Programming Assignment #1 Report

0516007 高誌佑

1. Experiments and Results

Below are some settings in all experiments:

- Machine (CPU): AMD Ryzen 7-3700X 3.6GHz
- Operating System: Ubuntu 18.04
- Language: **C++**
- Compiler Version: gcc 7.5.0
- All 5 algorithms are implemented with **graph search** version
- Use the heuristic function given in the spec, and g(x) is **the length from the starting position**

a. Optimal Test

I test whether these five search algorithms can find the optimal path. With the board size 8x8, I compare the length of output paths from (0,0) to every other position for each searching algorithm to check the optimality. The result is shown below.

Algorithm	Optimal
BFS	Yes
DFS	No
IDS	Yes
A*	Yes
IDA*	Yes

b. Time and Expanded Nodes Comparison

This test compares the execution time and the number of expanded nodes for each searching algorithm with six different board size. For IDS and IDA*, the number of expanded nodes is the summation of that of every iteration. The following tables show the results.

- Board Size: 8 x 8, from (0, 0) to (7, 7)

Algorithm	Time	Expanded Nodes
BFS	78 us	63
DFS	106 us	63
IDS	124 us	157
A*	100 us	46
IDA*	103 us	92

- Board Size: 16 x 16, from (0, 0) to (15, 15)

Algorithm	Time	Expanded Nodes
BFS	166 us	253
DFS	350 us	206
IDS	503 us	963
A*	253 us	98
IDA*	132 us	79

- Board Size: 32 x 32, from (0, 0) to (31, 31)

Algorithm	Time	Expanded Nodes
BFS	485 us	1023
DFS	1256 us	1023
IDS	4395 us	10182
A*	1288 us	574
IDA*	617 us	822

- Board Size: 128 x 128, from (0, 0) to (127, 127)

Algorithm	Time	Expanded Nodes
BFS	7.409 ms	16383
DFS	26.312 ms	16383
IDS	149.929 ms	636318
A*	18.854 ms	8638
IDA*	10.737 ms	13270

- Board Size: 512 x 512, from (0, 0) to (511, 511)

Algorithm	Time	Expanded Nodes
BFS	0.006 s	262143
DFS	0.277 s	223632
IDS	9.145 s	40448998
A*	0.198 s	118947
IDA*	0.272 s	211542

- Board Size: 1024 x 1024 from (0, 0) to (1023, 1023)

Algorithm	Time	Expanded Nodes
BFS	0.226 s	1048573
DFS	1.140 s	1048574
IDS	74.584 s	323255520
A*	0.808 s	475588
IDA*	2.188 s	844721

2. Observations and Interpretations

a. Optimality of DFS

From the optimal test, we can see that DFS is the only non-optimal one. Since DFS traverses the graph until it finds a legal and returns, it is not guaranteed to find the optimal path.

If we want to use DFS the find the optimal path, we should modify the process of searching. Instead of stopping when a legal path is found, we need to traverse all the path in graph. We also need to record the distance from the starting position for every visited position. Once we visit the same node again, update the distance if needed (because there may be multiple paths passing the same node with different length). After all possible path are travered, we can still find the optimal path, but it will takes a lot of time. In my opinion, it is not recommended to find the optimal path by DFS.

b. Time and Expanded Nodes Analysis

Among BFS, DFS, and IDS, the number of expanded nodes in IDS is much more since it counts for all the iterations. As the board size grows, the time difference between IDS and the other two is also obvious. For BFS, we can discover that it expands almost all the nodes in the board. I guess it is because the distance of the two corner is usually the longest, and BFS needs to expand all the nodes to find the goal.

Among A* and IDA*, IDA* is faster in small board size but slower in large board size. I think the reason is that IDA* needs to search the repeated nodes for every iteration. As the board size grows, the repeated search takes more time. Besides, IDA* expands more nodes than A* since it counts for all the iterations. I think if I implement the tree search version of A* and IDA*, the execution time of these two algorithm will grow a lot.

For the large board size, A* and IDA* expand less nodes than the other three algorithms. It is apparently that the evaluation function helps. It guides the algorithm to expand the node with higher value. As long as we use a proper evaluation function, we can avoid unecessary expansion. In addition, we can find IDA* runs faster than IDS. We can also say that the evaluation function helps us to save time.

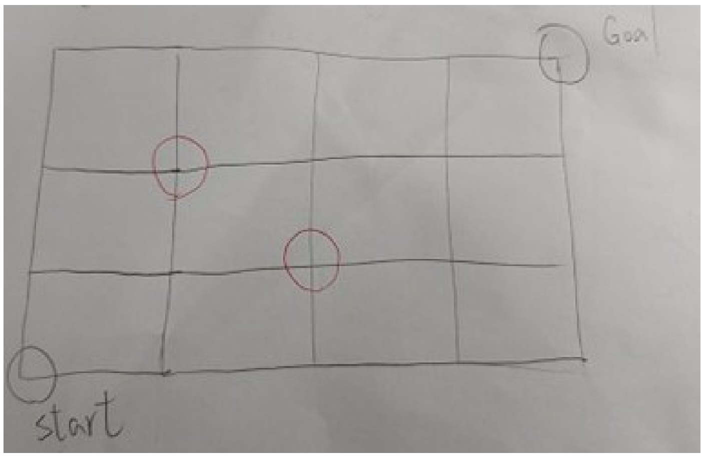
From the results list in the above tables, we can conclude that we should not use IDS with the large board size. And BFS is a time-efficient searching algorithm for the optimal path, but it expands more nodes. Besides, it is a relatively bad choice to use DFS to find the optimal path. Finally, an appropriate evalution function will enhance the efficiency of searching.

3. Things I have learned

Before this assignment, I have implemented only BFS and DFS before. Although I have heard of the other three algorithms, I do not know the implementations at all. I learn the details and the implementation during this assignment. Moreover, I try to analyze the different performance between each algorithm. It is a good practice to figure out the reasons that cause the difference, and I benefit a lot from the analysis. Hope that I can handle this kind of problem with ease in the future.

4. Remaining Questions and Future Investigations

Although I finish this assignment, I do not understand why the heuristic function given in the spec works. Look at the following picture, the starting position is at the left lower corner and the goal is at the right upper corner. The two red positions both have the same value according to the heuristic function. However, the optimal steps from two red positions to goal are different. One is two steps away and the other is four step away. It is clear that the two positions are not equivalent. In the future, I will try to think about this question and attemp to find a better heuristic function to further improve the performance.



Appendix: Source code

- Compile Command: g++ -g -Wall -o search main.cpp
- main.cpp

```
#include <bits/stdc++.h>
using namespace std;

struct Node {
    int x, y, f;
    Node() : x(-1), y(-1), f(INT_MAX) {}
    Node(int x, int y, int f = INT_MAX) : x(x), y(y), f(f) {}
};

// for the priority queue used in A*
struct cmp {
    bool operator() (Node a, Node b) { return a.f > b.f; }
};

const int BOARD_SIZE = 8;
const int MAX_LAYER = BOARD_SIZE * BOARD_SIZE;
bool explored[BOARD_SIZE][BOARD_SIZE];
Node parent[BOARD_SIZE][BOARD_SIZE];
int layer[BOARD_SIZE][BOARD_SIZE];

const int dx[8] = {-2, -1, 1, 2, -2, -1, 1, 2};
const int dy[8] = {1, 2, 2, 1, -1, -2, -2, -1};

// check whether the position is beyond the board
bool is_outside(int x, int y) {
    return x < 0 || x >= BOARD_SIZE || y < 0 || y >= BOARD_SIZE;
}

// use the recorded parent to backtrack the path and then output information
void output(const Node& leaf, int num_expand) {
    vector<string> points;
    Node node = leaf;

    points.clear();
    while (node.x != -1 && node.y != -1) {
        string str = "(" + to_string(node.x) + "," + to_string(node.y) + ")";
        points.push_back(str);
        node = parent[node.x][node.y];
    }
    reverse(points.begin(), points.end());

    cout << "path: ";
    for (const string& point : points) cout << point << " ";
    cout << "\n";
    cout << "Total " << points.size() - 1 << " steps\n";
    cout << "Total " << num_expand << " expanded nodes\n";
    // cout << points.size() - 1 << " ";
}

// heuristic function given in the spec
int heuristic(int sx, int sy, int gx, int gy) {
    int dx = sx - gx;
    int dy = sy - gy;
    return (abs(dx) + abs(dy)) / 3;
}

int BFS(const Node& start, const Node& goal) {
    printf("Using BFS, from (%d,%d) to (%d,%d)\n\n", start.x, start.y, goal.x, goal.y);

    int num_expand = 0;
    queue<Node> frontier;
    frontier.push(start);
    memset(explored, false, sizeof(explored));
    parent[start.x][start.y] = Node{-1, -1};

    while (!frontier.empty()) {
        Node node = frontier.front(); frontier.pop();
        if (explored[node.x][node.y]) continue;
        if (node.x == goal.x && node.y == goal.y) {
            output(node, num_expand);
            break;
        }

        explored[node.x][node.y] = true;
        num_expand++;
        for (int i = 0; i < 8; i++) {
            int x = node.x + dx[i];
            int y = node.y + dy[i];
            if (is_outside(x, y))   continue;
            if (explored[x][y])     continue;
            frontier.push(Node{x, y});
            parent[x][y] = node;
        }
    }
    return num_expand;
}

int DFS(const Node& start, const Node& goal) {
    printf("Using DFS, from (%d,%d) to (%d,%d)\n\n", start.x, start.y, goal.x, goal.y);
    int num_expand = 0;
    stack<Node> frontier;
    frontier.push(start);
    memset(explored, false, sizeof(explored));
    parent[start.x][start.y] = Node{-1, -1};

    while (!frontier.empty()) {
        Node node = frontier.top(); frontier.pop();
        if (explored[node.x][node.y]) continue;
        if (node.x == goal.x && node.y == goal.y) {
            output(node, num_expand);
            break;
        }

        explored[node.x][node.y] = true;
        num_expand++;
        for (int i = 0; i < 8; i++) {
            int x = node.x + dx[i];
            int y = node.y + dy[i];
            if (is_outside(x, y))   continue;
            if (explored[x][y])     continue;
            frontier.push(Node{x, y});
            parent[x][y] = node;
        }
    }
    return num_expand;
}

int IDS(const Node& start, const Node& goal) {
    printf("Using IDS, from (%d,%d) to (%d,%d)\n\n", start.x, start.y, goal.x, goal.y);

    int num_expand = 0;
    for (int l = 0; l < MAX_LAYER; l++) {
        stack<Node> frontier;
        frontier.push(start);
        memset(explored, false, sizeof(explored));
        parent[start.x][start.y] = Node{-1, -1};

        for (int i = 0; i < BOARD_SIZE; i++) for (int j = 0; j < BOARD_SIZE; j++)
            layer[i][j] = INT_MAX;
        layer[start.x][start.y] = 0;

        while (!frontier.empty()) {
            Node node = frontier.top(); frontier.pop();
            if (explored[node.x][node.y]) continue;
            if (node.x == goal.x && node.y == goal.y) {
                output(node, num_expand);
                return num_expand;
            }
            // cut node if we reach the layer limit
            if (layer[node.x][node.y] >= l)    continue;

            explored[node.x][node.y] = true;
            num_expand++;
            for (int i = 0; i < 8; i++) {
                int x = node.x + dx[i];
                int y = node.y + dy[i];
                if (is_outside(x, y))   continue;
                if (explored[x][y])     continue;
                frontier.push(Node{x, y});
                // to record the correct parent
                if (layer[node.x][node.y] + 1 < layer[x][y]) {
                    parent[x][y] = node;
                    layer[x][y] = layer[node.x][node.y] + 1;
                }
            }
        }
    }
    return num_expand;
}

int A_star(const Node& start, const Node& goal) {
    printf("Using A*, from (%d,%d) to (%d,%d)\n\n", start.x, start.y, goal.x, goal.y);

    int num_expand = 0;
    priority_queue<Node, vector<Node>, cmp > frontier;
    int h = heuristic(start.x, start.y, goal.x, goal.y);
    frontier.push(Node{start.x, start.y, h});
    memset(explored, false, sizeof(explored));
    parent[start.x][start.y] = Node{-1, -1};

    for (int i = 0; i < BOARD_SIZE; i++) for (int j = 0; j < BOARD_SIZE; j++)
        layer[i][j] = INT_MAX;
    layer[start.x][start.y] = 0;

    while (!frontier.empty()) {
        Node node = frontier.top(); frontier.pop();
        if (explored[node.x][node.y]) continue;
        if (node.x == goal.x && node.y == goal.y) {
            output(node, num_expand);
            break;
        }
        explored[node.x][node.y] = true;
        num_expand++;
        for (int i = 0; i < 8; i++) {
            int x = node.x + dx[i];
            int y = node.y + dy[i];
            if (is_outside(x, y))   continue;
            if (explored[x][y])     continue;

            // to record the correct parent
            if (layer[node.x][node.y] + 1 < layer[x][y]) {
                parent[x][y] = node;
                layer[x][y] = layer[node.x][node.y] + 1;
            }
            int h = heuristic(x, y, goal.x, goal.y);
            frontier.push(Node{x, y, h + layer[x][y]});
        }
    }
    return num_expand;
}

int IDA_star(const Node& start, const Node& goal) {
    printf("Using IDA*, from (%d,%d) to (%d,%d)\n\n", start.x, start.y, goal.x, goal.y);

    int num_expand = 0;
    for (int l = 0; l < MAX_LAYER; l++) {
        stack<Node> frontier;
        int h = heuristic(start.x, start.y, goal.x, goal.y);
        frontier.push(Node{start.x, start.y, h});
        memset(explored, false, sizeof(explored));
        parent[start.x][start.y] = Node{-1, -1};

        for (int i = 0; i < BOARD_SIZE; i++) for (int j = 0; j < BOARD_SIZE; j++)
            layer[i][j] = INT_MAX;
        layer[start.x][start.y] = 0;

        while (!frontier.empty()) {
            Node node = frontier.top(); frontier.pop();
            if (explored[node.x][node.y]) continue;

            if (node.x == goal.x && node.y == goal.y) {
                output(node, num_expand);
                return num_expand;
            }
            // cut node if we reach the layer(f) limit
            if (node.f >= l)    continue;
            explored[node.x][node.y] = true;
            num_expand++;
            for (int i = 0; i < 8; i++) {
                int x = node.x + dx[i];
                int y = node.y + dy[i];
                if (is_outside(x, y))   continue;
                if (explored[x][y])     continue;

                // to record the correct parent
                if (layer[node.x][node.y] + 1 < layer[x][y]) {
                    parent[x][y] = node;
                    layer[x][y] = layer[node.x][node.y] + 1;
                }
                int h = heuristic(x, y, goal.x, goal.y);
                frontier.push(Node{x, y, h + layer[x][y]});
            }
        }
    }
    return num_expand;
}

int search(int type, int sx, int sy, int gx, int gy) {
    Node start(sx, sy);
    Node goal(gx, gy);

    if (type == 1) return BFS(start, goal);
    if (type == 2) return DFS(start, goal);
    if (type == 3) return IDS(start, goal);
    if (type == 4) return A_star(start, goal);
    if (type == 5) return IDA_star(start, goal);
    return 0;
}

int main(int argc, char **argv) {
    if (argc < 6) {
        printf("Usage: ./search [type] [start x] [start y] [goal x] [goal y]\n");
        printf("type: 1.BFS 2.DFS 3.IDS 4.A* 5.IDA*\n");
        exit(EXIT_FAILURE);
    }
    auto t1 = std::chrono::high_resolution_clock::now();
    search(atoi(argv[1]), atoi(argv[2]), atoi(argv[3]), atoi(argv[4]), atoi(argv[5]));
    auto t2 = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>( t2 - t1 ).count();
    cout << duration << " us\n";
    return 0;
}
```