

Chapter 22

Game Playing

We will have a look at some of the principles governing game playing with computers. The games we are going to discuss are called *two-person, perfect-information* games, such as chess, checkers, and *go*.

In these games, there are two players that make moves in turn, and both of them have the complete information of the current situation of the game. The game is over when a position is reached that qualifies as 'terminal' by the rules of the game, such as a 'mate' in chess.

Represent a game

Such a game can be represented as a *game tree*, in which the nodes represent situations, and the arcs(edges) represent moves, and the initial situation is the root of such a tree, and the terminal situations correspond to the leaves.

We only consider games with two outcomes, our 'win', and our 'loss', and call the two parties 'us' and 'them'.

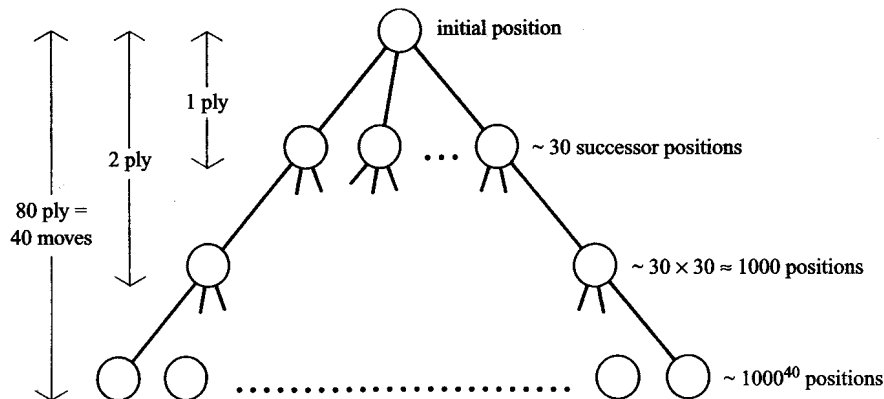
'Us' win in a terminal position when 'them' has not move to make.

'Us' wins a game in a non-terminal 'us-to-move' position if there is a legal move that leads to a win position. On the other hand, a non-terminal 'them-to-move' position will also be a win for us, if all the legal moves from that position lead to a win for us.

Game tree is complicated

Since there are often a large collection of legal moves for each situation, a game tree could be huge.

Below shows such an example.



For the Chinese game, *go*, there are 361 positions in a 19 by 19 board, each of them can have either a white piece, or a black piece, or empty, thus, there are 3^{361} possible situations.

As a result, an efficient search mechanism becomes a necessity for playing any interesting games.

To start with...

Below implements the idea as discussed in the last paragraph.

```
won(Pos):-terminal(Pos).  
won(Pos):-not(terminal(Pos)),move(Pos,Pos1),  
            not(move(Pos1,Pos2),not(won(Pos2))).
```

In the above, by `won(Pos)`, we mean a position in which we will win.

Thus, the last rule says that there is no way, 'them' can move to a new position, `Pos2`, where we will not win. In other words, every move that 'them' can move leads to a win for us.

In terms of formal logic,

$$\begin{aligned} \neg(\text{move}(P_1, P_2) \wedge \neg \text{won}(P_2)) &\equiv \neg \text{move}(P_1, P_2) \vee \text{won}(P_2) \\ &\equiv (\text{move}(P_1, P_2) \rightarrow \text{won}(P_2)) \end{aligned}$$

The above strategy follows a basic DFS approach. Thus, it might lead to infinite loops if a game allows repetitive application of the same rules.

But, practically, this will not happen with the help of additional rules.

For example, in Chess, a *draw*, i.e., *no-win* in our term, will be declared, if a three time repetition occurs.

A more efficient process

It is clear that the smaller the depth of a tree is, the less the computation will be. Thus, one way to do the search is to explore the tree to a certain depth, then evaluate the leaves of this partially developed tree.

The results will then be propagated backwards according to the *minimax principle*. This will then yield to values for all the nodes in the partial tree. The move that leads from the initial position, i.e., the root, to its *most promising* successor is then actually played out, and the process continues from there.

The key is that we will not try to search through the whole game tree, but only part of it, which will save us quite some time, as we will see.

The *minimax* principle

Now, let's check out this principle, which is usually based on an heuristic estimator that estimates the winning chance of one of the players, usually, 'us'.

We call the two players *Max* and *Min* since one of them, *Max*, should maximize the chance of winning, while *Min* wants to win, thus minimizing the chance of winning for the other, *Max*.

More specifically, whenever *Max* is to move, s/he chooses a move that maximize the value; while *Min* will choose a move that minimizes this value, thus reduces *Max*'s chance of wining the game.

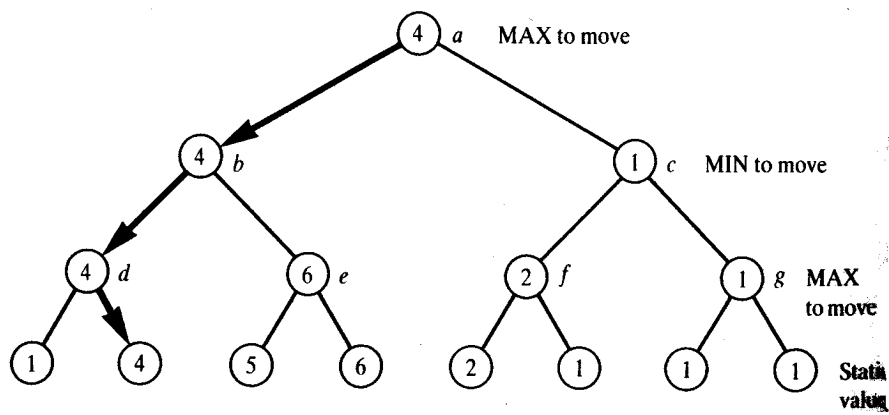
Remember the assumption of *perfect information*, i.e., we know exactly what we are doing, and we assume 'them' is smart also.

More specifically,...

after we are aware of the values of the bottom-level of the current search tree, we can follow this minimax principle to determine the values of all the other positions in the search tree. We can then cut out some of the branches; while continue working with the more promising branches.

The reporting process is carried out bottom up, level by level, until the root level is reached.

Below is an example of this evaluation and report process.



A bit more details

As demonstrated by the above example, the reporting process is carried out from bottom up, level by level, until the root level is reached. Then, we would know the best moves. For example, the best move sequence is a,b,d.

This sequence is also called the *main variation*, which defines the *minimax optimal* play for both sides, but from Max's perspective. Notice the value of nodes along this path is the same.

We also distinguish the values for the bottom level and those obtained during the reporting process. The formers are refereed to as *static*, since they are obtained with a 'static' evaluation process; while the latter are rather *dynamic*.

The propagation process

Let the static value of a position, P , be denoted as $v(P)$; let the backed-up value be $V(P)$, and let P_1, P_2, \dots, P_n be legal successor positions of P . Then, the relation between static values and the backed-up ones can be defined as follows:

1) If P is terminal position, when $v(P)$ is static,

$$V(P) = v(P).$$

2) If P is a Max-to-move position

$$V(P) = \max_i V(P_i).$$

3) If P is a Min-to-move position

$$V(P) = \min_i V(P_i).$$

Possible improvement

The above code systematically visits all the positions in the search tree, down to its terminal positions in a depth-first fashion, and evaluates all the leaves of this tree, then goes back to assign dynamic values to all the positions, until it assigns one for the root.

Usually, not all these work is necessary in order to correctly calculate the minimax values of the root position.

The basic idea is this: between two positions to be further explored, if one position has been demonstrated to be inferior to the other, then it can be immediately dropped, without knowing exactly what its value is.

An example

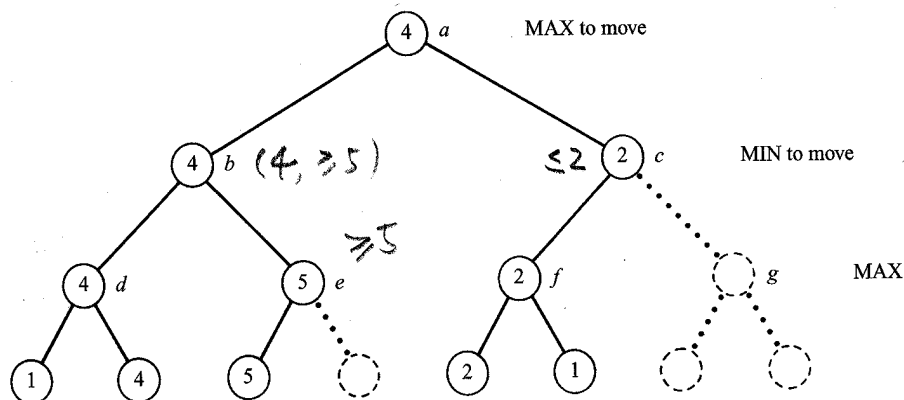
In the following figure, the search tree proceeds as follows: Start with a ; move down to b ; move down to d ; take the maximum of d 's successors yielding $V(d) = 4$; backtracks to b and moves down to e ; Now, since e is a Max position, and we find out that the static value of its first successor is 5, thus, e 's value is at least 5, which is clearly larger than 4, the other successor of b , a Min position. Thus, b clearly will drop the subtree rooted at e .

Thus, it is quite OK for us to assign an approximate value to e , which has no impact at all on the calculation of a dynamic value for b , as well as that for a , since the corresponding tree is simply ignored.

Alpha-beta algorithm

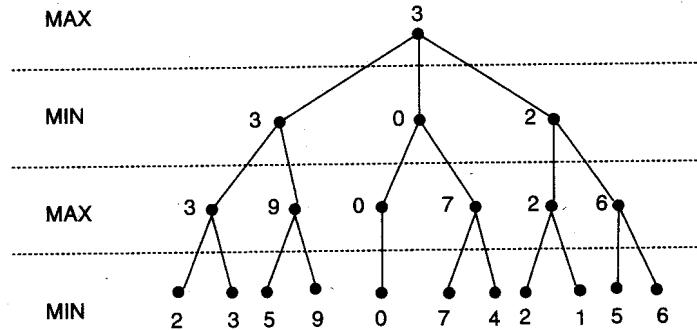
The aforementioned idea is basically what's behind the Alpha-beta algorithm, which is an efficient implementation of the minimax approach.

Below shows the continuing example: once we find out the value of f , we know that the value for c , the only other successor of a will be at most 2, since c is a Min position. Thus, there is no way that the 4 Max has achieved in b can be improved. We now know for sure that the dynamic value for a must be 4. This saves us three static evaluation.

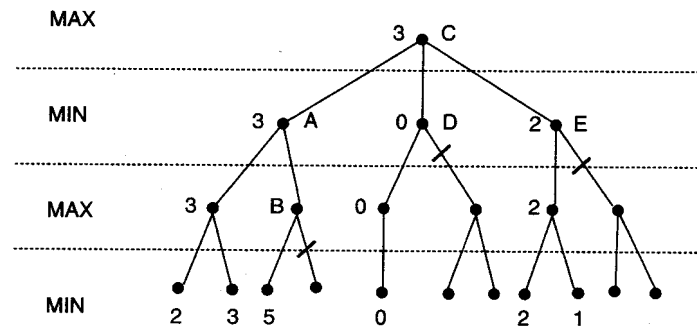


Another example

Below shows the minimax scan.



In the following graph, we find a β cutoff value for A of 3. We then find the α value of B is at least 5, then, by the previous rule 1, we stop the search of the subtree rooted at B . Similarly, since the β value of D is 0, and the α value of its other child is 7; this subtree can also be dropped. The case of dropping the tree at E is the same.



How good is it?

Now, let's compare the $\alpha-\beta$ algorithm with the original minimax search algorithm. It has been proved that, in the best case, when we always search for the strongest position first, we only need to evaluate \sqrt{N} static values, compared to N , which an exhaustive minimax algorithm would have to go through. On the other hand, in the worst case, when we pick up a wrong order of position visiting, then it might end up with the same work as that of the minimax.

Another way to look at it is in terms of branching factor, i.e., number of children, of the search tree. Assume that the factor is b , then, in the best case, the $\alpha-\beta$ algorithm only explores \sqrt{b} successors per node. As an example, in playing chess, this means cut the choice per step from about 30 to about 6.

Application

The *alpha*–*beta* principle provides the basic strategy for many useful two person and full-information game playing algorithms.

§22.5 and §22.6 present a chess playing program based on this principle, coupled with an advice table. When it is them's turn, the program reads in a move; and will select a move according to the advice table, when it is us's turn.