Experiment 1:

AIM: To write the python program to solve 8-Puzzle problem

Algorithm:

1. **Define Classes and Functions**:
   * Define a class **priorityQueue** to implement the priority queue data structure using heapq module.
   * Define a class **nodes** to represent each node in the search tree.
   * Define functions for calculating costs, creating new nodes, printing matrix, checking safety, and printing the path.
2. **Calculate Costs Function** (**calculateCosts**):
   * Calculates the cost (number of misplaced tiles) between the current state and the final state.
3. **Create New Nodes Function** (**newNodes**):
   * Generates new nodes by swapping empty tiles with adjacent tiles and calculates the costs for each new node.
4. **Print Matrix Function** (**printMatrix**):
   * Prints the matrix representation.
5. **Safety Check Function** (**isSafe**):
   * Checks if a move is safe or not (within the bounds of the puzzle).
6. **Print Path Function** (**printPath**):
   * Recursively prints the path from the root node to the final node.
7. **Solve Function** (**solve**):
   * Implements the A\* search algorithm to find the solution.
   * Initializes the priority queue, creates the root node, and pushes it into the priority queue.
   * Continues until the priority queue is not empty:
     + Pops the node with the minimum cost from the priority queue.
     + If the cost is 0, prints the path.
     + Generates child nodes for valid moves and adds them to the priority queue.

Code:

import copy

from heapq import heappush, heappop

n = 3

rows = [ 1, 0, -1, 0 ]

cols = [ 0, -1, 0, 1 ]

class priorityQueue:

def \_\_init\_\_(self):

self.heap = []

def push(self, key):

heappush(self.heap, key)

def pop(self):

return heappop(self.heap)

def empty(self):

if not self.heap:

return True

else:

return False

class nodes:

def \_\_init\_\_(self, parent, mats, empty\_tile\_posi,

costs, levels):

self.parent = parent

self.mats = mats

self.empty\_tile\_posi = empty\_tile\_posi

self.costs = costs

self.levels = levels

def \_\_lt\_\_(self, nxt):

return self.costs < nxt.costs

def calculateCosts(mats, final) -> int:

count = 0

for i in range(n):

for j in range(n):

if ((mats[i][j]) and

(mats[i][j] != final[i][j])):

count += 1

return count

def newNodes(mats, empty\_tile\_posi, new\_empty\_tile\_posi,

levels, parent, final) -> nodes:

new\_mats = copy.deepcopy(mats)

x1 = empty\_tile\_posi[0]

y1 = empty\_tile\_posi[1]

x2 = new\_empty\_tile\_posi[0]

y2 = new\_empty\_tile\_posi[1]

new\_mats[x1][y1], new\_mats[x2][y2] = new\_mats[x2][y2], new\_mats[x1][y1]

costs = calculateCosts(new\_mats, final)

new\_nodes = nodes(parent, new\_mats, new\_empty\_tile\_posi,

costs, levels)

return new\_nodes

def printMatsrix(mats):

for i in range(n):

for j in range(n):

print("%d " % (mats[i][j]), end = " ")

print()

def isSafe(x, y):

return x >= 0 and x < n and y >= 0 and y < n

# Printing the path from the root node to the final node

def printPath(root):

if root == None:

return

printPath(root.parent)

printMatsrix(root.mats)

print()

def solve(initial, empty\_tile\_posi, final):

pq = priorityQueue()

costs = calculateCosts(initial, final)

root = nodes(None, initial,

empty\_tile\_posi, costs, 0)

pq.push(root)

while not pq.empty():

minimum = pq.pop()

if minimum.costs == 0:

printPath(minimum)

return

for i in range(n):

new\_tile\_posi = [

minimum.empty\_tile\_posi[0] + rows[i],

minimum.empty\_tile\_posi[1] + cols[i], ]

if isSafe(new\_tile\_posi[0], new\_tile\_posi[1]):

# Creating a child node

child = newNodes(minimum.mats,

minimum.empty\_tile\_posi,

new\_tile\_posi,

minimum.levels + 1,

minimum, final,)

# Adding the child to the list of live nodes

pq.push(child)

initial = [ [ 1, 2, 3 ],

[ 5, 6, 0 ],

[ 7, 8, 4 ] ]

final = [ [ 1, 2, 3 ],

[ 5, 8, 6 ],

[ 0, 7, 4 ] ]

empty\_tile\_posi = [ 1, 2 ]

solve(initial, empty\_tile\_posi, final)

INPUT: initial = [ [ 1, 2, 3 ],

[ 5, 6, 0 ],

[ 7, 8, 4 ] ]

final = [ [ 1, 2, 3 ],

[ 5, 8, 6 ],

[ 0, 7, 4 ] ]

empty\_tile\_posi = [ 1, 2 ]

output: 1 2 3

5 6 0

7 8 4

1 2 3

5 0 6

7 8 4

1 2 3

5 8 6

7 0 4

1 2 3

5 8 6

0 7 4

Total Cost=4

RESULT: Hence the output is verified.