Kevin Langer

CS4100

Tick-Tac-Toe documentation

I picked a minimax algorithm to be used for my tick-tack-toe game. This decision tree was a logical pick because the game is fully observable and it is a zero sum game. Moreover, because tick-tack-toe is comprised of 9 positions each of which are filled after each move, the solution space is 9!, which is easily within the limits of brute force computation. The minimax algorithm works by taking a board in its current state and producing all possible boards for the next move. It recursively produces each collection of boards until a depth of 8 is reached. At this point it evaluates the boards and a value is propagated upwards in the tree. A winning game is qualified as a *four* for the AI, a tie is a *three*, and a loss is a *two*. Because the AI produces all possible moves, not just moves it expects to take, it logically propagates the tree with the AI picking moves with a *four* only picking moves with a *three* and under normal conditions never picking moves with a *two.* However, moves in which would be the human player the AI propagates a *two* and then a *three* and under normal conditions never a *four.* This results in a minimax tree. Because of the small search space, alpha-beta pruning was not required to get acceptable performance. However, for the first move, the computer randomly chooses one of the corners to go to. This is because the first move would take the longest and always result in the computer picking the top left space (minimax is deterministic). In order to make the computer seem more humanlike and provide for more possible games, the first move adds two bits of entropy by picking form any one of the corners. The minimax tree is very versatile. It can be used starting from any board condition and the rules of winning can easily be modified. Under the normal conditions and rules it will never fail to win or draw a game. The validity of this algorithm was play-tested against humans (one of which found a bug!) and against random inputs.