**Type of Searching Used**

The engine uses a minimax algorithm to determine the best move to make to increase its odds of winning against the other player. Minimax is a backtracking algorithm that is used in decision-making games to find the best move for a player to make. The algorithm has two players (maximizing Player and non-maximising Player), which compete against each other to win the game. The maximizing Player attempts to get the highest score possible while the non-maximizing Player does the opposite and tries to get the lowest score possible.

The minimax algorithm goes to certain depth and analyses each board state at that depth which is given an evaluation score by the evaluation function. The minimax algorithm is then applied to find the best move for the player to make so that it’s chance of winning increase.

**how your evaluation function has appropriate values for the win, loss, and draw results**

The evaluation function works by giving an evaluation score for each board state which is a score of which player is the closet to winning the game. The evaluation first checks to see if any player has won the game. This is done through the ‘checkIfWon’ function which returns an integer value based on the player that has won the game. A value of ‘1’ is returned if the non- maximizing Player has won, ‘0’ if the maximizing Player has won and ‘-1’ if no one has won the game.

The evaluation function then checks the value returned from the ‘checkIfWon’ function. So if the maximizing player has won the game, a score of 1000 is returned and a score of -1000 is returned if non-maximizing Player has won. If no player has won the game then the function analyses the state of the board. Note, no points are awarded to a player if it has only has one piece by itself.

The function first checks the columns to see if any player is close to winning the game. Every combination of achieving 4 in a row is checked and added to an integer value called ‘value’. The combinations are checked with the help of a ‘checkMoves’ function which takes in an array of size 4. The function checks how many pieces of maximizing Player and non-maximizing Player are in the array. A value is returned based on the number of pieces a players has and if it is possible to achieve 4 in a row with that combination.

If a column is full then that column is not checked as there is no chance of achieving four in a row for any player. The value returned from each combination is multiplied by a value in the ‘preference’ array that associates a value with each column. This is primarily done so that it pushes the player to go towards the centre of the board as it provides the best chance of getting 4 in a row then say if it were to place a piece on the ends of the board.

Similar to the column checking evaluation, the function checks each combination of getting 4 in a row for each row. If the row is full then there is no point in checking that row as no player can get 4 in a row in that row. The function then checks both the positive slope and negative slope diagonals on the board. Only diagonals where it is possible to get 4 in a row is checked as it is the only place where a player can get 4 in a row and win the game. Each diagonal has a ‘startpos’ array which is used to help the function know where the diagonal starts. Like the horizontal and vertical evaluations, each combination is checked in the diagonal.

After all combinations in the board is checked and added to the integer ‘value’, it is returned.

**Discontinue of an Internal Nodes**

At each depth, the engine check to see if any player has won the game with the help of the ‘checkIfWon’ function. If a player has won the game, then no more children are generated after that moment as there is no point in doing so. If the maximizing player has won the game (a value of 0 is returned), a score of 1000 is returned and a score of -1000 is returned if non-maximizing Player has won (a value of 1 is returned). This is also applied in the perft function but instead, it returns a value of 1.

**Improvements to Evaluation Function**

As stated by the lecturer, there is no such thing as a perfect evaluation function as one simple evaluation function can beat a complex evaluation function if it performs an unexpected move. In saying that, I would still make some further improvements to the evaluation function so that it performs better. A change I looked into was defensive moves that a player can make so that the other player cannot win in their next move. Due to the complexity of doing this, I never implemented it to the engine but it would have been very beneficial if it was as it would prevent a player from achieving easy wins.

**Conclusion**

In conclusion, the evaluation function and minimax algorithm operates as designed. The engine was tested against a random move generator was capable of beating it. Even though the engine works as intended, there are places where improvements can be made and areas in the code that can be optimised to perform faster.