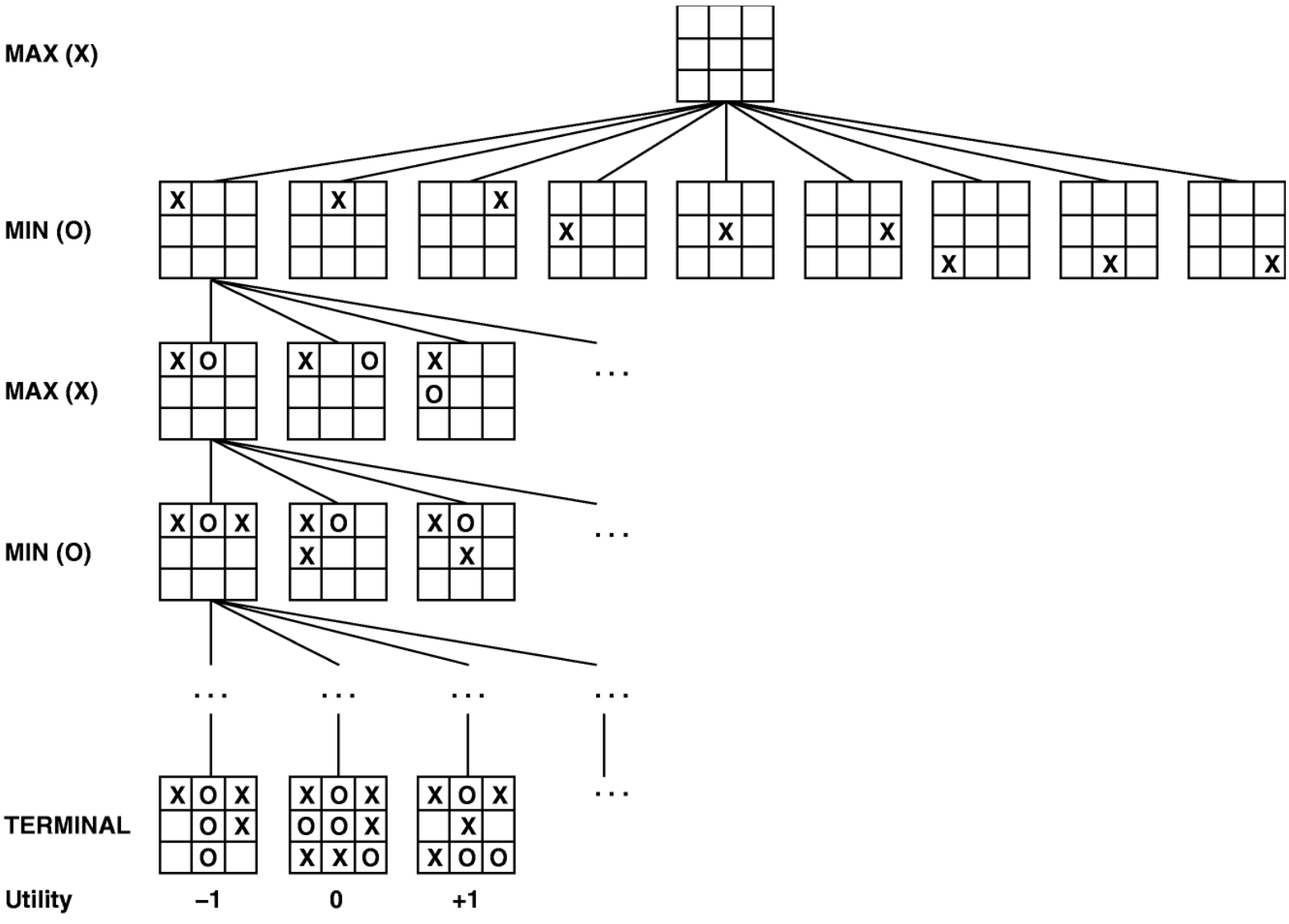
Games

Games require different search methods since we are often dealing with an adversary.

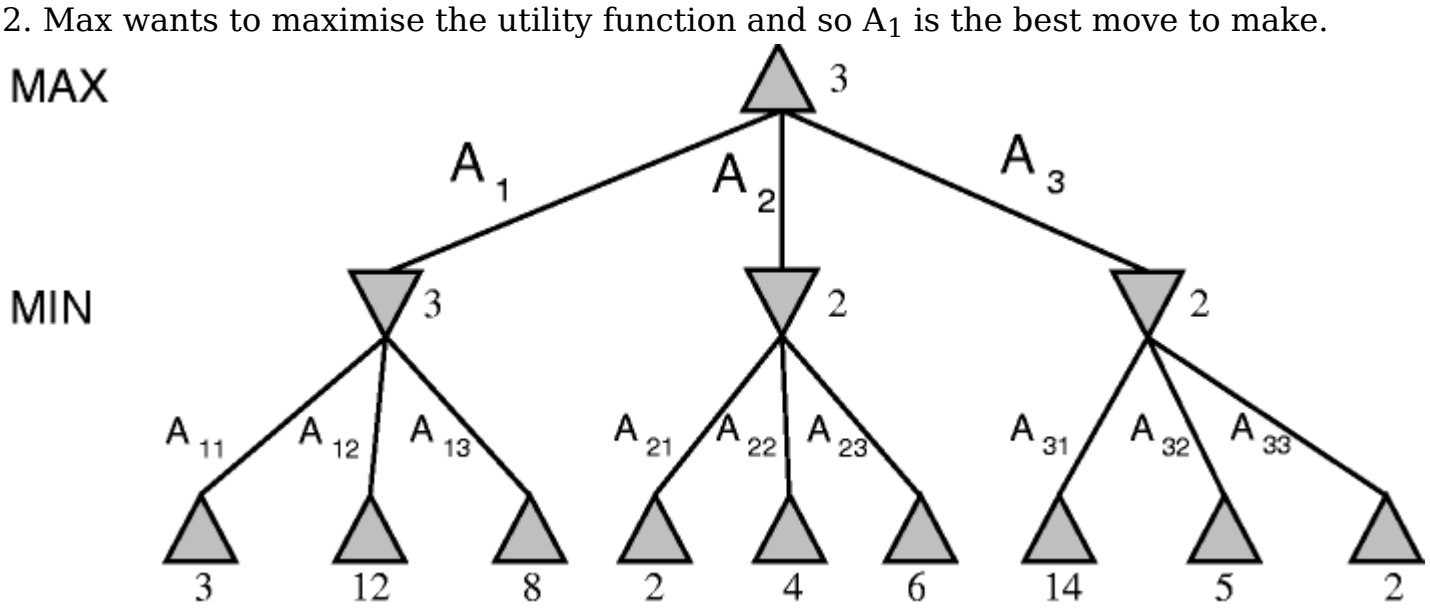
Game Tree

A tree in which the nodes denote board configurations, and the branches indicate how one board configuration can be transformed into another by a single move. A pair of moves is often called a **ply**.

Need a way to rank game outcomes. The ranking function is called a **utility function**. A simple utility function may give -1,0 or +1 for lose, draw and win.
Consider noughts and crosses with two players called MAX and MIN. If MAX is X and goes first then the tree is as follows.



Max will always try and maximize the utility function for the terminal state and min will always try to minimise it.
An exhaustive search of the game tree for a solution is usually not possible.
Consider the following simple game tree.
The utility function is only applied to terminal nodes. If max makes move A_1 then Min should make move A_{11}. Thus the result of making move $A_1$is a value of 3 for the utility function. Similarly A_2 leads to a value of 2 and A_3 a value of 2. Max wants to maximise the utility function and so A_1 is the best move to make.



Minimax consists of 5 steps.

1. Generate the complete game tree.
2. Apply the utility function to all the terminal states.
3. Use the utility of the terminal states to calculate a utility value for their parents (either max or min) depending

on depth.
4. Continue up to root node.
5. Choose move with highest value from root node.

The chosen move is called the **minimax decision** because it maximises the utility under the assumption that the opponent will play perfectly to try and minimise it.

function MINIMAX-DECISION(*game*) **returns** *an operator*

  for each *op* **in** OPERATORS[*game*] **do**
    VALUE[*op*] ← MINIMAX-VALUE(APPLY(*op*, *game*), *game*)
  end
  return the *op* with the highest VALUE[*op*]

---

function MINIMAX-VALUE(*state*, *game*) **returns** *a utility value*

  if TERMINAL-TEST[*game*](*state*) **then**
    return UTILITY[*game*](*state*)
  else if MAX is to move in *state* **then**
    return the highest MINIMAX-VALUE of SUCCESSORS(*state*)
  else
    return the lowest MINIMAX-VALUE of SUCCESSORS(*state*)

If the maximum depth of the tree is m and there are b moves at each node then the time complexity of minimax is $O(b^m)$. The algorithm is depth first so memory requirement is O(bm).

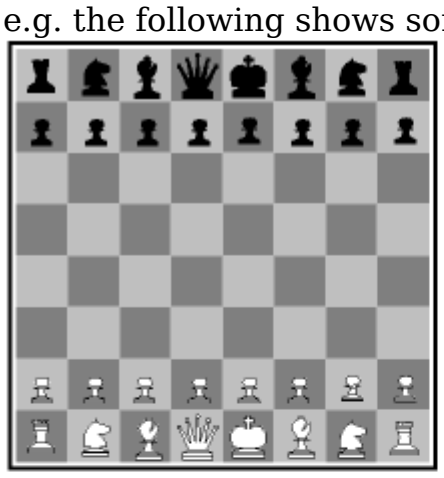
Evaluation functions

Minimax assumes that we can search all the way to the terminal states, this is not often practical.

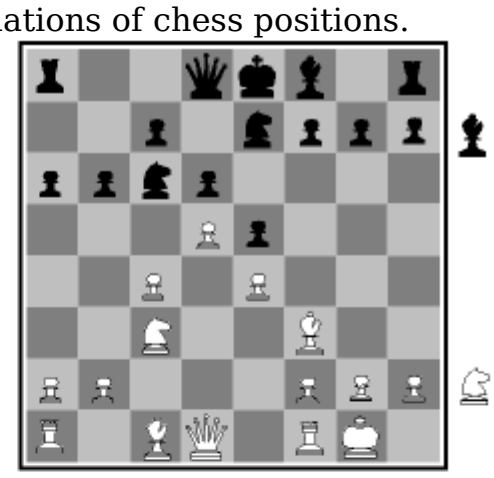
Instead of doing this we could use a heuristic **evaluation function** to estimate which moves leads to better terminal states. Now a cutoff test must be used to limit the search.

Choosing a good evaluation function is very important to the efficiency of this method. The evaluation function must agree with the utility function for terminal states and it must be a good predictor of the terminal values. If the evaluation function is infallible then no search is necessary, just choose the move that gives the best position. The better the evaluation function the less search need to be done.

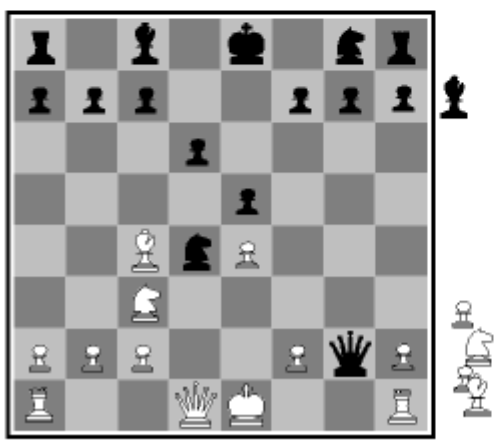
e.g. the following shows some evaluations of chess positions.



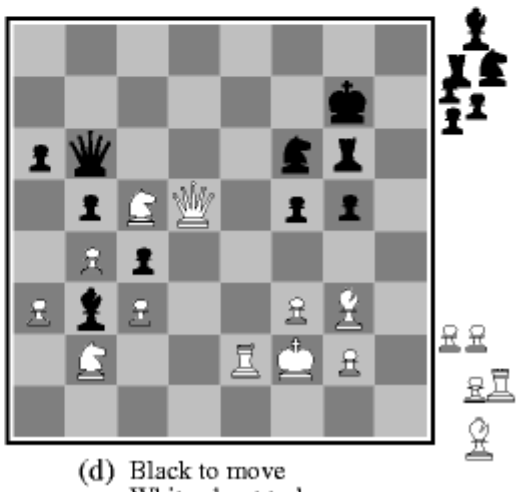
(a) White to move
Fairly even



(b) Black to move
White slightly better



(c) White to move
Black winning



(d) Black to move
White about to lose

The evaluation function is often a linear combination of features.

Cutting off search

The easiest way to stop the search is to have a fixed depth or to use iterative deepening.

Heuristic Continuation

You search for N ply in a tree, and find a very good move. But, perhaps, if you had searched just one ply further you would have discovered
that this move is actually very bad.

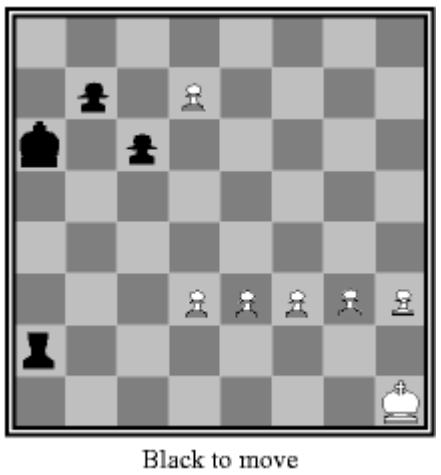
In general:

The analysis may stop just before your opponent captures one of your pieces or the analysis stops just before you capture one your opponent's pieces.

This is called the horizon effect: a good (or bad) move may be just over the horizon.

The Singular Extension Heuristic:

Search should continue as long as one moves static value stands out from the rest. If we don't use this heuristic, we risk harm from the Horizon Effect.
e.g. consider the following chess position. Black is ahead in material but if white can reach the eighth row with it's pawn then it can win. Black can stall this for some time by checking the white and so will never see that this is a bad position.



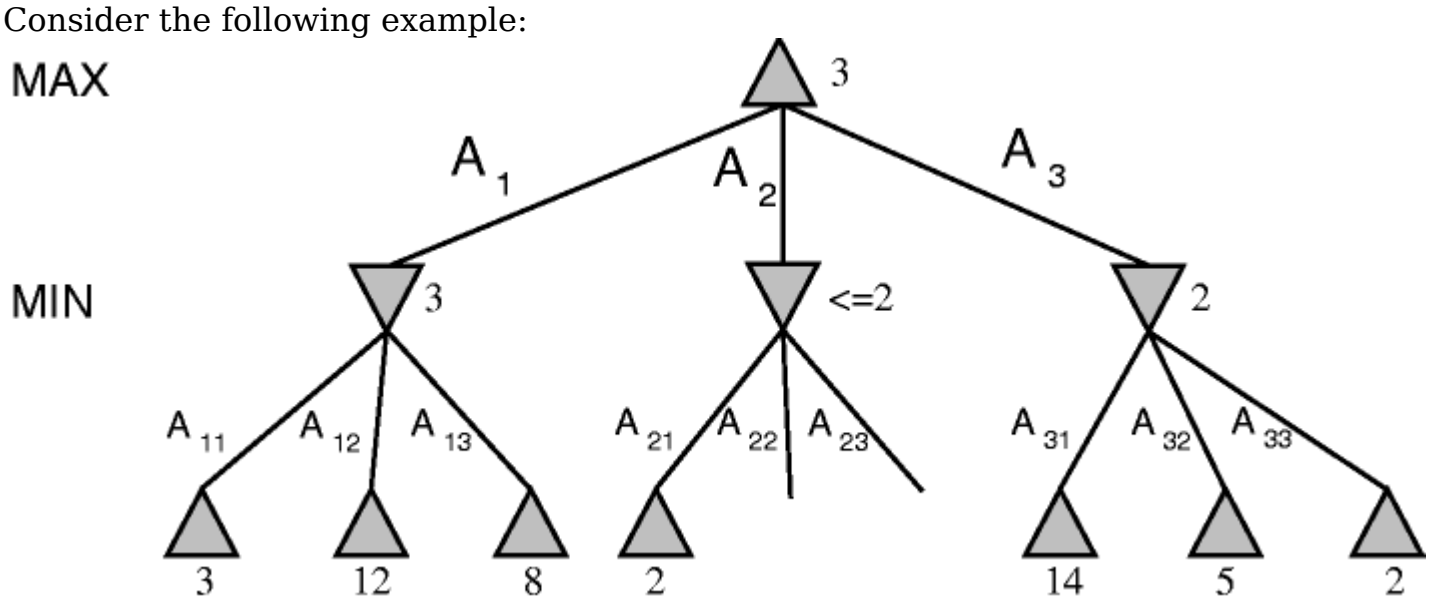
Black to move

Alpha Beta Pruning

Using the Minimax algorithm allows a chess playing program to look ahead about 3 or 4 ply. This is roughly as good an intermediate human player, but still fairly easy to beat. Minimax, however is wasteful because it evaluates some branches of the game tree that it need not. The principle behind Alpha-Beta pruning is:

If you have an idea that is surely bad, do not take time to see how terrible it is.

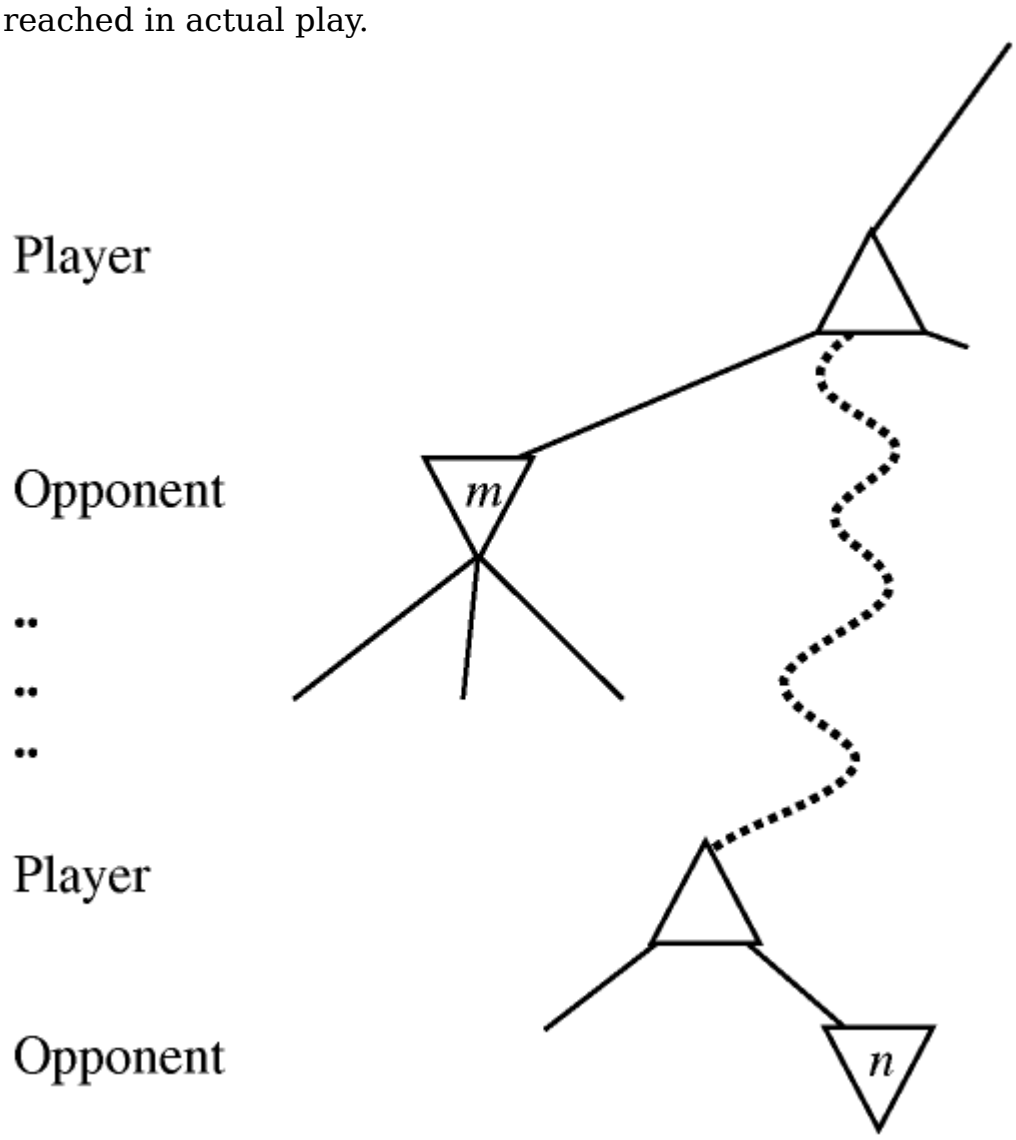
Consider the following example:



Search proceeds exactly as for minimax evaluating A_{11} A_{12} A_{13} and A_1 until, after looking at A_{21}, we find that A_2 must have a utility less than 2. We now know that A_2 is a bad choice because A_1 has a higher utility and we need not evaluate any more branches below A_2 .

The following diagram shows this for a general case.

If we have a better choice *m,* either at the parent node of *n* or at any choice point further up, then *n* will never be reached in actual play.



Minimax is depth first, so all that need be remembered is the best choice found so far for the player (MAX) and the best choice found so far for the opponent (MIN). These are called alpha and beta respectively.

The following algorithm illustrates this.

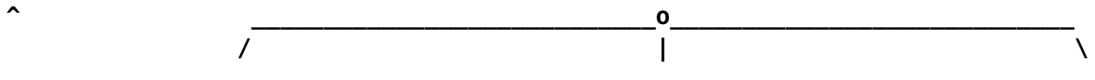
```
function MAX-VALUE(state, game, α, β) returns the minimax value of state
  inputs: state, current state in game
          game, game description
          α, the best score for MAX along the path to state
          β, the best score for MIN along the path to state

  if CUTOFF-TEST(state) then return EVAL(state)
  for each s in SUCCESSORS(state) do
      α ← MAX(α, MIN-VALUE(s, game, α, β))
      if α ≥ β then return β
  end
  return α
```

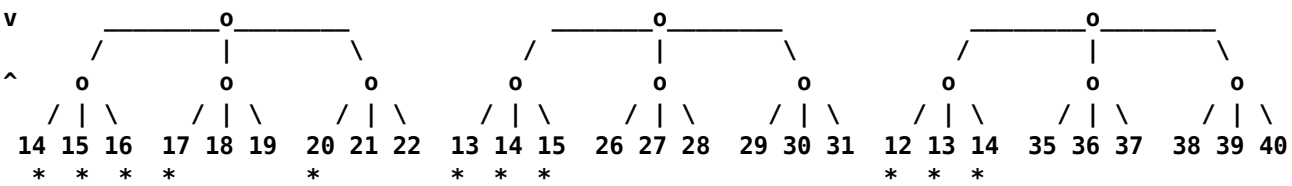
```
function MIN-VALUE(state, game, α, β) returns the minimax value of state

  if CUTOFF-TEST(state) then return EVAL(state)
  for each s in SUCCESSORS(state) do
      β ← MIN(β, MAX-VALUE(s, game, α, β))
      if β ≤ α then return α
  end
  return β
```

Effectiveness of alpha-beta pruning.

What are the maximum savings possible?

Suppose the tree is ordered as follows:





Only those nodes marked (*) need be evaluated.
How many static evaluations are needed?.

If b is the branching factor (3 above) and d is the depth (3 above)

s = $2b^{d/2}$ - 1 IF d even

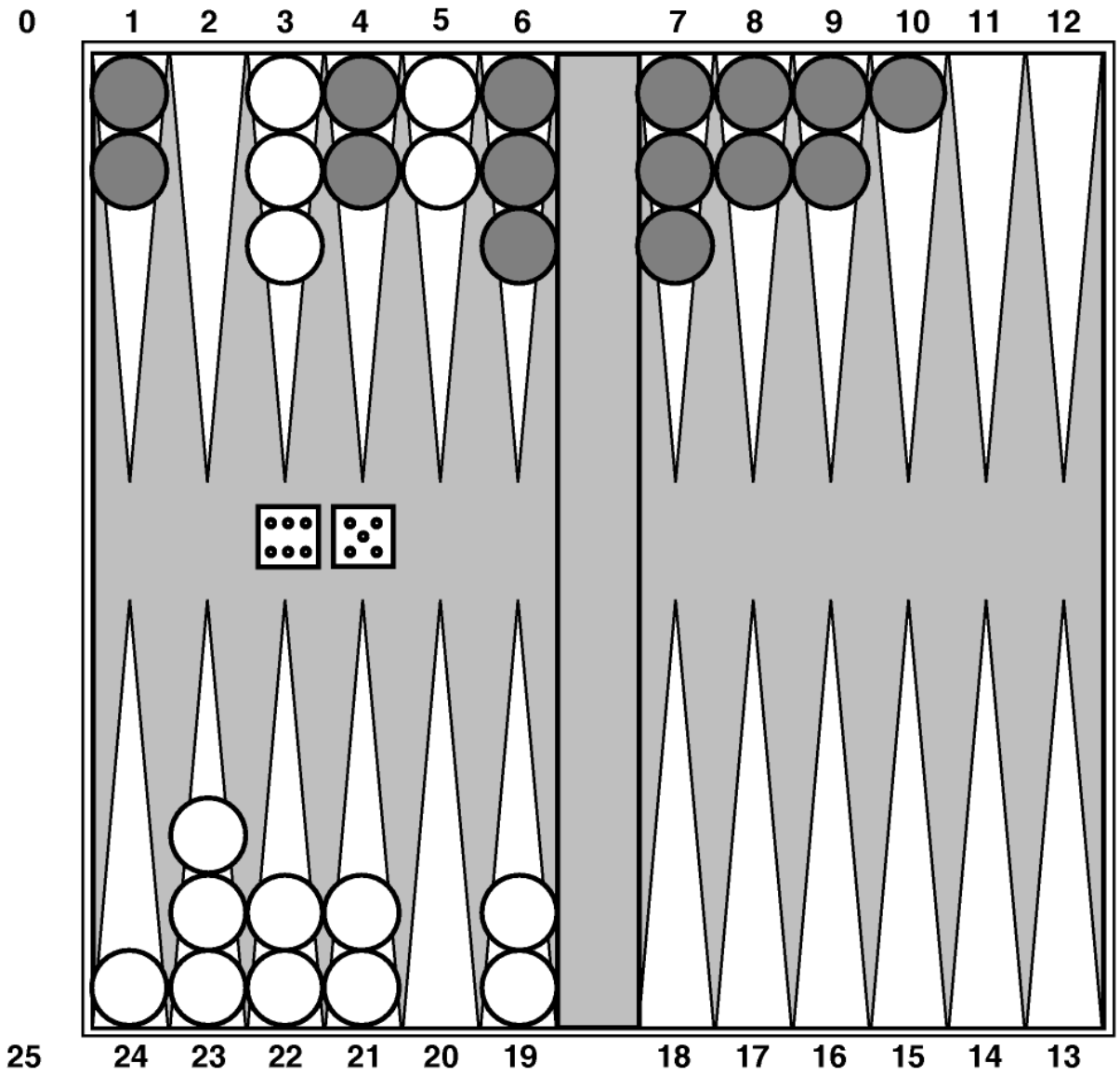
s = $b^{(d+1)/2}$ + $b^{(d-1)/2}$ - 1 IF d odd

For our tree d=3, b=3 and so s=11.

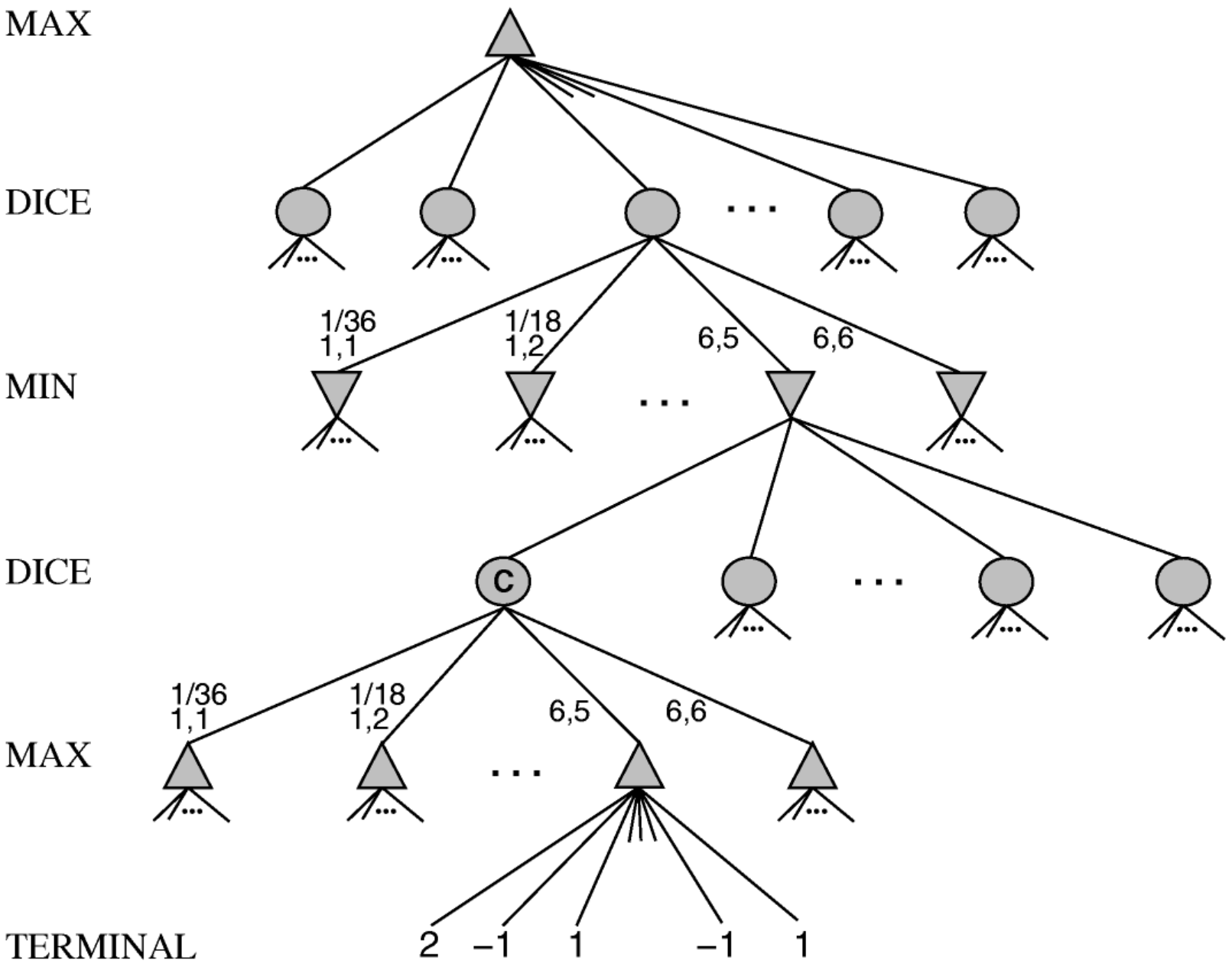
This is only for the perfectly arranged tree. It gives a lower bound on the number of evaluations of approximately $2b^{d/2}$ .The worst case is b^{d} (minimax)
In practice, for reasonable games, the complexity is O($b^{3d/4}$). Using minimax with alpha beta pruning allows us to look ahead about half as far again as without.

Games of Chance

Many games, such as backgammon, involve chance.



To represent this game we need to add **chance nodes** to the game tree. In the tree below chance nodes are shown as circles. Below each circle are the possible outcomes of the dice throw with the probability of the throw occurring. (1/36 for a double, 1/18 otherwise).



Now each position has no known outcome only an expected outcome. The expected value for a chance node above a max node is:

$$expectimax(C) = \sum_i (p(d_i) \max_s (utility(s)))$$

Where

C is a chance node
d_i is a dice roll
$p(d_i)$ is the probability a dice roll occurring
$\max_s(utility(s))$ is the maximum of the possible utility values for all the moves that can be made after dice roll d_i.

The expectimin value is similarly defined for a chance node above a min node.

$$expectimin(C) = \sum_i (p(d_i) \min_s (utility(s)))$$

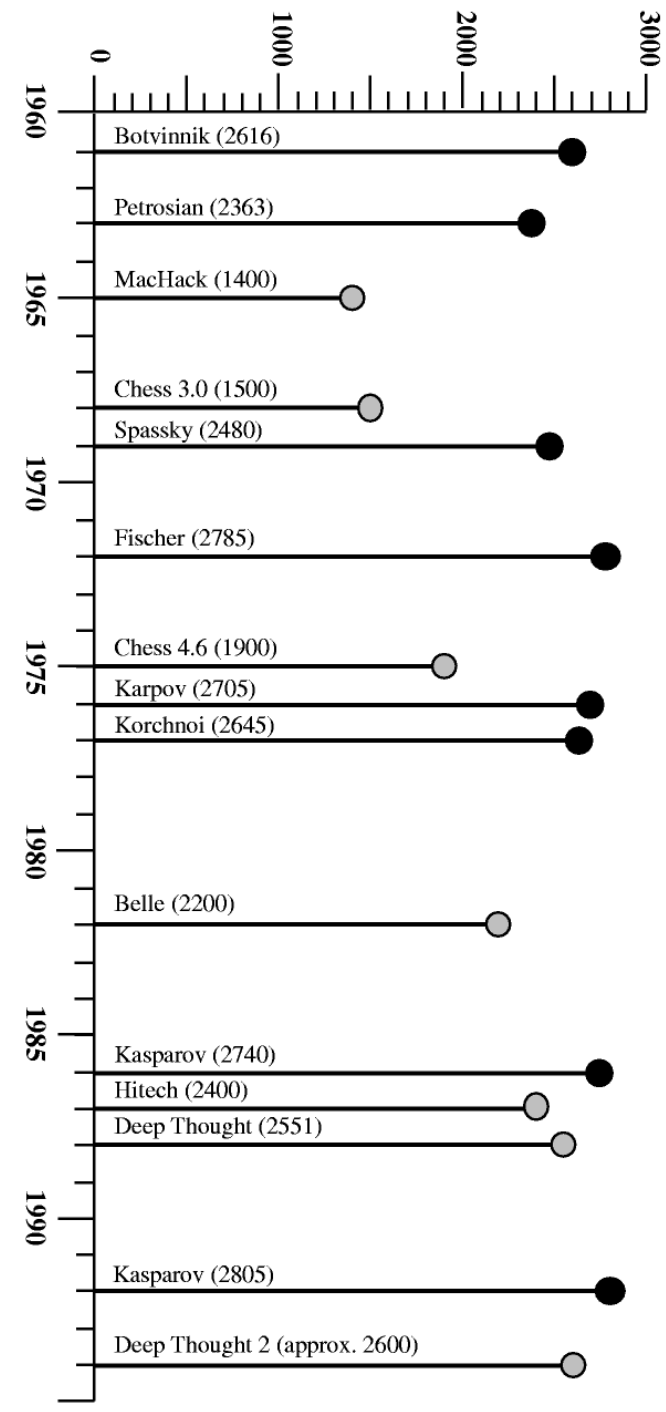
Complexity of expectiminimax

Minimax has a complexity of $O(b^m)$ where b is the branching factor and m is the depth. If there are now also a number (n) of chance nodes at each level, then the complexity becomes: $O(b^m n^m)$
This extra cost can make games of chance very difficult to solve.

State of the art

The following diagram suggests that in 1997 a chess program is as good as the best human. This is in fact what happened.
Deep Thought has 32 PowerPC CPUs and 256 dedicated hardware devices for evaluating board positions. It also has 100 years worth of grandmaster chess games and an opening and closing move database.

see: http://www.chess.ibm.com/meet/html/d.3.2.html for more details of Deep Blue


0
1000
2000
3000
1960
Botvinnik (2616)
Petrosian (2363)
1965
MacHack (1400)
Chess 3.0 (1500)
Spassky (2480)
1970
Fischer (2785)
Chess 4.6 (1900)
1975
Karpov (2705)
Korchnoi (2645)
1980
Belle (2200)
1985
Kasparov (2740)
Hitech (2400)
Deep Thought (2551)
1990
Kasparov (2805)
Deep Thought 2 (approx. 2600)