|  |  |  |
| --- | --- | --- |
| **Algorithms analysis** | Section | 02 |
| Student number | 21600635 |
| **Homework 4 – Graph Search** | Name | Jung, BoMoon |

□ Choose the target problem you solved: *Puzzle* *Game*

□ Num. of lines in your code: 240 lines

□ Num. of functions in your code: 13 functions

□ Flowchart of your algorithm



□ Describe your algorithm

First, I implemented queues and sets. In addition, when implementing a queue for outputting the result later, the address of the parent node is also added. After that, declare the necessary functions, store the puzzles of the given three states in sequence in a string array, and save the final value ‘123804765’ in the final variable. In this case, a blank space is considered as '0'. Also, declare a ‘set’ to check whether a puzzle with a changed state is a duplicate, and declare a queue to move the puzzle one by one and implement Breadth-First Search.

First, the root puzzle is put into the set and queue, and the puzzle at the same level is compared with the final. At this time, if the comparison result is the same, the number of times the puzzle has been moved and the swapped process are printed out and the next problem is moved on. If the comparison result is not the same, search the index of '0' to move the puzzle, and find the row and column values. Also, after checking whether it can move up, down, left and right from the position of ‘0’ through the IF conditional statement, if it can be moved, change the state of the puzzle through swap\_puzzle, check the set, and if there is no identical puzzle state, put it in the queue and set.

If you change the state of a puzzle at the same level and put it all in the queue, it counts how many states have changed, and repeats the same operation for the next level of puzzles.

□ Screenshots of your program running

텍스트, 원격, 전자기기이(가) 표시된 사진

자동 생성된 설명텍스트, 원격, 전자기기이(가) 표시된 사진

자동 생성된 설명텍스트, 원격이(가) 표시된 사진

자동 생성된 설명텍스트, 원격, 전자기기, 제어이(가) 표시된 사진

자동 생성된 설명텍스트, 전자기기, 원격이(가) 표시된 사진

자동 생성된 설명텍스트, 원격이(가) 표시된 사진

자동 생성된 설명

□ Discussion about the results

BFS searches from the root node to adjacent nodes. That is, it searches broadly before searching deeply. Therefore, in this task, it is implemented to find the shortest path by applying the BFS concept as an algorithm for solving a puzzle. For efficient implementation, a linked list is used to point to the parent's node. However, when using the BFS method, there is a risk that we may fall into an infinite loop if we do not know which node we have heard from. Therefore, there is no need to worry about duplication by using a data container called set to know which node has been visited, and it is also easy to know which node has been visited, eliminating the risk of falling into an infinite loop. In addition, BFS was implemented by visiting the next level in sequence using a queue. The running time of BFS using queue is O(V) when enqueue, O(V) when dequeue, and O(E) when scan adjacency list, so it is O(V+E).

Finding a solution to avoid falling into an infinite loop was the most difficult. After much searching and studying the container, I was able to find the set and eliminate the risk of falling into an infinite loop.

The limitation of my algorithm is that the size of the puzzle is fixed in advance and implemented, so if you put a size other than 3x3 for the size of the puzzle, the value may come out differently.

Therefore, the solution to make a better algorithm is an algorithm that can solve the puzzle according to the size of the puzzle after receiving the size of the puzzle from the user.

By doing this task, I learned exactly how BFS is implemented, and by implementing the code to solve the puzzle by applying it, I also learned a good container called set. In addition, I was able to study not only BFS but also DFS, so my knowledge was wider. I also clearly realized the advantages of trees. I directly compared the case of storing the values in the order inserted into the container called set and the case of storing the values using a tree. At this time, it was confirmed that the speed of finding a specific value was much more efficient when using a tree, and the importance of complex time was also recognized.

□ Codes

#include <iostream>

#include <cstring>

using namespace std;

struct Node{

string puzzle;

struct Node \*next;

struct Node \*parent;

};

struct Queue{

Node \*front, \*rear;

int len = 0;

Queue(){

front = rear = NULL;

}

int size(){

return len;

}

bool empty(){

return len == 0;

}

void push(string puz, Node \*par){

Node \*node = new Node;

node->puzzle = puz;

node->parent = par;

node->next = NULL;

if (empty()){

front = rear = node;

}

else{

rear->next = node;

rear = rear->next;

}

len++;

}

void pop(){

Node \*delNode = front;

front = delNode->next;

len--;

}

Node\* q\_front(){

return front;

}

bool find\_puzzle(string temp){

Node \*iter = front;

for(int i = 0; i < size(); i++){

if(iter->puzzle == temp){

return false;

}

iter = iter->next;

}

return true;

}

};

struct set{

string puzzle;

set \*left;

set \*right;

};

struct Set{

set \*\_head = NULL;

Set(){

\_head = NULL;

}

void insert(string puz){

set \*temp = new set;

set \*iter = \_head;

temp->puzzle = puz;

temp->left = NULL;

temp->right = NULL;

if(\_head == NULL){

\_head = temp;

return;

}

else{

set \*parent = NULL;

while(iter != NULL){

if(puz > iter->puzzle){

parent = iter;

iter = iter->right;

}

else if(puz < iter->puzzle){

parent = iter;

iter = iter->left;

}

else

break;

}

if(puz > parent->puzzle)

parent->right = temp;

else

parent->left = temp;

}

}

bool find(string puz){

set \*iter = \_head;

while(iter != NULL){

if(puz == iter->puzzle)

return true;

if(puz > iter->puzzle)

iter = iter->right;

else if(puz < iter->puzzle)

iter = iter->left;

}

return false;

}

set \*head(){

return \_head;

}

};

string state[3] = {"132078564", "370162548", "756342108"};

string final = "123804765";

int find\_position(string state)

{

for(int i = 0; i < state.size(); i++){

if(state[i] == '0')

return i;

}

return -1;

}

void print(string state){

for(int i = 0; i < state.size(); i++){

cout << state[i] << " ";

if(i % 3 == 2)

cout << "\n";

}

return;

}

string swap\_puzzle(string check, int ind, int change\_ind){

char temp = check[ind];

check[ind] = check[change\_ind];

check[change\_ind] = temp;

return check;

}

int main(){

int result = 0;

int isfind = 0;

for(int i = 0; i < 3; i++){

string puzzle = state[i];

cout << "-------------\n";

cout << i + 1 << " puzzle\n";

Queue q;

Set done;

result = 0;

isfind = 0;

q.push(puzzle, NULL);

done.insert(puzzle);

// 같은 level node 검사하기

while(!q.empty()){

int size = q.size();

for(int j = 0; j < size; j++){

Node \*check = q.q\_front();

q.pop();

// 만약 완료 됐으면 끝

if(check->puzzle == final){

cout << "how many move the puzzle? " << result << "\n";

for(int level = result + 1; level > 0; level--){

Node \*temp\_Node = check;

for(int temp\_level = 0; temp\_level < level - 1; temp\_level++)

temp\_Node = temp\_Node->parent;

print(temp\_Node->puzzle);

cout << "\n";

}

isfind = 1;

break;

}

// 0 찾기

int ind = find\_position(check->puzzle);

int row, col;

// row값 찾기

if(ind == 0)

row = 0;

else

row = ind / 3;

// col값 찾기

if(ind == 0)

col = 0;

else

col = ind % 3;

// 위치 바꾸고 큐에 넣기

string temp;

string temp1 = check->puzzle;

Node \*parent = check;

if(ind - 3 >= 0){

temp = swap\_puzzle(temp1, ind, ind - 3);

// 없었던 값이면 set에 넣기

if(!done.find(temp)){

q.push(temp, parent);

done.insert(temp);

}

}

if(ind - 1 >= 0 && (ind - 1) / 3 == row){

temp = swap\_puzzle(temp1, ind, ind - 1);

// 없었던 값이면 set에 넣기

if(!done.find(temp)){

q.push(temp, parent);

done.insert(temp);

}

}

if(ind + 1 <= 8 && (ind + 1) / 3 == row){

temp = swap\_puzzle(temp1, ind, ind + 1);

// 없었던 값이면 set에 넣기

if(!done.find(temp)){

q.push(temp, parent);

done.insert(temp);//여기서 문제

}

}

if(ind + 3 <= 8){

temp = swap\_puzzle(temp1, ind, ind + 3);

// 없었던 값이면 set에 넣기

if(!done.find(temp)){

q.push(temp, parent);

done.insert(temp);

}

}

}

//찾은게 있으면 다음 puzzle풀기

if(isfind == 1)

break;

//깊이

result++;

}

if(isfind == 0)

cout << "there is no path back to the correct state\n";

}

cout << "finished\n";

return 0;

}