

## LAB-B: WRITE A PROGRAM TO SOLVE 8-PUZZLE PROBLEM USING

**A: BFS**

**B: DFS**

**C: ITERATIVE DEPENDING DFS**

### Problem:

We also know the eight-puzzle problem by the name of N puzzle problem or sliding puzzle problem.

N-puzzle that consists of N tiles (N+1 tiles with an empty tile) where N can be 8, 15, 24 and so on.

In our example  $N = 8$ . (That is square root of  $(8+1) = 3$  rows and 3 columns).

In the same way, if we have  $N = 15, 24$  in this way, then they have Row and columns as follow (square root of  $(N+1)$  rows and square root of  $(N+1)$  columns).

That is if  $N=15$  than number of rows and columns= 4, and if  $N= 24$  number of rows and columns= 5.

So, basically in these types of problems we have given a initial state or initial configuration (Start state) and a Goal state or Goal Configuration.

Here We are solving a problem of 8 puzzle that is a 3x3 matrix.

7	2	4
5		6
8	3	1

Start State

	1	2
3	4	5
6	7	8

Goal State

### Solution:

The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal state.

Rules of solving puzzles

Instead of moving the tiles in the empty space we can visualize moving the empty space in place of the tile.

The empty space can only move in four directions (Movement of empty space)

1. Up
2. Down
3. Right or
4. Left

The empty space cannot move diagonally and can take only one step at a time.

## A: Breadth First Search

**OBJECTIVE:** To implement 8-puzzle using BFS.

**ILLUSTRATION:**

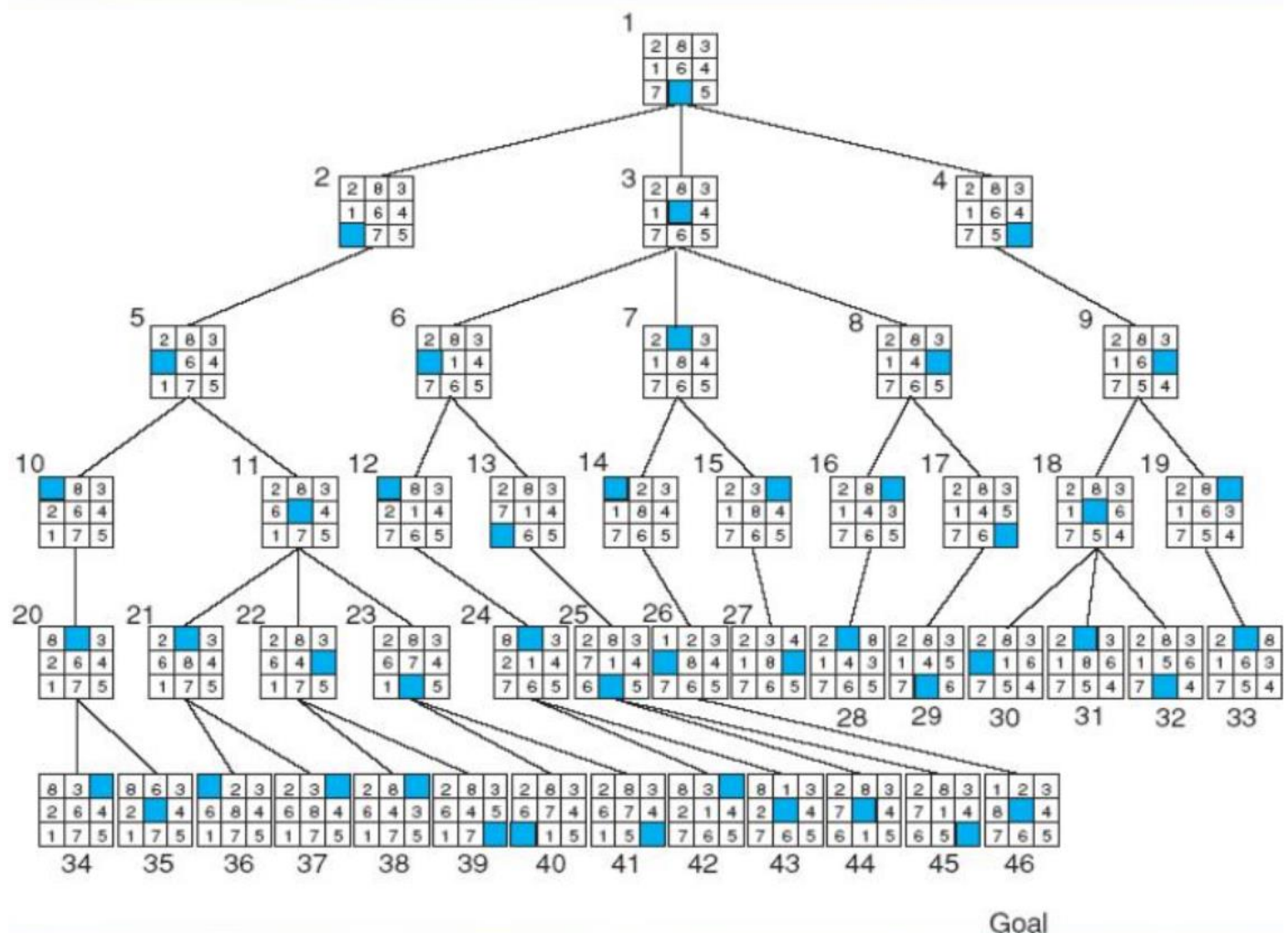


Fig: BFS of 8 puzzle, showing order in which states were removed from open.

## THEORY:

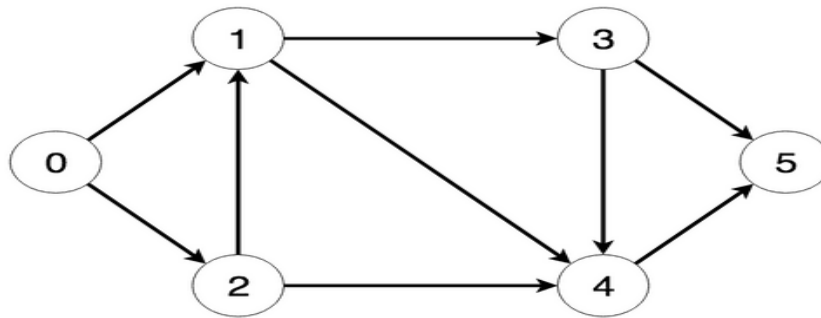
The Breadth-first search algorithm is an algorithm used to solve the shortest path problem in a graph without edge weights (i.e. a graph where all nodes are the same “distance” from each other, and they are either connected or not). This means that given a number of nodes and the edges between them, the Breadth-first search algorithm finds the shortest path from the specified start node to all other nodes. Nodes are sometimes referred to as vertices (plural of vertex) - here, we’ll call them nodes.

The basic principle behind the Breadth-first search algorithm is to take the current node (the start node in the beginning) and then add all of its neighbors that we haven’t visited yet to a queue. Continue this with the next node in the queue (in a queue that is the “oldest” node). Before we add a node to the queue, we set its distance to the distance of the current node plus 1 (since all edges are weighted equally), with the distance to the

start node being 0. This is repeated until there are no more nodes in the queue (all nodes are visited).

### Example of the Algorithm

Consider the following graph:



The steps the algorithm performs on this graph if given node 0 as a starting point, in order, are:

1. Visiting node 0
2. Visited nodes: [true, false, false, false, false, false], Distances: [0, 0, 0, 0, 0, 0]
  - a. Removing node 0 from the queue...
  - b. Visiting node 1, setting its distance to 1 and adding it to the queue
  - c. Visiting node 2, setting its distance to 1 and adding it to the queue
3. Visited nodes: [true, true, true, false, false, false], Distances: [0, 1, 1, 0, 0, 0]
  - a. Removing node 1 from the queue...
  - b. Visiting node 3, setting its distance to 2 and adding it to the queue
  - c. Visiting node 4, setting its distance to 2 and adding it to the queue
4. Visited nodes: [true, true, true, true, true, false], Distances: [0, 1, 1, 2, 2, 0]
  - a. Removing node 2 from the queue...
  - b. No adjacent, unvisited nodes
5. Visited nodes: [true, true, true, true, true, false], Distances: [0, 1, 1, 2, 2, 0]
  - a. Removing node 3 from the queue...
  - b. Visiting node 5, setting its distance to 3 and adding it to the queue
6. Visited nodes: [true, true, true, true, true, true], Distances: [0, 1, 1, 2, 2, 3]
  - a. Removing node 4 from the queue...
  - b. No adjacent, unvisited nodes
7. Visited nodes: [true, true, true, true, true, true], Distances: [0, 1, 1, 2, 2, 3]
  - a. Removing node 5 from the queue...
8. No more nodes in the queue. Final distances: [0, 1, 1, 2, 2, 3]

### Runtime Complexity of the Algorithm

The runtime complexity of Breadth-first search is  $O(|E| + |V|)$  ( $|V|$  = number of Nodes,  $|E|$  = number of Edges) if adjacency-lists are used. If we simply search all nodes to find connected nodes in each step, and use a matrix to look up whether two nodes are adjacent, the runtime complexity increases to  $O(|V|^2)$ .

Depending on the graph this might not matter, since the number of edges can be as big as  $|V|^2$  if all nodes are connected with each other.

### Space Complexity of the Algorithm

The space complexity of Breadth-first search depends on how it is implemented as well and is equal to the runtime complexity.

## ALGORITHMH:

1. Initialize the distance to the starting node as 0. The distances to all other node do not need to be initialized since every node is visited exactly once.
2. Set all nodes to “unvisited”
3. Add the first node to the queue and label it visited.
4. While there are nodes in the queue:
  - a. Take a node out of the queue
  - b. For all nodes next to it that we haven’t visited yet, add them to the queue, set their distance to the distance to the current node plus 1, and set them as “visited”

In the end, the distances to all nodes will be correct.

## PROGRAM:

```
from copy import deepcopy
from collections import deque

q = deque() # A queue to perform the BFS search.

visited = set() # A set to avoid reaching the previously visited state.

# To print the intermediate states while performing the BFS.

def print_current_state(a):
    for i in a:
        for j in i:
            print(j, end = " ")
        print()
        print()

def left(a):
    b = deepcopy(a);
    for i in range(len(b)):
        for j in range(len(b)):
            if a[i][j] == 0:
                if j - 1 >= 0:
                    b[i][j - 1], b[i][j] = b[i][j], b[i][j - 1]
    c = tuple(map(tuple, b)) # Since lists are mutable it is required to be converted into some
                             # immutable container which can be used as a key in the dictionary.
    if c in visited:
        return
    visited.add(c)
    q.append(b)
    return

def right(a):
    b = deepcopy(a);
    for i in range(len(b)):
        for j in range(len(b)):
            if a[i][j] == 0:
                if j + 1 < len(b):
```

```

b[i][j + 1], b[i][j] = b[i][j], b[i][j + 1]
c = tuple(map(tuple, b))
if c in visited:
    return
visited.add(c)
q.append(b)
return

```

```

def up(a):
    b = deepcopy(a);
    for i in range(len(b)):
        for j in range(len(b)):
            if a[i][j] == 0:
                if i - 1 >= 0:
                    b[i - 1][j], b[i][j] = b[i][j], b[i - 1][j]
                    c = tuple(map(tuple, b))
                    if c in visited:
                        return
                    visited.add(c)
                    q.append(b)
    return

```

```

def down(a):
    b = deepcopy(a);
    for i in range(len(b)):
        for j in range(len(b)):
            if a[i][j] == 0:
                if i + 1 < len(b):
                    b[i + 1][j], b[i][j] = b[i][j], b[i + 1][j]
                    c = tuple(map(tuple, b))
                    if c in visited:
                        return
                    visited.add(c)
                    q.append(b)
    return

```

```

def bfs(a, b):

```

```

    q.append(a)

```

```

    count = 0
    while len(q) is not 0:
        count += 1;

```

```

    now = q.popleft()

```

```

    print_current_state(now)

```

```

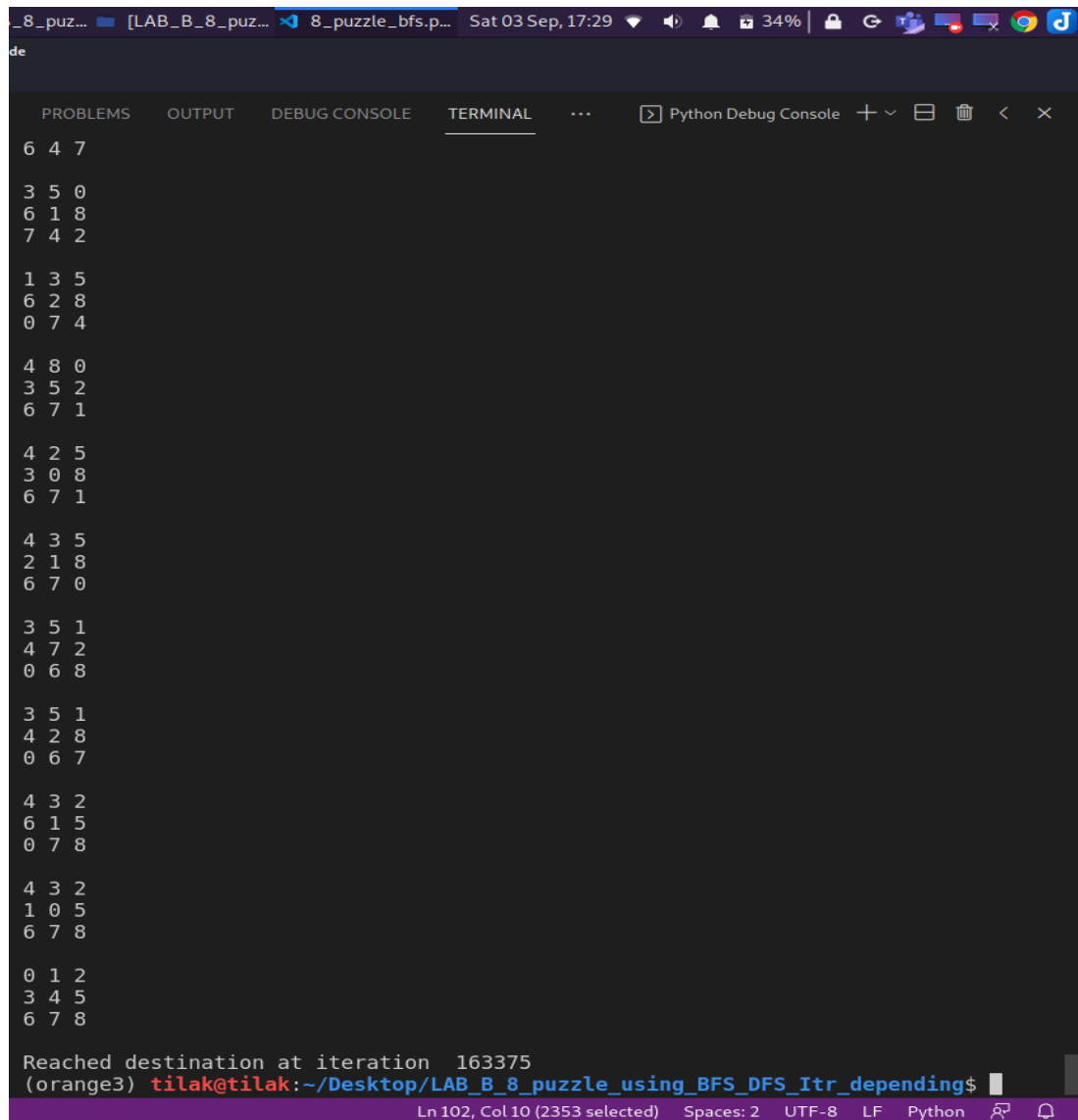
    if now == b:
        print("Reached destination at iteration ", count)
        return
    for i in range(len(now)):
        for j in range(len(now)):
            if now[i][j] == 0:

```

```
left(now)
up(now)
down(now)
right(now)
a = [[7, 2, 4], [5, 0, 6], [8, 3, 1]] # Initial State
b = [[0, 1, 2], [3, 4, 5], [6, 7, 8]] # Goal State

bfs(a, b)
```

## OUTPUT:



```
de
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ... Python Debug Console + - [] [] < x
6 4 7
3 5 0
6 1 8
7 4 2
1 3 5
6 2 8
0 7 4
4 8 0
3 5 2
6 7 1
4 2 5
3 0 8
6 7 1
4 3 5
2 1 8
6 7 0
3 5 1
4 7 2
0 6 8
3 5 1
4 2 8
0 6 7
4 3 2
6 1 5
0 7 8
4 3 2
1 0 5
6 7 8
0 1 2
3 4 5
6 7 8
Reached destination at iteration 163375
(orange3) tilak@tilak:~/Desktop/LAB_B_8_puzzle_using_BFS_DFS_Itr_dependin$
```

## CONCLUSION:

Hence, we implemented BFS to solve 8 puzzle problem and reached the destination at iteration 163375.

## B: Depth First Search

**OBJECTIVE:** To implement 8-puzzle using DFS.

**ILLUSTRATION:**

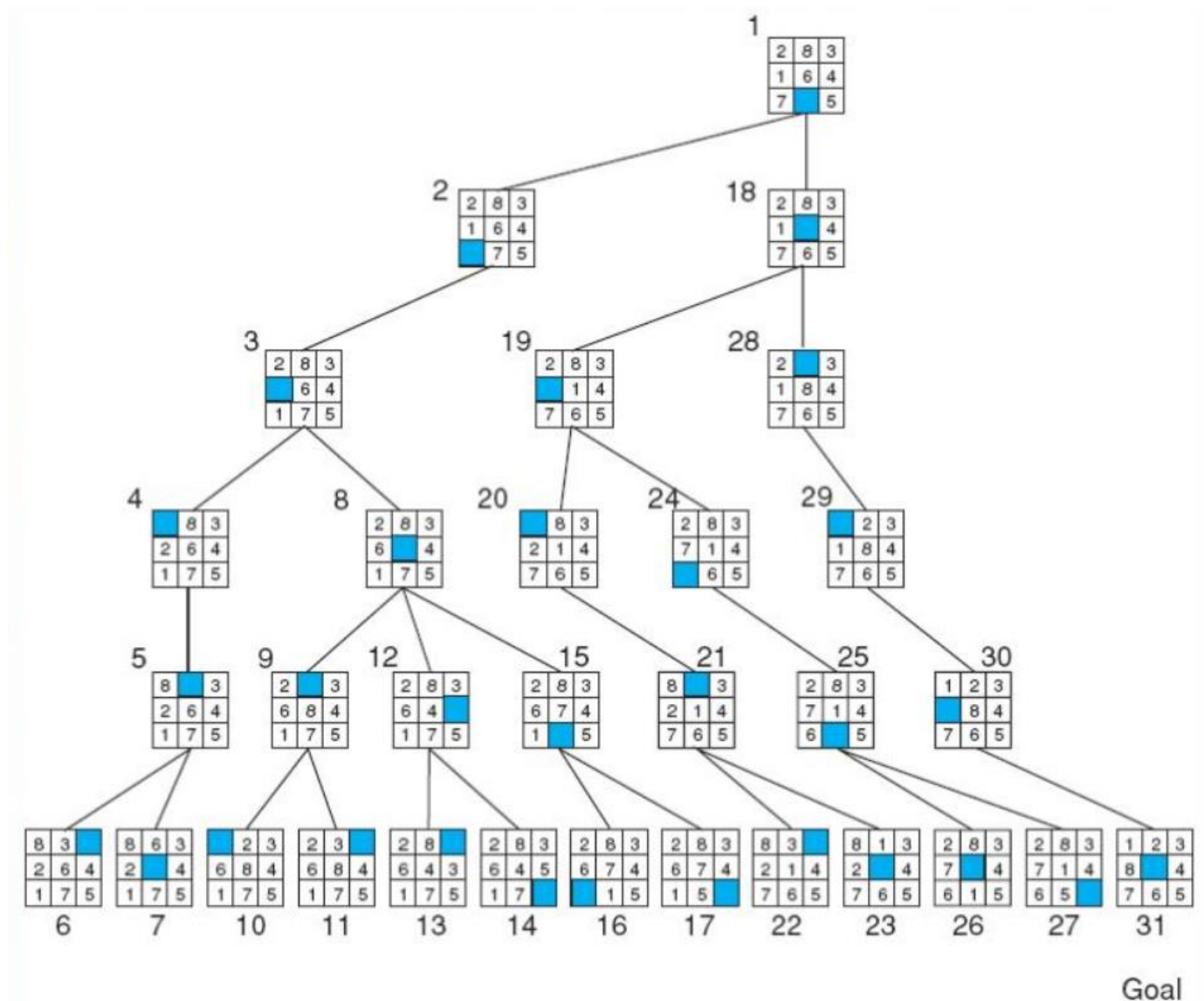


fig: DFS of 8 puzzle with a depth bound of 5.

## THEORY:

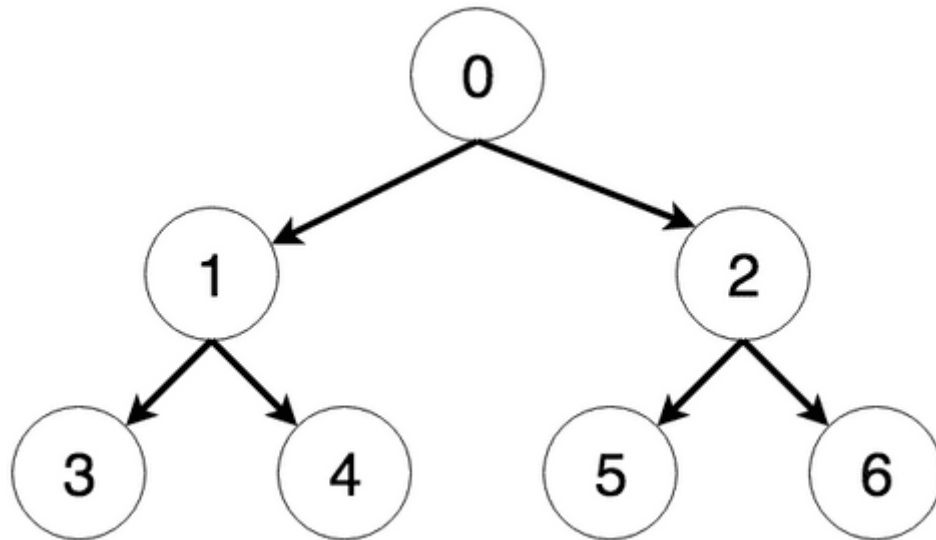
The Depth-First Search (also DFS) algorithm is an algorithm used to find a node in a tree. This means that given a tree data structure, the algorithm will return the first node in this tree that matches the specified condition (i.e. being equal to a value). Nodes are sometimes referred to as vertices (plural of vertex) - here, we'll call them nodes. The edges have to be unweighted. This algorithm can also work with unweighted graphs if a mechanism to keep track of already visited nodes is added.

The basic principle of the algorithm is to start with a start node, and then look at the first child of this node. It then looks at the first child of that node (grandchild of the start node) and so on, until a node has no more children (we've reached a leaf node). It then goes up one level, and looks at the next child. If there are no more children, it goes up

one more level, and so on, until it find more children or reaches the start node. If hasn't found the goal node after returning from the last child of the start node, the goal node cannot be found, since by then all nodes have been traversed.

### Example of the Algorithm

Consider the following tree:



The steps the algorithm performs on this tree if given node 0 as a starting point, in order, are:

- 1) Visiting Node 0
- 2) Visiting Node 1
- 3) Visiting Node 3
- 4) Went through all children of 3, returning to it's parent.
- 5) Visiting Node 4
- 6) Went through all children of 4, returning to it's parent.
- 7) Went through all children of 1, returning to it's parent.
- 8) Visiting Node 2
- 9) Visiting Node 5
- 10) Went through all children of 5, returning to it's parent.
- 11) Visiting Node 6
- 12) Found the node we're looking for!

### Runtime of the Algorithm

The runtime of regular Depth-First Search (DFS) is  $O(N)$  ( $N$  = number of Nodes in the tree), since every node is traversed at most once. The number of nodes is equal to  $b^d$ , where  $b$  is the branching factor and  $d$  is the depth, so the runtime can be rewritten as  $O(b^d)$ .

### Space of the Algorithm

The space complexity of Depth-First Search (DFS) is, if we exclude the tree itself,  $O(d)$ , with  $d$  being the depth, which is also the size of the call stack at maximum depth. If we include the tree, the space complexity is the same as the runtime complexity, as each node needs to be saved.

## ALGORITHM:



- 1.For each child of the current node
- 2.If it is the target node, return. The node has been found.
- 3.Set the current node to this node and go back to 1.
- 4.If there are no more child nodes to visit, return to the parent.
- 5.If the node has no parent (i.e. it is the root), return. The node has not been found.

## PROGRAM:

```

from copy import deepcopy

stack = [] # Using a list in python as a stack to perform the DFS search.

visited = set() # A set to avoid reaching the previously visited state.

# To print the intermediate states while performing the BFS.

def print_current_state(a):
    for i in a:
        for j in i:
            print(j, end = " ")
        print()
        print()

def left(a):
    b = deepcopy(a);
    for i in range(len(b)):
        for j in range(len(b)):
            if a[i][j] == 0:
                if j - 1 >= 0:
                    b[i][j - 1], b[i][j] = b[i][j], b[i][j - 1]
                    c = tuple(map(tuple, b)) # Since lists are mutable it is required to be converted into some
                    immutable container which can be used as a key in the dictionary.
                    if c in visited:
                        return
                    visited.add(c)
                    stack.append(b)
    return

def right(a):
    b = deepcopy(a);
    for i in range(len(b)):
        for j in range(len(b)):
            if a[i][j] == 0:
                if j + 1 < len(b):
                    b[i][j + 1], b[i][j] = b[i][j], b[i][j + 1]
                    c = tuple(map(tuple, b))
                    if c in visited:
                        return
                    visited.add(c)
                    stack.append(b)
    return

def up(a):
    b = deepcopy(a);

```

```

for i in range(len(b)):
    for j in range(len(b)):
        if a[i][j] == 0:
            if i - 1 >= 0:
                b[i - 1][j], b[i][j] = b[i][j], b[i - 1][j]
            c = tuple(map(tuple, b))
            if c in visited:
                return
            visited.add(c)
            stack.append(b)
            return

def down(a):
    b = deepcopy(a);
    for i in range(len(b)):
        for j in range(len(b)):
            if a[i][j] == 0:
                if i + 1 < len(b):
                    b[i + 1][j], b[i][j] = b[i][j], b[i + 1][j]
                c = tuple(map(tuple, b))
                if c in visited:
                    return
                visited.add(c)
                stack.append(b)
                return

def dfs(a, b):

    stack.append(a)

    count = 0
    while len(stack) is not 0:
        count += 1;

    now = stack.pop()

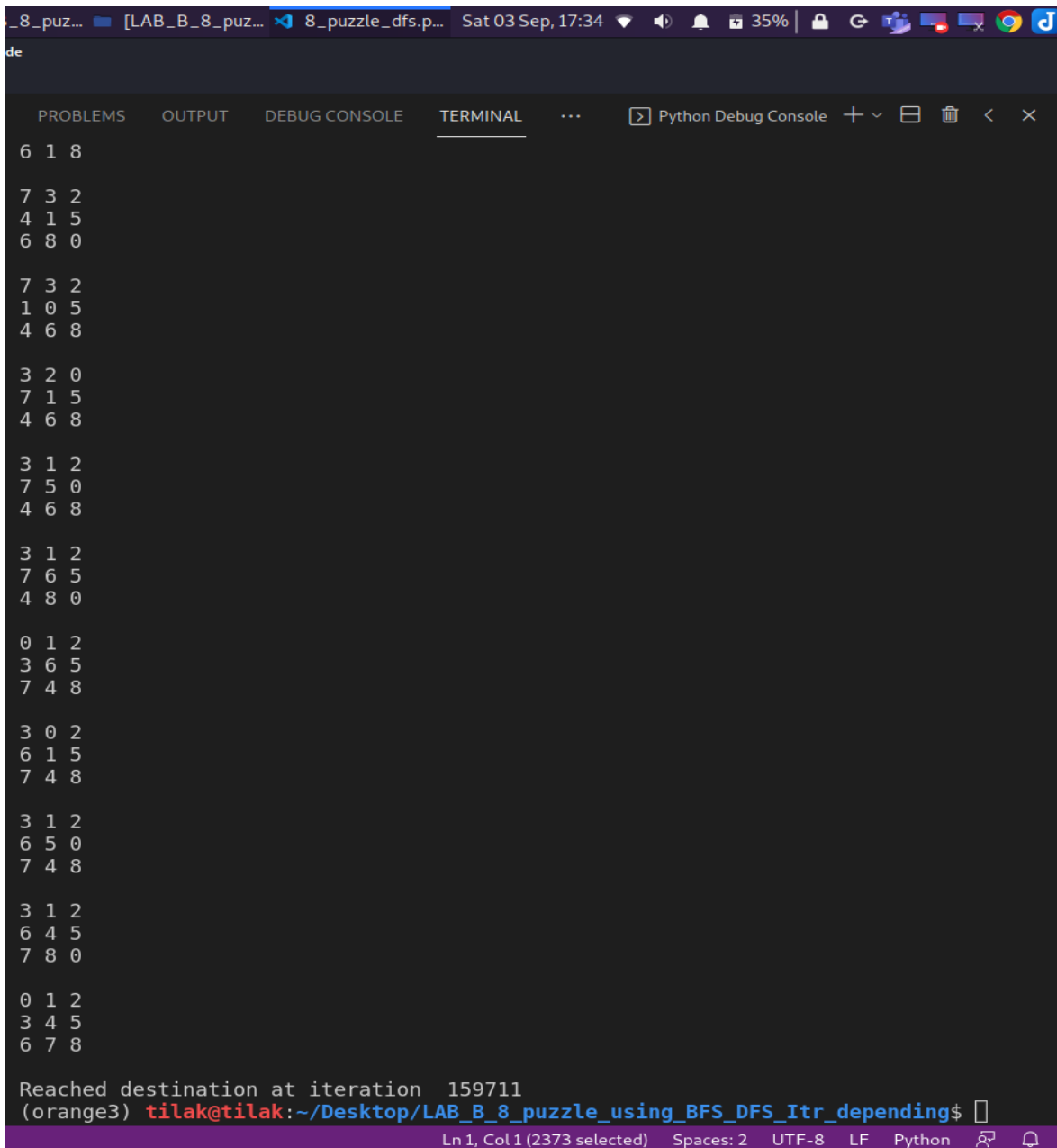
    print_current_state(now)

    if now == b:
        print("Reached destination at iteration ", count)
        return
    for i in range(len(now)):
        for j in range(len(now)):
            if now[i][j] == 0:
                right(now)
                up(now)
                down(now)
                left(now)
    a = [[7, 2, 4], [5, 0, 6], [8, 3, 1]] # Initial State
    b = [[0, 1, 2], [3, 4, 5], [6, 7, 8]] # Goal State

    dfs(a, b)

```

## OUTPUT:



```
de
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL ... Python Debug Console + - [ ] [X] < X
6 1 8
7 3 2
4 1 5
6 8 0
7 3 2
1 0 5
4 6 8
3 2 0
7 1 5
4 6 8
3 1 2
7 5 0
4 6 8
3 1 2
7 6 5
4 8 0
0 1 2
3 6 5
7 4 8
3 0 2
6 1 5
7 4 8
3 1 2
6 5 0
7 4 8
3 1 2
6 4 5
7 8 0
0 1 2
3 4 5
6 7 8
Reached destination at iteration 159711
(orange3) tilak@tilak:~/Desktop/LAB_B_8_puzzle_using_BFS_DFS_Itr_dependin$
```

**CONCLUSION:** Hence, we implemented DFS to solve 8 puzzle problem and reached the destination at iteration 159711.

## C: Iterative Depending Search

**OBJECTIVE:** To implement 8-puzzle using iterative depending search.

### **THEORY:**

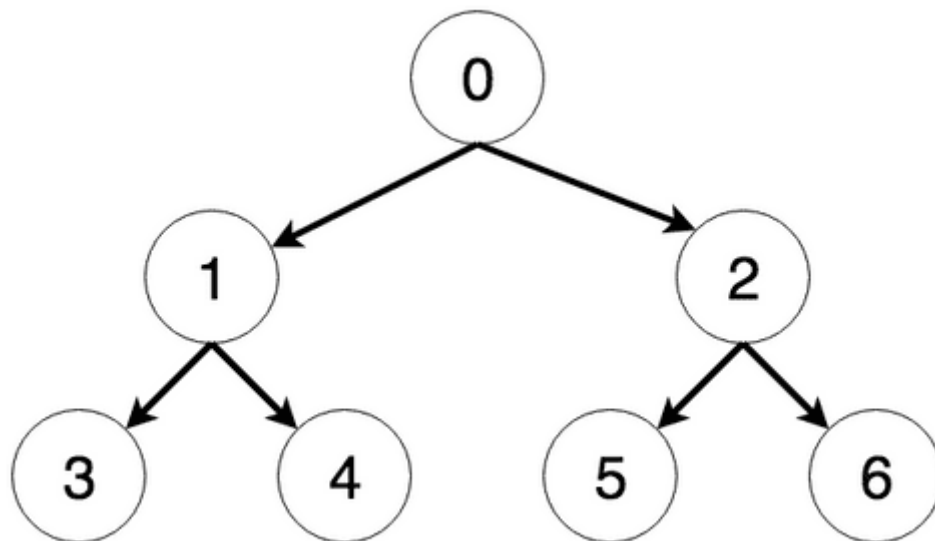
The Iterative Deepening Depth-First Search (also ID-DFS) algorithm is an algorithm used to find a node in a tree. This means that given a tree data structure, the algorithm will return the first node in this tree that matches the specified condition. Nodes are sometimes referred to as vertices (plural of vertex) - here, we'll call them nodes. The edges have to be unweighted. This algorithm can also work with unweighted graphs if mechanism to keep track of already visited nodes is added.

The basic principle of the algorithm is to start with a start node, and then look at the first child of this node. It then looks at the first child of that node (grandchild of the start node) and so on, until a node has no more children (we've reached a leaf node). It then goes up one level, and looks at the next child. If there are no more children, it goes up one more level, and so on, until it find more children or reaches the start node. If hasn't found the goal node after returning from the last child of the start node, the goal node cannot be found, since by then all nodes have been traversed.

So far this has been describing Depth-First Search (DFS). Iterative deepening adds to this, that the algorithm not only returns one layer up the tree when the node has no more children to visit, but also when a previously specified maximum depth has been reached. Also, if we return to the start node, we increase the maximum depth and start the search all over, until we've visited all leaf nodes (bottom nodes) and increasing the maximum depth won't lead to us visiting more nodes.

### **Example of the Algorithm**

Consider the following tree:



The steps the algorithm performs on this tree if given node 0 as a starting point, in order, are:

1. Visiting Node 0
2. Visiting Node 1

3. Current maximum depth reached, returning...
4. Visiting Node 2
5. Current maximum depth reached, returning...
6. Increasing depth to 2
7. Visiting Node 0
8. Visiting Node 1
9. Visiting Node 3
10. Current maximum depth reached, returning...
11. Visiting Node 4
12. Current maximum depth reached, returning...
13. Visiting Node 2
14. Visiting Node 5
15. Current maximum depth reached, returning...
16. Visiting Node 6
17. Found the node we're looking for, returning...

### Runtime of the Algorithm

If we double the maximum depth each time we need to go deeper, the runtime complexity of Iterative Deepening Depth-First Search (ID-DFS) is the same as regular Depth-First Search (DFS), since all previous depths added up will have the same runtime as the current depth ( $1/2 + 1/4 + 1/8 + \dots < 1$ ). The runtime of regular Depth-First Search (DFS) is  $O(N)$  ( $N$  = number of Nodes in the tree), since every node is traversed at most once. The number of nodes is equal to  $b^d$ , where  $b$  is the branching factor and  $d$  is the depth, so the runtime can be rewritten as  $O(b^d)$ .

### Space of the Algorithm

The space complexity of Iterative Deepening Depth-First Search (ID-DFS) is the same as regular Depth-First Search (DFS), which is, if we exclude the tree itself,  $O(d)$ , with  $d$  being the depth, which is also the size of the call stack at maximum depth. If we include the tree, the space complexity is the same as the runtime complexity, as each node needs to be saved.

### ALGORITHM:

1. For each child of the current node
2. If it is the target node, return
3. If the current maximum depth is reached, return
4. Set the current node to this node and go back to 1.
5. After having gone through all children, go to the next child of the parent (the next sibling)
6. After having gone through all children of the start node, increase the maximum depth and go back to 1.
7. If we have reached all leaf (bottom) nodes, the goal node doesn't exist.

### PROGRAM:

```
# Deepening Limited Search as DLS
def DLS(problem,limit):
    return recDLS(problem.INITIAL_STATE(),problem,limit)
```

```

# Iterative Deepening Search as IDS
def IDS(problem):
    for depth in range(1,50):# instead of infinite, taking 50
    result=DLS(problem,depth)
    if result == 1:
    return "Result found"
    elif result == -1:
    return "Failed to find Result\n"
    else:
    print("\n\ncut_off_occurred @ depth =",depth)

#finding blank's location
def findBlankPosition(array):
    for i in range(3):
    for j in range(3):
    if array[i][j]==0:
    return i,j

# creating child node for particular action
def childNode(problem,node,array_loc):
    i,j=array_loc
    if i<0 or j<0 or j>2 or i>2:#for avoiding out of bound case
    return None
    else:
    iB,jB,= findBlankPosition(node)
    node[iB][jB]=node[i][j]; node[i][j]=0
    return node

# Recursive DLS as recDLS
"""NOTE: using direct 2D-array (as node) instead of node.state (in IDS)"""
def recDLS(node,problem,limit,cut_off=0,failure=-1):
    if problem.GOAL_TEST(node):
    print("\n\nWow! finally, You have done your Job :)\n\nYour solution is:\n ",node)
    return 1 #solution(node)
    elif limit == 0:
    return cut_off
    else:
    cut_off_occurred = False
    for action_loc in problem.ACTIONS(node):
    child = deepcopy(childNode(problem,node,action_loc))
    # print("\nchild_Node",child)
    if child:# to avoid node!=None, i.e. childNode's response as None
    result = recDLS(child,problem,limit-1)
    if result == cut_off:
    cut_off_occurred = True
    elif result != failure: #result found
    return result
    if cut_off_occurred:#no solution found till this limit

```

```

return cut_off
else: #no solution found
print("\n:( very bad, Failure")
return failure

```

```

#class for problem object
class n_puzzle:# n=8
def __init__(self):
self.node=[[0,0,0],[0,0,0],[0,0,0]]
# self.goal=[[1,2,3],[4,5,6],[7,8,0]]
self.goal=[[0,1,2],[3,4,5],[6,7,8]]
def ACTIONS(self,arr): #find 0 and take actions
self.node=arr
i,j=findBlankPosition(self.node)
return [[i,j-1],[i,j+1],[i-1,j],[i+1,j]] #sending all four, boundry will check there only
def GOAL_TEST(self,arr):
self.node=arr
if self.node == self.goal:
return True
else:
return False
def INITIAL_STATE(self):
array= [ [[0,2,1],[3,4,5],[6,7,8]], [[1,2,3],[4,5,6],[0,7,8]], [[8,0,2],[4,3,1],[7,6,5]],
[[1,2,0],[3,4,5],[6,7,8]], [[8,1,2],[4,3,5],[7,6,0]], [[5,6,7],[8,3,4],[1,2,0]] ]
# array[4] result after 11 iterations (time 0:0:07.35, 1-11.6)
# array[5] result after 15 iterations (time 0:4:57.39 ,1-15.4)
#self.node=array[np.random.randint(0,6)]#for generating random index between 0 to 5
(but it changes every depth)
self.node= array[4]
print("\nStarting state is:",self.node)
return self.node

```

```

#creating starting board 8-puzzle
"NOTE: But it can't be used due to unsolvable case might generate."
def RandomArray():
arr=np.zeros([3,3],dtype=int)
for i in range(3): #generate random 2D array of size 3x3 with one number repeated (form
1 to 8)
for j in range(3):
k=0;x=np.random.randint(1,9)
while(x in arr and k<100):
x=np.random.randint(1,9);k+=1
arr[i][j]=x
dic={} #for finding repeated number from that 2D array (It can't be done by predefined
functions)
for i in range(3): #creating dictionary for keeping count
for j in range(3):
x=arr[i][j]
dic[x]=0 if x not in dic else 1
for key, value in dic.items(): # finding that number
if value==1:
num=key

```

```

break
x,y=np.where(arr==num) #finding index of the number in the array
print(arr)
print(x,y)
arr[x[0]][x[1]]=0 # replacing that number with zero
print("\nstarting Array:\n",arr)
return [arr] # finally, we got a random array with 0 to 8 number in 3x3 matrix

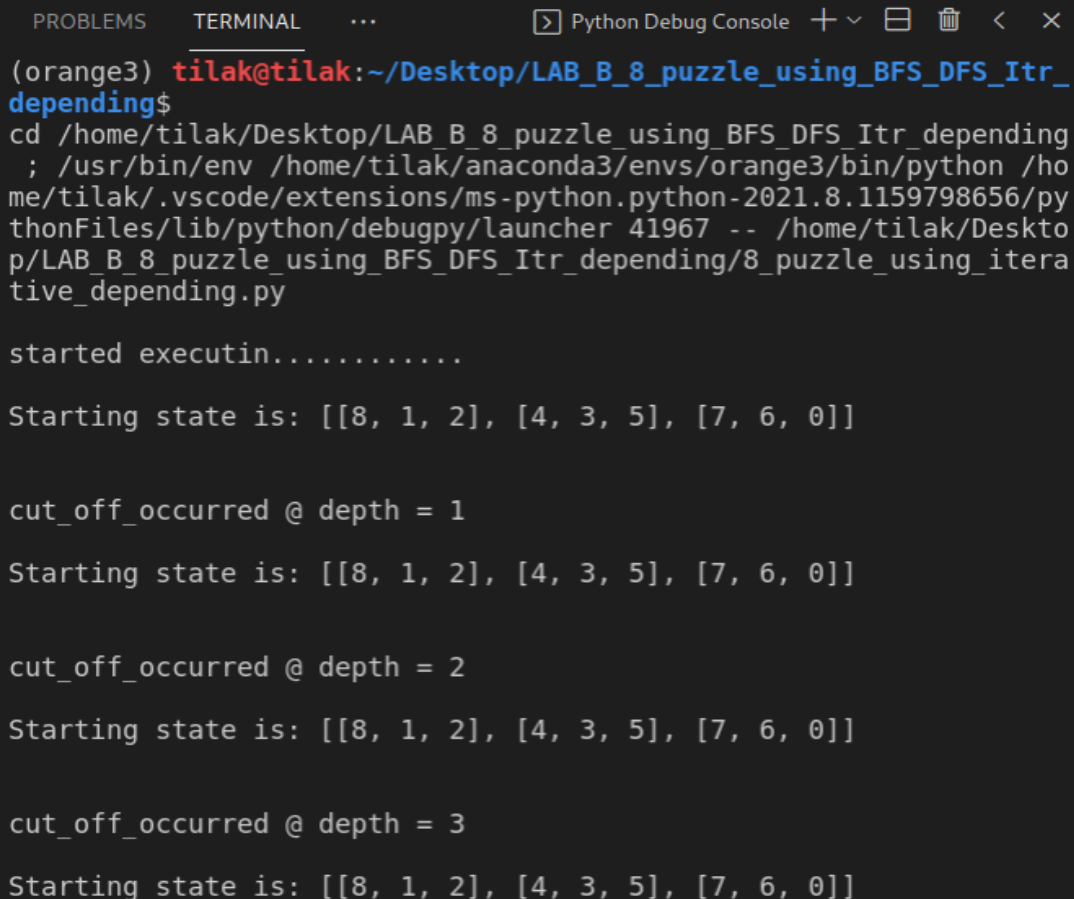
```

```

import numpy as np
from copy import deepcopy
if __name__=="__main__":
    print("\nstarted executin.....")
    problem = n_puzzle() #object
    print("\nIDS result: ",IDS(problem))
    # RandomArray()

```

## OUTPUT:



```

(orange3) tilak@tilak:~/Desktop/LAB_B_8_puzzle_using_BFS_DFS_Itr_dependin$
cd /home/tilak/Desktop/LAB_B_8_puzzle_using_BFS_DFS_Itr_dependin
; /usr/bin/env /home/tilak/anaconda3/envs/orange3/bin/python /home/tilak/.vscode/extensions/ms-python.python-2021.8.1159798656/pythonFiles/lib/python/debugpy/launcher 41967 -- /home/tilak/Desktop/LAB_B_8_puzzle_using_BFS_DFS_Itr_dependin/8_puzzle_using_iterative_dependin.py

started executin.....

Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

cut_off_occurred @ depth = 1
Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

cut_off_occurred @ depth = 2
Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

cut_off_occurred @ depth = 3
Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

```

```

cut_off_occurred @ depth = 4
Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

cut_off_occurred @ depth = 5
Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

```



```
cut_off_occurred @ depth = 6
Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

cut_off_occurred @ depth = 7
Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

cut_off_occurred @ depth = 8
Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

cut_off_occurred @ depth = 9
Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

cut_off_occurred @ depth = 10
Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

cut_off_occurred @ depth = 11
Starting state is: [[8, 1, 2], [4, 3, 5], [7, 6, 0]]

Wow! finally, You have done your Job :)

Your solution is:
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]

IDS result: Result found
(orange3) tilak@tilak:~/Desktop/LAB_B_8_puzzle_using_BFS_DF(orang
e3) tilak@tilak:~/Desktop/LAB_B_8_puzzle_using_BFS_(orange3) tila
(orange3) tilak@tilak:~/Desktop/LAB_B_8_puzzle_using_BFS_DFS_Itr
_depending$
```

Ln 124, Col 20 (4514 selected) Spaces: 4 UTF-8 LF Python

## CONCLUSION:

Hence, we implemented Iterative DFS to solve 8 puzzle problem and reached the destination at step 11 as shown in the output above.