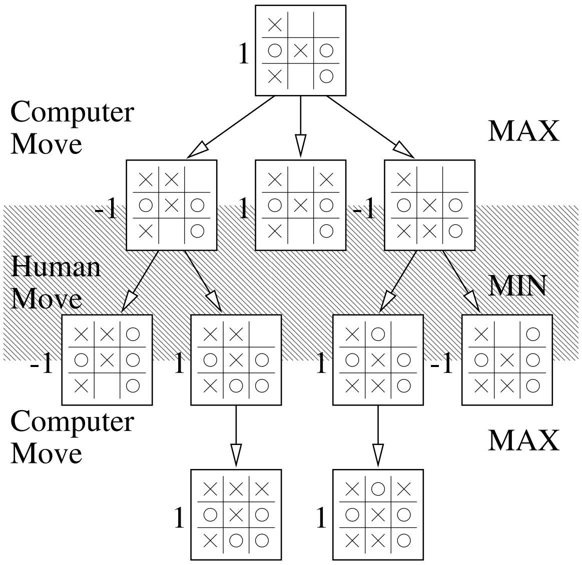
CS 61B: Lecture 22
Wednesday, October 14

Game tree search

How could we design a program that plays Tic Tac Toe? The standard technique searches for the best moves by using a *game tree*, which looks much like a family tree. The ancestor at the top of the tree is the current grid, accompanied by a record of whose turn it is. The children of the ancestor form the set of possible grids that result from considering each legal move. These children have children of their own, which represent each of the opponent's possible countermoves.



If we assume that both opponents are infinitely intelligent and will always choose the best move, then we can determine the computer's best move using the game tree. To any legal grid (coupled with information about whose turn it is), we assign a numerical score that indicates how optimistic we are about winning. For the sake of argument, give a grid a score of 1 if the computer is guaranteed a win (assuming it plays perfectly), a score of -1 if the computer's opponent is guaranteed a win, and a score of 0 if perfect players will draw. (It's better to give these constants names like `COMPUTER_WIN`, `HUMAN_WIN`, and `DRAW` as Weiss does, but these names are too long for the figures herein.)

How do we score a given grid? Suppose the computer is playing X. If there are three X's in a row, assign the grid a 1; for three collinear O's, assign it a -1. If the grid has no empty squares left, and nobody has won, assign it a 0.

The score of any other grid is computed using a *minimax* algorithm. We consider each possible move, and recursively assign a score to each grid that results. If it's the computer's turn, we choose the move that yields the maximum score, and assign the same score to the current grid. If it's the opponent's turn, we assume that the opponent plays perfectly; hence, we choose the move that yields the minimum score, and assign that score to the current grid.

This procedure is recursive, and the figure above illustrates the method calls executed when evaluating several grids' scores. Each grid's score is printed to its left. Because the top-level grid has a score of one, the computer is guaranteed a win, which is obtained by choosing the second of the three possible moves.

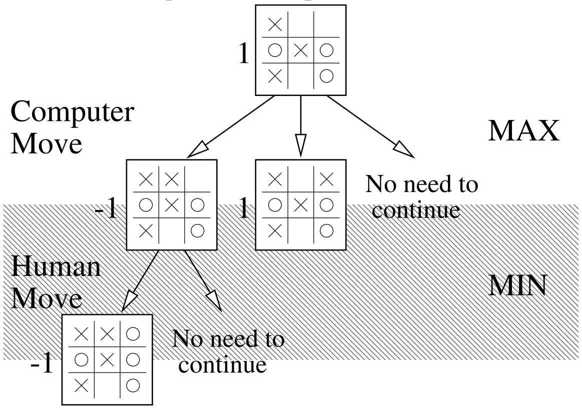
The following pseudocode, modeled on Weiss' Figure 7.27, computes a grid's score and the best move from that grid (which determines the grid's score). A `Best` object holds a record of the score and best move.

```
public Best chooseMove(int side) {
  Best myMove = new Best();    // My best move
  Best reply;          // Opponent's best reply

  if (the current grid is full or has a win) {
    return the grid's score, with no move;
  }

  if (side == COMPUTER) {
    otherSide = HUMAN;     myMove.score = -2;
  } else {
    otherSide = COMPUTER;  myMove.score = 2;
  }
  for (each legal move m) {
    perform move m;
    reply = chooseMove(otherSide);
    undo move m;
    if (((side == COMPUTER) &&
         (reply.score > myMove.Score)) ||
        ((side == HUMAN) &&
         (reply.score < myMove.Score))) {
      myMove.move = m;
      myMove.score = reply.score;
    }
  }
  return myMove;
}
```

NOTE: By setting the initial score out of range (-2 or 2), I ensure that at least one legal move will be assigned to `myMove`, thereby fixing a bug in Weiss' code.

Simple pruning

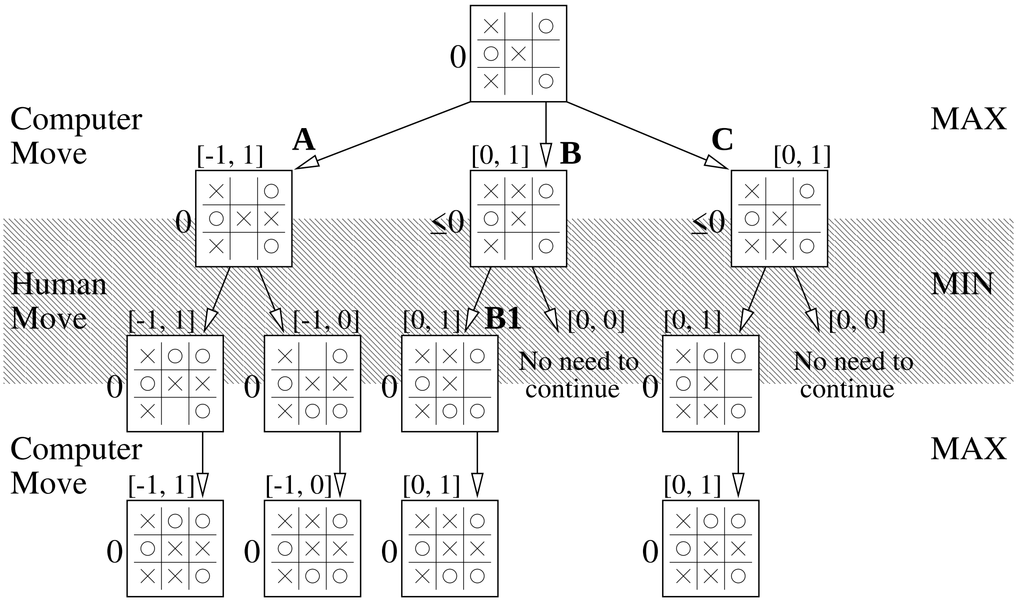
If, at any grid in the game tree, the player who has the turn has discovered a guaranteed winning move, there's no reason to continue to search for a better move. Hence, we can prune away some recursive calls. The previous example now looks like this:



Alpha-beta pruning

A more aggressive pruning technique (which subsumes simple pruning) is based on the following observation. (See the figure on the next page.) Suppose the computer has discovered a move (A) that guarantees a draw, and is investigating an alternative move (B). The computer begins to consider all the moves its opponent could make from grid B. It discovers that if the opponent makes move B1, the opponent can force a draw.

It would be a waste of the computer's time to continue investigating moves the opponent could make from grid B. Since the opponent can, at the very least, force a draw, move B is no better for the computer than move A (and might be worse). The computer should simply go on and consider move C.



To turn this insight into an algorithm, we pass two additional parameters to the `chooseMove()` method: α and β. The parameter α is a score that the computer knows with certainty it can achieve; for instance, if $\alpha = 0$, then the computer knows it can force a draw, and is only interested in searching for moves guaranteed to do better. Conversely, β is a guarantee that the opponent can achieve a score of β or lower. For any grid, we maintain values of α and β based on our current knowledge of the best moves discovered thus far. If α becomes equal to or greater than β, then further investigation of the current grid is useless.

In the figure above, for instance, grid A is found to have a score of 0. The computer knows that it cannot do worse than a draw (the top grid is a "MAX" grid), so it sets $\alpha = 0$ and uses the parameters $[\alpha, \beta] = [0, 1]$ to begin investigating grid B. Grid B1 is found to have a score of 0. Because the computer recognizes that its opponent can force a draw from grid B (which is a "MIN" grid), it sets $\beta = 0$ for grid B. Because the parameters $[\alpha, \beta] = [0, 0]$ have converged, there is no point in investigating other children of grid B. We never compute the exact score of B, but we know it is less than or equal to zero (because B is a "MIN" grid). Hence, move B cannot be better than move A, so we go on to move C. After the computer considers move C's first child, C's remaining children are similarly pruned. Adapting from Weiss' Figure 10.12:

```
public Best chooseMove(int side, int alpha,
                       int beta) {
  Best myMove = new Best();    // My best move
  Best reply;          // Opponent's best reply

  if (the current grid is full or has a win) {
    return the grid's score, with no move;
  }

  if (side == COMPUTER) {
    otherSide = HUMAN;     myMove.score = alpha;
  } else {
    otherSide = COMPUTER;  myMove.score = beta;
  }
  for (each legal move m) {
    perform move m;
    reply = chooseMove(otherSide, alpha, beta);
    undo move m;
    if ((side == COMPUTER) &&
        (reply.score > myMove.Score)) {
      myMove.move = m;
      alpha = myMove.score = reply.score;
    } else if ((side == HUMAN) &&
               (reply.score < myMove.Score)) {
      myMove.move = m;
      beta = myMove.score = reply.score;
    }
    if (alpha >= beta) { return myMove; }
  }
  return myMove;
}
```

Transposition tables

An inefficiency of unadorned game tree search is that some grids can be reached through many different sequences of moves, and so the same grid might be evaluated many times at great expense. To save some of this cost, we maintain a record of previously encountered grids, called a *transposition table*. We implement this table using a dictionary (a.k.a. a hash table). The hash table stores objects, each consisting of a grid and its associated score. The *key* that allows us to look up such an object is a grid (without its associated score). Each time `chooseMove()` evaluates a grid, it should first check whether the grid is in the transposition table; if so, its score is returned immediately. Otherwise, its score is evaluated recursively and stored in the transposition table.

However, we do not store grids that occur below a certain (experimentally determined) depth in the tree, because high-depth nodes are too numerous and are repeated too rarely.

Combinatorially huge game trees

Game trees grow exponentially with tree depth. Even with every technique at our disposal, we cannot hope to explore the entire tree for a game of chess. Hence, game tree search typically limits its recursion to a specified depth. Positions at the maximum depth are evaluated not by recursive search, but by less accurate heuristic means. These heuristics generally compute numerical estimates of the computer's optimism, which might be any real numbers between -1 and 1. Even if we adopt an infinite-valued (rather than three-valued) scoring system, alpha-beta search still works correctly; in fact, it is at its best in such circumstances.