Sam Chott
Project 2: Connect 4

Alpha-beta

Alpha-beta lowered the search requirements for the Connect 4 player significantly. Evaluating the first move, without alpha-beta, required evaluating 1296 different boards, when traversing the tree to a depth of 4. With alpha-beta, it only required evaluating 271 boards. This is a significant difference, and was actually perceivable, in terms of lag time, when playing the game. Presumably, if the branching factor of the min-max tree were greater, as it would be with a wider Connect 4 board, the pruning would even be more effective, when compared to the non-pruning algorithm run on the same state.

Evaluation function

Unfortunately, I didn't have/make enough time to come up with a completely original evaluation function. Mine functioned by essentially counting the number of black tokens and white tokens in each 4-long horizontal, vertical, and diagonal. If there were both black tokens and white tokens, it was assigned a value of 0, since that space is neither a threat nor a possibility of winning. If there were black tokens or white tokens only, then a negative, for white tokens, or positive constant, for black tokens, would be returned, depending on how many there were. The constants returned for every possible 4-long were then added together, and that was used as the evaluation variable. I took the concept of using constants and 'tuning' them from my experience with evolutionary computation.

Luckily, I found a good combination of constants quickly, after a minimum of experimentation. The constant is high for 3-long token sets, but not so high that the program won't try to make 2-long token sets. Either it works pretty well, or it just happens to be the right combination of constants to beat the ff.fas program at Connect 4. Either way, I'm not complaining.

At first, I did try to simply add 1 to a subscore for each black token, subtract 1 for each white token, and do nothing for empty spaces. My plan was then to square the subscore and add it into the total score. This didn't work well, mostly because it would lead the program to still want to add more black tokens to, say, (WHITE BLACK BLACK EMPTY), which, depending on the surrounding spaces, is often wasteful. Another problem with this scheme is that it was less tunable, since the values for each subscore couldn't be individually adjusted.

I decided to keep my evaluation function simple, because complex problem-solving is already used in the min-max tree. There was no need to search deeply in the evaluation function, because the tree was already ensuring that searches were sufficiently deep.

Functions added

mm-ab.lisp

get-children: Returns nil if there are no children, otherwise returns the output of 'successors' function.

winning-or-losing: Returns 1 if computer wins on the given board, -1 if the player wins, and nil if there are no wins on the given board.

win_lose_with_array: Does the actual computation for winning-or-losing.

eval.lisp

win_score

win_to_win_array

rank_win

These were all various helper functions made to avoid an overly-long rank_board function.

Functions modified:

eval.lisp

rank_board: Implemented evaluation.

mm-ab.lisp

minimax_decision: Implemented.

max-value: Implemented.

min-value: Implemented.

c4.lisp

Connect4: Slightly modified to allow for undo (added past_boards).

Play: Slightly modified to allow undo (added past_boards, passed more vars to Human-Move)

Human-Move: Modified to allow undo (passes vars to Player-Chooses-Slot).

Player-Chooses-Slot: Handles actual computation of undoing moves.

Drop-Token: Modified to help games where undo had been done to end smoothly.