The next step is to understand how to make correct decisions. Obviously, we still want to pick the move that leads to the best position. However, positions do not have definite minimax values; instead, we can only calculate the expected value of a position: the average over all possible outcomes of the chance nodes.

This leads us to generalize the minimax value for deterministic games to an expected minimax value for games with chance nodes. Terminal nodes and MAX and MIN nodes (for which the dice roll is known) work exactly the same way as before. For chance nodes we compute the expected value, which is the sum of the value over all outcomes, weighted by the probability of each chance action:

\[
\text{EXPECTIMINIMAX}(s) = \begin{cases} 
\text{UTILITY}(s) & \text{if TERMINAL-TEST}(s), \\
\max_a \text{EXPECTIMINIMAX RESULT}(s, a) & \text{if PLAYER}(s) = \text{MAX}, \\
\min_a \text{EXPECTIMINIMAX RESULT}(s, a) & \text{if PLAYER}(s) = \text{MIN}, \\
\sum_r P(r) \text{EXPECTIMINIMAX RESULT}(s, r) & \text{if PLAYER}(s) = \text{CHANCE} 
\end{cases}
\]

where \( r \) represents a possible dice roll (or other chance event) and RESULT\((s, r)\) is the same state as \( s \) with the additional fact that the result of the dice roll is \( r \).

5.5.1 Evaluation functions for games of chance

As with minimax, the obvious approximation to make with expectiminimax is to cut the search off at some point and apply an evaluation function to each leaf. One might think that evaluation functions for games such as backgammon should be just like evaluation functions...
for chess—they just need to give higher scores to better positions. But in fact, the presence of chance nodes means that one has to be more careful about what the evaluation values mean. Figure 5.12 shows what happens: with an evaluation function that assigns the values \([1, 2, 3, 4]\) to the leaves, move \(a_1\) is best; with values \([1, 20, 30, 400]\), move \(a_2\) is best. Hence, the program behaves totally differently if we make a change in the scale of some evaluation values! It turns out that to avoid this sensitivity, the evaluation function must be a positive linear transformation of the probability of winning from a position (or, more generally, of the expected utility of the position). This is an important and general property of situations in which uncertainty is involved, and we discuss it further in Chapter 16.

![Figure 5.12](image)

**Figure 5.12** An order-preserving transformation on leaf values changes the best move.

If the program knew in advance all the dice rolls that would occur for the rest of the game, solving a game with dice would be just like solving a game without dice, which minimax does in \(O(b^m)\) time, where \(b\) is the branching factor and \(m\) is the maximum depth of the game tree. Because expectiminimax is also considering all the possible dice-roll sequences, it will take \(O(b^{m+1})\), where \(n\) is the number of distinct rolls.

Even if the search depth is limited to some small depth \(d\), the extra cost compared with that of minimax makes it unrealistic to consider looking ahead very far in most games of chance. In backgammon \(n\) is 21 and \(b\) is usually around 20, but in some situations can be as high as 4000 for dice rolls that are doubles. Three plies is probably all we could manage.

Another way to think about the problem is this: the advantage of alpha–beta is that it ignores future developments that just are not going to happen, given best play. Thus, it concentrates on likely occurrences. In games with dice, there are no likely sequences of moves, because for those moves to take place, the dice would first have to come out the right way to make them legal. This is a general problem whenever uncertainty enters the picture: the possibilities are multiplied enormously, and forming detailed plans of action becomes pointless because the world probably will not play along.

It may have occurred to you that something like alpha–beta pruning could be applied