Chess Engine AI Algorithms

Ishaan Gupta, ***CSE BTech*****, Ajitesh Tripathi,** ***CSE BTech*****, Chayan Pant,** ***CSE BTech***
Ishaan Srivastava, ***CSE BTech***

Computer Science Department,School of Engineering, Bennett University,Plot Nos 8-11, TechZone 2, Greater Noida, Uttar Pradesh 201310 India

Abstract: Two most widely used algorithms to built AI bots in game theory are Minimax and Monte-Carlo tree search (MCTS). For first time in history of AI, machine was able to surpass human potential in complicated games. In this you will understand how we merged these two algorithms to come up with a strong bot. We applied core concepts of reinforcement learning without using any external library in the form of Monte Carlo tree and modified MCTS with the the help of Stockfish in order to achieve strong chess bot.

Index Terms: Monte Carlo Tree Search, Chess AI,Mini max,Reinforcement Learning.

1. Introduction

As humans, we have always wanted to improve on pre-existing intellect and technologies. On February 10, 1996, Deep Blue, beat Garry Kasparov —the first time a computer had ever beat a human in a formal chess game. It used custom VLSI chips to execute the alpha-beta search algorithm in parallel, an example of Good Old-Fashioned Artificial Intelligence. Almost a decade later, we now experience exponentially complex software that is generated based on deep learning. Technology is ever-improving, and so is the ambition to achieve ”excellence” or even ”perfection”. Nowadays there already exist countless engines, each better than the other. Some can be as simple as just using a decision tree and pruning the worst branches out, some can be as complex as the Stockfish chess engine. In Stockfish, the evaluation function was handcrafted for over a decade by many grandmasters all over the world. This evaluation function was then fed into Minimax Algorithm to maximize this function. Another groundbreaking result was observed by the performance of Google’s Deepmind creation AlphaZero which won 28 games and 72 draws against Stockfish with just 4 hours of self play and learning. In this paper we will see how this was made possible by Google. Although Google never released AlphaZero’s code but it provided enough details in [16] about it’s algorithm which enabled us to study underlying principles of it’s creation.

Due to limited computation power, the use of replicated neural network used by AlphaZero was avoided[16]. However, the idea of infusing these two engine(Alpha Zero and Stockfish) into one was quite intriguing. So that’s what was done in order to construct a strong bot. The exact mechanics will be discussed in further sections.

2. Related Work

2.1. Deep Blue

On February 10, 1996, Deep Blue, beat Garry Kasparov - the first run through a PC had ever beat a human in a proper chess game. It utilized custom VLSI chips to execute the alpha-beta search algorithm in parallel, a case of Good Old-Fashioned Artificial Intelligence[2].

2.2. Stockfish

Its been 12 years since Stockfish release and its been one the best CPU-only chess engine. This engine uses a set of algorithms and techniques like alpha-beta search and bitboards. Consistently from the first edition, the creators of stockfish improved it's performance, the lastest version(stochfish 12) has the updatable neural network which makes the increased strength visible[3].

2.3. Alpha Zero

In 2017, AlphaZero was introduced by Deep Mind, a single system that could teach itself from scratch, and defeat masters accross domains- in the game of chess, Shogi(Japanese chess) and GO. It applies reinforcement learning, where it learns from its own mistakes. It starts off playing randomly, but eventually grabs pace and is more likely to choose advantageous moves later on. It takes 9 hours to prepare itself for chess. Komodo relies on evaluation, rather than depth(much like AlphaZero). It also applies the monte carlo search engine[1].

3. Methods

In this section we will go through both the algorithms and how they were implemented from scratch using python. We will also prove why these algorithms work and which one is more effective than the other. We have discarded *Traditional Tree search Algorithm* from this paper because it is not feasible to store all the possible states of the game like chess. Chess has 10^{120} states which is more than number of atoms in the universe!

With that being said, let's dive into the algorithms and their working code.

3.1. Monte Carlo Tree Search

Monte Carlo Tree Search(MCTS) is widely used algorithm in AI in the field of game theory. In this section we will talk about working of this algorithm in the ***context of chess***. There are essentially just four steps in each iteration for this algorithm : Selection $\rightarrow$ Expansion $\rightarrow$ Rollout $\rightarrow$ Backpropagation. The tree state is stored for each iteration. The general principle of MCTS is to get closer to its optimum move with each iteration. If we make a system that has infinite memory, we will be able to design an engine which will give us the ultimate best move at all the states. Before exploring all the steps of the algorithm it is important to understand meaning of an extremely important term used in Reinforcement learning known as Upper Confidence Bound(UCB).

3.1.1. Upper Confidence Bound

Upper Confidence Bound(UCB) is mainly used to assign priority to each action from the current state, higher the UCB score of the node, more likely it is to be selected. The equation of UCB is given by:

$$\textbf{UCB(S}_i, A_j) = arg\,max(V(S_j) + c * \sqrt{\tfrac{\ln N}{n_i}}) \quad (1)$$

Here, S_i is the current state A_j is the j^{th} action which leads to state S_j. N is the number of times S_i's parent node has been visited and n_i is the number of times S_i has been visited. c is the exploration constant, in this paper c is assumed to be 2 for faster results. V(S_j) gives the winning score of S_j state.

3.1.2. Selection

This is the very first stage of starting of a given iteration. Here we simply calculate UCB(S_{curr},A_j) for every action j possible from current state and return the S_j with maximum value and pass it on to expansion() method.

```
#Algorithm of selection function
def selection(curr_node):
    max_ucb = -inf
```

```
    selected_child = None
    if(curr_node.children.isEmpty()):
        curr_node.children = generate_all_possible_states(curr_node)
    for i in curr_node.children:
        curr_ucb = ucb(i)
        if(curr_ucb>max_ucb):
            max_ucb = curr_ucb
            selected_child = i
    return (selected_child)
```

3.1.3. Expansion

When selection() returns the best child of the current node, we recursively apply selection() on current node till we encounter leaf node or reach end of the game.

```
#Algorithm of expansion function
def expansion(curr_node):
    if(curr_node.children.isEmpty()):
        return curr_node
    selected_child = selection(curr_node)
    return expansion(selected_child)
```

3.1.4. Rollout

When we receive leaf node from expansion(), we perform a rollout i.e. we will randomly select actions of current node till we reach the end of the game. We will return +1 if white wins, -1 if black wins or 0 in case of tie.

```
#Algorithm of rollout function
def rollout(curr_node):
    if(curr_node.isGameOver()):
        if(curr_node.color=="white" and curr_node.wins):
            return (1,curr_node)
        elif(curr_node.color=="black" and curr_node.wins):
            return (-1,curr_node)
        else:
            return(0,curr_node)
    selected_child = random.choice(curr_node.children)
    return rollout(selected_child)
```

3.1.5. Backpropogation

When we reach the terminal node returned by rollout(), we want to back propagate reward/loss to each of the node involved in that process, so that in future that path can be exploited/avoided in future iterations.

```
#Algorithm of Backpropogation
def backpropogation(curr_node,reward):
    if(curr_node.parent==None):
        curr_node.score+=reward
        return curr_node
    return(backpropogation(curr_node.parent))
```

3.1.6. Complexity Analysis

Time Complexity of MCTS is O($\frac{mKI}{C}$) where m is number of randomly selected children, K is number of parallel processes, I is number of iterations and C is number of cores available. Space Complexity is O(m*k)[17].

3.2. Mini-Max Algorithm

The first approach is to make a chess engine in which the minimax algorithm is used to generate game states. Minimax is a backtracking algorithm by nature that is generally used in decision-making in certain games to find the best move assuming that the opposition also selects the best move. By this introduction, we get the idea that the use of this algorithm is beneficial in two-player games where the players can take turns, which in our case is Chess. In Minimax these two players are termed as minimizer and maximizer. The whole crux of the algorithm runs on the concept of a single score range which decides the side that has the advantage with the concurrent game state. In chess, this range defines an advantage to white when the score is positive and to black when the score is negative. The extent of advantage is also determined by the numeric value of the score. For every state or position in the game, there is a score associated with it. A heuristic function is used to calculate the score, this function can be of different forms. A function of a similar sort for a chess engine can take the form in which it takes many parameters like material, king's position, central blocks dominance, stability, mobility of pieces, and mating patterns under consideration to evaluate a score for the game state. An example of this can be the DeepBlue chess engine which uses a very complex heuristic function, which they have generated with the help of professional chess players. As each piece has a defined weightage our approach towards this is a bit different. For our chess engine this evaluation function is:

score=$\sum$ White Pieces - $\sum$ Black Pieces

The minimax algorithm takes this score and generates a tree with a possible set of moves and predicts optimal moves by the process of backtracking. The maximizer which is the white player aims to derive the highest score possible from a given state while the minimizer which is the black player does the opposite and tries to get the lowest score possible from a given state. The root node of the tree is a maximizer if the engine is assigned as white, and a minimizer if the engine is assigned as black.

3.2.1. Minimax Algorithm

```
def MiniMax(position, maxPlayer, depth):
    if (depth == 0) or (game is over):
        return evaluation

    if (maxPlayer):
        maxEvaluation = - infinity
        for(every child of current position):
                evaluation = MiniMax(child, false, depth-1)
                maxEvaluation = max(maxEvaluation, evaluation)
        return maxEvaluation
    else:
        minEvaluation = + infinity
        for(every child of current position):
            evaluation = MiniMax(child, false, depth-1)
            minEvaluation = min(minEvaluation, evaluation)
        return minEvaluation
```

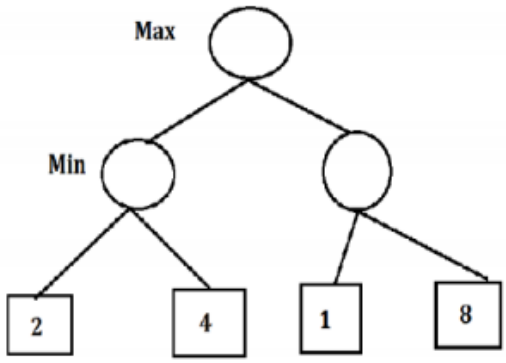

Fig 1. Initial minimax tree

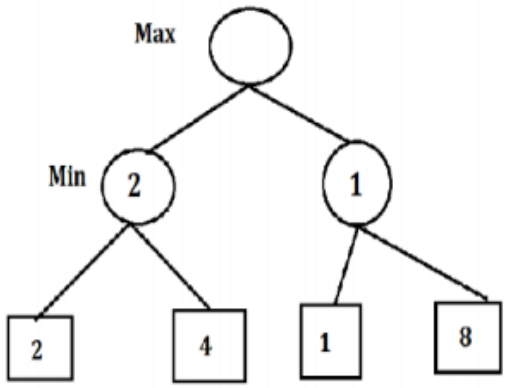

Fig 2. Final minimax tree

3.2.2. Some Observations

Minimax, being a backtracking algorithm, analyses each and every possible set of moves and selects the most optimal one out of them. There are just two possible sets of moves for the players at any given position in the above example(Fig1. Fig2.), but in chess generally, this number shoots up to the range of 30 and more. As the number of moves increases, so does the width of the tree which directly affects the amount of computation power required to run the algorithm which leads to an exponential increase in the time complexity. So in real-life circumstances, the minimax algorithm turns out to be very inefficient as we increase the depths of the tree to improve decision making.

1. For depth=1, the time for each move is 0.5s.
2. For depth=2, the time for each move is 55s.
3. For depth=3, the time for each move is 578s

3.2.3. Alpha Beta pruning optimization

To optimize the minimax algorithm to compute the states to greater depths in a faster way, we use the technique of alpha-beta pruning. The application of alpha-beta pruning reduces the number of nodes that need to be computed by eliminating some branches which are worthless to explore. Alpha-beta pruning can not be termed as a different algorithm from minimax altogether. Instead, it's termed as an optimized version of the minimax algorithm. The idea behind the technique is built upon the base of assurance for the most minimized or maximized score for a given node that can be generated by its subsequent branches. The maximizing node is assured of a maximized score from its branches which is stored by 'alpha' and similarly, the minimizing node is assured of a minimized score from its branches which is stored by 'beta'. These two values of alpha and beta are constantly updated by the minimax function itself, which are then used to determine worthless branches and therefore eliminate the need of exploring them. This process is known as the pruning of branches, hence the name 'Alpha-Beta Pruning' is derived.

3.2.4. Alpha Beta pruning Algorithm

```
def MiniMax(position, maxPlayer, depth, alpha, beta){
    if (depth = 0) or (game is over):
        return evaluation
```

```
if (maxPlayer):
    maxEvaluation = - infinity
    for(every child of current position):
        evaluation = MiniMax(child, false, depth-1, alpha, beta)
            maxEvaluation = max(maxEvaluation, evaluation)
        alpha = maximum of (alpha, evaluation)

        if (beta is smaller than or equal to alpha):
            break
    return maxEvaluation

else:
    minEvaluation = + infinity
    for(every child of current position):
        evaluation = MiniMax(child, false, depth-1, alpha, beta)
        minEvaluation = minimum of (minEvaluation, evaluation)
        beta = minimum of (beta, evaluation)

        if (beta is smaller than or equal to alpha):
            break
    return minEvaluation
```

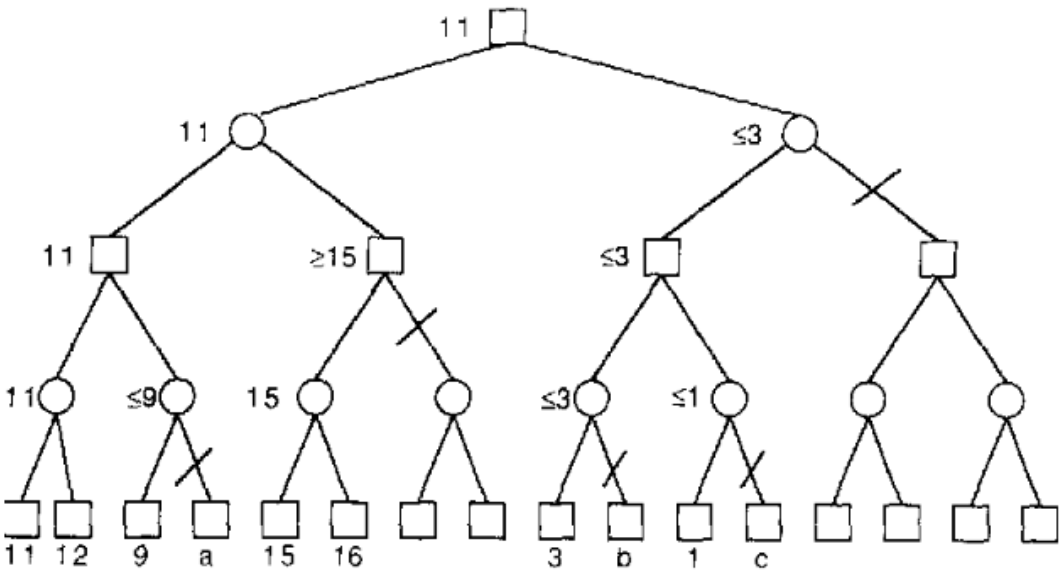

Fig 3: Alpha Beta pruning depth = 4

3.2.5. Observation

After applying alpha-beta pruning the time per move reduced dramatically.

For depth = 2, time = 6.78sec/move
For depth = 3, time = 15sec/move
For depth = 4, time = 90sec/move

3.2.6. Complexity Analysis

Without Alpha-Beta pruning the *time complexity* of the algorithm is O(n^d) where n is number of legal moves and d is depth of the tree. *space complexity* is however, O(n*d).

It is visible from observations above that alpha-beta pruning saves time. But question is how much? In best case scenario, node examines 2*n-1 grandchildren's to determine it's value, in worst case it is O(n^2). Hence overall complexity is O($n^{(d/2)}$) which is theoretically same as O($n^{(d)}$), but in reality it makes significant difference.

3.3. Minmax VS MCTS

Before getting on with the experiment, the comparison made between these two algorithms is based on the depth and number of iterations provided to each of them. Each of these algorithms are unique and are useful in different way.

So, to determine which chess bot performs better, lets make them fight with one another and see who wins. Here, MinimaxBot is initialized with depth=2 and MCTSBot is initialized with iterations=10. Here MCTS played as white and Minimax played as Black. Here are the results:-

Game:- 1. h4 Nh6 2. a4 Rg8 3. c4 Rh8 4. b3 Rg8 5. h5 Rh8 6. Na3 Rg8 7. Nb1 Rh8 8. c5 Na6 9. Nh3 Nxc5 10. Qc2 e6 11. e3 Qh4 12. Qg6 hxg6 13. Ke2 Nxb3 14. Ra3 Nxc1+ 15. Kd1 Bxa3 16. Ng1 Qxh1 17. hxg6 Qxg1 18. d3 Qxf1+ 19. Kc2 Qxd3

Fig 4(a): Final position of MCTS vs MinimaxBot

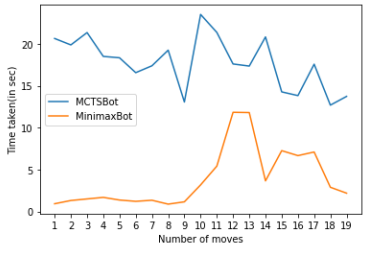

Fig 4(b): Move-time graph of MCTSBot Vs MinimaxBot

3.3.1. Explanation

As it is clearly visible that time taken by MinimaxBot took very less time as compared to MCTSBot. In fact, Mean time taken by MinimaxBot is 3.866secs/move while MCTSBot took 17.774 secs/-move. The reason being rollout() functionality in MCTS. As discussed previously, rollout() takes

random moves until it reaches the end of the game, which disables the algorithm to reach terminal stage fast.

In the case of MinimaxBot, the alpha-beta pruning helps it to make faster decisions and evaluation function helps it to make better decisions than MCTSBot. Therefore, MinimaxBot won this game in just 19 moves.

4. Building Our Bot

After a lot of hit and trial, we finally came up with a way to build a strong bot which is hopefully better than Stockfish chess engine. We did a very simple thing, instead of randomly choosing moves in rollout function() we loaded Stockfish engine[3] and used this to select moves till the end for a given state.

4.1. Why Does it Work?

Stockfish is a chess engine which uses mini-max algorithm to maximize or minimize it's evaluation function. This evaluation function is very complex and has been under development for over a decade. Our MCTS algorithm forces Stockfish's evaluation function to explore new paths, hence giving it an edge over old Stockfish engine.

4.2. Reconstructed Rollout Function

To make things more clear,here is the reconstructed rollout() function:

```
def rollout_stockfish(curr_node):
    if(curr_node.isGameOver()):
        if(curr_node.color=="white" and curr_node.wins):
            return (1,curr_node)
        elif(curr_node.color=="black" and curr_node.wins):
            return (-1,curr_node)
        else:
            return(0,curr_node)
    selected_child = StockFish_make_move(curr_node)
    return rollout(selected_child)
```

4.3. Motivation of This Idea

AlphaZero which is best chess engine so far has implemented Neural Networks and Monte-Carlo Tree Search[1]. They however used super-computer to train their bot which was quite impossible for us due to lack of memory in our system. So we decided to build upon already existing Stockfish to beat itself with slight manipulation.

5. Results and Discussion

So after we were done with constructing our bot we compared it with our previously experimented bots and also the one and only Stockfish. Let's name our bot as the BUBot. Here games are represented in ***Standard Algebraic Notations (SAN)***[1]. In *move-time* graph time is in seconds.

5.1. BUBot Vs MinimaxBot

BUBot was able to win against MinimaxBot which has depth=2 and BUBot's number of iterations was set to 30. Here BuBot played as black and won in just 23 moves:

Game:

[1]Standard algebraic notation (SAN) is a system for recording chess moves. Moves are represented by the name of the piece and the square to which it is being moved.

Nh3 d5 Ng5 e5 Nf3 e4 Ne5 f6 Nf7 Kxf7 Rg1 f5 Rh1 c5 Rg1 Nc6 Rh1 Be6 Rg1 Nf6 Rh1 d4 a3 Bd6 h3 Ne5 Rh2 c4 a4 Rc8 Rh1 f4 h4 d3 exd3 cxd3 Rh2 dxc2 Qxc2 Rxc2 Rh1 Rxc1+ Ke2 Bc5 Rh2 Qd3#

Fig 5(a): Final postion of BUBot vs MinimaxBot

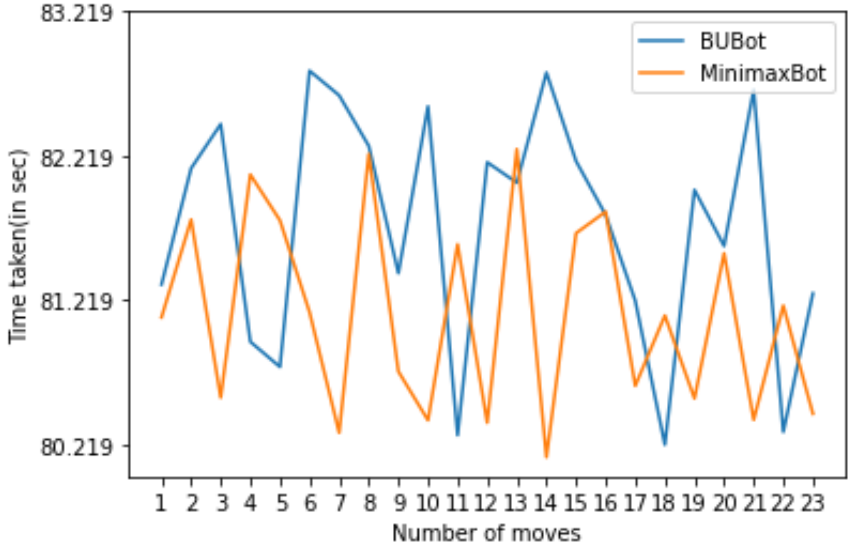

Fig 5(b): Move-time graph of BUBot Vs MinimaxBot

5.2. BUBot Vs MCTSBot

BUBot was able to win against MCTSBot too in just 19 moves. MCTSBot's number of iterations was set to 30 and BUBot's number of iterations was set to 30.

Game:

e4 c6 b4 e5 Bc4 Nf6 f3 d5 a4 dxc4 Qe2 Bxb4 Ra2 Be6 Kf2 c3 Qe3 Na6 g3 Bxa2 Ke2 Bxb1 Ke1 Bxc2 Qxc3 Bxc3 Nh3 Qd4 Kf1 Qd3+ Ke1 Qe3+ Kf1 Qxf3+ Nf2 Nxe4 Kg1 Qxf2#

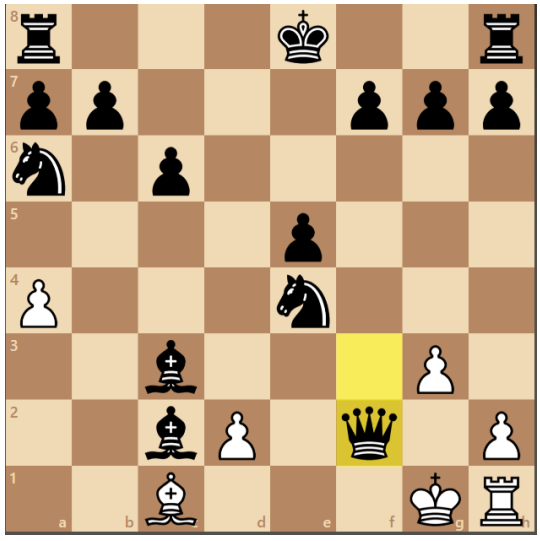

Fig 6(a): Final postion of BUBot Vs MCTSBot

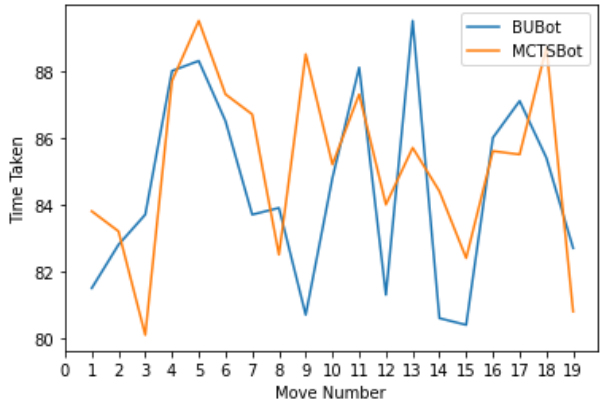

Fig 6(b): Move-time graph of BUBot Vs MCTSBot

5.3. BUBot Vs StockFish

Now this the most interesting match of our entire experiment. In this match BUBot plays as white while StockFish plays as black. Number of iterations of our engine=30. This battle lasted for 73 moves.

Game: 1. e4 Nc6 2. Nf3 e5 3. Nc3 Nf6 4. Nd5 Nxe4 5. Bc4 Nd6 6. d3 Nxc4 7. dxc4 d6 8.O-O h6 9. Qe2 a5 10. b3 Be7 11. c3 O-O 12. Re1 Re8 13. b4 Bf8 14. b5 Ne7 15. Be3 f5 16. Rad1 Bd7 17. Bc1 Nc8 18. c5 c6 19. Ne3 Qf6 20. cxd6 e4 21. Nd4 f4 22.bxc6 bxc6 23. Nc4 f3 24. gxf3 c5 25. Nc2 Bc6 26. d7 Qg6+ 27. Kh1 exf3 28. Qf1 Rxe1 29. Nxe1 Nd6 30. Bf4 Nf7 31. Nb6 Rd8 32. Bc7 Be7 33. Bxd8 Bxd8 34. Nd5 Bxd7 35. Qc4 Kh7 36. Nf4 Qf5 37. Qd3 Qxd3 38. Nexd3 Bb5 39. Nxc5 Be7 40. Ne4 Bc6 41. Nd5 Bd8 42. c4 Ne5 43. Rd4 Kg8 44. Nc5 Kf7 45. Rf4+ Bf6 46. Kg1 Bxd5 47. cxd5 g5 48. Re4 Be7 49. Nb7 Bf6 50. Nxa5 Nd7 51. Nc6 Nc5 52. Rc4 Nd7 53. Rb4 Nc5 54. a4 Be7 55. Nxe7 Kxe7 56. a5 g4 57. Rb6 g3 58. fxg3 Nd7 59. Rxh6 Nc5 60. a6 f2+ 61. Kxf2 Ne4+ 62. Kf3 Ng5+ 63. Kg4 Nf7 64. Rh7 Kf6 65. Rxf7+ Kxf7 66. a7 Ke7 67.a8=Q Kd6 68. Qc6+ Ke5 69. Kf3 Kd4 70. d6 Ke5 71. d7 Kd4 72. d8=R+ Ke5 73. Rd5#

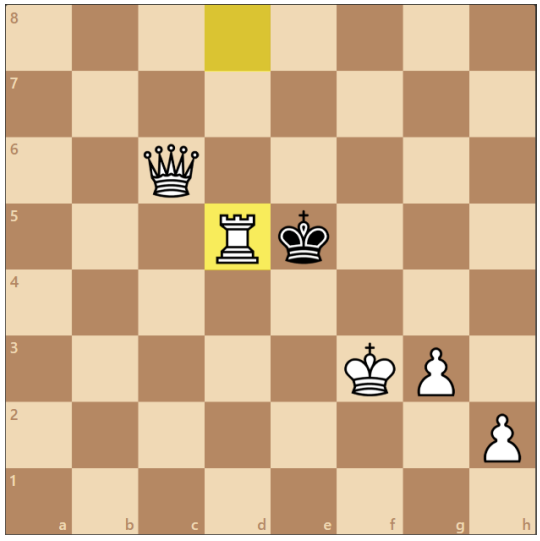

Fig 7(a): Final postion of BUBot Vs Stockfish match

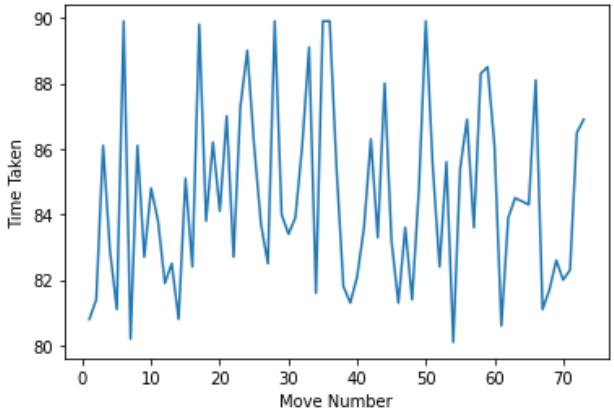

Fig 7(b): Move-time graph of BUBot against Stockfish

6. Conclusion

From our experiment and results so obtained, it can be concluded that integrating Min-max and MCTS can yield exceptionally good chess bot. It can also be seen that our bot is quite strong and it did not have to consume a lot of computational power to achieve such good results. We were also successful in converting general algorithm to the chess code for our purpose. This can be reused by other chess programmers to make better and more efficient bots.

7. References

[1] David Silver, Thomas Hubert, Julian Schrittwieser et al, "A general reinforcement learning algorithm that masters chess, shogi and Go through self-play ", Science, 6th December 2018.

[2] Feng-hsiung Hsu, Murray S Campbell, and A Joseph Hoane Jr. ,"Deep blue overview", https://www.wikizero.com/en/Deep$_{B}lue_{(}chess_{c}omputer)$.

[3] Stockfishchess.org, 2015, URL http://stockfishchess.org .

[4] Luís Tarrataca, Andreas Wichert, "Tree search and quantum computation", ResearchGate, August 2011, DOI: 10.1007/s11128-010-0212-z.

[5] Shirish Chinchalkar, ”Number of reachable positions. ICCA JOURNAL, 19(3):181–183, 1996”
[6] Daniel Shawul and Remi Coulom, ”Paired comparisons with ties: Modeling game outcomes in chess”,https://www.arxiv-vanity.com/papers/1509.01549/.
[7] Wei-Li lutetium, Yu-Shuen Wang, Wen-Chieh carver, ”Chess Evolution Visualization”.
[8] Monty Newborn, ”Deep Blue”, Springer-Verlag New York, 2003, DOI: 10.1007/978-0-387-21790-1.
[9] Stephen JJ Smith, ”An analysis of forwarding pruning”, citeseerx, 1994.
[10] Pedro Pires, Petia Georgieva, Pattern ”Recognition and Image Analysis”, 9th Iberian Conference, IbPRIA 2019, Madrid, Spain, July 1–4, 2019, Proceedings.
[11] Hongming Zhang, Tianyang Yu, ”Deep Reinforcement Learning” Springer, Singapore, DOI: 10.1007/978-981-15-4095-0.
[12] Chyi-Yeu carver, Po-Chia Jo, Chang-Kuo Tseng, ”Design and realization of a two-armed multifunctional companion mechanism with electronic game board”.
[13] Moral cheating in on-line chess, 2007.
[14] Mark Craven and David Page,”Reinforcement Learning with DNNs: AlphaGo to AlphaZero”,CS760: Machine Learning Spring 2018.
[15] Yifan Jin, Shaun Benjamin,CME 323, Report.