

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

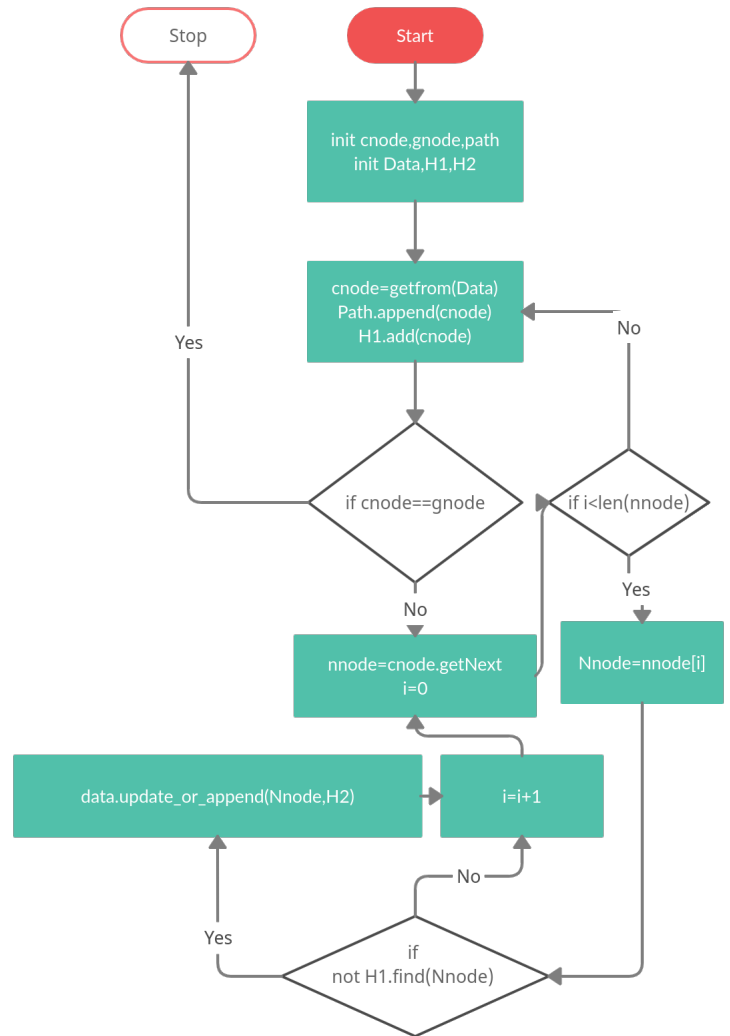
- Write a pseudocode for a graph search agent. Represent the agent in the form of a flow chart. Clearly mention all the implementation details with reasons.
- Write a collection of functions imitating the environment for Puzzle-8.
- Describe what is Iterative Deepening Search.
- Considering the cost 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.
- Generate Puzzle-8 instances with the goal state at depth “d”.
- Prepare a table indicating the memory and time requirements to solve Puzzle-8 instances (depth “d”) using your graph search agent.

II. SOLUTIONS

- 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)(initialized 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 $\text{Node.cost} < \text{Data.node.cost}$, else just add to data.

Fig. 1. Graph search agent



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

```

1 class Node:
2     #Initilise the node
3     def __init__(self,puzzle,tmp=0):
4         self.children=list()
5         self.parent=None
6         self.puzzle=list()
7         self.x=0
8         self.setPuzzle(puzzle)
9         self.tmp=tmp
10        self.next=None
11        #Children: contain nodes to children
12        #Parent points to the parent of current
13        node
14        #Puzzle contain state
15        #X contain location of blank
16        #tmp contain cost to node from source
17        #next used in hashing in case 2 nodes
18        have same hashing
19        #Inserts state into self state from given
20        state
21        def setPuzzle(self,puzzle):
22            for i in range(len(puzzle)):
23                self.puzzle.append(puzzle[i])
24            #Check if current state is goal state
25            def goalTest(self):
26                return self.puzzle==goal
27            #Perform right move
28            def moveToRight(self,p,i):
29                #p is state
30                #i is current blank place
31                if i%3<2:#Check if not right most
32                    pc=[]
33                    self.copyPuzzle(pc,p)
34
35                    temp=pc[i+1]
36                    pc[i+1]=pc[i]
37                    pc[i]=temp
38
39                    child=Node(pc,self.tmp+1)
40                    self.children.append(child)
41                    child.parent=self
42            #Perform left move
43            def moveToLeft(self,p,i):
44                if i%3>0:
45                    pc=[]
46                    self.copyPuzzle(pc,p)
47
48                    temp=pc[i-1]
49                    pc[i-1]=pc[i]
50                    pc[i]=temp
51
52                    child=Node(pc,self.tmp+1)
53                    self.children.append(child)
54                    child.parent=self
55            #Perform up move
56            def moveToUp(self,p,i):
57                if i-3>=0:
58                    pc=[]
59                    self.copyPuzzle(pc,p)
60
61                    temp=pc[i-3]
62                    pc[i-3]=pc[i]
63                    pc[i]=temp
64
65                    child=Node(pc,self.tmp+1)
66                    self.children.append(child)
67                    child.parent=self
68            #Perform down move
69            def moveToDown(self,p,i):
70                if i+3<len(self.puzzle):
71                    pc=[]

```

```

69        self.copyPuzzle(pc,p)
70
71        temp=pc[i+3]
72        pc[i+3]=pc[i]
73        pc[i]=temp
74
75        child=Node(pc,self.tmp+1)
76        self.children.append(child)
77        child.parent=self
78
79        #Print puzzle
80        def printPuzzle(self):
81            for i in range(len(self.puzzle)):
82                if i%3==0:
83                    print('\n')
84                    print(self.puzzle[i],end=" ")
85            #Return true when p is same as current
86            state
87            def isSamePuzzle(self,p):
88                samePuzzle=True
89                for i in range(len(p)):
90                    if self.puzzle[i]!=p[i]:
91                        samePuzzle=False
92                return samePuzzle
93            #Explore current node
94            def expandNode(self):
95                #get position of blank tile
96                for i in range(len(self.puzzle)):
97                    if self.puzzle[i]==0:
98                        self.x=i
99                #make move for them
100                self.moveToRight(self.puzzle,self.x)
101                self.moveToLeft(self.puzzle,self.x)
102                self.moveToUp(self.puzzle,self.x)
103                self.moveToDown(self.puzzle,self.x)
104            #Copy contents from b to a
105            def copyPuzzle(self,a,b):
106                for i in range(len(b)):
107                    a.append(b[i])
108            #PriorityQueue is used to get smallest always
109            from queue import PriorityQueue
110            #To return first element of given parameter
111            def data(n):
112                return n[0]
113            #appends path from n to root in path
114            def pathTrace(path,n):
115                print("\nTracing path...")
116                current=n
117                path.append(current)
118                while(current.parent!=None):
119                    current=current.parent
120                    path.append(current)
121            #Heuristic function
122            def heuristic_for_misplaced_tiles(p):
123                cost=0
124                for i in range(len(p)):
125                    if p[i]!=goal[i] and p[i]!=0:
126                        cost+=1
127                return cost
128            #BestFirst search
129            class BestFirst:
130                #Algo starts here
131                def bestfirst(self,root):
132                    pathToSolution=[]
133                    #To Explore and explored
134                    openList=[]
135                    closedList=[]
136                    #Append root in to explore
137                    openList.append((0,root))
138                    #Goal flag
139                    goalFound=False
140                    #Till toExplore is not empty or goal
141                    not found
142                    while len(openList)!=0 and goalFound==
143                    False:

```

```

140         #Sort to explore list as per data
attribute
141         openList=sorted(openList,key=data)
142         #get minimum
143         _, currentNode=openList[0]
144         #Append to explored
145         closedList.append(currentNode)
146         #Remove from to explore
147         openList.pop(0)
148         #Explore current node
149         currentNode.expandNode()
150         #stores cost,node pair
151         mis=[]
152         for i in range(len(currentNode.
children)):
153             #append child node with its
cost from goal node to children node
154             mis.append((
heuristic_for_misplaced_tiles(currentNode.
children[i].puzzle),currentNode.children[i
]))
155             #Process all children of explored
node
156             for i in range(len(currentNode.
children)):
157                 #Take ith child
158                 currentChild=currentNode.
children[i]
159                 #Check if its a goal
160                 if currentChild.goalTest():
161                     #Trace path from current
node to root and append it in
pathToSolution
162                     print("\nGoal Found")
163                     goalFound=True
164                     pathTrace(pathToSolution,
currentChild)
165                     break
166                     #If child node is not yet
explored or in to explore then add it to,
to explore list
167                     if(self.contains(openList,
currentChild,"gg")==False and self.contains
(closedList,currentChild,"ff")==False):
168                         openList.append(mis[i])
169                     #Return path from goal to root(Reverse)
170                     return pathToSolution
171             #Check if lis(list) contains c(Node) with s
(mode)
172             def contains(self,lis,c,s):
173                 #Flag
174                 contains=False
175                 for i in range(len(lis)):
176                     #open list have cost and node
177                     if s=="gg":
178                         _,l=lis[i]
179                     #closed list only have node
180                     else:
181                         l=lis[i]
182                     #Return true when l and c have same
puzzle(State) attribute
183                     if l.isSamePuzzle(c.puzzle)==True:
184                         contains=True
185                     #Return flag
186                     return contains
187             #Execution starts here
188             #Source
189             puzzle=[1,2,4,3,0,5,7,6,8]
190             #Goal
191             goal=[0,1,2,3,4,5,6,7,8]
192             root=Node(puzzle)
193             ui=BestFirst()
194             solution=ui.bestfirst(root)
195             #Solution is from goal to root so reverse it

```

```

196         solution.reverse()
197         if len(solution)>0:
198             for i in range(len(solution)):
199                 solution[i].printPuzzle()
200                 if i<len(solution)-1:
201                     print("\n\n*****Next move
*****",i)
202             else:
203                 print("\nNo path to solution is found")

```

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

Output steps using BestFirst

```

1
2 Goal Found
3
4 Tracing path...
5 1 2 4
6 3 0 5
7 7 6 8
8
9 *****Next move***** 0
10 1 2 4
11 3 5 0
12 7 6 8
13
14 *****Next move***** 1
15 1 2 0
16 3 5 4
17 7 6 8
18
19 *****Next move***** 2
20 1 0 2
21 3 5 4
22 7 6 8
23
24 *****Next move***** 3
25 0 1 2
26 3 5 4
27 7 6 8
28
29 *****Next move***** 4
30 3 1 2
31 0 5 4
32 7 6 8
33
34 *****Next move***** 5
35 3 1 2
36 5 0 4
37 7 6 8
38
39 *****Next move***** 6
40 3 1 2
41 5 6 4
42 7 0 8
43
44 *****Next move***** 7
45 3 1 2
46 5 6 4
47 0 7 8
48
49 *****Next move***** 8
50 3 1 2
51 0 6 4
52 5 7 8
53
54 *****Next move***** 9
55 3 1 2
56 6 0 4
57 5 7 8
58
59 *****Next move***** 10
60 3 1 2
61 6 4 0

```

```
62 5 7 8
63
64 *****Next move***** 11
65 3 1 2
66 6 4 8
67 5 7 0
68
69 *****Next move***** 12
70 3 1 2
71 6 4 8
72 5 0 7
73
74 *****Next move***** 13
75 3 1 2
76 6 4 8
77 0 5 7
78
79 *****Next move***** 14
80 3 1 2
81 0 4 8
82 6 5 7
83
84 *****Next move***** 15
85 3 1 2
86 4 0 8
87 6 5 7
88
89 *****Next move***** 16
90 3 1 2
91 4 5 8
92 6 0 7
93
94 *****Next move***** 17
95 3 1 2
96 4 5 8
97 6 7 0
98
99 *****Next move***** 18
100 3 1 2
101 4 5 0
102 6 7 8
103
104 *****Next move***** 19
105 3 1 2
106 4 0 5
107 6 7 8
108
109 *****Next move***** 20
110 3 1 2
111 0 4 5
112 6 7 8
113
114 *****Next move***** 21
115 0 1 2
116 3 4 5
117 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.

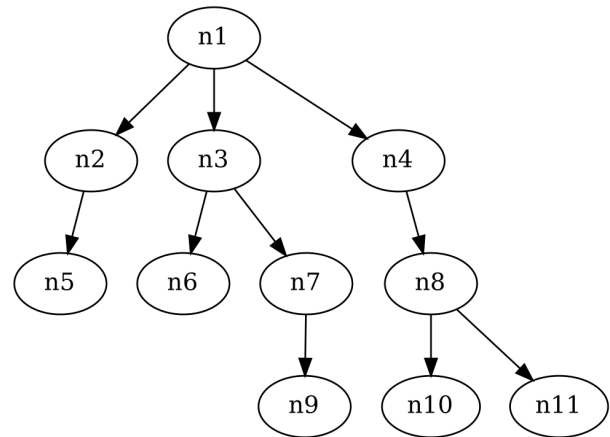
For level 2 we apply DFS on n1, so comparison sequence will be like this, n1,n2,n5,n3,n6,n7,n4,n8. Here also we dont find goal so head to next level.

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.

```

1 class Node:
2     #Initilise the node
3     def __init__(self,puzzle,tmp=0):
4         self.children=list()
5         self.parent=None
6         self.puzzle=list()
7         self.x=0
8         self.setPuzzle(puzzle)
9         self.tmp=tmp
10        self.next=None
11        #Children: contain nodes to children
12        #Parent points to the parent of current
13        node
14        #Puzzle contain state
15        #X contain location of blank
16        #tmp contain cost to node from source
17        #next used in hashing in case 2 nodes
18        have same hashing
19        #Inserts state into self state from given
20        state
21        def setPuzzle(self,puzzle):
22            for i in range(len(puzzle)):
23                self.puzzle.append(puzzle[i])
24        #Check if current state is goal state
25        def goalTest(self):
26            return self.puzzle==goal
27        #Perform right move
28        def moveToRight(self,p,i):
29            #p is state
30            #i is current blank place
31            if i%3<2:#Check if not right most
32                pc=[]
33                self.copyPuzzle(pc,p)
34
35                temp=pc[i+1]
36                pc[i+1]=pc[i]
37                pc[i]=temp
38
39                child=Node(pc,self.tmp+1)
40                self.children.append(child)
41                child.parent=self
42        #Perform left move
43        def moveToLeft(self,p,i):
44            if i%3>0:
45                pc=[]
46                self.copyPuzzle(pc,p)
47
48                temp=pc[i-1]
49                pc[i-1]=pc[i]
50                pc[i]=temp
51
52                child=Node(pc,self.tmp+1)
53                self.children.append(child)
54                child.parent=self
55        #Perform up move
56        def moveToUp(self,p,i):
57            if i-3>=0:
58                pc=[]
59                self.copyPuzzle(pc,p)
60
61                temp=pc[i-3]
62                pc[i-3]=pc[i]
63                pc[i]=temp
64
65                child=Node(pc,self.tmp+1)
66                self.children.append(child)
67                child.parent=self
68        #Perform down move

```

```

66 def moveToDown(self,p,i):
67     if i+3<len(self.puzzle):
68         pc=[]
69         self.copyPuzzle(pc,p)
70
71         temp=pc[i+3]
72         pc[i+3]=pc[i]
73         pc[i]=temp
74
75         child=Node(pc,self.tmp+1)
76         self.children.append(child)
77         child.parent=self
78     #Print puzzle
79     def printPuzzle(self):
80         for i in range(len(self.puzzle)):
81             if i%3==0:
82                 print('\n')
83             print(self.puzzle[i],end=" ")
84     #Return true when p is same as current
85     state
86     def isSamePuzzle(self,p):
87         samePuzzle=True
88         for i in range(len(p)):
89             if self.puzzle[i]!=p[i]:
90                 samePuzzle=False
91         return samePuzzle
92     #Explore current node
93     def expandNode(self):
94         #get position of blank tile
95         for i in range(len(self.puzzle)):
96             if self.puzzle[i]==0:
97                 self.x=i
98         #make move for them
99         self.moveToRight(self.puzzle,self.x)
100        self.moveToLeft(self.puzzle,self.x)
101        self.moveToUp(self.puzzle,self.x)
102        self.moveToDown(self.puzzle,self.x)
103    #Copy contents from b to a
104    def copyPuzzle(self,a,b):
105        for i in range(len(b)):
106            a.append(b[i])
107    #For hashing
108    from collections import OrderedDict
109    dict=OrderedDict
110    #Get all nodes in path from n to root node
111    def pathTrace(path,n):
112        current=n
113        path.append(current)
114        while(current.parent!=None):
115            current=current.parent
116            path.append(current)
117    #Class to perform uniform search
118    class uninformedSearch:
119        #Hashing function
120        def hashing(self,x):
121            hash_val=0
122            # simply sum of index*state
123            for i in range(len(x)):
124                hash_val+=(i+1)**x[i]
125            return hash_val
126        #Perform uniform search
127        def bfs(self,root):
128            #No of explored nodes
129            explored=0
130            #No of children faced
131            child=0
132            #No of children put in memory at start
133            root is in memory
134            childm=1
135            #Maximum memory needed for program
136            maxm=childm
137            #Stored solution path here
138            pathToSolution=[]

```

```

138     #2 hash tables (Dictionary)
139     #To explore
140     openList=dict()
141     #Explored
142     closedList=dict()
143     #Adding root to to explore list
144     openList[self.hashing(root.puzzle)]=
root
145     #Falg if goal found
146     goalFound=False
147     #Check if current is goal state and
print it
148     if root.goalTest():
149         print("\nGoal Found")
150         goalFound=True
151         #Get path from current node(Goal)
to source
152         pathTrace(pathToSolution, root)
153         return pathToSolution, explored,
child, childm
154         #Stop when there is nothing to explore
or goal is found
155         while len(openList)!=0 and goalFound==
False:
156             #Get one node from list which
contain node to explore
157             _, currentNode=openList.popitem(last
=False)
158             #One node is out of memory
159             childm-=1
160             #Get hast value for that one node
161             chash=self.hashing(currentNode.
puzzle)
162             #Mark node as explored as this node
is going to be explored
163             #It may have same hash also so
check next
164             if closedList.get(chash)!=None:
165                 tmp1=closedList.get(chash)
166                 while tmp1.next!=None:tmp1=tmp1
.next
167                 tmp1.next=currentNode
168             else:closedList[chash]=currentNode
169             #Explore the node
170             currentNode.expandNode()
171             #Increament counter for explored
172             explored+=1
173             #For all childrens check goal state
174             for i in range(len(currentNode.
children)):
175                 #take ith child
176                 currentChild=currentNode.
children[i]
177                 #Child is faced in process so
increment counter
178                 child+=1
179                 #Check if this is goal state
180                 if currentChild.goalTest():
181                     goalFound=True
182                     #Get path to root
183                     pathTrace(pathToSolution,
currentChild)
184                     break
185                 #Check if node in openList or
closedList
186                 if(self.con(openList,
currentChild,"open")==False and self.con(
closedList,currentChild,"closed")==False):
187                     #If not then just append it
to open list
188                     openList[self.hashing(
currentChild.puzzle)]=currentChild
189                     #So its added to open list
so memory++

```

```

190         childm+=1
191         #Update maxm required
192         if childm>maxm:maxm=childm
193         return pathToSolution, explored, child,
maxm
194         #pathToSolution: contain path from goal
to root
195         #explored: Number of nodes explored
196         #child: Number of child faced
197         #maxm: Maximum memory(Nodes) required
to run code
198         #Check if lis(HashTable) contains c(
CurrentNode) with s(MODE)
199         def con(self, lis, c, s):
200             #Check if node is present in dictionary
for hash value
201             if lis.get(self.hashing(c.puzzle)) !=
None:
202                 #Check if node at hash value is
same
203                 if lis[self.hashing(c.puzzle)].
isSamePuzzle(c.puzzle):
204                     #Update cost to node if needed
and in open list
205                     if s=="open" and lis[self.
hashing(c.puzzle)].tmp>c.tmp:
206                         lis[self.hashing(c.puzzle)
].tmp=c.tmp
207                         lis[self.hashing(c.puzzle)
].parent=c.parent
208                         lis[self.hashing(c.puzzle)
].children=c.children
209                         #Node is already in list, so
true
210                     return True
211                     #if same hash for 2 or more nodes
212                     else:
213                         #temporar pointer
214                         tmp1=lis[self.hashing(c.puzzle)
]
215                         #Check all pointers
216                         while tmp1.next!=None:
217                             tmp1=tmp1.next
218                             if tmp1.isSamePuzzle(c.
puzzle):
219                                 #Update cost to node if
needed and in open list
220                                 if s=="open" and tmp1.
tmp>c.tmp:
221                                     tmp1.tmp=c.tmp
222                                     tmp1.parent=c.
parent
223                                     tmp1.children=c.
children
224                                     #Node is already in list
, so true
225                                 return True
226                                 if(s=="open"):
227                                     tmp1.next=c
228                                     return True
229                                 #False otherwise
230                                 return False
231
232 #Execution starts here
233 #My randoly generated state
234 puzzle=[1,2,4,3,0,5,7,6,8]
235 #My goal state
236 goal=[0,1,2,3,4,5,6,7,8]
237 root=Node(puzzle)
238 ui=uninformedSearch()
239 #Call uniform search
240 solution, explored, child, childm=ui.bfs(root)
241 #Reverse it as solution is reverse(from goal to
source)

```

```

242 solution.reverse()
243 #Print solution
244 if len(solution)>0:
245     for i in range(len(solution)):
246         print("\n\nMove",i)
247         solution[i].printPuzzle()
248         if i<len(solution)-1:
249             print("\n\n*****Next move
*****")
250         print("\n\nDepth of solution is : ",
solution[-1].tmp)
251         print("\nTotal moves required : ",len(
solution)-1)
252         print("\nTotal nodes explored : ",explored)
253         print("\nTotal childrens : ",child)
254         print("\nTotal children in memory at once :
",childm)
255 else:
256     print("\nNo path to solution is found")

```

Listing 2. 8-puzzle using Uniform Search

Output steps using Uniform

```

1 Move 0
2 1 2 4
3 3 0 5
4 7 6 8
5
6 *****Next move*****
7 Move 1
8 1 2 4
9 3 6 5
10 7 0 8
11
12 *****Next move*****
13 Move 2
14 1 2 4
15 3 6 5
16 7 8 0
17
18 *****Next move*****
19 Move 3
20 1 2 4
21 3 6 0
22 7 8 5
23
24 *****Next move*****
25 Move 4
26 1 2 0
27 3 6 4
28 7 8 5
29
30 *****Next move*****
31 Move 5
32 1 0 2
33 3 6 4
34 7 8 5
35
36 *****Next move*****
37 Move 6
38 0 1 2
39 3 6 4
40 7 8 5
41
42 *****Next move*****
43 Move 7
44 3 1 2
45 0 6 4
46 7 8 5
47
48 *****Next move*****
49 Move 8
50 3 1 2
51 6 0 4

```

```

52 7 8 5
53
54 *****Next move*****
55 Move 9
56 3 1 2
57 6 4 0
58 7 8 5
59
60 *****Next move*****
61 Move 10
62 3 1 2
63 6 4 5
64 7 8 0
65
66 *****Next move*****
67 Move 11
68 3 1 2
69 6 4 5
70 7 0 8
71
72 *****Next move*****
73 Move 12
74 3 1 2
75 6 4 5
76 0 7 8
77
78 *****Next move*****
79 Move 13
80 3 1 2
81 0 4 5
82 6 7 8
83
84 *****Next move*****
85 Move 14
86 0 1 2
87 3 4 5
88 6 7 8
89
90 Depth of solution is : 14
91 Total moves required : 14
92 Total nodes explored : 3457
93 Total childrens : 9562
94 Total children in memory at once : 2163

```


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

```

1 #Same as before but return nodes at given depth
2 class configurations:
3     #For hashing
4     def hashing(self,x):
5         hash_val=0
6         for i in range(len(x)):
7             hash_val+=(i+1)**x[i]
8         return hash_val
9     #To get nodes at depth depth
10    def dbfs(self, root, depth):
11        #root: Root node
12        #depth: Depth at which to return nodes
13        explored=0
14        child=0
15        childm=0
16        maxm=0
17        #This stores nodes which have required
18        depth
19        solutions=[]
20        pathToSolution=[]
21        openList=dict()
22        closedList=dict()
23        openList[self.hashing(root.puzzle)]=
24        root
25        goalFound=False
26        #Check if current depth of root is same
27        as required
28        if root.tmp==depth:
29            goalFound=True
30            #Node is found, return path to it
31            from source
32            pathTrace(pathToSolution,root)
33            solutions.append(pathToSolution)
34            return solutions
35            while len(openList)!=0 and goalFound==
36            False:
37                _,currentNode=openList.popitem(last
38                =False)
39                chash=self.hashing(currentNode.
40                puzzle)
41                if closedList.get(chash)!=None:
42                    tmp1=closedList.get(chash)
43                    while tmp1.next!=None:tmp1=tmp1
44                    .next
45                    tmp1.next=currentNode
46                else:closedList[chash]=currentNode
47                currentNode.expandNode()
48                explored+=1
49                for i in range(len(currentNode.
50                children)):
51                    currentChild=currentNode.
52                    children[i]
53                    #Skip nodes which are above
54                    depth
55                    if currentChild.tmp>depth:
56                        goalFound=True
57                        #Go out as we got what we
58                        needed
59                        break
60                    child+=1
61                    if (self.con(openList,
62                    currentChild,"open")==False and self.con(
63                    closedList,currentChild,"closed")==False):
64                        openList[self.hashing(
65                        currentChild.puzzle)]=currentChild
66                        childm+=1
67                        if childm>maxm:maxm=childm
68                        #If node matches with depth
69                        then add it to solutions
70                        if currentChild.tmp==depth:
71                            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 already 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 already 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:

```

```

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

```

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

Output top 10 solutions with depth 8

```

1 Found 114 solutions or configurations with
  depth 8
2 Showing top : 10
3 Solution 1 with depth 8
4 2 5 0
5 1 3 4
6 6 7 8
7
8 *****Next solution*****
9 Solution 2 with depth 8
10 2 3 5
11 1 0 4
12 6 7 8
13
14 *****Next solution*****
15 Solution 3 with depth 8
16 1 2 5
17 6 3 4
18 7 8 0
19
20 *****Next solution*****
21 Solution 4 with depth 8
22 1 2 5
23 6 0 4
24 7 3 8
25
26 *****Next solution*****
27 Solution 5 with depth 8
28 1 5 4
29 3 2 8
30 6 7 0
31
32 *****Next solution*****
33 Solution 6 with depth 8
34 3 1 5
35 2 0 4
36 6 7 8
37
38 *****Next solution*****
39 Solution 7 with depth 8
40 3 1 5
41 6 2 4
42 0 7 8
43
44 *****Next solution*****
45 Solution 8 with depth 8
46 1 2 5
47 3 0 7
48 6 8 4
49
50 *****Next solution*****
51 Solution 9 with depth 8
52 1 2 0
53 3 7 5
54 6 8 4
55
56 *****Next solution*****
57 Solution 10 with depth 8
58 1 2 5
59 7 0 4
60 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.

```

1 import time as t
2 import sys
3 node_size=sys.getsizeof(Node
  ([0,1,2,3,4,5,6,7,8]))
4 #Tabular format
5 goal=[0,1,2,3,4,5,6,7,8]
6 print("\begin{center}")
7 print("\begin{tabular}{ |c|c|c| }")
8 print("Init state & Time(msec) & Memory\\")
9 count=0
10 #Solutions is list of instances of 8-puzzle
11 for i in solutions:
12     count+=1
13     tmp=t.time()
14     puzzle=i[0].puzzle
15     root=Node(puzzle)
16     ui=uninformedSearch()
17     solution,explored,child,childm=ui.bfs(root)
18     #to divide in half
19     if count==35:
20         print("\end{tabular}")
21         print("\end{center}")
22         print("\begin{center}")
23         print("\begin{tabular}{ |c|c|c| }")
24         print("Init state & Time(msec) & Memory
25         \\")
26         if solution[0].tmp==d:
27             print("text{",solution[-1].puzzle,"} &
28             ",int((t.time()-tmp)*1000000)," & ",childm*
29             node_size,end="\\")
30 print("\end{tabular}")
31 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/>