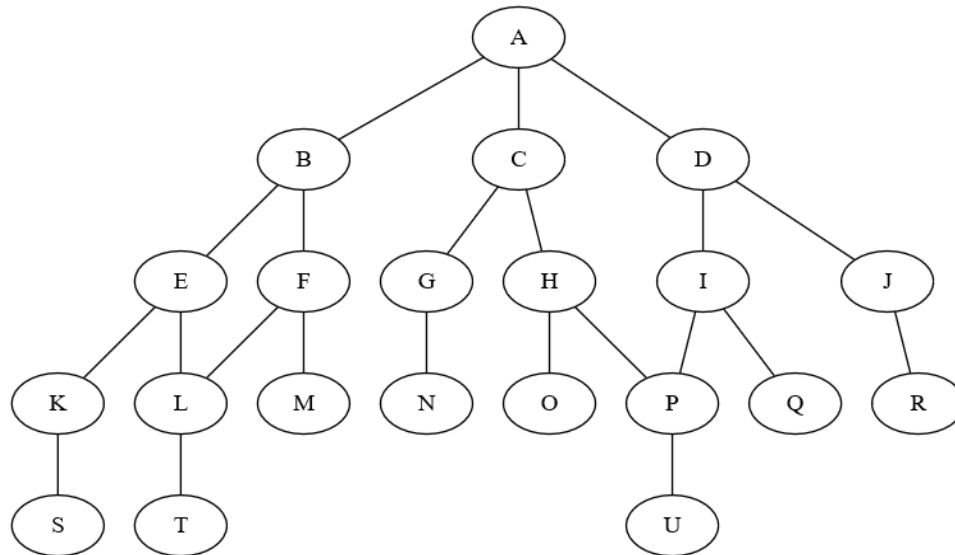


1. Implement Exhaustive search techniques using

a.BFS

Program:

```
n=int(input("Enter number of nodes in graph:")) #READING INPUT
for i in range(n):
    print(chr(65+i),end=' ')
print("are the nodes")
d={}
for i in range(n):
    print("Enter the child nodes of",chr(i+65),end="")
    c=list(map(str,input(":").split()))
    if c[0]!='0':
        d[chr(i+65)]=c
    else:
        d[chr(i+65)]=[]
source=input("Enter source node:")
key=input("Enter destination node:")
#INITIALISING OPEN AND CLOSED LISTS
open_list=[source]
closed_list=[]
print("Open List--Closed List")
print(" ".join(open_list)," ".join(closed_list),sep='--')
while open_list:
    source=open_list.pop(0)
    closed_list.append(source)
    if source==key:
        print('Required node is found',key)
        break
    for i in d[source]:
        if i not in closed_list and i not in open_list:
            open_list.append(i)
#PRINTING OPEN AND CLOSED LIST FOR EACH ITERATION.
print(" ".join(open_list)," ".join(closed_list),sep='--')
```

OUTPUT:**I.****GRAPH:**

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

Open List--Closed List

A--

B--A

BC--A

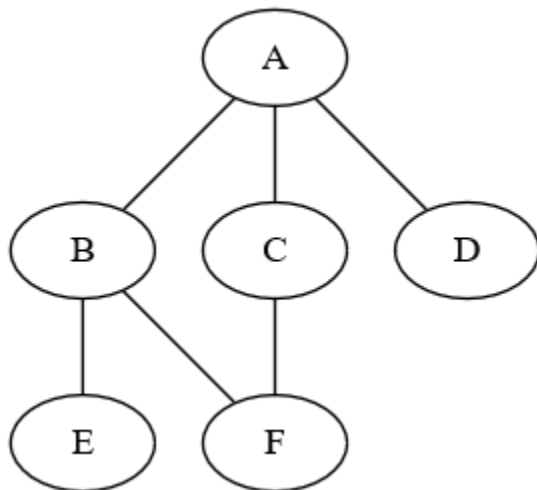
```

BCD--A
CDE--AB
CDEF--AB
DEFG--ABC
DEFGH--ABC
EFGHI--ABCD
EFGHIJ--ABCD
FGHIJK--ABCDE
FGHIJKL--ABCDE
GHIJKL--ABCDEF
GHIJKLM--ABCDEF
HIJKLMN--ABCDEFG
IJKLMNO--ABCDEFGH
IJKLMNOP--ABCDEFGH
JKLMNOP--ABCDEFGHI
JKLMNOPQ--ABCDEFGHI
KLMNOPQR--ABCDEFGHIJ
LMNOPQRS--ABCDEFGHIJK
MNOPQRST--ABCDEFGHIJKL
QRSTU--ABCDEFGHIJKLMNOP
Required node is found U

```

II.

GRAPH:



```

Enter number of nodes in graph:6
A B C D E F 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:F
Enter the child nodes of D:0
Enter the child nodes of E:0
Enter the child nodes of F:0
Enter source node:A
Enter destination node:E

```

Open List--Closed List

A--

B--A

BC--A

BCD--A

CDE--AB

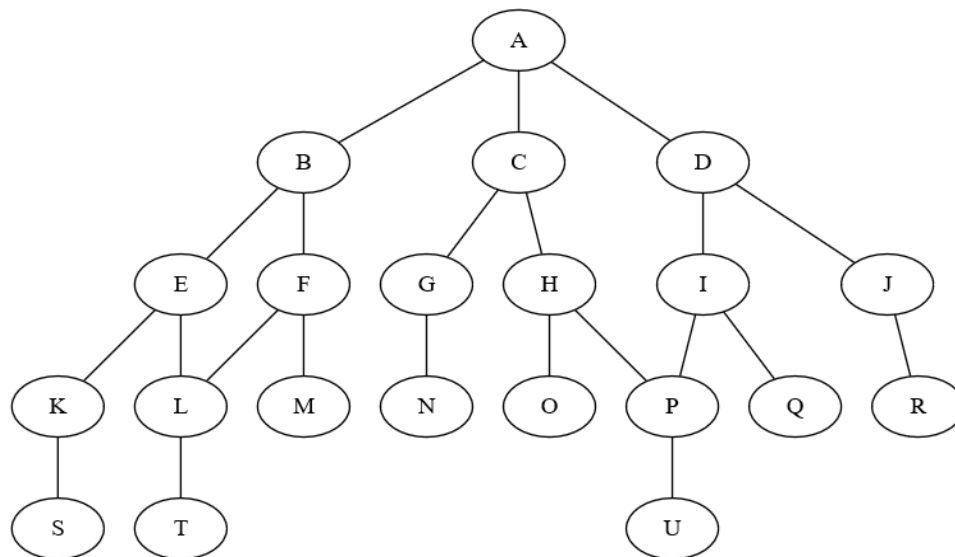
CDEF--AB

DEF--ABC

Required node is found E

b.DFS**Program:**

```
n=int(input("Enter number of nodes in graph:")) # READING INPUT
for i in range(n):
    print(chr(65+i),end=' ')
print("are the nodes")
d={}
for i in range(n):
    print("Enter the child nodes of",chr(i+65),end="")
    c=list(map(str,input(":").split()))
    if c[0]!='0':
        d[chr(i+65)]=c
    else:
        d[chr(i+65)]=[]
source=input("Enter source node:")
key=input("Enter destination node:")
#INITIALISING OPEN AND CLOSED LISTS.
open_list=[source]
closed_list=[]
print("Open List--Closed List")
print(" ".join(open_list)," ".join(closed_list),sep='--')
while open_list:
    source=open_list.pop(0)
    closed_list.append(source)
    if source==key:
        print('Required node is found',key)
        break
    for i in d[source]:
        if i not in closed_list and i not in open_list:
            open_list.insert(0,i)
#PRINTING OPEN AND CLOSED LIST FOR EACH ITERATION.
print(" ".join(open_list)," ".join(closed_list),sep='--')
```

OUTPUT:**I.****GRAPH:**

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

Open List--Closed List

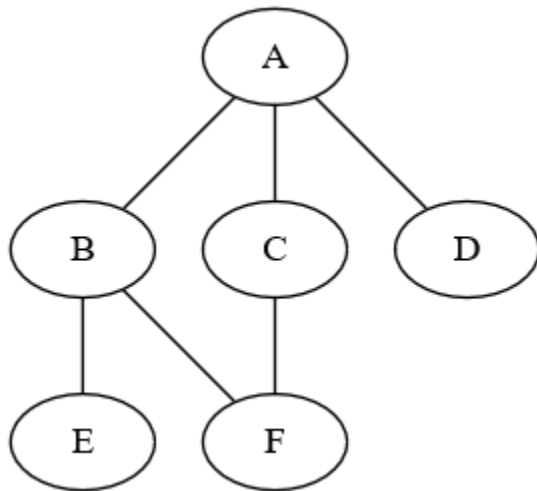
A--

B--A

CB--A
 DCB--A
 ICB--AD
 JICB--AD
 RICB--ADJ
 PCB--ADJRI
 QPCB--ADJRI
 UCB--ADJRIQP
 Required node is found U

II.

GRAPH:



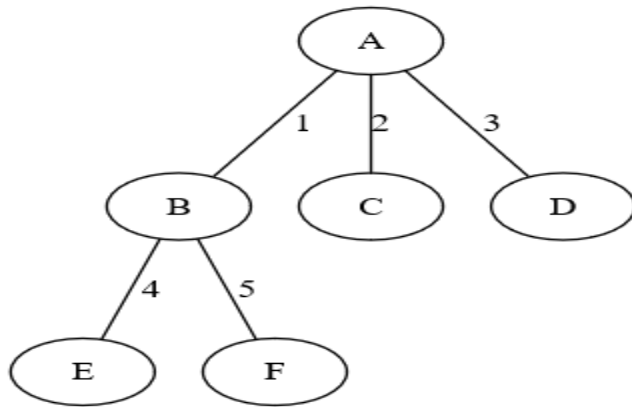
Enter number of nodes in graph:6
 A B C D E F are the nodes
 Enter the child nodes of A:B C D
 Enter the child nodes of B:E F
 Enter the child nodes of C:F
 Enter the child nodes of D:0
 Enter the child nodes of E:0
 Enter the child nodes of F:0
 Enter source node:A
 Enter destination node:E
 Open List--Closed List
 A--
 B--A
 CB--A
 DCB--A
 FB--ADC
 E--ADCFB
 E--ADCFB
 Required node is found E

c.Uniform Cost Search

Program:

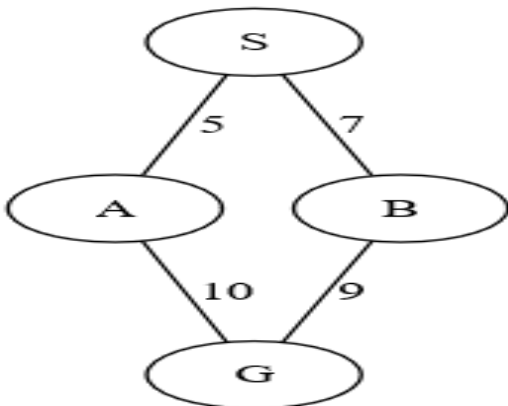
```
def UCS(graph, s, goal):
# FRONTIER IS THE DICTIONARY THAT STORES THE PATH AND ITS COST.
    frontier = {s: 0}
#EXPLORED IS A LIST THAT HAS ALL THE NODES THAT ARE ALREADY EXPLORED TO
#AVOID INFINITE LOOPING.
    explored = []
    while frontier:
        print(f"Open list: {frontier}")
        print(f"Closed list: {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"
# A FUNCTION TO READ INPUT GRAPH
def create_graph():
    num_nodes = int(input("Enter number of nodes in graph: "))
    graph = { }
    nodes = input("Enter the nodes separated by space: ").split()
    for node in nodes:
        children = input(f"Enter the child nodes of {node}: ").split()
        graph[node] = { }
        for child in children:
            if child != '0':
                cost = int(input(f"Enter the cost from {node} to {child}: "))
                graph[node][child] = cost
    return graph

graph = create_graph() #READING INPUT
s = input("Enter source node: ")
g = input("Enter goal node: ")
print(UCS(graph, s, g)) #FUNCTION CALL
```


OUTPUT:**I.****Graph:**

```

Enter number of nodes in graph: 6
Enter the nodes separated by space: A B C D E F
Enter the child nodes of A: B C D
Enter the child nodes of B: E F
Enter the child nodes of C: F
Enter the child nodes of D: 0
Enter the child nodes of E: 0
Enter the child nodes of F: 0
Enter source node: A
Enter goal node: E
Enter the cost from A to B: 1
Enter the cost from A to C: 2
Enter the cost from A to D: 3
Enter the cost from B to E: 4
Enter the cost from B to F: 5
Open list: {'A': 0}
Closed list: []
A : 0
Open list: {'B': 1, 'C': 2, 'D': 3}
Closed list: ['A']
B : 1
Open list: {'C': 2, 'D': 3, 'E': 5, 'F': 6}
Closed list: ['A', 'B']
E : 5
Goal reached with cost: 5
  
```

II.**Graph:**

```
Enter number of nodes in graph: 4
Enter the nodes separated by space: S A B G
Enter the child nodes of S: A B
Enter the child nodes of A: G
Enter the child nodes of B: G
Enter the child nodes of G: 0
Enter source node: S
Enter goal node: G
Enter the cost from S to A: 5
Enter the cost from S to B: 7
Enter the cost from A to G: 10
Enter the cost from B to G: 9
Open list: {'S': 0}
Closed list: []
S : 0
Open list: {'A': 5, 'B': 7}
Closed list: ['S']
A : 5
Open list: {'B': 7, 'G': 15}
Closed list: ['S', 'A']
B : 7
Open list: {'G': 15}
Closed list: ['S', 'A', 'B']
G : 15
Goal reached with cost: 15
```

d.Depth-First Iterative Deepening

Program:

```
def dfs(d, source, key):
```

```
#BASIC DFS FUNCTION
```

```
    open_list = [source]
```

```
    closed_list = []
```

```
    while open_list:
```

```
        source = open_list.pop(0)
```

```
        closed_list.append(source)
```

```
        if source == key:
```

```
            return [closed_list, open_list, 1]
```

```
        for i in d[source]:
```

```
            if i not in closed_list and i not in open_list:
```

```
                open_list.insert(0, i)
```

```
    return [closed_list, open_list, 0]
```

```
def depthxtree(graph, depth, source):
```

```
#THIS FUNCTION GENERATES TREES/GRAPHS TILL A CERTAIN LEVEL (HEIGHT)
```

```
    c = 1
```

```
    d = { }
```

```
    templis = []
```

```
    all_nodes = []
```

```
    all_nodes.append(source + str(c))
```

```
    while c <= depth:
```

```
        if c == 1:
```

```
            d[source] = []
```

```
            templis = graph[source]
```

```
            parent = source
```

```
        if c == 2:
```

```
            for i in templis:
```

```
                d[i] = []
```

```
                d[parent].append(i)
```

```
                all_nodes.append(i + str(c))
```

```
        else:
```

```
            for j in all_nodes:
```

```
                if j[-1] == str(c - 1):
```

```
                    templis = graph[j[:-1]]
```

```
                    for i in templis:
```

```
                        d[i] = []
```

```
                        d[j[:-1]].append(i)
```

```
                        all_nodes.append(i + str(c))
```

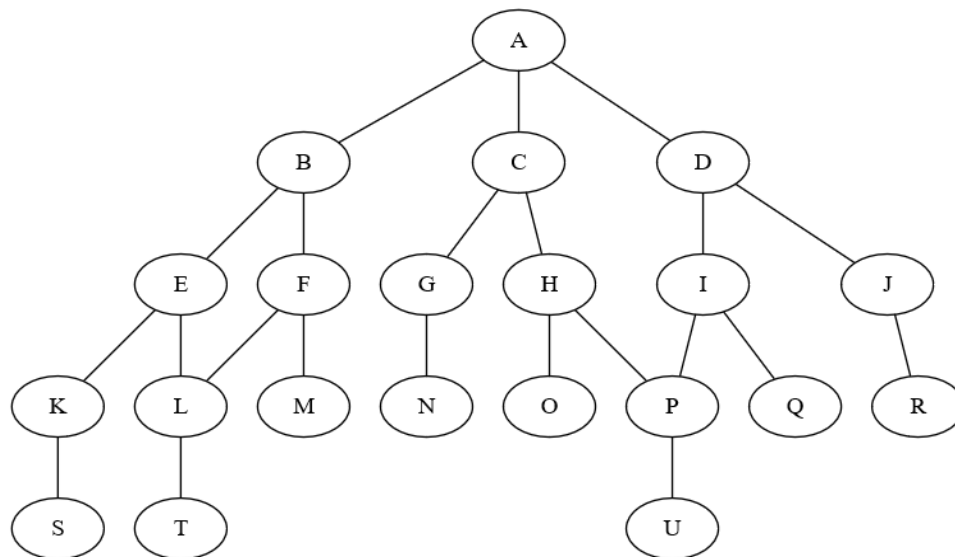
```
        c += 1
```

```
    return d
```

```
def iddfs(graph, source, key):
```

```
depth = 1
while 1:
    d = depthxtree(graph, depth, source)
    y = dfs(d, source, key)
    print("DEPTH=", depth)
    print("Open_list=", y[1])
    print("Closed_list=", y[0])
    depth += 1
    if len(d) == len(graph) or y[2] == 1:
        break
# READING INPUT
n = int(input("Enter number of nodes in graph: "))
for i in range(n):
    print(chr(65 + i), end=' ')
print("are the nodes")
graph = { }
for i in range(n):
    print("Enter the child nodes of", chr(65 + i), end="")
    c = list(map(str, input(": ").split()))
    if c[0] != '0':
        graph[chr(65 + i)] = c
    else:
        graph[chr(65 + i)] = []

source = input("Enter source node: ")
destination = input("Enter destination node: ")
iddfs(graph, source, destination) #FUNCUTION CALL
```

OUTPUT:**I.****GRAPH:**

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

DEPTH= 1

Open_list= []

Closed_list= ['A']

DEPTH= 2

Open_list= []

Closed_list= ['A', 'D', 'C', 'B']

DEPTH= 3

Open_list= []

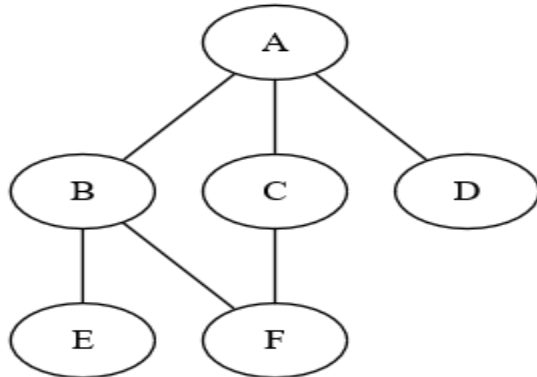
```

Closed_list= ['A', 'D', 'J', 'I', 'C', 'H', 'G', 'B', 'F', 'E']
DEPTH= 4
Open_list= []
Closed_list= ['A', 'D', 'J', 'R', 'I', 'Q', 'P', 'C', 'H', 'O', 'G',
'N', 'B', 'F', 'M', 'L', 'E', 'K']
DEPTH= 5
Open_list= ['C', 'B']
Closed_list= ['A', 'D', 'J', 'R', 'I', 'Q', 'P', 'U']

```

II.

GRAPH:



```

Enter number of nodes in graph: 6
A B C D E F 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: F
Enter the child nodes of D: 0
Enter the child nodes of E: 0
Enter the child nodes of F: 0
Enter source node: A
Enter destination node: F
DEPTH= 1
Open_list= ['B', 'C', 'D']
Closed_list= ['A']
DEPTH= 2
Open_list= ['E', 'F']
Closed_list= ['A', 'B', 'C', 'D']
DEPTH= 3
Open_list= []
Closed_list= ['A', 'B', 'C', 'D', 'E', 'F']

```

e.Bidirectional Search

Program:

```
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

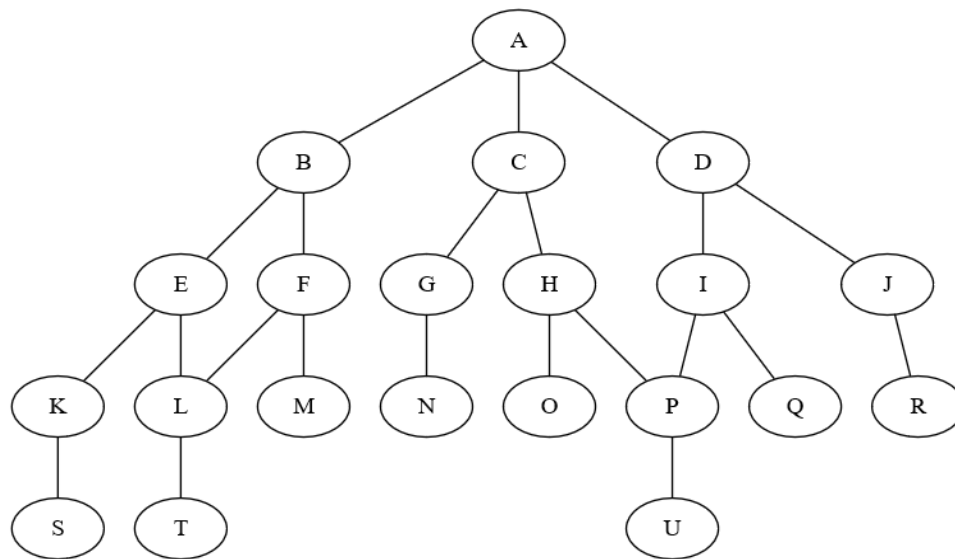
def isIntersecting(reachedF, reachedB):
    intersecting = set(reachedF).intersection(set(reachedB))
    return list(intersecting)[0] if intersecting else -1

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: "))
    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:**I.****GRAPH:**

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

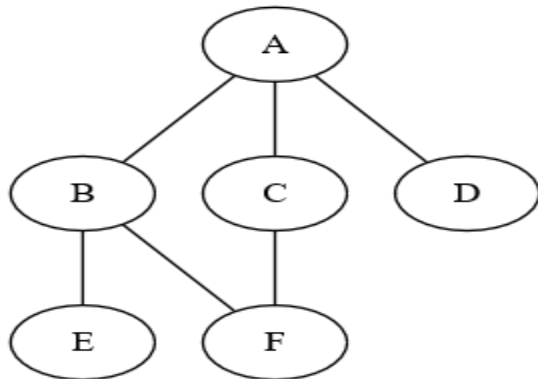
Enter source node: A

Enter destination node: U

From front:

```
    Frontier: ['A']
    Reached: ['A']
From back:
    Frontier: ['U']
    Reached: ['U']
Path found!
Path: ['A', 'P', 'U'] ['A']
```

II. GRAPH:



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) Implement water jug problem with Search tree generation using BFS

Program:

```
def fill(x,jug,i): # FILL FUNCTION RETURNS A STATE WITH MENTIONED FILLED JUG
    x=list(x)
    x[i]=jug
    return tuple(x)
def empty(x,i): # EMPTY FUNCTION RETURNS A STATE WITH MENTIONED JUG EMPTY
    x=list(x)
    x[i]=0
    return tuple(x)
def transfer(x,jug1,jug2,i): # TRANSFER FUNCTION RETURNS A STATE WITH POSSIBLE
TRANSFER FROM JUG i TO j.
    x=list(x)
    if i==1:
        available=jug2-x[1]
        if x[0]>available:
            x[0]-=available
            x[1]+=available
        else:
            x[1]+=x[0]
            x[0]=0
    elif i==2:
        available=jug1-x[0]
        if x[1]>available:
            x[1]-=available
            x[0]+=available
        else:
            x[0]+=x[1]
            x[1]=0
    return tuple(x)
def make_tree(d,jug1,jug2,vol): #GENERATING STATE SPACE TREE
    all_nodes=[(0,0)]
    flag=1
    while(flag):
        if flag==2:
            break
        flag=0
        source=all_nodes[0]
        for i in [1,2]:
            x=jug2
            if i==1:
                x=jug1
            f=fill(source,x,i-1)
            if f not in all_nodes:
                all_nodes.insert(0,f)
```

```

        d[source].append(f)
        d[f]=[]
        flag=1
        if f[0]==vol or f[1]==vol:
            break
    f=empty(source,i-1)
    if f not in all_nodes:
        all_nodes.insert(0,f)
        d[source].append(f)
        d[f]=[]
        flag=1
        if f[0]==vol or f[1]==vol:
            break
    f=transfer(source,jug1,jug2,i)
    if f not in all_nodes:
        all_nodes.insert(0,f)
        d[source].append(f)
        d[f]=[]
        flag=1
        if f[0]==vol or f[1]==vol:
            flag=2
            break

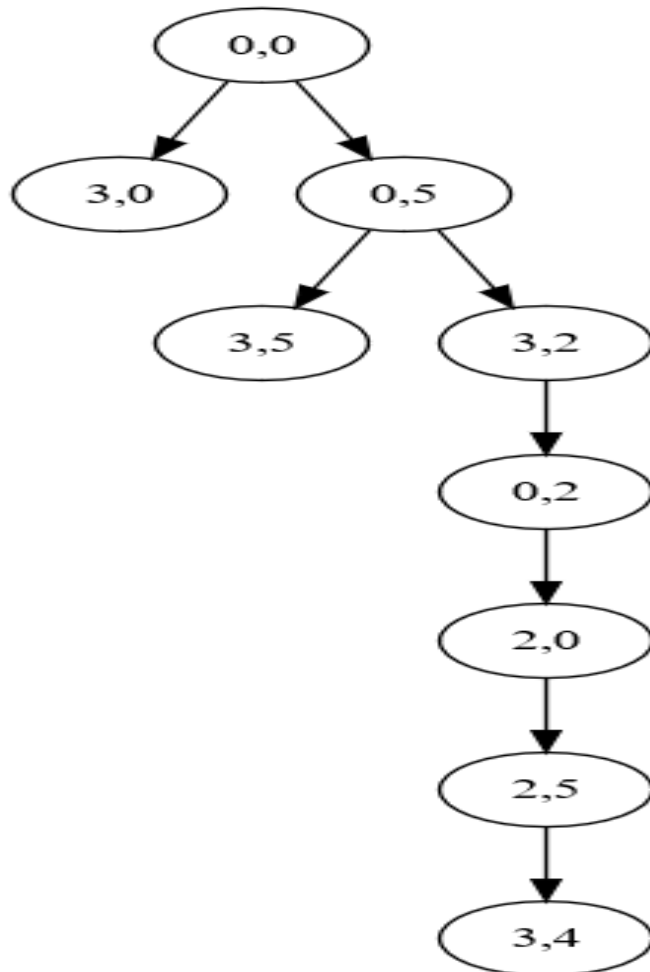
jug1,jug2=map(int,input('Enter capacities of jug1 and jug2:').split()) #INPUT READING
vol=int(input('Enter Volume to make:'))
d={}
d[(0,0)]=[]
make_tree(d,jug1,jug2,vol) #FUNCTION CALL TO GENERATE STATE SPACE TREE
final=[] # FINAL[-1] HOLDS THE FINAL REQUIRED STATE
for i,j in d.items():
    if j==[]:
        if i[0]!=vol and i[1]!=vol:
            pass
        else:
            final.append(i)
    else:
        final.append(i)
#BFS
open_list=[(0,0)]
closed_list=[]
print("Open List--Closed List")
print(" ".join(str(open_list)), " ".join(str(closed_list)), sep='--')
while open_list:
    source=open_list.pop(0)
    closed_list.append(source)
    if source==final[-1]:

```

```

break
for i in d[source]:
    if i not in closed_list and i not in open_list:
        open_list.append(i)
print(".join(str(open_list)), ".join(str(closed_list)), sep='--', end='\n\n')

```

OUTPUT:**I.****GRAPH:**

```

Enter capacities of jug1 and jug2:3 5
Enter Volume to make:4
Open List--Closed List
[(0, 0)]--[]
[(3, 0), (0, 5)]--[(0, 0)]

[(0, 5)]--[(0, 0), (3, 0)]

[(3, 5), (3, 2)]--[(0, 0), (3, 0), (0, 5)]

[(3, 2)]--[(0, 0), (3, 0), (0, 5), (3, 5)]

```

$[(0, 2)] \leftrightarrow [(0, 0), (3, 0), (0, 5), (3, 5), (3, 2)]$

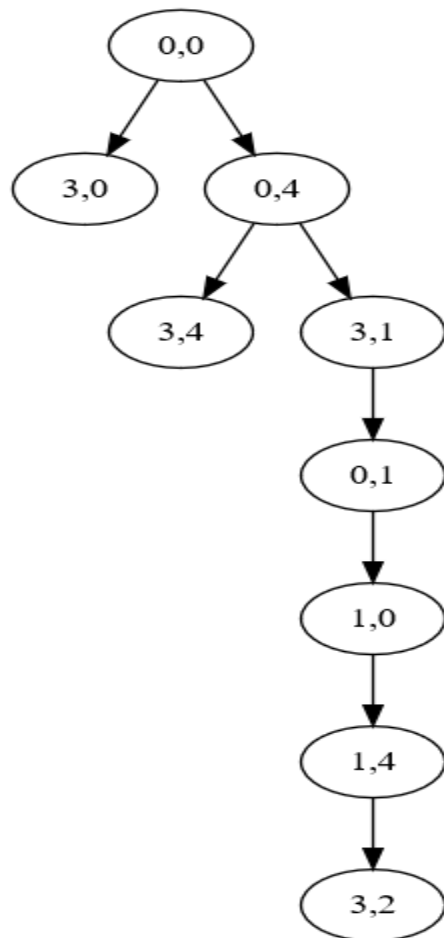
$[(2, 0)] \leftrightarrow [(0, 0), (3, 0), (0, 5), (3, 5), (3, 2), (0, 2)]$

$[(2, 5)] \leftrightarrow [(0, 0), (3, 0), (0, 5), (3, 5), (3, 2), (0, 2), (2, 0)]$

$[(3, 4)] \leftrightarrow [(0, 0), (3, 0), (0, 5), (3, 5), (3, 2), (0, 2), (2, 0), (2, 5)]$

II.

GRAPH:



Enter capacities of jug1 and jug2:3 4

Enter Volume to make:2

Open List--Closed List

$[(0, 0)] \leftrightarrow []$

$[(3, 0), (0, 4)] \leftrightarrow [(0, 0)]$

$[(0, 4)] \leftrightarrow [(0, 0), (3, 0)]$

$[(3, 4), (3, 1)] \leftrightarrow [(0, 0), (3, 0), (0, 4)]$

$[(3, 1)] \leftrightarrow [(0, 0), (3, 0), (0, 4), (3, 4)]$

$[(0, 1)] \dashv\vdash [(0, 0), (3, 0), (0, 4), (3, 4), (3, 1)]$

$[(1, 0)] \dashv\vdash [(0, 0), (3, 0), (0, 4), (3, 4), (3, 1), (0, 1)]$

$[(1, 4)] \dashv\vdash [(0, 0), (3, 0), (0, 4), (3, 4), (3, 1), (0, 1), (1, 0)]$

$[(3, 2)] \dashv\vdash [(0, 0), (3, 0), (0, 4), (3, 4), (3, 1), (0, 1), (1, 0), (1, 4)]$

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

Program:

```
def fill(x,jug,i):
    x=list(x)
    x[i]=jug
    return tuple(x)
def empty(x,i):
    x=list(x)
    x[i]=0
    return tuple(x)
def transfer(x,jug1,jug2,i):
    x=list(x)
    if i==1:
        available=jug2-x[1]
        if x[0]>available:
            x[0]-=available
            x[1]+=available
        else:
            x[1]+=x[0]
            x[0]=0
    elif i==2:
        available=jug1-x[0]
        if x[1]>available:
            x[1]-=available
            x[0]+=available
        else:
            x[0]+=x[1]
            x[1]=0
    return tuple(x)
def make_tree(d,jug1,jug2,vol):
    all_nodes=[(0,0)]
    flag=1
    while(flag):
        if flag==2:
            break
        flag=0
        source=all_nodes[0]
        for i in [1,2]:
            x=jug2
            if i==1:
                x=jug1
            f=fill(source,x,i-1)
            if f not in all_nodes:
                all_nodes.insert(0,f)
                d[source].append(f)
```

```

        d[f]=[]
        flag=1
        if f[0]==vol or f[1]==vol:
            break
        f=empty(source,i-1)
        if f not in all_nodes:
            all_nodes.insert(0,f)
            d[source].append(f)
            d[f]=[]
            flag=1
            if f[0]==vol or f[1]==vol:
                break
        f=transfer(source,jug1,jug2,i)
        if f not in all_nodes:
            all_nodes.insert(0,f)
            d[source].append(f)
            d[f]=[]
            flag=1
            if f[0]==vol or f[1]==vol:
                flag=2
                break

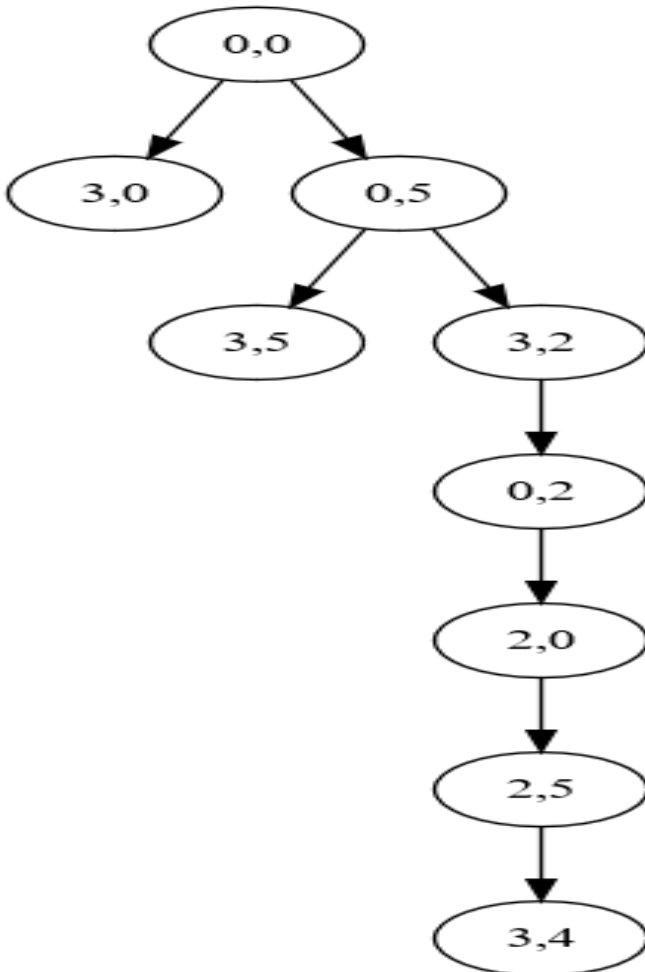
jug1,jug2=map(int,input('Enter capacities of jug1 and jug2:').split())
vol=int(input('Enter Volume to make:'))
d={}
d[(0,0)]=[]
make_tree(d,jug1,jug2,vol)
final=[]
for i,j in d.items():
    if j==[]:
        if i[0]!=vol and i[1]!=vol:
            pass
        else:
            final.append(i)
    else:
        final.append(i)
#DFS
open_list=[(0,0)]
closed_list=[]
print("Open List--Closed List")
print(" ".join(str(open_list)), " ".join(str(closed_list)), sep='--')
while open_list:
    source=open_list.pop(0)
    closed_list.append(source)
    if source==final[-1]:
        break

```

```

for i in d[source]:
    if i not in closed_list and i not in open_list:
        open_list.insert(0,i)
print(".join(str(open_list)), ".join(str(closed_list)), sep='--', end='\n\n')

```

OUTPUT:**I.****GRAPH:**

```

Enter capacities of jug1 and jug2:3 5
Enter Volume to make:4
Open List--Closed List
[(0, 0)]--[]
[(0, 5), (3, 0)]--[(0, 0)]

[(3, 2), (3, 5), (3, 0)]--[(0, 0), (0, 5)]
[(0, 2), (3, 5), (3, 0)]--[(0, 0), (0, 5), (3, 2)]
[(2, 0), (3, 5), (3, 0)]--[(0, 0), (0, 5), (3, 2), (0, 2)]

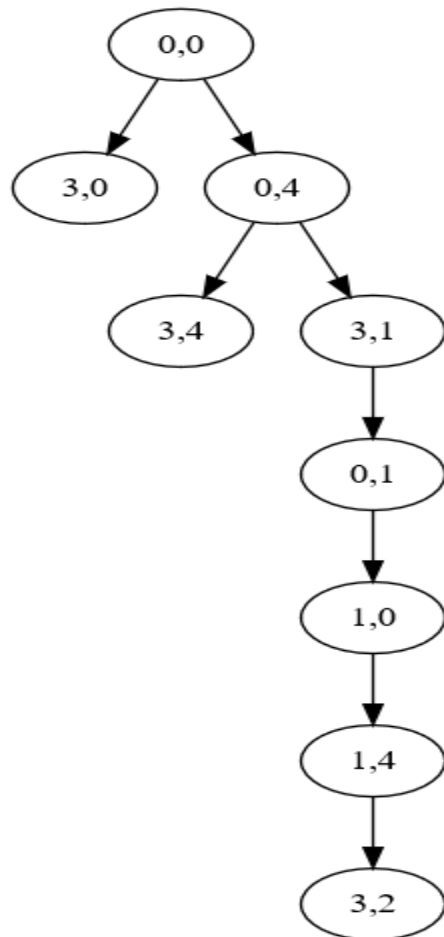
```

$[(2, 5), (3, 5), (3, 0)] \leftrightarrow [(0, 0), (0, 5), (3, 2), (0, 2), (2, 0)]$

$[(3, 4), (3, 5), (3, 0)] \leftrightarrow [(0, 0), (0, 5), (3, 2), (0, 2), (2, 0), (2, 5)]$

II.

GRAPH:



Enter capacities of jug1 and jug2:3 4

Enter Volume to make:2

Open List--Closed List

$[(0, 0)] \leftrightarrow []$

$[(0, 4), (3, 0)] \leftrightarrow [(0, 0)]$

$[(3, 1), (3, 4), (3, 0)] \leftrightarrow [(0, 0), (0, 4)]$

$[(0, 1), (3, 4), (3, 0)] \leftrightarrow [(0, 0), (0, 4), (3, 1)]$

$[(1, 0), (3, 4), (3, 0)] \leftrightarrow [(0, 0), (0, 4), (3, 1), (0, 1)]$

$[(1, 4), (3, 4), (3, 0)] \leftrightarrow [(0, 0), (0, 4), (3, 1), (0, 1), (1, 0)]$

$[(3, 2), (3, 4), (3, 0)] \dashv\dashv [(0, 0), (0, 4), (3, 1), (0, 1), (1, 0), (1, 4)]$

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

Program:

def travel(state,s): # TRAVEL FUNCTION RETURNS THE POSSIBLE STATE IF TRAVEL ACTION IS PERFORMED ELSE RETURNS A NULL STRING .

```

state=list(state)
state[0]=list(state[0])
state[1]=list(state[1])
if state[0][-1]==1:
    for i in [0,1]:
        state[0][i]-=s[i]
        state[1][i]+=s[i]
    state[0][-1]=0
    state[1][-1]=1
    state[0]=tuple(state[0])
    state[1]=tuple(state[1])
    state=tuple(state)
    return state
elif state[1][-1]==1:
    for i in [0,1]:
        state[1][i]-=s[i]
        state[0][i]+=s[i]
    state[1][-1]=0
    state[0][-1]=1
    state[0]=tuple(state[0])
    state[1]=tuple(state[1])
    state=tuple(state)
    return state
else:
    return ""

```

def make_tree(start,end): # GENERATES STATE SPACE TREE

```

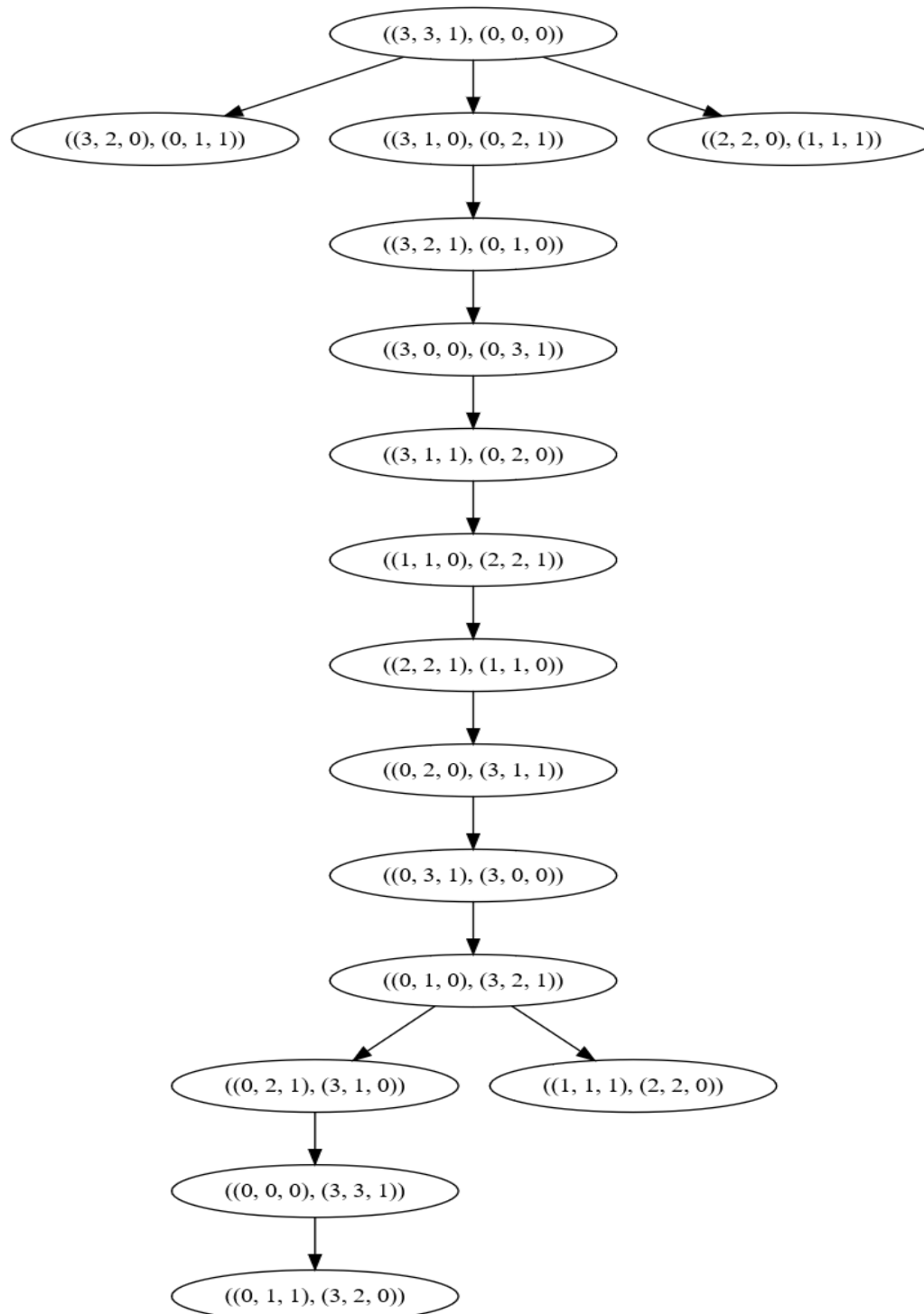
d={ }
d[start]=[]
all_nodes=[start]
c=1
h=-1
source=start
while(h<c and source!=end):
    h+=1
    source=all_nodes[h]
    for i in [0,1,2]: #EITHER 0,1 OR 2 MISSIONARIES
        for j in [0,1,2]: # EITHER 0,1 OR 2 CANNIBALS
            if i+j<=2 and i+j>0: # ATLEAST 1 ON BOAT OR ATMOST 2
                state=travel(source,[i,j])

```

```

        if state!=" " and state not in all_nodes: # GENERATING ONLY NEW STATES.
            if (state[0][0]>=state[0][1] or state[0][0]==0) and (state[1][0]>=state[1][1] or
state[1][0]==0):
                if state[0][0]>=0 and state[0][1]>=0 and state[1][0]>=0 and state[1][1]>=0:
                    all_nodes.append(state)
                    d[source].append(state)
                    d[state]=[]
                    if state==end:
                        break
            c=len(all_nodes)
        return d
m,n=map(int,input("Enter the number of missionaries and cannibals:").split()) #INPUT READING
start = ((m,n,1),(0,0,0))
end = ((0,0,0),(m,n,1))
key=end
state=start
d=make_tree(start,end) #FUNCTION CALL
#BFS
open_list=[start]
closed_list=[]
print("Open List--Closed List")
print(".join(str(open_list)), ".join(str(closed_list)),sep='--')
while open_list:
    source=open_list.pop(0)
    closed_list.append(source)
    if source==key:
        print('Required node is found',key)
        break
    for i in d[source]:
        if i not in closed_list and i not in open_list:
            open_list.append(i)
print(".join(str(open_list)), ".join(str(closed_list)),sep='--',end='\n\n')

```

OUTPUT:**I.****GRAPH:**

Enter the number of missionaries and cannibals:3 3

Open List--Closed List

[((3, 3, 1), (0, 0, 0))]-[]

$[((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1))]--[((3, 3, 1), (0, 0, 0))]$

$[((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1))]$

$[((2, 2, 0), (1, 1, 1)), ((3, 2, 1), (0, 1, 0))]--[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1))]$

$[((3, 2, 1), (0, 1, 0))]--[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1))]$

$[((3, 0, 0), (0, 3, 1))]--[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1)), ((3, 2, 1), (0, 1, 0))]$

$[((3, 1, 1), (0, 2, 0))]--[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1))]$

$[((1, 1, 0), (2, 2, 1))]--[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0))]$

$[((2, 2, 1), (1, 1, 0))]--[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1))]$

$[((0, 2, 0), (3, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0))]$

$[((0, 3, 1), (3, 0, 0))]--[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0)), ((0, 2, 0), (3, 1, 1))]$

$[((0, 1, 0), (3, 2, 1))]--[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0)), ((0, 2, 0), (3, 1, 1)), ((0, 3, 1), (3, 0, 0))]$

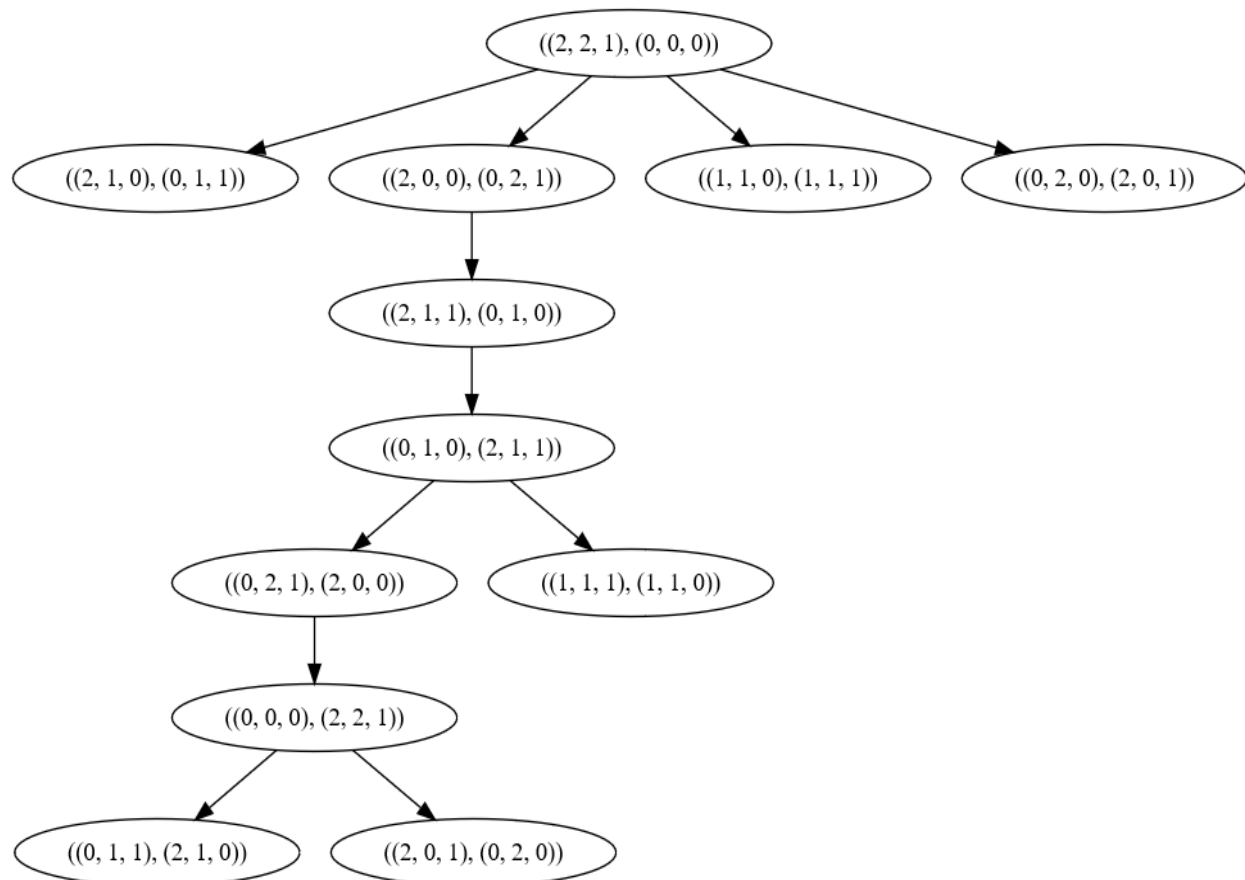
$[((0, 2, 1), (3, 1, 0)), ((1, 1, 1), (2, 2, 0))]--[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0)), ((0, 2, 0), (3, 1, 1)), ((0, 3, 1), (3, 0, 0)), ((0, 1, 0), (3, 2, 1))]$

```
[((1, 1, 1), (2, 2, 0)), ((0, 0, 0), (3, 3, 1))]-[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0)), ((0, 2, 0), (3, 1, 1)), ((0, 3, 1), (3, 0, 0)), ((0, 1, 0), (3, 2, 1)), ((0, 2, 1), (3, 1, 0))]
```

```
[((0, 0, 0), (3, 3, 1))]-[((3, 3, 1), (0, 0, 0)), ((3, 2, 0), (0, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((2, 2, 0), (1, 1, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0)), ((0, 2, 0), (3, 1, 1)), ((0, 3, 1), (3, 0, 0)), ((0, 1, 0), (3, 2, 1)), ((0, 2, 1), (3, 1, 0)), ((1, 1, 1), (2, 2, 0))]
```

Required node is found ((0, 0, 0), (3, 3, 1))

II. GRAPH:



Enter the number of missionaries and cannibals:2 2

Open List--Closed List

```
[((2, 2, 1), (0, 0, 0))]-[[]]
[((2, 1, 0), (0, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((1, 1, 0), (1, 1, 1)), ((0, 2, 0), (2, 0, 1))]-[((2, 2, 1), (0, 0, 0))]
```

```
[((2, 0, 0), (0, 2, 1)), ((1, 1, 0), (1, 1, 1)), ((0, 2, 0), (2, 0, 1))]-[((2, 2, 1), (0, 0, 0)), ((2, 1, 0), (0, 1, 1))]
```

```
[((1, 1, 0), (1, 1, 1)), ((0, 2, 0), (2, 0, 1)), ((2, 1, 1), (0, 1, 0))]-[((2, 2, 1), (0, 0, 0)), ((2, 1, 0), (0, 1, 1)), ((2, 0, 0), (0, 2, 1))]
```

```
[((0, 2, 0), (2, 0, 1)), ((2, 1, 1), (0, 1, 0))]-[((2, 2, 1), (0, 0, 0)), ((2, 1, 0), (0, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((1, 1, 0), (1, 1, 1))]
```

```
[((2, 1, 1), (0, 1, 0))]-[((2, 2, 1), (0, 0, 0)), ((2, 1, 0), (0, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((1, 1, 0), (1, 1, 1)), ((0, 2, 0), (2, 0, 1))]
```

```
[((0, 1, 0), (2, 1, 1))]-[((2, 2, 1), (0, 0, 0)), ((2, 1, 0), (0, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((1, 1, 0), (1, 1, 1)), ((0, 2, 0), (2, 0, 1)), ((2, 1, 1), (0, 1, 0))]
```

```
[((0, 2, 1), (2, 0, 0)), ((1, 1, 1), (1, 1, 0))]-[((2, 2, 1), (0, 0, 0)), ((2, 1, 0), (0, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((1, 1, 0), (1, 1, 1)), ((0, 2, 0), (2, 0, 1)), ((2, 1, 1), (0, 1, 0)), ((0, 1, 0), (2, 1, 1))]
```

```
[((1, 1, 1), (1, 1, 0)), ((0, 0, 0), (2, 2, 1))]-[((2, 2, 1), (0, 0, 0)), ((2, 1, 0), (0, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((1, 1, 0), (1, 1, 1)), ((0, 2, 0), (2, 0, 1)), ((2, 1, 1), (0, 1, 0)), ((0, 1, 0), (2, 1, 1)), ((0, 2, 1), (2, 0, 0))]
```

```
[((0, 0, 0), (2, 2, 1))]-[((2, 2, 1), (0, 0, 0)), ((2, 1, 0), (0, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((1, 1, 0), (1, 1, 1)), ((0, 2, 0), (2, 0, 1)), ((2, 1, 1), (0, 1, 0)), ((0, 1, 0), (2, 1, 1)), ((0, 2, 1), (2, 0, 0)), ((1, 1, 1), (1, 1, 0))]
```

Required node is found ((0, 0, 0), (2, 2, 1))

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

Program:

```
def travel(state,s):
    state=list(state)
    state[0]=list(state[0])
    state[1]=list(state[1])
    if state[0][-1]==1:
        for i in [0,1]:
            state[0][i]-=s[i]
            state[1][i]+=s[i]
        state[0][-1]=0
        state[1][-1]=1

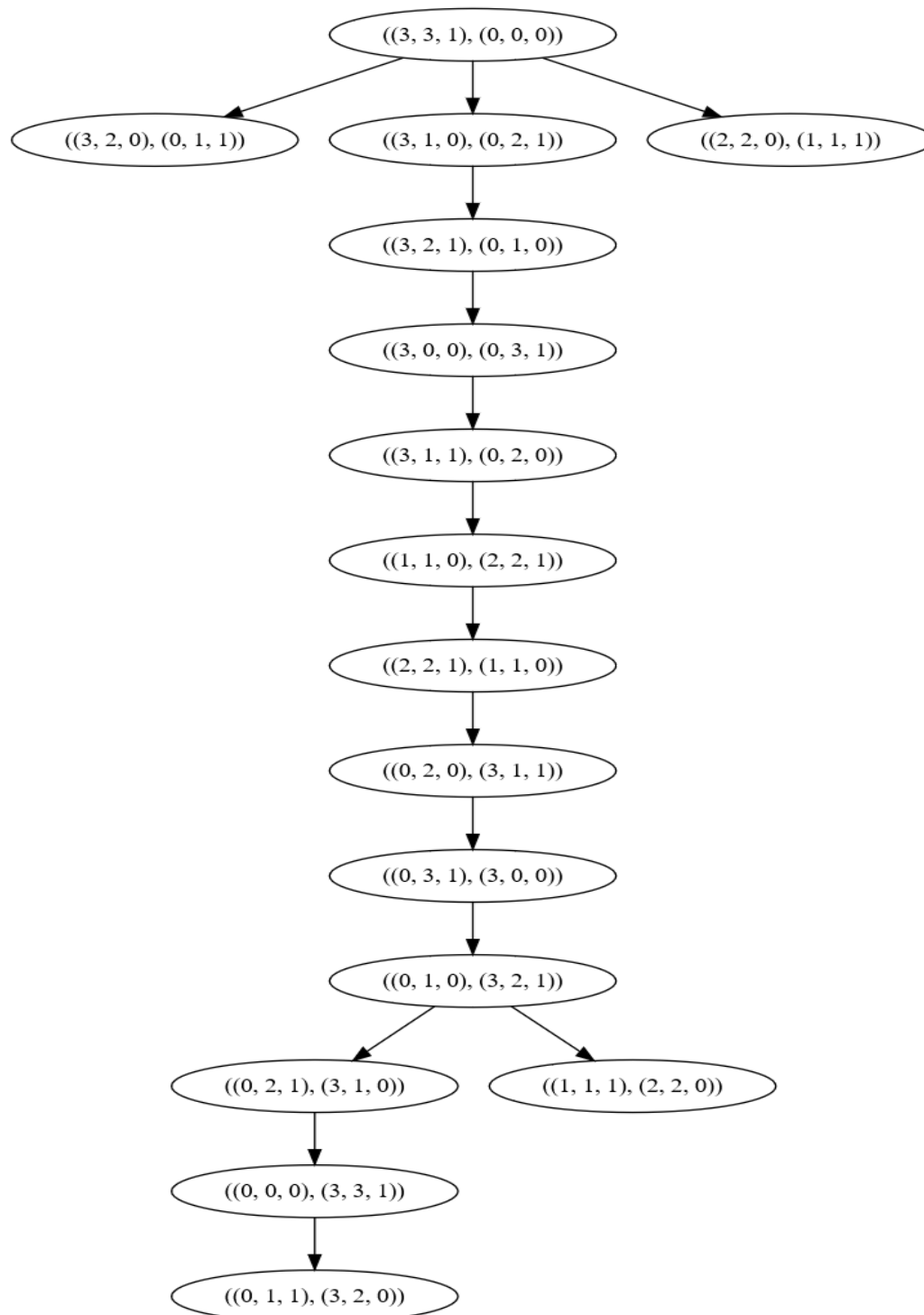
        state[0]=tuple(state[0])
        state[1]=tuple(state[1])
        state=tuple(state)
        return state
    elif state[1][-1]==1:
        for i in [0,1]:
            state[1][i]-=s[i]
            state[0][i]+=s[i]
        state[1][-1]=0
        state[0][-1]=1
        state[0]=tuple(state[0])
        state[1]=tuple(state[1])
        state=tuple(state)
        return state
    else:
        return ""

def make_tree(start,end):
    d={ }
    d[start]=[]
    all_nodes=[start]
    c=1
    h=-1
    source=start
    while(h<c and source!=end):
        h+=1
        source=all_nodes[h]
        for i in [0,1,2]:
            for j in [0,1,2]:
                if i+j<=2 and i+j>0:
                    state=travel(source,[i,j])
                    if state!="" and state not in all_nodes:
```

```

        if (state[0][0]>=state[0][1] or state[0][0]==0) and (state[1][0]>=state[1][1]or
state[1][0]==0):
            if state[0][0]>=0 and state[0][1]>=0 and state[1][0]>=0 and state[1][1]>=0:
                all_nodes.append(state)
                d[source].append(state)
                d[state]=[]
                if state==end:
                    break
            c=len(all_nodes)
        return d
m,n =map(int,input("Enter the number of missionaries and cannibals:").split())
start = ((m,n,1),(0,0,0))
end = ((0,0,0),(m,n,1))
key=end
state=start
d=make_tree(start,end)
#DFS
open_list=[start]
closed_list=[]
print("Open List--Closed List")
print(" ".join(str(open_list)), ".join(str(closed_list)),sep='--')
while open_list:
    source=open_list.pop(0)
    closed_list.append(source)
    if source==key:
        print('Required node is found',key)
        break
    for i in d[source]:
        if i not in closed_list and i not in open_list:
            open_list.insert(0,i)
print(" ".join(str(open_list)), ".join(str(closed_list)),sep='--',end='\n\n')

```

OUTPUT:**I.****GRAPH:**

Enter the number of missionaries and cannibals:3 3

Open List--Closed List

[((3, 3, 1), (0, 0, 0))]-=[]

$[((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0))]$

$[((3, 1, 0), (0, 2, 1)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1))]$

$[((3, 2, 1), (0, 1, 0)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1))]$

$[((3, 0, 0), (0, 3, 1)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((3, 2, 1), (0, 1, 0))]$

$[((3, 1, 1), (0, 2, 0)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1))]$

$[((1, 1, 0), (2, 2, 1)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0))]$

$[((2, 2, 1), (1, 1, 0)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1))]$

$[((0, 2, 0), (3, 1, 1)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0))]$

$[((0, 3, 1), (3, 0, 0)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0)), ((0, 2, 0), (3, 1, 1))]$

$[((0, 1, 0), (3, 2, 1)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0)), ((0, 2, 0), (3, 1, 1)), ((0, 3, 1), (3, 0, 0))]$

$[((1, 1, 1), (2, 2, 0)), ((0, 2, 1), (3, 1, 0)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0)), ((0, 2, 0), (3, 1, 1)), ((0, 3, 1), (3, 0, 0)), ((0, 1, 0), (3, 2, 1))]$

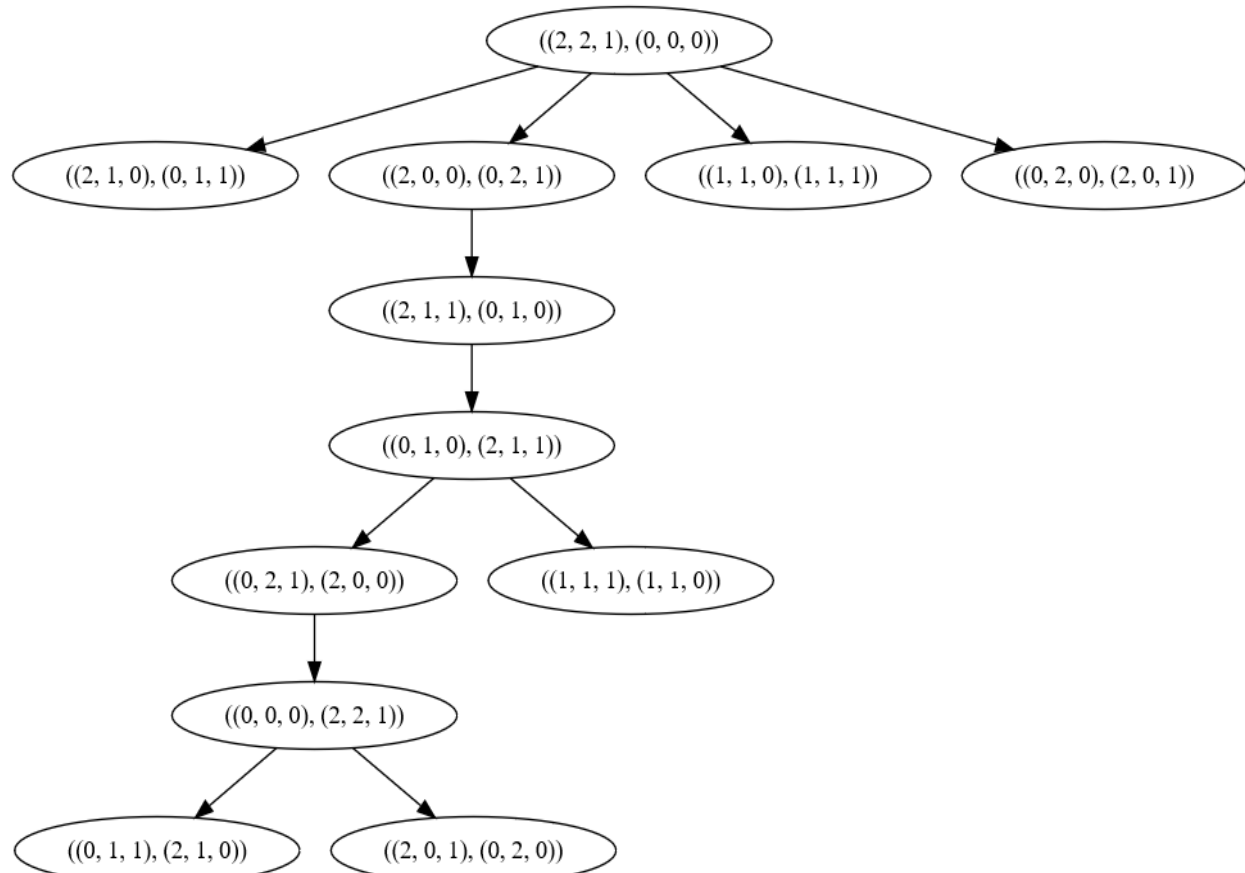
$[((0, 2, 1), (3, 1, 0)), ((3, 2, 0), (0, 1, 1))]--[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0)), ((0, 2, 0), (3, 1, 1)), ((0, 3, 1), (3, 0, 0)), ((0, 1, 0), (3, 2, 1)), ((1, 1, 1), (2, 2, 0))]$

```
[((0, 0, 0), (3, 3, 1)), ((3, 2, 0), (0, 1, 1))]-[((3, 3, 1), (0, 0, 0)), ((2, 2, 0), (1, 1, 1)), ((3, 1, 0), (0, 2, 1)), ((3, 2, 1), (0, 1, 0)), ((3, 0, 0), (0, 3, 1)), ((3, 1, 1), (0, 2, 0)), ((1, 1, 0), (2, 2, 1)), ((2, 2, 1), (1, 1, 0)), ((0, 2, 0), (3, 1, 1)), ((0, 3, 1), (3, 0, 0)), ((0, 1, 0), (3, 2, 1)), ((1, 1, 1), (2, 2, 0)), ((0, 2, 1), (3, 1, 0))]
```

Required node is found ((0, 0, 0), (3, 3, 1))

II.

GRAPH:



Enter the number of missionaries and cannibals:2 2

Open List--Closed List

```
[((2, 2, 1), (0, 0, 0))]-[[]]
[((0, 2, 0), (2, 0, 1)), ((1, 1, 0), (1, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((2, 1, 0), (0, 1, 1))]-[((2, 2, 1), (0, 0, 0))]
```

```
[((1, 1, 0), (1, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((2, 1, 0), (0, 1, 1))]-[((2, 2, 1), (0, 0, 0)), ((0, 2, 0), (2, 0, 1))]
```

```
[((2, 0, 0), (0, 2, 1)), ((2, 1, 0), (0, 1, 1))]-[((2, 2, 1), (0, 0, 0)), ((0, 2, 0), (2, 0, 1)), ((1, 1, 0), (1, 1, 1))]
```



```
[((2, 1, 1), (0, 1, 0)), ((2, 1, 0), (0, 1, 1))]-[((2, 2, 1), (0, 0, 0)), ((0, 2, 0), (2, 0, 1)), ((1, 1, 0), (1, 1, 1)), ((2, 0, 0), (0, 2, 1))]
```

```
[((0, 1, 0), (2, 1, 1)), ((2, 1, 0), (0, 1, 1))]-[((2, 2, 1), (0, 0, 0)), ((0, 2, 0), (2, 0, 1)), ((1, 1, 0), (1, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((2, 1, 1), (0, 1, 0))]
```

```
[((1, 1, 1), (1, 1, 0)), ((0, 2, 1), (2, 0, 0)), ((2, 1, 0), (0, 1, 1))]-[((2, 2, 1), (0, 0, 0)), ((0, 2, 0), (2, 0, 1)), ((1, 1, 0), (1, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((0, 2, 1), (2, 0, 0)), ((2, 1, 1), (0, 1, 0)), ((0, 1, 0), (2, 1, 1))]
```

```
[((0, 2, 1), (2, 0, 0)), ((2, 1, 0), (0, 1, 1))]-[((2, 2, 1), (0, 0, 0)), ((0, 2, 0), (2, 0, 1)), ((1, 1, 0), (1, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((2, 1, 1), (0, 1, 0)), ((0, 1, 0), (2, 1, 1)), ((1, 1, 1), (1, 1, 0))]
```

```
[((0, 0, 0), (2, 2, 1)), ((2, 1, 0), (0, 1, 1))]-[((2, 2, 1), (0, 0, 0)), ((0, 2, 0), (2, 0, 1)), ((1, 1, 0), (1, 1, 1)), ((2, 0, 0), (0, 2, 1)), ((2, 1, 1), (0, 1, 0)), ((0, 1, 0), (2, 1, 1)), ((1, 1, 1), (1, 1, 0)), ((0, 2, 1), (2, 0, 0))]
```

Required node is found ((0, 0, 0), (2, 2, 1))

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

Program:

```
def move(state): #FUNCTION TO PERFROM LEFT TO RIGHT OR RIGHT TO LEFT MOVE.
```

```
    state=list(state)
    temp=state[1]
    state[1]=state[-1]
    state[-1]=temp
    return ".join(state)
```

```
def clean(state): #FUNCTION TO CLEAN THE DIRT IF EXISTS.
```

```
    state=list(state)
    if state[0]=='1' and state[1]=='1':
        state[0]='0'
    if state[2]=='1' and state[3]=='1':
        state[2]='0'
    return ".join(state)
```

```
def make_tree(start,final): # GENERATE STATE SPACE TREE.
```

```
    d={ }
    d[start]=[]
    all_nodes=[start]
    c=1
    h=-1
    source=start
    while(h<c and source not in final):
        h+=1
        source=all_nodes[h]
        state=move(source)
        if state not in all_nodes:
            all_nodes.append(state)
            d[source].append(state)
            d[state]=[]
            if state in final:
                break
        state=clean(source)
        if state not in all_nodes:
            all_nodes.append(state)
            d[source].append(state)
            d[state]=[]
            if state in final:
                break
        c=len(all_nodes)
    return d,state
```

```
start=input("Enter start state:") #READING INPUT
```

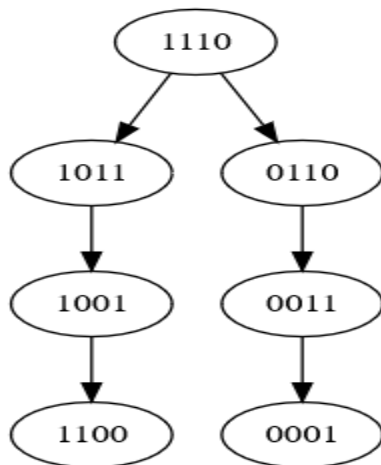
```
final=['0100','0001']
```

```
d,st=make_tree(start,final)
```

```

print(d)
#BFS
open_list=[(0,0)]
closed_list=[]
print("Open List--Closed List")
print(".join(str(open_list))",".join(str(closed_list)),sep='--')
while open_list:
    source=open_list.pop(0)
    closed_list.append(source)
    if source==final[-1]:
        break
    for i in d[source]:
        if i not in closed_list and i not in open_list:
            open_list.append(i)
    print(".join(str(open_list))",".join(str(closed_list)),sep='--',end='\n\n')

```

OUTPUT:**I.****GRAPH:**

Enter start state:1110

```
{'1110': ['1011', '0110'], '1011': ['1001'], '0110': ['0011'], '1001': ['1100'], '0011': ['0001'], '1100': [], '0001': []}
```

Open List--Closed List

```
['1110']--[]
```

```
['1011', '0110']--['1110']
```

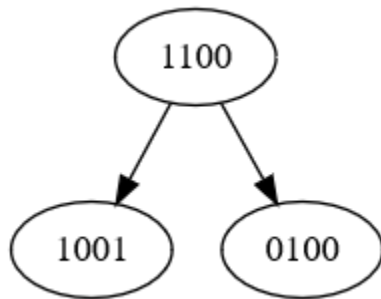
```
['0110', '1001']--['1110', '1011']
```

```
['1001', '0011']--['1110', '1011', '0110']
```

```
['0011', '1100']--['1110', '1011', '0110', '1001']
```

```
['1100', '0001']--['1110', '1011', '0110', '1001', '0011']
```

```
['0001']--['1110', '1011', '0110', '1001', '0011', '1100']
```

II.**GRAPH:**

```
Enter start state:1100
```

```
{'1100': ['1001', '0100'], '1001': [], '0100': []}
```

```
Open List--Closed List
```

```
['1100']--[]
```

```
['1001', '0100']--['1100']
```

```
['0100']--['1100', '1001']
```

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

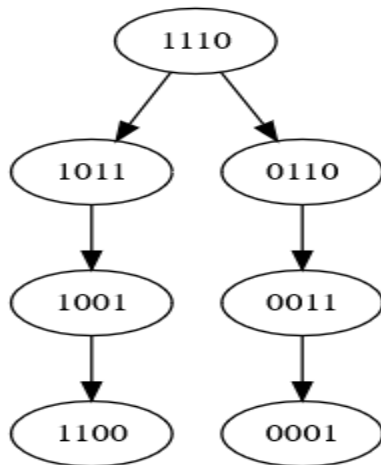
Program:

```
def move(state):
    state=list(state)
    temp=state[1]
    state[1]=state[-1]
    state[-1]=temp
    return ".join(state)
def clean(state):
    state=list(state)
    if state[0]=='1' and state[1]=='1':
        state[0]='0'
    if state[2]=='1' and state[3]=='1':
        state[2]='0'
    return ".join(state)
def make_tree(start,final):
    d={ }
    d[start]=[]
    all_nodes=[start]
    c=1
    h=-1
    source=start
    while(h<c and source not in final):
        h+=1
        source=all_nodes[h]
        state=move(source)
        if state not in all_nodes:
            all_nodes.append(state)
            d[source].append(state)
            d[state]=[]
            if state in final:
                break
        state=clean(source)
        if state not in all_nodes:
            all_nodes.append(state)
            d[source].append(state)
            d[state]=[]
            if state in final:
                break
        c=len(all_nodes)
    return d,state
start=input("Enter start state:")
final=['0100','0001']
d,st=make_tree(start,final)
```

```

print(d)
#DFS
open_list=[start]
closed_list=[]
print("Open List--Closed List")
print(".".join(str(open_list)).join(str(closed_list)),sep='--')
while open_list:
    source=open_list.pop(0)
    closed_list.append(source)
    if source in final:
        break
    for i in d[source]:
        if i not in closed_list and i not in open_list:
            open_list.insert(0,i)
    print(".".join(str(open_list)).join(str(closed_list)),sep='--',end='\n\n')

```

OUTPUT:**I.****GRAPH:**

```

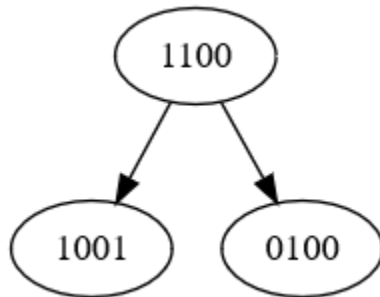
Enter start state:1110
{'1110': ['1011', '0110'], '1011': ['1001'], '0110': ['0011'], '1001': ['1100'], '0011': ['0001'], '1100': [], '0001': []}
Open List--Closed List
['1110']--[]
['0110', '1011']--['1110']

['0011', '1011']--['1110', '0110']

['0001', '1011']--['1110', '0110', '0011']

```

II. GRAPH:



Enter start state:1100

```
{'1100': ['1001', '0100'], '1001': [], '0100': []}
```

Open List--Closed List

```
['1100']--[]
```

```
['0100', '1001']--['1100']
```

5.Implement the following

a)Greedy Best First Search

Program:

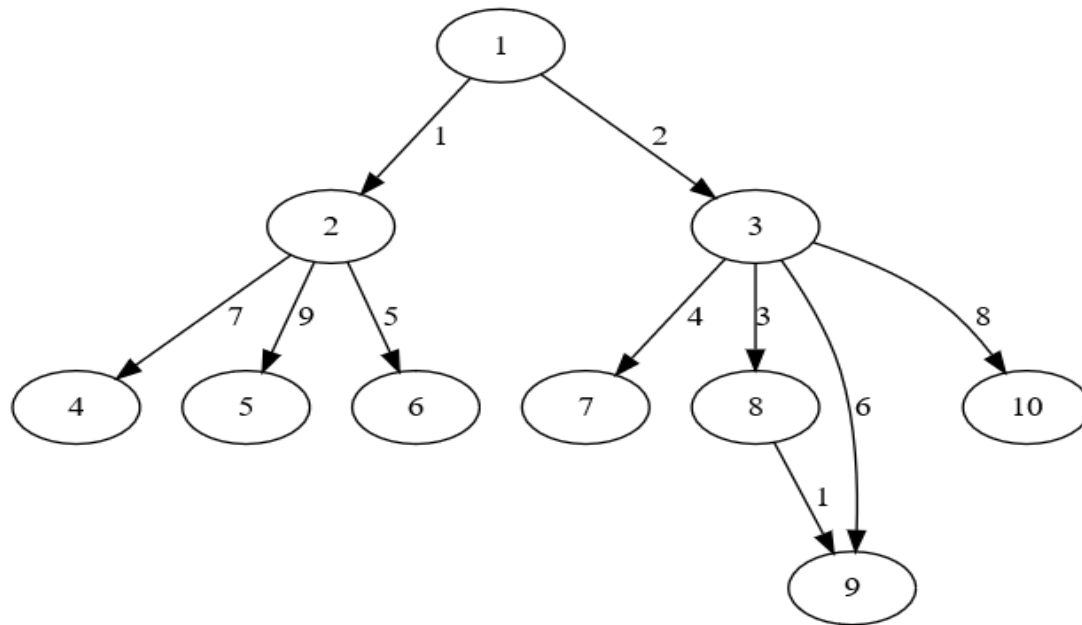
```

n=int(input("Enter number of nodes in the graph:")) #INPUT READING
cost = [[0] * n for i in range(n)]
for i in range(n):
    print(i+1,end=' ')
print("are the names of nodes")
print("Enter path costs as follows for each node accordingly and x to quit reading costs:")
print("<from_node> <to_node> <cost>")
while True:
    s=input("Enter:")
    if s=='x' or s=='X':
        break
    s=s.split()
    cost[int(s[0])-1][int(s[1])-1]=int(s[2])
h= list(map(int,input("Enter heuristic values in order:").split()))
source=int(input("Enter source node:"))-1
destination=int(input("Enter destination node:"))-1
def greed(cost,h,source,destination):
    op=[] #OPEN LIST
    c=[0] # CLOSED LIST
    while source!=destination:
        op.append(source)
        children=[]
        children_f=[]
        for i in range(len(cost[0])):
            if cost[source][i]>0: # TO CHECK IF CHILD EXISTS
                children.append(i)
                children_f.append(h[i])
        if len(children)>0:
            index=children_f.index(min(children_f)) #FINDING CHEAPEST PATH AVAILABLE
            source=children[index] #UPDATING SOURCE
        op.append(source)
    return op,c
op,c=greed(cost,h,source,destination)
print('Nodes travelled are:')
for i in op[:-1]:
    print(i+1,end='->')
print(op[-1]+1)

```

OUTPUT:

I. GRAPH:



Enter number of nodes in the graph:10

1 2 3 4 5 6 7 8 9 10 are the names of nodes

Enter path costs as follows for each node accordingly and x to quit
reading costs:

<from_node> <to_node> <cost>

Enter:1 2 1

Enter:1 3 2

Enter:2 4 7

Enter:2 5 9

Enter:2 6 5

Enter:3 7 4

Enter:3 8 3

Enter:3 9 6

Enter:3 10 8

Enter:8 9 1

Enter:x

Enter heuristic values in order:8 10 4 15 14 12 7 2 0 4

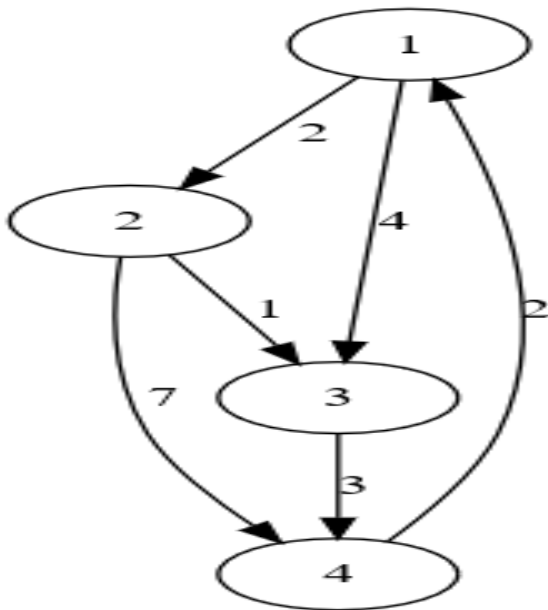
Enter source node:1

Enter destination node:9

Nodes travelled are:

1->3->9

II. GRAPH:



Enter number of nodes in the graph: 4

1 2 3 4 are the names of nodes

Enter path costs as follows for each node accordingly and x to quit
reading costs:

<from_node> <to_node> <cost>

Enter: 1 2 2

Enter: 1 3 4

Enter: 2 3 1

Enter: 2 4 7

Enter: 3 4 3

Enter: 4 1 2

Enter: x

Enter heuristic values in order: 5 3 2 0

Enter source node: 1

Enter destination node: 4

Nodes travelled are:

1->2->3->4

b)A* algorithm**Program:**

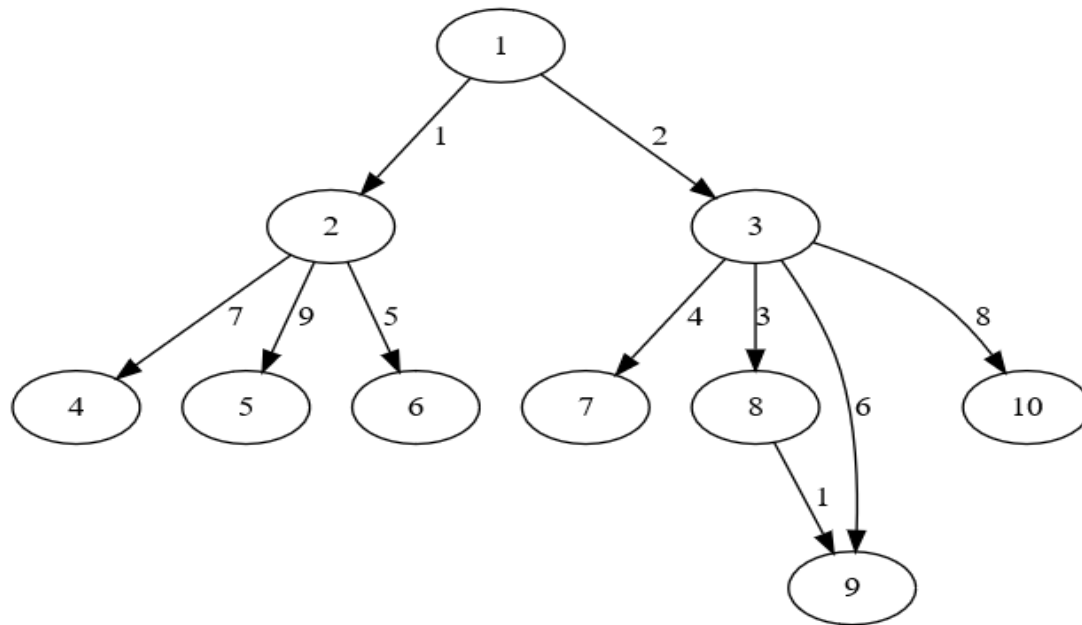
```

n=int(input("Enter number of nodes in the graph:")) #INPUT READING
cost = [[0] * n for i in range(n)]
for i in range(n):
    print(i+1,end=' ')
print("are the names of nodes")
print("Enter path costs as follows for each node accordingly and x to quit reading costs:")
print("<from_node> <to_node> <cost>")
while True:
    s=input("Enter:")
    if s=='x' or s=='X':
        break
    s=s.split()
    cost[int(s[0])-1][int(s[1])-1]=int(s[2])
h= list(map(int,input("Enter heuristic values in order:").split()))
source=int(input("Enter source node:"))-1
destination=int(input("Enter destination node:"))-1
def Astar(cost,h,source,destination):
    op=[]
    c=[0]
    while source!=destination:
        op.append(source)
        if len(op)>1:
            c.append(cost[op[-2]][op[-1]])
        children=[]
        children_f=[] #F(N) VALUES
        for i in range(len(cost[0])):
            if cost[source][i]>0:
                children.append(i)
                children_f.append(sum(c)+h[i]+cost[source][i]) #F(N)=G(N)+H(N)
        if len(children)>0:
            index=children_f.index(min(children_f)) #SLECTING CHILD BASED ON F(N) VLAUES
            source=children[index]
        op.append(source)
        c.append(cost[op[-2]][op[-1]])
    return op,c
op,c=Astar(cost,h,source,destination)
print('Nodes travelled are:')
for i in op[:-1]:
    print(i+1,end='->')
print(op[-1]+1)
print("Cost to traverse is ",sum(c))

```

OUTPUT:

I. GRAPH:



Enter number of nodes in the graph:10

1 2 3 4 5 6 7 8 9 10 are the names of nodes

Enter path costs as follows for each node accordingly and x to quit
reading costs:

<from_node> <to_node> <cost>

Enter:1 2 1

Enter:1 3 2

Enter:2 4 7

Enter:2 5 9

Enter:2 6 5

Enter:3 7 4

Enter:3 8 3

Enter:3 9 6

Enter:3 10 8

Enter:8 9 1

Enter:x

Enter heuristic values in order:8 10 4 15 14 12 7 2 0 4

Enter source node:1

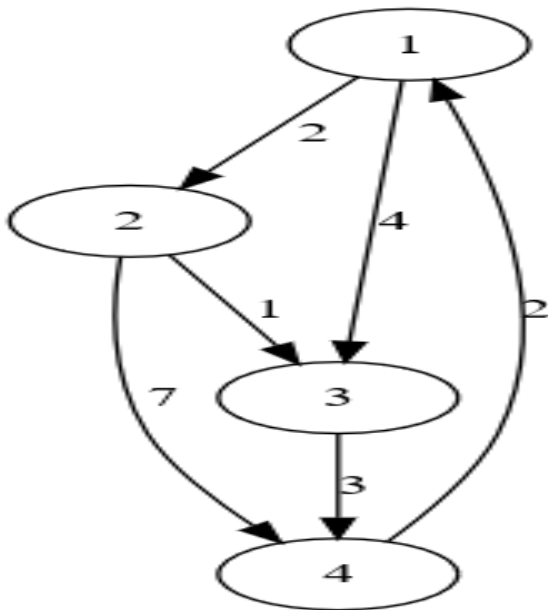
Enter destination node:9

Nodes travelled are:

1->3->8->9

Cost to traverse is 6

II. GRAPH:



Enter number of nodes in the graph: 4

1 2 3 4 are the names of nodes

Enter path costs as follows for each node accordingly and x to quit
reading costs:

<from_node> <to_node> <cost>

Enter: 1 2 2

Enter: 1 3 4

Enter: 2 3 1

Enter: 2 4 7

Enter: 3 4 3

Enter: 4 1 2

Enter: x

Enter heuristic values in order: 5 3 2 0

Enter source node: 1

Enter destination node: 4

Nodes travelled are:

1->2->3->4

Cost to traverse is 6

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

Program:

```

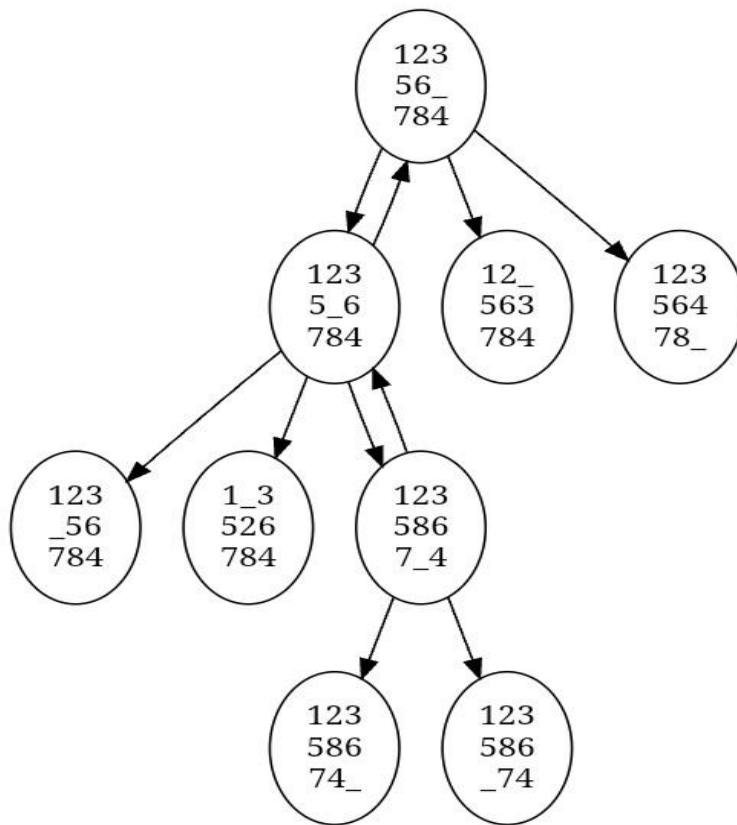
print("If the puzzle is \n1 2 3\n4 6\n7 5 8\ninput shall be given as 1234_6758")
start=input("Enter puzzle start state:")
final=input("Enter puzzle expected end state:")
def allpossible(state):
    l=[]
    index=state.index("_")
    if index in [0,1,3,4,6,7]: # MOVE RIGHT
        source=state[:]
        source[index],source[index+1]=source[index+1],source[index]
        l.append("".join(source))
    if index in [1,2,4,5,7,8]: #MOVE LEFT
        source=state[:]
        source[index],source[index-1]=source[index-1],source[index]
        l.append("".join(source))
    if index in [3,4,5,6,7,8]: #MOVE UP
        source=state[:]
        source[index],source[index-3]=source[index-3],source[index]
        l.append("".join(source))
    if index in [0,1,2,3,4,5]: #MOVE DOWN
        source=state[:]
        source[index],source[index+3]=source[index+3],source[index]
        l.append("".join(source))
    return l
def heuristics(state,final): #HEURISTIC FUNCTION FINDS NUMBER OF WRONG TILES.
    state=list(state)
    final=list(final)
    c=0
    for i in range(len(state)):
        if state[i]!=final[i]:
            c+=1
    return c
def EightPuzzle(source,final):
    all_nodes=[source]
    closed_list=[]
    d={}
    cost=0
    while source!=final:
        closed_list.append(source)
        l=allpossible(list(source))
        heur=[]
        for i in l:
            heur.append(heuristics(i,final))
        index=heur.index(min(heur))

```

```

d[source]=l
source=l[index]
cost=cost+1
closed_list.append(source)
return d,closed_list
EightPuzzle(start,final) #FUNCTION CALL

```

OUTPUT:**I.****GRAPH:**

If the puzzle is

1 2 3

4 6

7 5 8

input shall be given as 1234_6758

Enter puzzle start state:12356_784

Enter puzzle expected end state:123586_74

```

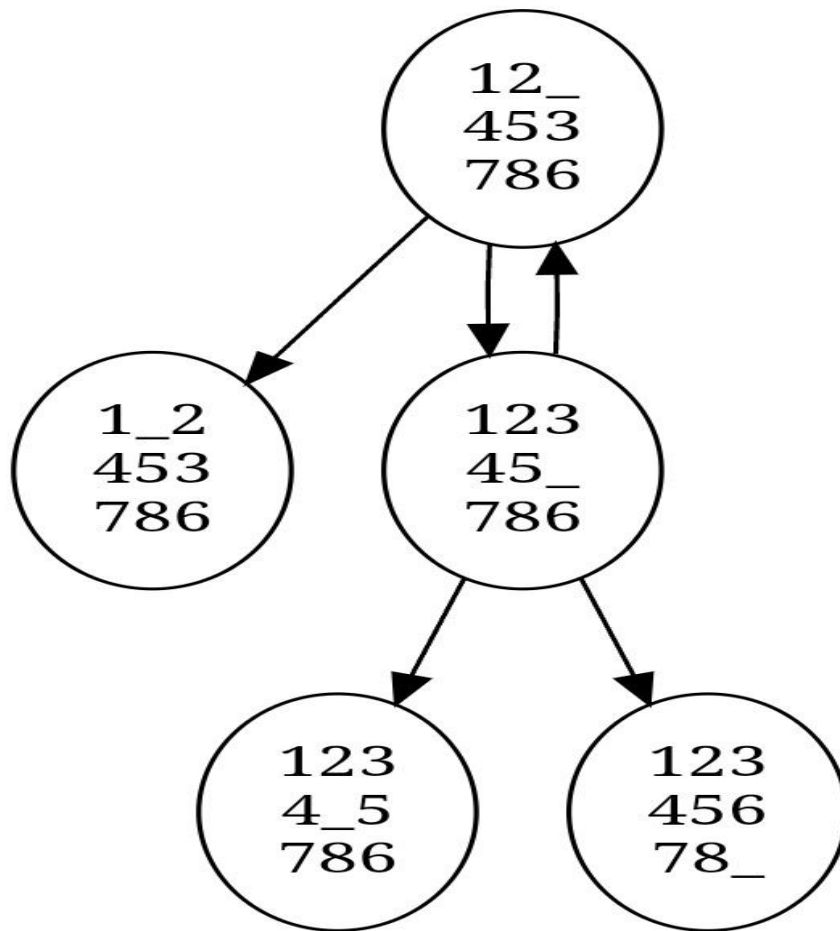
({'12356_784': ['1235_6784', '12_563784', '12356478_'],
 '1235_6784': ['12356_784', '123_56784', '1_3526784', '1235867_4'],
 '1235867_4': ['12358674_', '123586_74', '1235_6784']},

```

```
['12356_784', '1235_6784', '1235867_4', '123586_74']])
```

II.

GRAPH:



If the puzzle is

```
1 2 3
```

```
4   6
```

```
7 5 8
```

input shall be given as 1234_6758

Enter puzzle start state:12_453786

Enter puzzle expected end state:12345678_

```
{'12_453786': ['1_2453786', '12345_786'],
 '12345_786': ['1234_5786', '12_453786', '12345678_']},
['12_453786', '12345_786', '12345678_']])
```


7. Implement AO* algorithm for General graph problem.

Program:

```
def cost(H, condition, weight=1):
    costs = {}
    if 'AND' in condition:
        AND_nodes = condition['AND']
        Path_A = ' AND '.join(AND_nodes)
        PathA = sum(H[node] + weight for node in AND_nodes)
        costs[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)
        costs[Path_B] = PathB
    return costs

def update_cost(H, Conditions, weight=1):
    main_nodes = list(Conditions.keys())
    main_nodes.reverse()
    least_cost = {}
    for key in main_nodes:
        condition = Conditions[key]
        print(key, ':', Conditions[key], '>>>', cost(H, condition, weight))
        c = cost(H, condition, weight)
        H[key] = min(c.values())
        least_cost[key] = cost(H, condition, weight)
    return least_cost

def shortest_path(Start, Updated_cost, H):
    Path = Start
    if Start in Updated_cost.keys():
        Min_cost = min(Updated_cost[Start].values())
        key = list(Updated_cost[Start].keys())
        values = list(Updated_cost[Start].values())
        Index = values.index(Min_cost)
        Next = key[Index].split()
        if len(Next) == 1:
            Start = Next[0]
            Path += '<--' + shortest_path(Start, Updated_cost, H)
        else:
            Path += '<--(' + key[Index] + ') '
```

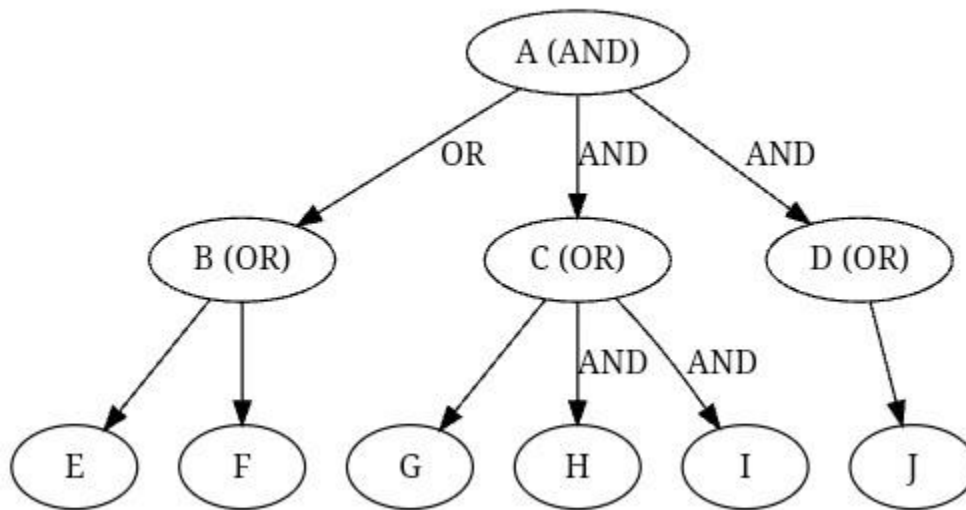
```

Start = Next[0]
Path += '[' + shortest_path(Start, Updated_cost, H) + ' + '

Start = Next[-1]
Path += shortest_path(Start, Updated_cost, H) + ']'

return Path
H = eval(input('Enter nodes with heuristic costs: '))
Conditions = eval(input('Enter graph: '))
weight = 1
print('Updated Cost:')
Updated_cost = update_cost(H, Conditions, weight=1)
print('Shortest Path:\n', shortest_path('A', Updated_cost, H))

```

OUTPUT:**I.****GRAPH:**

```

Enter nodes with heuristic costs: {'A': -1, 'B': 5, 'C': 2, 'D': 4,
'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I': 0, 'J': 0}
Enter graph: {'A': {'OR': ['B'], 'AND': ['C', 'D']}, 'B': {'OR': ['E',
'F']}, 'C': {'OR': ['G'], 'AND': ['H', 'I']}, 'D': {'OR': ['J']}}

```

Updated Cost:

```

D : {'OR': ['J']} >>> {'J OR': 0}
C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'G OR': 3, 'H AND I': 2}
B : {'OR': ['E', 'F']} >>> {'E OR F': 6}
A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'B OR': 5, 'C AND D': 6}

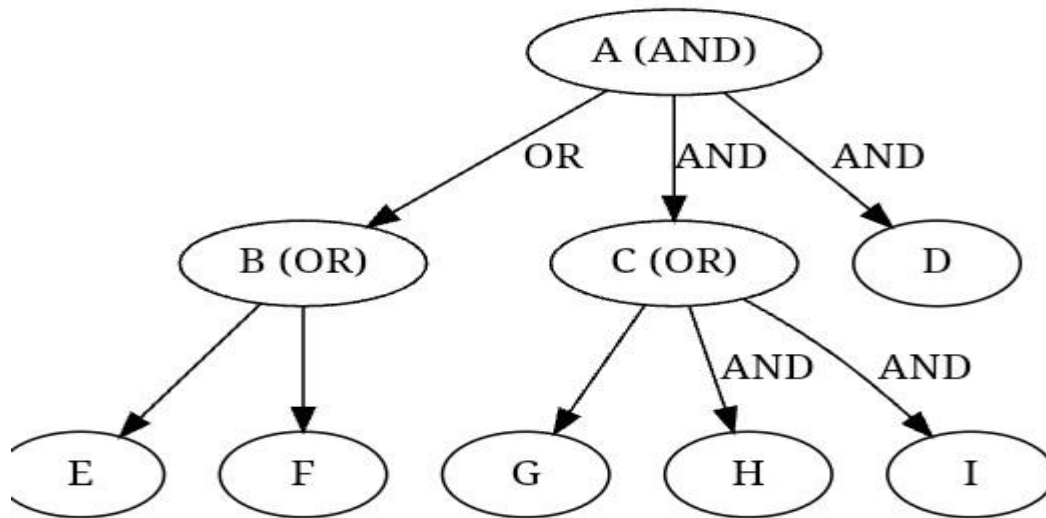
```

Shortest Path:

A \leftarrow -(B OR) [F + E] + (C AND D) [I + H]

II.

GRAPH:



Enter nodes with heuristic costs: {'A': -1, 'B': 5, 'C': 2, 'D': 4, 'E': 7, 'F': 9, 'G': 3, 'H': 0, 'I': 0, 'J': 0}

Enter graph: {'A': {'OR': ['B'], 'AND': ['C', 'D']}, 'B': {'OR': ['E', 'F']}, 'C': {'OR': ['G'], 'AND': ['H', 'I']}}

Updated Cost:

C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'H AND I': 2, 'G': 4}

B : {'OR': ['E', 'F']} >>> {'E OR F': 8}

A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 8, 'B': 9}

Shortest Path:

A \leftarrow -(C AND D) [C \leftarrow -(H AND I) [H + I] + D]

8. Implement Game trees using

a)MINIMAX algorithm

Program:

```
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()))
```

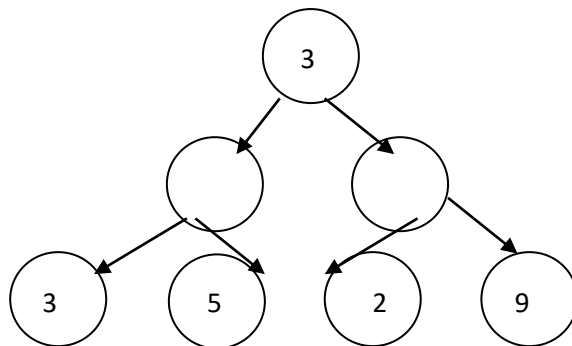
```
treeDepth = int(math.log(len(scores), 2))
```

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

OUTPUT:

I.

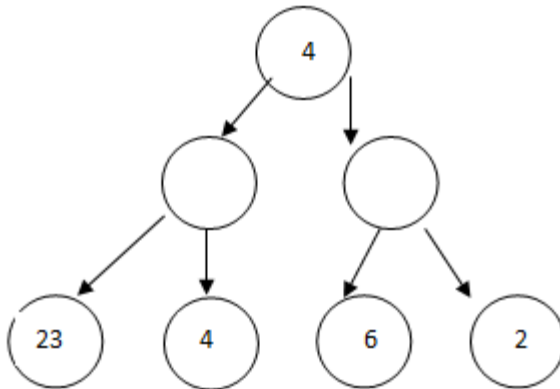
GRAPH:



```
Enter scores:3 5 2 9
```

```
The optimal value is: 3
```

II. GRAPH:



Enter scores:23 4 6 2

The optimal value is: 4

b) Alpha-Beta pruning

Program:

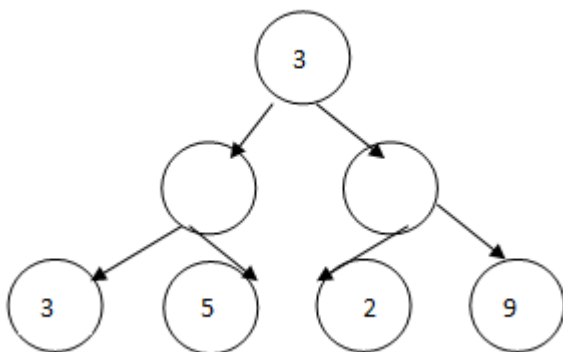
MAX, MIN = 1000, -1000

```
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:

I.

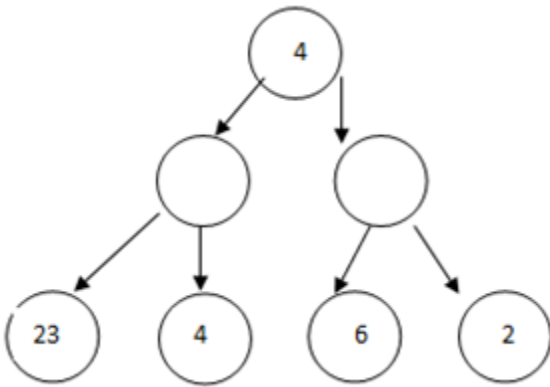
GRAPH:



Enter scores:3 5 2 9

The optimal value is: 12

II.

GRAPH:

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

9. Implement Crypt arithmetic problems.

Program:

```
import itertools
def number(word, digit_map):
    return int(''.join(str(digit_map[letter]) for letter in word))
def solve_cryptarithmic(puzzle): # TRYING ALL POSSIBLE MOVES WITH CONSTRIANTS
    words = puzzle.split()
    unique_characters = set(''.join(words))
    if len(unique_characters) > 10: #NO MORE THAN 10 UNIQUE CHARS (0-9)
        return "Invalid puzzle: More than 10 unique characters"
    leading_characters = set(word[0] for word in words)
    if len(leading_characters) > 2:
        return "Invalid puzzle: More than 2 words start with the same character"
    for digits in itertools.permutations(range(10), len(unique_characters)):
        digit_map = dict(zip(unique_characters, digits))
        if all(digit_map[word[0]] != 0 for word in leading_characters):
            if sum(number(word, digit_map) for word in words[:-1]) == number(words[-1], digit_map):
                return digit_map
    return "No solution found"
puzzle = input("Enter the cryptarithmic puzzle (words separated by spaces): ")
solution = solve_cryptarithmic(puzzle)
print("Solution:", solution)
```

OUTPUT:

I.

```
Enter the cryptarithmic puzzle (words separated by spaces): send
more money
Solution: {'s': 9, 'e': 5, 'n': 6, 'd': 7, 'm': 1, 'o': 0, 'r': 8,
'y': 2}
```

II.

```
Enter the cryptarithmic puzzle (words separated by spaces): TWO TWO
FOUR
Solution: {'O': 4, 'R': 8, 'T': 7, 'U': 6, 'F': 1, 'W': 3}
```