**CSC384 Assignment 2 Design: Checkers**

**Calculating Utility of Terminal States**

The utility of a terminal state is either a very large positive number for the winning player or a very large negative number for the losing player, and this function only runs after verifying that the board is at a terminal state with an is\_terminal helper function. This is calculated by iterating through every character in the board and counting the number of pieces present for each player. If the turn is red (and red is the maximum player), for example, and there are no black pieces (including kings) present, the function returns +1000000/depth since the red player has won. This operates similarly for the black player. The depth is a parameter to differentiate between a winning path that is 2 moves long versus 4 moves long, for example, because the quicker the win for the red, the better. That is, the lower the depth, the better the move for the red player.

Furthermore, if neither condition of having an entire player wiped out is met, the function checks if the player of the turn is equal to the maximizing player of the alpha beta iteration. If it is, since it knows it is at a terminal state, then the player must not have any moves left; thus, the utility is returned -1000000/depth. Conversely, if it’s not the maximizing player’s turn at a terminal state (and there are still pieces from both players on the board since the previous if statements did not run), then the player is at a winning state.

**Estimating Utility of Non-Terminal States: Evaluation Function**

For non-terminal states, an evaluation function was used to estimate a player’s utility. The search for terminal states stops at a specified depth limit, which was 7 in this case. Once the depth limit was met, the evaluation was called if it was a non-terminal state.

The evaluation function for a given player was calculated by iterating through every piece on the board. For each player, a regular piece counted as 1 point, while a king counted as 2.5 points. The final utility estimate for the player was found by subtracting the opponent’s point total from the given player’s point total for that state.

**Additional Optimizations**

Node ordering of the successor states was implemented in both the min\_value and max\_value helper functions for the alpha beta search. This was done using the built-in “.sort” functions, and the states were sorted according to their estimated utility value (using the evaluation function). In max\_value, successors were sorted in order of descending utility because the maximum player wants to maximize its utility, and vice versa for the min\_value function.

Caching was implemented in alpha beta pruning to store the utility values associated with each state. Each state was converted to a string to store in the cache. In the max and min\_value functions, if a terminal state was discovered, the code will first check if the state is already in the cache; if so, then return the utility value associated with that state. Otherwise, use the utility function on that state and store them both in the cache.