Project1-26-Puzzel Problem

Groupmates: Runze Li (Nid: Nxxxxxxx)

Jiayi Li (Nid: Nxxxxxxx)

Date: Nov 5, 2023

26-puzzle Problem

Elaboration of A* Algorithm

1. Generate the root node, which contains the initial state of the 26-puzzle problem.
2. Add the root node into the frontier, which is arranged in ascending order by the evaluation function f(n).
3. Add the initial state and the corresponding value of f(n) into the reached state table.
4. While the frontier is not empty, pop the first node of it. Check if that node contains the goal state. If not, expand that node. If it contains the goal state, the algorithm finds the optimal solution.
    - 'Expand' means that, apply all applicable actions to the current node, generating child nodes. Calculate f(n) for each child node.
5. Check if each child node has a repeated state of the existing state in the reached state table. If the current state is not in the table, add the state into the table and add the current child node into the frontier. Do not need to compare f(n) value of the new node with existing node in the reached table because that the hueristic function in this problem is Manhattan distance, which is admissable and consistent.

Explanation of the Code

- The comments of the code is included in the 'Source Code' section of the report.
- Programming language: C++
- The origin of the coordinate is defined at the lower left corner of the bottommost grid.
- Definition of data types
    - Define a struct 'node' which includes 7 variables:

```
typedef struct node {
  vector<vector<vector<int>>> state;
  string action;
  int depth;
  int manhattan_Distance;
  int f;
  vector<int> f_sequence;
  vector<int> blank_Position;
} node;
```

        - state of the node, which is represented by three-dimensional vector
        - action sequence from the root node to the current node
        - depth of current node, that is, g(n)
        - manhattan distance of current node, that is, h(n)
        - evaluation function of the current node, that is, f(n), which is equals to g(n)+h(n)
        - sequence of f(n) of each node along the path from the root node to the current node
        - the coordinate of the blank position of the current node
    - The reached state table is reprensented by the 'map'. The key is the state of the node and value is f(n) of the node.

26-puzzle Problem

Elaboration of A* Algorithm

1. Generate the root node, which contains the initial state of the 26-puzzle problem.
2. Add the root node into the frontier, which is arranged in ascending order by the evaluation function f(n).
3. Add the initial state and the corresponding value of f(n) into the reached state table.
4. While the frontier is not empty, pop the first node of it. Check if that node contains the goal state. If not, expand that node. If it contains the goal state, the algorithm finds the optimal solution.
    - 'Expand' means that, apply all applicable actions to the current node, generating child nodes. Calculate f(n) for each child node.
5. Check if each child node has a repeated state of the existing state in the reached state table. If the current state is not in the table, add the state into the table and add the current child node into the frontier. Do not need to compare f(n) value of the new node with existing node in the reached table because that the hueristic function in this problem is Manhattan distance, which is admissable and consistent.

Explanation of the Code

- The comments of the code is included in the 'Source Code' section of the report.
- Programming language: C++
- The origin of the coordinate is defined at the lower left corner of the bottommost grid.
- Definition of data types
    - Define a struct 'node' which includes 7 variables:

```
typedef struct node {
  vector<vector<vector<int>>> state;
  string action;
  int depth;
  int manhattan_Distance;
  int f;
  vector<int> f_sequence;
  vector<int> blank_Position;
} node;
```

        - state of the node, which is represented by three-dimensional vector
        - action sequence from the root node to the current node
        - depth of current node, that is, g(n)
        - manhattan distance of current node, that is, h(n)
        - evaluation function of the current node, that is, f(n), which is equals to g(n)+h(n)
        - sequence of f(n) of each node along the path from the root node to the current node
        - the coordinate of the blank position of the current node
    - The reached state table is reprensented by the 'map'. The key is the state of the node and value is f(n) of the node.

```
map<vector<vector<vector<int>>>, int> reached;
```

- The frontier is represented by the 'Priority_queue'. The queue is ordered by 'cmp', an operator, which is overloaded to compare the f(n) value of the nodes in the frontier.

```
priority_queue<node*, vector<node*>, cmp> frontier;
```

- Definition of functions
  - Calculate h(n) of the node with the function below, using three nested loops to add the absolute values of difference of three axis between the current state of the goal state.

```
int calculate_Manhattan_Distance(vector<vector<vector<int>>> state)
```

  - Figure out the coordinate of the blank position with the function below.

```
vector<int> find_Blank_Position(vector<vector<vector<int>>> state)
```

  - Check if the current state is the same as the goal state with the function below.

```
bool check_Goal_State(vector<vector<vector<int>>> state)
```

  - Expand the child nodes of the current node, which has been popped from the frontier.

```
vector<node*> expand_State(node* front_node)
```

    - Get the x, y, z value of the coordinate of the blank position as blank_x, blank_y, blank_z
    - For different current states, there are different applicable action sets. For instance, if blank_x is equal to or greater than 2, the 'East' action can not be applied. It needs 6 conditional statements to determine the actions for each node.
    - If the action is applicable, add the action to the action sequence of the current child node. Add one to the current child node. Calculate the h(n) and f(n) of the child node. Add the f(n) to the sequence of f(n) of the current child node. Obtain the blank position of the child node. Add the child node to the child nodes of the front_node.
    - Return all the child nodes of the front_node.
- Main function
  - Read the input file using the ifstream. Store the data in the input file into two variables: initial state and goal state.
  - Initialize the problem by creating the root node.

- While the frontier is not empty:
  - Pop the first node.

```
frontier.pop();
```

  - Check if the popped node is the goal node.

```
if (check_Goal_State(front_node->state))
```

  - Expand the non-goal node.

```
vector<node*> children_states = expand_State(front_node);
```

  - For each of the child nodes, check if the state it has is a repeated state.

```
if (reached.find(child_state->state) != reached.end()) continue;
else {
    reached.emplace(child_state->state, child_state->f);
    frontier.push(child_state);
    }
```

- Write the result into the output file using ofstream. The line 25 shows the depth of the optimal goal node. The line 26 shows the counts of nodes generated by the algorithm. The line 27 shows the action sequence along the optimal path. The line 28 shows the f(n) value of each node on the solution path.

Instructions of Compiling and Running the Code

- Input the below instructions in cmd in Windows:

```
g++ -o 26-puzzle 26-puzzle.cpp
./26-puzzle.exe
```

- If we use a txt file with another name as an input file or output file, we need to modify the name of the input file in the 26-puzzle.cpp program. For example, if we use 'input1.txt' as input file and 'output1.txt' as output file, we need to modify two lines(line 291 and line 413) in the program:

```
ifstream infile("input1.txt", ios::in);
outfile.open("output.txt", ios::out);
```

Output Files of Three Test Input Files

- Output file for the Input1 file:

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

9 10 11
12 13 14
15 16 17

18 19 20
21 22 23
24 25 26

1 2 3
4 13 5
6 7 8

9 10 11
15 12 14
24 16 17

18 19 20
21 0 23
25 22 26

6
23
D W S D E N
6 6 6 6 6 6 6
```

- Output file for the Input2 file:

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

9 10 11
12 13 14
15 16 17

18 19 20
21 22 23
24 25 26

1 10 2
4 5 3
6 7 8
```

```
9 13 11
21 12 14
15 16 17

18 0 20
24 19 22
25 26 23

13
44
E N W D S W D S E E N W N
13 13 13 13 13 13 13 13 13 13 13 13 13 13
```

- Output file for the Input3 file:

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

9 10 11
12 13 14
15 16 17

18 19 20
21 22 23
24 25 26

0 2 3
1 7 14
6 8 5

12 9 10
4 13 11
21 16 17

18 19 20
22 25 23
15 24 26

16
59
S E N D N W W S D E S W U N U N
16 16 16 16 16 16 16 16 16 16 16 16 16 16 16 16
```

Source Code

```
#include<iostream>
#include<fstream>
```

```
#include<iomanip>
#include<vector>
#include<map>
#include<queue>
#include<cmath>

using namespace std;

// The structure of state, including the state, f(n), etc.
typedef struct node {
    vector<vector<vector<int>>> state;
    string action;              // store the action along the path
    int depth;                  // g(n)
    int manhattan_Distance;     // h(n)
    int f;                      // f(n) = g(n) + h(n)
    vector<int> f_sequence;     // store the f(n) along the path
    vector<int> blank_Position; // the position of blank position: x, y, z
} node;

struct cmp {
    bool operator()(node* a, node* b) {
        return a->f > b->f;
    }
};


/*
 * We use a map in C++ programming language as hash table,
 * which is used to determine whether the state is reached.
 *
 * reached[state] = f(state)
 */
map<vector<vector<vector<int>>>, int> reached;
/*
 * Initialize the initial state and goal state(3*3*3)
 */
vector<vector<vector<int>>> initial_state(3, vector<vector<int>>(3, vector<int>(3,
0)));
vector<vector<vector<int>>> goal_state(3, vector<vector<int>>(3, vector<int>(3,
0)));
/*
 * Initialize the index of every value in goal state(27*3)
 * index_of_goal_state[value] = {x, y, z}
 */
vector<vector<int>> index_of_goal_state(27, vector<int>(3, 0));
priority_queue<node*, vector<node*>, cmp> frontier;

/*
 * Calculate the manhattan distance = abs(delta(x)) + abs(delta(y)) +
abs(delta(z))
 */
int calculate_Manhattan_Distance(vector<vector<vector<int>>> state) {
    int manhattan_Distance = 0;
    for (int x=0; x<3; x++) {
```

```
        for (int y=0; y<3; y++) {
            for (int z=0; z<3; z++) {
                if (state[x][y][z] == 0) continue;
                manhattan_Distance += abs(x - index_of_goal_state[state[x][y][z]]
[0]);
                manhattan_Distance += abs(y - index_of_goal_state[state[x][y][z]]
[1]);
                manhattan_Distance += abs(z - index_of_goal_state[state[x][y][z]]
[2]);
            }
        }
    }
    return manhattan_Distance;
}

/*
 * Find the index of blank position {x, y, z}
 */
vector<int> find_Blank_Position(vector<vector<vector<int>>> state) {
    vector<int> blank_position(3, 0);
    for (int x=0; x<3; x++) {
        for (int y=0; y<3; y++) {
            for (int z=0; z<3; z++) {
                if (state[x][y][z] == 0) {
                    blank_position[0] = x;
                    blank_position[1] = y;
                    blank_position[2] = z;
                    return blank_position;
                }
            }
        }
    }
    return blank_position;
}

/*
 * Check whether it is goal state
 */
bool check_Goal_State(vector<vector<vector<int>>> state) {
    for (int x=0; x<3; x++) {
        for (int y=0; y<3; y++) {
            for (int z=0; z<3; z++) {
                if (state[x][y][z] != goal_state[x][y][z]) {
                    return false;
                }
            }
        }
    }
    return true;
}

/*
 * Expand the node and get node's children
 */
```

```
vector<node*> expand_State(node* front_node) {
    vector<node*> children;
    int blank_x = front_node->blank_Position[0];
    int blank_y = front_node->blank_Position[1];
    int blank_z = front_node->blank_Position[2];

    /*
     * Get next valid state using a action in {E, W, N, S, U, D}
     */

    /*
     * Action East
     */
    if (blank_x < 2) {
        node* child_East = new node();
        /*
         * Move blank position to the east
         */
        child_East->state.assign(front_node->state.begin(), front_node->state.end());
        swap(child_East->state[blank_x][blank_y][blank_z], child_East->state[blank_x + 1][blank_y][blank_z]);
        /*
         * Get the new action
         */
        child_East->action = front_node->action;
        child_East->action += "E";
        /*
         * Add one to the depth level of child state
         */
        child_East->depth = front_node->depth + 1;
        /*
         * Get the new manhattan distance
         */
        child_East->manhattan_Distance = calculate_Manhattan_Distance(child_East->state);
        /*
         * f(n) = g(n) + h(n)
         */
        child_East->f = child_East->depth + child_East->manhattan_Distance;
        /*
         * Add the new f(n) to the sequence
         */
        child_East->f_sequence.assign(front_node->f_sequence.begin(), front_node->f_sequence.end());
        child_East->f_sequence.push_back(child_East->f);
        /*
         * Get the new index of blank position
         */
        child_East->blank_Position = find_Blank_Position(child_East->state);

        children.push_back(child_East);
    }
```

```
    /*
     * Action West
     */
    if (blank_x > 0) {
        node* child_West = new node();

        child_West->state.assign(front_node->state.begin(), front_node-
>state.end());
        swap(child_West->state[blank_x][blank_y][blank_z], child_West-
>state[blank_x - 1][blank_y][blank_z]);

        child_West->action = front_node->action;
        child_West->action += "W";

        child_West->depth = front_node->depth + 1;

        child_West->manhattan_Distance = calculate_Manhattan_Distance(child_West-
>state);

        child_West->f = child_West->depth + child_West->manhattan_Distance;

        child_West->f_sequence.assign(front_node->f_sequence.begin(), front_node-
>f_sequence.end());
        child_West->f_sequence.push_back(child_West->f);

        child_West->blank_Position = find_Blank_Position(child_West->state);

        children.push_back(child_West);
    }

    /*
     * Action North
     */
    if (blank_y < 2) {
        node* child_North = new node();

        child_North->state.assign(front_node->state.begin(), front_node-
>state.end());
        swap(child_North->state[blank_x][blank_y][blank_z], child_North-
>state[blank_x][blank_y + 1][blank_z]);

        child_North->action = front_node->action;
        child_North->action += "N";

        child_North->depth = front_node->depth + 1;

        child_North->manhattan_Distance =
calculate_Manhattan_Distance(child_North->state);

        child_North->f = child_North->depth + child_North->manhattan_Distance;

        child_North->f_sequence.assign(front_node->f_sequence.begin(), front_node-
>f_sequence.end());
        child_North->f_sequence.push_back(child_North->f);
```

```
        child_North->blank_Position = find_Blank_Position(child_North->state);

        children.push_back(child_North);
    }

    /*
     * Action South
     */
    if (blank_y > 0) {
        node* child_South = new node();

        child_South->state.assign(front_node->state.begin(), front_node-
>state.end());
        swap(child_South->state[blank_x][blank_y][blank_z], child_South-
>state[blank_x][blank_y - 1][blank_z]);

        child_South->action = front_node->action;
        child_South->action += "S";

        child_South->depth = front_node->depth + 1;

        child_South->manhattan_Distance =
calculate_Manhattan_Distance(child_South->state);

        child_South->f = child_South->depth + child_South->manhattan_Distance;

        child_South->f_sequence.assign(front_node->f_sequence.begin(), front_node-
>f_sequence.end());
        child_South->f_sequence.push_back(child_South->f);

        child_South->blank_Position = find_Blank_Position(child_South->state);

        children.push_back(child_South);
    }

    /*
     * Action Up
     */
    if (blank_z < 2) {
        node* child_Up = new node();

        child_Up->state.assign(front_node->state.begin(), front_node-
>state.end());
        swap(child_Up->state[blank_x][blank_y][blank_z], child_Up->state[blank_x]
[blank_y][blank_z + 1]);

        child_Up->action = front_node->action;
        child_Up->action += "U";

        child_Up->depth = front_node->depth + 1;

        child_Up->manhattan_Distance = calculate_Manhattan_Distance(child_Up-
>state);
```

```
        child_Up->f = child_Up->depth + child_Up->manhattan_Distance;

        child_Up->f_sequence.assign(front_node->f_sequence.begin(), front_node-
>f_sequence.end());
        child_Up->f_sequence.push_back(child_Up->f);

        child_Up->blank_Position = find_Blank_Position(child_Up->state);

        children.push_back(child_Up);
    }

    /*
     * Action Down
     */
    if (blank_z > 0) {
        node* child_Down = new node();

        child_Down->state.assign(front_node->state.begin(), front_node-
>state.end());
        swap(child_Down->state[blank_x][blank_y][blank_z], child_Down-
>state[blank_x][blank_y][blank_z - 1]);

        child_Down->action = front_node->action;
        child_Down->action += "D";

        child_Down->depth = front_node->depth + 1;

        child_Down->manhattan_Distance = calculate_Manhattan_Distance(child_Down-
>state);

        child_Down->f = child_Down->depth + child_Down->manhattan_Distance;

        child_Down->f_sequence.assign(front_node->f_sequence.begin(), front_node-
>f_sequence.end());
        child_Down->f_sequence.push_back(child_Down->f);

        child_Down->blank_Position = find_Blank_Position(child_Down->state);

        children.push_back(child_Down);
    }

    return children;
}

int main() {

    // Read the file
    ifstream infile("input3.txt", ios::in);
    if (!infile.is_open()) {
        cout << "open error!" << endl;
        return 0;
    }
```

```
    /*
     * Store the initial state and goal state into the array
     *
     */
    for (int z=2; z>=0; z--) {
        for (int y=2; y>=0; y--) {
            for (int x=0; x<3; x++) {
                if (infile.eof()) {
                    cout << "read error" << endl;
                    return 0;
                }
                infile >> initial_state[x][y][z];
            }
        }
    }
    for (int z=2; z>=0; z--) {
        for (int y=2; y>=0; y--) {
            for (int x=0; x<3; x++) {
                if (infile.eof()) {
                    cout << "read error" << endl;
                    return 0;
                }
                infile >> goal_state[x][y][z];
            }
        }
    }

    /*
     * Store the index of every value in goal state
     */
    for (int x=0; x<3; x++) {
        for (int y=0; y<3; y++) {
            for (int z=0; z<3; z++) {
                index_of_goal_state[goal_state[x][y][z]][0] = x;
                index_of_goal_state[goal_state[x][y][z]][1] = y;
                index_of_goal_state[goal_state[x][y][z]][2] = z;
            }
        }
    }

    node* start_node = new node();
    start_node->state.assign(initial_state.begin(), initial_state.end());
    start_node->action = "";
    start_node->depth = 0;
    start_node->manhattan_Distance = calculate_Manhattan_Distance(start_node-
>state);
    start_node->f = start_node->depth + start_node->manhattan_Distance;
    start_node->f_sequence.push_back(start_node->f);
    start_node->blank_Position = find_Blank_Position(start_node->state);


    /*
     * Firstly, we need to push the start node into priority queue.
     */
```

```
    frontier.push(start_node);
    reached.emplace(start_node->state, start_node->f);

    node* front_node;
    bool find_Goal = false;

    while (!frontier.empty()) {
        /*
         * Get the head of the frontier
         */

        front_node = frontier.top();
        frontier.pop();


        /*
         * Determine whether the node is goal state
         */
        if (check_Goal_State(front_node->state)) {
            /*
             * We have successfully found the result.
             */
            find_Goal = true;
            break;
        }


        /*
         * Expand the node, i.e. get the next step of the state
         */
        vector<node*> children_states = expand_State(front_node);


        for (auto child_state : children_states) {


            /*
             * Check whether the child state isn't in 'reached' table
             *
             */
            if (reached.find(child_state->state) != reached.end()) continue;
            else {
                /*
                 * Add the child state to 'reached' table and push it into
frontier
                 */
                reached.emplace(child_state->state, child_state->f);
                frontier.push(child_state);
            }
        }
    }

    if (find_Goal == false) {
        cout << "We can't find the optimal path from initial state to goal state."
```

```
<< endl;
        return 0;
    }

    /*
     * Write the initial state and goal state in the output file.
     * Line 1-11:  initial state
     * Line 13-23: goal state
     *
     */
    ofstream outfile;
    outfile.open("output.txt", ios::out);
    for (int z=2; z>=0; z--) {
        for (int y=2; y>=0; y--) {
            for (int x=0; x<3; x++) {
                outfile << initial_state[x][y][z] << " ";
            }
            outfile << endl;
        }
        outfile << endl;
    }
    for (int z=2; z>=0; z--) {
        for (int y=2; y>=0; y--) {
            for (int x=0; x<3; x++) {
                outfile << goal_state[x][y][z] << " ";
            }
            outfile << endl;
        }
        outfile << endl;
    }

    /*
     * We output the last 4 line.
     *       Line 25: the depth level d of the shallowest goal node as found
     *                by the A* algorithm (assume the root node is at level 0).
     *       Line 26: the total number of nodes N generated in my tree
     *                (including the root node).
     *       Line 27: the solution, i.e. a sequence of actions from root node
     *                to goal node represented by A's. The A's are separated
     *                by blank spaces. Each A is a character from the set
     *                {E, W, N, S, U, D}, representing the movements of the
     *                blank position(0).
     *       Line 28: f(n) value of the nodes along the solution path, from
     *                the root node to the goal node, separated by blank spaces.
     */
    outfile << front_node->depth << endl;
    outfile << reached.size() << endl;
    for (int i=0; i < front_node->action.length(); i++) {
        outfile << front_node->action[i] << " ";
    }
    outfile << endl;
    for (int i=0; i < front_node->f_sequence.size(); i++) {
        outfile << front_node->f_sequence[i] << " ";
    }
```

```
    outfile << endl;

    // Close the file
    outfile.close();

    return 0;
}
```