Chess

Ping-Jung Liu October 5th 2017

1. Introduction

In this project, I implemented a chess artificial intelligence with minimax algorithm and its numerous extentions. The objective of the project, obviously, is to defeat the other player, human or AI. The required extentions include evaluation function, cutoff test, alpha-beta pruning, transposition table, and iterative deepening search. The extentions for bonus credits include modification of transposition table, opening moves to make AIs look smarter, large-scale reordering of moves to further enhance alpha-beta pruning, and an overview of a related research paper.

2. Model

We used the python chess package as the model of this project, which imbedded all the chess rules and game over conditions.The challenge of this project is to implemented a huge amount of algorithms and potential extentions, which is why I decided to show the testing results after explaining every algorithms and decisions I made.

3. Minimax

The main idea of minimax is to look for optimal moves in the search tree. Given a maximum depth of D, the program will search D moves away from current position, assuming both players are logical and will always maximize their own utilities.

Minimax, implemented in MinimaxAI.py consists of three functions, minimax decision, min value, and max value. The AI will call minimax decision to make the actual decision. minimax decision will call min value to decide which of the legal moves from current position will result in highest utility. min value and max value will then recursively call each other until they reach the depth limit.

The codes given will be a condensed version that can still show the main idea of a function, because many of the extentions were already in the actual function.

minimax_decision:

```
for move in list(board.legal_moves):

    board.push(move)

    # find the best move in min nodes
    moveUtil = self.min_value(board, 1)
    if moveUtil >= bestUtil:
        bestUtil = moveUtil
        bestMove = move

    board.pop()
```

min_value:

```
for move in list(board.legal_moves):

    board.push(move)
    minUtil = min(minUtil, self.max_value(board, depth + 1))
    board.pop()
```

max_value:

```
for move in list(board.legal_moves):

    board.push(move)
    maxUtil = max(maxUtil, self.min_value(board, depth + 1))
    board.pop()
```

4. Alpha-Beta Pruning

Alpha-Beta Pruning is the most well-known modification of minimax, which records the lower and upper bound utilities along the way, so that the program can ignore a whole subtree if those nodes are impossible to be selected by min and max.

Alpha-Beta Pruning in min_value:

```
for move in new_list:
    #list(board.legal_moves):

    board.push(move)
    minUtil = min(minUtil, self.max_value(board, depth + 1, alpha, beta))
    board.pop()

    # pruning
    if minUtil <= alpha:
        return minUtil
    # modify beta
    beta = min(minUtil, beta)
```

Alpha is the lower bound and Beta the upper. If a min value encountered a utility that is less than the lower bound, it recognizes that its parent is unlikely to choose any move in this subtree as optimal move, so just return and move on to the next subtree.

5-1. Feature: Cutoff Function

Unfortunately, it is practically impossible to search the whole true, given the average number of legal moves for each position is about 38. We have to set up cutoff situation so that the program can end when it reaches a certain depth. The program will use a evaluation function to estimate the utility of the current position and return it. I will discuss the evaluation function in the next section.

cutoff:

```
def cutoff(self, board, depth):
    return depth > self.currentDepth or board.is_game_over()
```

Please note that self.currentDepth is the current depth limit.

5-2. Feature: Evaluation Function:

The utility of winning is the largest integer in python; the losing utility is negative of that; I decided the draw utility to be 0. These are the simple cases because the game will inevitably end when the game reaches any of the situations.

I followed the text book's suggestion to implement a utility function that can calculate any possible board position. The idea is to give each unit a weight W, find the numbers of each unit, and sum up Wi * Ni of the AI, then subtract 0.5 of opponent's total score to obtain the current utility. A bit confusing so I will use a simple example: Say the weight of queen is 5, king is 6. White has one queen and one king; black has one king. The current utility of white is therefore 1 * 5 + 1 * 6 - 0.5 * 1 * 6 = 8. The utility of black is 1 * 6 - 0.5 * (1 * 5 + 1 * 6) = 0.5. The idea behind this is that if the opponent has more strong units left, the chance to win is also smaller.

get_utility (I used the sample utility function for both minimax and alpha-beta):

```
def get_utility(self, board):
    # whose turn is it
    turn = int(board.turn)
    u = 0

    if board.is_game_over():
        if board.is_checkmate():
            if self.side == turn :
                return self.lossU
            else:
                return self.winU
        else:
            return 0
    else:
        for i in range(1, 7):
            # board.pieces return the positions of a piece of one player
            u = u + i * len(board.pieces(i, self.side))
            u = u - 0.5 * i * len(board.pieces(i, (self.side + 1) % 2))

    return u
```

5-3. Feature: Iterative Deepening Search:

The purpose of using IDS here is to ensure the AI always has a move to perform. In real life situation there might be time constraints on each decision, so the AI needs to make sure it has a somewhat logical move. Another reason is to make reordering of moves easier with the aid of transposition table; this will be discussed in later sections.

```
for i in range(0, self.maxDepth):

    self.currentDepth = i

    (move, util) = self.minimax_decision(board)

    # if encounter winning utility, return
    if util == self.winU:
        print(util)
        return move
```

5-4. Feature and Discussion: Transposition Table (Extra):

Just like memoization in dynamic programming, a transposition table revord previously visited board postions for future references. I used a python set to implement the transposition table in this project. Since the board class cannot be hashed, I used str(board) as the keys for the table.

In my first attempt, I only saved the utilities in the table. When the program reaches a visited position, it will return the utility. It turned out terrible because the utility of a position could be obtained in numerous ways; the search depth left also differs, which make it unwise to treat transposition table as a normal memoization set. This implementation allowed me to reach depth 25, which sadly could not beat an AI with max depth 5 because of the above reasons.

In my second attempt, I saved the search depth of a postion as well. If the program reaches a visited position, it will check whether the recorded depth is greater than the current search depth, and return if this is true. The idea is that the deepest search contains the most information, and thus would be a better utility to return.

In my third attempt, I modified the table further more for alpha-beta pruning. In addition to utility and depth, the table now store a third element, which indicates what what type of utility it is. Types include, lower bound, upper bound, and exact values, which means the postion is not pruned. When the programmed reach a visited postion, it will check the depth and utility type before returning, which will be shown in the code.

minimax return scenario:

```
if str(board) in self.transp and self.t:
    if self.transp[str(board)][1] > self.currentDepth:
        return self.transp[str(board)][0]
```

minimax replacement scenario:

```
if self.cutoff(board, depth):
    util = self.get_utility(board)
    # modify transposition table
    if self.t and (not str(board) in self.transp or depth > self.transp[str(board)][1]):
        self.transp[str(board)] = (util, depth)

    return util
```

alpha-beta return scenario:

```
if str(board) in self.transp and self.t:
    if self.transp[str(board)][1] > self.currentDepth:
        util = self.transp[str(board)][0]

        if self.transp[str(board)][2] == "EXACT":
            return util
        elif self.transp[str(board)][2] == "LOW" and util <= alpha:
            return util
        elif self.transp[str(board)][2] == "UP" and util >= beta:
            return util
```

alpha-beta replacement scenario:

```
if self.cutoff(board, depth):

    util = self.get_utility(board)

    if self.t:
        if util <= alpha:
            self.transp[str(board)] = (util, depth, "LOW")
        elif util >= beta:
            self.transp[str(board)] = (util, depth, "UP")
        else:
            self.transp[str(board)] = (util, depth, "EXACT")

    return self.get_utility(board)
```

5-5. Feature and Discussion: Reordering (Extra):

Given the way alpha-beta pruning works, it is obvious that the order of legal moves can affect the running time. With the support of transposition table and iterative deepening search, I implemented a large scale reordering to further decrease the runtime of alpha-beta pruning. For max value, I will sort the whole legal move list in descending order, so that the move with highest potential utility appears at the front, opposite for min value. The reorder function takes a board position and return a list of moves sorted by utilities. If a postion has been visited, use the utility saved in transposition table, if not, use get utility function to obtain a estimation.

```
def reorder(self, board):

    move_list = list(board.legal_moves)
    temp = []

    # loop through the list
    for move in move_list:

        board.push(move)
        # if visited, get the utility from transposition table
        if str(board) in self.transp:
            temp.append((self.transp[str(board)][0], move))
        # if not visited, calculate utility
        else:
            temp.append((self.get_utility(board), move))

        board.pop()

    # sort the the list based on utilities
    temp = sorted(temp, key=lambda util: util[0])
    # reverse it so it begins with the largest utility
    temp.reverse()

    # obtain a list of moves for convenience
    move_list = []
    for i in range(0, len(temp)):
        move_list.append(temp[i][1])
    return move_list
```

5-6. Feature and Discussion: Opening Moves (Extra):

After I mostly finished the whole project, I realized that the AI can smartly respond to my actions, but when I let two AIs fight each other, they tend to perform repetitive and illogical moves at the beginning. I found out that many of the existing chess AIs have hard-coded opening moves to act smarter. I implemented the Belgrade Gambit opening and the AI indeed acted more like an actual player. Note that I used a counter to keep track of move number.

```
self.moveNum = self.moveNum + 1
if self.moveNum == 1:
    if self.side == 1:
        return chess.Move.from_uci("b1c3")
    else:
        return chess.Move.from_uci("g8f6")
elif self.moveNum == 2:
    if self.side == 1:
        return chess.Move.from_uci("g1f3")
    else:
        return chess.Move.from_uci("b8c6")
elif self.moveNum == 3:
    if self.side == 1:
        return chess.Move.from_uci("e2e4")
    else:
        return chess.Move.from_uci("e7e5")
```

6. Testing:

The testing of all these algorithms and features is a tedious process, so I will try to make them as concise and clear as possible.

Result of two MinimaxAI with depth 2 fighting each other, to show original minimax works:

```
Black to move

2147483647
Move:
c1a1
r . . . . b . Q
. . . . . . . P
k . . . . . . .
n . p . N P . .
b . P . . . . .
. n . . . . . P
K . . . . . . .
q . . . . . . .
---------------
a b c d e f g h

Black Won
```

The utility of black is the winning utility 2147483647, and we can observe that the white K is obviously check-mated.

Result of two AlphaBetaAI with depth 1 fighting each other, to show original alphabeta works:

```
Black to move

Move Util:
0
Nodes Visited:
2
Move:
a1a2
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . N . . . . .
. . . . . . . .
. . . . . . . .
k . . . . B . .
. . K . . . . .
---------------
a b c d e f g h

DRAW!
```

I played with both minimax and alphabeta to verify that they indeed found moves with equal utility, but alphabeta took less time.

minimax depth 3 respond to moves:

```
White to move

Please enter your move:
b2b4
____________________________________________________________________________________
Move Util:
19.0
Nodes Visited
26661
Move:
f8b4
____________________________________________________________________________________
White to move

Please enter your move:
a2a4
____________________________________________________________________________________
Black to move

Move Util:
19.0
Nodes Visited
34946
Move:
a7a5
```

alphabeta depth 3 respond to moves:

```
White to move

Please enter your move:
b2b4
____________________________________________________________________________________

Black to move

Move Util:
19.0
Nodes Visited:
3391
Move:
f8b4
____________________________________________________________________________________
White to move

Please enter your move:
a2a4
____________________________________________________________________________________
Black to move

Move Util:
19.0
Nodes Visited:
15158
Move:
a7a5
```

We can see that the move utilities found by alpha-beta were the same as minimax, but visited significantly less nodes.

I will now compare nodes visited of regular minimax and transposition table implemented minimax.

Without transposition table:

```
Nodes Visited:
26661

Nodes Visited:
34946

Nodes Visited:
33235

Nodes Visited:
51133
```

With transposition table, respond to same moves:

```
Nodes Visited:
19453

Nodes Visited:
88

Nodes Visited:
20884

Nodes Visited:
100
```

We can see that though the runtime saved by transposition table varied among different situations, overall it saves a decent amount of nodes visited.

I will then compare nodes visited of transposition table alpha-beta and transposition table + reordering alpha-beta.

Without reordering:

```
Nodes Visited:
4865

Nodes Visited:
2298

Nodes Visited:
7488

Nodes Visited:
2670
```

With reordering respond to same moves:

```
Nodes Visited:
1782

Nodes Visited:
1860

Nodes Visited:
1993

Nodes Visited:
2240
```

Again, even though the runtime saved by reordering varied, it still saves nodes visited.

With all the additional features, the AI can calculate depth 4 in 5-10 seconds, and can calculate depth 5 in about a minute.

7. Literature Review (Undergrad):

I reviewed "Genetic Algorithms for Evolving Computer Chess Programs by Omid E. David, H. Jaap van den Herik, Moshe Koppel, and Nathan S. Netanyahu" to further investigate the current state of artificial chess players.

Many of the top chess programs used hard coded openings and grandmaster level evaluation functions to crush opponents. With parallel computing the AI can search deep into the tree to outplay others. Another approach, which the team had taken before as well, combined the design process with machine learning techniques and allowed evaluation function parameters to learn and evolve from games. In this paper, in addition to evolve the evaluation function, the team managed to evolve the selective search mechanism as well. The objective is to show that a chess program with little assumption on grandmaster evaluation function and selective search mechanism can still become a decent chess player through machine learning.

Numerous popular selective search mechanisms include null-move pruning, futility pruning, multicut pruning, etc. Given that these mechanisms are all controlled by some critical parameters, the team used existing fitness function to train the program. As stated in the paper, even though the resulting AI acts differently from the AIs with fixed search mechanism and evolved evaluation function, it is capable of playing against tournament level chess AIs, which was a breakthrough such that the evaluation function and search mechanism were initialized with random parameters, so given enough time and game cases, the program can surely evolved into more advanced AI.