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

Source code:

```
graph = {'A':['B', 'C', 'D'], 'B':['A', 'E', 'F'], 'C':['A', 'F', 'G'], 'D':['A'], 'E':['B', 'H',
'I'], 'F':['B','J'], 'G':['C'], 'H':['E'], 'I':['E'], 'J':['F']}

print("Graph : ", graph)


def BFS(g, root, goal):
    open = [root]
    closed = []
    while(goal not in closed):
        print(open, closed)
        x = open.pop(0)
        if x in g.keys():
            for i in g[x]:
                if i not in open+closed:
                    open.append(i)
        closed.append(x)
    print(open, closed)


root = input("Enter root node : ")
goal = input("Enter goal node : ")
BFS(graph, root, goal)
```

Output:

Graph : {'A': ['B', 'C', 'D'], 'B': ['A', 'E', 'F'], 'C': ['A', 'F', 'G'], 'D': ['A'], 'E': ['B', 'H', 'I'], 'F': ['B', 'J'], 'G': ['C'], 'H': ['E'], 'I': ['E'], 'J': ['F']}

Enter root node : A

Enter goal node : G

['A'] []

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

['C', 'D', 'E', 'F'] ['A', 'B']

['D', 'E', 'F', 'G'] ['A', 'B', 'C']

['E', 'F', 'G'] ['A', 'B', 'C', 'D']

['F', 'G', 'H', 'I'] ['A', 'B', 'C', 'D', 'E']

['G', 'H', 'I', 'J'] ['A', 'B', 'C', 'D', 'E', 'F']

['H', 'I', 'J'] ['A', 'B', 'C', 'D', 'E', 'F', 'G']

b) DFS

Source code:

```
graph = {'A':['B', 'C', 'D'], 'B':['A', 'E', 'F'], 'C':['A', 'F', 'G'], 'D':['A'], 'E':['B', 'H',
'I'], 'F':['B','J'], 'G':['C'], 'H':['E'], 'I':['E'], 'J':['F']}

print("Graph : ", graph)


def DFS(g, root, goal):
    open = [root]
    closed = []
    while(goal not in closed):
        print(open, closed)
        x = open.pop(0)
        closed.append(x)
        if x in g.keys():
            for i in g[x][::-1]:
                if i not in open+closed:
                    open.insert(0, i)
    print(open, closed)


root = input("Enter root node : ")
goal = input("Enter goal node : ")
DFS(graph, root, goal)
```

Output:

Graph : {'A': ['B', 'C', 'D'], 'B': ['A', 'E', 'F'], 'C': ['A', 'F', 'G'], 'D': ['A'], 'E': ['B', 'H', 'I'], 'F': ['B', 'J'], 'G': ['C'], 'H': ['E'], 'I': ['E'], 'J': ['F']}

Enter root node : A

Enter goal node : G

['A'] []

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

['E', 'F', 'C', 'D'] ['A', 'B']

['H', 'I', 'F', 'C', 'D'] ['A', 'B', 'E']

['I', 'F', 'C', 'D'] ['A', 'B', 'E', 'H']

['F', 'C', 'D'] ['A', 'B', 'E', 'H', 'I']

['J', 'C', 'D'] ['A', 'B', 'E', 'H', 'I', 'F']

['C', 'D'] ['A', 'B', 'E', 'H', 'I', 'F', 'J']

['G', 'D'] ['A', 'B', 'E', 'H', 'I', 'F', 'J', 'C']

['D'] ['A', 'B', 'E', 'H', 'I', 'F', 'J', 'C', 'G']

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

Source code:

```
m, n = [int(x) for x in input("Enter jug capacities : ").split()]
res = int(input("Enter final result : "))

root, dest = [0, 0], [res, 0]

def water_jug_bfs(r, d, m, n): #5, 3
    visited, open  = [], [r]
    graph = {}
    while(d not in open):
        layer = []
        for i in open:
            pos = []
            if(i[0]==0):
                pos.append([m, i[1]])
            if(i[1]==0):
                pos.append([i[0], n])
            if(i[0]==m):
                pos.append([0, i[1]])
            if(i[1]==n):
                pos.append([i[0], 0])
            if(i[1]>0 and m-i[0]>0):
                a=m-i[0]
                if a>=i[1]:
                    pos.append([i[0]+i[1], 0])
```

```
                else:
                    pos.append([m, i[1]-a])
            if(i[0]>0 and n-i[1]>0):
                a=n-i[1]
                if a>=i[0]:
                    pos.append([0, i[0]+i[1]])
                else:
                    pos.append([i[0]-a, n])
            graph[tuple(i)]=pos
            layer+=pos
        for j in layer:
            if j in visited:
                layer.remove(j)
        visited+=open
        open=layer
    return graph

g = water_jug_bfs(root, dest, m, n)
```

```
def path_taken(g,  r,  d):
    x = d
    ans = [tuple(d)]
    while(x != r):
        for i in g.keys():
            if x in g[i]:
                ans.insert(0, i)
                break
        x = list(ans[0])
    return ans
path_taken(g, root, dest)
```

Output:

[(0, 0), (5, 0), (1, 4), (1, 0), (0, 1), (5, 1), (2, 4), (2, 0)]

b) DFS

Source code:

```
j1=int(input("Enter capacity of jug1 : "))
j2=int(input("Enter capacity of jug2 : "))
goal_node=list(map(int,input("enter the goal node : ").split()))
initial_node=[0,0]
def dfs(initial_node,goal_node,j1,j2):
    path=[]
    open_list=[]
    closed_list=[]
    open_list.append(initial_node)
    while open_list:
        s=open_list.pop()
        path.append(s)
        if s[0]==goal_node[0] and s[1]==goal_node[1]:
            print("Path:")
            return path
        closed_list.append([s[0],s[1]])
        if(s[0]<j1 and ([j1,s[1]] not in closed_list)):
            open_list.append([j1,s[1]])
            closed_list.append([j1,s[1]])
        if(s[1]<j2 and ([s[0],j2] not in closed_list)):
            open_list.append([s[0],j2])
            closed_list.append([s[0],j2])
        if(s[0]>0 and ([0,s[1]] not in closed_list)):
            open_list.append([0,s[1]])
            closed_list.append([0,s[1]])
        if(s[1]>0 and ([s[0],0] not in closed_list)):
```

```
            open_list.append([s[0],0])
            closed_list.append([s[0],0])
        if((s[0]+s[1])<=j1 and s[1]>=0 and ([(s[0]+s[1]),0] not in closed_list)):
            open_list.append([(s[0]+s[1]),0])
            closed_list.append([(s[0]+s[1]),0])
        if((s[0]+s[1])<=j2 and s[0]>0 and ([0,(s[0]+s[1])] not in closed_list)):
            open_list.append([0,(s[0]+s[1])])
            closed_list.append([0,(s[0]+s[1])])
        if((s[0]+s[1])>=j1 and s[1]>=0 and ([j1,(s[1]-(j1-s[0]))] not in closed_list)):
            open_list.append([j1,(s[1]-(j1-s[0]))])
            closed_list.append([j1,(s[1]-(j1-s[0]))])
        if((s[0]+s[1])>=j2 and s[0]>0 and ([(s[0]-(j2-s[1])),j2] not in closed_list)):
            open_list.append([(s[0]-(j2-s[1])),j2])
            closed_list.append([(s[0]-(j2-s[1])),j2])
    return "no path"

dfs(initial_node,goal_node,j1,j2)
```

Output:

Enter capacity of jug1 : 5

Enter capacity of jug2 : 3

enter the goal node : 2 0

Path:

[[0, 0], [0, 3], [3, 0], [3, 3], [5, 1], [0, 1], [1, 0], [1, 3], [4, 0], [4, 3], [5, 2], [0, 2], [2, 0]]

3. Implement Exhaustive search techniques using :

a) Uniform cost search

Source code:

```
import copy
graph = {'A':{'B':6, 'C':4}, 'B':{'D':4}, 'C':{'F':8}, 'D':{'G':2}, 'F':{'E':9, 'H':7},
'G':{'E':9, 'H':3}, 'H':{'E':4}, 'E':{}}
print("Graph : ", graph)
def select_node(g):
    node, least = ' ', 9999
    for i in g.keys():
        if g[i]<least:
            node, least = i, g[i]
    return node


def UCS(g, root, goal):
    open, closed = {root:0}, {}
    leastcost = 9999
    while len(open)>0:
        print(open, closed)
        x =  select_node(open)
        cost = open.pop(x)
        closed[x]=cost
        to_be_updated = copy.deepcopy(g[x])
        for i in to_be_updated.keys():
            to_be_updated[i]+=cost
        for i in to_be_updated.keys():
            if i in open.keys():
                if open[i]>to_be_updated[i]:
```

```
                open[i]=to_be_updated[i]
            else:
                open[i]=to_be_updated[i]
        if goal in open.keys() and leastcost>open[goal]:
            leastcost=open[goal]
    print(open, closed)
    return leastcost

root = input("Enter root node : ")
goal = input("Enter goal node : ")
least_cost = UCS(graph, root, goal)
print("Least cost form root to goal node is : ", least_cost)
```

Output:

```
Graph :  {'A': {'B': 6, 'C': 4}, 'B': {'D': 4}, 'C': {'F': 8}, 'D': {'G': 2}, 'F': {'E': 9,
'H': 7}, 'G': {'E': 9, 'H': 3}, 'H': {'E': 4}, 'E': {}}
Enter root node : A
Enter goal node : E
{'A': 0} {}
{'B': 6, 'C': 4} {'A': 0}
{'B': 6, 'F': 12} {'A': 0, 'C': 4}
{'F': 12, 'D': 10} {'A': 0, 'C': 4, 'B': 6}
{'F': 12, 'G': 12} {'A': 0, 'C': 4, 'B': 6, 'D': 10}
{'G': 12, 'E': 21, 'H': 19} {'A': 0, 'C': 4, 'B': 6, 'D': 10, 'F': 12}
{'E': 21, 'H': 15} {'A': 0, 'C': 4, 'B': 6, 'D': 10, 'F': 12, 'G': 12}
{'E': 19} {'A': 0, 'C': 4, 'B': 6, 'D': 10, 'F': 12, 'G': 12, 'H': 15}
{} {'A': 0, 'C': 4, 'B': 6, 'D': 10, 'F': 12, 'G': 12, 'H': 15, 'E': 19}
Least cost form root to goal node is : 19
```

b) Depth-First Iterative Deepening

Source code:

```
graph = {1:[2, 3],
      2:[4, 5],
      3:[6, 7],
      4:[8, 9],
      5:[10, 11]}
graph.update(dict.fromkeys([6,  7,  8,  9,  10,  11],  []))
print(graph)

def BFS_level(g, root, h):
   level =  [root]
   sub_level = []
   graph = {}
   for i in range(h):
      while len(level)>0:
         x = level.pop(0)
         graph[x]=g[x]
         sub_level.extend(g[x])
      level.extend(sub_level)
      sub_level=[]
   return level, graph

def DFS(g, root, goal):
   open = [root]
   closed = []
   while(goal not in closed):
      x = open.pop(0)
```

```
        closed.append(x)
        if x in g.keys():
            l = g[x][::-1]
            for i in l:
                if i not in open:
                    open.insert(0, i)
        # print(open, closed)
    print("Path is : ",closed)

def DFID(g, root, goal):
    h=0
    nodes, sub_graph = BFS_level(g, root, h)
    while goal not in nodes:
        h+=1
        nodes, sub_graph = BFS_level(g, root, h)
    print("level :", h+1)
    DFS(sub_graph, root, goal)

DFID(graph, 1, 5)
```

Output:

level : 3

Path is : [1, 2, 4, 5]

c) Bidirectional search

Source code:

```
graph = {
    'A': ['B', 'C', 'D'],
    'B': ['E', 'F', 'A'],
    'C': ['F', 'G', 'A'],
    'D': ['A'],
    'E': ['H', 'I', 'B'],
    'F': ['J', 'B', 'C'],
    'G': ['C'],
    'H': ['E'],
    'I': ['E'],
    'J': ['F']
}
print(graph)

def Bidirectional_BFS(g, root, goal):
    flag = 0
    open1 = [root]
    closed1 = []
    open2 = [goal]
    closed2 = []
    graph1 = {}
    graph2={}
    while(len(set(open1+open2))==len(open1+open2)):
        # Front
        x1 = open1.pop(0)
        closed1.append(x1)
```

```
    sub_nodes = g[x1]
    graph1[x1]=[]
    for i in sub_nodes:
        if i not in open1+closed1:
            graph1[x1].append(i)
            open1.append(i)

    #Back
    x2 = open2.pop(0)
    closed2.append(x2)
    sub_nodes = g[x2]
    graph2[x2]=[]
    for i in sub_nodes:
        if i not in open2+closed2:
            graph2[x2].append(i)
            open2.append(i)
# print(graph1, graph2)
path1, path2 = [], []
match_ele = ''
for i in open1:
    if i in open2:
        match_ele = i
        break
print("Matched Element is : ", match_ele)
x1, x2 = match_ele, match_ele
while root!=x1:
    for i in graph1.keys():
        if x1 in graph1[i]:
```

```
                path1.insert(0, i)
                break
            x1 = path1[0]
    path1.append(match_ele)

    while goal!=x2:
        for i in graph2.keys():
            if x2 in graph2[i]:
                path2.insert(0, i)
                break
        x2 = path2[0]

    print("Final Path : ",path1+path2[::-1])

root = input('Enter root node : ')
goal = input('Enter Goal node : ')
Bidirectional_BFS(graph, root, goal)
```

Output:

```
{'A': ['B', 'C', 'D'], 'B': ['E', 'F', 'A'], 'C': ['F', 'G', 'A'], 'D': ['A'], 'E': ['H', 'I', 'B'],
'F': ['J', 'B', 'C'], 'G': ['C'], 'H': ['E'], 'I': ['E'], 'J': ['F']}
Enter root node : A
Enter Goal node : J
Matched Element is : C
Final Path :  ['A', 'C', 'F', 'J']
```

4. Implement Missionaries and Cannibals problem with Search tree generation using :

```
def Possibilities(m1, c1, m2, c2, x):
    pos = []
    if x[0][2]==1: # boat is on left side
        if m1>=2 and (m1-2>=c1 or m1-2==0) and m2+2>=c2:
            pos.append([[m1-2, c1, 0], [m2+2, c2, 1]])
        if m1>=1 and c1>=1 and m1-1>=c1-1 and m2+1>=c2+1:
            pos.append([[m1-1, c1-1, 0], [m2+1, c2+1, 1]])
        if c1>=2 and (m2>=c2+2 or m2==0):
            pos.append([[m1, c1-2, 0], [m2, c2+2, 1]])
        if m1>=1 and (m1-1>=c1 or m1-1==0) and m2+1>=c2:
            pos.append([[m1-1, c1, 0], [m2+1, c2, 1]])
        if c1>=1 and (m2>=c2+1 or m2==0):
            pos.append([[m1, c1-1, 0], [m2, c2+1, 1]])

    elif x[1][2] == 1: # boat is on right side
        if m2>=2 and m1+2 >= c1 and (m2-2 >= c2 or m2-2==0):
            pos.append([[m1+2, c1, 1], [m2-2, c2, 0]])
        if m2>=1 and c2>=1 and m1+1 >= c1+1 and m2-1 >= c2-1:
            pos.append([[m1+1, c1+1, 1], [m2-1, c2-1, 0]])
        if c2>=2 and ((m1 >= c1+2 and m2>=c2-2) or m1==0):
            pos.append([[m1, c1+2, 1], [m2, c2-2, 0]])
        if m2>=1 and m1+1 >= c1 and (m2-1 >= c2 or m2-1==0):
            pos.append([[m1+1, c1, 1], [m2-1, c2, 0]])
        if c2>=1 and (m1 >= c1+1 or m1==0):
            pos.append([[m1, c1+1, 1], [m2, c2-1, 0]])
    return pos
```

a) BFS

Source code:

```
M = int(input("Enter no.of Missionaries : "))
C = int(input("Enter no.of Cannibals : "))
Root, goal = [[M, C, 1], [0, 0, 0]], [[0, 0, 0], [M, C, 1]]

def Missionaries_and_Cannibals_BFS(r, g, m, c):
    open, closed = [], []
    open.append(r)
    while g not in open:
        x = open.pop(0)
        m1, c1, m2, c2 = x[0][0], x[0][1], x[1][0], x[1][1]
        pos = Possibilities(m1, c1, m2, c2, x)
        closed.append(x)
        for i in closed+open:
            if i in pos:
                pos.remove(i)
        open.extend(pos)
    print(closed)
Missionaries_and_Cannibals_BFS(root, goal, M, C)
```

Output:

Enter no.of Missionaries : 3

Enter no.of Cannibals : 3

[[[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, 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]]]

b) DFS

Source code:

```
def Missionaries_and_Cannibals_DFS(r, g, m, c):
    open = []
    closed = []
    open.append(r)
    while g not in open:
        x = open.pop(0)
        m1, c1, m2, c2 = x[0][0], x[0][1], x[1][0], x[1][1]
        pos = Possibilities(m1, c1, m2, c2, x)
        closed.append(x)
        for i in closed+open:
            if i in pos:
                pos.remove(i)
        open.extend(pos[::-1])
    print(closed)

Missionaries_and_Cannibals_DFS(root, goal, M, C)
```

Output:

[[[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]]]

5. Implement Vacuum World problem with Search tree generation using :

```
pos = input("Enter position of the Vacuum machine, either 'A' or 'B' : ")
# ['Dirty', 'Position']
root = [[1, 0], [1, 0]]
if pos=='A':
    root[0][-1]=1
else:
    root[-1][-1]=1
goal = [[[0, 1], [0, 0]], [[0, 0], [0, 1]]]
print(root, goal)
```

Output:

```
Enter position of the Vacuum machine, either 'A' or 'B' : A
[[1, 1], [1, 0]] [[[0, 1], [0, 0]], [[0, 0], [0, 1]]]
```

```
import copy
def Operation(c):
    cond, pos = copy.deepcopy(c), []
    location = 0 if cond[0][-1]==1 else 1
    if cond[location][0]==1:
        cond[location][0]=0
        pos.append(cond)
    cond = copy.deepcopy(c)
    cond[location][-1]=0
    location = 1 if location==0 else 0
    cond[location][-1]=1
    pos.append(cond)
    return pos
```

a) BFS

Source code:

```
def Vacuum_World_BFS(r, g):
    open, closed = [r], []
    graph = {}
    while (g[0] not in open):
        print(open)
        x = open.pop(0)
        closed.append(x)
        pos = Operation(x)
        graph[str(x)]=pos
        for i in pos:
            if i not in open+closed:
                open.append(i)
    return graph
graph = Vacuum_World_BFS(root, goal)
```

Output:

```
[[[1, 1], [1, 0]]]
[[[0, 1], [1, 0]], [[1, 0], [1, 1]]]
[[[1, 0], [1, 1]], [[0, 0], [1, 1]]]
[[[0, 0], [1, 1]], [[1, 0], [0, 1]]]
[[[1, 0], [0, 1]], [[0, 0], [0, 1]]]
[[[0, 0], [0, 1]], [[1, 1], [0, 0]]]
```

b) DFS

Source code:

```
def Vacuum_World_DFS(r, g):
    open = [r]
    closed = []
    graph = {}
    while (g[0] not in open):
        print(open)
        x = open.pop(0)
        closed.append(x)
        pos = Operation(x)
        graph[str(x)]=pos
        for i in pos[::-1]:
            if i not in open+closed:
                open.append(i)
    return graph
graph = Vacuum_World_DFS(root, goal)
```

Output:

```
[[[1, 1], [1, 0]]]
[[[1, 0], [1, 1]], [[0, 1], [1, 0]]]
[[[0, 1], [1, 0]], [[1, 0], [0, 1]]]
[[[1, 0], [0, 1]], [[0, 0], [1, 1]]]
[[[0, 0], [1, 1]], [[1, 1], [0, 0]]]
[[[1, 1], [0, 0]], [[0, 0], [0, 1]]]
```

6. Implement the following :

```
graph = {
    'A':{'B':1, 'C':2},
    'B':{'D':7, 'E':9, 'F':5},
    'C':{'G':4, 'H':3, 'I':6, 'J':8},
    'D':{'B':7},
    'E':{'B':9},
    'F':{'B':5},
    'G':{'C':4},
    'H':{'I':1, 'C':3},
    'I':{'H':1, 'C':6},
    'J':{'C':8}
}
h = {'A':8, 'B':10, 'C':4, 'D':15, 'E':14, 'F':12, 'G':7, 'H':2, 'I':0, 'J':4}
print(graph, h)

def select_key(open):
    least = 10000
    node = ''
    for i in open.keys():
        if least>open[i]:
            node = i
            least=open[i]
    return node
```

a) Greedy Best First Search

Source code:

```
def Greedy_BFS(g, h, start, goal):
    open = {start:h[start]}
    closed = []
    cost = 0
    while goal not in closed:
        node = select_key(open)
        cost = open.pop(select_key(open))
        closed.append(node)
        new = g[node].copy()
        for i in new:
            new[i]=h[i]
        open.update(new)
        print(open, closed)
    cost = 0
    for i in range(len(closed)-1):
        cost+=g[closed[i]][closed[i+1]]
    print("Least cost : ",cost)
Greedy_BFS(graph, h, 'A', 'I')
```

Output:

```
{'B': 10, 'C': 4} ['A']
{'B': 10, 'G': 7, 'H': 2, 'I': 0, 'J': 4} ['A', 'C']
{'B': 10, 'G': 7, 'H': 2, 'J': 4, 'C': 4} ['A', 'C', 'I']
Least cost : 8
```

b) A* algorithm

Source code:

```
def A_star(g, h, start, goal):
    open, closed = {start:h[start]}, []
    least_cost = 10000
    while(goal not in closed):
        node = select_key(open)
        cost = open.pop(select_key(open))
        closed.append(node)
        to_be_updated = g[node].copy()
        for i in to_be_updated.keys():
            to_be_updated[i]=(cost-h[node]) + to_be_updated[i] + h[i]
        open.update(to_be_updated)
        if goal in open.keys():
            if open[goal]<least_cost:
                least_cost=open[goal]
        print(open, closed)
    print("Least cost : ", least_cost)
A_star(graph, h, 'A', 'I')
```

Output:

```
{'B': 11, 'C': 6} ['A']
{'B': 11, 'G': 13, 'H': 7, 'I': 8, 'J': 14} ['A', 'C']
{'B': 11, 'G': 13, 'I': 6, 'J': 14, 'C': 12} ['A', 'C', 'H']
{'B': 11, 'G': 13, 'J': 14, 'C': 16, 'H': 9} ['A', 'C', 'H', 'I']
Least cost : 6
```

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

Source cost:

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

import copy
def  Modified(s, i, f):
    state = copy.deepcopy(s)
    state[f[0]][f[1]], state[i[0]][i[1]]  = state[i[0]][i[1]], state[f[0]][f[1]]
    return state

def Possibility(state, goal, g, previous):
    n = len(state)
    for i in range(n):
        if 0 in state[i]:
            x, y = i, state[i].index(0)
            break
    current = [x, y]
    pos = []
    move = []
    if x > 0 and previous!='u':
        pos.append([x-1, y])
        move.append('d')
    if x < n-1 and previous!='d':
```

```
        pos.append([x+1, y])
        move.append('u')
    if y > 0 and previous!='r':
        pos.append([x, y-1])
        move.append('l')
    if y < n-1 and previous!='l':
        pos.append([x, y+1])
        move.append('r')
    res = []
    for i in pos:
        a = Modified(state, current, i)
        res.append(a)
    f_n =  []
    for i in res:
        f_n.append(Heuristic(i, goal)+g)
    least = f_n.index(min(f_n))
    return res[least], move[least]

def Heuristic(state, goal):
    h = 0
    for i in range(len(state)):
        for j in range(len(state[0])):
            if state[i][j]!=goal[i][j]:
                h+=1
    return h-1

def Eight_Puzzle(initial, goal):
    g=0
```

```
    open = [initial]
    closed = []
    previous = ''
    while goal not in open:
        print(open)
        x = open.pop(0)
        pos, previous = Possibility(x, goal, g, previous)
        open.append(pos)
        g+=1
        time.sleep(.5)
    print(open)

Eight_Puzzle(Initial, Goal)
```

Output:

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

8. Implement AO* algorithm for General graph problem.

Source code:

```
graph = {
    'A': {'OR': ['B'], 'AND': ['C', 'D']},
    'B': {'OR': ['E', 'F']},
    'C': {'OR': ['G'], 'AND': ['H', 'I']},
    'D': {'OR': ['J']},
    'E': {},
    'F': {},
    'G': {},
    'H': {},
    'I': {},
    'J': {}
}
h = {'A':-1, 'B': 4, 'C': 2, 'D': 3, 'E': 6, 'F': 8, 'G': 2, 'H': 0, 'I':0, 'J':0}

def AO_star(g, h, root, goal):
    open = [root]
    closed = []
    while goal not in closed:
        print(open, closed)
        x = open.pop(0)
        closed.append(x)
        sub_graph = g[x]
        f_n_or = []
        f_n_and = 9999
        chk1, chk2 = 0, 0
        if 'OR' in sub_graph.keys():
```

```
        chk1 = 1
        or_part = sub_graph['OR']
        for i in or_part:
            f_n_or.append(1+h[i])
    if 'AND' in sub_graph.keys():
        chk2 = 1
        and_part = sub_graph['AND']
        f_n_and = 0
        for i in and_part:
            f_n_and += 1+h[i]
    if chk1==1 or chk2==1:
        min_f_n_or = min(min(f_n_or), 9999)
        if min_f_n_or<=f_n_and:
            open.append(or_part[f_n_or.index(min_f_n_or)])
        else:
            open.extend(and_part)
        m = min(f_n_and, min_f_n_or)
        if x!=root and h[x]!=m:
            h[x]=m
            for i in g.keys():
                a = []
                for j in g[i].values():
                    a+=j
                if x in a:
                    open = [i]
                    for j in closed[::-1]:
                        if j!=i:
                            closed.remove(j)
```

```
                    closed.remove(i)
                    break
    print(open, closed)
    return closed

path = AO_star(graph, h, 'A', 'J')
print("path = ",path)
```

Output:

```
['A'] []
['B'] ['A']
['A'] []
['C', 'D'] ['A']
['D', 'H', 'I'] ['A', 'C']
['A'] []
['C', 'D'] ['A']
['D', 'H', 'I'] ['A', 'C']
['H', 'I', 'J'] ['A', 'C', 'D']
['I', 'J'] ['A', 'C', 'D', 'H']
['J'] ['A', 'C', 'D', 'H', 'I']
[] ['A', 'C', 'D', 'H', 'I', 'J']
path =  ['A', 'C', 'D', 'H', 'I', 'J']
```

9. Implement Game trees using :

a) MINIMAX algorithm

Source code:

```
def MIN_MAX(leafs, mode):
    flag = mode #max 1
    if flag == 1:
        print("Max level : ", leafs)
    else:
        print("Min level : ", leafs)
    while(len(leafs)>1):
        new = []
        while len(leafs)>1:
            x = leafs.pop(0)
            y = leafs.pop(0)
            if flag==1:
                new.append(max(x, y))
            else:
                new.append(min(x, y))
        if len(leafs)==1:
            new.append(leafs.pop())
        leafs = new
        if flag == 1:
            flag=0
            print("Min level : ", leafs)
        else:
            flag=1
            print("Max level : ", leafs)
```

```
    return leafs[0]

leaf_nodes = list(map(int, input("Enter leaf nodes : ").split()))
mode = int(input("Enter 1 if MAX, else enter 0 : "))

res = MIN_MAX(leaf_nodes, mode)
print("Root node of final MIN_MAX tree is : ", res)
```

Output:

```
Enter leaf nodes : 10 5 -10 7 5 -7 -5
Enter 1 if MAX, else enter 0 : 1
Max level :  [10, 5, -10, 7, 5, -7, -5]
Min level :  [10, 7, 5, -5]
Max level : [7, -5]
Min level :  [7]
Root node of final MIN_MAX tree is :  7
```

b) 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(0, 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(0, 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

if__name__== "__main__":
    values = [3, 5, 6, 9, 1, 2, 0, -1]
    print("The optimal value is :", minimax(0, 0, True, values, MIN, MAX))
```

Output:

The optimal value is : 5

10. Implement Crypt arithmetic problems.

Source code:

```
import itertools
def get_value(word, substitution):
    s = 0
    factor = 1
    for letter in reversed(word):
        s += factor * substitution[letter]
        factor *= 10
    return s


def solve2(equation):
    left, right = equation.lower().replace(' ', '').split('=')
    left = left.split('+')
    letters = set(right)
    for word in left:
        for letter in word:
            letters.add(letter)
    letters = list(letters)
    digits = range(10)
    for perm in itertools.permutations(digits, len(letters)):
        sol = dict(zip(letters, perm))
        if sum(get_value(word, sol) for word in left) == get_value(right, sol):
            print(' + '.join(str(get_value(word, sol)) for word in left) + " = {}
(mapping : {})".format(get_value(right, sol), sol))


solve2('SEND + MORE = MONEY')
```

Output:

9567 + 1085 = 10652 (mapping : {'o': 0, 'e': 5, 'n': 6, 'r': 8, 's': 9, 'd': 7, 'm': 1, 'y': 2})

2817 + 368 = 3185 (mapping : {'o': 3, 'e': 8, 'n': 1, 'r': 6, 's': 2, 'd': 7, 'm': 0, 'y': 5})

2819 + 368 = 3187 (mapping : {'o': 3, 'e': 8, 'n': 1, 'r': 6, 's': 2, 'd': 9, 'm': 0, 'y': 7})

3719 + 457 = 4176 (mapping : {'o': 4, 'e': 7, 'n': 1, 'r': 5, 's': 3, 'd': 9, 'm': 0, 'y': 6})

3712 + 467 = 4179 (mapping : {'o': 4, 'e': 7, 'n': 1, 'r': 6, 's': 3, 'd': 2, 'm': 0, 'y': 9})

3829 + 458 = 4287 (mapping : {'o': 4, 'e': 8, 'n': 2, 'r': 5, 's': 3, 'd': 9, 'm': 0, 'y': 7})

3821 + 468 = 4289 (mapping : {'o': 4, 'e': 8, 'n': 2, 'r': 6, 's': 3, 'd': 1, 'm': 0, 'y': 9})

5731 + 647 = 6378 (mapping : {'o': 6, 'e': 7, 'n': 3, 'r': 4, 's': 5, 'd': 1, 'm': 0, 'y': 8})

5732 + 647 = 6379 (mapping : {'o': 6, 'e': 7, 'n': 3, 'r': 4, 's': 5, 'd': 2, 'm': 0, 'y': 9})

5849 + 638 = 6487 (mapping : {'o': 6, 'e': 8, 'n': 4, 'r': 3, 's': 5, 'd': 9, 'm': 0, 'y': 7})

6419 + 724 = 7143 (mapping : {'o': 7, 'e': 4, 'n': 1, 'r': 2, 's': 6, 'd': 9, 'm': 0, 'y': 3})

6415 + 734 = 7149 (mapping : {'o': 7, 'e': 4, 'n': 1, 'r': 3, 's': 6, 'd': 5, 'm': 0, 'y': 9})

6524 + 735 = 7259 (mapping : {'o': 7, 'e': 5, 'n': 2, 'r': 3, 's': 6, 'd': 4, 'm': 0, 'y': 9})

6853 + 728 = 7581 (mapping : {'o': 7, 'e': 8, 'n': 5, 'r': 2, 's': 6, 'd': 3, 'm': 0, 'y': 1})

6851 + 738 = 7589 (mapping : {'o': 7, 'e': 8, 'n': 5, 'r': 3, 's': 6, 'd': 1, 'm': 0, 'y': 9})

7316 + 823 = 8139 (mapping : {'o': 8, 'e': 3, 'n': 1, 'r': 2, 's': 7, 'd': 6, 'm': 0, 'y': 9})

7429 + 814 = 8243 (mapping : {'o': 8, 'e': 4, 'n': 2, 'r': 1, 's': 7, 'd': 9, 'm': 0, 'y': 3})

7539 + 815 = 8354 (mapping : {'o': 8, 'e': 5, 'n': 3, 'r': 1, 's': 7, 'd': 9, 'm': 0, 'y': 4})

7531 + 825 = 8356 (mapping : {'o': 8, 'e': 5, 'n': 3, 'r': 2, 's': 7, 'd': 1, 'm': 0, 'y': 6})

7534 + 825 = 8359 (mapping : {'o': 8, 'e': 5, 'n': 3, 'r': 2, 's': 7, 'd': 4, 'm': 0, 'y': 9})

7649 + 816 = 8465 (mapping : {'o': 8, 'e': 6, 'n': 4, 'r': 1, 's': 7, 'd': 9, 'm': 0, 'y': 5})

7643 + 826 = 8469 (mapping : {'o': 8, 'e': 6, 'n': 4, 'r': 2, 's': 7, 'd': 3, 'm': 0, 'y': 9})

8324 + 913 = 9237 (mapping : {'o': 9, 'e': 3, 'n': 2, 'r': 1, 's': 8, 'd': 4, 'm': 0, 'y': 7})

8432 + 914 = 9346 (mapping : {'o': 9, 'e': 4, 'n': 3, 'r': 1, 's': 8, 'd': 2, 'm': 0, 'y': 6})

8542 + 915 = 9457 (mapping : {'o': 9, 'e': 5, 'n': 4, 'r': 1, 's': 8, 'd': 2, 'm': 0, 'y': 7})