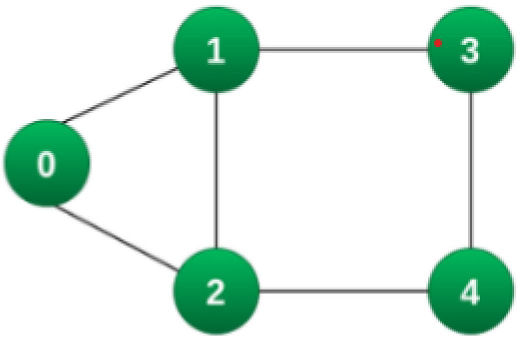
Output 1 :

```
Enter source node: 0
Enter goal node: 4
Frontier: [0]
Explored: [0]
0
Frontier: [1, 2]
Explored: [0, 1, 2]
1
Frontier: [2, 3]
Explored: [0, 1, 2, 3]
2
Frontier: [3, 4]
Explored: [0, 1, 2, 3, 4]
3
Frontier: [4]
Explored: [0, 1, 2, 3, 4]
4

Goal reached: 4
```

Output 2 :

```
Enter source node: 4
Enter goal node: 1
Frontier: [4]
Explored: [4]
4
Frontier: [2, 3]
Explored: [4, 2, 3]
2
Frontier: [3, 0, 1]
Explored: [4, 2, 3, 0, 1]
3
Frontier: [0, 1]
Explored: [4, 2, 3, 0, 1]
0
Frontier: [1]
Explored: [4, 2, 3, 0, 1]
1

Goal reached: 1
```



1 AIM : Implement Exhaustive search techniques using

a. BFS(Breadth First Search)

Source Code :

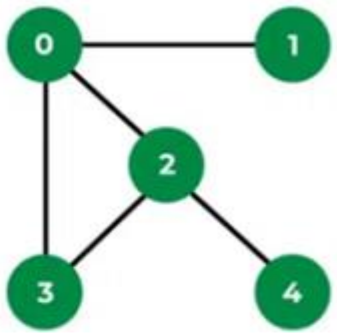
```
def BFS(graph, v, goal):
    explored = [v]
    frontier = [v]
    path = []
    while frontier:
        print(f"Frontier: {frontier}")
        print(f"Explored: {explored}")
        v = frontier.pop(0)
        print(v)
        if v == goal:
            print("\nGoal reached:", v)
            return path
        path.append(v)
        for i in graph[v]:
            if i not in explored:
                frontier.append(i)
                explored.append(i)
    print("\nGoal not reached")
    return None
graph = {
    0: [1, 2],
    1: [0, 2, 3],
    2: [0, 1, 4],
    3: [1, 4],
    4: [2, 3]
}
s = int(input("Enter source node: "))
g = int(input("Enter goal node: "))
path = BFS(graph, s, g)
```

Output 1 :

```
Enter source node: 0
Enter goal node: 4
Frontier: [0]
Explored: [0]
0
Frontier: [1, 2]
Explored: [0, 1, 2]
2
Frontier: [1, 3, 4]
Explored: [0, 1, 2, 3, 4]
4
Goal reached:  4
```

Output 2 :

```
Enter source node: 4
Enter goal node: 1
Frontier: [4]
Explored: [4]
4
Frontier: [2]
Explored: [4, 2]
2
Frontier: [0, 3]
Explored: [4, 2, 0, 3]
3
Frontier: [0]
Explored: [4, 2, 0, 3]
0
Frontier: [1]
Explored: [4, 2, 0, 3, 1]
1
Goal reached:  1
```



b. DFS((Depth First Search)

Source Code :

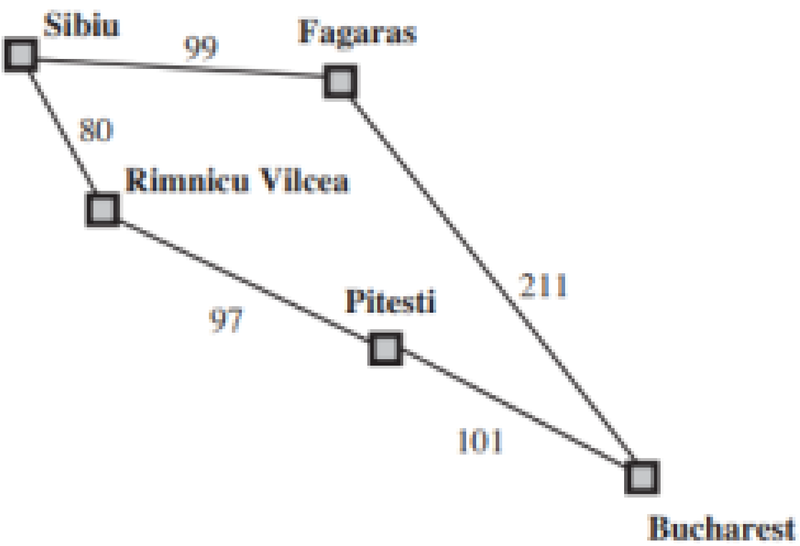
```
def DFS(graph, v, goal):
    explored = [v]
    frontier = [v]
    while frontier:
        print(f"Frontier: {frontier} ")
        print(f"Explored: {explored} ")
        v = frontier.pop(-1)
        print(v)
        if v == goal:
            print("Goal reached: ", v)
            return
        for i in graph[v]:
            if i not in explored:
                frontier.append(i)
                explored.append(i)
    print("Goal not found")
graph = {
    0: [1, 2],
    1: [0],
    2: [0, 3, 4],
    3: [0, 2],
    4: [2]
}
s = int(input("Enter source node: "))
g = int(input("Enter goal node: "))
DFS(graph, s, g)
```

Output 1 :

```
Enter source node: S
Enter goal node: B
Frontier: {'S': 0}
Explored: []
S  :  0
Frontier: {'F': 99, 'R': 80}
Explored: ['S']
R  :  80
Frontier: {'F': 99, 'P': 177}
Explored: ['S', 'R']
F  :  99
Frontier: {'P': 177, 'B': 310}
Explored: ['S', 'R', 'F']
P  :  177
Frontier: {'B': 278}
Explored: ['S', 'R', 'F', 'P']
B  :  278
Goal reached with cost: 278
```

Output 2 :

```
Enter source node: R
Enter goal node: B
Frontier: {'R': 0}
Explored: []
R  :  0
Frontier: {'P': 97}
Explored: ['R']
P  :  97
Frontier: {'B': 198}
Explored: ['R', 'P']
B  :  198
Goal reached with cost: 198
```



c. Uniform Cost Search

Source Code :

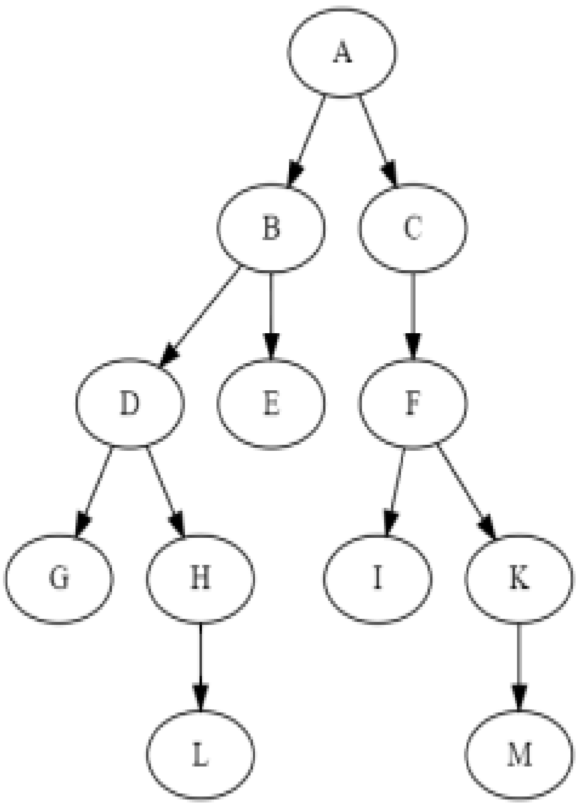
```
def UCS(graph, s, goal):
    frontier = {s: 0}
    explored = []
    while frontier:
        print(f"Frontier: {frontier} ")
        print(f"Explored: {explored} ")
        node = min(frontier, key=frontier.get)
        val = frontier[node]
        print(node, " : ", val)
        del frontier[node]
        if goal == node:
            return f"Goal reached with cost: {val}"
        explored.append(node)
        for neighbour, pathCost in graph[node].items():
            if neighbour not in explored or neighbour not in frontier:
                frontier.update({neighbour: val + pathCost})
            elif neighbour in frontier and pathCost > val:
                frontier.update({neighbour: val})
    return "Goal not found"
graph = {
    'S': {'F': 99, 'R': 80},
    'F': {'B': 211},
    'R': {'P': 97},
    'P': {'B': 101},
    'B': {}
}
s = input("Enter source node: ")
g = input("Enter goal node: ")
print(UCS(graph, s, g))
```

Output 1 :

```
Enter source node: A
Enter goal node: M

A
A B C
A B D E C F
A B D G H E C F I K
Goal found at depth: 4
```

Output 2 :

```
Enter source node: B
Enter goal node: H

B
B D
Goal found at depth: 2
```



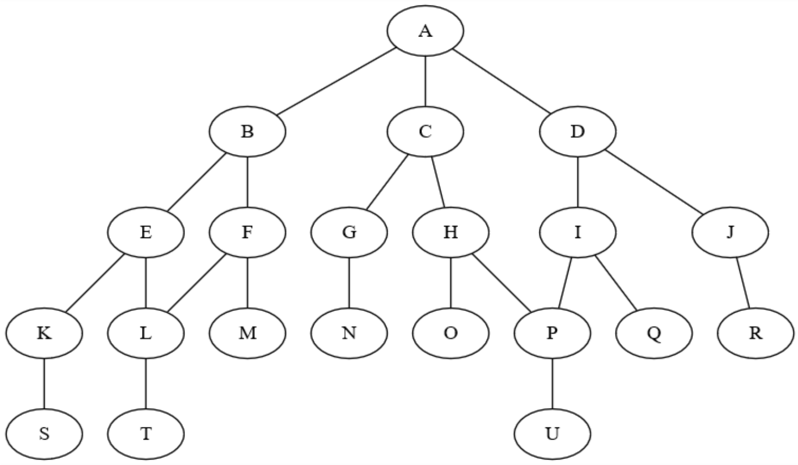
d. Depth-First Iterative Deepening Search

Source Code :

```
def recursiveDLS(graph, v, goal, limit):
    if v == goal:
        return 'GOAL'
    elif limit == 0:
        return 'LIMIT'
    else:
        cutoff = False
        print(v, end=' ')
        for neighbour in graph[v]:
            result = recursiveDLS(graph, neighbour, goal, limit-1)
            if result == 'LIMIT':
                cutoff = True
            elif result != 'FAIL':
                return result
        return 'LIMIT' if cutoff else 'FAIL'

def IDS(graph, v, goal):
    for depth in range(100):
        result = recursiveDLS(graph, v, goal, depth)
        print()
        if result != 'LIMIT':
            return result, depth

graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': ['G', 'H'],
    'E': [],
    'F': ['I', 'K'],
    'G': [],
    'H': ['L'],
    'I': [],
    'K': ['M'],
    'L': [],
    'M': []
}

s = input("Enter source node: ")
g = input("Enter goal node: ")
res, depth = IDS(graph, s, g)
if res == 'GOAL':
    print("Goal found at depth:", depth)
else:
    print("Goal not found")
```

Output 1 :

```
Enter number of nodes in graph: 21
A B C D E F G H I J K L M N O P Q R S T U are the nodes
Enter the child nodes of A: B C D
Enter the child nodes of B: E F
Enter the child nodes of C: G H
Enter the child nodes of D: I J
Enter the child nodes of E: K L
Enter the child nodes of F: L M
Enter the child nodes of G: N
Enter the child nodes of H: O P
Enter the child nodes of I: P Q
Enter the child nodes of J: R
Enter the child nodes of K: S
Enter the child nodes of L: T
Enter the child nodes of M: 0
Enter the child nodes of N: 0
Enter the child nodes of O: 0
Enter the child nodes of P: U
Enter the child nodes of Q: 0
Enter the child nodes of R: 0
Enter the child nodes of S: 0
Enter the child nodes of T: 0
Enter the child nodes of U: 0
Enter source node: A
Enter destination node: U
From front:
       Frontier: ['A']
       Reached: ['A']
From back:
       Frontier: ['U']
       Reached: ['U']
Path found!
Path: ['A', 'P', 'U'] ['A']
```



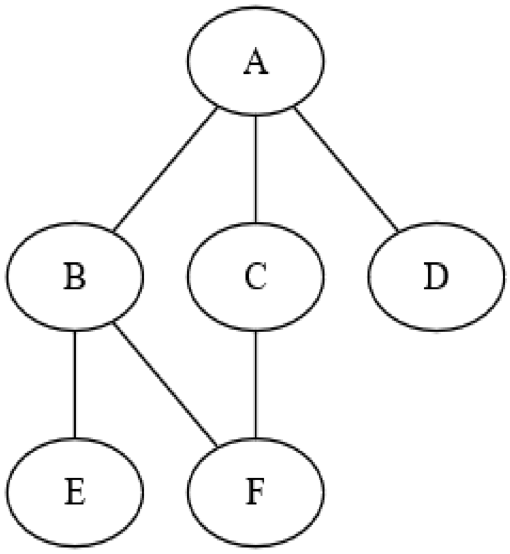
e. Bidirectional Search

Source Code :

```
def BFS(direction, graph, frontier, reached):
    if direction == 'F':  # FROM ONE SIDE(SAY FRONT F)
        d = 'c'
    elif direction == 'B': :  # FROM ONE SIDE(SAY BACK B)
        d = 'p'
    node = frontier.pop(0)
    for child in graph[node][d]:
        if child not in reached:
            reached.append(child)
            frontier.append(child)
    return frontier, reached

def isIntersecting(reachedF, reachedB):
    intersecting = set(reachedF).intersection(set(reachedB))
    return list(intersecting)[0] if intersecting else -1

def BidirectionalSearch(graph, source, dest):
    frontierF = [source]
    frontierB = [dest]
    reachedF = [source]
    reachedB = [dest]
    while frontierF and frontierB:
        print("From front: ")
        print(f"\tFrontier: {frontierF}")
        print(f"\tReached: {reachedF}")
        print("From back: ")
        print(f"\tFrontier: {frontierB}")
        print(f"\tReached: {reachedB}")
        frontierF, reachedF = BFS('F', graph, frontierF, reachedF)
        frontierB, reachedB = BFS('B', graph, frontierB, reachedB)
        intersectingNode = isIntersecting(reachedF, reachedB)
        if intersectingNode != -1:
            print("From front: ")
            print(f"\tFrontier: {frontierF}")
            print(f"\tReached: {reachedF}")
            print("From back: ")
            print(f"\tFrontier: {frontierB}")
            print(f"\tReached: {reachedB}")
            print("Path found!")
            path = reachedF[:-1] + reachedB[::-1]
            return path
    print("No path found!")
    return []

def create_graph():
    graph = {}
    n = int(input("Enter number of nodes in graph: "))
```

Output 2 :

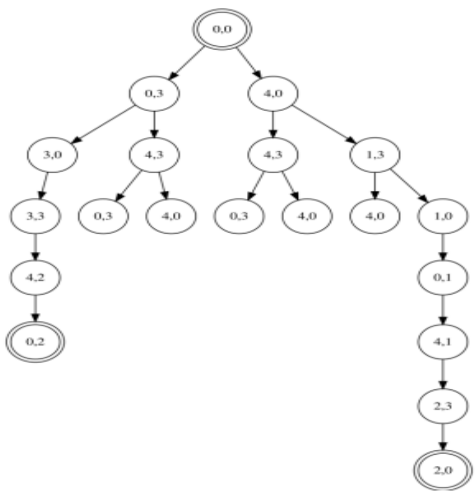
```
Enter number of nodes in graph: 7
A B C D E F G are the nodes
Enter the child nodes of A: B C
Enter the child nodes of B: D
Enter the child nodes of C: D E
Enter the child nodes of D: F
Enter the child nodes of E: F G
Enter the child nodes of F: 0
Enter the child nodes of G: 0
Enter source node: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



```
for i in range(n):
        print(chr(65 + i), end=' ')
    print("are the nodes")
    for i in range(n):
        node = chr(65 + i)
        children = input(f"Enter the child nodes of {node}: ").split()
        graph[node] = {'c': [], 'p': []}
        for child in children:
            if child != '0':
                graph[node]['c'].append(child)
                graph[child]['p'].append(node)
    return graph

s = input("Enter source node: ")
g = input("Enter goal node: ")

graph = create_graph()
path = BidirectionalSearch(graph, s, g)
if len(path):
    print("Path:", path)
```

Output 1 :

```
Enter capacity of first jug: 4
Enter capacity of second jug: 3
Enter target volume: 2
BFS / DFS: BFS
Frontier: [(0, 0)]
Reached: [(0, 0)]
Frontier: [(4, 0), (0, 3)]
Reached: [(0, 0), (4, 0), (0, 3)]
Frontier: [(0, 3), (4, 3), (1, 3)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (1, 3)]
Frontier: [(4, 3), (1, 3), (3, 0)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (1, 3), (3, 0)]
Frontier: [(1, 3), (3, 0)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (1, 3), (3, 0)]
Frontier: [(3, 0), (1, 0)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (1, 3), (3, 0), (1, 0)]
Frontier: [(1, 0), (3, 3)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (1, 3), (3, 0), (1, 0), (3, 3)]
Frontier: [(3, 3), (0, 1)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (1, 3), (3, 0), (1, 0), (3, 3), (
0, 1)]
Frontier: [(0, 1), (4, 2)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (1, 3), (3, 0), (1, 0), (3, 3), (
0, 1), (4, 2)]
Frontier: [(4, 2), (4, 1)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (1, 3), (3, 0), (1, 0), (3, 3), (
0, 1), (4, 2), (4, 1)]
Frontier: [(4, 1), (0, 2)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (1, 3), (3, 0), (1, 0), (3, 3), (
0, 1), (4, 2), (4, 1), (0, 2)]
Frontier: [(0, 2), (2, 3)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (1, 3), (3, 0), (1, 0), (3, 3), (
0, 1), (4, 2), (4, 1), (0, 2), (2, 3)]
Solution path
(0, 0)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
(0, 2)
```



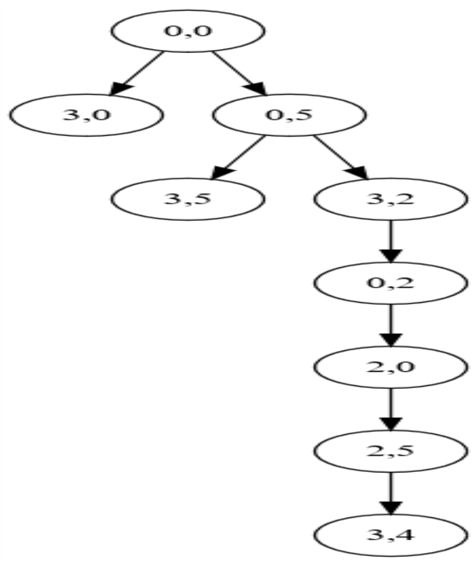
2a. AIM : Implement water jug problem with Search tree generation using BFS

Source Code :

```
def generateStates(state, capacity1, capacity2):
    x, y = state
    states = []
    if x < capacity1:
        states.append((capacity1, y))
    if y < capacity2:
        states.append((x, capacity2))
    if x > 0:
        states.append((0, y))
    if y > 0:
        states.append((x, 0))
    if x+y <= capacity1 and y > 0:
        states.append((x+y, 0))
    if x+y <= capacity2 and x > 0:
        states.append((0, x+y))
    if x+y >= capacity1 and y > 0:
        states.append((capacity1, x+y-capacity1))
    if x+y >= capacity2 and x > 0:
        states.append((x+y-capacity2, capacity2))
    return states

def water_jug_problem(searchAlgo, capacity1, capacity2, target):
    state = (0, 0)
    reached = {state: None}
    frontier = [state]
    if searchAlgo == 'BFS':
        popping = 0
    elif searchAlgo == 'DFS':
        popping = -1
    while frontier:
```

Output 2 :

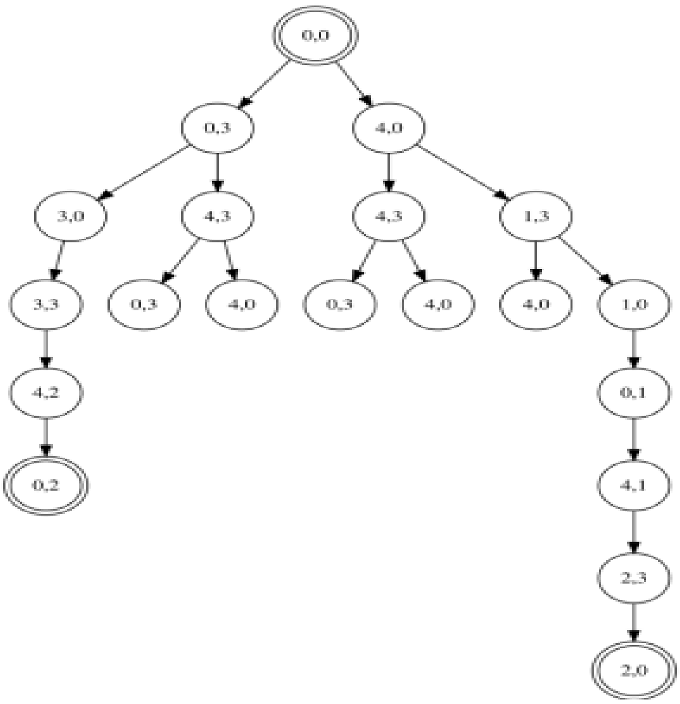
```
Enter capacity of first jug: 5
Enter capacity of second jug: 3
Enter target volume: 4
BFS / DFS: BFS
Frontier: [(0, 0)]
Reached: [(0, 0)]
Frontier: [(5, 0), (0, 3)]
Reached: [(0, 0), (5, 0), (0, 3)]
Frontier: [(0, 3), (5, 3), (2, 3)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3)]
Frontier: [(5, 3), (2, 3), (3, 0)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3), (3, 0)]
Frontier: [(2, 3), (3, 0)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3), (3, 0)]
Frontier: [(3, 0), (2, 0)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3), (3, 0), (2, 0)]
Frontier: [(2, 0), (3, 3)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3), (3, 0), (2, 0), (3, 3
)]
Frontier: [(3, 3), (0, 2)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3), (3, 0), (2, 0), (3, 3
), (0, 2)]
Frontier: [(0, 2), (5, 1)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3), (3, 0), (2, 0), (3, 3
), (0, 2), (5, 1)]
Frontier: [(5, 1), (5, 2)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3), (3, 0), (2, 0), (3, 3
), (0, 2), (5, 1), (5, 2)]
Frontier: [(5, 2), (0, 1)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3), (3, 0), (2, 0), (3, 3
), (0, 2), (5, 1), (5, 2), (0, 1)]
Frontier: [(0, 1), (4, 3)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3), (3, 0), (2, 0), (3, 3
), (0, 2), (5, 1), (5, 2), (0, 1), (4, 3)]
Frontier: [(4, 3), (1, 0)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3), (3, 0), (2, 0), (3, 3
), (0, 2), (5, 1), (5, 2), (0, 1), (4, 3), (1, 0)]
Frontier: [(1, 0), (4, 0)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (2, 3), (3, 0), (2, 0),
(3, 3), (0, 2), (5, 1), (5, 2), (0, 1),
 (4, 3), (1, 0), (4, 0)]
Frontier: [(4, 0), (1, 3)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3),
(2, 3), (3, 0), (2, 0), (3, 3), (0, 2),
(5, 1), (5, 2), (0, 1), (4, 3), (1, 0),
(4, 0), (1, 3)]
Solution path
(0, 0)
(5, 0)
(2, 3)
(2, 0)
(0, 2)
(5, 2)
(4, 3)
(4, 0)
```



```
print(f"Frontier: {frontier} ")
        print(f"Reached: {list(reached.keys())} ")
        state = frontier.pop(popping)
        if state == (target, 0) or state == (0, target):
            path = []
            while state:
                path.append(state)
                state = reached[state]
            path.reverse()
            return path
        states = generateStates(state, capacity1, capacity2)
        for new_state in states:
            if new_state not in reached:
                reached[new_state] = state
                frontier.append(new_state)
    return None

capacity1 = int(input("Enter capacity of first jug: "))
capacity2 = int(input("Enter capacity of second jug: "))
target = int(input("Enter target volume: "))
algo = input("BFS / DFS: ")
path = water_jug_problem(algo, capacity1, capacity2, target)
if path is None:
    print("No solution found.")
else:
    print("Solution path")
    for state in path:
        print(state)
```

Output 1 :

```
Enter capacity of first jug: 4
Enter capacity of second jug: 3
Enter target volume: 2
BFS / DFS: DFS
Frontier: [(0, 0)]
Reached: [(0, 0)]
Frontier: [(4, 0), (0, 3)]
Reached: [(0, 0), (4, 0), (0, 3)]
Frontier: [(4, 0), (4, 3), (3, 0)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (3, 0)]
Frontier: [(4, 0), (4, 3), (3, 3)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (3, 0), (3, 3)]
Frontier: [(4, 0), (4, 3), (4, 2)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (3, 0), (3, 3), (4, 2)]
Frontier: [(4, 0), (4, 3), (0, 2)]
Reached: [(0, 0), (4, 0), (0, 3), (4, 3), (3, 0), (3, 3), (4, 2), (0, 2
)]
Solution path
(0, 0)
(0, 3)
(3, 0)
(3, 3)
(4, 2)
(0, 2)
```



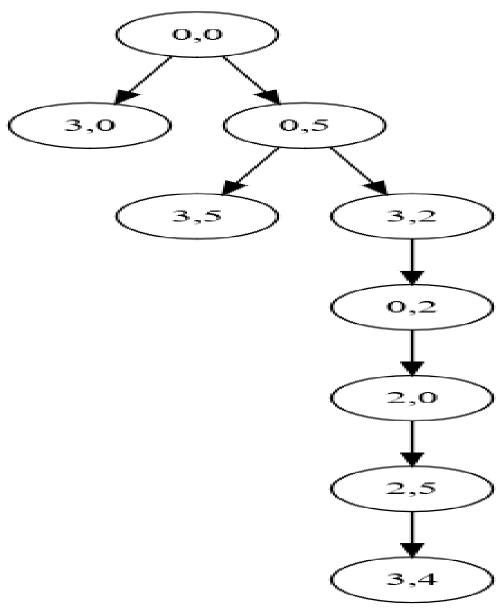
2b .AIM : Implement water jug problem with Search tree generation using DFS

Source Code :

```
def generateStates(state, capacity1, capacity2):
    x, y = state
    states = []
    if x < capacity1:
        states.append((capacity1, y))
    if y < capacity2:
        states.append((x, capacity2))
    if x > 0:
        states.append((0, y))
    if y > 0:
        states.append((x, 0))
    if x+y <= capacity1 and y > 0:
        states.append((x+y, 0))
    if x+y <= capacity2 and x > 0:
        states.append((0, x+y))
    if x+y >= capacity1 and y > 0:
        states.append((capacity1, x+y-capacity1))
    if x+y >= capacity2 and x > 0:
        states.append((x+y-capacity2, capacity2))
    return states

def water_jug_problem(searchAlgo, capacity1, capacity2, target):
    state = (0, 0)
    reached = {state: None}
    frontier = [state]
    if searchAlgo == 'BFS':
        popping = 0
    elif searchAlgo == 'DFS':
        popping = -1
    while frontier:
        print(f"Frontier: {frontier} ")
        print(f"Reached: {list(reached.keys())} ")
        state = frontier.pop(popping)
        if state == (target, 0) or state == (0, target):
            path = []
            while state:
                path.append(state)
                state = reached[state]
            path.reverse()
            return path
        states = generateStates(state, capacity1, capacity2)
        for new_state in states:
            if new_state not in reached:
                reached[new_state] = state
```

Output 2 :

```
Enter capacity of first jug: 5
Enter capacity of second jug: 3
Enter target volume: 4
BFS / DFS: DFS
Frontier: [(0, 0)]
Reached: [(0, 0)]
Frontier: [(5, 0), (0, 3)]
Reached: [(0, 0), (5, 0), (0, 3)]
Frontier: [(5, 0), (5, 3), (3, 0)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (3, 0)]
Frontier: [(5, 0), (5, 3), (3, 3)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (3, 0), (3, 3)]
Frontier: [(5, 0), (5, 3), (5, 1)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (3, 0), (3, 3), (5, 1)]
Frontier: [(5, 0), (5, 3), (0, 1)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (3, 0), (3, 3), (5, 1), (0, 1
)]
Frontier: [(5, 0), (5, 3), (1, 0)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (3, 0), (3, 3), (5, 1), (0, 1
), (1, 0)]
Frontier: [(5, 0), (5, 3), (1, 3)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (3, 0), (3, 3), (5, 1), (0, 1
), (1, 0), (1, 3)]
Frontier: [(5, 0), (5, 3), (4, 0)]
Reached: [(0, 0), (5, 0), (0, 3), (5, 3), (3, 0), (3, 3), (5, 1), (0, 1
), (1, 0), (1, 3), (4, 0)]
Solution path
(0, 0)
(0, 3)
(3, 0)
(3, 3)
(5, 1)
(0, 1)
(1, 0)
(1, 3)
(4, 0)
```



```
                frontier.append(new_state)
    return None

capacity1 = int(input("Enter capacity of first jug: "))
capacity2 = int(input("Enter capacity of second jug: "))
target = int(input("Enter target volume: "))
algo = input("BFS / DFS: ")
path = water_jug_problem(algo, capacity1, capacity2, target)
if path is None:
    print("No solution found.")
else:
    print("Solution path")
    for state in path:
        print(state)
```

Output :

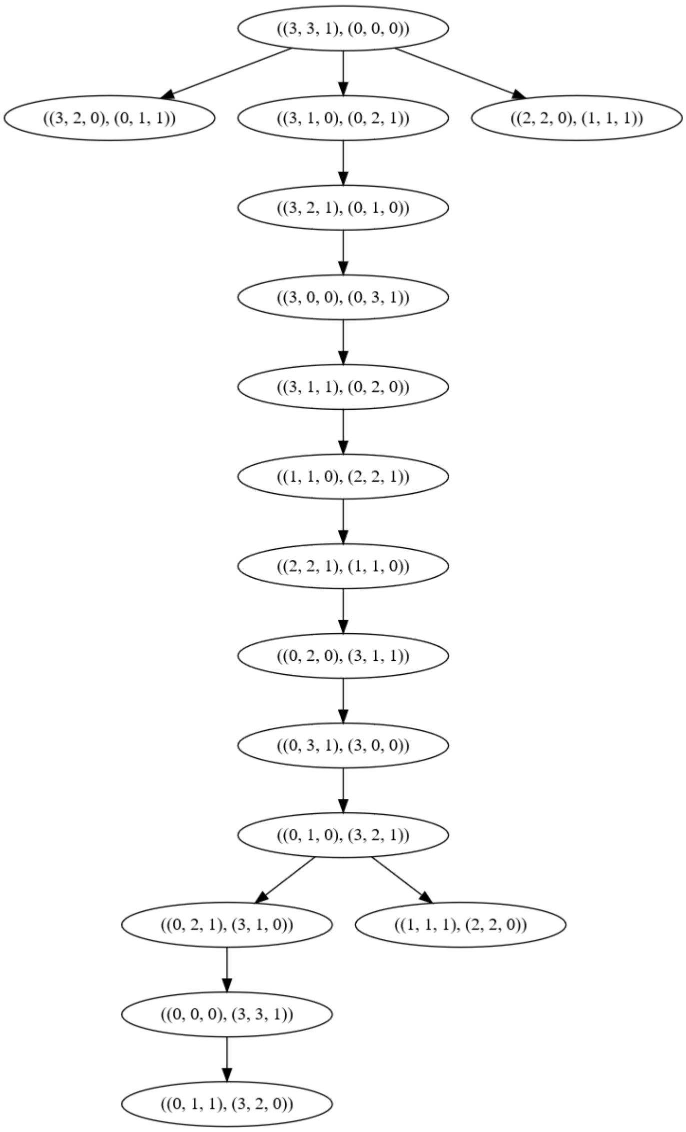
```
Enter number of missionaries: 3
Enter number of cannibals: 3
BFS/DFS: BFS
Frontier: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 2 CL: 2 B: 1 MR: 1 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1]
Frontier: [ML: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1]
Frontier: [ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2 B: 0 MR: 0 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1]
Frontier: [ML: 3 CL: 2 B: 0 MR: 0 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1]
Frontier: [ML: 3 CL: 0 B: 1 MR: 0 CR: 3]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3]
Frontier: [ML: 3 CL: 1 B: 0 MR: 0 CR: 2]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2]
Frontier: [ML: 1 CL: 1 B: 1 MR: 2 CR: 2]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2]
Frontier: [ML: 2 CL: 2 B: 0 MR: 1 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1]
Frontier: [ML: 0 CL: 2 B: 1 MR: 3 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1, ML:
0 CL: 2 B: 1 MR: 3 CR: 1]
Frontier: [ML: 0 CL: 3 B: 0 MR: 3 CR: 0]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1, ML:
0 CL: 2 B: 1 MR: 3 CR: 1, ML: 0 CL: 3 B: 0 MR: 3 CR: 0]
Frontier: [ML: 0 CL: 1 B: 1 MR: 3 CR: 2]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1, ML:
```

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

Source Code :

```
class State:
    def __init__(self, ml, cl, mr, cr, b, maxM, maxC):
        self.parent , self.actions , self.ml , self.cl = None,[],ml,cl
        self.mr, self.cr, self.b, self.maxM, self.maxC = mr,cr,b,maxM,maxC
    def is_valid(self):
        if self.ml < 0 or self.mr < 0 or self.cl < 0 or self.cr < 0:
            return False
        elif self.ml > self.maxM or self.mr > self.maxM or self.cl > self.maxC or self.cr > self.maxC:
            return False
        elif self.ml + self.mr != self.maxM or self.cl + self.cr != self.maxC:
            return False
        elif self.ml == 0 and (self.mr < self.cr):
            return False
        elif self.mr == 0 and (self.ml < self.cl):
            return False
        elif self.ml != 0 and self.mr != 0 and (self.ml < self.cl or self.mr < self.cr):
            return False
        return True
    def expand(self):
        passengers = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]
        for m, c in passengers:
            if self.b == 0:
                newState = State(self.ml-m, self.cl-c, self.mr+m, self.cr+c, 1, self.maxM, self.maxC)
            else:
                newState = State(self.ml+m, self.cl+c, self.mr-m, self.cr-c, 0, self.maxM, self.maxC)
            if newState.is_valid():
                newState.parent = self
                self.actions.append(newState)
    def printPath(self):
        node = self
        path = []
```

```
0 CL: 2 B: 1 MR: 3 CR: 1, ML: 0 CL: 3 B: 0 MR: 3 CR: 0, ML: 0 CL: 1 B:
1 MR: 3 CR: 2]
Frontier: [ML: 1 CL: 1 B: 0 MR: 2 CR: 2, ML: 0 CL: 2 B: 0 MR: 3 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1, ML:
0 CL: 2 B: 1 MR: 3 CR: 1, ML: 0 CL: 3 B: 0 MR: 3 CR: 0, ML: 0 CL: 1 B:
1 MR: 3 CR: 2, ML: 1 CL: 1 B: 0 MR: 2 CR: 2, ML: 0 CL: 2 B: 0 MR: 3 CR:
1]
Frontier: [ML: 0 CL: 2 B: 0 MR: 3 CR: 1, ML: 0 CL: 0 B: 1 MR: 3 CR: 3]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1, ML:
0 CL: 2 B: 1 MR: 3 CR: 1, ML: 0 CL: 3 B: 0 MR: 3 CR: 0, ML: 0 CL: 1 B:
1 MR: 3 CR: 2, ML: 1 CL: 1 B: 0 MR: 2 CR: 2, ML: 0 CL: 2 B: 0 MR: 3 CR:
1, ML: 0 CL: 0 B: 1 MR: 3 CR: 3]
Frontier: [ML: 0 CL: 0 B: 1 MR: 3 CR: 3]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1, ML:
0 CL: 2 B: 1 MR: 3 CR: 1, ML: 0 CL: 3 B: 0 MR: 3 CR: 0, ML: 0 CL: 1 B:
1 MR: 3 CR: 2, ML: 1 CL: 1 B: 0 MR: 2 CR: 2, ML: 0 CL: 2 B: 0 MR: 3 CR:
1, ML: 0 CL: 0 B: 1 MR: 3 CR: 3]
Goal reached
ML: 3 CL: 3 B: 0 MR: 0 CR: 0
ML: 3 CL: 1 B: 1 MR: 0 CR: 2
ML: 3 CL: 2 B: 0 MR: 0 CR: 1
ML: 3 CL: 0 B: 1 MR: 0 CR: 3
ML: 3 CL: 1 B: 0 MR: 0 CR: 2
ML: 1 CL: 1 B: 1 MR: 2 CR: 2
ML: 2 CL: 2 B: 0 MR: 1 CR: 1
ML: 0 CL: 2 B: 1 MR: 3 CR: 1
ML: 0 CL: 3 B: 0 MR: 3 CR: 0
ML: 0 CL: 1 B: 1 MR: 3 CR: 2
ML: 1 CL: 1 B: 0 MR: 2 CR: 2
ML: 0 CL: 0 B: 1 MR: 3 CR: 3
Length of path: 12
```



```
        while node:
            path.append(node)
            node = node.parent
        path.reverse()
        for n in path:
            print(n)
        print(f"Length of path: {len(path)}")
    def __eq__(self, other):
        return self.ml == other.ml and self.mr == other.mr and self.cl == other.cl and self.cr == other.cr and self.b ==
other.b
    def __repr__(self):
        return f"ML: {self.ml} CL: {self.cl} B: {self.b} MR: {self.mr} CR: {self.cr}"
def MissionaryCannibal(mCount, cCount, searchAlgo):
    initialState = State(mCount, cCount, 0, 0, 0, mCount, cCount)
    goalState = State(0, 0, mCount, cCount, 1, mCount, cCount)
    frontier = [initialState]
    reached = [initialState]
    p = 0 if searchAlgo == 'BFS' else -1
    while frontier:
        print(f"Frontier: {frontier}")
        print(f"Reached: {reached}")
        node = frontier.pop(p)
        if node == goalState:
            print("Goal reached")
            node.printPath()
            return
        node.expand()
        for state in node.actions:
            if state not in reached:
                frontier.append(state)
                reached.append(state)
    print("Goal not found")
mCount = int(input("Enter number of missionaries: "))
cCount = int(input("Enter number of cannibals: "))
algo = input("BFS/DFS: ")
MissionaryCannibal(mCount, cCount, algo)
```

Output :

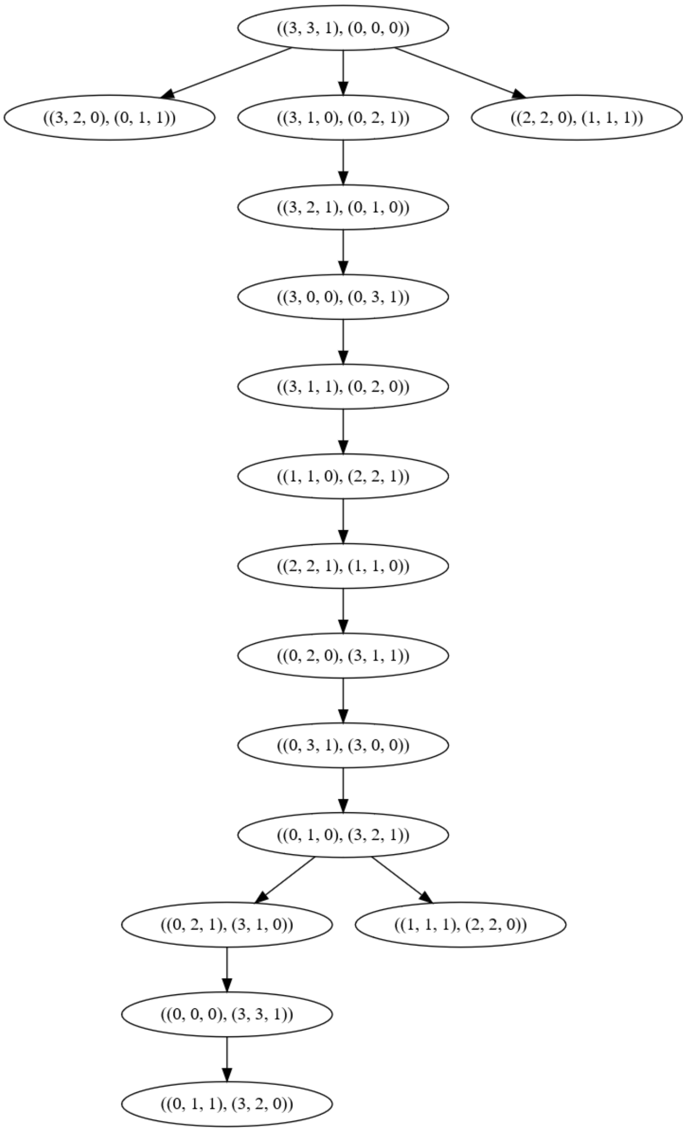
```
Enter number of missionaries: 3
Enter number of cannibals: 3
BFS/DFS: DFS
Frontier: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 2 CL: 2 B: 1 MR: 1 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 3 CL: 2 B: 0 MR: 0 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 3 CL: 0 B: 1 MR: 0 CR: 3]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 3 CL: 1 B: 0 MR: 0 CR: 2]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 1 CL: 1 B: 1 MR: 2 CR: 2]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 2 CL: 2 B: 0 MR: 1 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 0 CL: 2 B: 1 MR: 3 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1, ML:
0 CL: 2 B: 1 MR: 3 CR: 1]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 0 CL: 3 B: 0 MR: 3 CR: 0]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1, ML:
0 CL: 2 B: 1 MR: 3 CR: 1, ML: 0 CL: 3 B: 0 MR: 3 CR: 0]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 0 CL: 1 B: 1 MR: 3 CR: 2]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
```

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

Source Code :

```
class State:
    def __init__(self, ml, cl, mr, cr, b, maxM, maxC):
        self.parent , self.actions , self.ml , self.cl = None,[],ml,cl
        self.mr, self.cr, self.b, self.maxM, self.maxC = mr,cr,b,maxM,maxC
    def is_valid(self):
        if self.ml < 0 or self.mr < 0 or self.cl < 0 or self.cr < 0:
            return False
        elif self.ml > self.maxM or self.mr > self.maxM or self.cl > self.maxC or self.cr > self.maxC:
            return False
        elif self.ml + self.mr != self.maxM or self.cl + self.cr != self.maxC:
            return False
        elif self.ml == 0 and (self.mr < self.cr):
            return False
        elif self.mr == 0 and (self.ml < self.cl):
            return False
        elif self.ml != 0 and self.mr != 0 and (self.ml < self.cl or self.mr < self.cr):
            return False
        return True
    def expand(self):
        passengers = [(1, 0), (2, 0), (0, 1), (0, 2), (1, 1)]
        for m, c in passengers:
            if self.b == 0:
                newState = State(self.ml-m, self.cl-c, self.mr+m, self.cr+c, 1, self.maxM, self.maxC)
            else:
                newState = State(self.ml+m, self.cl+c, self.mr-m, self.cr-c, 0, self.maxM, self.maxC)
            if newState.is_valid():
                newState.parent = self
                self.actions.append(newState)
    def printPath(self):
        node = self
```

```
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1, ML:
0 CL: 2 B: 1 MR: 3 CR: 1, ML: 0 CL: 3 B: 0 MR: 3 CR: 0, ML: 0 CL: 1 B:
1 MR: 3 CR: 2]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 1 CL: 1 B: 0 MR: 2 CR: 2, ML: 0 CL: 2 B: 0 MR: 3 CR: 1]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1, ML:
0 CL: 2 B: 1 MR: 3 CR: 1, ML: 0 CL: 3 B: 0 MR: 3 CR: 0, ML: 0 CL: 1 B:
1 MR: 3 CR: 2, ML: 1 CL: 1 B: 0 MR: 2 CR: 2, ML: 0 CL: 2 B: 0 MR: 3 CR:
1]
Frontier: [ML: 3 CL: 2 B: 1 MR: 0 CR: 1, ML: 3 CL: 1 B: 1 MR: 0 CR: 2,
ML: 1 CL: 1 B: 0 MR: 2 CR: 2, ML: 0 CL: 0 B: 1 MR: 3 CR: 3]
Reached: [ML: 3 CL: 3 B: 0 MR: 0 CR: 0, ML: 3 CL: 2 B: 1 MR: 0 CR: 1, M
L: 3 CL: 1 B: 1 MR: 0 CR: 2, ML: 2 CL: 2 B: 1 MR: 1 CR: 1, ML: 3 CL: 2
B: 0 MR: 0 CR: 1, ML: 3 CL: 0 B: 1 MR: 0 CR: 3, ML: 3 CL: 1 B: 0 MR: 0
CR: 2, ML: 1 CL: 1 B: 1 MR: 2 CR: 2, ML: 2 CL: 2 B: 0 MR: 1 CR: 1, ML:
0 CL: 2 B: 1 MR: 3 CR: 1, ML: 0 CL: 3 B: 0 MR: 3 CR: 0, ML: 0 CL: 1 B:
1 MR: 3 CR: 2, ML: 1 CL: 1 B: 0 MR: 2 CR: 2, ML: 0 CL: 2 B: 0 MR: 3 CR:
1, ML: 0 CL: 0 B: 1 MR: 3 CR: 3]
Goal reached
ML: 3 CL: 3 B: 0 MR: 0 CR: 0
ML: 2 CL: 2 B: 1 MR: 1 CR: 1
ML: 3 CL: 2 B: 0 MR: 0 CR: 1
ML: 3 CL: 0 B: 1 MR: 0 CR: 3
ML: 3 CL: 1 B: 0 MR: 0 CR: 2
ML: 1 CL: 1 B: 1 MR: 2 CR: 2
ML: 2 CL: 2 B: 0 MR: 1 CR: 1
ML: 0 CL: 2 B: 1 MR: 3 CR: 1
ML: 0 CL: 3 B: 0 MR: 3 CR: 0
ML: 0 CL: 1 B: 1 MR: 3 CR: 2
ML: 0 CL: 2 B: 0 MR: 3 CR: 1
ML: 0 CL: 0 B: 1 MR: 3 CR: 3
Length of path: 12
```



```
        path = []
        while node:
            path.append(node)
            node = node.parent
        path.reverse()
        for n in path:
            print(n)
        print(f"Length of path: {len(path)}")
    def __eq__(self, other):
        return self.ml == other.ml and self.mr == other.mr and self.cl == other.cl and self.cr == other.cr and self.b ==
other.b
    def __repr__(self):
        return f"ML: {self.ml} CL: {self.cl} B: {self.b} MR: {self.mr} CR: {self.cr}"
def MissionaryCannibal(mCount, cCount, searchAlgo):
    initialState = State(mCount, cCount, 0, 0, 0, mCount, cCount)
    goalState = State(0, 0, mCount, cCount, 1, mCount, cCount)
    frontier = [initialState]
    reached = [initialState]
    p = 0 if searchAlgo == 'BFS' else -1
    while frontier:
        print(f"Frontier: {frontier}")
        print(f"Reached: {reached}")
        node = frontier.pop(p)
        if node == goalState:
            print("Goal reached")
            node.printPath()
            return
        node.expand()
        for state in node.actions:
            if state not in reached:
                frontier.append(state)
                reached.append(state)
    print("Goal not found")
mCount = int(input("Enter number of missionaries: "))
cCount = int(input("Enter number of cannibals: "))
algo = input("BFS/DFS: ")
MissionaryCannibal(mCount, cCount, algo)
```

Output 1 :

```
Left room Dirty(D)/Clean(C): D
Right room Dirty(D)/Clean(C)D
Vaccum in L/R room: L
BFS/DFS: BFS
Frontier: [World: [True, True], Vacuum: 0]
Reached: [World: [True, True], Vacuum: 0]
Frontier: [World: [True, True], Vacuum: 1, World: [False, True], Vacuum
: 0]
Reached: [World: [True, True], Vacuum: 0, World: [True, True], Vacuum:
1, World: [False, True], Vacuum: 0]
Frontier: [World: [False, True], Vacuum: 0, World: [True, False], Vacuu
m: 1]
Reached: [World: [True, True], Vacuum: 0, World: [True, True], Vacuum:
1, World: [False, True], Vacuum: 0, World: [True, False], Vacuum: 1]
Frontier: [World: [True, False], Vacuum: 1, World: [False, True], Vacuu
m: 1]
Reached: [World: [True, True], Vacuum: 0, World: [True, True], Vacuum:
1, World: [False, True], Vacuum: 0, World: [True, False], Vacuum: 1, Wo
rld: [False, True], Vacuum: 1]
Frontier: [World: [False, True], Vacuum: 1, World: [True, False], Vacuu
m: 0]
Reached: [World: [True, True], Vacuum: 0, World: [True, True], Vacuum:
1, World: [False, True], Vacuum: 0, World: [True, False], Vacuum: 1, Wo
rld: [False, True], Vacuum: 1, World: [True, False], Vacuum: 0]
Frontier: [World: [True, False], Vacuum: 0, World: [False, False], Vacu
um: 1]
Reached: [World: [True, True], Vacuum: 0, World: [True, True], Vacuum:
1, World: [False, True], Vacuum: 0, World: [True, False], Vacuum: 1, Wo
rld: [False, True], Vacuum: 1, World: [True, False], Vacuum: 0, World:
[False, False], Vacuum: 1]
Frontier: [World: [False, False], Vacuum: 1, World: [False, False], Vac
uum: 0]
Reached: [World: [True, True], Vacuum: 0, World: [True, True], Vacuum:
1, World: [False, True], Vacuum: 0, World: [True, False], Vacuum: 1, Wo
rld: [False, True], Vacuum: 1, World: [True, False], Vacuum: 0, World:
[False, False], Vacuum: 1, World: [False, False], Vacuum: 0]
Goal state reached!
World: [True, True], Vacuum: 0
World: [False, True], Vacuum: 0
World: [False, True], Vacuum: 1
World: [False, False], Vacuum: 1
Path cost:3
```

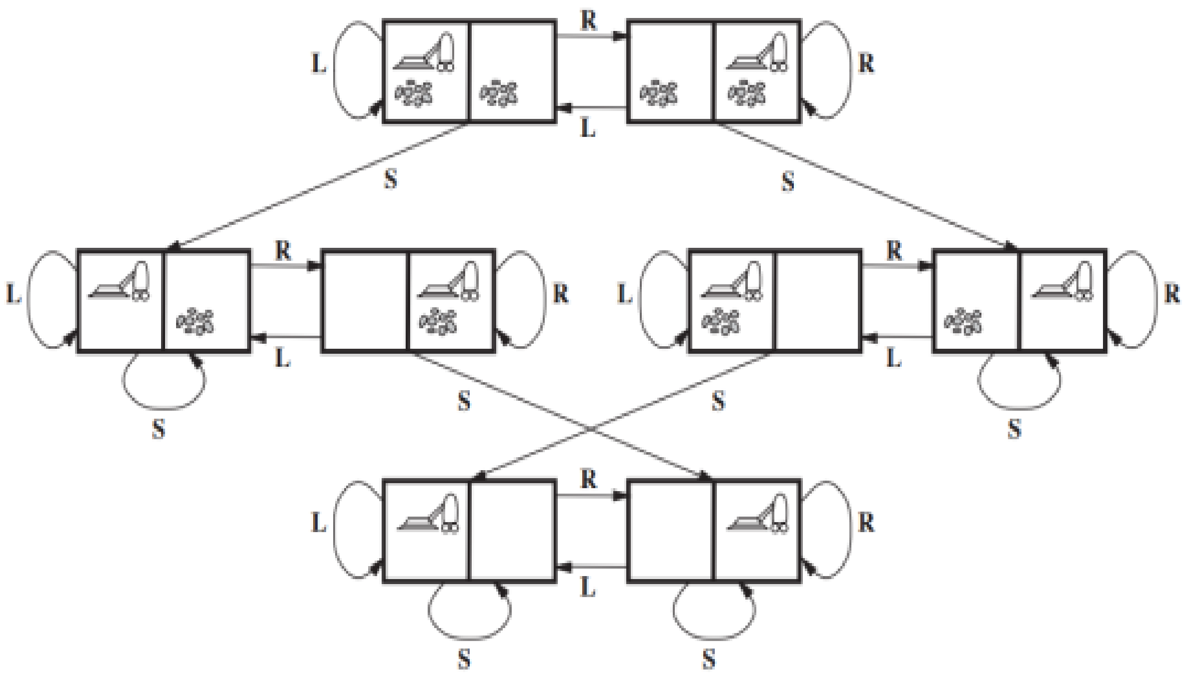
4a. AIM : Implement Vacuum World problem with Search tree generation using BFS.

Source Code :

```
class Node:
    def __init__(self, world, vacuum):
        self.parent = None
        self.children = []
        self.world = world
        self.vacuum = vacuum
    def goalTest(self):
        return (not self.world[0] and not self.world[1])
    def __eq__(self, other):
        return isinstance(other, Node) and self.world == other.world and self.vacuum == other.vacuum
    def moveLeft(self):
        if self.vacuum == 1:
            child = Node(self.world, 0)
            child.parent = self
            self.children.append(child)
    def moveRight(self):
        if self.vacuum == 0:
            child = Node(self.world, 1)
            child.parent = self
            self.children.append(child)
    def suck(self):
        if self.world[self.vacuum]:
            w = self.world[:]
            w[self.vacuum] = False
            child = Node(w, self.vacuum)
            child.parent = self
            self.children.append(child)
    def expandNode(self):
        self.moveLeft()
        self.moveRight()
        self.suck()
```

Output 2 :

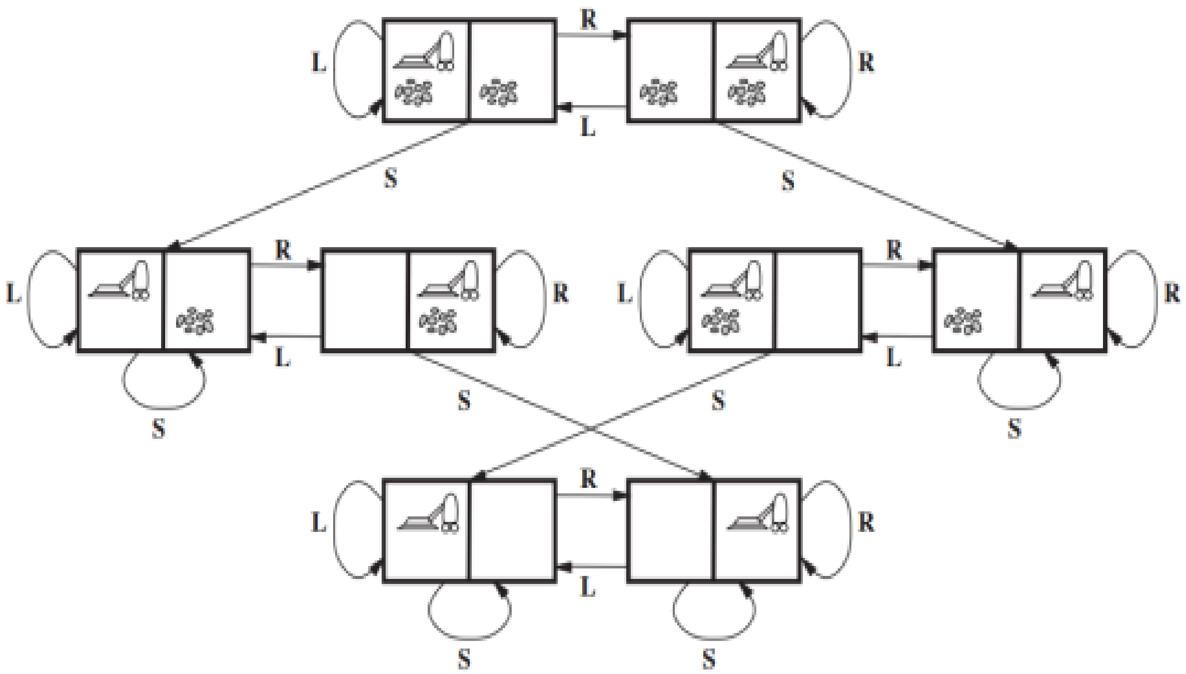
```
Left room Dirty(D)/Clean(C): C
Right room Dirty(D)/Clean(C)D
Vaccum in L/R room: R
BFS/DFS: BFS
Frontier: [World: [False, True], Vacuum: 1]
Reached: [World: [False, True], Vacuum: 1]
Frontier: [World: [False, True], Vacuum: 0, World: [False, False], Vacu
um: 1]
Reached: [World: [False, True], Vacuum: 1, World: [False, True], Vacuum
: 0, World: [False, False], Vacuum: 1]
Frontier: [World: [False, False], Vacuum: 1]
Reached: [World: [False, True], Vacuum: 1, World: [False, True], Vacuum
: 0, World: [False, False], Vacuum: 1]
Goal state reached!
World: [False, True], Vacuum: 1
World: [False, False], Vacuum: 1
Path cost :1
```



```
    def __repr__(self):
        return f"World: {self.world}, Vacuum: {self.vacuum}"
    def printPath(self):
        node = self
        path = [node]
        while node.parent:
            node = node.parent
            path.append(node)
        path = path[::-1]
        for p in path:
            print(p)
def search(world, vacuum, algo):
    node = Node(world, vacuum)
    frontier = [node]
    reached = [node]
    popping = 0 if algo == 'BFS' else -1
    while frontier:
        print(f"Frontier: {frontier}")
        print(f"Reached: {reached}")
        node = frontier.pop(popping)
        if node.goalTest():
            print("Goal state reached!")
            node.printPath()
            return
        node.expandNode()
        for child in node.children:
            if child not in reached:
                reached.append(child)
                frontier.append(child)
    print("Goal not found")
    return
l = input("Left room Dirty(D)/Clean(C): ") == 'D'
r = input("Right room Dirty(D)/Clean(C)") == 'D'
v = input("Vaccum in L/R room: ")
v = 0 if v == 'L' else 1
algo = input("BFS/DFS: ")
search([l, r], v, algo)
```

Output 1 :

```
Left room Dirty(D)/Clean(C): D
Right room Dirty(D)/Clean(C)D
Vaccum in L/R room: L
BFS/DFS: DFS
Frontier: [World: [True, True], Vacuum: 0]
Reached: [World: [True, True], Vacuum: 0]
Frontier: [World: [True, True], Vacuum: 1, World: [False, True], Vacuum
: 0]
Reached: [World: [True, True], Vacuum: 0, World: [True, True], Vacuum:
1, World: [False, True], Vacuum: 0]
Frontier: [World: [True, True], Vacuum: 1, World: [False, True], Vacuum
: 1]
Reached: [World: [True, True], Vacuum: 0, World: [True, True], Vacuum:
1, World: [False, True], Vacuum: 0, World: [False, True], Vacuum: 1]
Frontier: [World: [True, True], Vacuum: 1, World: [False, False], Vacuu
m: 1]
Reached: [World: [True, True], Vacuum: 0, World: [True, True], Vacuum:
1, World: [False, True], Vacuum: 0, World: [False, True], Vacuum: 1, Wo
rld: [False, False], Vacuum: 1]
Goal state reached!
World: [True, True], Vacuum: 0
World: [False, True], Vacuum: 0
World: [False, True], Vacuum: 1
World: [False, False], Vacuum: 1
```



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

Source Code :

```
class Node:
  def __init__(self, world, vacuum):
    self.parent = None
    self.children = []
    self.world = world
    self.vacuum = vacuum
  def goalTest(self):
    return (not self.world[0] and not self.world[1])
  def __eq__(self, other):
    return isinstance(other, Node) and self.world == other.world and self.vacuum == other.vacuum
  def moveLeft(self):
    if self.vacuum == 1:
      child = Node(self.world, 0)
      child.parent = self
      self.children.append(child)
  def moveRight(self):
    if self.vacuum == 0:
      child = Node(self.world, 1)
      child.parent = self
      self.children.append(child)
  def suck(self):
    if self.world[self.vacuum]:
      w = self.world[:]
      w[self.vacuum] = False
      child = Node(w, self.vacuum)
      child.parent = self
      self.children.append(child)
  def expandNode(self):
    self.moveLeft()
```

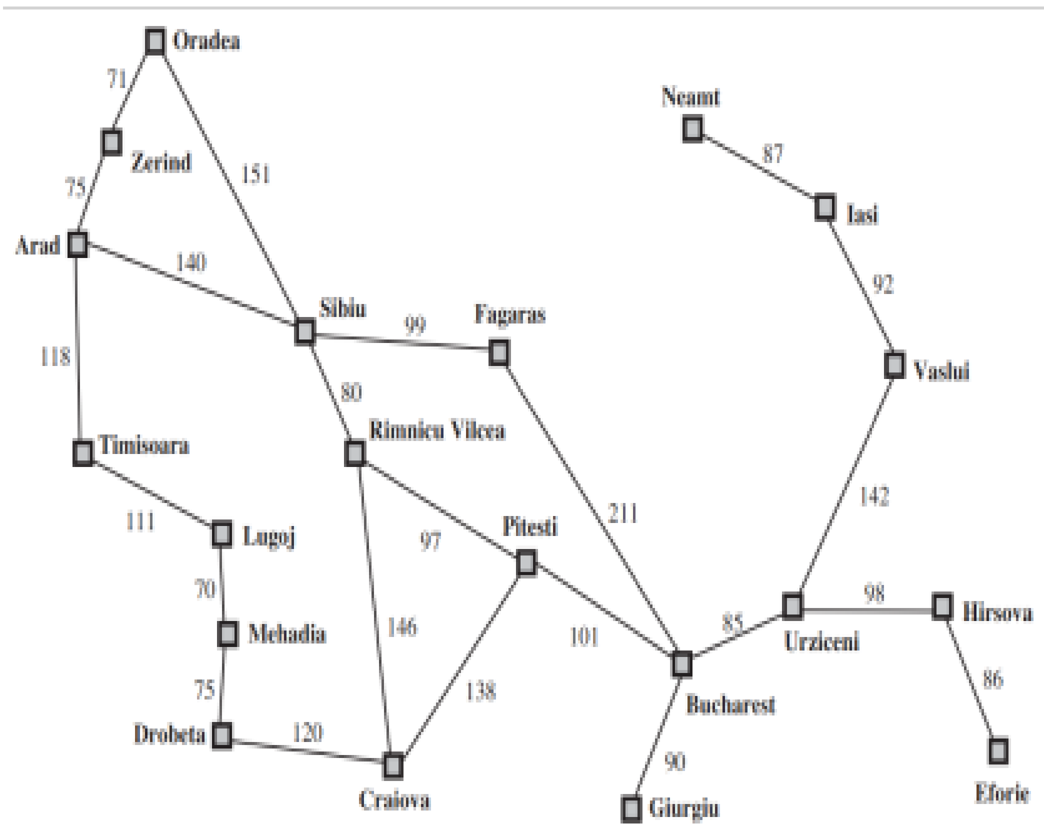
Output 2 :

```
Left room Dirty(D)/Clean(C): C
Right room Dirty(D)/Clean(C)D
Vaccum in L/R room: R
BFS/DFS: DFS
Frontier: [World: [False, True], Vacuum: 1]
Reached: [World: [False, True], Vacuum: 1]
Frontier: [World: [False, True], Vacuum: 0, World: [False, False], Vacu
um: 1]
Reached: [World: [False, True], Vacuum: 1, World: [False, True], Vacuum
: 0, World: [False, False], Vacuum: 1]
Goal state reached!
World: [False, True], Vacuum: 1
World: [False, False], Vacuum: 1
Path cost : 1
```

```
        self.moveRight()
        self.suck()
    def __repr__(self):
        return f"World: {self.world}, Vacuum: {self.vacuum}"
    def printPath(self):
        node = self
        path = [node]
        while node.parent:
            node = node.parent
            path.append(node)
        path = path[::-1]
        for p in path:
            print(p)
def search(world, vacuum, algo):
    node = Node(world, vacuum)
    frontier = [node]
    reached = [node]
    popping = 0 if algo == 'BFS' else -1
    while frontier:
        print(f"Frontier: {frontier}")
        print(f"Reached: {reached}")
        node = frontier.pop(popping)
        if node.goalTest():
            print("Goal state reached!")
            node.printPath()
            return
        node.expandNode()
        for child in node.children:
            if child not in reached:
                reached.append(child)
                frontier.append(child)
    print("Goal not found")
    return
l = input("Left room Dirty(D)/Clean(C): ") == 'D'
r = input("Right room Dirty(D)/Clean(C)") == 'D'
v = input("Vaccum in L/R room: ")
v = 0 if v == 'L' else 1
algo = input("BFS/DFS: ")
search([l, r], v, algo)
```

Output 1 :

```
Enter source node: A
Frontier: {'A': 366}
Reached: ['A']
A  :  366
Frontier: {'S': 253, 'T': 329, 'Z': 374}
Reached: ['A', 'S', 'T', 'Z']
S  :  253
Frontier: {'T': 329, 'Z': 374, 'F': 176, 'O': 380, 'R': 193}
Reached: ['A', 'S', 'T', 'Z', 'F', 'O', 'R']
F  :  176
Frontier: {'T': 329, 'Z': 374, 'O': 380, 'R': 193, 'B': 0}
Reached: ['A', 'S', 'T', 'Z', 'F', 'O', 'R', 'B']
B  :  0
Goal reached
```



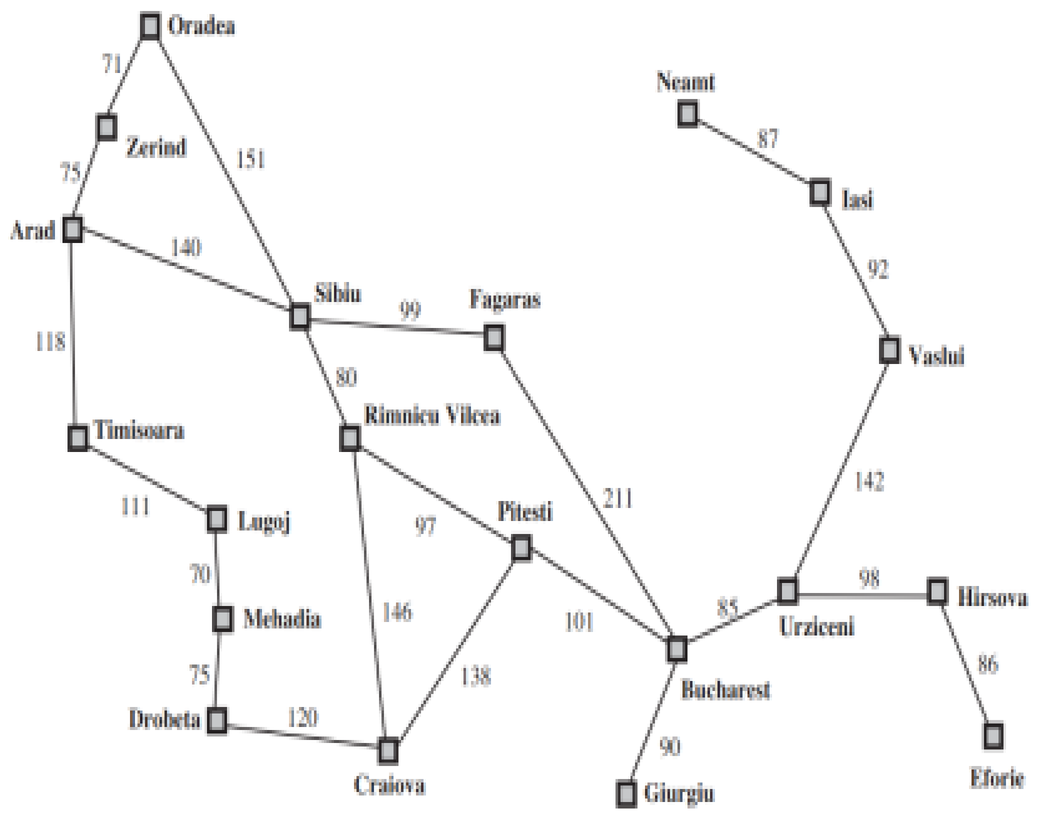
5a. AIM : Implement the Greedy Best First Search

Source Code :

```
romania = {
'A' : {'S' : 140, 'T' : 118, 'Z' : 75},
'B' : {'F' : 211, 'G' : 90, 'P' : 101, 'U' : 85},
'C' : {'D' : 120, 'P' : 138, 'R' : 146},
'D' : {'C' : 120, 'M' : 75},
'E' : {'H' : 86},
'F' : {'B' : 211, 'S' : 99},
'G' : {'B' : 90},
'H' : {'E' : 86, 'U' : 98},
'I' : {'N' : 87, 'V' : 92},
'L' : {'M' : 70, 'T' : 111},
'M' : {'D' : 75, 'L' : 70},
'N' : {'I' : 87},
'O' : {'S' : 151, 'Z' : 71},
'P' : {'B' : 101, 'C' : 138, 'R' : 97},
'R' : {'C' : 146, 'P' : 97, 'S' : 80},
'S' : {'A' : 140, 'F' : 99, 'O' : 151, 'R' : 80},
'T' : {'A' : 118, 'L' : 111},
'U' : {'B' : 85, 'H' : 98, 'V' : 142},
'V' : {'I' : 92, 'U' : 142},
'Z' : {'A' : 75, 'O' : 71}
}
hSLD = {'A' : 366, 'B' : 0, 'C' : 160, 'D' : 242, 'E' : 161, 'F' : 176, 'G' : 77,
'H' : 151, 'I' : 226, 'L' : 244, 'M' : 241, 'N' : 234, 'O' : 380, 'P' : 100,
'R' : 193, 'S' : 253, 'T' : 329, 'U' : 80, 'V' : 199, 'Z' : 374
}
def greedyBestFirstSearch(problem, h, initial, goal):
    node = initial
```

Output 2 :

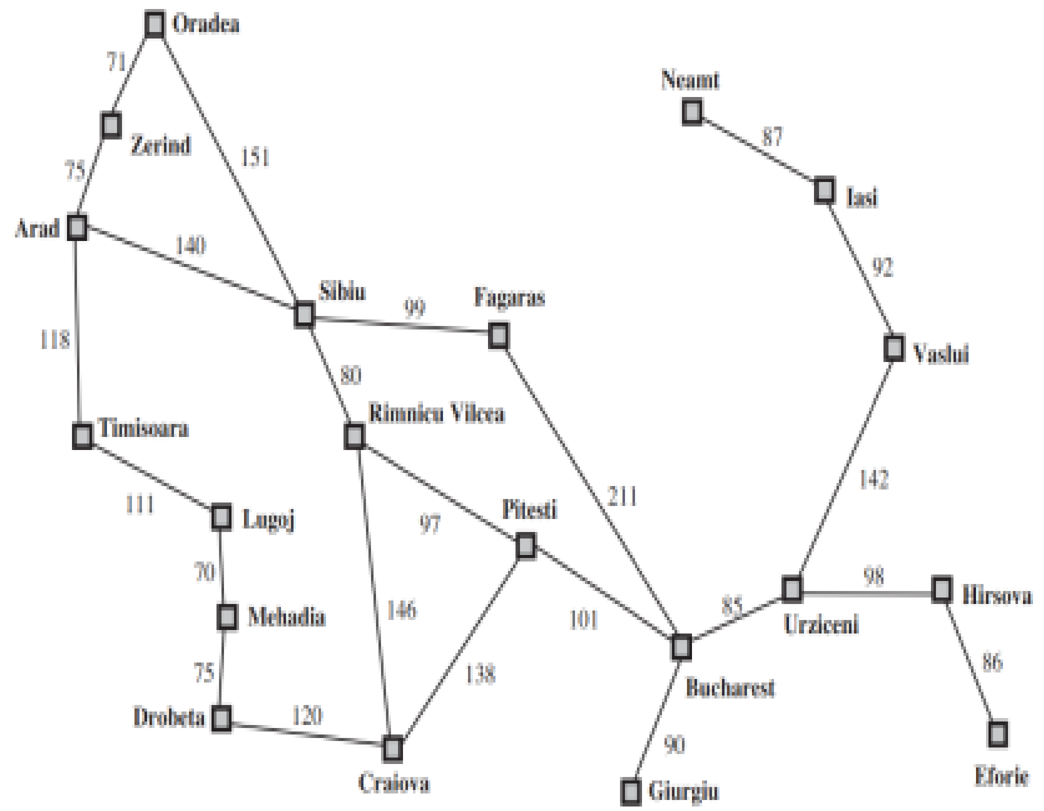
```
Enter source node: T
Frontier: {'T': 329}
Reached: ['T']
T  :  329
Frontier: {'A': 366, 'L': 244}
Reached: ['T', 'A', 'L']
L  :  244
Frontier: {'A': 366, 'M': 241}
Reached: ['T', 'A', 'L', 'M']
M  :  241
Frontier: {'A': 366, 'D': 242}
Reached: ['T', 'A', 'L', 'M', 'D']
D  :  242
Frontier: {'A': 366, 'C': 160}
Reached: ['T', 'A', 'L', 'M', 'D', 'C']
C  :  160
Frontier: {'A': 366, 'P': 100, 'R': 193}
Reached: ['T', 'A', 'L', 'M', 'D', 'C', 'P', 'R']
P  :  100
Frontier: {'A': 366, 'R': 193, 'B': 0}
Reached: ['T', 'A', 'L', 'M', 'D', 'C', 'P', 'R', 'B']
B  :  0
Goal reached
```



```
    frontier = {initial: h[initial]}
    reached = [initial]
    while frontier:
        print(f"Frontier: {frontier} ")
        print(f"Reached: {reached} ")
        node = min(frontier, key=frontier.get)
        print(node, " : ", frontier[node])
        del frontier[node]
        if node == goal:
            print("Goal reached")
            return
        for neighbour in problem[node]:
            if neighbour not in reached:
                reached.append(neighbour)
                frontier[neighbour] = h[neighbour]
    return "Goal not found!"

s = input("Enter source node: ")
greedyBestFirstSearch(romania, hSLD, s, 'B')
```

Output 1 :

```
Enter source node: A
Frontier: {'A': 366}
Reached: {'A'}
A : 366
Frontier: {'S': 393, 'T': 447, 'Z': 449}
Reached: {'S', 'Z', 'A', 'T'}
S : 393
Frontier: {'T': 447, 'Z': 449, 'A': 646, 'F': 415, 'O': 671, 'R': 413}
Reached: {'O', 'R', 'S', 'Z', 'A', 'T', 'F'}
R : 413
Frontier: {'T': 447, 'Z': 449, 'A': 646, 'F': 415, 'O': 671, 'C': 526,
'P': 417, 'S': 553}
Reached: {'O', 'R', 'C', 'P', 'S', 'Z', 'A', 'T', 'F'}
F : 415
Frontier: {'T': 447, 'Z': 449, 'A': 646, 'O': 671, 'C': 526, 'P': 417,
'S': 553, 'B': 450}
Reached: {'O', 'R', 'C', 'B', 'P', 'S', 'Z', 'A', 'T', 'F'}
P : 417
Frontier: {'T': 447, 'Z': 449, 'A': 646, 'O': 671, 'C': 526, 'S': 553,
'B': 418, 'R': 607}
Reached: {'O', 'R', 'C', 'B', 'P', 'S', 'Z', 'A', 'T', 'F'}
B : 418
Goal state reached with cost: 418
```



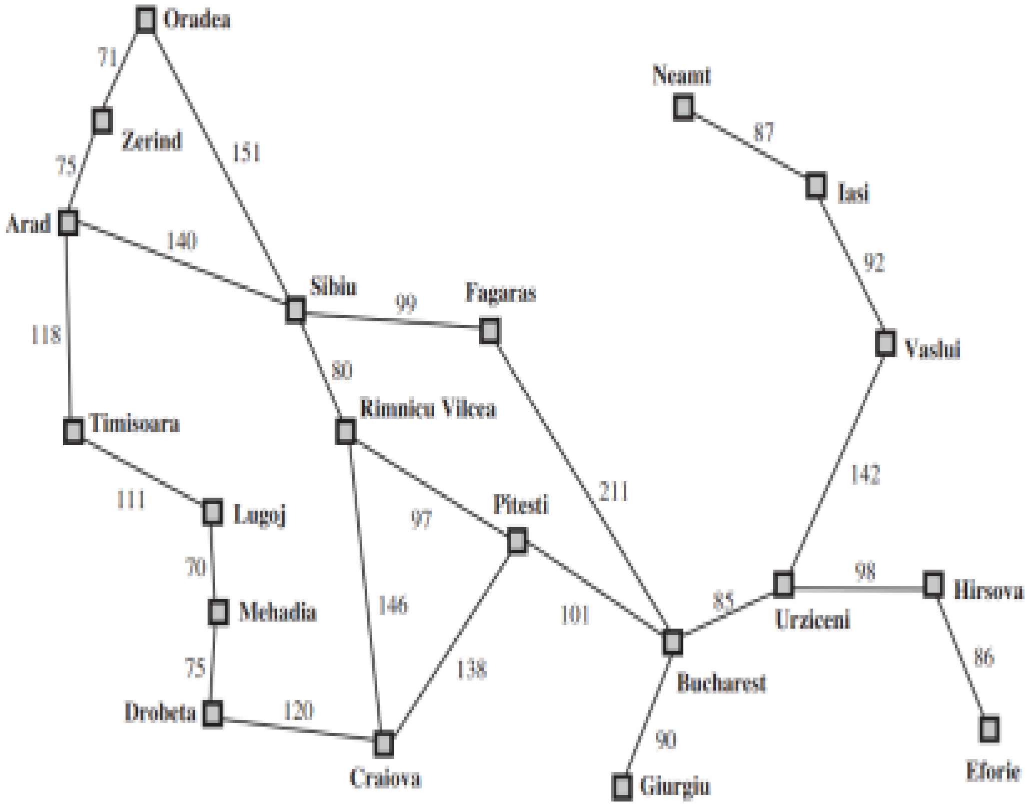
5b . AIM : Implement the A* Algorithm.

Source Code :

```
romania = {
'A' : {'S' : 140, 'T' : 118, 'Z' : 75},
'B' : {'F' : 211, 'G' : 90, 'P' : 101, 'U' : 85},
'C' : {'D' : 120, 'P' : 138, 'R' : 146},
'D' : {'C' : 120, 'M' : 75},
'E' : {'H' : 86},
'F' : {'B' : 211, 'S' : 99},
'G' : {'B' : 90},
'H' : {'E' : 86, 'U' : 98},
'I' : {'N' : 87, 'V' : 92},
'L' : {'M' : 70, 'T' : 111},
'M' : {'D' : 75, 'L' : 70},
'N' : {'I' : 87},
'O' : {'S' : 151, 'Z' : 71},
'P' : {'B' : 101, 'C' : 138, 'R' : 97},
'R' : {'C' : 146, 'P' : 97, 'S' : 80},
'S' : {'A' : 140, 'F' : 99, 'O' : 151, 'R' : 80},
'T' : {'A' : 118, 'L' : 111},
'U' : {'B' : 85, 'H' : 98, 'V' : 142},
'V' : {'I' : 92, 'U' : 142},
'Z' : {'A' : 75, 'O' : 71}
}
hSLD = {'A' : 366, 'B' : 0, 'C' : 160, 'D' : 242, 'E' : 161, 'F' : 176, 'G' : 77,
'H' : 151, 'I' : 226, 'L' : 244, 'M' : 241, 'N' : 234, 'O' : 380, 'P' : 100,
'R' : 193, 'S' : 253, 'T' : 329, 'U' : 80, 'V' : 199, 'Z' : 374
}
def aStar(problem, h, initial, goal):
  node = initial
```

Output 2 :

```
Enter source node: M
Frontier: {'M': 241}
Reached: {'M'}
M : 241
Frontier: {'D': 317, 'L': 314}
Reached: {'D', 'L', 'M'}
L : 314
Frontier: {'D': 317, 'M': 381, 'T': 510}
Reached: {'D', 'T', 'L', 'M'}
D : 317
Frontier: {'M': 381, 'T': 510, 'C': 355}
Reached: {'L', 'C', 'M', 'D', 'T'}
C : 355
Frontier: {'M': 381, 'T': 510, 'D': 557, 'P': 433, 'R': 534}
Reached: {'L', 'C', 'M', 'D', 'P', 'T', 'R'}
M : 381
Frontier: {'T': 510, 'D': 457, 'P': 433, 'R': 534, 'L': 454}
Reached: {'L', 'C', 'M', 'D', 'P', 'T', 'R'}
P : 433
Frontier: {'T': 510, 'D': 457, 'R': 534, 'L': 454, 'B': 434, 'C': 631}
Reached: {'L', 'C', 'B', 'M', 'D', 'P', 'T', 'R'}
B : 434
Goal state reached with cost: 434
```



```
    frontier = {node: h[node]+0}
    pathCost = {node: 0}
    reached = set([node])
    while frontier:
        print(f"Frontier: {frontier} ")
        print(f"Reached: {reached} ")
        node = min(frontier, key=frontier.get)
        val = frontier[node]
        del frontier[node]
        print(node, ":", val)
        if node == goal:
            return f"Goal state reached with cost: {val}"
        for child in problem[node]:
            if child not in reached or child not in frontier:
                reached.add(child)
                pathCost[child] = pathCost[node] + problem[node][child]
                frontier[child] = pathCost[child] + h[child]
            elif child in frontier and problem[node][child] + pathCost[node] + h[child] < frontier[child]:
                pathCost[child] = pathCost[node] + problem[node][child]
                frontier[child] = pathCost[child] + h[child]
    return "Goal state not found"
s = input("Enter source node: " )
ans = aStar(romania, hSLD, s, 'B' )
print(ans)
```

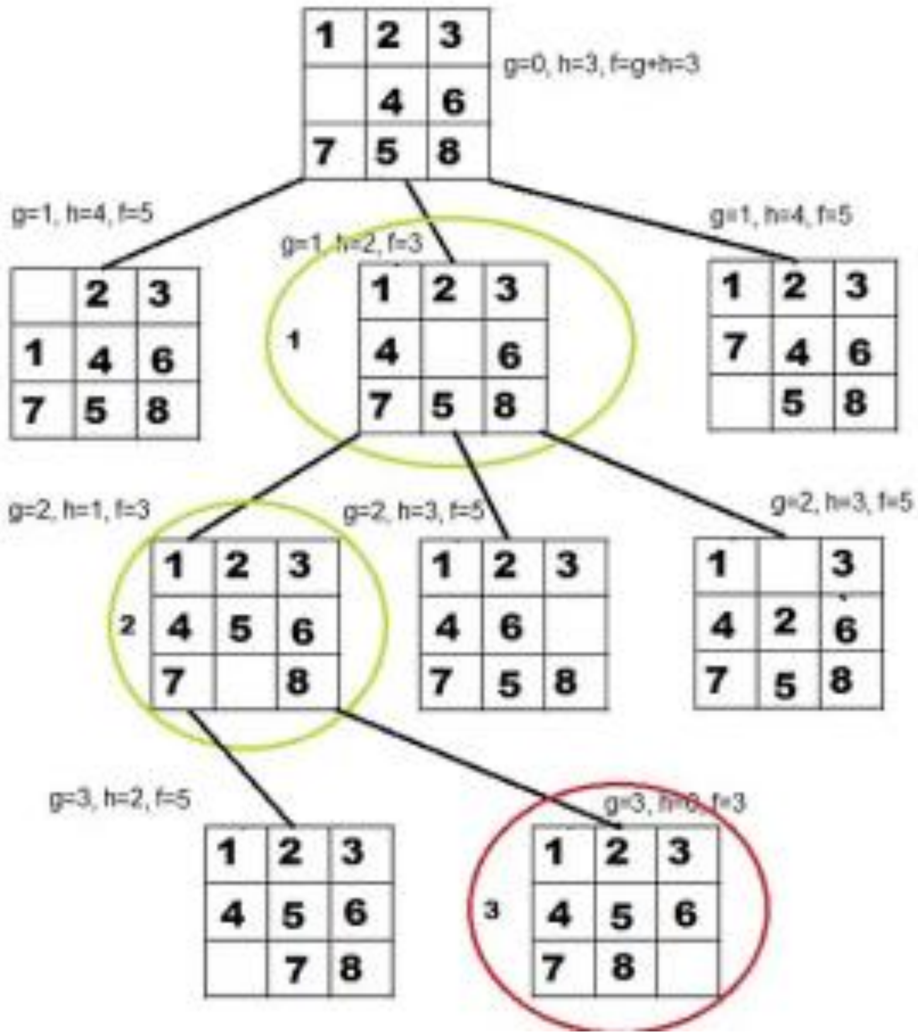
Output 1 :

```
Enter the initial state:
1 2 3
0 4 6
7 5 8
Enter the goal state:
1 2 3
4 5 6
7 8 0
Frontier: [[[1, 2, 3], [0, 4, 6], [7, 5, 8]]]
Reached: [[[1, 2, 3], [0, 4, 6], [7, 5, 8]]]
Frontier: [[[1, 2, 3], [4, 0, 6], [7, 5, 8]], [[1, 2, 3], [7, 4, 6], [0
, 5, 8]], [[0, 2, 3], [1, 4, 6], [7, 5, 8]]]
Reached: [[[1, 2, 3], [0, 4, 6], [7, 5, 8]], [[1, 2, 3], [4, 0, 6], [7,
5, 8]], [[0, 2, 3], [1, 4, 6], [7, 5, 8]], [[1, 2, 3], [7, 4, 6], [0, 5
, 8]]]
Frontier: [[[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 6, 0], [7
, 5, 8]], [[1, 2, 3], [7, 4, 6], [0, 5, 8]], [[1, 0, 3], [4, 2, 6], [7,
5, 8]], [[0, 2, 3], [1, 4, 6], [7, 5, 8]]]
Reached: [[[1, 2, 3], [0, 4, 6], [7, 5, 8]], [[1, 2, 3], [4, 0, 6], [7,
5, 8]], [[0, 2, 3], [1, 4, 6], [7, 5, 8]], [[1, 2, 3], [7, 4, 6], [0, 5
, 8]], [[1, 2, 3], [4, 6, 0], [7, 5, 8]], [[1, 0, 3], [4, 2, 6], [7, 5,
8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]]]
Frontier: [[[1, 2, 3], [4, 5, 6], [7, 8, 0]], [[1, 2, 3], [4, 6, 0], [7
, 5, 8]], [[1, 2, 3], [7, 4, 6], [0, 5, 8]], [[1, 2, 3], [4, 5, 6], [0,
7, 8]], [[1, 0, 3], [4, 2, 6], [7, 5, 8]], [[0, 2, 3], [1, 4, 6], [7, 5
, 8]]]
Reached: [[[1, 2, 3], [0, 4, 6], [7, 5, 8]], [[1, 2, 3], [4, 0, 6], [7,
5, 8]], [[0, 2, 3], [1, 4, 6], [7, 5, 8]], [[1, 2, 3], [7, 4, 6], [0, 5
, 8]], [[1, 2, 3], [4, 6, 0], [7, 5, 8]], [[1, 0, 3], [4, 2, 6], [7, 5,
8]], [[1, 2, 3], [4, 5, 6], [7, 0, 8]], [[1, 2, 3], [4, 5, 6], [7, 8, 0
]], [[1, 2, 3], [4, 5, 6], [0, 7, 8]]]
Goal Reached
[[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]]
Path cost: 3
```

6. AIM : Implement 8-puzzle problemusing A* algorithm.

Source Code :

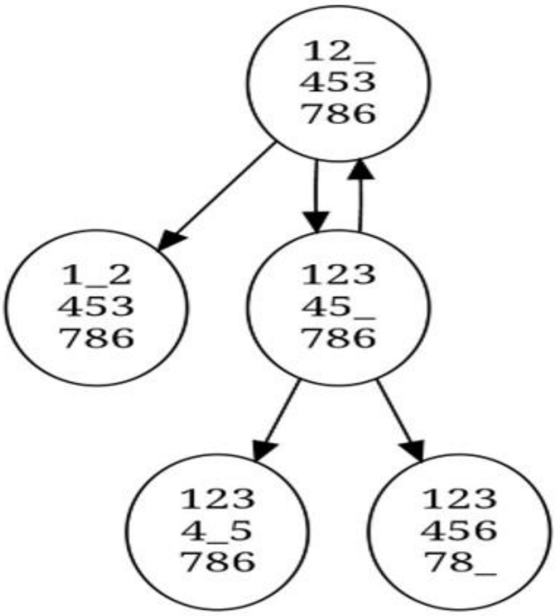
```
import copy
class Node:
    def __init__(self, puzzle, g):
        self.parent = None  # parent of the current Node
        self.puzzle = puzzle[:][:]  # the matrix representing the current state
        self.children = []  # a list to store the children nodes
        self.g = g  # the pathCost value
        self.x = []  # the position of the space
    def MD(self, pos1, pos2):
        return abs(pos1[0] - pos2[0]) + abs(pos1[1] - pos2[1])
    def h(self, goal):
        goalPositions = {}
        for i in range(3):
            for j in range(3):
                goalPositions[goal[i][j]] = [i, j]
        val = 0
        for i in range(3):
            for j in range(3):
                val += self.MD([i, j], goalPositions[self.puzzle[i][j]])
        return val
    def goalTest(self, goal):
        for i in range(3):
            for j in range(3):
                if self.puzzle[i][j] != goal[i][j]:
                    return False
        return True
    def addChild(self, puzzle):
        child = Node(puzzle, self.g+1)
        child.parent = self
```


g=0, h=3, f=g+h=3
g=1, h=4, f=5
g=1, h=2, f=3
g=1, h=4, f=5
1
g=2, h=1, f=3
g=2, h=3, f=5
g=2, h=3, f=5
2
g=3, h=2, f=5
3

```
            self.children.append(child)
    def moveRight(self):
        if self.x[1] != 2:
            pc = copy.deepcopy(self.puzzle)
            s = self.x
            pc[s[0]][s[1]], pc[s[0]][s[1]+1] = pc[s[0]][s[1]+1], pc[s[0]][s[1]]
            self.addChild(pc)
    def moveLeft(self):
        if self.x[1] != 0:
            pc = copy.deepcopy(self.puzzle)
            s = self.x
            pc[s[0]][s[1]], pc[s[0]][s[1]-1] = pc[s[0]][s[1]-1], pc[s[0]][s[1]]
            self.addChild(pc)
    def moveUp(self):
        if self.x[0] != 0:
            pc = copy.deepcopy(self.puzzle)
            s = self.x
            pc[s[0]][s[1]], pc[s[0]-1][s[1]] = pc[s[0]-1][s[1]], pc[s[0]][s[1]]
            self.addChild(pc)
    def moveDown(self):
        if self.x[0] != 2:
            pc = copy.deepcopy(self.puzzle)
            s = self.x
            pc[s[0]][s[1]], pc[s[0]+1][s[1]] = pc[s[0]+1][s[1]], pc[s[0]][s[1]]
            self.addChild(pc)
    def expandNode(self):
        for i in range(3):
            for j in range(3):
                if self.puzzle[i][j] == 0:
                    self.x = [i, j]
                    break
```

Output 2 :

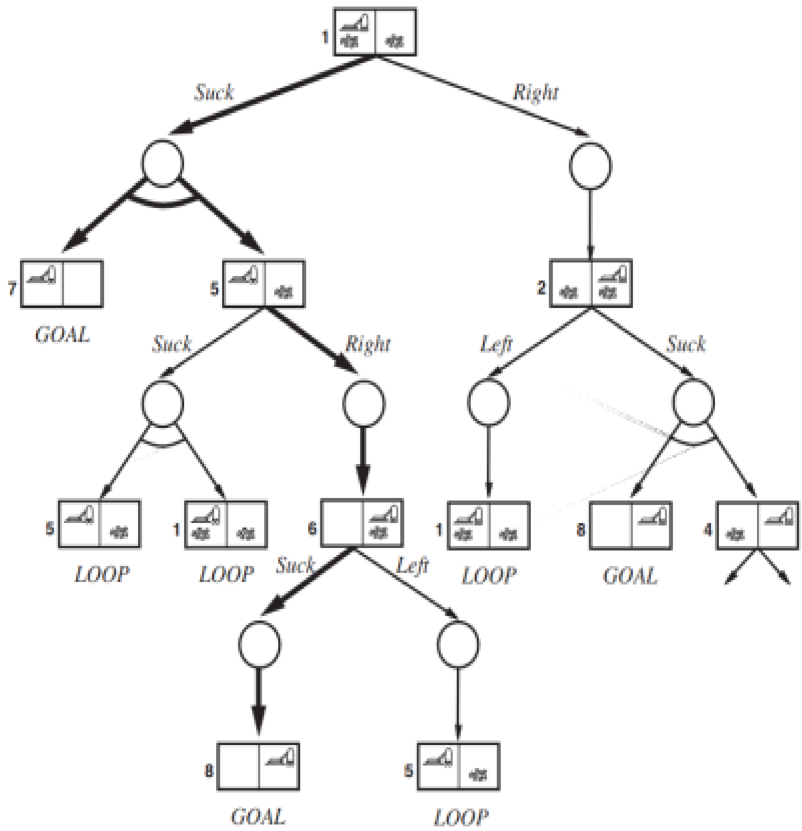
```
Enter the initial state:
1 2 0
4 5 3
7 8 6
Enter the goal state:
1 2 3
4 5 6
7 8 0
Frontier: [[[1, 2, 0], [4, 5, 3], [7, 8, 6]]]
Reached: [[[1, 2, 0], [4, 5, 3], [7, 8, 6]]]
Frontier: [[[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 0, 2], [4, 5, 3], [7
, 8, 6]]]
Reached: [[[1, 2, 0], [4, 5, 3], [7, 8, 6]], [[1, 0, 2], [4, 5, 3], [7,
8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]]]
Frontier: [[[1, 2, 3], [4, 5, 6], [7, 8, 0]], [[1, 2, 3], [4, 0, 5], [7
, 8, 6]], [[1, 0, 2], [4, 5, 3], [7, 8, 6]]]
Reached: [[[1, 2, 0], [4, 5, 3], [7, 8, 6]], [[1, 0, 2], [4, 5, 3], [7,
8, 6]], [[1, 2, 3], [4, 5, 0], [7, 8, 6]], [[1, 2, 3], [4, 0, 5], [7, 8
, 6]], [[1, 2, 3], [4, 5, 6], [7, 8, 0]]]
Goal Reached
[[1, 2, 0], [4, 5, 3], [7, 8, 6]]
[[1, 2, 3], [4, 5, 0], [7, 8, 6]]
[[1, 2, 3], [4, 5, 6], [7, 8, 0]]
Path cost: 2
```



```
            self.moveRight()
            self.moveLeft()
            self.moveUp()
            self.moveDown()
    def printPath(self):
        node = self
        path = []
        while node:
            path.append(node.puzzle)
            node = node.parent
        path = path[::-1]
        for p in path:
            print(p)
        print(f"Path cost: {len(path)-1}")
    def __repr__(self):
        return f"{self.puzzle}"
def AStar(initial, goal):
    node = Node(initial[:], 0)
    frontier = [node]
    reached = [node.puzzle]
    while frontier:
        frontier.sort(key=lambda x: x.h(goal)+x.g)
        print(f"Frontier: {frontier}")
        print(f"Reached: {reached}")
        node = frontier.pop(0)
        if node.goalTest(goal):
            print("Goal Reached")
            node.printPath()
            return
        node.expandNode()
        for child in node.children:
            if child.puzzle not in reached:
                frontier.append(child)
                reached.append(child.puzzle)
    return "Goal not found"
initial = []
print("Enter the initial state:")
for i in range(3):
    t = [int(x) for x in input().split()]
    initial.append(t)
goal = []
print("Enter the goal state:")
for i in range(3):
    t = [int(x) for x in input().split()]
    goal.append(t)
AStar(initial, goal)
```

Output 1 :

```
Left room (d/c): d
Right room (d/c): d
Vacuum (l/r): l
d, d, l
  suck
  c, d, l
    move right
    c, d, r
      suck
      c, c, r
  c, c, l
  move right
  d, d, r
    suck
    d, c, r
      move left
      d, c, l
        suck
        c, c, l
    c, c, r
```



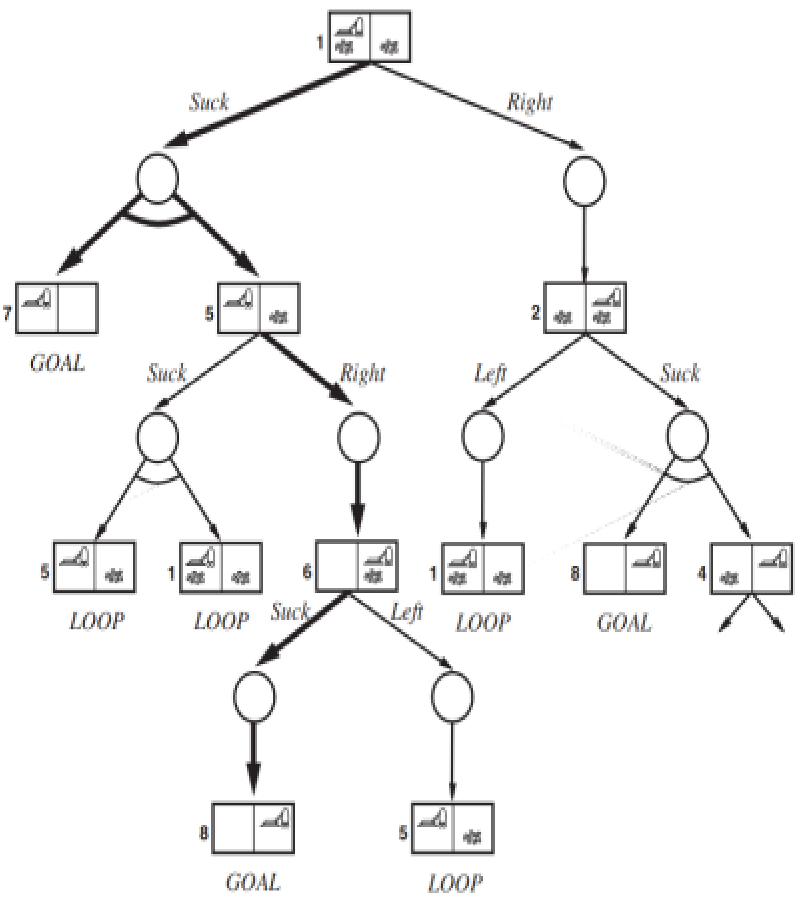
7. AIM : Implement AO* algorithm for General graph problem.

Source Code :

```
class State:
    def __init__(self, l='d', r='d', v='l'):
        self.l = l
        self.r = r
        self.v = v
        self.actions = {}
    def moveLeft(self):
        if self.v == 'r':
            self.actions['move left'] = [State(self.l, self.r, 'l')]
    def moveRight(self):
        if self.v == 'l':
            self.actions['move right'] = [State(self.l, self.r, 'r')]
    def suck(self):
        if self.v == 'r':
            if self.r == 'c':
                self.actions['suck'] = [self]
                self.actions['suck'].append(State(self.l, 'd', self.v))
            else:
                self.actions['suck'] = [State(self.l, 'c', self.v)]
            if self.l == 'd':
                self.actions['suck'].append(State('c', 'c', self.v))
        if self.v == 'l':
            if self.l == 'c':
                self.actions['suck'] = [self]
                self.actions['suck'].append(State('d', self.r, self.v))
            else:
                self.actions['suck'] = [State('c', self.r, self.v)]
            if self.r == 'd':
                self.actions['suck'].append(State('c', 'c', self.v))
```

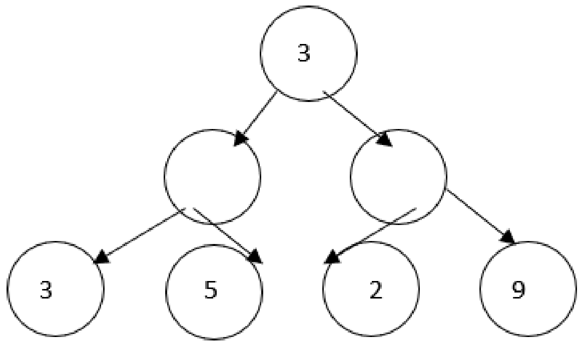
Output 2 :

```
Left room (d/c): d
Right room (d/c): c
Vacuum (l/r): l
d, c, l
  suck
  c, d, l
    move right
    c, d, r
      suck
      c, c, r
  c, c, l
  move right
  d, d, r
    suck
    d, c, r
      move left
      d, c, l
        suck
        c, c, l
    c, c, r
```



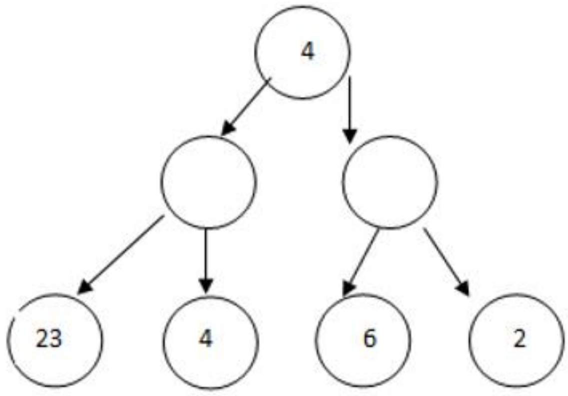
```
    def explore(self):
        self.suck()
        self.moveLeft()
        self.moveRight()
    def goalTest(self):
        return self.r == 'c' and self.l == 'c'
    def __eq__(self, other):
        return self.l == other.l and self.r == other.r and self.v == other.v
    def __repr__(self):
        return f"{self.l}, {self.r}, {self.v}"
def and_or_search(state):
    def or_search(state, path):
        possible = []
        if state.goalTest():
            return []
        if state in path:
            return None
        state.explore()
        for action in state.actions.keys():
            plan = and_search(state.actions[action], path + [state])
            if plan is not None:
                possible.append([action, plan])
        return possible
    def and_search(states, path):
        plan = {}
        for s in states:
            plan[str(s)] = or_search(s, path)
            if plan[str(s)] is None:
                return None
        return plan
    return or_search(state, [])
def visualize(root, indent=0):
    for v in root:
        for i in v:
            if isinstance(i, str):
                print(f"{' ' * indent}{i}")
            elif isinstance(i, dict):
                for k in i.keys():
                    print(f"{' ' * indent}{k}")
                    visualize(i[k], indent + 2)
l = input("Left room (d/c): ")
r = input("Right room (d/c): ")
v = input("Vacuum (l/r): ")
ans = and_or_search(State('d', 'd', 'l'))
print(State(l, r, v))
visualize(ans, indent=2)
```

Output 1 :




```
Enter scores:3 5 2 9
The optimal value is: 3
```

Output 2 :



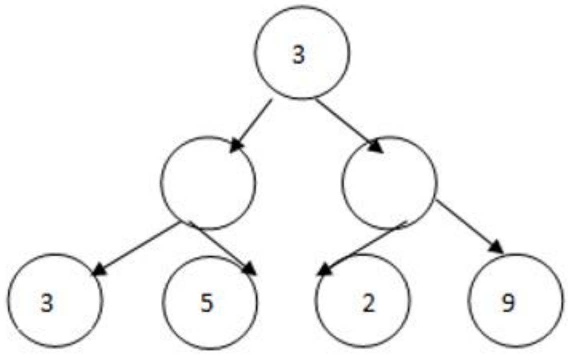

```
Enter scores:23 4 6 2
The optimal value is: 4
```

8a.AIM : Implement Game trees using MINIMAX algorithm

Source Code :

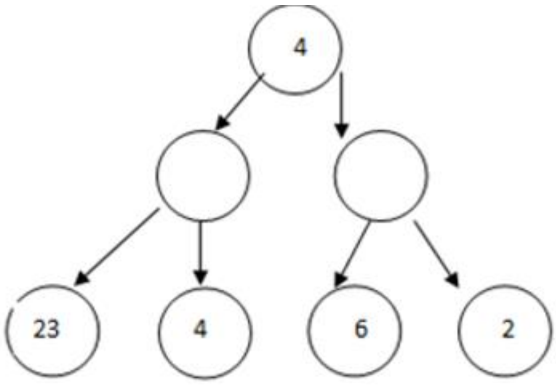
```
import math

def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth):
    if curDepth == targetDepth:
        return scores[nodeIndex]

    if maxTurn:
        return max(minimax(curDepth + 1, nodeIndex * 2, False, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1, False, scores, targetDepth))
    else:
        return min(minimax(curDepth + 1, nodeIndex * 2, True, scores, targetDepth),
                   minimax(curDepth + 1, nodeIndex * 2 + 1, True, scores, targetDepth))

scores = list(map(int,input("Enter scores:").split()))
treeDepth = int(math.log(len(scores), 2))

print("The optimal value is:", minimax(0, 0, True, scores, treeDepth))
```

Output 1 :




```
Enter scores:3 5 2 9
The optimal value is: 12
```

Output 2 :




```
Enter scores: 23 4 6 2
The optimal value is: 6
```

8b.AIM : Implement Game trees using Alpha-Beta pruning

Source Code :

```
MAX, MIN = 1000, -1000
def minimax(depth, nodeIndex, maximizingPlayer, values, alpha, beta):
    if depth == 3:
        return values[nodeIndex]
    if maximizingPlayer:
        best = MIN
        for i in range(2):
            val = minimax(depth + 1, nodeIndex * 2 + i, False, values, alpha, beta)
            best = max(best, val)
            alpha = max(alpha, best)
            if beta <= alpha:
                break
        return best
    else:
        best = MAX
        for i in range(2):
            val = minimax(depth + 1, nodeIndex * 2 + i, True, values, alpha, beta)
            best = min(best, val)
            beta = min(beta, best)
            if beta <= alpha:
                break
        return best
values = list(map(int,input("Enter scores:").split()))
print("The optimal value is:", minimax(0, 0, True, values, MIN, MAX))
```

Output 1 :

```
Enter words: SEND MORE MONEY
The solution is: {'M': 1, 'D': 7, 'E': 5, 'N': 6, 'O': 0, 'R': 8, 'S':
9, 'Y': 2}
```

Output 2 :

```
Enter words: TWO TWO FOUR
The solution is: {'F': 1, 'O': 4, 'R': 8, 'T': 7, 'U': 6, 'W': 3}
```

9. AIM : Implement Crypt arithmetic problems.

Source Code :

```
def solve_cryptarithmetic(puzzle):
    letters = sorted(list(set(char for word in puzzle for char in word) - {puzzle[2][0]}))
    letter_to_digit = {puzzle[2][0]: 1}
    def backtrack(index):
        if index == len(letters):
            if is_valid():
                return True
            return False
        for digit in set(range(10)) - {1}:
            if digit not in letter_to_digit.values():
                letter_to_digit[letters[index]] = digit
                if backtrack(index + 1):
                    return True
                del letter_to_digit[letters[index]]
        return False
    def is_valid():
        numeric_puzzle = [int(''.join([str(letter_to_digit[char]) for char in word])) for word in puzzle[:-1]]
        return sum(numeric_puzzle) == int(''.join([str(letter_to_digit[char]) for char in puzzle[-1]]))
    if backtrack(0):
        return letter_to_digit
    else:
        return None
puzzle_example = input("Enter words: ").split()
solution = solve_cryptarithmetic(puzzle_example)
if solution:
    print("The solution is:", solution)
```