Cutting off search

The most straightforward approach to controlling the amount of search is to set a fixed depth limit, so that the cutoff test succeeds for all nodes at or below depth *d*. The depth is chosen so that the amount of time used will not exceed what the rules of the game allow. A slightly more robust approach is to apply iterative deepening, as defined in Chapter 3. When time runs out, the program returns the move selected by the deepest completed search.

These approaches can have some disastrous consequences because of the approximate nature of the evaluation function. Consider again the simple evaluation function for chess based on material advantage. Suppose the program searches to the depth limit, reaching the position shown in Figure 5.4(d). According to the material function, white is ahead by a knight and therefore almost certain to win. However, because it is black's move, white's queen is lost because the black knight can capture it without any recompense for white. Thus, in reality the position is won for black, but this can only be seen by looking ahead one more ply.

QUIESCENT

Obviously, a more sophisticated cutoff test is needed. The evaluation function should only be applied to positions that are **quiescent**, that is, unlikely to exhibit wild swings in value in the near future. In chess, for example, positions in which favorable captures can be made are not quiescent for an evaluation function that just counts material. Nonquiescent positions can be expanded further until quiescent positions are reached. This extra search is called a **quiescence search**; sometimes it is restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

QUIESCENCE SEARCH

HORIZON PROBLEM

The **horizon problem** is more difficult to eliminate. It arises when the program is facing a move by the opponent that causes serious damage and is ultimately unavoidable. Consider the chess game in Figure 5.5. Black is slightly ahead in material, but if white can advance its pawn from the seventh row to the eighth, it will become a queen and be an easy win for white. Black can forestall this for a dozen or so ply by checking white with the rook, but inevitably the pawn will become a queen. The problem with fixed-depth search is that it believes that these stalling moves have avoided the queening move—we say that the stalling moves push the inevitable queening move "over the horizon" to a place where it cannot be detected. At present, no general solution has been found for the horizon problem.

5.4 ALPHA-BETA PRUNING

Let us assume we have implemented a minimax search with a reasonable evaluation function for chess, and a reasonable cutoff test with a quiescence search. With a well-written program on an ordinary computer, one can probably search about 1000 positions a second. How well will our program play? In tournament chess, one gets about 150 seconds per move, so we can look at 150,000 positions. In chess, the branching factor is about 35, so our program will be able to look ahead only three or four ply, and will play at the level of a complete novice! Even average human players can make plans six or eight ply ahead, so our program will be easily fooled.

Fortunately, it is possible to compute the correct minimax decision without looking at every node in the search tree. The process of eliminating a branch of the search tree from consideration

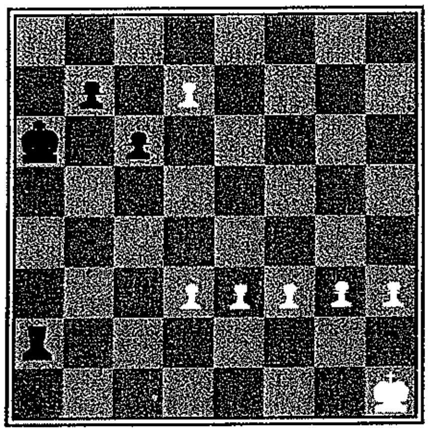

Figure 5.5 The horizon problem. A series of checks by the black rook forces the inevitable queening move by white "over the horizon" and makes this position look like a slight advantage for black, when it is really a sure win for white.

PRUNING

ALPHA-BETA PRUNING

without examining it is called **pruning** the search tree. The particular technique we will examine is called **alpha-beta pruning**. When applied to a standard minimax tree, it returns the same move as minimax would, but prunes away branches that cannot possibly influence the final decision.

Consider the two-ply game tree from Figure 5.2, shown again in Figure 5.6. The search proceeds as before: A_1, then A_{11}, A_{12}, A_{13}, and the node under A_1 gets minimax value 3. Now we follow A_2, and A_{21}, which has value 2. At this point, we realize that if MAX plays A_2, MIN has the option of reaching a position worth 2, and some other options besides. Therefore, we can say already that move A_2 is worth *at most* 2 to MAX. Because we already know that move A_1 is worth 3, there is no point at looking further under A_2. In other words, we can prune the search tree at this point and be confident that the pruning will have no effect on the outcome.

The general principle is this. Consider a node n somewhere in the tree (see Figure 5.7), such that Player has a choice of moving to that node. If Player has a better choice m either at the parent node of n, or at any choice point further up, then *n will never be reached in actual play.* So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

Remember that minimax search is depth-first, so at any one time we just have to consider the nodes along a single path in the tree. Let α be the value of the best choice we have found so far at any choice point along the path for MAX, and β be the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN. Alpha-beta search updates the value of α and β as it goes along, and prunes a subtree (i.e., terminates the recursive call) as soon as it is known to be worse than the current α or β value.

The algorithm description in Figure 5.8 is divided into a MAX-VALUE function and a MIN-VALUE function. These apply to MAX nodes and MIN nodes, respectively, but each does the same thing: return the minimax value of the node, except for nodes that are to be pruned (in

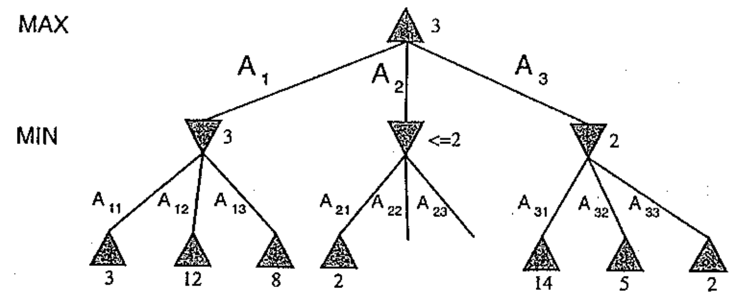

Figure 5.6 The two-ply game tree as generated by alpha-beta.

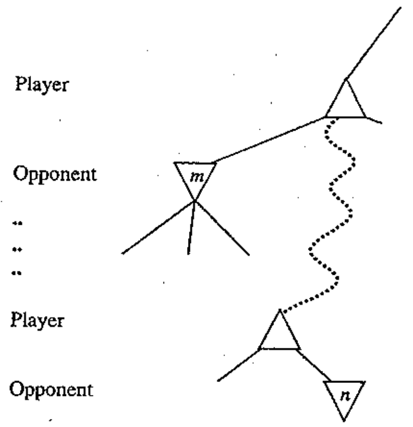

Figure 5.7 Alpha-beta pruning: the general case. If m is better than n for Player, we will never get to n in play.

which case the returned value is ignored anyway). The alpha-beta search function itself is just a copy of the MAX-VALUE function with extra code to remember and return the best move found.

Effectiveness of alpha-beta pruning

The effectiveness of alpha-beta depends on the ordering in which the successors are examined. This is clear from Figure 5.6, where we could not prune A_3 at all because A_{31} and A_{32} (the worst moves from the point of view of MIN) were generated first. This suggests it might be worthwhile to try to examine first the successors that are likely to be best.

If we assume that this can be done,[3] then it turns out that alpha-beta only needs to examine $O(b^{d/2})$ nodes to pick the best move, instead of $O(b^d)$ with minimax. This means that the effective branching factor is $\sqrt{b}$ instead of b—for chess, 6 instead of 35. Put another way, this means

[3] Obviously, it cannot be done perfectly, otherwise the ordering function could be used to play a perfect game!

```
function MAX-VALUE(state, game, $\alpha$, $\beta$) returns the minimax value of state
  inputs: state, current state in game
          game, game description
          $\alpha$, the best score for MAX along the path to state
          $\beta$, the best score for MIN along the path to state

  if CUTOFF-TEST(state) then return EVAL(state)
  for each s in SUCCESSORS(state) do
      $\alpha \leftarrow$ MAX($\alpha$, MIN-VALUE(s, game, $\alpha$, $\beta$))
      if $\alpha \geq \beta$ then return $\beta$
  end
  return $\alpha$
```

```
function MIN-VALUE(state, game, $\alpha$, $\beta$) returns the minimax value of state

  if CUTOFF-TEST(state) then return EVAL(state)
  for each s in SUCCESSORS(state) do
      $\beta \leftarrow$ MIN($\beta$, MAX-VALUE(s, game, $\alpha$, $\beta$))
      if $\beta \leq \alpha$ then return $\alpha$
  end
  return $\beta$
```

Figure 5.8 The alpha-beta search algorithm. It does the same computation as a normal minimax, but prunes the search tree.

that alpha-beta can look ahead twice as far as minimax for the same cost. Thus, by generating 150,000 nodes in the time allotment, a program can look ahead eight ply instead of four. By thinking carefully about *which computations actually affect the decision*, we are able to transform a program from a novice into an expert.

The effectiveness of alpha-beta pruning was first analyzed in depth by Knuth and Moore (1975). As well as the best case described in the previous paragraph, they analyzed the case in which successors are ordered randomly. It turns out that the asymptotic complexity is $O((b/\log b)^d)$, which seems rather dismal because the effective branching factor $b/\log b$ is not much less than b itself. On the other hand, the asymptotic formula is only accurate for $b > 1000$ or so—in other words, not for any games we can reasonably play using these techniques. For reasonable b, the total number of nodes examined will be roughly $O(b^{3d/4})$. In practice, a fairly simple ordering function (such as trying captures first, then threats, then forward moves, then backward moves) gets you fairly close to the best-case result rather than the random result. Another popular approach is to do an iterative deepening search, and use the backed-up values from one iteration to determine the ordering of successors in the next iteration.

TREE MODEL

It is also worth noting that all complexity results on games (and, in fact, on search problems in general) have to assume an idealized **tree model** in order to obtain their results. For example, the model used for the alpha-beta result in the previous paragraph assumes that all nodes have the same branching factor b; that all paths reach the fixed depth limit d; and that the leaf evaluations are randomly distributed across the last layer of the tree. This last assumption is seriously flawed: for example, if a move higher up the tree is a disastrous blunder, then most of its descendants will look bad for the player who made the blunder. The value of a node is therefore likely to be highly correlated with the values of its siblings. The amount of correlation depends very much on the particular game and indeed the particular position at the root. Hence, there is an unavoidable component of *empirical science* involved in game-playing research, eluding the power of mathematical analysis.

5.5 Games That Include an Element of Chance

In real life, unlike chess, there are many unpredictable external events that put us into unforeseen situations. Many games mirror this unpredictability by including a random element such as throwing dice. In this way, they take us a step nearer reality, and it is worthwhile to see how this affects the decision-making process.

Backgammon is a typical game that combines luck and skill. Dice are rolled at the beginning of a player's turn to determine the set of legal moves that is available to the player. In the backgammon position of Figure 5.9, white has rolled a 6-5, and has four possible moves.

Although white knows what his or her own legal moves are, white does not know what black is going to roll, and thus does not know what black's legal moves will be. That means white cannot construct a complete game tree of the sort we saw in chess and Tic-Tac-Toe. A game tree in backgammon must include **chance nodes** in addition to MAX and MIN nodes. Chance nodes are shown as circles in Figure 5.10. The branches leading from each chance node denote the possible dice rolls, and each is labelled with the roll and the chance that it will occur. There are 36 ways to roll two dice, each equally likely; but because a 6-5 is the same as a 5-6, there are only 21 distinct rolls. The six doubles (1-1 through 6-6) have a 1/36 chance of coming up, the other fifteen distinct rolls a 1/18 chance.

CHANCE NODES

The next step is to understand how to make correct decisions. Obviously, we still want to pick the move from $A_1, \ldots, A_n$ that leads to the best position. However, each of the possible positions no longer has a definite minimax value (which in deterministic games was the utility of the leaf reached by best play). Instead, we can only calculate an average or **expected value**, where the average is taken over all the possible dice rolls that could occur.

EXPECTED VALUE

It is straightforward to calculate expected values of nodes. For terminal nodes, we use the utility function, just like in deterministic games. Going one step up in the search tree, we hit a chance node. In Figure 5.10, the chance nodes are circles; we will consider the one labelled C. Let d_i be a possible dice roll, and $P(d_i)$ be the chance or probability of obtaining that roll. For each dice roll, we calculate the utility of the best move for MIN, and then add up the utilities, weighted by the chance that the particular dice roll is obtained. If we let $S(C, d_i)$ denote the set of positions generated by applying the legal moves for dice roll $P(d_i)$ to the position at C, then we can calculate the so-called **expectimax value** of C using the formula

EXPECTIMAX VALUE

$$expectimax(C) = \sum_i P(d_i) \max_{s \in S(C,d_i)}(utility(s))$$