Figure 5.7 The alpha beta search algorithm. Notice that these routines are the same as the MINMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain a and $\beta$ (and the bookkeeping to pass these parameters along).

Adding dynamic move-ordering schemes, such as trying first the moves that were found to be best in the past, brings us quite close to the theoretical limit. The past could be the previous move—often the same threats remain—or it could come from previous exploration of the current move. One way to gain information from the current move is with iterative deepening search. First, search 1 ply deep and record the best path of moves. Then search 1 ply deeper, but use the recorded path to inform move ordering. As we saw in Chapter 3, iterative deepening on an exponential game tree adds only a constant fraction to the total search time, which can be more than made up from better move ordering. The best moves are often called killer moves and to try them first is called the killer move heuristic.

In Chapter 3, we noted that repeated states in the search tree can cause an exponential increase in search cost. In many games, repeated states occur frequently because of transpositions—different permutations of the move sequence that end up in the same position. For example, if White has one move, $a_1$, that can be answered by Black with $b_1$ and an unrelated move $a_2$ on the other side of the board that can be answered by $b_2$, then the sequences $[a_1, b_1, a_2, b_2]$ and $[a_2, b_2, a_1, b_1]$ both end up in the same position. It is worthwhile to store the evaluation of the resulting position in a hash table the first time it is encountered so that we don’t have to recompute it on subsequent occurrences. The hash table of previously seen positions is traditionally called a transposition table; it is essentially identical to the explored
list in GRAPH-SEARCH (Section 3.3). Using a transposition table can have a dramatic effect, sometimes as much as doubling the reachable search depth in chess. On the other hand, if we are evaluating a million nodes per second, at some point it is not practical to keep all of them in the transposition table. Various strategies have been used to choose which nodes to keep and which to discard.

5.4 IMPERFECT REAL-TIME DECISIONS

The minimax algorithm generates the entire game search space, whereas the alpha-beta algorithm allows us to prune large parts of it. However, alpha-beta still has to search all the way to terminal states for at least a portion of the search space. This depth is usually not practical, because moves must be made in a reasonable amount of time—typically a few minutes at most. Claude Shannon's paper Programming a Computer for Playing Chess (1950) proposed instead that programs should cut off the search earlier and apply a heuristic evaluation function to states in the search, effectively turning nonterminal nodes into terminal leaves. In other words, the suggestion is to alter minimax or alpha-beta in two ways: replace the utility function by a heuristic evaluation function EVAL, which estimates the position's utility, and replace the terminal test by a cutoff test that decides when to apply EVAL. That gives us the following for heuristic minimax for state $s$ and maximum depth $d$:

$$H_{\text{MINIMAX}}(s, d) =$$

\[
\begin{cases} 
  \text{EVAL}(s) & \text{if CUTOFF-TEST}(s, d) \\
  \max_{a \in \text{Actions}(s)} H_{\text{MINIMAX}}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MAX} \\
  \min_{a \in \text{Actions}(s)} H_{\text{MINIMAX}}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MIN}.
\end{cases}
\]

5.4.1 Evaluation functions

An evaluation function returns an estimate of the expected utility of the game from a given position, just as the heuristic functions of Chapter 3 return an estimate of the distance to the goal. The idea of an estimator was not new when Shannon proposed it. For centuries, chess players (and aficionados of other games) have developed ways of judging the value of a position because humans are even more limited in the amount of search they can do than are computer programs. It should be clear that the performance of a game-playing program depends strongly on the quality of its evaluation function. An inaccurate evaluation function will guide an agent toward positions that turn out to be lost. How exactly do we design good evaluation functions?

First, the evaluation function should order the terminal states in the same way as the true utility function: states that are wins must evaluate better than draws, which in turn must be better than losses. Otherwise, an agent using the evaluation function might err even if it can see ahead all the way to the end of the game. Second, the computation must not take too long! (The whole point is to search faster.) Third, for nonterminal states, the evaluation function should be strongly correlated with the actual chances of winning.