FAI Final Report

Mason Leon and Naveen Kumar Chiluka
Github Link

CS 5100
Foundations of Artificial Intelligence
Dr. Lawson L.S. Wong
Fall 2019

Khoury College of Computer Sciences
Northeastern University
Boston,MA

Contents

1 Introduction

1.1 Problem

Chess has been a challenging game both for human players as well as artificially intelligent computer programs. Although chess is not quite a NP-complete problem due to the fact that it has a defined, although, incredibly large state space of around 10^{50} states which makes it computationally impossible to play a game without any approximation. In order to effectively play chess within the game's official rules and time constraints, a good AI agent requires heuristics for every move as well as cleverness on behalf of the programmer.

1.2 Motivation

The game provides a good opportunity to learn about designing evaluation functions based on the features of the game. This assignment has helped us understand the nuances of chess, its complexities, and the sheer number of approaches computer scientists have undertaken in order to program an AI agent capable of beating a human player. Additionally this project has helped us apply concepts learned throughout the course to an interesting puzzle.

1.3 Objective

To build an AI Chess Agent using Minimax Algorithm, Alpha-beta pruning along with a reasonable evaluation function.

1.4 Problem formulation

- State space: All the valid board configurations.
- Action space: All the valid moves of the pieces.
- Goal state: The state where the AI agent wins, loses or draws.
- Utility or Evaluation function: Heuristic based on key features.

1.5 Ideal Outcome

Build an AI agent using different evaluation functions and analyze the performances of each of these cases.

2 Approach

2.1 Environment

We initially planned to use Javascript environment for the project but as both of us are completely new to front-end development it was taking longer than expected to comfortably code in JS. As we researched implementation of our algorithms, we discovered python-chess, a pure Python chess library with move generation, move validation and support for common chess formats. This project was exciting for us because it allowed us to work on the project in a language we were comfortable with as well as utilize data science tools such as Jupyter Notebooks, Google Colab, and Pandas for running, benchmarking, and performing statistical analysis of our results. Additionally, this library allows for integration of popular chess engines for the purpose of testing our algorithm against another intelligent chess player agent.In reading the python-chess documentation, we were directed to a Jupyter Notebook from Dr. Douglas Blank's CS371: Introduction to Cognitive Science course at Bryn Mawr College. In this starter code we learned about the library's basic Board, Game, and Piece data structures and state representation of the library. We adapted the examples of a random agent, a human agent, naive agent, and their respective evaluation functions for our own project and have set up the chess game environment.

2.2 Evaluation Functions

2.2.1 Random Evaluation Function

We have started with a Random Evaluation function which picks a random move from the available legal moves. As expected, the results were random.

2.2.2 Improved Random Evaluation Function

This is similar to the Random Evaluation Function except that it prefers moves that result in a capture over others.

2.2.3 Naive Evaluation Function

So, after doing some research on the chess evaluation strategies, we discovered Material scores for the board. Material score is a heuristic that computes the utility of a particular state of the board based on the available pieces and the corresponding weights.

$$MaterialScore = \sum_{\forall\, i\, in\, pieces} w_p(i) * (nW(i) - nB(i))$$

where
i = piece type
nW(i) = number of white pieces of type i
nB(i) = number of black pieces of type i

Based on the reasoning provided on this wiki Chess Programming Wiki we decided on the following values for the weights for our material score heuristic:
Pawn = 100
Knight = 320
Bishop = 330
Rook = 500
Queen = 900

2.2.4 Improved Naive Evaluation Function

We found that the Material Score heuristic is of not much help since its effect is seen only when there is a capture. For that to happen the agent's pieces should be in positions where it's possible to capture the opponent's pieces. We observed that the Naive agent was resulting in draws when pitted against a Random Agent more often than not. So, we imparted a little more intelligence to it by adding a couple of additional features like:

- score would be increased by 50 if the move results in a capture.
- score would be increased by 9999 if the move results in a checkmate.
- score would be decreased by 500 if the move results in a check to the current player.

- score would be increased by 900 if the move results in a check to the opponent player.
- score starts from a random seed unlike in the previous evaluation where it started from zero.

Adding these features, showed some positive results in the performance of the agent and resulting in wins most of the time against the Random Agent.

2.2.5 Advanced Evaluation Function

The Improved Naive Agent was a reasonable player and picked better moves over Naive agent but it would still end up picking sub-optimal moves since the features are not enough. In search of a better heuristic we found Piece-Square tables. The tables can be thought of as Reward functions for each of type of piece. There would be positive and negative rewards for each square of the board for every type of piece. So, we created an Advanced Agent which relies on both Material Score and Piece-Square Tables. Please refer to section 6 for further details on the tables used in the project. This agent performed far better than its predecessor against the Random Agent. To illustrate the concept of Piece-Square Tables, consider the following Pawn Table: In the below table, the top rows represent the white side while the bottom two rows represent the black side and the table is used for white pieces.The positive values are rewards for the agent if a pawn is moved to that square while negative values are penalties for moving to that square. This can dictate the movement of pawns in the intended directions.

- As per the table, the central pawns are levied with heavy penalty if they don't move forward and make room for minor pieces and queen. All the pawns are encouraged to make it to the other end so that they can be promoted to more powerful pieces.

```
pawntable = [
    0,  0,  0,  0,  0,  0,  0,  0,
    5, 10, 10, -20, -20, 10, 10, 5,
    5, -5, -10, 0, 0, -10, -5, 5,
    0, 0, 0, 20, 20, 0, 0, 0,
    5, 5, 10, 25, 25, 10, 5, 5,
    10, 10, 20, 30, 30, 20, 10, 10,
    50, 50, 50, 50, 50, 50, 50, 50,
    0, 0, 0, 0, 0, 0, 0, 0]
```

- Similarly, the Knights are encouraged to go to the center as they have the highest degree of freedom there.

```
knightstable = [
    -50, -40, -30, -30, -30, -30, -40, -50,
    -40, -20, 0, 5, 5, 0, -20, -40,
    -30, 5, 10, 15, 15, 10, 5, -30,
    -30, 0, 15, 20, 20, 15, 0, -30,
    -30, 5, 15, 20, 20, 15, 5, -30,
    -30, 0, 10, 15, 15, 10, 0, -30,
    -40, -20, 0, 0, 0, 0, -20, -40,
    -50, -40, -30, -30, -30, -30, -40, -50]
```

- The Bishops are discouraged from moving to corners and borders. They are encouraged to move to the center as they serve better purpose there.

```
bishopstable = [
    -20, -10, -10, -10, -10, -10, -10, -20,
    -10, 5, 0, 0, 0, 0, 5, -10,
    -10, 10, 10, 10, 10, 10, 10, -10,
    -10, 0, 10, 10, 10, 10, 0, -10,
    -10, 5, 5, 10, 10, 5, 5, -10,
    -10, 0, 5, 10, 10, 5, 0, -10,
    -10, 0, 0, 0, 0, 0, 0, -10,
    -20, -10, -10, -10, -10, -10, -10, -20]
```

- The Rooks are discouraged from moving to the edges as they are underutilized there. They are incentivized either to go to the opponent's side or not-penalized for being in the central square.

```
rookstable = [
    0, 0, 0, 5, 5, 0, 0, 0,
    -5, 0, 0, 0, 0, 0, 0, -5,
    -5, 0, 0, 0, 0, 0, 0, -5,
    -5, 0, 0, 0, 0, 0, 0, -5,
    -5, 0, 0, 0, 0, 0, 0, -5,
    -5, 0, 0, 0, 0, 0, 0, -5,
    5, 10, 10, 10, 10, 10, 10, 5,
    0, 0, 0, 0, 0, 0, 0, 0]
```

- The queens have the highest power on the board - as their powers combined that of knights and rooks. So, in order for a player to get the maximum advantage out of a queen, it would be ideal to move them towards the central squares.

```
queenstable = [
    -20, -10, -10, -5, -5, -10, -10, -20,
    -10, 0, 0, 0, 0, 0, 0, -10,
    -10, 5, 5, 5, 5, 5, 0, -10,
    0, 0, 5, 5, 5, 5, 0, -5,
    -5, 0, 5, 5, 5, 5, 0, -5,
    -10, 0, 5, 5, 5, 5, 0, -10,
    -10, 0, 0, 0, 0, 0, 0, -10,
    -20, -10, -10, -5, -5, -10, -10, -20]
```

- The kings should always be secured as the game would end if the king is under a check-mate. The safest and best place for the king would be in the first two ranks of the player's side.

```
kingstable = [
    20, 30, 10, 0, 0, 10, 30, 20,
    20, 20, 0, 0, 0, 0, 20, 20,
    -10, -20, -20, -20, -20, -20, -20, -10,
    -20, -30, -30, -40, -40, -30, -30, -20,
    -30, -40, -40, -50, -50, -40, -40, -30,
    -30, -40, -40, -50, -50, -40, -40, -30,
    -30, -40, -40, -50, -50, -40, -40, -30,
    -30, -40, -40, -50, -50, -40, -40, -30]
```

Note: All the above tables are for the white side, for the black side we need to consider the mirrored values.

2.3 Agents

2.3.1 Random Agent

This agent plays the game using the random evaluation function.

2.3.2 Improved Random Agent

This agent uses Improved Random Evaluation Function.

2.3.3 Naive Agent

This agent uses Naive Evaluation Function.

2.3.4 Improved Naive Agent

This agent uses Improved Naive Evaluation Function.

2.3.5 Advanced Agent

This agent uses Advanced Evaluation Function.

2.3.6 Mini-max Agent

Just the evaluation functions were not enough for our agent to perform well against other intelligent chess agents. The agent needed to look ahead into the future and predict the moves of the opponent and then make decisions. So, we implemented mini-max search algorithm along with each of the evaluation functions.

2.3.7 Mini-max-Alpha-Beta Agent

Although the Mini-max Agent was faring well with other agents, it was taking too long to make a move when the depth of the search was more than 3. To tackle this problem, we improved the Mini-max algorithm by incorporating the Alpha-Beta pruning technique. This technique will help the agent avoid expanding unnecessary moves. We also incorporated endgame table base probing in all the Improved and Advanced Agents so that the agents win certainly. When the number of pieces in the game is 7 or lesser, the agent starts querying the Web API for the next best move. This was by far the best agent we have created.

2.4 Tools/Libraries

Development Environment	Libraries
Jupyter, Pycharm	python-chess
Anaconda	stockfish
Pandas	Syzygy API
Ubuntu 18.04.3 LTS	

3 Results

These are some of the results we obtained when we pitted different agents against each other.

- Naive Agent vs Random Agent

round_nu	iterations	depth	white agent	black agent	white_victory	winner	moves_played	remain_w_pieces	remaining_b_pieces
1	10		naive_agent	random_agent	FALSE	draw: claim	79	14	1
2	10		naive_agent	random_agent	FALSE	draw: claim	39	15	6
3	10		naive_agent	random_agent	FALSE	draw: claim	47	16	1
4	10		naive_agent	random_agent	FALSE	draw: claim	127	11	1
5	10		naive_agent	random_agent	FALSE	draw: claim	137	11	1
6	10		naive_agent	random_agent	FALSE	draw: claim	57	15	1
7	10		naive_agent	random_agent	FALSE	draw: claim	121	12	1
8	10		naive_agent	random_agent	FALSE	draw: claim	57	13	1
9	10		naive_agent	random_agent	FALSE	draw: claim	103	14	2
10	10		naive_agent	random_agent	FALSE	draw: claim	115	11	1

Most of the games result in a draw because of some of the obscure draw-rules in chess. We believe it is either due to fifty-move-rule or three-fold-repetition.

- Improved Naive Agent vs Random Agent

round_nu	iterations	depth	white agent	black agent	white_victory	winner	moves_played	remain_w_pieces	remaining_b_pieces
1	10		improved_agent	random_agent	FALSE	draw: claim	125	5	1
2	10		improved_agent	random_agent	FALSE	draw: claim	128	11	1
3	10		improved_agent	random_agent	FALSE	draw: claim	113	6	1
4	10		improved_agent	random_agent	FALSE	draw: claim	189	4	2
5	10		improved_agent	random_agent	TRUE	checkmate: White wins!	63	14	4
6	10		improved_agent	random_agent	TRUE	checkmate: White wins!	63	10	4
7	10		improved_agent	random_agent	TRUE	checkmate: White wins!	75	8	3
8	10		improved_agent	random_agent	TRUE	checkmate: White wins!	75	11	2
9	10		improved_agent	random_agent	TRUE	checkmate: White wins!	67	12	1
10	10		improved_agent	random_agent	TRUE	checkmate: White wins!	77	12	4

The Improved Agent wins almost 6 out of 10 times because of the additional features provided to the agent.

- Advanced Agent vs Random Agent

round_nu	iterations	depth	white agent	black agent	white_victory	winner	moves_played	remain_w_pieces	remaining_b_pieces
1	10		advanced_agent	random_agent	TRUE	checkmate: White wins!	45	12	3
2	10		advanced_agent	random_agent	TRUE	checkmate: White wins!	43	13	3
3	10		advanced_agent	random_agent	TRUE	checkmate: White wins!	61	15	1
4	10		advanced_agent	random_agent	TRUE	checkmate: White wins!	67	15	1
5	10		advanced_agent	random_agent	TRUE	checkmate: White wins!	63	12	1
6	10		advanced_agent	random_agent	TRUE	checkmate: White wins!	61	14	1
7	10		advanced_agent	random_agent	TRUE	checkmate: White wins!	37	13	3
8	10		advanced_agent	random_agent	TRUE	checkmate: White wins!	17	15	12
9	10		advanced_agent	random_agent	TRUE	checkmate: White wins!	39	13	3
10	10		advanced_agent	random_agent	TRUE	checkmate: White wins!	69	13	1

The Advanced Agent wins almost all the times owing to its added intelligence in the form of piece-square tables.

- Advanced-Mini-max Agent vs Random Agent

round_nu	iterations	depth	white agent	black agent	white_victory	winner	moves_played	remain_w_pieces	remaining_b_pieces
1	10	1	advanced_minimax_agent	random_agent	FALSE	draw: stalemate	57	13	2
2	10	1	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	19	16	15
3	10	1	advanced_minimax_agent	random_agent	FALSE	draw: stalemate	63	15	1
4	10	1	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	19	16	12
5	10	1	advanced_minimax_agent	random_agent	FALSE	draw: stalemate	53	15	2
6	10	1	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	39	15	8
7	10	1	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	25	16	12
8	10	1	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	33	16	10
9	10	1	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	43	15	3
10	10	1	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	49	15	3
1	10	2	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	37	16	8
2	10	2	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	49	15	6
3	10	2	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	11	16	16
4	10	2	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	21	15	11
5	10	2	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	15	16	12
6	10	2	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	39	14	10
7	10	2	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	31	14	10
8	10	2	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	21	16	15
9	10	2	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	25	15	13
10	10	2	advanced_minimax_agent	random_agent	TRUE	checkmate: White wins!	49	14	9

The Advanced-Mini-max Agent wins almost every time when the depth is 2 and 7 out of 10 times when depth is 1. As depth increases, the agent's performance against Random Agent gets better.

- Advanced-Mini-max Agent vs Stockfish

round_nu	iterations	depth	white agent	black agent	white_vict	winner	moves_played	remain_w_pieces	remaining_b_pieces
1	10	1	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	56	7	11
2	10	1	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	46	8	11
3	10	1	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	44	7	11
4	10	1	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	56	6	10
5	10	1	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	30	10	12
6	10	1	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	56	3	10
7	10	1	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	48	6	10
8	10	1	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	48	6	9
9	10	1	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	46	9	11
10	10	1	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	54	5	9
1	10	2	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	66	6	9
2	10	2	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	48	7	11
3	10	2	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	50	9	11
4	10	2	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	70	8	9
5	10	2	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	62	4	9
6	10	2	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	46	8	11
7	10	2	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	54	7	12
8	10	2	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	42	10	11
9	10	2	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	76	4	9
10	10	2	advanced_minimax_agent	stockfish	FALSE	checkmate: Black wins!	28	12	14

The Advanced-Mini-max Agent loses all the time to Stockfish. We couldn't quite figure out the skill level of the Stockfish agent in this game. Setting the skill level of the Stockfish Agent and then pitting against our agents would give us better insights into the performance of our agents.

- Advanced-Mini-max-Alpha-Beta Agent vs Random Agent

round_nu	iterations	depth	white agent	black agent	white_victory	winner	moves_played	remain_w_pieces	remaining_b_pieces
1	10	1	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	37	15	4
2	10	1	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	45	15	5
3	10	1	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	25	14	11
4	10	1	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	27	15	9
5	10	1	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	35	15	9
6	10	1	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	31	15	7
7	10	1	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	37	16	6
8	10	1	advanced_alpha-beta_minimax_agent	random_agent	FALSE	draw: stalemate	59	15	1
9	10	1	advanced_alpha-beta_minimax_agent	random_agent	FALSE	draw: stalemate	65	15	4
10	10	1	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	37	16	6
1	10	2	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	39	14	4
2	10	2	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	47	14	5
3	10	2	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	41	15	4
4	10	2	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	35	15	7
5	10	2	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	61	13	1
6	10	2	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	31	14	10
7	10	2	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	77	11	1
8	10	2	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	27	14	8
9	10	2	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	49	13	3
10	10	2	advanced_alpha-beta_minimax_agent	random_agent	TRUE	checkmate: White wins!	27	15	9

As expected, the Advanced-Mini-max-Alpha-Beta Agent wins 10 out of 10 times when depth is 2 and 8 out of 10 times when the depth is 1.

- Advanced-Mini-max-Alpha-Beta Agent vs Stockfish

round_nu	iterations	depth	white agent	black agent	white_victory	winner	moves_played	remain_w_pieces	remaining_b_pieces
1	10	1	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	30	10	12
2	10	1	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	56	7	11
3	10	1	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	44	6	13
4	10	1	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	44	7	13
5	10	1	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	68	3	11
6	10	1	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	30	10	12
7	10	1	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	42	6	11
8	10	1	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	58	6	11
9	10	1	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	48	6	10
10	10	1	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	36	7	12
1	10	2	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	36	7	11
2	10	2	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	54	2	13
3	10	2	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	38	6	14
4	10	2	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	32	6	14
5	10	2	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	36	6	14
6	10	2	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	38	6	14
7	10	2	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	56	2	13
8	10	2	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	36	6	14
9	10	2	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	38	6	12
10	10	2	advanced_alpha-beta_minimax_agent	stockfish	FALSE	checkmate: Black wins!	38	6	14

The Advanced-Mini-max-Alpha-Beta Agent loses all the time to Stockfish. This needs further investigation into the Stockfish internal algorithm and features used so that we can improve our agents.

The Advanced-Mini-max-Alpha-Beta Agent is now improved due to the Advanced Evaluation Function in the following ways:

- Reduced number of moves required to win
- Reduced overall losses (game ending in either a win or a draw)

4 Problems Faced

- Analyzing the game and deciding on the features was tougher than we thought.
- Due to the space and time complexity of the game, it was difficult to debug the game as every game would take long time to run till the end depending on the evaluation function. We could use PyCharm for debugging but it was not a feasible option for us due to the limitations of our local machines. So, our only option was to run on google co-lab servers but the debugging features of the Jupyter notebook were limited and not very helpful.
- We encountered difficulties while integrating our agent with another trained, intelligent chess agent like Stockfish. The problem was the Jupyter support with File IO. We had permission errors while uploading the executable file of the Stockfish engine. Adding to this, we each had a different OS. So, we fixed this issue by using an Ubuntu-based Virtual Machine as the common platform for the project. We compiles a Stockfish executable from its binary and used this in our Linux based VMs.
- There was confusion over obscure chess rules automatically built into the library resulting in draws. The documentation for Python-Chess did not have the clearest examples of every function's use and purpose.
- We chose to use a Syzygy endgame table base API because we did not have enough local storage for the 7-man endgame table-base (ie: all combinations starting from 7 pieces left to end of game) as this required 17 TB of SSD space. However, by querying the Syzygy endgame table base API we became bound by the network and poor documentation that did not explain the rate limits. The HTTP request would throw *Error 429 Too many requests* at times. This happens when our agent reaches the endgame and hits the API too many times in a given time frame while evaluating moves. This was fixed by adding a Retry block whenever it faced such an error. The agent would retry after waiting for sometime. Although we address this with error handling in our code, in practice we timed out the server by requesting too many calls to the API and we were unable to finish a single game starting from a 7-man endgame.

5 Future Work

Over the course of the project, we learned a lot of things ranging from collaboration, research, debugging, structuring the code, etc. Although we have achieved the set objective, there is a lot of room for building on this project.

- After seeing the results of our agents with each other as well as Stockfish, we realized that improvement of the Mini-max would would be unfeasible for us as depths of 3 or higher could take hours when generating data for multiple games.
- The heuristics we have used are static throughout the game. Since the game strategy and tactics change as the game progresses, using multiple heuristics for different stages of the game would yield better results. An example would be to use different weights in the material score heuristic for each of start game, middle game and end game stages. We could also use other "centipawn" heuristic values from famous chess Grandmasters and computer scientists.
- Using chess opening playbooks would be a good idea to start the game by an established and majorly used move over a random one.
- Alpha-beta pruning works better when the move ordering is right. This is something that needs to be explored further to reduce the time complexity of the search.
- It would be interesting to see how an Expectimax Search would perform in this game. The probability model can be designed in such a way as to avoid sub-optimal moves. We haven't given much thought to this but would very much like to try it out in our free time in the future.
- The Monte Carlo algorithm along with some sort of learning approach could possibly be an interesting approach to eliminating some specific moves or scenarios that may be less probable for an agent to win.
- All of our agents lost to the Stockfish Agent. We didn't really understand the skill level of the Stockfish Agent our agents were playing against. Additionally, the time Stockfish took to calculate and make a superior move to our agent was vastly superior to our approach. We can gain better insights into the performance of our agents if we can set the skill level of Stockfish and then pit our agents against it.
- As our program utilized a single core, we believe there is great performance efficiency to be possibly gained through the use of multi-processing or multi-threaded programming. This is a suggested approach by the Python-Chess library however, neither of us have experience in this field, and this is a future enhancement.

Note: Please refer to the README.md in the project code base (Github Link) to understand how to run the program and integrate with stockfish agent.

6 References

- Pure Python chess library - Github repository `https://github.com/niklasf/python-chess`
- Pure Python chess library. Read the Docs. `https://python-chess.readthedocs.io/en/latest/index.html`
- Chess Strategies `https://en.wikipedia.org/wiki/Chess_strategy`
- `https://www.freecodecamp.org/news/simple-chess-ai-step-by-step-1d55a9266977/`

- https://static.aminer.org/pdf/PDF/000/226/325/genetically_programmed_strategies_for_chess_endgame.pdf
- Piece-Square tables and Material Score https://www.chessprogramming.org/Simplified_Evaluation_Function
- Stockfish Github repository https://github.com/official-stockfish/Stockfish
- Endgame table bases https://syzygy-tables.info/

final_report_stats

December 13, 2019

1 ai-chess-agent statistics

[1]:
```
import numpy as np
import pandas as pd
import csv
import os
import matplotlib.pyplot as plt
```

[2]:
```
def get_csv_paths(dir):
    files = dict()
    for (dirpath, dirnames, filenames) in os.walk(dir):
        for file in filenames:
            if file.endswith(".csv"):
                files[file] = os.path.join(dirpath, file)
    file_list = []
    for item in sorted(files.keys()):
        file_list.append([files[item], item.split(".")[0]])

    return file_list
```

[3]:
```
f_list = get_csv_paths("./../src/driver_notebooks/results/")
COLUMNS = ['round_num','iterations', 'depth', 'white_agent','black_agent',
           'white_victory','winner','moves_played','remaining_w_pieces',
           'remaining_b_pieces', 'remaining_tot_pieces']
```

[4]:
```
df_from_each_file = (pd.read_csv(f[0], names=COLUMNS, header=0) for f in f_list)
```

[5]:
```
df = pd.concat(df_from_each_file, ignore_index=True)
```

1.0.1 Total number of games played

[6]:
```
games = df['round_num'].count()
games
```

[6]: 340

[7]:
```
white_checkmate_df = df.apply(lambda x: True if x['winner'] == 'checkmate:␣
↪White wins!' else False , axis=1)
white_checkmate_num = len(white_checkmate_df[white_checkmate_df == True].index)
```

[8]:
```
black_checkmate_df = df.apply(lambda x: True if x['winner'] == 'checkmate:␣
↪Black wins!' else False , axis=1)
black_checkmate_num = len(black_checkmate_df[black_checkmate_df == True].index)
```

[9]:
```
draw_stalemate_df = df.apply(lambda x: True if x['winner'] == "draw: stalemate"␣
↪else False , axis=1)
draw_stalemate_num = len(draw_stalemate_df[draw_stalemate_df == True].index)
```

[10]:
```
draw_fivefold_df = df.apply(lambda x: True if x['winner'] == "draw: 5-fold␣
↪repetition" else False , axis=1)
draw_fivefold_num = len(draw_fivefold_df[draw_fivefold_df == True].index)
```

[11]:
```
draw_insufficient_material_df = df.apply(lambda x: True if x['winner'] == "draw:
↪ insufficient material" else False , axis=1)
draw_insufficient_material_num =␣
↪len(draw_insufficient_material_df[draw_insufficient_material_df == True].
↪index)
```

[12]:
```
draw_claim_df = df.apply(lambda x: True if x['winner'] == "draw: claim" else␣
↪False , axis=1)
draw_claim_num = len(draw_claim_df[draw_claim_df == True].index)
```

1.0.2 Overall results

[13]:
```
winner_df = pd.DataFrame([(white_checkmate_num, black_checkmate_num,␣
↪draw_stalemate_num, draw_fivefold_num, draw_insufficient_material_num,␣
↪draw_claim_num)],
columns = ['checkmate: White', 'checkmate: Black', 'draw: stalemate', 'draw:␣
↪5-fold repetition', 'draw: insufficient material', 'draw: claim'])
winner_df.style.hide_index()
```

[13]:
```
<pandas.io.formats.style.Styler at 0x7fa39890d2e8>
```

1.0.3 Overall Percentages

[14]:
```
winnings = df['winner']
counts = winnings.value_counts()
percent = winnings.value_counts(normalize=True)
percent100 = winnings.value_counts(normalize=True).mul(100).round(1).
↪astype(str) + '%'
win_df = pd.DataFrame({'counts': counts, 'percent': percent, 'percent 100':␣
↪percent100 })
```

```
win_df
```

[14]:

	counts	percent	percent 100
checkmate: Black wins!	172	0.505882	50.6%
checkmate: White wins!	106	0.311765	31.2%
draw: claim	38	0.111765	11.2%
draw: stalemate	22	0.064706	6.5%
draw: insufficient material	2	0.005882	0.6%

1.0.4 Top 10 games, ordered by moves played ascending

[15]:
```
df.sort_values(by=['moves_played'], inplace=False, ascending=True).head(10)
```

[15]:

	round_num	iterations	depth	white_agent	black_agent
323	4	10	NaN	random_agent	stockfish
146	7	10	1.0	improved_minimax_agent	random_agent
334	5	10	NaN	improved_random_agent	stockfish
12	3	10	2.0	advanced_minimax_agent	random_agent
144	5	10	1.0	improved_minimax_agent	random_agent
198	9	10	2.0	improved_minimax_agent	random_agent
158	9	10	2.0	improved_minimax_agent	random_agent
191	2	10	2.0	improved_minimax_agent	random_agent
320	1	10	NaN	random_agent	stockfish
14	5	10	2.0	advanced_minimax_agent	random_agent

	white_victory	winner	moves_played	remaining_w_pieces
323	False	checkmate: Black wins!	6	16
146	True	checkmate: White wins!	7	16
334	False	checkmate: Black wins!	8	15
12	True	checkmate: White wins!	11	16
144	True	checkmate: White wins!	11	16
198	True	checkmate: White wins!	13	16
158	True	checkmate: White wins!	13	16
191	True	checkmate: White wins!	13	16
320	False	checkmate: Black wins!	14	14
14	True	checkmate: White wins!	15	16

	remaining_b_pieces	remaining_tot_pieces
323	16	32
146	16	32
334	14	29
12	16	32
144	13	29
198	15	31
158	15	31
191	15	31
320	16	30

```
14                    12                     28
```

1.0.5 Top 10 games where the white agent wins, ordered by moves played ascending

[16]:
```python
df.loc[df['winner'] == 'checkmate: White wins!'].
↪sort_values(by=['moves_played'], inplace=False, ascending=True).head(10)
```

[16]:
```
     round_num  iterations  depth            white_agent   black_agent  \
146          7          10    1.0  improved_minimax_agent  random_agent
144          5          10    1.0  improved_minimax_agent  random_agent
12           3          10    2.0  advanced_minimax_agent  random_agent
158          9          10    2.0  improved_minimax_agent  random_agent
191          2          10    2.0  improved_minimax_agent  random_agent
198          9          10    2.0  improved_minimax_agent  random_agent
14           5          10    2.0  advanced_minimax_agent  random_agent
107          8          10    NaN          advanced_agent  random_agent
1            2          10    1.0  advanced_minimax_agent  random_agent
3            4          10    1.0  advanced_minimax_agent  random_agent

     white_victory                 winner  moves_played  remaining_w_pieces  \
146           True  checkmate: White wins!             7                  16
144           True  checkmate: White wins!            11                  16
12            True  checkmate: White wins!            11                  16
158           True  checkmate: White wins!            13                  16
191           True  checkmate: White wins!            13                  16
198           True  checkmate: White wins!            13                  16
14            True  checkmate: White wins!            15                  16
107           True  checkmate: White wins!            17                  15
1             True  checkmate: White wins!            19                  16
3             True  checkmate: White wins!            19                  16

     remaining_b_pieces  remaining_tot_pieces
146                  16                    32
144                  13                    29
12                   16                    32
158                  15                    31
191                  15                    31
198                  15                    31
14                   12                    28
107                  12                    27
1                    15                    31
3                    12                    28
```

1.0.6 Games where the white agent wins and Oppnent Agent is Stockfish Engine, ordered by moves played ascending

[17]:
```
df[(df['winner'] == 'checkmate: White wins!') & (df['black_agent'] ==␣
↪'stockfish')]
```

[17]:
```
Empty DataFrame
Columns: [round_num, iterations, depth, white_agent, black_agent, white_victory,
winner, moves_played, remaining_w_pieces, remaining_b_pieces,
remaining_tot_pieces]
Index: []
```

1.0.7 Top 10 games with the fewest remaining black pieces, ordered by pieces remaining ascending

[18]:
```
df.sort_values(by=['remaining_b_pieces'], inplace=False, ascending=True).
↪head(10)
```

[18]:
```
     round_num  iterations  depth                      white_agent  \
109         10          10    NaN                   advanced_agent
315          6          10    NaN            improved_random_agent
312          3          10    NaN            improved_random_agent
86           7          10    NaN                      naive_agent
87           8          10    NaN                      naive_agent
268          9          10    1.0     naive_alpha-beta_minimax_agent
56           7          10    2.0  advanced_alpha-beta_minimax_agent
89          10          10    NaN                      naive_agent
54           5          10    2.0  advanced_alpha-beta_minimax_agent
47           8          10    1.0  advanced_alpha-beta_minimax_agent

      black_agent  white_victory                  winner  moves_played  \
109  random_agent           True  checkmate: White wins!            69
315  random_agent          False         draw: stalemate           101
312  random_agent          False         draw: stalemate           149
86   random_agent          False             draw: claim           121
87   random_agent          False             draw: claim            57
268  random_agent          False         draw: stalemate           129
56   random_agent           True  checkmate: White wins!            77
89   random_agent          False             draw: claim           115
54   random_agent           True  checkmate: White wins!            61
47   random_agent          False         draw: stalemate            59

     remaining_w_pieces  remaining_b_pieces  remaining_tot_pieces
109                  13                   1                    14
315                  13                   1                    14
312                  11                   1                    12
86                   12                   1                    13
```

```
87                    13                    1                    14
268                   15                    1                    16
56                    11                    1                    12
89                    11                    1                    12
54                    13                    1                    14
47                    15                    1                    16
```

1.0.8 Top 10 games with fewest remaining black pieces where white wins, ordered by black pieces remaining ascending

[19]:
```
df[(df['winner'] == 'checkmate: White wins!')].
↪sort_values(by=['remaining_b_pieces'], inplace=False, ascending=True).
↪head(10)
```

[19]:
```
     round_num  iterations  depth                      white_agent  \
310          1          10    NaN            improved_random_agent
105          6          10    NaN                   advanced_agent
104          5          10    NaN                   advanced_agent
103          4          10    NaN                   advanced_agent
102          3          10    NaN                   advanced_agent
98           9          10    NaN                   improved_agent
56           7          10    2.0  advanced_alpha-beta_minimax_agent
54           5          10    2.0  advanced_alpha-beta_minimax_agent
109         10          10    NaN                   advanced_agent
148          9          10    1.0           improved_minimax_agent

      black_agent  white_victory                  winner  moves_played  \
310  random_agent           True  checkmate: White wins!           291
105  random_agent           True  checkmate: White wins!            61
104  random_agent           True  checkmate: White wins!            63
103  random_agent           True  checkmate: White wins!            67
102  random_agent           True  checkmate: White wins!            61
98   random_agent           True  checkmate: White wins!            67
56   random_agent           True  checkmate: White wins!            77
54   random_agent           True  checkmate: White wins!            61
109  random_agent           True  checkmate: White wins!            69
148  random_agent           True  checkmate: White wins!           125

     remaining_w_pieces  remaining_b_pieces  remaining_tot_pieces
310                   9                   1                    10
105                  14                   1                    15
104                  12                   1                    13
103                  15                   1                    16
102                  15                   1                    16
98                   12                   1                    13
56                   11                   1                    12
54                   13                   1                    14
```

109	13	1	14
148	12	1	13

1.0.9 Top 10 games with the fewest remaining white pieces, ordered by pieces remaining ascending

[20]:
```
df.sort_values(by=['remaining_w_pieces'], inplace=False, ascending=True).
↪head(10)
```

[20]:

	round_num	iterations	depth	white_agent
301	2	10	NaN	random_agent
306	7	10	NaN	random_agent
309	10	10	NaN	random_agent
303	4	10	NaN	random_agent
76	7	10	2.0	advanced_alpha-beta_minimax_agent
135	6	10	NaN	advanced_agent
305	6	10	NaN	random_agent
302	3	10	NaN	random_agent
308	9	10	NaN	random_agent
307	8	10	NaN	random_agent

	black_agent	white_victory	winner	moves_played
301	random_agent	False	draw: stalemate	164
306	random_agent	False	draw: insufficient material	416
309	random_agent	False	draw: insufficient material	519
303	random_agent	False	draw: stalemate	266
76	stockfish	False	checkmate: Black wins!	56
135	stockfish	False	checkmate: Black wins!	54
305	random_agent	False	draw: claim	394
302	random_agent	False	draw: claim	497
308	random_agent	False	draw: claim	297
307	random_agent	False	draw: claim	410

	remaining_w_pieces	remaining_b_pieces	remaining_tot_pieces
301	1	7	8
306	1	2	3
309	1	2	3
303	1	6	7
76	2	13	15
135	2	11	13
305	2	2	4
302	2	1	3
308	2	3	5
307	2	3	5

1.0.10 Top 10 games with fewest remaining white pieces where black wins, ordered by white pieces remaining ascending

[21]:
```
df[(df['winner'] == 'checkmate: Black wins!')].
↪sort_values(by=['remaining_w_pieces'], inplace=False, ascending=True).
↪head(10)
```

[21]:

	round_num	iterations	depth	white_agent
135	6	10	NaN	advanced_agent
71	2	10	2.0	advanced_alpha-beta_minimax_agent
76	7	10	2.0	advanced_alpha-beta_minimax_agent
64	5	10	1.0	advanced_alpha-beta_minimax_agent
176	7	10	2.0	improved_minimax_agent
25	6	10	1.0	advanced_minimax_agent
200	1	10	1.0	improved_minimax_agent
206	7	10	1.0	improved_minimax_agent
132	3	10	NaN	advanced_agent
179	10	10	2.0	improved_minimax_agent

	black_agent	white_victory	winner	moves_played
135	stockfish	False	checkmate: Black wins!	54
71	stockfish	False	checkmate: Black wins!	54
76	stockfish	False	checkmate: Black wins!	56
64	stockfish	False	checkmate: Black wins!	68
176	stockfish	False	checkmate: Black wins!	72
25	stockfish	False	checkmate: Black wins!	56
200	stockfish	False	checkmate: Black wins!	50
206	stockfish	False	checkmate: Black wins!	52
132	stockfish	False	checkmate: Black wins!	50
179	stockfish	False	checkmate: Black wins!	52

	remaining_w_pieces	remaining_b_pieces	remaining_tot_pieces
135	2	11	13
71	2	13	15
76	2	13	15
64	3	11	14
176	3	8	11
25	3	10	13
200	3	10	13
206	3	10	13
132	3	9	12
179	4	11	15