### **Introduction**

The CheckerBoard program is written in Python language. I’m more familiar with the Python language and it is easier to implement the logic of the minimax algorithm with alpha beta pruning using this language. I use the *pygame* library to draw the GUI. There are 4 classes defined which are Game class, Board class , Piece class and AI bot class. I also have a main program where I call the Game object and update the event for each game from the main. Plus, I created a constants file to store the configuration of the program there.

Game class is where I implemented the rules of the game and controlled the game state. On the other hand, Board class is where I draw the GUI and update the GUI for every move user and AI made. Pieces class represents each piece on the board. In the class I can control the color state and the king state. Finally the AI bot class is where the minimax and alpha beta pruning algorithm is implemented.

In the main class, I instantiated Game and AI objects. Then, the program will loop through until the game ends. Mouse interaction is captured using the Pygame event method. It will captured any mouse event and from the event I handle the movement of the pieces.

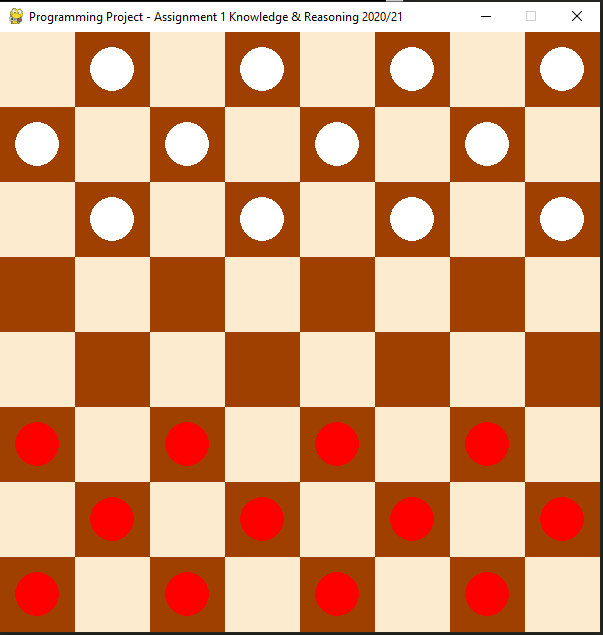


Figure x shows the Graphical User Interface for a start game.

### **Description of Program Functionality**

2. Search algorithm **(20 marks)**

a. Appropriate and efficient state representation

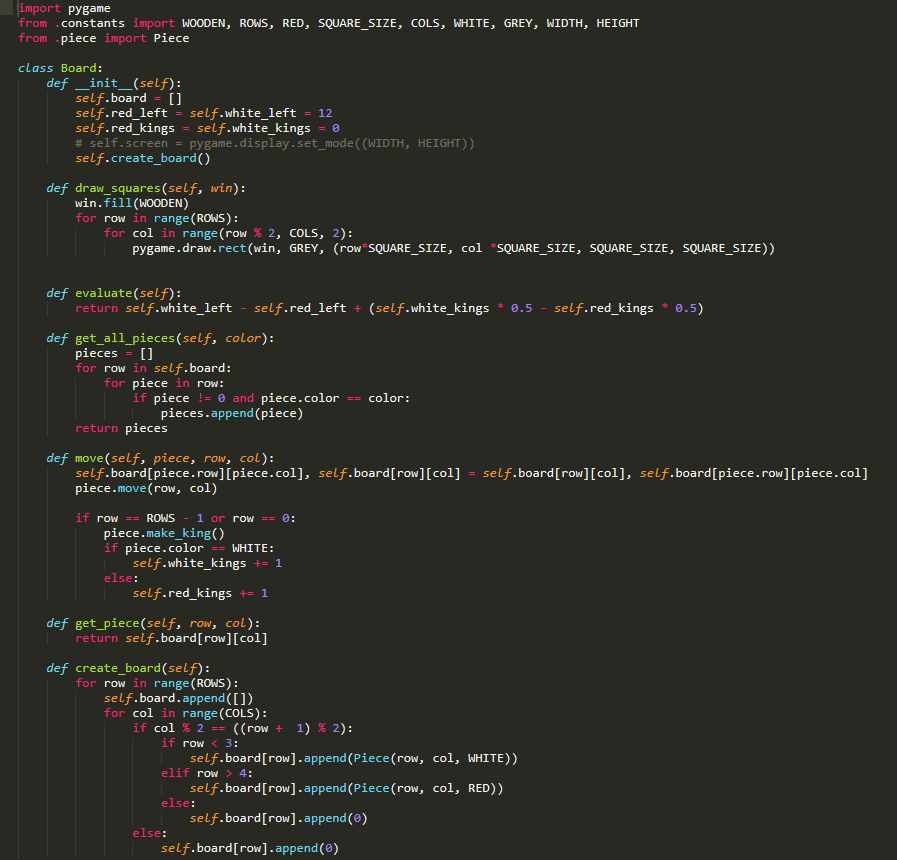


Figure 1. shows the snippet code for Board class.

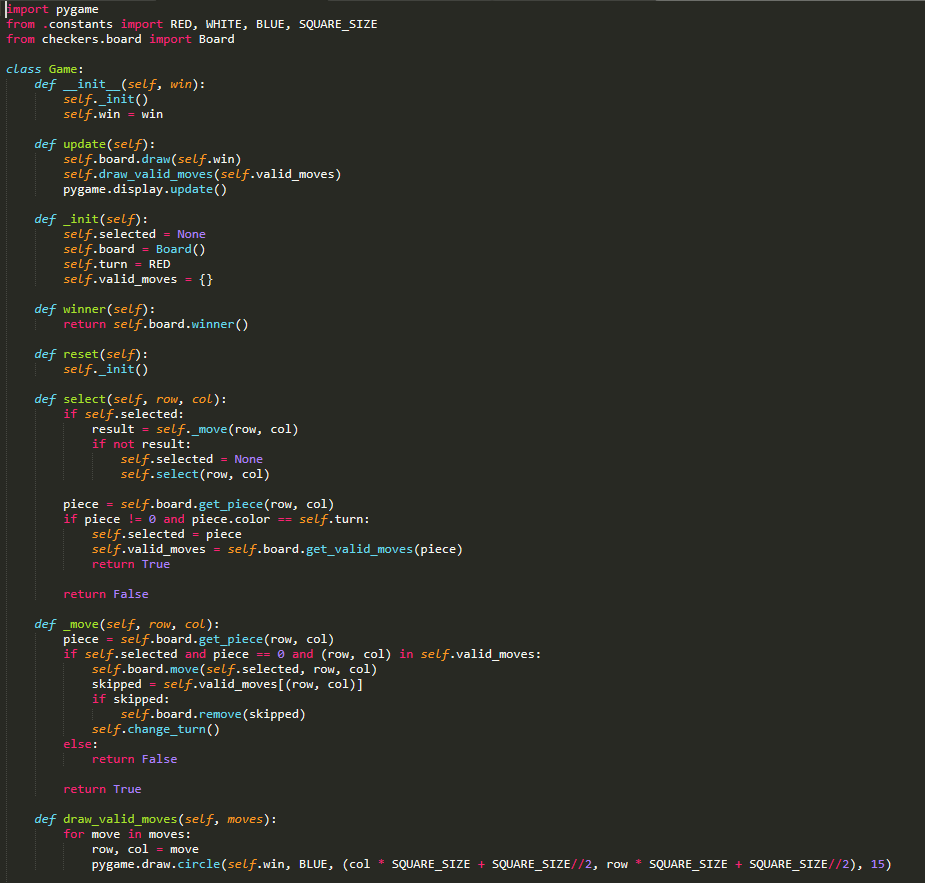


Figure 2 shows snippet of game state.

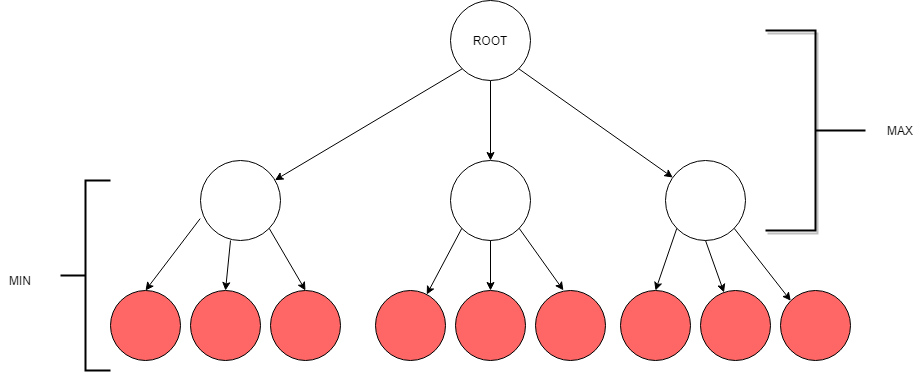
There are 5 classes defined. Each class has its own state. For example, board class has a number of red ,white left states and king for each color state. On the other hand, Game class is where Checkerboard rules are implemented. In this class where I control the turn state, winner state and valid moves state.

**b. Reasonable successor function to generate AI moves**

To generate nodes, I use a recursive function to get min and max value for each node. At each traverse of node, I implemented a routine to compare and update the alpha and beta value.

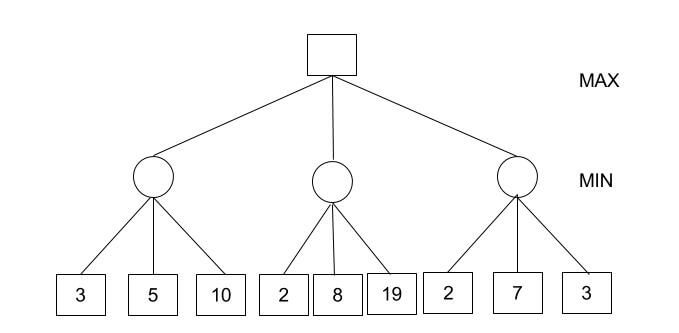
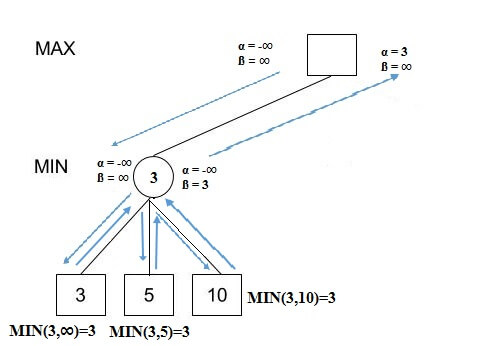
**c. Minimax evaluation**

The algorithm is set up to 5 tree depth.



This is a simple 2 tree depth visualization of the minimax algorithm. On the first layer (root), it represents a whitle piece. There are 3 possible moves the white piece can make. Hence, it created a second layer of 3 white nodes. For each white move, the opponent(user) has 3 possible moves as well. Hence, on the third layer, each white node has 3 red nodes which represent possible moves from the user.

**d. Alpha-Beta pruning**

1. Initialize alpha = -infinity and beta = infinity as the worst possible cases. The condition to prune a node is when alpha becomes greater than or equal to beta.
2. Start with assigning the initial values of alpha and beta to root and since alpha is less than beta we don’t prune it.
3. Carry these values of alpha and beta to the child node on the left. And now from the utility value of the terminal state, we will update the values of alpha and be, so we don’t have to update the value of beta. Again, we don’t prune because the condition remains the same. Similarly, the third child node also. And then backtracking to the root we set alpha=3 because that is the minimum value that alpha can have.  
   
4. Now, alpha=3 and beta=infinity at the root. So, we don’t prune. Carrying this to the center node, and calculating MIN{2, infinity}, we get alpha=3 and beta=2.
5. Prune the second and third child nodes because alpha is now greater than beta.
6. Alpha at the root remains 3 because it is greater than 2. Carrying this to the rightmost child node, evaluate MIN{infinity,2}=2. Update beta to 2 and alpha remains 3.
7. Prune the second and third child nodes because alpha is now greater than beta.
8. Hence, we get 3, 2, 2 at the left, center, and right MIN nodes, respectively. And calculating MAX{3,2,2}, we get 3. Therefore, without even looking at four leaves we could correctly find the minimax decision.

e. **Appropriate use of heuristics**

The heuristic function I implemented is to reduce red pieces score and increase the white pieces score. In other words, MAX function is used on the white nodes and MIN function is applied on red nodes.

3. Validation of moves **(16 marks)**

a. No invalid moves carried out by the AI /

The AI moves according as stated from the rules. The AI should capture if there any move that can trigger

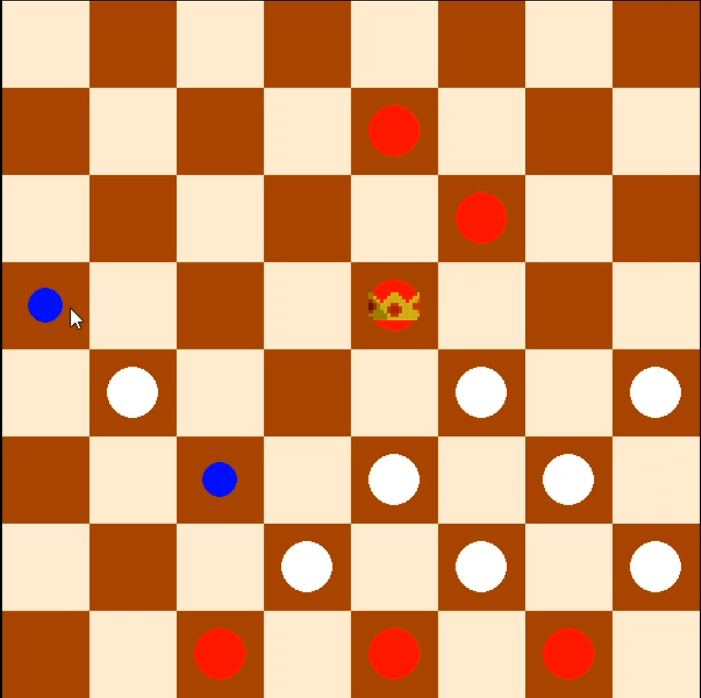
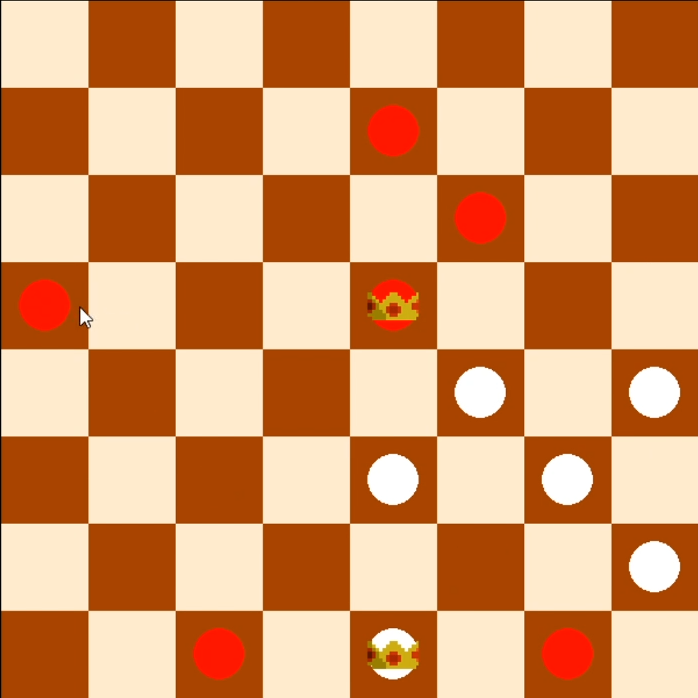
b. Automatic check for valid user moves /

c. Rejection of invalid user moves, with a specific explanation given

d. Forced capture - an opportunity to capture an enemy piece has to be taken. If there is more than one capturing opportunity in the same turn, the user may choose which one to take.

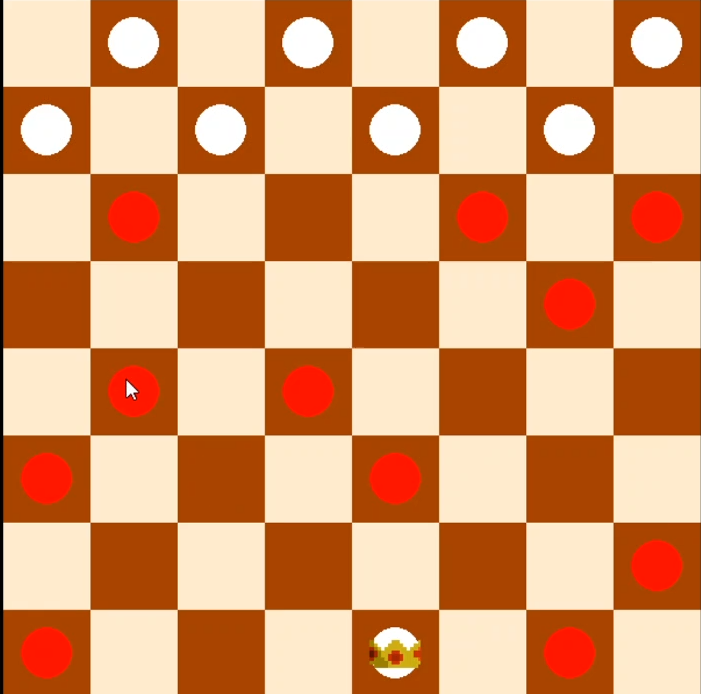
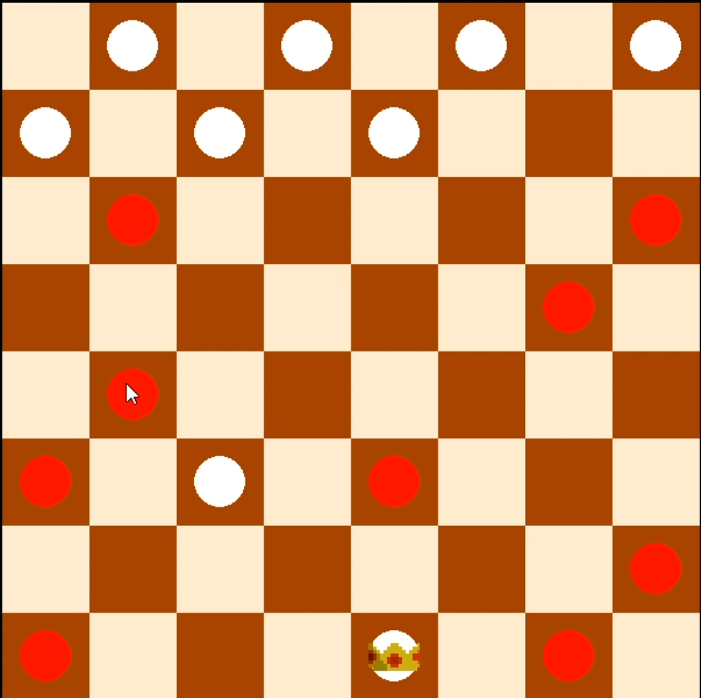
4. Other features

a. Multi-leg capturing moves for the user (RED)

 → 

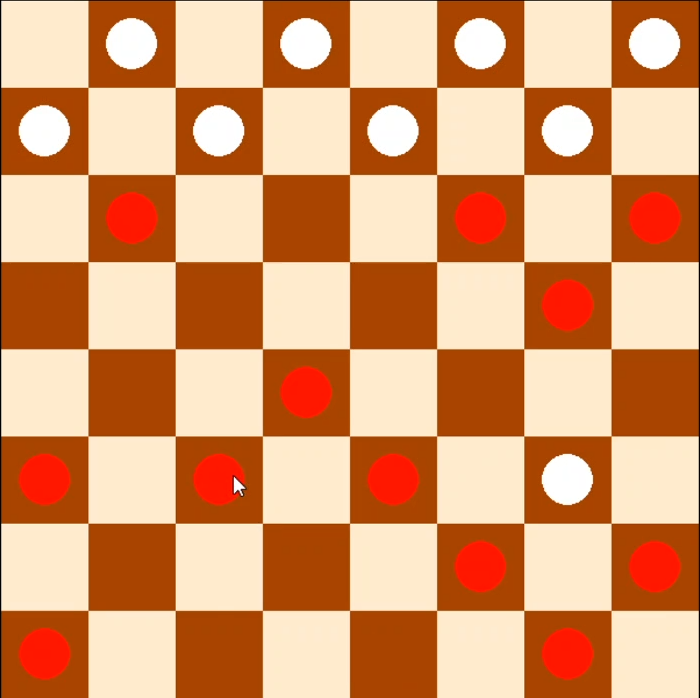
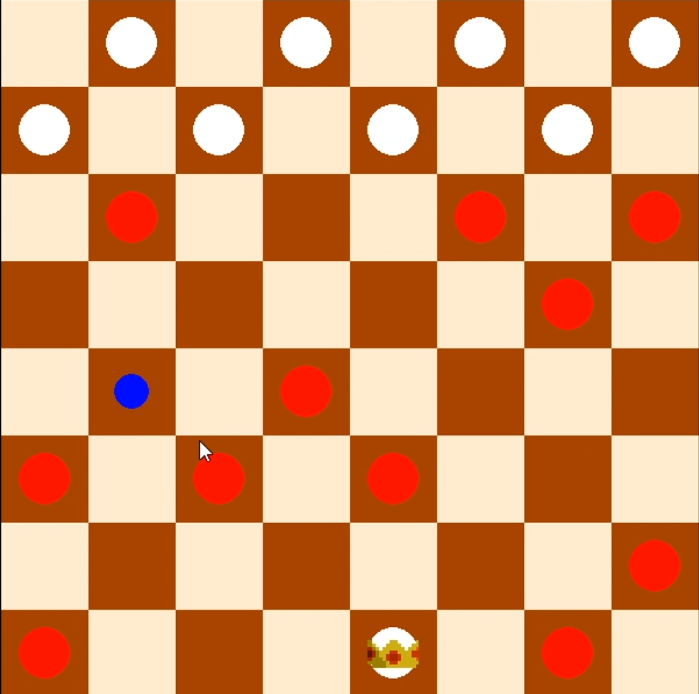
From the above figures, the user can attack 2 White pieces in one move.

b. Multi-leg capturing moves for the AI (WHITE)

→ 

From the image above, the AI bot (white) can attack 2 pieces of red in a turn. On the other hand, this rules can also be used by human (user).

c. King conversion at baseline (The king’s row) as per the normal rules

 → 

From the above figures, White pieces can be converted into King as it reach the end tiles for red region. This rules follows for the Red King conversion as well.