

Checkers Game Project

With Smart AI using Minimax Algorithm



15 August, 2022

IST

CS-02 A



DEPARTMENT OF COMPUTER SCIENCES

OOP LAB PROJECT

**Checkers Game**

**Submitted To:**

Mr. Tufail Shah

**Submitted By:**

Waleed Malik (210201040)

Shahab Ahmed (210201021)

Siraj Ahmad Khan (210201046)

**BATCH:** CS-02 (A)

**Objectives:**

Our project name is going to be Checkers, an advancement made from the previous project we made, which was a Tic Tac Toe game featuring an AI that never lost using minimax algorithm. Checkers is a board game featuring two plays which involve diagonal moves of pieces which are identical for both players and mandatory captures by jumping over opponent pieces. There are a couple of variations to this game, with the project focusing on the American variation which is the most popular.

The program features a fully-function GUI powered with PySide6 and QtDesigner both belonging to The Qt Company with GPL/LGPL licensing. We also use an AI that is powered through an enhanced Minimax algorithm that uses alpha beta pruning. Not including the code for UI (which is mostly generated by QtDesigner, a graphical designing tool that can be used as point and drag), the main code is written in over 1000 lines, all in Object Oriented format and linked together.

1. **Overview:**

This game can be played in a8x8 grid (shown in the figure 2.1) .The game can be played by two players. Players will have the option to play against either a human player or the computer (which will be featuring the minimax algorithm.

Background pattern

Description automatically generated

The starting setup; White moves first  
Figure: 2.1

* 1. Starting Position

Each player starts with 12 men on the dark squares of the three rows closest to that player's side (see diagram). The row closest to each player is called the kings row or crown head. The player with the darker-colored pieces moves first. Then turns alternate.

* 1. Move rules

There are two different ways to move in English draughts:

* Simple move: A simple move consists of moving a piece one square diagonally to an adjacent unoccupied dark square. Uncrowned pieces can move diagonally forward only; kings can move in any diagonal direction.
* Jump: A jump consists of moving a piece that is diagonally adjacent an opponent's piece, to an empty square immediately beyond it in the same direction (thus "jumping over" the opponent's piece front and back). Men can jump diagonally forward only; kings can jump in any diagonal direction. A jumped piece is considered "captured" and removed from the game. Any piece, king, or man can jump a king.

**Jumping is always mandatory**: if a player has the option to jump, they must take it, even if doing so results in disadvantage for the jumping player. For example, a mandated single jump might set up the player such that the opponent has a multi-jump in reply.

Multiple jumps are possible, if after one jump, another piece is immediately eligible to be jumped by the moved piece—even if that jump is in a different diagonal direction. If more than one multi-jump is available, the player can choose which piece to jump with, and which sequence of jumps to make. The sequence chosen is not required to be the one that maximizes the number of jumps in the turn; however, a player must make all available jumps in the sequence chosen.

1.2.1 Kings

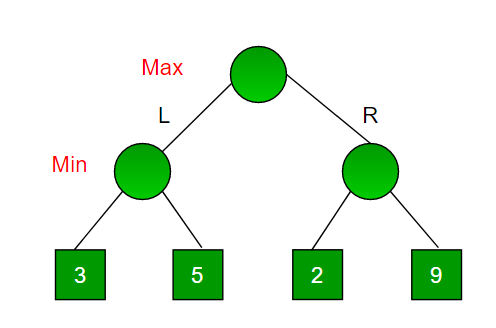
If a man moves into the king’s row on the opponent's side of the board, it is crowned as a king and gains the ability to move both forward and backward. If a man moves into the king’s row or if it jumps into the king’s row, the current move terminates; the piece is crowned as a king but cannot jump back out as in a multi-jump until the next move.

* 1. End of Game

A player wins by capturing all the opponent's pieces or by leaving the opponent with no legal move. The game is a draw if neither side can force a win, or by agreement (one side offering a draw, the other accepting).

1. Minimax Algorithm

Previously demonstrated in the Tic Tac Toe project, minimax is a recursive algorithm very popular in game theory where it considers the current state of the game, then plays every possible move by both players and evaluates how good each move is. It can then select the best possible move for both players and thus choose the one most favorable for the AI.



Creating multiple branches for the AI, each for all the possible moves, then branching out for the moves after it, and when no more possible moves can be made, it gives a score to how good the entire sequence of moves was  
Figure 2.1

Since this is a backtracking-based algorithm, it tries all possible moves, then backtracks and decides.

* Maximizer goes LEFT: It is now the minimizers turn. The minimizer now has a choice between 3 and 5. Being the minimizer it will choose the least among both, that is 3
* Maximizer goes RIGHT: It is now the minimizers turn. The minimizer now has a choice between 2 and 9. He will choose 2 as it is the least among the two values.
  1. Alpha Beta Pruning

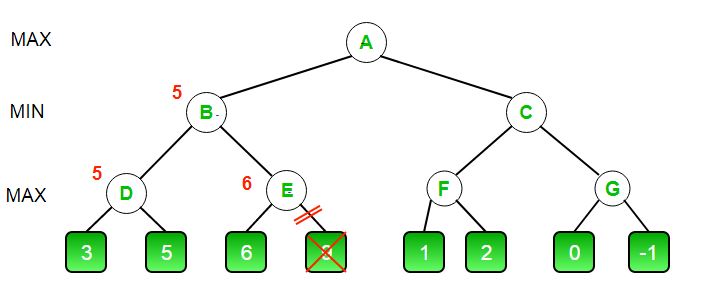
However, the previously mentioned Minimax algorithm runs into performance issues which makes it difficult for us to explore as much depth and as many moves as we would wish to. For this we apply alpha beta pruning, which cuts off certain potential moves when there is already a better one found.

This allows us to search much faster and even go into deeper levels in the game tree. It cuts off branches in the game tree which need not be searched because there already exists a better move available. It is called Alpha-Beta pruning because it passes 2 extra parameters in the minimax function, namely alpha and beta.

Let’s define the parameters alpha and beta.

Alpha is the best value that the maximizer currently can guarantee at that level or above.

Beta is the best value that the minimizer currently can guarantee at that level or above.



Starting from left most end-nodes, during maximization the maximizer can guarantee 5 as the highest value for the node, which the minimizer accepts. If the next tree finds a value greater than 5, then the recursive calls break, and the remaining nodes are ‘pruned’  
Figure 2.1.1

1. Graphical User Interface (GUI)

We have designed to use QtDesigner, an application for graphical designing that uses point and drag widgets to create a desired UI, then generates the code for it in PySide6, which would be in a class-format. We can import this class into our code, more specifically the file main.py from where we run the program. We create another class that will be a child class of the original so we can not only use the UI, but we can also manually connect all it’s functionalities and widgets to the rest of the code (things like buttons and more).

1. Implementation

The implementation involved first off researching the rules of the game and so on.

Then before there was any code written, we created a rough draft on a notebook on the logic of the code and what structure it will follow.

After which we conducted research on similar checkers games written in Python, namely Tech with Tim who also wrote it in an Object-Oriented pattern and included a Minimax algorithm.

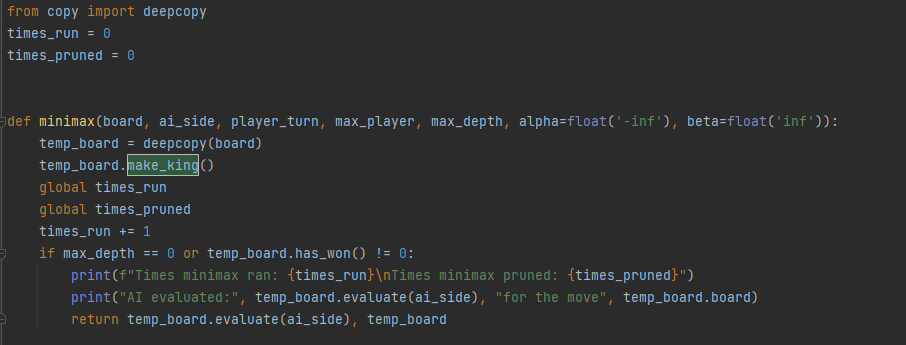
We also found multiple sources for in-depth explanations for Minimax algorithm. The code in itself is not very length or long or technical, however the logic behind it needs to be thoroughly understood. There is always variations in the algorithm as each has to be tailored to the needs of each individual program, although most of the pattern is similar. There is a maximizing function, a minimizing function, an end function where either the game is over or we have searched the maximum depth set custom by ourselves to ensure the computation does not take too long, as each depth adds an exponential factor to the computation time.   
The AI code is contained entirely in ai.py

After a very pain-staking challenge of custom creating a GUI for the first time and connecting it to the program, a lot of the logic behind the gameboard itself was not very difficult after having already made a Tic Tac Toe game. Most of the logic for the board and moves is contained entirely inside game.py, which provides the object Board itself to main.py and possesses multiple functions that return changes to the board to main.py, as game.py itself contains no reference to the UI. All changes to the UI are made after game.py returns the changes to main.py, and in return all player input is provided to game.py using the AI in main.py

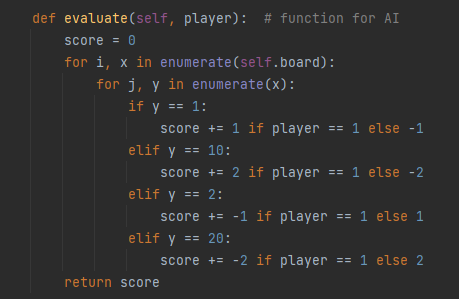
The research and rough notes-drafting took course over several weeks before development started, while the actual development took an average of 8-10 hours over the course of five days, with a few changes and bugs-fixing done afterwards.

1. Code for the AI

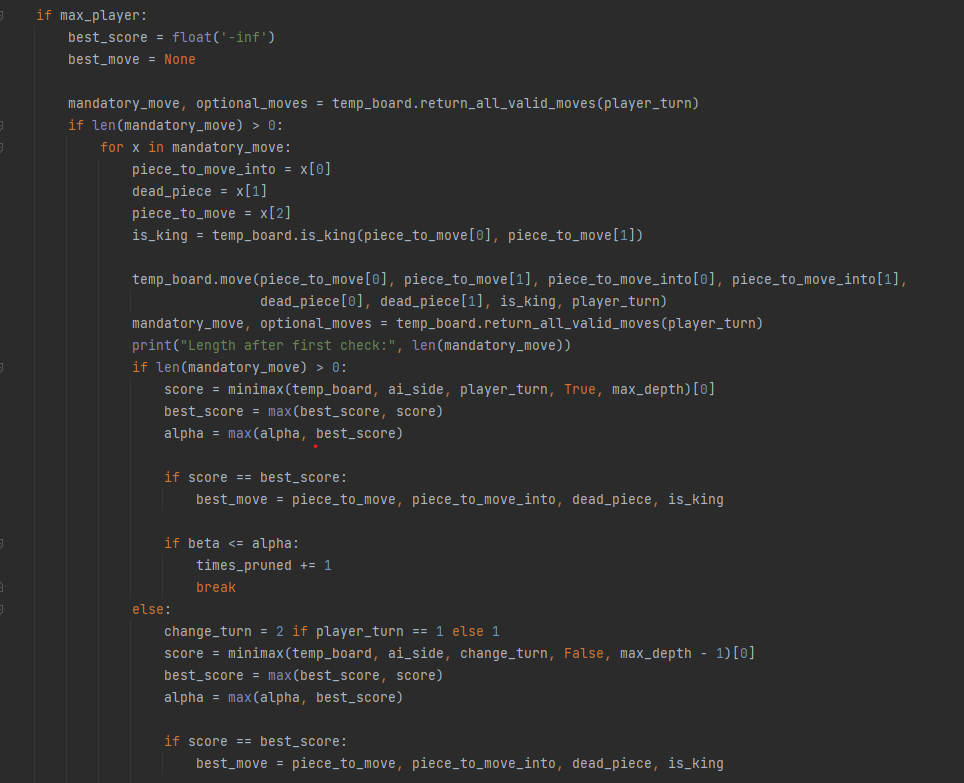
All code for Minimax AI is contained within ai.py, which is imported in main.py



We use deep copy to create a full copy of the object passed onto minimax. The global variables of times run, and times pruned are for debugging/analytical purposes only. At the start of each recursive call, we check if there is any king to create, as the previous calls will not check them at their ends. The first condition which for recursive algorithm is the end condition checks if max depth has been reached or if the game win state is not 0, the zero implying the game is still running, 1 implying white won, 2 implying black won, and 3 implying a draw. When we reach this end condition, we call a function within the Board object called evaluate, that returns a score.



This code is contained within the Board object or game.py, the minimax algorithm simply calls it as it stores a copy of the object  
Figure 5.1



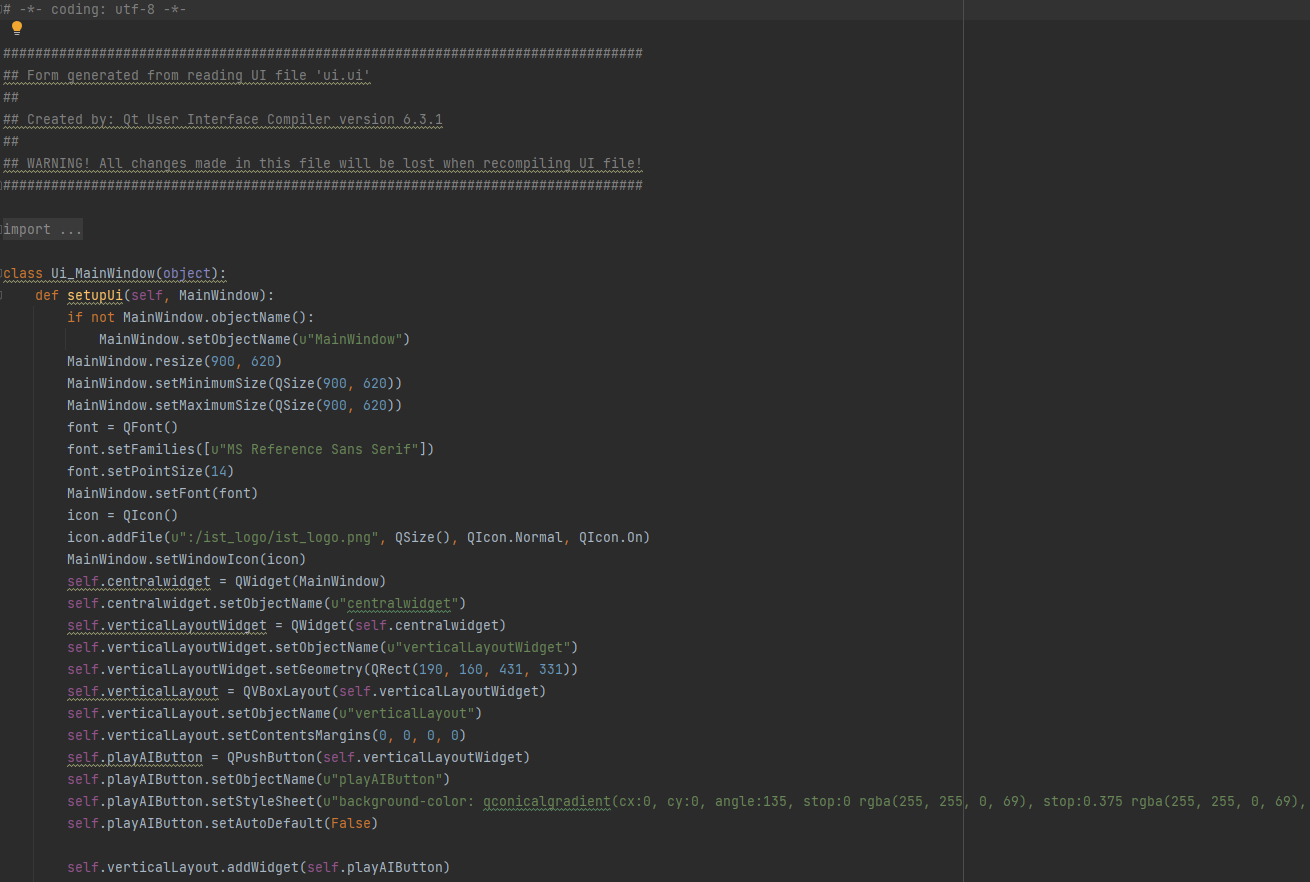
Text

Description automatically generated

In the above screenshots features the code for the maximizing function of the minimax algorithm, which is divided into two parts, the mandatory and the optional moves. In case there is a kill-move available, that move is mandatory, and the AI cannot take any move other than that, except in the case there are multiple kill-moves available, in which case the AI can choose and try each one of them to explore different scenarios. In case there are multiple jumps available, the AI will recall the function again without decreasing its depth to make sure it fully explores it’s turn before increasing the max depth counter again.

1. Sample of the code for UI

Each different UI window such as the main screen, the play screen, and the about screen require a completely different UI design to be made, which is saved in its own .ui file, which can then be compiled into Python code. Each time the UI is edited inside QtDesigner, the whole file is overwritten, that is why every window’s UI file is kept separately and then inherited inside main.py, so our code doesn’t get erased each time we add or remove something in the QtDesigner.



1. Main Program Code

The main.py is the one that starts an application instance and sets up the UI. It also has the main responsibility of using the input from the AI and connecting the rest of the code and producing output.

Text

Description automatically generated

This is how an application instance is created in PySide6, MainWindow() is a class inside main.py whose parent is the class Ui\_MainWindow() shown in the picture within the sample of the code of UI.



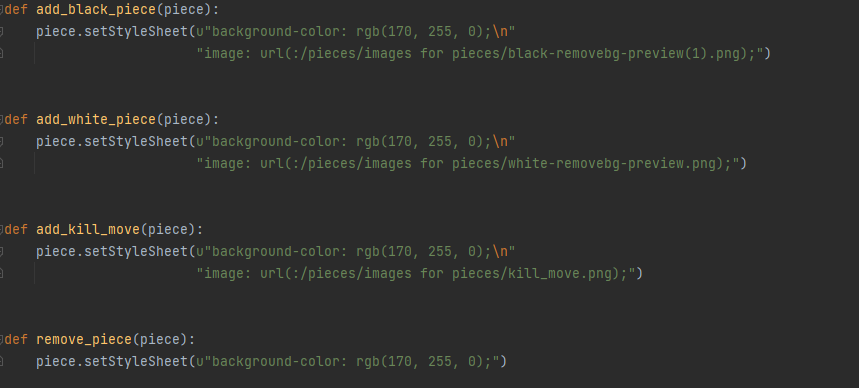
Text

Description automatically generated

Text

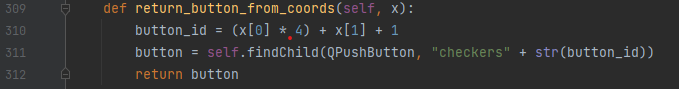
Description automatically generated

The names for buttons are written inside QtDesigner and are variables of the original parent class inside ui.py. The clicked.connect() function calls any function we decide when the buttons that we have written are clicked. We have written QMessageBox which is a widget in PySide6 library that can with very little code create a message box and can return what buttons were selected for the Message box. In some of our options there is a need to request a more detailed message box, for that we create a whole separate UI window and use it for our purposes a message box. You can see this happening above with the side\_dialog = ChooseSide() code, that uses the child class for the UI we created to choose between sides, that features a completely custom UI as our message box. This is entirely different from QMessageBox and is not to be confused with.



Within the images\_rc.py there are images stored. With these functions we can change the design of the board to fit our board and the piece that is supposed to be on it. Upon giving it as its argument the button object, it will change its display immediately.

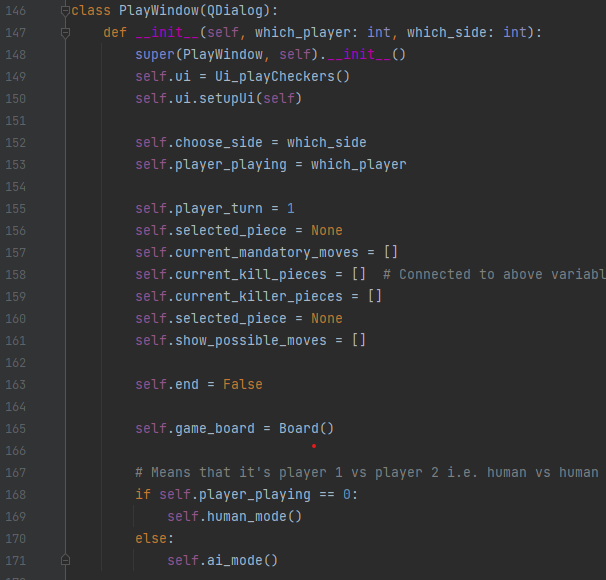
All playable buttons on the board are named in the format “checkersX” where x represents its number on the game board, which in checkers starts from top-left in a row-wise manner, from 1 to 32. We can always find the button object or know its exact coordinates in the gameboard by using these names for our buttons.



Text

Description automatically generated

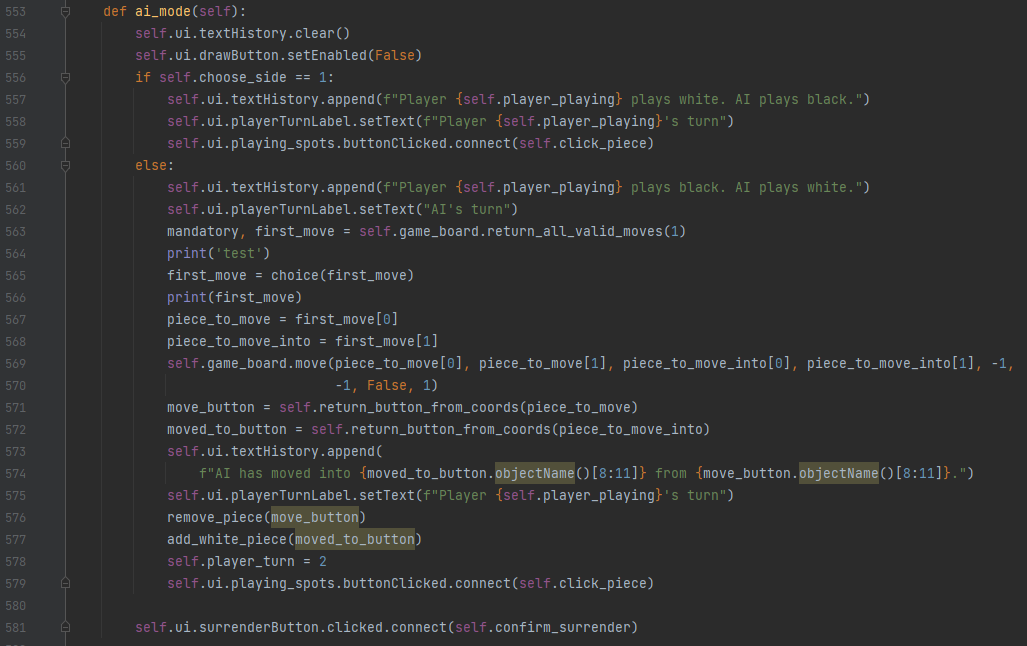
Finally, on to the Play Window itself, when we have actually run a game, the class PlayWindow, child of QDialog which is a widget in PySide6 and references the imported class Ui\_playCheckers() which is the original class containing all UI designs and objects.



This is where we display an already designed UI and make an object self.game\_board of class Board() from game.py, our first reference to game.py which contains all the logic of the game.



Depending on what option our player chose, whether it was Human vs Human mode or AI, we call the appropriate function. All playable spots are grouped in a button group, which when clicked calls the function for click\_piece.

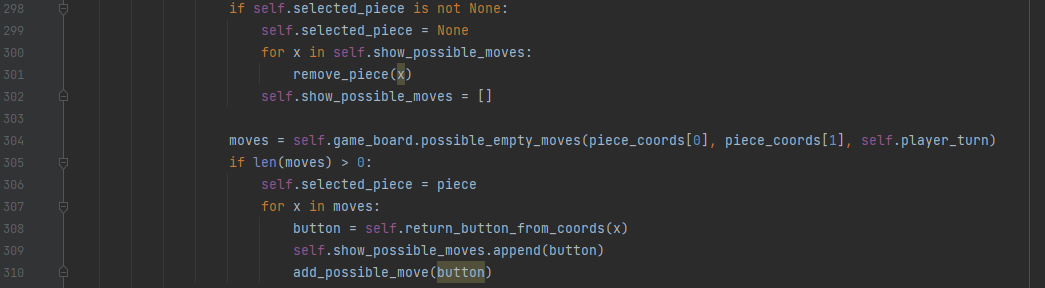


And the above for the AI function.

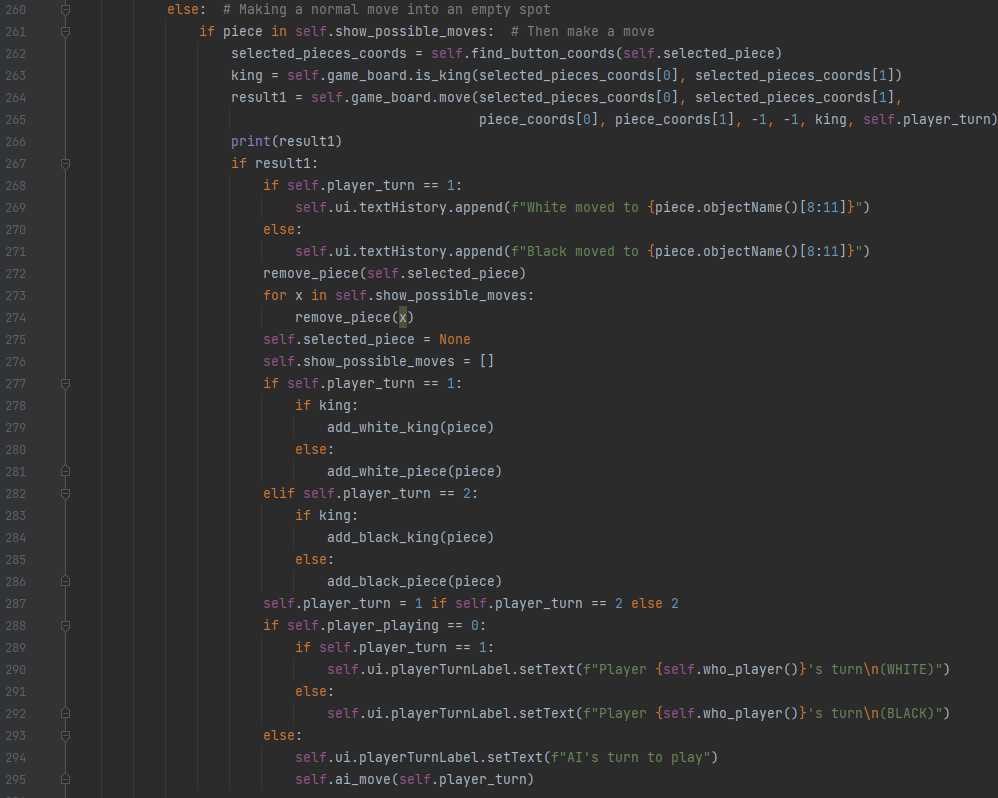
Text

Description automatically generated

This function deals with every time the player clicks a playing spot on the game board. The class inside main.py contains some of the game logic as well, since it is responsible for saving whatever button, the player chooses and display possible options.



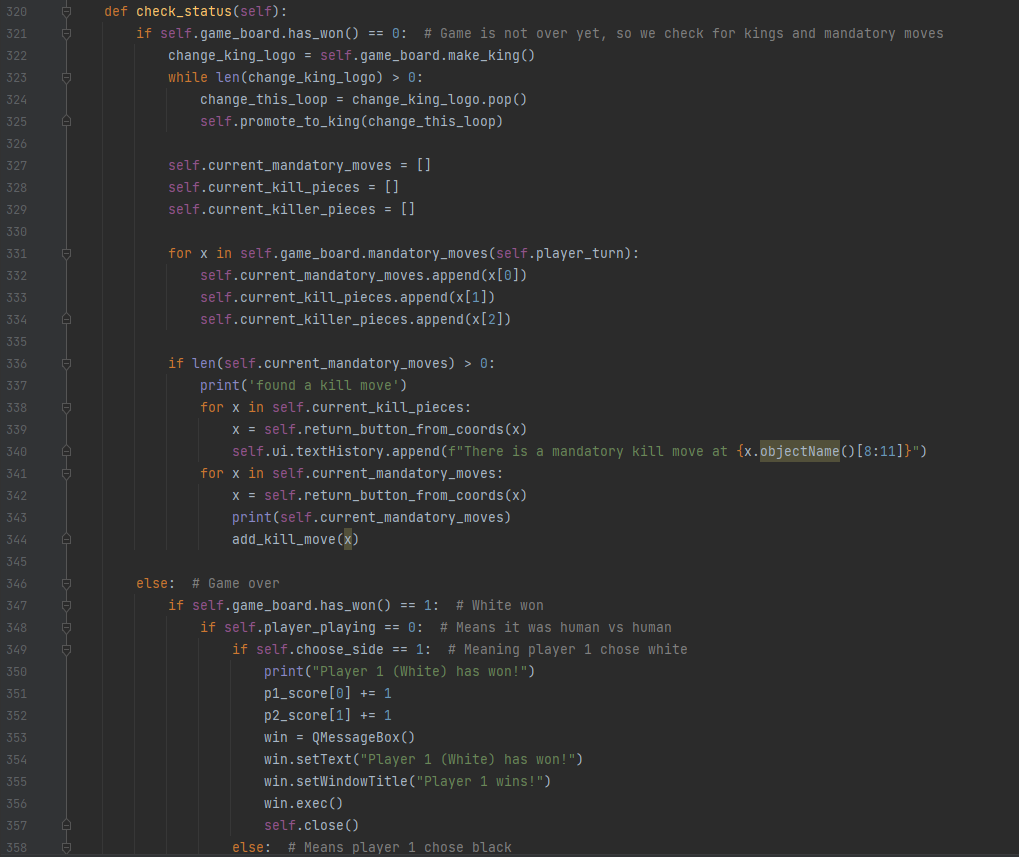
If there isn’t a piece already selected, this code will select it and request possible moves from game.py, then display them on the board. If a piece was already selected but no move was made, then the current selected piece is cancelled, and possible moves are reset.



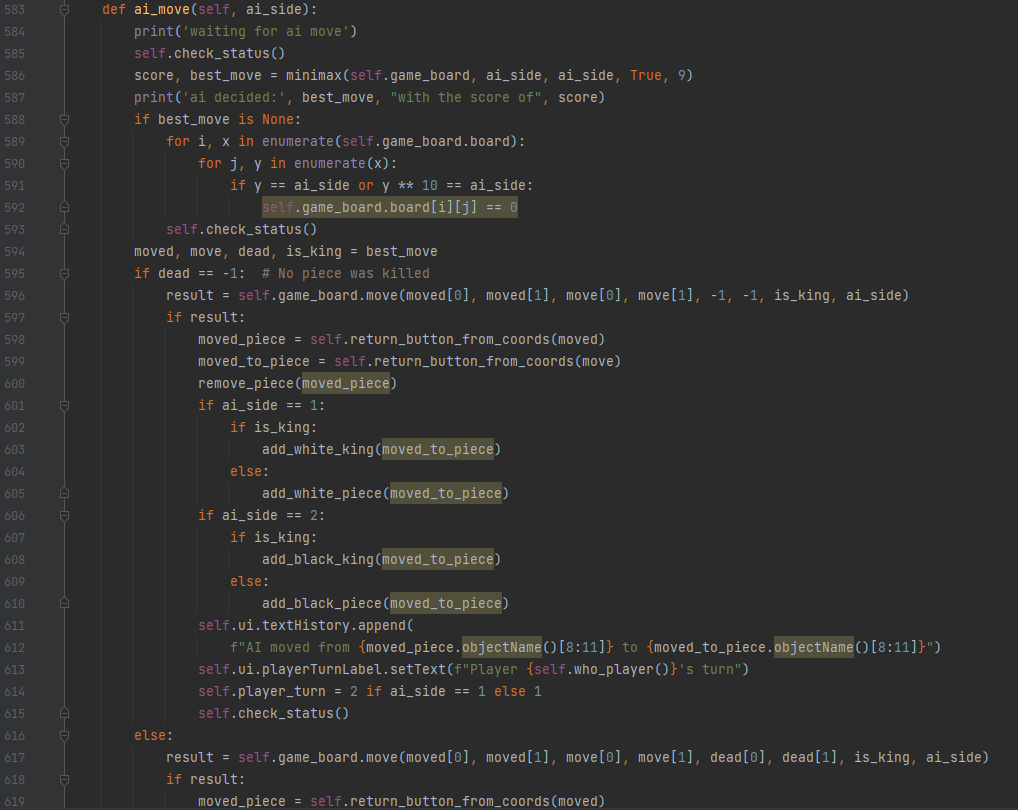
If we have already selected a piece, and the player clicks a button where he can make a move, the above code is responsible for making the move by referencing the code in game.py, and then game.py will return if the move made was successful or not. If it was successful, then we update the UI and the positions in the gameboard. Afterwards the turn is passed on either to the other player, or the AI.

There is another section that deals with mandatory moves that is similar to above code but is slightly more complicated to explain. The short version is that the game needs to check the entire board if the player whose turn it is, can kill any piece, then it is **COMPULSORY** for the player to make that move and we do not allow him to select any other piece or make any other move other than that. This is covered in the rules of the American version of Checkers, along with many other variations of Checkers.

After every move we also must check if any side has won or not, we also must check in the case a piece was killed if the player can kill another piece immediately after allowing for a double jump. We also need to check if the move resulted in any piece being promoted to king and update our game if true.

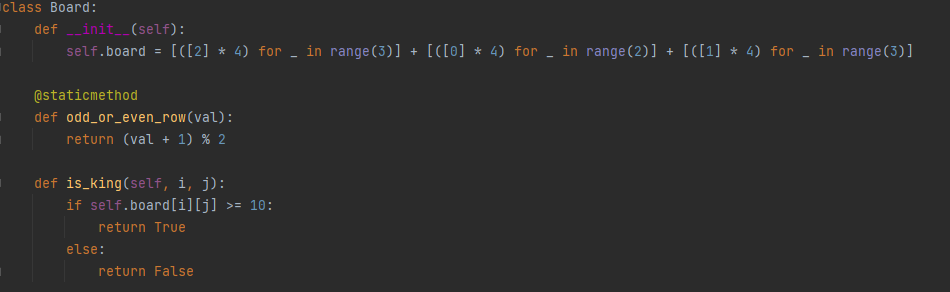


Thus, we run this function after every move to check those conditions and update. It also handles what happens if either side happened to win.



This function will call the minimax algorithm and execute the move suggested by the minimax algorithm. Then if it doesn’t have a double-jump available, it will change the turn back over to the player.

1. The Game Code



This will create a list that represents our game board and all the positions inside it.



There are only 4 playing spots per row. The number two represents a black piece and the number one represents a white piece. Zeros represent an empty spot. If we multiply the value of a position by 10 we create a king, for example 10 would be a white king, and 20 would be a black king.



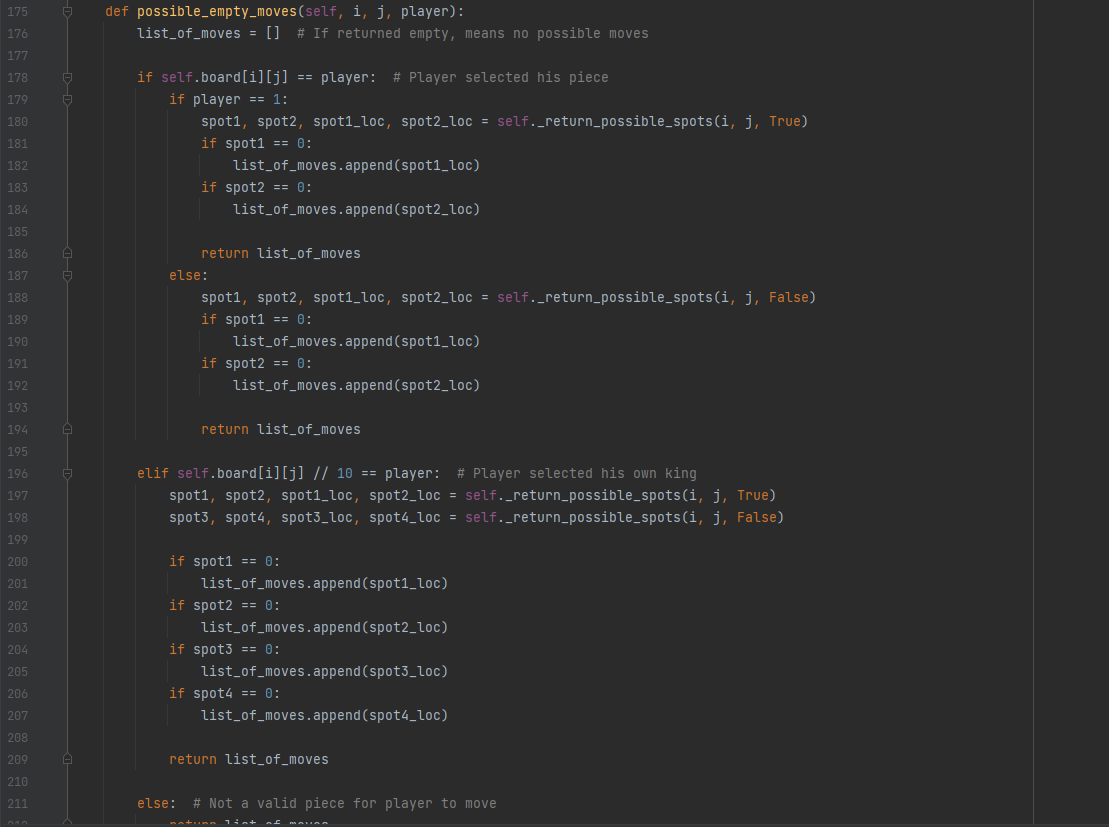
The above code checks if the board is empty of one side’s pieces, which would mean the other side has won. If not, it checks if it possible for both sides to make any sort of move. In some cases, a piece may become stuck, so the function would return 3, representing a draw.

Otherwise, it means the game is still running.

Text

Description automatically generated

The above is a sample of the code relevant for making moves. The UI provides all the arguments, then the function will double check if the move is legal or not, and then if it is, it will make the move, then return True or False for the main.py to know if the move was made or not. Finally main.py will update the gameboard UI.

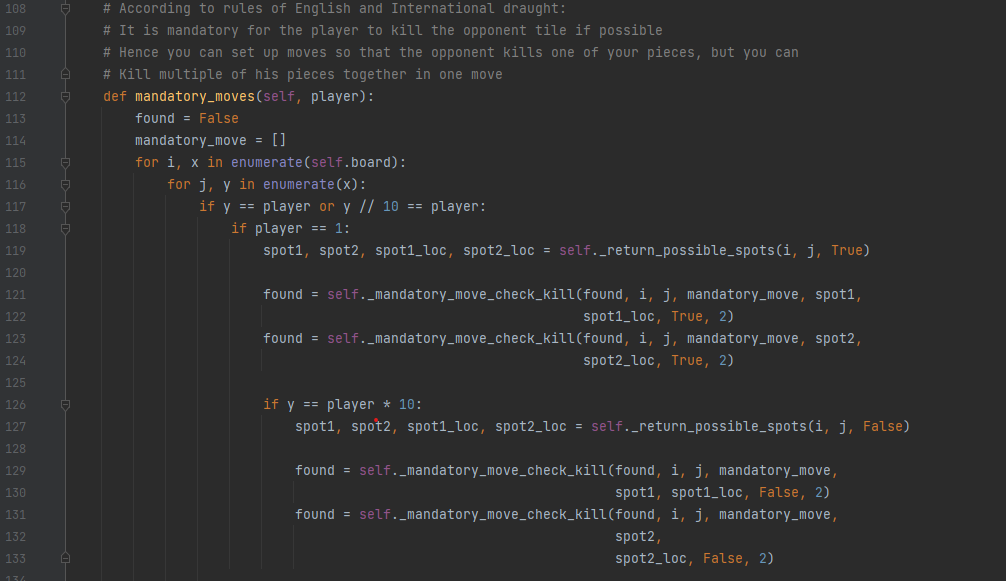


Text

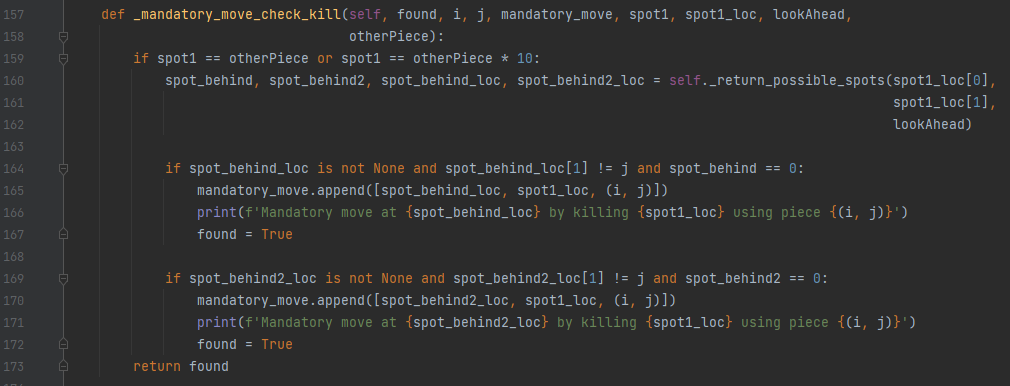
Description automatically generated

Note how the in the second picture the function is declared private/protected. This is because it is a function that performs repetitive tasks, checking what spots are diagonal of any piece, and is used for other functions inside the class and serves no purpose independently. It can check diagonals by simple math, the column number in the above row is always a diagonal, and the second one is usually one column number lesser, or larger, and this can be decided by checking if the row is odd or even.

Optional move uses this function to find the diagonal spots, then check if they are empty or not, then return all those empty spots for the rest of the program.



Here we pass the argument player which represents which side to check (white or black) and then we scan the entire board to check if that side can kill any piece. We do this by considering every spot that has that’s player piece or king, then see if the appropriate diagonals have the opponent’s piece, and finally we check if there is an empty spot directly behind in a zig-zag direction of the enemy piece. We collect a list of all possible moves like this and return. The last of the two checks are repetitive and so we created another private/protected function for that.



1. Future Revisions and Review

After experimentation with the code, we have set the maximum depth for Minimax algorithm to search to 9, although in the case of multiple-jump-kills each jump does not add to the depth otherwise the AI would often perform very poor in situations where such double-kills can happen whether itself is performing them or the opponent. As such these double kills not only impact the performance of the program heavily but can run into bugs where the AI cannot successfully conduct all those possible moves and get stuck, although effort is made to minimize the chances of this happening.

Even despite having alpha-beta pruning and the maximum depth of 9, the program sometimes takes several second to process a move on a decent processor. Through debugging we have found that for simple moves the AI can search up to 300 possible scenarios, and for more complicated moves where there can be several kills happening, this can go up to tens of thousands of scenarios per move.

In debugging we also found that the program successfully prunes half or more than half of these scenarios, allowing us to even be able to reach the depth of 9. However, as the minimax algorithm is sometimes called up to 10000 times, even forgoing all our previous efforts, Python itself takes a long time for even simple loops performed that many times as Minimax algorithm does. Thus, we still run into performance issues.

A possible solution to this might be using other libraries that may allow us to conduct those recursive calls faster, which needs more research and knowledge. Perhaps multi-processing libraries can also help with this. However, the most optimal way to solve this would be using C++ as Python will always be limited when it comes to performance. By writing the code for Minimax algorithm and integrating that C++ file into our Python project, we can achieve far greater speeds and deeper searches.

This however also requires not only additional knowledge and mastery of C++ but also needs research on how to use code written in C++ as libraries in Python.

There is also an option in the main menu called Recent Games, which was initially designed so that the original object Board would be saved every time a game would finish with the history of the game written along with it, and recent games would allow us to go text/binary files containing those objects and allow us to not only show results of the previous games but see every move made, and also allow the Minimax algorithm to analyze the best chance each side has to win for each move. This is a complicated feature that may get added in the future.

Finally, there is the fact that often AI makes moves immediately after our moves, we often cannot see our own move made until the AI has already processed and made its own move. It can be difficult to track what move the AI is making. There is a need for the program to slow down and show the piece being dragged into its new position, however both parts are complicated and difficult and involves complete mastery of the more advanced libraries and functions of PySide6 and Python itself.