1. **How does your program calculate the utility of each terminal/non-terminal state? Describe briefly.**

   The program calculates the utility of each terminal/non-terminal state by calling the *evaluate_state()* function.

   The function returns an integer which represents the score of the current state and follows the formula $\sum \text{red pieces} - \sum \text{black pieces}$. Regardless of colour, King pieces have a value of 5 and normal pieces have a value of $3 +$ *evaluate_piece()*. The function *evaluate_piece()* considers gives a weighted score to a piece based on how advanced they are following the formula $28 - (\text{row} \times 4)$ for red pieces and $\text{row} \times 4$ for black pieces. This is because a piece that is more advanced on the board for its respective player would bring more utility than a piece closer to their starting position. Red pieces can only move up and turns into a King when they reach row 0 and black pieces can only move down and turns into a King when they reach row 7.

   Finally, we considering the shielding effect from diagonally neighbouring ally pieces and add 2 for red pieces subtracting 2 for black pieces. This is because diagonally neighbouring ally pieces prevents opponent pieces from capturing said piece via jumping.

2. **Does your program perform other optimizations, such as node ordering or state caching? If so, describe each optimization in a few sentences.**

   - Node ordering takes place in alpha-beta pruning when functions *max_value()* and *min_value()* are called for the maximizing player (red player) and minimizing player (black player) respectively. This is done by sorting the list of successor states via our *evaluate_state()* function. For the maximizing player, the list is sorted in descending order based on utility and for the minimizing player, the list is sorted in ascending order. This allows for pruning to be expedited. Once the criteria for pruning is met, we break out of the for loop and this minimizes the number of successor states we need to visit.
   - State caching is also built into the program. Best successor states will be cached in a global hashmap. This is done by setting the string representation of the state as key and its utility value as the value. This prevents calculating utility values and recursively exploring a state that has already been explored and stored in our cache and speeds up our algorithm's run time.
   - When our program searches for all possible moves, jump moves are prioritized. This happens in the *get_all_moves()* and *get_all_succcessors()* function. If jump moves exists, these 2 functions would only consider the jump moves and ignore all simple moves even if some exist. This prevents unnecessary computational resources devoted to finding simple moves that would not be executed due to the rules of Checkers.