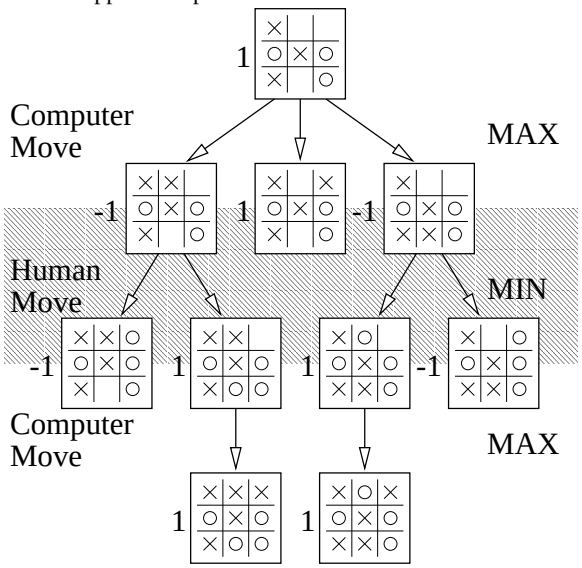
CS 61B: Lecture 18
Friday, October 8, 2010

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. A game tree is *not* a data structure; it is the structure of a sequence of recursive method calls.

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. Each legal grid (coupled with information about whose turn it is) is assigned a numerical score that indicates how optimistic we are about winning. For concreteness, 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`, but these names are too long for the figures herein.)

How do we score a given grid? It's easy if the game is over. 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 by a *minimax* algorithm. We consider each possible move, and determine the child grid that each move creates. We assign a score to each child grid by calling the minimax algorithm recursively. Then we assign a score to the current (parent) grid. 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. So we choose the move that yields the minimum score, and assign that score to the current grid.

Minimax is recursive, and the figure above illustrates the recursive method calls that minimax executes when evaluating several grids' scores. Again, no tree data structure is created! At any one point in time, the program stack represents one path down the tree (but the root of the tree is at the bottom of the stack). 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 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 best `move` and its `score`.

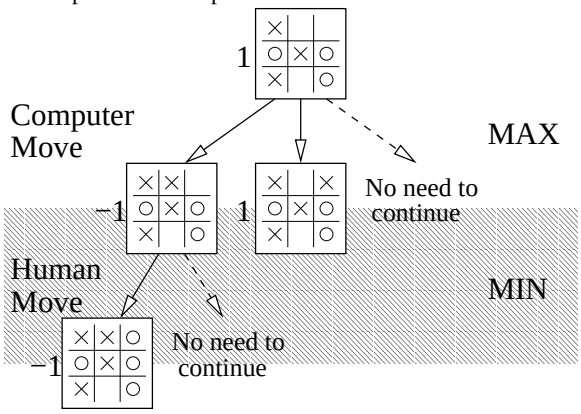
```
public class Grid {
  public Best chooseMove(boolean side) {
    Best myBest = new Best();     // My best move
    Best reply;          // Opponent's best reply

    if (the current Grid is full or has a win) {
      return a Best with Grid's score, no move;
    }
    if (side == COMPUTER) {
      myBest.score = -2;
    } else {
      myBest.score = 2;
    }
    for (each legal move m) {
      perform move m;     // Modifies "this" Grid
      reply = chooseMove(! side);
      undo move m;        // Restores "this" Grid
      if (((side == COMPUTER) &&
           (reply.score > myBest.Score)) ||
          ((side == HUMAN) &&
           (reply.score < myBest.Score))) {
        myBest.move = m;
        myBest.score = reply.score;
      }
    }
    return myBest;
  }
}
```

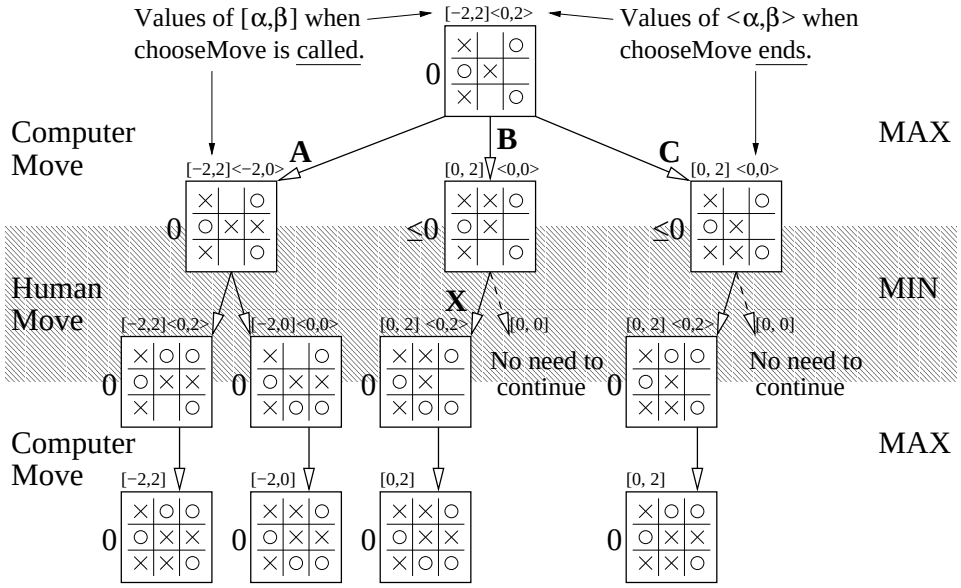
Why is `myBest.score` initially set to -2 or 2? By setting the initial score out of range, we ensure that at least one legal move will be assigned to `myBest`. It is a simple way, albeit not the most transparent way, to make sure that `chooseMove` always accepts the first move investigated.

Each grid in the game tree at left represents one invocation of `chooseMove`. The children of each grid represent recursive calls, which are executed in order from left to right. When any particular grid is invoked, that grid and its ancestors (parent, grandparent, etc.) are stack frames on the program stack.

Simple pruning

If, at any grid in the game tree, a player discovers a guaranteed winning move, there's no reason to continue to search for a better move. Hence, we can save time by pruning 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 above.) 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 X in response, 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—though it 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 β becomes equal to or less than α, then further investigation of the current grid is useless.

In the figure above, for instance, grid A has a score of 0. The top grid is a MAX grid, so the computer cannot get lower than zero (worse than a draw), and it sets $\alpha = 0$ at the topmost grid and uses the parameters $[\alpha, \beta] = [0, 2]$ to begin investigating grid B.

Minimax computes that Grid X has 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 met in the middle, 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 child is likewise pruned.

The following pseudocode formalizes game tree search with alpha-beta pruning. Note that α only changes during a computer (MAX) move, and β only changes during an opponent (MIN) move. The top-level chooseMove invocation should be called with $[\alpha, \beta] = [-2, 2]$ to make sure that some move is always selected.

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

  if (the current Grid is full or has a win) {
    return a Best with the Grid's score, no move;
  }
  if (side == COMPUTER) {
    myBest.score = alpha;
  } else {
    myBest.score = beta;
  }
  for (each legal move m) {
    perform move m;       // Modifies "this" Grid
    reply = chooseMove(! side, alpha, beta);
    undo move m;          // Restores "this" Grid
    if ((side == COMPUTER) &&
        (reply.score > myBest.Score)) {
      myBest.move = m;
      myBest.score = reply.score;
      alpha = reply.score;
    } else if ((side == HUMAN) &&
               (reply.score < myBest.Score)) {
      myBest.move = m;
      myBest.score = reply.score;
      beta = reply.score;
    }
    if (alpha >= beta) { return myBest; }
  }
  return myBest;
}
```

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 heuristics called *evaluation functions*. Evaluation functions compute numerical estimates of the computer's optimism. These scores are not just -1, 0, or 1, but can take on a continuous range between (for example) -1.0 and 1.0. Even with this infinite-valued (rather than three-valued) scoring system, alpha-beta search still works correctly; in fact, it is at its best in such circumstances.