CFC11F: Advanced Data Structures and Algorithms
Heuristic Search for 15 Puzzle Problem Report

Pham Thi Huong Giang (m5252116@u-aizu.ac.jp)

1 Problem Description

The 15-puzzle problem is played on a 4x4 board with 15 numbered cells (tiles), and one of the cells is empty space. The problem goal is to rearrange the tiles in the shortest path (i.e., the fewest steps) from the initial state of the puzzle to a particular goal configuration by sliding one piece at each step into the blank space.

We use the heuristic search to find the solution, where several heuristic techniques are applied to bound and prune several possible paths to increase the search efficiency. This report presents two heuristic search algorithms, Iterative Deepening A* (IDA*) and A* algorithms, to solve the 15-puzzle problem.

IDA* algorithm is a heuristic algorithm, combining the depth-first search DFS with depth-limit search (iterative deepening) and a pruning technique based on estimate values (heuristic). The heuristic value h is estimated to give a lower limit to reach the goal from the current state. With g is the current depth, if g + h exceeds the limit d (of depth-limit search), we can prune the search on the current path. We continue increasing the limit d and iteratively search until the solution is found.

A* algorithm is also a heuristic algorithm, where the estimate values (heuristic) are applied with a breath-first search (BFS) algorithm. The puzzle states with its heuristic value are managed by a priority queue to find the shortest path. The algorithm selects the top state of the priority queue with the smallest heuristic value to speed up the BFS. A map is maintained to keep track of the visited states.

Both algorithms calculate the heuristic value based on the Manhattan distance. The codes with comments are provided to explain the algorithms.

2 Source Code

2.1 *Iterative Deepening A* Algorithm (IDA*)*

```
#include <iostream>

#include <algorithm>

#include <cstring>

using namespace std;
```

```
#define N 4
#define TILE_LEN 16
#define BLANK_TILE 0
#define STEP_LIMIT 45 // the given puzzle is solvable in at most 45 steps, depth_limit < 45

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 f[TILE_LEN]; // puzzle configuration
    int blank_tile; // index of the blank tile
    int md; // Manhattan distance for the current puzzle state
};

int manhattan_distance[TILE_LEN][TILE_LEN]; // precomputed Manhattan distances for all tile pairs
Puzzle puz; // current puzzle state
int depth_limit; // current depth limit
int path[STEP_LIMIT]; // path of movements leads to the solution

void initMahattantDistance();
int getMahattantDistance(Puzzle);
bool dfs(int, int);
string iterative_deepening(Puzzle);

int main()
{
    Puzzle in;

    for (int i = 0; i < TILE_LEN; i++)
    {
        cin >> in.f[i];
        if (in.f[i] == BLANK_TILE)
        {
            in.f[i] = TILE_LEN;
            in.blank_tile = i;
        }
    }

    // precompute the Manhattan distances
    initMahattantDistance();

    string rs = iterative_deepening(in);

    // rs.size(): number of steps required to reach the solution
    cout << rs.size() << endl;
    return 0;
}

// precompute the Manhattan distances
void initMahattantDistance()
{
    for (int i = 0; i < TILE_LEN; i++)
    {
        for (int j = 0; j < TILE_LEN; j++)
        {
            // calculate Manhattan distance between tile (i/N, i%N) and (j/N, j%N)
            manhattan_distance[i][j] = abs(i / N - j / N) + abs(i % N - j % N);
        }
    }
}

// calculate the Manhattan distance for a given puzzle state
int getMahattantDistance(Puzzle p)
{
    int cost = 0;
    for (int i = 0; i < TILE_LEN; i++)
    {
        // cost of blank tile = 0
        if (p.f[i] == TILE_LEN)
            continue;
        // i: desired tile value , p.f[i] - 1: current title value
        cost += manhattan_distance[i][p.f[i] - 1];
    }

    return cost;
}

// DFS with depth-limited search and the Manhattan distance heuristic
bool dfs(int depth, int prev)
{
    if (puz.md == 0)
        return true; // puzzle solved

    // ddd a heuristic to the current depth and cut a branch if the limit is exceeded
    if (depth + puz.md > depth_limit)
        return false;

    int sx, sy, tx, ty;
    Puzzle tmp;

    // current blank_tile location
    sx = puz.blank_tile / N;
    sy = puz.blank_tile % N;

    for (int dir = 0; dir < 4; dir++)
    {
        // new blank_tile location
        tx = sx + dx[dir];
        ty = sy + dy[dir];

        // prune some paths
        // skip the out of bounds direction
        if (tx < 0 || tx >= N || ty < 0 || ty >= N)
            continue;
        // prevent moving back to the previous state
        if (max(prev, dir) - min(prev, dir) == 2)
            continue;

        tmp = puz; // store the current state for backtracking

        // update the mahantan distance
        // tx * N + ty: new blank_tile location, cost = 0
        // f[tx * N + ty]: current value at the new blank_tile location, minus this cost (to 0)
        // sx * N + sy: current blank_tile location, update cost with the new tile value f[tx * N + ty]
        puz.md -= manhattan_distance[tx * N + ty][puz.f[tx * N + ty] - 1];
        puz.md += manhattan_distance[sx * N + sy][puz.f[tx * N + ty] - 1];

        // change puzzle configure
        swap(puz.f[tx * N + ty], puz.f[sx * N + sy]);
        // update new location of the blank_tile
        puz.blank_tile = tx * N + ty;

        // recursively explore the next state
        // dfs return true, puzzle solved
        if (dfs(depth + 1, dir))
        {
            // store the movement direction
            path[depth] = dir;
            return true;
        }

        // puzzle not solved, backtrack to the previous state
        puz = tmp;
    }

    // puzzle not solved at this depth, return false
    return false;
```

```
}

// Iterative deepening algorithm to find the solution.
string iterative_deepening(Puzzle in)
{
    // calculate the md of the initial state
    in.md = getMahattantDistance(in);

    for (depth_limit = in.md; depth_limit < STEP_LIMIT; depth_limit++)
    {
        // initialize the puzzle state
        puz = in;

        // current depth: 0, previous direction: -100 (arbitrary number)
        if (dfs(0, -100))
        {
            // dfs return true, solution is found
            string rs = "";
            for (int i = 0; i < depth_limit; i++)
            {
                // construct the path, convert movement directions to characters
                rs += direction[path[i]];
            }
            return rs;
        }
    }
    // no solution found within the step limit
    return "no answer";
}
```

2.2 *A* Search Algorithm*

```
#include <iostream>
#include <algorithm>
#include <cstring>
#include <queue>
```

```
#include <map>
using namespace std;

// Constants and Definitions
#define N 4
#define TILE_LEN 16
#define BLANK_TILE 0

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 f[TILE_LEN]; // puzzle configuration
    int blank_tile; // index of the blank tile
    int md; // Manhattan distance for the current puzzle state
    int cost; // number of moves to reach this state from the initial state
    bool operator<(const Puzzle &p) const
    {
        for (int i = 0; i < TILE_LEN; i++)
        {
            if (f[i] == p.f[i])
                continue;
            return f[i] < p.f[i];
        }
        return false;
    }
};

struct State
{
    Puzzle puzzle; // current puzzle state
    int estimated; // estimated cost from the current state to the goal state
    bool operator<(const State &s) const
    {
        return estimated > s.estimated;
    }
};

int manhattan_distance[TILE_LEN][TILE_LEN]; // precomputed Manhattan distances for all tile pairs
priority_queue<State> pqueue; // priority queue for A* search.
map<Puzzle, bool> puzzle_map; // map to keep track of visited puzzle states.

void initMahattantDistance();
int getMahattantDistance(Puzzle);
int astar(Puzzle &);

int main()
{
    Puzzle in;

    for (int i = 0; i < TILE_LEN; i++)
    {
        cin >> in.f[i];
        if (in.f[i] == BLANK_TILE)
        {
            in.f[i] = TILE_LEN;
            in.blank_tile = i;
        }
    }

    // precompute the Manhattan distances
    initMahattantDistance();

    cout << astar(in) << endl;

    return 0;
}
```

```
// precompute the Manhattan distances
void initMahattantDistance()
{
    for (int i = 0; i < TILE_LEN; i++)
    {
        for (int j = 0; j < TILE_LEN; j++)
        {
            // calculate Manhattan distance between tile (i/N, i%N) and (j/N, j%N)
            manhattan_distance[i][j] = abs(i / N - j / N) + abs(i % N - j % N);
        }
    }
}

// calculate the Manhattan distance for a given puzzle state
int getMahattantDistance(Puzzle p)
{
    int cost = 0;
    for (int i = 0; i < TILE_LEN; i++)
    {
        // cost of blank tile = 0
        if (p.f[i] == TILE_LEN)
            continue;
        // i: desired tile value , p.f[i] - 1: current title value
        cost += manhattan_distance[i][p.f[i] - 1];
    }

    return cost;
}

// A* search algorithm
int astar(Puzzle &p_in)
{
    // calculate the md of the initial state
    p_in.md = getMahattantDistance(p_in);
    p_in.cost = 0;
```

```
State s_in;
s_in.puzzle = p_in;
s_in.estimated = p_in.md;
pqueue.push(s_in);

while (!pqueue.empty())
{
    State s = pqueue.top();
    pqueue.pop();

    //if puzzle solved, return the number of moves
    if (s.puzzle.md == 0)
        return s.puzzle.cost;

    int sx, sy, tx, ty;

    // current blank_tile location
    sx = s.puzzle.blank_tile / N;
    sy = s.puzzle.blank_tile % N;

    for (int dir = 0; dir < 4; dir++)
    {
        // new blank_tile location
        tx = sx + dx[dir];
        ty = sy + dy[dir];

        // prune some paths
        // skip the out of bounds direction
        if (tx < 0 || tx >= N || ty < 0 || ty >= N)
            continue;

        Puzzle puz = s.puzzle;

        // update the mahantan distance
```

```
            // tx * N + ty: new blank_tile location, cost = 0
            // f[tx * N + ty]: current value at the new blank_tile location, minus this cost (to 0)
            // sx * N + sy: current blank_tile location, update cost with the new tile value f[tx * N + ty]
            puz.md -= manhattan_distance[tx * N + ty][puz.f[tx * N + ty] - 1];
            puz.md += manhattan_distance[sx * N + sy][puz.f[tx * N + ty] - 1];
            swap(puz.f[tx * N + ty], puz.f[sx * N + sy]);
            puz.blank_tile = tx * N + ty;

            if (!puzzle_map[puz])
            {
                puzzle_map[s.puzzle] = true; // mark the visited state in map
                puz.cost++;
                State s_new;
                s_new.puzzle = puz;
                s_new.estimated = puz.cost + puz.md;
                pqueue.push(s_new); // add the new state to the priority queue
            }
        }
    }
    return -1; // no solution found
}
```