**EXPERIMENT NO : 08 DATE : 16/04/24**

**Aim**: Implement 8 Puzzle Problem using Best First Search Algorithm

**Theory**:

It has set off a 3x3 board having 9 block spaces out of which 8 blocks having tiles bearing number from 1 to 8. One space is left blank. The tile adjacent to blank space can move into it. We have to arrange the tiles in a sequence for getting the goal state”. The 8-puzzle problem is invented and popularized by Noyes Palmer Chapman in the year 1870. It is played on a 3-by-3 grid with 8 square locks/tiles labelled 1 through 8 and a blank square. The goal of this 8-puzzle problem is to rearrange the given blocks in the correct order. The tiles can be shifted vertically or horizontally using the empty square.

Best-First Search:

We now describe an algorithmic solution to the problem that illustrates a general artificial intelligence methodology known as the A\* search algorithm. We define a state of the game to be the board position, the number of moves made to reach the board position, and the previous state. First, insert the initial state (the initial board, 0 moves, and a null previous state) into a priority queue. Then, delete from the priority queue the state with the minimum priority, and insert onto the priority queue all neighbouring states (those that can be reached in one move). Repeat this procedure until the state dequeued is the goal state. The success of this approach hinges on the choice of priority function for a state. We consider two priority functions:

1. Hamming priority function: The number of blocks in the wrong position, plus the number of

moves made so far to get to the state. Intuitively, a state with a small number of blocks in the

wrong position is close to the goal state, and we prefer a state that have been reached using a

small number of moves.

2. Manhattan priority function: The sum of the distances (sum of the vertical and horizontal

distance) from the blocks to their goal positions, plus the number of moves made so far to get to

the state.

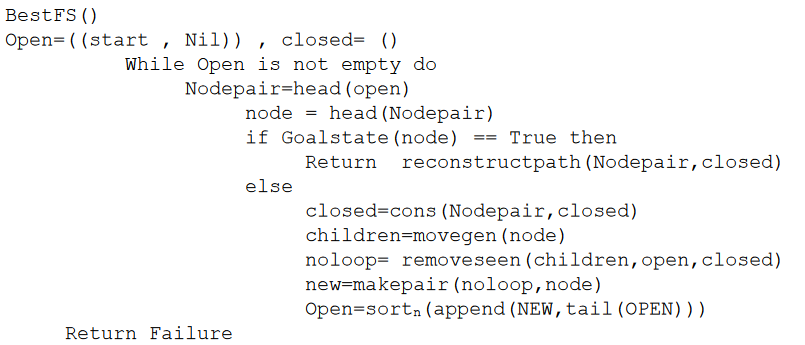
Rules Of Solving Puzzle:

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) Up , Down ,Right

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

**Algorithm:**



**Code:**

start\_state = [[1, 2, 3],

[5, 6, 0],

[7, 8, 4]]

goal\_state = [[1, 2, 3],

[5, 8, 6],

[0, 7, 4]]

def print\_state(state):

for row in state:

print(row)

def get\_blank\_position(state):

for i in range(3):

for j in range(3):

if state[i][j] == 0:

return i, j

def gen\_moves(state):

moves = []

i, j = get\_blank\_position(state)

if i == 0:

if j == 0:

moves.append((i, j + 1))

moves.append((i + 1, 0))

elif j == 1:

moves.append((i, j - 1))

moves.append((i, j + 1))

moves.append((i + 1, j))

elif j == 2:

moves.append((i, j - 1))

moves.append((i + 1, j))

elif i == 1:

if j == 0:

moves.append((i, j + 1))

moves.append((i - 1, j))

moves.append((i + 1, j))

elif j == 1:

moves.append((i, j - 1))

moves.append((i, j + 1))

moves.append((i + 1, j))

moves.append((i - 1, j))

elif j == 2:

moves.append((i, j - 1))

moves.append((i + 1, j))

moves.append((i - 1, j))

elif i == 2:

if j == 0:

moves.append((i - 1, j))

moves.append((i, j + 1))

elif j == 1:

moves.append((i, j - 1))

moves.append((i, j + 1))

moves.append((i - 1, j))

elif j == 2:

moves.append((i, j - 1))

moves.append((i - 1, j))

return moves

def make\_move(state, moves):

i, j = get\_blank\_position(state)

states = []

for new\_i, new\_j in moves:

current = [row[:] for row in state]

current[i][j], current[new\_i][new\_j] = current[new\_i][new\_j], current[i][j]

states.append(current)

return states

def manhattan\_distance(state, goal\_state):

total\_distance = 0

# Create dictionaries to store the position of each number in the goal state

goal\_positions = {}

for i in range(3):

for j in range(3):

goal\_positions[goal\_state[i][j]] = (i, j)

# Calculate Manhattan distance for each tile in the current state

for i in range(3):

for j in range(3):

if state[i][j] != 0: # Ignore the empty tile

num = state[i][j]

goal\_pos = goal\_positions[num]

distance = abs(i - goal\_pos[0]) + abs(j - goal\_pos[1])

total\_distance += distance

return total\_distance

def hill\_climbing(start):

# Define a class to represent a state in the search

class State:

def \_\_init\_\_(self, state, parent=None, heuristic=0):

self.state = state

self.parent = parent

self.heuristic = heuristic

current\_state = State(start\_state) # Start state

print("Initial State: ")

print\_state(current\_state.state)

if start\_state == goal\_state:

print("Goal reached")

return

open = [current\_state] # open to store states to be explored

while open:

open.sort(key=lambda x: x.heuristic) # Sort open based on heuristic values

current\_state = open.pop(0) # Pop the state with the lowest heuristic value

print("Current State: ")

print\_state(current\_state.state)

print("Heuristic value = " + str(current\_state.heuristic))

if current\_state.state == goal\_state:

print("Goal state found.")

print("Final state: ")

print\_state(goal\_state)

break

moves = gen\_moves(current\_state.state)

for move in moves:

new\_state = make\_move(current\_state.state, [move])[0]

heuristic\_value = manhattan\_distance(new\_state, goal\_state)

new\_state\_obj = State(new\_state, current\_state, heuristic\_value)

open.append(new\_state\_obj)

# Trace back the path to the start state

path = []

while current\_state:

path.append(current\_state.state)

current\_state = current\_state.parent

# Reverse the path to get the correct sequence of steps

path.reverse()

print("Sequence of steps to reach the goal state:")

for step in path:

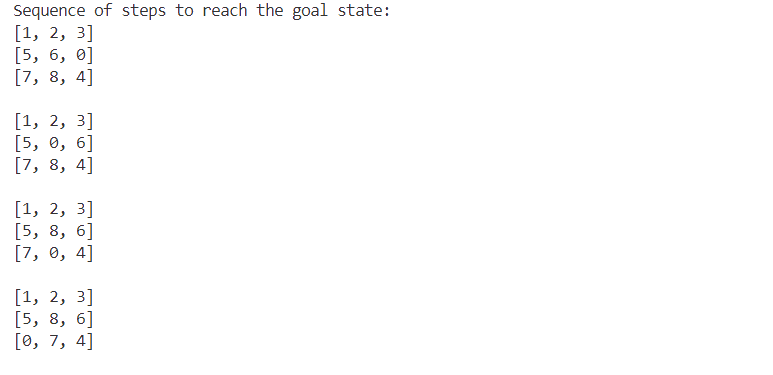
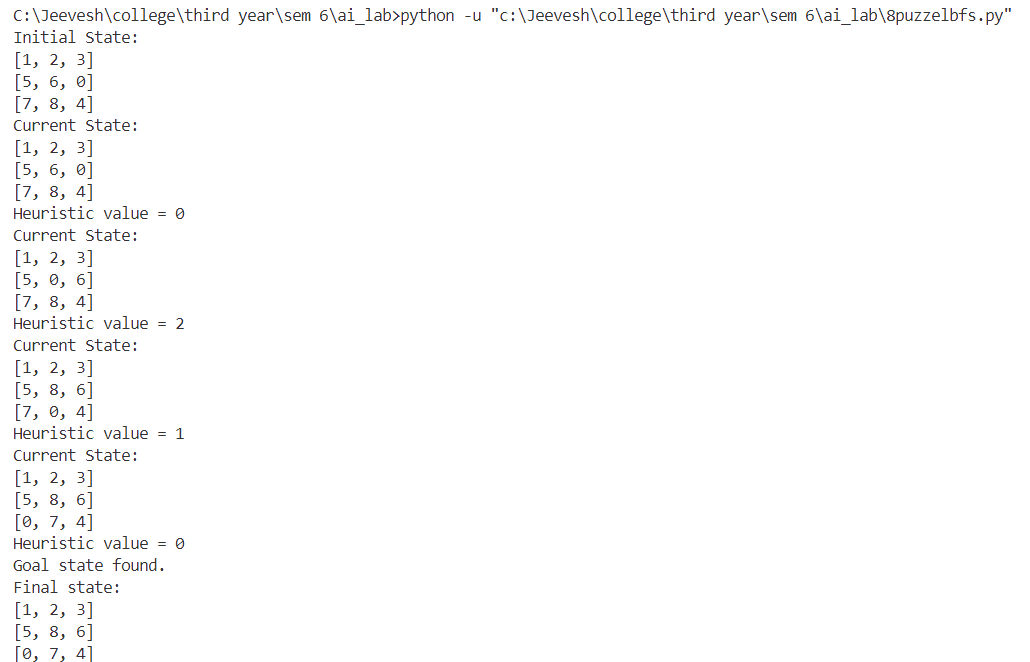
print\_state(step)

print()

# Test the hill climbing function

hill\_climbing(start\_state)

**Output:**

****

**Conclusion:**

Implementation of the 8 Puzzles Problem Using Best First Search was carried out by tracing the algorithm and above output was obtained during the same.