5	2	6	3	Initial arrangement
1	2	6	3	**A**oife takes four from pile one
1	2	6	0	**B**rendan takes three from pile four
1	2	1	0	**A**oife takes five from pile three
1	1	1	0	**B**rendan takes one from pile two
1	0	1	0	**A**oife takes one from pile two
1	0	0	0	**B**rendan takes one from pile two
0	0	0	0	**A**oife takes one from pile one and wins

Table 1: An example of two players playing nim; this illustrates game play, neither player is playing optimally. The four columns give the number of counters in the four piles directly after the move described in the same row.

15 - adversarial search

Here we will consider adversarial search: algorithms designed to allow a computer to develop an optimal strategy for an adversarial game, or a process that can be rephrased as a game. In particular we will consider the minimax algorithm, a naïve AI which has been useful; it is naïve in the sense that it does not perform sophisticated pattern recognition using modern biometic neural networks like deep-learning networks, instead it relies on the brutish computational power of a computer to 'play ahead' and see the consequence of different moves. With tweaks, this is the algorithm that allowed Deep Blue to beat Garry Kasparov, the world chess champion, in 1997, realizing a proposal, originally due to Claude Shannon, that computers can play chess. It is not the algorithm that allowed AlphaGo to beat Lee Sidol, a 9th Dan Go champion, in 2016, that was a deep learning network.

Roughly speaking we are thinking about a game without chance where two players alternate in taking a move which changes the game state. In chess the game state is the position of the pieces on the board, in go the position of the stones, in tic-tac-toe the number and position of the Xs and Os on the grid. In adverserial search a tree is constructed with the current state of the game as the root: the nodes will be game states and the different edges from a node will correspond to different possible moves. At first the leaves, the terminal nodes, correspond to final game states, which will be a win for one or other player or a draw. The overall strategy is to score the leaves and to propagate those scores upwards under the assumption that each player plays optimally. It is easiest to describe this through an example.

Nim example

Nim is a simple game using counters for two players; the counters are divided into some number of piles and the players take turns removing counters. In their turn a player must take at least one counter and can take as many counters as they like, provided they only come from one pile. Thus, in each turn each player removes counters from only one pile. The winner is the player who takes the last piece. An example game is given in Table 1

Nim can be enjoyable to play, but it is mostly beloved of mathematicians and computer scientists because it is a good example to use in explanations like this one and because it was solved as a game, for all numbers of piles and counters, in 1902 by Charles L. Bouton, in a paper which is considered to have founded combinatorial game theory. What we are interested in here stops short of solving the game analytically, instead in Fig. 1 we map out a tree giving

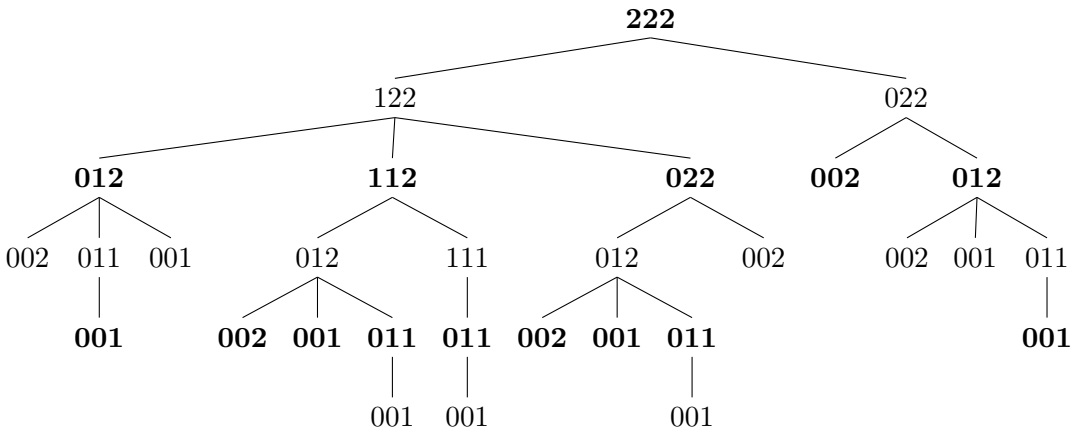


Figure 1: The whole game tree for nim starting with $(2, 2, 2)$; some equivalent choices have been left out, so it includes $(1, 2, 2)$ on the second ply, but not $(2, 1, 2)$; for clarity the triples are arranges in ascending order. Alternative ply are bolded to help keep track of who is playing, Aoife is bold, whereas her opponent is unbolded, so for a bolded node Aoife takes the next move.

all possible moves for nim with three piles of two. Thus, from the initial configuration $(2, 2, 2)$ leaving out alternatives that are equivalent to each other, there are two possible moves, the player can take one or two counters from one of the piles, leading to $(1, 2, 2)$ or $(0, 2, 2)$. From $(1, 2, 2)$ there are then three alternatives, leading to $(0, 1, 2)$, $(1, 1, 2)$ and $(0, 2, 2)$, and so on.

Now the leaves of the tree all have two zeros, whoever has the next move wins. This means we can score the leaves; for definiteness let's imagine Aoife has the first go so if this is to be an AI for a computer then Aoife is the computer and the purpose of this calculation is to choose which move Aoife should make at $(2, 2, 2)$ to win. Obviously the different levels in the tree alternate between the two players, so the level of the root corresponds to Aoife moving, the next level, with $(1, 2, 2)$ and $(0, 2, 2)$ to Brendan and so on. The different levels are sometimes called *ply* in this context.

We will score +1 for leaves where Aoife has the next go and is thus able to win, -1 if Brendan does. This is shown in Fig. 2; the problem is that the leaves are far removed from the current move, Aoife is at $(2, 2, 2)$ and decided between $(1, 2, 2)$ and $(0, 2, 2)$; we need to move the scoring up from the leaves to the root.

In Fig. 3 the scoring has been moved upwards in a simple way, since the only legal moves from $(0, 1, 1)$ lead to $(0, 0, 1)$ or equivalently $(0, 1, 0)$ this means $(0, 1, 1)$ leads to the same result as $(0, 0, 1)$ and so the score from the leaf is moved up one.

However this reasoning isn't enough if a node has more than one leaf, as for example is the case for 012:a in Fig. 3. Here minimax search uses the assumption that each player is playing optimally, so Aoife makes the choice that maximizes the score, and Brendan the choice that minimizes it. In the case of 012:a, it is Brendan's turn so he will choose the move that will give the lowest of the three scores on offer, that is the one that gives -1, hence this node is also worth -1; similarly for **012**:b it is Aoife's go so she will choose the move that gives +1. This assumption allows the scores to be propagated upwards, with each time the parent inheriting either the lowest or highest of the scores of its children, depending on whose go it corresponds

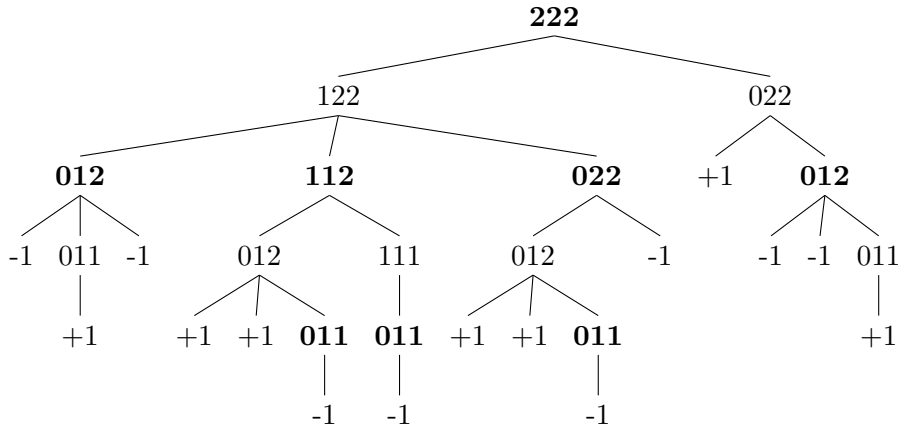


Figure 2: Here the leaves have been scored, +1 for a win for Aoife, -1 if Brendan wins. Since each player takes just one move in each turn, it is easy to score the leaves, the bold ones go to +1, the unbolded to -1.

to.

The first step for this process is shown in Fig. 4; notices that in the case of **112**:a the two children each have score -1 so although this is Aoife's turn, she is forced to choose a losing move and **112**:a is scored -1. Finally in Fig. 5 the scores have propagated to the top and we see that if Aoife takes the right option, $(0, 2, 2)$, she will win no matter what Brendan does, whereas if she takes the right option Brendan can win if he plays optimally.

Scoring

Obviously one disadvantage of adverserial search is that it ignores any psychology. Another obvious example game is tic-tac-toe, tic-tac-toe is a silly game because, as we all know, optimal play always leads to a draw and so adversial search will propagate a value of 0 to the root; however, if for example playing against a child, it might be worth playing to win on the assumption your opponent won't play optimally; this would involve guessing which strategy is most likely to trick them. However a bigger issue is the combinatorial explosion implicit in most games; for (2,2,2) there is only a modest number of valid moves at each level, a larger number of counters and piles would obviously expand this, however this is still modest compared to the number of choices for a game like chess or go.

In a chess game there are typically 37 available moves and a typical chess game has 100 moves in total, clearly this makes exploring the game to the end infeasible. In go the situation is even more extreme; the number of possible moves is usually more than 100 and a game usually involves more turns than chess. This creates a horizon, a distance down the tree beyond which it is infeasible, given computer speed and game tempo, to search which means that the leaves, the terminal nodes, often don't represent a final resolution of the game and so they can't be attributed a score to represent win, loss or draw.

The way out of this is to score the leaves using a heuristic, some machine-calculable number describing how good a given game configuration is. In the case of chess, for example, Shannon's original proposal was that each player was given a score based on their remaining pieces, one point for each pawn, three for each knight or bishop, five for a rook and nine for the queen,

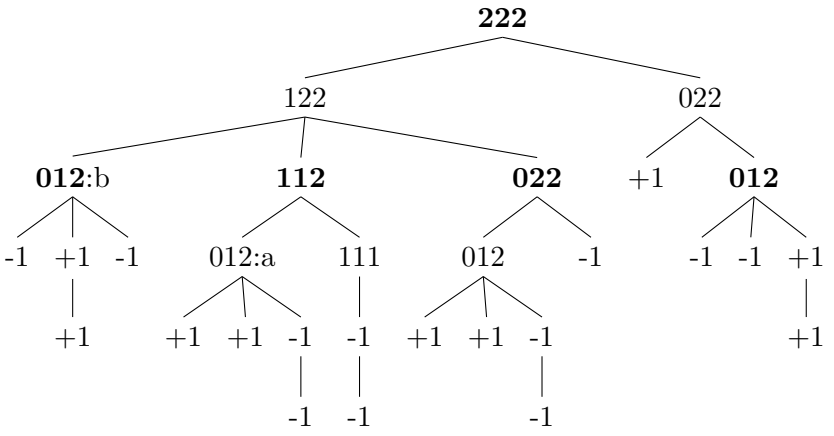


Figure 3: Here the score of the leaves with no siblings have been moved up to their parent. Two of the nodes have been labelled with letters 012:a and **012**:b, these letters are just to help refer to the nodes in the text and have no meaning in the context of the game.

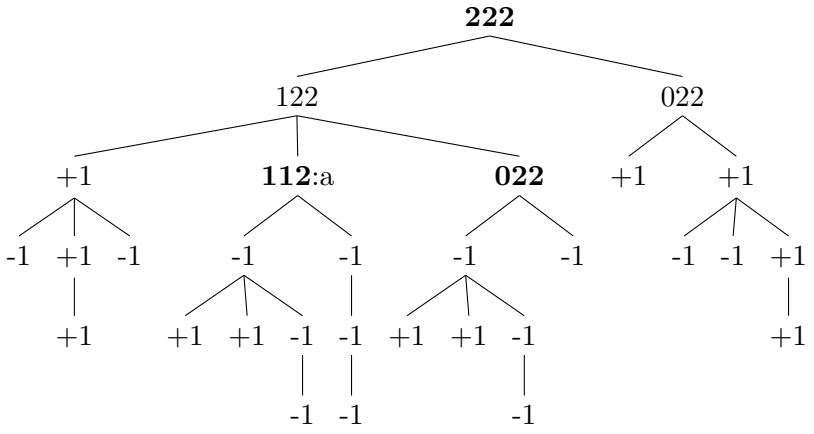


Figure 4: Some of the scores have been propogated upwards, this process now propagates until the current decision point is reached. Again, the label **112**:a has no meaning in the game context but is there to help discussion in the text.

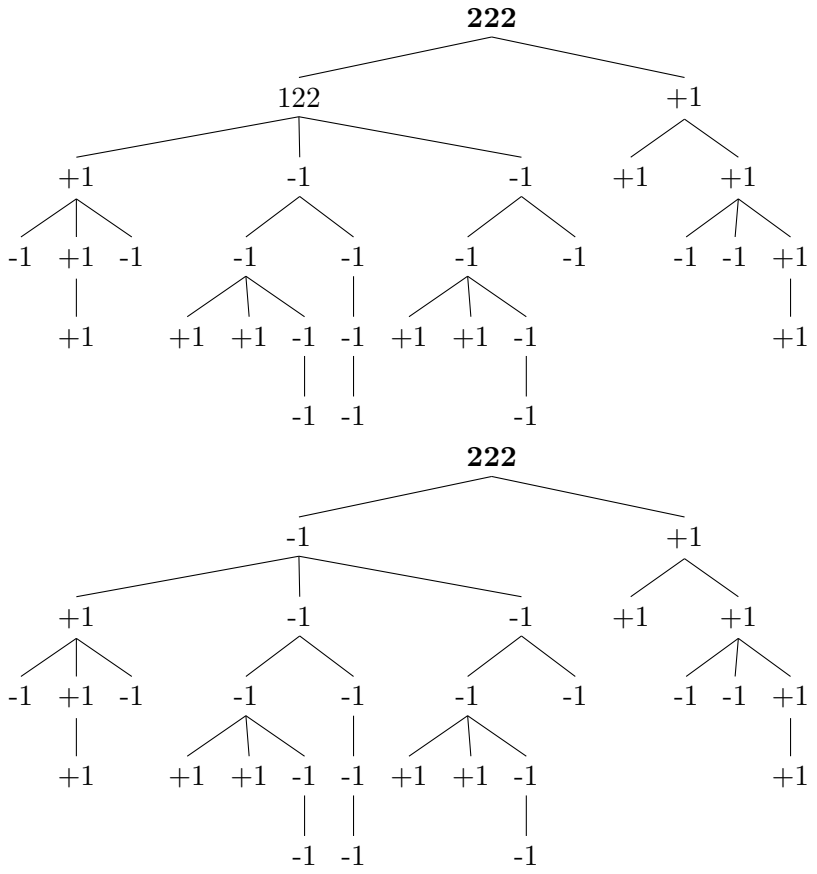


Figure 5: The scores are propagated to the root showing that Aoife can either force a win or risk losing.

on their position, so for example there is a half point taken off for each doubled pawn, that is one pawn behind another and on their mobility, with 0.1 points for each available move. In addition a checkmate had a value of 200 to trump all other considerations. The score for the position is the difference between the two players. Now, in this scheme, each leaf is evaluated using the heuristic and the score is propagated backwards under the assumption that each player plays optimally.

A sketch of a program implementing minimax is given in Table 2.

In practice a poor chess player looks forward four ply; a great one eight but a great player also has more global ideas and intuitions about the game, basically their heuristic is better. The computer that beat Kasparov could look forward roughly 12 ply, the extra four ply allowed it to beat the human; however, Deep Blue also did pruning, it didn't look down every branch. This is what we discuss next.

Alpha-beta pruning

In practice this scheme is supplemented by lots of tricks, for a start different scoring schemes could be experimented on, a 'book' of well known opening sequences and endgames could stored; even in the mid-game a playbook of zingers could help with common plays. Often 'interesting' branches are explored deeper. Another, more concrete, improvement is provided by pruning. This is easy to understand given an example; first consider the simple example given by the portion of the nim tree given in Fig. 6; imagine we are trying to score the $(1, 2, 2)$ node as part of a larger evaluation. In Fig. 6B we see that after the left-most branch has been scored we have a score of -1, since here Brendan is to play at $(1, 2, 2)$ and -1 is the best he can do, we don't need to evaluate the other two branches, they cannot beat the -1 we already have. Of course, this was partly a matter of luck, if we had done the rightmost branch first it would have given a $+1$, and at least one more branch needs to have been evaluated.

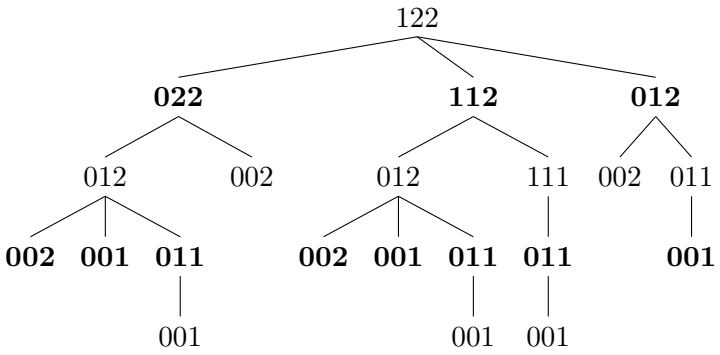
Alpha-beta pruning is a kind of second order version of pruning. Consider the tree in Fig. 7; in this tree the aim is to get the score for the node labelled 'unknown score a', in this example this node is the maximizing node. So far the left sub-branch has been scored to give 20 but the right branch hasn't been finished, in fact the first, leftmost, branch of the right branch has been scored to give 10; now 'unknown score b' is a minimizing node, so although most of its children haven't been scored we know that it will score no more than 10 because that score is already available. Since 'unknown score a' is a maximimizing node it will never choose 'unknown score b' since 20 is bigger than 10. For this reason the other unscored children of 'unknown score b' don't need to be evaluated and this whole branch can be pruned, freeing up more time to explore elsewhere. This is alpha-beta pruning.

Outline code for running minimax with alpha-beta pruning is given in Table 3. This code might not seem completely transparent at first since it is tricky to see how to keep track of when to prune, to see how it works consider running it on the tree in Fig. 7. The root calls `max_value` with `below` set to $-\infty$, or equivalent, and `above` set to ∞. The first node gives 20, so `below` is set to 20; this means `min_value` is called on the right branch with `below` as 20; when the first branch returns 10 this is compared to `below` and since $10 < 20$ this returns 10 and the other nodes are not evaluated.

```
1  int which_child(Node* a)
2  {
3     max_value(a);
4     return a.best_child;
5  }
6
7  float max_value(Node* a)
8  {
9     if(terminal_node(a)) return utility(a);
10
11    int i;
12    float v=big_number,u;
13
14    for(i=0;i<number_of_children(a);i++){
15       u=min_value(get_child(a,i));
16       if(u>v){
17          v=u;
18          a->best_child=i;
19       }
20    }
21    return v;
22 }
23
24 float min_value(Node* a)
25 {
26    if(terminal_node(a)) return utility(a);
27
28    int i;
29    float v=-big_number,u;
30
31    for(i=0;i<number_of_children(a);i++){
32       u=max_value(get_child(a,i));
33       if(u<v){
34          v=u;
35          a->best_child=i;
36       }
37    }
38    return v;
39 }
```

Table 2: Minimax. This code implements the minimax algorithm recursively, the recursion alternates, with `max_value` calling `min_value` and visa versa. It is not full working code in the sense that it relies on a `Node` struct which is not given along with a number of functions: `terminal_node` returns true for a leaf and false otherwise, `utility` returns the score, `number_of_children` returns the number of children and `get_child` returns the `i`th child. It also has `best_child` to keep track of the child with the optimal score.

A)



B)

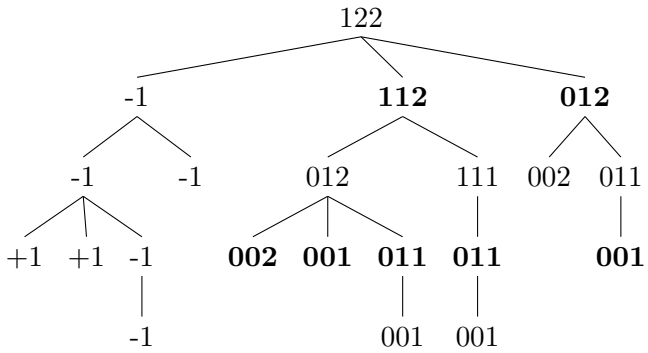


Figure 6: A portion of the nim tree. In A) the whole tree is shown, in B) the left most branch has been score by propagating upwards from the leaves.

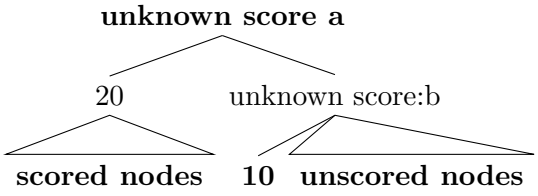


Figure 7: Here the left branch has been fully scored, the right branch has only been partially scored, the left part gives 10, the rest hasn't been considered yet.

```
1  int which_child(Node* a)
2  {
3     max_value(a,-big_number,big_number);
4     return a.best_child;
5  }
6
7  float max_value(Node* a,float below,float above)
8  {
9     if(terminal_node(a)) return utility(a);
10
11    int i;
12    float v=big_number,u;
13
14    for(i=0;i<number_of_children(a);i++){
15       u=min_value(get_child(a,i),below,above);
16       if(u>v){
17          v=u;
18          a->best_child=i;
19       }
20       if(v>=above) return v;
21       if(v>below) below=v;
22    }
23    return v;
24 }
25
26 float min_value(Node* a)
27 {
28    if(terminal_node(a)) return utility(a);
29
30    int i;
31    float v=-big_number,u;
32
33    for(i=0;i<number_of_children(a);i++){
34       u=max_value(get_child(a,i),below,above);
35       if(u<v){
36          v=u;
37          a->best_child=i;
38       }
39       if(v<=below) return v;
40       if(v<above) above=v;
41    }
42    return v;
43 }
```

Table 3: Minimax with alpha-beta pruning. Here `above` and `below` ae used to keep track of pruning values; the clever thing is how the pruning values are pushed down from above.