Mid Term Report: Heuristic function & Breadth first search

CSE-0408 Summer 2021

Md.Rafiul Alam Durjoy
Department of Computer Science and Engineering
State University of Bangladesh (SUB)
Dhaka, Bangladesh
mdurjoy148@gmail.com

***Abstract*—Heuristic Function is a function that estimates the cost of getting from one place to another (from the current state to the goal state.) Also called as simply a heuristic.**

And Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

***Index Terms*—8-puzzle,BFS,python,C++**

I. Introduction

A. Heuristic Function:

N-Puzzle or sliding puzzle is a popular puzzle that consists of N tiles where N can be 8, 15, 24, and so on. In our example N = 8. The puzzle is divided into sqrt(N+1) rows and sqrt(N+1) columns.An 8-Puzzle will have 3 rows and 3 columns. The puzzle consists of N tiles and one empty space where the tiles can be moved. Start and Goal configurations (also called state) of the puzzle are provided. The puzzle can be solved by moving the tiles one by one in the single empty space and thus achieving the Goal configuration.

B. Breadth First Search:

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.

II. Literature Review

A. Heuristic Function:

The puzzle was invented by Noyes Palmer Chapman, a postmaster in Canastota, New York, who indicated it to friends and companions, as right on time as 1874, a previous puzzle consisting of 16 grids that were to be assembled in 4 rows of equal length, each row adding to 34.

B. Breadth First Search:

The faster presentation of best-first tree width search reflects the difference in size connecting a search tree and a search graph. Unfortunately, the scalability of best-first search is limited by its memory necessities, which tend to grow exponentially with the search depth. To improve scalability, use a memory-efficient version of best-first search called breadth-first heuristic search.

III. Proposed Methodology

A. Heuristic Function:

The Initial state is [8,1,2],[3,6,4],[0,7,5] and goal state is [1,2,3],[8,0,4],[7,6,5]. It is represented by h(n) and it calculates the cost of an optional path between the pair of states. The value of the heuristic function is always positive. Instead of moving the tiles in the empty space, we can visualize moving the empty space in place of the tile, basically swapping the tile with the empty space. The empty space can only move in four directions.

1. Up
2.Down
3. Right
4. Left

B. Breadth First Search:

Breadth-first search (BFS) is a method for exploring a tree or graph. In a BFS, you first explore all the nodes one step away, then all the nodes two steps away, etc.

Rule 1 Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.

Rule 2 If no adjacent vertex is found, remove the first vertex from the queue.

Rule 3 Repeat Rule 1 and Rule 2 until the queue is empty

C. CODE FOR ASSIGNMENT-1: 8-PUZZLE

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

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


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_junction + dash + dash + dash + right_up_angle + Fore.RESET
+ Style.RESET_ALL
middle_line = Style.BRIGHT + Fore.CYAN + left_junction + dash + dash + dash +

middle_junction + dash + dash + dash + middle_junction + dash + dash + dash +
right_junction + Fore.RESET + Style.RESET_ALL
last_line = Style.BRIGHT + Fore.CYAN + left_down_angle + dash + dash + dash +

bottom_junction + dash + dash + dash + bottom_junction + dash + dash + dash +

right_down_angle + Fore.RESET + Style.RESET_ALL


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)
```

```
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.append(Node(newState, node.current_node, node.g + 1,

            euclidianCost(newState), dir))

    return listNode

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)

        adj_node = getAdjNode(test_node)
        for node in adj_node:
            if str(node.current_node) in closed_set.keys() or

            str(node.current_node) in open_set.keys() and open_set[
                str(node.current_node)].f() < node.f():
                continue
            open_set[str(node.current_node)] = node

        del open_set[str(test_node.current_node)]


if __name__ == '__main__':
    br = main([[8, 1, 2],
               [3, 6, 4],
               [0, 7, 5]])

    print('total steps : ', len(br) - 1)
    print()
    print( "INPUT",)
    for b in br:
        if b['dir'] != '':
            letter = ''
            if b['dir'] == 'U':
                letter = 'UP'
            elif b['dir'] == 'R':
                letter = "RIGHT"
```

```
            elif b['dir'] == 'L':
                letter = 'LEFT'
            elif b['dir'] == 'D':
                letter = 'OUTPUT'
        print_puzzle(b['node'])
        print()
```

```
total steps :  6

INPUT
```

8	1	2
3	6	4
	7	5

8	1	2
3	6	4
7		5

8	1	2
3		4
7	6	5

8	1	2
	3	4
7	6	5

Fig. 1. output of code.

	1	2
8	3	4
7	6	5

1		2
8	3	4
7	6	5

1	3	2
8		4
7	6	5

Fig. 2. output of code.

D. CODE FOR ASSIGNMENT-2: BFS

```
            #include <bits/stdc++.h>
using namespace std;
int z,l;
int parent[100];
int cost [1000][10000];
int find(int i)
{

    while (parent[i] != i)
        i = parent[i];
    return i;
}
void union1(int i, int j)
{
        int f = find(i);
        int h = find(j);
        parent[f] = h;
}
void BFsMST()
{
        int mincost = 0;
        int edge_count = 0;
        while (edge_count < z - 1)
    {
                int min = INT_MAX, f = -1, h = -1;
                for (int i = 0; i < z; i++) {
                        for (int j = 0; j < z; j++)
            {
                                if (find(i) != find(j) && cost[i][j] < min)
                                {
                                        min = cost[i][j];
                                        f = i;
                                        h = j;
                                }
                        }
                }

                union1(f,h);
        cout<<"Edge "<<edge_count++<<":("<<f<<" "<<h<<") cost:"<<min<<endl;
                mincost += min;
        }
        cout<<endl<<"Minimum cost= "<<mincost;
}
```

```
int main()
{

        z=8;
        l=9;
    for(int i=0;i<z;i++)
    {
        for(int j=0; j<z; j++)
        {
            cost[i][j]= INT_MAX;
        }
    }
cost[0][1]=1;
cost[0][2]=2;
cost[2][3]=3;
cost[3][4]=4;
cost[3][5]=5;
cost[5][4]=100;
cost[5][6]=7;
cost[6][7]=101;
cost[4][7]=8;
    for (int i = 0; i < z; i++)
        parent[i] = i;
        BFsMST();
        return 0;
}
```

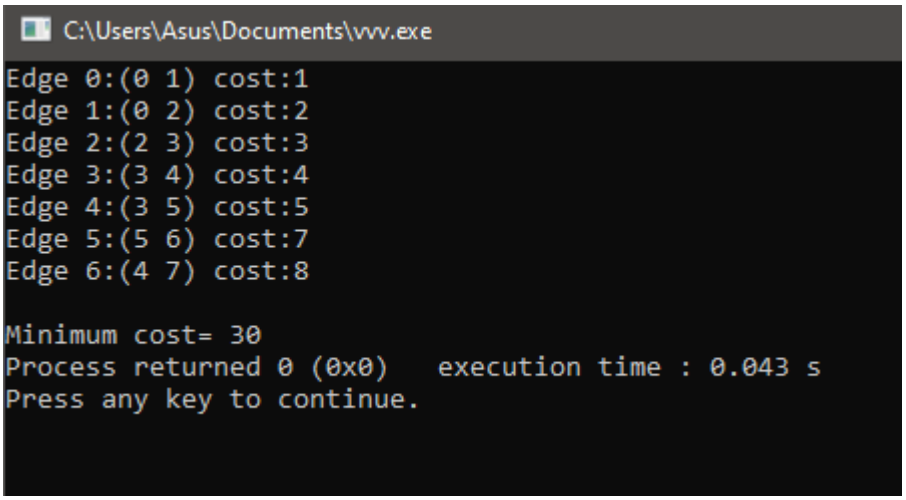

Fig. 3. output of code.

IV. Conclusion

A. Heuristic Function:

An approach for solving the 8-puzzle problem has been proposed which minimizes the memory required while achieving optimum results. All the heuristics have been applied to a* algorithm to bring uniformity and the results were shown.

B. Breadth First Search:

The Breadth First Search 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] I. S. Jacobs and C. P. Bean, "Fine particles, thin films and exchange anisotropy," in Magnetism, vol. III, G. T. Rado and H. Suhl, Eds. New York: Academic, 1963, pp. 271–350.

[2] Y. Yorozu, M. Hirano, K. Oka, and Y. Tagawa, "Electron spectroscopy studies on magneto-optical media and plastic substrate interface," IEEE Transl. J. Magn. Japan, vol. 2, pp. 740–741, August 1987 [Digests 9th Annual Conf. Magnetics Japan, p. 301, 1982].