Lab 2: 8 Puzzle Single Player Game (BFS)

In [1]:

```
#Import the necessary libraries
from time import time
from queue import Queue
```

In [2]:

```
#Creating a class Puzzle
class Puzzle:
    #Setting the goal state of 8-puzzle
    goal_state=[1,2,3,8,0,4,7,6,5]
    num_of_instances=0
    #constructor to initialize the class members
    def __init__(self,state,parent,action):
        self.parent=parent
        self.state=state
        self.action=action
        #TODO: incrementing the number of instance by 1
        Puzzle.num_of_instances+= 1

    #function used to display a state of 8-puzzle
    def __str__(self):
        return str(self.state[0:3])+'\n'+str(self.state[3:6])+'\n'+str(self.state[6

    #method to compare the current state with the goal state
    def goal_test(self):
        #TODO: include a condition to compare the current state with the goal state
        if self.state == Puzzle.goal_state:
            return True
        return False

    #static method to find the legal action based on the current board position
    @staticmethod
    def find_legal_actions(i,j):
        legal_action = ['U', 'D', 'L', 'R']
        if i == 0:
            # if row is 0 in board then up is disable
            legal_action.remove('U')
        elif i == 2:
            legal_action.remove('D')
        if j == 0:
            legal_action.remove('L')
        elif j == 2:
            legal_action.remove('R')

        return legal_action

    #method to generate the child of the current state of the board
    def generate_child(self):
        #TODO: create an empty list
        children=[]
        x = self.state.index(0)
        i = int(x / 3)
        j = int(x % 3)
        #TODO: call the method to find the legal actions based on i and j values
        legal_actions = Puzzle.find_legal_actions(i,j)

        #TODO:Iterate over all legal actions
        for action in legal_actions:
            new_state = self.state.copy()
            #if the legal action is UP
            if action=='U':
                #Swapping between current index of 0 with its up element on the boa
                new_state[x], new_state[x-3] = new_state[x-3], new_state[x]
            elif action=='D':
                #TODO: Swapping between current index of 0 with its down element on
```

```
                new_state[x], new_state[x+3] = new_state[x+3], new_state[x]
            elif action=='L':
                #TODO: Swapping between the current index of 0 with its left elemen
                new_state[x], new_state[x-1] = new_state[x-1], new_state[x]
            elif action=='R':
                #TODO: Swapping between the current index of 0 with its right eleme
                new_state[x], new_state[x+1] = new_state[x+1], new_state[x]
            children.append(Puzzle(new_state,self,action))
        #TODO: return the children
        return children
    #method to find the solution
    def find_solution(self):
        solution = []
        solution.append(self.action)
        path = self
        while path.parent != None:
            path = path.parent
            solution.append(path.action)
        solution = solution[:-1]
        solution.reverse()
        return solution
```

In [3]:

```
#method for breadth first search
#TODO: pass the initial_state as parameter to the breadth_first_search method
def breadth_first_search(initial_state):
    start_node = Puzzle(initial_state, None, None)
    print("Initial state:")
    print(start_node)
    if start_node.goal_test():
        return start_node.find_solution()
    q = Queue()
    #TODO: put start_node into the Queue
    q.put(start_node)
    #TODO: create an empty list of explored nodes
    explored=[]
    #TODO: Iterate the queue until empty. Use the empty() method of Queue
    while not(q.empty()):
        #TODO: get the current node of a queue. Use the get() method of Queue
        node=q.get()
        #TODO: Append the state of node in the explored list as node.state
        explored.append(node.state)
        #TODO: call the generate_child method to generate the child nodes of curren
        children=Puzzle.generate_child(node)
        #TODO: Iterate over each child node in children
        for child in children:
            if child.state not in explored:
                if child.goal_test():
                    return child.find_solution()
                q.put(child)
    return
```

In [4]:

```
#Start executing the 8-puzzle with setting up the initial state
#Here we have considered 3 initial state intitalized using state variable
state=[[1, 3, 4,
        8, 6, 2,
        7, 0, 5],

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

       [2, 8, 1,
        4, 6, 3,
        0, 7, 5]]
#Iterate over number of initial_state
for i in range(0,3):
    #TODO: Initialize the num_of_instances to zero
    Puzzle.num_of_instances=0
    #Set t0 to current time
    t0=time()
    bfs=breadth_first_search(state[i])
    #Get the time t1 after executing the breadth_first_search method
    t1=time()-t0
    print('BFS:', bfs)
    print('space:',Puzzle.num_of_instances)
    print('time:',t1)
    print()
print('----------------------------------------')
```

```
Initial state:
[1, 3, 4]
[8, 6, 2]
[7, 0, 5]
BFS: ['U', 'R', 'U', 'L', 'D']
space: 66
time: 0.0020859241485595703

Initial state:
[2, 8, 1]
[0, 4, 3]
[7, 6, 5]
BFS: ['U', 'R', 'R', 'D', 'L', 'L', 'U', 'R', 'D']
space: 591
time: 0.019245624542236328

Initial state:
[2, 8, 1]
[4, 6, 3]
[0, 7, 5]
BFS: ['R', 'U', 'L', 'U', 'R', 'R', 'D', 'L', 'L', 'U', 'R', 'D']
space: 2956
time: 0.06805133819580078

----------------------------------------
```