Lisa Ye
04/09/2020
CS 4613
15 Puzzle

Instructions to Run

*Along with the requested source code in plain text file, I have also attached the original .cpp file

Programming Language: C++
IDE used: Visual Studio C++ 2017

To Compile the Code:

<u>If you are using an IDE:</u>

1. Include the source code (probably need the .cpp file) in the project/workspace/solution
2. Hit whatever button that is equivalent to run on the IDE.

<u>If you are using command line:</u>

1. Make sure you are inside the correct directory
2. Type in the following to compile:
   gcc main.cpp -o puzzle
3. Afterwards, type in the following to run the program:
   puzzle

To Run the Program:

- The program will prompt you to input the input file name and the output file name you want to generate.
- Make sure you have the input files in the same folder as the source code. After you run the program, the output will be generated in the same folder as the source code named after the name you input.
- On Windows, you need to include the ".txt" extension while entering the file name. Not sure on other operating systems, but include it just to be safe.

On the following page I will include first the source code (it's a bit long) then Output1-Output4 respectively.

```
// Lisa Ye
// CS 4613 AI Project 1 15 Puzzle
// This program solves the 15 puzzle problem by using A* Search

#include<string>
#include<iostream>
#include<fstream>
#include<vector>
#include<algorithm>
#include<queue>
#include<cmath>
#include <sstream>
using namespace std;

char UP = 'U';
char DOWN = 'D';
char LEFT = 'L';
char RIGHT = 'R';
char NOACTION = 'N';
bool DEBUG = false;

class Board {
	// overloading output operator
	friend ostream& operator<<(ostream& os, const Board& rhs) {
		// prints out board in ixj(4x4) formatting
		for (size_t i = 0; i < rhs.data.size(); i++) {
			for (size_t j = 0; j < rhs.data.size(); j++) {
				os << rhs.data[i][j] << " ";
			}
			os << endl;
		}
		return os;
	}

	// overloading output file operator
	friend ofstream& operator<<(ofstream& os, const Board& rhs) {
		// prints out board in ixj(4x4) formatting
		for (size_t i = 0; i < rhs.data.size(); i++) {
			for (size_t j = 0; j < rhs.data.size(); j++) {
				os << rhs.data[i][j] << " ";
			}
			os << endl;
		}
		return os;
	}
public:
	Board(const vector<vector<int>>& data) : data(data) {}
```

```
// check if another board is same as this board
bool isSameBoard(const Board& other) const {
	// goes through whole board to check each tile
	for (size_t i = 0; i < other.data.size(); i++) {
		for (size_t j = 0; j < other.data.size(); j++) {
			if (data[i][j] != other.data[i][j]) {
				return false;
			}
		}
	}
	return true;
}

/*
 returns position of target number in a board
 pair.first = row number
 pair.second = column number
*/
pair<size_t, size_t> getPosition(int target) const {
	pair<size_t, size_t> result(data.size(), data.size());
	for (size_t i = 0; i < data.size(); i++) {
		for (size_t j = 0; j < data.size(); j++) {
			// check if match target
			if (data[i][j] == target) {
				// record position and return
				result.first = i;
				result.second = j;
				return result;
			}
		}
	}
	return result;
}

// calculates and return Manhattan distance
int getManhattanDistance(const Board& goal) const {
	int result = 0;
	for (int i = 0; i < 4; i++) {
		for (int j = 0; j < 4; j++) {
			// ignores empty tile
			if (data[i][j] != 0) {
				// finds position of num in goal state
				// converts size_t to int because subtracting
				pair<int, int> goalPos = goal.getPosition(data[i][j]);
				// adds Manhttan distance of curr tile to total Manhattan distance
				result += abs(goalPos.first - i);
				result += abs(goalPos.second - j);
			}
```

```
                }
        }
        return result;
}
// returns a vector of possible moves
// 'U' - blank space able to move up
// 'D' - blank sapce able to move down
// 'L' - blank space able to move left
// 'R' - blank spce able to move right
vector<char> getActions() const {
        vector<char> result;
        pair<size_t, size_t> zeroPos = getPosition(0);
        // based on position of zero/blank, adds possible action to our list
        if (zeroPos.first > 0) {
                result.push_back(UP);
        }
        if (zeroPos.first < data.size() - 1) {
                result.push_back(DOWN);
        }
        if (zeroPos.second > 0) {
                result.push_back(LEFT);
        }
        if (zeroPos.second < data[0].size() - 1) {
                result.push_back(RIGHT);
        }
        // for debugging, prints out the list
        if (DEBUG) {
                for (char action : result) {
                        cout << action << ", ";
                }
                cout << endl;
        }

        return result;
}
// returns a new Board after an action has been applied
Board doAction(char action) const {
        // checks if we can actually do this action with this board
        vector<char> actions = getActions();
        bool valid = false;
        for (char act : actions) {
                if (act == action) {
                        valid = true;
                        break;
                }
        }
        if (!valid) {
                cerr << "action cannot be performed" << endl;
```

```
                return *this;
            }
            // makes a copy of current board
            Board result = *this;
            // get position of blank
            pair<size_t, size_t> zeroPos = getPosition(0);

            if (action == UP) {
                result.data[zeroPos.first][zeroPos.second] = result.data[zeroPos.first - 1][zeroPos.second];
                result.data[zeroPos.first - 1][zeroPos.second] = 0;
            }
            else if (action == DOWN) {
                result.data[zeroPos.first][zeroPos.second] = result.data[zeroPos.first + 1][zeroPos.second];
                result.data[zeroPos.first + 1][zeroPos.second] = 0;
            }
            else if (action == LEFT) {
                result.data[zeroPos.first][zeroPos.second] = result.data[zeroPos.first][zeroPos.second - 1];
                result.data[zeroPos.first][zeroPos.second - 1] = 0;
            }
            else if (action == RIGHT) {
                result.data[zeroPos.first][zeroPos.second] = result.data[zeroPos.first][zeroPos.second + 1];
                result.data[zeroPos.first][zeroPos.second + 1] = 0;
            }
            else { cerr << "invalid action" << endl; }
            return result;
        }

private:
        vector<vector<int>> data;
};

// data structure to hold states for graph search
class TreeNode {

public:
        TreeNode(const TreeNode* parent, const Board& data, const Board& goal, char action)
            :parent(parent), board(data), goal(goal), lastAction(action){

            // sets up path cost, which is just the depth
            if (parent) {
                g = parent->g + 1;
            }
            else { // root node, g = 0
                g = 0;
            }
        }

        // returns path cost
```

```
        int getG() const { return g; }
        // returns estmiated total cost
        int getF() const {
                return board.getManhattanDistance(goal) + g;
        }
        // returns the action taken to get to board this node holds
        char getLastAction() const { return lastAction; }
        // return data
        Board getBoard() const {
                return board;
        }
        // returns a list of action taken to get to board in this node from initial board
        // NOTE: resulting list is in reserved order
        // if curr node = root node, path only has one node, aka the root node
        vector<char> getPathAction() const {
                vector<char> result;
                const TreeNode* nodeP = this;

                while (nodeP) {
                        if (nodeP->lastAction != NOACTION) {
                                result.push_back(nodeP->lastAction);
                        }
                        nodeP = nodeP->parent;
                }
                return result;
        }
        // returns a path from root node to current node, starting at root node
        // NOTE: the resulting path would be in reserved order
        vector<const TreeNode*> getPath() const {
                vector<const TreeNode*> result;
                const TreeNode* nodeP = this;
                while (nodeP) {
                        result.push_back(nodeP);
                        nodeP = nodeP->parent;
                }
                return result;
        }

private:
        const TreeNode* parent; // parent node
        Board board; // board that this node holds
        Board goal; // goal state
        char lastAction; // represents the action taken to get to curr node, empty char for root node
        int g; // path cost up to this node
};

// compare functor for prioroity queue
// a smaller f value means you're on top of the queue
```

```
class cmpFunction {
public:
	int operator()(TreeNode* lhs, TreeNode* rhs) const {
		return lhs->getF() > rhs->getF();
	}
};

// reads in input file and translate text to initial and goal boards
void readInput(ifstream& ifs, vector<vector<int>>& initial, vector<vector<int>>& goal);

// runs the A* search and produce an output file
void run(const Board& initial, const Board& goal);

int main() {
	// vectors to hold data
	vector<vector<int>> initialData(4, vector<int>(4, 0));
	vector<vector<int>> goalData(4, vector<int>(4, 0));
	// opens file
	ifstream input;
	cout << "Enter input file name(include the .txt):" << endl;
	string filename;
	cin >> filename;
	input.open(filename);
	// check if file is valid
	if (!input) {
		cout << "failed to open file" << endl;
		exit(1);
	}
	// fill vector from input
	readInput(input, initialData, goalData);
	// create Board object with data
	Board initialBoard(initialData);
	Board goalBoard(goalData);

	// debug print outs
	if (DEBUG) {
		cout << "----initial board----" << endl;
		cout << initialBoard << endl;
		cout << "----goal board----" << endl;
		cout << goalBoard << endl;
		cout << "Manhttan Distance: " << initialBoard.getManhattanDistance(goalBoard) << endl;

	}
	// testing getAction()
	if (DEBUG) {
		cout << "Possible Action of initial Board: " << endl;
		initialBoard.getActions();
		cout << "Possible Action of goal Board: " << endl;
```

```
		goalBoard.getActions();
	}
	// testing doAction()
	if (DEBUG) {
		cout << "move initial baord up" << endl;
		Board result = initialBoard.doAction(UP);
		cout << result << endl;
		cout << "move goal board right" << endl;
		result = goalBoard.doAction(RIGHT);
		cout << result << endl;
	}

	// run the search
	run(initialBoard, goalBoard);
	// close filestream when finished
	input.close();

}

void readInput(ifstream& ifs, vector<vector<int>>& initial, vector<vector<int>>& goal) {
	string row;
	int num;
	// fills up initial board
	for (int i = 0; i < 4; i++) {
		getline(ifs, row);
		istringstream ss(row);
		size_t j = 0;
		while (ss >> num) {
			initial[i][j] = num;
			j++;
		}
	}
	// skips empty line
	getline(ifs, row);
	// fills up goal board
	for (int i = 0; i < 4; i++) {
		getline(ifs, row);
		istringstream ss(row);
		size_t j = 0;
		while (ss >> num) {
			goal[i][j] = num;
			j++;
		}
	}
}

void run(const Board& initial, const Board& goal) {
	// create output file stream
```

```
	// note that if same name of file exist, will overwrite
	cout << "Enter name of outfile to be generated(include the .txt): " << endl;
	string outputFileName;
	cin >> outputFileName;
	ofstream output(outputFileName);
	// keep track of explored Boards(states)
	vector<Board> explored = vector<Board>();
	int numNodes = 1;		// total num of nodes generated, initialized to 1 bc root node
	vector<char> solutionAction; // keeps the sequence of action
	vector<const TreeNode*> solutionPath; // keeps the path
	int d = 0; // level of the shallowest goal node
	priority_queue<TreeNode*, vector<TreeNode*>, cmpFunction> queue;
	vector<TreeNode*> cleanUp;
	queue.push(new TreeNode(nullptr, initial, goal, NOACTION));
	// runs the search
	while (!queue.empty()) {
		Board currBoard = queue.top()->getBoard();
		if (currBoard.isSameBoard(goal)) { // found the goal node
			// assign d value, which is just the depth of the Node
			d = queue.top()->getG();
			// trace back on solution action
			solutionAction = queue.top()->getPathAction();
			// trace back on nodes
			solutionPath = queue.top()->getPath();
			break; // get outta this while loop
		}
		// adds current state to the explored vector
		explored.push_back(currBoard);
		if (DEBUG) {
			cout << "Current Board:" << endl;
			cout << currBoard << endl;
			cout << "Expanding..." << endl;
		}
		// calculates which actions can be done
		vector<char> actions = currBoard.getActions();
		// each doable action is a child(if not repeat)
		for (char action : actions) {
			// generates a new board after we take the action
			Board childBoard = currBoard.doAction(action);
			if (DEBUG) {
				cout << "Action Taken: " << action << "; Resulting Board: " << endl;
				cout << childBoard;
				cout << "Manhattan Distance: " << childBoard.getManhattanDistance(goal) << endl <<
endl;
			}
			// checks if this Board is a repeat state
			bool hasExplored = false;
			for (const Board& board : explored) {
```

```
                        if (childBoard.isSameBoard(board)) {
                                hasExplored = true;
                                break;
                        }
                }
                // not a repeat state, safe to add to our queue
                if (!hasExplored) {
                        // create new child Node
                        TreeNode* child = new TreeNode(queue.top(), childBoard, goal, action);
                        numNodes++; // increment node count
                        queue.push(child); // adds new node to queue
                }
        }
        // done expanding this current node, can remove from queue now
        // add to cleanUp list to free up memory later
        cleanUp.push_back(queue.top());
        queue.pop();
}

// write to output file
output << initial << endl;
output << goal << endl;
output <<  d << endl;
output <<  numNodes << endl;
// write out the actions
for (size_t i = solutionAction.size(); i >= 1; i--) {
        output << solutionAction[i - 1] << " ";
}
output << endl;
// write out the f values
for (size_t i = solutionPath.size(); i >= 1; i--) {
        output << solutionPath[i - 1]->getF() << " ";
}

// closing an output stream creates the file
output.close();
// free up memory
while (!queue.empty()) {
        delete queue.top();
        queue.pop();
}
for (TreeNode* tn : cleanUp) {
        delete tn;
}
cleanUp.clear();
}
```

1 2 3 4
5 6 0 7
8 9 10 11
12 13 14 15

1 2 3 4
5 9 6 7
8 13 0 11
12 14 10 15

d: 5
N: 15
Actions: L D D R U
F values: 5 5 5 5 5 5

1 5 3 13
8 0 6 4
15 10 7 9
11 14 2 12

1 5 3 13
8 10 6 4
0 15 2 9
11 7 14 12

d: 6
N: 20
Actions: D R D L U L
F values: 6 6 6 6 6 6 6

9 13 7 4
12 3 0 1
2 15 5 6
14 10 11 8

13 3 7 4
9 1 0 6
12 2 5 8
14 15 10 11

d: 12
N: 27
Actions: R D D L L U L U U R D R
F values: 12 12 12 12 12 12 12 12 12 12 12 12 12

13 12 2 11
10 1 8 9
0 3 15 14
6 4 7 5

10 13 12 11
8 1 2 9
3 4 15 5
6 0 14 7

d: 16
N: 172
Actions: R U R U L L D R D R R D L U L D
F values: 12 12 14 14 14 14 14 14 14 14 16 16 16 16 16 16 16