Edmund Lai and Zoheb Siddiqui

March 7th, 2018

Prof. Tanimoto/CSE 415

Report for Final Project

Report

1) Title: Baroque Chess AI agent.
2) A) Zoheb Siddiqui: Created the static eval, minimax(without αβ pruning), IDDFS and Zobrist hash functions.
   B) Edmund Lai: Created the move generator, including the findNewMove for each type of piece and findCaptureMove for each type of piece. Implemented the alpha-beta pruning in minimax. Responsible for general debugging and code design.
3) The program is designed to play the game of baroque chess smartly, using various AI techniques learnt in class.
4) Minimax: This function takes in a state and a specified depth. It figures out the best possible move that can be made in the current state by analyzing the static eval values for successor states and maximizes/ minimizes them depending on whose turn it is. This process is repeated till all states for the given depth have been analyzed or if the method runs out of time. In both cases the best possible state analyzed(yet) is returned.
   Zobrist Hashing: This technique is used to map a state to a unique hash value. The unique hash is the key to the static eval value of the state. Storing static eval values in a hash table saves time while running the minimax as it avoids recomputing values for states that have been analyzed before.
   αβ pruning: This is another technique to optimize minimax to save time and avoid computing unnecessary states. The alpha-beta technique exploits the fact that at each node, the maximizing player will choose the node that gives them the max value, and the minimizing player will choose in order to receive the min value, so any decision tree where a state is possible where the end value is worse for the currently evaluating player can be cut off prematurely and not fully explored, thus saving time.

5) Screenshot:

```
Minimax backed up value of 2.9 at ply of 1
Minimax backed up value of -2.0 at ply of 1
Minimax backed up value of -0.6 at ply of 1
Minimax backed up value of -0.6 at ply of 2
Time used in makeMove: 4.5333 seconds out of  5.0
WHITE's move: the P at (6, 5) to (5, 5).
Turn 7: Move is by WHITE
Magnus Carlsen says: How am I supposed to choose from 60 possible moves? This is insanity!
c l i w k i l f
p p p p p p - -
- P - - - - - -
- - - - - - - -
- - - - - - p -
I - F - - P - p
P - P P P - P P
- L - W K I L C
BLACK's move

Minimax backed up value of -0.8 at ply of 1
Minimax backed up value of 4.1 at ply of 1
Minimax backed up value of 4.3 at ply of 1
Minimax backed up value of 3.2 at ply of 1
Minimax backed up value of 3.2 at ply of 2
Time used in makeMove: 4.5405 seconds out of  5.0
BLACK's move: the l at (0, 6) to (2, 6).
Turn 8: Move is by BLACK
Magnus Carlsen says: If there were 892 fish in the sea, I would have at least that many Zobrist hash values to stop you!
c l i w k i - f
p p p p p p - -
- P - - - - l -
- - - - - - - -
- - - - - - p -
I - F - - P - p
P - P P P - P P
- L - W K I L C
WHITE's move
```

6) Demo instructions:
   In order to play Baroque Chess, you need to navigate to the folder where the files are contained and type in the command:

   python3 BaroqueGameMaster.py nameOfFirstAgent nameOfSecondAgent timeLimit

7) Code excerpt:

```
def staticEval(state):
    weights = [0,0,1,1,5,5,3,3,6,6,4,4,10,10,4,4]
    board = state.board
    white_list = []
    black_list = []
    piece_present_sum = 0
    can_move_sum = 0
    kill_sum = 0
    for i in range(0,8):
        for j in range(0,8):
            piece = board[i][j]
            if piece != 0 and piece != 1:
                if piece % 2 == 0:
                    piece_present_sum -= weights[piece]
                    black_list.append([piece,(i,j)])
                else:
                    piece_present_sum += weights[piece]
                    white_list.append([piece,(i,j)])
```

```
    to_return = 0.3*(piece_present_sum)
    w_moves = generateNewMoves(state, BC.WHITE)
    b_moves = generateNewMoves(state, BC.BLACK)
    can_move_sum = len(w_moves) - len(b_moves)
    for white_move in w_moves:
        kill_list = white_move[2]
        for elem in kill_list:
            kill_sum += weights[elem]
    for black_move in b_moves:
        kill_list = black_move[2]
        for elem in kill_list:
            kill_sum -= weights[elem]
    to_return += 0.6*(kill_sum)
    to_return += 0.1*(can_move_sum)

    return to_return
```

This is the static eval function. It returns a value for the state depending on if it's good for white or black. If the state is good for white a high value is returned and if it is good for black a low value is returned.
Each piece is given a value depending on it's importance. E.g the king is 10 and pincer is 1. The value returned consists of 3 parts. The first is the weighted difference between pieces remaining for white and black. This makes up 30% of the value. The 2nd is the difference between the total number of moves possible for white and black. This makes up 10% of the value. 60% of the value consists of the difference in kill sum for white and black. The kill sum is a weighted sum of all the pieces that can be killed for each opponent in the current state.

8) A) Zoheb Siddiqui:

I learnt how to implement Zobrist hashing and integrate it into minimax. I also learnt how to correctly implement a time limit for the minimax function. I also had to think about different strategies for baroque chess to implement a good static eval method.

B) Edmund Lai:

I learned how to think procedurally like a computer as to how to check what new positions any given piece could reach given the current board state. I also learned how to organize my different functions for each simpler piece such as the Coordinator, King, and Withdrawer such that it would allow easy implementation for more complicated pieces such as the Leaper or the Imitator. I also became quite familiar with the importance of testing each individual piece in order to eliminate possible causes for bugs in the code, especially in a complicated game like Baroque Chess. I also became more familiar with the minimax method implemented with alpha-beta and learned how to do Zobrist hashing.

9) If we had more time we would have implemented Q learning to teach the agent about the consequences to moving to a new state to make it smarter. We would also try to optimize our current code to work more efficiently to process more states in a limited time.
10) We used Wikipedia to learn about the different ways the individual pieces act and how Baroque chess is played.

    https://en.wikipedia.org/wiki/Baroque_chess

    We also used chessprogramming to design a working evaluation function for Baroque Chess. Despite the focus on the original game of Chess, the basic methodology is quite similar.

    https://chessprogramming.wikispaces.com/Evaluation

    We also used inference.org to clarify the specific details on complicated interactions between pieces, such as the imitator capturing multiple pieces at once, or the immobility of a piece with the presence of an ally imitator and an enemy freezer next to each other.

    http://www.inference.org.uk/mackay/ultima/ultima.html

    We used geeksforgeeks.org to better understand the implementation of Zobrist hashing, and understand why it is used in chess.

    https://www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-5-zobrist-hashing/

11) Partners' Reflections
    A) Zoheb Siddiqui:
    Role- Designed the AI aspect of the project and optimization.

    Challenges: I failed to implement a time limit in A4 and I was unable to implement Zobrist hashing in it too. Thus, this was my first time implementing them both successfully. It was challenging to test my code using python as the move generator function was quite difficult to code and it was needed to completely test my code.

    Benefits: The division of the work was quite clear and each person was responsible for their code. We met from time to time to discuss our ideas for the agent and it was a good strategy as we had to justify every idea we had and it made our implementation smarter.

    B) Edmund Lai:
    Role in Project- Designing the Move generator for the Baroque Chess agent, testing

    Challenges:

    One of the challenges were about being able to meet up and communicate effectively about progress done on the project. Most of the collaboration was done via pushing and pulling code on Github, which was an effective tool for use in this scenario, but

communication could have been a bit better with regard to how far each person could get done within a certain time frame.

Another one of the challenges was self-created due to time constraints. Edmund chose to do the move generation due to that aspect of the project being interesting and cool to implement, but it was quite a large undertaking for one person and required a lot of time and effort, which was made more evident once it became closer to the milestone deadlines.

Benefits:

The benefits are that it was very easy to simply focus on one thing at a time due to each person being responsible for a given task. It was also very beneficial to be able to collaborate and bounce ideas off of one another in order to solve the problem at hand. Additionally, it was an extra motivation because you are not only responsible for your own grade, but also your partner's grade as well, so there is further reason to try your best to complete the assignment as well as you can.