Min-Max Trees

One other use of recursion is a technique called backtracking. Backtracking essentially involves a recursive exhaustive search that tries to maximize a particular choice. A popular use of backtracking is in computer players of games, such as chess.

The basic idea is as follows: when the computer is moving, it "tries" out all of its possible moves. It wants to pick the move which gives it the best advantage. One way to do this would be to look at each "answering" move the human player could make. Imagine that each of these moves had a score such that the higher the score, the more favorable the move is for the human player. Our goal as the computer of course, would be to *minimize* the score achieved by the human. We would naturally assume that the human makes its best possible move, ie. the *maximum* valued move of all of its possibilities. Thus, as the computer, we should pick the *minimum* of all of these *maximums.* Thus, the term mini-max tree is given to this sort of search.

Consider the following diagram:

Rather than just search two levels of this tree however, we can recursively search to the bottom of the tree for simple games like Tic-Tac-Toe. Let's analyze this pseudocode:

```
public Best chooseMove(int side) {

  Best reply;
  int opp, dc, simpleEval, value;
  int bestRow=0, bestCol=0;

  // Base case: Board is already done!
  if ( (simpleEval = positionValue()) !=
        UNCLEAR)

    return new Best(simpleEval);

  // Set up the opponent variable.
  if (side == COMPUTER) {
    opp = HUMAN; value = HUMAN_WIN;
  }
  else
    opp = COMPUTER; value = COMPUTER_WIN;
  }
  // Loop through each possible move.
  for (int row=0; row<3; row++)
    for (int col=0; col<3; col++)

      if (squareIsEmpty(row, col)) {

        // Try out this move.
        place(row, col, side);

        // Recursively get this move's score
        reply = chooseMove(opp);

        // Undo the temporary move.
        place(row, col, EMPTY);

        // If this move improves our
        // position, update the appropriate
        // variables.
        if ((side == COMPUTER &&
             reply.val > value) ||
            (side == HUMAN &&
             reply.val < value)) {

          value = reply.val;
          bestRow = row; bestCol = col;
        }
      }

  // Return the best move found.
  return new Best(value, bestRow, bestCol);
}
```