# BIL441 Artificial Intelligence Project Report
# Chess Engine

Nevzat Umut Demirseren
Department of Computer Sciences
TOBB University of Economics and Technology
Ankara, Turkey
nevzatumut2001@gmail.com

***Abstract* – In chess engine models developed using Artificial Intelligence, multiple approaches are used to determine the best move with the usage of heuristics. This paper gives information about the Minimax algorithm in chess engines and how it can be improved using alpha-beta pruning, move ordering and killer heuristics. Without the help of Machine Learning, AI models using Minimax algorithm are compared on speed metric since all Minimax models give the same answer if given enough time.**

***Keywords* – Python, Artificial Intelligence, Chess Engine, Minimax Algorithm, Alpha-beta pruning, Move ordering, Killer Heuristics**

## I. INTRODUCTION

Chess is a two-player strategy board game played on a checkered board with 64 squares arranged in an 8x8 grid. Each player starts with 16 pieces that are used to attack and defend their opponent's pieces and ultimately capture their opponent's king. The objective of the game is to checkmate the opponent's king, which means to put it in a position where it is under attack and there is no legal move to escape the attack.

Chess engines which are using Artificial Intelligence only mostly use the Minimax approach. This approach determines the best move calculating which move is the most logical for the opponent to play by scoring each possible move considering how the game will go on. Chess engines try to optimize this function by using various methods such as alpha-beta pruning, move ordering and heuristics. ***In this model, the aim is to determine if a more aggressive approach works faster than a model which tries to determine deeper levels of the game.*** This model prioritizes the taking moves using a look-up table and memorizes the previously calculated moves according to evaluations.

*This implementation is written using Python and python-chess Chess library.*

## II. BUILDING THE MODEL

The model uses various techniques to determine the best move and to evaluate the current position of the game.

### A. Board Evaluation

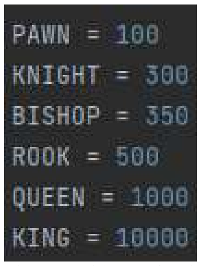
The essential part of chess engines is the evaluation part. Each chess engine uses unique techniques to evaluate the position on the board. Most successful chess engines use a hybrid model where each piece is evaluated according to the position with additional parameters (such as if the rooks are connected) and Machine Learning where the engine checks if the game could lead to a win using the game data set.

This model uses a function to evaluate the position by counting the pieces and multiplying each piece count by the piece coefficient using a look-up table. Also, each piece has a bonus depending on the position. For example, knights are stronger if they are placed in the center of the board. However, they are not as valuable if they are closer to the corners. This is determined by using a two-dimensional array and adding the value depending on the value. Each piece's value is calculated and the sum is the score of that player.

W = The sum of white pieces' score

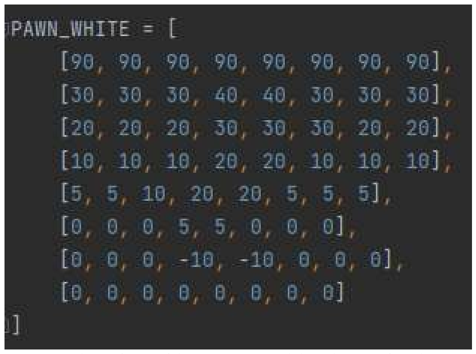
B = The sum of black pieces' score

Evaluation = W - B



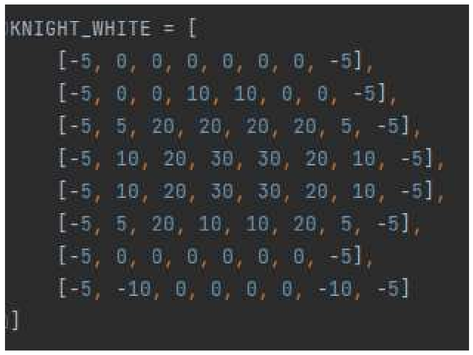
```
PAWN = 100
KNIGHT = 300
BISHOP = 350
ROOK = 500
QUEEN = 1000
KING = 10000
```

Evaluations used in Evaluations.py



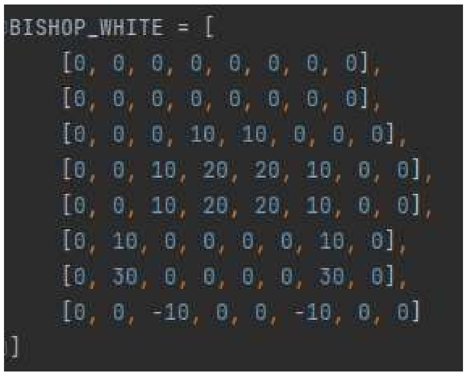
```
PAWN_WHITE = [
    [90, 90, 90, 90, 90, 90, 90, 90],
    [30, 30, 30, 40, 40, 30, 30, 30],
    [20, 20, 20, 30, 30, 30, 20, 20],
    [10, 10, 10, 20, 20, 10, 10, 10],
    [5, 5, 10, 20, 20, 5, 5, 5],
    [0, 0, 0, 5, 5, 0, 0, 0],
    [0, 0, 0, -10, -10, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
]
```

Bonus Points for White Pawn Position in Evaluations.py



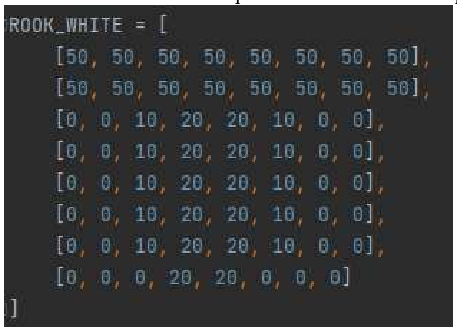
```
KNIGHT_WHITE = [
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [-5, 0, 0, 10, 10, 0, 0, -5],
    [-5, 5, 20, 20, 20, 20, 5, -5],
    [-5, 10, 20, 30, 30, 20, 10, -5],
    [-5, 10, 20, 30, 30, 20, 10, -5],
    [-5, 5, 20, 10, 10, 20, 5, -5],
    [-5, 0, 0, 0, 0, 0, 0, -5],
    [-5, -10, 0, 0, 0, 0, -10, -5]
]
```

Bonus Points for White Knight Position in Evaluations.py



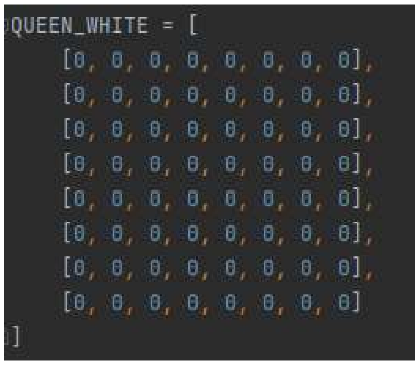
```
BISHOP_WHITE = [
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 10, 10, 0, 0, 0],
    [0, 0, 10, 20, 20, 10, 0, 0],
    [0, 0, 10, 20, 20, 10, 0, 0],
    [0, 10, 0, 0, 0, 0, 10, 0],
    [0, 30, 0, 0, 0, 0, 30, 0],
    [0, 0, -10, 0, 0, -10, 0, 0]
]
```

Bonus Points for White Bishop Position in Evaluations.py



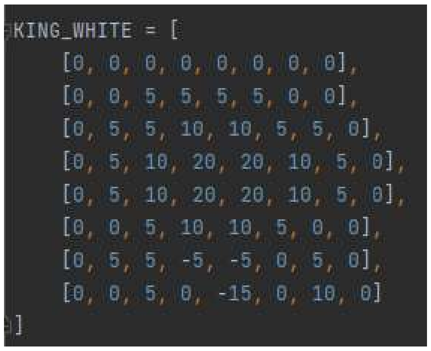
```
ROOK_WHITE = [
    [50, 50, 50, 50, 50, 50, 50, 50],
    [50, 50, 50, 50, 50, 50, 50, 50],
    [0, 0, 10, 20, 20, 10, 0, 0],
    [0, 0, 10, 20, 20, 10, 0, 0],
    [0, 0, 10, 20, 20, 10, 0, 0],
    [0, 0, 10, 20, 20, 10, 0, 0],
    [0, 0, 10, 20, 20, 10, 0, 0],
    [0, 0, 0, 20, 20, 0, 0, 0]
]
```

Bonus Points for White Rook Position in Evaluations.py



```
QUEEN_WHITE = [
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 0, 0, 0, 0, 0, 0]
]
```

Bonus Points for White Queen Position in Evaluations.py



```
KING_WHITE = [
    [0, 0, 0, 0, 0, 0, 0, 0],
    [0, 0, 5, 5, 5, 5, 0, 0],
    [0, 5, 5, 10, 10, 5, 5, 0],
    [0, 5, 10, 20, 20, 10, 5, 0],
    [0, 5, 10, 20, 20, 10, 5, 0],
    [0, 0, 5, 10, 10, 5, 0, 0],
    [0, 5, 5, -5, -5, 0, 5, 0],
    [0, 0, 5, 0, -15, 0, 10, 0]
]
```

Bonus Points for White King Position in Evaluations.py

*The evaluations for the black pieces are the reversed versions of the arrays.*

After this calculation is completed, the fastest checkmating move is awarded with a bonus (which is a variable) and a depth coefficient.

## B. *Check Openings*

The chess engine uses a library of openings at the beginning of the game. If a certain opening is played by the other player, the engine continues to play the opening in the theory. Each opening has the resulting Stockfish evaluation in the openings file and compares for the best opening possible.

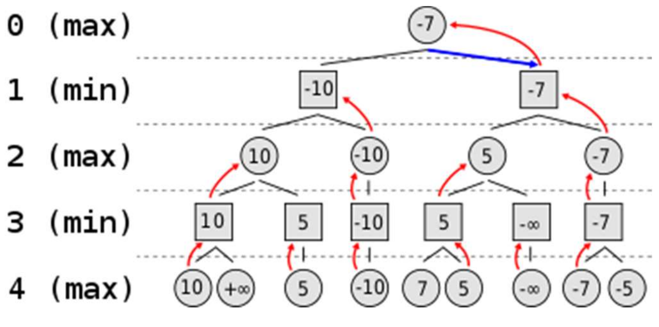
The engine has 14 openings in the library (in order):

- Italian Opening
  - Giuoco Piano Variant
  - Two Knights Defense
- Ruy Lopez Opening
  - Berlin Defense
  - Morphy Defense
- Caro Kann Opening
  - Exchange Variation
  - Nf3 Defense
- Queen's Gambit Declined
- Queen's Gambit Accepted
- Indian Defense
- London System
  - Gambit Declined
  - Gambit Accepted
- French Defense
- King's Gambit Accepted
- King's Gambit Declined

This list contains the 14 most played openings in online chess games. This list can be expanded but the likelihood of changing performance is small.

## C. *Minimax Algorithm*

The minimax algorithm is a decision-making algorithm used in two-player games, which assumes both players are playing optimally. The algorithm explores all possible moves and evaluates them based on their effect on the outcome of the game. It creates a game tree and alternates between maximizing and minimizing the player's chances of winning until a terminal state is reached, assigning a score to each leaf node based on whether it represents a win, loss, or draw. The algorithm then chooses the move that leads to the highest score when the player is trying to maximize their chances of winning, and the lowest score when the player is trying to minimize their chances of losing, and the game continues until a terminal state is reached.



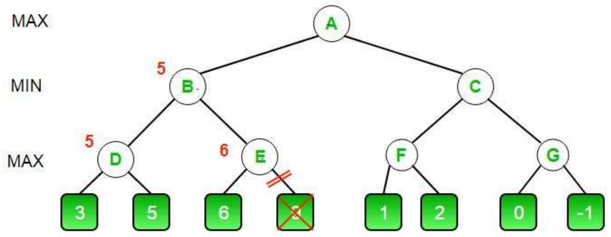
Example implementation of Minimax Algorithm

The implementation used in the chess engine evaluates the position using a previously defined function piece count, piece values and piece positions.

On the model, the move is calculated 5 plies deep which means the minimax algorithm is programmed to calculate the best move according to 5 moves ahead. Chess engines try to maximize this parameter by staying in the standard calculation times (in an average 10-minute chess game each move takes 30 seconds to 1 minute). The best chess engines which use the Minimax Algorithm with a hybrid approach to Machine Learning Heuristics and other search algorithms such as Monte Carlo Tree Search can go up to 50 ply deep using cloud computing. However, for a naïve model which only utilizes the CPU and does not use cloud computing 5 ply depth is considered good. Further comparison is made accordingly.

## D. *Alpha-beta Pruning for the Minimax Algorithm*

Alpha-beta pruning is a technique used in the minimax algorithm to reduce the number of nodes that need to be evaluated during the search process. It works by cutting off branches of the search tree that are guaranteed to be worse than other branches that have already been evaluated. The algorithm maintains two values, alpha and beta, which represent the best score found so far for the maximizing player and the minimizing player, respectively. As the search progresses, if the algorithm finds a move that leads to a score worse than the current alpha or beta value, it can prune that branch of the search tree and move on to the next branch, without evaluating the remaining nodes in that branch. This can greatly reduce the number of nodes that need to be evaluated, making the search process more efficient and faster.

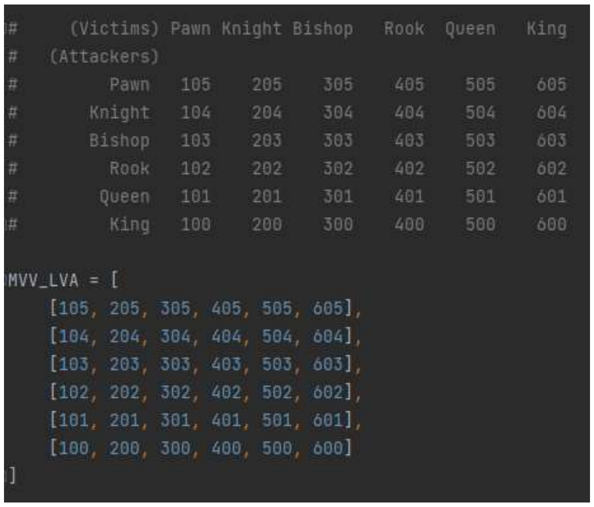


Example implementation of Alpha-beta Pruning

In this implementation, alpha-beta pruning prunes the nodes of the Minimax Tree where a better result is found.

## E. *Move Ordering using MVV-LVA*

Move ordering using MVV-LVA (Most Valuable Victim – Least Valuable Aggressor) is a commonly used approach in chess engines using naïve AI. This module sorts the moves by prioritizing the taking moves. Each taking move has a score depending on the taking piece and taken piece. If the taking piece's value is low and the taken piece's value is high, the move has more priority.



```
#     (Victims) Pawn Knight Bishop   Rook  Queen   King
#   (Attackers)
#         Pawn   105    205    305    405    505    605
#       Knight   104    204    304    404    504    604
#       Bishop   103    203    303    403    503    603
#         Rook   102    202    302    402    502    602
#        Queen   101    201    301    401    501    601
#         King   100    200    300    400    500    600

MVV_LVA = [
    [105, 205, 305, 405, 505, 605],
    [104, 204, 304, 404, 504, 604],
    [103, 203, 303, 403, 503, 603],
    [102, 202, 302, 402, 502, 602],
    [101, 201, 301, 401, 501, 601],
    [100, 200, 300, 400, 500, 600]
]
```

MVV-LVA Look-up Table in Evaluations.py

Additionally, for quiet moves if there exists a queen promotion the move also has priority.

## F. *Killer Heuristic*

The Killer Heuristic is a technique used in the alpha-beta pruning algorithm to improve its efficiency by reducing the number of nodes that need to be evaluated during the search process. The heuristic maintains a list of "killer moves" - moves that were previously identified as causing a cutoff (i.e., a beta cutoff for the maximizing player or an alpha cutoff for the minimizing player) in a similar position. When searching a new node, the algorithm first tries the killer moves before exploring other moves, as they are likely to be strong moves that lead to cutoffs. This can greatly reduce the number of nodes that need to be evaluated, as many of the strong moves are identified and tried early in the search process, leading to faster convergence towards the optimal move.

This model stores 2 killer moves. The killer moves are determined by checking if they are pruned by beta-cutoff.

## G. *Memoization*

Memoization is a technique used in computer programming to optimize the performance of recursive functions by caching their results. The technique involves storing the results of expensive function calls and returning the cached result when the same inputs occur again. When a function is called with a set of inputs, the memoization system first checks if the result has already been computed and cached. If it has, the cached result is returned immediately without recomputing the function, saving time and computation resources. If the result has not been cached, the function is evaluated and the result is stored in the cache for future use.

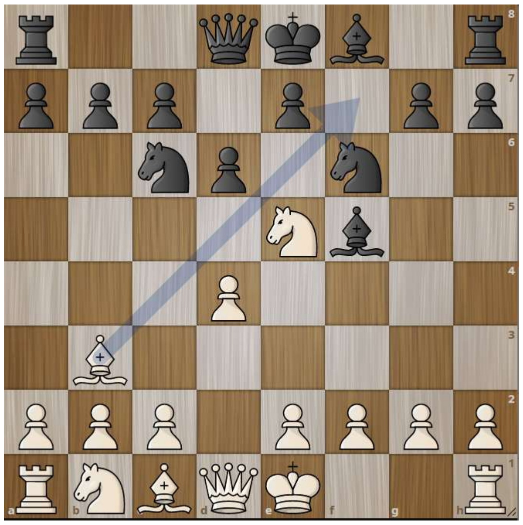
This model creates an identifier value for each position and memoizes the evaluation for each turn.

## III. TESTING THE MODEL

*The chess engine has solved various puzzles with different difficulty levels. In addition, the engine was tested playing various games against human players.*

Because of the checkmate detection in the evaluation function, checkmate in X puzzles can be solved. However, it must be noted that this is an experiment of how good the AI plays when the game is played aggressively. This model cannot calculate the best move on any played game given since it does not play by the conventional chess tactics and human heuristics. The model plays a good move, however it may not always be the best move.

Example puzzles are provided in the Puzzles.py file. It must be noted that since this is a naïve AI, it still must calculate moves to find the checkmate in 1.


Puzzle 2 from Puzzles.py

## IV. ANALYZING THE RESULTS

As mentioned before, this project's aim is to determine if an aggressive approach for chess engines using naïve AI has a better win rate. To experiment, various positions were played by the engine.

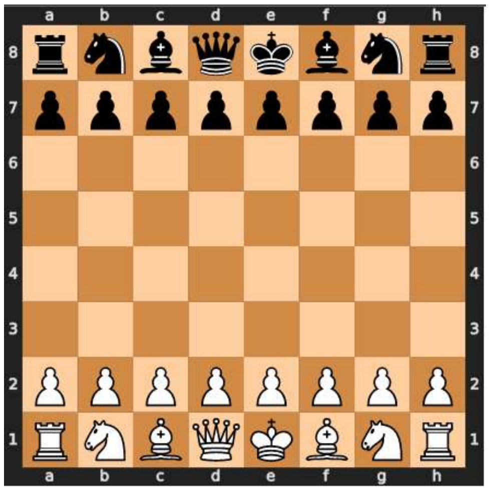
On an aggressive game, the model generally can find the best possible move (considering that this is a naïve AI model with 5 depth). However, in a game with a more tactical approach, the model may not be able to find the best move.

From the results gathered, an aggressive approach is not the best approach in every position. However, if the game is played with general aggression the engine usually generates a good move.

## V. FURTHER TESTS AND DEMO

Further tests can be done by running the main.py file contained in the project. The m file contains a visualized demo. The board is displayed on a browser via SVG and the move is typed to the console in UCI (Universal Chess Interface) format. It must be noted that the game part has no validation and illegal moves can be made.

Also, more puzzles can be written in the Puzzles.py file. The provided puzzle examples are basic examples to understand that the engine can play the best move on simple positions. Also, the puzzles provided has a depth of 1 for simplicity purposes.



Example SVG Board from main.py

## CONCLUSION

According to the results of the experiments, an aggressive approach for chess AI modes is a good approach. However, it must be noted that it is not always the best approach. In higher level chess games which require deep thinking and positional advantage through sacrifices can be easily missed. The engine's aim is to take as many pieces as possible and attack for a checkmate in a simpler position.

## REFERENCES

[1] https://github.com/maksimKorzh/chess_programming Reference Chess AI written in C, 2020
[2] https://python-chess.readthedocs.io/en/latest/ python-chess Docs
[3] https://en.wikipedia.org/wiki/Minimax Minimax Wikipedia Page
[4] https://www.chessprogramming.org/ Chess Programming Wiki
[5] https://rustic-chess.org/ Heuristic methods for chess AI
[6] https://lichess.org/study/WiuSw3ga/c9rkZk4L Puzzle Positions

## FURTHER SOURCES

[1] https://github.com/TheCrimsonNeV0/BIL441Project GitHub Repository Page of the Project
[2] https://youtu.be/p1vIyOVHiK4 YouTube Video for the Project