Playing Games With AI - Checkers Version

by Svitlana Midianko, Spring 2021. For best experience, please read this report by following this link.

1. Problem Definition

A checkerboard game *Checkers* has been active for over 5000 years and gained popularity in multiple parts of the world, including Ukraine, where I used to play this game as a child. *Checkers* is a zero-sum game played by two players and involves a high degree of strategic and tactical thinking. Even though there are many variations of the game, the standard version of it is described below [1].

The game's initial state consists of 12×2 checkers placed on opposite sides of 8×8 checkerboard in a checker-like manner. Each player starts with the symmetric position and takes turns alternating.

While playing, the **players are allowed to:**

1. make diagonal moves of uniform game pieces;
    1. make mandatory captures by jumping over opponent pieces.

And the **rules of the game are as follows:**

a. Capture is possible only in the forward direction. The exception is when the capture is a part of multiple stages captures on the same move, then the capture is possible also backward.

b. When a checker reaches the last line, it is crowned to Queen (in the Ukrainian version; in the standard version, it is often named King). Queens can move both backward and forward, and they're also able to capture both backward and forward.

The **player's goal** is to eliminate all the opponent's checkers or surround the remaining checkers, which determines a victory.

Checkers is a *competitive multi-agent* environment. It is also *deterministic,* meaning that the $state_{i+1}$ of the game/environment is fully determined by the $state_i$ and action executed by the player. The state of the game is *static* so that it cannot change white the player is deliberating (if the game is played without a clock). It is also *fully observable* for players to see what opponents' and their state of checkers are. The game is also *sequential*, so the player's best move will depend on the opponent's previous move. Because there is a set of actions, this game also has a *discrete* environment. The search space of the game is enormous — around 10^{20}, but not as big as in chess (~ 10^{120}). Branching factor is 2.8 [2, 3] .

2. Solution Specification

To solve this problem, I developed this game in Python. Using this code, it is possible to: (1) watch two AIs playing against each other; (2) play against chosen AI. There are multiple configurations of AI that can be tried out, such as AI using solely minimax, AI using minimax and alpha-beta pruning, AI using either one of the two evaluation functions.

The `MiniMax` algorithm is an appropriate choice of algorithm for this game. It relies on having zero-sum, opposite interests between the players — each of them wanting to win. `MiniMax` constructs the tree branching out of each possible board state , starting from the root node (current state). On each level of the tree, the goal is to either maximize or minimize the value of the evaluation function. This is because, assuming that the opponent plays smart, we know that the opponent wants to minimize the score, which would happen every other move. It is a recursive algorithm, so it repeats for each of the states of the board, resulting in exponential growth with extra additional depth level. See figure 1 for visualization.

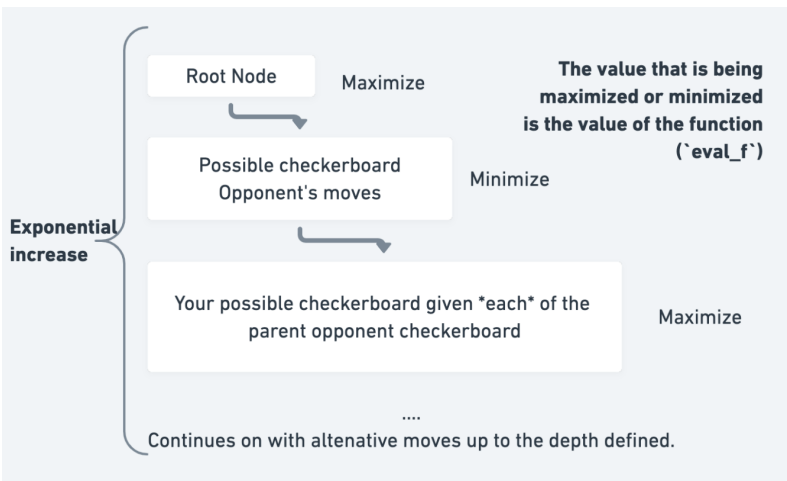


Figure 1. Visualization of the MiniMax algorithm for *Checkers*. On each of the depth the goal is to either maximize or minimize value of the evaluation function. With alpha-beta pruning number of evaluations decreases because of disregard of the states that wouldn't improve the player's score.

One of the weaknesses of such an algorithm is its almost brute-like structure, where the player/AI has to check *all* possible movements and evaluate what their outcomes are. This requires a lot of computation power. To optimize this, I also implemented alpha-beta pruning, which stores two extra values of α, β and prunes away the branches to minimize the number of `MiniMax` evaluations. Essentially, on each level of the tree, the algorithm stops evaluating branches on that depth of the tree once there was at least one possibility found that proves the next move to be worse than a previously examined move. It returns the same result as simple `MiniMax` but prunes away the branches on the way hence minimizing the number of minimax evaluations needed.

Figure 3 displays the diagram of the classes and functions implemented to make it work. Since the evaluation of the whole tree is not possible, I limit the depth of the tree.

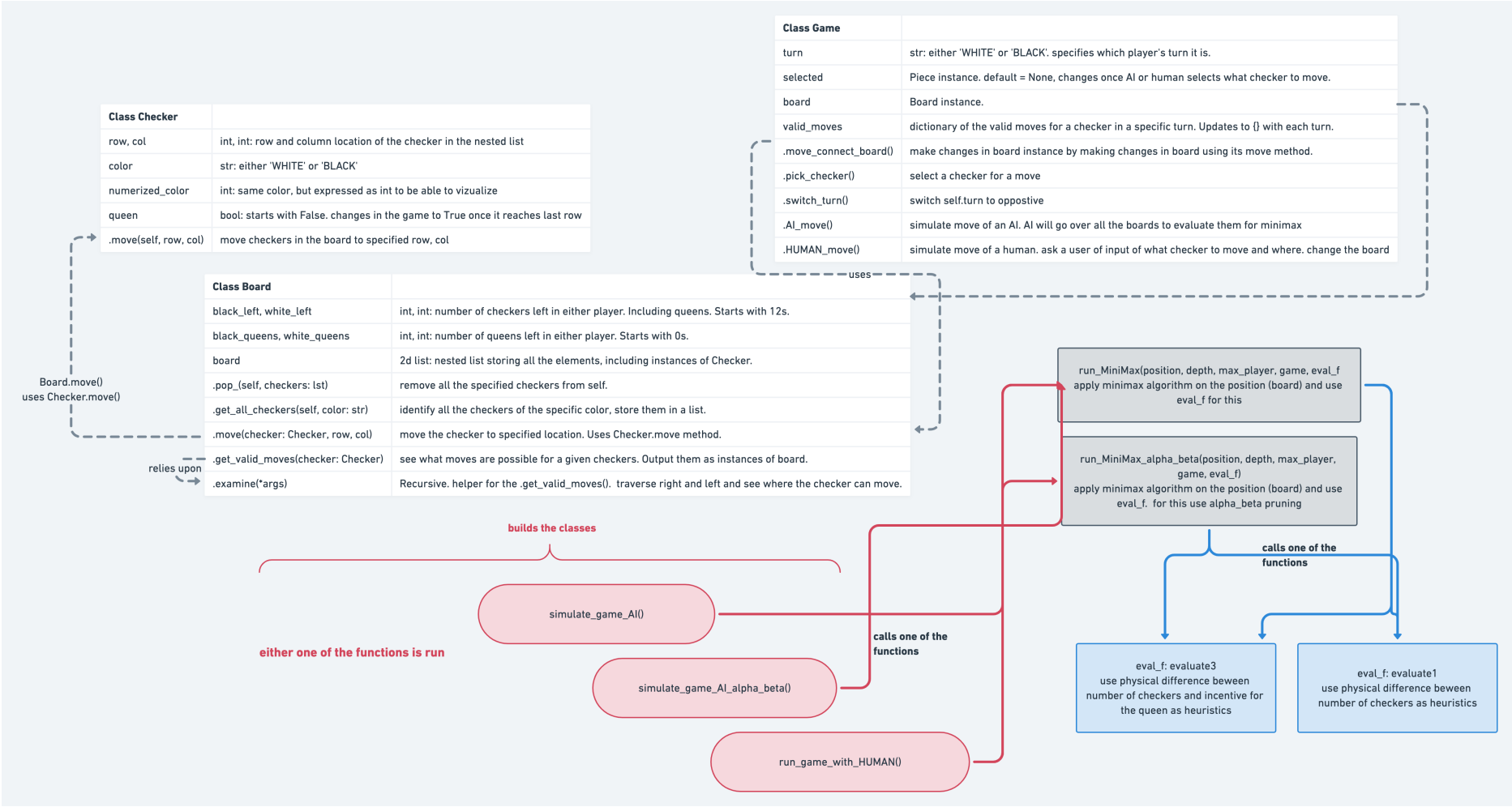


Figure 2. Diagram of the classes and functions implemented for the checkers game. The tables manifest the classes, its attributes and methods. The red modes manifest the functions that the user can call at first. Next, two versions of MiniMax functions (with and without alpha-beta pruning) are implemented (in grey). They rely upon either one of the evaluation functions (in blue). See image in better resolution here.

3. Analysis of Solution

Using the Notebook attached, you can play with the AI I developed. Furthermore, I made AIs play against each other to determine which evaluate function is better. There are two evaluation functions:

$evalF_1 = Num_{white} - Num_{black}$. This function is a simple substraction of the number of checkers which allows us to incentivize AI to minimize the checkers.

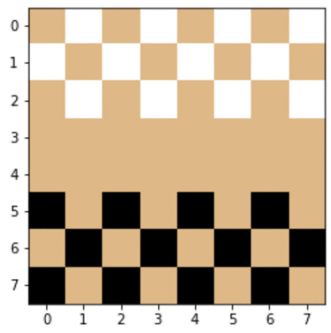
$evalF_2 = Num_{white} - Num_{black} + (0.3 \times Num_{white} - 0.3 \times Num_{black})$. This function does the same as the previous one, but also incentivizes AI to make queens.

Given that the algorithm does not have any random component (it deterministic), I only could run a single experiment for each of the variations. See below the dynamic gifs displaying the game. It was clear from experiments that the higher depth of the AI, the better it plays, which can be rationalized by the fact that AI, with a higher-depth search tree examines more options and finds a better solution. Furthermore, the $evalF2$ performed better, which is also intuitive given that the queens are capable of catching more of the opponent's checkers.

Tournament 1:

`AI-depth3` **VS** `AI-depth1`

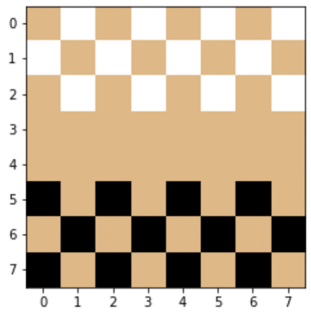
With `eval_fuction1`. White is higher depth.



Tournament 2:

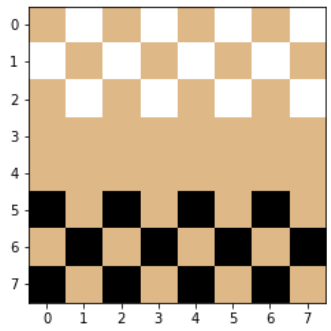
`AI-depth5` **VS** `AI-depth1`

With `eval_fuction1`.White is higher depth.



Tournament 3: `AI-evalF1` **VS** `AI-evalF2`

With depth3 and α-β. White is evalF2.



Furthermore, the alpha-beta pruning version substantially decreased the time it takes to run the algorithm. Indeed, the time taken is just a proxy measurement and is not as accurate because of the potential other computational processes happening in the meantime that could affect either of the results. A better measurement would be the number of minimax evaluations done. Nevertheless, one can see in figure 3 that running algorithms of the depth of 1 has no difference in time, whereas with each extra depth added the time taken by a simple minimax grows exponentially.

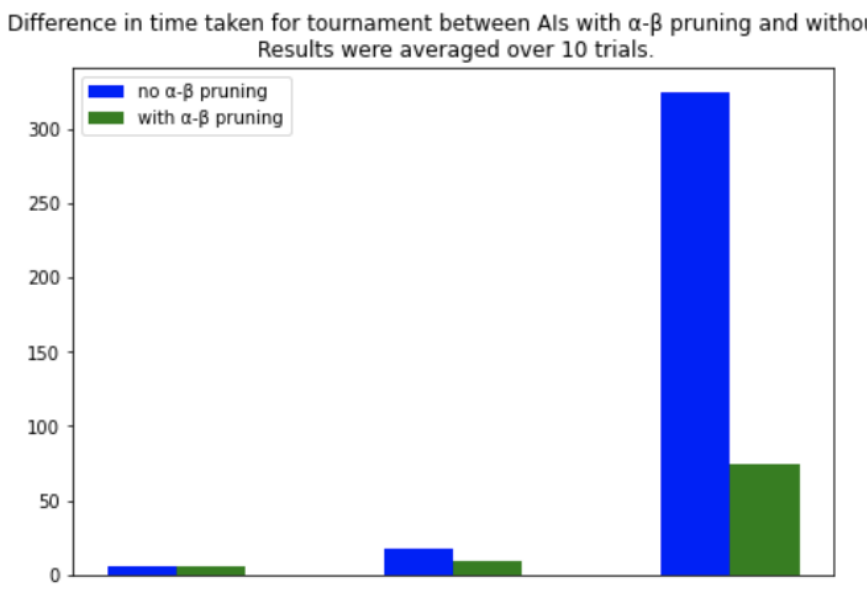


Figure 3. Time taken to run the game between two AIs. First bars resemble running the algorithm with just a depth of 1. The second chunk of bars is for the depth of 3. And the last one is for the depth of 5.

References

[1] Draughts. (2021, April 18). Retrieved April 23, 2021, from https://en.wikipedia.org/wiki/Draughts

[2] Schaeffer, J., Burch, N., Bjornsson, Y., Kishimoto, A., Muller, M., Lake, R., . Sutphen, S. (2007). Checkers Is Solved. *Science, 317(*5844), 1518-1522. doi:10.1126/science.1144079

[3] Allis. V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. University of Limburg, Maastricht, The Netherlands. ISBN 90-900748-8-0.