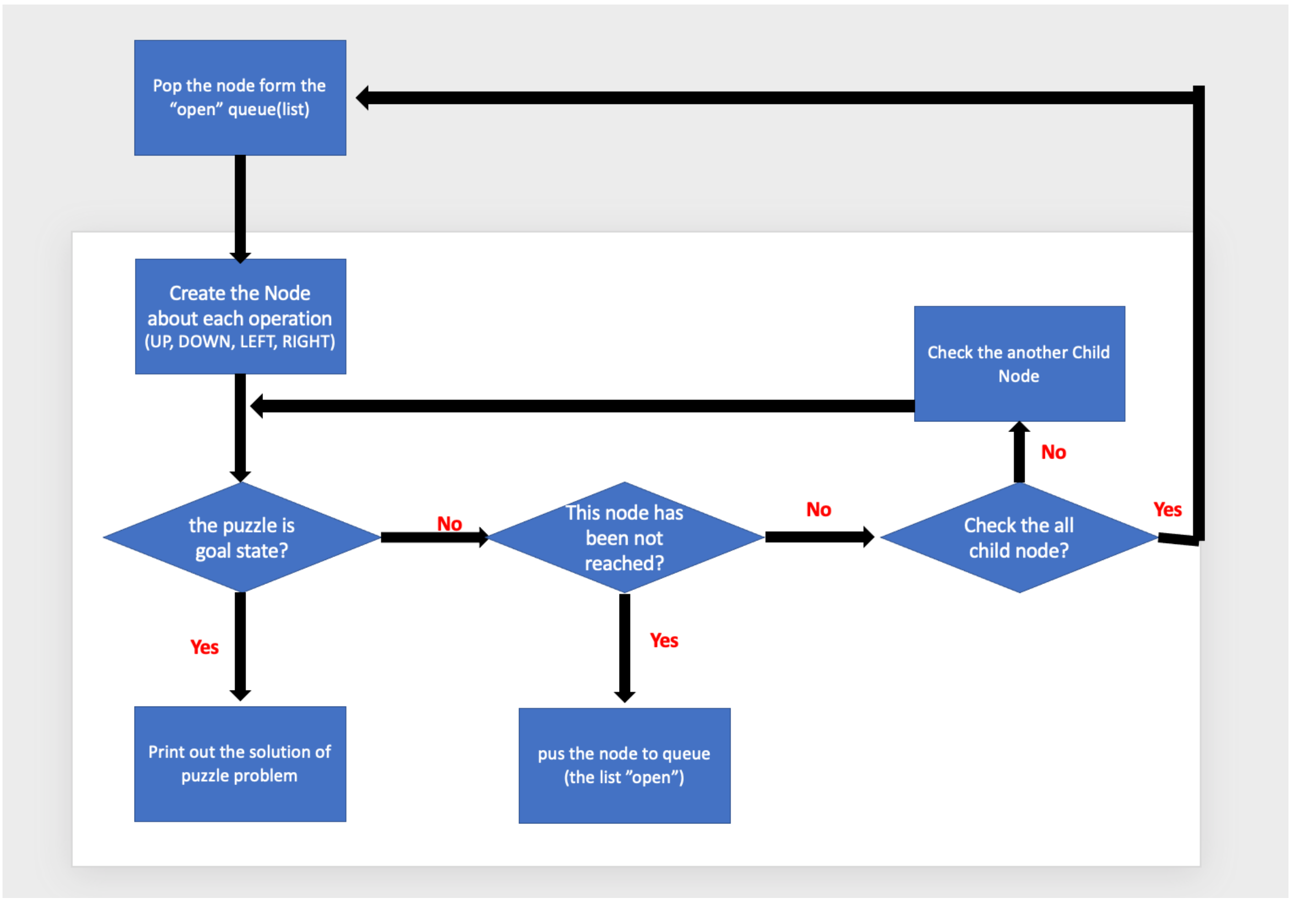
Algorithms analysis	Section	02
	Student number	21700760
Homework 4 – Graph Search	Name	Juyoung Ha

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

□ Num. of lines in your code: 220 lines

□ Num. of functions in your code: 7+3 functions (7 : class method, 3 : function)

□ Flowchart of your algorithm


Pop the node form the "open" queue(list)
Create the Node about each operation (UP, DOWN, LEFT, RIGHT)
Check the another Child Node
No
the puzzle is goal state?
No
This node has been not reached?
No
Check the all child node?
Yes
Yes
Yes
Print out the solution of puzzle problem
pus the node to queue (the list "open")

□ Describe your algorithm

In my solution program, I defined the 'Node' class that means vertexs in the graph like below.

```
class Node
{
public:
    int puzzle[9];           // puzzle data array
    list<Node*> child_node; // child node of node
    Node* parent;           // parent node

    void set_puzzle(const int arr[]);
    int empty_index();
    bool isGoal(const int arr[]);
    void block_move(int op);
    void clone_puzzle(int* new_puzzle, const int* origin_puzzle);
    void print_puzzle();
    bool equal_check(int arr[]);
};
```

First, each node has the array "puzzle". In this program, I represent the puzzle by using the one-dimensional array. For example, if I want to represent the puzzle "123/4 5/678", I use an array like this "123405678". As you know, zero means empty space of puzzles. Second, each node has the "list" that saves the pointer of the "Node" object. By using this list, each node can represent the child node of the node easily. Third, each node has the Node pointer "parent" that represents the parent node. By using this pointer, each node can maintain its parent node easily. Additionally, the class "Node" has the method to manipulate nodes and puzzles. Each method's functions are as below.

Method Name	Functions
set_puzzle	Set the puzzle of Node by taking a one-dimensional array through the parameter
empty_index	Find and return the index of empty space of puzzles
isGoal	Check whether the puzzles reached to goal states
block_move	Generate new Node by moving puzzle blocks. The parameter "op" means operator that puzzles can execute (UP, DOWN, LEFT, RIGHT). All of these constants are defined as preprocessors.
clone_puzzle	Clone the "origin_puzzle" parameter puzzle states to the "new_puzzle" parameter.
print_puzzle	Print out the puzzle states visually (3x3 shape).
equal_check	Check the two puzzles have the same state by comparing each one-dimensional array.

Next, let's consider about overall my program logic through the main function.

```
int main()
{
    int init1[9] = {1, 3, 2, 0, 7, 8, 5, 6, 4}; // initial case 1
    int init2[9] = {3, 7, 0, 1, 6, 2, 5, 4, 8}; // initial case 2
    int init3[9] = {7, 5, 6, 3, 4, 2, 1, 0, 8}; // initial case 3
    int target[9] = {1, 2, 3, 8, 0, 4, 7, 6, 5}; // target(final) case

    Node* root = new Node;
    root->set_puzzle(init1);

    list<Node*>* solution = breadth_first_search(root, target);
    cout << "step = " << solution->size() << endl;
    solution->reverse();

    if(!solution->empty()) {
        for(Node* node : *solution) {
            node->print_puzzle();
            cout << endl;
        }
    }
    else
        cout << "Failed to find solutions..." << endl;
    return 0;
}
```

This is my main funtion code.

As you can see, the array "init1", "init2", "init3" is the initial state of the puzzles and the array "target" is the goal states of the puzzles. In order to solve the puzzle problem, I use the "BFS" algorithm. So, I make the object of the Node class which named root and set the puzzles by using the "set_puzzles" method. If you want to use another initial state, you just make the one-dimensional array and give them to the "set_puzzles" method. Because the above source code offers the "init1" array to the "set_puzzle" method as the parameter, this program sets the initial states of the puzzle to init1.

Next, in order to use the "BFS" algorithm, I use the "breadth_first_search" function. The specific logic of this method will be introduced later. The function "breadth_first_search" has two parameters. In the above code, "root" means the initial states, "target" means the goal states. If this function walks successfully, these functions return the pointer of the list that stores the "Node" object pointer. This list means the path can lead to a solution. By using this list, we can print out the all steps of the solution to the puzzle problem. Because this list makes the reverse sequence of the solution, by using the reverse() method of the list, we can arrange the steps of the solution can properly. Lastly, by accessing the list "solution" sequentially and print, we can see the all steps of the solutions. So, let's look at the specific logic of the function "breadth_first_search".

```
list<Node*>* breadth_first_search(Node* root, int* target)
{
    list<Node*>* solution = new list<Node*>;
    list<Node*>* open = new list<Node*>;
    list<Node*>* close = new list<Node*>;

    open->push_back(root);
    bool goalCheck = false;

    while(!open->empty() && !goalCheck) {
        Node* curNode = open->front();
        close->push_back(curNode);
        open->pop_front();

        curNode->block_move(UP);
        curNode->block_move(DOWN);
        curNode->block_move(LEFT);
        curNode->block_move(RIGHT);

        for(Node* node : curNode->child_node)
        {
            if(node->isGoal(target)) {
                cout << "solution was founded..." << endl;
                goalCheck = true;
                solution_trace(solution, node);
            }

            if(!contains(open, node) && !contains(close, node))
                open->push_back(node);
        }
    }
    return solution;
}
```

The "breadth_first_search" function, maintain three lists (solution, open, close). The "solution" list store the steps of the solution and the "open" list store the pointer of the nodes that reached but not finished (represented "Gray" color in the textbook). So, this list takes the role of the queue in the BFS algorithm. And the "close" list store the pointer of the nodes that finished (represented "Black" color in the textbook). And the parameter "root" means the root node of the graph that includes the initial state of the puzzles, and "target" means the goal state of the puzzles by representing a one-dimensional array. Next, push the root node to the list "open" and initialize boolean value "goalCheck" to false. If "goalCheck" is ture, that means the program reach the goal puzzle state.

Until the "open" list is empty and the "goalCheck" is true, the program repeat this function.

1. pop the node from the queue(the list "open") and push them to the another queue(the list "close").

2. About the node that poped step1 excute the block_move function (UP, DOWN, LEFT, RIGHT direction).

As you can see, the below source code is the method "block_move". Through the "if" statement, this logic checks the operation is available(up, down, left, right) and if available, creates the new node and assigns the new puzzle state, push to the list "child_node", and assign the parent node.

```
void Node:: block_move(int op)
{
    int zero_index = empty_index();
    int tmp_arr[9];
    int tmp;

    if(op == UP && zero_index / 3 != 0) {
        clone_puzzle(tmp_arr, puzzle);
        tmp = tmp_arr[zero_index-3];
        tmp_arr[zero_index-3] = 0;
        tmp_arr[zero_index] = tmp;

        Node* child = new Node;
        child->set_puzzle(tmp_arr);
        child_node.push_back(child);
        child->parent = this;
    }
    else if(op == DOWN && zero_index / 3 != 2) {
        clone_puzzle(tmp_arr, puzzle);

        tmp = tmp_arr[zero_index+3];
        tmp_arr[zero_index+3] = 0;
        tmp_arr[zero_index] = tmp;

        Node* child = new Node;
        child->set_puzzle(tmp_arr);
        child_node.push_back(child);
        child->parent = this;
    }
    else if(op == LEFT && zero_index % 3 != 0) {
        clone_puzzle(tmp_arr, puzzle);

        tmp = tmp_arr[zero_index-1];
        tmp_arr[zero_index-1] = 0;
        tmp_arr[zero_index] = tmp;

        Node* child = new Node;
        child->set_puzzle(tmp_arr);
        child_node.push_back(child);
        child->parent = this;
    }
    else if(op == RIGHT && zero_index % 3 != 2) {
        clone_puzzle(tmp_arr, puzzle);

        tmp = tmp_arr[zero_index+1];
        tmp_arr[zero_index+1] = 0;
        tmp_arr[zero_index] = tmp;

        Node* child = new Node;
        child->set_puzzle(tmp_arr);
        child_node.push_back(child);
        child->parent = this;
    }
}
```

3. About the node that popped from the queue, check each child node whether reach the goal state by using "isGoal" function.

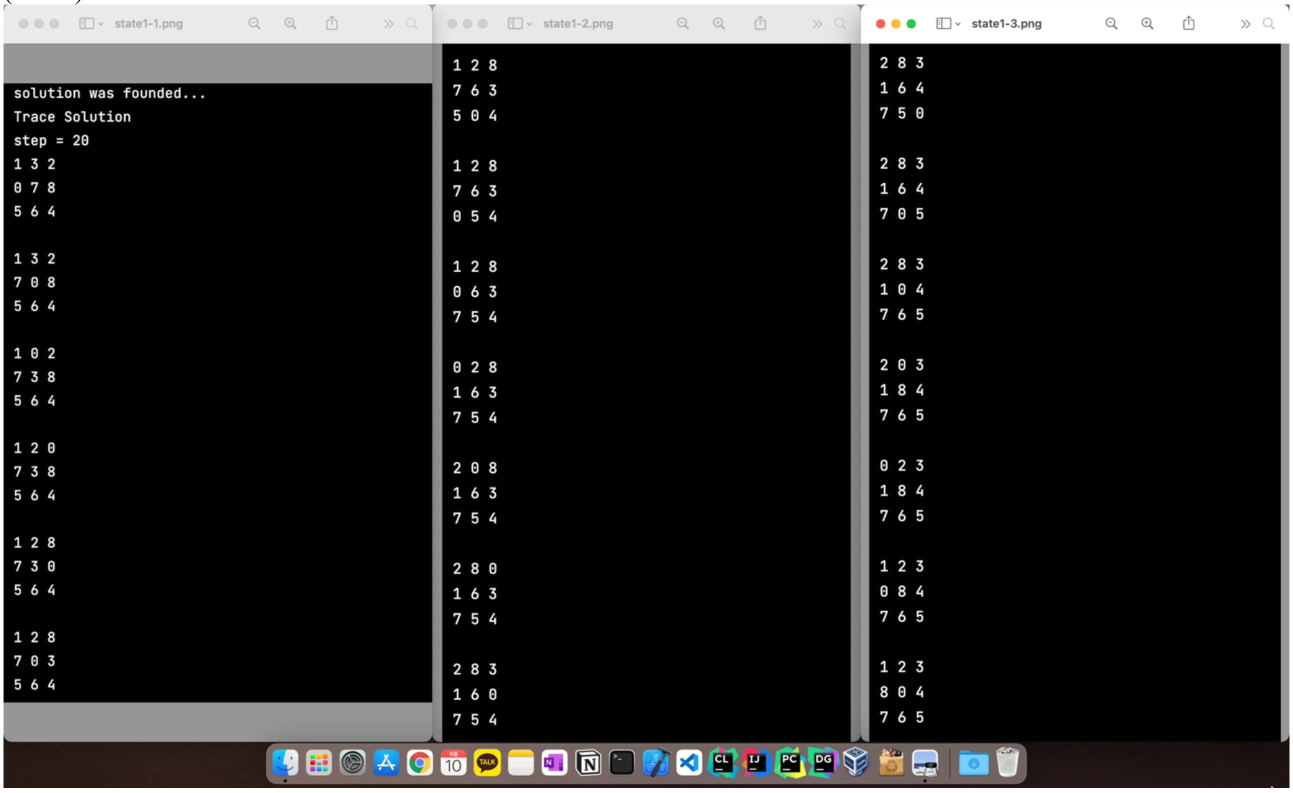
4. If the node was founded that reaches the goal state, print out the "solution was founded..." message and execute the function "solution_trace" that organizes the solution list. If the node was not founded that reach to the goal state, check the duplication of the node from the "open" and "close" list. If these two list does not include this node, push this node to the list "open".

And all of these sequence are terminated, return the list "solution".

□ Screenshots of your program running

If my program found the solution, the program print out the message "solution was founded...". Next, the program print out the total number of steps that lead to solutions. Lastly, the program print out each state of puzzles like below. As you can see, all initial states offered in the assignment can lead to goal states.

(state1)

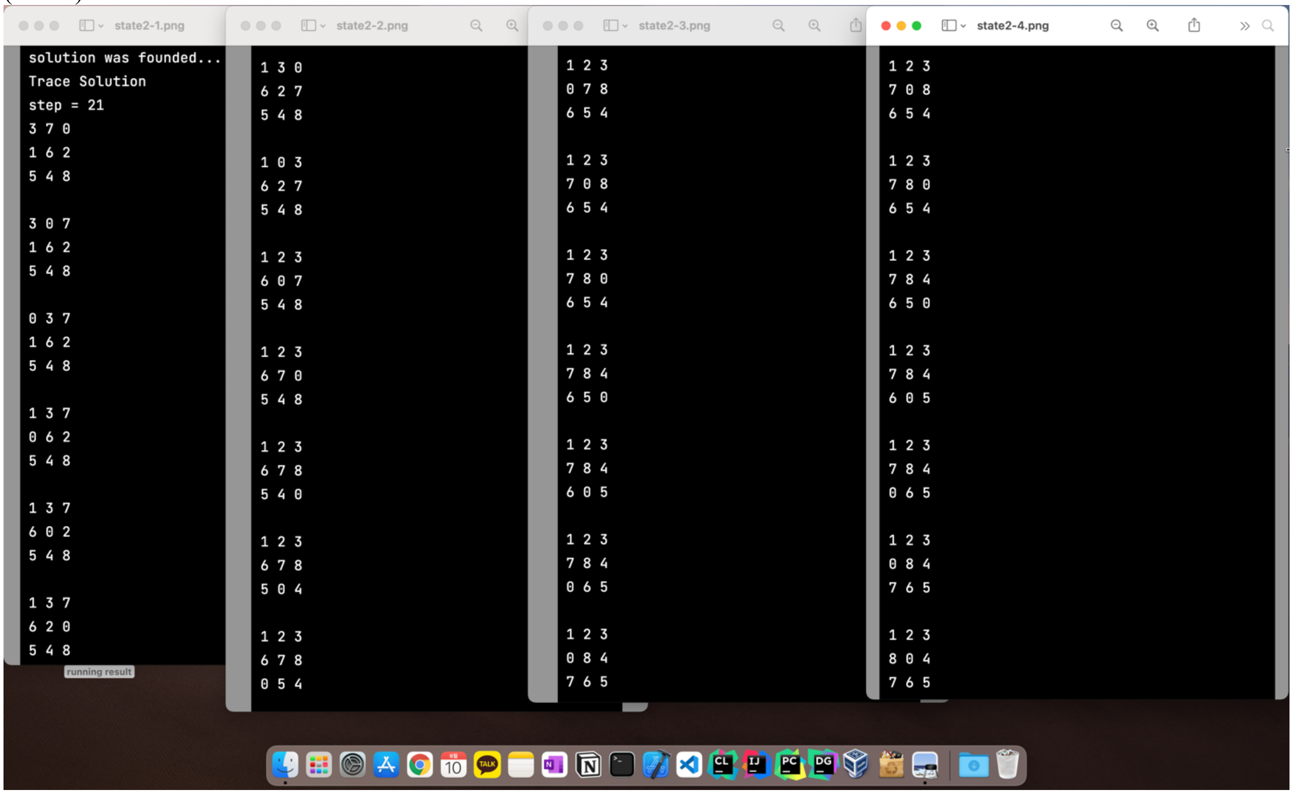


```
solution was founded...
Trace Solution
step = 20
1 3 2
0 7 8
5 6 4

1 3 2
7 0 8
5 6 4

1 0 2
7 3 8
5 6 4

1 2 0
7 3 8
5 6 4

1 2 8
7 3 0
5 6 4

1 2 8
7 0 3
5 6 4
```

```
1 2 8
7 6 3
5 0 4

1 2 8
7 6 3
0 5 4

1 2 8
0 6 3
7 5 4

0 2 8
1 6 3
7 5 4

2 0 8
1 6 3
7 5 4

2 8 0
1 6 3
7 5 4

2 8 3
1 6 0
7 5 4
```

```
2 8 3
1 6 4
7 5 0

2 8 3
1 6 4
7 0 5

2 8 3
1 0 4
7 6 5

2 0 3
1 8 4
7 6 5

0 2 3
1 8 4
7 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5
```

(state2)



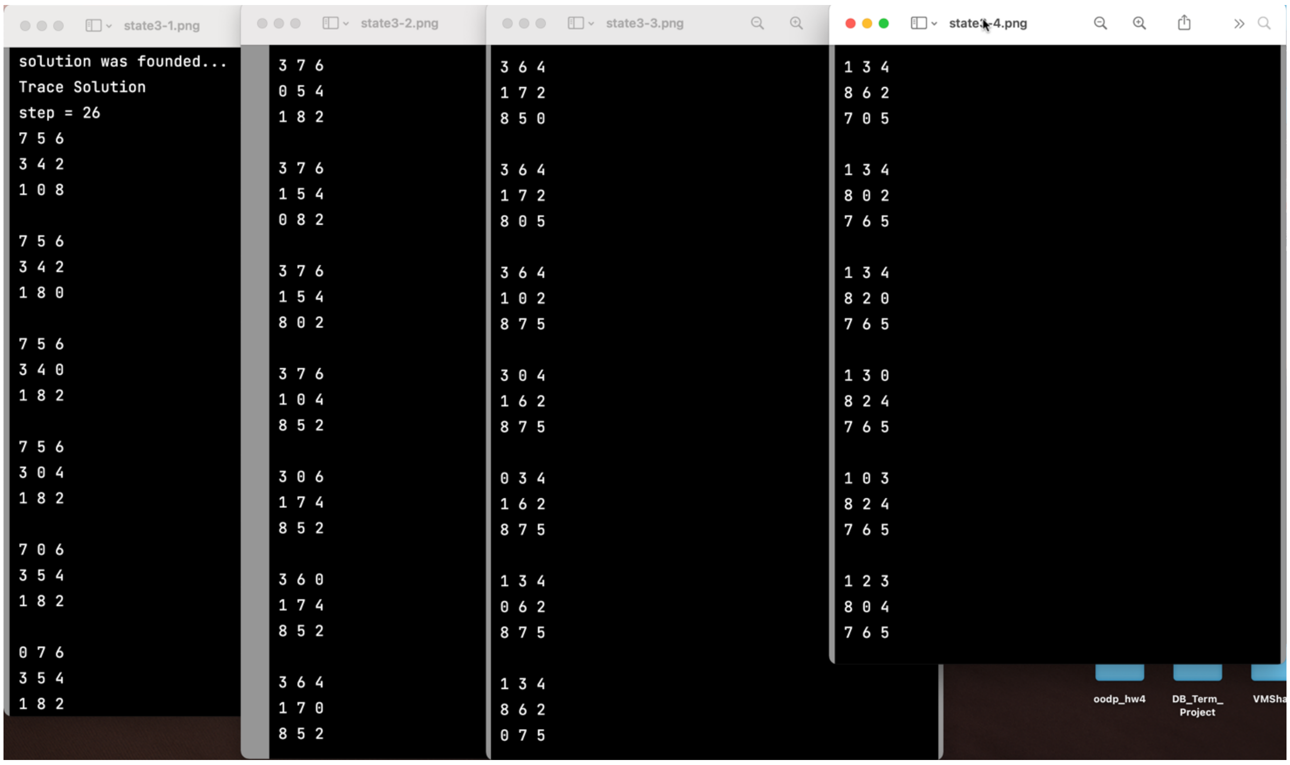
```
solution was founded...
Trace Solution
step = 21
3 7 0
1 6 2
5 4 8

3 0 7
1 6 2
5 4 8

0 3 7
1 6 2
5 4 8

1 3 7
0 6 2
5 4 8

1 3 7
6 0 2
5 4 8

1 3 7
6 2 0
5 4 8
```

```
1 3 0
6 2 7
5 4 8

1 0 3
6 2 7
5 4 8

1 2 3
6 0 7
5 4 8

1 2 3
6 7 0
5 4 8

1 2 3
6 7 8
5 4 0

1 2 3
6 7 8
5 0 4

1 2 3
6 7 8
0 5 4
```

```
1 2 3
0 7 8
6 5 4

1 2 3
7 0 8
6 5 4

1 2 3
7 8 0
6 5 4

1 2 3
7 8 4
6 5 0

1 2 3
7 8 4
6 0 5

1 2 3
7 8 4
0 6 5

1 2 3
0 8 4
7 6 5
```

```
1 2 3
7 0 8
6 5 4

1 2 3
7 8 0
6 5 4

1 2 3
7 8 4
6 5 0

1 2 3
7 8 4
6 0 5

1 2 3
7 8 4
0 6 5

1 2 3
0 8 4
7 6 5

1 2 3
8 0 4
7 6 5
```

(state3)


state3-1.png

```
solution was founded...
Trace Solution
step = 26
7 5 6
3 4 2
1 0 8

7 5 6
3 4 2
1 8 0

7 5 6
3 4 0
1 8 2

7 5 6
3 0 4
1 8 2

7 0 6
3 5 4
1 8 2

0 7 6
3 5 4
1 8 2
```

state3-2.png

```
3 7 6
0 5 4
1 8 2

3 7 6
1 5 4
0 8 2

3 7 6
1 5 4
8 0 2

3 7 6
1 0 4
8 5 2

3 0 6
1 7 4
8 5 2

3 6 0
1 7 4
8 5 2

3 6 4
1 7 0
8 5 2
```

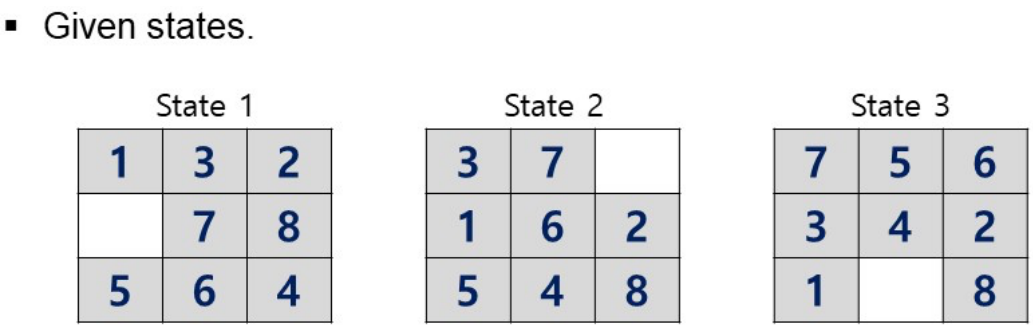
state3-3.png

```
3 6 4
1 7 2
8 5 0

3 6 4
1 7 2
8 0 5

3 6 4
1 0 2
8 7 5

3 0 4
1 6 2
8 7 5

0 3 4
1 6 2
8 7 5

1 3 4
0 6 2
8 7 5

1 3 4
8 6 2
0 7 5
```

state3-4.png

```
1 3 4
8 6 2
7 0 5

1 3 4
8 0 2
7 6 5

1 3 4
8 2 0
7 6 5

1 3 0
8 2 4
7 6 5

1 0 3
8 2 4
7 6 5

1 2 3
8 0 4
7 6 5
```

In this example, I use the initial states of the puzzle offered in the assignment description like below.


- Given states.

State 1

1	3	2
	7	8
5	6	4

State 2

3	7	
1	6	2
5	4	8

State 3

7	5	6
3	4	2
1		8

□ Discussion about the results

During the implementation of my solution code, many objects were created and used, and I spent a lot of time thinking about how to deliver this object to other objects in the beginning. Through this difficulty, I learned and practiced the concept of the pointer.

Previously, the BFS algorithm we studied was to explore the already formed graph, but it was impressive in that this Puzzle Problem creates a new case every time, checks if the case is the goal state puzzle, and expands it further if not. In other words, it was very impressive that it was not done with a graph that had already been completed, but rather expanded together with the search process. In addition, I had never actively utilized graph search algorithms to write solutions to any problems before, but this assignment provided a good opportunity to solve problems using graph search algorithms.

In the future study, I want to implement the solution to this problem by using the "DFS" and "A*" algorithms.

□ Codes

```
#include <iostream>
#include <list>
using namespace std;

// constant for indicate
#define UP 1
#define DOWN 2
#define LEFT 3
#define RIGHT 4

// class definition
class Node
{
public:
    int puzzle[9];          // puzzle data array
    list<Node*> child_node; // child node of node
    Node* parent;           // parent node

    void set_puzzle(const int arr[]);
    int empty_index();
    bool isGoal(const int arr[]);
    void block_move(int op);
    void clone_puzzle(int* new_puzzle, const int* origin_puzzle);
    void print_puzzle();
    bool equal_check(int arr[]);
};

// function definition
list<Node*>* breadth_first_search(Node* root, int *target);
bool contains (list<Node*>* list, Node *np);
void solution_trace(list<Node*>* solution, Node* np);

int main()
{
    int init1[9] = {1, 3, 2, 0, 7, 8, 5, 6, 4}; // initial case 1
    int init2[9] = {3, 7, 0, 1, 6, 2, 5, 4, 8}; // initial case 2
    int init3[9] = {7, 5, 6, 3, 4, 2, 1, 0, 8}; // initial case 3
    int target[9] = {1, 2, 3, 8, 0, 4, 7, 6, 5}; // target(final) case

    Node* root = new Node;
    root->set_puzzle(init1);

    list<Node*>* solution = breadth_first_search(root, target);
    cout << "step = " << solution->size() << endl;
    solution->reverse();

    if(!solution->empty()) {
        for(Node* node : *solution) {
            node->print_puzzle();
            cout << endl;
        }
    }
    else
        cout << "Failed to find solutions..." << endl;
    return 0;
}

// method definition
void Node:: set_puzzle(const int arr[])
{
    for(int i=0 ; i<9 ; i++)
        puzzle[i] = arr[i];
}

int Node:: empty_index()
{
    for(int i=0 ; i<9 ; i++) {
        if(puzzle[i] == 0)
            return i;
    }
    return -1;
}

bool Node:: isGoal(const int arr[])
{
    int counter = 0;
    for(int i=0 ; i<9 ; i++) {
        if(puzzle[i] != arr[i]) break;
        counter++;
    }
    if(counter < 9) return false;
    else return true;
}

void Node:: clone_puzzle(int *new_puzzle, const int *origin_puzzle)
{
    for(int i=0 ; i<9 ; i++)
        new_puzzle[i] = origin_puzzle[i];
}

void Node:: block_move(int op)
{
    int zero_index = empty_index();
    int tmp_arr[9];
    int tmp;

    if(op == UP && zero_index / 3 != 0) {
        clone_puzzle(tmp_arr, puzzle);
        tmp = tmp_arr[zero_index-3];
        tmp_arr[zero_index-3] = 0;
        tmp_arr[zero_index] = tmp;

        Node* child = new Node;
        child->set_puzzle(tmp_arr);
        child_node.push_back(child);
        child->parent = this;
    }
    else if(op == DOWN && zero_index / 3 != 2) {
        clone_puzzle(tmp_arr, puzzle);

        tmp = tmp_arr[zero_index+3];
        tmp_arr[zero_index+3] = 0;
        tmp_arr[zero_index] = tmp;

        Node* child = new Node;
        child->set_puzzle(tmp_arr);
        child_node.push_back(child);
        child->parent = this;
    }
    else if(op == LEFT && zero_index % 3 != 0) {
        clone_puzzle(tmp_arr, puzzle);

        tmp = tmp_arr[zero_index-1];
        tmp_arr[zero_index-1] = 0;
        tmp_arr[zero_index] = tmp;

        Node* child = new Node;
        child->set_puzzle(tmp_arr);
        child_node.push_back(child);
        child->parent = this;
    }
    else if(op == RIGHT && zero_index % 3 != 2) {
        clone_puzzle(tmp_arr, puzzle);

        tmp = tmp_arr[zero_index+1];
        tmp_arr[zero_index+1] = 0;
        tmp_arr[zero_index] = tmp;

        Node* child = new Node;
        child->set_puzzle(tmp_arr);
        child_node.push_back(child);
        child->parent = this;
    }
}

void Node:: print_puzzle()
{
    for(int i=0 ; i<9 ; i++) {
        if((i+1) % 3 == 0)
            cout << puzzle[i] << endl;
        else
            cout << puzzle[i] << " ";
    }
}

bool Node::equal_check(int arr[])
```

```
{
    for(int i=0 ; i<9 ; i++)
        if(puzzle[i] != arr[i])
            return false;
    return true;
}

// function definition
bool contains(list<Node*>* list, Node* np)
{
    bool contains = false;

    for(Node* node : *list) {
        if(node->equal_check(np->puzzle))
            contains = true;
    }
    return contains;
}

list<Node*>* breadth_first_search(Node* root, int* target)
{
    list<Node*>* solution = new list<Node*>;
    list<Node*>* open = new list<Node*>;
    list<Node*>* close = new list<Node*>;

    open->push_back(root);
    bool goalCheck = false;

    while(!open->empty() && !goalCheck) {
        Node* curNode = open->front();
        close->push_back(curNode);
        open->pop_front();

        curNode->block_move(UP);
        curNode->block_move(DOWN);
        curNode->block_move(LEFT);
        curNode->block_move(RIGHT);

        for(Node* node : curNode->child_node)
        {
            if(node->isGoal(target)) {
                cout << "solution was founded..." << endl;
                goalCheck = true;
                solution_trace(solution, node);
            }

            if(!contains(open, node) && !contains(close, node))
                open->push_back(node);
        }
    }
    return solution;
}

void solution_trace(list<Node*>* solution, Node* np)
{
    cout << "Trace Solution" << endl;
    solution->push_back(np);
    while(np->parent != NULL)
    {
        np = np->parent;
        solution->push_back(np);
    }
}
```