School of Data and Computer Science, Sun Yat-sen University

E03 Othello Game ($\alpha - \beta$ pruning)

18364066 Yanzuo Lu

September 22, 2020

Contents

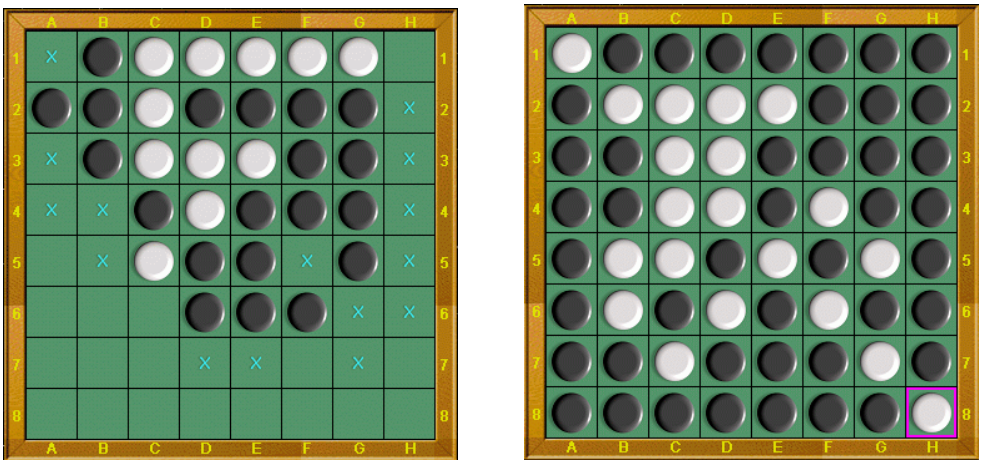

Figure 1: Othello Game

1 Othello

Othello (or Reversi) is a strategy board game for two players, played on an 8×8 uncheckered board. There are sixty-four identical game pieces called disks (often spelled "discs"), which are light on one side and dark on the other. Please see figure 1.

Players take turns placing disks on the board with their assigned color facing up. During a play, any disks of the opponent's color that are in a straight line and bounded by the disk just placed and another disk of the current player's color are turned over to the current player's color.

The object of the game is to have the majority of disks turned to display your color when the last playable empty square is filled.

You can refer to `http://www.tothello.com/html/guideline_of_reversed_othello.html` for more information of guideline, meanwhile, you can download the software to have a try from `http://www.tothello.com/html/download.html`. The game installer `tothello_trial_setup.exe` can also be found in the current folder.

2 Tasks

1. In order to reduce the complexity of the game, we think the board is 6×6.

2. There are several evaluation functions that involve many aspects, you can turn to `http://www.cs.cornell.edu/~yuli/othello/othello.html` for help. In order to reduce the difficulty of the task, I have gaven you some hints of evaluation function in the file `Heuristic Function for Reversi (Othello).cpp`.

3. Please choose an appropriate evaluation function and use min-max and $\alpha - \beta$ prunning to

implement the Othello game. The framework file you can refer to is `Othello.cpp`. Of course, I wish your program can beat the computer.

4. Write the related codes and take a screenshot of the running results in the file named E03_StudentNumber.pdf and send it to ai_2020@foxmail.com, the **deadline** is 2020.09.20 23:59:59.

3 Codes

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

constexpr double kMaxValue = (double)0x3f3f3f3f;
constexpr int kSearchDepth = 9;
constexpr int kDirection[8][2] = {{-1, 0}, {-1, 1}, {0, 1}, {1, 1},
    {1, 0}, {1, -1}, {0, -1}, {-1, -1}};

enum class Piece                        // 棋子
{
    SPACE = -1,                         // 空位置
    WHITE,                              // 白棋
    BLACK                               // 黑棋
};
Piece max_player;                       // AI执黑/执白

typedef struct Move                     // 落子
{
    int pos_x;                          // 竖方向
    int pos_y;                          // 横方向
}Move;

typedef struct Othello
{
    Piece board[6][6];                  // 棋盘
    int black_reverse[6][6];            // 可翻转黑棋数
    int white_reverse[6][6];            // 可翻转白棋数
    int white_num;                      // 白棋数
    int black_num;                      // 黑棋数

    void InitBoard(Othello* my_board);
    void CopyBoard(Othello* dest_board, Othello* src_board);
    void PrintBoard(Othello* my_board);
    int Action(Othello* my_board, Move* choice, Piece player);
    void ReverseCount(Othello* my_board);
    int ValidPlaces(Othello* my_board, Piece player);
    double MiniMax(Othello* my_board, int depth, Piece player,
        double alpha, double beta, Move* choice);
    double Evaluation(Othello* my_board);
```

```
    void IsGoal(Othello* my_board);
}Othello;

void Othello::InitBoard(Othello* my_board)
{
    // 初始化棋盘
    for (int i = 0; i < 6; i++)
    {
        for (int j = 0; j < 6; j++)
        {
            my_board->board[i][j] = Piece::SPACE;
            my_board->black_reverse[i][j] = 0;
            my_board->white_reverse[i][j] = 0;
        }
    }
    my_board->white_num = 2;
    my_board->black_num = 2;
    my_board->board[2][2] = my_board->board[3][3] = Piece::WHITE;
    my_board->board[2][3] = my_board->board[3][2] = Piece::BLACK;
    ReverseCount(my_board);
}

void Othello::CopyBoard(Othello* dest_board, Othello* src_board)
{
    // 复制棋盘
    for (int i = 0; i < 6; i++)
    {
        for (int j = 0; j < 6; j++)
        {
            dest_board->board[i][j] = src_board->board[i][j];
            dest_board->black_reverse[i][j] = src_board->black_reverse[i][j];
            dest_board->white_reverse[i][j] = src_board->white_reverse[i][j];
        }
    }
    dest_board->white_num = src_board->white_num;
    dest_board->black_num = src_board->black_num;
}

void Othello::PrintBoard(Othello* my_board)
{
    // 打印棋盘
    cout << "————————————" << endl;
    for (int i = 0; i < 6; i++)
    {
        cout << "|";
        for (int j = 0; j < 6; j++)
        {
            if (my_board->board[i][j] == Piece::SPACE) cout << "␣|";
            else if (my_board->board[i][j] == Piece::BLACK) cout << "●|";
```

```
            else cout << " |";
        }
        cout << endl << "---------------" << endl;
    }
    cout << "Black Piece: " << my_board->black_num << "\tWhite Piece: "
        << my_board->white_num << endl;
}

int Othello::Action(Othello* my_board, Move* choice, Piece player)
{
    // 落子
    int x, y, player_reverse;
    x = choice->pos_x;
    y = choice->pos_y;
    if (player == Piece::BLACK)
        player_reverse = my_board->black_reverse[x][y];
    else player_reverse = my_board->white_reverse[x][y];

    if (x < 0 || x > 5 || y < 0 || y > 5 || my_board->board[x][y] !=
        Piece::SPACE || player_reverse == 0)
        return 0;
    my_board->board[x][y] = player;
    if (player == Piece::BLACK) my_board->black_num++;
    else my_board->white_num++;

    // 八个方向依次遍历
    for (int i = 0; i < 8; i++)
    {
        int dx = kDirection[i][0];
        int dy = kDirection[i][1];
        int new_x = x;
        int new_y = y;
        Piece another_player;
        if (player == Piece::BLACK) another_player = Piece::WHITE;
        else another_player = Piece::BLACK;

        int straight_count = 0;
        int flag = 0;
        // 同一方向一直走下去直到遇到边界、空白这两个不合法情况
        while (true)
        {
            new_x += dx;
            new_y += dy;

            if (new_x < 0 || new_x > 5 || new_y < 0 || new_y > 5 ||
                my_board->board[new_x][new_y] == Piece::SPACE) break;
            if (my_board->board[new_x][new_y] == another_player)
                straight_count++;
            else if (my_board->board[new_x][new_y] == player)
```

```
            {
                flag++;
                break;
            }
        }

        if (flag)
        {
            new_x = x;
            new_y = y;
            for (int j = 0; j < straight_count; j++)
            {
                new_x += dx;
                new_y += dy;
                my_board->board[new_x][new_y] = player;
            }
            if (player == Piece::BLACK)
            {
                my_board->black_num += straight_count;
                my_board->white_num -= straight_count;
            }
            else
            {
                my_board->white_num += straight_count;
                my_board->black_num -= straight_count;
            }
        }
    }
    ReverseCount(my_board);
    return 1;
}

void Othello::ReverseCount(Othello* my_board)
{
    // 计算当前棋盘上每个空白位置落子所能翻转对手棋子的个数
    for (int i = 0; i < 6; i++)
    {
        for (int j = 0; j < 6; j++)
        {
            if (my_board->board[i][j] != Piece::SPACE)
            {
                my_board->black_reverse[i][j] = 0;
                my_board->white_reverse[i][j] = 0;
                continue;
            }
            my_board->black_reverse[i][j] = 0;
            my_board->white_reverse[i][j] = 0;
            for (int k = 0; k < 8; k++)
            {
```

```
                int dx = kDirection[k][0];
                int dy = kDirection[k][1];
                int new_x = i + dx;
                int new_y = j + dy;
                if (new_x < 0 || new_x > 5 || new_y < 0 || new_y > 5 ||
                    my_board->board[new_x][new_y] == Piece::SPACE) continue;
                Piece player = my_board->board[new_x][new_y];
                Piece another_player;
                if (player == Piece::BLACK) another_player = Piece::WHITE;
                else another_player = Piece::BLACK;

                int straight_count = 0;
                while (true)
                {
                    if (new_x < 0 || new_x > 5 || new_y < 0 || new_y > 5 ||
                        my_board->board[new_x][new_y] == Piece::SPACE) break;
                    if (my_board->board[new_x][new_y] == another_player)
                    {
                        if (another_player == Piece::BLACK)
                            my_board->black_reverse[i][j] += straight_count;
                        else my_board->white_reverse[i][j] += straight_count;
                        break;
                    }
                    else straight_count++;
                    new_x = new_x + dx;
                    new_y = new_y + dy;
                }
            }
        }
    }
}

int Othello::ValidPlaces(Othello* my_board, Piece player)
{
    // 遍历棋盘上所有空白位置，读取翻转数组，计算当前player
    // 可落子的空白位置数量
    int valid_places = 0;
    for (int i = 0; i < 6; i++)
    {
        for (int j = 0; j < 6; j++)
        {
            if (player == Piece::BLACK &&
                my_board->black_reverse[i][j] > 0)
                valid_places++;
            else if (player == Piece::WHITE &&
                my_board->white_reverse[i][j] > 0)
                valid_places++;
        }
    }
```

```
    return valid_places;
}

double Othello::MiniMax(Othello* my_board, int depth, Piece player,
    double alpha, double beta, Move* choice)
{
    choice->pos_x = -1;
    choice->pos_y = -1;

    Piece another_player;
    if (player == Piece::BLACK) another_player = Piece::WHITE;
    else another_player = Piece::BLACK;

    if (depth <= 0)
        return Evaluation(my_board);

    if (ValidPlaces(my_board, player) == 0)
    {
        if (ValidPlaces(my_board, another_player) != 0)
        {
            // player当前不可落子
            if (player == max_player)
            {
                Othello temp_board;
                CopyBoard(&temp_board, my_board);
                Move next_choice;
                double temp_alpha = MiniMax(&temp_board, depth - 1,
                    another_player, alpha, beta, &next_choice);
                if (temp_alpha >= alpha) alpha = temp_alpha;
                return alpha;
            }
            else
            {
                Othello temp_board;
                CopyBoard(&temp_board, my_board);
                Move next_choice;
                double temp_beta = MiniMax(&temp_board, depth - 1,
                    another_player, alpha, beta, &next_choice);
                if (temp_beta <= beta) beta = temp_beta;
                return beta;
            }

        }
        else
        {
            // 双方均不可落子
            return Evaluation(my_board);
        }
    }
```

```
if (player == max_player)
{
    for (int k = 0; k < 36; k++)
    {
        int i = k / 6;
        int j = k - i * 6;

        if (my_board->board[i][j] != Piece::SPACE) continue;
        if (player == Piece::BLACK)
        {
            if (my_board->black_reverse[i][j] > 0)
            {
                Othello temp_board;
                CopyBoard(&temp_board, my_board);
                Move temp_choice = { i, j };
                Action(&temp_board, &temp_choice, player);
                Move next_choice;
                // temp_board.PrintBoard(&temp_board);
                double temp_alpha = MiniMax(&temp_board, depth - 1,
                    another_player, alpha, beta, &next_choice);
                if (temp_alpha >= alpha)
                {
                    choice->pos_x = i;
                    choice->pos_y = j;
                    alpha = temp_alpha;
                }
                if (alpha >= beta) break;
            }
        }
        else
        {
            if (my_board->white_reverse[i][j] > 0)
            {
                Othello temp_board;
                CopyBoard(&temp_board, my_board);
                Move temp_choice = { i, j };
                Action(&temp_board, &temp_choice, player);
                Move next_choice;
                // temp_board.PrintBoard(&temp_board);
                double temp_alpha = MiniMax(&temp_board, depth - 1,
                    another_player, alpha, beta, &next_choice);
                if (temp_alpha >= alpha)
                {
                    choice->pos_x = i;
                    choice->pos_y = j;
                    alpha = temp_alpha;
                }
                if (alpha >= beta) break;
```

```
            }
        }
    }
    return alpha;
}
else
{
    for (int k = 0; k < 36; k++)
    {
        int i = k / 6;
        int j = k - i * 6;
        if (my_board->board[i][j] != Piece::SPACE) continue;
        if (player == Piece::BLACK)
        {
            if (my_board->black_reverse[i][j] > 0)
            {
                Othello temp_board;
                CopyBoard(&temp_board, my_board);
                Move temp_choice = { i, j };
                Action(&temp_board, &temp_choice, player);
                Move next_choice;
                // temp_board.PrintBoard(&temp_board);
                double temp_beta = MiniMax(&temp_board, depth - 1,
                    another_player, alpha, beta, &next_choice);
                if (temp_beta <= beta)
                {
                    choice->pos_x = i;
                    choice->pos_y = j;
                    beta = temp_beta;
                }
                if (beta <= alpha) break;
            }
        }
        else
        {
            if (my_board->white_reverse[i][j] > 0)
            {
                Othello temp_board;
                CopyBoard(&temp_board, my_board);
                Move temp_choice = { i, j };
                Action(&temp_board, &temp_choice, player);
                Move next_choice;
                // temp_board.PrintBoard(&temp_board);
                double temp_beta = MiniMax(&temp_board, depth - 1,
                    another_player, alpha, beta, &next_choice);
                if (temp_beta <= beta)
                {
                    choice->pos_x = i;
                    choice->pos_y = j;
```

```
                    beta = temp_beta;
                }
                if (beta <= alpha) break;
            }
        }
    }
    return beta;
}
}

double Othello::Evaluation(Othello* my_board)
{
    int my_tiles = 0, opp_tiles = 0, i, j, k, my_front_tiles = 0,
        opp_front_tiles = 0, x, y;
    double p = 0, c = 0, l = 0, m = 0, f = 0, d = 0;
    int V[6][6] =
    {
        {20, -3, 11, 11, -3, 20},
        {-3, -7, -4, -4, -7, -3},
        {11, -4, 2, 2, -4, 11},
        {11, -4, 2, 2, -4, 11},
        {-3, -7, -4, -4, -7, -3},
        {20, -3, 11, 11, -3, 20}
    };

    Piece another_player;
    if (max_player == Piece::BLACK) another_player = Piece::WHITE;
    else another_player = Piece::BLACK;

    for (i = 0; i < 6; i++)
    {
        for (j = 0; j < 6; j++)
        {
            if (my_board->board[i][j] == max_player)
            {
                d += V[i][j];
                my_tiles++;
            }
            else if (my_board->board[i][j] == another_player)
            {
                d -= V[i][j];
                opp_tiles++;
            }
            if (my_board->board[i][j] != Piece::SPACE)
            {
                for (k = 0; k < 8; k++)
                {
                    x = i + kDirection[k][0];
                    y = j + kDirection[k][1];
```

```
                    if (x >= 0 && x < 6 && y >= 0 && y < 6 &&
                        my_board->board[x][y] == Piece::SPACE)
                    {
                        if (my_board->board[i][j] == max_player)
                            my_front_tiles++;
                        else opp_front_tiles++;
                        break;
                    }
                }
            }
        }
    }

    if (my_tiles > opp_tiles)
        p = (100.0 * my_tiles) / ((double)my_tiles + (double)opp_tiles);
    else if (my_tiles < opp_tiles)
        p = -(100.0 * opp_tiles) / ((double)my_tiles + (double)opp_tiles);
    else
        p = 0;

    if (my_front_tiles > opp_front_tiles)
        f = -(100.0 * my_front_tiles) / ((double)my_front_tiles +
            (double)opp_front_tiles);
    else if (my_front_tiles < opp_front_tiles)
        f = (100.0 * opp_front_tiles) / ((double)my_front_tiles +
            (double)opp_front_tiles);
    else f = 0;

    my_tiles = opp_tiles = 0;
    if (my_board->board[0][0] == Piece::SPACE)
    {
        if (my_board->board[0][1] == max_player) my_tiles++;
        else if (my_board->board[0][1] == another_player) opp_tiles++;
        if (my_board->board[1][1] == max_player) my_tiles++;
        else if (my_board->board[1][1] == another_player) opp_tiles++;
        if (my_board->board[1][0] == max_player) my_tiles++;
        else if (my_board->board[1][0] == another_player) opp_tiles++;
    }

    if (my_board->board[0][5] == Piece::SPACE)
    {
        if (my_board->board[0][4] == max_player) my_tiles++;
        else if (my_board->board[0][4] == another_player) opp_tiles++;
        if (my_board->board[1][4] == max_player) my_tiles++;
        else if (my_board->board[1][4] == another_player) opp_tiles++;
        if (my_board->board[1][5] == max_player) my_tiles++;
        else if (my_board->board[1][5] == another_player) opp_tiles++;
    }
```

```
    if (my_board->board[5][0] == Piece::SPACE)
    {
        if (my_board->board[5][1] == max_player) my_tiles++;
        else if (my_board->board[5][1] == another_player) opp_tiles++;
        if (my_board->board[4][1] == max_player) my_tiles++;
        else if (my_board->board[4][1] == another_player) opp_tiles++;
        if (my_board->board[4][0] == max_player) my_tiles++;
        else if (my_board->board[4][0] == another_player) opp_tiles++;
    }

    if (my_board->board[5][5] == Piece::SPACE)
    {
        if (my_board->board[4][5] == max_player) my_tiles++;
        else if (my_board->board[4][5] == another_player) opp_tiles++;
        if (my_board->board[4][4] == max_player) my_tiles++;
        else if (my_board->board[4][4] == another_player) opp_tiles++;
        if (my_board->board[5][4] == max_player) my_tiles++;
        else if (my_board->board[5][4] == another_player) opp_tiles++;
    }

    l = -12.5 * ((double)my_tiles - (double)opp_tiles);

    my_tiles = ValidPlaces(my_board, max_player);
    opp_tiles = ValidPlaces(my_board, another_player);
    if (my_tiles > opp_tiles)
        m = (100.0 * my_tiles) / ((double)my_tiles + (double)opp_tiles);
    else if (my_tiles < opp_tiles)
        m = -(100.0 * opp_tiles) / ((double)my_tiles + (double)opp_tiles);
    else m = 0;

    double score = (10 * p) + (801.724 * c) + (382.026 * l) + (78.922 * m) +
        (74.396 * f) + (10 * d);
    return score;
}

void Othello::IsGoal(Othello* my_board)
{
    if (my_board->black_num > my_board->white_num)
    {
        if (max_player == Piece::BLACK) cout << "AI获胜!!!" << endl;
        else cout << "你获胜了!!!" << endl;
    }
    else if (my_board->white_num > my_board->black_num)
    {
        if (max_player == Piece::WHITE) cout << "AI获胜!!!" << endl;
        else cout << "你获胜了!!!" << endl;
    }
    else
    {
```

```
        cout << "平局!!!" << endl;
    }
}

int main()
{
    Othello* my_board = new Othello;
    my_board->InitBoard(my_board);
    Piece human_player, current_player;

    while (true)
    {
        cout << "请键入数字选择执黑棋(1)/白棋(0):";
        int temp_player;
        cin >> temp_player;

        if (temp_player == 1)
        {
            max_player = Piece::WHITE;
            human_player = Piece::BLACK;
            break;
        }
        else if (temp_player == 0)
        {
            max_player = Piece::BLACK;
            human_player = Piece::WHITE;
            break;
        }
        else
        {
            cout << "请键入合法数字!!!" << endl;
        }
    }

    // 黑棋先行
    current_player = Piece::BLACK;
    while (true)
    {
        if (current_player == human_player)
        {
            if (my_board->ValidPlaces(my_board, human_player) == 0)
            {
                if (my_board->ValidPlaces(my_board, max_player) == 0)
                {
                    my_board->IsGoal(my_board);
                    break;
                }
                current_player = max_player;
                continue;
```

```
        }

        while (true)
        {
            my_board->PrintBoard(my_board);
            cout << "请输入需要落子的位置:";
            int temp_x, temp_y;
            cin >> temp_x >> temp_y;
            temp_x--;
            temp_y--;
            Move temp_choice = { temp_x, temp_y };
            if (!my_board->Action(my_board, &temp_choice, human_player))
                cout << "输入位置非法!!!" << endl;
            else
            {
                my_board->PrintBoard(my_board);
                break;
            }
        }

        current_player = max_player;
    }
    else
    {
        if (my_board->ValidPlaces(my_board, max_player) == 0)
        {
            if (my_board->ValidPlaces(my_board, human_player) == 0)
            {
                my_board->IsGoal(my_board);
                break;
            }
            current_player = human_player;
            continue;
        }

        Move ai_choice;
        double alpha = my_board->MiniMax(my_board, kSearchDepth,
            max_player, -kMaxValue, kMaxValue, &ai_choice);

        my_board->Action(my_board, &ai_choice, max_player);
        my_board->PrintBoard(my_board);
        cout << "AI选择落子(" << ai_choice.pos_x + 1 << ", " <<
            ai_choice.pos_y + 1 << ")" << endl;
        // cout << alpha << " " << my_board->Evaluation(my_board)
            << endl;

        current_player = human_player;
    }
}
```

```
    delete my_board;
    return 0;
}
```

4 Results

本次实验我对整个代码的框架进行了重写，因为示例代码中的 MiniMax 极大极小值搜索函数是经过了优化和修改的，和我们在课上学习到的伪代码结果有所差异，所以我只参考了示例代码中的启发式函数，其他代码全部是按照自己的理解来重写的。

首先我设置了一个 Othello 结构来包括所有常用的和需要的函数，比较重要的分别是 Action、ReverseCount 和 ValidPlaces 这三个，它们的功能分别是落子（负责翻转所有可翻转的棋子）、计算棋盘上每个空白位置的黑棋和白棋分别可翻转对手棋子的个数、统计棋盘上黑棋或白棋能够落子的位置数量。这三个功能函数无论是在 MiniMax 极大极小值搜索中，还是在 Evaluation 评价函数中都发挥着重要的作用。

然后是 Evaluation 评价函数，值得注意的是我在全局变量中设置了一个 maxplayer 来指代 AI 方面需要极大化的是黑棋还是白棋，所以我只需要按照示例代码中的逻辑将 mycolor 和 oppcolor 分别替换成 maxplayer 和 anotherplayer 即可，anotherplayer 则是用户执黑或执白的表示，通过简单的 if 语句即可获得。

最后是最关键的 MiniMax 函数，参考刘老师在课上跟我们着重讲解的伪代码，因为 Evaluation 返回的总是对于 AI 来说的一个场面分数，对于 AI 在 MAX 层就需要选择极大节点，对于用户在 MIN 层就需要选择极小节点，而不是示例代码中的将二者混合在一起，尽管结合在一起的效率可能更高，但分开来应该是更好理解的。根据伪代码中的逻辑，我们需要在每一层判断是否需要更新 alpha 和 beta 值，尤其需要注意黑白棋是有可能出现二者均无子可下但棋盘未满的情况的，因此我们需要在每一层首先判定是否已经到达了叶节点，若到达则直接对场面进行评估并返回即可。

总结得来说，经过更换了若干个启发式函数，起初按照 pdf 链接中的指导发现 AI 棋力还不如我这种新手，后来更换为示例代码中的较为复杂的动态启发式函数以后，棋力有明显地提升，至少我本人是无法打败 AI 的，但是对于示例程序来说确实是毫无胜算，下面将给出我编写的 AI 互搏的截图，搜索深度为 9 层。

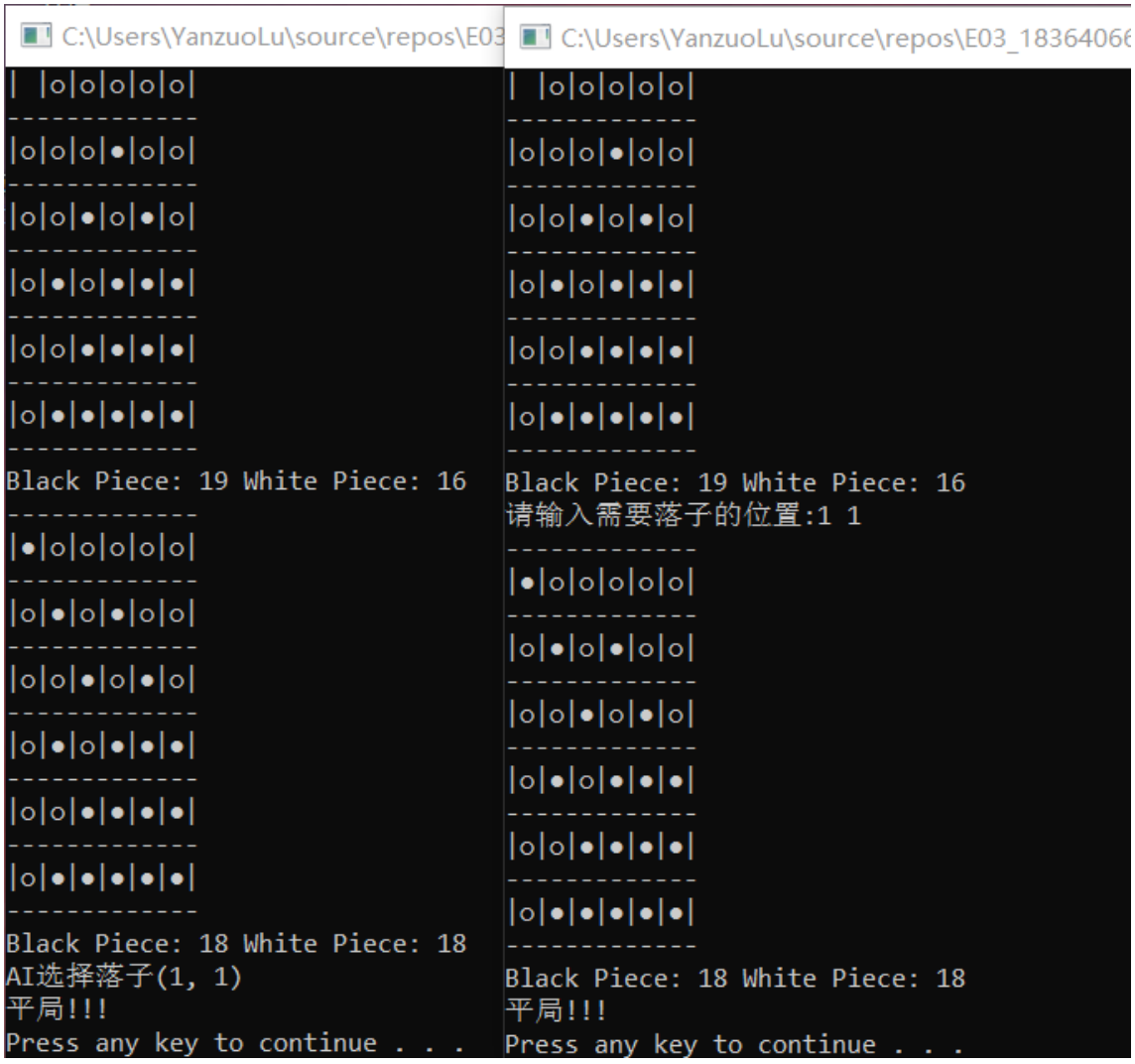

Figure 2: AI FIGHT