CSC111 Final Project: **Check Your Move!**

Dimitrios Chlympatsos, Samarth Sharma, Lakshman Nair, Peter James

April 1, 2023

Problem Description and Research Question

Checkers is a strategy board game played on a 8×8 board with each player having 12 pieces, called "stones". The player moves one stone when it is their turn, and the stone can only move diagonally forward, or capture other pieces by "jumping" diagonally forward (in one move, there can be as many jumps as potential captures). Moreover, when the stone reaches the end of the opponent's side, which makes the stone a "king," it can also move diagonally backward (and, analogously, capture rival stones backwards, as well). The game ends when either one of the players loses all of their stones (the player with stones remaining wins) or when a player is left with no possible moves to make (where the player that still has possible moves to play wins).

The main objective of this project is **to create artificially intelligent algorithms -*based on decision tree recursion*- that can -on average- win against a random player for a *modification* of Checkers** (this modification will be analyzed below). This objective also requires us to make an (object-oriented) implementation of the Checkers game (i.e. a Checkers interface), and also a visual component to display our results. Our motivation for this project stems from the second assignment, where we were guided to create intelligent algorithms to play "Wordle". The ability to create an algorithm that, throughout a whole game, performs better than a random player, piqued our interest and caused us to explore whether this is possible for other turn-based games or zero-sum board games in particular. During our research, we came across Checkers, a zero-sum game that is conceptually similar to Chess. But while Chess has a branching factor of 35 [1], Checkers has a branching factor of 10 (Eppes, 2019). So, although not as complex as Chess, the number of possibilities grow exponentially with the number of moves played, making the task of determining effective algorithms challenging one, and one we were all excited to dive into.

Our Modification of Checkers: For our project, we made two simplifying assumptions with regard to the rules of the game:

1. There are no kings. Once a stone reaches the opposite side, it stays there and no longer moves.
2. In the official Checkers Game, there is the possibility of consecutive captures wherein there are multiple opposing stones in the path of the player stone which allows it to capture multiple stones. We have made one modification to consecutive captures in our game which only allows for consecutive captures to take place in a "straight" diagonal line with no turns. An example of a valid consecutive capture is when a stone moves from (1,1) to (5,5), this is a valid move and stones at (2,2) and (4,4) are captured.

These simplifying assumptions slightly reduce the complexity of the design of the Checkers interface. We did this because our focus is not on making a perfect implementation of the Checkers game, but rather on designing intelligent algorithms (based on tree recursion) that play optimally (for the given game).

Due to these simplifying assumptions, it is natural that the winning condition adapts to the modified rules. For our modification, a player A (playing against a player B) wins when one of the following is satisfied (these conditions are implemented in the `CheckersGame.get_winner()` method, discussed in the Computational Overview section):

1. B has lost all of his stones.
2. A has no possible moves and he has more stones on the board.
3. B has no possible moves to make and A has more stones on the board.

[1]This means that, on average, for any game state and for any turn, there are 35 possible moves a player can make.

Computational Overview

Checkers Implementation

First, we had to design and implement an interface that allowed playing a Checkers Game, and supplying all the necessary data and metadata about that game. Our interface consists of three classes. All these classes are located in the `structures.py` module. Although the in-code documentation of these classes is very thorough, information about them follows.

1. `Stone`: This class represents a particular piece in a game of Checkers. It has four instance attributes: `ID`, `state`, `position`, and `color`. Furthermore, it includes the methods `kill` (which changes the state of a piece from `True` to `False`) and `get_poss_moves`, which returns all possible moves that this particular stone can make given a game state. It also includes two private helper methods (since these are private, they are not part of the interface).

2. `Move`: This class represents a Move. It has two instance attributes: `stone_to_move` (which is the Stone object that this move relates to) and `new_position` (a tuple for the new position of the stone). As such, a `Move` object *uses-a* `Stone` object. Furthermore, this class includes the important method `positions_captured`, which returns any positions that would be captured if this move were to take place [2].

3. `CheckersGame`: This is the "master" class of our interface, and represents the state of a Checkers game (in other words, it represents a Checkers game itself). In includes six instance attributes:

    - `board`: This is a nested list (matrix) that represents the board on which the game is played. Any particular "cell" in this matrix stores either the `ID` of the `Stone` object that "occupies", or `-1` if it is unoccupied. The helper method `_initialize_stones_and_matrix` (existing in the `structures` module) is invoked to return a matrix (nested list) of stone IDs in their configuration at the beginning of a Checkers game.
    - `stones`: This is a dictionary with all the `Stone` objects in this game, mapping `ID` to the corresponding `Stone` object. As such, a `CheckersGame` object has a `Stone`, in a *one-to-many* relationship.
    - `black_history`: A list with all the moves that have taken place by the *Black* player up to this point in the game, in chronological order. As such, a `CheckersGame` object also has a `Move`, in a one-to-many relationship.
    - `red_history`: A list with all the moves that have taken place by the *Red* player up to this point in the game, in chronological order.
    - `black_survivors`: A set with all `ID`'s of black stones that are still alive.
    - `red_survivors`: A set with all `ID`'s of red stones that are still alive.

    Furthermore, this class has several crucial methods. Firstly, `record_move` is a mutating method responsible for registering the occurrence of a move (represented with a `Move` object) by updating all instance attributes (i.e. CheckersGame.board, CheckersGame.stones, etc.) as required. Also, it contains a `get_black_moves` and a `get_red_moves` method that returns a list with all the possible moves that the black and red player can make given the current state of the game. Additionally, it includes a `_copy` method that performs a **deep copy** on this `CheckersGame` object; this means that a new object for the container is created, new objects for the attributes of the container, for the attributes of the attributes, and so on. The result is that we fully avoid "mutation at distance", which would disastrous for the intelligent algorithms that will soon be described (that want to play out what would happen if a particular move was made, without actually making that move). The method `copy_and_record_move` utilizes the `record_move` and the `_copy` methods.

[2] Here lies the reason for our second simplifying assumption (that capturing can only take place in a "straight line"). If we allowed for non-straight line consecutive captures, there would have been an ambiguity in the positions captured by a move. For example, if a stone moved from (2,2) to (2,6), this move could only take place if there were consecutive captures taking place but now there are two possibilities for which positions are captured, one is that there are stones at (3,3) and (3,5) and these are captured OR there are stones at (1,3) and (1,5) and these are the positions captured. These possibilities created a sense of ambiguity and uncertainity in our implementation thus leading us to making the simplifying assumption of only allowing "straight" line captures

Finally we have the `GameTree` class which is used for the `Aggressor` algorithm and for the `PrunelessMinimaxerwithTree` implementation.

`GameTree` class:

Attributes

- `move: Move | str`. The move attribute is used to expand the GameTree by using record_move and add_subtree
- `score: Optional[float]`. This is the score attribute, used for the minimax algorithm.
- `_subtrees: list[GameTree]`. The possible moves that the black player can make. This should be updated every time the `record` methods are called.
- `game: Optional[CheckersGame]`. This attribute contains the game from which the gametree being initialized was constructed. It is optional, because while it is necessary for the Aggressive Algorithm, it is not needed for minimax.

Methods

- add_subtree: This method adds a new subtree to the current list of subtrees of a gametree.
- get_subtrees: This method returns a list of subtrees for a gametree.
- getaggroscore: This method takes a game tree and a depth as arguments and recursively calculates a score for each move in the tree. The score represents the expected number of opponent pieces that the move will capture in the next sequence of moves, where the sequence is as long as the specified depth. For example, if the score for a move is 3, it means that the player expects to capture 3 opponent pieces in the next sequence of moves if they choose that initial move.
- increasedepth: This method takes a game tree, and mutates it, increasing its depth by 1. It does this recursively, by finding leafnodes of a gametree, and replacing them with the same leafnodes, except with depth 1, instead of depth 0. This method allows us to optimize the makemove method for the Aggressive algorithm, as it is faster to take a subtree of a previous game, and extend it to the required depth, instead of initializating a whole new gametree.

Minimax Search Algorithm

Related Module: `minimax.py`

Introducing Minimax

Minimax is an AI decision-making algorithm that is heavily used in game theory (Lazar, 2021). I will begin by explaining the main idea behind Minimax, and proceed with a more detailed explanation of how it makes a choice (in our context, a choice for which move to make).

In zero-sum turn-based strategy games, ”great” players are usually those who choose what move to make by looking forward many moves and taking into account how he can maximize his chance of winning, assuming his opponent plays logically. This is precisely the main idea behind the Minimax search algorithm.

To understand how the Minimax algorithm works, we begin a decision tree, where each node represents a particular state (or a move that led to this state). The decision tree is at most as deep as a preset depth limit, and a leaf represents a node with either a winning/losing state, or a node that is at the maximum allowed depth.

There are two agents (that could be viewed as two players); the *Maximizer* and the *Minimizer*. Note that all nodes at depth d of the tree represent possible d'th moves.

Now, each node of these nodes needs to be assigned a score (this is the main part of the minimax), which happens as follows: if it is the Maximizer's turn (at the particular depth where the node is located), then he chooses the child (move) with the highest associated score, and this highest score is the score that will be assigned to the node itself. Similarly, if the player is the Minimizer, he chooses the child with the lowest score, and again this lowest score is the score that will be assigned to the node itself.

But notice: A Maximizer node selects the child node with the highest score, and each child node (which is a Minimizer) selects the child node with the lowest score, and each such child (which again represents a Maximizer) selects the child with the highest score, and so on. This means the process of score assignment is mutually recursive!

So what is the base case? The base case occurs when a node represents a winning/losing game, or when a certain depth has been reached (Checkers has a branching factor of 10 and each game consists of dozens of moves, so computing a complete tree is computationally infeasible). In that case, we use a utility function –formally called a static evaluation– a function that takes the game state of a leaf and returns a score for that state. Once the terminal/leaf nodes have been evaluated, we start "moving up" in the tree and peeling off layers of the recursive call stack (until the root of our tree has a score) in the way described above, with the Maximizer choosing the child with the highest score and the minimizer choosing the child with the lowest score.

Note that (unlike algorithms seen in A2), the Minimax algorithm creates and uses a new decision tree at every turn of the player that uses it.

Implementation of Minimax

Our implementation of the Minimax algorithm relies on two methods that are mutually recursive (i.e. one calls the other until a base case is reached); a maximizer method and a minimizer method. We wrote three implementations of a Minimax player.

1. `PrunelessMinimaxerWithTree`: This player uses the `maximize_with_tree` and `minimize_with_tree` functions (from the `minima.py` module), which recursively creates and assigns scores to a decision tree (a `GameTree` object) for a Checkers Game, and actually returns the produced tree with all the scores.

2. `PrunelessMinimaxer`: It turns out that actually creating a tree is not necessary, and the Minimax algorithm can still recurse down the possibilities (recurse down the hypothetical decision tree) and choose the best scores/moves without simultaneously building the tree. T

   Thus, this player uses the `maximize` and `minimize` functions (from the `minima.py` module), which perform the same job as the functions `maximize_with_tree` and `minimize_with_tree` but without actually creating and returning a new tree.

   This leads to an observable improvement in efficiency, since there is no time and space spent on actually building and returning a tangible `GameTree`.

3. `PrunefulMinimaxer`: This is my "apex" player, which uses a modified version of Minimax, Minimax with alpha-beta pruning.

   In the same way you prune "bad" branches of a tree so the tree as a whole grows more using the same resources, pruning a search tree refers to fully ignoring ("cutting off") subtrees that we are certain will not help the search; so we do not recurse on these pruned subtrees, thus significantly improving efficiency.

   In the context of the MiniMax search algorithm, to perform pruning, we add two more parameters to the MiniMax algorithm; alpha, which is the best score (best option) that the maximizer can choose (with the information so far) down the path from the root to the node we are in now, and beta, which is the best score that the minimizer can choose in the path from the root to the node we are in now; assuming that both players play optimally. Note that if we are using the implementation under which we have two separate maximize and minimize functions, then, from our understanding, to perform pruning you again need to pass two additional parameters to each of these two functions, let's say gamma, which is the best score the opponent can get (best can mean highest or lowest, depending on which function we are in), assuming they play optimally.

   I implemented this method of "pruning" the game tree with the functions `maximize_with_pruning` and `minimize_with_pruning`.

   Based on published research, on average, alpha-beta pruning allows the minimax algorithm to go almost twice as deep in the same amount of time. Therefore, this algorithm is the most efficient of all three, and because it is much faster, it can go deeper, so it can also perform much better.

Note that the mutual recursion taking place with the `maximizer` and `minimizer` functions (and all their modifications) is an example of recursion that does not (and must not) mutate the accumulator; in this case, the accumulator is the `state` parameter (which is a `CheckersGame` object. The reason is that we want the algorithm to "look ahead" by seeing what would happen hypothetically if certain moves were made, but without actually making these moves (it would be completely wrong to mutate the gamestate).

Furthermore, I have defined a utility function called `utility` (that serves as the "static evaluation function" for the Minimax algorithm) in the `minimax.py` module. After trying out a few different candidate functions, I ended up with this one, which seems to do be quite rational. Generally, a utility function is meant to return a score for a given state, where higher scores suggest this state is good for one player (in our case, the Black), and lower scores suggest the opposite (i.e. that the state is good for the Red). My implementation works as follows: if the passed state is a winning state for the black player, the function returns positive infinity; if it is a winning state for the red player, it returns negative infinity. If neither of these two is the case, it takes into account two other metrics:

1. The signed difference between the number of black survivors and the number of red survivors.
2. The average distance of each black stone to the opposite side. I believed that the closer on average black stones are to the opposite side, the more likely they are to win the game in the end.

Aggressive Algorithm (Simple)

The minimax algorithm is a well known algorithm that has been heavily investigated in the past. We decided to explore another Checkers algorithm that uses an entirely different strategy. We called this algorithm the "Aggressive Algorithm", because it's primary goal is to play moves which capture the most pieces.

First we created the SimpleAggressor algorithm which inherits the Player class. This Player is designed to make the most aggressive move that it can play next. This player is initialized with a color, and has one method: the play method.

The play method for the SimpleAggressor player takes into account the current state of the game and was implemented using the following method. Note: It has to be the players turn for this method to execute. First we calculate which moves are the most aggressive by getting a list of moves that the player can make next, and then filtering for the moves which capture the most pieces. If there are ties, we choose randomly from the set of most aggressive moves. The play method returns, with output being the chosen move. In this way, the player selects a move that results in the maximum reduction of the opponent's pieces in one turn.

Overall, this implementation of the "Aggressive Strategy" was simple, as it only looked one move ahead. Immediate flaws in this algorithm are evident, as it focuses on capturing enemy pieces even if it came at the cost of the players own pieces. Moreover, it does not take into account any strategic long-term considerations or potential future moves that may arise from its current choice. Perhaps if we made the algorithm look further ahead, some of these problems would be fixed.

Aggressive Algorithm (Advanced)

The Advanced Aggressive Algorithm attempts to fix the issues seen in the Simple Aggressive Algorithm. It does this by taking into account long-term strategic considerations, by prioritizing the reduction of opponent pieces in the longterm, rather than just in the next move alone.

We created the AdvancedAggressor algorithm which inherits the Player class. This Player is designed to make the series of d moves which capture the most pieces. Here, d is an input value that the algorithm must be initialized with, and it is the number of moves that we want the Algorithm to calculate ahead, or the "depth". Advanced Aggressor is also initialized with the current CheckersGame.

The move method for this player uses a more sophisticated strategy that depends on constructing a game tree and choosing the move with the highest or lowest score.

We implemented the move method in the following way. First we take a CheckersGame object that represents the current state of the game. We then call gametreewithdepth on this object to create a gametree for the current state of the game, of depth equal to the depth specified in the AdvancedAggressor Player. After this we find the Aggression Score of all subtrees of the gametree we generated. We do this using a getmove method that we implement for this player, which works by calling getAggroScore on all subtrees of the generated gametree.

The function getAggroScore works by recursively assigning an aggression score to a gametree that corresponds to the average number of captures across all moves in that tree up to a certain depth.

Once this score is calculated for each subtree using the getmove method, the method finally finds the subtree with the maximum aggression score, and returns the move for this subtree. The move method for the algorithm then executes this move.

Immediately, this move method looks more sophisticated than the Simple implementation for our Aggressive Strategy, because it thinks ahead. However one issue we had was that it was computationally slow. Each time we made a move, the Player would create need to recursively create a new gametree object consisting of all moves up to a certain depth based on the current state of the game. After this we would have to assign the subtrees of this gametree a AggressionScore, which is done recursively again, by checking all nodes in these subtrees.

To optimize this process, we made CheckersGame an instant attribute of each GameTree. This ensured that each GameTree contained the CheckersGame which was used to create it. This helped speed up the getAggroScore method, because we needed to access the CheckersGame for every node in a gametree to check how many pieces were captured.

Additionally, we made an "update" method for the Player. This greatly reduced the computational demand. Each time the player made a move, it would need to initialize a new gametree from the current state of the board. However using this update function, after the first move, when the player makes a move, it takes the relevant subtree of the current gametree, increases its depth by 1, using the increasedepth method, and then updates the players gametree to this subtree. This means that we don't have to compute an entire new gametree each time we need to make a move.

Overall, this player strategy is designed to choose moves that are expected to result in the maximum number of opponent pieces being captured, based on a game tree evaluation. The depth parameter controls how many moves ahead the player considers, and a higher value leads to better evaluations but also takes longer to compute.

Visualization

Use of `pygame`: The Pygame library plays an integral role in the visualization of our project. It is used to visualize the simulation of a full checkers game between two AIs and generate an interactive interface that allows users to pick which AIs they want to see face one another. While this already requires extensive use of Pygame, we decided to create a menu, guide and winner screen to improve the aesthetic and user experience of our program. All actions that run through Pygame represent the front-end of our program. To provide further insight as to how the program works and the process connecting the back-end and front-end, there are functions and a class that must be broken down. The simplest way to analyze the whole body of work is to break down the structure first, and then each function relative to when it is called in the code.

Thus, the visualization was structured using variables that acted as a toggle that would trigger when to draw each screen. For example, running, menu and is_guide are examples of these types of variables which are simply Boolean values set to True when a certain window or screen must be generated. Once they are set to True, the code checks for this condition using an if statement and draws the corresponding screen for this variable being True. Since the first visualization the viewer should see is the menu, the menu variable is initially True, which causes the draw_menu() function to be called. This function draws a background, rectangles for buttons and then it renders fonts and positions them. This was accomplished by downloading .ttf files and using .fonts which is one of the many modules of Pygame we used (which was possible due to the pygame.init() call). To make the simulation interactive we ran an event loop that tracks events that happen whilst the user is in the Pygame window. This keeps track of positions where the mouse clicks which allowed us to specify the boundaries of the button. When the event mousebuttonup was registered in these areas menu would toggle back to False, and another would become True, generating a window like draw_guide or draw_choose_ai instead. These are buttons in the menu, all of which work as designed. The guide is part of the user experience and explains the important aspects of the interface to someone unfamiliar with the specifics of our implementation or the game of checkers. Then we implemented another button-like implementation with mouse events so that the user can get back to the menu without closing the window and reopening it. The exit feature works similarly.

As expected, the most significant button is the 'Play' button which uses the toggle process to draw the screen which displays the possible AI combinations that can be simulated. While we could have, and originally planned to, only simulate one intelligent algorithm to satisfy our goal of creating an artificially intelligent player, we thought it would be much more interesting to implement multiple different types of AI players. For curiosity and symmetry, we also allowed each AI combination to change piece colour. A significant connection between the front-end and back-end occurs at this point. Once the button is selected, the toggle for that button initializes the two AI it is labelled with. Then, these AI are set against each other using the run_game function and a list of CheckersGame objects are generated. Using these, we run the draw_boards function which calls the two most important drawing functions. These will be explained first, then the draw_boards will appear simpler.

The first of these functions is the draw_squares function which fills a red background and then draws black squares on the even positions in each column and row. This function is the first statement in the next function, draw, which draws a checkers board based on the CheckersGame.board attribute. This is because the draw_squares function is responsible for drawing over the pieces of the old game state so that new pieces can be drawn on top of a blank board.

The draw function is the most important function in the visualization and required extreme attention. The parameters include a screen, the same as draw_squares, but also a game parameter that is a CheckersGame object. This game allowed us to access the game.board attribute, which as aforementioned, is a nested list in the format of a checkers board with pieces represented by IDs. By creating a nested loop to access these IDs we could access the Stone object corresponding to the ID by indexing the game.stones attribute with that ID. Once we have the Stone, we must draw it, which creates an organizational issue with separating the back-end from the front-end. Thus, we decided to implement a similar class to the Stone class, Piece, responsible for only front-end methods. The Piece class initializes the position of the Stone object and its colour, the attributes necessary to call the pygame.draw.circle built-in function. Since Piece's main purpose was to draw the stone at that ID, a method to calculate its position in the centre of a square was created, and then a method to draw the piece itself using pygame.draw.circle. Thus, for each ID in the CheckersGame.board, a corresponding piece would be drawn by extracting the back-end data from the class and implementing it with the front-end class and functions.

Using these two key functions, draw_boards, which has as a parameter a list of CheckersGame objects, loops through each individual game object and calls draw on them to draw the board for that given state. As the loop runs the game.board changes as simulated moves are made, which mimics the progression of a game.

Since the board is read each column at a time, it will be drawn with the pieces perpendicular to the viewer rather than in front of them. Since it is easier to watch the game from the perspective of a player, we rotated the board 90 degrees and then updated it. As part of the visualization, we also made a half second delay per move so that the viewer has time to see what is happening, while not spending too long on each move or game.

Once this loop finishes and the simulation ends, a last screen is automatically generated. The draw_win screen tracks the winner using the CheckersGame method get_winner. This enabled us to create two different visuals depending on which side won the game. This signifies the end of the game and the end of the visuals.

The visual aspect is critical for analyzing the differences in behaviour of the different AI. It was a challenging and rewarding aspect of the project that provides character and depth to the AI algorithms we generated, while also adding an appealing aesthetic and unique user experience.

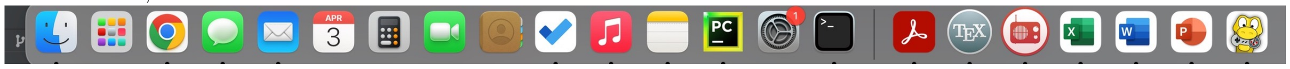
Running Our Program

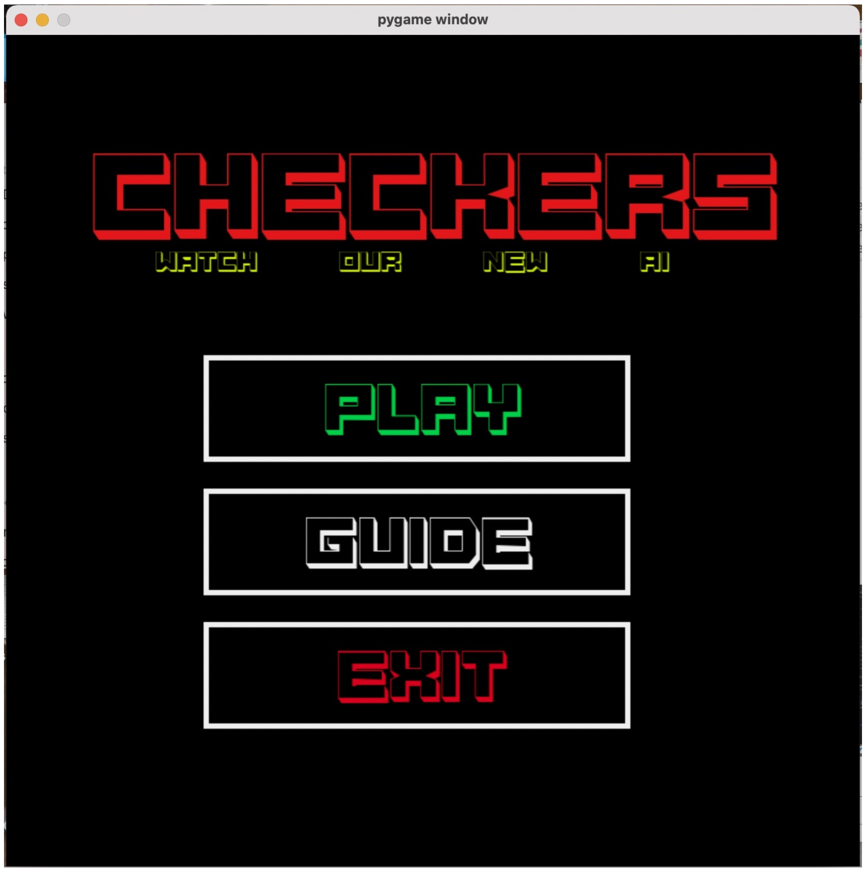
Install/update `pygame` and `plotly` libraries from `requirement.txts` correctly.

Ensure `font_files` is decompressed properly into a folder with the same name and stored in the same directory as the `main.py` file to ensure the directory calls to access these fonts in the files are correct.

When running `main.py`, opt for the "*Run File in Python Console*" option since our prompts require inputs from the user. The user will be initially asked to prompt 'Y' or 'N'; Typing 'Y' will launch the game, while typing 'N' will request further inputs which are described below.

When 'Y' is inputted, there will be a pygame window formed and presented in your dock. Locate the `PyGame` snake icon (in your dock) and press it (because the PyGame window, although it will surely open, might not pop up to you). Follow the instructions on the screen to play the game. Sometimes buttons may need double-clicking to load the next window (usually when choosing what type of player configuration you want to see); but please be patient with the application since the algorithms are heavy and the games (which consist of 50 or more moves) take time to execute, record and draw.



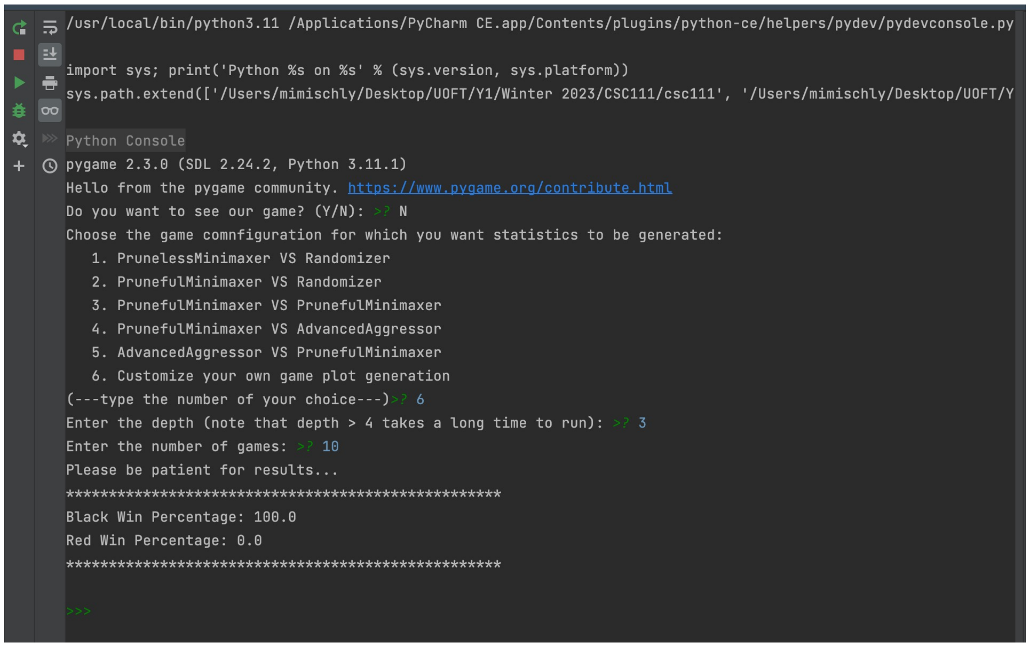


If 'N' is inputted [3], you will be able to observe plots of simulations of numerous games with different configurations of black/red players. You will be prompted to type an integer from 1 to 6 (these prompts can be observed in the screenshot attached below).

Typing any number from 1 to 5 will automatically lead to the generation of a plot of the configuration appears next to the number (as seen in the attached image). Please be patient, as this may take some time. Moreover, once the simulation has run you will see printed the percentage of the black player's wins and the percentage of the red player's wins for that particular simulation.

If you type 6, you will be further prompted to input a depth of your choice (from a feasibility perspective, this should not exceed 4), as well as the number of games to run. Then, the program will automtically run a simulation (with as many games as the amount you inputted) where the black player is `PrunefulMinimaxer` operating at the depth you inputted and the red player is a Randomizer. You will see a plot of the simulation and the corresponding stats printed in the console.

[3] ignore the `pygame` window that might appear in the dock



```
/usr/local/bin/python3.11 /Applications/PyCharm CE.app/Contents/plugins/python-ce/helpers/pydev/pydevconsole.py

import sys; print('Python %s on %s' % (sys.version, sys.platform))
sys.path.extend(['/Users/mimischly/Desktop/UOFT/Y1/Winter 2023/CSC111/csc111', '/Users/mimischly/Desktop/UOFT/Y

Python Console
pygame 2.3.0 (SDL 2.24.2, Python 3.11.1)
Hello from the pygame community. https://www.pygame.org/contribute.html
Do you want to see our game? (Y/N): >? N
Choose the game comnfiguration for which you want statistics to be generated:
   1. PrunelessMinimaxer VS Randomizer
   2. PrunefulMinimaxer VS Randomizer
   3. PrunefulMinimaxer VS PrunefulMinimaxer
   4. PrunefulMinimaxer VS AdvancedAggressor
   5. AdvancedAggressor VS PrunefulMinimaxer
   6. Customize your own game plot generation
(---type the number of your choice---)>? 6
Enter the depth (note that depth > 4 takes a long time to run): >? 3
Enter the number of games: >? 10
Please be patient for results...
***************************************************
Black Win Percentage: 100.0
Red Win Percentage: 0.0
***************************************************

>>>
```

Details about the Game Launcher: Our program produces an interactive display before the execution of the simulation. The primary interface is the menu screen that is generated in a `PyGame` window when the game is run. This is only generated if, after the `main.py` file is run, the user types 'Y' in response to the question and opens the `PyGame` window in their computer dock. This menu includes three options, all fairly self-explanatory. For new users, especially those who lack information on the way the simulation works, the guide is the first place to go and is likely a helpful feature. It can be simply selected using a cursor, which creates a new screen outlining the game of checkers and the way our simulation works. This was implemented to help all users traverse the simulation and familiarize them with the fundamentals of checkers itself. Once the user is finished reading the text in the guide, they can press the bottom line/button of the guide which will send them back to the menu so they do not have to close their window to actually start a game. Unless a user wants to leave, for which they can use the exit button, they will select the play button with their cursor next. This will take them to another screen that allows them to select which AI combination they would like to simulate and what what the configuration should be (e.g. black is `Pruneful Maximizer` and red is `Randomizer` or any other configuration). Once they have selected the button with their mouse, they can wait a few seconds and the game will begin and execute smoothly. Once the game is finished, a "win screen" will be generated, signifying the end of the simulation, at which point the user can exit the PyGame window at their own discretion. As you will be able to see, the AI's always win against a `Randomizer`!

Changes to Project Plan

One of the major changes between our project plan and proposal was changing from a playable checkers game to a simulation. While we originally planned for a person to be able to select a piece with their cursor and make a move themselves, we decided to switch to a simulation to focus more on making and improving our AI algorithms. While it would have been interesting, we were having difficulties with selecting and moving pieces around. It also seemed that the project would have been hyper-focused on the user interface aspect opposed to the back-end algorithms which were our priority. There were a number of difficulties that implementation posed all around, which is why we decided to focus on creating a simulation. For this reason, we focused on generating and improving our AIs more than originally intended.

Moreover, during the creation of the game we made a couple simplifying assumptions. We acknowledge the differences between traditional checkers and our simulation in the guide so that users are not confused when they do not see a piece become a king. While we originally intended to include this aspect in the game, a piece that could move in four different directions made the computation for the number of possible moves and future moves too large. Including this condition would have likely drastically increased the time it takes for the simulation to run. Additionally, we made the simplifying assumption that if a piece is capturing more than one piece it cannot change direction while capturing (as explained earlier). This is because it was too difficult to track the moves on either side of an opposing piece for each piece on the board. Again, this likely would have caused some computational difficulties and increased the amount of time and lag for the program.

Discussion

After several weeks of consistent work, the results are very promising, but getting to these results was not an easy road and we faced multiple challenges along the way. We will begin by discussing some of those challenges.

Limitations/Challenges

Mutation of `Stone.position` in `CheckersGame.record_move`:

In our initial implementation of record_move we came across an issue that posed a great challenge to us, but the lesson we learned was very important; initially, in our `record_move` implementation, we had an assignment statement that mutated the moving stone's position to be the new position given by the move and then added the move to the `black_history` and `red_history` accordingly. We thought this implementation was right and would not cause any issues and it even passed our first round of testing inputs. However, an issue arose when we attempted to draw games using the `draw_boards` function which depended heavily on `CheckersGame.record_move`. After several hours of debugging and testing, we recognized our error; the mutation to the moving stones' position was causing the problem. This is because the `Move` class has two attributes; the Stone and the `new_position`, and the `Stone` class has a `position` attribute that stores the position of the stone. When we mutate a stone's position to become the new position after recording a move, then the stone "loses" its initial position, which causes problems when trying to draw game boards because we want the stone to go from its initial position to the next one, and not from the position it ended up at the end of the game. To explain with an example, let's say we are following the black stone with `ID = 10`. Assume that this stone in the end has reached position `(4,7)`. With our initial implementation of `record_move` we would be adding all of the moves that stone 10 made into black_history which would then contain a list of Moves. But since after all the mutations of `Stone.position`, the black stone 10's position ends up as `(4,7)`. So if stone 10's first move was to go to `(7,3)`, this would mean that in `black_history` a `Move` would be stored to move stone 10 to that position, but since the final position of stone 10 is (4,7) the `Move` stored in black_history is attempting to move stone 10 from `(4,7)` to `(7,3)`, which is an invalid move, thus causing the game boards to be erroneous. This error is similar to the concepts discussed in "Mutation at a distance" and posed an interesting challenge to us. We ended up resolving this error by using `copy.deepcopy()` which is briefly explained on page 2. `copy.deepcopy(move)`, where move is a `Move` object, would return a new `Move` object, and all the attribute of that `Move` object will also be new objects (i.e. they will not be aliases with the attributes of the other `Move` object).

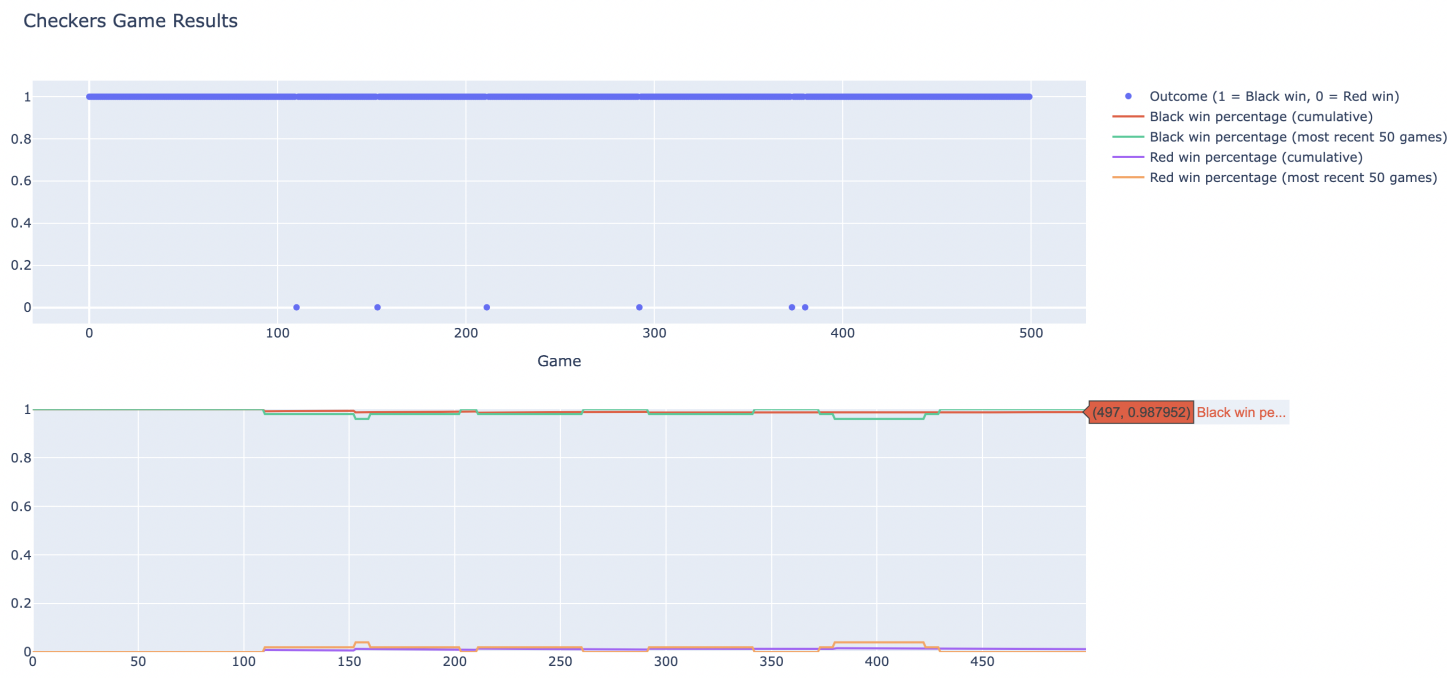
Minimax Algorithm

As noted above, we implemented three players for the Minimax decision tree algorithm, with each player utilizing a (different) version of Minimax; `PrunelessMinimaxerWithTree`, `PrunelessMinimaxer`, `PrunefulMinimaxer`. However, for a given depth d, all these algorithms are equally effective since the strategy used to assign scores and select amongst the possible moves is the same. What differs is the running time efficiency. The slowest is that `PrunelessMinimaxerWithTree`, since for every node it creates a tree. The second slowest is the `PrunelessMinimaxer`, and the fastest of all is the `PrunelfulMinimaxer`, which is very efficient because it performs alpha-beta pruning, avoiding recursion on subtrees that we know for certain will not contribute to our score assignment.

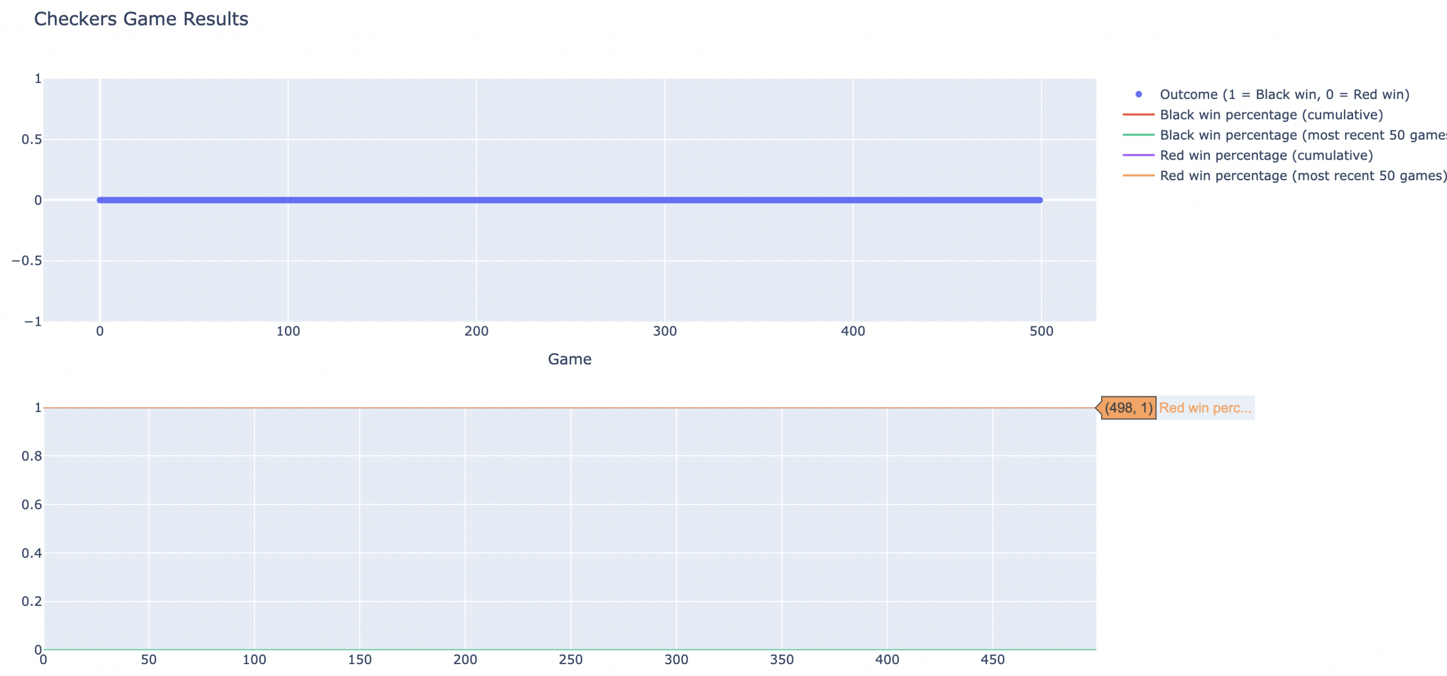
We decided to run numerous simulations of Checkers games with the black player and red player taking various roles, for different depths (we did this using the `run_games_plot` function from the `simulation.py` module.

Thankfully, the Minimax players perform extremely well against a random player (the `Randomizer`, who chooses completely randomly amongst all possible moves for a given state and turn), typically winning against a random player almost 100% of the time!

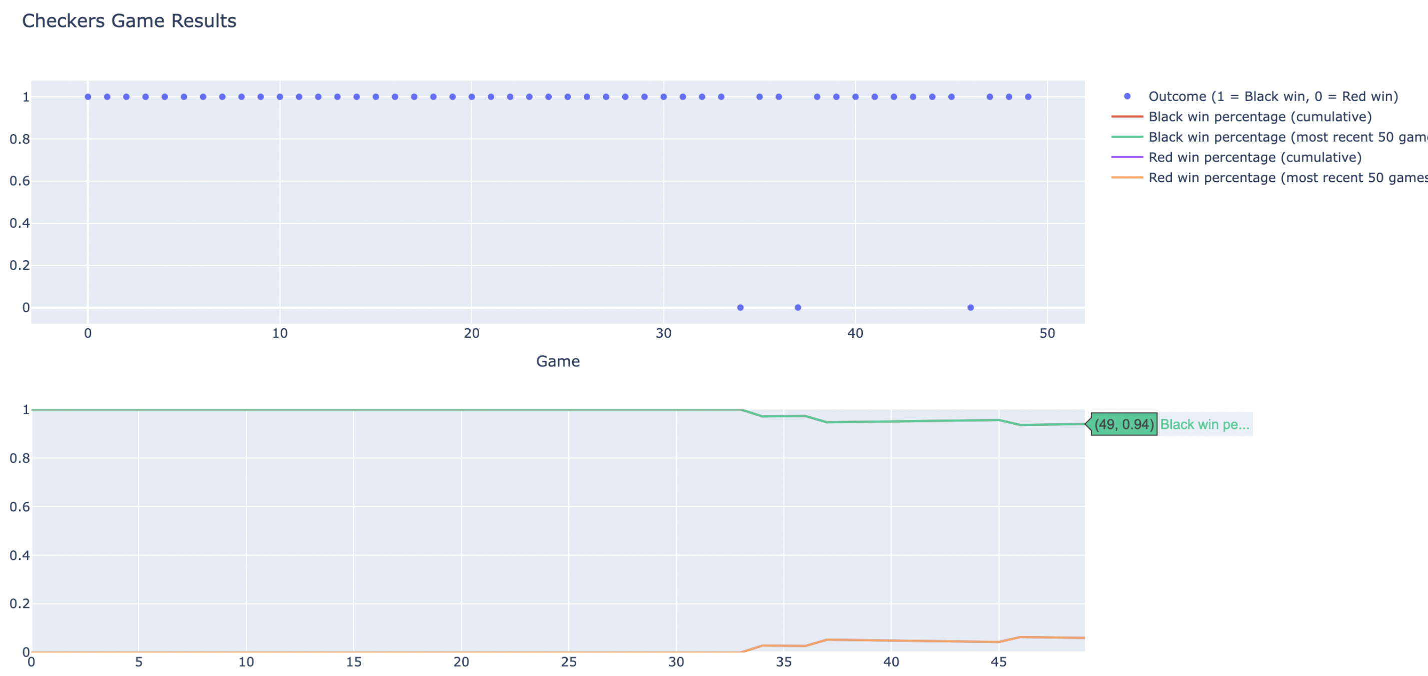
The below visual is the result of a simulation of 500 complete Checkers games where the black player is the `PrunefulMinimaxer` algorithm with depth 2, and the red player is a `Randomizer`. This plot shows that the Minimax algorithm won a staggering 98.8% of the time!



The following result shows that this is also the case if the black player is a `Randomizer` and the red player is a `PrunefulMinimaxer` (rather than the other way around), with teh `PrunefulMinimaxer` (who is now red) winning 100% of the time!



Furthermore, `PrunefulMinimaxer` also seems to perform better than `AdvancedAggressor`. This is clear from the simulation of 50 games with the black player being the `PrunefulMinimaxer` and the red player being the `AdvancedAggressor`, both with depth 2, and another simulation of 50 games but with the players in the opposite order (again both with depth 2). The results of both simulations are attached below. When `PrunefulMinimaxer` was the black player, he won 94% of the time, and when he was the red player, he won 100% of the time.

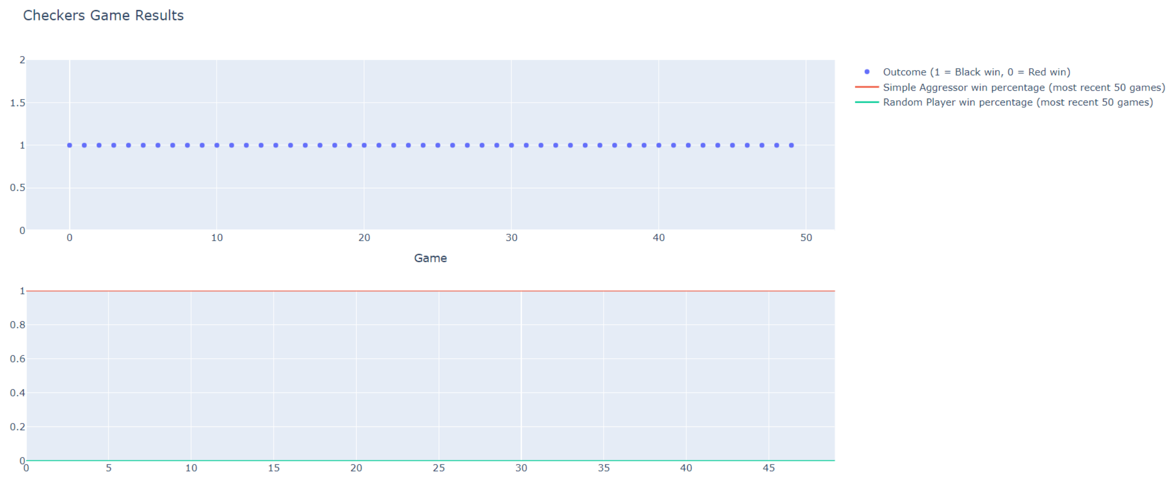


What is truly significant is that these results (that showed the unarguable dominance of the Minimaxer) is that the `PrunefulMinimaxer` operated with only a depth of 2. In a live game, the `PrunefulMinimaxer` (which is the most efficient of all three) can play with a depth of 5 and still be within the time limit making a play. It is difficult to imagine the power of a `PrunefulMinimaxer` operating with depth 5.

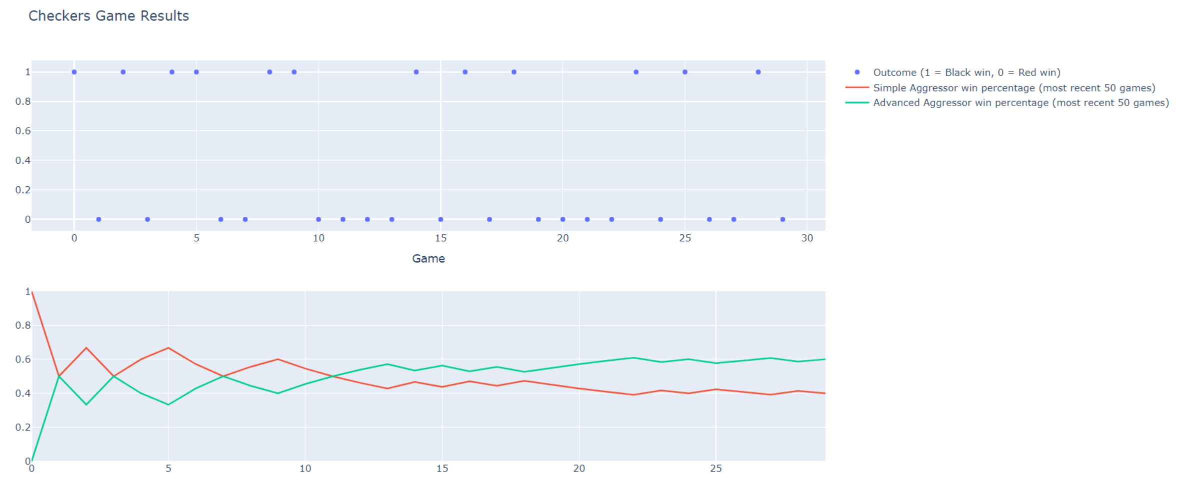
Another crucial note to keep in mind is the dependence of the Minimax algorithm (for all three of the implementations) on our utility function. Choosing another utility function could drastically change the results and the decision-making behavior of our algorithms. Ideally, *the utility (or "static evaluation") function should be a measure of the proximity of a state to a winning state.* Although we tried to implement our current utility function with this consideration, there is lots of room for improvement, and here are some potential further steps. In a future version of the project, we could "improve" (again, we do not know what this word actually means for our context) our utility function by taking into account more information about the state of the game. But this comes with another discussion point; right now, the utility function runs in linear time because for a given states, it calculates the average position of each black piece (to see how close Black is to the other side). Maybe we could change the implementation of the `CheckersGame` class to include such a number (and other quantities we may want to use in the utility function) as an attribute, so accessing it will take constant time, like `len`.

Aggressive Algorithm

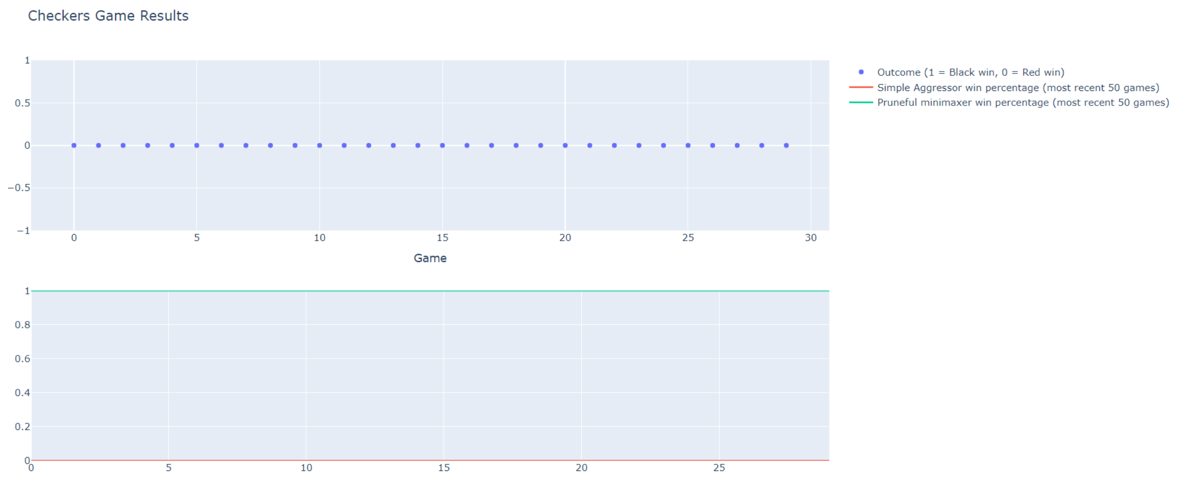
: Making a simulation with the Simple Aggressive player as that black player and a Randomizer as the red player, we immediately see that the Simple Aggressive Algorithm was much better than a Player who made Random moves.



However, when we compared the Simple Aggressive Algorithm with the Advanced Aggressive player, we noticed that the Advanced Counterpart performed much better, even when it looked just two moves ahead. Before optimization of the Advanced Aggressive algorithm, it took us 5 minutes to compute 30 games. After optimization, it took us around a minute. Part of this optimization included recursing down the tree once instead of twice (as initially we recursed both to assign moves to the nodes and then recursed again to assign scores).



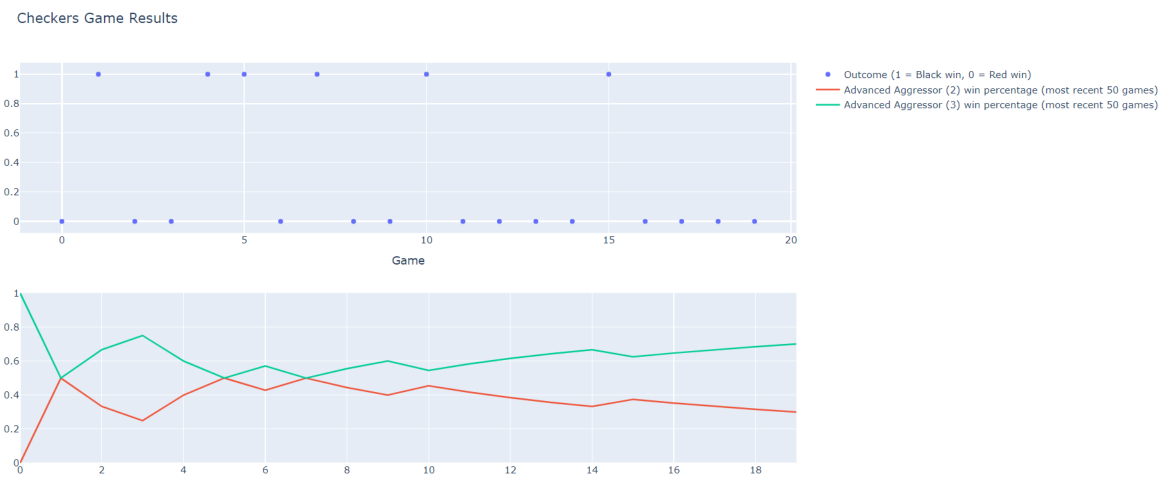
This shows us that the "Aggressive" Strategy worked far better when we looked ahead. Finally, we compared the Simple Aggressor Algorithm against the Minimax Strategy:



The Minimax strategy clearly completely outperformed the Simple Aggressive Strategy, beating it every time. This sheds light on some of the flaws in the Aggression Strategy. Although capturing the most pieces may seem valuable in the short run, in the long run, the most aggressive moves can put us in bad positions in the future, which will lead to us losing the game. On the other hand, the Minimax strategy more holistically assigns scores to moves, by looking ahead and using a more complicated utility function that values winning the game over anything else.

Advanced Aggressive Algorithm

This implementation of the Aggressive Strategy fixed some of the issues with the simpler counterpart. It allows for more long-term thinking, making moves that would capture more pieces down the line, rather than in the short term. We already saw that it outperforms the Simple Aggressive Player, so we decided to compare how two advanced Aggressive Players, one of depth 2, and the other of depth 3 performed against one another.



As shown in the plot above, the Advanced Aggressive Player of depth 3 won 70% of the time against the same Player but with depth 2. This highlights how important the depth of the Player is, as by just increasing this value by 1, we get a significantly better algorithm. During our implementation of this algorithm, it was interesting to see how important optimization was in our computations. Before the computations outlined in the Advanced Aggressive Algorithm description, computing games with depth 3 and 4 were essentially unfeasible. However after these optimizations, we were able to compute many games at these depths in 5 minutes or less.

To conclude, both artificial intelligence tree-based algorithms we implemented, `AdvancedAgressor` and `PrunefulMinimaxer` win against a `Randomizer` almost 100% of the time, with the `PrunefulMinimaxer` seemingly outperforming `AdvancedAgressor`.

References

Haussmann, Aden. “Build an Unbeatable Board Game AI.” Medium, Towards Data Science, 31 Mar. 2021, https://towardsdatascience.com/build-an-unbeatable-board-game-ai-68719308a17.

Srinivasan, Aishwarya. “The First of Its Kind AI Model- Samuel’s Checkers Playing Program.” Medium, IBM Data Science in Practice, 4 Dec. 2020, https://medium.com/ibm-data-ai/the-first-of-its-kind-ai-model-samuels-checkers-playing-program-1b712fa4ab96.

Foster, David. “How to Build Your Own Ai to Play Any Board Game.” Medium, Applied Data Science, 4 Feb. 2021, https://medium.com/applied-data-science/how-to-train-ai-agents-to-play-multiplayer-games-using-self-play-deep-reinforcement-learning-247d0b440717.

Madrigal, Alexis C. “How Checkers Was Solved.” The Atlantic, Atlantic Media Company, 19 July 2017, https://www.theatlanti
tinsley-checkers/534111/.

Pygame front page. Pygame Front Page - pygame v2.4.0 documentation. (n.d.). Retrieved from https://www.pygame.org/docs

“What Is the Minimax Algorithm? - Artificial Intelligence.” YouTube, YouTube, 6 Mar. 2017, https://www.youtube.com/watc
4vw.

Eppes, Marissa. “How a Computerized Shess Opponent ‘Thinks’-the Minimax Algorithm.” Medium, Towards Data Science, 6 Oct. 2019, https medium.comtowards-data-sciencehow-a-chess-playing-computer-thinks-about-its-next-move.

Lazar, Dorian. “Understanding the Minimax Algorithm.” Medium, Towards Data Science, 14 May 2021, https://towardsdatasc
the-minimax-algorithm-726582e4f2c6.