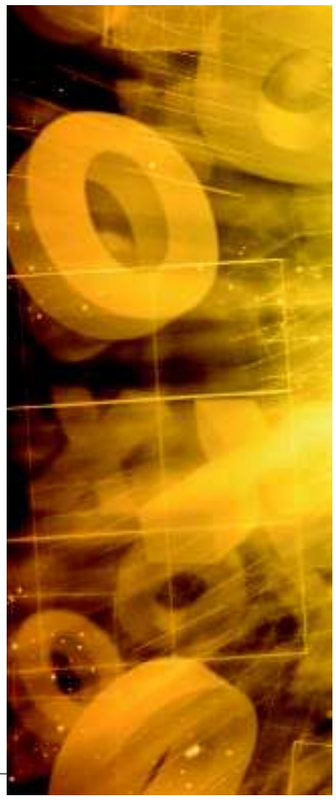
Behind the minimax algorithm



Ever fancied programming two-player zero-sum games? Here's everything you need to know

One of the most interesting avenues of computer science is that of programming a computer to play a game against a human opponent. Examples abound, with the most famous that of programming a computer to play chess. But no matter what the game is, the programming tends to follow an algorithm called minimax, with various attendant sub-algorithms in tow.

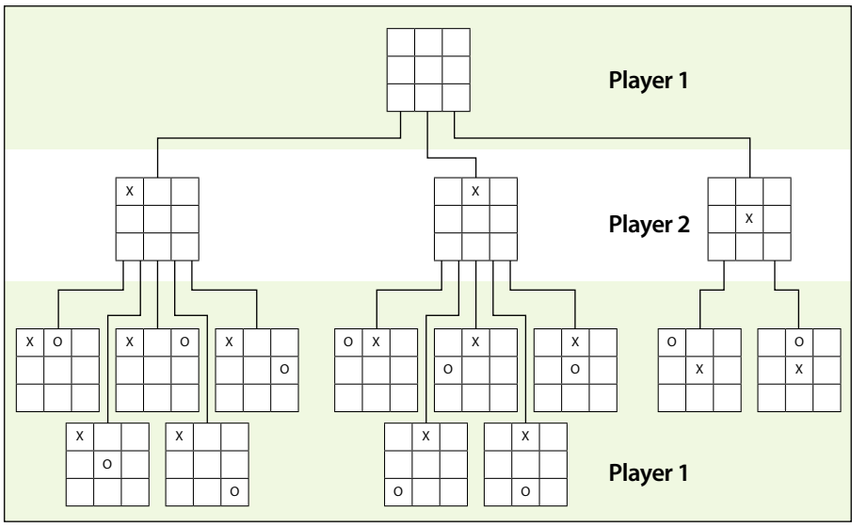
First, a definition: a two-player zero-sum game is one played between two players where the players play alternately, the whole game is visible to both and there's a winner and a loser (or there's a draw). It's zero-sum because if the game is played for money, the loser pays the winner and overall there's no loss of money. (A bit like energy in a reaction: no money is created or destroyed.)

One of the simplest two-player zero-sum games is noughts and crosses, where the players alternately place Xs and Os in a 3 x 3 grid, with the winner being the first player to place three of their symbol in a row, column or diagonal line. Like me, you probably played this as a child and, as you played it, you learned how to force a win or draw every time. In fact, once both players get that insight, every game is guaranteed to result in a draw. The only way to win is to play a novice player.

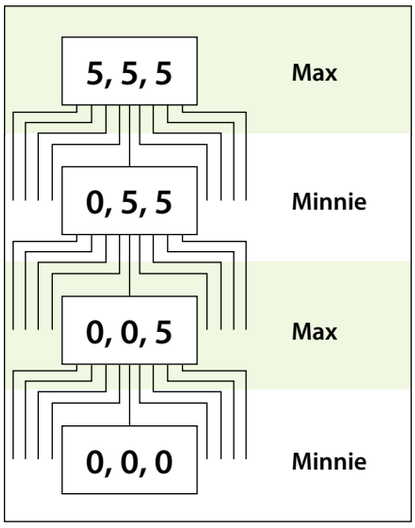
The algorithm

Analysing noughts and crosses with the minimax algorithm is pretty standard in game theory, so I'll discuss a different game called Nim to illustrate minimax and its variants. Nim is interesting because it's easily understood, fairly unfamiliar and simply modelled. Plus, there are no draws in Nim, so the whole winner/loser thing is much simpler: someone always wins. But who?

In Nim, the players face three piles of stones with, say, five stones in each pile. Each player ▸



▲ **Figure 1: The first few levels in the noughts and crosses game tree.**



▲ **Figure 2: An allowable but idiotic game play for Nim, resulting in Max losing.**

takes it in turn to play by removing from a single pile anything from one stone to the entire pile. The loser is the one who is forced to remove the final stone from the final pile, leaving all three piles empty. (Another way of looking at it is that the winner is the first player to be faced with three empty piles.)

For example, suppose our two players are named Max and Minnie. Max starts (he always does, not being a gentleman) and decides to remove all the stones from pile one. Minnie then removes all but two stones from pile two. Max thinks for a while, then removes all but two stones from pile three. Minnie resigns, because no matter what she does, Max will win. (If she removes one stone from a pile, Max removes both stones from the other, and she's left with the final stone. If she removes both stones from a pile, Max removes one stone from the other, leaving her with the final stone.)

Traversing nodes

Games such as Nim are modelled as game trees. You start off with the initial state of the game as a node, the root of the tree. From this node, each possible move is modelled as a link to another node, which stands in for another state or position of the game.

So, for example, in noughts and crosses, the root node is the empty grid. Traditionally X starts and there are three possible moves: the centre, a corner and the middle cell along an edge (all the cells are equivalent to one of those three). So, the initial root node has three links to other game states. Each of those new nodes has different possible moves for O, as shown in Figure 1. You can imagine going further and drawing more levels.

Nim's tree is more complex. The initial state has 15 possible links, corresponding to removing one, two, three, four or five stones from each of the three piles. Each of these 15 possible states of the game then has up to 14 possible links to other states for the second player, and so on. You can imagine that the number of game states (that is, nodes in the game tree) explodes pretty quickly.

If you happened to have a big enough piece of paper, it would be possible to map out the entire game tree for the version of Nim that I described. For the leaf nodes of the tree (that is, the nodes with no links coming out of them), you would be able to identify the loser of the game for the path taken through the tree to each particular leaf. Figure 2 shows a particularly daft path through the tree where the players take all the stones from each pile in turn (not exactly an insightful game, but nevertheless a possible one under the rules). The loser is Max, because he takes all the stones from pile three in the third move.

We can assign a value to each leaf node to indicate who wins (or loses). To make sure we don't get completely confused, we assign a monetary value from the viewpoint of the first player, Max. Let's say the winner of the path to the leaf receives £1 and the loser has to pay out that amount – so if the winner is Max, the value of the node is £1, while if the winner is Minnie, the value is -£1 (since Max has to pay that amount to her).

Player one

Let's imagine that we set up the entire game tree from the viewpoint of Max, the player who makes his move first. Each game position corresponds to a node in the tree, and if you think about it, a whole level of the tree will correspond to a given player. So, the root of the tree is what Max is faced with at the very start of the game: five stones in each of the three piles, and 15 possible game positions to leave for Minnie. What does Max choose to play in this situation?

What he should do is analyse all possible moves from the bottom up and assign a value to each node as he works his way up the tree,

Claude Shannon

In 1950, Claude Shannon published a paper called *Programming a Computer for Playing Chess*, which was the first such paper to consider this particular game tree. In it, he reached an upper bound for the number of nodes in a game tree for chess to be about 10^{120}, which meant, as he put it, that a "machine operating at the rate of one [node] per micro-second would require over 10^{90} years to calculate the first move". Shannon's paper was remarkable because it contained the insight of an evaluation function for the strength of a position, and using it to be able to calculate node values to several levels deep. ■

Spotlight on… Alpha-beta pruning

First proposed by John McCarthy at a conference in 1956 (although only named as such later on), alpha-beta pruning is a method for cutting off whole branches of the game tree so that they don't have to be evaluated with minimax.

In essence, the algorithm maintains two extra values during the minimax recursion: alpha and beta. Alpha is the minimum value for Max (biggest loss for him) and beta is the maximum value for Minnie (biggest win for Max). They start out as negative infinity for alpha and positive infinity for beta. As the minimax recursion proceeds, the value for alpha is replaced when a new minimax value that is larger is found (ditto for beta, when a smaller value is calculated). If they cross at any time, the branch of the tree currently being investigated is no good for either player and can be further ignored, or pruned.

It can be shown that this algorithm doesn't mistakenly prune branches that will benefit either player and so it's widely used in minimax implementations. ■

Expectiminimax

Games such as Nim and chess have outcomes that are solely dependent on the skill of the players, or, presumably, their access to a well-written minimax analyser. Games such as backgammon are different, because their outcomes also depend on a randomisation factor such as the roll of a dice. The minimax algorithm has been expanded to suit such games (leading to the expectiminimax algorithm) by including what are known as chance nodes that incorporate the expected value of the randomisation agent (for backgammon, this would be the dice). ■

according to the amount he could win on that node if he played optimally.

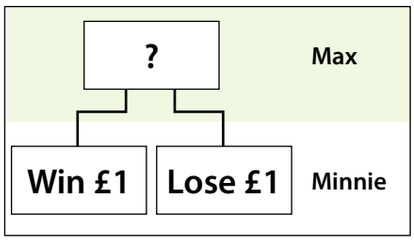
Let's take a look at a made-up example, shown in Figure 3. Here, the root node shows a game position from which Max must play. There are two possibilities: playing the left-hand option goes to a game position that he's already worked out means he wins £1; playing the right-hand option goes to a game position where he loses £1. (Remember, all payouts are from Max's viewpoint.) I don't know about you, but I'd choose the first play. This means that the current game position also has a value of £1. For every game position where it's his turn to play, Max would choose the option that would maximise his winnings.

Minnie, who is just as perceptive as Max, would, of course, choose plays that would result in the best result for her and ignore all the others. So she would always choose a play that maximised her winnings, which, from Max's perspective, means minimising his.

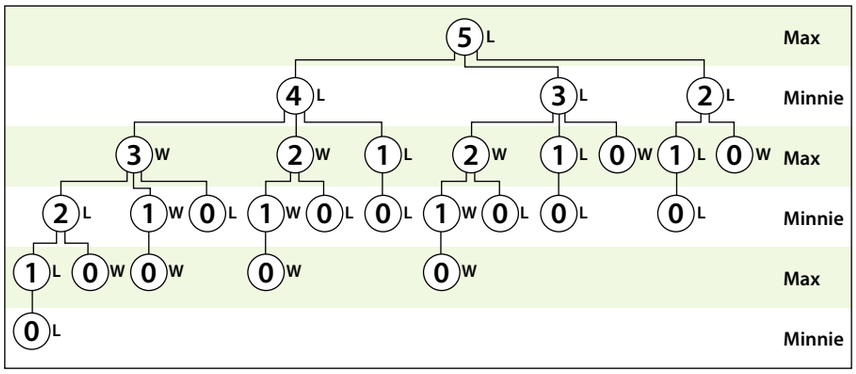
If you had the entire tree, you could work out a value for each node working from the bottom up. If it was a 'Max node' (that is, Max had to play from it), it would have a value that was the maximum of the child nodes. If it was a 'Minnie node' it would have a value that was the smallest (the minimum) of the child nodes. This, in essence, is the minimax algorithm: build the tree, work out the value of each node using an alternate minimise/maximise constraint, and the value of the root is the value of the entire game for player one (Max, as we called him).

The recursive method

Instead of building the entire tree and then analysing it, the best approach is to traverse the tree recursively (a postfix traversal, in fact) and calculate what you need when you need it



▲ **Figure 3: A simple choice in a game tree, to calculate the minimax value of the root node.**



▲ **Figure 4: The complete game tree for a simplified Nim game.**

(and destroy the stuff you don't need when you're done). In essence, since a tree is defined recursively, you calculate the minimax value by calculating the maximum (or minimum) of the minimaxes of all the child trees. Remember that the levels alternate between maximising and minimising (sometimes you look at it from Minnie's viewpoint instead of Max's).

Figure 4 shows a very simplified Nim game (one pile of five stones, you can remove one, two or three stones each play), fully expanded into a game tree. The number inside each node is the number of stones left in the pile after the move, and the letter alongside each node is the minimax value for Max (W = win, L = lose). Note that the value of the game is L – that is, Max will always lose (if you like, this simplified Nim is always a win for the second player).

Although the minimax algorithm is always guaranteed to find the best play for Max, there is a big problem. The game tree can be huge – mind-bogglingly huge. Consider chess, the classic archetype of a two-player zero-sum game. At each game position there could be something like 30 possible moves. Since each chess game is made up of about 80 plays (40 back-and-forths), it would mean that the lowest level of the tree would have something like 10^{118} nodes. (Note that in tournaments it's rare for a game to go to checkmate – the losing player is likely to resign well before then.) As a comparison, there are around 10^{80} atoms in the observable universe, meaning that, in essence, there's no possible way for a computer to map the entire chess game tree. So what can we do?

The first optimisation is to limit the depth to which we evaluate the game tree using the minimax algorithm. Since we may not actually reach a leaf node in doing this, we make use of an approximation function – a heuristic – to approximate the value of the node or game position. Of necessity, this value is not going to be accurate, but it will enable us to apply the minimax algorithm without having to evaluate all the nodes down to the leaves. The better the heuristic, the better the chances of devising a winning game play and the more accurate our minimax values will be.

Chess programs

Computer chess is one of the most fertile avenues of game research. Modern chess programs running on standard hardware use sophisticated pruning techniques to cull unprofitable areas of the game tree, use advanced heuristics to evaluate a game position and have large databases of standard opening games (so that they don't have to analyse an opening game at all) and common endgames (so that they can more easily force checkmates without having to analyse a multitude of game positions towards the end of a game). ■

Limiting depth

In our recursive algorithm for minimax, we'll need to limit the depth of the recursion instead of allowing the recursion to reach the leaves. The simplest way to do this is to pass a depth parameter to the recursive minimax function and decrement its value at every recursive call. At the lowest level of the recursion, we use the heuristic function to calculate the minimax value of the current game position.

Now, the resulting minimax value at the root of the game tree is only going to be an approximation. The deeper we allow the partial minimax algorithm to go, the more accurate its value will be (because we're more likely to find leaf nodes in our traversal), but the longer the traversal will take. We have to strike a balance between accuracy and the time taken to calculate the minimax value (and hence the move to play).

Once it's our turn again to make a move, we should recalculate the minimax value at our new game position, making it, in effect, the root of the current state of the game. Every move would be made after a new minimax calculation based on the current game state.

In many chess programs that run on standard PC hardware, the depth of the minimax search is limited to some six full-width levels – around a billion possible game positions. Any more than that and the time taken to analyse the game positions would be far too long to be practical. For example, analysing positions at a rate of a million per second, six full-width levels would take about a quarter of an hour. ■

Julian M Bucknall has worked for companies ranging from TurboPower to Microsoft and is now CTO for Developer Express.
feedback@pclus.co.uk