```
import time

#Node to store state , index and parent index
class Node:
    #constructor
    def __init__(self, state, index, parent_index):
        # 3x3 matrix representing the state of the node
        self.Node_State_i = [row[:] for row in state]
        #Index of node
        self.Node_Index_i = index
        #Index of PArent Node
        self.Parent_Node_Index_i = parent_index
    def clone(self):
        return Node(self.Node_State_i, self.Node_Index_i, self.Parent_Node_Index_i)

#Function to check for final state
def check_final_state(Node):
    counter = 0
    #Loop to check final state
    for i in range(0,3):
        for j in range(0,3):
            counter+=1
            if(i!=2 or j!=2):
                if(Node.Node_State_i[i][j] != counter):
                    return False #if not matching with final state
    return True

#Functiont avoid repeating patterns by checking all visited nodes
def check_repeating(Node):
    global visited_nodes
    for index, val in enumerate(visited_nodes):
        flag = False
        counter =0
        i=0
        while(i<3 and flag==False):
            for j in range(0,3):
                if(visited_nodes[index].Node_State_i[i][j] != Node.Node_State_i[i]
[j]):
                    flag = True
                    break
                else:
                    counter += 1
                if(counter == 8):
                    return True
            i+=1
    return False

#Function for locating blank tile with subfunctions of left,right ,up and down
def locate_blank_tile(currentNode_i):
    for i in range(0,3):
        for j in range(0,3):
            if(currentNode_i.Node_State_i[i][j] == 0):
                blank_tile = (i,j)

    #Index counter
    global index_counter

    #Sub Function to check if a blank tile can move left
    def ActionMoveLeft(currentNode):
```

```
        global index_counter
        #If column is not 0 then left movement is possible
        if(blank_tile[1]!=0):
            index_counter += 1
            #Setting children = currenttile
            NewNode = Node(currentNode.Node_State_i, 0, 0)
            #Swapping empty tile with left one
            NewNode.Node_State_i[blank_tile[0]][blank_tile[1]] =
NewNode.Node_State_i[blank_tile[0]][blank_tile[1]-1]
            NewNode.Node_State_i[blank_tile[0]][blank_tile[1]-1] = 0
            #Setting Index and Parent Node
            NewNode.Node_Index_i = index_counter
            NewNode.Parent_Node_Index_i = currentNode.Node_Index_i
            #Left Movement is possible therefor status is true
            status = True
            return [status, NewNode]
        else:
            status = False
            return [status, None]

    #Sub Function to check if a blank tile can move right
    def ActionMoveRight(currentNode):
        global index_counter
        #If column is not 2 then right movement is possible
        if(blank_tile[1]!=2):
            index_counter += 1
            #Setting children = currenttile
            NewNode = Node(currentNode.Node_State_i, 0, 0)
            #Swapping empty tile with right one
            NewNode.Node_State_i[blank_tile[0]][blank_tile[1]] =
NewNode.Node_State_i[blank_tile[0]][blank_tile[1]+1]
            NewNode.Node_State_i[blank_tile[0]][blank_tile[1]+1] = 0
            #Setting Index and Parent Node
            NewNode.Node_Index_i = index_counter
            NewNode.Parent_Node_Index_i = currentNode.Node_Index_i
            #Right Movement is possible therefor status is true
            status = True
            return [status, NewNode]
        else:
            status = False
            return [status, None]


    #Sub Function to check if a blank tile can move up
    def ActionMoveUp(currentNode):
        global index_counter
        #If Row is not 0 then up movement is possible
        if(blank_tile[0]!=0):
            index_counter += 1
            #Setting children = currenttile
            NewNode = Node(currentNode.Node_State_i, 0, 0)
            #Swapping empty tile with upmost one
            NewNode.Node_State_i[blank_tile[0]][blank_tile[1]] =
NewNode.Node_State_i[blank_tile[0]-1][blank_tile[1]]
            NewNode.Node_State_i[blank_tile[0]-1][blank_tile[1]] = 0
            #Setting Index and Parent Node
            NewNode.Node_Index_i = index_counter
            NewNode.Parent_Node_Index_i = currentNode.Node_Index_i
            #Upmost Movement is possible therefor status is true
```

```
            status = True
            return [status, NewNode]
        else:
            status = False
            return [status, None]


    #Sub Function to check if a blank tile can move down
    def ActionMoveDown(currentNode):
        global index_counter
        #If Row is not 2 then down movement is possible
        if(blank_tile[0]!=2):
            index_counter += 1
            #Setting children = currenttile
            NewNode = Node(currentNode.Node_State_i, 0, 0)
            #Swapping empty tile with downmost one
            NewNode.Node_State_i[blank_tile[0]][blank_tile[1]] =
NewNode.Node_State_i[blank_tile[0]+1][blank_tile[1]]
            NewNode.Node_State_i[blank_tile[0]+1][blank_tile[1]] = 0
            #Setting Index and Parent Node
            NewNode.Node_Index_i = index_counter
            NewNode.Parent_Node_Index_i = currentNode.Node_Index_i
            #Upside Movement is possible therefor status is true
            status = True
            return [status, NewNode]
        else:
            status = False
            return [status, None]


    Moved_Nodes = []

    #Calling movement nodes

    MoveLeft  = ActionMoveLeft(currentNode_i)
    #Checking if status is true
    if(MoveLeft[0] == True):
        #Check if the children is final state
        if(check_final_state(MoveLeft[1]) == True):
            Moved_Nodes.append(MoveLeft[1])
            return True, Moved_Nodes

        else:
            #Check repeating nodes
            if(check_repeating(MoveLeft[1]) == False):
                Moved_Nodes.append(MoveLeft[1])
            else:
                index_counter -= 1

    MoveRight = ActionMoveRight(currentNode_i)
    if(MoveRight[0] == True):
        if(check_final_state(MoveRight[1]) == True):
            Moved_Nodes.append(MoveRight[1])
            return True, Moved_Nodes
        else:
            if(check_repeating(MoveRight[1]) == False):
                Moved_Nodes.append(MoveRight[1])
            else:
                index_counter -= 1
```

```
    MoveUp    = ActionMoveUp(currentNode_i)
    if(MoveUp[0] == True):
        if(check_final_state(MoveUp[1]) == True):
            Moved_Nodes.append(MoveUp[1])
            return True, Moved_Nodes
        else:
            if(check_repeating(MoveUp[1]) == False):
                Moved_Nodes.append(MoveUp[1])
            else:
                index_counter -= 1

    MoveDown  = ActionMoveDown(currentNode_i)
    if(MoveDown[0] == True):
        if(check_final_state(MoveDown[1]) == True):
            Moved_Nodes.append(MoveDown[1])
            return True, Moved_Nodes
        else:
            if(check_repeating(MoveDown[1]) == False):
                Moved_Nodes.append(MoveDown[1])
            else:
                index_counter -= 1

    return False, Moved_Nodes


#BFS Algorithm
def bfs(initial_node):
    global list_nodes

    #Defining explored node to store all explored nodes
    global visited_nodes

    #Removing root node and storing it in explored
    removed_node = list_nodes.pop(0)
    visited_nodes.append(removed_node)
    FinalReached, Children_Nodes = locate_blank_tile(removed_node)

    list_nodes = list_nodes + Children_Nodes

    #Function to backtrack the initial node form first node
    def generate_path():
        output_nodes = []
        total_nodes = visited_nodes + list_nodes
        i = len(total_nodes)-1
        output_nodes.append(total_nodes[i])
        i=i-1
        index=0
        while total_nodes[i].Node_Index_i != 0:
            current_node = output_nodes[index]
            if(total_nodes[i].Node_Index_i == current_node.Parent_Node_Index_i):
                output_nodes.append(total_nodes[i])
                index += 1
            i -= 1
        output_nodes.append(initial_node)

        output_nodes.reverse()
```

```
print("==============================Output==================================")
        for i, val in enumerate(output_nodes):
            for row in output_nodes[i].Node_State_i:
                for element in row:
                    print(element, end=' ')
                print()  # Move to the next line after printing each row
            print("\n")

print("============================End=========================================")


        for i, val in enumerate(output_nodes):
            print(output_nodes[i].Node_Index_i, end=' ')

        #Writing to text files
        # Open a text file in write mode ('w')
        with open("Nodes.txt", "w") as file:
            # Write some text to the file
            for i, val in enumerate(output_nodes):
                for row in output_nodes[i].Node_State_i:
                    for element in row:
                        file.write(str(element))
                        file.write(" ")
                file.write("\n")

        # Open a text file in write mode ('w')
        with open("NodesInfo.txt", "w") as file:
            file.write("Node_Index   Parent_Node_Index          Node\n")
            # Write some text to the file
            for i, val in enumerate(output_nodes):
                file.write(str(output_nodes[i].Node_Index_i))
                if(output_nodes[i].Node_Index_i>9):
                    file.write("                      ")
                else:
                    file.write("                       ")
                file.write(str(output_nodes[i].Parent_Node_Index_i))
                file.write("           ")
                for row in output_nodes[i].Node_State_i:
                    for element in row:
                        file.write(str(element))
                        file.write(" ")
                file.write("\n")

        # Open a text file in write mode ('w')
        with open("nodePath.txt", "w") as file:
            # Write some text to the file
            for index, val in enumerate(output_nodes):
                for i in range(0,3):
                    for j in range(0,3):
                        element = output_nodes[index].Node_State_i[j][i]
                        file.write(str(element))
                        file.write(" ")
                file.write("\n")


    if(FinalReached==True):
        generate_path()
```

```
    return FinalReached


#Input State of 8 puzzle
#1 0 3
#4 5 6
#7 8 2

Node_State_1 = [[1, 0, 3], [4, 5, 6], [7, 8, 2]]

#Index of root node is 1
Node_Index_1 = 0

#Parent node is None since it is the parent node
Parent_Node_Index_1 = 0

#Creating Root node with initial state
node1 = Node(Node_State_1, Node_Index_1, Parent_Node_Index_1)

#Using List datastructure to store all the Nodes
list_nodes = []

#Adding root node to dict
list_nodes.append(node1)

visited_nodes = []

index_counter = Node_Index_1

#Main func
if __name__ == "__main__":
    start = time.time()
    while(True):
        if(bfs(node1) ==  True):
            break #break when final state reached
    end = time.time()
    #Time
    print("\nElapsed time for 8 puzzle is",(end-start))
```