|  |  |  |
| --- | --- | --- |
| **Algorithms analysis** | Section | 02 |
| Student number | 21900371 |
| **Homework 4 – Graph Search** | Name | Seo, Hyo Gyeong |

*If your explanation is less informative and insufficient, then you may not get any points.*

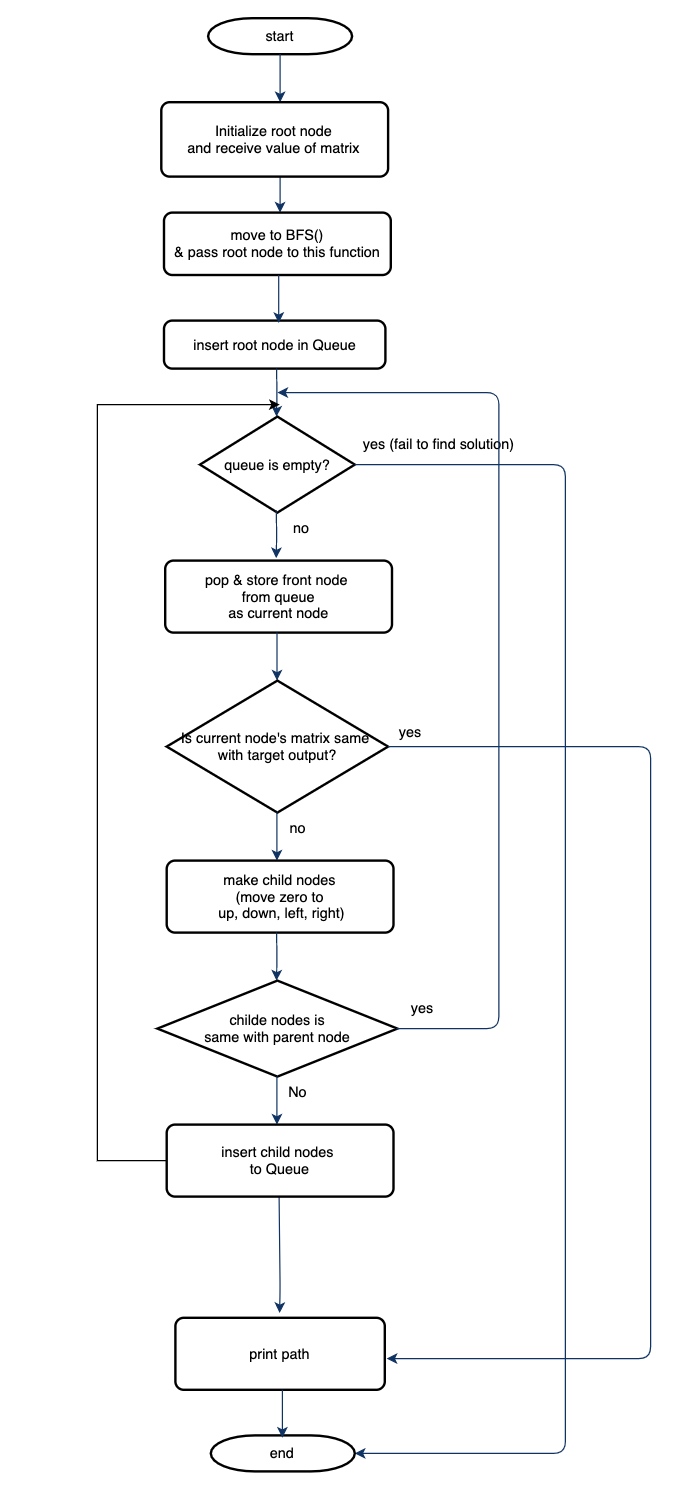
*Also, you should provide discussion, otherwise you will get penalty.*

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

□ Num. of lines in your code: 326 lines

□ Num. of functions in your code: 13 functions (except functions in class)

□ Flowchart of your algorithm



□ Describe your algorithm

I used BFS to solve this puzzle. And I also created nodes as structures for BFS. A node consists of a two-dimensional array that stores matrix, variables that store zero positions(row, col), and a pointer that connects parent nodes.  
  
First, initialized the root node and then get input puzzle matrix. The root node is then passed to the BFS() to solve the puzzle. In BFS(), insert the passed root node in queue. And while the queue is not empty, repeat the following in the while statement.

.

1. Save the front node of Queue as the current\_node variable and pop the front node.
2. compare the matrix of Current\_node with the target matrix.

* If the matrix is the same (solving the problem), print the current matrix and the matrix of intermediate processes connected to parent via the procedure(). And then exit the loop
* If it's different, move on to 3.

1. For BFS, create a child node that moves up, down, left, and right positions of zero based on current\_node.

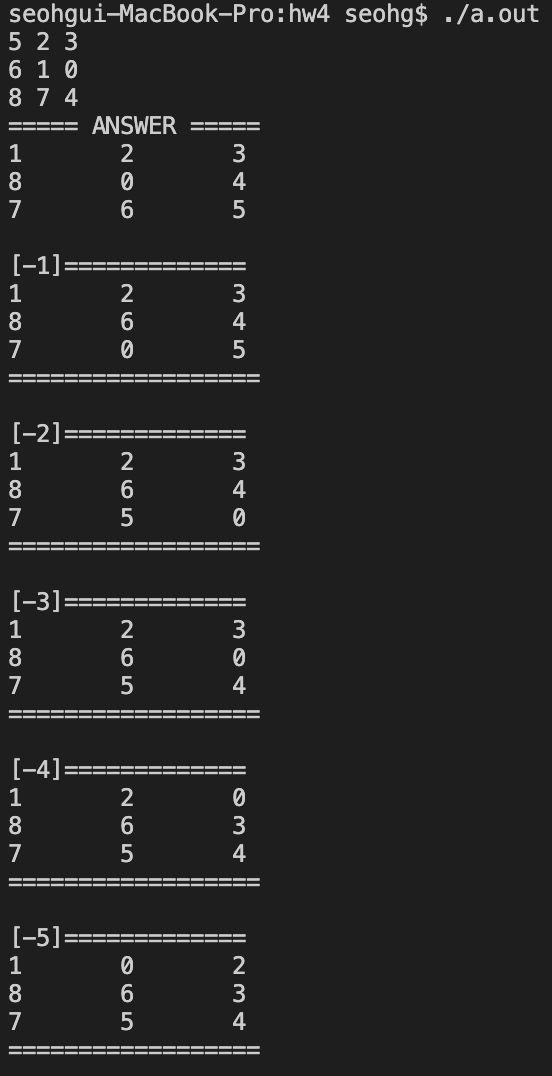
3-1 : If the current\_node's row is not 0, create a child node that has moved upward, and insert it in queue if the path connected by the parent link does not have the same matrix.  
  
3-2: If the current\_node's row is not 2, create a children node that has moved downward, and insert it in queue if the path connected by the parent link does not have the same matrix.

3-3: If the current\_node's col is not 0, create a child node that has shifted to the left, and insert it in queue if the path connected by the parent link does not have the same matrix.

3-4: If the current\_node's col is not 2, create a children node that has shifted to the right, and insert it in queue if the path connected by the parent link does not have the same matrix.

If q is not empty after moving, go back to 1. again.

□ Screenshots of your program running

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

□ Discussion about the results

// Please discuss about your algorithm, any trouble you encountered, limitation of your algorithm,

// any possible better solution (algorithm). Please share what you feel, what you learn from this assignment.

1. About BFS

**- algorithm**

I used the BFS algorithm in my code. A BFS algorithm is an algorithm that explores nodes close to the parent node. In other words, it is not a deep search, it is a broad search. The BFS has a time complexity of O(V+E) with a comparison of E times and a visit time of V. BFS uses queue, a data structure that can be taken out and used after saving visited nodes in turn, and is suitable for finding the shortest path.

**- pros**

It is fast when the number of nodes is small and shallow, and it preferentially explores the width, ensuring the shortest path even if there are multiple paths that can be the answer.

**- cons**

Because nodes that need to be explored are stored in queues and taken out, It require a lot of storage space. Moreover, when the number of nodes is large, it takes a very long time if the target node is in the deep stage.

2. compare with other algorithms

**- compare with DFS**

DFS is an algorithm that explores depth first. Therefore, DFS is efficient because it requires less storage than BFS. However, the BFS guarantees the shortest path, while the DFS does not. Therefore, it is inappropriate to use DFS in situations where the puzzle needs to be solved with minimal movement. Also, if depth is not restricted, it can fall too deep into one branch. However, if the node we are looking for is in the deep stage, using DFS can achieve results faster than using BFS.

**- compare with A\***

The A\* algorithm is an algorithm that uses priority queues to explore the nodes with the lowest cost, ensuring that the shortest path is found, as is the case with BFS. It is important to determine the appropriate heuristic function because its performance depends on the heuristic function, which is the method of calculating the cost. Therefore, there is a disadvantage that the performance varies even if the same function is used for different purposes.

3. Others

- Although I did not put the implemented code into the homework code (because I experimented with STL), I saw that when I tried to solve some problem with DFS, It show the result with about 10000 depth or takes too long time. Therefore, I thought that when we use DFS, we need code in case of excessive depth.

- The order of moving zero values by creating child nodes is up, down, left, and right.  
This sequence would change the order in which the queue was inserted, and I thought this change would make a big difference in obtaining results. In the case of DFS, I believe that this sequence can either make depth very long or very short.

- When running the code, it took about 3.8 seconds to print the result with a depth of 25, and 0.03 seconds to print the result with a depth of 17. This shows that the order of exploration at the same level is also important, but the change in execution time is large with changes in depth.

4. What I feel & learn

- I had some difficulty implementing the queue. There was a problem with implementing the use of nodes for BFS as variable types in structure for Queue. Through these difficulties, I could practice implementing Queue and Tree a lot in solving this problem. Through this process, I felt that implementing algorithms is important, but it is also necessary to practice implementing basic data structures skillfully.

-  A\* algorithm may be a little more difficult to implement because there is more things to consider than DFS and BFS and requires appropriate functions that calculating cost. But I think it would be very efficiently by using the appropriate cost calculation function, so I would like to implement code that solves the problem using this algorithm later.

- In situations where there are many navigating nodes, such as increasing the size of the puzzle (row, column increasing), the difference in computational speed according to the algorithm will be very large. Therefore, I thought that it is important to select and apply algorithms by considering the conditions to solve the problem.

□ Codes

// Please copy & paste you code here.

// You should also submit the separate executable C or C++ files, TA will try run your code.

#include <iostream>

#include <string.h>

#include <stdlib.h>

#include <cstdio>

#include <ctime>

using namespace std;

/\* sturctures for BFS \*/

typedef struct Node{ //Node for BFS

int mat[3][3]; // puzzle board

int row, col; //position of zero

Node \*parent; // link parent node

}Node;

typedef struct qNode{ //Node for BFS queue

struct Node\* qnode; //node structure

struct qNode\* next; //for link nodes

}qNode;

typedef struct Vector{ //this is simple vector structure for storing path

Node\* vnode; //node

Vector\* next; //link

}Vector;

/\* queue (data structure) using class\*/

class queue{

qNode \*front, \*rear; //link head & tail

public:

queue(); //constructure

void insert\_q(Node\*t); //insert node

void delete\_q(); //delete node

Node\* queue\_front(); //show front node

bool queue\_empty(); //check queue is empty or not

};

queue::queue(){

front=NULL; //init front

rear=NULL; //init tail

}

void queue::insert\_q(Node\* t){

qNode\* temp = new qNode; //make new qNode

temp->qnode = t; //store node

temp->next = NULL; //init link ptr as null

if(rear==NULL){ //if node is frist node

rear = temp; // tail ptr -> new qNode

front = temp; // head ptr -> new qNode

return;

}

rear->next = temp; //if node is not first node

rear=temp; //push new qNode in back

}

void queue::delete\_q(){

if(!queue\_empty()){ // queue is not empty

qNode\* tmp = front; //store the front node temporary

front=front->next; //move front ptr to next node

if(front==NULL){ //if there is no next node

rear=NULL; //make tail ptr to null

}

delete tmp; //delete node

}

}

Node\* queue::queue\_front(){

if(!queue\_empty()){ //if queue is not empty

Node\* temp = new Node; //make new Node

temp = front->qnode; //store front node to temp temporary

return temp; //return temp node

}

return NULL; //if queue is empty, return null

}

bool queue::queue\_empty(){

if(front==NULL&&rear==NULL){ //if head, tail ptr is null

return true; //return true

}else{

return false; // if not, return false

}

}

/\* target matrix \*/

int output[3][3] = {{1,2,3},{8,0,4},{7,6,5}};

/\* functions \*/

void init\_node(Node\* node); //initiallize node

void get\_matrix(Node\* node); //get input matrix

void print\_node(Node\* node); //print node's matrix

bool compare(int cur[][3], int out[][3]); //compare the matrix

void b\_move(Node\* node); //make child node (moving zero)

void move\_up(Node\* node);//move zero to upside

void move\_down(Node\* node);//move zero to downside

void move\_left(Node\* node);//move zero to left

void move\_right(Node\* node); //move zero to right

void copy\_node(Node\* newNode, Node\* orgNode); //copy nodes (deepcopy)

bool is\_exist(Node\* node); //check if there is same node in parent(path)

void procedure(Node\* node); //print path

int BFS(Node\* node); //for BFS

/\* global \*/

queue q; //for queue

Vector\* path\_head = new Vector; //for saving path(head)

Vector\* path = new Vector; //for saving path (other nodes)

int main(){

struct Node\* node = new Node; //make root node for DFS

clock\_t startTime, endTime;

init\_node(node); //init root node

get\_matrix(node); //recieved value of matrix (save position of zero)

startTime = clock();

BFS(node); //solve puzzle using bfs

endTime = clock();

cout<<"running time : "<<double(endTime - startTime)/CLOCKS\_PER\_SEC<<"s"<<endl;

return 0;

}

/\* compare current node matrix to target matrix \*/

bool compare(int cur[][3], int out[][3]){

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

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

if(cur[i][j]!=out[i][j]){

return false;

}

}

}

return true;

}

/\* init node \*/

void init\_node(Node\* node){

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

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

node->mat[i][j] = 0;

}

}

node->row=0;

node->col=0;

node->parent = NULL;

}

/\* get matrix \*/

void get\_matrix(Node\* node){

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

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

int temp;

scanf("%d",&temp); //get matrix element

node->mat[i][j] = temp;

if(temp == 0){

node->row = i;

node->col = j;

}

}

}

}

/\* print node \*/

void print\_node(Node\* node){

if(node!=NULL){

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

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

printf("%d\t",node->mat[i][j]);

}

printf("\n");

}

}

}

/\* print procecure (path) \*/

void procedure(Node\* node){

int a = 0; // for printing step number

struct Node\* parent = node;

path\_head->next = path; //link path\_head and path

path->vnode=parent; //save path's node

path->next = NULL; //init link ptr as null

while(parent!=NULL){ //exit if node is root node

Vector\* temp = new Vector;

a--;

parent = parent->parent; //load parent node

path->next = temp; //link new node to vector

temp->vnode = parent; //save parent to node

temp->next = NULL; //init ptr as null

if(parent==NULL){

break;

}

printf("\n[%d]=============\n",a);

print\_node(temp->vnode);

printf("==================\n");

}

}

/\* move zero \*/

void move\_up(Node\* node){

node->mat[node->row][node->col] = node->mat[node->row-1][node->col];

node->mat[node->row-1][node->col]=0;

node->row = node->row-1;

}

void move\_down(Node\* node){

node->mat[node->row][node->col] = node->mat[node->row+1][node->col];

node->mat[node->row+1][node->col]=0;

node->row = node->row+1;

}

void move\_left(Node\* node){

node->mat[node->row][node->col] = node->mat[node->row][node->col-1];

node->mat[node->row][node->col-1]=0;

node->col = node->col-1;

}

void move\_right(Node\* node){

node->mat[node->row][node->col] = node->mat[node->row][node->col+1];

node->mat[node->row][node->col+1]=0;

node->col = node->col+1;

}

void b\_move(Node\* node){

for(int i=0;i<4;i++){ //up, down, right, left

struct Node \*newNode = new Node; //for child node

copy\_node(newNode, node); //copy the origin node

newNode->parent = node; //link the origin node as parent

switch(i){

case 0:

if(newNode->row!=0){

move\_up(newNode); // in child node modify the position of zero to upside

if(!is\_exist(newNode)){ //if modified node is not same with any parent(in path) node

q.insert\_q(newNode); //insert this node to queue

}

}

break;

case 1:

if(newNode->row!=2){

move\_down(newNode);

if(!is\_exist(newNode)){

q.insert\_q(newNode);

}

}

break;

case 2:

if(newNode->col!=0){

move\_left(newNode);

if(!is\_exist(newNode)){

q.insert\_q(newNode);

}

}

break;

case 3:

if(newNode->col!=2){

move\_right(newNode);

if(!is\_exist(newNode)){

q.insert\_q(newNode);

}

}

break;

default:;

}

}

}

/\* check whether the node is same with parent node \*/

bool is\_exist(Node\* node) {

struct Node\* temp = node->parent;

while(temp!= NULL) {

if(compare(node->mat, temp->mat)) { //if there is same matrix

return true; //return true

}

temp = temp->parent;

}

return false;

}

/\* copy node\*/

void copy\_node(Node\* newNode, Node\* orgNode){ //copy node's element

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

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

newNode->mat[i][j] = orgNode->mat[i][j];

}

}

newNode->row=orgNode->row;

newNode->col=orgNode->col;

newNode->parent =NULL;

}

/\*\*\*\*\* BFS algorithm \*\*\*\*\*/

int BFS(Node\* node){

struct Node\* current\_node = new Node;

q.insert\_q(node); //push root node

while(!q.queue\_empty()){ //iterate until queue is empty or find solution

current\_node = q.queue\_front(); //load front node

q.delete\_q(); //delete loaded node

if(compare(current\_node->mat, output)){ //if it find solution

printf("===== ANSWER =====\n");

print\_node(current\_node);

procedure(current\_node); //save path

return 1;

}

b\_move(current\_node); //make new node and move zero (child node)

}

return 0;

}