Assignment No:01

CSE-0408 Summer 2021

Md. Khaled Saifullah Sadi
Department of Computer Science and Engineering
State University of Bangladesh (SUB)
Dhaka, Bangladesh
mdsadi4@gmail.com

***Abstract*—The 8-puzzle is the largest puzzle of its type that can be completely solved. It is simple, and yet obeys a combinatorially large problem space of 9!/2 states. The N x N extension of the 8-puzzle is NP-hard.**

n

***Index Terms*—heuristic, 8 puzzle**

I. Introduction

Heuristic is a function which is used in Informed Search, and it finds the most promising path.Many problems, such as game-playing and path-finding, can be solved by search algorithms. To do so, the problems are represented by a search graph or tree in which the nodes correspond to the states of the problem.

II. Literature Review

The 8-puzzle is a prominent workbench model for measuring the performance of heuristic search algorithms [Gaschnig, 1979; Nilsson, 1980; Pearl, 1985; Russell, 1992], learning methods [Laird et a/., 1987] and the use of macro operators [Korf, 1985a].

III. Proposed Methodology

The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. Your goal is to rearrange the blocks so that they are in order

IV. Some Screansoot from Code

```
from copy import deepcopy
from colorama import Fore, Back, Style

DIRECTIONS = {"U": [-1, 0], "D": [1, 0], "L": [0, -1], "R": [0, 1]}
END = [[1, 2, 3], [4, 5, 6], [7, 8, 0]]

# unicode
left_down_angle = '\u2514'
right_down_angle = '\u2518'
right_up_angle = '\u2510'
left_up_angle = '\u250C'

middle_junction = '\u253C'
top_junction = '\u252C'
bottom_junction = '\u2534'
right_junction = '\u2524'
left_junction = '\u251C'

bar = Style.BRIGHT + Fore.CYAN + '\u2502' + Fore.RESET + Style.RESET_ALL
dash = '\u2500'

first_line = Style.BRIGHT + Fore.CYAN + left_up_angle + dash + dash + dash + top_junction + dash + dash + dash + top_junctior
middle_line = Style.BRIGHT + Fore.CYAN + left_junction + dash + dash + dash + middle_junction + dash + dash + dash + middle_
last_line = Style.BRIGHT + Fore.CYAN + left_down_angle + dash + dash + dash + bottom_junction + dash + dash + dash + bottom_


def print_puzzle(array):
    print(first_line)
    for a in range(len(array)):
        for i in array[a]:
            if i == 0:
                print(bar, Back.RED + ' ' + Back.RESET, end=' ')
            else:
                print(bar, i, end=' ')
        print(bar)
        if a == 2:
            print(last_line)
        else:
            print(middle_line)
```

Fig. 1. Code

```
class Node:
    def __init__(self, current_node, previous_node, g, h, dir):
        self.current_node = current_node
        self.previous_node = previous_node
        self.g = g
        self.h = h
        self.dir = dir

    def f(self):
        return self.g + self.h


def get_pos(current_state, element):
    for row in range(len(current_state)):
        if element in current_state[row]:
            return (row, current_state[row].index(element))


def euclidianCost(current_state):
    cost = 0
    for row in range(len(current_state)):
        for col in range(len(current_state[0])):
            pos = get_pos(END, current_state[row][col])
            cost += abs(row - pos[0]) + abs(col - pos[1])
    return cost


def getAdjNode(node):
    listNode = []
    emptyPos = get_pos(node.current_node, 0)

    for dir in DIRECTIONS.keys():
        newPos = (emptyPos[0] + DIRECTIONS[dir][0], emptyPos[1] + DIRECTIONS[dir][1])
        if 0 <= newPos[0] < len(node.current_node) and 0 <= newPos[1] < len(node.current_node[0]):
            newState = deepcopy(node.current_node)
            newState[emptyPos[0]][emptyPos[1]] = node.current_node[newPos[0]][newPos[1]]
            newState[newPos[0]][newPos[1]] = 0
            # listNode += [Node(newState, node.current_node, node.g + 1, euclidianCost(newState), dir)]
            listNode.append(Node(newState, node.current_node, node.g + 1, euclidianCost(newState), dir))

    return listNode
```

Fig. 2. Code

```
def getBestNode(openSet):
    firstIter = True

    for node in openSet.values():
        if firstIter or node.f() < bestF:
            firstIter = False
            bestNode = node
            bestF = bestNode.f()
    return bestNode


def buildPath(closedSet):
    node = closedSet[str(END)]
    branch = list()

    while node.dir:
        branch.append({
            'dir': node.dir,
            'node': node.current_node
        })
        node = closedSet[str(node.previous_node)]
    branch.append({
        'dir': '',
        'node': node.current_node
    })
    branch.reverse()

    return branch


def main(puzzle):
    open_set = {str(puzzle): Node(puzzle, puzzle, 0, euclidianCost(puzzle), "")}
    closed_set = {}

    while True:
        test_node = getBestNode(open_set)
        closed_set[str(test_node.current_node)] = test_node

        if test_node.current_node == END:
            return buildPath(closed_set)
```

Fig. 3. Code

V. Some Screansoot from OutPut

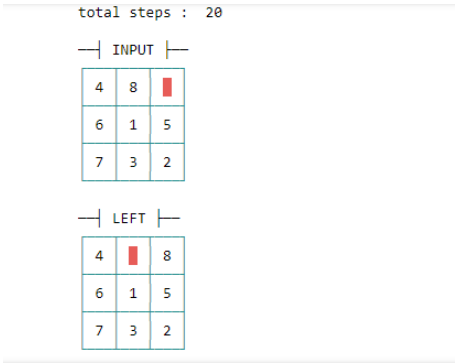

Fig. 1. Output

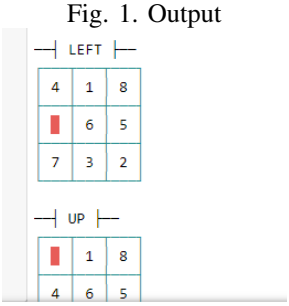

Fig. 2. Output

VI. Conclusion

We tested our code to see how many states it would take to get from the current state to the goal state, and we came up with seven.

Acknowledgment

I would like to thank my honourable**Khan Md. Hasib Sir** for his time, generosity and critical insights into this project.

References

[1] Piltaver, R., Lustrek, M., and Gams, M. (2012). The pathology of heuristic ˇ search in the 8-puzzle. Journal of Experimental and Theoretical Artificial Intelligence, 24(1), 65-94