CSC384 – Assignment 2 Document

Ethan Rajah

My program calculates the utility of each terminal state by taking a state and current turn as inputs to a function called ‘compute_utility’ and returns a large number if the player representing the current turn won the game, and a very small number if the player representing the current turn lost the game. Both 100000 and -100000 were arbitrarily chosen as the numbers to use for identifying the utility of the terminal state as the evaluations of non-terminal states would never reach that high of a value. The utility is then set to the satellite value dedicated to storing the evaluation of the state so that it can easily be referenced later.

To calculate the utility of non-terminal states, I created a function called, ‘eval_fnc’ which takes both the state and turn as inputs and returns a heuristic estimate of what the true utility is for that state. The evaluation function is based on taking the difference between the number and type of pieces each player has, as well as the difference between the number of pieces each player has at the edges of the board, as those are safe from capture pieces. The function iterates through the state board and if it encounters a red piece, it adds to the evaluation, whereas if it encounters a black piece, it subtracts from the evaluation. Furthermore, regular pieces have a value of 1 and king pieces have a value of 2. When the function identifies that a piece is an edge of the board, it either adds or subtracts a value of 1 from the evaluation, depending on if the piece is red or black. I originally had a more advanced evaluation function, where I included an extra evaluation to account for the fact that it is better to have your pieces further advanced along the board, however, after testing, I found that this made my program worse in efficiency and optimality by a fairly large margin, so I decided to omit it from the final submission. Overall, I found that the final evaluation function with a depth limit of 10 best suited my AI.

To optimize the AI for efficiency, I implemented both node ordering and state caching into my program. To implement node ordering, I added a few lines of code in my successor generating function, where the successors would be sorted either in decreasing order or increasing order based on their evaluations, depending on if we were at a max node or a min node in minimax. This optimization increases the amount of pruning that occurs within the algorithm, thus decreasing runtimes of the program. To implement state caching, I created a dictionary that was passed through the alpha-beta pruning algorithm. When we reach a terminal state or depth limit, the AI checks if the state is already within the cache dictionary, but if it is not, the evaluation of the state is calculated and placed within the cache, with the key being the state object. Thus, if the state is already in the cache, we can avoid redoing its evaluation calculation, in turn improving the efficiency of the AI.