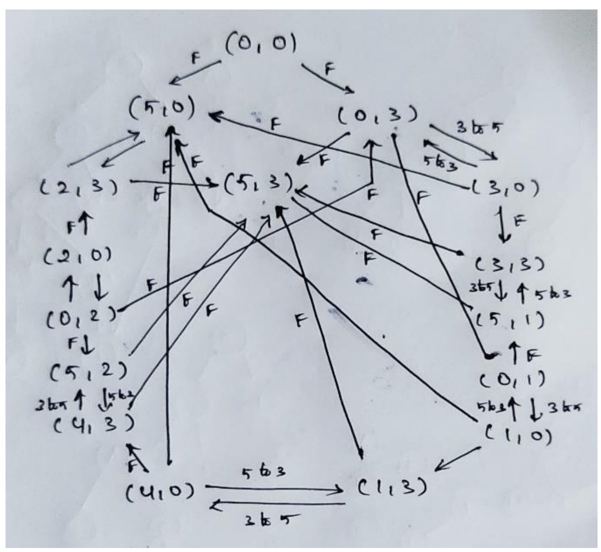
Implement water jug problem with Search tree generation using a. BFS

Program:

```
import copy
class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action
def SOLUTION(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return list(reversed(path))
def CHILD_NODE(problem, parent, action):
    child_state = copy.deepcopy(parent.state)
    if action in ('F1', 'F2'):
        index = int(action[1]) - 1
        if child_state[index] == 0:
            child_state[index] = problem.jug_size[index]
    elif action in ('E1', 'E2'):
        index = int(action[1]) - 1
        child_state[index] = 0
    elif action in ('1T2', '2T1'):
        from_jug, to_jug = int(action[0]),int(action[2])
        problem.POUR(from_jug, to_jug, child_state)
    return Node(child_state, parent, action)
def isPresent(node, frontier):
    return any(n.state == node.state for n in frontier)
def BREADTH_FIRST_SEARCH(problem):
    node = Node(problem.initial_state, None, None)
    if problem.GOAL_TEST(node.state):
        return SOLUTION(node)
    frontier = [node]
    explored = []
    while frontier:
        node = frontier.pop(0) #BFS
        explored.append(node.state)
        print("\nE:",explored)
        for action in problem.ACTIONS(node.state):
            child = CHILD_NODE(problem, node, action)
            if child.state and child.state not in explored and not isPresent(child, frontier):
                if problem.GOAL_TEST(child.state):
                    return SOLUTION(child)
                frontier.append(child)
        print("F:", [n.state for n in frontier])
    return "Failure"
class Problem:
```

```
    def __init__(self, j1, j2, g):
        self.initial_state = [0, 0]
        self.goal_state = [g, 'x']
        self.jug_size = [j1, j2]
    def ACTIONS(self, state): return ['F1', 'F2', 'E1', 'E2', '1T2', '2T1']
    def GOAL_TEST(self, state):  return state[0] == self.goal_state[0]
    def POUR(self, from_jug, to_jug, child_state):
        j1, j2 = from_jug - 1, to_jug - 1
        avail = child_state[j1]
        if child_state[j2] + avail >= self.jug_size[j2]:
            child_state[j1] -= self.jug_size[j2] - child_state[j2]
            child_state[j2] = self.jug_size[j2]
        else:
            child_state[j2] += avail
            child_state[j1] = 0
b = int(input("Enter Big jug capacity: "))
s = int(input("Enter Small jug capacity: "))
t = int(input("Enter target jug capacity: "))
problem = Problem(b,s,t)
solution = BREADTH_FIRST_SEARCH(problem)
print("Solution:", solution)
```

State Space Tree 1:

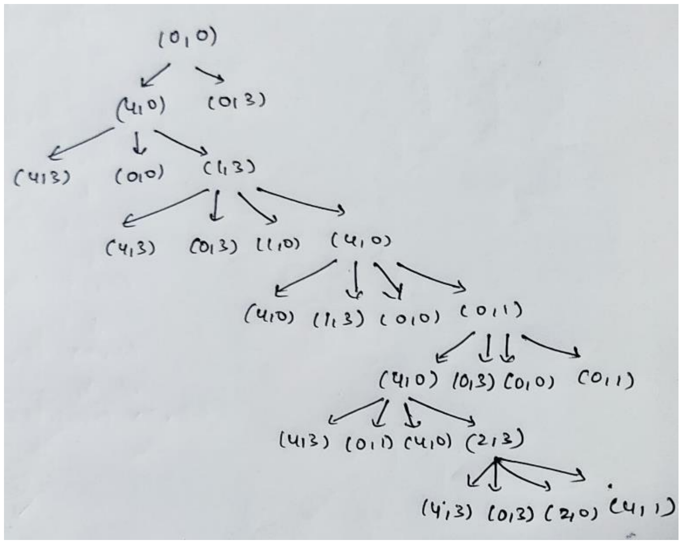


Output 1:

```
Enter Big jug capacity: 5
Enter Small jug capacity: 3
Enter target jug capacity: 4
E: [[0, 0]]
F: [[5, 0], [0, 3]]
E: [[0, 0], [5, 0]]
F: [[0, 3], [5, 3], [2, 3]]
E: [[0, 0], [5, 0], [0, 3]]
F: [[5, 3], [2, 3], [3, 0]]
E: [[0, 0], [5, 0], [0, 3], [5, 3]]
F: [[2, 3], [3, 0]]
E: [[0, 0], [5, 0], [0, 3], [5, 3], [2, 3]]
F: [[3, 0], [2, 0]]
```

E: [[0, 0], [5, 0], [0, 3], [5, 3], [2, 3], [3, 0]]
F: [[2, 0], [3, 3]]
E: [[0, 0], [5, 0], [0, 3], [5, 3], [2, 3], [3, 0], [2, 0]]
F: [[3, 3], [0, 2]]
E: [[0, 0], [5, 0], [0, 3], [5, 3], [2, 3], [3, 0], [2, 0], [3, 3], [0, 2]]
F: [[5, 1], [5, 2]]
E: [[0, 0], [5, 0], [0, 3], [5, 3], [2, 3], [3, 0], [2, 0], [3, 3], [0, 2], [5, 1]]
F: [[5, 2], [0, 1]]
E: [[0, 0], [5, 0], [0, 3], [5, 3], [2, 3], [3, 0], [2, 0], [3, 3], [0, 2], [5, 1], [5, 2]]
Solution: [[0, 0], [5, 0], [2, 3], [2, 0], [0, 2], [5, 2], [4, 3]]

State Space Tree 2:



Output 2:
Enter Big jug capacity: 4
Enter Small jug capacity: 3
Enter target jug capacity: 2
E: [[0, 0]]
F: [[4, 0], [0, 3]]
E: [[0, 0], [4, 0]]
F: [[0, 3], [4, 3], [1, 3]]
E: [[0, 0], [4, 0], [0, 3]]
F: [[4, 3], [1, 3], [3, 0]]
E: [[0, 0], [4, 0], [0, 3], [4, 3]]
F: [[1, 3], [3, 0]]
E: [[0, 0], [4, 0], [0, 3], [4, 3], [1, 3]]
F: [[3, 0], [1, 0]]
E: [[0, 0], [4, 0], [0, 3], [4, 3], [1, 3], [3, 0]]
F: [[1, 0], [3, 3]]
E: [[0, 0], [4, 0], [0, 3], [4, 3], [1, 3], [3, 0], [1, 0], [3, 3]]
F: [[0, 1], [4, 2]]
E: [[0, 0], [4, 0], [0, 3], [4, 3], [1, 3], [3, 0], [1, 0], [3, 3], [0, 1]]
F: [[4, 2], [4, 1]]
E: [[0, 0], [4, 0], [0, 3], [4, 3], [1, 3], [3, 0], [1, 0], [3, 3], [0, 1], [4, 2]]
F: [[4, 1], [0, 2]]
E: [[0, 0], [4, 0], [0, 3], [4, 3], [1, 3], [3, 0], [1, 0], [3, 3], [0, 1], [4, 2], [4, 1]]
Solution: [[0, 0], [4, 0], [1, 3], [1, 0], [0, 1], [4, 1], [2, 3]]

Lab Program 2b:

Aim:

Implement water jug problem with Search tree generation using

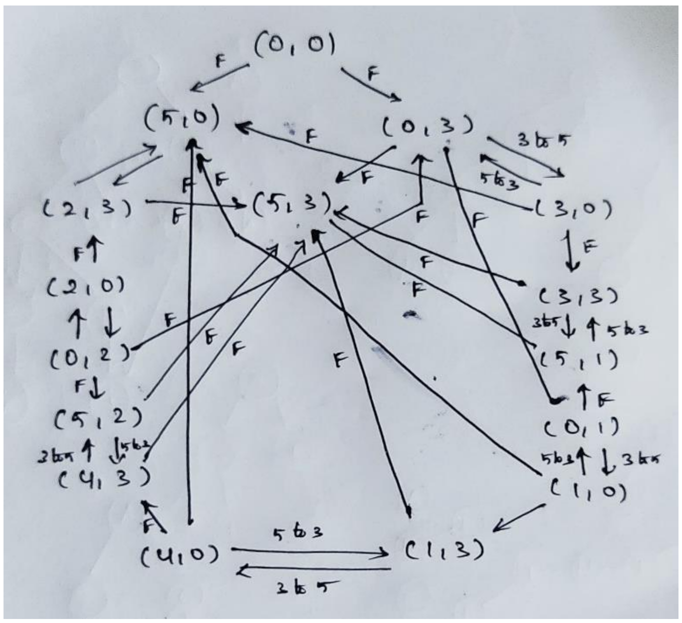
b. DFS

Program:

```
import copy
class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action
def SOLUTION(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return list(reversed(path))
def CHILD_NODE(problem, parent, action):
    child_state = copy.deepcopy(parent.state)
    if action in ('F1', 'F2'):
        index = int(action[1]) - 1
        if child_state[index] == 0:
            child_state[index] = problem.jug_size[index]
    elif action in ('E1', 'E2'):
        index = int(action[1]) - 1
        child_state[index] = 0
    elif action in ('1T2', '2T1'):
        from_jug, to_jug = int(action[0]),int(action[2])
        problem.POUR(from_jug, to_jug, child_state)
    return Node(child_state, parent, action)
def isPresent(node, frontier):
    return any(n.state == node.state for n in frontier)

def DEPTH_FIRST_SEARCH(problem):
    node = Node(problem.initial_state, None, None)
    if problem.GOAL_TEST(node.state):
        return SOLUTION(node)
    frontier = [node]
    explored = []
    while frontier:
        node = frontier.pop() #DFS
        explored.append(node.state)
        print("\nE:",explored)
        for action in problem.ACTIONS(node.state):
            child = CHILD_NODE(problem, node, action)
            if child.state and child.state not in explored and not isPresent(child, frontier):
                if problem.GOAL_TEST(child.state):
                    return SOLUTION(child)
                frontier.append(child)
        print("F:", [n.state for n in frontier])
```

```
        return "Failure"
class Problem:
    def __init__(self, j1, j2, g):
        self.initial_state = [0, 0]
        self.goal_state = [g, 'x']
        self.jug_size = [j1, j2]
    def ACTIONS(self, state):
        return ['F1', 'F2', 'E1', 'E2', '1T2', '2T1']
    def GOAL_TEST(self, state):
        return state[0] == self.goal_state[0]
    def POUR(self, from_jug, to_jug, child_state):
        j1, j2 = from_jug - 1, to_jug - 1
        avail = child_state[j1]
        if child_state[j2] + avail >= self.jug_size[j2]:
            child_state[j1] -= self.jug_size[j2] - child_state[j2]
            child_state[j2] = self.jug_size[j2]
        else:
            child_state[j2] += avail
            child_state[j1] = 0
b = int(input("Enter Big jug capacity: "))
s = int(input("Enter Small jug capacity: "))
t = int(input("Enter target jug capacity: "))
problem = Problem(b,s,t)
solution = DEPTH_FIRST_SEARCH(problem)
print("Solution:", solution)
```

State Space Tree 1:



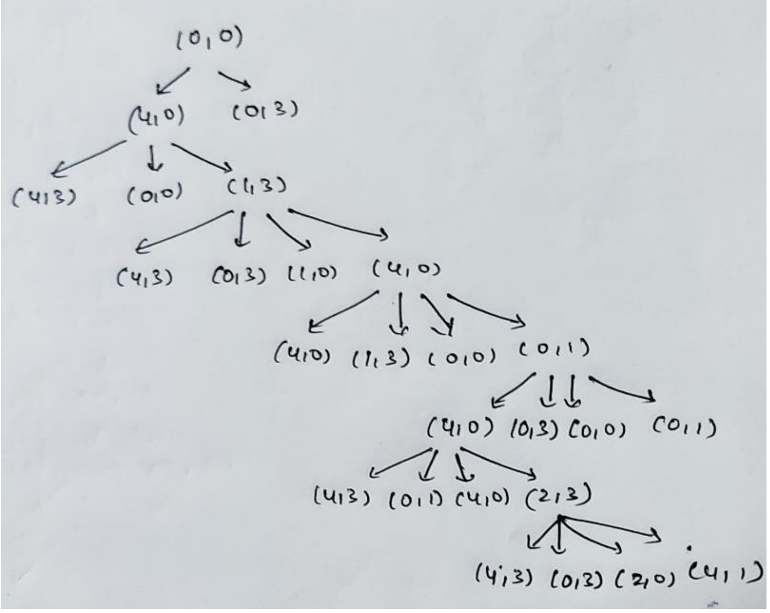
Output 1

Enter Big jug capacity: 5
Enter Small jug capacity: 3
Enter target jug capacity: 4
E: [[0, 0]]
F: [[5, 0], [0, 3]]
E: [[0, 0], [0, 3]]

F: [[5, 0], [5, 3], [3, 0]]
E: [[0, 0], [0, 3], [3, 0]]
F: [[5, 0], [5, 3], [3, 3]]
E: [[0, 0], [0, 3], [3, 0], [3, 3]]
F: [[5, 0], [5, 3], [5, 1]]
E: [[0, 0], [0, 3], [3, 0], [3, 3], [5, 1]]
F: [[5, 0], [5, 3], [0, 1]]
E: [[0, 0], [0, 3], [3, 0], [3, 3], [5, 1], [0, 1]]
F: [[5, 0], [5, 3], [1, 0]]
E: [[0, 0], [0, 3], [3, 0], [3, 3], [5, 1], [0, 1], [1, 0]]
F: [[5, 0], [5, 3], [1, 3]]
E: [[0, 0], [0, 3], [3, 0], [3, 3], [5, 1], [0, 1], [1, 0], [1, 3]]

Solution: [[0, 0], [0, 3], [3, 0], [3, 3], [5, 1], [0, 1], [1, 0], [1, 3], [4, 0]]

State Space Tree 2:



Output 2:

Enter Big jug capacity: 4
Enter Small jug capacity: 3
Enter target jug capacity: 2
E: [[0, 0]]
F: [[4, 0], [0, 3]]
E: [[0, 0], [0, 3]]
F: [[4, 0], [4, 3], [3, 0]]
E: [[0, 0], [0, 3], [3, 0]]
F: [[4, 0], [4, 3], [3, 3]]
E: [[0, 0], [0, 3], [3, 0], [3, 3]]
F: [[4, 0], [4, 3], [4, 2]
E: [[0, 0], [0, 3], [3, 0], [3, 3], [4, 2]]
F: [[4, 0], [4, 3], [0, 2]
E: [[0, 0], [0, 3], [3, 0], [3, 3], [4, 2], [0, 2]]

Solution: [[0, 0], [0, 3], [3, 0], [3, 3], [4, 2], [0, 2], [2, 0]]

Lab Program 3a:

Aim:

Implement Missionaries and Cannibals problem with Search tree generation using

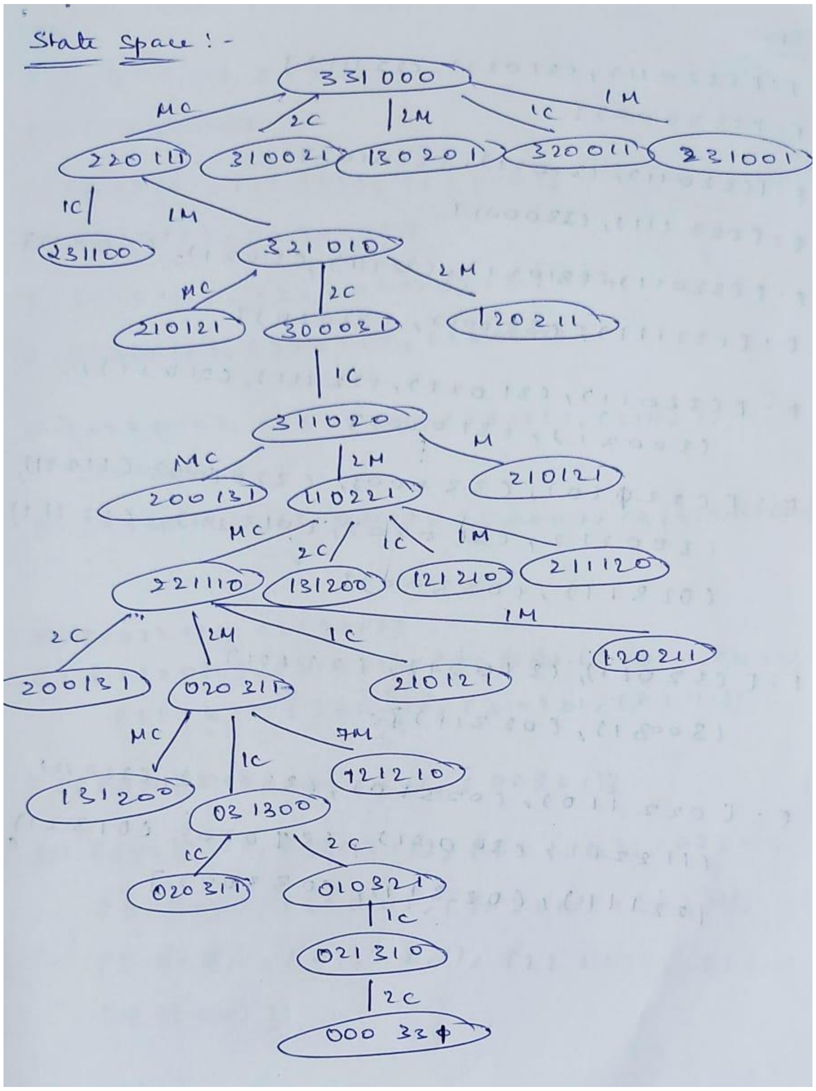
a. BFS

Program:

```
import copy
class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action
def SOLUTION(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return list(reversed(path))
def CHILD_NODE(parent, action):
    child_state = copy.deepcopy(parent.state)
    changed = 0
    if child_state[0][2] == 1:
        start, end = child_state[0], child_state[1]
    else:
        start, end = child_state[1], child_state[0]

    actions_dict = {
        '1M': (1, 0),
        '1C': (0, 1),
        '2M': (2, 0),
        '2C': (0, 2),
        '1M1C': (1, 1)
    }
    if action in actions_dict:
        move = actions_dict[action]
        if start[0] >= move[0] and start[1] >= move[1]:
            start[0] -= move[0]
            start[1] -= move[1]
            end[0] += move[0]
            end[1] += move[1]
            changed = 1
    if changed and ((child_state[0][0] >= child_state[0][1] or child_state[0][0] == 0) and
(child_state[1][0] >= child_state[1][1] or child_state[1][0] == 0)):
        start[2] = 0
        end[2] = 1
        return Node(child_state, parent, action)
def isPresent(node, frontier):
    return any(i.state == node.state for i in frontier)
def BREADTH_FIRST_SEARCH(problem):
    node = Node(problem.initial_state, None, None)
    if problem.GOAL_TEST(node.state):
```

```
        return SOLUTION(node)
    frontier = [node]
    explored = []
    while frontier:
        node = frontier.pop(0) #BFS
        explored.append(node.state)
        print("\nE:",explored)
        for action in problem.ACTIONS(node.state):
            child = CHILD_NODE(problem, node, action)
            if child.state and child.state not in explored and not isPresent(child, frontier):
                if problem.GOAL_TEST(child.state):
                    return SOLUTION(child)
                frontier.append(child)
        print("F:", [n.state for n in frontier])
    return "Failure"
def Bfs_MCP(problem):
    node = Node(problem.initial_state, None, None)
    if problem.GOAL_TEST(node.state):
        return SOLUTION(node)
    frontier = [node]
    explored = []
    while frontier:
        node = frontier.pop(0) #BFS
        explored.append(node.state)
        print("\nE:",explored)
        for action in problem.ACTIONS(node.state):
            child = CHILD_NODE(node, action)
            if child and child.state not in explored and not isPresent(child, frontier):
                if problem.GOAL_TEST(child.state):
                    print("\nE:",explored)
                    print("F:", [n.state for n in frontier])
                    return SOLUTION(child)
                frontier.append(child)
        print("F:", [n.state for n in frontier])
    return "Failure"
class Problem:
    def __init__(self):
        self.initial_state = [[3, 3, 1], [0, 0, 0]]
        self.goal_state = [[0, 0, 0], [3, 3, 1]]
        self.graph = ['1M', '1C', '2M', '2C', '1M1C']
    def ACTIONS(self, state):
        return self.graph
    def GOAL_TEST(self, state):
        return state == self.goal_state
problem = Problem()
solution = Bfs_MCP(problem)
print("Path:")
for i, step in enumerate(solution):
    print("Step", i+1, step)
```

State Space Tree:



Output:

E: [[[3, 3, 1], [0, 0, 0]]]
F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]]]
F: [[[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]]]
F: [[[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]]]
F: [[[3, 2, 1], [0, 1, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]]]
F: [[[3, 0, 0], [0, 3, 1]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]]]
F: [[[3, 1, 1], [0, 2, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]]]
F: [[[1, 1, 0], [2, 2, 1]]]

E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]]]
F: [[[2, 2, 1], [1, 1, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]]]
F: [[[0, 2, 0], [3, 1, 1]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]], [[0, 2, 0], [3, 1, 1]]]
F: [[[0, 3, 1], [3, 0, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]], [[0, 2, 0], [3, 1, 1]], [[0, 3, 1], [3, 0, 0]]]
F: [[[0, 1, 0], [3, 2, 1]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]], [[0, 2, 0], [3, 1, 1]], [[0, 3, 1], [3, 0, 0]], [[0, 1, 0], [3, 2, 1]]]
F: [[[1, 1, 1], [2, 2, 0]], [[0, 2, 1], [3, 1, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]], [[0, 2, 0], [3, 1, 1]], [[0, 3, 1], [3, 0, 0]], [[0, 1, 0], [3, 2, 1]], [[1, 1, 1], [2, 2, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]], [[0, 2, 0], [3, 1, 1]], [[0, 3, 1], [3, 0, 0]], [[0, 1, 0], [3, 2, 1]], [[1, 1, 1], [2, 2, 0]]]
F: [[[0, 2, 1], [3, 1, 0]]]

Path:
Step 1 [[3, 3, 1], [0, 0, 0]]
Step 2 [[3, 1, 0], [0, 2, 1]]
Step 3 [[3, 2, 1], [0, 1, 0]]
Step 4 [[3, 0, 0], [0, 3, 1]]
Step 5 [[3, 1, 1], [0, 2, 0]]
Step 6 [[1, 1, 0], [2, 2, 1]]
Step 7 [[2, 2, 1], [1, 1, 0]]
Step 8 [[0, 2, 0], [3, 1, 1]]
Step 9 [[0, 3, 1], [3, 0, 0]]
Step 10 [[0, 1, 0], [3, 2, 1]]
Step 11 [[1, 1, 1], [2, 2, 0]]
Step 12 [[0, 0, 0], [3, 3, 1]]

Lab Program 3b:

Aim:

Implement Missionaries and Cannibals problem with Search tree generation using b. DFS

Program:

```
import copy
class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action
def SOLUTION(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return list(reversed(path))
def CHILD_NODE(parent, action):
    child_state = copy.deepcopy(parent.state)
    changed = 0
    if child_state[0][2] == 1:
        start, end = child_state[0], child_state[1]
    else:
        start, end = child_state[1], child_state[0]

    actions_dict = {
        '1M': (1, 0),
        '1C': (0, 1),
        '2M': (2, 0),
        '2C': (0, 2),
        '1M1C': (1, 1)
    }
    if action in actions_dict:
        move = actions_dict[action]
        if start[0] >= move[0] and start[1] >= move[1]:
            start[0] -= move[0]
            start[1] -= move[1]
            end[0] += move[0]
            end[1] += move[1]
            changed = 1
    if changed and ((child_state[0][0] >= child_state[0][1] or child_state[0][0] == 0) and
(child_state[1][0] >= child_state[1][1] or child_state[1][0] == 0)):
        start[2] = 0
        end[2] = 1
        return Node(child_state, parent, action)
def isPresent(node, frontier):
    return any(i.state == node.state for i in frontier)
def Dfs_MCP(problem):
    node = Node(problem.initial_state, None, None)
    if problem.GOAL_TEST(node.state):
        return SOLUTION(node)
```

```
    frontier = [node]
    explored = []
    while frontier:
        node = frontier.pop() #DFS
        explored.append(node.state)
        for action in problem.ACTIONS(node.state):
            child = CHILD_NODE(node, action)
            if child and child.state not in explored and not isPresent(child, frontier):
                if problem.GOAL_TEST(child.state):
                    return SOLUTION(child)
                frontier.append(child)
    return "Failure"
class Problem:
    def __init__(self):
        self.initial_state = [[3, 3, 1], [0, 0, 0]]
        self.goal_state = [[0, 0, 0], [3, 3, 1]]
        self.graph = ['1M', '1C', '2M', '2C', '1M1C']
    def ACTIONS(self, state):  return self.graph
    def GOAL_TEST(self, state):  return state == self.goal_state
problem = Problem()
solution = Dfs_MCP(problem)
print("Path")
for i, step in enumerate(solution):
    print("Step", i+1, step)
```

Output:

E: [[[3, 3, 1], [0, 0, 0]]]
F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 1]]]
E: [[[3, 3, 1], [0, 0, 0]], [[2, 2, 0], [1, 1, 1]]]
F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[3, 2, 1], [0, 1, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]]]
F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[3, 0, 0], [0, 3, 1]]]
E: [[[3, 3, 1], [0, 0, 0]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]]]
F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[3, 1, 1], [0, 2, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]]]
F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[1, 1, 0], [2, 2, 1]]]
E: [[[3, 3, 1], [0, 0, 0]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]]]
F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 1], [1, 1, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]]]
F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[0, 2, 0], [3, 1, 1]]]
E: [[[3, 3, 1], [0, 0, 0]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]], [[0, 2, 0], [3, 1, 1]]]
F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[0, 3, 1], [3, 0, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]], [[0, 2, 0], [3, 1, 1]], [[0, 3, 1], [3, 0, 0]]]
F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[0, 1, 0], [3, 2, 1]]]
E: [[[3, 3, 1], [0, 0, 0]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]], [[0, 2, 0], [3, 1, 1]], [[0, 3, 1], [3, 0, 0]], [[0, 1, 0], [3, 2, 1]]]

F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[1, 1, 1], [2, 2, 0]], [[0, 2, 1], [3, 1, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]], [[0, 2, 0], [3, 1, 1]], [[0, 3, 1], [3, 0, 0]], [[0, 1, 0], [3, 2, 1]], [[0, 2, 1], [3, 1, 0]]]
E: [[[3, 3, 1], [0, 0, 0]], [[2, 2, 0], [1, 1, 1]], [[3, 2, 1], [0, 1, 0]], [[3, 0, 0], [0, 3, 1]], [[3, 1, 1], [0, 2, 0]], [[1, 1, 0], [2, 2, 1]], [[2, 2, 1], [1, 1, 0]], [[0, 2, 0], [3, 1, 1]], [[0, 3, 1], [3, 0, 0]], [[0, 1, 0], [3, 2, 1]], [[0, 2, 1], [3, 1, 0]]]
F: [[[3, 2, 0], [0, 1, 1]], [[3, 1, 0], [0, 2, 1]], [[1, 1, 1], [2, 2, 0]]]

Path:
Step 1 [[3, 3, 1], [0, 0, 0]]
Step 2 [[2, 2, 0], [1, 1, 1]]
Step 3 [[3, 2, 1], [0, 1, 0]]
Step 4 [[3, 0, 0], [0, 3, 1]]
Step 5 [[3, 1, 1], [0, 2, 0]]
Step 6 [[1, 1, 0], [2, 2, 1]]
Step 7 [[2, 2, 1], [1, 1, 0]]
Step 8 [[0, 2, 0], [3, 1, 1]]
Step 9 [[0, 3, 1], [3, 0, 0]]
Step 10 [[0, 1, 0], [3, 2, 1]]
Step 11 [[0, 2, 1], [3, 1, 0]]
Step 12 [[0, 0, 0], [3, 3, 1]]

Lab Program 4a:

Aim:

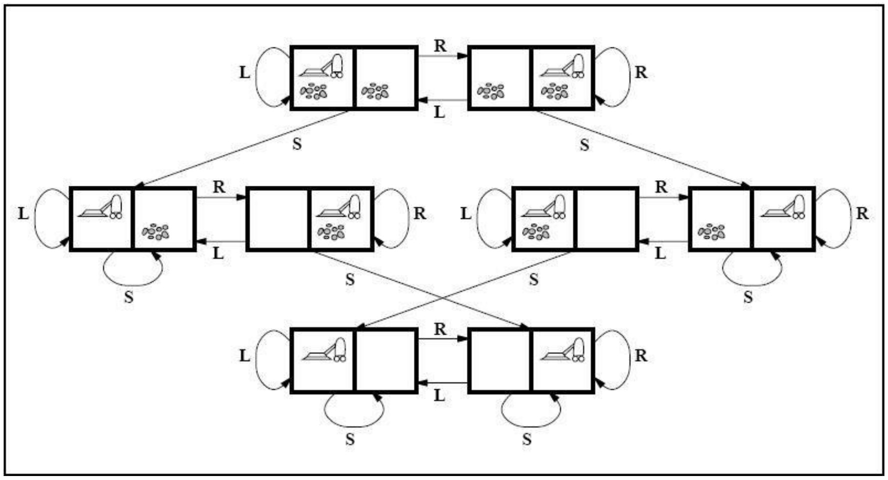
Implement Vacuum World problem with Search tree generation using
a. BFS

Program:

```
import copy

class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action
def SOLUTION(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return list(reversed(path))
def isPresent(node, frontier):
    return any(i.state == node.state for i in frontier)
def CHILD_NODE(problem, parent, action):
  child_state = copy.deepcopy(parent.state)
  if(action=='sweep'):
    child_state=problem.sweep(child_state)
  elif(action=='move_r'):
    child_state=problem.move_r(child_state)
  elif(action=='move_l'):
    child_state=problem.move_l(child_state)
  return Node(child_state,parent,action)
def Bfs_VWP(problem):
    node = Node(problem.initial_state, None, None)
    if problem.GOAL_TEST(node.state):
        return SOLUTION(node)
    frontier = [node]
    explored = []
    iteration = 1
    while frontier:
        print(f"Iteration {iteration}:")
        print("f:", [n.state for n in frontier])
        print("e:", explored)
        iteration += 1
        node = frontier.pop(0)  # BFS
        explored.append(node.state)
        for action in problem.ACTIONS(node.state):
            child = CHILD_NODE(problem, node, action)
            if child and child.state not in explored and not isPresent(child, frontier):
                if problem.GOAL_TEST(child.state):
                    print(f"Iteration {iteration}:")
                    print("f:", [n.state for n in frontier])
                    print("e:", explored)
```

```
                return SOLUTION(child)
            frontier.append(child)
    return "Failure"
class Problem:
    def __init__(self, ini,goal):
        self.initial_state = ini
        self.goal_state = goal
        self.actions=['sweep','move_r','move_l']
    def ACTIONS(self, state):
        return self.actions
    def GOAL_TEST(self, state):
        a=state[0] == self.goal_state
        b=state[1]==self.goal_state
        return a and b
    def sweep(self,state):
        room1,room2,pos=state
        if pos=="right" and room2=="dirty":
            room2="clean"
        if pos=="left" and room1=="dirty":
            room1="clean"
        return [room1,room2,pos]
    def move_l(self,state):
        room1,room2,pos=state
        if pos == 'right':
            pos = 'left'
        return [room1,room2,pos]
    def move_r(self,state):
        room1,room2,pos=state
        if pos == 'left':
            pos = 'right'
        return [room1,room2,pos]
room1 = 'dirty'
room2 = 'dirty'
pos = input("Enter position of the vaccum: ")
problem=Problem([room1, room2, pos],'clean')
solution=Bfs_VWP(problem)
print("Solution:",solution)
```

State Space Tree:



Output:
Enter position of the vaccum: right
Iteration 1:
f: [['dirty', 'dirty', 'right']]
e: []
Iteration 2:
f: [['dirty', 'clean', 'right'], ['dirty', 'dirty', 'left']]
e: [['dirty', 'dirty', 'right']]
Iteration 3:
f: [['dirty', 'dirty', 'left'], ['dirty', 'clean', 'left']]
e: [['dirty', 'dirty', 'right'], ['dirty', 'clean', 'right']]
Iteration 4:
f: [['dirty', 'clean', 'left'], ['clean', 'dirty', 'left']]
e: [['dirty', 'dirty', 'right'], ['dirty', 'clean', 'right'], ['dirty', 'dirty', 'left']]
Iteration 5:
f: [['clean', 'dirty', 'left']]
e: [['dirty', 'dirty', 'right'], ['dirty', 'clean', 'right'], ['dirty', 'dirty', 'left'], ['dirty', 'clean', 'left']]
Solution: [['dirty', 'dirty', 'right'], ['dirty', 'clean', 'right'], ['dirty', 'clean', 'left'], ['clean', 'clean', 'left']]
Output 2:
Enter position of the vaccum: left
Iteration 1:
f: [['dirty', 'dirty', 'left']] e: []
Iteration 2:
f: [['clean', 'dirty', 'left'], ['dirty', 'dirty', 'right']]
e: [['dirty', 'dirty', 'left']]
Iteration 3:
f: [['dirty', 'dirty', 'right'], ['clean', 'dirty', 'right']]
e: [['dirty', 'dirty', 'left'], ['clean', 'dirty', 'left']]
Iteration 4:
f: [['clean', 'dirty', 'right'], ['dirty', 'clean', 'right']]
e: [['dirty', 'dirty', 'left'], ['clean', 'dirty', 'left'], ['dirty', 'dirty', 'right']]
Iteration 5:
f: [['dirty', 'clean', 'right']]
e: [['dirty', 'dirty', 'left'], ['clean', 'dirty', 'left'], ['dirty', 'dirty', 'right'], ['clean', 'dirty', 'right']]
Solution: [['dirty', 'dirty', 'left'], ['clean', 'dirty', 'left'], ['clean', 'dirty', 'right'], ['clean', 'clean', 'right']]

Lab Program 4b:
Aim:

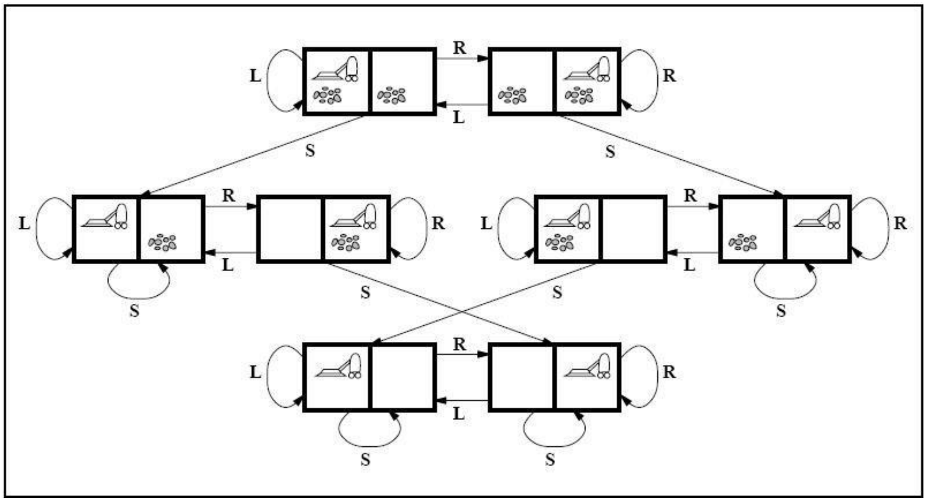
Implement Vacuum World problem with Search tree generation using
b. DFS

Program:

```
import copy
class Node:
    def __init__(self, state, parent, action):
        self.state = state
        self.parent = parent
        self.action = action

def SOLUTION(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return list(reversed(path))
def isPresent(node, frontier):
    return any(i.state == node.state for i in frontier)
def CHILD_NODE(problem, parent, action):
  child_state = copy.deepcopy(parent.state)
  if(action=='sweep'):
    child_state=problem.sweep(child_state)
  elif(action=='move_r'):
    child_state=problem.move_r(child_state)
  elif(action=='move_l'):
    child_state=problem.move_l(child_state)
  return Node(child_state,parent,action)
def Dfs_VWP(problem):
    ''' Only one line change'''
    node = Node(problem.initial_state, None, None)
    if problem.GOAL_TEST(node.state):
        return SOLUTION(node)
    frontier = [node]
    explored = []
    iteration = 1
    while frontier:
        print(f"Iteration {iteration}:")
        print("f:", [n.state for n in frontier])
        print("e:", explored)
        iteration += 1
        node = frontier.pop()  # DFS
        explored.append(node.state)
        for action in problem.ACTIONS(node.state):
            child = CHILD_NODE(problem, node, action)
            if child and child.state not in explored and not isPresent(child, frontier):
                if problem.GOAL_TEST(child.state):
                    print(f"Iteration {iteration}:")
                    print("f:", [n.state for n in frontier])
```

```
                    print("e:", explored)
                    return SOLUTION(child)
                frontier.append(child)
    return "Failure"
class Problem:
    def __init__(self, ini,goal):
        self.initial_state = ini
        self.goal_state = goal
        self.actions=['sweep','move_r','move_l']
    def ACTIONS(self, state):
        return self.actions
    def GOAL_TEST(self, state):
        a=state[0] == self.goal_state
        b=state[1]==self.goal_state
        return a and b
    def sweep(self,state):
        room1,room2,pos=state
        if pos=="right" and room2=="dirty":
            room2="clean"
        if pos=="left" and room1=="dirty":
            room1="clean"
        return [room1,room2,pos]
    def move_l(self,state):
        room1,room2,pos=state
        if pos == 'right':
            pos = 'left'
        return [room1,room2,pos]
    def move_r(self,state):
        room1,room2,pos=state
        if pos == 'left':
            pos = 'right'
        return [room1,room2,pos]
room1 = 'dirty'
room2 = 'dirty'
pos = input("Enter position of the vaccum: ")
problem=Problem([room1, room2, pos],'clean')
solution=Dfs_VWP(problem)
print("Solution:",solution)
```

State Space Tree:



Output 1:
Enter position of the vaccum: right
Iteration 1:
f: [['dirty', 'dirty', 'right']] e: []
Iteration 2:
f: [['dirty', 'clean', 'right'], ['dirty', 'dirty', 'left']]
e: [['dirty', 'dirty', 'right']]
Iteration 3:
f: [['dirty', 'clean', 'right'], ['clean', 'dirty', 'left']]
e: [['dirty', 'dirty', 'right'], ['dirty', 'dirty', 'left']]
Iteration 4:
f: [['dirty', 'clean', 'right'], ['clean', 'dirty', 'right']]
e: [['dirty', 'dirty', 'right'], ['dirty', 'dirty', 'left'], ['clean', 'dirty', 'left']]
Iteration 5:
f: [['dirty', 'clean', 'right']]
e: [['dirty', 'dirty', 'right'], ['dirty', 'dirty', 'left'], ['clean', 'dirty', 'left'], ['clean', 'dirty', 'right']]
Solution: [['dirty', 'dirty', 'right'], ['dirty', 'dirty', 'left'], ['clean', 'dirty', 'left'], ['clean', 'dirty', 'right'], ['clean', 'clean', 'right']]

Output 2:
Enter position of the vaccum: left
Iteration 1:
f: [['dirty', 'dirty', 'left']] e: []
Iteration 2:
f: [['clean', 'dirty', 'left'], ['dirty', 'dirty', 'right']]
e: [['dirty', 'dirty', 'left']]
Iteration 3:
f: [['clean', 'dirty', 'left'], ['dirty', 'clean', 'right']]
e: [['dirty', 'dirty', 'left'], ['dirty', 'dirty', 'right']]
Iteration 4:
f: [['clean', 'dirty', 'left'], ['dirty', 'clean', 'left']]
e: [['dirty', 'dirty', 'left'], ['dirty', 'dirty', 'right'], ['dirty', 'clean', 'right']]
Iteration 5:
f: [['clean', 'dirty', 'left']]
e: [['dirty', 'dirty', 'left'], ['dirty', 'dirty', 'right'], ['dirty', 'clean', 'right'], ['dirty', 'clean', 'left']]
Solution: [['dirty', 'dirty', 'left'], ['dirty', 'dirty', 'right'], ['dirty', 'clean', 'right'], ['dirty', 'clean', 'left'], ['clean', 'clean', 'left']]

Lab Program 5a:

Aim:

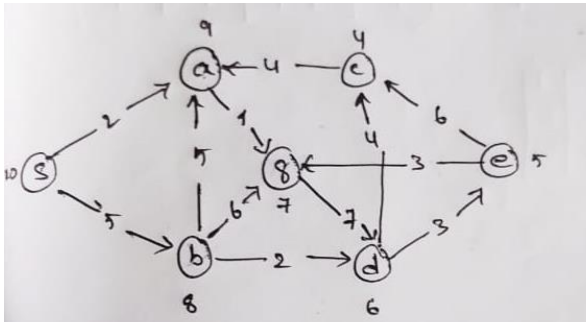
Implement the following a. Greedy Best First Search

Program:

```
from queue import PriorityQueue
class Node:
    def __init__(self, state, parent, action, heuristic):
        self.state = state
        self.parent = parent
        self.action = action
        self.heuristic = heuristic
def solution(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return list(reversed(path))
class Problem:
    def __init__(self,i,g,graph):
        self.initial_state = i
        self.goal_state = g
        self.graph = graph
    def actions(self, state):
        return self.graph.get(state, [])
    def goal_test(self, heuristic):
        return heuristic == self.goal_state[1]
def greedy_best_first_search(problem):
    node = Node(problem.initial_state[0], None, None, problem.initial_state[1])
    if problem.goal_test(node.heuristic):
        return solution(node)
    frontier = PriorityQueue()
    frontier.put((node.heuristic, node))
    explored = []
    h=0
    while not frontier.empty():
        node = frontier.get()[1]
        h+=node.heuristic
        explored.append(node.state)
        print("F: ", [n[1].state for n in frontier.queue])
        print("E", explored)
       # print("Exploring ", node.state)
        if problem.goal_test(node.heuristic):
            print("Path Cost:",h)
            return solution(node)
        for action in problem.actions(node.state):
            child = Node(action[0], node, action, action[1])
            if child.state not in explored:
                frontier.put((child.heuristic, child))
                #print("Adding ", child.state, " to Frontier")
    return "Failure"
```

```
graph=eval(input("Enter the graph:"))
i=input("Enter initial state:")
g=input("Enter goal state:")
problem = Problem(i,g,graph)
solution = greedy_best_first_search(problem)
print("Solution:", solution)
```

Graph:



Output 1:

Enter graph: {'s': {'a': 2, 'b': 5, 'X': 10}, 'a': {'g': 1, 'X': 9}, 'b': {'a': 5, 'g': 6, 'd': 2, 'X': 8}, 'g': {'d': 7, 'X': 7}, 'd': {'c': 4, 'e': 3, 'X': 6}, 'c': {'a': 4, 'X': 4}, 'e': {'c': 6, 'g': 3, 'X': 5}}
Enter start node:s
Enter goal node:g
F: []
E: []
F: ['a']
E: ['s']
F: ['g', 'a', 'a']
E: ['s', 'b']
F: ['e', 'g', 'a', 'a']
E: ['s', 'b', 'd']
F: ['g', 'a', 'a', 'a']
E: ['s', 'b', 'd', 'c']
F: ['g', 'a', 'a', 'a']
E: ['s', 'b', 'd', 'c', 'e']
Goal node found

Output 2:

Enter graph: {'s': {'a': 2, 'b': 5, 'X': 10}, 'a': {'g': 1, 'X': 9}, 'b': {'a': 5, 'g': 6, 'd': 2, 'X': 8}, 'g': {'d': 7, 'X': 7}, 'd': {'c': 4, 'e': 3, 'X': 6}, 'c': {'a': 4, 'X': 4}, 'e': {'c': 6, 'g': 3, 'X': 5}}
Enter start node:a
Enter goal node:e
F: []
E: []
F: []
E: ['a']
F: []
E: ['a', 'g']
F: ['e']
E: ['a', 'g', 'd']
F: []
E: ['a', 'g', 'd', 'c']
Goal node found

Lab Program 5b:

Aim:

Implement the following

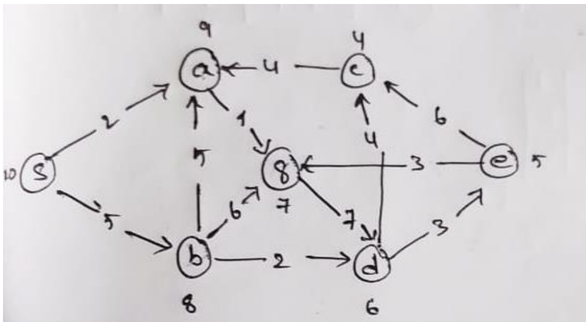
b. A* algorithm

Program:

```
from queue import PriorityQueue
class Node:
    def __init__(self, state, parent, action, cost, heuristic):
        self.state = state
        self.parent = parent
        self.action = action
        self.cost = cost
        self.heuristic = heuristic
        self.total_cost = cost + heuristic

    def __lt__(self, other):
        return self.total_cost < other.total_cost
def solution(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return list(reversed(path))
class Problem:
    def __init__(self,i,g,graph):
            self.initial_state = i
            self.goal_state = g
            self.graph = graph
    def actions(self, state):
        return self.graph.get(state, [])
    def goal_test(self, state):
        return state == self.goal_state
def a_star_search(problem):
    node = Node(problem.initial_state[0], None, None, problem.initial_state[1], problem.initial_state[2])
    if problem.goal_test(node.state):
        return solution(node)
    frontier = PriorityQueue()
    frontier.put(node)
    explored = []
    while not frontier.empty():
        node = frontier.get()
        explored.append(node.state)
        print("Frontier: ", [(n.state, n.total_cost) for n in frontier.queue])
        print("Explored", explored)
       # print("Exploring ", node.state)
        if problem.goal_test(node.state):
            print("Path Cost:",node.total_cost)
            return solution(node)
        for action in problem.actions(node.state):
            child = Node(action[0], node, action, action[1] + node.cost, action[2])
```

```
                if child.state not in explored:
                    frontier.put(child)
                    #print("Adding ", child.state, " to Frontier")
        return "Failure"
graph=eval(input("Enter the graph:"))
i=input("Enter initial state:")
g=input("Enter goal state:")
problem = Problem(i,g,graph)
solution = a_star_search(problem)
print("Solution:", solution
```

Graph:



Output 1:

Enter graph: {'s': {'a': 2, 'b': 5, 'X': 10}, 'a': {'g': 1, 'X': 9}, 'b': {'a': 5, 'g': 6, 'd': 2, 'X': 8}, 'g': {'d': 7, 'X': 7}, 'd': {'c': 4, 'e': 3, 'X': 6}, 'c': {'a': 4, 'X': 4}, 'e': {'c': 6, 'g': 3, 'X': 5}}
Enter start node:s
Enter goal node:g
F: ['s']
E: []
F: ['a', 'b']
E: ['s']
F: ['b', 'g']
E: ['s', 'a']
Goal node found

Output 2:

Enter graph: {'s': {'a': 2, 'b': 5, 'X': 10}, 'a': {'g': 1, 'X': 9}, 'b': {'a': 5, 'g': 6, 'd': 2, 'X': 8}, 'g': {'d': 7, 'X': 7}, 'd': {'c': 4, 'e': 3, 'X': 6}, 'c': {'a': 4, 'X': 4}, 'e': {'c': 6, 'g': 3, 'X': 5}}
Enter start node:a
Enter goal node:e
F: ['a']
E: []
F: ['g']
E: ['a']
F: ['d']
E: ['a', 'g']
F: ['c', 'e']
E: ['a', 'g', 'd']
F: ['e']
E: ['a', 'g', 'd', 'c']
Goal node found

Lab Program 6:
Aim:

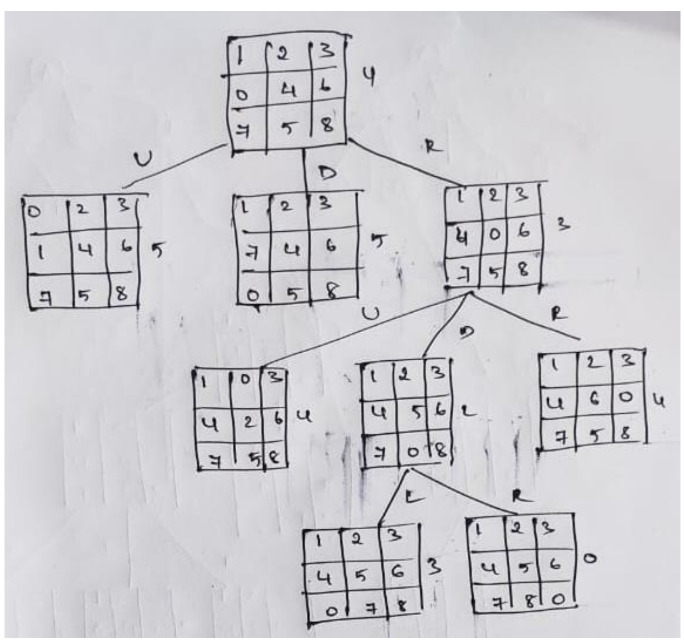
Implement 8-puzzle problem using A* algorithm

Program:

```
from queue import PriorityQueue
class Node:
    def __init__(self, state, parent, action, cost, heuristic):
        self.state = state
        self.parent = parent
        self.action = action
        self.cost = cost
        self.heuristic = heuristic
        self.total_cost = cost + heuristic
    def __lt__(self, other):
        return self.total_cost < other.total_cost
    def __eq__(self, other):
        return self.total_cost == other.total_cost
def solution(node):
    path = []
    while node:
        path.append(node.state)
        node = node.parent
    return list(reversed(path))
def cal_h(parent, child):
    heuristic = 0
    for i in range(len(parent)):
        if parent[i] != child[i]:
            heuristic += 1
    return heuristic
def child_node(problem, parent, action):
    child_state = parent.state.copy()
    actual_position = parent.state.index(0)
    next_position = {
        'r': actual_position + 1,
        'l': actual_position - 1,
        'u': actual_position - 3,
        'd': actual_position + 3
    }[action]
    child_state[actual_position], child_state[next_position] = child_state[next_position],
child_state[actual_position]
    child_cost = parent.cost + 1
    child_heuristic = cal_h(problem.goal_state, child_state)
    return Node(child_state, parent, action, child_cost, child_heuristic)
def a_star_search(problem):
    node = Node(problem.initial_state, None, None, 0, cal_h(problem.goal_state, problem.initial_state))
    if problem.goal_test(node.state):
        return solution(node)
    frontier = PriorityQueue()
    frontier.put((node.total_cost, node))
    explored = set()
```

```
    while not frontier.empty():
        print("Frontier: ", [(n[1].state, n[1].total_cost) for n in frontier.queue])
        print("Explored", explored)
        print()
        print()
        _, node = frontier.get()
        explored.add(tuple(node.state))
        if problem.goal_test(node.state):
            return solution(node)
        for action in problem.actions(node.state):
            child = child_node(problem, node, action)
            if tuple(child.state) not in explored:
                frontier.put((child.total_cost, child))
    return "Failure"
class Problem:
   def __init__(self,i,g,graph):
        self.initial_state = i
        self.goal_state = g
        self.graph = graph
    def actions(self, state):
        return self.action_map.get(state.index(0), [])
    def goal_test(self, state):
        return state == self.goal_state
graph=eval(input("Enter the graph:"))
i=eval(input("Enter initial state:"))
g=eval(input("Enter goal state:"))
problem = Problem(i,g,graph)
solution = a_star_search(problem)
print("Solution:", solution)
```

Graph 1:



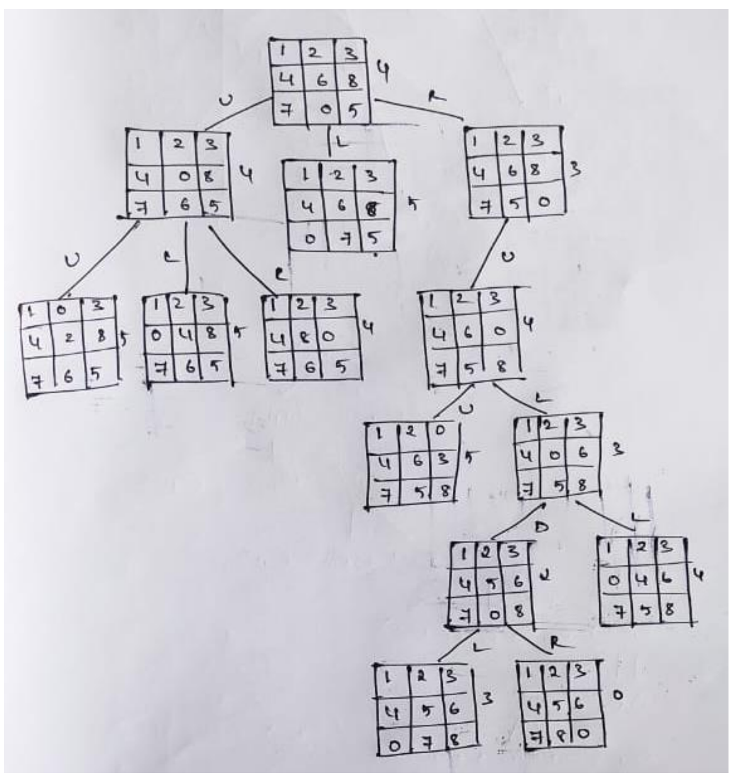
Output 1:
Enter start state: [1,2,3,0,4,6,7,5,8]
Enter goal state: [1,2,3,4,5,6,7,8,0]
Frontier: [[[1, 2, 3, 0, 4, 6, 7, 5, 8], 4]]
Explored: []
State: [1, 2, 3, 0, 4, 6, 7, 5, 8] Heuristic: 4

Frontier: [[[0, 2, 3, 1, 4, 6, 7, 5, 8], 5], [[1, 2, 3, 7, 4, 6, 0, 5, 8], 5], [[1, 2, 3, 4, 0, 6, 7, 5, 8], 3]]
Explored: [[1, 2, 3, 0, 4, 6, 7, 5, 8]]
State: [1, 2, 3, 4, 0, 6, 7, 5, 8] Heuristic: 3

Frontier: [[[1, 2, 3, 7, 4, 6, 0, 5, 8], 5], [[0, 2, 3, 1, 4, 6, 7, 5, 8], 5], [[1, 0, 3, 4, 2, 6, 7, 5, 8], 4], [[1, 2, 3, 4, 5, 6, 7, 0, 8], 2], [[1, 2, 3, 4, 6, 0, 7, 5, 8], 4]]
Explored: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8]]
State: [1, 2, 3, 4, 5, 6, 7, 0, 8] Heuristic: 2

Frontier: [[[1, 0, 3, 4, 2, 6, 7, 5, 8], 4], [[1, 2, 3, 4, 6, 0, 7, 5, 8], 4], [[0, 2, 3, 1, 4, 6, 7, 5, 8], 5], [[1, 2, 3, 7, 4, 6, 0, 5, 8], 5], [[1, 2, 3, 4, 5, 6, 0, 7, 8], 3], [[1, 2, 3, 4, 5, 6, 7, 8, 0], 0]]
Explored: [[1, 2, 3, 0, 4, 6, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8], [1, 2, 3, 4, 5, 6, 7, 0, 8]]
State: [1, 2, 3, 4, 5, 6, 7, 8, 0] Heuristic: 0

Goal state achieved: [1, 2, 3, 4, 5, 6, 7, 8, 0]
Graph 2:



Output 2:
Enter start state: [1,2,3,4,6,8,7,0,5]
Enter goal state: [1,2,3,4,5,6,7,8,0]
Frontier: [[[1, 2, 3, 4, 6, 8, 7, 0, 5], 4]]
Explored: []

State: [1, 2, 3, 4, 6, 8, 7, 0, 5] Heuristic: 4

Frontier: [[[1, 2, 3, 4, 0, 8, 7, 6, 5], 4], [[1, 2, 3, 4, 6, 8, 0, 7, 5], 5], [[1, 2, 3, 4, 6, 8, 7, 5, 0], 3]]
Explored: [[1, 2, 3, 4, 6, 8, 7, 0, 5]]
State: [1, 2, 3, 4, 6, 8, 7, 5, 0] Heuristic: 3

Frontier: [[[1, 2, 3, 4, 0, 8, 7, 6, 5], 4], [[1, 2, 3, 4, 6, 8, 0, 7, 5], 5], [[1, 2, 3, 4, 6, 0, 7, 5, 8], 4]]
Explored: [[1, 2, 3, 4, 6, 8, 7, 0, 5], [1, 2, 3, 4, 6, 8, 7, 5, 0]]
State: [1, 2, 3, 4, 0, 8, 7, 6, 5] Heuristic: 4

Frontier: [[[1, 2, 3, 4, 6, 0, 7, 5, 8], 4], [[1, 2, 3, 4, 6, 8, 0, 7, 5], 5], [[1, 0, 3, 4, 2, 8, 7, 6, 5], 5], [[1, 2, 3, 0, 4, 8, 7, 6, 5], 5], [[1, 2, 3, 4, 8, 0, 7, 6, 5], 4]]
Explored: [[1, 2, 3, 4, 6, 8, 7, 0, 5], [1, 2, 3, 4, 6, 8, 7, 5, 0], [1, 2, 3, 4, 0, 8, 7, 6, 5]]
State: [1, 2, 3, 4, 6, 0, 7, 5, 8] Heuristic: 4

Frontier: [[[1, 2, 3, 4, 8, 0, 7, 6, 5], 4], [[1, 0, 3, 4, 2, 8, 7, 6, 5], 5], [[1, 2, 3, 0, 4, 8, 7, 6, 5], 5], [[1, 2, 3, 4, 6, 8, 0, 7, 5], 5], [[1, 2, 0, 4, 6, 3, 7, 5, 8], 5], [[1, 2, 3, 4, 0, 6, 7, 5, 8], 3]]
Explored: [[1, 2, 3, 4, 6, 8, 7, 0, 5], [1, 2, 3, 4, 6, 8, 7, 5, 0], [1, 2, 3, 4, 0, 8, 7, 6, 5], [1, 2, 3, 4, 6, 0, 7, 5, 8]]
State: [1, 2, 3, 4, 0, 6, 7, 5, 8] Heuristic: 3

Frontier: [[[1, 2, 3, 4, 8, 0, 7, 6, 5], 4], [[1, 2, 3, 0, 4, 8, 7, 6, 5], 5], [[1, 2, 3, 4, 6, 8, 0, 7, 5], 5], [[1, 2, 0, 4, 6, 3, 7, 5, 8], 5], [[1, 0, 3, 4, 2, 8, 7, 6, 5], 5], [[1, 0, 3, 4, 2, 6, 7, 5, 8], 4], [[1, 2, 3, 4, 5, 6, 7, 0, 8], 2], [[1, 2, 3, 0, 4, 6, 7, 5, 8], 4]]
Explored: [[1, 2, 3, 4, 6, 8, 7, 0, 5], [1, 2, 3, 4, 6, 8, 7, 5, 0], [1, 2, 3, 4, 0, 8, 7, 6, 5], [1, 2, 3, 4, 6, 0, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8]]
State: [1, 2, 3, 4, 5, 6, 7, 0, 8] Heuristic: 2

Frontier: [[[1, 0, 3, 4, 2, 6, 7, 5, 8], 4], [[1, 2, 3, 4, 8, 0, 7, 6, 5], 4], [[1, 2, 3, 0, 4, 6, 7, 5, 8], 4], [[1, 0, 3, 4, 2, 8, 7, 6, 5], 5], [[1, 2, 3, 0, 4, 8, 7, 6, 5], 5], [[1, 2, 3, 4, 6, 8, 0, 7, 5], 5], [[1, 2, 0, 4, 6, 3, 7, 5, 8], 5], [[1, 2, 3, 4, 5, 6, 0, 7, 8], 3], [[1, 2, 3, 4, 5, 6, 7, 8, 0], 0]]
Explored: [[1, 2, 3, 4, 6, 8, 7, 0, 5], [1, 2, 3, 4, 6, 8, 7, 5, 0], [1, 2, 3, 4, 0, 8, 7, 6, 5], [1, 2, 3, 4, 6, 0, 7, 5, 8], [1, 2, 3, 4, 0, 6, 7, 5, 8], [1, 2, 3, 4, 5, 6, 7, 0, 8]]
State: [1, 2, 3, 4, 5, 6, 7, 8, 0] Heuristic: 0

Goal state achieved: [1, 2, 3, 4, 5, 6, 7, 8, 0]

Lab Program 7:

Aim:

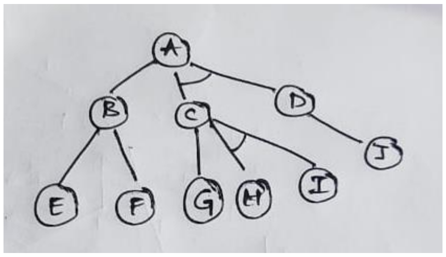
Implement AO* algorithm for general graph problem

Program:

```
def Cost(H, condition, weight=1):
    cost = {}
    if 'AND' in condition:
        AND_nodes = condition['AND']
        Path_A = ' AND '.join(AND_nodes)
        PathA = sum(H[node] + weight for node in AND_nodes)
        cost[Path_A] = PathA
    if 'OR' in condition:
        OR_nodes = condition['OR']
        Path_B = ' OR '.join(OR_nodes)
        PathB = min(H[node] + weight for node in OR_nodes)
        cost[Path_B] = PathB
    return cost
def update_cost(H, Conditions, weight=1):
    Main_nodes = list(Conditions.keys())
    Main_nodes.reverse()
    least_cost = {}
    for key in Main_nodes:
        condition = Conditions[key]
        print(key, ':', Conditions[key], '>>>', Cost(H, condition, weight))
        c = Cost(H, condition, weight)
        H[key] = min(c.values())
        least_cost[key] = Cost(H, condition, weight)
    return least_cost
def shortest_path(Start, Updated_cost, H):
    Path = Start
    if Start in Updated_cost.keys():
        Min_cost = min(Updated_cost[Start].values())
        key = list(Updated_cost[Start].keys())
        values = list(Updated_cost[Start].values())
        Index = values.index(Min_cost)
        Next = key[Index].split()
        if len(Next) == 1:
            Start = Next[0]
            Path += '<--' + shortest_path(Start, Updated_cost, H)
        else:
            Path += '<--(' + key[Index] + ') '
            Start = Next[0]
            Path += '[' + shortest_path(Start, Updated_cost, H) + ' + '
            Start = Next[-1]
            Path += shortest_path(Start, Updated_cost, H) + ']'
    return Path
H = eval(input('Enter nodes with heuristic costs: '))
Conditions = eval(input('Enter graph: '))
weight = 1
print('Updated Cost:')
```

Updated_cost = update_cost(H, Conditions, weight=1)
print('Shortest Path:\n', shortest_path('A', Updated_cost, H))

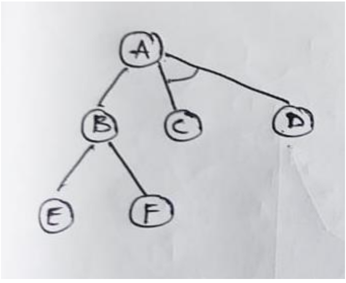
Graph 1:



Output 1:

Enter nodes with heuristic costs: {'A':-1,'B':5,'C':2,'D':4,'E':7,'F':9,'G':3,'H':0 ,'I':0,'J':0}
Enter graph: {'A':{'OR':['B'],'AND':['C','D']},'B':{'OR':['E','F']},'C':{'OR':['G'] ,'AND':['H','I']},'D':{'OR':['J']}}
Updated Cost:
D : {'OR': ['J']} >>> {'J': 1}
C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'H AND I': 2, 'G': 4}
B : {'OR': ['E', 'F']} >>> {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 5, 'B': 9}
Shortest Path:
A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]

Graph 2:



Output 2:

Enter nodes with heuristic costs: {'A':-1,'B':5,'C':2,'D':4,'E':7,'F':9}
Enter graph: {'A':{'OR':['B'],'AND':['C','D']},'B':{'OR':['E','F']}}
Updated Cost:
B : {'OR': ['E', 'F']} >>> {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 8, 'B': 9}
Shortest Path:
A<--(C AND D) [C + D]

Lab Program 8a:

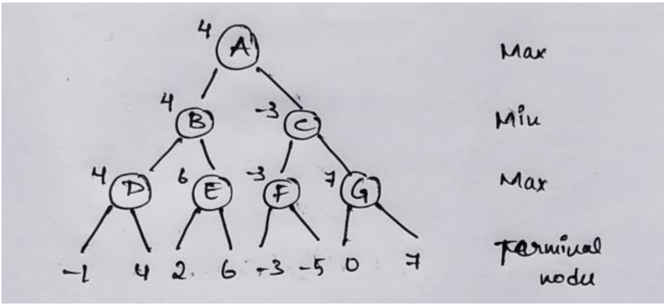
Aim:

Implement Game trees using
a. MINIMAX algorithm

Program:

```
import math
def minimax(game, state):
    def maxvalue(game, state):
        if state not in game:
            return state
        v = -math.inf
        for a in game[state]:
            v=max(v,minvalue(game,a))
        print("Max value for ",state," : ",v)
        return v
    def minvalue(game,state):
        if state not in game:
            return state
        v = math.inf
        for a in game[state]:
            v = min(v,maxvalue(game,a))
        print("Min value for ",state," : ",v)
        return v
    return max([minvalue(game,a) for a in game[state]])
graph = eval(input("Enter graph: "))
ans = minimax(graph,'A')
print("The optimal value is: ",ans)
```

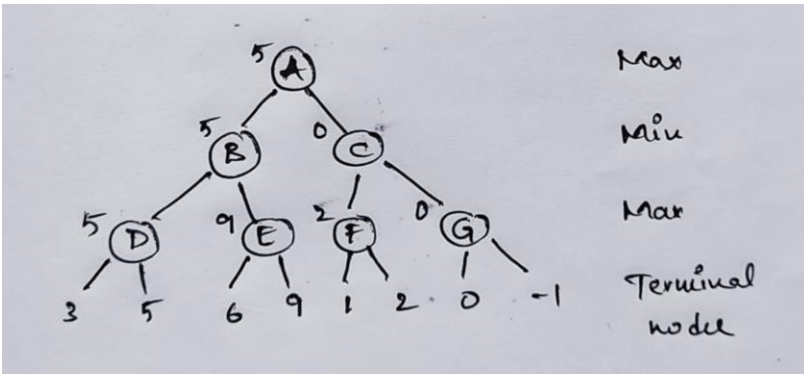
Graph 1:



Output 1:

```
Enter graph: {'A':['B','C'],'B':['D','E'],'C':['F','G'],'D':[-1,4],'E':[2,6],'F':[-3,-5],'G':[0,7]}
Max value for  D  :  4
Max value for  E  :  6
Min value for  B  :  4
Max value for  F  :  -3
Max value for  G  :  7
Min value for  C  :  -3
The optimal value is:  4
```

Graph 2:



Output 2:

Enter graph: {'A':['B','C'],'B':['D','E'],'C':['F','G'],'D':[3,5],'E':[6,9],'F':[1,2],'G':[0,-1]}
Max value for D : 5
Max value for E : 9
Min value for B : 5
Max value for F : 2
Max value for G : 0
Min value for C : 0
The optimal value is: 5

Lab Program 8b:

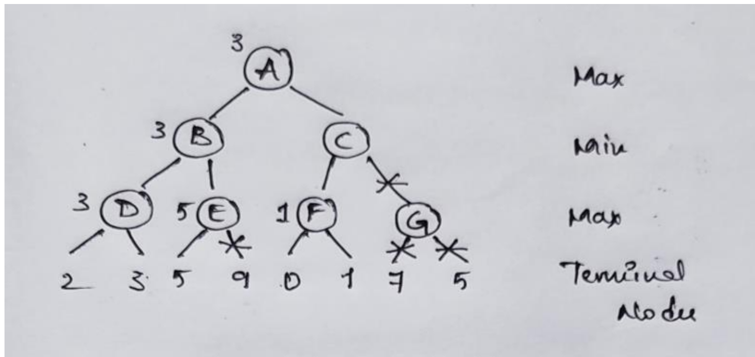
Aim:

Implement Game trees using
b. Alpha-Beta pruning

Program:

```
import math
def alphaBeta(game, state):
    def maxvalue(game, state,alpha,beta):
        if state not in game:
            return state
        v = -math.inf
        for a in game[state]:
            v=max(v,minvalue(game,a,alpha,beta))
            if v >= beta:
                print("Pruned subtree at state:", state, "with value:", v)
                return v
            alpha = max(alpha,v)
        print("State: ",state," V: ",v," Alpha: ",alpha," Beta: ",beta)
        return v
    def minvalue(game,state,alpha,beta):
        if state not in game:
            return state
        v = math.inf
        for a in game[state]:
            v=min(v,maxvalue(game,a,alpha,beta))
            if v <= alpha:
                print("Pruned subtree at state:", state, "with value:", v)
                return v
            beta = min(beta,v)
        print("State: ",state," V: ",v," Alpha: ",alpha," Beta: ",beta)
        return v
    alpha = -math.inf
    beta = math.inf
    v = maxvalue(game,state,alpha,beta)
    return v
graph = eval(input("Enter graph: "))
ans = alphaBeta(graph,'A')
print("The optimal value is: ",ans)
```

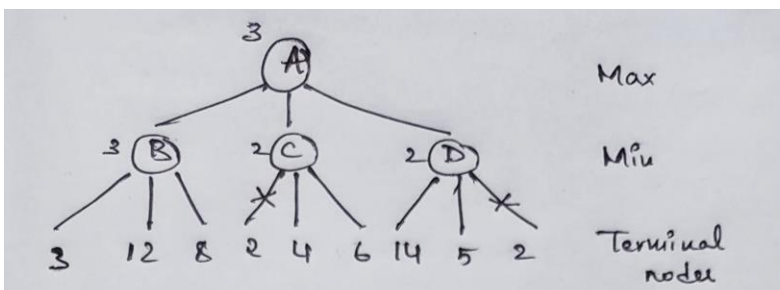
Graph 1:



Output 1:

Enter graph: {'A':['B','C'],'B':['D','E'],'C':['F','G'],'D':[2,3],'E':[5,9],'F':[0,1],'G':[7,5]}
State: D V: 3 Alpha: 3 Beta: inf
Pruned subtree at state: E with value: 5
State: B V: 3 Alpha: -inf Beta: 3
State: F V: 1 Alpha: 3 Beta: inf
Pruned subtree at state: C with value: 1
State: A V: 3 Alpha: 3 Beta: inf
The optimal value is: 3

Graph 2:



Output 2:

Enter graph: {'A':['B','C','D'],'B':[3,12,8],'C':[2,4,6],'D':[14,5,2]}
State: B V: 3 Alpha: -inf Beta: 3
Pruned subtree at state: C with value: 2
Pruned subtree at state: D with value: 2
State: A V: 3 Alpha: 3 Beta: inf
The optimal value is: 3

Lab Program 9:

Aim:

Implement Crypt arithmetic problems.

Program:

```
import itertools
def number(n, d):
    t = 0
    for i in n:
        t = d[i] + (t * 10)
    return t
def test(l, s, d):
    total = 0
    for i in l: total += number(i, d)
    return total == number(s, d)
def check(d, c):
    for i in d.keys():
        if i in c and d[i] == 0:
            return True
    return False
def solve(l, s):
    c = [i[0] for i in l] + [s[0]]
    p = list(set(''.join(l) + s))
    q = len(p)
    permutations = itertools.permutations(range(0, 10), q)
    for perm in permutations:
        d = {p[j]: perm[j] for j in range(q)}
        if check(d, c):
            continue
        if test(l, s, d):  return d
    return None
if __name__ == "__main__":
    l = input('Enter list of strings: ').split()
    s = input('Enter output string: ')
    solution = solve(l, s)
    if solution:
        print(solution)
        print('Solution found')
    else:
        print('No solution found')
```

Output 1:

Enter list of strings: TWO TWO
Enter output string: FOUR
{'O': 4, 'F': 1, 'T': 7, 'U': 6, 'R': 8, 'W': 3}
Solution found

Output 2:

Enter list of strings: SEND MORE
Enter output string: MONEY
{'O': 0, 'S': 9, 'E': 5, 'R': 8, 'Y': 2, 'M': 1, 'D': 7, 'N': 6}
Solution found