Artificial Intelligence

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_junctio
middle_line = Style.BRIGHT + Fore.CYAN + left_junction + dash + dash + dash + middle_junction + dash + dash + dash + middle_j
last_line = Style.BRIGHT + Fore.CYAN + left_down_angle + dash + dash + dash + bottom_junction + dash + dash + dash + bottom_j


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

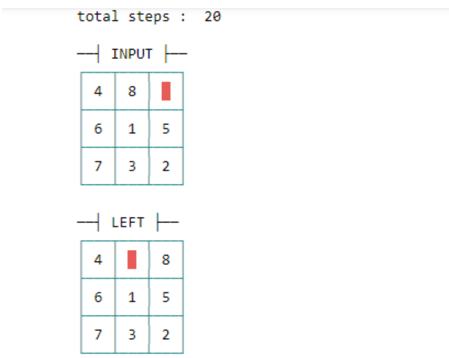

Fig. 1. Output

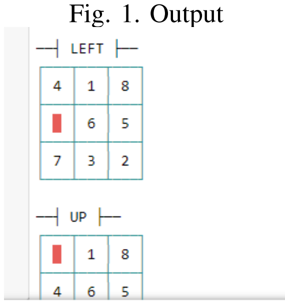

Fig. 2. Output

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

Assignment No:02

***Abstract*—Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.**

***Index Terms*—BFS, Shortest Path**

VII. Introduction

Breadth First Search (BFS) is an algorithm for traversing or searching layerwise in tree or graph data structures.If we consider searching as a form of traversal in a graph, an uninformed search algorithm would blindly traverse to the next node in a given manner without considering the cost associated with that step

VIII. Literature Review

Breadth First Search (BFS) is a prominent workbench model for measuring the performance of heuristic search algorithms [Nilsson, 1969; Gaschnig, 1970; Pearl, 1995; Fardeen, 1992], learning methods [Laird et a/., 1977] and the use of macro operators [Korf, 1995a].

IX. Proposed Methodology

.Begin the search algorithm, by knowing the key which is to be searched. Once the key/element to be searched is decided the searching begins with the root (source) first

1) procedure BFS(G, root) is
2) let Q be a queue
3) label root as explored
4) Q.enqueue(root)
5) while Q is not empty do
6) v := Q.dequeue()
7) if v is the goal then
8) return v
9) for all edges from v to w in
10) for all edges from v to w in
11) G.adjacentEdges(v) do
12) if w is not labeled as explored then
13) label w as explored
14) Q.enqueue(w)

We are using NetworkX for creating Graph. NetworkX is a Python package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks(Graph). Also using matplotlib.pyplot, we can graphically represent our graph.

X. Some Screansoot from Code

```
import networkx as nx
import matplotlib.pyplot as plt
from collections import deque
import random

def CreateGraph(node, edge):
    G = nx.Graph()
    for i in range(1, node+1):
        G.add_node(i)
    for i in range(edge):
        u, v = random.randint(1, node), random.randint(1, node)
        G.add_edge(u, v)
    return G

def DrawGraph(G, color):
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels = True, node_color = color, edge_color = 'black' ,width = 1, alpha = 0.7)

def DrawIteratedGraph(G,col_val):
    pos = nx.spring_layout(G)
    color = ["green", "blue", "yellow", "pink", "red", "black", "gray", "brown", "orange", "plum"]
    values = []
    for node in G.nodes():
        values.append(color[col_val[node]])
    nx.draw(G, pos, with_labels = True, node_color = values, edge_color = 'black' ,width = 1, alpha = 0.7)

def BFS(start):
    queue = deque()
    queue.append(start)
    visited[start] = True
    level[start] = 0

    while queue:
        u = queue.popleft()
        print(u, " -> ", end = "")
        for v in G.adj[u]:
            if not visited[v]:
                queue.append(v)
                visited[v] = True
                level[v] = level[u] + 1
```

Fig. 1. Code

```
        DrawIteratedGraph(G, level)
        plt.title('From {}:'.format(u), loc='left')
        plt.title('Level {}:'.format(level[u]), loc='right')
        plt.show()

    print("End")

if __name__ == "__main__":

    print("Enter no of Node")
    node = int(input())
    print("Enter no of Edges")
    edge = int(input())

    G = CreateGraph(node, edge)
    print("Nodes: ", G.nodes)
    DrawGraph(G, "green")
    plt.show()
    visited = [False for i in range(node+1)]
    level = [0 for i in range(node+1)]
    parent = [0 for i in range(node+1)]
    root = 1
    BFS(root)
```

Fig. 2. Code

XI. Some Screansoot from OutPut

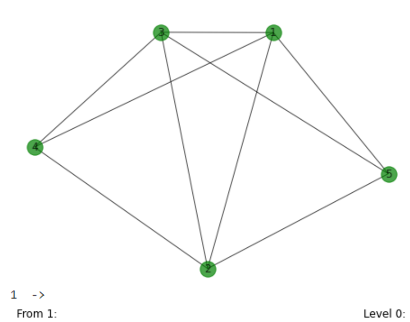

Fig. 1. Output

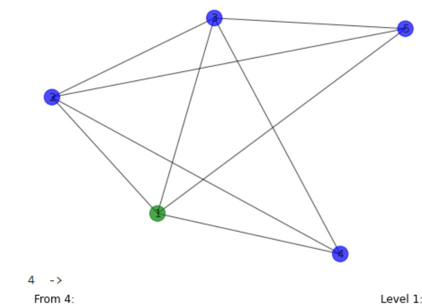

Fig. 2. Output

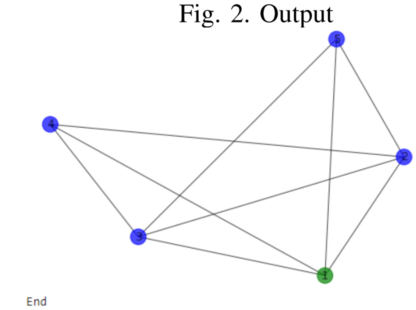

Fig. 2. Output

XII. Conclusion

The BFS algorithm is useful for analyzing the nodes in a graph and constructing the shortest path of traversing through these.

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. (2015). Breadth First Search (BFS). Journal of Experimental and Theoretical Artificial Intelligence, 24(1), 65-94