Artificial Intelligence: Project Report

I. Overview

1) Scope of this project

➢ Provide ideas and initial implementations of 5 search algorithms used in chess game: Minimax, Negamax, Alpha-Beta pruning, Quiescent Search, Null-move heuristic search.

➢ Provide implementation of two simple heuristic methods, used to evaluate a chess position.

➢ Provide measurement in time of each technique running on various depths to visualise time complexity over the course of a chess game.

➢ Conclude, remark on measurement (if possible)

2) What this project does not cover

We simply aren't big-brained enough to construct a convincing accuracy-measuring method. There are three reasons:

- There's no definition of an absolutely correct move.
- Our heuristic position evaluator is too simple, and thus:

Different implementations of heuristic evaluator will thrive in different types of positions (will be discussed later in II.3)

3) Third-party packages/softwares used

This project runs on python3.8.

The champion package that helped us with chess board management is python-chess (https://github.com/niklasf/python-chess).

GUI is made with Godot3.4 (https://godotengine.org) (but you probably don't care about this).

How to install these packages/softwares is detailed in README. If you don't see that file, please email to lam.t194787@sis.hust.edu.vn and remind him that he screwed up this project.[1]

---

[1] Or any of my team member. They would gladly bombard him with complains, frustration, desperation, regret, anger, disappointment, 50 shades of F midterm... *shiver.

II.   Details of Solutions

1) Problem Representation

The problem: Given a chess position, determine the next best move that will lead to a winning position for the side-to-move.

To solve this problem, we define the search space as a directed graph:

- A node of the graph is a Board.
- A Board consists of the layout of pieces on the board and the side to move.
- A directed edge on the graph is a Move that transform one Board to another.

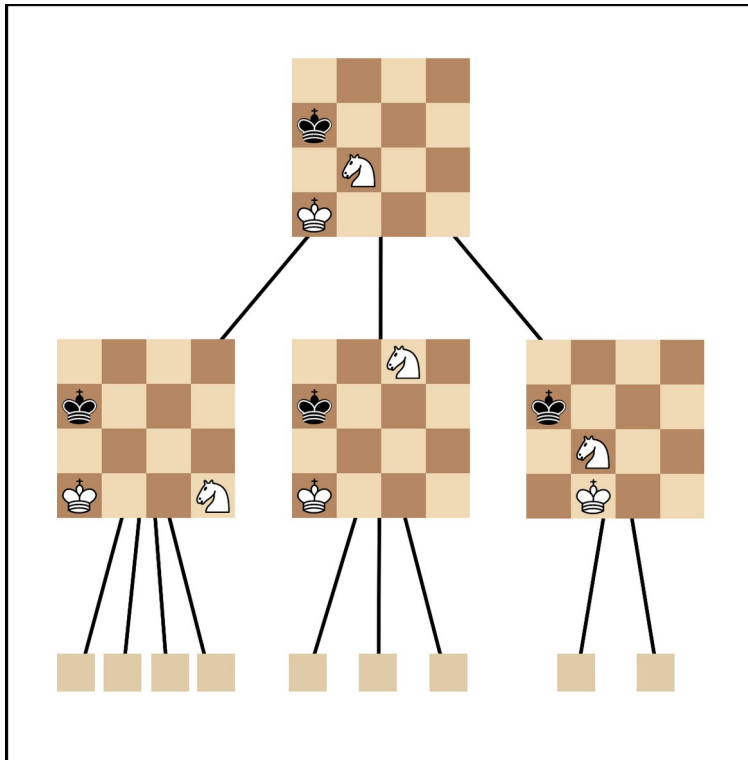Board and Move are implemented by python-chess library.

Figure 1: An example of how the search space looks like

2) Measuring Method

Each search algorithm will evaluate each Board of the Evergreen game[2], and then output the evaluation time. A Board is evaluated many times and taken average to ensurethat the result does not bias too much from the mean actual value. Because some algorithms are too slow, we only evaluate each position 3 times.

Measurement is conducted via EvergreenTimer.py. If you are curious, or want to validate the results, you can tweak and run it. Graphs are drawn by an online tool[3].

Again, this is time-measurement. We <u>Do Not</u> perform accuracy-measurement.

---

2   A splendid examplar of the Romantic Era of chess: https://www.chess.com/terms/evergreen-game-chess
3   https://www.rapidtables.com/tools/line-graph.html. Pretty neat little thing.

3) *Solving Algorithms*

Each topic in Details of Solutions will follow this structure (more or less):

- Motivation: Why it exists, what does it do.
- Implementation: How it works.
- Measurement: Some plotted graphs to give you a sense of algorithm quickness.
- Remark: Some things we noticed and explained (if possible).
- Improvements: How it could have been better, but we didn't have time (nor confidence) to implement it.

Right, so let's not beating around the bush anymore. To the point!

➢ Evaluator

- *Motivation:*
  We need to quantify how "good" a Board is. Evaluator is the heuristic evaluation function. It takes a Board and output a number – the score estimated for a Board. Positive number indicates that white is taking the lead, and vice versa.

- *Implementation:*
  In our project, we have two Evaluators: EvaluatorSunfish[4] and EvaluatorPosition. They are almost identical in working mechanism, except for King handling. Let's talk about their common points first:

```
'P': (  0,   0,   0,   0,   0,   0,  0,   0,
       78,  83,  86,  73, 102,  82, 85,  90,
        7,  29,  21,  44,  40,  31, 44,   7,
      -17,  16,  -2,  15,  14,   0, 15, -13,
      -26,   3,  10,   9,   6,   1,  0, -23,
      -22,   9,   5, -11, -10,  -2,  3, -19,
      -31,   8,  -7, -37, -36, -14,  3, -31,
        0,   0,   0,   0,   0,   0,  0,   0)

'N': ( -66, -53, -75, -75, -10, -55, -58, -70,
        -3,  -6, 100, -36,   4,  62,  -4, -14,
        10,  67,   1,  74,  73,  27,  62,  -2,
        24,  24,  45,  37,  33,  41,  25,  17,
        -1,   5,  31,  21,  22,  35,   2,   0,
       -18,  10,  13,  22,  18,  15,  11, -14,
       -23, -15,   2,   0,   2,   0, -23, -20,
       -74, -23, -26, -24, -19, -35, -22, -69),
```

♟ = -100

♞ = 320

Figure 2: Example Position-Score mapping & Piece Score, EvaluatorPosition

1. Each chess piece is assigned a score (also called Material Point).
2. Each chess piece has a map of position-on-board with a score.
3. The score estimated for a Board is the sum of all pieces' score and their position score.

In EvaluatorSunfish, the King is assigned a value very big, so that it won't trade the king for any other piece. In EvaluatorPosition, the King is not assigned any value, but still has its position-score mapping present.

We use EvaluatorPosition because it is faster than EvaluatorSunfish, and there has not been any out-of-this-world dumb move recorded when we use this Evaluator for various algorithms, and played against it.

- *Measurement:*

---

4   Adapted from Sunfish chess engine: https://github.com/thomasahle/sunfish/blob/master/sunfish.py

EvaluatorSunfish takes 0.000102 second to evaluate a position.
EvaluatorPosition takes 0.000082.
So, EvaluatorPosition is faster than Sunfish by 25%.

- *Remark:*
  None

- *Improvements:*
  Based on Ethereal chess engine[5] evaluation criteria

  Our Evaluator has a serious drawback: It is only effective in opening/middlegame phase. For example, the position-score mapping for king piece highly promotes king's safety. It says that "The closer your king is to your home, the more advantageous it is" but this is not true in endgame phase. In endgame, your king becomes a powerful piece and must head to the battlefield itself.

  Some concepts are ignored here, such as "pinning", "isolated pawn", "knight outpost"… which make pieces more powerful, or cool down their threat level.

  Pieces with more positions to move make their presence on the board deadlier, and should also be noted.

  In summary:
  1. Changes position-score mapping dynamically over the game.
  2. More mobility → More score
  3. Integrate other chess concepts
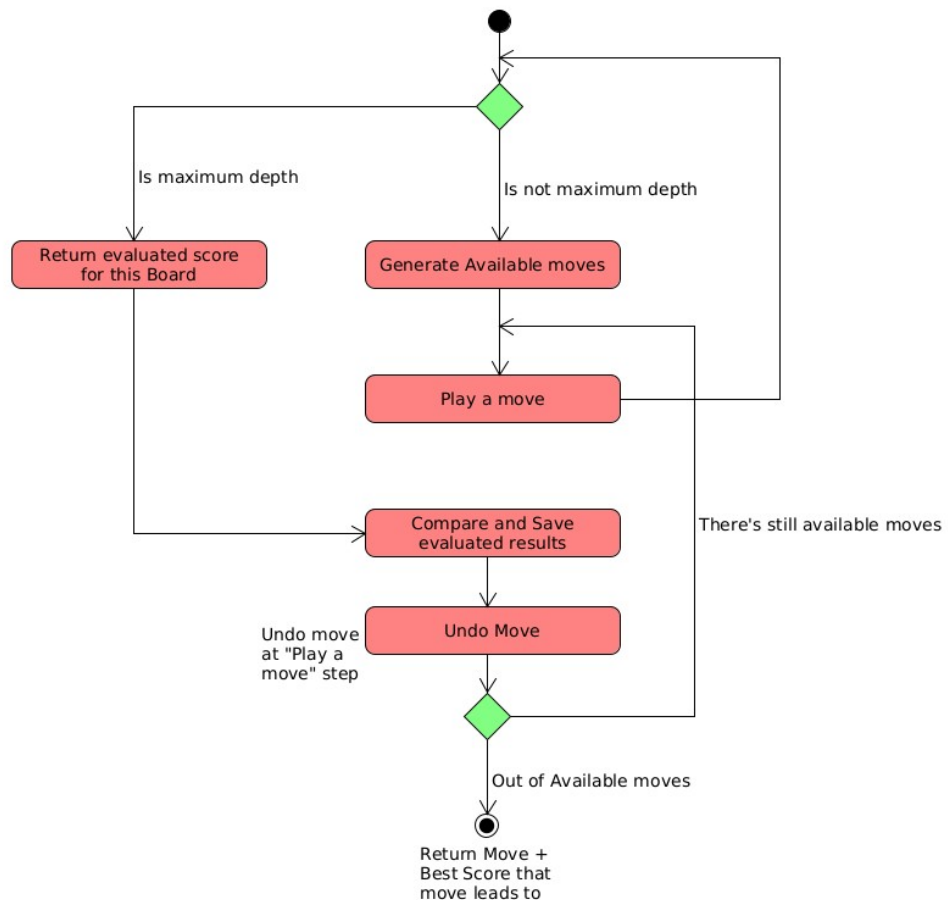
---

5   https://github.com/AndyGrant/Ethereal

➢ Minimax/Negamax

- *Motivation*

  Search the whole game space for the move that lead to MAXimum advantage for self, or MINImize advantage of the opponent.

- *Implementation*

  Because the game space is too big to search, we implement Minimax as a Depth-limited search algorithm:

Negamax algorithm is basically Minimax algorithm but instead of having separate subroutines for finding Min and Max, we combine those two into one and passed one part as negative, one as positive as follows:

max(a, b) = -min(-a, -b)
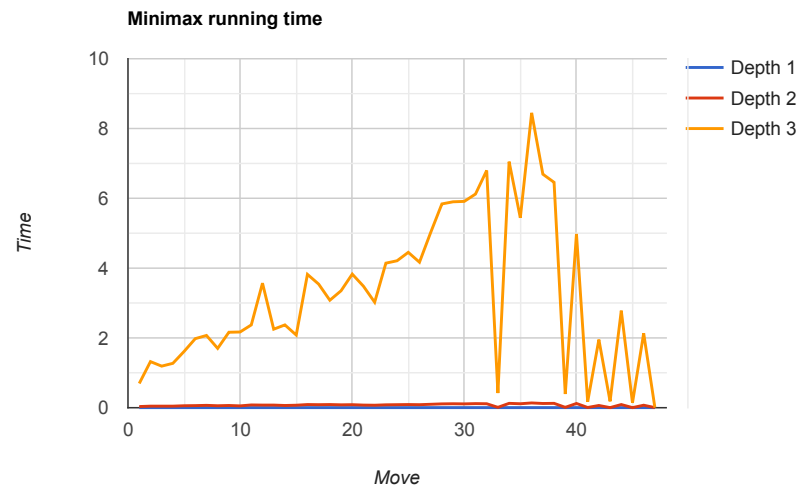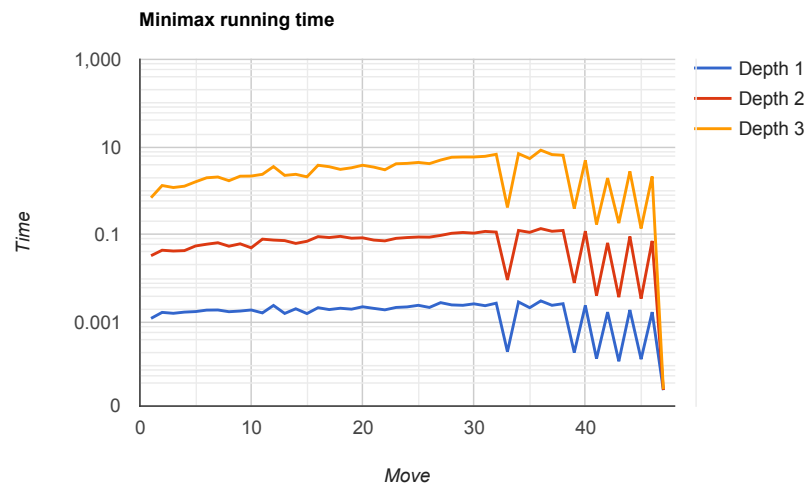
- *Measurement*

Figure 3: Minimax linear measurement

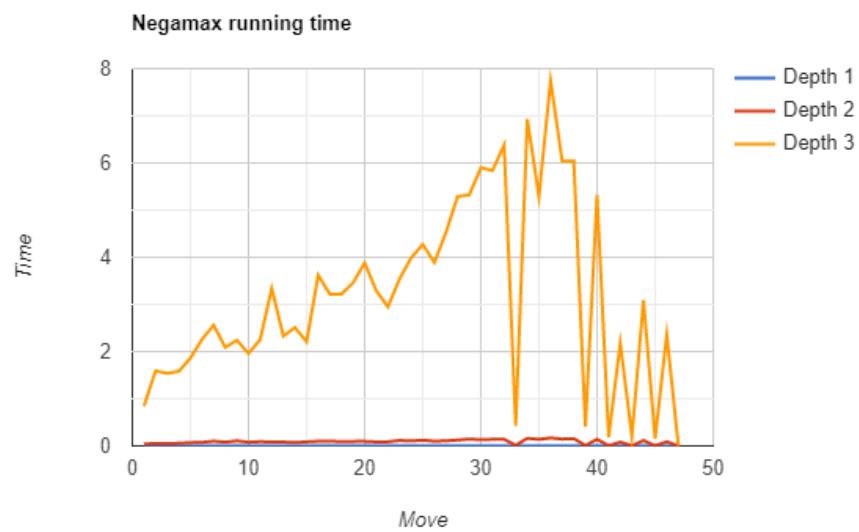*Figure 4: Minimax logarithm measurement*

Figure 5: Negamax linear measurement

- *Remarks*

  There are some sharp cuts in the graph. That's because a king is in check in these moves. Consecutive checking cuts down the search space drastically, because it leaves the mover with very few options.

  The search time rises from the start to middle of the game (half-move 30), and then go down from middle to endgame. That's because at the middle game phase, pieces are released from their starting positions, and thus there're more options to consider. And then at the end of the game, pieces are captured, disappear from the board, number of options reduces, so the search time go down.

  In logarithm graph, the graph forms are alike. This might indicate that the proportion of search space between (any) two depths are approximately fixed.

- *Improvements:*

  Some Boards are exactly identical (excluding move numbering info). But Minimax/Negamax still re-evaluate them many times. To speed up, we have an idea to use dynamic programming with hash table to store the FEN strings of traversed Board. But the problem is that the search space might overwhelm hardware memory, and caching effectiveness dies. Thus, it remains a speculation.
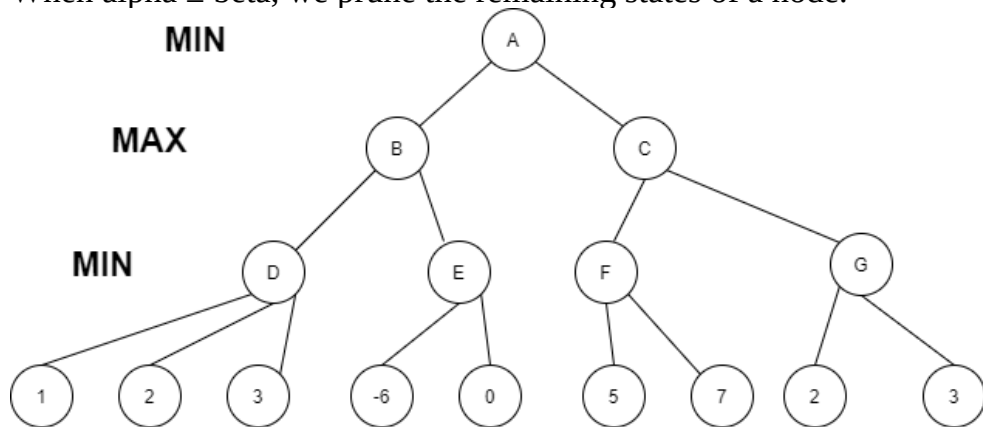
➢ Alpha-Beta Pruning

- *Motivation:*

  Cut off search space, using knowledge about the way players pick their moves.

- *Implementation:*

  Initialize alpha and beta variables with initial values -INF, INF respectively. White will update alpha and Black will update beta.

  When alpha ≥ beta, we prune the remaining states of a node.

Step 1: betaD = INF, alphaD = -INF
Step 2: Min(1, betaD) = 1 -> betaD = 1
Step 3: Min(betaD, 2) = 1 -> betaD = 1
Step 4: Min(betaD, 3) = 1 -> betaD = 1
Step 5: D = 1 because min(1,2,3) = 1
Step 6: betaB = INF, alphaB = -INF

Step 7:
     Max(D, alphaB) = 1 -> alphaB = 1
     → alphaE = 1, betaE = INF

     Min(-6, betaE) = -6 -> betaE = -6 but alphaE >= betaE pruning remain state

Do like this step by step finally, at node A, alphaA = -INF, betaA = INF then best move is have score = 1 and the computer will choose this leftmost move.

- *Measurement:*

AlphaBeta running time

AlphaBeta running time
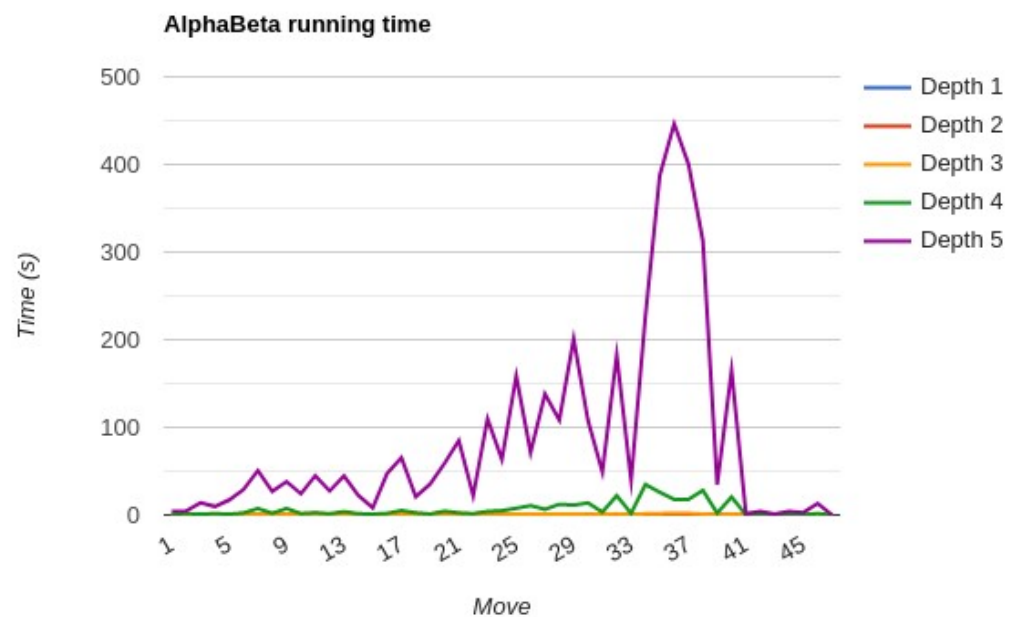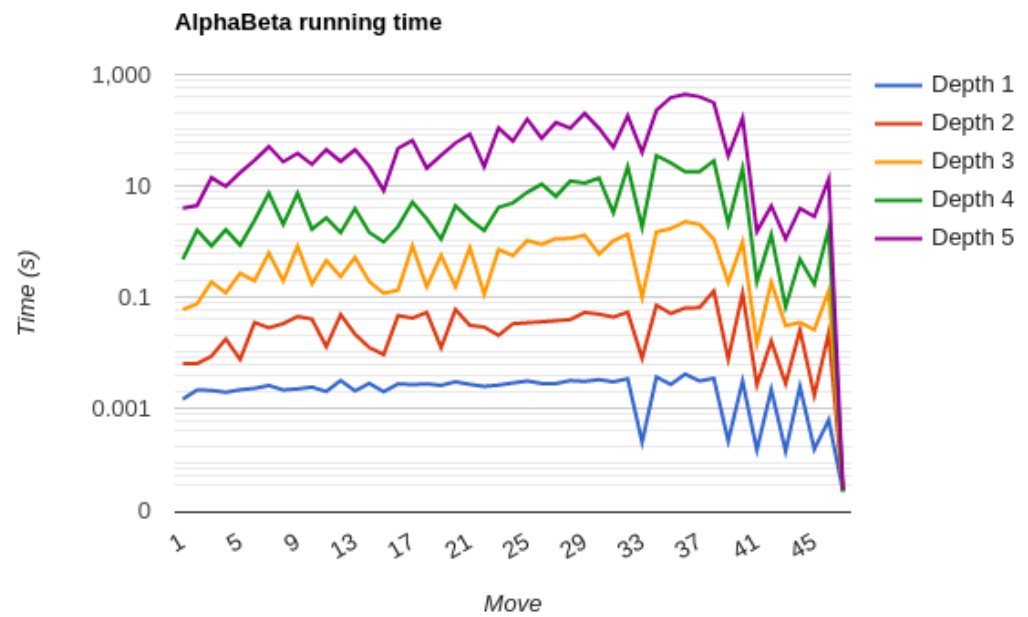
- *Remark:*

This algorithm is the improved version of the Minimax algorithm in running speed. It will return the exact move Minimax chooses, but in quicker time.

- *Improvements:*

None

➢ Quiescent Search

- *Motivation:*

  Increase accuracy. Does extra work against moves that have tremendous impact to evaluation score (in other fancy words: deals with the "horizon effect").

  The normal search is trying to use fixed depth for each time searching for best Position. However, leaf of the search tree doesn't contains the information about whether the next move is a capture or not. It could affect the whole result of a match if the rival is using the same or better search algorithm with higher or equal depth.

- *Implementation:*

  The main idea of this search method is to try combine previous search methods with the ability to "predict the future with some short depth". In our method, we are trying to integrate this idea with Alpha Beta fixed-depth search. When Alpha Beta complete its initial search, the algorithm would test for quietness of the current Board (is there any capture moves or not). If it is, then we return this move. But if it is not, the search will extend a few more depths to ensure this isn't a blunder.

  Since this algorithm depends not only on the number of Positions spawned from root but also the number of captured move each time the normal search reaches the depth equal = 0. Then we also proposed 2 methods to find the find the captured move which can improves this algorithm.

  1. Naive idea: Each time searching, we list all possible move is the captured move, then if current Position not in that list, we return the move else continue with quiescence for deeper search.

  2. Improvement: Combine find captured move and en passant move and the comparison the score of board if be captured to reduce the space of the captured list.

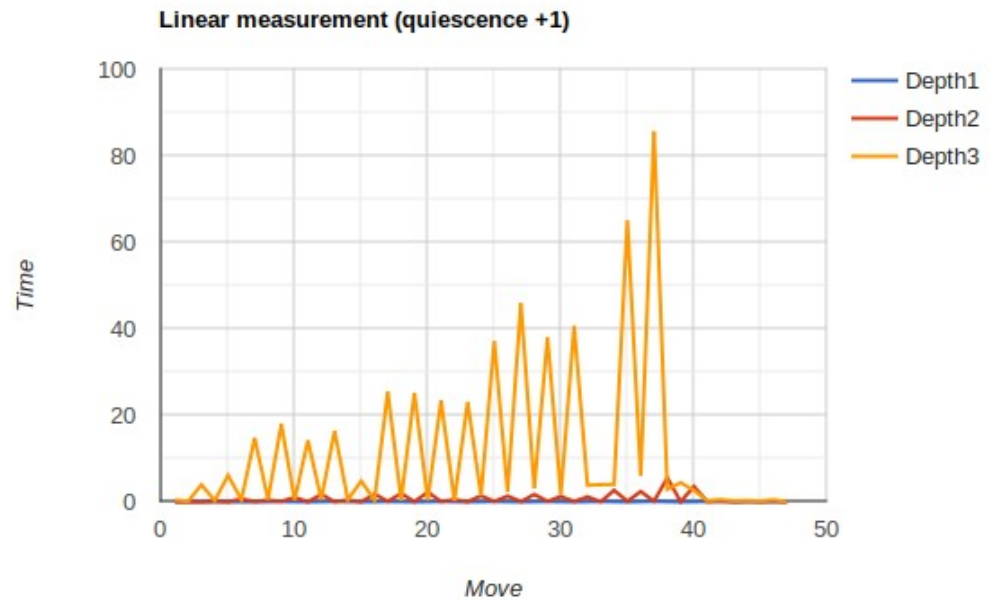*Measurement:*

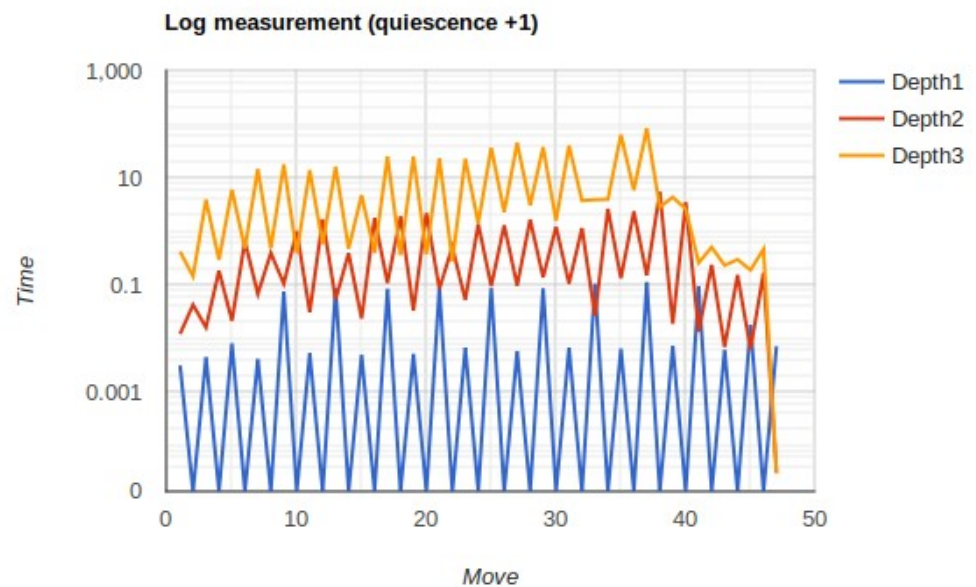These are the measured result from Naive idea

Figure 6: Naive Idea

Figure 7: Naive Idea

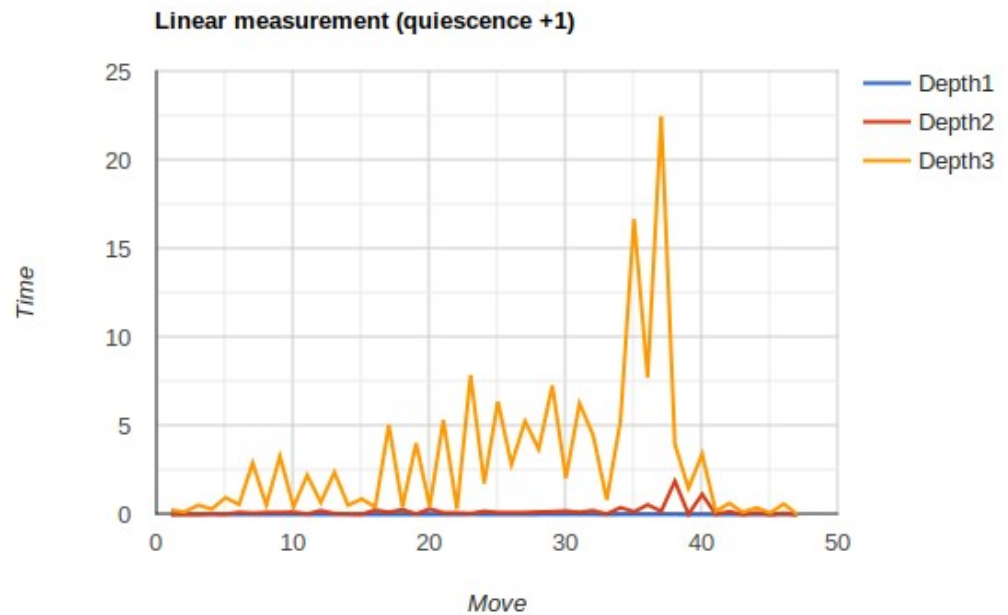And these are measurement result from Improvement Idea:
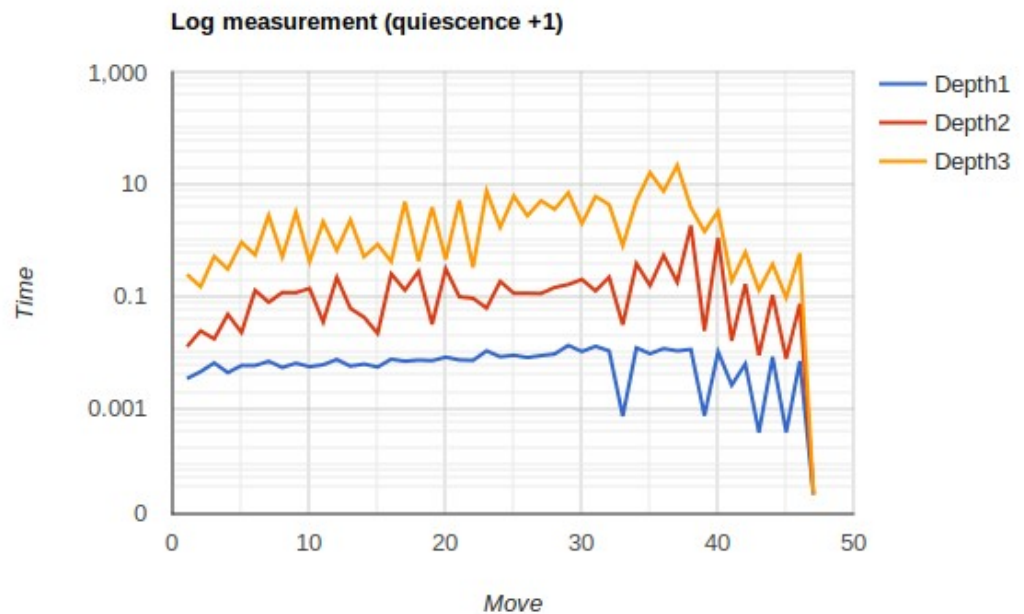

Figure 8: Improved idea linear

Figure 9: Improved Idea Log

Clearly we can see the improved method runs 4 times faster than the naive idea with just list all captured move. The peak of this algorithm approximately 23 seconds for depth = 3 (equals depth = 4 of fixed depth algorithm). This means we always has benefit of score board compared with rival with same depth (since quiescence step, in our code is increasing 1 depth).

- *Remark:*

  Time complexity:
  Quiescence search also use DFS
  Worst case: O(b^m)
  Ideal case: O(b^m/2)
  Space complexity:
  Worst case: O(bm)
  Ideal case: O(b(m/2))
  Completeness:
  Quiescence is complete, definitely find solution, if exists in finite search tree
  Optimality:
  Optimal if both player plays optimally.

- *Improvements:*

  We can implement many other algorithm with faster perform like negamax or null move,… for initial search to increases the performance of this algorithm.

➢ Null-Move Heuristic

- *Motivation:*

  Trade off accuracy for speed. If one player makes a sequence of moves so bad that the other player can skip their turn, then the initial sequence of moves should not be tried.

- *Implementation:*

  Swap the side whose turn it is to move, then performs an alpha-beta search on the resulting position to a shallower depth. If this shallow search produces a cutoff, it assumes the full-depth search (in the absence of a forfeited turn) would also have produced a cutoff.

- *Measurement:*

AB
Time (ms)
Chess Position

Depth 1
Depth 2
Depth 3
Depth 4
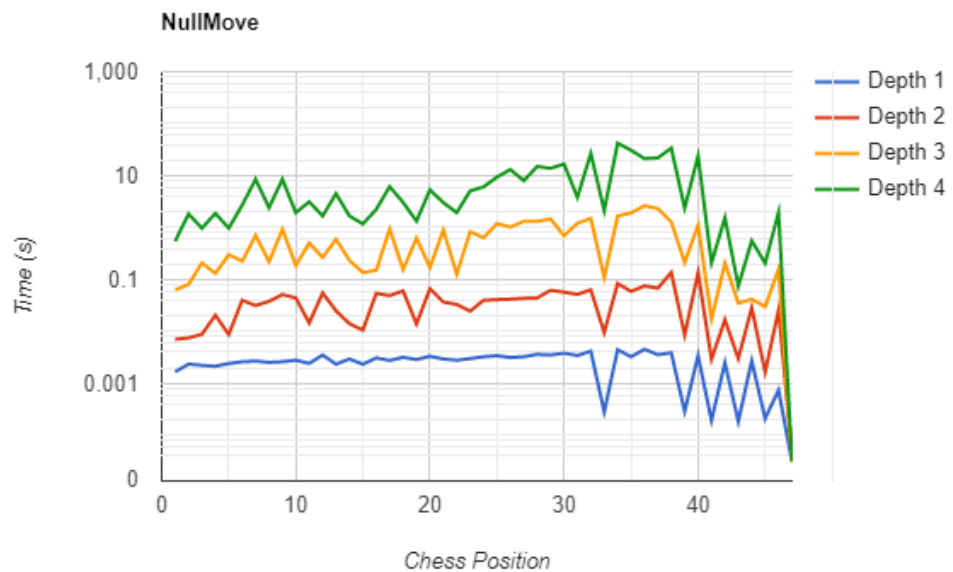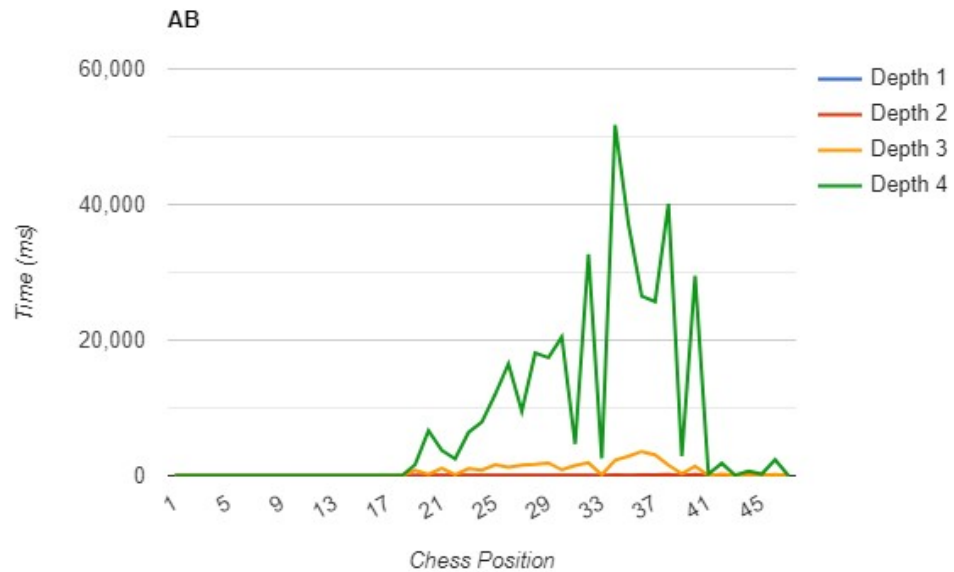
- *Remark:*

  Null-move have to make no move without any change in piece layout of a Board, just switch the side to move – which is an illegal move in python Chess, however we have to do some evaluate in the new chess position and then undo all of it, so we have to make a null-move that can be push or pop like a legal move, which the provided function chess.move.null in python chess can't do.

  Unlike alpha-beta, which reduce workload without reduce accuracy compared to minimax, null-move pruning may return to a worse move (it still doing the maximize and minimize job as it based on alpha-beta, but with reduced accuracy due to reduced depth)

  So the main difficulty here is that we have to choose R (level of depth reduced) to balance the workload reduce and accuracy reduce.
  We haved tested some value of R (with depth 8) and the result is that R=2 is the best thing we got, R=1 is too small to have workload significantly reduce, but R=3 is too large, making the null-move too likely to miss tactics.

- *Improvements:*
  None

III.   Team Assignment

1) Tran Lam 20194787
   Coordinate and distribute workload to team members.
   Quality check.
   Code: EvaluatorSunfish, Minimax, EvergreenTimer, pychess, GUI
   Presentation: Alpha-Beta pruning

2) Vu Tuan Minh 2019
   Code: Null-move heuristic
   Presentation: Null-move heuristic

3) Dao Duc Manh 2019
   Code: Quiescent search
   Presentation: Quiescent search
   Write & test installation on Linux

4) Chu Thach Thao 2019
   Code: Negamax
   Presentation: Minimax, Negamax

5) Dang Quang Minh 2019
   Code: Alpha-Beta
   Presentation: Evaluator
   Write & test installation on Windows
   Found this awesome link which describes these 5 methods implemented in this
   project. Ohhhh! What would we do without him!
   https://www.cs.cornell.edu/boom/2004sp/ProjectArch/Chess/algorithms.html