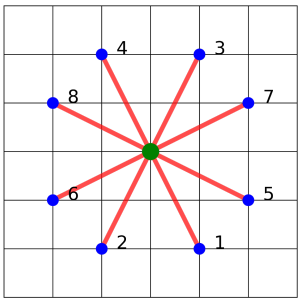
Homework 1 Report

0516034 楊翔鈞

作業題目為給定一張 8 * 8 的棋盤，以及起點跟終點的位置，棋子從起點開始走，按照象棋中「馬」的走法移動，找出一條路徑使得從起點走到終點所花費的步數最少，為了方便書寫，這邊將最佳路徑定義為花費步數最少的路徑。

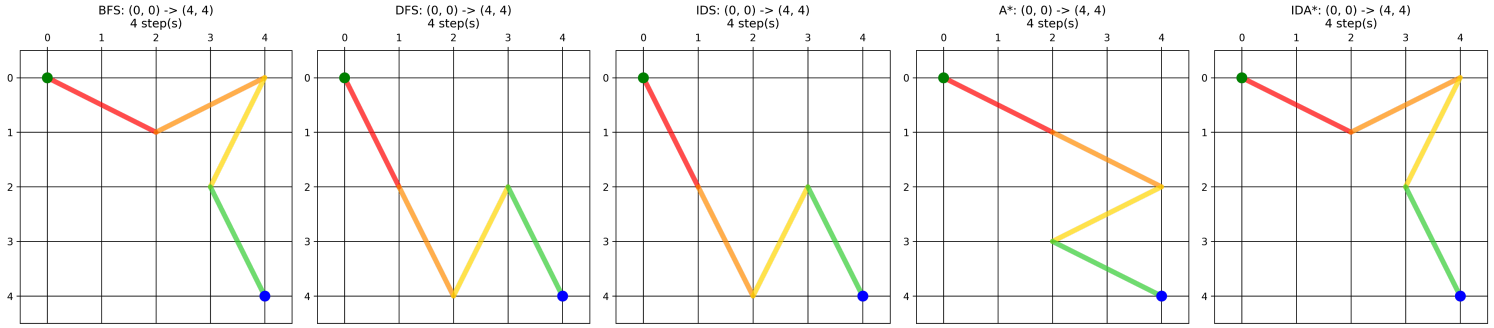
這次作業會利用 BFS、DFS、IDS、A* 還有 IDA* 這五種演算法來算最佳路徑，下面會用不同大小的棋盤來呈現路徑搜尋結果。每種大小的棋盤我都會嘗試三種起、終點的組合，分別是從左上角走到右下角（對角線），左上角走到右上角（棋盤的邊），還有隨機選擇兩個點作為起點與終點，假設這三種組合中有任一種的搜尋時間超過兩分鐘，就不討論該演算法，對於這三種組合，我會挑一個比較能夠看出各個演算法之間的差異的情況來討論。

另外補充一點，在我的程式碼中有一個 `get_next_steps` 的函數，功能是：給定一個位置，函數會回傳一個 array，array 中是下一步的所有可能位置，如右圖所示，綠點為目前位置，藍點為下一步可以到達的地方，DFS 與 IDS 就是按照圖上所標示的順序 traverse。



5 * 5 的棋盤

由左而右分別是 BFS、DFS、IDS、A*、IDA*，綠點為起點，藍點為終點。



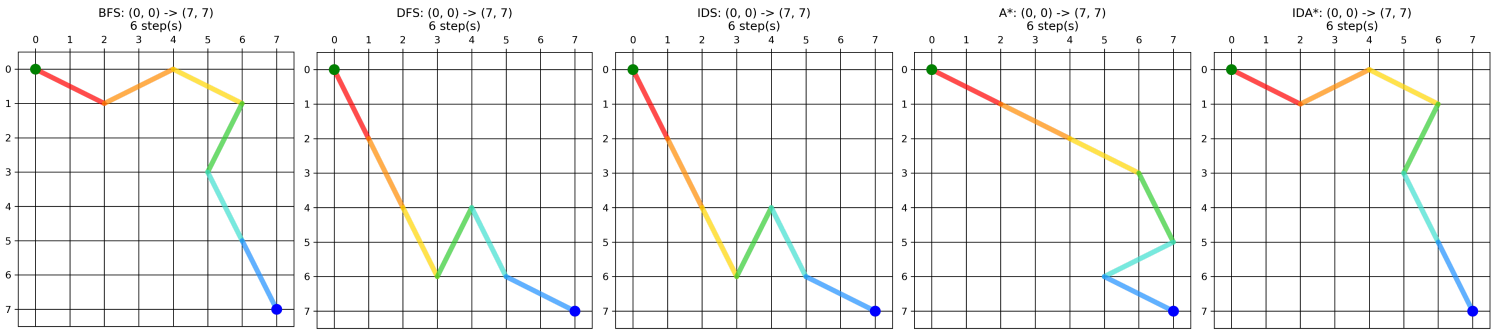
從（0, 0）走到（4, 4）的最佳路徑為 4 步，執行時間與 expand 的數量為：

- BFS：0.005 秒、38 個 node
- DFS：0.006 秒、111 個 node
- IDS：0.005 秒、60 個 node
- A*：0.005 秒、15 個 node
- IDA*：0.005 秒、27 個 node

因為棋盤很小，各個演算法的執行時間還沒有明顯的差異。

8 * 8 的棋盤（作業環境）

由左而右分別是 BFS、DFS、IDS、A*、IDA*。



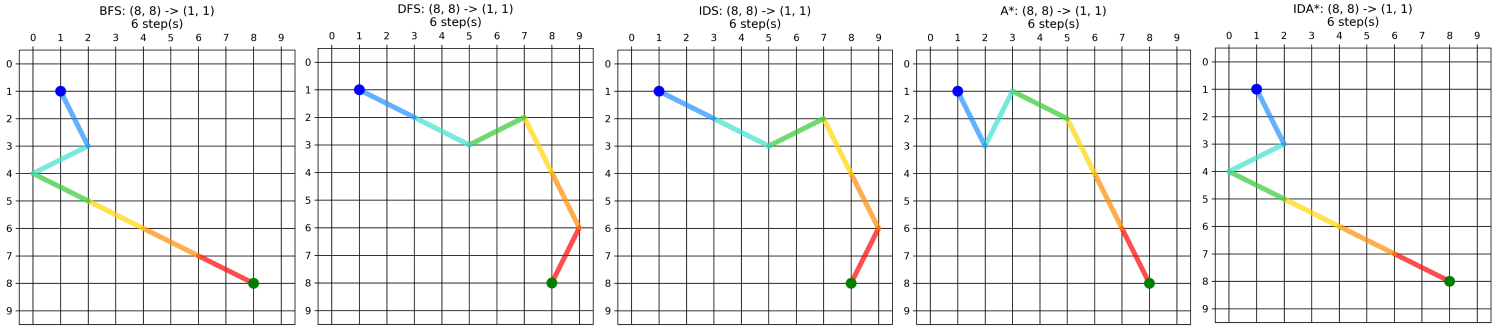
從（0, 0）走到（7, 7）的最佳路徑為 6 步，執行時間與 expand 的數量為：

- BFS：0.006 秒、583 個 node
- DFS：0.034 秒、7874 個 node
- IDS：0.015 秒、2017 個 node
- A*：0.005 秒、38 個 node
- IDA*：0.005 秒、177 個 node

可以發現 DFS 與 IDS 的執行時間與 expand 數量明顯超過其他三種演算法。

10 * 10 的棋盤

由左而右分別是 BFS、DFS、IDS、A*、IDA*。



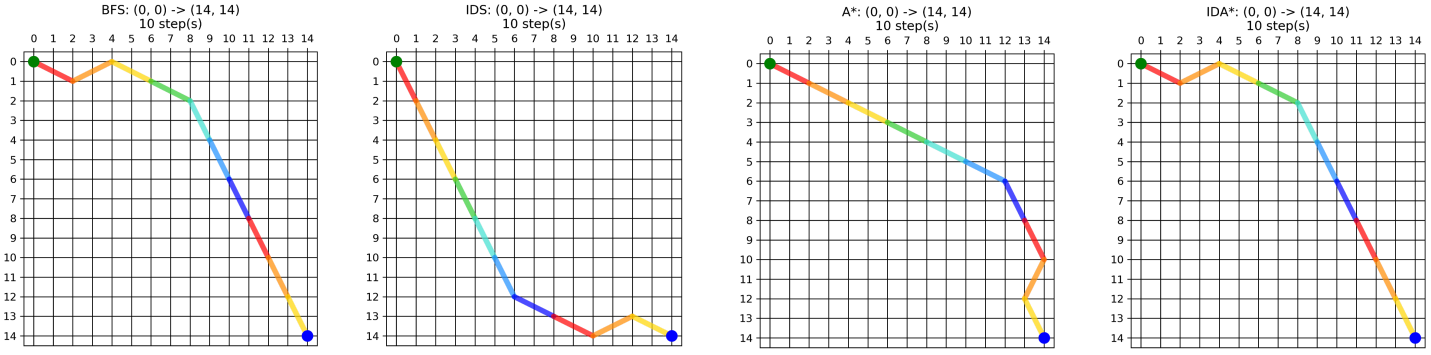
從（8, 8）走到（1, 1）的最佳路徑為 6 步，執行時間與 expand 的數量為：

- BFS：0.007 秒、1360 個 node
- DFS：0.477 秒、98615 個 node
- IDS：0.027 秒、5541 個 node
- A*：0.006 秒、43 個 node
- IDA*：0.006 秒、227 個 node

DFS 的執行時間已經是其他演算法的 60 到 70 倍了。

15 * 15 的棋盤

由左而右分別是 BFS、IDS、A*、IDA*。



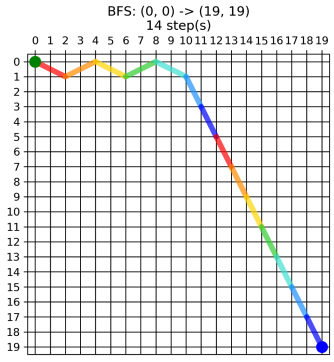
從（0, 0）走到（14, 14）的最佳路徑為 10 步，執行時間與 expand 的數量為：

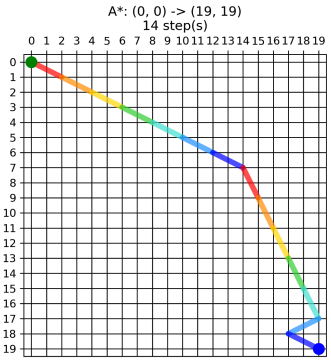
- BFS：0.034 秒、42958 個 node
- IDS：9 秒、2867779 個 node
- A*：0.006 秒、73 個 node
- IDA*：0.007 秒、1058 個 node

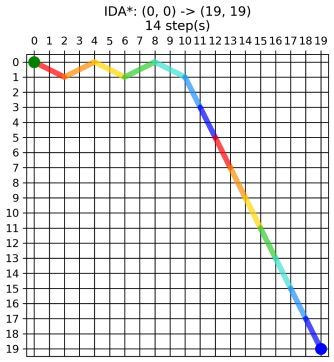
當棋盤變成 15 * 15 時，DFS 對於左上角到右上角的最佳路徑已經沒辦法在短時間內算出來，同時 IDS 消耗的時間與 expand 的數量也遠超過其他三種演算法。

20 * 20 的棋盤

由左而右分別是 BFS、A*、IDA*。







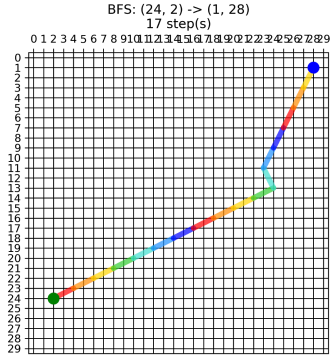
從（0, 0）走到（19, 19）的最佳路徑為 14 步，執行時間與 expand 的數量為：

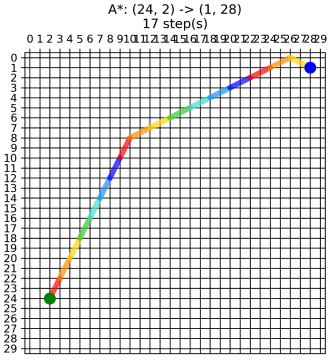
- BFS：0.479 秒、1085575 個 node
- A*：0.007 秒、230 個 node
- IDA*：0.095 秒、1127888 個 node

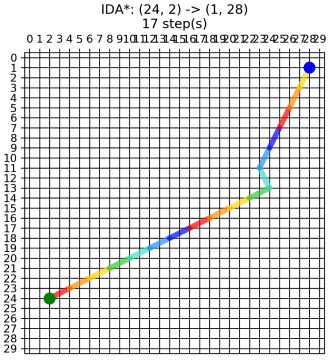
此時 BFS 與 A*、IDA* 的速度也逐漸拉開了，而且 IDS 已經無法在短時間內找出最佳路徑。

30 * 30 的棋盤

由左而右分別是 BFS、A*、IDA*。





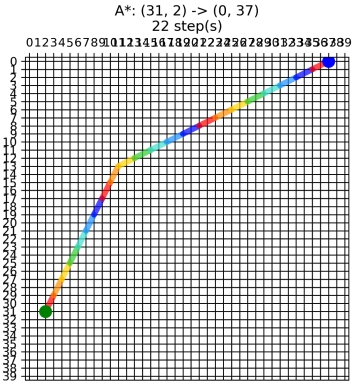


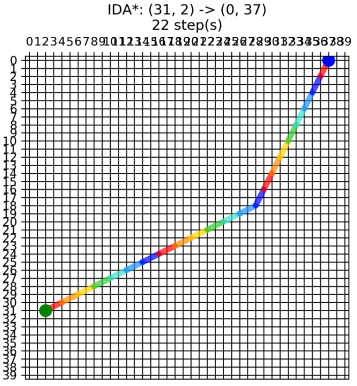
從（24, 2）走到（1, 28）的最佳路徑為 17 步，執行時間與 expand 的數量為：

- BFS：18.798 秒、40359450 個 node
- A*：0.007 秒、192 個 node
- IDA*：0.21 秒、210376 個 node

40 * 40 的棋盤

由左而右分別是 A*、IDA*。

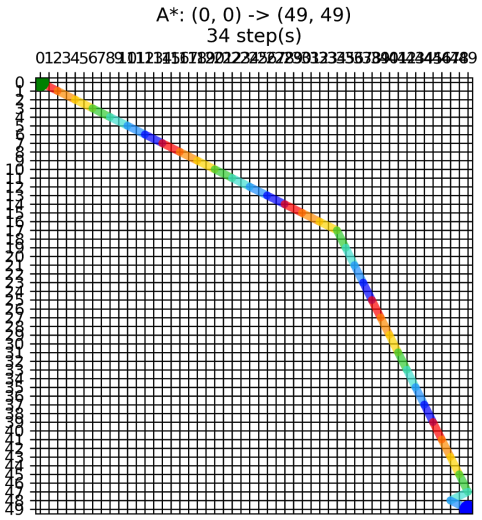




從（31, 2）走到（0, 37）的最佳路徑為 22 步，執行時間與 expand 的數量為：

- A*：0.008 秒、256 個 node
- IDA*：0.01 秒、2085 個 node

50 * 50 的棋盤



從（0, 0）走到（49, 49）的最佳路徑為 34 步，A* 花了 0.018 秒，並且 expand 了 1210 個 node，當棋盤變成 50 * 50 時，只剩下 A* 有辦法在限時內找出最佳路徑。

觀察

- 最佳路徑
  - DFS 跟 IDS：兩者找出的最佳路徑的一樣，因為 IDS 本身也是透過 DFS 來達成，所以 traverse 的順序相同。另外，這兩種演算法的第一步往往跟另外三種不一樣，原因是對於下一步可能的位置，DFS traverse 的順序與其他演算法不同。
  - BFS 跟 IDA*：兩者找出的最佳路徑一樣，但這部分我不太清楚為什麼，我有嘗試將移動方式改為（±1, ±3）、（±3, ±1），並把 heuristic function 改為 Manhattan distance 除以四，BFS 與 IDA* 算出來的結果還是一樣。
  - A*：A* 的每一步幾乎都是朝著終點的方向在移動，因為 A* 是根據與終點的預估距離在尋找下一步可能的落點。
- 速度
  - 執行速度：A* > IDA* > BFS > IDS > DFS
  - DFS：深度優先，走完一條完整的路徑才會走下一條，當找到一條由起點到終點的路徑時，無法確定該路徑為最佳路徑，必須把所有可能的路徑都走過才能找出最佳路徑，而且棋盤的邊長每加一，執行時間就會多好幾倍，非常很耗時間。
  - BFS：廣度優先，以起點為中心，每次向外擴張一個深度，直到碰到終點，所以一旦找到一條起點到終點的路徑，則必定為最佳路徑，不過隨著起點與終點間的距離越來越大，BFS 需要搜尋的範圍也就越來越大，執行時間也就變長了。
  - IDS：改良自 DFS，重複執行有深度限制的 DFS，並在每次執行結束後，將深度限制加大，直到找到終點。假設最佳路徑需要 n 步（終點在距離起點深度為 n 的地方），那深度限制為 n 的 DFS 執行時間應該會跟 BFS 所需的時間差不多，但是因為 IDS 先前必須把深度限制 1 到 n-1 的 DFS 都跑過，所以 IDS 整體的執行速度會比 BFS 慢，但解決了 BFS 記憶體使用過多的問題，所以 IDS 算是綜合了 BFS 的速度與 DFS 的記憶體使用量。
  - A*：利用 heuristic function 來估算可能的路徑長度，所以相較於 BFS 無條件擴張搜尋深度，A* 是有計畫地搜尋可能是最佳路徑的路線，所以即便棋盤大小加大到 50 * 50，還是可以在極短的時間內就找到最佳路徑，我後來把棋盤改成 1000 * 1000，然後找左上角到右下角的最佳路徑， A* 花了 15 秒就找出來了。
  - IDA*：解決了 A* 過高的空間複雜度，但缺點是同樣的路徑會被搜尋很多遍，而且在資料結構方面，A* 是使用 min-heap 來選擇要搜尋的點，IDS* 則是去疊代每個可能的點，所以執行時間會比 A* 長。
- Expand 數量
  - 數量：DFS > IDS > BFS > IDA* > A*
  - DFS：目前的路走到底才會搜尋另一條路，所以 expand 的數量很龐大。
  - BFS：主要跟路徑深度有關，最佳路徑的深度內的 node 都會被 expand。

    - IDS：由於有深度限制，所以不會像 DFS 一樣走到底才回頭，不過因為一直執行有深度限制的 DFS，所以 expand 的數量會比 BFS 多。
    - A*：因為只會搜尋較可能是最佳路徑的路線，所以 expand 的數量極少。
    - IDA*：跟 IDS 的原因類似，不斷搜尋已經搜尋過的路線，所以 expand 的次數會比 A* 多。
- 記憶體
    - 記憶體消耗：BFS > IDS > DFS > A* > IDA*
    - 在執行 BFS 的時候電腦記憶體的使用量有明顯的上升，這與 BFS 的空間複雜度有關，因為是廣度優先，所以會同時將各種相同深度、不同的方向的路徑存在記憶體內，空間複雜度就是 O(b^d)，其中 b 是每個 node 的最大分支數量，d 是路徑深度。
    - 另外講義上也提到 A* 同樣存在空間複雜度過高的問題，不過我從 5 * 5 一路跑到 50 * 50，A* 的記憶體使用量並沒有明顯的變化，後來我把棋盤放大到 1000 * 1000 才有些許上升。

總結

嘗試了各種不同大小的棋盤，互相比較之後 A* 是最穩定的一種搜尋方法，IDA* 則因為一直重複搜尋同樣的路徑，而而對於較大的棋盤，DFS 跟 BFS 都不太適合做最佳路徑搜尋，前者是執行時間太長，後者是記憶體用量太高，IDS 雖然綜合了 BFS 跟 DFS 的優點，但是對於大型棋盤一樣沒辦法在短時間求出最佳路徑。

疑問

不清楚為何 BFS 跟 IDA* 的執行結果會相同，我覺得有可能是因為這次用的 heiristic function 與棋子的移動方式，剛好讓這兩個演算法的解相同，希望之後能嘗試其他不同的 heuristic function 跟移動方式，來看看是不是真的會受這些原因影響。

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

typedef pair<int, int> Point;
typedef vector<Point> Point_vec;
typedef set<Point> Point_set;

const int INF = 1e9;
const int FOUND = -1;
int board_size;
Point_vec optimal_path;

// custom hash function for unordered_map
struct hash_pair {
    template <class T1, class T2>
    size_t operator()(const pair<T1, T2>& p) const {
        auto hash1 = hash<T1>{}(p.first);
        auto hash2 = hash<T2>{}(p.second);
        return hash1 ^ hash2;
    }
};


// return the coordinates of next 8 steps
// move policy:
//     (1, 2), (-1, 2), (1, -2), (-1, -2),
//     (2, 1), (-2, 1), (2, -1), (-2, -1)
Point_vec get_next_steps(Point &pos) {
    int r, c;
    int new_r, new_c;
    r = pos.first;
    c = pos.second;
    Point_vec next_steps;
    for(int dr=-1; dr<=1; dr+=2) {
        for(int dc=-2; dc<=2; dc+=4) {
            new_r = r + dr;
            new_c = c + dc;
            if(!(new_r<0 || new_r>=board_size || new_c<0 ||
new_c>=board_size)) {
                next_steps.push_back(make_pair(new_r, new_c));
            }
        }
    }
    for(int dr=-2; dr<=2; dr+=4) {
        for(int dc=-1; dc<=1; dc+=2) {
```

```
            new_r = r + dr;
            new_c = c + dc;
            if(!(new_r<0 || new_r>=board_size || new_c<0 ||
new_c>=board_size)) {
                next_steps.push_back(make_pair(new_r, new_c));
            }
        }
    }
    return next_steps;
}


// check if the elem exists in vec
bool is_in(Point_vec &vec, Point &elem) {
    for(int i=0; i<vec.size(); i++) {
        if(vec[i] == elem) return true;
    }
    return false;
}


// compute Manhattan distance
int get_manhattan_dis(Point &p1, Point &p2) {
    return abs(p1.first-p2.first)+abs(p1.second-p2.second);
}


// heuristic function used in this assignment
int get_h_score(Point &p1, Point &p2) {
    return (int)floor(get_manhattan_dis(p1, p2) / 3.0);
}


int bfs(Point start, Point goal) {
    Point_vec path, next_steps;
    Point_set explored_set;
    queue<Point_vec> q; // implement the bfs with queue
    Point node, pos;
    int expanded = 0;

    optimal_path = Point_vec{}; // initialize the optimal path to empty
vector
    q.push(Point_vec{start}); // push the starting point into the queue

    // while the queue is not empty
    while(q.size()) {
        path = q.front(); // get the front element of the queue
        q.pop();
        node = path[path.size()-1]; // current node is the last node in
```

```
the path

        // if the path is found
        if(node == goal) {
            optimal_path = path;
            return expanded;
        }

        next_steps = get_next_steps(node); // get the next 8 steps
        explored_set.insert(node); // current node is expanded, add it
to the explored set
        expanded++;
        for(int i=0; i<next_steps.size(); i++) {
            pos = next_steps[i];

            // push this position into queue if it is not yet expanded
            if(explored_set.find(pos) == explored_set.end()) {
                Point_vec new_path(path);
                new_path.push_back(pos);
                q.push(new_path);
            }
        }
    }
    return expanded;
}


int dfs(Point start, Point goal) {
    stack<pair<Point_vec, Point_set>> s; // implement the dfs with stack
    pair<Point_vec, Point_set> state;
    pair<int, Point> candidate;
    Point_vec path, next_steps;
    Point_set explored_set;
    Point node, pos;
    int minimum_step = 1e5;
    int expanded = 0;

    optimal_path = Point_vec{}; // initialize the optimal path to empty
vector
    s.push(make_pair(Point_vec{start}, Point_set{})); // push the
starting point and corresponding status set into the queue

    // while the stack is not empty
    while(s.size()) {
        state = s.top(); // get the top element of the stack
        s.pop();
        path = state.first;
        explored_set = state.second;
        node = path[path.size()-1]; // current node is the last node in
```

```
the path

        // if the path is found
        if(node == goal) {
            // update the optimal path if the current path is better
than that
            if(path.size() < minimum_step) {
                minimum_step = path.size();
                optimal_path = path;
            }
            continue; // keep going until all the path are found
        }

        // if the current length of path is longer than the one of
optimal path,
        // it is impossible to become a new optimal path
        if(path.size() > minimum_step) continue;

        next_steps = get_next_steps(node); // get the next 8 steps
        explored_set.insert(node); // current node is expanded, add it
to the explored set
        expanded++;
        for(int i=0; i<next_steps.size(); i++) {
            pos = next_steps[i];

            // push this position into stack if it is not yet expanded
            if(explored_set.find(pos) == explored_set.end()) {
                Point_vec new_path(path);
                new_path.push_back(pos);
                s.push(make_pair(new_path, explored_set));
            }
        }
    }
    return expanded;
}


// DFS with limited depth
pair<Point_vec, int> deep_limited_search(Point start, Point goal, int
depth) {
    stack<tuple<Point_vec, Point_set, int>> s; // implement the dfs with
stack
    tuple<Point_vec, Point_set, int> state;
    pair<int, Point> candidate;
    Point_vec path, next_steps;
    Point_set explored_set;
    Point node, pos;
    int cur_depth;
    int expanded = 0;
```

```
    s.push(make_tuple(Point_vec{start}, Point_set{}, depth)); // push
the starting point and corresponding status into the queue

    // while the stack is not empty
    while(s.size()) {
        state = s.top(); // get the top element of the stack
        s.pop();
        path = get<0>(state);
        explored_set = get<1>(state);
        cur_depth = get<2>(state);
        node = path[path.size()-1]; // current node is the last node in
the path

        // if the path is found
        if(node == goal) return make_pair(path, expanded);

        // abandon this state if it reaches the depth limit
        if(cur_depth <= 0) continue;

        next_steps = get_next_steps(node); // get the next 8 steps
        explored_set.insert(node); // current node is expanded, add it
to the explored set
        expanded++;
        for(int i=0; i<next_steps.size(); i++) {
            pos = next_steps[i];

            // push this position into stack if it is not yet expanded
            if(explored_set.find(pos) == explored_set.end()) {
                Point_vec new_path(path);
                new_path.push_back(pos);
                s.push(make_tuple(new_path, explored_set, cur_depth-1));
            }
        }
    }
    return make_pair(Point_vec{}, expanded);
}


int ids(Point start, Point goal) {
    pair<Point_vec, int> result;
    Point_vec path;
    int expanded = 0;
    optimal_path = Point_vec{}; // initialize the optimal path to empty
vector

    // run repeatedly with increasing depth limits until the goal is
found
    for(int depth=1; depth<=board_size*board_size; depth++) {
```

```
        result = deep_limited_search(start, goal, depth);
        path = result.first;
        expanded += result.second;

        // if the path is found, its size will be greater than zero
        if(path.size()) {
            optimal_path = path;
            return expanded;
        }
    }
    return expanded;
}


int a_star(Point start, Point goal) {
    priority_queue<
        pair<int, Point_vec>,
        vector<pair<int, Point_vec>>,
        greater<pair<int, Point_vec>>> open_set; // implement the A*
with min-heap
    unordered_map<Point, int, hash_pair> g_score, h_score, f_score;
    pair<int, Point_vec> state;
    Point_vec path, next_steps;
    Point_set closed_set;
    Point node, pos;
    int g, h, f;
    int expanded = 0;

    optimal_path = Point_vec{}; // initialize the optimal path to empty
vector
    g_score[start] = 0; // initial g-score of starting point is 0
    h_score[start] = get_h_score(start, goal); // initial h-score of
starting point is h(start)
    f_score[start] = g_score[start] + h_score[start];
    open_set.push(make_pair(f_score[start], Point_vec{start})); // push
the starting point and corresponding status into the min-heap

    // while min-heap is not empty
    while(open_set.size()) {
        state = open_set.top(); // get the top element of the min-heap
        open_set.pop();
        path = state.second;
        node = path[path.size()-1]; // current node is the last node in
the path

        // if the path is found
        if(node == goal) {
            optimal_path = path;
            return expanded;
```

```
        }

        next_steps = get_next_steps(node); // get the next 8 steps
        closed_set.insert(node); // current node is expanded, add it to
the closed set
        expanded++;
        for(int i=0; i<next_steps.size(); i++) {
            pos = next_steps[i];

            // consider this position if it is not yet expanded
            if(closed_set.find(pos) == closed_set.end()) {
                g = g_score[node] + 1;
                h = get_h_score(pos, goal);
                f = g + h; // update the g-score and compute h(pos)

                // push this position into min-heap if
                //     1. this position is not yet visited
                //     2. current g-score if smaller than the old one
                //        (it means that this position can be reached
with less costs)
                if(g_score.find(pos)==g_score.end() || g<g_score[pos]) {
                    g_score[pos] = g;
                    h_score[pos] = h;
                    f_score[pos] = f;
                    Point_vec new_path(path);
                    new_path.push_back(pos);
                    open_set.push(make_pair(f, new_path));
                }
            }
        }
    }
    return expanded;
}


pair<int, int> ida_star_search(Point_vec path, Point goal, int g, int
thres) {
    pair<int, int> result;
    Point_vec next_steps;
    Point node, pos;
    int min, f, t;
    int expanded = 0;

    node = path[path.size()-1]; // current node is the last node in the
path
    f = g + get_h_score(node, goal); // compute the f-score from current
node to goal

    // update the threshold if f-score is greater than it
```

```
    if(f > thres) return make_pair(f, expanded);

    // if the path is found
    if(node == goal) {
        optimal_path = path;
        return make_pair(FOUND, expanded);
    }

    min = INF;
    next_steps = get_next_steps(node); // get the next 8 steps
    expanded++;

    for(int i=0; i<next_steps.size(); i++) {
        pos = next_steps[i];

        // if ths position is not in current path
        if(!is_in(path, pos)) {
            Point_vec new_path(path);
            new_path.push_back(pos);
            result = ida_star_search(new_path, goal, g+1, thres); // go
to next step and see if it can find the path or update the threshold
            t = result.first;
            expanded += result.second;

            // if the path id found
            if(t == FOUND) return make_pair(t, expanded);

            // update the threshold
            if(t < min) min = t;
        }
    }
    return make_pair(min, expanded);
}


int ida_star(Point start, Point goal) {
    int t, thres = get_h_score(start, goal);
    int expanded = 0;
    pair<int, int> result;
    Point_vec path{start};
    optimal_path = Point_vec{};
    while(1) {
        result = ida_star_search(path, goal, 0, thres);
        t = result.first; // keep going and update the threshold until
the path is found
        expanded += result.second;
        if(t == FOUND) return expanded;
        if(t > INF) optimal_path = Point_vec{}; // if the path doesn't
exist
```

```
        thres = t;
    }
    return expanded;
}


int solve(int search_func, int start_x, int start_y, int goal_x, int
goal_y) {
    Point start = make_pair(start_x, start_y);
    Point goal = make_pair(goal_x, goal_y);
    int expanded;
    switch(search_func) {
    case 0:
        return bfs(start, goal);
        break;
    case 1:
        return dfs(start, goal);
        break;
    case 2:
        return ids(start, goal);
        break;
    case 3:
        return a_star(start, goal);
        break;
    case 4:
        return ida_star(start, goal);
        break;
    default:
        return -1;
        break;
    }
}


// write the optimal path to the file
void output_path() {
    fstream fout;
    fout.open("path", ios::out);
    for(int i=0; i<optimal_path.size(); i++) {
        fout << optimal_path[i].first << " " << optimal_path[i].second
<< endl;
    }
    fout.close();
}


int main(int argc, char *argv[]) {
    if(argc < 7) {
        cout << "Usage: ./HW1 <searching-method> <board-size> <start-x>
```

```
<start-y> <goal-x> <goal-y>\n";
        cout << "\nSearching method:\n0) BFS\n1) DFS\n2) IDS\n3) A*\n4)
IDA*\n";
    }
    else {
        int search_func = stoi(string(argv[1]));
        board_size = stoi(string(argv[2]));
        int start_x = stoi(string(argv[3]));
        int start_y = stoi(string(argv[4]));
        int goal_x = stoi(string(argv[5]));
        int goal_y = stoi(string(argv[6]));
        int expanded = 0;

        if(search_func > 4) {
            cout << "Unknown searching method.\n";
        }
        else if(start_x<0 || start_x>=board_size || goal_x<0 ||
goal_x>=board_size) {
            cout << "Constraint:\n0 <= x, y <= " << board_size << endl;
        }
        else {
            vector<string> algo{"BFS", "DFS", "IDS", "A*", "IDA*"};
            expanded = solve(search_func, start_x, start_y, goal_x,
goal_y);
            cout << algo[search_func] << endl;
            cout << "Expanded: " << expanded << endl;
            cout << "Steps: " << optimal_path.size()-1 << endl;
            cout << "--------------------\n";
            output_path();
        }
    }
    return 0;
}
```