Matthew Daniel

CS 470

March 10, 2017

Project 2 Write-Up

**Abstract:**

I wrote a program that allows a user to play Connect Four against another user or against a computer player. During its turn, the AI selects a move via the negamax algorithm. The negamax algorithm employs A/B pruning, move ordering, and concurrent searching of the depth one children. The algorithm looks 10 moves ahead if it is playing as red (the player who moves first), or depth 9 if it is playing as blue. The program recognizes all tested win and draw states, and always selects the best move based upon its heuristic.

**Algorithm:**

1. I utilized the negamax variant of minmax search. This enabled me to ignore the minimum/maximum choice based upon the passed depth, and instead multiply the color by -1 at each depth. I used a built-in timing function to determine what depth I could search to. After experimentation, I found that a depth of 10 moves kept the search time under 10 seconds when playing as red, but a depth of 9 was needed when playing as blue to keep search times from exceeding the limit.
2. I added the following additional features to my negamax algorithm:
   1. A/B pruning. I implemented the A/B pruning pseudocode found in the Wikipedia entry. The algorithm implements pruning from the children of the children of the current game state. The reason for this is discussed in point c.
   2. Move ordering. The algorithm processes the moves from the center of the board outward. This is because the pieces in the center of the board are involved in more winning scenarios, and are likely to receive higher evaluation scores.
   3. Concurrent search. The first children of the current game state run their searches asynchronously, taking advantage of the 8 virtual cores on my machine. This may slightly decrease the effectiveness of A/B pruning, but greatly increases the speed of the search in general.
3. The evaluation function works by examining only the newest inserted piece and its connections. It searches in n/s, e/w, ne/sw, and nw/se pairs, assigning a value of 10^(number of pieces in a row – 1) for 1 to 3 pieces in a row, assuming that there is enough empty spaces to complete a win. If the algorithm finds a winning state, the board is evaluated to Int.max, or infinity. The other component of the evaluation function is a metric that examines how many winning combinations that particular slot could ever be involved in. Assuming the board is not a win, the two components are each scaled by a constant and then added together. These constants were chosen after playing various games against an online AI. This value is added to the score for the current player from previous calculated boards, and the final heuristic value is the (sum of red scores) – (sum of blue scores) \* color. While this evaluation is far from a perfect representation of what the state of the game is, it is relatively fast to calculate.
4. By using the ±Infinity evaluation for a winning state, the AI agent always takes a win and blocks a loss when possible. One problem that occurred toward the end of games was that the computer would make illegal moves. I set the value of full columns to negative infinity, and told the AI to take the center-most move if two moves achieved the same value. If it knew it would lose on any move, it would attempt to take an illegal move, as everything evaluated to negative infinity. I addressed this issue, however.

**Conclusions:**

1. Strengths and Weaknesses:
   1. Strengths. My program utilizes nearly all the optimizations that were presented in class with the A/B pruning. Additionally, the parallel search takes advantage of all of my CPU. The graphical interface makes the game enjoyable to play. The tuning of the weights for the two portions of the heuristic function allowed the AI to play well.
   2. Weaknesses: Even though my program does not look through the entire board every time, improvements could likely be made to speed up that function. Additionally, the heuristic may not have as much success against a player that has a much different style from itself.
2. After playing many games against my AI, I believe that it performs reasonably well. Though I have learned some of the patterns that it plays by, and can trick it a little bit at times, it usually wins if it can. It obviously is at a disadvantage when it goes second, as any player would be.
3. To improve the program I would:
   1. Update the heuristic function: I could represent the boards in a more compact way, perhaps a 1D array, and evaluate using fewer steps. This may even enable me to look at the entire board and not just the most recently placed piece.
   2. Negamax improvements: There was a variant of the negamax algorithm that utilized some memorization. This in combination with an evaluation function that looks at the board as a whole could be very useful.
   3. Timed cutoff: When columns are full or nearly full, the algorithm runs much more quickly. I could potentially search more deeply as the board filled up, and cutoff at 10 seconds for earlier moves.