Shi Xingyue

Dr. Rao Yanghui

Artificial Intelligence

22 November 2018

Lab 8: Game Tree Search – Othello

[Abstract]

In this experiment, I implement Othello game with friendly graphic interface. I try three search strategies, namely minimax search (without pruning), Alpha-beta search and Alpha-beta iterating deepening search on this game-tree search problem. Two sharply different evaluation functions are designed and tried out, the first of which is based on some features indicating how desirable a decision is, while the second is statically based on piece values.

1 Methods

1.1 Algorithm

1.1.1 Search Strategy

1.1.1.1 Minimax Search

Minimax's main idea is to "minimize the payoff that could be gained by the other player". By minimizing the other player's payoff, we maximize our own payoff.

Using Minimax search algorithm, we choose the move that forces the opponent to get the worst utility value which is got by recursively calling Minimax going through the set search depth.

1.1.1.2 Alpha-beta Pruned Search

Alpha–beta pruning is a search algorithm that attempts to decrease the number of nodes to evaluate in the minimax algorithm in its search tree. It stops completely evaluating a move when at least one possibility has been found that proves the move to be worse than a previously examined move. Such moves need not be evaluated further, namely pruned from the minimax search tree.

When alpha-beta pruning is applied to the minimax search algorithm, the move chosen is NOT changed, alpha-beta simply saves us from unnecessary evaluations.

1.1.2 Evaluation Functions

I evaluate how desirable a state led to by an action is considering 4 aspects. The first one is the most natural and simple feature – the number of pieces (black or white separately) on board. I would like to call it as “score”. The second one is mobility which take into account both restricting opponent’s mobility and mobilizing myself. Othello strategies put that corners are quite important when playing mainly because they cannot be reversed, so the third feather I take is corners captured. The last feature “piece values” counts the total estimated value of my pieces.

Different deciding strategies are taken at different stage of game. At the very end of game (means the board will be fully captured after this move), I use simply “score” to decide. Otherwise I use the linear combination of 4 features to decide.

1.1.2.1 Score

I use terminology “score” to refer to the numbers of black/ white pieces on board. Score is defined as:

$$\mathrm{score} = \mathrm{nPlayer} - \mathrm{nOther}$$

where nPlayer means the number of pieces in my color, nOther the number of pieces in opponent's color. nPlayer and nOther are quite easy to get by simply checking the entire board.

Obviously, the higher score is, the more desirable a move is. Specially, score is set as $+\infty$ when nOther is 0 showing this move is great, and as $-\infty$ when nPlayer is 0 showing this move is so terrible that not acceptable.

1.1.2.2 Mobility

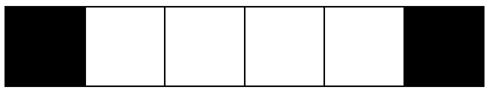
Mobility evaluates whether I am still free to have further move after this one. We prefer moves having higher mobility because they offer us more choices later, while moves having poorer mobility may force us to choose some undesirable move later.

I define "mobility" as:

$$\text{mobility} = \#\ \text{valid moves}$$

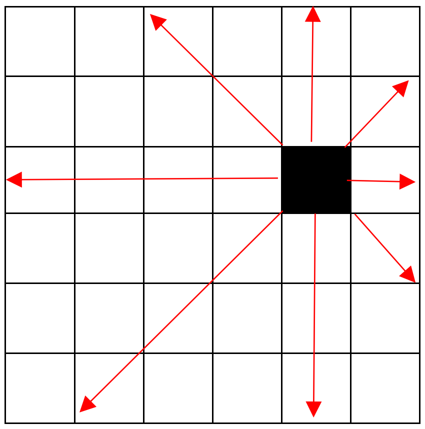
where the number of valid moves (after this one and without considering opponent's next move) is returned from function "Valid".

Function "Valid" counts the number of valid moves of a selected player on current board. When saying "valid", we mean this move can cause a reversion. To be more detailed, a valid move always has similar format as below:



The leftmost grid has been occupied by me, and the rightmost grid is a place to make VALID move. The key point is that all grids between the leftmost and rightmost ones should be occupied by my opponent! They cannot be occupied by me or be blank or otherwise this move is NOT valid.

A string of grids in this format above can also be in other directions. Actually, there are in total 8 possible directions to construct such a string of grids. I need try all of them to find the number of valid moves. Detailed implementation can be found in file “agent.cpp”.



The higher mobility is, the more desirable a move is (or at least I think it is).

1.1.2.3 Corners Captured

I count the number of corners captured by black/white pieces respectively, and define “bonus” as:

$$\mathrm{bonus} = \mathrm{cPlayer} - \mathrm{cOther}.$$

In the first place, I think the higher “bonus” is, the more desirable a move is (namely the higher return value of evaluation function is). But afterward, I find that higher bonus does not necessarily leads to better state for me [2]. To be more exact, bonus should be set to a tiny value when both players have captured corners because this state is not beneficial to minimize my opponent’s offset. After this modification, bonus is defined as:

$$bonus = \begin{cases} a\ tiny\ value, & cPlayer \geq 1\ or\ cOther \geq 1 \\ cPlayer - cOther, & otherwise \end{cases}.$$

I am still slightly confused about why this modification works, but I try out both versions of bonus's definitions and it turns out that the newer one enables the game agent to have more pieces when game ends (but does it mean this evaluation function is better since both colors are played by computer?).

1.1.2.4 Piece values

"Piece values" is a terminology widely used in chess games such as international chess. However, I do not find widely acknowledged piece value for Othello, like international chess. Instead, I get the set of piece values by repeating playing Othello using different piece values. Detailed implementation is recorded in Section 1.4.1.

1.1.2.5 Different Version of Evaluation Functions

I try different evaluation functions during this experiment. However, I do not know how to tell which one is the best for not knowing criteria to or baseline. I only know that all of them succeed in winning the game.

	EVALUATION FUNCTION #1	**EVALUATION FUNCTION #2**	**EVALUATION FUNCTION #3**
SCORE	✓	✓	✓
MOBILITY	✓	✓	✓
CORNERS		✓	✓
PIECEVALUES	✓ (ver. 1)	✓ (ver.2)	✓

1.2 Pseudo-code

1.2.1 Minimax Search

I implement the Depth-First version of minimax search algorithm.

```
minimax (player, board, depth, r_out, c_out)


if depth==0 //if has reach the set depth
        return eval(board)//evaluation function
else
        MoveList = all valid next moves of n
        if MoveList==NULL
                return eval(board)
        else
                for each move in MoveList:
                        result = -minimax(player.switch(), board, depth-1, r_out, c_out)
                bestmove=move having MAX(result)
                r_out=bestmove.r
                c_out=bestmove.c
                return MAX(result)
```

1.2.2 Alpha-beta Pruned Minimax Search

Adding alpha-beta pruning to Minimax search algorithm does not change its overall algorithm structure or running results. Pseudo-code of alpha-beta pruned Minimax search is as below:

```
minimax (alpha, beta, player, board, depth, r_out, c_out)
//need additional parameters, alpha and beta

if depth==0 //if has reach the set depth
        return eval(board)//evaluation function
else
        MoveList = all valid next moves of n
        if MoveList==NULL
                return eval(board)
        else
                for each move in MoveList:
                        result = -minimax(alpha, beta, player.switch(), board, depth-1, r_out, c_out)
                        alpha=MAX(result,alpha)
                        beta=MIN(result,beta)
                        //alpha-beta pruning
                        if (beta<=alpha)
                                break;
                bestmove=move having MAX(result)
                r_out=bestmove.r
                c_out=bestmove.c
                return MAX(result)
```

1.3 Key Codes with Comments

1.3.1 Alpha-beta Pruned Minimax Search

Codes in this function is quite long, I divide it into parts to make it more readable.

The Minimax search is implemented with recursion. The first boundary condition of recursion is that “search has gone through the set depth”:

```
if (depth == 0){
	//boundary condition 1
	//reach deepest search depth
	return (*eval)(player, board, mGrid, mValid);
}
```

If the first boundary condition is not reach, I get all the possible moves by calling function “GetOrderMove”. Moves are sorted by descending priority and stored in vector “moves”.

```
vector<cMove> moves = GetOrderMove(player, board, mValid, mGrid);
```

Then I test whether the second boundary condition is reached. The second boundary condition is that the search tree has reach a leaf, namely the vector “moves” is empty. In this situation, I check whether my opponent has valid moves. If my opponent still has valid moves, I choose the move minimizing its payoff by “`return -MiniMax(next, board, depth, mGrid, mValid, eval, alpha, beta,r_out, c_out, searchtimes);`” (note the minus!). If my opponent has no valid moves either, I consider current state as “quite desirable” because it restricts my opponent mobility greatly.

```
//boundary condition 2
//no valid move -> evaluate utility using uDiff
if (moves.size() == 0){
	//nMove is the number of valid moves of NEXT player
	int nMove = board.Valid(next, mValid);
	if (nMove != 0){
		return -MiniMax(next, board, depth, mGrid, mValid, eval, alpha, beta,r_out, c_out, searchtimes);
	}
	else{
		int nPlayer = board.Count(player);
		int nOther = board.Count(next);
		int nDiff = nPlayer - nOther;
		return INFINITY / (nROW * nCOL) * nDiff;
	}
}
```

If neither boundary condition 1 nor 2 is reached, it means the search process goes on and I the player still has some valid move(s). Please note that alpha-beta pruning is added into this part.

```
//if there is some valid action(s)
cMove mBest(-1, -1, -1);
double nScore = -INFINITY * 2;
for (int m = 0; m<moves.size(); m++){
        int r = moves[m].r;
        int c = moves[m].c;


        //try taking this move
        cMatrix<bool> mUndo;
        mUndo.Resize(nROW, nCOL, false);
        board.Move(player, r, c, mUndo);
        board.Set(r, c, player);

        //get opponent's payoff
        //I want to minimize this payoff
        double result = -MiniMax(next, board, depth - 1, \
        mGrid, mValid, eval,alpha, beta, r_out, c_out,searchtimes) ;
        searchtimes++;


        //update nScore=MAX(result)
        if (result > nScore){
                nScore = result;
                mBest = moves[m];
        }


        //alpha-beta pruning
        alpha = (alpha >= result) ? alpha : result;
        beta = (beta <= result) ? beta : result;
        if (beta <= alpha) {
                //cout << "Alpha-beta pruned here!" << endl;
                cout << endl << endl;
                //undo the move
                board.Remove(r, c);
                board.Undo(mUndo);
                //prune this branch!
                break;
        }



        //undo the move
        board.Remove(r, c);
        board.Undo(mUndo);
}
```

After the searching done in the "for" block above, I find the best move which minimizes opponent's payoff. Grid's co-ordinate of this best move is passed by parameter "r_out" and "c_out". My own payoff is returned as return value.

```
r_out = mBest.r;
c_out = mBest.c;
return nScore;
```

1.3.2 Evaluation Functions

First, I calculate 4 features one by one. Some of them can be easily got so I calculate them directly in evaluation functions, while the other needs more work so I implement them in other functions.

As for evaluation function #3, the return value is a linear combination of these 4 features defined as:

$$\text{return value} = \text{score} * 0.1 + \text{mobility} + \text{piecevalues} + \text{bonus}.$$

```
double Evaluate3(const cPlayer player, cBoard& board, cMatrix<double>& mGrid,
cMatrix<bool>& mValid){
        //clarify role
        cPlayer other;
        if (player == gBLACK) other = gWHITE;
        else if (player == gWHITE) other = gBLACK;

        //feature 1: score
        int nPlayer = board.Count(player);
        int nOther = board.Count(other);
        double score = nPlayer - nOther;

        //if opp has no piece
        if (nOther == 0) return INFINITY;

        //if player has no piece
        if (nPlayer == 0) return -INFINITY;


        //if this is the very end of the game, pieces decide strategy
        if (nPlayer + nOther == nROW * nCOL){
                return score;
        }

        //feature 2: mobility
        int mobPlayer = Mobility(player, board, mValid);
        int mobOther = Mobility(other, board, mValid);
        int mob = mobPlayer - mobOther;

        //feature 3: piece values
        double posPlayer = PieceValue2(player, board, mGrid);
        double posOther = PieceValue2(other, board, mGrid);
        double pos = posPlayer - posOther;
        //feature 4: corner captured
        double bonus = 0;
        int cPlayer = 0;
        int cOther = 0;
        int r, c;

        r = 0; c = 0;
        if (board[r][c] == player) cPlayer++;
        else if (board[r][c] == other) cOther++;

        r = nROW - 1; c = 0;
        if (board[r][c] == player) cPlayer++;
        else if (board[r][c] == other) cOther++;

        r = 0; c = nCOL - 1;
        if (board[r][c] == player) cPlayer++;
        else if (board[r][c] == other) cOther++;

        r = nROW - 1; c = nCOL - 1;
        if (board[r][c] == player) cPlayer++;
        else if (board[r][c] == other) cOther++;

        //discourage when both has corner
        //confusing?
        if (cPlayer >= 1 && cOther >= 1) bonus = -85* mGrid[2][2];
        else bonus = cPlayer - cOther;

        return (score*0.1 + mob + pos + bonus);
    }
```

1.4 Innovation and Optimization

1.4.1 Self-learnt Piece Values

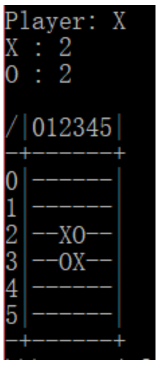
Piece values I use are not set manually but learnt by repeating competition [3]. I start with a set of piece values named as "winner". Then I generate random numbers to make another set of piece values named as "challenger". Agent use the same evaluation function with different set of piece values to play "X" and "O" twice, with "X" and "O" each starting game once. If "challenger" wins both games, replace "winner" with "challenger". Repeat this process until no more replacement.

1.4.2 Friendly Graphic User Interface

After making sure all the algorithms do work, I try making a friendlier GUI using what I learnt from CG course last semester! I use OpenGL to make a simple interface. My attempt is shown in "Result" section.

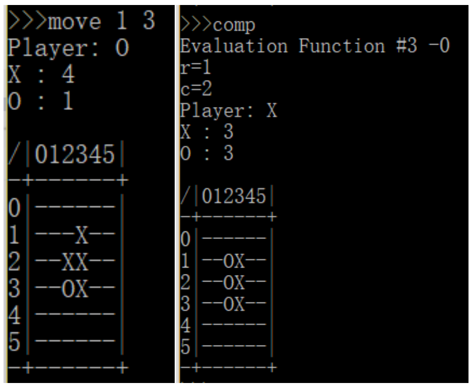
2 Results and Analysis

2.1 Result Examples



```
Player: X
X : 2
O : 2

/|012345|
-+------+
0|------|
1|------|
2|--XO--|
3|--OX--|
4|------|
5|------|
-+------+
```

Figure 1 Initial board



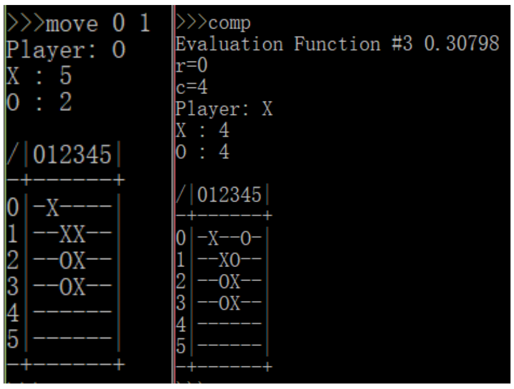
```
>>>move 1 3
Player: O
X : 4
O : 1

/|012345|
-+------+
0|------|
1|---X--|
2|--XX--|
3|--OX--|
4|------|
5|------|
-+------+
```

```
>>>comp
Evaluation Function #3 -0
r=1
c=2
Player: X
X : 3
O : 3

/|012345|
-+------+
0|------|
1|--OX--|
2|--OX--|
3|--OX--|
4|------|
5|------|
-+------+
```

Figure 2 Round 1.

I place "X" on (1,3). Agent places "O" on (1,2) with evaluation function returning 0.

After this round, there are 3 "X" and 3 "O" on board.



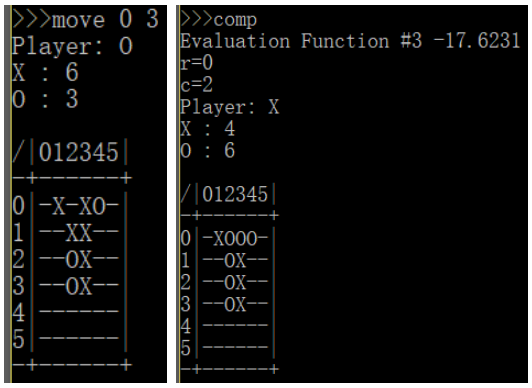
```
>>>move 0 1
Player: O
X : 5
O : 2

/|012345|
-+------+
0|-X----|
1|--XX--|
2|--OX--|
3|--OX--|
4|------|
5|------|
-+------+
```

```
>>>comp
Evaluation Function #3 0.30798
r=0
c=4
Player: X
X : 4
O : 4

/|012345|
-+------+
0|-X--O-|
1|--XO--|
2|--OX--|
3|--OX--|
4|------|
5|------|
-+------+
```

Figure 3 Round 2.

I place "X" on (0,1). Agent places "O" on (0,4) with evaluation function returning 0.30798.

After this round, there are 4 "X" and 4 "O" on board.



```
>>>move 0 3
Player: O
X : 6
O : 3

/|012345|
-+------+
0|-X-XO-|
1|--XX--|
2|--OX--|
3|--OX--|
4|------|
5|------|
-+------+
```

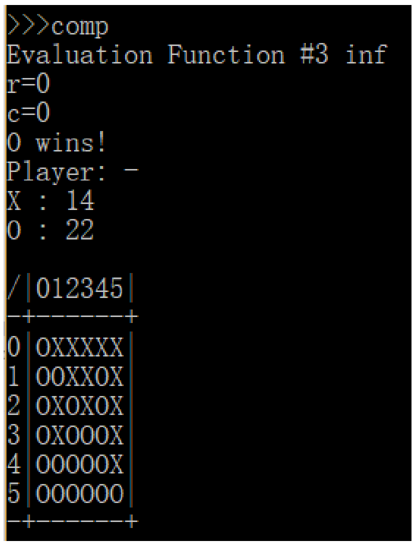
```
>>>comp
Evaluation Function #3 -17.6231
r=0
c=2
Player: X
X : 4
O : 6

/|012345|
-+------+
0|-XOOO-|
1|--OX--|
2|--OX--|
3|--OX--|
4|------|
5|------|
-+------+
```

Figure 4 Round 3.

I place "X" on (0,3). Agent places "O" on (0,2) with evaluation function returning -17.6231.

After this round, there are 4 "X" and 6 "O" on board.

When running this example game, I use Evaluation Function #2 to guide me and the agent uses Evaluation Function #3. After dozens of rounds, the game is over with this result.



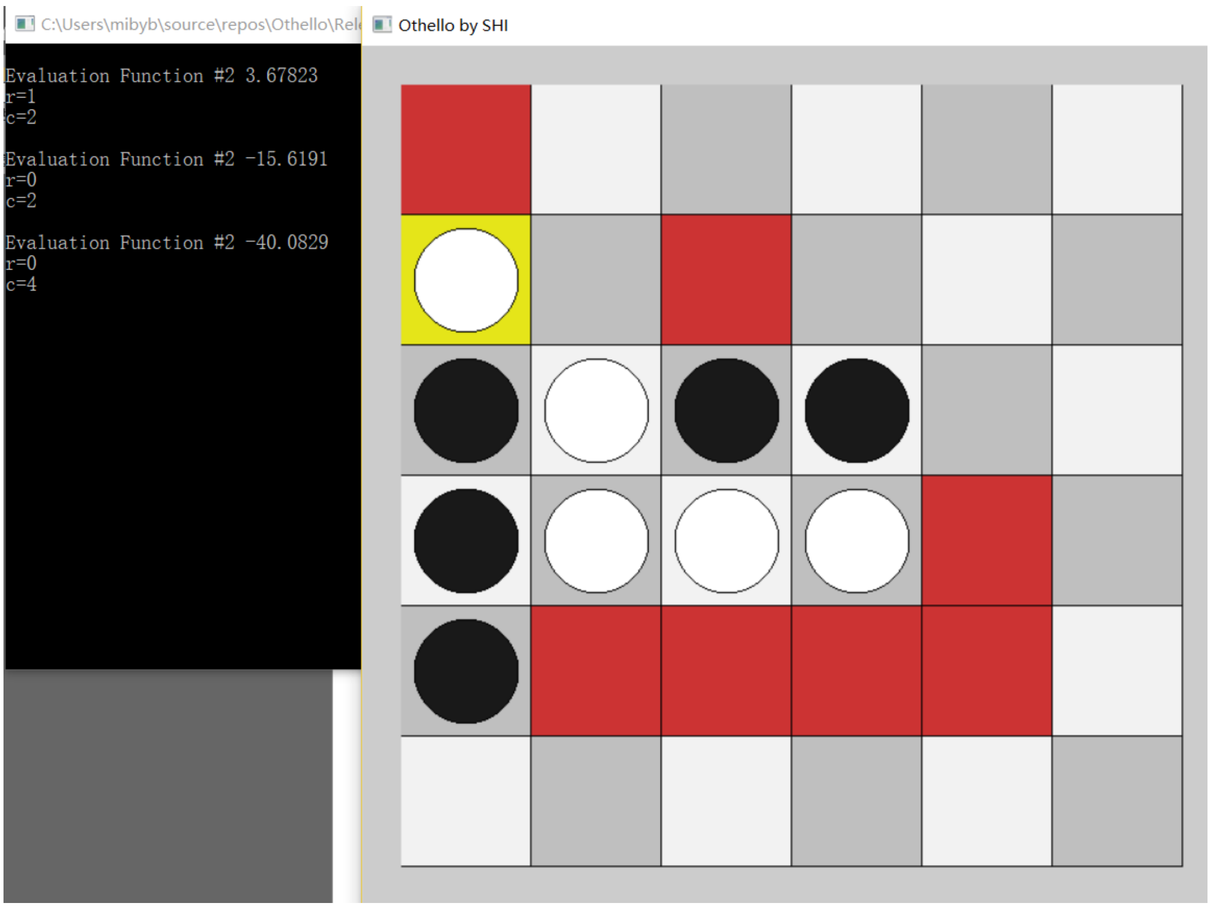
```
>>>comp
Evaluation Function #3 inf
r=0
c=0
O wins!
Player: -
X : 14
O : 22

/|012345|
-+------+
0|OXXXXX|
1|OOXXOX|
2|OXOXOX|
3|OXOOOX|
4|OOOOOX|
5|OOOOOO|
-+------+
```

Figure 5 Game Over!

Agent using Evaluation Function #3 win!

2.2 GUI Version

GUI version of this game is shown below. The return value of evaluation function of agent's move is shown in command line.



```
Evaluation Function #2 3.67823
r=1
c=2

Evaluation Function #2 -15.6191
r=0
c=2

Evaluation Function #2 -40.0829
r=0
c=4
```

Figure 6 GUI version.

The search algorithm and evaluation functions are identical to previous ones.

3 Reflection

This experiment is the one I am most interested in up to now. I learn together with my agent how to play Othello during this experiment because I am a truly beginner in it. Researchers have tried tons of evaluation functions and some of them are really elegant and interesting.

It is a little pity that I still don’t know how to judge an evaluation function. Maybe competing repeatedly using this evaluation function with other players is an approach. I would love to learn more about this field.

Work Cited

[1] Sannidhanam，Vaishnavi. and. Annamalai，Muthukaruppan. *An Analysis of Heuristics in Othello. 2004.*

[2] http://www.cs.cornell.edu/~yuli/othello/othello.html

[3] https://www.cs.cmu.edu/~chengwen/reversi/Reversi.html