黑白棋设计报告

课程名称: 人工智能

姓　　名: 肖潼 周晟皓 赵宣栋

学　　院: 计算机学院

系: 计算机科学与技术

专　　业: 计算机科学与技术

学　　号: 3150103078 3150102103 3150104910

指导教师: 李玺

目录：

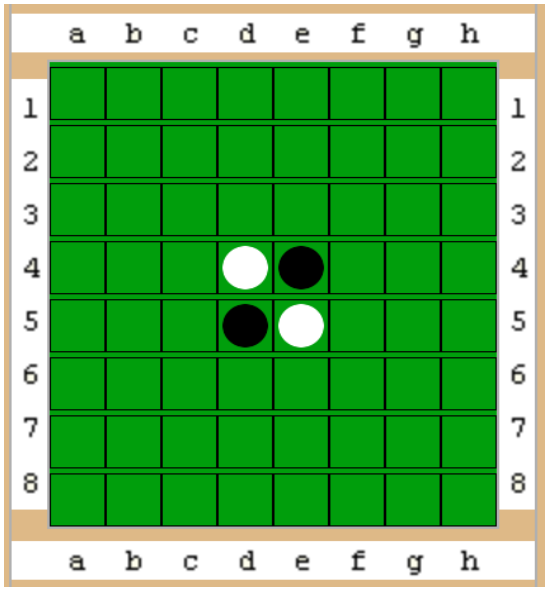
1. 引言

1.1. 简介

黑白棋（Reversi、Othello），也称翻转棋，是一个经典的策略性游戏。它使用 8×8 的棋盘，由双方轮流下棋，棋子分正反两面，分别为黑色和白色。下棋过程中如果一方的棋子把对方的棋子“夹住”了，就可以把被夹住的棋子翻转过来，成为自己的棋子。一步合法的着子，至少翻转一枚对方的棋子，如果某一方无子可下，就要停步一次，让对方先走。当双方都无子可下时，比赛结束，子多方为胜方。在 90 年代中期互联网开始普及，亦冒起了一些大型游戏网站。最先出现的是微软的 Microsoft Games 网站，被吸引进去下黑白棋的都是各国的好手，因为是外国网站，当时下棋的华人还是比较少。及后纷纷出现其他网站，中国具有代表性的网站有 Yahoo 游戏、中国游戏网、联众游戏。

1.2. 游戏规则

1. 棋局开始时黑棋位于 e4 和 d5，白棋位于 d4 和 e5，如图所示。



2. 黑方先行，双方交替下棋。
3. 一步合法的棋步包括：在一个空格新落下一个棋子，并且翻转对手一个或多个棋子。

4. 新落下的棋子与棋盘上已有的同色棋子间，对方被夹住的所有棋子都要翻转过来。可以是横着夹，竖着夹，或是斜着夹。夹住的位置上必须全部是对手的棋子，不能有空格。
5. 一步棋可以在数个（横向，纵向，对角线）方向上翻棋，任何被夹住的棋子都必须被翻转过来，棋手无权选择不去翻某个棋子。
6. 除非至少翻转了对手的一个棋子，否则就不能落子。如果一方没有合法棋步，也就是说不管他下到哪里，都不能至少翻转对手的一个棋子，那他这一轮只能弃权，而由他的对手继续落子直到他有合法棋步可下。
7. 如果一方至少有一步合法棋步可下，他就必须落子，不得弃权。
8. 棋局持续下去，直到棋盘填满或者双方都无合法棋步可下。
9. 如果某一方落子时间超过 1 分钟，则判该方失败。

2. 算法介绍

下棋的一般流程为，当下棋过程中的某一步有好几种走法可供选择时，棋手就要作出决策，选择对自己最有利的走法，那么如何判断是否有利呢？最简单的办法就是找一个的评估函数，这个函数对每一种局面给出一个估值，估值越高表明对自己越有利。用 V(S)表示对状态 S 的评估，如果评估函数足够好，能充分反应当前状态对自己的利弊，那么显然状态 S 下，应该采取的最佳行动。但是最佳行动很难直接得到，一般 1. 采用多步搜索，在搜索过程的终止状态再调用评估函数；2. 利用机器学习的方法学习一个较好的评估函数来指导下一步的动作。

在本次项目中，我们使用了蒙特卡罗搜索和 alpha-beta 剪枝两种策略，并基于 C++和 Python 实现，因为 alpha-beta 表现超出蒙特卡罗搜索，所以最终选用 alpha-beta 剪枝策略。

2.1. 整体环境

2.1.1. 整体环境模拟

我们设计了如下的数据结构来代表整个环境，环境包括一个棋盘，同时提供置位，获取当前状态，判断棋局状况等接口。

```
class Env
{
private:
    Board board;
public:
    Env();
    void setState(Board& board);
    Board getState();
    void step(Position p, int color);
    vector<Position> getLegalActions(int color);
    int count(int color);
    bool isTerminate();
    int winner();
    void display();
};
```

2.1.2. 棋子棋盘结构

同时还有棋子位置，棋盘的数据结构，其中涉及到多种符号的重载：

```
class Position
{
public:
    int x, y;
    double value;
    Position(int x, int y);
    Position(const Position& pos);
    Position& operator=(const Position& pos);
    bool operator < (const Position &pos) const;
};

class Board
{
```

```
private:
    char state[64];
public:
    Board();
    Board(const Board& board);
    char * operator[](int k);
    Board& operator=(const Board& board);
    BitBoard toBit();
    string toString();
    bool operator < (const Board &board) const;
};
```

对于当前棋局的判断，最为复杂的是判断什么地方是可以落子的。对于每个棋子的八个方向进行判断，然后找出所有符合条件的落子位置。

```
void addLegal(vector<Position> &list, Board &board, Position
&pos, int color, int dx, int dy) {
    int x, y;
    int state = 0;
    x = pos.x + dx;
    y = pos.y + dy;
    while (x >= 0 && x <= 7 && y >= 0 && y <= 7) {
        if (state == 0) {
            if (color == BLACK && board[x][y] != WHITE) return;
            if (color == WHITE && board[x][y] != BLACK) return;
            state = 1;
        } else {
            if (board[x][y] == EMPTY) {
                list.push_back(Position(x, y));
                return;
            }
            if (color == board[x][y]) return;
        }
        x += dx;
```

```
        y += dy;
    }
}

vector<Position> Env::getLegalActions(int color) {
    int i, j;
    vector<Position> list = vector<Position>();
    if (count(EMPTY) < 30) {
        for (i = 0; i < 8; i++)
            for (j = 0; j < 8; j++) {
                if (board[i][j] != color) continue;
                Position pos = Position(i, j);
                addLegal(list, board, pos, color, -1, -1);
                addLegal(list, board, pos, color, -1, 0);
                addLegal(list, board, pos, color, -1, 1);
                addLegal(list, board, pos, color, 0, -1);
                addLegal(list, board, pos, color, 0, 1);
                addLegal(list, board, pos, color, 1, -1);
                addLegal(list, board, pos, color, 1, 0);
                addLegal(list, board, pos, color, 1, 1);
            }
    } else {
        for (i = 0; i < 8; i++)
            for (j = 0; j < 8; j++) {
                if (board[i][j] != EMPTY) continue;
                Position pos = Position(i, j);
                if (_test(board, pos, color, -1, -1)
                    || _test(board, pos, color, -1, 0)
                    || _test(board, pos, color, -1, 1)
                    || _test(board, pos, color, 0, -1)
                    || _test(board, pos, color, 0, 1)
                    || _test(board, pos, color, 1, -1)
```

```
                || _test(board, pos, color, 1, 0)
                || _test(board, pos, color, 1, 1))
                list.push_back(pos);
        }
    }
    return list;
}
```

2.1.3. 程序整体逻辑

由于我们使用网络进行对决，所以程序整体的逻辑是 AI 相互对战的判断，先判断自己所处的角色，然后通过 alpha-beta 进行 7 层的搜索，在本机模拟当前棋局，将最优选择返回服务器。

```
void aivsai(bool first, int ai_seconds, int id) {
    vector<Position> list;
    vector<Position>::iterator iter;
    double value, bestValue;
    Position bestPos(-1, -1);
    Env env;
    Board board;
    Network net;
    int SESSION_ID = id;
    string player = net.create_session(id);
    cout << "Game begin " << player << endl;
    env.display();
    if (player == "B") {
        cout << "I am black" << endl;
        while (!env.isTerminate()) {
            if (net.get_trun(SESSION_ID) == player) {
                board = net.board_string(SESSION_ID);
                env.setState(board);
                env.display();
```

```
            cout << "Oppoent choose " << net.board_last(SESSION_ID) << endl;

            list = env.getLegalActions(BLACK);
            board = env.getState();
            bestValue = -1e9;
            if (list.size() != 0) {
                for (iter = list.begin(); iter != list.end(); iter++) {
                    env.setState(board);
                    env.step(*iter, BLACK);
                    value = AB(env, 7, WHITE, MIN, -1e9, 1e9);
                    if (value > bestValue) {
                        bestValue = value;
                        bestPos = *iter;
                    }
                }
                env.setState(board);
                env.step(bestPos, BLACK);
            }
            cout << "My AI choose: (" << bestPos.x << " " << bestPos.y << ")" << endl;
            net.get_move(bestPos.x, bestPos.y, SESSION_ID, player);
            env.display();
        } else {
            Sleep(2);
        }
    }
} else if (player == "W") {...//similar with "B"}
}
```

2.2. 蒙特卡罗搜索

黑白棋 AI 使用的主体思想是蒙特卡洛搜索树。蒙特卡洛树是基于统计和随机的广度搜索的思想。算法主要分为四个部分。

1. 首先是选择，从蒙特卡洛树的根节点开始，寻找可扩展的节点。一个节点是可扩展的需要满足两个条件，即该节点代表非终结状态，且该节点有未被访问过的子节点。
2. 然后是扩展，扩展即是根据当前状态下可采取的动作，将多个子节点加入 MCT（Monte-Carlo Tree，蒙特卡洛树）中。
3. 然后是模拟，模拟就是从第一步中选择的节点开始，采取一定策略行动，直至完成整场决策。
4. 最后是回溯，即将第三步模拟的结果的信息添加到 MCT 中从根节点到第一步选择节点的 path 上的所有节点信息中，作为之后的统计信息和判断依据。

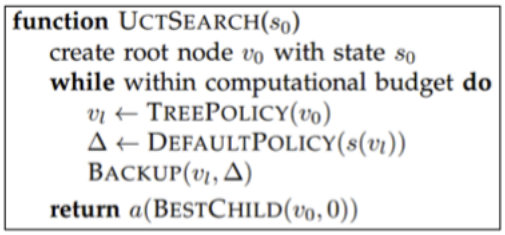
工程采用面向对象的思想，每个节点构造了一个对象，搜索节点的定义如下：

```
class UCTNode
{
public:
    UCTNode(Board& state, shared_ptr<UCTNode> father, int color);
    unsigned int getAccessTime() const { return _access_time; }
    int getReward() const { return _reward; }
    shared_ptr<UCTNode> getFather() { return _father; }
    int getColor() const { return _color; }
    void addAccessTime() { _access_time += 1; }
    void addReward(int delta) { _reward += delta; }
    void  addChild(Position  action,  shared_ptr<UCTNode>  child)
{ _children.insert(make_pair(action, child)); }
    int getN(Board state, Position action);
    double getValue(Position action, int role, double c);
private:
    Board _board;
```

```
    unsigned int _access_time;
    int _reward;
    int _color;
    shared_ptr<UCTNode> _father;
    map<Position, shared_ptr<UCTNode>> _children;
    map<pair<Board, Position>, int>N;
};
```

2.2.1. UCTSearch

UCTSearch 是 MCTS 算法的入口，伪代码如下：



在该函数中，我们限制了算法所需要的总时间，如果时间耗尽，如果到达最大层数，则退出。

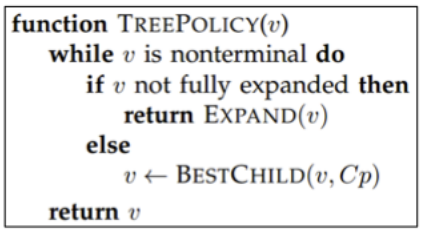
```
Position MCTS::search(Board &state) {
    Timer T;
    while (T.elapsed_seconds() < t) {
        simulate(state);
    }
    return getMove(state, _role);
}

void MCTS::simulate(Board &state) {
    env.setState(state);
    shared_ptr<UCTNode> end = simulateTree();
    // the tree is expanded until end, and env is changed to this
point as well
    int reward = simulateDefault();
```

```
    backup(end, reward);
}
```

2.2.2. 选择和拓展

在 treepolicy 函数中完成对于当前 MCT 的遍历，相关伪代码实现如下：

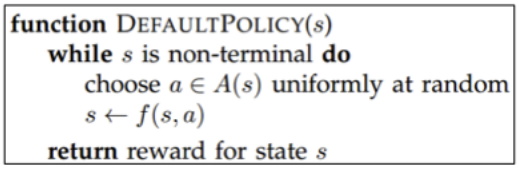


函数实现如下：

```
shared_ptr<UCTNode> MCTS::simulateTree() {
    int role_cur = _role;
    Board *state_cur;
    while (!env.isTerminate()) {
        state_cur = &(env.getState());
        if (tree.find(*state_cur) == tree.end()) { // new node
            shared_ptr<UCTNode> expand_child =
                  newNode(state_cur,      tree[*state_cur],      -
role_cur);
            return expand_child;
        }
        Position action = getMove(*state_cur, role_cur);
        env.step(action, role_cur);
        role_cur = -role_cur;
    }
    return tree[*state_cur];
}
```

2.2.3. 模拟

模拟棋局的进行，伪代码如下：




```
int MCTS::simulateDefault() {
    int role_cur = _role;
    while (!env.isTerminate()) {
        Board state = env.getState();
        Position action = DefaultPoicy(state, role_cur);  // Can
we avoid this construct every time
        role_cur = -role_cur;
    }
    return env.winner();
}
Position MCTS::DefaultPoicy(Board &state, int role, double T) {
    Env new_env;
    vector<Position> poses;
    vector<Position>::iterator iter;
    double value, bestValue;
    Position bestPos(-1, -1);
    bestValue = -1e9;
    new_env.setState(state);
    poses = new_env.getLegalActions(role);
    if (poses.size() == 0)
        return Position(-1, -1);
    for (iter = poses.begin(); iter != poses.end(); iter++) {
        new_env.setState(state);
        new_env.step(*iter, role);
        value = AB(new_env, 7, -role, MIN, -1e9, 1e9);
```

```
        if (value > bestValue) {
            bestValue = value;
            bestPos = *iter;
        }
    }
    return bestPos;
}
```

此处做了一个优化，最原始的 defaultpolicy 中，所有选定子之后的落子都是纯随机的。我们在接下来的探索中使用 alpha-beta 剪枝并且配合棋盘每个位置的权重进行优化，优化过后，选定子之后的落子使得选择角上的子的概率，选择边上的子的概率>选择中间的子的概率。

2.2.4. 回溯

对于模拟终局的结果，作为 reword 回溯更新每一个祖先节点，伪代码如下：

function BACKUP(v, Δ)
  while v is not null **do**
    $N(v) \leftarrow N(v) + 1$
    $Q(v) \leftarrow Q(v) + \Delta(v, p)$
    $v \leftarrow$ parent of v

具体实现：

```
void MCTS::backup(shared_ptr<UCTNode> v, int reward) {
    shared_ptr<UCTNode> tmp = v;
    while (tmp) {
        tmp->addAccessTime();
        tmp->addReward(reward);
        tmp = tmp->getFather();
    }
}
```

2.2.5. 估值最高的子节点

如果轮到黑棋走，就选对于黑棋有利的；如果轮到白棋走，就选对于黑棋最不利的。但不能太贪心，不能每次都只选择“最有利的/最不利的”，因为这会意味着搜索树的广度不够，容易忽略实际更好的选择。伪代码如下：

function BESTCHILD(v, c)

return $\underset{v' \in \text{children of } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2\ln N(v)}{N(v')}}$

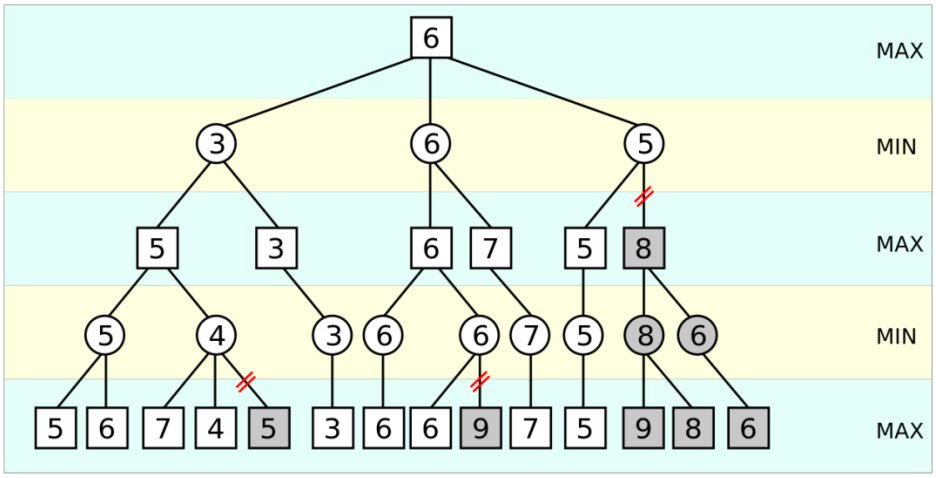
具体实现

```
Position MCTS::getMove(Board &state, int role, double c = 0.1)
{
    vector<Position> actions = env.getLegalActions(role);
    shared_ptr<UCTNode> node = tree[state];
    if (actions.size() == 0)
        return Position(-1, -1);
    Position best_action = actions[0];
    double best_value = node->getValue(best_action, role, c);
    double value;
    for (auto action: actions) {
        value = node->getValue(action, role, c);
        if (role == 1 && value >= best_value) {
            best_value = value;
            best_action = action;
        } else if (role == -1 && value <= best_value) {
            best_value = value;
            best_action = action;
        }
    }
    return best_action;
}
```

2.3. Alpha-Beta 剪枝

Alpha-Beta 剪枝是 MinMax 搜索的一种经典的优化算法，可以避免对大量局面的进一步搜索，从而减少搜索时间，达到更深的搜索深度。

如果上一层搜索层是 max，且最好结果为 α（估值最大），这一层搜索层是 min，且这个节点当前搜索的最好结果是 β（估值最小），如果 β 小于 α，意味着继续搜索不会更新上一层 max 的最好结果，所以可以停止搜索。同理，如果上一层搜索层是 min，且最好结果为 β（估值最小），这一层搜索层是 max，且这个节点当前搜索的最好结果是 α（估值最大），如果 α 大于 β，意味着继续搜索不会更新上一层 min 的最好结果，可以停止搜索。



Alpha-Beta 剪枝设定了 alpha 和 beta 两个值，初始时 alpha 值为负无穷，beta 值为正无穷。之后随着搜索的进行，会不断更新 alpha 和 beta，并将 alpha 和 beta 传递给下一层的搜索。Alpha-Beta 剪枝的伪代码如下：

```
01 function alphabeta(node, depth, α, β, maximizingPlayer)
is
02     if depth = 0 or node is a terminal node then
03         return the heuristic value of node
04     if maximizingPlayer then
05         v := -∞
06         for each child of node do
07             v := max(v, alphabeta(child, depth - 1, α,
β, FALSE))
```

```
08            α := max(α, v)
09            if β ≤ α then
10                break (* β cut-off *)
11        return v
12    else
13        v := +∞
14        for each child of node do
15            v := min(v, alphabeta(child, depth - 1, α,
β, TRUE))
16            β := min(β, v)
17            if β ≤ α then
18                break (* α cut-off *)
19        return v
```

我们对 alpha-beta 剪枝的具体实现如下：

```
double AB(Env& env, int depth, int color, int role, double a,
double b)
{
    vector<Position> list;
    vector<Position>::iterator iter;
    Board board;
    double v, temp;

    if (env.isTerminate() || depth == 0)
        return (dynamic_heuristic_evaluation_function(env, USER,
-USER));

    board = env.getState();
    list = env.getLegalActions(color);
    if (role == MIN)
    {
        v = 1e9;
        if (list.empty())
```

```
        {
            if (color == BLACK)
                temp = AB(env, depth-1, WHITE, MAX, a, b);
            else
                temp = AB(env, depth-1, BLACK, MAX, a, b);
            v = v<temp? v : temp;
            return v;
        }
        //sort(list.begin(), list.end(), comp1);
        for (iter = list.begin(); iter != list.end(); iter++)
        {
            env.setState(board);
            env.step(*iter, color);
            if (color == BLACK)
                temp = AB(env, depth-1, WHITE, MAX, a, b);
            else
                temp = AB(env, depth-1, BLACK, MAX, a, b);
            v = v<temp? v : temp;
            b = b<v? b : v;
            if (b <= a)
                break;
        }
        return v;
    }
    else
    {
        v = -1e9;
        if (list.empty())
        {
            if (color == BLACK)
                temp = AB(env, depth-1, WHITE, MIN, a, b);
            else
```

```
                temp = AB(env, depth-1, BLACK, MIN, a, b);
            v = v>temp? v : temp;
            return v;
        }
        //sort(list.begin(), list.end(), comp2);
        for (iter = list.begin(); iter != list.end(); iter++)
        {
            env.setState(board);
            env.step(*iter, color);
            if (color == BLACK)
                temp = AB(env, depth-1, WHITE, MIN, a, b);
            else
                temp = AB(env, depth-1, BLACK, MIN, a, b);
            v = v>temp? v : temp;
            a = a>v? a : v;
            if (a >= b)
                break;
        }
        return v;
    }
}
```

在黑白棋 AI 的设计中，由于黑白棋的搜索空间很大，无法在规定的时间内完成对所有可能性的 alpha-beta 剪枝，我们限制了 alpha-beta 剪枝的搜索深度，达到一定搜索深度以后我们将使用启发式的估值函数对棋局进行评估。如果启发式的估值函数足够好，则可以准确判断出搜索到棋局的胜负情况，则限定了深度的 alpha-beta 剪枝与搜到底的 alpha-beta 剪枝得到的结果将一致。

2.4. 估值函数

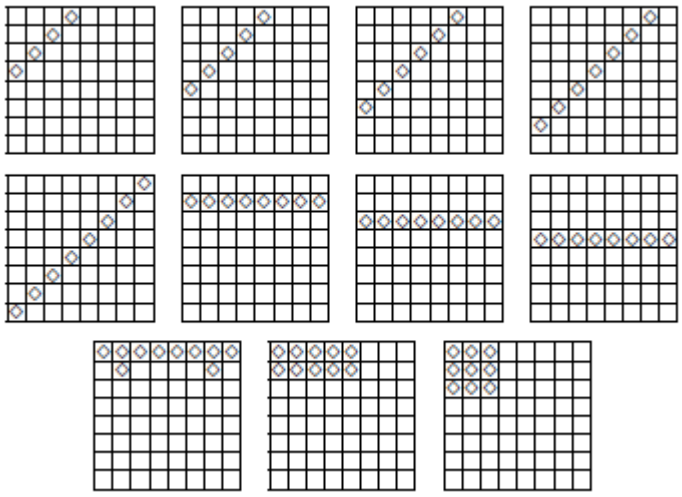
估值函数和搜索深度决定了 alpha-beta 剪枝算法结果的好坏，由于搜索的深度受到程序结构和计算机计算能力的限制，相比估值函数更难优化。估值函数

我们阅读了“From Simple Features to Sophisticated Evaluation

Functions”这篇论文，论文中介绍了如何自动地从最基本的棋局特征，比如某一个格子的状态，生成更高级的组合特征，比如若干格子的某一状态。最终获得一个状态的集合，并为每一种状态分配权重，估值函数的值就是当前局面出现的状态的权重之和。

论文中还提到了对不同对局阶段拟合不同的估值函数的做法。先对最后的若干步棋盘进行处理，使用 minmax 的搜索方式获得输赢的 ground truth，并训练出状态和相应的权重。之后再往前倒推若干步，使用相同深度的 minmax 进行搜索，并使用上一次的估值函数进行估值，得到新的 ground truth，并训练出状态和相应的权重，依此类推，直到获得了各个对局阶段的状态集合和相应的权重。

我们尝试实现了论文描述的状态生成和权重分配。我们首先从 botzone 网站上获取了 3 月份和 4 月份的机器对局数据，从而获得了几十万个局面。我们筛选出了最后几步的局面，并用 minmax 的搜索方式得出输赢的结果，并进行了标记。之后我们尝试构造了论文中所描述的状态，如下图所示：



但我们在实现的过程中发现所需要的内存空间太大，没有办法运行，于是去除了几个状态。但之后又发现判断棋局中是否出现了某个状态耗时太长，没有办法进行训练，于是不得不放弃这种做法。

之后我们在网上搜索找到了其他人以前做好的估值函数，于是我们对他们写好的估值函数进行了学习，这也是我们最终使用的估值函数。该估值函数的代码实现如下：

```
double dynamic_heuristic_evaluation_function(Env env, int
my_color, int opp_color) {
    Board grid = env.getState();
```

```
    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 X1[] = {-1, -1, 0, 1, 1, 1, 0, -1};
    int Y1[] = {0, 1, 1, 1, 0, -1, -1, -1};
    int V[8][8] =
           {{20, -3, 11, 8,  8,  11, -3, 20},
            {-3, -7, -4, 1,  1,  -4, -7, -3},
            {11, -4, 2,  2,  2,  2,  -4, 11},
            {8,  1,  2,  -3, -3, 2,  1,  8},
            {8,  1,  2,  -3, -3, 2,  1,  8},
            {11, -4, 2,  2,  2,  2,  -4, 11},
            {-3, -7, -4, 1,  1,  -4, -7, -3},
            {20, -3, 11, 8,  8,  11, -3, 20}};

// Piece difference, frontier disks and disk squares
    for (i = 0; i < 8; i++)
       for (j = 0; j < 8; j++) {
          if (grid[i][j] == my_color) {
             d += V[i][j];
             my_tiles++;
          } else if (grid[i][j] == opp_color) {
             d -= V[i][j];
             opp_tiles++;
          }
          if (grid[i][j] != EMPTY) {
             for (k = 0; k < 8; k++) {
                x = i + X1[k];
                y = j + Y1[k];
                if (x >= 0 && x < 8 && y >= 0 && y < 8 &&
grid[x][y] == EMPTY) {
```

```
                    if      (grid[i][j]      ==      my_color)
my_front_tiles++;
                    else opp_front_tiles++;
                    break;
                }
            }
        }
    }
    if (my_tiles > opp_tiles)
        p = (100.0 * my_tiles) / (my_tiles + opp_tiles);
    else if (my_tiles < opp_tiles)
        p = -(100.0 * opp_tiles) / (my_tiles + opp_tiles);
    else p = 0;

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

// Corner occupancy
    my_tiles = opp_tiles = 0;
    if (grid[0][0] == my_color) my_tiles++;
    else if (grid[0][0] == opp_color) opp_tiles++;
    if (grid[0][7] == my_color) my_tiles++;
    else if (grid[0][7] == opp_color) opp_tiles++;
    if (grid[7][0] == my_color) my_tiles++;
    else if (grid[7][0] == opp_color) opp_tiles++;
    if (grid[7][7] == my_color) my_tiles++;
    else if (grid[7][7] == opp_color) opp_tiles++;
```

```
    c = 25 * (my_tiles - opp_tiles);

// Corner closeness
    my_tiles = opp_tiles = 0;
    if (grid[0][0] == EMPTY) {
        if (grid[0][1] == my_color) my_tiles++;
        else if (grid[0][1] == opp_color) opp_tiles++;
        if (grid[1][1] == my_color) my_tiles++;
        else if (grid[1][1] == opp_color) opp_tiles++;
        if (grid[1][0] == my_color) my_tiles++;
        else if (grid[1][0] == opp_color) opp_tiles++;
    }
    if (grid[0][7] == EMPTY) {
        if (grid[0][6] == my_color) my_tiles++;
        else if (grid[0][6] == opp_color) opp_tiles++;
        if (grid[1][6] == my_color) my_tiles++;
        else if (grid[1][6] == opp_color) opp_tiles++;
        if (grid[1][7] == my_color) my_tiles++;
        else if (grid[1][7] == opp_color) opp_tiles++;
    }
    if (grid[7][0] == EMPTY) {
        if (grid[7][1] == my_color) my_tiles++;
        else if (grid[7][1] == opp_color) opp_tiles++;
        if (grid[6][1] == my_color) my_tiles++;
        else if (grid[6][1] == opp_color) opp_tiles++;
        if (grid[6][0] == my_color) my_tiles++;
        else if (grid[6][0] == opp_color) opp_tiles++;
    }
    if (grid[7][7] == EMPTY) {
        if (grid[6][7] == my_color) my_tiles++;
        else if (grid[6][7] == opp_color) opp_tiles++;
        if (grid[6][6] == my_color) my_tiles++;
```

```
        else if (grid[6][6] == opp_color) opp_tiles++;
        if (grid[7][6] == my_color) my_tiles++;
        else if (grid[7][6] == opp_color) opp_tiles++;
    }
    l = -12.5 * (my_tiles - opp_tiles);

// Mobility
    my_tiles = num_valid_moves(env, my_color);
    opp_tiles = num_valid_moves(env, opp_color);
    if (my_tiles > opp_tiles)
        m = (100.0 * my_tiles) / (my_tiles + opp_tiles);
    else if (my_tiles < opp_tiles)
        m = -(100.0 * opp_tiles) / (my_tiles + opp_tiles);
    else m = 0;

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

在这个估值函数中，综合考虑了双方拥有的棋子个数，双方棋子位置，双方可以下的棋子位置，双方对角的占领（下在角上）和封锁（角是空的，下在角的邻近格），双方稳定子的个数。这个估值函数对棋局的每一个位置都设定了权重，如果这个位置为我方的棋子则加上这个权重，如果有对方的棋子，则减去这个权重，从而可以综合考虑双方棋子的个数和位置。

在这个函数的最后，将这些特征的得分通过进行加权求和得到对局面的估值，加权求和时使用的权重是预先训练得到的。

相比论文中的那种特征选择的方法，这种方法特征数目要少的多（前者有十几万的特征），运算速度更快，也更符合 AI 对战的需求。这种估值函数中涉及到的特征都可以被论文中的特征所包括（论文中提到稳定子和机动性可以通过格子特征的组合表达），但更简单，不需要大量的计算和训练。这个估值函数相比论文中的估值函数，没有考虑棋局的不同阶段。它在棋局的所有阶段，使用的估值特征和权重相同，这就导致了它不能在对局的整个过程中很好的拟合各个局面的

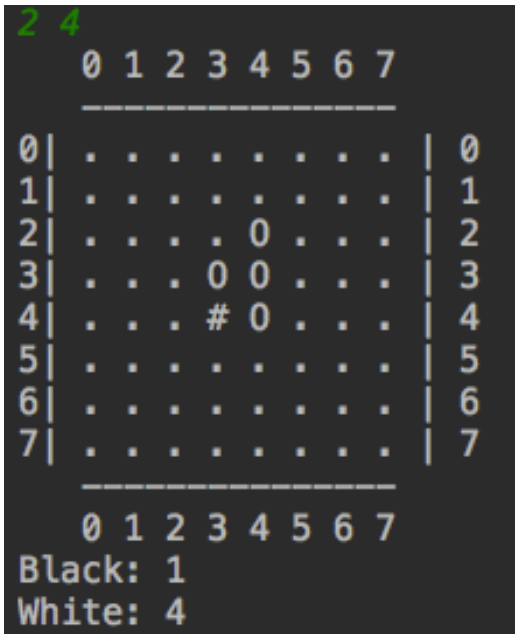
真实输赢情况，可能只在对局的某一阶段拟合效果较好。

2.5. 附加功能

2.5.1. 交互界面

棋盘一开始在 Python 下写了图形化界面，便于对算法进行比较和改进，但是因为可以使用网络来进行交互，所以之后便使用简单的命令行形式进行处理。

输入当前所要落子的位置，然后棋盘更新，打印棋盘。

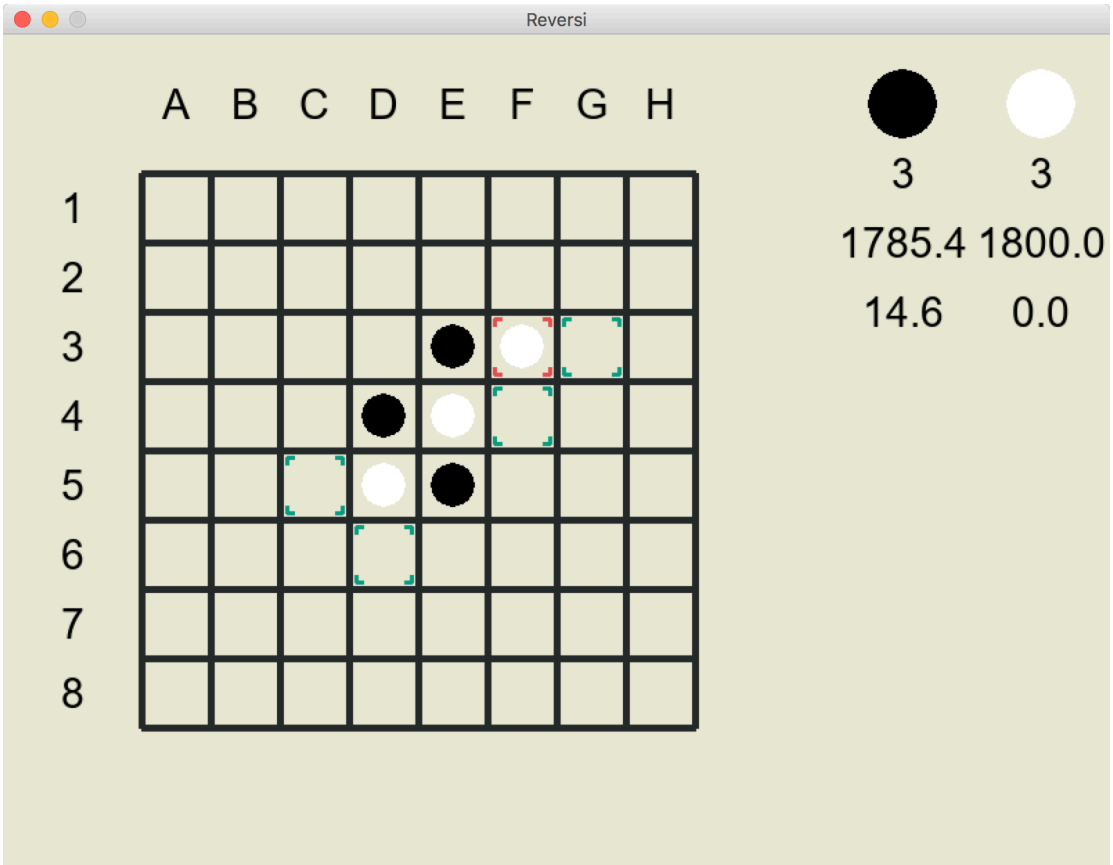



```
void Env::display() {
    int i, j;
    cout << "  0 1 2 3 4 5 6 7" << endl;
    cout << "  ---------------" << endl;
    for (i = 0; i < 8; i++) {
        cout << i << "| ";
        for (j = 0; j < 8; j++)
            if (board[i][j] == BLACK)
                cout << "# ";
            else if (board[i][j] == WHITE)
                cout << "0 ";
```

```
            else
                cout << ". ";
        cout << "| " << i << endl;
    }
    cout << "  ---------------" << endl;
    cout << "  0 1 2 3 4 5 6 7" << endl;
    cout << "Black: " << count(BLACK) << endl;
    cout << "White: " << count(WHITE) << endl;
}
```

之前参考开源代码中的实现制作的图形化界面：

在进入图形化界面之前选好先后手，然后展示棋局的图形界面。可以看到有落子的候选区域，然后对于当前棋局判断的评分。



2.5.2. 网络接口部分

助教老师辛苦实现了在线的测试，使得我们的对弈变得更加便捷，我们使用

HTTP get 和 post 请求来和服务器进行交互。

在 Python 中的实现较为简单，但在 C++中需要使用 curl 库来实现网络交互。

Network 对外接口：

```
class Network {
private:
    const std::string IP = "http://10.180.58.5:5000/";
public:
    std::string board_last(int num);
    std::string create_session(int num);
    Board board_string(int num);
    std::string  get_move(int  x,  int  y,  int  num,  std::string
color);
    std::string get_trun(int num);
    std::string network(std::string content);
};
```

Network 使用 curl_easy_setopt 设置对外的参数

```
std::string Network::network(std::string content) {
    std::string host = IP + content;
    std::stringstream out;
    while (true) {
        void *curl = curl_easy_init();
        // set URL
        curl_easy_setopt(curl, CURLOPT_URL, host.c_str());
        // set func and var
        curl_easy_setopt(curl,              CURLOPT_WRITEFUNCTION,
write_data);
        curl_easy_setopt(curl, CURLOPT_WRITEDATA, &out);
        // http get
        CURLcode res = curl_easy_perform(curl);
        if (res != CURLE_OK) {
            fprintf(stderr,  "curl_easy_perform()  failed:  %s\n",
```

```
curl_easy_strerror(res));
        }
        if (out.str().length() != 0) {
            break;
        }
        Sleep(0.1);
    }
    return out.str().substr(0, out.str().length() - 1);
}
```

例如创建新的对弈房间，直接改变 string 的值：

```
std::string Network::create_session(int num) {
    std::string content= "create_session/" + std::to_string(num);
    std::string res;
    res = network(content);
    return res;
}
```

3. 测试和改进

3.1. 测试结果

我们主要使用我们的算法和已有的 AI 进行对比，以及和同学进行友谊赛，因为网络接口已经关闭，所以我们只能展示当时比赛的一些记录：

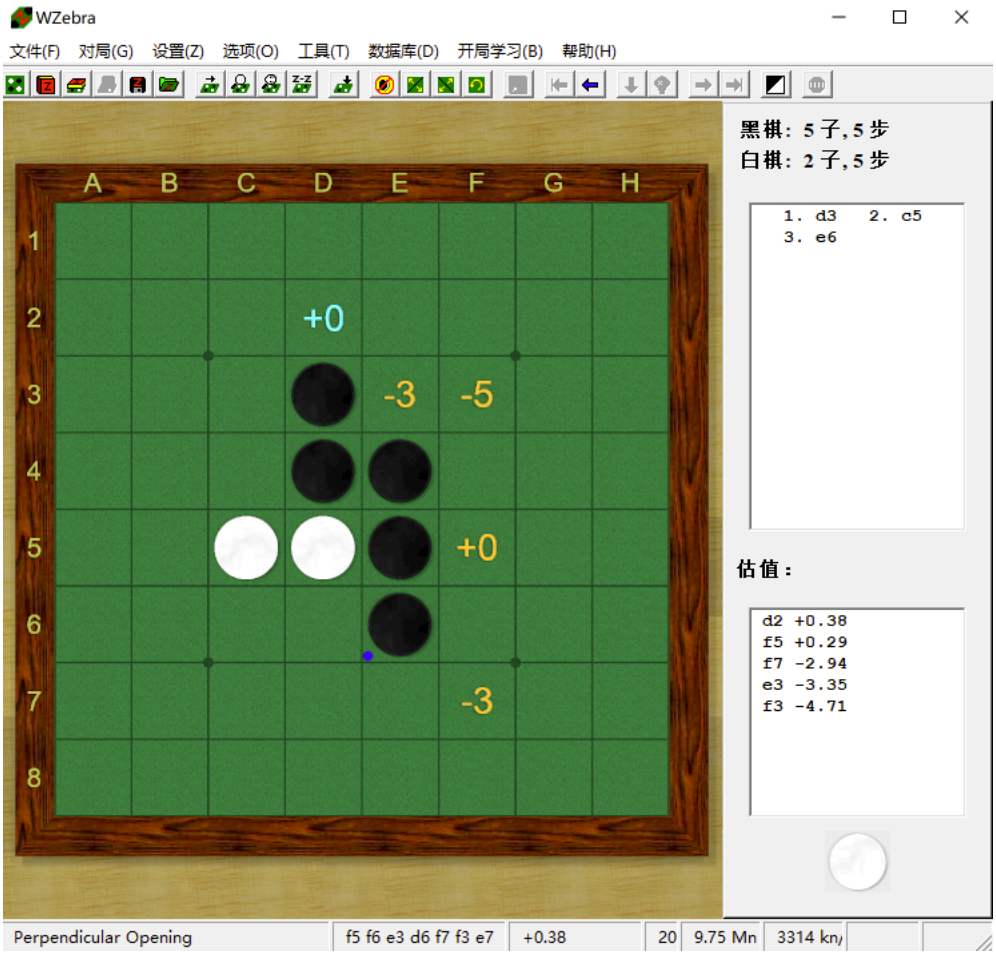
```
   0 1 2 3 4 5 6 7
  ---------------
0| . . . . . . . . | 0
1| . . . . . . . . | 1
2| . . . # O . . . | 2
3| . . . # O O . . | 3
4| . . # # O # . . | 4
5| . . . . . . . . | 5
```

```
6| . . . . . . . . | 6
7| . . . . . . . . | 7
  ---------------
  0 1 2 3 4 5 6 7
Black: 5
White: 4
oppoent choose 2.4.W
choose: (3 6)
  0 1 2 3 4 5 6 7
  ---------------
0| . . . . . . . . | 0
1| . . . . . . . . | 1
2| . . . # O . . . | 2
3| . . . # # # # . | 3
4| . . # # O # . . | 4
5| . . . . . . . . | 5
6| . . . . . . . . | 6
7| . . . . . . . . | 7
  ---------------
  0 1 2 3 4 5 6 7
Black: 8
White: 2
0.0.0.0.0.0.0.0;0.0.0.0.0.0.0.0;0.0.0.B.W.0.W.0;0.0.0.B.B.
W.B.0;0.0.B.B.W.B.0.0;0.0.0.W.0.0.0.0;0.0.0.0.0.0.0.0;0.0.
0.0.0.0.0.0.0
  0 1 2 3 4 5 6 7
  ---------------
0| . . . . . . . . | 0
1| . . . . . . . . | 1
2| . . . # O . O . | 2
3| . . . # # O # . | 3
4| . . # # O # . . | 4
```

```
5| . . . . . . . . | 5
6| . . . . . . . . | 6
7| . . . . . . . . | 7
  ----------------
  0 1 2 3 4 5 6 7
Black: 7
White: 4
oppoent choose 2.6.W
choose: (2 5)
  0 1 2 3 4 5 6 7
  ----------------
0| . . . . . . . . | 0
1| . . . . . . . . | 1
2| . . . # # # O . | 2
3| . . . # # # # . | 3
4| . . # # O # . . | 4
5| . . . . . . . . | 5
6| . . . . . . . . | 6
7| . . . . . . . . | 7
  ----------------
  0 1 2 3 4 5 6 7
Black: 10
White: 2
0.0.0.0.0.0.0.0;0.0.0.0.W.0.0.0;0.0.0.B.W.B.W.0;0.0.0.B.W.
B.B.0;0.0.B.B.W.B.0.0;0.0.0.W.0.0.0.0;0.0.0.0.0.0.0.0;0.0.
0.0.0.0.0.0.0
  0 1 2 3 4 5 6 7
  ----------------
0| . . . . . . . . | 0
1| . . . . O . . . | 1
2| . . . # O # O . | 2
3| . . . # O # # . | 3
```

```
4| . . # # O # . . | 4
5| . . . . . . . . | 5
6| . . . . . . . . | 6
7| . . . . . . . . | 7
   ---------------
   0 1 2 3 4 5 6 7
Black: 8
White: 5
oppoent choose 1.4.W
choose: (1 3)
   0 1 2 3 4 5 6 7
   ---------------
0| . . . . . . . . | 0
1| . . . # O . . . | 1
2| . . . # # # O . | 2
3| . . . # O # # . | 3
4| . . # # O # . . | 4
5| . . . . . . . . | 5
6| . . . . . . . . | 6
7| . . . . . . . . | 7
   ---------------
   0 1 2 3 4 5 6 7
Black: 10
White: 4
```

我们还与当前一款已经开发地很好的软件进行对比和优化，WZebra 是一个为黑白棋游戏而编写的自由软件，供练习和分析棋局。我们使用它的打谱模式发现，alpha-beta 剪枝总可以选择到相对最优的位置，可能此款软件也是基于此算法实现的。



我们在测试过程中发现，alpha-beta 剪枝对于大部分的 MCTS 算法都有优势，我们自己写的 MCTS 和助教老师的 MCTS 算法相比，我们的更胜一筹。但都被 7 层搜索的 alpha-beta 剪枝打败。

所以我们最后采用 alpha-beta 剪枝与班上同学进行比赛。

3.2. 反思改进

比赛之前我们对 MCTS 和 alpha-beta 剪枝两种方法都进行了测试。我们尝试修改了 alpha-beta 的搜索深度，搜索深度为 5 的时候，大部分情况可以在一秒内得到结果，有些情况需要等待十几秒，当搜索深度为 7 的时候，有些局面需要接近一分钟才能得到结果，但对战的效果要比搜索深度为 5 的时候好不少。搜索深度更深的时候等待的时间变得更长，对局太慢。

我们尝试对 alpha-beta 剪枝进行了改进，我们在搜索的时候按局面的估值函数先进行排序，min 节点按从小到大排，max 节点按从大到小排，这样就更有可能出现剪枝的情况，从而加快搜索的速度。但实验的结果显示排序以后的时间

变化不大，几乎没有变化，于是我们舍弃了这个优化。

我们也尝试使用了多线程来加速 alpha-beta 剪枝的搜索过程，但使用多线程之后搜索反而变慢了，于是我们舍弃了这个优化。

我们对 MCTS 的算法也进行了多种尝试。我们主要是针对 MCTS 算法的随机走子过程进行优化，由于 MCTS 得到的结果是与双方的下棋策略有关的，双方下棋策略与实际中下棋策略越接近，得到的结果就越有价值。我们尝试使用估值函数对每一次可以下的位置计算价值，并按 softmax 之后的概率进行随机走子，这样就可以让双方有更大的几率走有利于自己的棋子。我们也尝试了贪心策略，但最后决定将 MCTS 与 alpha-beta 剪枝的策略结合起来。结合之后的 MCTS 每秒的搜索次数减少了，但效果有所提升。

非常不幸，我们第一场就与本次黑白棋大赛第二名相遇，遗憾进行了一轮游。但是我们的友谊赛结果较好，对于纯 MCTS 算法有巨大的优势，基本可以稳赢，但对改进过的 MCTS 算法还是力不从心。

我们对比了自己的算法和对方的算法，发现主要的差距来自于搜索的次数。我们没有使用 bitboard 对落子进行加速，这导致我们的搜索次数少于对手，如果能够更快地模拟落子，使搜索的深度加深，对战的结果将更好。

4. 参考

1. From Simple Features to Sophisticated Evaluation Functions, Michael Buro, NEC Research Institute 4 Independence Way Princeton NJ 08540, USA
2. https://en.wikipedia.org/wiki/Monte_Carlo_tree_search
3. 28 天自制你的 AlphaGo（五）：蒙特卡洛树搜索（MCTS）基础。
4. 详解 AlphaGo 背后的力量：蒙特卡洛树搜索入门指南 —— 机器之心
5. 双人博弈问题中的蒙特卡洛树搜索算法的改进 季辉 ， 丁泽军