Report Name: Puzzle problem Solving and Breadth-First search(BFS) in Graph representation problem solving.

CSE-0408 Summer 2021

Name:Kaspia Tabbassom
Id:UG02-47-18-046
Department of Computer Science and Engineering
State University of Bangladesh (SUB)
Dhaka, Bangladesh
email address:kaspiaishita@gmail.com

I. INTRODUCTION

For puzzle : We can solve Heuristic function by the eight puzzle prob-lem.It is also known as the name of N puzzle problem orsliding puzzle problem.N-puzzle that consists of N tiles (N+1 titles with an emptytile) where N can be 8, 15, 24 and so on. in these types ofproblems we have given a initial state or initial configuration(Start state) and a Goal state or Goal Configuration.It is playedon a 3-by-3 grid with 8 square blocks labeled 1 through 8 and ablank square. Your goal is to rearrange the blocks so that theyare in order. You are permitted to slide blocks horizontallyor vertically into the blank square. The following shows asequence of legal moves from an initial board position (left)to the goal position (right)

For BFS: Breadth first search is a graph traversal algorithm that starts traversing the graph from root node and explores all the neighbouring nodes. Then, it selects the nearest node and explore all the unexplored nodes. The algorithm follows the same process for each of the nearest node until it finds the goal.

II. VARIANTS OF BREADTH FIRST SEARCH

The two variants of Breadth FIRST SEARCH are Greedy Best First Search and A* Best First Search.

Greedy BFS: Algorithm selects the path which appears to be the best, it can be known as the combination of depth-first search and breadth-first search. Greedy BFS makes use of Heuristic function and search and allows us to take advantages of both algorithms.
A* BFS: Is an informed search algorithm, or a best-first search, meaning that it is formulated in terms of weighted graphs: starting from a specific starting node of a graph, it aims to find a path to the given goal node having the smallest cost (least distance travelled, shortest time, etc.)

III. RULES

Rules for 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)
1. Up
2. Down
3. Right or
4. Left
The empty space cannot move diagonally and can take only one step at a time.

Code For Puzzle Solve:

include ¡bits/stdc++.h¿

using namespace std;

define N 3

// state space tree nodes

struct Node

// stores parent node of current node
// helps in tracing path when answer is found Node* parent;

// stores matrix

int mat[N][N];

// stores blank tile cordinates
int x, y;

// stores the number of misplaced tiles int cost;

// stores the number of moves so far int level;

;

// Function to print N x N matrix

int printMatrix(int mat[N][N])

for (int i = 0; i ¡ N; i++)

for (int j = 0; j ¡ N; j++)

printf("
printf("");

// Function to allocate a new node
Node* newNode(int mat[N][N], int x, int y, int newX,
int newY, int level, Node* parent)

Node* node = new Node;

// set pointer for path to root node-¿parent = parent;

// copy data from parent node to current node
memcpy(node-¿mat, mat, sizeof node-¿mat);

// move tile by 1 postion
swap(node-¿mat[x][y], node-¿mat[newX][newY]);

// set number of misplaced tiles
node-¿cost = INT$_{M}AX$;

// set number of moves so far node-¿level = level;

// update new blank tile cordinates
node-¿x = newX;
node-¿y = newY;

return node;

// botton, left, top, right
int row[] = 1, 0, -1, 0 ;
int col[] = 0, -1, 0, 1 ;

// Function to calculate the the number of misplaced tiles
// ie. number of non-blank tiles not in their goal position
int calculateCost(int initial[N][N], int final[N][N])

int count = 0;
for (int i = 0; i ¡ N; i++)
for (int j = 0; j ¡ N; j++)
if (initial[i][j] initial[i][j] != final[i][j]) count++;
return count;

// Function to check if (x, y) is a valid matrix cordinate
int isSafe(int x, int y)

return (x ¿= 0 x ¡ N y ¿= 0 y ¡ N);

// print path from root node to destination node
void printPath(Node* root)

if (root == NULL)
return; printPath(root-¿parent);
printMatrix(root-¿mat);

printf("");

// Comparison object to be used to order the heap struct comp
bool operator()(const Node* lhs, const Node* rhs) const

return (lhs-¿cost + lhs-¿level) ¿ (rhs-¿cost + rhs- ¿level);

;

// Function to solve N*N - 1 puzzle algorithm using
// Branch and Bound. x and y are blank tile coordinates
// in initial state
void solve(int initial[N][N], int x, int y,
int final[N][N])

// Create a priority queue to store live nodes of
// search tree;
priority$_{q}ueue < Node*, std :: vector < Node* >, comp > pq$;

// create a root node and calculate its cost
Node* root = newNode(initial, x, y, x, y, 0, NULL);
root-¿cost = calculateCost(initial, final);

// Add root to list of live nodes;
pq.push(root);

// Finds a live node with least cost,
// add its childrens to list of live nodes and
// finally deletes it from the list.
while (!pq.empty())
// Find a live node with least estimated cost
Node* min = pq.top();

// The found node is deleted from the list of
// live nodes
pq.pop();

// if min is an answer node
if (min-¿cost == 0)

// print the path from root to destination;
printPath(min);
return;

// do for each child of min
// max 4 children for a node
for (int i = 0; i ¡ 4; i++)

if (isSafe(min-¿x + row[i], min-¿y + col[i]))

// create a child node and calculate
// its cost
Node* child = newNode(min-¿mat, min-¿x,
min-¿y, min-¿x + row[i],
min-¿y + col[i],
min-¿level + 1, min);
child-¿cost = calculateCost(child-¿mat, final);

// Add child to list of live nodes
pq.push(child);

// Driver code
int main()

// Initial configuration
// Value 0 is used for empty space
int initial[N][N] =

1, 2, 3,
5, 6, 0,
7, 8, 4
;

// Solvable Final configuration
// Value 0 is used for empty space
int final[N][N] =

1, 2, 3,

5, 8, 6,

0, 7, 4

;

// Blank tile coordinates in initial

// configuration

int x = 1, y = 2;

solve(initial, x, y, final);

return 0;

Algorithm for BFS:

Let S be the root/starting node of the graph.

Step 1: Start with node S and enqueue it to the queue.

Step 2: Repeat the following steps for all the nodes in the graph.

Step 3: Dequeue S and process it.

Step 4: Enqueue all the adjacent nodes of S and process them. [END OF LOOP]

Step 6: EXIT

Code For BFS:

In[ ]:

from collections import defultdict

from queue import Queue

In[ ]:

class Graph():

def$_init_(self, directed)$:

self.graph = defaultdict(list)

self.directed =directed

def add$_edge(self, u, v)$:

if self.directed:

self.graph[u].append(v)

else:

self.graph[u].aparend(v)

self.graph[v].aparend(u)

def bfs(self, vertex):

visited =[]

queue = Queue()

queue.put(vertex)

while not queue empty():
vertex=queue.get()

if vertex in visited:

continue

print(vertex,end =” ”)

visited.aparend(vertex)

for neighbour in self.graph[vertex]:

if neighbour != None:

queue.put(neighbour)

In[ ]:

g = Graph(True)

In[4]:

g.add$_edge('s','r')$

g.add$_edge('s','v')$

g.add$_edge('s','x')$

g.add$_edge('r','t')$

g.add$_edge('v','w')$

g.add$_edge('x','r')$

g.add$_edge('x','u')$

g.add$_edge('t','x')$

g.add$_edge('t','u')$

g.add$_edge('t','y')$

g.add$_edge('w','s')$

g.add$_edge('w','y')$

g.add$_edge('u',None)$

g.add$_edge('y','u')$

In[5]:
g.graph

In[1]:

g.bfs(’s’)

IV. Conclusion

For Puzzle : The heuristic function is a way to inform the search about the direction to a goal. It provides an informed way to guess which neighbor of a node will lead to a goal. There is nothing magical about a heuristic function. It must use only information that can be readily obtained about a node.

For BFS: 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.