8 PUZZLE PROBLEM

To design a graph search agent and understand the use of a hash table, queue in state space search.

Ankur Singh
MTECH CSE (2020-22)
IIIT Vadodara
Gandhinagar, India
202061001@iiitvadodara.ac.in

Anand Mundhe
MTECH CSE (2020-22)
IIIT Vadodara
Gandhinagar, India
202061006@iiitvadodara.ac.in

I. Problem Statement

A. Write a pseudocode for a graph search agent. Represent the agent in the form of a flow chart. Clearly mention all the implementation details with reasons.

B. Write a collection of functions imitating the environment for Puzzle-8.

C. Describe what is Iterative Deepening Search.

D. Considering the cost associated with every move to be the same (uniform cost), write a function which can backtrack and produce the path taken to reach the goal state from the source/ initial state.

E. Generate Puzzle-8 instances with the goal state at depth "d".

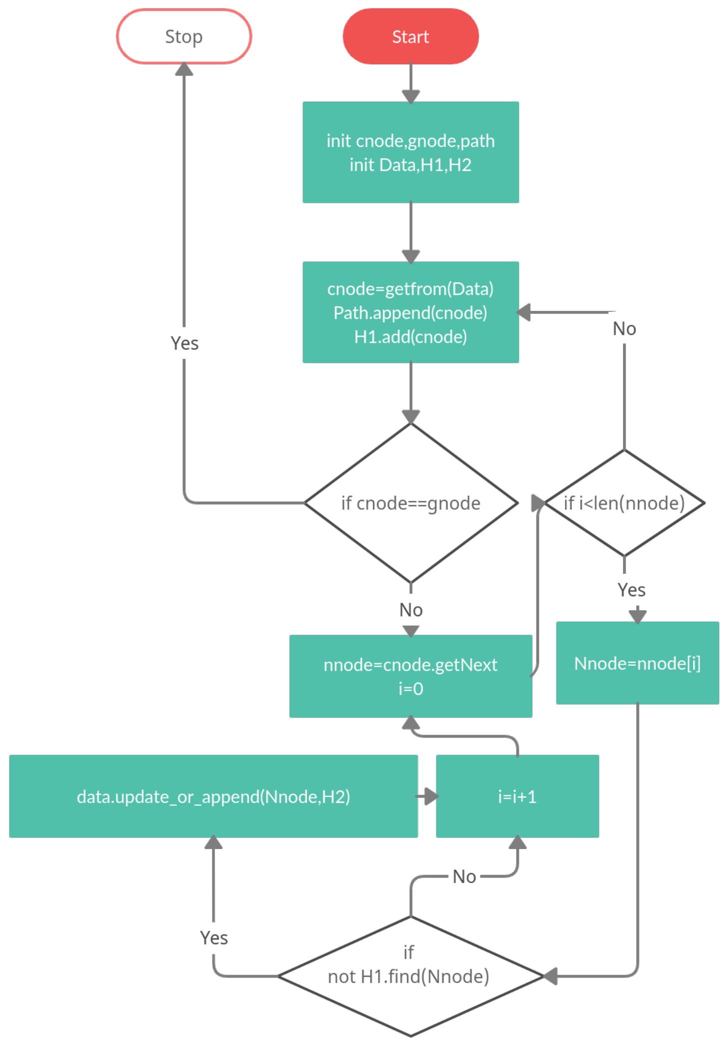
F. Prepare a table indicating the memory and time requirements to solve Puzzle-8 instances (depth "d") using your graph search agent.

II. Solutions

1) Write a pseudo code for a graph search agent. Represent the agent in the form of a flow chart. Clearly mention all the implementation details with reasons.

   cnode: Pointer to current node.
   gnode: Pointer to goal node.
   Path: Contain path from root to goal.
   Data: Data storage(Queue,Stack)(initiallized with root).
   H1: To track explored node(Hash).
   H2: To track to explore nodes(Hash).
   getfrom(Data): Gets one node out from data.
   Path.append(Node): Append one node in path.
   H1.add(Node): Adds node in hash table.
   cnode.getNext: List of all next possible states.
   H1.find(Node): Return true when Node is in hash table.
   data.update_or_append(Node,H): Update node using H when Node.cost<Data.node.cost, else just add to data.

Fig. 1. Graph search agent



2) Write a collection of functions imitating the environment for Puzzle-8.

```
class Node:
    #Initilise the node
    def __init__(self,puzzle,tmp=0):
        self.children=list()
        self.parent=None
        self.puzzle=list()
        self.x=0
        self.setPuzzle(puzzle)
        self.tmp=tmp
        self.next=None
        #Children: contain nodes to children
        #Parent points to the parent of current node
        #Puzzle contain state
        #X contain location of blank
        #tmp contain cost to node from source
        #next used in hashing in case 2 nodes have same hashing
    #Inserts state into self state from given state
    def setPuzzle(self,puzzle):
        for i in range(len(puzzle)):
            self.puzzle.append(puzzle[i])
    #Check if current state is goal state
    def goalTest(self):
        return self.puzzle==goal
    #Perform right move
    def moveToRight(self,p,i):
        #p is state
        #i is current blank place
        if i%3<2:#Check if not right most
            pc=[]
            self.copyPuzzle(pc,p)

            temp=pc[i+1]
            pc[i+1]=pc[i]
            pc[i]=temp

            child=Node(pc,self.tmp+1)
            self.children.append(child)
            child.parent=self
    #Perform left move
    def moveToLeft(self,p,i):
        if i%3>0:
            pc=[]
            self.copyPuzzle(pc,p)

            temp=pc[i-1]
            pc[i-1]=pc[i]
            pc[i]=temp

            child=Node(pc,self.tmp+1)
            self.children.append(child)
            child.parent=self
    #Perform up move
    def moveToUp(self,p,i):
        if i-3>=0:
            pc=[]
            self.copyPuzzle(pc,p)

            temp=pc[i-3]
            pc[i-3]=pc[i]
            pc[i]=temp

            child=Node(pc,self.tmp+1)
            self.children.append(child)
            child.parent=self
    #Perform down move
    def moveToDown(self,p,i):
        if i+3<len(self.puzzle):
            pc=[]
            self.copyPuzzle(pc,p)

            temp=pc[i+3]
            pc[i+3]=pc[i]
            pc[i]=temp

            child=Node(pc,self.tmp+1)
            self.children.append(child)
            child.parent=self
    #Print puzzle
    def printPuzzle(self):
        for i in range(len(self.puzzle)):
            if i%3==0:
                print('\n')
            print(self.puzzle[i],end=" ")
    #Return true when p is same as current state
    def isSamePuzzle(self,p):
        samePuzzle=True
        for i in range(len(p)):
            if self.puzzle[i]!=p[i]:
                samePuzzle=False
        return samePuzzle
    #Explore current node
    def expandNode(self):
        #get position of blank tile
        for i in range(len(self.puzzle)):
            if self.puzzle[i]==0:
                self.x=i
        #make move for them
        self.moveToRight(self.puzzle,self.x)
        self.moveToLeft(self.puzzle,self.x)
        self.moveToUp(self.puzzle,self.x)
        self.moveToDown(self.puzzle,self.x)
    #Copy contents from b to a
    def copyPuzzle(self,a,b):
        for i in range(len(b)):
            a.append(b[i])
#PriorityQueue is used to get smallest always
from queue import PriorityQueue
#To return first element of given parameter
def data(n):
        return n[0]
#appends path from n to root in path
def pathTrace(path,n):
        print("\nTracing path...")
        current=n
        path.append(current)
        while(current.parent!=None):
            current=current.parent
            path.append(current)
#Heuristic function
def heuristic_for_misplaced_tiles(p):
    cost=0
    for i in range(len(p)):
        if p[i]!=goal[i] and p[i]!=0:
            cost+=1
    return cost
#BestFirst search
class BestFirst:
    #Algo starts here
    def bestfirst(self,root):
        pathToSolution=[]
        #To Explore and explored
        openList=[]
        closedList=[]
        #Append root in to explore
        openList.append((0,root))
        #Goal flag
        goalFound=False
        #Till toExplore is not empty or goal not found
        while len(openList)!=0 and goalFound==False:
```

```python
            #Sort to explore list as per data attribute
            openList=sorted(openList,key=data)
            #get minimum
            _,currentNode=openList[0]
            #Append to explored
            closedList.append(currentNode)
            #Remove from to explore
            openList.pop(0)
            #Explore current node
            currentNode.expandNode()
            #stores cost,node pair
            mis=[]
            for i in range(len(currentNode.children)):
                #append child node with its cost from goal node to children node
                mis.append((heuristic_for_misplaced_tiles(currentNode.children[i].puzzle),currentNode.children[i]))
            #Process all children of explored node
            for i in range(len(currentNode.children)):
                #Take ith child
                currentChild=currentNode.children[i]
                #Check if its a goal
                if currentChild.goalTest():
                    #Trace path from current node to root and append it in pathToSolution
                    print("\nGoal Found")
                    goalFound=True
                    pathTrace(pathToSolution,currentChild)
                    break
                #If child node is not yet explored or in to explore then add it to, to explore list
                if(self.contains(openList,currentChild,"gg")==False and self.contains(closedList,currentChild,"ff")==False):
                    openList.append(mis[i])
        #Return path from goal to root(Reverse)
        return pathToSolution
    #Check if lis(list) contains c(Node) with s(mode)
    def contains(self,lis,c,s):
        #Flag
        contains=False
        for i in range(len(lis)):
            #open list have cost and node
            if s=="gg":
                _,l=lis[i]
            #closed list only have node
            else:
                l=lis[i]
            #Return true when l and c have same puzzle(State) attribute
            if l.isSamePuzzle(c.puzzle)==True:
                contains=True
        #Return flag
        return contains
#Execution starts here
#Source
puzzle=[1,2,4,3,0,5,7,6,8]
#Goal
goal=[0,1,2,3,4,5,6,7,8]
root=Node(puzzle)
ui=BestFirst()
solution=ui.bestfirst(root)
#Solution is from goal to root so reverse it
solution.reverse()
if len(solution)>0:
    for i in range(len(solution)):
        solution[i].printPuzzle()
        if i<len(solution)-1:
            print("\n\n******Next move**********",i)
else:
    print("\nNo path to solution is found")
```

Listing 1. Collection of functions for envirnoment for puzzle-8

Output steps using BestFirst

```

Goal Found

Tracing path...
1 2 4
3 0 5
7 6 8

******Next move********** 0
1 2 4
3 5 0
7 6 8

******Next move********** 1
1 2 0
3 5 4
7 6 8

******Next move********** 2
1 0 2
3 5 4
7 6 8

******Next move********** 3
0 1 2
3 5 4
7 6 8

******Next move********** 4
3 1 2
0 5 4
7 6 8

******Next move********** 5
3 1 2
5 0 4
7 6 8

******Next move********** 6
3 1 2
5 6 4
7 0 8

******Next move********** 7
3 1 2
5 6 4
0 7 8

******Next move********** 8
3 1 2
0 6 4
5 7 8

******Next move********** 9
3 1 2
6 0 4
5 7 8

******Next move********** 10
3 1 2
6 4 0
```

```
5 7 8

******Next move********** 11
3 1 2
6 4 8
5 7 0

******Next move********** 12
3 1 2
6 4 8
5 0 7

******Next move********** 13
3 1 2
6 4 8
0 5 7

******Next move********** 14
3 1 2
0 4 8
6 5 7

******Next move********** 15
3 1 2
4 0 8
6 5 7

******Next move********** 16
3 1 2
4 5 8
6 0 7

******Next move********** 17
3 1 2
4 5 8
6 7 0

******Next move********** 18
3 1 2
4 5 0
6 7 8

******Next move********** 19
3 1 2
4 0 5
6 7 8

******Next move********** 20
3 1 2
0 4 5
6 7 8

******Next move********** 21
0 1 2
3 4 5
6 7 8
```

3) Describe what is Iterative Deepening Search.

We have BFS but it takes lot of memory to execute, but will give solution no matter what.
Then we have DFS which require less memory to execute, but it may or may not give solution.
Idea is to combine BFS algorithm's reliability and DFS algorithm's memory consumption. In iterative deepening search, we do level wise DFS.
First perform DFS for level 0 which is root and check if it is matching to our goal, if not then we just increase level to 1.
Perform the DFS up to level 1, when next node is out of graph then we stop. [2]

Lets take one example:

Suppose our goal is n10 node, At start we go with level 0 DFS, where we encounter only n1 and we check if n1 is equal to goal(n10), which is not so we go to level level 1.

At level 1 we encounter 4 nodes which are n1,n2,n3 and n4, so apply DFS on n1, which is to compare if n1 is goal then n2, n3 and n4 respectively. As no goal is found so we go to next level.

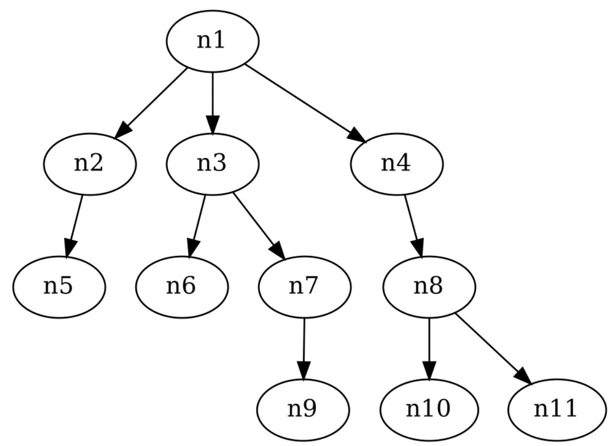
For level 2 we apply DFS on n1, so comparison sequence will be like this, n1,n2,n5,n3,n6,n7,n4,n8. Here also we dont find goal so head to next level.

For level 3 we apply DFS on n1, so comparison sequence will be like this, n1,n2,n5,n3,n6,n7,n9,n4,n8,n10. Here we find n10 which is goal state so we can stop here.

Advantages:

1. Always finds optimal solution as it scans by level.
2. Less memory required as its a level wise DFS.

Fig. 2. Tree for example



4) Considering the cost associated with every move to be the same (uniform cost), write a function which can backtrack and produce the path taken to reach the goal state from the source/ initial state.

```python
class Node:
    #Initilise the node
    def __init__(self,puzzle,tmp=0):
        self.children=list()
        self.parent=None
        self.puzzle=list()
        self.x=0
        self.setPuzzle(puzzle)
        self.tmp=tmp
        self.next=None
        #Children: contain nodes to children
        #Parent points to the parent of current node
        #Puzzle contain state
        #X contain location of blank
        #tmp contain cost to node from source
        #next used in hashing in case 2 nodes have same hashing
    #Inserts state into self state from given state
    def setPuzzle(self,puzzle):
        for i in range(len(puzzle)):
            self.puzzle.append(puzzle[i])
    #Check if current state is goal state
    def goalTest(self):
        return self.puzzle==goal
    #Perform right move
    def moveToRight(self,p,i):
        #p is state
        #i is current blank place
        if i%3<2:#Check if not right most
            pc=[]
            self.copyPuzzle(pc,p)

            temp=pc[i+1]
            pc[i+1]=pc[i]
            pc[i]=temp

            child=Node(pc,self.tmp+1)
            self.children.append(child)
            child.parent=self
    #Perform left move
    def moveToLeft(self,p,i):
        if i%3>0:
            pc=[]
            self.copyPuzzle(pc,p)

            temp=pc[i-1]
            pc[i-1]=pc[i]
            pc[i]=temp

            child=Node(pc,self.tmp+1)
            self.children.append(child)
            child.parent=self
    #Perform up move
    def moveToUp(self,p,i):
        if i-3>=0:
            pc=[]
            self.copyPuzzle(pc,p)

            temp=pc[i-3]
            pc[i-3]=pc[i]
            pc[i]=temp

            child=Node(pc,self.tmp+1)
            self.children.append(child)
            child.parent=self
    #Perform down move
    def moveToDown(self,p,i):
        if i+3<len(self.puzzle):
            pc=[]
            self.copyPuzzle(pc,p)

            temp=pc[i+3]
            pc[i+3]=pc[i]
            pc[i]=temp

            child=Node(pc,self.tmp+1)
            self.children.append(child)
            child.parent=self
    #Print puzzle
    def printPuzzle(self):
        for i in range(len(self.puzzle)):
            if i%3==0:
                print('\n')
            print(self.puzzle[i],end=" ")
    #Return true when p is same as current state
    def isSamePuzzle(self,p):
        samePuzzle=True
        for i in range(len(p)):
            if self.puzzle[i]!=p[i]:
                samePuzzle=False
        return samePuzzle
    #Explore current node
    def expandNode(self):
        #get position of blank tile
        for i in range(len(self.puzzle)):
            if self.puzzle[i]==0:
                self.x=i
        #make move for them
        self.moveToRight(self.puzzle,self.x)
        self.moveToLeft(self.puzzle,self.x)
        self.moveToUp(self.puzzle,self.x)
        self.moveToDown(self.puzzle,self.x)
    #Copy contents from b to a
    def copyPuzzle(self,a,b):
        for i in range(len(b)):
            a.append(b[i])

#For hashing
from collections import OrderedDict
dict=OrderedDict
#Get all nodes in path from n to root node
def pathTrace(path,n):
        current=n
        path.append(current)
        while(current.parent!=None):
            current=current.parent
            path.append(current)
#Class to perform uniform search
class uninformedSearch:
    #Hashing function
    def hashing(self,x):
        hash_val=0
        # simply sum of index*state
        for i in range(len(x)):
            hash_val+=(i+1)**x[i]
        return hash_val
    #Perform uniform search
    def bfs(self,root):
        #No of explored nodes
        explored=0
        #No of children faced
        child=0
        #No of children put in memory at start root is in memory
        childm=1
        #Maximum memory needed for program
        maxm=childm
        #Stored solution path here
        pathToSolution=[]
```

```
        #2 hash tables (Dictionary)
        #To explore
        openList=dict()
        #Explored
        closedList=dict()
        #Adding root to to explore list
        openList[self.hashing(root.puzzle)]=root
        #Falg if goal found
        goalFound=False
        #Check if current is goal state and print it
        if root.goalTest():
            print("\nGoal Found")
            goalFound=True
            #Get path from current node(Goal) to source
            pathTrace(pathToSolution,root)
            return pathToSolution,explored,child,childm
        #Stop when there is nothing to explore or goal is found
        while len(openList)!=0 and goalFound==False:
            #Get one node from list which contain node to explore
            _,currentNode=openList.popitem(last=False)
            #One node is out of memory
            childm-=1
            #Get hast value for that one node
            chash=self.hashing(currentNode.puzzle)
            #Mark node as explored as this node is going to be explored
            #It may have same hash also so check next
            if closedList.get(chash)!=None:
                tmp1=closedList.get(chash)
                while tmp1.next!=None:tmp1=tmp1.next
                tmp1.next=currentNode
            else:closedList[chash]=currentNode
            #Explore the node
            currentNode.expandNode()
            #Increament counter for explored
            explored+=1
            #For all childrens check goal state
            for i in range(len(currentNode.children)):
                #take ith child
                currentChild=currentNode.children[i]
                #Child is faced in process so increment counter
                child+=1
                #Check if this is goal state
                if currentChild.goalTest():
                    goalFound=True
                    #Get path to root
                    pathTrace(pathToSolution,currentChild)
                    break
                #Check if node in openList or closedList
                if(self.con(openList,currentChild,"open")==False and self.con(closedList,currentChild,"closed")==False):
                    #If not then just append it to open list
                    openList[self.hashing(currentChild.puzzle)]=currentChild
                    #So its added to open list so memory++
                    childm+=1
                    #Update maxm required
                    if childm>maxm:maxm=childm
        return pathToSolution,explored,child,maxm
        #pathToSolution: contain path from goal to root
        #explored: Number of nodes explored
        #child: Number of child faced
        #maxm: Maximum memory(Nodes) required to run code
    #Check if lis(HashTable) contains c(CurrentNode) with s(MODE)
    def con(self,lis,c,s):
        #Check if node is present in dictionary for hash value
        if lis.get(self.hashing(c.puzzle)) != None:
            #Check if node at hash value is same
            if lis[self.hashing(c.puzzle)].isSamePuzzle(c.puzzle):
                #Update cost to node if needed and in open list
                if s=="open" and lis[self.hashing(c.puzzle)].tmp>c.tmp:
                    lis[self.hashing(c.puzzle)].tmp=c.tmp
                    lis[self.hashing(c.puzzle)].parent=c.parent
                    lis[self.hashing(c.puzzle)].children=c.children
                #Node is alredy in list, so true
                return True
            #if same hash for 2 or more nodes
            else:
                #temporar pointer
                tmp1=lis[self.hashing(c.puzzle)]
                #Check all pointers
                while tmp1.next!=None:
                    tmp1=tmp1.next
                    if tmp1.isSamePuzzle(c.puzzle):
                        #Update cost to node if needed and in open list
                        if s=="open" and tmp1.tmp>c.tmp:
                            tmp1.tmp=c.tmp
                            tmp1.parent=c.parent
                            tmp1.children=c.children
                        #Node is alredy in list, so true
                        return True
                if(s=="open"):
                    tmp1.next=c
                    return True
        #False otherwise
        return False

#Execution starts here
#My randoly generated state
puzzle=[1,2,4,3,0,5,7,6,8]
#My goal state
goal=[0,1,2,3,4,5,6,7,8]
root=Node(puzzle)
ui=uninformedSearch()
#Call uniform search
solution,explored,child,childm=ui.bfs(root)
#Reverse it as solution is reverse(from goal to source)
```

```
solution.reverse()
#Print solution
if len(solution)>0:
    for i in range(len(solution)):
        print("\n\n\nMove",i)
        solution[i].printPuzzle()
        if i<len(solution)-1:
            print("\n\n******Next move
    *********")
    print("\n\n\nDepth of solution is : ",
    solution[-1].tmp)
    print("\nTotal moves required : ",len(
    solution)-1)
    print("\nTotal nodes explored : ",explored)
    print("\nTotal childrens : ",child)
    print("\nTotal children in memory at once :
     ",childm)
else:
    print("\nNo path to solution is found")
```

Listing 2. 8-puzzle using Uniform Search

Output steps using Uniform

```
Move 0
1 2 4
3 0 5
7 6 8

******Next move*********
Move 1
1 2 4
3 6 5
7 0 8

******Next move*********
Move 2
1 2 4
3 6 5
7 8 0

******Next move*********
Move 3
1 2 4
3 6 0
7 8 5

******Next move*********
Move 4
1 2 0
3 6 4
7 8 5

******Next move*********
Move 5
1 0 2
3 6 4
7 8 5

******Next move*********
Move 6
0 1 2
3 6 4
7 8 5

******Next move*********
Move 7
3 1 2
0 6 4
7 8 5

******Next move*********
Move 8
3 1 2
6 0 4
7 8 5

******Next move*********
Move 9
3 1 2
6 4 0
7 8 5

******Next move*********
Move 10
3 1 2
6 4 5
7 8 0

******Next move*********
Move 11
3 1 2
6 4 5
7 0 8

******Next move*********
Move 12
3 1 2
6 4 5
0 7 8

******Next move*********
Move 13
3 1 2
0 4 5
6 7 8

******Next move*********
Move 14
0 1 2
3 4 5
6 7 8

Depth of solution is :  14
Total moves required :  14
Total nodes explored :  3457
Total childrens :  9562
Total children in memory at once :  2163
```

5) Generate Puzzle-8 instances with the goal state at depth “d”.

```python
#Same as before but return nodes at given depth
class configurations:
    #For hashing
    def hashing(self,x):
        hash_val=0
        for i in range(len(x)):
            hash_val+=(i+1)**x[i]
        return hash_val
    #To get nodes at depth depth
    def dbfs(self,root,depth):
        #root: Root node
        #depth: Depth at which to return nodes
        explored=0
        child=0
        childm=0
        maxm=0
        #This stores nodes which have required depth
        solutions=[]
        pathToSolution=[]
        openList=dict()
        closedList=dict()
        openList[self.hashing(root.puzzle)]=root
        goalFound=False
        #Check if current depth of root is same as required
        if root.tmp==depth:
            goalFound=True
            #Node is found, return path to it from source
            pathTrace(pathToSolution,root)
            solutions.append(pathToSolution)
            return solutions
        while len(openList)!=0 and goalFound==False:
            _,currentNode=openList.popitem(last=False)
            chash=self.hashing(currentNode.puzzle)
            if closedList.get(chash)!=None:
                tmp1=closedList.get(chash)
                while tmp1.next!=None:tmp1=tmp1.next
                tmp1.next=currentNode
            else:closedList[chash]=currentNode
            currentNode.expandNode()
            explored+=1
            for i in range(len(currentNode.children)):
                currentChild=currentNode.children[i]
                #Skip nodes which are above depth
                if currentChild.tmp>depth:
                    goalFound=True
                    #Go out as we got what we needed
                    break
                child+=1
                if(self.con(openList,currentChild,"open")==False and self.con(closedList,currentChild,"closed")==False):
                    openList[self.hashing(currentChild.puzzle)]=currentChild
                    childm+=1
                    if childm>maxm:maxm=childm
                    #If node matches with depth then add it to solutions
                    if currentChild.tmp==depth:
                        pathTrace(pathToSolution,currentChild)
                        solutions.append(pathToSolution)
                        pathToSolution=[]

        return solutions
    #Check if lis(HashTable) contains c(CurrentNode) with s(MODE)
    def con(self,lis,c,s):
        #Check if node is present in dictionary for hash value
        if lis.get(self.hashing(c.puzzle)) != None:
            #Check if node at hash value is same
            if lis[self.hashing(c.puzzle)].isSamePuzzle(c.puzzle):
                #Update cost to node if needed and in open list
                if s=="open" and lis[self.hashing(c.puzzle)].tmp>c.tmp:
                    lis[self.hashing(c.puzzle)].tmp=c.tmp
                    lis[self.hashing(c.puzzle)].parent=c.parent
                    lis[self.hashing(c.puzzle)].children=c.children
                #Node is alredy in list, so true
                return True
            #if same hash for 2 or more nodes
            else:
                #temporar pointer
                tmp1=lis[self.hashing(c.puzzle)]
                #Check all pointers
                while tmp1.next!=None:
                    tmp1=tmp1.next
                    if tmp1.isSamePuzzle(c.puzzle):
                        #Update cost to node if needed and in open list
                        if s=="open" and tmp1.tmp>c.tmp:
                            tmp1.tmp=c.tmp
                            tmp1.parent=c.parent
                            tmp1.children=c.children
                        #Node is alredy in list, so true
                        return True
                if(s=="open"):
                    tmp1.next=c
                    return True
        #False otherwise
        return False

#Execution starts here
puzzle=[0,1,2,3,4,5,6,7,8]
d=8
show=10
root=Node(puzzle)
ui=configurations()
solutions=ui.dbfs(root,d)
print("Found",len(solutions),"solutions or configurations with depth",d,"\nShowing top :",show)
for i in range(len(solutions)):
    if len(solutions[i])>0:
        print("\n\nSolution",i+1,"with depth",d)
        solutions[i][0].printPuzzle()
    if i<len(solutions)-1 and i<show-1:
```

```
            print("\n\n******Next solution
    *********")
    if i>=show-1:break
```

Listing 3. Functions to generate 8 puzzle instances with depth d

Output top 10 solutions with depth 8

```
Found 114 solutions or configurations with
    depth 8
Showing top : 10
Solution 1 with depth 8
2 5 0
1 3 4
6 7 8

******Next solution*********
Solution 2 with depth 8
2 3 5
1 0 4
6 7 8

******Next solution*********
Solution 3 with depth 8
1 2 5
6 3 4
7 8 0

******Next solution*********
Solution 4 with depth 8
1 2 5
6 0 4
7 3 8

******Next solution*********
Solution 5 with depth 8
1 5 4
3 2 8
6 7 0

******Next solution*********
Solution 6 with depth 8
3 1 5
2 0 4
6 7 8

******Next solution*********
Solution 7 with depth 8
3 1 5
6 2 4
0 7 8

******Next solution*********
Solution 8 with depth 8
1 2 5
3 0 7
6 8 4

******Next solution*********
Solution 9 with depth 8
1 2 0
3 7 5
6 8 4

******Next solution*********
Solution 10 with depth 8
1 2 5
7 0 4
3 6 8
```

6) Prepare a table indicating the memory and time requirements to solve Puzzle-8 instances (depth “d”) using your graph search agent.

```
import time as t
import sys
node_size=sys.getsizeof(Node([0,1,2,3,4,5,6,7,8]))
#Tabular format
goal=[0,1,2,3,4,5,6,7,8]
print("\\begin{center}")
print("\\begin{tabular}{ |c|c|c| }")
print("Init state & Time(msec) & Memory\\\\")
count=0
#Solutions is list of instances of 8-puzzle
for i in solutions:
    count+=1
    tmp=t.time()
    puzzle=i[0].puzzle
    root=Node(puzzle)
    ui=uninformedSearch()
    solution,explored,child,childm=ui.bfs(root)
    #to devide in half
    if count==35:
        print("\\end{tabular}")
        print("\\end{center}")
        print("\\begin{center}")
        print("\\begin{tabular}{ |c|c|c| }")
        print("Init state & Time(msec) & Memory\\\\")
    if solution[0].tmp==d:
        print("text{",solution[-1].puzzle,"} & ",int((t.time()-tmp)*1000000)," & ",childm*node_size,end="\\\\\n")
print("\\end{tabular}")
print("\\end{center}")
```

Listing 4. To generate table of memory and time

Output Table

Init state	Time(msec)	Memory
text [2, 5, 0, 1, 3, 4, 6, 7, 8]	2992	3528
text [2, 3, 5, 1, 0, 4, 6, 7, 8]	7930	6384
text [1, 2, 5, 6, 3, 4, 7, 8, 0]	3051	3528
text [1, 2, 5, 6, 0, 4, 7, 3, 8]	7608	7112
text [1, 5, 4, 3, 2, 8, 6, 7, 0]	4118	5544
text [3, 1, 5, 2, 0, 4, 6, 7, 8]	4641	5208
text [3, 1, 5, 6, 2, 4, 0, 7, 8]	6667	6216
text [1, 2, 5, 3, 0, 7, 6, 8, 4]	4260	4760
text [1, 2, 0, 3, 7, 5, 6, 8, 4]	6371	6216
text [1, 2, 5, 7, 0, 4, 3, 6, 8]	5102	5712
text [1, 2, 5, 4, 0, 8, 3, 6, 7]	4903	5600
text [1, 2, 0, 3, 8, 5, 6, 4, 7]	6658	5824
text [1, 2, 5, 3, 8, 7, 6, 4, 0]	8205	5768
text [1, 2, 5, 6, 3, 8, 0, 4, 7]	5725	5880
text [1, 5, 0, 3, 2, 8, 6, 4, 7]	4883	4872
text [0, 1, 5, 3, 2, 8, 6, 4, 7]	4673	4816
text [3, 1, 4, 5, 0, 2, 6, 7, 8]	6357	5096
text [3, 1, 4, 6, 5, 2, 0, 7, 8]	6465	6048
text [0, 5, 4, 1, 3, 2, 6, 7, 8]	4750	5488
text [1, 5, 4, 6, 3, 2, 0, 7, 8]	4502	5488
text [1, 5, 4, 3, 7, 2, 6, 8, 0]	4515	4816

Init state	Time(msec)	Memory
text [4, 2, 5, 1, 3, 8, 6, 7, 0]	4483	6048
text [4, 3, 0, 1, 5, 2, 6, 7, 8]	3867	5208
text [4, 3, 2, 1, 5, 8, 6, 7, 0]	3949	5152
text [4, 3, 2, 1, 7, 5, 6, 8, 0]	3444	4480
text [4, 3, 2, 1, 7, 5, 0, 6, 8]	3935	4480
text [1, 4, 2, 6, 0, 3, 7, 8, 5]	4078	4592
text [1, 4, 0, 6, 3, 2, 7, 8, 5]	4493	6048
text [1, 4, 0, 6, 5, 2, 7, 3, 8]	4442	5880
text [1, 4, 2, 6, 5, 8, 7, 3, 0]	4406	5880
text [0, 4, 2, 1, 6, 5, 7, 3, 8]	4477	5824
text [1, 4, 2, 7, 6, 5, 0, 3, 8]	4132	4704
text [1, 2, 0, 6, 4, 5, 7, 3, 8]	3790	4984
text [0, 1, 2, 6, 4, 5, 7, 3, 8]	3658	4984
text [0, 4, 2, 1, 3, 7, 6, 8, 5]	4253	5376
text [1, 4, 2, 6, 3, 7, 0, 8, 5]	4060	5432
text [1, 2, 0, 3, 4, 7, 6, 8, 5]	3332	4312
text [0, 1, 2, 3, 4, 7, 6, 8, 5]	3947	4256
text [1, 4, 2, 3, 8, 7, 0, 6, 5]	3347	4312
text [0, 1, 4, 3, 7, 2, 6, 8, 5]	3113	3920
text [1, 7, 4, 3, 0, 2, 6, 8, 5]	5289	6328
text [1, 4, 0, 7, 5, 2, 3, 6, 8]	4036	5432
text [1, 4, 2, 7, 5, 8, 3, 6, 0]	4031	5432
text [1, 2, 0, 7, 4, 5, 3, 6, 8]	4111	4704
text [0, 1, 2, 7, 4, 5, 3, 6, 8]	3545	4704
text [1, 4, 2, 7, 6, 5, 3, 8, 0]	3550	4704
text [4, 2, 0, 1, 7, 5, 3, 6, 8]	3009	3920
text [4, 7, 2, 1, 0, 5, 3, 6, 8]	6476	6664
text [4, 3, 1, 5, 0, 2, 6, 7, 8]	5295	5040
text [4, 3, 1, 6, 5, 2, 0, 7, 8]	5234	5936
text [3, 5, 0, 4, 2, 1, 6, 7, 8]	3042	3976
text [3, 5, 1, 4, 2, 8, 6, 7, 0]	5394	5488
text [0, 5, 1, 3, 4, 2, 6, 7, 8]	4133	5432
text [3, 5, 1, 6, 4, 2, 0, 7, 8]	4157	5488
text [3, 5, 1, 4, 7, 2, 6, 8, 0]	3538	4760
text [3, 5, 1, 4, 7, 2, 0, 6, 8]	4099	4760
text [3, 1, 2, 5, 0, 8, 4, 6, 7]	5011	5488
text [0, 1, 2, 3, 5, 8, 4, 6, 7]	4488	5992
text [3, 1, 0, 4, 8, 2, 6, 5, 7]	4162	5712
text [3, 1, 2, 4, 8, 7, 6, 5, 0]	3286	4256
text [3, 1, 2, 6, 4, 8, 0, 5, 7]	5217	5768
text [3, 2, 0, 4, 1, 8, 6, 5, 7]	145050	4704
text [0, 3, 2, 4, 1, 8, 6, 5, 7]	6787	4760
text [0, 2, 5, 3, 4, 1, 6, 7, 8]	6798	5152
text [3, 2, 5, 6, 4, 1, 0, 7, 8]	6681	5152
text [0, 3, 5, 4, 2, 1, 6, 7, 8]	3854	4032
text [3, 2, 5, 4, 7, 1, 6, 8, 0]	5151	3976
text [3, 2, 5, 4, 7, 1, 0, 6, 8]	7434	4032
text [3, 2, 5, 4, 1, 8, 0, 6, 7]	8552	3808
text [3, 2, 5, 4, 0, 8, 6, 1, 7]	11505	6888
text [4, 3, 2, 6, 1, 5, 7, 8, 0]	9118	3752
text [4, 3, 2, 6, 0, 5, 7, 1, 8]	9634	7392
text [3, 1, 2, 6, 4, 7, 0, 8, 5]	9235	5376
text [3, 2, 0, 4, 1, 7, 6, 8, 5]	9645	4200
text [0, 3, 2, 4, 1, 7, 6, 8, 5]	5359	4256

REFERENCES

[1] Artificial Intelligence: a Modern Approach, Russell and Norvig (Fourth edition) Chapter 2 and 3

[2] Iterative Deepening Search(IDS) or Iterative Deepening Depth First Search(IDDFS) https://www.geeksforgeeks.org/iterative-deepening-searchids-iterative-deepening-depth-first-searchiddfs/