1.a) Aim: Implement Exhaustive search techniques using BFS

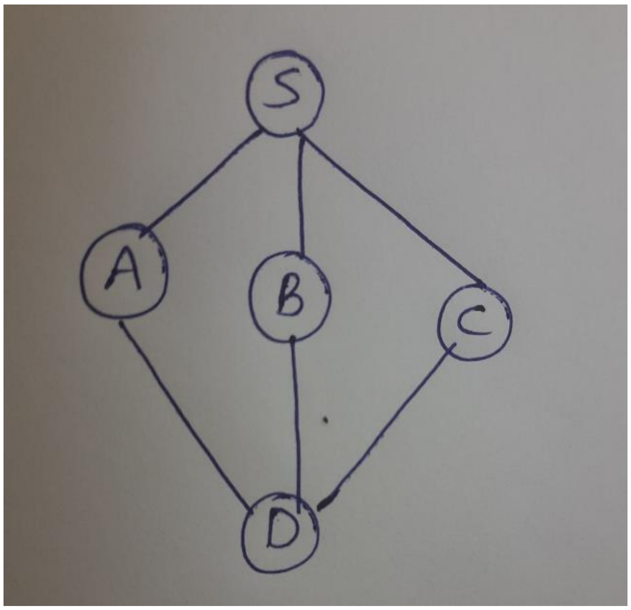
Source code:

```
def bfs(graph,start,goal):
    explored=[]
    frontier=[]
    explored.append(start)
    frontier.append(start)
    while frontier:
      node=frontier.pop(0)
      print(node, end=" ")
      for neighbor in graph[node]
        if neighbor not in explored:
          explored.append(neighbor)
          frontier.append(neighbor)
        print('\nfrontier:',frontier)
        print('explored:',explored)
        if neighbor==goal:
          print("\n goal reached:",neighbor)
          return
if __name__ =='__main__':
  #graph={ 0:[1,2], 1:[2], 2:[0,3],3:[3]}
  #dfs(graph,'A')
  graph=eval(input('enter the graph'))
  source=input('enter the source node')
  dest=input('enter the destination node')
  bfs(graph,source,dest)
```

Output1:

```
enter
graph{'a':['s','d'],'b':['s','d'],'s':['a','b','c'],'c':['s','d'],'d':['a','b
','c']}
enter source nodes
enter destination noded
s
frontier: ['a']
explored: ['s', 'a']

frontier: ['a', 'b']
explored: ['s', 'a', 'b']

frontier: ['a', 'b', 'c']
explored: ['s', 'a', 'b', 'c']
a
frontier: ['b', 'c']
explored: ['s', 'a', 'b', 'c']

frontier: ['b', 'c', 'd']
explored: ['s', 'a', 'b', 'c', 'd']

 goal reached: d
```

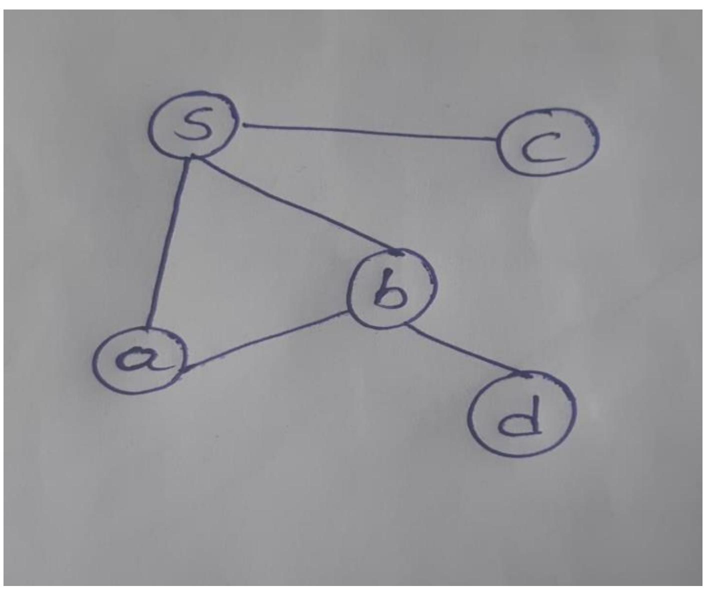
Graph:



Output2:

```
enter
graph{'s':['a','b','c'],'b':['s','a','d'],'a':['s','b'],'c':['s'],'d':['b']}
enter source nodes
enter destination noded
s
frontier: ['a']
explored: ['s', 'a']

frontier: ['a', 'b']
explored: ['s', 'a', 'b']

frontier: ['a', 'b', 'c']
explored: ['s', 'a', 'b', 'c']
a
frontier: ['b', 'c']
explored: ['s', 'a', 'b', 'c']

frontier: ['b', 'c']
explored: ['s', 'a', 'b', 'c']
b
frontier: ['c']
explored: ['s', 'a', 'b', 'c']
frontier: ['c']
explored: ['s', 'a', 'b', 'c']

frontier: ['c', 'd']
explored: ['s', 'a', 'b', 'c', 'd']

 goal reached: d
```

Graph



1.b) Aim: Implement Exhaustive search techniques using DFS

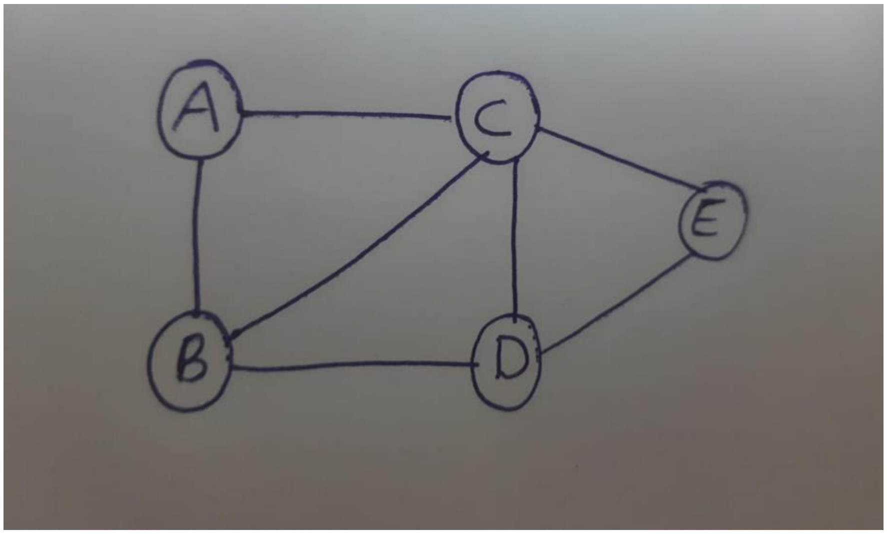
Source code:

```
def dfs(graph,start,goal):
    explored=[]
    frontier=[]
    explored.append(start)
    frontier.append(start)
    while frontier:
        node=frontier.pop()
        print(node, end=" ")
        for neighbor in graph[node]:
            if neighbor not in explored:
                explored.append(neighbor)
                frontier.append(neighbor)
            print('\nfrontier:',frontier)
            print('explored:',explored)
            if neighbor==goal:
                    print("\n goal reached:",neighbor)
                    return
if __name__ =='__main__':
    #graph={ 0:[1,2], 1:[2], 2:[0,3],3:[3]}
    #dfs(graph,'A')
    graph=eval(input('enter the graph:'))
    source=input('enter the source node')
    dest=input('enter the destination node')
    dfs(graph,'B','E')
```

Output1:

```
enter the
graph{'A':['B','C'],'B':['A','C','D'],'C':['A','D','E'],'D':['B','C','E'],'E'
:['C','D']}
enter the source nodeB
enter the destination nodeE
B
frontier: ['A']
explored: ['B', 'A']

frontier: ['A', 'C']
explored: ['B', 'A', 'C']

frontier: ['A', 'C', 'D']
explored: ['B', 'A', 'C', 'D']
D
frontier: ['A', 'C']
explored: ['B', 'A', 'C', 'D']

frontier: ['A', 'C']
explored: ['B', 'A', 'C', 'D']

frontier: ['A', 'C', 'E']
explored: ['B', 'A', 'C', 'D', 'E']

 goal reached: E
```

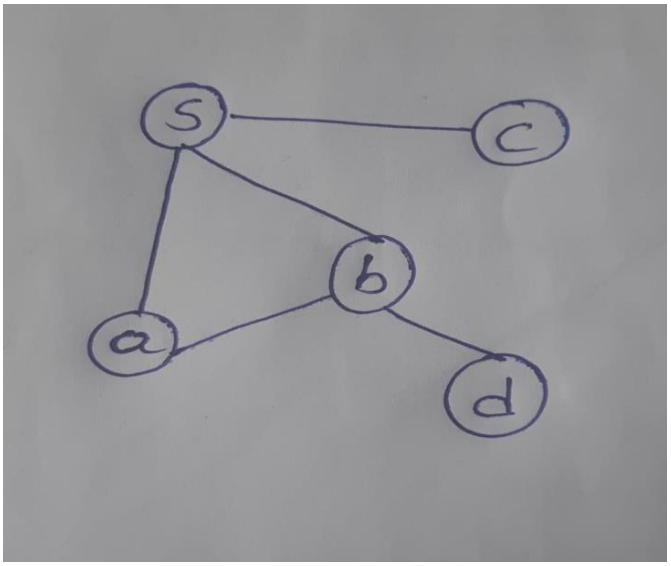
Graph:



Output2:

```
enter the
graph{'s':['a','b','c'],'b':['s','a','d'],'a':['s','b'],'c':['s'],'d':['b']}
enter the source nodes
enter the destination noded
s
frontier: ['a']
explored: ['s', 'a']

frontier: ['a', 'b']
explored: ['s', 'a', 'b']

frontier: ['a', 'b', 'c']
explored: ['s', 'a', 'b', 'c']
c
frontier: ['a', 'b']
explored: ['s', 'a', 'b', 'c']
b
frontier: ['a']
explored: ['s', 'a', 'b', 'c']

frontier: ['a']
explored: ['s', 'a', 'b', 'c']

frontier: ['a', 'd']
explored: ['s', 'a', 'b', 'c', 'd']

 goal reached: d
```

Graph:



1.c) Implement Exhaustive search techniques using Uniform Cost Search

Source code:

```
def ucs(graph,s,goal):
    frontier={s:0}
    explored=[]
    while True:
        if(len(frontier))==0:
            return 'FAIL'
        node=min(frontier,key=frontier.get)
        val=frontier[node]
        print(node,':',val)
        del frontier[node]
       # print('frontier:',frontier)
        #print('explored:',explored)
        if goal==node:
            return f"Goal reached with cost:{val}"
        explored.append(node)
        for neighbor,pathcost in graph[node].items():
            if neighbor not in explored or neighbor not in frontier:
                frontier.update({neighbor:val+pathcost})
            elif neigbor in frontier and pathcost>val:
                frontier.update({neighbor:val})
        print('frontier:',frontier)
        print('explored:',explored)
graph=eval(input('enter the graph'))
```

source=input('enter the source node')

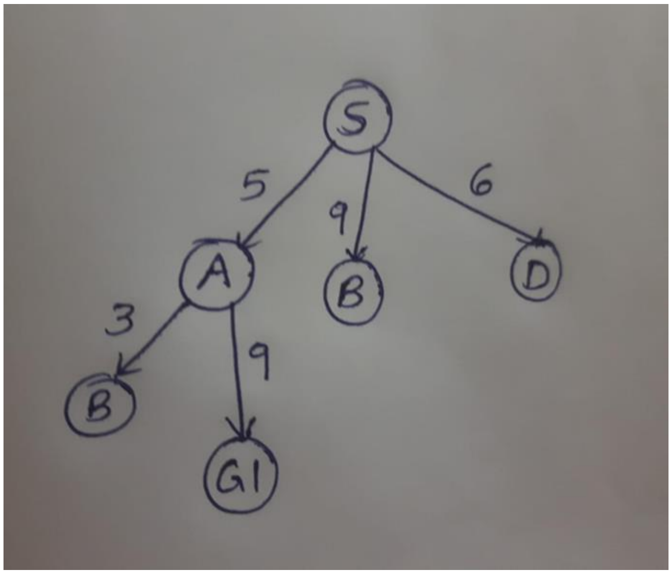
dest=input('enter the destination node')

print(ucs(graph,source,dest))

Output1:

```
enter the graph{'s':{'a':5,'b':9,'d':6},'a':{'b':3,'g1':9},'d':{},'b':{}}
enter the source nodes
enter the destination nodeg1
s : 0
frontier: {'a': 5, 'b': 9, 'd': 6}
explored: ['s']
a : 5
frontier: {'b': 8, 'd': 6, 'g1': 14}
explored: ['s', 'a']
d : 6
frontier: {'b': 8, 'g1': 14}
explored: ['s', 'a', 'd']
b : 8
frontier: {'g1': 14}
explored: ['s', 'a', 'd', 'b']
g1 : 14
Goal reached with cost:14
```

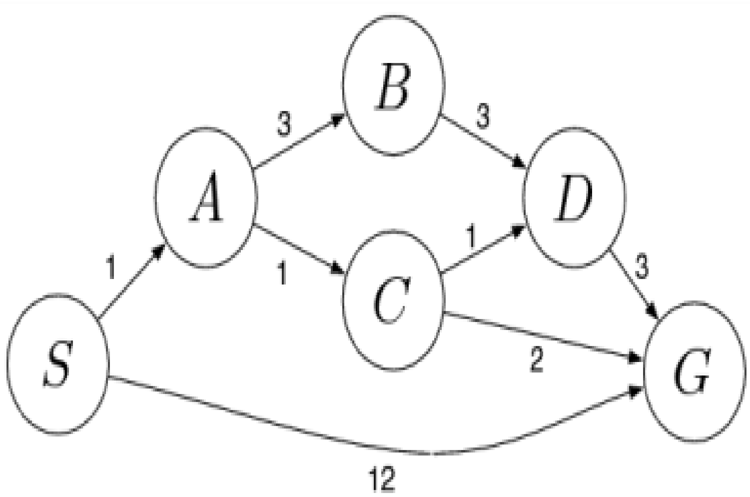
Graph:



Output2:

```
enter the
graph{'S':{'A':1,'G':12},'A':{'B':3,'C':1},'B':{'D':3},'C':{'D':1,'G':2},'D':
{'G':3}}
enter the source nodeS
enter the destination nodeD
S : 0
frontier: {'A': 1, 'G': 12}
explored: ['S']
A : 1
frontier: {'G': 12, 'B': 4, 'C': 2}
explored: ['S', 'A']
C : 2
frontier: {'G': 4, 'B': 4, 'D': 3}
explored: ['S', 'A', 'C']
D : 3
Goal reached with cost:3
```

Graph:



1.d) Aim : Implement Exhaustive search techniques using Depth-First Iterative Deepening.

<u>Source code:</u>

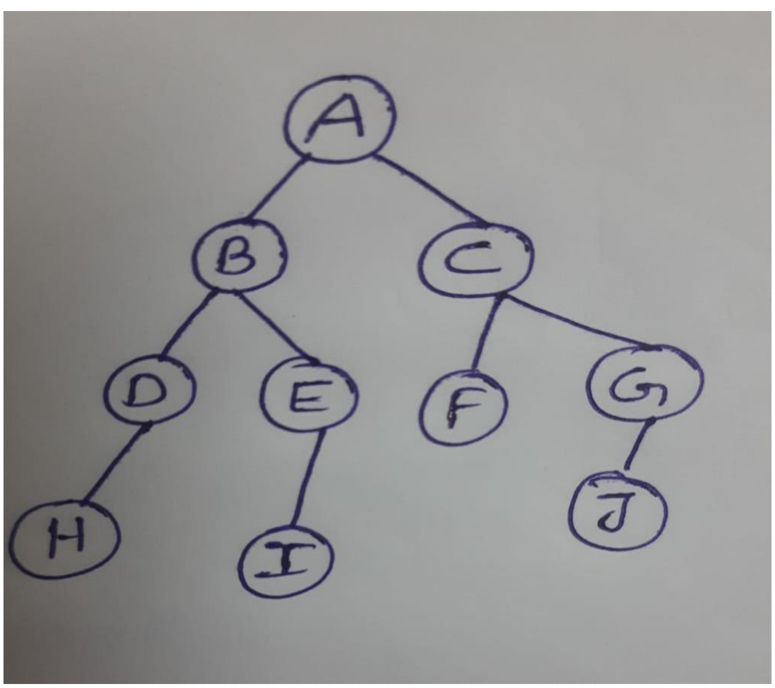
```
# Define the graph
g = eval(input('Enter graph: '))
# Input source, target, and depth limit
s = int(input('Enter source: '))
t = int(input('Enter target: '))
depth = int(input('Enter depth: '))
def DFS(d):
    visited = [s]
    stack = [s]
    check = [0]
    while stack:
        f = stack.pop()
        print(f, end=' ')
        p = check.pop()
        if f == t:
            print(' Target found within given depth')
            return True
        if p + 1 > d:
            continue
        for neighbor in g[f]:
            if neighbor not in visited:
                check.append(p + 1)
                visited.append(neighbor)
```

```
                stack.append(neighbor)
    return False
def IDDFS():
    for i in range(depth + 1):
        print('Depth', i, ': ', end='')
        if DFS(i):
            return
    print('Target not found within given depth')
IDDFS()
```

Output1:

```
Enter graph:
{'A':['B','C'],'B':['D','E'],'C':['F','G'],'D':['H'],'E':['I'],'G':['J']}
Enter source: A
Enter target: G
Enter depth: 4
Depth 0 : A Depth 1 : A C B Depth 2 : A C G  Target found within given depth
```

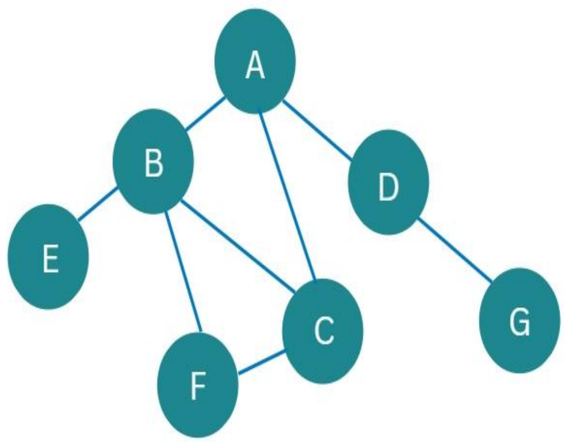
Graph:



Output2

```
Enter graph:
{'A':['B','C','D'],'B':['E','C','F'],'C':['A','B','F'],'D':['A','G'],'E':['B'
],'F':['B','C'],'G':['D']}
Enter source: A
Enter target: F
Enter depth: 1
Depth 0 : A Depth 1 : A D C B Target not found within given depth
```

Graph



1.e) Aim: Implement Exhaustive search techniques using Bidirectional search

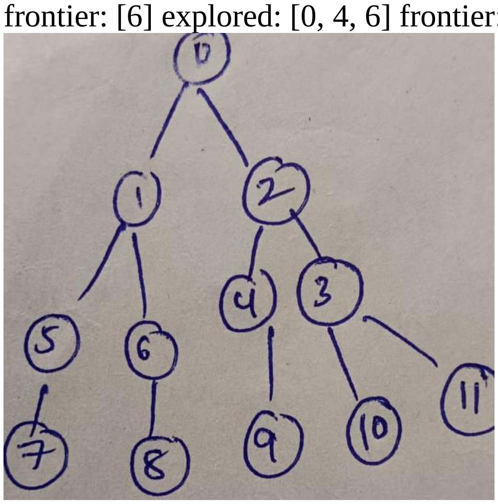
Source code:

```
def bfs(direction,graph,frontier,reached):
    if direction=='F':
        d='c'
    elif direction=='B':
        d='p'
    node=frontier.pop(0)
    for child in graph[node][d]:
        if child not in reached:
            reached.append(child)
            frontier.append(child)
        print("frontier:",frontier)
        print("explored:",reached)
        return frontier,reached
def isintersecting(rf,rb):
    i=set(rf).intersection(set(rb))
    return list(i)[0] if i else -1
def bds(graph,source,dest):
    ff=[source]
    fb=[dest]
    rf=[source]
    rb=[dest]
    while ff and fb:
        ff,rf=bfs('F',graph,ff,rf)
```

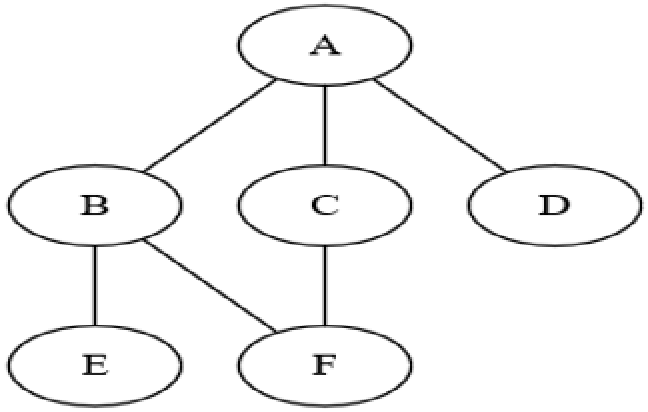
```
        fb,rb=bfs('B',graph,fb,rb)
        inode=isintersecting(rf,rb)
        if inode!=-1:
            print("path found:")
            path=rf[:-1]+rb[::-1]
            return path
    print("no path found")
    return[]
g={0:{'c':[4],'p':[]},1:{'c':[4],'p':[]},2:{'c':[6],'p':[0,1]},3:{'c':[5],'p':[1,2]},4:{'c':[6],'p':[2,3]},5:{'c'
:[7],'p':[1,2]},6:{'c':[7],'p':[4,5]},7:{'c':[8],'p':[6]},8:{'c':[9],'p':[0,2]},9:{'c':[9,10],'p':[7,8]}}
s=int(input("enter the staring state:"))
go=int(input("enter the goal state"))
print(bds(g,s,go))
```

Output1:

```
enter the staring state:0
enter the goal state7
frontier: [4] explored: [0, 4]
frontier: [6] explored: [7, 6]
frontier: [6] explored: [0, 4, 6] frontier: [4] explored: [7, 6, 4]path found: [0, 4, 4, 6, 7]
```



Output2:




```
Enter number of nodes in graph: 7
A B C D E F G are the nodes
Enter the child nodes of A: B C
Enter the child nodes of B: D
Enter the child nodes of C: D E
Enter the child nodes of D: F
Enter the child nodes of E: F G
Enter the child nodes of F: 0
Enter the child nodes of G: 0
Enter source node: A
Enter goal node: G
From front:
      Frontier: ['A']
      Reached: ['A']
From back:
      Frontier: ['G']
      Reached: ['G']
Path found!
Path: ['A', 'C', 'E', 'G']
```

2.a) Aim: Implement water jug problem with Search tree generation using bfs

Source code:

```
from collections import deque

x_capacity = int(input("Enter Jug 1 capacity: "))

y_capacity = int(input("Enter Jug 2 capacity: "))

end = int(input("Enter target volume: "))

def bfs(start, end, x_capacity, y_capacity):

    path = []

    front = deque()

    front.append(start)

    visited = []

    frontier = []  # Frontier to keep track of nodes being explored

    explored = {}  # Explored graph to keep track of explored nodes and their parents

    while front:

        current = front.popleft()

        frontier.append(current)  # Add the node to the frontier

        visited.append(current)

        x = current[0]

        y = current[1]

        path.append(current)

        if x == end or y == end:

            print("Found")

            print("Path:", path)

            print("Path Cost:", len(path) - 1)

            print("Frontier:", frontier)
```

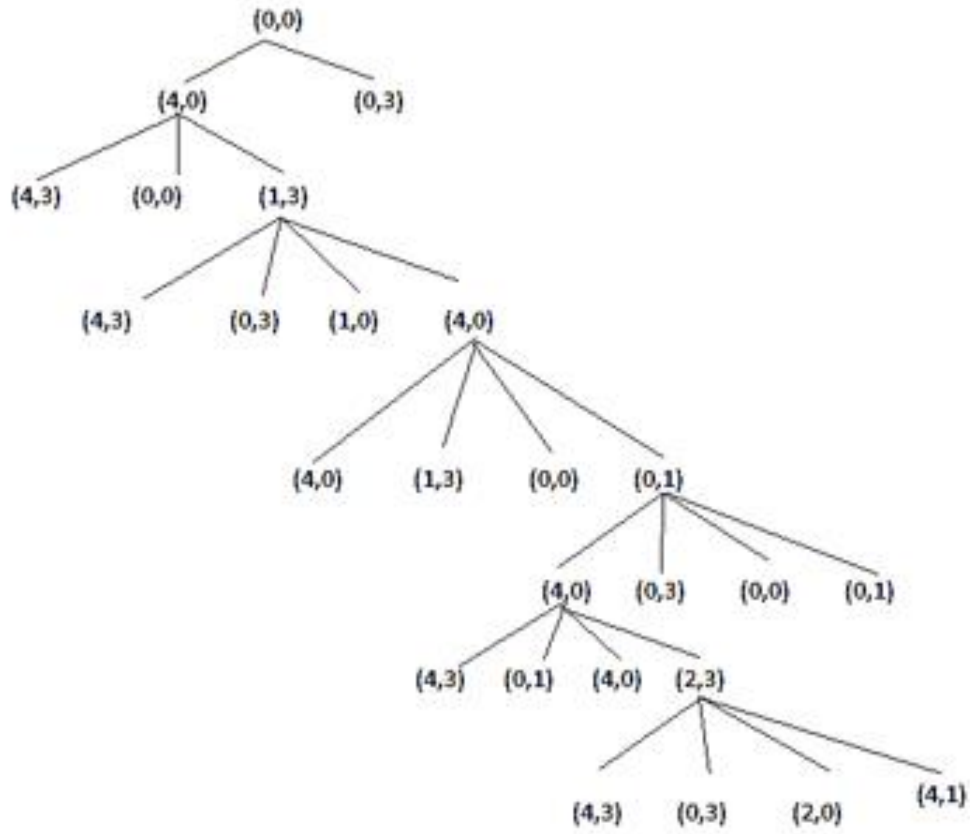
```
    print("Explored graph:", explored)
    return path
# rule 1
if current[0] < x_capacity and ([x_capacity, current[1]] not in visited):
    front.append([x_capacity, current[1]])
    visited.append([x_capacity, current[1]])
    explored[(x_capacity, current[1])] = current
# rule 2
if current[0] > 0 and ([0, current[1]] not in visited):
    front.append([0, current[1]])
    visited.append([0, current[1]])
    explored[(0, current[1])] = current
# rule 3
if current[1] < y_capacity and ([current[0], y_capacity] not in visited):
    front.append([current[0], y_capacity])
    visited.append([current[0], y_capacity])
    explored[(current[0], y_capacity)] = current
# rule 4
if current[1] > 0 and ([current[0], 0] not in visited):
    front.append([current[0], 0])
    visited.append([current[0], 0])
    explored[(current[0], 0)] = current
# rule 5
if (current[0] + current[1]) <= x_capacity and current[1] > 0 and \
        ([current[0] + current[1], 0] not in visited):
    front.append([current[0] + current[1], 0])
    visited.append([current[0] + current[1], 0])
```

```
            explored[(current[0] + current[1], 0)] = current
        # rule 6
        if (current[0] + current[1]) <= y_capacity and current[0] > 0 and \
                ([0, current[0] + current[1]] not in visited):
            front.append([0, current[0] + current[1]])
            visited.append([0, current[0] + current[1]])
            explored[(0, current[0] + current[1])] = current
        # rule 7
        if (current[0] + current[1]) >= x_capacity and current[1] > 0 and \
                ([x_capacity, current[1] - (x_capacity - current[0])] not in visited):
            front.append([x_capacity, current[1] - (x_capacity - current[0])])
            visited.append([x_capacity, current[1] - (x_capacity - current[0])])
            explored[(x_capacity, current[1] - (x_capacity - current[0]))] = current
        # rule 8:
        if (current[0] + current[1]) >= y_capacity and current[0] > 0 and \
                ([current[0] - (y_capacity - current[1]), y_capacity] not in visited):
            front.append([current[0] - (y_capacity - current[1]), y_capacity])
            visited.append([current[0] - (y_capacity - current[1]), y_capacity])
            explored[(current[0] - (y_capacity - current[1]), y_capacity)] = current
    return "Not found"
start = [0, 0]
print(bfs(start, end, x_capacity, y_capacity))
```

Output1:

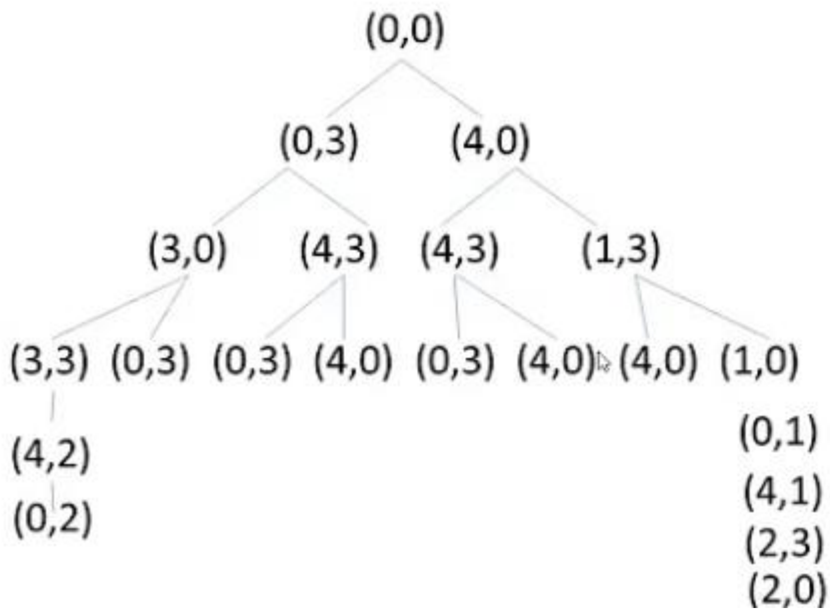
```
Enter Jug 1 capacity:  5
Enter Jug 2 capacity:  3
Enter target volume:  2
Found
Path: [[0, 0], [5, 0], [0, 3], [5, 3], [2, 3]]
Path Cost: 4
Frontier: [[0, 0], [5, 0], [0, 3], [5, 3], [2, 3]]
```

Explored graph: {(5, 0): [0, 0], (0, 3): [0, 0], (5, 3): [5, 0], (2, 3): [5, 0], (3, 0): [0, 3]}
[[0, 0], [5, 0], [0, 3], [5, 3], [2, 3]]



<u>Output2:</u>

```
Enter Jug 1 capacity: 4
Enter Jug 2 capacity: 3
Enter target volume: 2
Found
Path: [[0, 0], [4, 0], [0, 3], [4, 3], [1, 3], [3, 0], [1, 0], [3, 3], [0,
1], [4, 2]]
Path Cost: 9
Frontier: [[0, 0], [4, 0], [0, 3], [4, 3], [1, 3], [3, 0], [1, 0], [3, 3],
[0, 1], [4, 2]]
Explored graph: {(4, 0): [0, 0], (0, 3): [0, 0], (4, 3): [4, 0], (1, 3): [4,
0], (3, 0): [0, 3], (1, 0): [1, 3], (3, 3): [3, 0], (0, 1): [1, 0], (4, 2):
[3, 3], (4, 1): [0, 1]}
[[0, 0], [4, 0], [0, 3], [4, 3], [1, 3], [3, 0], [1, 0], [3, 3], [0, 1], [4,
2]]
```


(0,0)
(0,3)
(4,0)
(3,0)
(4,3)
(4,3)
(1,3)
(3,3)
(0,3)
(0,3)
(4,0)
(0,3)
(4,0)
(4,0)
(1,0)
(4,2)
(0,2)
(0,1)
(4,1)
(2,3)
(2,0)

Date: (12-02-2024)

2.b) Aim: Implement water jug problem with Search tree generation using dfs

Source code:

```
x_capacity = int(input("Enter Jug 1 capacity: "))
y_capacity = int(input("Enter Jug 2 capacity: "))
end = int(input("Enter target volume: "))
def dfs(start, end, x_capacity, y_capacity):
    path = []
    front = []
    front.append(start)
    visited = []
    while front:
        current = front.pop()
        x = current[0]
        y = current[1]
        path.append(current)
        if x == end or y == end:
            print("Found")
            print("Path:", path)
            print("Path Cost:", len(path) - 1)
            print("Frontier:", front)
            print("Explored graph:", visited)
            return path
        # Rule 1: Fill Jug 1
        if x < x_capacity and [x_capacity, y] not in visited:
            front.append([x_capacity, y])
```

```
        visited.append([x_capacity, y])
    # Rule 2: Empty Jug 1
    if x > 0 and [0, y] not in visited:
        front.append([0, y])
        visited.append([0, y])
    # Rule 3: Fill Jug 2
    if y < y_capacity and [x, y_capacity] not in visited:
        front.append([x, y_capacity])
        visited.append([x, y_capacity])
    # Rule 4: Empty Jug 2
    if y > 0 and [x, 0] not in visited:
        front.append([x, 0])
        visited.append([x, 0])
    # Rule 5: Pour water from Jug 2 to Jug 1 until Jug 1 is full or Jug 2 is empty
    if (x + y) <= x_capacity and y > 0 and [x + y, 0] not in visited:
        front.append([x + y, 0])
        visited.append([x + y, 0])
    # Rule 6: Pour water from Jug 1 to Jug 2 until Jug 2 is full or Jug 1 is empty
    if (x + y) <= y_capacity and x > 0 and [0, x + y] not in visited:
        front.append([0, x + y])
        visited.append([0, x + y])
    # Rule 7: Pour water from Jug 2 to Jug 1 until Jug 1 is full or Jug 2 is empty
    if (x + y) >= x_capacity and y > 0 and [x_capacity, y - (x_capacity - x)] not in visited:
        front.append([x_capacity, y - (x_capacity - x)])
        visited.append([x_capacity, y - (x_capacity - x)])
    # Rule 8: Pour water from Jug 1 to Jug 2 until Jug 2 is full or Jug 1 is empty
    if (x + y) >= y_capacity and x > 0 and [x - (y_capacity - y), y_capacity] not in visited:
```

```
            front.append([x - (y_capacity - y), y_capacity])
            visited.append([x - (y_capacity - y), y_capacity])
    return "Not found"
def gcd(a, b):
    if a == 0:
        return b
    return gcd(b % a, a)
start = [0, 0]
if end % gcd(x_capacity, y_capacity) == 0:
    print(dfs(start, end, x_capacity, y_capacity))
else:
    print("No solution possible for this combination.")
```

Output1:

Enter Jug 1 capacity: 5

Enter Jug 2 capacity: 3

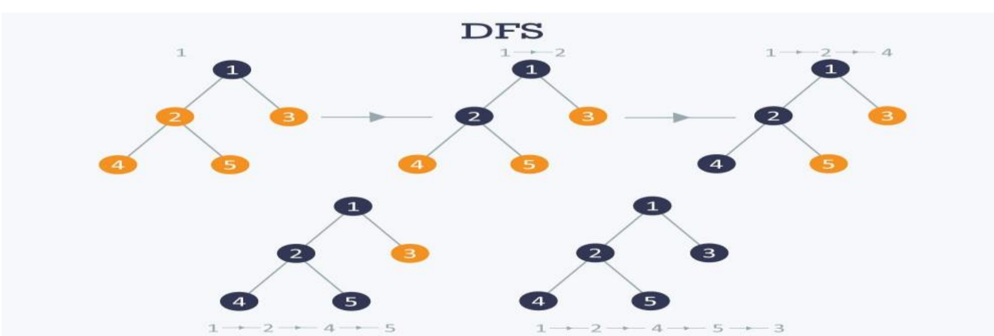
Enter target volume: 4

Found Path: [[0, 0], [0, 3], [3, 0], [3, 3], [5, 1], [0, 1], [1, 0], [1, 3], [4, 0]]

Path Cost: 8

Frontier: [[5, 0], [5, 3], [0, 0]] Explored graph: [[5, 0], [0, 3], [5, 3], [0, 0], [3, 0], [3, 3], [5, 1], [0, 1], [1, 0], [1, 3], [4, 0]] [[0, 0], [0, 3], [3, 0], [3, 3], [5, 1], [0, 1], [1, 0], [1, 3], [4, 0]]

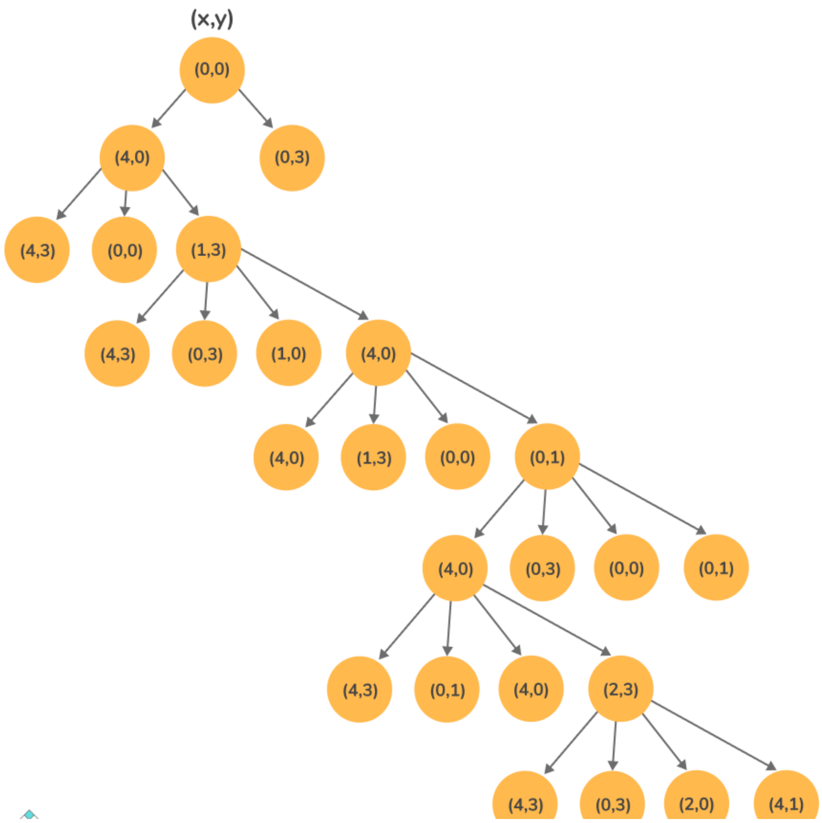
Graph:



Output2:

```
Enter Jug 1 capacity: 4
Enter Jug 2 capacity: 3
Enter target volume: 2
Found
Path: [[0, 0], [0, 3], [3, 0], [3, 3], [4, 2]]
Path Cost: 4
Frontier: [[4, 0], [4, 3], [0, 0]]
Explored graph: [[4, 0], [0, 3], [4, 3], [0, 0], [3, 0], [3, 3], [4, 2]]
[[0, 0], [0, 3], [3, 0], [3, 3], [4, 2]]
```

Graph:



3.a) Aim: Implement Missionaries and Cannibals problem with Search tree generation using BFS

Source code:

```
from collections import deque

def is_valid_state(state):

    m_left, c_left, m_right, c_right, boat = state

    if (m_left < 0 or c_left < 0 or m_right < 0 or c_right < 0 or

        (m_left != 0 and m_left < c_left) or

        (m_right != 0 and m_right < c_right)):

        return False

    return True

def generate_next_states(state):

    m_left, c_left, m_right, c_right, boat = state

    possible_states = []

    if boat == 'left':

        for m in range(3):

            for c in range(3):

                if m + c > 2 or m + c == 0:

                    continue

                new_state = (m_left - m, c_left - c, m_right + m, c_right + c, 'right')

                if is_valid_state(new_state):

                    possible_states.append(new_state)

    else:

        for m in range(3):

            for c in range(3):

                if m + c > 2 or m + c == 0:
```

```
                continue
            new_state = (m_left + m, c_left + c, m_right - m, c_right - c, 'left')
            if is_valid_state(new_state):
                possible_states.append(new_state)
    return possible_states
def bfs(start_state, goal_state):
    visited = set()
    queue = deque([(start_state, [])])
    while queue:
        current_state, path = queue.popleft()
        visited.add(current_state)
        if current_state == goal_state:
            return path
        for next_state in generate_next_states(current_state):
            if next_state not in visited:
                queue.append((next_state, path + [next_state]))
                visited.add(next_state)
    return []
start_state = eval(input(‘enter the start state’))
goal_state = eval(input(‘enter the goal state’))
print("BFS solution:")
solution = bfs(start_state, goal_state)
if solution:
    for i, state in enumerate(solution):
        print(f"Step {i+1}: {state}")
else:
    print("No solution found.")
```

Output1:

BFS solution:

Step 1: (3, 1, 0, 2, 'right')

Step 2: (3, 2, 0, 1, 'left')

Step 3: (3, 0, 0, 3, 'right')

Step 4: (3, 1, 0, 2, 'left')

Step 5: (1, 1, 2, 2, 'right')

Step 6: (2, 2, 1, 1, 'left')

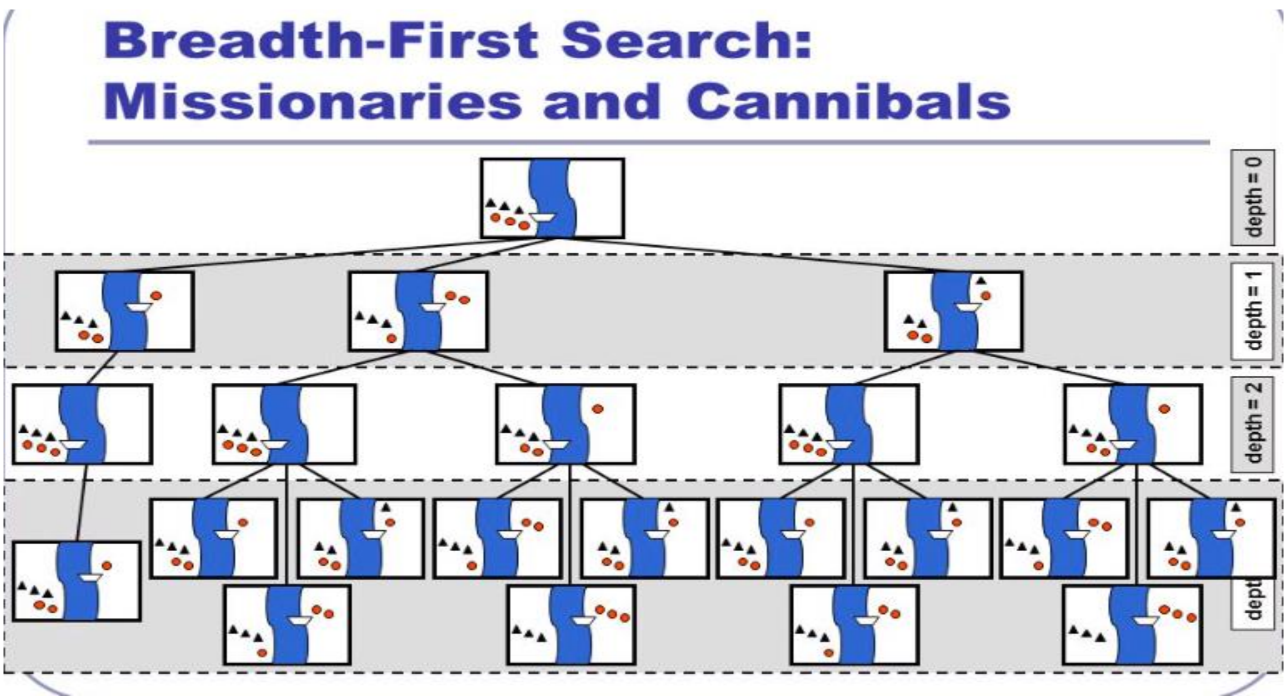
Step 7: (0, 2, 3, 1, 'right')

Step 8: (0, 3, 3, 0, 'left')

Step 9: (0, 1, 3, 2, 'right')

Step 10: (0, 2, 3, 1, 'left')

Step 11: (0, 0, 3, 3, 'right')



Output2:

```
enter the start state(4,4,0,0,'left')
enter the end state(0,0,4,4,'right')
BFS solution:
No solution found.
```

3.b) Aim : Implement Missionaries and Cannibals problem with Search tree generation using DFS

Source code:

```
from collections import deque

def is_valid_state(state):

    m_left, c_left, m_right, c_right, boat = state

    if (m_left < 0 or c_left < 0 or m_right < 0 or c_right < 0 or

            (m_left != 0 and m_left < c_left) or

            (m_right != 0 and m_right < c_right)):

        return False

    return True

def generate_next_states(state):

    m_left, c_left, m_right, c_right, boat = state

    possible_states = []

    if boat == 'left':

        for m in range(3):

            for c in range(3):

                if m + c > 2 or m + c == 0:

                    continue

                new_state = (m_left - m, c_left - c, m_right + m, c_right + c, 'right')

                if is_valid_state(new_state):

                    possible_states.append(new_state)

    else:

        for m in range(3):

            for c in range(3):

                if m + c > 2 or m + c == 0:
```

```
                continue
            new_state = (m_left + m, c_left + c, m_right - m, c_right - c, 'left')
            if is_valid_state(new_state):
                possible_states.append(new_state)
    return possible_states
def dfs(current_state, goal_state, path, visited):
    visited.add(current_state)
    if current_state == goal_state:
        return path
    for next_state in generate_next_states(current_state):
        if next_state not in visited:
            solution = dfs(next_state, goal_state, path + [next_state], visited)
            if solution:
                return solution
    return None
start_state = (3, 3, 0, 0, 'left')
goal_state = (0, 0, 3, 3, 'right')
visited = set()
print("dfs solution:")
solution = dfs(start_state, goal_state, [start_state], visited)
if solution:
    for i, state in enumerate(solution):
        print(f"Step {i+1}: {state}")
else:
    print("No solution found.")
```

Output1:

dfs solution:

Step 1: (3, 3, 0, 0, 'left')

Step 2: (3, 1, 0, 2, 'right')

Step 3: (3, 2, 0, 1, 'left')

Step 4: (3, 0, 0, 3, 'right')

Step 5: (3, 1, 0, 2, 'left')

Step 6: (1, 1, 2, 2, 'right')

Step 7: (2, 2, 1, 1, 'left')

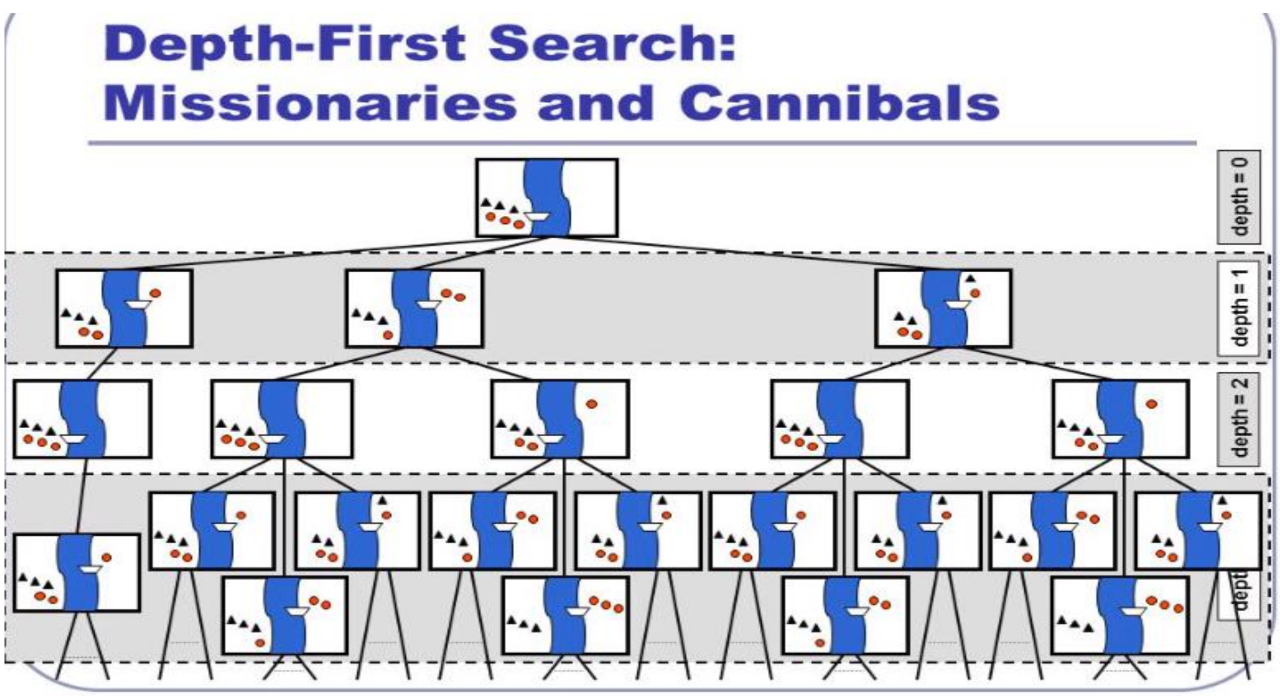
Step 8: (0, 2, 3, 1, 'right')

Step 9: (0, 3, 3, 0, 'left')

Step 10: (0, 1, 3, 2, 'right')

Step 11: (0, 2, 3, 1, 'left')

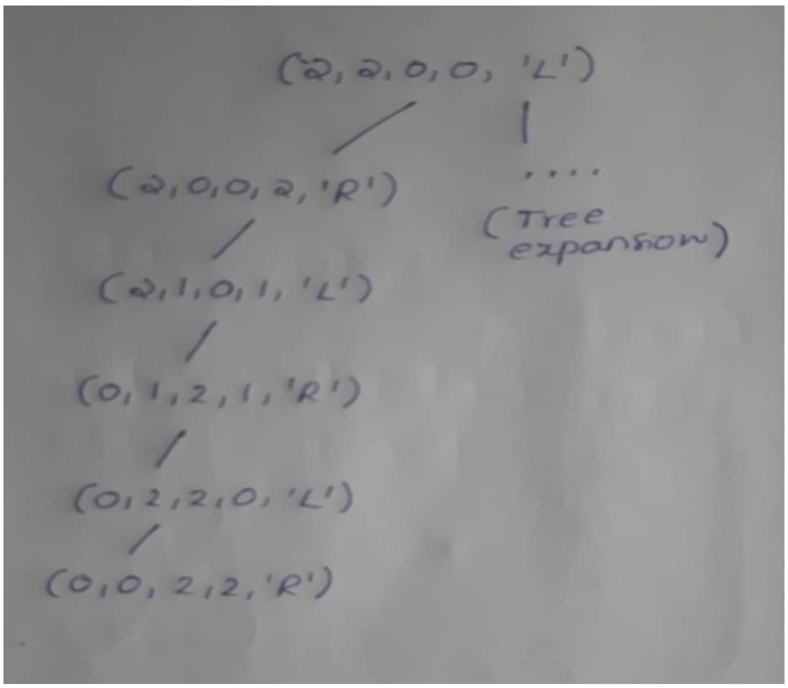
Step 12: (0, 0, 3, 3, 'right')



Output2:

```
enter the start state(2,2,0,0,'left')
enter the goal state(0,0,2,2,'right')
dfs solution:
Step 1: (2, 2, 0, 0, 'left')
Step 2: (2, 0, 0, 2, 'right')
Step 3: (2, 1, 0, 1, 'left')
Step 4: (0, 1, 2, 1, 'right')
Step 5: (0, 2, 2, 0, 'left')
Step 6: (0, 0, 2, 2, 'right')
```

Graph:



4.a) Aim: Implement Vacuum World problem with Search tree generation using BFS

Source code:

```
##vaccume cleaner problem using bfs
from queue import Queue
class State:
    def __init__(self, agent_position, room_a, room_b):
        self.agent_position = agent_position
        self.room_a = room_a
        self.room_b = room_b
        self.left = None
        self.right = None
        self.clean = None
    def get_states(self):
        next_states = []
        if self.left:
            next_states.append(self.left)
        if self.right:
            next_states.append(self.right)
        if self.clean:
            next_states.append(self.clean)
        return next_states
class Agent:
    def __init__(self, first_state, goal_state1, goal_state2):
        self.first_state = first_state
        self.goal_state1 = goal_state1
```

```
        self.goal_state2 = goal_state2
    def run_bfs(self):
        is_initial_state = False
        queue = Queue()
        checked = set()
        queue.put(self.first_state)
        checked.add(self.first_state)
        while not queue.empty():
            current = queue.get()
            if not is_initial_state:
                is_initial_state = True
                print("\nInitial State:", current.state_name)
            else:
                print("\nMove to state", current.state_name)
                self.prompt_attributes(current)
            if current == self.goal_state1 or current == self.goal_state2:
                print("Final State:", current.state_name)
                return True
            elif not current.get_states():
                print("Final state wasn't found or reached!")
                return False
            else:
                print("Next possible states:", [state.state_name for state in current.get_states()])
                for state in current.get_states():
                    if state not in checked:
                        queue.put(state)
                        checked.add(state)
```

```
    def prompt_attributes(self, current):
        print("Vacuum is in room", (current.agent_position))
        if current.room_a:
            print("Room A is clean")
        else:
            print("Room A is dirty")
        if current.room_b:
            print("Room B is clean")
        else:
            print("Room B is dirty")
def main():
    # Input Section
    print("Enter the initial state of each room where: 1 - Clean | 2 - Dirty")
    status_a = int(input("Enter the state of the first room: ")) == 1
    status_b = int(input("Enter the state of the second room: ")) == 1
    print("Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B")
    position = int(input("Enter the initial position: "))
    # Create state instances
    state1 = State(None, status_a, status_b)
    state2 = State(None, status_a, status_b)
    state3 = State(None, status_a, status_b)
    state4 = State(None, status_a, status_b)
    state5 = State(None, status_a, status_b)
    state6 = State(None, status_a, status_b)
    state7 = State(None, status_a, status_b)
    state8 = State(None, status_a, status_b)
    # Setup and connect each state
```

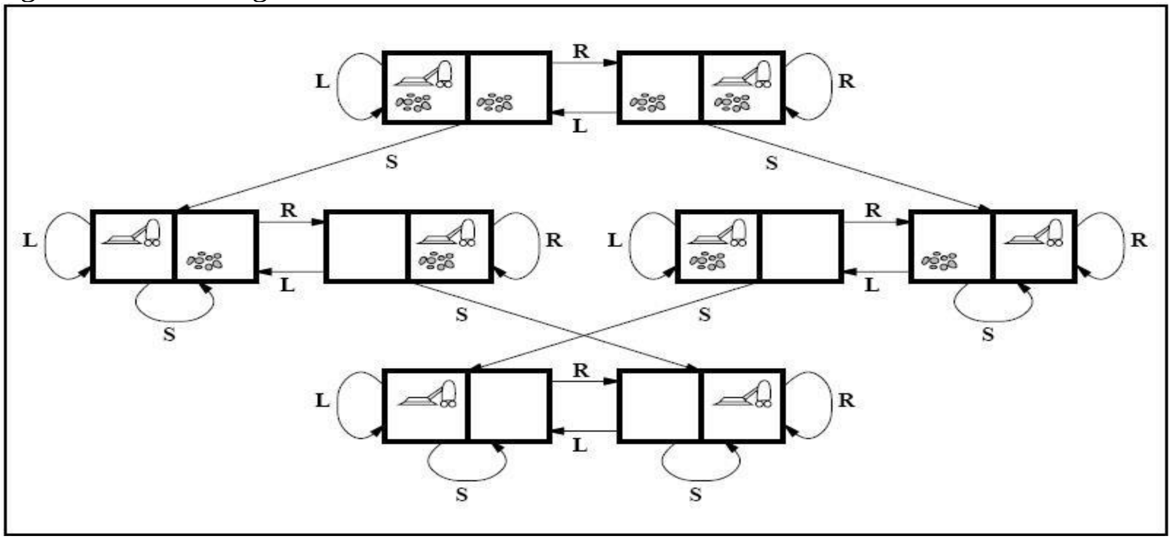
```
state1.agent_position = state1
state1.right = state2
state1.clean = state5
state1.state_name = "State 1"
state2.agent_position = state2
state2.left = state2
state2.right = state4
state2.state_name = "State 2"
state3.left = state3
state3.right = state4
state3.clean = state7
state3.state_name = "State 3"
state4.agent_position = state4
state4.left = state4
state4.right = state4
state4.state_name = "State 4"
state5.left = state5
state5.right = state6
state5.state_name = "State 5"
state6.agent_position = state6
state6.left = state6
state6.right = state8
state6.state_name = "State 6"
state7.left = state7
state7.right = state8
state7.state_name = "State 7"
state8.agent_position = state8
```

```
state8.left = state8
state8.right = state8
state8.state_name = "State 8"
# Run agent
initial_state = None
if position == 1:
    if status_a and status_b:
        initial_state = state7
    elif status_a:
        initial_state = state5
    elif status_b:
        initial_state = state3
    else:
        initial_state = state1
elif position == 2:
    if status_a and status_b:
        initial_state = state8
    elif status_a:
        initial_state = state6
    elif status_b:
        initial_state = state4
    else:
        initial_state = state2
else:
    print("\nInitial state is unknown...")
    return
agent = Agent(initial_state, state7, state8)
```

```python
    if agent.run_bfs():
        print("\nAgent achieved the goal.")
    else:
        print("\nAgent failed to reach the goal.")
if __name__ == "__main__":
    main()
```

Output1:

Enter the initial state of each room where: 1 - Clean | 2 - Dirty
Enter the state of the first room: 2
Enter the state of the second room: 2
Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B
Enter the initial position: 1
Initial State: State 1
Next possible states: ['State 2', 'State 5']
Move to state State 2
Vacuum is in room <__main__.State object at 0x000002154755C210>
Room A is dirty
Room B is dirty
Next possible states: ['State 2', 'State 4']
Next possible states: ['State 6', 'State 8']
Move to state State 8
Vacuum is in room <__main__.State object at 0x0000021547677410>
Room A is dirty
Room B is dirty
Final State: State 8
Agent achieved the goal.



Output2:

Enter the initial state of each room where: 1 - Clean | 2 - Dirty
Enter the state of the first room: 1
Enter the state of the second room: 1
Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B Enter the initial
position: 1
Initial State: State 7
Room A is clean
Room B is clean
Final State: State 7
Agent achieved the goal.

4.b)Aim:Implement Vacuum World problem with Search tree generation using DFS

Source code:

```
class State:
    def __init__(self, agent_position, room_a, room_b):
        self.agent_position = agent_position
        self.room_a = room_a
        self.room_b = room_b
        self.left = None
        self.right = None
        self.clean = None
    def get_states(self):
        next_states = []
        if self.left:
            next_states.append(self.left)
        if self.right:
            next_states.append(self.right)
        if self.clean:
            next_states.append(self.clean)
        return next_states
class Agent:
    def __init__(self, first_state, goal_state1, goal_state2):
        self.first_state = first_state
        self.goal_state1 = goal_state1
        self.goal_state2 = goal_state2
    def run_dfs(self):
```

```
        visited = set()
        return self.dfs(self.first_state, visited)
    def dfs(self, current_state, visited):
        if current_state in visited:
            return False
        visited.add(current_state)
        print("\nMove to state", current_state.state_name)
        self.prompt_attributes(current_state)
        if current_state == self.goal_state1 or current_state == self.goal_state2:
            print("Final State:", current_state.state_name)
            return True
        for state in current_state.get_states():
            if self.dfs(state, visited):
                return True
        return False
    def prompt_attributes(self, current):
        print("Vacuum is in room", str(current.agent_position))
        if current.room_a:
            print("Room A is clean")
        else:
            print("Room A is dirty")
        if current.room_b:
            print("Room B is clean")
        else:
            print("Room B is dirty")
def main():
    # Input Section
```

```
print("Enter the initial state of each room where: 1 - Clean | 2 - Dirty")
status_a = int(input("Enter the state of the first room: ")) == 1
status_b = int(input("Enter the state of the second room: ")) == 1
print("Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B")
position = int(input("Enter the initial position: "))
# Create state instances
state1 = State(None, status_a, status_b)
state2 = State(None, status_a, status_b)
state3 = State(None, status_a, status_b)
state4 = State(None, status_a, status_b)
state5 = State(None, status_a, status_b)
state6 = State(None, status_a, status_b)
state7 = State(None, status_a, status_b)
state8 = State(None, status_a, status_b)
# Setup and connect each state
state1.agent_position = state1
state1.right = state2
state1.clean = state5
state1.state_name = "State 1"
state2.agent_position = state2
state2.left = state2
state2.right = state4
state2.state_name = "State 2"
state3.left = state3
state3.right = state4
state3.clean = state7
state3.state_name = "State 3"
```

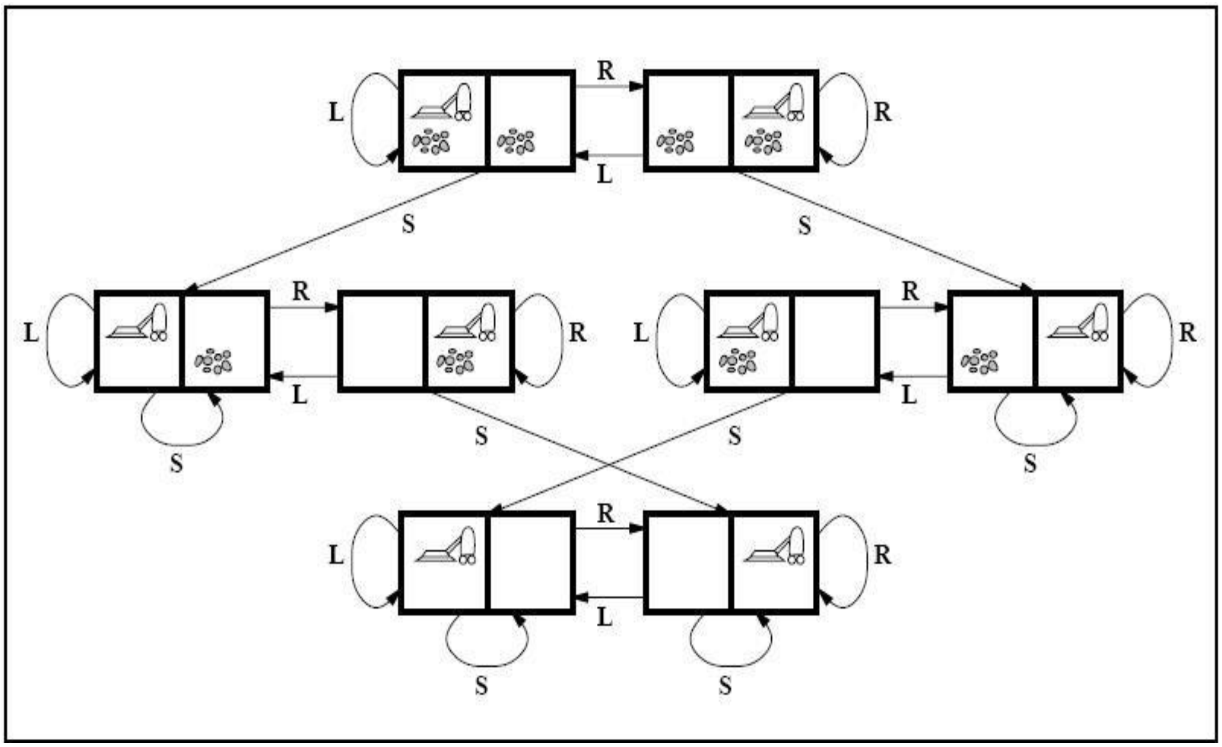
```
state4.agent_position = state4
state4.left = state4
state4.right = state4
state4.state_name = "State 4"
state5.left = state5
state5.right = state6
state5.state_name = "State 5"
state6.agent_position = state6
state6.left = state6
state6.right = state8
state6.state_name = "State 6"
state7.left = state7
state7.right = state8
state7.state_name = "State 7"
state8.agent_position = state8
state8.left = state8
state8.right = state8
state8.state_name = "State 8"
# Run agent
initial_state = None
if position == 1:
    if status_a and status_b:
        initial_state = state7
    elif status_a:
        initial_state = state5
    elif status_b:
        initial_state = state3
```

```
        else:
            initial_state = state1
    elif position == 2:
        if status_a and status_b:
            initial_state = state8
        elif status_a:
            initial_state = state6
        elif status_b:
            initial_state = state4
        else:
            initial_state = state2
    else:
        print("\nInitial state is unknown...")
        return


    agent = Agent(initial_state, state7, state8)
    if agent.run_dfs():
        print("\nAgent achieved the goal.")
    else:
        print("\nAgent failed to reach the goal.")
if __name__ == "__main__":
    main()
```

Output1:

```
Enter the initial state of each room where: 1 - Clean | 2 - Dirty
Enter the state of the first room:  2
Enter the state of the second room:  1
Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B
Enter the initial position:  1
Move to state State 3
Vacuum is in room None
```

Room A is dirty
Room B is clean
Move to state State 4
Vacuum is in room <__main__.State object at 0x0000021547677D90>
Room A is dirty
Room B is clean
Move to state State 7
Vacuum is in room None
Room A is dirty
Room B is clean
Final State: State 7
Agent achieved the goal.



<u>Output2:</u>
Enter the initial state of each room where: 1 - Clean | 2 - Dirty
Enter the state of the first room: 2
Enter the state of the second room: 1
Enter the initial position of the vacuum where: 1 - Room A | 2 - Room B Enter the initial position: 1
Move to state State 3
Room A is dirty
Room B is clean
Move to state State 4
Room A is dirty

Room B is clean
Move to state State 7
Room A is dirty
Room B is clean
Final State: State 7
Agent achieved the goal.

5.a)Aim: Implement the following Greedy Best First Search

Source code:

```
class Node:
    def __init__(self, v, weight):
        self.v = v
        self.weight = weight
class PathNode:
    def __init__(self, node, parent=None):
        self.node = node
        self.parent = parent
def addEdge(u, v, weight):
    adj[u].append(Node(v, weight))
def GBFS(h, V, src, dest):
    openList = []
    closeList = []
    openList.append(PathNode(src, None))
    frontier = []
    explored = []
    while openList:
        currentNode = openList[0]
        currentIndex = 0
        for i in range(len(openList)):
            if h[openList[i].node] < h[currentNode.node]:
                currentNode = openList[i]
                currentIndex = i
        openList.pop(currentIndex)
```

```
        closeList.append(currentNode)
        explored.append(currentNode.node)
        if currentNode.node == dest:
            path = []
            cur = currentNode
            while cur:
                path.append(cur.node)
                cur = cur.parent
            path.reverse()
            return path, frontier, explored
        for node in adj[currentNode.node]:
            if node.v not in [x.node for x in openList] and node.v not in [x.node for x in closeList]:
                openList.append(PathNode(node.v, currentNode))
                frontier.append(node.v)
    return [], frontier, explored
V = int(input("Enter the number of vertices: "))
adj = [[] for _ in range(V)]
# Getting input for edges and weights
print("Enter the edges (u v weight), one per line (press enter to stop):")
while True:
    edge = input().split()
    if len(edge) != 3:
        break
    u, v, weight = map(int, edge)
    addEdge(u, v, weight)
# Getting user input for the source and destination nodes.
src = int(input("Enter the source node: "))
```

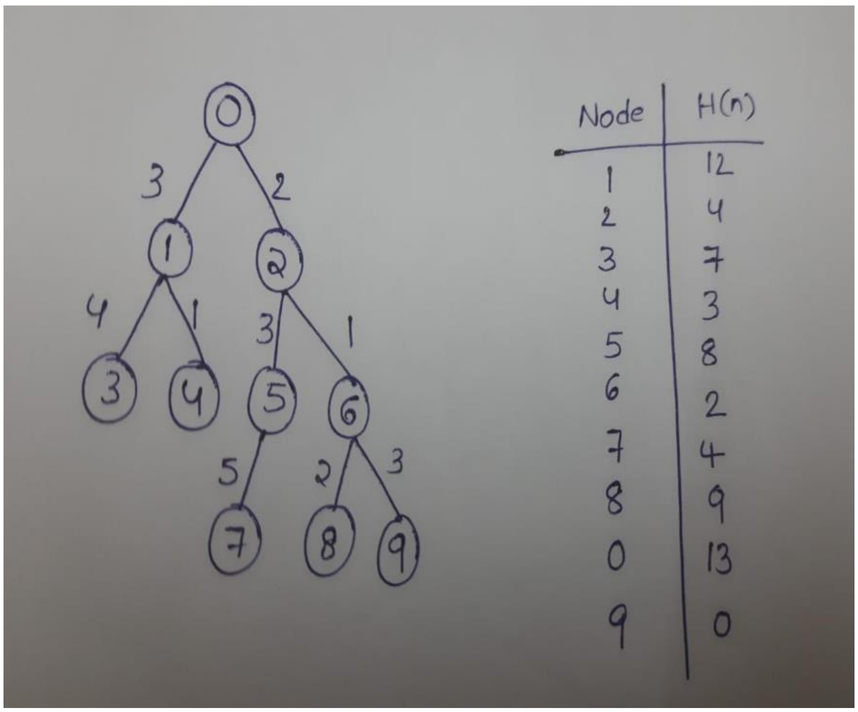
```
dest = int(input("Enter the destination node: "))


# Getting the heuristic values for each node.
h = []
for i in range(V):
    h_value = int(input("Enter the heuristic value for node " + str(i) + ": "))
    h.append(h_value)
path, frontier, explored = GBFS(h, V, src, dest)
if path:
    print("Shortest path:", " -> ".join(str(node) for node in path))
else:
    print("No path found from source to destination.")
# Frontier
print("Frontier:", frontier)
# Explored
print("Explored:", explored)
# Path Cost
print("Path Cost:", len(path))
# State Space Tree
print("State Space Tree:")
for node in path:
    print("Node:", node, "Parent:", path[path.index(node) - 1] if path.index(node) != 0 else None)
```

<u>Output1:</u>

```
Enter the number of vertices: 10
Enter the edges (u v weight), one per line (press enter to stop):
0 1 3
0 2 2
1 3 4
1 4 1
2 5 3
2 6 1
```

```
5 7 5
6 8 2
6 9 3

Enter the source node: 0
Enter the destination node: 7
Enter the heuristic value for node 0: 13
Enter the heuristic value for node 1: 12
Enter the heuristic value for node 2: 4
Enter the heuristic value for node 3: 7
Enter the heuristic value for node 4: 3
Enter the heuristic value for node 5: 8
Enter the heuristic value for node 6: 2
Enter the heuristic value for node 7: 4
Enter the heuristic value for node 8: 9
Enter the heuristic value for node 9: 0
Shortest path: 0 -> 2 -> 5 -> 7
Frontier: [1, 2, 5, 6, 8, 9, 7]
Explored: [0, 2, 6, 9, 5, 7]
Path Cost: 4
State Space Tree:
Node: 0 Parent: None
Node: 2 Parent: 0
Node: 5 Parent: 2
Node: 7 Parent: 5
```

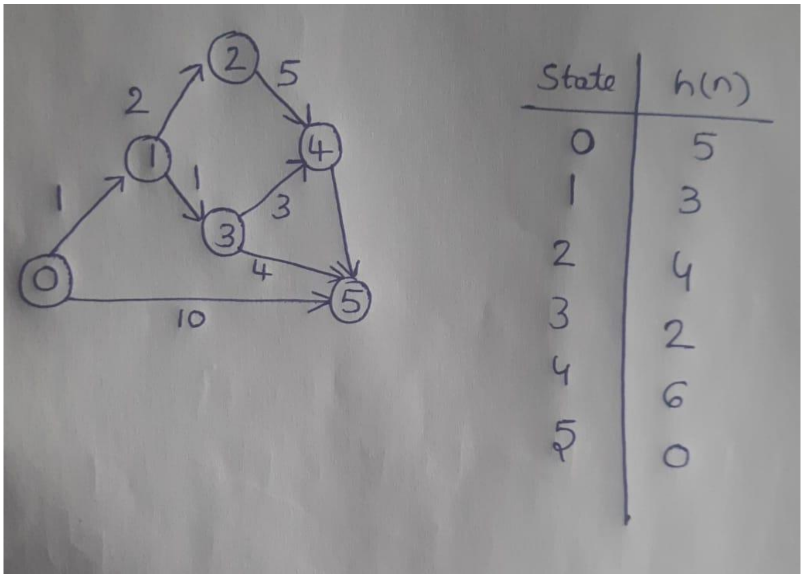
Graph:



<u>Output2:</u>

```
Enter the number of vertices: 6
Enter the edges (u v weight), one per line (press enter to stop):
0 1 1
1 2 2
2 4 5
1 3 1
3 4 3
3 5 4
4 5 1
0 5 10

Enter the source node: 0
Enter the destination node: 5
Enter the heuristic value for node 0: 5
Enter the heuristic value for node 1: 3
Enter the heuristic value for node 2: 4
Enter the heuristic value for node 3: 2
Enter the heuristic value for node 4: 6
Enter the heuristic value for node 5: 0
Shortest path: 0 -> 5
Frontier: [1, 5]
Explored: [0, 5]
Path Cost: 2
State Space Tree:
Node: 0 Parent: None
Node: 5 Parent: 0
```

Graph:



Date: (18-03-2024)

5.b) Aim : Implement the A* algorithm.

Source code:

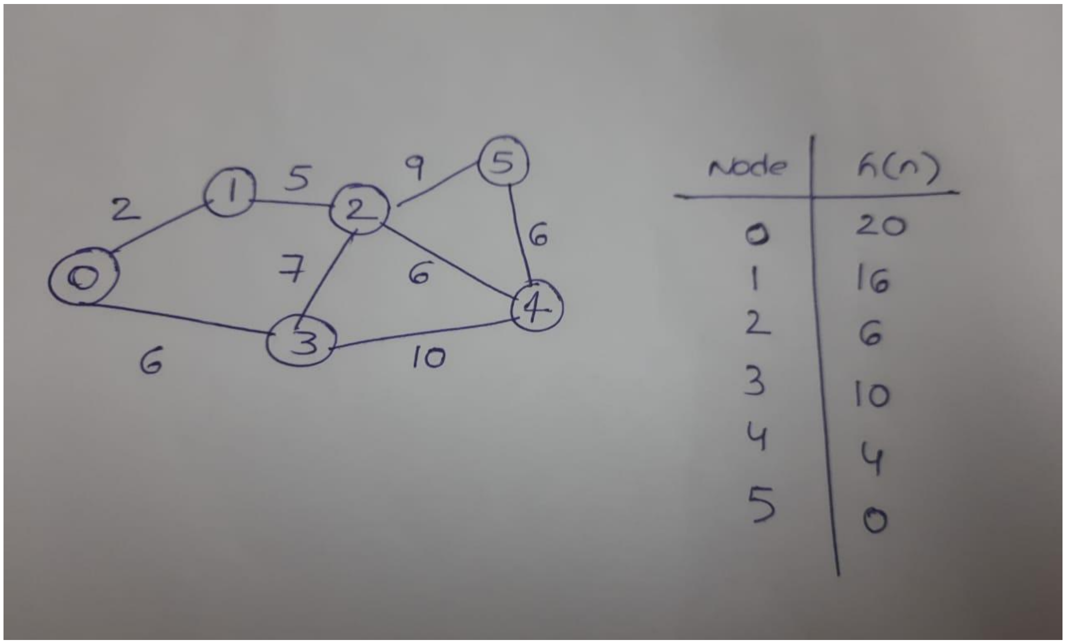
```
from collections import deque


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


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


    def h(self, n, H):
        return H[n]


    def a_star_algorithm(self, start_node, stop_node, H):
        open_list = set([start_node])
        closed_list = set([])
        g = {}
        g[start_node] = 0
        parents = {}
        parents[start_node] = start_node


        while len(open_list) > 0:
            n = None
            for v in open_list:
```

```
        if n is None or g[v] + self.h(v, H) < g[n] + self.h(n, H):
            n = v
    if n is None:
        print('Path does not exist!')
        return None
    if n == stop_node:
        reconst_path = []
        while parents[n] != n:
            reconst_path.append(n)
            n = parents[n]
        reconst_path.append(start_node)
        reconst_path.reverse()
        print('Path found: {}'.format(reconst_path))
        return reconst_path

    for (m, weight) in self.get_neighbors(n):
        if m not in open_list and m not in closed_list:
            open_list.add(m)
            parents[m] = n
            g[m] = g[n] + weight
        else:
            if g[m] > g[n] + weight:
                g[m] = g[n] + weight
                parents[m] = n
                if m in closed_list:
                    closed_list.remove(m)
                    open_list.add(m)
```

```
            open_list.remove(n)
            closed_list.add(n)
            print("open list:", open_list)
            print("closed_list:", closed_list)
        print('Path does not exist!')
        return None


# Taking inputs for adjacency list
adjacency_list = {}
while True:
    node = input("Enter node (or type 'done' to finish): ")
    if node.lower() == 'done':
        break
    edges = input("Enter edges for node {} (format: neighbor,weight, separate by space, type 'done' to finish): ".format(node)).split()
    adjacency_list[node] = [(edge.split(',')[0], int(edge.split(',')[1])) for edge in edges]


# Taking inputs for heuristic values
H = {}
for node in adjacency_list.keys():
    H[node] = int(input("Enter heuristic value for node {}: ".format(node)))


graph = Graph(adjacency_list)
start_node = input("Enter the start node: ")
goal_node = input("Enter the goal node: ")
graph.a_star_algorithm(start_node, goal_node, H)
```

Output1:

```
Enter node (or type 'done' to finish): 0
Enter edges for node 0 (format: neighbor,weight, separate by space, type
'done' to finish): 1,2 3,6
Enter node (or type 'done' to finish): 1
Enter edges for node 1 (format: neighbor,weight, separate by space, type
'done' to finish): 0,2 2,5
Enter node (or type 'done' to finish): 2
Enter edges for node 2 (format: neighbor,weight, separate by space, type
'done' to finish): 1,5 3,7 5,9 4,6
Enter node (or type 'done' to finish): 3
Enter edges for node 3 (format: neighbor,weight, separate by space, type
'done' to finish): 0,6 2,7 4,10
Enter node (or type 'done' to finish): 4
Enter edges for node 4 (format: neighbor,weight, separate by space, type
'done' to finish): 2,6 3,10 5,6
Enter node (or type 'done' to finish): 5
Enter edges for node 5 (format: neighbor,weight, separate by space, type
'done' to finish): 2,9 4,6
Enter node (or type 'done' to finish): done
Enter heuristic value for node 0: 20
Enter heuristic value for node 1: 16
Enter heuristic value for node 2: 6
Enter heuristic value for node 3: 10
Enter heuristic value for node 4: 4
Enter heuristic value for node 5: 0
Enter the start node: 0
Enter the goal node: 5
open list: {'1', '3'}
closed_list: {'0'}
open list: {'1', '4', '2'}
closed_list: {'0', '3'}
open list: {'4', '2'}
closed_list: {'0', '1', '3'}
open list: {'4', '5'}
closed_list: {'0', '1', '3', '2'}
Path found: ['0', '1', '2', '5']
```

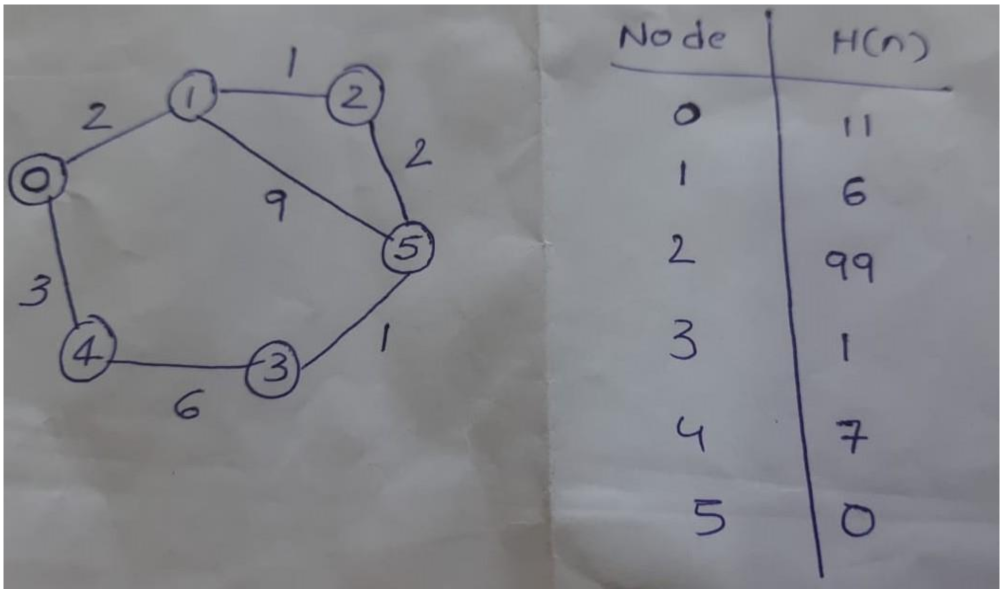


<u>Output2:</u>

```
Enter node (or type 'done' to finish): 0
Enter edges for node 0 (format: neighbor,weight, separate by space, type
'done' to finish): 1,2 4,3
Enter node (or type 'done' to finish): 1
Enter edges for node 1 (format: neighbor,weight, separate by space, type
'done' to finish): 0,2 2,1 5,9
Enter node (or type 'done' to finish): 2
Enter edges for node 2 (format: neighbor,weight, separate by space, type
'done' to finish): 1,2 5,2
Enter node (or type 'done' to finish): 3
Enter edges for node 3 (format: neighbor,weight, separate by space, type
'done' to finish): 4,6 5,1
Enter node (or type 'done' to finish): 4
Enter edges for node 4 (format: neighbor,weight, separate by space, type
'done' to finish): 0,3 3,6
Enter node (or type 'done' to finish): 5
Enter edges for node 5 (format: neighbor,weight, separate by space, type
'done' to finish): 1,9 2,2 3,1
Enter node (or type 'done' to finish): done
Enter heuristic value for node 0: 11
Enter heuristic value for node 1: 6
Enter heuristic value for node 2: 99
Enter heuristic value for node 3: 1
Enter heuristic value for node 4: 7
Enter heuristic value for node 5: 0
Enter the start node: 0
Enter the goal node: 5
open list: {'4', '1'}
closed_list: {'0'}
open list: {'4', '5', '2'}
closed_list: {'0', '1'}
open list: {'3', '5', '2'}
```

```
closed_list: {'4', '0', '1'}
open list: {'5', '2'}
closed_list: {'4', '0', '1', '3'}
Path found: ['0', '4', '3', '5']
```

Graph:



6.Aim : Implement 8-puzzle problem using A* algorithm.

Source code:

```
from heapq import heappop, heappush
class PuzzleNode:
    def __init__(self, state, parent=None, g=0, h=0):
        self.state = state
        self.parent = parent
        self.g = g
        self.h = h
    def __lt__(self, other):
        return (self.g + self.h) < (other.g + other.h)
def get_blank_position(state):
    for i in range(3):
        for j in range(3):
            if state[i][j] == 0:
                return i, j
def get_manhattan_distance(row1, col1, row2, col2):
    return abs(row1 - row2) + abs(col1 - col2)
def get_heuristic_value(state, goal_state):
    h = 0
    for i in range(3):
        for j in range(3):
            if state[i][j] != 0:
                value = state[i][j]
                goal_row, goal_col = find_position(goal_state, value)
                h += get_manhattan_distance(i, j, goal_row, goal_col)
```

```
    return h
def get_valid_moves(row, col):
    moves = []
    if row > 0:
        moves.append((-1, 0)) # Move blank tile up
    if row < 2:
        moves.append((1, 0)) # Move blank tile down
    if col > 0:
        moves.append((0, -1)) # Move blank tile left
    if col < 2:
        moves.append((0, 1)) # Move blank tile right
    return moves
def get_new_state(state, move):
    row, col = get_blank_position(state)
    new_state = [row[:] for row in state]
    new_row, new_col = row + move[0], col + move[1]
    new_state[row][col] = new_state[new_row][new_col]
    new_state[new_row][new_col] = 0
    return new_state
def print_state(state):
    for row in state:
        print(row)
    print()
def is_goal_state(state, goal_state):
    return state == goal_state
def find_position(state, value):
    for i in range(3):
```

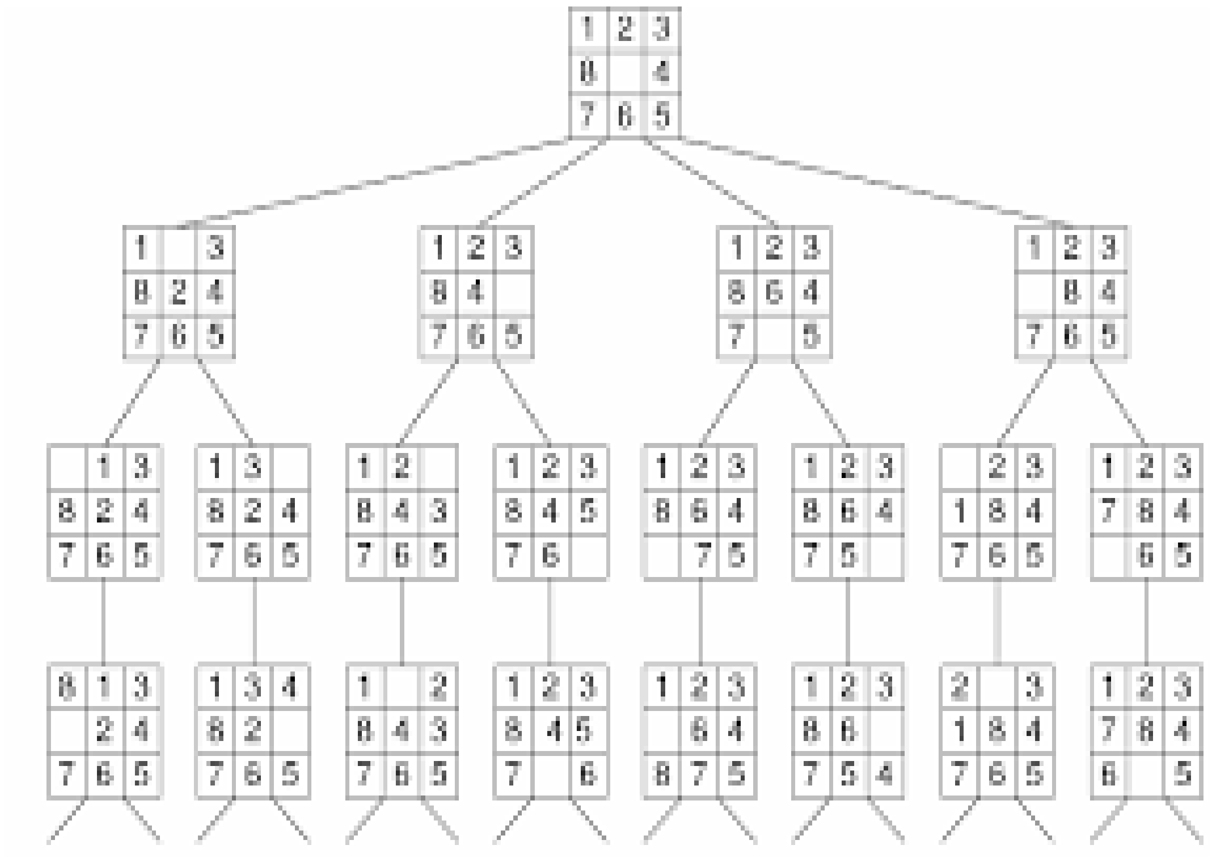
```
        for j in range(3):
            if state[i][j] == value:
                return i, j
def get_solution_path(node):
    path = []
    while node is not None:
        path.append(node.state)
        node = node.parent
    path.reverse()
    return path
def solve_puzzle(initial_state, goal_state):
    open_set = []
    closed_set = set()
    h = get_heuristic_value(initial_state, goal_state)
    initial_node = PuzzleNode(initial_state, g=0, h=h)
    heappush(open_set, initial_node)
    while open_set:
        current_node = heappop(open_set)
        closed_set.add(tuple(map(tuple, current_node.state)))
        if is_goal_state(current_node.state, goal_state):
            return get_solution_path(current_node)
        row, col = get_blank_position(current_node.state)
        moves = get_valid_moves(row, col)
        for move in moves:
            new_state = get_new_state(current_node.state, move)
            if tuple(map(tuple, new_state)) not in closed_set:
                g = current_node.g + 1
```

```
            h = get_heuristic_value(new_state, goal_state)
            new_node = PuzzleNode(new_state, parent=current_node, g=g, h=h)
            heappush(open_set, new_node)
    return None
print("Enter the initial state (0 represents the blank tile):")
initial_state = []
for i in range(3):
    row = list(map(int, input().split()))
    initial_state.append(row)
print("Enter the goal state:")
goal_state = []
for i in range(3):
    row = list(map(int, input().split()))
    goal_state.append(row)
solution = solve_puzzle(initial_state, goal_state)
if solution is not None:
    print("Solution found!")
    for state in solution:
        print_state(state)
else:
    print("No solution found.")
```

Output1:

```
Enter the initial state (0 represents the blank tile): 1 2 3
0 4 6
7 5 8
Enter the goal state: 1 2 3
4 5 6
```

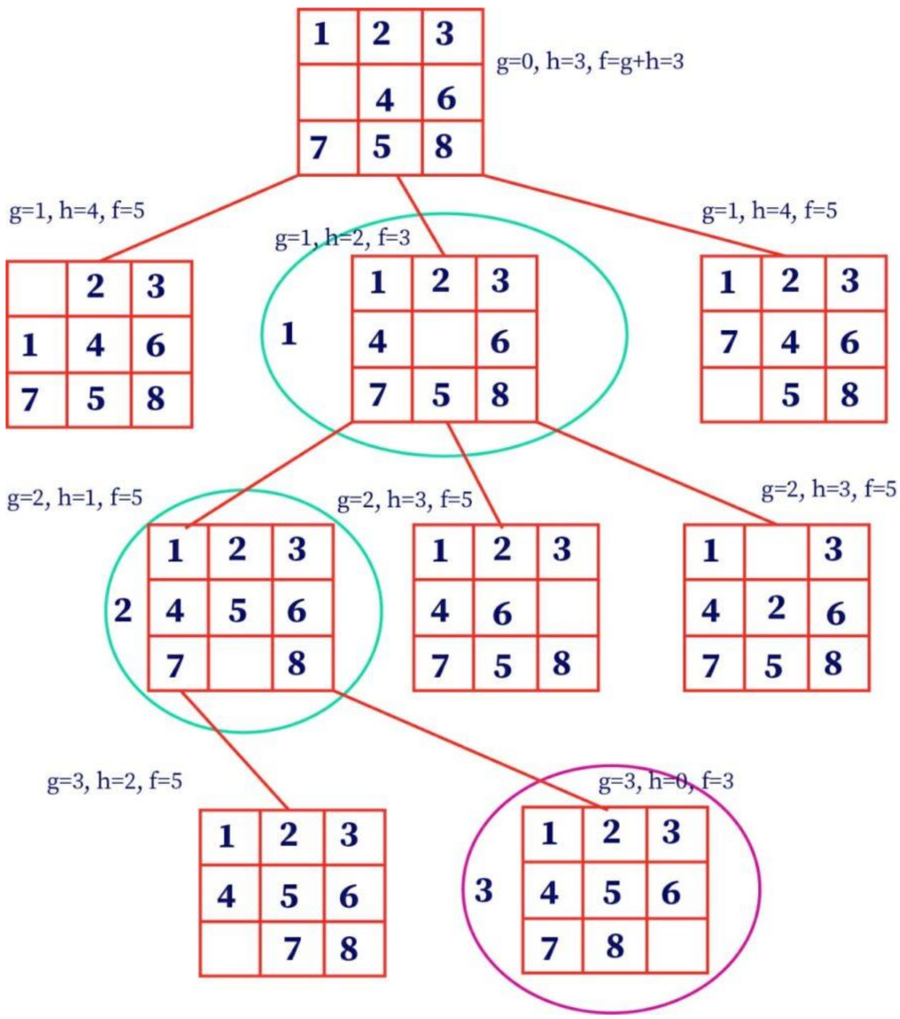
```
 7 8 0

Solution found!

 [[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] ]

[[1, 2, 3] [4, 5, 6] [7, 8, 0]]
```



Output2:

```
Enter the initial state (0 represents the blank tile):
1 2 3
0 4 6
7 5 8
Enter the goal state:
1 2 3
4 5 6
7 8 0
Solution found!
[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]

[1, 2, 3]
[4, 5, 6]
[7, 8, 0]
```



7. Aim: Implement AO* algorithm for General graph problem

Source code:

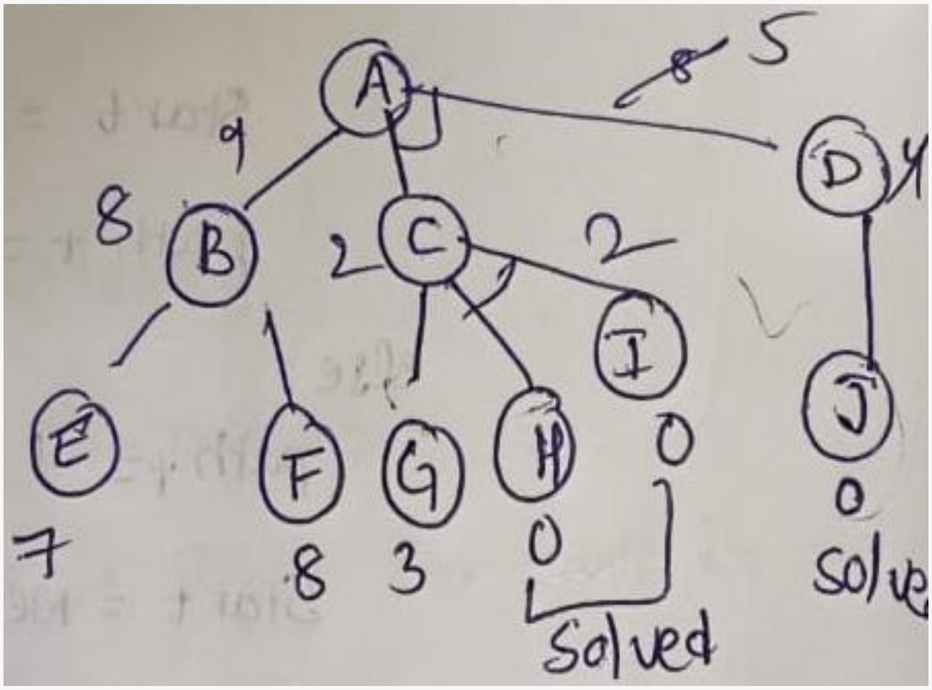
```
def Cost(H, condition, weight=1):
cost = {}
if 'AND' in condition:
AND_nodes = condition['AND']
Path_A = ' AND '.join(AND_nodes)
PathA = sum(H[node] + weight for node in AND_nodes)
cost[Path_A] = PathA
if 'OR' in condition:
OR_nodes = condition['OR']
Path_B = ' OR '.join(OR_nodes)
PathB = min(H[node] + 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]
c = Cost(H, condition, weight)
H[key] = min(c.values())
least_cost[key] = c
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))
```

Output1:

```
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]
Path Cost:8
```

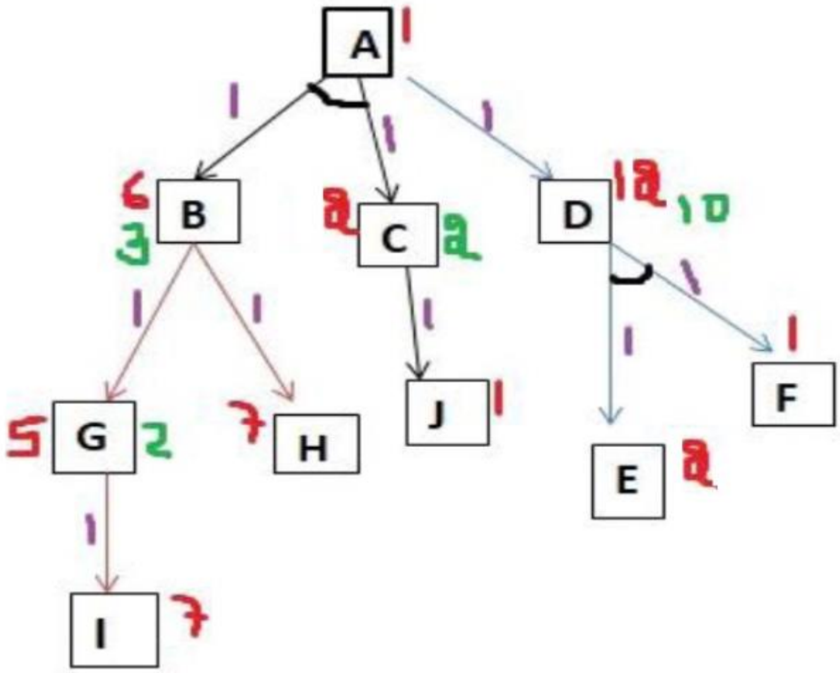
Graph Structure:



Output2:

```
Enter nodes with heuristic costs: {'A': 1, 'B': 6, 'C': 2, 'D': 12, 'E': 2,
'F': 1, 'G': 5, 'H': 7, 'I': 7, 'J': 1}
Enter graph:
{'A':{'OR':['D'],'AND':['B','C']},'B':{'OR':['G','H']},'C':{'OR':['J']},'D':{
'AND':['E','F']},'G':{'OR':['I']}}
Updated Cost :
Shortest Path :
 A <-- D <-- (E AND F) [E + F]
```

Graph:



8. a)Implement Game trees using MINIMAX algorithm

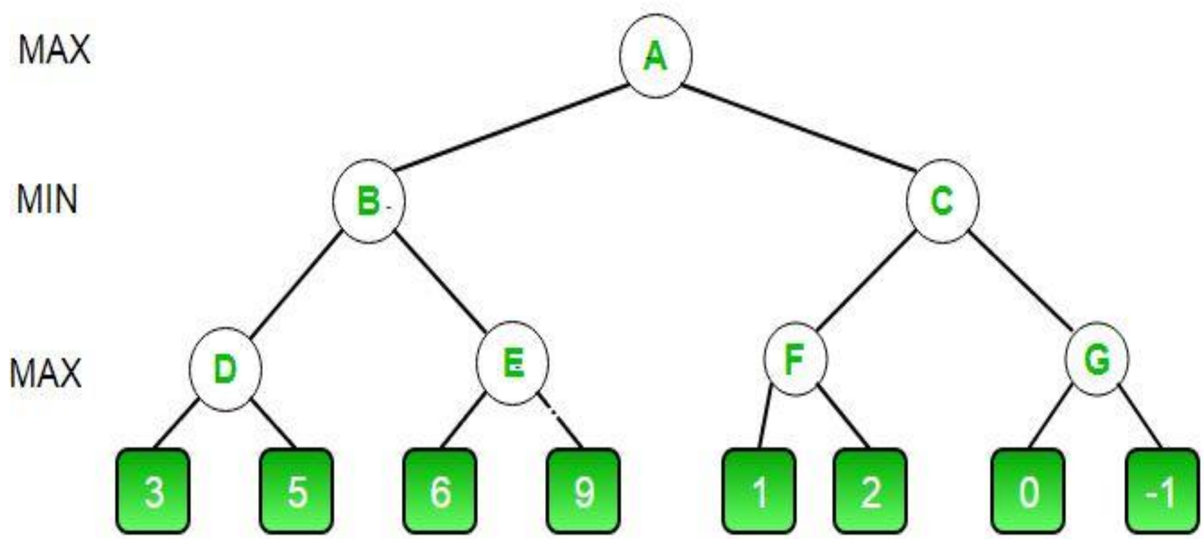
Source code:

```
import math

def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth, path_cost):

    # Base case: targetDepth reached

    if curDepth == targetDepth:

        return scores[nodeIndex], path_cost[nodeIndex]

    if maxTurn:

        left_score, left_cost = minimax(curDepth + 1, nodeIndex * 2, False, scores, targetDepth,
path_cost)

        right_score, right_cost = minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores,
targetDepth, path_cost)

        return max(left_score, right_score), left_cost + right_cost

    else:

        left_score, left_cost = minimax(curDepth + 1, nodeIndex * 2, True, scores, targetDepth,
path_cost)

        right_score, right_cost = minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores,
targetDepth, path_cost)

        return min(left_score, right_score), left_cost + right_cost

num_nodes = int(input("Enter the number of nodes: "))

scores = []

path_cost = []

print("Enter the scores for each node:")

for I in range(num_nodes):

    score = int(input(f"Node {i}: "))

    scores.append(score)

    path_cost.append(i) # Initialize path cost with node index for simplicity

treeDepth = math.log(len(scores), 2)
```

```
optimal_value, total_cost = minimax(0, 0, True, scores, int(treeDepth), path_cost)

print("The optimal value is:", optimal_value)

print("Total Path Cost:", total_cost)
```

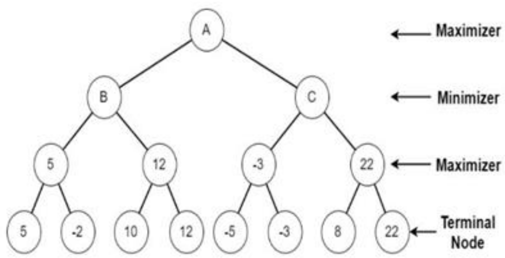
Output1:

```
Enter the number of nodes: 8
Enter the scores for each node:
Node 0: 3
Node 1: 5
Node 2: 6
Node 3: 9
Node 4: 1
Node 5: 2
Node 6: 0
Node 7: -1
The optimal value is: 5
Total Path Cost: 28
```



Output2:

```
Enter the number of nodes:8
Enter the scores for each node:
Node 0: 5
Node 1: -2
Node 2: 10
Node 3: 12
Node 4: -5
Node 5: -3
Node 6: 8
Node 7: 22
The optimal value is: 5
Total Path Cost: 28
```



8.b)Aim: Implement Game trees using algorithm. Alpha-Beta pruning

Source code:

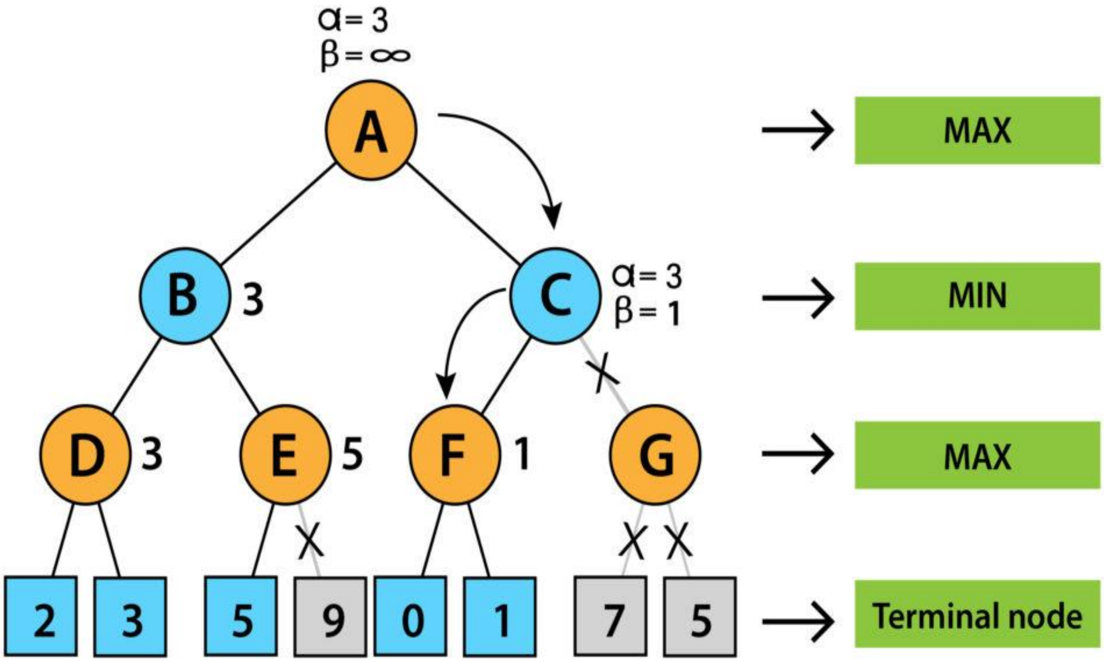
```
MAX, MIN = 1000, -1000

def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta, path_cost):

    if depth == 0:

        return values[nodeIndex], path_cost[nodeIndex]

    if maximizingPlayer:

        best = MIN

        for i in range(0, 2):

            val, path_cost_i = minimax(depth - 1, nodeIndex * 2 + i, False, values, alpha, beta,
path_cost)

            best = max(best, val)

            alpha = max(alpha, best)

            # Alpha Beta Pruning

            if beta <= alpha:

                break

        return best, path_cost[nodeIndex] + path_cost_i

    else:

        best = MAX

        for i in range(0, 2):

            val, path_cost_i = minimax(depth - 1, nodeIndex * 2 + i, True, values, alpha, beta,
path_cost)

            best = min(best, val)

            beta = min(beta, best)

            # Alpha Beta Pruning

            if beta <= alpha:

                break
```

```
        return best, path_cost[nodeIndex] + path_cost_i
if __name__ == "__main__":
    # Take user input for values
    values = []
    path_cost = []
    num_nodes = int(input("Enter the number of nodes: "))
    print("Enter the values for each node:")
    for i in range(num_nodes):
        value = int(input(f"Node {i}: "))
        values.append(value)
        path_cost.append(i)
 print(path_cost)
   depth = int(input("Enter the depth: "))
    print("The optimal value is:", minimax(depth, 0, True, values, MIN, MAX, path_cost))
```

Output1:

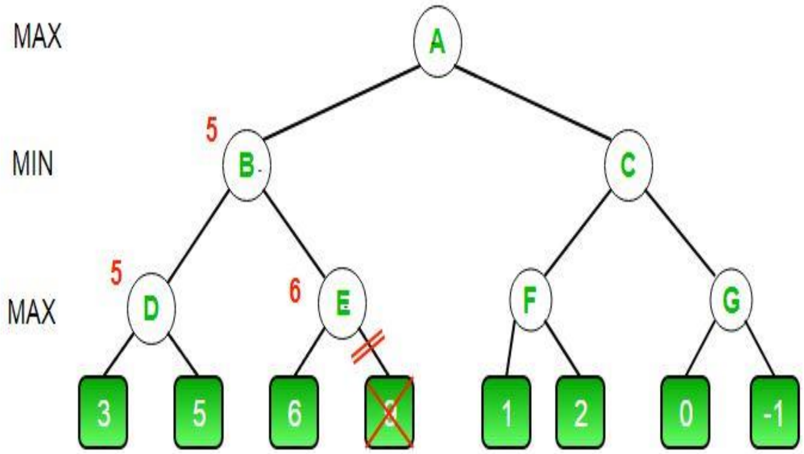
```
Enter the number of nodes: 8
 Enter the values for each node:
Node 0: 2
Node 1: 3
Node 2: 5
Node 3: 9
Node 4: 0
Node 5: 1
Node 6: 7
Node 7: 5
[0, 1, 2, 3, 4, 5, 6, 7]
Enter the depth: 3
```

The optimal value is: (3, 8)



<u>Output2:</u>

```
Enter the number of nodes: 8
Enter the values for each node:
Node 0: 3
Node 1: 5
Node 2: 6
Node 3: 9
Node 4: 1
Node 5: 2
Node 6: 0
Node 7: -1
[0, 1, 2, 3, 4, 5, 6, 7]
Enter the depth: 3
The optimal value is: (5, 8)
```



MAX
MIN
MAX
A
B
C
D
E
F
G
5
5
6
3
5
6
1
2
0
-1

9.Aim: Implement Crypt arithmetic problems.

Source code:

```
import itertools
def number(n, d):
    t = 0
    for i in n:
        t = d[i] + (t * 10)
    return t
def test(l, s, d):
    sum = 0
    for i in l:
        sum = sum + number(i, d)
    if sum == number(s, d):
        return 1
    return 0
def check(d):
    for i in d.keys():
        if i in c and d[i] == 0:
            return 1
    return 0
l = input('Enter list of strings: ').split()
s = input('Enter output string: ')
c = []
for i in l:
    c.append(i[0])
    c.append(s[0])
```

```
p = list(set(''.join(l) + s))
q = len(p)
k = list(itertools.permutations(range(0, 10), q))
d = {}
f = 0
for i in k:
    for j in range(q):
        d[p[j]] = i[j]
    if check(d):
        continue
    if test(l, s, d) == 1:
        f = 1
        print(d)
        print('Solution found')
        break
if f == 0:
    print('No solution found')
```

Output1:

```
Enter list of strings: send more
Enter output string: moneys
{'s': 9, 'r': 8, 'y': 2, 'n': 6, 'o': 0, 'e': 5, 'd': 7,
'm': 1}
```

```
  SEND
+ MORE
----------------
 MONEY
```

```
Solution found
```

Output2:

```
Enter list of strings: cross roads
Enter output string: danger
```

```
{'e': 4, 'c': 9, 'r': 6, 'd': 1, 'g': 7, 'n': 8, 's': 3, 'o': 2, 'a': 5}
Solution found
```

CROSS	9 6 2 3 3
ROADS	6 2 5 1 3
------------	----------------
DANGER	1 5 8 7 4 6