Rachel Chiang

Project 3: 4-in-a-Line

CS 420.01

Minimax with Alpha-Beta Pruning for 4-in-a-Line Game on an 8 × 8 Board

In this project, the minimax algorithm with alpha beta pruning is implemented to attempt to play 4-in-a-Line on an 8 × 8 board. This game is similar to Go and Tic-Tac-Toe in that it is a two-player game and the participants compete to create a line of four consecutive pieces belonging to oneself, but it is distinct in that this version requires only horizontal and vertical lines, so diagonals are ignored. The formal game problem formulation is provided below.

Initial State	An empty 8 × 8 board
Player	A player is one who controls the X pieces or the O pieces
Successor Function	An action entails the placement of one piece on the board, where a piece previously does not exist, and players must alternate turns.
Goal Test	There are three types of potential terminal states: a draw, the X player wins, or the O player wins. A draw occurs when the board is full (though a draw may be evident beforehand). A win occurs when there is a line of four consecutive X's or four consecutive O's.
Utility Function	The utility will be estimated by an evaluation function, which will be detailed below.
Game Tree	The initial state and subsequent legal moves according to the successor function.

(Above) Game problem formulation for the 4-in-a-Line game.

The artificial player will be implemented using the minimax algorithm with alpha-beta pruning. The minimax algorithm generates minimax values for every node of the game tree, and the values that the algorithm uses to pass judgement is generally provided by a utility function which is defined at a goal state. In a game, one player may be Max and is trying to maximize the outcome of the game such that it ends in his favor, but he assumes that his opponent or opponents, the Mins, are also seeking to maximize their outcomes; from Max's point of view, they are attempting to minimize the utility value. Given an extremely simple game with a very small game tree, this algorithm is complete and works well, but realistically, game trees tend to be far too large for an ordinary minimax algorithm because it would have to traverse the entire tree. Consequently, certain strategies can be used to try to circumvent the requirement of the entire tree.

One such strategy is to skip unnecessary branches of the game tree using alpha-beta pruning. Alpha-beta refers to a pair of values, (α, β). These values are propagated down the tree as it is being searched and allows the tree to prune branches that provide less optimal utility values than ones that have been found already. This still may not be enough to solve a game problem however, since the tree will likely still be very large, and in many cases, reaching the true leaf node of the branch to calculate the utility value is not feasible. Thus, it may be helpful to also implement an evaluation function, which, given one node of the tree, it will estimate the utility value of the node according to the state of the game.

The alpha-beta pruning algorithm implementation is rather straightforward, as it is simply a depth-first search with extra control statements to eject early according to the alpha and beta values. In this project implementation, the flavor of depth-first search used is iterative-deepening, since there is a time limit for the algorithm to run. At five seconds, the algorithm can search to a depth of five. There is also a somewhat simplistic move-ordering system in place. Firstly, if the algorithm has yet to search at some depth (which corresponds to one piece's move), it will start at the opponent player's last move, which loosely imitates how a human might play against another human. Then, if the algorithm has already searched at this depth, that means that the evaluation function will have already passed through these states, so it will prioritize searching down actions that have more favorable estimated utility values.

To put bluntly, the evaluation function strategy used is not very intelligent. The program as it is never wins, and it only pulls a draw from a human player if the human player is not actually paying attention or is more focused on teasing the agent. The final form of the evaluation function will be described. First, the game will test for a game-over scenario; that is, if one of the pieces has a four-line or if the board is full (a draw). If it is at a winning/losing state, it will add or subtract a weight, called the `fourWeight`, or if it is a draw, it will add nothing. Then, it will count how many positive four-lines can be made and add this according to different weights for lines already containing three or two pieces in it, the values `threeWeight` and `twoWeight`, respectively. Then it will count the number of enemy potential four-lines that are being blocked by it, and the value it returns is a sum of counters multiplied by the same weights as previously. Finally, a slight center bias is added to the calculation, which comes with, as one may expect, its own weight, the `centerWeight`. The final values for each of these were locked in as follows:

Weight	Value
`centerWeight`	2
`blockWeight`	1000
`fourWeight`	10000
`threeWeight`	100
`twoWeight`	10

(Above) Weights used for the utility value approximation and their values.

These calculations sum up to four large for-loop groups and one smaller loop, so the evaluation function ends up being somewhat complex. Unfortunately, it also is not a particularly intelligent evaluation function. Quite consistently, the agent will, after four to six turns in, place pieces where they hold no true benefit to attaining a win or a draw. Different weight values were played with, but the greatest weakness of the agent most likely lies in its evaluation function.

The evaluation function seems too time-consuming computationally. Aside from possibly simplifying the evaluation function, possible improvements (or full adjustments) to the evaluation function include the following:

- Prioritize or more clearly emphasize moves that accomplish more than one strategic goal. For instance, an action that both blocks opponent four-lines and contributes to the construction of a new four-line would be a better choice than doing only one or neither of those. As it is, the algorithm simply accomplishes this as a side-effect of summing values together, but it may be worth consideration to look into a more sophisticated way of calculating these than simply adding weighted counters. For instance, the opponent will play d5, and the agent will respond with e4, and although it is better to place a piece toward the center, that piece is on a diagonal with the opponent's, which does not help contribute to blocking.
- Make more efficient blocks. The fault for the issue above may actually lie in the blocking mechanism in general. The blocking evaluation actually uses the same counter that counts how many four-lines can be created in one or two moves. This may shed light on the fact that toward mid-game, the agent picks seemingly poor moves; it is because it does not consider that it may have already blocked a specific winning row—for instance: below, f4 was played after b4 even though a4 was still open at the time. (The exact game sequence was Opponent versus Player with (1) d4, d5; (2) c4, e4; (3) c3, c2; (4) c5, c6; (5) b4, f4; (6) a4. More examples can be viewed in the appendix.)

```
  1 2 3 4 5 6 7 8
A - - - X - - - -
B - - - X - - - -
C - O X X X O - -
D - - - X O - - -
E - - - O - - - -
F - - - O - - - -
G - - - - - - - -
H - - - - - - - -
```

- Keep track of remaining valid potential winning lines. Initially, 80 four-lines are possible. As more and more pieces are placed on the board, this value decreases universally, as well as for each individual—one line may be voided for X, but that does not mean O cannot accomplish the four-line. This could be used in the evaluation function itself, or perhaps it could persist throughout the search algorithm itself, which could help when picking actions, or maybe it could be useful in both.
- Look for opportunities for special two-spaces. From playing the game against the agent, it is evident that the agent fails to foresee that leaving an "open-two" will most certainly end in its defeat. An example of an open-two is "`Y - X X - Z`" and it is `X`'s turn, where the hyphens are open spaces and at least one of `Y` or `Z` must also be open. The cause of or fix for this behavior has yet to be identified, but it may again be the fault of the poor blocking mechanism, since blocking directly adjacent to recently-placed opponent pieces should avoid most of these situations. The evaluation function may also simply be too shortsighted in finding actual threats. Even given thirty seconds of tree traversal time, it still makes fatal errors.

In retrospect, there was a rather belated realization that the mini part of the minimax algorithm actually also uses the same data structure for sorting previously-explored levels, so it is sorting in the same manner that the max is sorting. This means that the mini component is searching much less efficiently, but since it still is seeking minimal values, this should not affect the honesty of its decision-making. Furthermore, careful attention was possibly misplaced in managing memory even though depth-first searches tend to hold linear space complexities anyway, but binary is more interesting to use than arrays and should also yield generally faster computations. For instance, to locate a winning four-line, rather than moving and accessing four memory locations of an array, a single bitwise operation can check all four “locations” at once. Using standard minimum data type sizes (for C++), four different implementations were considered, which can be viewed in the appendix.

Appendix

Game Examples

Unfortunately, the agent did not win or draw any of these games.

Example 1.

```
  1 2 3 4 5 6 7 8
A - - - X - - - -
B - - - X - - - -
C - O X X X O - -
D - - - X O - - -
E - - - O - - - -
F - - - O - - - -
G - - - - - - - -
H - - - - - - - -
Opponent vs. Player
1. d4  d5
2. c4  e4
3. c3  c2
4. c5  c6
5. b4  f4
6. a4
```

Example 2.

```
  1 2 3 4 5 6 7 8
A - - - - - - - -
B - - - - O - O -
C - O X X X X - -
D - - - - X - - -
E - - - O X - - -
F - - - - O - - -
G - - - - - - - -
H - - - - - - - -
Opponent vs. Player
        1. d5  e4
        2. e5  f5
        3. c5  b5
        4. c4  b7
        5. c3  c2
        6. c6
```

Example 3.

```
  1 2 3 4 5 6 7 8
A - - - - - - - -
B - - - O - - - -
```

```
C - - X X X X O -
D O X X X O O - -
E - - - O - - - -
F - - - - - - - -
G - - - - - - - -
H - - - - - - - -
Opponent vs. Player
        1. d4  d5
        2. d3  d6
        3. d2  d1
        4. c4  e4
        5. c5  b4
        6. c6  c7
        7. c3
```

State Data Type Considerations

Consideration	Bits	Some Thoughts
Two-dimensional integer array	1024	Mostly straightforward to implement, but moving back and forth between indices to check the conditions of states may be annoying. Eats a lot of space.
Two-dimensional character array	512	Mostly straightforward to implement, but see above. Eats some space.
One-dimensional integer array	128	Checking the conditions of states seems like it would be a big pain. Neat and tidy. Somewhat small.
Two 64-bit integers	128	Checking the conditions of states should be convenient. Bits are fun. Somewhat small.