Zafir Hasan 11/5/18
CS-UY 4613

Compile And Run

1. Download MinGW from https://osdn.net/projects/mingw/releases/
2. Install the mingw32-gcc-g++-bin (version 6.3.0-1) from the MinGW installation manager
3. Add C:\MinGW\bin to PATH environment variable
4. Open terminal and cd into directory where main.cpp lives
5. Run **g++ -g main.cpp -o main.exe** in the terminal (make sure all input files are in same directory as main.cpp
6. If you encounter an error on step 4, run **g++ -std=c++11 main.cpp -o main.exe**
7. Run **main** from terminal

Output1.txt

2 8 3
1 6 4
7 0 5

1 2 3
8 0 4
7 6 5

5
12
U U L D R

Output2.txt

2 8 3
7 1 6
0 5 4

1 2 3
8 0 4
7 6 5

10
33
U R U L D R R D L U

Output3.txt

1 0 6
4 2 3
7 5 8

7 4 3
5 0 6
2 8 1

17
166
L D D R U U R D L U L D D R R U L

Output4.txt

8 1 2
3 6 4
5 0 7

4 2 1
8 7 0
3 5 6

14
111
U R U L D R D L L U U R D R

main.cpp

```cpp
#include <iostream>
#include <fstream>
#include <string>
#include <vector>
#include <cstdlib>
#include <queue>
#include <array>
#include <math.h>

using namespace std;

// The node class will contain member variables that contain the state of the
node, the path-cost up to that node from the root node, and the path up to that
node from the root node.
class Node {
public:
```

```
    Node(vector<vector<int>> state) : state(state), pathCost(0), path({}) {}
    Node(vector<vector<int>> state, int pathCost, vector<char> path) :
state(state), pathCost(pathCost), path(path) {}

    void displayState() const {
        string stringstate;
        for (size_t i = 0; i < state.size(); i++) {
            for (size_t j = 0; j < state[i].size(); j++) {
                stringstate += to_string(state[i][j]) + ' ';
            }
            stringstate += '\n';
        }
        cout << "State: \n" << stringstate << endl;
    }
    void displayPathCost() const {
        cout << "Path cost: " << pathCost << endl;
    }
    void displayPath() const {
        string strpath;
        for (size_t i = 0; i < path.size(); i++) {
                strpath = strpath + path[i] + " ";
        }
        cout << "Path: " << strpath << endl;

    }

    int getPathCost() const {
        return pathCost;
    }

    vector<char> getPath() const {
        return path;
    }

    vector<vector<int>> getState() const {
        return state;
    }

private:
    vector<vector<int>> state;
    int pathCost;
    vector<char> path;

};
```

```
// This function creates the output file and writes to the file in the format
specified in the project problem once the solution node has been found
void solution(const string ofstring, const Node* node, const vector<vector<int>>&
initialState, const vector<vector<int>>& goalState, const int generated) {
    ofstream outputFile(ofstring);
    string fstring;
    for (size_t i = 0; i < initialState.size(); i++) {
        for (size_t j = 0; j < initialState[i].size(); j++) {
            fstring += to_string(initialState[i][j]) + " ";
        }
        fstring += '\n';
    }
    fstring += '\n';
    for (size_t i = 0; i < goalState.size(); i++) {
        for (size_t j = 0; j < goalState[i].size(); j++) {
            fstring += to_string(goalState[i][j]) + " ";
        }
        fstring += '\n';
    }
    fstring += '\n';
    vector<char> path = node->getPath();
    fstring += to_string(path.size()) + '\n';
    fstring += to_string(generated) + '\n';
    for (size_t i = 0; i < path.size(); i++) {
            fstring = fstring + path[i] + " ";
            cout << path[i] << " " << endl;
    }
    outputFile << fstring;
    outputFile.close();
}

// This function returns the manhattan distance of one tile from its goal
position
int tileDist(int tile, const vector<vector<int>>& currState, const
vector<vector<int>>& goalState) {

    vector<int> stateCoor;
    vector<int> goalCoor;
    for (size_t i = 0; i < currState.size(); i++) {
        for (size_t j = 0; j < currState[i].size(); j++) {
            if (currState[i][j] == tile) {
                stateCoor.push_back(i);
                stateCoor.push_back(j);
            }
            if (goalState[i][j] == tile) {
```

```
                goalCoor.push_back(i);
                goalCoor.push_back(j);
            }
        }
    }

    int xdiff = abs(stateCoor[0] - goalCoor[0]);
    int ydiff = abs(stateCoor[1] - goalCoor[1]);

    int tileDist = xdiff + ydiff;
    return tileDist;

}

// This function returns the sum of Manhattan distances of all tiles (except
blank) from their goal position
int heuristic(const vector<vector<int>>& currState, const vector<vector<int>>&
goalState) {
    int sumDist = 0;
    for (int i = 1; i < 9; i++) {
        sumDist += tileDist(i, currState, goalState);
    }
    return sumDist;
}

// This function returns true if the state is equivalent to the goal state. It is
equivalent when the sum of Manhattan distances of the tiles from their goal
position is equal to 0. It returns false otherwise.
bool goalTest(const vector<vector<int>>& state, const vector<vector<int>>& goal)
{
    return heuristic(state, goal) == 0;
}

// This function returns a vector containing all possible actions doable from a
given state
vector<char> actions(const vector<vector<int>>& state) {
    vector<char> vecActions;
    vector<int> coors;
    for (size_t i = 0; i < state.size(); i++) {
        for (size_t j = 0; j < state[i].size(); j++) {
            if (state[i][j] == 0) {
                coors.push_back(i);
                coors.push_back(j);
            }
        }
```

```
    }

    if (coors[0] == 0) {
        vecActions.push_back('D');
    }
    else if (coors[0] == 1) {
        vecActions.push_back('D');
        vecActions.push_back('U');
    }
    else {
        vecActions.push_back('U');
    }


    if (coors[1] == 0) {
        vecActions.push_back('R');
    }
    else if (coors[1] == 1) {
        vecActions.push_back('R');
        vecActions.push_back('L');
    }
    else {
        vecActions.push_back('L');
    }

    return vecActions;
}

// remove the node with the lowest pathCost from the frontier and return the
pointer
Node* pop(vector<Node*>& frontier) {
    int minCost = 100;
    int minIndex = 0;
    for (size_t i = 0; i < frontier.size(); i++) {
        int pathCost = frontier[i]->getPathCost();
        if (pathCost < minCost) {
            minCost = pathCost;
            minIndex = i;
        }
    }
    Node* nodeptr = frontier[minIndex];
    frontier.erase((frontier.begin()+minIndex));
    return nodeptr;
}
```

```
// This function creates a child node using the state of the current node, a
legal action, and the goal state for the problem
Node* childNode(const Node* currNode, const char action, const
vector<vector<int>>& goalState) {
    vector<vector<int>> childState = currNode->getState();

    vector<char> path = currNode->getPath();
    path.push_back(action);

    int fcost = currNode->getPathCost() - heuristic(currNode->getState(),
goalState) + 1;

    int temp;
    int x;
    int y;
    for (size_t i = 0; i < childState.size(); i++) {
        for (size_t j = 0; j < childState[i].size(); j++) {
            if (childState[i][j] == 0) {
                x = i;
                y = j;
            }
        }
    }
    if (action == 'U') {
        temp = childState[x-1][y];
        childState[x-1][y] = 0;
    }
    else if (action == 'D') {
        temp = childState[x+1][y];
        childState[x+1][y] = 0;
        // childState[x][y] = temp;
    }
    else if (action == 'L') {
        temp = childState[x][y-1];
        childState[x][y-1] = 0;
        // childState[x][y] = temp;
    }
    else {
        temp = childState[x][y+1];
        childState[x][y+1] = 0;
        // childState[x][y] = temp;
    }
    childState[x][y] = temp;
```

```
    int pathCost = heuristic(childState, goalState) + fcost;
    Node* child = new Node(childState, pathCost, path);
    return child;

}


bool equals(const vector<vector<int>>& s, const vector<vector<int>>& t) {
    for (size_t i = 0; i < s.size(); i++) {
        for (size_t j = 0; j < s[i].size(); j++) {
            if (s[i][j] != t[i][j]) {
                return false;
            }
        }
    }
    return true;
}

// Check if a state is inside frontier or explored
bool contains(const vector<vector<int>>& state, const vector<Node*>& nodevec) {

    for (size_t i = 0; i < nodevec.size(); i++) {
        if(equals(state, nodevec[i]->getState())) {
            return true;
        }
    }
    return false;
}



// GRAPH-SEARCH A* using Sum of Manhattan distances of tiles from their goal
position as heuristic.
void graphSearchA(const string file, const string ofile) {

    // Read from input file if it exists
    ifstream ifs(file);
    if (!ifs) {
        cerr << "Could not open the file.\n";
        exit(1);
    }
    else {
        cout << "File has been opened successfully" << endl;
    }
```

```
    int tile;
    int counter = 0;
    vector< vector<int> > initialState(3);
    vector< vector<int> > goalState(3);

    int i = 0;
    int j = 0;
    // Set up initial state and goal state using input file
    while (ifs >> tile) {
        if (i < 3) {
            initialState[i].push_back(tile);
        }
        else {
            goalState[i-3].push_back(tile);
        }
        j++;
        if (j % 3 == 0) {
            i++;
        }
    }
    ifs.close();

    // Set up root node and push it to frontier
    Node* root = new Node(initialState, heuristic(initialState, goalState), {});
    cout << "set up root" << endl;
    vector<Node*> frontier;
    frontier.push_back(root);
    vector<Node*> explored;
    // Set the number of nodes generated to 1
    int generated = 1;

    while(true) {
        // If the frontier is empty, the search has failed and we will return an
error
        if (frontier.empty()) {
            cerr << "Search has failed" << endl;
            exit(1);
        }
        // Otherwise, pop the frontier, getting the node with the lowest pathCost
        Node* n = pop(frontier);
        cout << "Popped frontier" << endl;

        // Outputting node parameters for testing purposes
        n->displayState();
        n->displayPath();
```

```
        n->displayPathCost();

        // If the popped node has a h(n) value of 0, we know its the goal node.
We will then return the solution that will generate the output file with all the
appropriate parameters
        if (goalTest(n->getState(), goalState)) {
            return solution(ofile, n, initialState, goalState, generated);
        }

        // Otherwise, push the node to the explored data structure
        explored.push_back(n);
        vector<char> possActions = actions(n->getState());
        // For every action possible from the node n, create the child node using
the action and n. Then, check if the state of the child node is not in frontier
and not in explored.
        for (size_t i = 0; i < possActions.size(); i++) {
            cout << possActions[i] << " " << endl;
            Node* child = childNode(n, possActions[i], goalState);
            // cout << "Created child" << endl;

            // Push child node to frontier and increase nodes generated by one if
the state of the child node is in neither the frontier nor the explored data
structure
            if (!(contains(child->getState(), frontier)) && !(contains(child-
>getState(), explored))) {
                generated++;
                cout << "in frontier: " << to_string(contains(child->getState(),
frontier)) << endl;
                cout << "in explored: " << to_string(contains(child->getState(),
explored)) << endl;
                frontier.push_back(child);
                cout << "Not in frontier or explored, let's add" << endl;

            }
        }
        cout << frontier.size() << endl;
    }
}

int main() {

    graphSearchA("Input1.txt", "Output1.txt");
    graphSearchA("Input2.txt", "Output2.txt");
    graphSearchA("Input3.txt", "Output3.txt");
    graphSearchA("Input4.txt", "Output4.txt");
```

```
}
```