Report II

09016428 SongZixing

1 Review

In the last report, I mainly cover the basic algorithm that is and will be implemented in my Gomoku AI. And I especially emphasized two main heuristic search algorithms, which are MiniMax Algorithm and Negamax Algorithm. However I only implemented the first one at that time due to the limited time I have. During the past two weeks, I successfully managed to put the second one into practice still based on the alpha-beta pruning algorithm. In the meanwhile, I compared these two algorithms, which refers to MiniMax Algorithm and Negamax Algorithm to vertify some of theoritical ideas so that the future optimization algorithms can be based on a safe and solid ground.

2 Negamax Algorithm

To make it easily understood, I still attach some introductions to the Negamax Algorithm, which can actually be considered as a changed version of MiniMax Algorithm with slightly less consumption of space and equal consumption of time.

2.1 Overview

Negamax search is a variant form of Minimax search that relies on the zero-sum property of a two-player game.

This algorithm relies on following fact.

$$max(a, b) = -min(-a, -b)$$

This obvious equation can simplify the implementation of the Minimax algorithm. More precisely, the value of a position to player A in such a game is the negation of the value to player B. Thus, the player on move looks for a move that maximizes the negation of the value resulting from the move: this successor position must by definition have been valued by the opponent. The reasoning of the previous sentence works regardless of whether A or B is on move. This means that a single procedure can be used to value both positions. This is a coding simplification over minimax, which requires that A selects the move with the maximum-valued successor while B selects the move with the minimum-valued successor.

2.2 Pseudocode

```
01 function negamax(node, depth, color)
02     if depth = 0 or node is a terminal node
03         return color * the heuristic value of node

04     bestValue := −∞
05     foreach child of node
06         v := −negamax(child, depth − 1, −color)
07         bestValue := max( bestValue, v )
08     return bestValue
```

```
//Initial call for Player A's root node
rootNegamaxValue := negamax( rootNode, depth, 1)
rootMinimaxValue := rootNegamaxValue
```

```
//Initial call for Player B's root node
rootNegamaxValue := negamax( rootNode, depth, -1)
rootMinimaxValue := -rootNegamaxValue
```

2.3 Analysis

The root node inherits its score from one of its immediate child nodes. The child node that ultimately sets the root node's best score also represents the best move to play. Although the negamax function shown only returns the node's best score as *bestValue*, practical negamax implementations may also retain and return both best move and best score for the root node. Assuming basic negamax, only the node's best score is essential with non-root nodes. And the node's best move isn't necessary to retain nor return for those nodes.

2.4 Negmax with alpha-beta pruning(Pseudocode)

Still, like MiniMax, I implement Negmax Algorithm with alpha-beta pruning. The following is pseudocode for it.

```
function negamax(node, depth, α, β, color)
02     if depth = 0 or node is a terminal node
03         return color * the heuristic value of node

04     childNodes := GenerateMoves(node)
05     childNodes := OrderMoves(childNodes)
06     bestValue := −∞
07     foreach child in childNodes
08         v := −negamax(child, depth − 1, −β, −α, −color)
09         bestValue := max( bestValue, v )
10         α := max( α, v )
11         if α ≥ β
12             break
```

```
13      return bestValue

//Initial call for Player A's root node
rootNegamaxValue := negamax( rootNode, depth, −∞, +∞, 1)
```

3 Implement

Based on the pseudocode for Negmax Algorithm with alpha-beta pruning, I implement it in C++ as follows. Here is the source code.

```
MoveValue Gomoku::negaMax(int depth, double alpha, double beta, int color)
{
    if (depth == 0 /*|| isGameOver()*/) {
            double res = EvaluateState();
            return MoveValue(res);
    }
    MoveValue bestMove(-DBL_MAX);

    vector<MOVE>LegalMoves = GenerateLegalMoves();
    vector<MOVE>::iterator movesIterator = LegalMoves.begin();
    while (movesIterator != LegalMoves.end()) {
        MOVE currentMove = *movesIterator;
        MOVE newMove = currentMove;
        movesIterator++;
        applyMove(currentMove);
        double val = -(negaMax(depth - 1, -beta,-alpha,color%2+1).returnValue);
        Grid[currentMove.x][currentMove.y] = 0;
        CurrentColor = CurrentColor % 2 + 1;
        //UnmakeMove();
        if (val > bestMove.returnValue) { bestMove.returnValue = val;
bestMove.returnMove = currentMove; }// bestValue := max( bestValue, v )
        if (val > alpha) { alpha = val; }//α := max( α, v )
        if (alpha >= beta) break;//pruning
    }
    return bestMove;
}
```

4 Performance Comparison

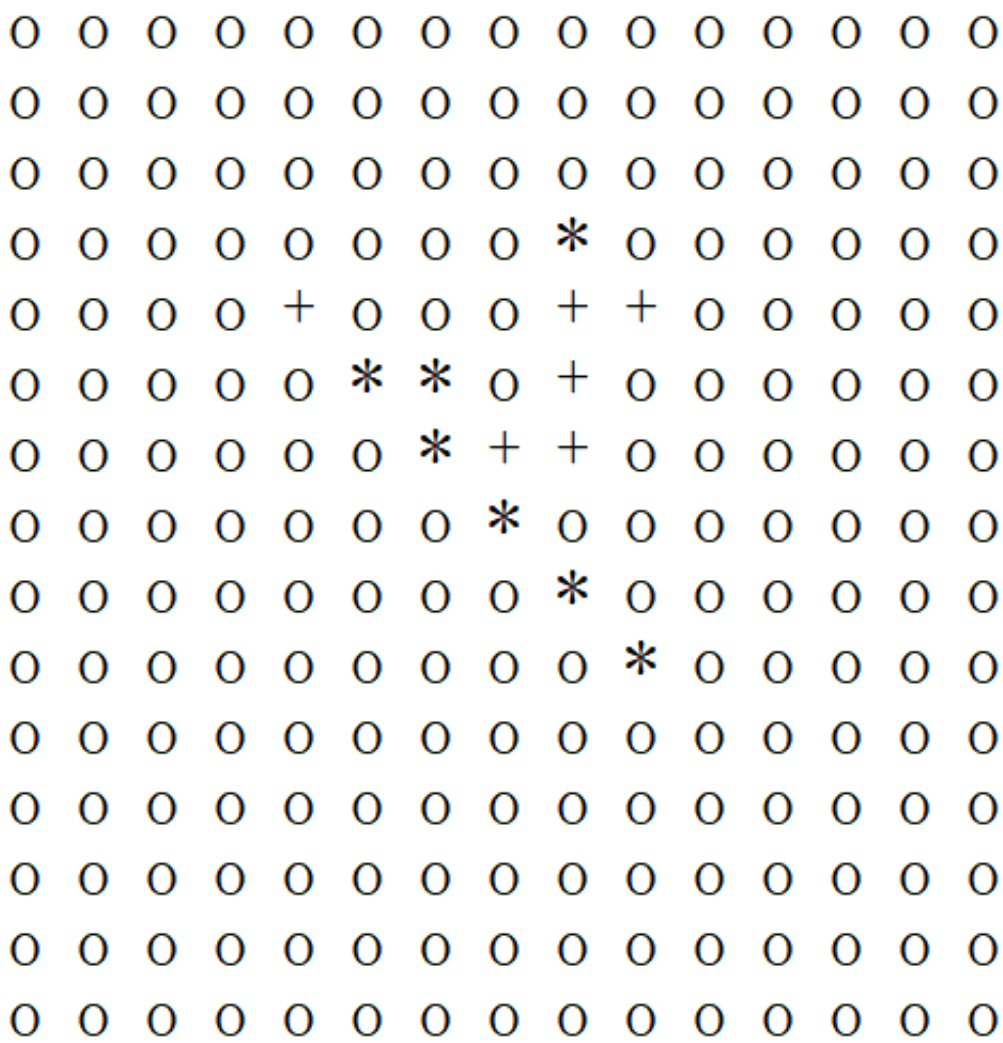
In order to compare the performance of the MiniMax Algorithm and Negamax Algorithm. I make some experiments. First, two AI engines are created separately based on these two different algorithms. Secondly, I make them compete against each other and meanwhile the programme outputs the state of the board when two AI engines take turns to make their own moves.

I make three different experiments to verify some theoritical ideas or conclusions.

To make it clear, in the following experiments, *AI_a* refers to the engine based on Negamax Algorithm while *AI_b* refers to the engine based on MiniMax Algorithm. And '*' means black while '+' means white.

4.1 Experiment I

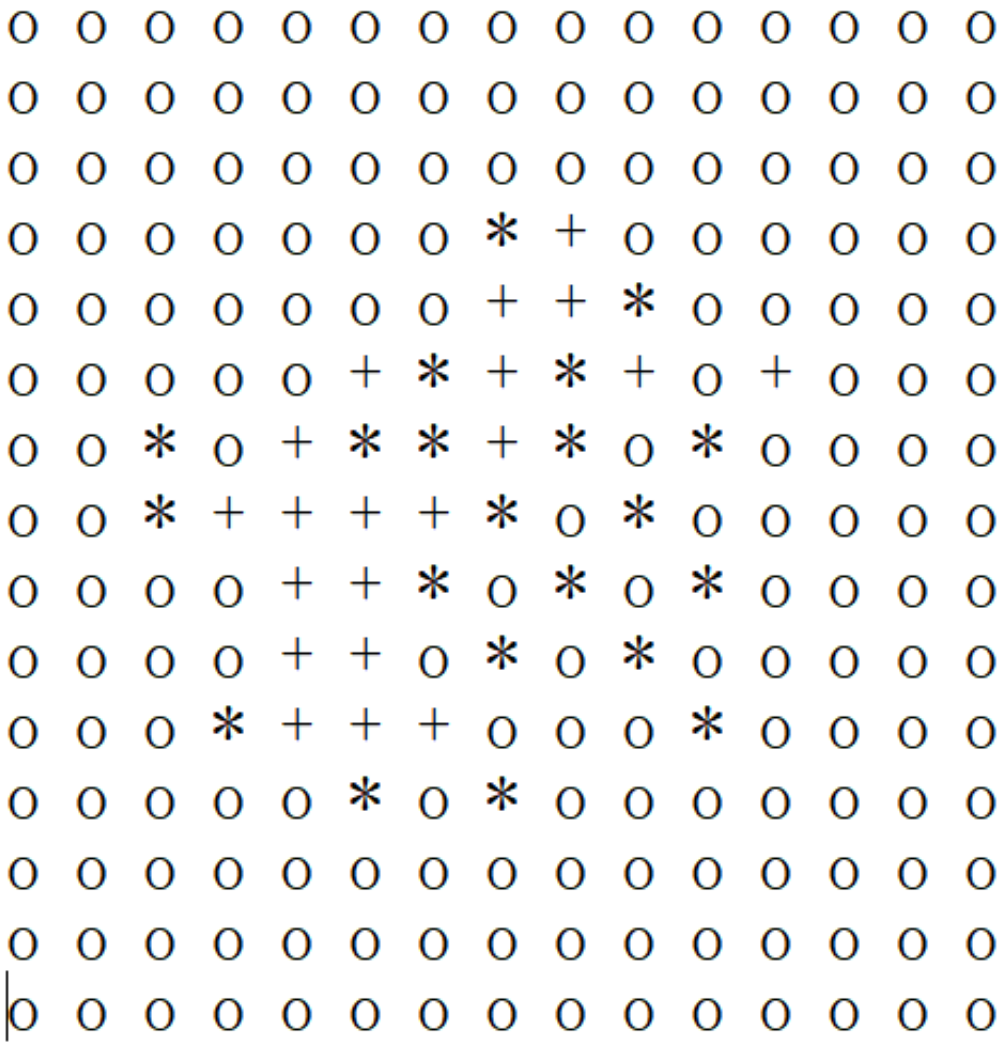
In this experiment, *AI_a* takes **black** and *AI_b* takes **white**. Also, *AI_a*'s search depth is **2** and *AI_b*'s search depth is **1**. What I want to verify is that the search step really counts a lot. The game result is as follows. And the whole game process can be seen in the 'Ex_1.txt' file.



We can easily see that *AI_a* easily beats *AI_b* using just a few steps. So the search step really counts a lot.

4.2 Experiment II

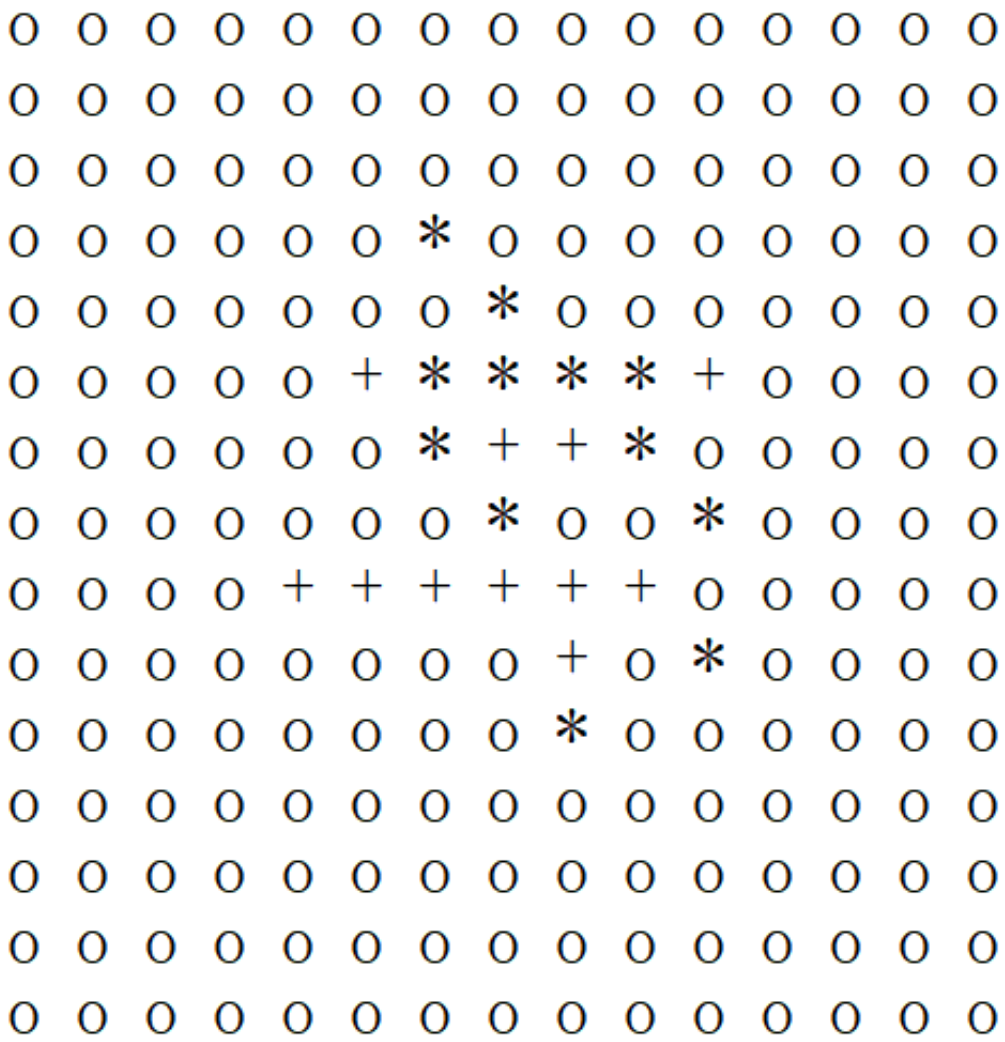
In this experiment, *AI_a* takes **white** and *AI_b* takes **black**. Also, *AI_a*'s search depth is **2** and *AI_b*'s search depth is also **2**. What I want to see is that which algorithm actually performs better. The game result is as follows. And the whole game process can be seen in the 'Ex_2.txt' file.



Finally *AI_a* wins within more steps. So we can safely draw the conclusion that Negamax Algorithm performs slightly better than MiniMax Algorithm to some extent.

4.3 Experiment III

In this experiment, *AI_a* takes **black** and *AI_b* takes **white**. Also, *AI_a*'s search depth is **2** and *AI_b*'s search depth is also **3**. What I want to verify is that even though Negamax Algorithm may perform slightly better than MiniMax Algorithm but what in fact matters most is search depth. The game result is as follows. And the whole game process can be seen in the 'Ex_3.txt' file.



In this experiment, *AI_b* wins within not many turns. So it is apparent that search depth really counts a lot.

5 Expectations

During some tests, I find that Negamax Algorithm still takes an increasing amount of time as the game goes on due to the growing size of search tree because the time complexity is not reduced. So I will focus on how to reduce the time of the algorithm takes by using some optimization algorithms to make some adjustments on alpha-beta pruning algorithm like Aspiration search or Transposition table.