Solving problems by search III: playing games

We now look at how an agent might act when the outcomes of its actions are not known because an adversary is trying to hinder it. We look specifically at the example of *playing games*.

Aims:

- to show how game-playing can be modelled as search;
- to introduce the *minimax* algorithm for game-playing;
- to look at some problems inherent in the use of minimax and to introduce methods for their solution;
- to introduce the concept of $\alpha - \beta$ *pruning*.

Reading: Russell and Norvig, chapter 6.

Playing games: search against an adversary

Something is missing from our existing formulation of problem-solving by search.

- What if we do not know the exact outcome of an action?
- Game playing is a good example: in chess, drafts, and so on an opponent *responds* to our moves.
- We don't know what their response will be, and so the outcome of our moves is not clear.

Game playing has traditionally been of interest in AI because it provides an *idealisation* of a world in which two agents act to *reduce* each other's well-being.

Playing games: search against an adversary

Nonetheless, game playing can be an excellent source of hard problems.

For instance with chess:

- the average branching factor is roughly 35;
- games can reach 50 moves per player;
- so a rough calculation gives the search tree 35^{100} nodes;
- even if only different, legal positions are considered it's about 10^{40}.

Playing games: search against an adversary

As well as dealing with uncertainty due to an opponent:

- we can't make a complete search to find the best move...
- ... so we have to act even though we're not sure about the best thing to do.

It therefore seems that games are a step closer to the complexities inherent in the world around us than are the standard search problems considered so far.

Playing games: search against an adversary

Note:

- "Go" is *much* harder than chess!
- The branching factor is about 360.

If you want to make yourself:

- rich (there's a $2,000,000$ dollar prize if your program can beat a top-level player), and;
- famous (nobody is anywhere near winning the prize);

then you should get to work.

Perfect decisions in a two-person game

Say we have two players, called Maxwell and Minny - Max and Min for short.

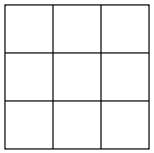
(Yes, there is a reason for this.)

- We'll use noughts and crosses as an initial example.
- Max moves first.
- The players alternate until the game ends.
- At the end of the game, prizes are awarded. (Or punishments administered.)

Perfect decisions in a two-person game

Games like this can be modelled as search problems.

- There is an *initial state*.

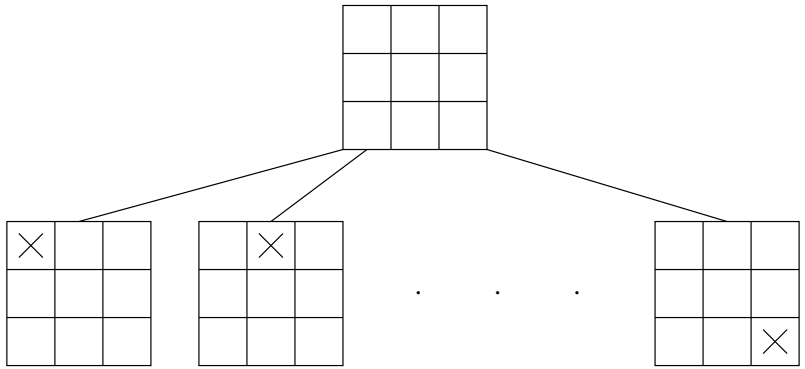


Max to move

- There is a set of *operators*. Here, Max can place a cross in any empty square, or Min a nought.
- There is a *terminal test*. Here, the game ends when three noughts or three crosses are in a row, or there are no unused spaces.
- There is a *utility* or *payoff* function. This tells us, numerically, what the outcome of the game is.

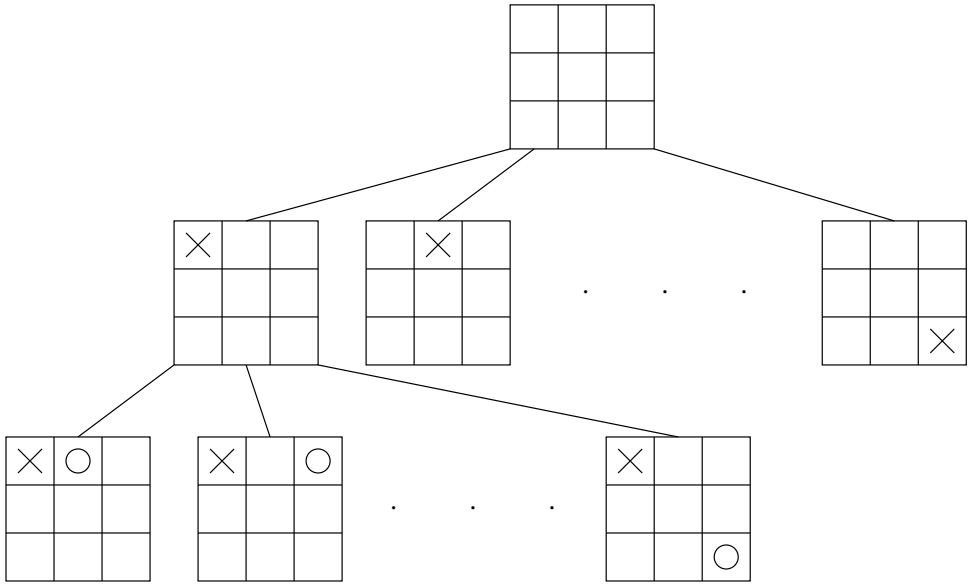
Perfect decisions in a two-person game

We can construct a tree to represent a game. From the initial state, Max can make nine possible moves:



Perfect decisions in a two-person game

In each case, Min has eight replies:



And so on.

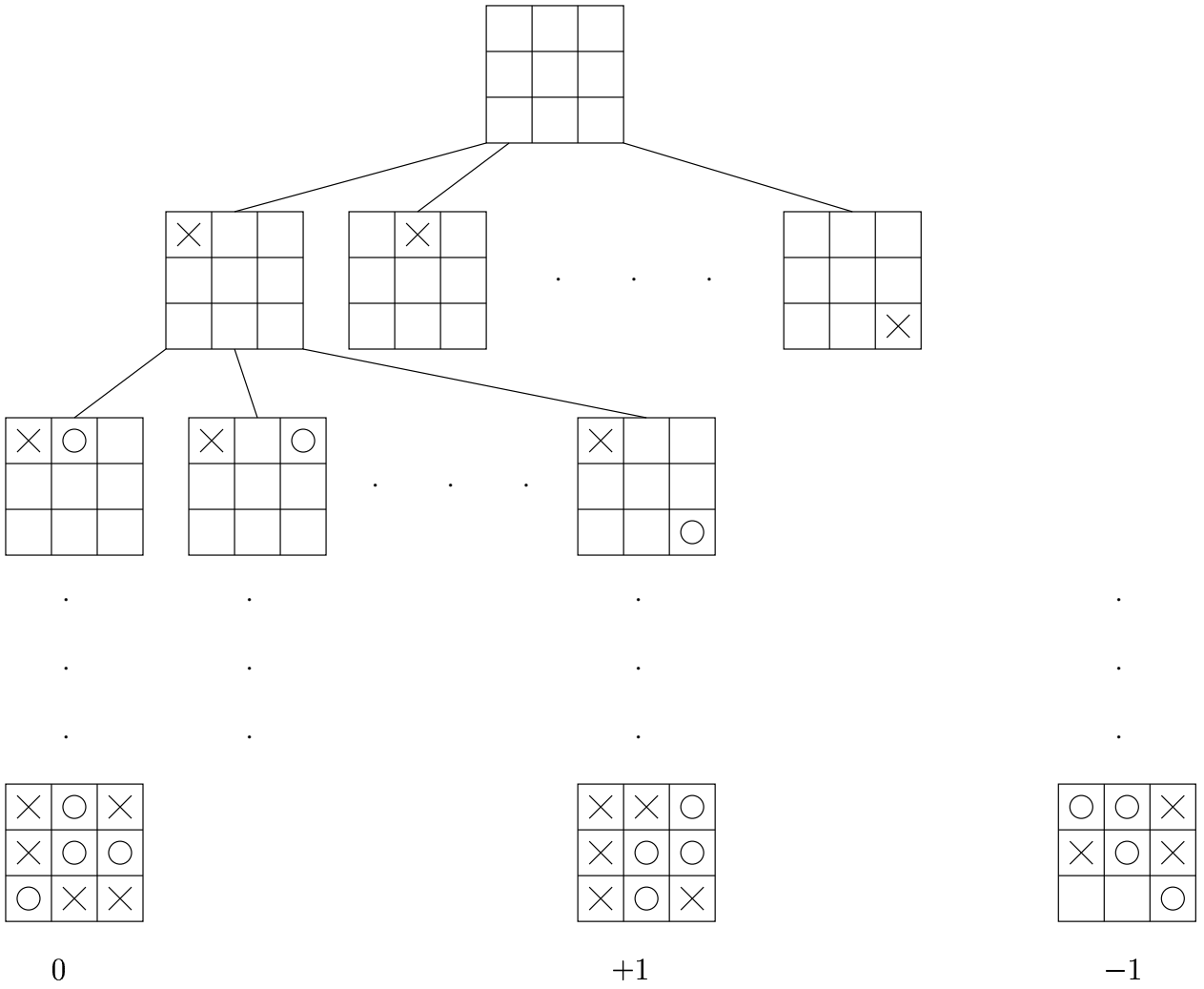
Perfect decisions in a two-person game

This construction can in principle be continued to represent *all* possibilities for the game.

The leaves are situations where a player has won, or there are no spaces.

Each leaf is labelled using the utility function.

Perfect decisions in a two-person game

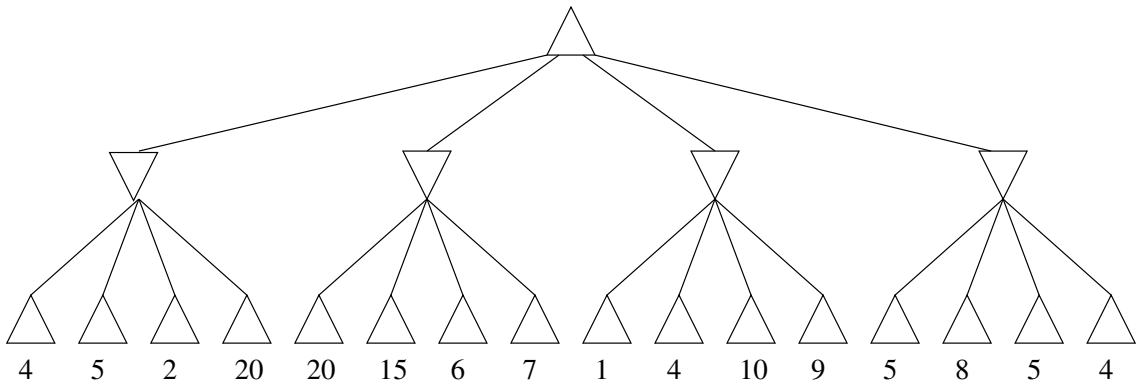


Perfect decisions in a two-person game

How can Max use this tree to decide on a move?

- if he is rational he will play to reach a position with the biggest utility possible;
- bit if Min is rational, she will play to *minimise* the utility available to Max.

Consider a much simpler tree:

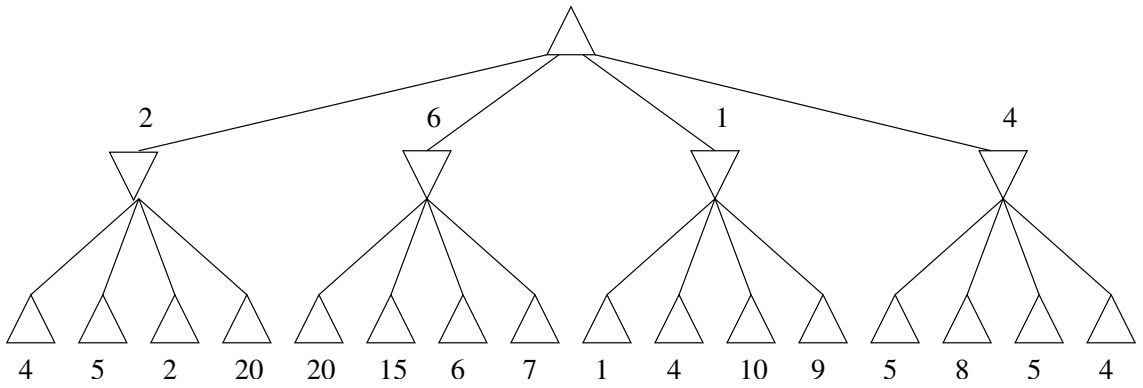


The minimax algorithm

There are only two moves—one for each player. Game theorists would call this one move, or two *ply* deep.

- Max's utility is labelled for each terminal state.
- The *minimax algorithm* allows us to infer the best move that the current player can make, given the utility function.

We work backward from the leaves. In the current example:



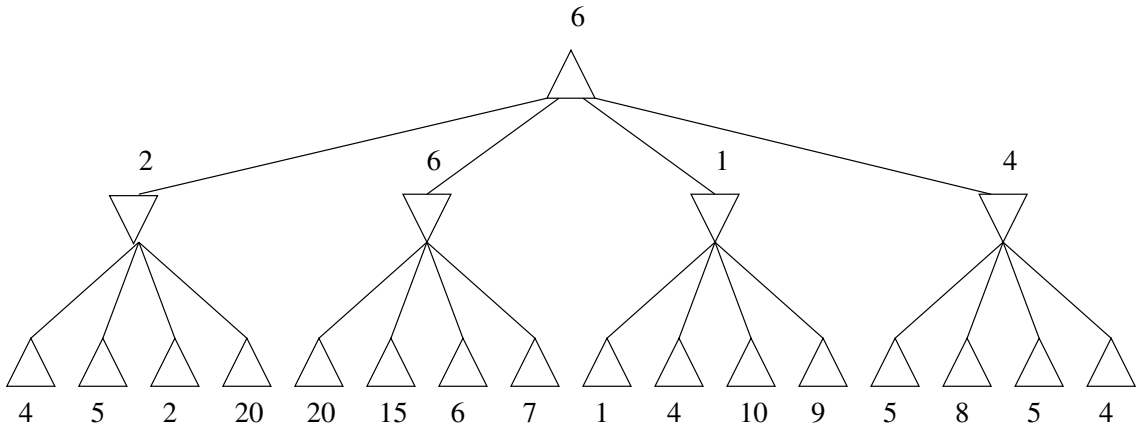
The minimax algorithm

Min takes the final move:

- If Min is in game position 1, her best choice is move 3. So from Max's point of view this node has a utility of 2.
- If Min is in game position 2, her best choice is move 3. So from Max's point of view this node has a utility of 6.
- If Min is in game position 3, her best choice is move 1. So from Max's point of view this node has a utility of 1.
- If Min is in game position 4, her best choice is move 4. So from Max's point of view this node has a utility of 4.

The minimax algorithm

Moving one further step up the tree:



We can see that Max's best opening move is move 2, as this leads to the node with highest utility.

The minimax algorithm

In general:

- generate the complete tree and label the leaves according to the utility function;
- working from the leaves of the tree upward, label the nodes depending on whether Max or Min is to move;
- if Min is to move label the current node with the *minimum* utility of any descendant;
- if Max is to move label the current node with the *maximum* utility of any descendant.

If the game is p ply and at each point there are q available moves then this process has $O(q^p)$ time complexity and space complexity linear in p and q.

Making imperfect decisions

We need to avoid searching all the way to the end of the tree. So:

- we generate only part of the tree: instead of testing whether a node is a leaf we introduce a *cut-off* test telling us when to stop;
- instead of a utility function we introduce an *evaluation function* for the evaluation of positions for an incomplete game.

The evaluation function attempts to measure the expected utility of the current game position.

Making imperfect decisions

How can this be justified?

- This is a strategy that humans clearly sometimes make use of.
- For example, when using the concept of *material value* in chess.
- The effectiveness of the evaluation function is *critical*...
- ... but it must be computable in a reasonable time.
- (In principle it could just be done using minimax!)

The evaluation function

Designing a good evaluation function can be extremely tricky:

- let's say we want to design one for chess by giving each piece its material value: pawn = 1, knight/bishop = 3, rook = 5 and so on;
- define the evaluation of a position to be the difference between the material value of black's and white's pieces

$$\text{eval}(\text{position}) = \sum_{\text{black's pieces } p_i} \text{value of } p_i \; - \sum_{\text{white's pieces } q_i} \text{value of } q_i$$

This seems like a reasonable first attempt. Why might it go wrong?

The evaluation function

Consider what happens at the start of a game:

- until the first capture the evaluation function gives 0, so in fact we have a *category* containing many different game positions with equal estimated utility.
- For example, all positions where white is one pawn ahead.
- The evaluation function for such a category should represent the probability that a position chosen at random from it leads to a win.

The evaluation function

Considering individual positions.

If on the basis of past experience a position has 50% chance of winning, 10% chance of losing and 40% chance of reaching a draw, we might give it an evaluation of

$$\text{eval}(\text{position}) = (0.5 \times 1) + (0.1 \times -1) + (0.4 \times 0) = 0.4.$$

Extending this to the evaluation of categories, we should then weight the positions in the category according to their likelihood of occurring.

The evaluation function

Using material advantage as suggested gives us a *weighted linear evaluation function*

$$\text{eval}(\text{position}) = \sum_{i=1}^{n} w_i f_i$$

where the w_i are *weights* and the f_i represent *features* of the position. In this example

$$f_i = \text{value of the } i\text{th piece}$$

$$w_i = \text{number of } i\text{th pieces on the board}$$

where black and white pieces are regarded as different and the f_i are positive for one and negative for the other.

The evaluation function

Evaluation functions of this type are very common in game playing.

There is no systematic method for their design.

Weights can be chosen by allowing the game to play itself and using *learning* techniques to adjust the weights to improve performance.

$\alpha - \beta$ pruning

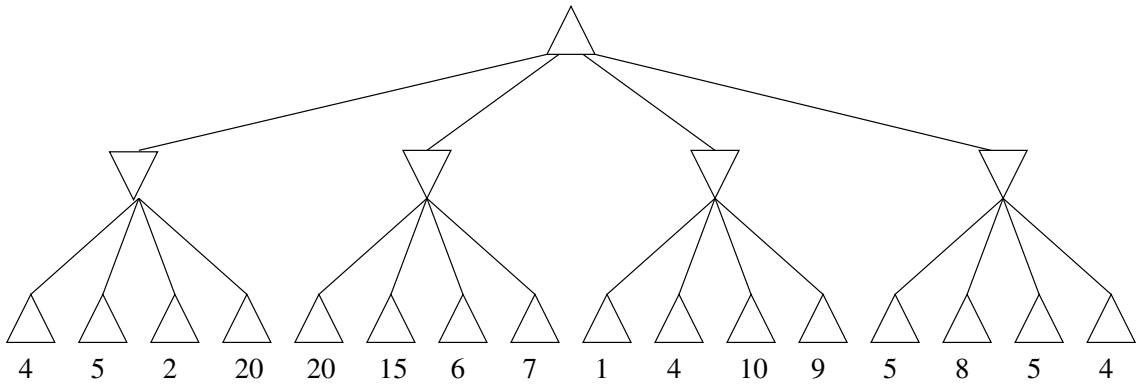
Even with a good evaluation function and cut-off test, the time complexity of the minimax algorithm makes it impossible to write a good chess program without some further improvement.

- Assuming we have 150 seconds to make each move, for chess we would be limited to a search of about 3 to 4 ply whereas...
- ...even an average human player can manage 6 to 8.

Luckily, it is possible to prune the search tree without affecting the outcome and without having to examine all of it.

$\alpha - \beta$ pruning

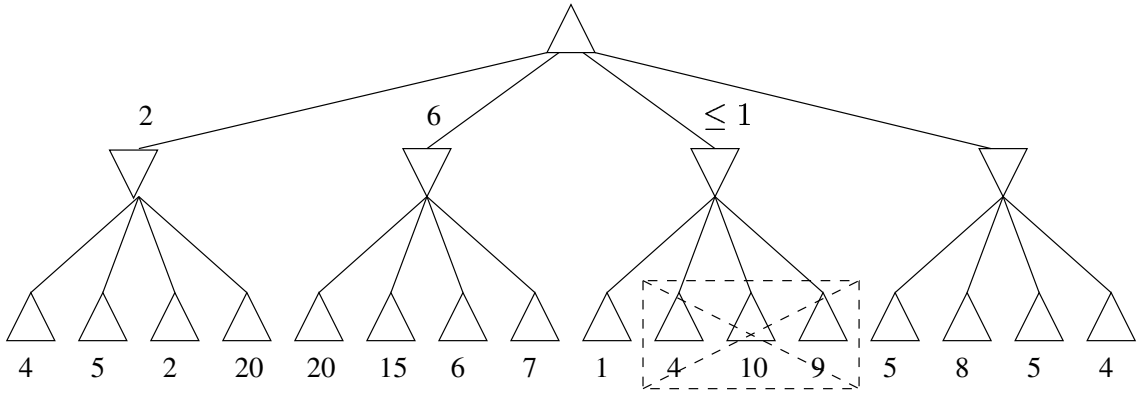
Returning for a moment to the earlier, simplified example:



The search is depth-first and left to right.

$\alpha - \beta$ pruning

The search continues as previously for the first 8 leaves.



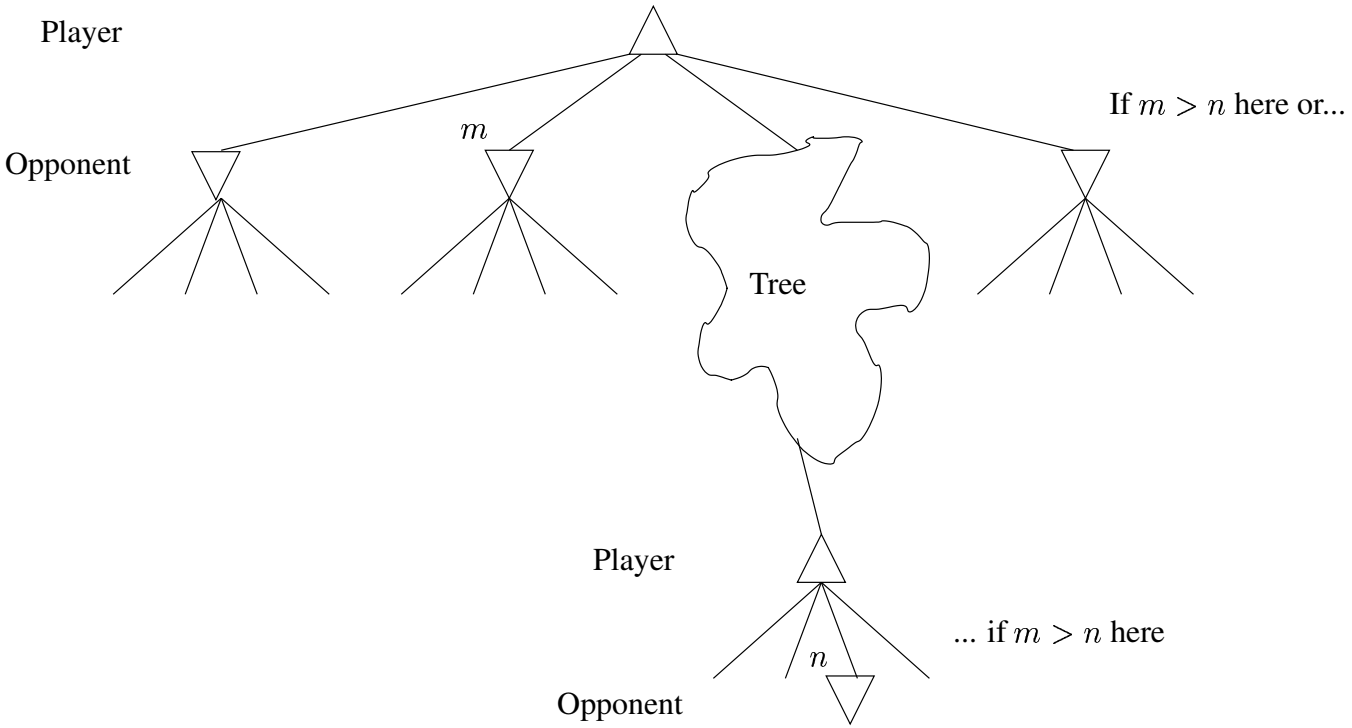
We then discover that should Max play move 3 as an opening, Min has the option of reaching a leaf with utility at most 1!

So: *we don't need to search any further under Max's opening move* 3.

This is because the search has *already established* that Max can do better by making opening move 2.

$\alpha - \beta$ pruning in general

If...



... then n will *never actually be reached*.

$\alpha - \beta$ pruning in general

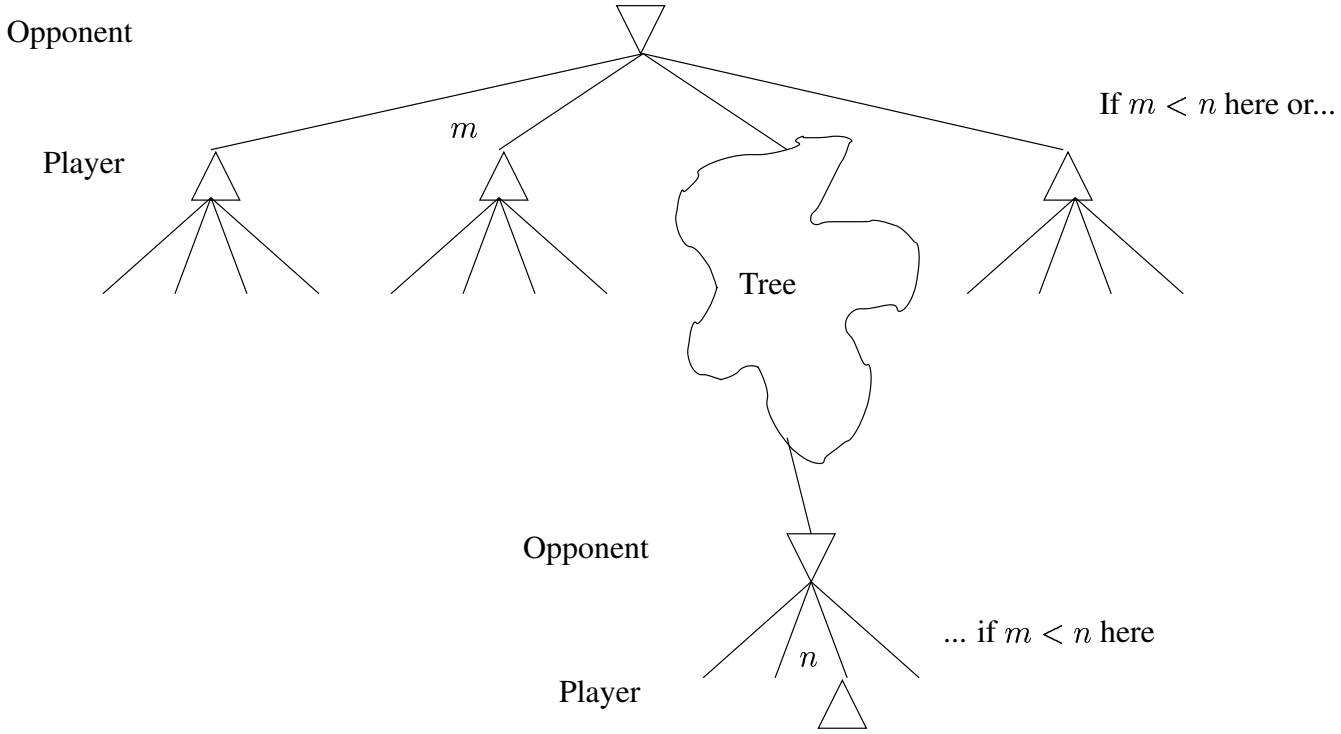
The search is depth-first, so we're only ever looking at one path through the tree.

We need to keep track of the value α where

$\alpha =$ the highest utility seen so far on the path for Max

$\alpha - \beta$ pruning in general

Similarly, if...



...then again n will never actually be reached. We keep track of $\beta =$ the lowest utility seen so far on the path for Min.

$\alpha - \beta$ pruning in general

Assume Max begins.

Initial values for α and β are

$$\alpha = -\infty$$

and

$$\beta = +\infty.$$

So: we call the function $\max(-\infty, +\infty, \texttt{root})$.

$\alpha - \beta$ pruning in general

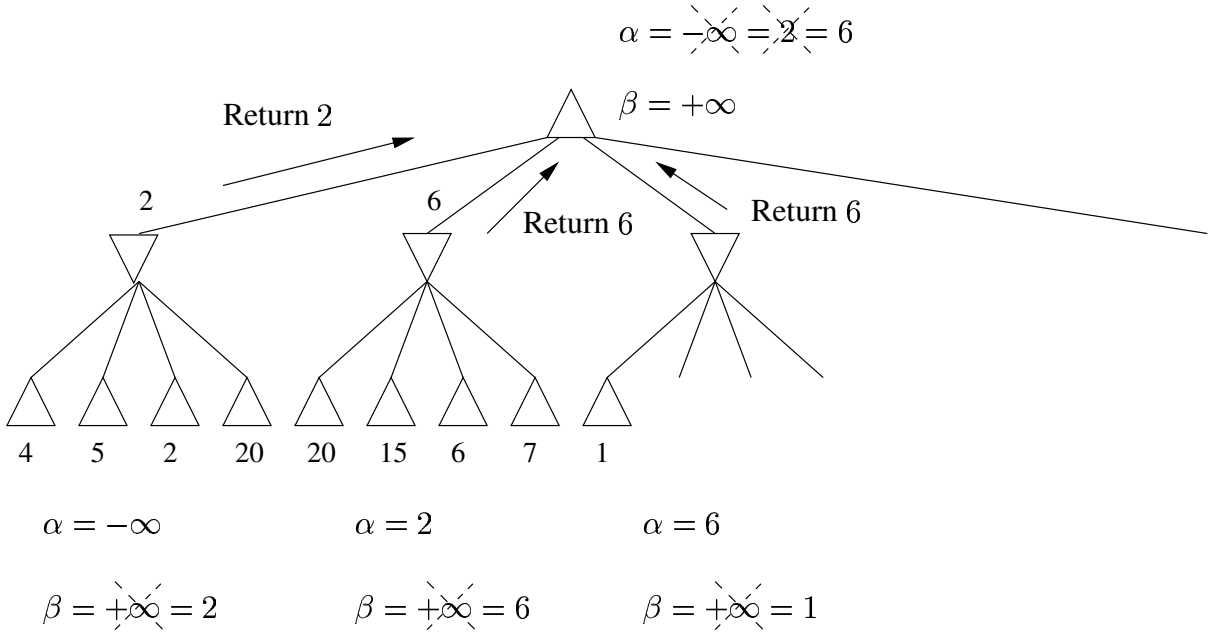
```
max(alpha,beta,node)
{
  if (node is at cut-off)
    return evaluation(node);
  else
    {
      for (each successor s of node)
        {
          alpha = maximum(alpha,min(alpha,beta,s));
          if (alpha >= beta)
            return beta; // pruning happens here.
        }
      return alpha;
    }
}
```

$\alpha - \beta$ pruning in general

```
min(alpha,beta,node)
{
  if (node is at cut-off)
    return evaluation(node);
  else
    {
      for (each successor s of node)
        {
          beta = minimum(beta,max(alpha,beta,s));
          if (beta <= alpha)
            return alpha; // pruning happens here.
        }
      return beta;
    }
}
```

$\alpha - \beta$ pruning in general

Applying this to the earlier example and keeping track of the values for α and β you should obtain:



How effective is $\alpha - \beta$ pruning?

(Warning! The theoretical results that follow are somewhat idealised.)

A quick inspection should convince you that the *order* in which moves are arranged in the tree is critical.

So, it seems sensible to try good moves first:

- if you were to have a perfect move-ordering technique the $\alpha - \beta$ pruning would be $O(q^{p/2})$ as opposed to $O(q^p)$;
- so the branching factor would effectively be $\sqrt{q}$ instead of q;
- and we would expect to be able to search ahead twice as many moves as before.

However, this is not realistic: if you had such an ordering technique you'd be able to play perfect games!

How effective is $\alpha - \beta$ pruning?

If moves are arranged at random then $\alpha - \beta$ pruning is:

- $O((q/\log q)^p)$ asymptotically when $q > 1000$ or;
- about $O(q^{3p/4})$ for reasonable values of q.

In practice simple ordering techniques can get close to the best case. For example, if we try captures, then threats, then moves forward *etc*.

Alternatively, we can implement an iterative deepening approach and use the order obtained at one iteration to drive the next.