Mini project 3: Gomoku AI

110062334 周峻平

Basic Structure

```
struct Point {
    int x;
    int y;
    Point() : x(-1), y(-1) {}
    Point(int x, int y) : x(x), y(y) {}
    bool operator== (const Point& rhs) const {
        return (x == rhs.x && y == rhs.y);
    }
};

class Node {
public:
    int color;
    array<std::array<int, SIZE>, SIZE> board;

    Node(array<std::array<int, SIZE>, SIZE> bd, int player) {
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                this->board[i][j] = bd[i][j];
            }
        }
        color = player;
    }

    Node(Node& rhs) {
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                this->board[i][j] = rhs.board[i][j];
            }
        }
        color = rhs.color;
    }
```

Class Node: GetPossibleMoves(), evaluate(), PutChess()

State Value Function

```
enum Chess_State {
    o = 0,
    b1 = 1,
    b2 = 2,
    b3 = 3,
    b4 = 4,
    b5 = 5,
    w1 = 6,
    w2 = 7,
    w3 = 8,
    w4 = 9,
    w5 = 10
};

void StateType() {
    memset(state, 0, sizeof(state));

    state[1][0][0][0][0] = b1;
    state[0][1][0][0][0] = b1;
    state[0][0][1][0][0] = b1;
    state[0][0][0][1][0] = b1;
    state[0][0][0][0][1] = b1;

    state[1][1][0][0][0] = b2;
    state[1][0][1][0][0] = b2;
    state[1][0][0][1][0] = b2;
    state[1][0][0][0][1] = b2;
    state[0][1][1][0][0] = b2;
    state[0][1][0][1][0] = b2;
    state[0][1][0][0][1] = b2;
    state[0][0][1][1][0] = b2;
    state[0][0][1][0][1] = b2;
    state[0][0][0][1][1] = b2;

    state[0][0][1][1][1] = b3;
    state[0][1][0][1][1] = b3;
    state[0][1][1][0][1] = b3;
    state[0][1][1][1][0] = b3;
    state[1][0][0][1][1] = b3;
    state[1][0][1][0][1] = b3;
    state[1][0][1][1][0] = b3;
    state[1][1][0][0][1] = b3;
    state[1][1][0][1][0] = b3;
    state[1][1][1][0][0] = b3;

    state[1][1][1][1][0] = b4;
    state[1][1][1][0][1] = b4;
    state[1][1][0][1][1] = b4;
    state[1][0][1][1][1] = b4;
    state[0][1][1][1][1] = b4;

    state[1][1][1][1][1] = b5;
```

```
state[2][0][0][0][0] = w1;
state[0][2][0][0][0] = w1;
state[0][0][2][0][0] = w1;
state[0][0][0][2][0] = w1;
state[0][0][0][0][2] = w1;

state[2][2][0][0][0] = w2;
state[2][0][2][0][0] = w2;
state[2][0][0][2][0] = w2;
state[2][0][0][0][2] = w2;
state[0][2][2][0][0] = w2;
state[0][2][0][2][0] = w2;
state[0][2][0][0][2] = w2;
state[0][0][2][2][0] = w2;
state[0][0][2][0][2] = w2;
state[0][0][0][2][2] = w2;
```

```
state[0][0][2][2][2] = w3;
state[0][2][0][2][2] = w3;
state[0][2][2][0][2] = w3;
state[0][2][2][2][0] = w3;
state[2][0][0][2][2] = w3;
state[2][0][2][0][2] = w3;
state[2][0][2][2][0] = w3;
state[2][2][0][0][2] = w3;
state[2][2][0][2][0] = w3;
state[2][2][2][0][0] = w3;

state[2][2][2][2][0] = w4;
state[2][2][2][0][2] = w4;
state[2][2][0][2][2] = w4;
state[2][0][2][2][2] = w4;
state[0][2][2][2][2] = w4;

state[2][2][2][2][2] = w5;
```

```
int evaluate_me(int type, int player) {
    int weight[11] = { 0, 1, 20, 400, 50000, 1000000, 2, 500, 8000, 100000, 10000000 };
    if (player == 1) {
        for (int i = 1; i <= 5; i++) {
            if (type == i) {
                return weight[i];
            }
        }

    }
    else if (player == 2) {
        for (int i = 6; i <= 10; i++) {
            if (type == i) {
                return weight[i - 5];
            }
        }

    }
    return 0;
}
```

```
int evaluate_opponent(int type, int player) {
    int weight[11] = { 0, 1, 20, 400, 50000, 1000000, 2, 500, 8000, 100000, 10000000 };
    if (player == 1) {
        for (int i = 6; i <= 10; i++) {
            if (type == i) {
                return weight[i];
            }
        }

    }
    else if (player == 2) {
        for (int i = 1; i <= 5; i++) {
            if (type == i) {
                return weight[i + 5];
            }
        }

    }
    return 0;
}
```

```
int evaluate(int player) {
    int val = 0;
    for (int i = 0; i < 15; i++) {
        for (int j = 0; j < 15; j++) {
            // row
            if (j + 4 < 15) {
                int type = state[board[i][j]][board[i][j + 1]][board[i][j + 2]][board[i][j + 3]][board[i][j + 4]];
                val += (evaluate_me(type, player) - evaluate_opponent(type, player));
            }
            //col
            if (i + 4 < 15) {
                int type = state[board[i][j]][board[i + 1][j]][board[i + 2][j]][board[i + 3][j]][board[i + 4][j]];
                val += (evaluate_me(type, player) - evaluate_opponent(type, player));
            }
            //diag
            if (i + 4 < 15 && j + 4 < 15) {
                int type = state[board[i][j]][board[i + 1][j + 1]][board[i + 2][j + 2]][board[i + 3][j + 3]][board[i + 4][j + 4]];
                val += (evaluate_me(type, player) - evaluate_opponent(type, player));
            }
            //diag
            if (i + 4 < 15 && j - 4 >= 0) {
                int type = state[board[i][j]][board[i + 1][j - 1]][board[i + 2][j - 2]][board[i + 3][j - 3]][board[i + 4][j - 4]];
                val += (evaluate_me(type, player) - evaluate_opponent(type, player));
            }
        }
    }

    return val;
}
```

main & write_valid_spot function

```
int main(int, char** argv) {
    std::ifstream fin(argv[1]);
    std::ofstream fout(argv[2]);
    read_board(fin);

    StateType();
    Node origin(board, player);
    write_valid_spot(fout, origin);

    fin.close();
    fout.close();
    return 0;
}
void write_valid_spot(std::ofstream& fout, Node& node) {
    int x, y;
    auto target = AlphaBeta(node, 3, INT_MIN, INT_MAX, true);
    //auto target = MiniMax(node, 3, true);
    x = target.second.x;
    y = target.second.y;
    fout << x << " " << y << "\n";
    fout.flush();

    return;
}
```

PutChess

```
void PutChess(Point p, bool me) {
    if (me) {
        board[p.x][p.y] = this->color;
    }
    else {
        board[p.x][p.y] = (3 - this->color);
    }
}
```

GetPossibleMoves

```
vector<Point> GetPossibleMoves() {
    bool HasChess = false, free[15][15];
    memset(free, false, sizeof(free));
    for (int i = 0; i < SIZE; i++) {
        for (int j = 0; j < SIZE; j++) {
            if (this->board[i][j] == EMPTY) {
                continue;
            }
            HasChess = true;
            int x1 = max(0, i - radius), x2 = min(14, i + radius);
            int y1 = max(0, j - radius), y2 = min(14, j + radius);
            for (int x = x1; x <= x2; x++) {
                for (int y = y1; y <= y2; y++) {
                    if (this->board[x][y] == EMPTY) {
                        free[x][y] = true;
                    }
                }
            }
        }
    }

    vector<Point> pos_vec;
    if (!HasChess) {
        pos_vec.emplace_back(Point(7, 7));
    }
    else {
        for (int i = 0; i < SIZE; i++) {
            for (int j = 0; j < SIZE; j++) {
                if (free[i][j]) {
                    pos_vec.emplace_back(Point(i, j));
                }
            }
        }
    }

    return pos_vec;
}
```

MiniMax

```
pair<int, Point> MiniMax(Node node, int depth, bool maximize) {
    if (depth == 0) {
        return make_pair(node.evaluate(node.color), Point(-1, -1));
    }
    if (maximize) {
        int max_val = INT_MIN;
        Point p;
        vector<Point> PossibleMoves = node.GetPossibleMoves();
        for (auto pos : PossibleMoves) {
            Node child(node);
            child.PutChess(pos, true);
            auto it = MiniMax(child, depth - 1, !maximize);
            if (it.first > max_val) {
                max_val = it.first;
                p = pos;
            }
        }
        return make_pair(max_val, p);
    }
    else { // minimize
        int min_val = INT_MAX;
        Point p;
        vector<Point> PossibleMoves = node.GetPossibleMoves();
        for (auto pos : PossibleMoves) {
            Node child(node);
            child.PutChess(pos, false);
            auto it = MiniMax(child, depth - 1, !maximize);
            if (it.first < min_val) {
                min_val = it.first;
                p = pos;
            }
        }
        return make_pair(min_val, p);
    }
}
```

AlphaBeta

```
pair<int, Point> AlphaBeta(Node node, int depth, int alpha, int beta, bool maximize) {
    if (depth == 0) {
        return make_pair(node.evaluate(node.color), Point(-1, -1));
    }
    if (maximize) {
        int max_val = INT_MIN;
        Point p;
        vector<Point> PossibleMoves = node.GetPossibleMoves();
        for (auto pos : PossibleMoves) {
            Node child(node);
            child.PutChess(pos, true);
            auto it = AlphaBeta(child, depth - 1, alpha, beta, !maximize);
            if (it.first > max_val) {
                max_val = it.first;
                p = pos;
            }
            alpha = max(alpha, max_val);
            if (alpha >= beta) {
                break;
            }
        }
        return make_pair(max_val, p);
    }

    else {
         int min_val = INT_MAX;
         Point p;
         vector<Point> PossibleMoves = node.GetPossibleMoves();
         for (auto pos : PossibleMoves) {
             Node child(node);
             child.PutChess(pos, false);
             auto it = AlphaBeta(child, depth - 1, alpha, beta, !maximize);
             if (it.first < min_val) {
                 min_val = it.first;
                 p = pos;
             }
             beta = min(beta, min_val);
             if (beta <= alpha) {
                 break;
             }
         }
         return make_pair(min_val, p);
    }
}
```

Version Control