

Materials:

1. Slide show of Tic-Tac-Toe game tree (ply 6), showing development of min-max values
2. Slide show of alpha-beta example

I. Introduction

- -----

- A. In the search methods we have discussed so far, we have assumed that we are in total control of the decision points along the way. This is not necessarily the case if we are working in an adversary environment. The classic illustration of this is two-player games, in which every other move is totally out of our control.
- B. Two player games constitute an interesting and important class of search problems.
 1. The ability to play games such as chess is often regarded as a mark of high intelligence, and efforts to produce programs that could do so began early in the history of AI.
 2. Game playing is a search problem, in the sense that whenever it is the computer's turn to move, it must choose the best move from a set of legal moves for the current board position. Typically, this is done by exploring the consequences of each move through several moves into the future - e.g. "if I do this, then my opponent can do ..., to which I respond by doing ...".
 3. That, is, at any given time, the computer is confronted by a tree of possible moves - its choices, possible opponents responses to each, possible countermoves by the computer ...

PROJECT: Tic-tac-toe example. Note terminology "6th ply"

- C. In addition to raising the problem of our controlling only half the moves, games present a second problem: for any interesting game, the search tree is much too large to be exhaustively searched.

Ex: Even an uninteresting game like tic-tac-toe has a search tree with about $9! = 362,880$ nodes at the outset (i.e. when choosing an initial move). For chess, the estimates range well over 10^{100} . (A game would hardly be fun if one could easily discover a path from the start state to a guaranteed win!)
- D. To address these two issues, we introduce two new ideas:
 1. A search technique known as min-maxing, which accounts for the back and forth nature of play between opponents. We will discuss this in a moment.
 2. A heuristic technique known as static evaluation functions.
- E. We will also consider various approaches to pruning the search tree in the face of combinatorial explosion. The ability to do this well is one of the key things that distinguishes great chess programs from just good ones.

II. Min-Max Search

-- -----

- A. For games, we use a search procedure called mini-maxing, which can be viewed as a variant of an and/or tree.
1. Levels of the tree alternate as to whose turn it is to move. At level 1, 3, 5 etc. it is our turn; at levels 2, 4, 6 etc. it is our opponent's turn.
 2. When it is our turn to move, we can choose any move that leads to our goal; thus we are happy with the node if any move leads to victory and the nodes corresponding to our moves are OR nodes - i.e. we control which branch to take, so a node is good if branch 1 is good or branch 2 is good or ... An or node is good for me if ANY of its branches is good for me.
 3. When it is our opponent's turn to move, we have to be able to cope with anything he might do - that is, we want to put him into a situation where any choice he makes still leaves it possible for us to achieve our goal. Thus, these are AND nodes - i.e. the node is good only if branch 1 is good and branch 2 is good ... An and node is good for me iff ALL of its branches are good for me.

PROJECT TIC-TAC TOE EXAMPLE: the top node is an or node; the next level are and nodes, etc. The arcs on the and nodes indicate that we must find an option further down the tree that works no matter which of the possibilities our opponent might choose at this level.

- B. The basic idea is to work backward from the leaves of the tree, toward the root, assigning a value to each node.
1. Nodes representing a final state in the game are assigned values as follows:
1 = win for us
-1 = loss for us (win for opponent)
0 = draw
 2. Once all the children of a non-terminal node are labelled, it can be labelled as follows:
 - a. If it is an or node, we MAXIMIZE - i.e. we choose the maximum value from among any of the children.

Rationale: We are in control, so we can choose the best path.
 - b. If it is an and node, we MINIMIZE - i.e. we choose the minimum value from among the children.

Rationale: Our opponent is in control, and will presumably choose the path which is best for him and hence worst for us.
 3. Ultimately, when we get to the top level, we will choose from among the moves available to us the one having the highest value.

SLIDE SHOW: Development of min-max values

- C. Although we think of min-maxing as being done bottom up, we actually do it top-down, using a recursive, depth first function. Because this is based on DFS, we won't need to keep all the values for all the nodes - only the values for the immediate children of the top-level node, which will then be used to make our choice. We will look at pseudo-code for this shortly.

III. Static Evaluation Functions

--- -----

- A. In our discussion of min-maxing, we assumed that it would be possible to generate the game tree all the way down to the terminal nodes. In general, this is not the case, for reasons we have noted earlier.
- B. Thus, at some level in the search, we have to assign values to nodes that are not terminal nodes for the game as a whole. We do this using a static evaluation function.
 - 1. A static evaluation function is a function that looks at a given board position and assigns it a score - without doing further search.
 - 2. If the board configuration does happen to represent a terminal state, then the static evaluation function can assign an appropriate value for win, lose, or draw.
 - 3. Otherwise, the function returns a value based on an estimate of how likely it is that configuration will ultimately lead to a win for us or our opponent.
- C. The development of such a function is a difficult task, and requires a good grasp of what is important in the game. What we want is a function that gives a large positive score if the situation is good for us, and a negative score if the situation is good for our opponent.
 - Ex: For checkers, one possibility (though not the best) is the difference between the number of my checkers remaining and the number of my opponent's - perhaps counting kings as 2.
 - Note: Typically, the static evaluation function returns values in an arbitrary range, not just -1 .. 1. We can consider a sure win for us as having as its value a very large positive integer (we will call this "infinity"), and a sure win for our opponent would have a very small negative value (which we will call - "infinity.) One can also get the static evaluation function result to lie in -1 .. 1 by dividing by some normalizing factor - e.g. for checkers difference between my number of pieces and my opponent's, divided by the total number of pieces in the game to start with (24).
- D. We combine the static evaluation function with a limit on the depth of a search. When we get to the prescribed depth limit, we evaluate the board using the static evaluation function.
 - 1. A simple variant of game tree search uses a fixed maximum depth at all times.
 - 2. More sophisticated variants may allow some "interesting" branches of the tree to be searched more deeply than others by using some game-specific heuristic.

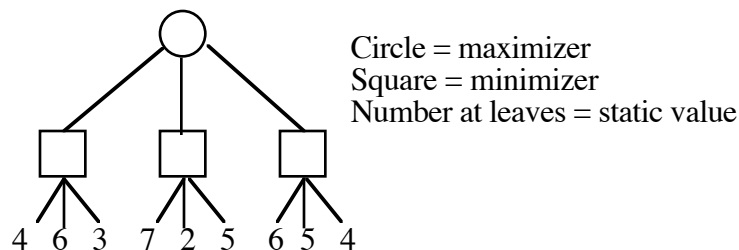
E. A function conventionally called "minimax" implements min-maxing with static evaluation at a specified maximum-depth:

```

int minimax(board, player, depth-so-far)
{
    if (depth-so-far < maximum-depth)
    {
        generate all moves for player given board;
        switch(player)
        {
            case minimizer:
                best-so-far = "infinity" (maximum positive integer);
                while there are still moves to explore
                {
                    new-board = result of applying one move to board;
                    value = minimax(new-board, other player,
                                   depth-so-far + 1);
                    if (value < best-so-far)
                        best-so-far = value;
                }
                return best-so-far
            case maximizer:
                best-so-far = - "infinity" (minimum negative integer);
                while there are still moves to explore
                {
                    new-board = result of applying one move to board;
                    value = minimax(new-board, other player,
                                   depth-so-far + 1);
                    if (value > best-so-far)
                        best-so-far = value;
                }
                return best-so-far;
        }
    }
    else
        return static evaluation of board (from perspective of
                                           top-level caller.)
}

```

Ex: Search to depth of 2 below top:

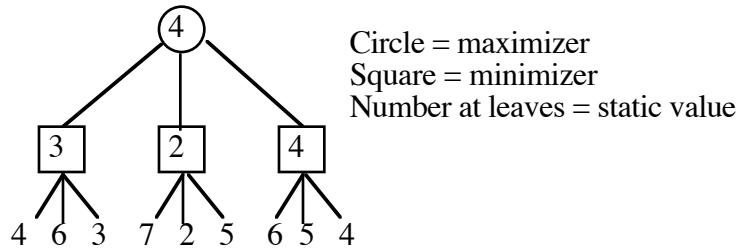


Search down left branch would set leftmost minimizer to 3, and top-level maximizer best-so-far to 3.

Search down middle branch would set middle minimizer to 2, leaving top-level maximizer best-so-far unchanged.

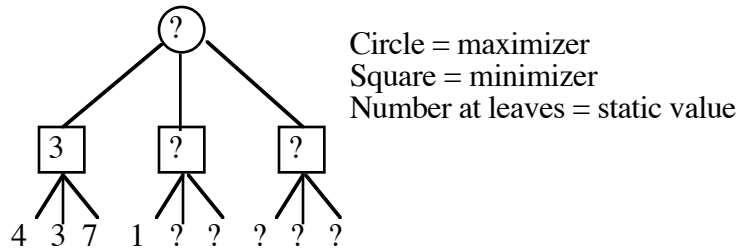
Search down right branch would set right minimizer value to 4, which also becomes value of top-level maximizer.

Final result:



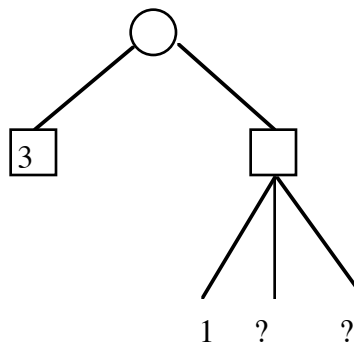
IV. Pruning the Tree

- A. One problem with game tree search is that combinatorial explosion quickly sets in, limiting the depth of the search. This effect can be postponed, but not eliminated, by a technique called alpha-beta pruning. At best, alpha-beta pruning can cut the exponent in the exponential growth in half; but in the worst case it may yield no improvement at all.
- B. The key idea is that at certain points in a min-max search, we can conclude that pursuing certain branches of the tree will yield no information of any value to us.
 1. Example: Consider the following partially-completed min-max search



Evaluating the remaining two children of the middle minimizer cannot possibly affect the outcome of the overall search. Since we know that the minimizer has one child of value 1, we can be sure that it will not return a value greater than 1 (though it could return something less). However, since the left subtree has value 3, the maximizer at the top will always prefer this 3 to a value of 1 or less, so knowing the exact value of the middle minimizer subtree is irrelevant.

2. Example: Consider the following (where irrelevant subtrees are omitted)

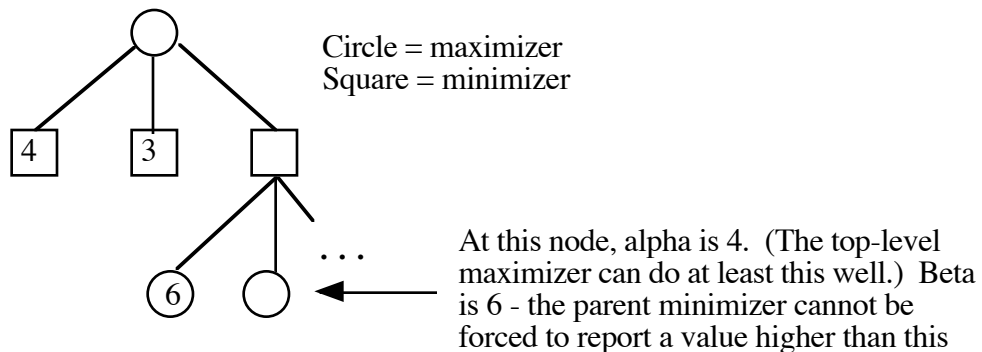


At this point, the remaining branches of the minimizer can be cut off. Why? ASK CLASS

(The fact that the top-level maximizer has seen a 3 means that any value coming up from its second child less than this will be rejected. Since the minimizer cannot now be forced to return a value greater than 1, its true value will have no effect on the ultimate outcome.)

This is known as DEEP CUTOFF.

3. We can implement this algorithmically as follows: To our search procedure, we add two parameters called alpha and beta. These serve to limit the range of values any specific call to minimax will consider. In the top-level call, alpha is set to - infinity and beta to + infinity.
 - a. Alpha represents the best guaranteed value that some maximizer up the tree has found. For example, if ours is the second branch to be considered by some maximizer ancestor of the current node, then alpha for us is the value it found for its first branch. Likewise, beta is the smallest guaranteed minimum some minimizer up the tree has found.
 - b. At any node, we only consider possibilities lying between alpha and beta.
 - i. If the node is a minimizer, and we find some branch with value less than alpha, we return this value without further exploration of other branches. (This node is guaranteed to return a value \leq the value we just found, and a maximizer further above is guaranteed to reject it because it has found an option at least as good as alpha.)
 - ii. Likewise, if a maximizer finds some branch with value greater than beta, it stops considering further branches. (A minimizer above has found an option at least as small as beta, so it will reject our branch in any case.)
 - c. Ex: Consider the following partial search tree (where only the final result of searching the first two branches is shown.)



If any branch below the node we are working on reports a value greater than 6, report it immediately, and stop searching further branches. (This node, as a maximizer, must report this value or greater, and the parent minimizer will prefer the 6 it has seen.)

If the node we are working on reports a value of 4 or less, then its parent minimizer can report it and stop searching its siblings. (If the parent minimizer reports a value less than 4, the top level maximizer will prefer the 4 it has already found.)

- d. In calling down to lower levels, a maximizer hands down max (alpha, best-so-far) and beta. A minimizer hands down alpha and min (beta, best-so-far.)

4. Minimax with alpha beta procedure

```

int minimax(board, player, depth-so-far, alpha, beta)
{
    if (depth-so-far < maximum-depth)
    {
        generate all moves for player given board;

        switch(player)
        {
            case minimizer:

                best-so-far = "infinity" (maximum positive integer);
                while there are still moves to explore
                    and best-so-far > alpha
                {
                    new-board = result of applying one move to board;
                    value = minimax(new-board, other player,
                                   depth-so-far + 1,
                                   alpha,
                                   best-so-far);
                    if (value < best-so-far)
                        best-so-far = value;
                }
                return best-so-far

            case maximizer:

                best-so-far = - "infinity" (minimum negative integer);
                while there are still moves to explore
                    and best-so-far < beta
                {
                    new-board = result of applying one move to board;
                    value = minimax(new-board, other player,
                                   depth-so-far + 1,
                                   best-so-far,
                                   beta);
                    if (value > best-so-far)
                        best-so-far = value;
                }
                return best-so-far;
        }
    }
    else
        return static evaluation of board (from perspective of
                                           top-level caller.)
}

```

5. EXAMPLE: SLIDE SHOW showing development of Alpha/Beta values
(Based on Winston figure 4-18, Page 119)

C. There are a number of other heuristics that can be applied to help decide how much of the search tree to actually consider, given search time constraints - e.g.

1. Quiescence

- It is better to stop the search in a portion of the tree where computed values are changing rather slowly, rather than in the middle of rapid change. (I.e. if in a region of rapid change, push search deeper in this portion of the tree.)

2. The Killer Heuristic

- Consider first the move that has the highest likelihood of being ultimately chosen

D. There is also a danger with minimaxing called the Horizon Effect

- The danger that a serious problem can lie just beyond the point where we stopped searching and did a static evaluation.