1.Implement Exhaustive search techniques using

a. <u>BFS</u>:

SOURCE CODE:

```
Import pandas as pd
gra={'a':['b','c','d'],'b':['e','f'],'c':['f','g'],'d':[],'e':['h','i'],'f':['j'],'g':[],'h':[],'i':[],'j':[]}
go=input("enter the goal state") init1=input("enter the root") n=len(gra) o=[]
c=[]
o.append(init1)
def
bfs(o,c,init1,n):
while(n>0):
n1=o[0]
l=len(gra[n1])
print(o,c)
   c.append(o[0])
   o.pop(0) for i in
   range(0,l):
     if(gra[n1][i] not in o):
       o.append(gra[n1][i])
   if(n1==go): n=-
     1
     print(o,c)
  if(n!=-1):
    print("goal state is not present")
    print(o,c)
```

```
bfs(o,c,init1,n)
```

OUTPUT:

```
enter the goal state j
enter the root a
['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']
['i', 'j'] ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
['j'] ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i']
[] ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j']
```

b. DFS:

SOURCE CODE:

```
graph = {
  '5' : ['3','7'],
  '3' : ['2', '4'],
  '7' : ['8'],
  '2' : [],
  '4' : ['8'],
  '8' : [] } visited = set() # Set to keep track of visited nodes
of graph.
def dfs(visited, graph, node): #function for dfs
    if node not in visited: print
        (node) visited.add(node) for
        neighbour in graph[node]:
            dfs(visited, graph, neighbour)
# Driver Code
print("Following is the Depth-First Search")
dfs(visited, graph, '5')
```

OUTPUT:

Following is the Depth-First Search

5 3 2 4 8 7

c. Uniform Cost Search:

SOURCE CODE:

```
from queue import PriorityQueue
gr={'a':{'b':3,'c':2},'b':{'d':6,'e':5,'a':3},'c':{'f':5,'a':2},'f':{'g':2,'c':5},'e':{'b':5},'g':{'f':
2},'d':{'b':6}}
parent={'a':[]} close=[]
path=[] def
backtrack(now,cost):
  if(cost<0):
    return
  if(cost==0):
    path.append(now
    ) print(path)
    return
  close.append(now)
  path.append(now)
  for i in parent[now]:
  if i not in close:
     backtrack(i,cost-gr[now][i])
  return
def bestfirst(gr,goal):
    open_list=PriorityQueue()
    closed_list=[]
    open_list.put((0,'a')) while
    open_list.empty()==False:
```

```
cost,now=open_list.get() if
now==goal:
      print("min cost required
      is",cost) backtrack(goal,cost);
      break
   closed_list.append(now)
   for edge in gr[now].keys():
      if edge not in closed_list:
         open_list.put((cost+gr[now][edge],edge))
         if edge not in parent.keys():
         parent[edge]=[now]
         else:
           parent[edge].append(now)
#print(closed_list) bestfirst(gr,'g')
```

OUTPUT:

min cost required is 9

['g', 'f', 'c', 'a']

d. <u>Depth-First Iterative Deepening</u>:

SOURCE CODE:

```
goal=(input("enter the goal node"))
croot=(input("enter the root node"))
maxdepth=int(input("enter the max depth"))
path=[] def
dfs(croot,goal,graph,maxdepth,path):
 path.append(croot)
 if croot==goal:
 return True
 if maxdepth<=0:
  return False
 for i in graph[croot]:
  if(dfs(i,goal,graph,maxdepth-1,path)):
   return True
def iterative(croot,goal,graph,maxdepth,path):
 for i in range(0,maxdepth):
  if(dfs(croot,goal,graph,i,path)):
  return True
if iterative(croot,goal,d,maxdepth,path):
 print("path found")
 print(path)
else:
 print("path not found")
 print(path)
```

OUTPUT:

```
enter the goal nodeH
enter the root nodeA
enter the max depth3
path found
['A', 'A', 'B', 'C', 'A', 'B', 'E', 'F', 'C', 'D', 'H']
```

e. **Bidirectional**:

SOURCE CODE:

```
gr={0:[5,1,2,11],5:[0,6,7],1:[7,0,3],7:[8,1],3:[9,4,1],2:[0,10,12],11:[0],6:[5],8:[7],
9:[3],4:[3],10:[2],12:[2]}

open_list1=[] closed_list1=[]

def

bfs(gr,open_list,closed_list):

   now=open_list[0]

   open_list.remove(now)

   if(now not in gr.keys()):

      closed_list.append(now)

   if now not in closed_list:

      temp=gr[now]

      #print(now)

      for i in temp:

         if i not in open_list:

            open_list.append(i)

      closed_list.append(now)

open_list2=[] closed_list2=[]

open_list1.append(5)

open_list2.append(4) def

intersection_list(list1, list2):

  list3 = [value for value in list1 if value in list2]

return list3 def intersection(l1,l2):

   if len(intersection_list(l1, l2)):
```

```
        return True
    return False
while(not intersection(open_list1,open_list2)):
    bfs(gr,open_list1,closed_list1) bfs(gr,open_list2,closed_list2)
print(intersection_list(open_list1, open_list2))
```

OUTPUT:

[1]

2. Implement water jug problem with Search tree generation using

a. BFS:

SOURCE CODE:

```
open_list=[] closed_list=[]

path=[] def

new(li,open_list,i,j): if(li[0]<i):

open_list.append([i,li[1]])

if(li[1]<j):

open_list.append([li[0],j])

   if(li[0]>0):

      open_list.append([0,li[1]])

   if(li[1]>0):

      open_list.append([li[0],0])

   if li[0]+li[1]<=i and li[1]>0:

      open_list.append([li[0]+li[1],0])

   if li[0]+li[1]<=j and li[0]>0:

      open_list.append([0,li[0]+li[1]])

   if li[0]+li[1]>=j and li[0]>0: open_list.append([li[0]-(j-

      li[1]),j])

   if li[0]+li[1]>=i and li[1]>0:

      open_list.append([i,li[1]-(i-li[0])])

def bfs(open_list,closed_list,goal,i,j):

   now=open_list[0]

   open_list.remove(now) if(now==goal):

   print("reached") return
```

```
    if now in closed_list:
        bfs(open_list,closed_list,goal,i,j)
    if now not in closed_list:
        new(now,open_list,i,j)
        closed_list.append(now) bfs(open_list,closed_list,goal,i,j)
i=int(input("enter  1st  jug  amount"))
j=int(input("enter  2nd  jug  amount"))
g=int(input("enter goal state amount"))
initial=[0,0]                    goal=[g,0]
closed_list.append(initial)
new(initial,open_list,i,j)
bfs(open_list,closed_list,goal,i,j)
#print(list(reversed(closed_list)))
print(path)
```

OUTPUT:

```
enter  1st  jug  amount  5
enter  2nd  jug  amount  4
enter goal state amount 3
reached
[]
```

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]])
       #Fill #rule1 if(s[0]<j1 and ([j1,s[1]] not in
       closed_list)):
          open_list.append([j1,s[1]])
          closed_list.append([j1,s[1]])
       #rule2 if(s[1]<j2 and ([s[0],j2] not in
       closed_list)):
          open_list.append([s[0],j2])
          closed_list.append([s[0],j2])
       #Empty #rule3 if(s[0]>0 and ([0,s[1]] not
       in closed_list)):
          open_list.append([0,s[1]])
          closed_list.append([0,s[1]])
```

```
#rule4 if(s[1]>0 and ([s[0],0] not in
closed_list)):
   open_list.append([s[0],0])
   closed_list.append([s[0],0])
#Transfer #rule5 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])
#rule6 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])])

#rule7 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]))])
#rule8 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 jug2:4

enter the goal node2 0

Path:

[[0, 0],

[0, 4],

[4, 0],

[4, 4],

[5, 3],

[0, 3],

[3, 0],

[3, 4],

[5, 2],

[0, 2],

[2, 0]]

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

a. <u>BFS</u>:

SOURCE CODE:

```
open_list=[]
closed_list=[]
def move(s):
  if s[0][2]==1:
    #move from left side to right side #moving 2 machinaries if ((s[0][0]-2>0 and
    s[0][0]-2>=s[0][1]) or s[0][0]-2==0) and s[1][0]+2>=s[1][1]:
      open_list.append([[s[0][0]-2,s[0][1],0],[s[1][0]+2,s[1][1],1]])
      #moving 2 cannibals if s[0][1]-2>=0 and ((s[0][0]>=s[0][1]-2 and s[0][0]!=0)
or s[0][0]==0) and ((s[1][0]!=0 and s[1][1]+2<=s[1][0]) or s[1][0]==0):
      open_list.append([[s[0][0],s[0][1]-2,0],[s[1][0],s[1][1]+2,1]])
      #moving 1 machinary and 1 cannibal if s[0][1]>=0 and ((s[0][0]-1>0 and
    s[0][0]-1>=s[0][1]-1) or s[0][0]-1==0) and
s[1][0]+1>=s[1][1]+1:
      open_list.append([[s[0][0]-1,s[0][1]-1,0],[s[1][0]+1,s[1][1]+1,1]])
    #moving 1 machinary
    if ((s[0][0]-1>0 and s[0][0]-1>=s[0][1]) or s[0][0]-1==0) and s[1][0]+1>=s[1][1]:
      open_list.append([[s[0][0]-1,s[0][1],0],[s[1][0]+1,s[1][1],1]])
    #moving 1 cannibal
    if s[0][1]-1>=0 and ((s[0][0]>=s[0][1]-1 and s[0][0]!=0) or s[0][0]==0) and
((s[1][0]!=0 and s[1][0]>=s[1][1]+1) or s[1][0]==0):
      open_list.append([[s[0][0],s[0][1]-1,0],[s[1][0],s[1][1]+1,1]])
```

```
  else:
   #move from right side to left side #moving 2 machinaries if ((s[1][0]-2>0 and
   s[1][0]-2>=s[1][1]) or s[1][0]-2==0) and s[0][0]+2>=s[0][1]:
    open_list.append([[s[0][0]+2,s[0][1],1],[s[1][0]-2,s[1][1],0]])
    #moving 2 cannibals if s[1][1]-2>=0 and ((s[1][0]!=0 and s[1][0]>=s[1][1]-2)
or s[1][0]==0) and ((s[0][1]+2<=s[0][0] and s[0][0]!=0) or s[0][0]==0):
    open_list.append([[s[0][0],s[0][1]+2,1],[s[1][0],s[1][1]-2,0]])
    #moving 1 machinary and 1 cannibal if s[1][1]>=0 and ((s[1][0]-1>0 and
   s[1][0]-1>=s[1][1]-1) or s[1][0]-1==0) and
s[0][0]+1>=s[0][1]+1:
    open_list.append([[s[0][0]+1,s[0][1]+1,1],[s[1][0]-1,s[1][1]-1,0]])
   #moving 1 machinary
   if ((s[1][0]-1>0 and s[1][0]-1>=s[1][1]) or s[1][0]-1==0) and s[0][0]+1>=s[0][1]:
    open_list.append([[s[0][0]+1,s[0][1],1],[s[1][0]-1,s[1][1],0]])
   #moving 1 cannibal
   if s[1][1]-1>=0 and ((s[1][0]!=0 and s[1][0]>=s[1][1]-1) or s[1][0]==0) and
((s[0][0]>=s[0][1]+1 and s[0][0]!=0) or s[0][0]==0):
    open_list.append([[s[0][0],s[0][1]+1,1],[s[1][0],s[1][1]-1,0]])
path=[] def
bfs(open_list,closed_list,goal):
   now=open_list[0]
   open_list.remove(now)
   if(now==goal):
   print("reached") return

   if now in closed_list:
```

```
        bfs(open_list,closed_list,goal)
    if now not in closed_list:
        move(now)
        closed_list.append(now)
        bfs(open_list,closed_list,goal)
        path.append(now)
initial=[[3,3,1],[0,0,0]]
move(initial)
closed_list.append(initial)
goal=[[0,0,0],[3,3,1]]
bfs(open_list,closed_list,goal)
print(closed_list)
```

OUTPUT:

reached

[[[3, 3, 1], [0, 0, 0]], [[3, 1, 0], [0, 2, 1]], [[2, 2, 0], [1, 1, 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]], [[0, 2, 1], [3, 1, 0]]]

b. DFS:

SOURCE CODE:

```
open_list=[]
closed_list=[]
def move(s):
  if s[0][2]==1:
    #move from left side to right side #moving 2 machinaries if ((s[0][0]-2>0 and
    s[0][0]-2>=s[0][1]) or s[0][0]-2==0) and s[1][0]+2>=s[1][1]:
      open_list.insert(0,[[s[0][0]-2,s[0][1],0],[s[1][0]+2,s[1][1],1]])
      #moving 2 cannibals if s[0][1]-2>=0 and ((s[0][0]>=s[0][1]-2 and s[0][0]!=0)
or s[0][0]==0) and ((s[1][0]!=0 and s[1][1]+2<=s[1][0]) or s[1][0]==0):
      open_list.insert(0,[[s[0][0],s[0][1]-2,0],[s[1][0],s[1][1]+2,1]])
      #moving 1 machinary and 1 cannibal if s[0][1]>=0 and ((s[0][0]-1>0 and
    s[0][0]-1>=s[0][1]-1) or s[0][0]-1==0) and
s[1][0]+1>=s[1][1]+1:
      open_list.insert(0,[[s[0][0]-1,s[0][1]-1,0],[s[1][0]+1,s[1][1]+1,1]])
    #moving 1 machinary
    if ((s[0][0]-1>0 and s[0][0]-1>=s[0][1]) or s[0][0]-1==0) and s[1][0]+1>=s[1][1]:
      open_list.insert(0,[[s[0][0]-1,s[0][1],0],[s[1][0]+1,s[1][1],1]])
    #moving 1 cannibal
    if s[0][1]-1>=0 and ((s[0][0]>=s[0][1]-1 and s[0][0]!=0) or s[0][0]==0) and
((s[1][0]!=0 and s[1][0]>=s[1][1]+1) or s[1][0]==0):
      open_list.insert(0,[[s[0][0],s[0][1]-1,0],[s[1][0],s[1][1]+1,1]])
  else:
    #move from right side to left side
```

```
#moving 2 machinaries if ((s[1][0]-2>0 and s[1][0]-2>=s[1][1]) or s[1][0]-
2==0) and s[0][0]+2>=s[0][1]:
    open_list.insert(0,[[s[0][0]+2,s[0][1],1],[s[1][0]-2,s[1][1],0]])
    #moving 2 cannibals if s[1][1]-2>=0 and ((s[1][0]!=0 and s[1][0]>=s[1][1]-2)
or s[1][0]==0) and ((s[0][1]+2<=s[0][0] and s[0][0]!=0) or s[0][0]==0):
    open_list.insert(0,[[s[0][0],s[0][1]+2,1],[s[1][0],s[1][1]-2,0]])
    #moving 1 machinary and 1 cannibal if s[1][1]>=0 and ((s[1][0]-1>0 and
s[1][0]-1>=s[1][1]-1) or s[1][0]-1==0) and
s[0][0]+1>=s[0][1]+1:
    open_list.insert(0,[[s[0][0]+1,s[0][1]+1,1],[s[1][0]-1,s[1][1]-1,0]])
#moving 1 machinary
if ((s[1][0]-1>0 and s[1][0]-1>=s[1][1]) or s[1][0]-1==0) and s[0][0]+1>=s[0][1]:
    open_list.insert(0,[[s[0][0]+1,s[0][1],1],[s[1][0]-1,s[1][1],0]])
#moving 1 cannibal
if s[1][1]-1>=0 and ((s[1][0]!=0 and s[1][0]>=s[1][1]-1) or s[1][0]==0) and
((s[0][0]>=s[0][1]+1 and s[0][0]!=0) or s[0][0]==0):
    open_list.insert(0,[[s[0][0],s[0][1]+1,1],[s[1][0],s[1][1]-1,0]])
def dfs(open_list,closed_list,goal):
    now=open_list[0]
    open_list.remove(now)
    if(now==goal):
    print("reached") return
    if now in closed_list:
        dfs(open_list,closed_list,goal)
    if now not in closed_list:
```

```
    move(now)
    closed_list.append(now)
    dfs(open_list,closed_list,goal)
initial=[[3,3,1],[0,0,0]]
move(initial)
closed_list.append(initial)
goal=[[0,0,0],[3,3,1]]
dfs(open_list,closed_list,goal)
print(closed_list)
```

OUTPUT:

```
reached
[[[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,
1, 0], [0, 2, 1]], [[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]]]
```

4. Implement Vacuum World problem with Search tree generation using

a. <u>BFS</u>

SOURCE CODE:

```
from queue import Queue
# Initializing a queue
queue = Queue() c=0
visited=[[0,0,0,0],[0,0,0,0],[0,0,0,0]]
mat=[[1,0,0,0],[0,1,0,1],[1,0,1,1]]
queue.put((0,0)) while(not
queue.empty()):
 i,j=queue.get()
 #print(i)
 #print(j)
 if(i<0 or j<0 or i>=len(mat) or
 j>=len(mat)):#print("entered") continue if(visited[i][j]):
  #print("entered")
  continue
 visited[i][j]=1
 if(mat[i][j]==0)
 : c+=1 else:
  c+=2
 mat[i][j]=0
 print(mat)
 queue.put((i+1,j))
```

```
  queue.put((i,j+1))
  queue.put((i-1,j))
  queue.put((i,j-1))
print(c-1)
```

OUTPUT:

```
[[0, 0, 0, 0], [0, 1, 0, 1], [1, 0, 1, 1]]
[[0, 0, 0, 0], [0, 1, 0, 1], [1, 0, 1, 1]]
[[0, 0, 0, 0], [0, 1, 0, 1], [1, 0, 1, 1]]
[[0, 0, 0, 0], [0, 1, 0, 1], [0, 0, 1, 1]]
[[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 1]]
[[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 1]]
[[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 1]]
[[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 1]]
[[0, 0, 0, 0], [0, 0, 0, 1], [0, 0, 0, 1]]
12
```

b. DFS

SOURCE CODE:

```
c=0
visited=[[0,0,0,0],[0,0,0,0],[0,0,0,0]]
def dfs(mat,i,j):
  if(i<0 or j<0 or i>=len(mat) or j>=len(mat[0])):
    return
  if(visited[i][j]):
  return
  visited[i][j]=1
  global c
  if(mat[i][j]==0):
  c+=1 else:
    c+=2
  mat[i][j]=0
  print(mat)
  dfs(mat,i+1,j)
  dfs(mat,i,j+1)
  dfs(mat,i-1,j)
  dfs(mat,i,j-1)
mat=[[1,0,0,0],[0,1,0,1],[1,0,1,1]]
dfs(mat,0,0) print(c-1)
```

OUTPUT:

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

```
[[0, 0, 0, 0], [0, 1, 0, 1], [1, 0, 1, 1]]
[[0, 0, 0, 0], [0, 1, 0, 1], [0, 0, 1, 1]]
[[0, 0, 0, 0], [0, 1, 0, 1], [0, 0, 1, 1]]
[[0, 0, 0, 0], [0, 1, 0, 1], [0, 0, 0, 1]]
[[0, 0, 0, 0], [0, 1, 0, 1], [0, 0, 0, 0]]
[[0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0]]
[[0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0]]
[[0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0]]
[[0, 0, 0, 0], [0, 1, 0, 0], [0, 0, 0, 0]]
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
17
```

5. Implement the following

a. <u>Greedy Best First Search</u>

SOURCE CODE:

```
from queue import PriorityQueue def
greedybestfirstsearch(gr,goal,heuristic):
   open_list=PriorityQueue()
   closed_list=[]
   open_list.put((8,'A',0)) while
   open_list.empty()==False:
   heu_cost,now,cost=open_list.get() if
   now==goal:
         print("min cost required is",cost)
         break
      closed_list.append(now)
      for edge in gr[now].keys():
         if edge not in closed_list:
            open_list.put((heuristic[edge],edge,cost+gr[now][edge]))
   print(closed_list)
graph={'A':{'B':1,'C':2},'B':{'D':7,'E':9,'F':4},'C':{'G':4,'H':3,'I':6,'J':8},'H':{'I':1}}
heuristic={'A':8,'B':10,'C':4,'D':15,'E':14,'F':12,'G':7,'H':2,'I':0,'J':4}
greedybestfirstsearch(graph,'I',heuristic)
```

OUTPUT:

```
min cost required is 8
['A', 'C']
```

b. A* algorithm

SOURCE CODE:

```
from queue import PriorityQueue
def astar(gr,goal,heuristic):
open_list=PriorityQueue()
closed_list=[]
open_list.put((8,'A',0)) while
open_list.empty()==False:
        heu_cost,now,cost=open_list.get()
        if now==goal:
            print("min cost required is",cost)
            break
        closed_list.append(now)
        for edge in gr[now].keys():
            if edge not in closed_list:
    open_list.put((heuristic[edge]+cost+gr[now][edge],edge,cost+gr[now][edge]))
    print(closed_list)
graph={'A':{'B':1,'C':2},'B':{'D':7,'E':9,'F':4},'C':{'G':4,'H':3,'I':6,'J':8},'H':{'I':1}}
heuristic={'A':8,'B':10,'C':4,'D':15,'E':14,'F':12,'G':7,'H':2,'I':0,'J':4}
astar(graph,'I',heuristic)
```

OUTPUT:

```
min cost required is 6
['A', 'C', 'H']
```

6. Implement 8-puzzle problem using A* algorithm

SOURCE CODE:

```
from queue import PriorityQueue
initial=[[1,2,3],[5,6,0],[7,8,4]]
goal=[[1,2,3],[5,8,6],[0,7,4]]          cost=0
open_list=PriorityQueue()   closed_list=[]
import copy def heuristic(now): cnt=0 for
i   in   range(len(now)):   for   j   in
range(len(now)):  if  now[i][j]!=0  and
now[i][j]!=goal[i][j]:
    cnt+=1
 return cnt+cost
def up(nw,i,j):
 now=copy.deepcopy(nw)
 temp=now[i-1][j] now[i-
 1][j]=now[i][j] now[i][j]=temp
 open_list.put((heuristic(now),now))
def down(nw,i,j):
 #print("down")
 now=copy.deepcopy(nw)
 temp=now[i+1][j]
 now[i+1][j]=now[i][j]
 now[i][j]=temp
 open_list.put((heuristic(now),n
 ow))
```

```
def left(nw,i,j):
  #print("left")
  now=copy.deepcopy(nw)
  temp=now[i][j-1] now[i][j-
  1]=now[i][j] now[i][j]=temp
  open_list.put((heuristic(now),now)
  )
def right(nw,i,j):
  now=copy.deepcopy(nw)
  temp=now[i][j+1]
  now[i][j+1]=now[i][j]
  now[i][j]=temp
  open_list.put((heuristic(now),now))
def      possibleiter(now):
  global   cost   cost=cost+1
  for  i  in  range(len(now)):
  for j in range(len(now)):
    if now[i][j]==0:
       x=i y=j if x==0
  and y==0:
  right(now,x,y)
  down(now,x,y)
  if x==0 and y==1:
    left(now,x,y)
```

```
  right(now,x,y)
  down(now,x,y)
if x==0 and y==2:
  left(now,x,y)
  down(now,x,y)
if x==1 and y==0:
  right(now,x,y)
  down(now,x,y)
  up(now,x,y)
if x==1 and y==1:
  left(now,x,y)
  right(now,x,y)
  down(now,x,y)
  up(now,x,y)
if x==1 and y==2:
  down(now,x,y)
  up(now,x,y)
  left(now,x,y)
if x==2 and y==0:
right(now,x,y)
up(now,x,y) if
x==2 and y==1:
left(now,x,y)
right(now,x,y)
up(now,x,y)
```

```
  if x==2 and y==2:
    left(now,x,y)
    up(now,x,y)
def astar(goal):
    open_list.put((0,initial))     while
    open_list.empty()==False:
    heu_cost,now=open_list.get()  if
    now==goal:
           print("min cost required is",cost)
           break
       closed_list.append(now)
       possibleiter(now)
astar(goal)
print(closed_list)
```

OUTPUT:

min cost required is 3

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

7. Implement AO* algorithm for General graph problem

SOURCE CODE:

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

        # ADD TO PATH FOR OR PATH

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

Conditions = {

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

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

'C': {'OR': ['G'], 'AND': ['H', 'I']},
```

```
'D': {'OR': ['J']}
}
# weight weight = 1 #
Updated                cost
print('Updated Cost :')
Updated_cost = update_cost(H, Conditions, weight=1)
print('*'*75)
print('Shortest Path :\n',shortest_path('A', Updated_cost,H))
```

OUTPUT:

```
Updated Cost :
D : {'OR': ['J']} >>> {'J': 1}
C : {'OR': ['G'], 'AND': ['H', 'I']} >>> {'H AND I': 2, 'G': 4}
B : {'OR': ['E', 'F']} >>> {'E OR F': 8}
A : {'OR': ['B'], 'AND': ['C', 'D']} >>> {'C AND D': 5, 'B': 9}
***************************************************************************
Shortest Path :
A<--(C AND D) [C<--(H AND I) [H + I] + D<--J]
```

8. Implement Game trees using

a. <u>MINIMAX algorithm</u>

SOURCE CODE:

```
import math def minimax (curDepth,maxTurn,
scores,targetDepth): if (curDepth == targetDepth):
if(maxTurn):
      return max(scores)
    else:
      return min(scores)
  if (maxTurn):
    maxnow=float('-inf')
    for i in scores:
      maxnow=max(maxnow,minimax(curDepth + 1,False, i, targetDepth))
    return maxnow
  else:
    minnow=float('inf')
    for i in scores:
      minnow=min(minnow,minimax(curDepth + 1,True, i, targetDepth))
    return minnow
scores = [[[2,3],[4,5],[6,7]],[[8,9],[10,11]]]
treeDepth = 3
print("The optimal value is : ", end = "")
print(minimax(1, True, scores, treeDepth))
```

OUTPUT:

The optimal value is : 9

b. Alpha-Beta pruning

SOURCE CODE:

```
import math path=[] def minimax (curDepth,maxTurn,
scores,targetDepth,alpha,beta): if (curDepth == targetDepth):
if(maxTurn):
      path.append(scores)
      return max(scores)
    else:
      path.append(scores)
      return min(scores)
  if              (maxTurn):
    maxnow=float('-
    inf') for i in scores:
      maxnow=max(maxnow,minimax(curDepth        +        1,False,        i,
targetDepth,alpha,beta)) alpha=max(alpha,maxnow) if beta<=alpha: break
    return maxnow
  else:
    minnow=float('inf')
    for i in scores:
      minnow=min(minnow,minimax(curDepth         +        1,True,         i,
targetDepth,alpha,beta)) beta=min(beta,minnow)
      if beta<=alpha:
        break
    return minnow
```

```
scores = [[[8,9],[10,11]],[[4,5],[6,7],[2,3]]]
treeDepth = 3
print("The optimal value is : ", end = "")
alpha=float('-inf')
beta=float('inf')
print(minimax(1, True, scores, treeDepth,alpha,beta))
print(path)
```

OUTPUT:

The optimal value is : 9

[[8, 9], [10, 11], [4, 5]]

9. 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:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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