CSC11F: Advanced Data Structures and Algorithms

Exercise 6 – Report

NGUYEN Van Tinh - s1272003

1. Problem description

The goal of the 15-puzzle problem is to complete pieces on 4×4 cells where one of the cells is empty space. In this problem, the space is represented by 0, and pieces are represented by integers from 1 to 15, as shown below.

1 2 3 4

6 7 8 0

5 10 11 12

9 13 14 15

You can move a piece toward the empty space in one step. Your goal is to make the pieces the following configuration in the shortest move (fewest steps).

1 2 3 4

5 6 7 8

9 10 11 12

13 14 15 0

Write a program that reads the initial state of the puzzle and prints the fewest steps to solve the puzzle.

a. Input

The 4×4 integers denoting the pieces or space are given.

b. Output

Print the fewest steps in a line.

c. Constraints

- The given puzzle is solvable in at most 45 steps.

2. A* Algorithm

Firstly, the A* algorithm with the Manhattan distance heuristic is adopted to solve the 15-puzzle problem.

The program defines two structs, 'Puzzle' and 'State', to represent puzzle configurations and states in the search process. The 'Puzzle' struct contains an array to store the puzzle configuration, the index of the blank tile, the Manhattan distance heuristic value, and the cost to reach that configuration from the initial state. The 'State' struct keeps track of a puzzle configuration and its estimated cost.

The program uses a priority queue to manage the states during the A* search. It also uses a map to keep track of visited puzzle configurations, avoiding revisiting already explored states.

The main functions are as follows:

a. ***initMahattanDistance()***: This function calculates and stores the Manhattan distance between all pairs of tiles on the puzzle board, which is a pre-computation step to optimize the heuristic calculations.
b. ***getMahattanDistance(Puzzle p):*** This function calculates the Manhattan distance heuristic value for a given puzzle configuration. It sums the Manhattan distances of each tile to its correct position on the puzzle board.
c. ***astar(Puzzle &current_puzzle):*** This is the main A* search function. It takes an initial puzzle configuration as input and searches for the optimal solution. It uses the Manhattan distance heuristic to estimate the cost to reach the goal state. The A* algorithm iteratively explores states with the lowest estimated cost until the goal state is reached (the Mahattan distance is 0) or the priority queue becomes empty.

3. Iterative deepening A* Algorithm

Next, I use the iterative deepening A* (IDA*) algorithm with the Manhattan distance heuristic to solve the problem.

For this program, the function ***initMahattanDistance()*** and ***getMahattanDistance(Puzzle p)*** remains the same, while two new functions

depthLimitedSearch() and ***iterativeDeepening()*** are introduced to apply the IDA* algorithm with the heuristic search. They can be briefly described as follows:

a. depthLimitedSearch(int depth, int prev)

- This function implements the depth-limited search with A* heuristic using the IDA* algorithm.
- It takes two parameters: depth (the current depth of the search) and prev (the previous direction of the blank tile movement).
- The function first checks if the current state is the goal state by checking if the Manhattan distance heuristic value md is zero.
- It then adds the heuristic value to the current depth and compares it to the depthLimit. If the sum exceeds the depthLimit, the function returns false, cutting the branch.
- The function proceeds to try moving the blank tile in all four directions (up, down, left, right) while avoiding moves that undo the previous move (i.e., moves in the opposite direction).
- For each move, it updates the md heuristic value based on the Manhattan distance changes and recursively explores the next state.
- If a solution is found during the exploration, the function stores the direction of the move in the solutionPath array and returns true.

b. iterativeDeepening(Puzzle initialPuzzle)

- This function performs the iterative deepening process to find the solution path using the IDA* algorithm.
- It takes the initial Puzzle as input and starts with a depth limit set to the Manhattan distance heuristic value of the initial puzzle configuration.
- It then incrementally increases the depth limit up to a predefined STEP_LIMIT.
- For each depth limit, it sets the currentPuzzle to the initial puzzle and calls the depthLimitedSearch() function to explore the search space within the depth limit.
- If a solution is found, it constructs the solution string (sequence of moves) based on the directions stored in the solutionPath array.

4. Source code

- The source code for the aforementioned programs is as follows. You can also find it in the two ".cpp" files submitted along with this report. Detailed comments are also provided in the code.

a. A* algorithm

```cpp
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
#include <map>
using namespace std;

#define BOARD_SIZE 4
#define BOARD_AREA 16
#define BLANK_TILE 0
#define STEP_LIMIT 100

static const int dx[4] = {0, -1, 0, 1};
static const int dy[4] = {1, 0, -1, 0};
static const char direction[4] = {'u', 'l', 'd', 'r'};

struct Puzzle
{
    int board[BOARD_AREA]; // The puzzle board configuration, represented as an
array of integers.
    int blank_tile;        // The index of the blank tile in the puzzle.
    int md;                // The Manhattan distance heuristic value for this
puzzle configuration.
    int cost;              // The cost (number of steps) to reach this
configuration from the initial state.

    // Operator overload for comparing two Puzzle structs.
    // It is used for keeping track of visited states in the search process.
    bool operator<(const Puzzle &p) const
    {
        for (int i = 0; i < BOARD_AREA; i++)
        {
            if (board[i] == p.board[i])
                continue;
            return board[i] < p.board[i];
        }
        return false;
    }
};
```

```
struct State
{
    Puzzle puzzle; // The current puzzle configuration.
    int estimated; // The estimated cost (heuristic + actual cost) to reach the
goal from this state.

    // Operator overload for comparing two State structs.
    // It is used for maintaining the priority queue during the A* search.
    bool operator<(const State &s) const
    {
        return estimated > s.estimated;
    }
};

int manhattan_distance[BOARD_AREA][BOARD_AREA]; // 2D array to store Manhattan
distances between tiles.
priority_queue<State> pqueue;                   // Priority queue to manage
states during the A* search.
map<Puzzle, bool> puzzle_map;                   // Map to keep track of visited
puzzle configurations.

void initMahattantDistance()
{
    // Calculate and store the Manhattan distance between all pairs of tiles
    for (int i = 0; i < BOARD_AREA; i++)
    {
        for (int j = 0; j < BOARD_AREA; j++)
        {
            manhattan_distance[i][j] = abs(i / BOARD_SIZE - j / BOARD_SIZE) +
abs(i % BOARD_SIZE - j % BOARD_SIZE);
        }
    }
}

int getMahattantDistance(Puzzle p)
{
    // Calculate the Manhattan distance heuristic value for the given puzzle
configuration.
    int cost = 0;
    for (int i = 0; i < BOARD_AREA; i++)
    {
        if (p.board[i] == BOARD_AREA)
            continue;
        cost += manhattan_distance[i][p.board[i] - 1];
    }
```

```
    return cost;
}

int astar(Puzzle &current_puzzle)
{
    current_puzzle.md = getMahattantDistance(current_puzzle); // Calculate the
initial heuristic value for the input puzzle.
    current_puzzle.cost = 0; // Initialize the cost to reach this state from the
initial state.

    State initial_state;
    initial_state.puzzle = current_puzzle;
    initial_state.estimated = current_puzzle.md; // The estimated cost is the
heuristic cost at the beginning.
    pqueue.push(initial_state); // Push the initial state into the priority
queue.

    while (!pqueue.empty())
    {
        State current_state = pqueue.top(); // Get the state with the lowest
estimated cost from the priority queue.
        pqueue.pop();

        if (current_state.puzzle.md == 0)
            return current_state.puzzle.cost; // If the heuristic is zero, the
goal state is reached, return the cost.

        int current_x, current_y, target_x, target_y;
        current_x = current_state.puzzle.blank_tile / BOARD_SIZE; // Get the x-
coordinate of the blank tile.
        current_y = current_state.puzzle.blank_tile % BOARD_SIZE; // Get the y-
coordinate of the blank tile.

        // Try moving the blank tile in all four directions (u, l, d, r)
        for (int dir = 0; dir < 4; dir++)
        {
            target_x = current_x + dx[dir]; // Calculate the new x-coordinate
after the move in the current direction.
            target_y = current_y + dy[dir]; // Calculate the new y-coordinate
after the move in the current direction.

            // Check if the new position is within the puzzle boundaries.
            if (target_x < 0 || target_x >= BOARD_SIZE || target_y < 0 ||
target_y >= BOARD_SIZE)
```

```
                continue;

            Puzzle new_puzzle = current_state.puzzle; // Create a copy of the
current puzzle configuration.

            // Update the Manhattan distance heuristic after the tile swap.
            new_puzzle.md -= manhattan_distance[target_x * BOARD_SIZE +
target_y][new_puzzle.board[target_x * BOARD_SIZE + target_y] - 1];
            new_puzzle.md += manhattan_distance[current_x * BOARD_SIZE +
current_y][new_puzzle.board[target_x * BOARD_SIZE + target_y] - 1];

            // Swap the tiles and update the position of the blank tile.
            swap(new_puzzle.board[target_x * BOARD_SIZE + target_y],
new_puzzle.board[current_x * BOARD_SIZE + current_y]);
            new_puzzle.blank_tile = target_x * BOARD_SIZE + target_y;

            // If the resulting puzzle configuration is not visited before,
proceed with it.
            if (!puzzle_map[new_puzzle])
            {
                puzzle_map[current_state.puzzle] = true; // Mark the current
puzzle configuration as visited.
                new_puzzle.cost++; // Increment the cost to reach this new
configuration.
                State new_state;
                new_state.puzzle = new_puzzle; // Create a new state with the
updated puzzle.
                new_state.estimated = new_puzzle.cost + new_puzzle.md; // Update
the estimated cost for the new state.
                pqueue.push(new_state); // Push the new state into the priority
queue.
            }
        }
    }

    return -1; // If the queue is empty and no solution is found, return -1.
}

int main()
{
    Puzzle initial_puzzle;

    // Read the initial puzzle configuration from the input.
    for (int i = 0; i < BOARD_AREA; i++)
    {
```

```
        cin >> initial_puzzle.board[i];
        if (initial_puzzle.board[i] == BLANK_TILE)
        {
            initial_puzzle.board[i] = BOARD_AREA;
            initial_puzzle.blank_tile = i;
        }
    }

    initMahattantDistance();
    cout << astar(initial_puzzle) << endl;

    return 0;
}
```

b. IDA* algorithm

```
#include <iostream>
#include <algorithm>
#include <cstring>
using namespace std;

#define BOARD_SIZE 4
#define BOARD_AREA 16
#define BLANK_TILE 0
#define STEP_LIMIT 100

static const int dx[4] = {0, -1, 0, 1};
static const int dy[4] = {1, 0, -1, 0};
static const char direction[4] = {'u', 'l', 'd', 'r'};

struct Puzzle
{
    int board[BOARD_AREA];
    int blank_tile;
    int md; // Manhattan distance heuristic
};

int manhattan_distance[BOARD_AREA][BOARD_AREA];
Puzzle currentPuzzle;
int depthLimit;
int solutionPath[STEP_LIMIT];

void initmanhattan_distance()
```

```
{
    for (int i = 0; i < BOARD_AREA; i++)
    {
        // Calculate and store the Manhattan distance between all pairs of tiles.
        for (int j = 0; j < BOARD_AREA; j++)
        {
            manhattan_distance[i][j] = abs(i / BOARD_SIZE - j / BOARD_SIZE) +
abs(i % BOARD_SIZE - j % BOARD_SIZE);
        }
    }
}

int getMahattantDistance(Puzzle p)
{
    // Calculate the Manhattan distance heuristic value for the given puzzle
configuration.
    int cost = 0;
    for (int i = 0; i < BOARD_AREA; i++)
    {
        if (p.board[i] == BOARD_AREA)
            continue;
        cost += manhattan_distance[i][p.board[i] - 1];
    }

    return cost;
}

// Depth-limited search with A* heuristic using IDA* algorithm
bool depthLimitedSearch(int depth, int prev)
{
    // If the current state is the goal state, return true
    if (currentPuzzle.md == 0)
        return true;

    // Add a heuristic to the current depth and cut a branch if the limit is
exceeded
    if (depth + currentPuzzle.md > depthLimit)
        return false;

    int current_x, current_y, target_x, target_y;
    Puzzle tempPuzzle;

    current_x = currentPuzzle.blank_tile / BOARD_SIZE;
    current_y = currentPuzzle.blank_tile % BOARD_SIZE;
```

```
    // Try moving the blank tile in all four directions (u, l, d, r)
    for (int dir = 0; dir < 4; dir++)
    {
        target_x = current_x + dx[dir];
        target_y = current_y + dy[dir];
        // Check if the new position is within the puzzle boundaries
        if (target_x < 0 || target_x >= BOARD_SIZE || target_y < 0 || target_y >=
BOARD_SIZE)
            continue;
        // Skip moves that undo the previous move (opposite direction)
        if (max(prev, dir) - min(prev, dir) == 2)
            continue;

        tempPuzzle = currentPuzzle;

        // Update the Manhattan distance heuristic after the tile swap.
        currentPuzzle.md -= manhattan_distance[target_x * BOARD_SIZE +
target_y][currentPuzzle.board[target_x * BOARD_SIZE + target_y] - 1];
        currentPuzzle.md += manhattan_distance[current_x * BOARD_SIZE +
current_y][currentPuzzle.board[target_x * BOARD_SIZE + target_y] - 1];

        // Swap the tiles and update the position of the blank tile.
        swap(currentPuzzle.board[target_x * BOARD_SIZE + target_y],
currentPuzzle.board[current_x * BOARD_SIZE + current_y]);
        currentPuzzle.blank_tile = target_x * BOARD_SIZE + target_y;

        // Recursively explore the next state
        if (depthLimitedSearch(depth + 1, dir))
        {
            // If a solution is found, store the direction and return true
            solutionPath[depth] = dir;
            return true;
        }
        // If the current move did not lead to a solution, backtrack
        currentPuzzle = tempPuzzle;
    }
    // If no solution is found for the current depth, return false
    return false;
}

// Main function to perform iterative deepening
string iterativeDeepening(Puzzle initialPuzzle)
{
    // Calculate the Manhattan distance for the initial puzzle state
    initialPuzzle.md = getMahattantDistance(initialPuzzle);
```

```
    // Start the depth limit from the Manhattan distance and increment it
    // until the STEP_LIMIT is reached (maximum depth allowed)
    for (depthLimit = initialPuzzle.md; depthLimit < STEP_LIMIT; depthLimit++)
    {
        currentPuzzle = initialPuzzle;

        // Perform depth-limited DFS using IDA*
        if (depthLimitedSearch(0, -100))
        {
            string result = "";
            // If a solution is found, construct the solution string
            for (int i = 0; i < depthLimit; i++)
            {
                result += direction[solutionPath[i]];
            }
            return result;
        }
    }
    // If no solution is found within the depth limit, return "no answer"
    return "no answer";
}

int main()
{
    Puzzle initial_puzzle;

    // Read the initial puzzle configuration from the input.
    for (int i = 0; i < BOARD_AREA; i++)
    {
        cin >> initial_puzzle.board[i];
        if (initial_puzzle.board[i] == BLANK_TILE)
        {
            initial_puzzle.board[i] = BOARD_AREA;
            initial_puzzle.blank_tile = i;
        }
    }

    initmanhattan_distance();
    string result = iterativeDeepening(initial_puzzle);
    cout << result.size() << endl;
    return 0;
}
```