Gomoku

Ver 2.0

09016401
顾婷瑄
Botzone id: Gutingxuan

1. Old versions review

Gomoku1.0 is simply based on an evaluation function and can not foresee future moves. Although it can comprehensively consider every available position and make the best move, it is not very competitive.

2. Literatures I read

[1] 王志水. 基于搜索算法的人工智能在五子棋博弈中的应用研究[D].山东：中国石油大学, 2006

[2] Blog: 五子棋基本棋型@我是老邱

五子棋估值算法@maxuewei2

3. Current idea of design

i. Essence

A function based on a new evaluation function and a MaxMin algorithm.

ii. Origin of the idea

The paper I’ ve read.

iii. Expectation

The function can make tentative moves, evaluate the value of each state on both sides and then make the best choice.

iv. Implementation

A new evaluation function of the board and a MaxMin algorithm.

v. Advantages of the design

With the attempt to make possible moves, it can take more moves into consideration and thus can make a better choice in comparison with merely evaluating the value of current available positions.

vi. Weakness

With the increase of search depth and with the increase number of made moves, the time spent on calculating will increase exponentially so it’ s quite slow when in practice. And since I didn’ t use any pruning algorithm, the deepest depth it can go with is 2. A depth of 3 will take a couple of minutes to make the move and I can’ t tolerate to test it.

vii. Improvement directions

I will try to optimize the searching algorithm in order to decrease the time spent on searching. And if possible, I will try to increase the searching depth. Perhaps I will use the alpha-beta pruning algorithm.

4. Evaluation of the performance

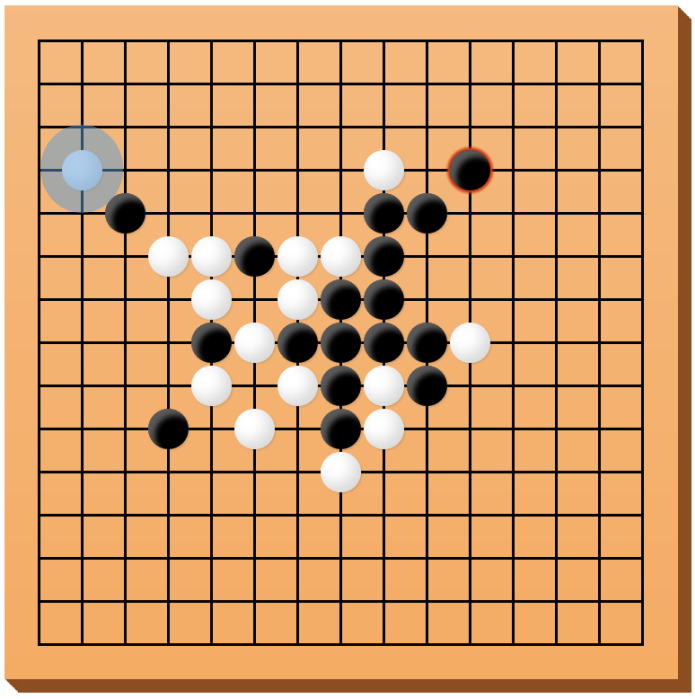
There are still some flaws in this version of Gomoku. Although it uses a more delicate algorithm, it didn’ t perform as well as I expected. For example, it sometimes ignores the obvious winning move of the opponent and then lose the game with no doubt.

Here are some recorded games with my Gomoku program on Botzone.org. As you can see, there still are some flaws in it. But from my perspective, it is better than the previous version.

Gutingxuan: me

Tttttttest: computer

Depth=1

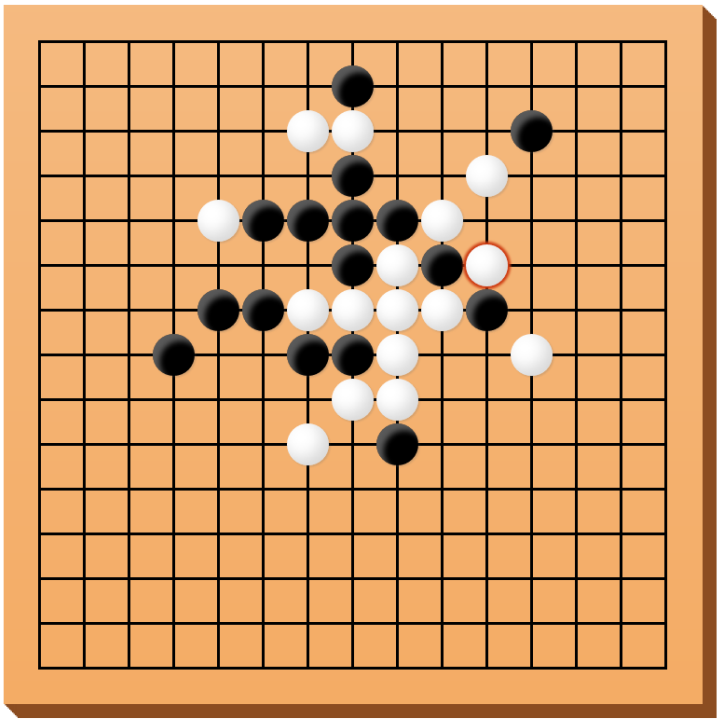


回合：34

黑方：Gutingxuan

白方：tttttttest【13】

I notice that when the computer finds that there is an inevitable failure, it just chooses the first vacant position it traversed to make the move. The stone circled is an example.



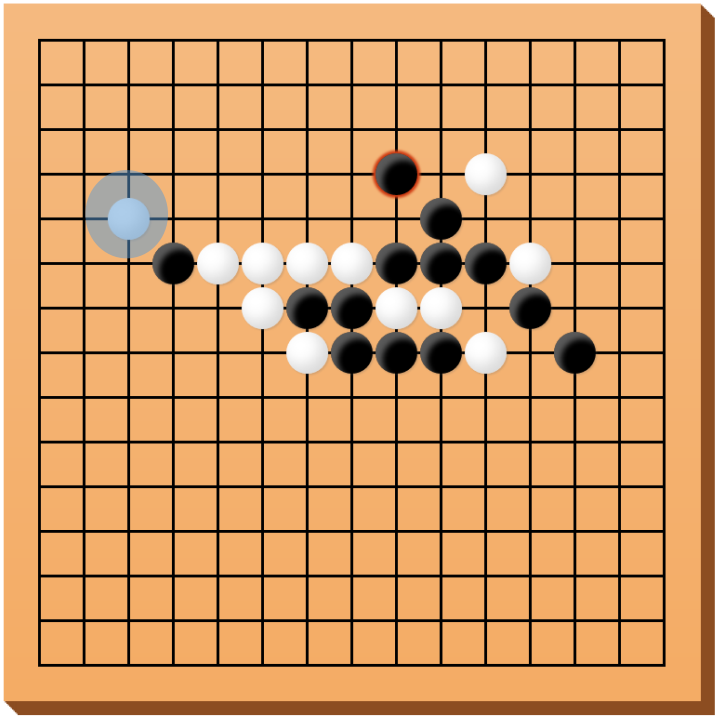
回合：33

黑方：ttttttttest【13】

白方：Gutingxuan

When the searching depth is 1, sometimes it gets too aggressive and neglects the winning move of the other player so it loses at the end.

Depth=2

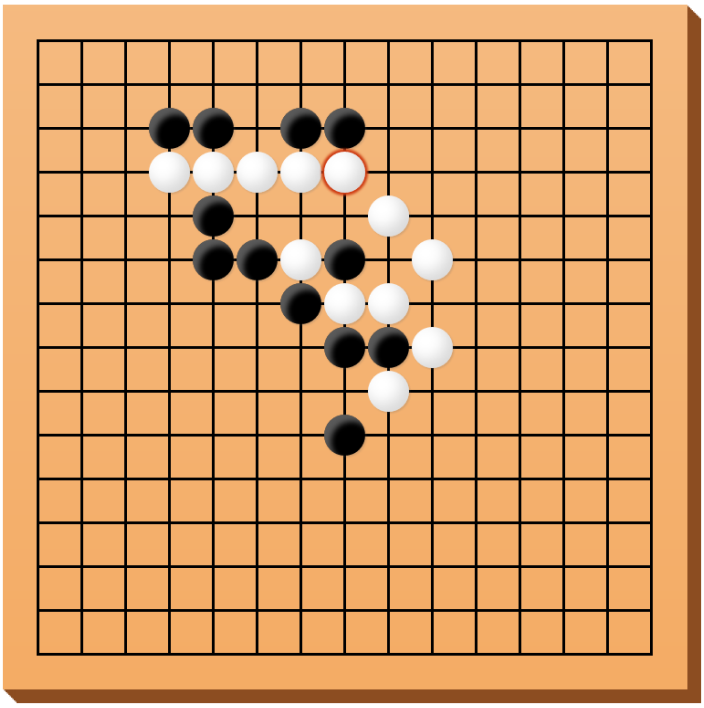


回合：26

黑方：你

白方：[Gutingxuan]ttttttttest

When the depth is 2, the computer keeps sticking to the player which is quite annoying. The stone circled is an unreasonable move. I will try to figure out what causes this kind of mistake.



回合：25

黑方：[Gutingxuan]ttttttttest

白方：你

Again, too aggressive. It forgets that the white stone wins ahead of it.

5. Code structure and detailed implementations

i. The new evaluation function

This function returns the current value of the board. It calculates the max position value of the player and the max position value of the computer and then makes a subtraction. The calculation of position value is based on former defense and attack evaluation functions.

Board value= computer value – player value

This function serves for the MaxMin algorithm.

```
int ChessBoard::evaluate()
{
    InitScore();
    int maxA = 0, maxD = 0; //最大进攻值和防守值
    int valueAI = 0, valueHM = 0;
    int move2[225][2];
    int k = 0;
    for (int i = 0; i < 225; i++)
    {
        move2[i][0] = -1;
        move2[i][1] = -1;
    }
    for (int x = 0; x < 15; x++)
    {
        for (int y = 0; y < 15; y++)
        {
            if (board[x][y] == -1)
            {
                for (int i = 1; i <= 8; i++)
                {
                    if (getLine(x, y, i, 1) == 0 || getLine(x, y, i, 1) == 1
                        || getLine(x, y, i, 2) == 0 || getLine(x, y, i, 2) == 1)
                    {
                        move2[k][0] = x;
                        move2[k][1] = y;
                        k++;
                        break;
                    }
                }
            }
        }
    }
```

```
}
for (int i = 0; i<k; i++)
{
    if (EvaluateA(move2[i][0], move2[i][1], choice) > maxA)
    {
        maxA = EvaluateA(move2[i][0], move2[i][1], choice);
    }

    if (EvaluateD(move2[i][0], move2[i][1], 1 - choice) > maxD)
    {
        maxD = EvaluateD(move2[i][0], move2[i][1], 1 - choice);
    }
}
if (maxA >= maxD)
{
    valueAI = maxA;
}
else if (maxD > maxA)
{
    valueAI = maxD;
}
maxA = 0;
maxD = 0;
InitScore();
for (int i = 0; i<k; i++)
{
    if (EvaluateA(move2[i][0], move2[i][1], 1 - choice) > maxA)
    {
        maxA = EvaluateA(move2[i][0], move2[i][1], 1 - choice);
    }

    if (EvaluateD(move2[i][0], move2[i][1], choice) > maxD)
    {
        maxD = EvaluateD(move2[i][0], move2[i][1], choice);
    }
}
if (maxA >= maxD)
{
    valueHM = maxA;
}
else if (maxD > maxA)
{
    valueHM = maxD;
}
return valueAI - valueHM;
```

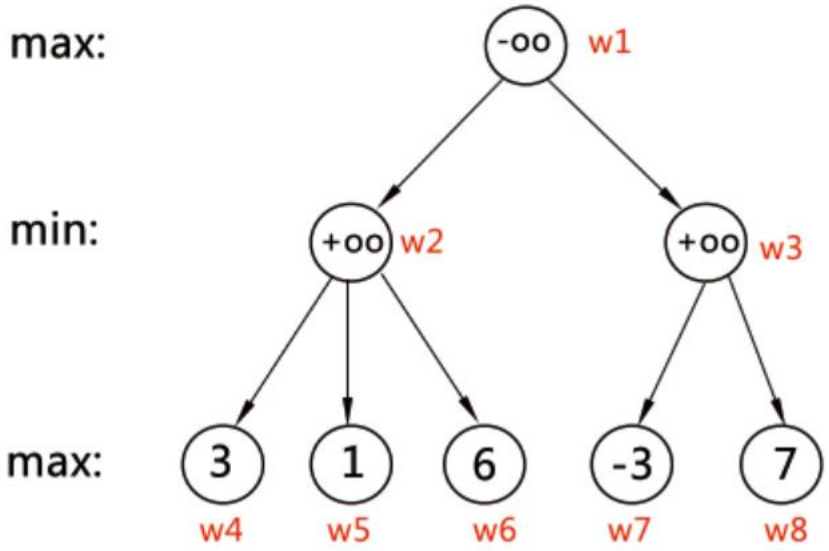
ii. The MaxMin algorithm

The evaluation function: We use this to evaluate the value of each state.

The board value = computer value – player value

Max state: This is the state that computer is the one to make the move. There are some possible moves and each of them will lead to a sub-state. Since computer is going to make the move, it will try to maximize the board value so this is called the Max state.

Min state: This is the state that opponent is going to make the move. The move it makes will also lead to different sub-states. From it' s perspective, it will try to minimize the board value so this is called the Min state.



Take the game tree above as an example.

Assume the first player is A, the second player is B, then A is a max state and B is a min state. A has 2 choices at first, w2 and w3. Whether it goes with w2 or w3 depends on the evaluation function. Since it is a max state, so it will choose the bigger one between f(w2) and f(w3).

Here are the searching paths:

w1->w2->w4: w2 is now 3

w1->w2->w5: w5 is less than w4 and w2 is a min state so it will go with w5. w2 is now 1.

w1->w2->w6: w6 is 6 so we don' t take it.

Finally, w2 is 1.

w1->w3->w7: w3 is now -3

w1->w3->w8: w8 is 7 so we don’ t take it.

Finally, w3 is -3.

Since w1 is a max state, so its next move will be w2.

Pseudocode from Wikipedia:

```
function minimax(node, depth)
    if node is a terminal node or depth = 0
        return the heuristic value of node
    if the adversary is to play at node
        let α := +∞
        foreach child of node
            α := min(α, minimax(child, depth-1))
    else {we are to play at node}
        let α := -∞
        foreach child of node
            α := max(α, minimax(child, depth-1))
    return α
```

Pseudocode from a blog: This helps me better understand the algorithm

```
// player = 1 表示轮到己方， player = 0 表示轮到对方
// cur_node 表示当前局面(结点)
maxmin(player, cur_node)
{
    if 达到终结局面
        return 该局面结点的估价值 f
    end
    if player == 1 // 轮到己方走
        best = -oo // 己方估价值初始化为 -oo
        for 每一种走法 do
            new_node = get_next(cur_node) // 遍历当前局面 cur_node 的所有子局面
            val = maxmin(player^1, new_node); // 把新产生的局面交给对方，对方返回一个该局面的估价值
            if val > best
                best = val;
            end
        end
        return best;
    else // 轮到对方走
        best = +oo // 对方估价值初始化为 +oo
        for 每一种走法 do
            new_node = get_next(cur_node) // 遍历当前局面 cur_node 的所有子局面
            val = maxmin(player^1, new_node); // 把新产生的局面交给对方，对方返回一个该局面的估价值
            if val < best
                best = val;
            end
        end
        return best;
    end
}
```

My MaxMin function:

The generator function part:

These lines basically form a generator function which collects available positions that have neighbors, which means there is a previous stone nearby. Otherwise there will be too many available positions that need to be traversed and this will increase the searching time.

```
int ChessBoard::MaxMin(int player, int deep)
{
    if (Winning() != -1 || deep <= 0)
        return evaluate();
    int best=0, val=0;
    int move1[225][2];
    int k = 0;
    for (int i = 0; i < 225; i++)
    {
        move1[i][0] = -1;
        move1[i][1] = -1;
    }
    for (int x = 0; x < 15; x++)
    {
        for (int y = 0; y < 15; y++)
        {
            if (board[x][y] == -1)
            {
                for (int i = 1; i <= 8; i++)
                {
                    if (getLine(x, y, i, 1) == 0 || getLine(x, y, i, 1) == 1)
                    {
                        move1[k][0] = x;
                        move1[k][1] = y;
                        k++;
                        break;
                    }
                }
            }
        }
    }
```

The body part of the algorithm:

The computer will make a move so as to simulate a sub-state when traversing sub-states. After traversing this sub-state, it will withdraw this move.

```
if (player == choice)
{
    best = -20000000;
    for (int i = 0; i<k; i++)
    {
        board[move1[i][0]][move1[i][1]] = choice;
        val = MaxMin(1-player,deep - 1);
        board[move1[i][0]][move1[i][1]] = -1;
        if (val > best)
        {
            best = val;
            if (deep == 2)
            {
                lastx = move1[i][0];
                lasty = move1[i][1];
            }
        }
    }
}
else
{
    best = 20000000;
    for (int i = 0; i<k; i++)
    {
        board[move1[i][0]][move1[i][1]] = 1 - choice;
        val = MaxMin(player, deep - 1);
        board[move1[i][0]][move1[i][1]] = -1;
        if (val < best)
        {
            best = val;
        }
    }
}
return best;
```

```
void ChessBoard::UseMaxMin()
{
    MaxMin(choice, 2);
    board[lastx][lasty] = choice;
}
```

Complexity analysis:

If there are (n) sub-states for each move, considering the depth as (d), the worst case would be O(n^b).