Adversarial Search

Adversarial Search / Games

- Two (or more) players
- Taking turns to move (take action, make a choice)
- Opposite goals – competition
- Zero-sum: how much one player gains, the opponent loses exactly the same amount

Examples of 2-player games:

  – tic-tac-toe, checkers, chess, go

History

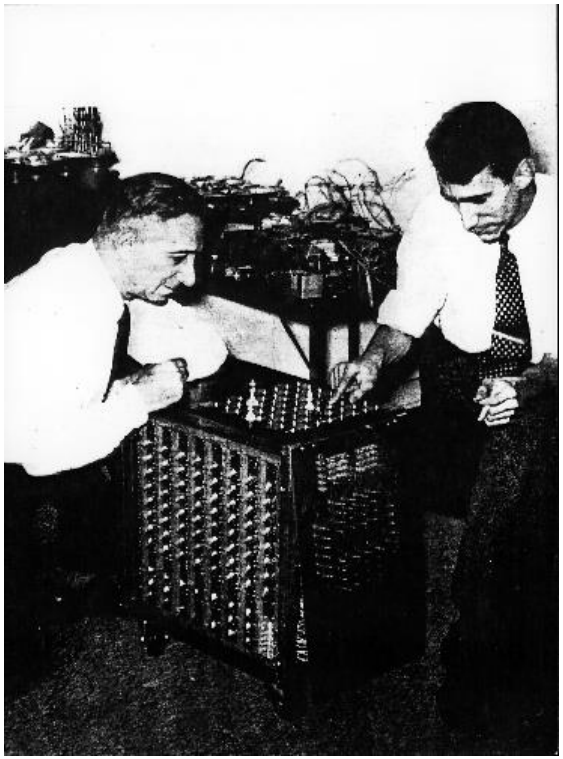
Much of the work in this area has been motivated by playing **chess**, which has always been known as a "**thinking person's game**".

In the sixties and seventies of the last century, several authors wrote that computers cannot play chess.

They were right: computers cannot play chess the way humans do.

Claude Shannon

- In **1949** Shannon published a ground-breaking paper on computer chess entitled "Programming a Computer for Playing Chess."
  - Describes how a computer could be made to play a **reasonable** game of chess.
  - His process for having the computer decide on which move to make is a **minimax** procedure, based on an evaluation function of a given chess position.

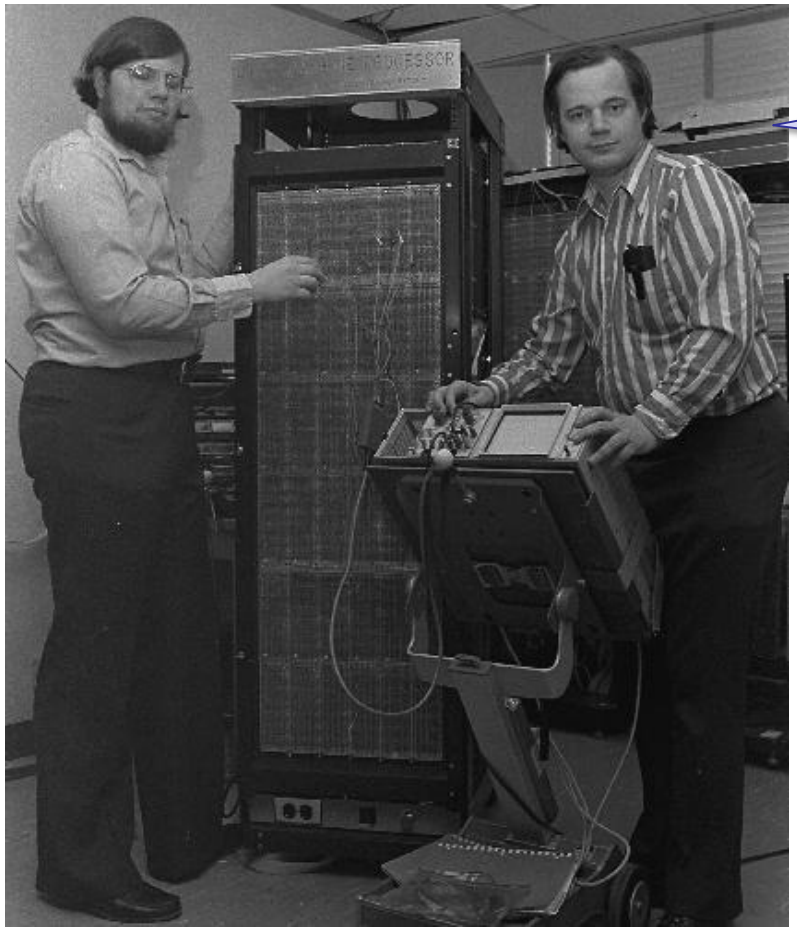


Alan Turing

- Shortly after, **Alan Turing** did a hand simulation of a program to play **checkers**, based on some of these ideas.

Richard Greenblatt

- The first programs to play real chess didn't arrive until almost ten years later, and it wasn't until **Greenblatt's MacHack 6, 1966** that a computer chess program defeated a good player.



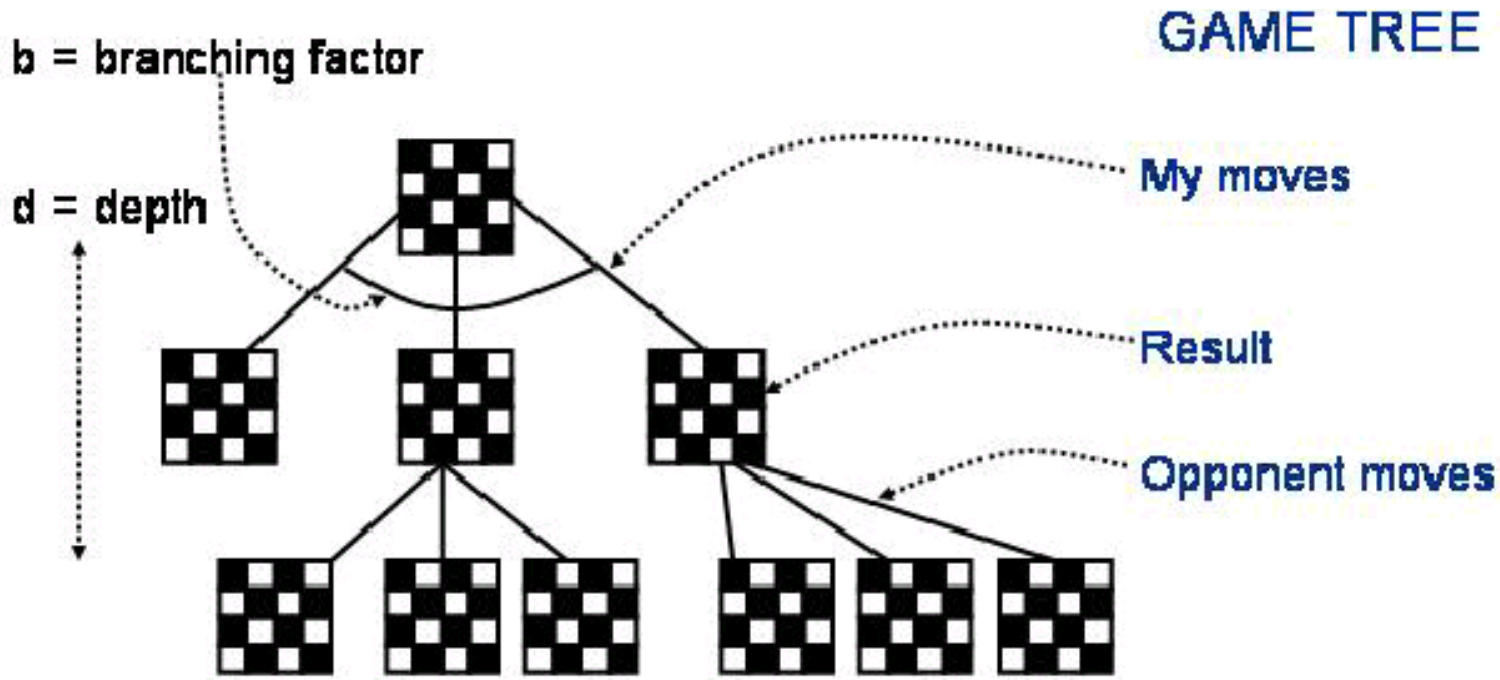
Richard Greenblatt

took one hour per move

IBM's Deep Blue

- Slow and steady progress eventually led to the defeat of reigning world champion **Garry Kasparov** against **IBM's Deep Blue** in May **1997**.

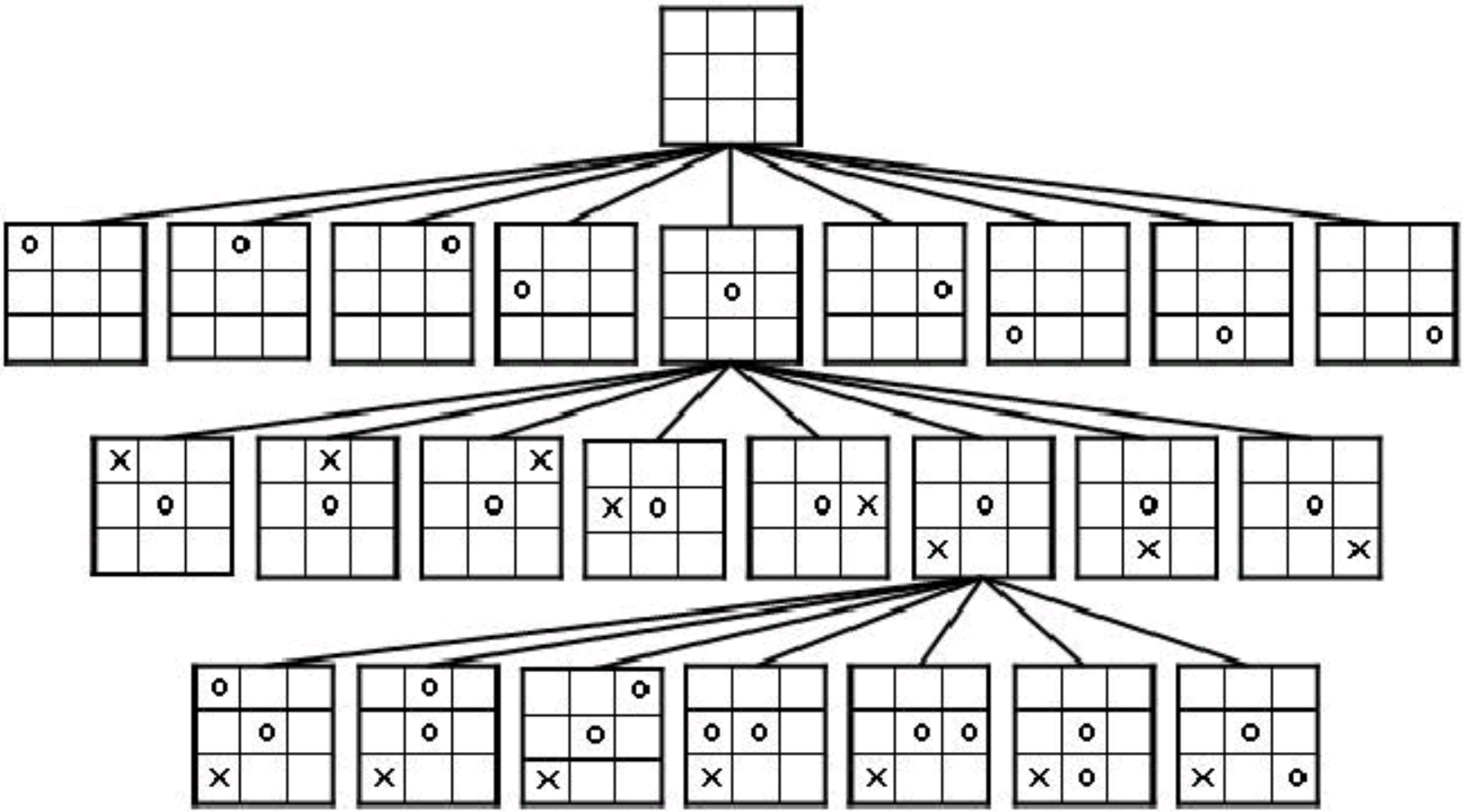
Games as Search



Chess
b = 36
d > 40

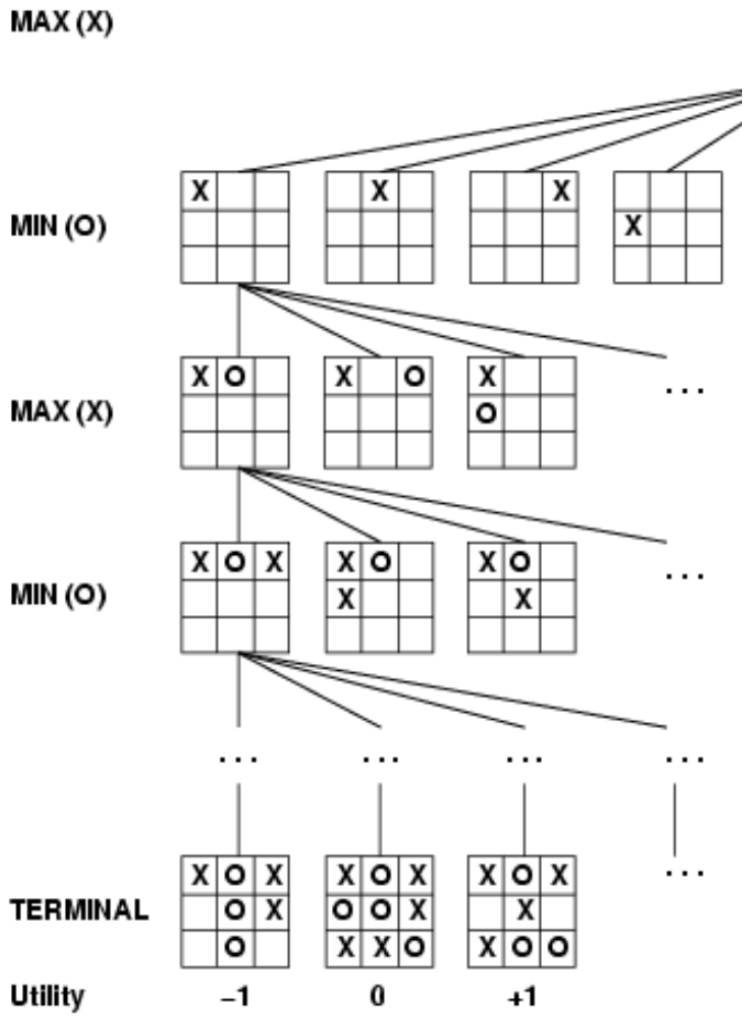
36^{40} is big!

Partial Game Tree for Tic-Tac-Toe



Here's a little piece of the game tree for Tic-Tac-Toe, starting from an empty board. Note that **branching is variable** depending on the level.

Partial Game Tree for Tic-Tac-Toe



- Several board positions from the leftmost part of the game tree for Tic-Tac-Toe. The last row shows the final (terminal) positions on the board, or **leaf nodes**. 255,168 leaves

- The Utility values under the last row indicate the result:
-1 means ‘lose’;
0 means ‘draw’; and
+1 means ‘win’.

- The result is always interpreted from the point of view of the first player.

- Note that **depth is variable**.

Chess as Search

Tic-Tac-Toe 255,168 leaf nodes are calculated – the game is “solved”.
Checkers are also “solved” recently.
Can we “solve” chess?

Chess
branching at each level: about 35
depth: 80 -100 (each player makes 40 - 50 moves, on average)
- - - - - - - - - - - - - - - - - -
leaf nodes: **10^{120}**

Resources
atoms in Universe: 10^{80}
nanoseconds in year: $10^{7} * 10^{9}$
years of Universe: $1.4 * 10^{10}$
- -
Total: **10^{106}**

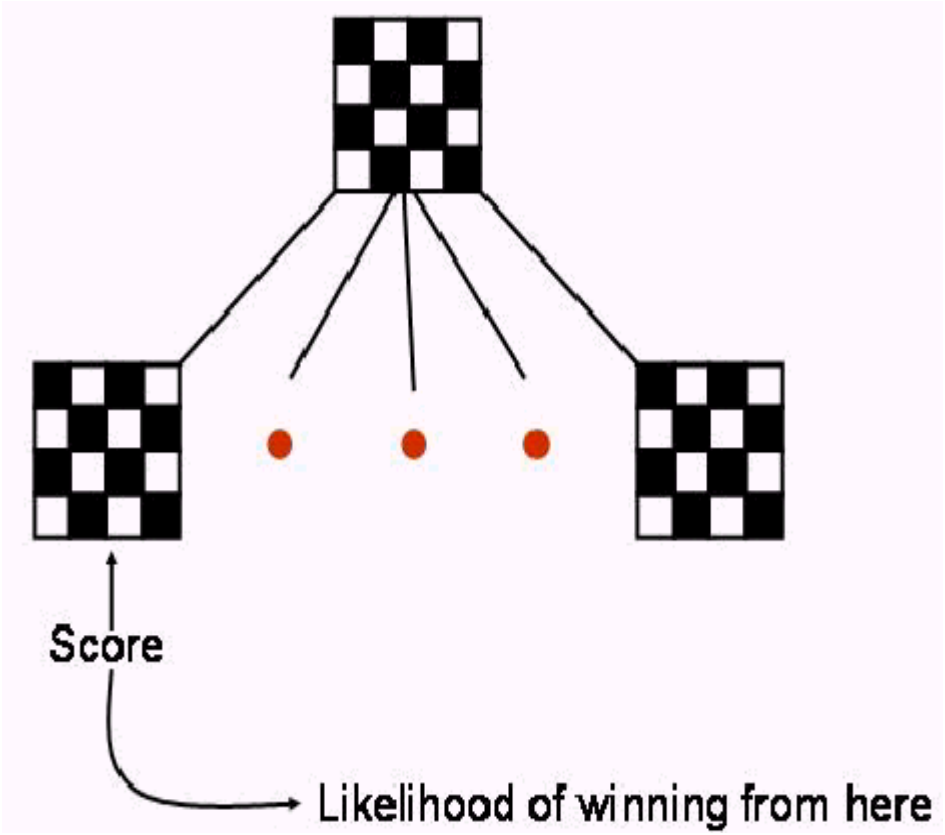
If from the beginning of time (from Big Bang), every atom in our Universe would calculate one chess leaf node per nanosecond, we would still fall short of solving chess.

Limited look ahead + scoring

- The key idea that underlies game playing programs (presented in Shannon's 1949 paper) is that of
  - **Limited look-ahead** and
  - **Min-Max algorithm**.
- Imagine we are going to look ahead in the game-tree to a depth of 2
  - (or 2 **ply**)

- We can use our scoring function to see what the values are at the **leaves of this tree**.
- These values are called the "**static evaluations.**"
- We compute a value for each of the nodes above this one in the tree by "**backing up**" these static evaluations in the tree.
- The player who is building the tree is trying to **maximize** his score. The opponent is trying to **minimize** the score.
- So, each layer of the tree can be classified into either a **maximizing layer** or a **minimizing layer**.

Scoring Function



- A crucial component of any game playing program. **Assigns a numerical value to a board position.**
- We can think of this value as capturing the **likelihood of winning from that position**.
- Since in these games one person's win is another's person loss, **we will use the same scoring function for both players**, simply **negating** the values to represent the opponent's scores.

Static evaluations

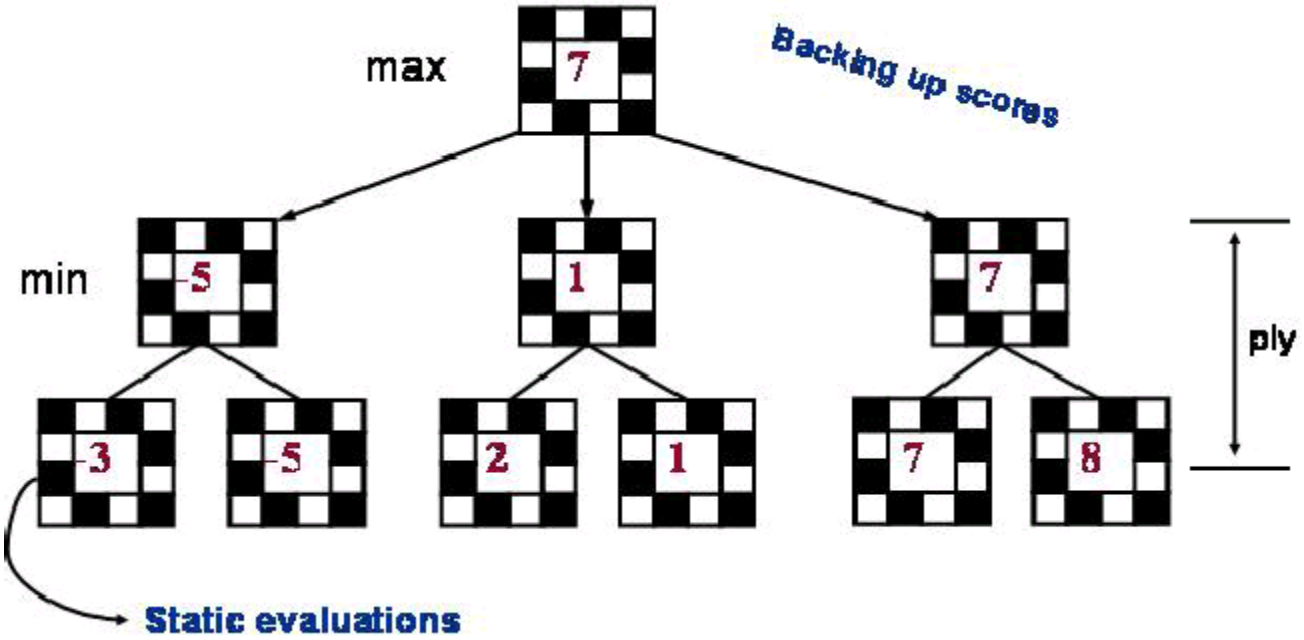
- For chess, we typically use linear weighted sum of features

$$\mathit{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- e.g., $w_1 = 9$ with

  $f_1(s)$ = (number of white queens) – (number of black queens), etc.

Min-Max Example



- In this example, the layer right above the leaves is a min layer, so we assign to each node in that layer the minimum score of any of its children.
- At the next layer up, we're maximizing so we pick the maximum of the scores available to us, that is, 7.
- So, this analysis tells us that we should pick the move that gives us the best guaranteed score, independent of what our opponent does. This is the **MIN-MAX** algorithm.

Min-Max

MAX-VALUE (state, depth)

```
if (depth==0) return EVAL (state)
v=-∞
for each s in SUCCESSORS (state) do
        v = MAX (v, MIN-VALUE (s, depth-1))
return v
```

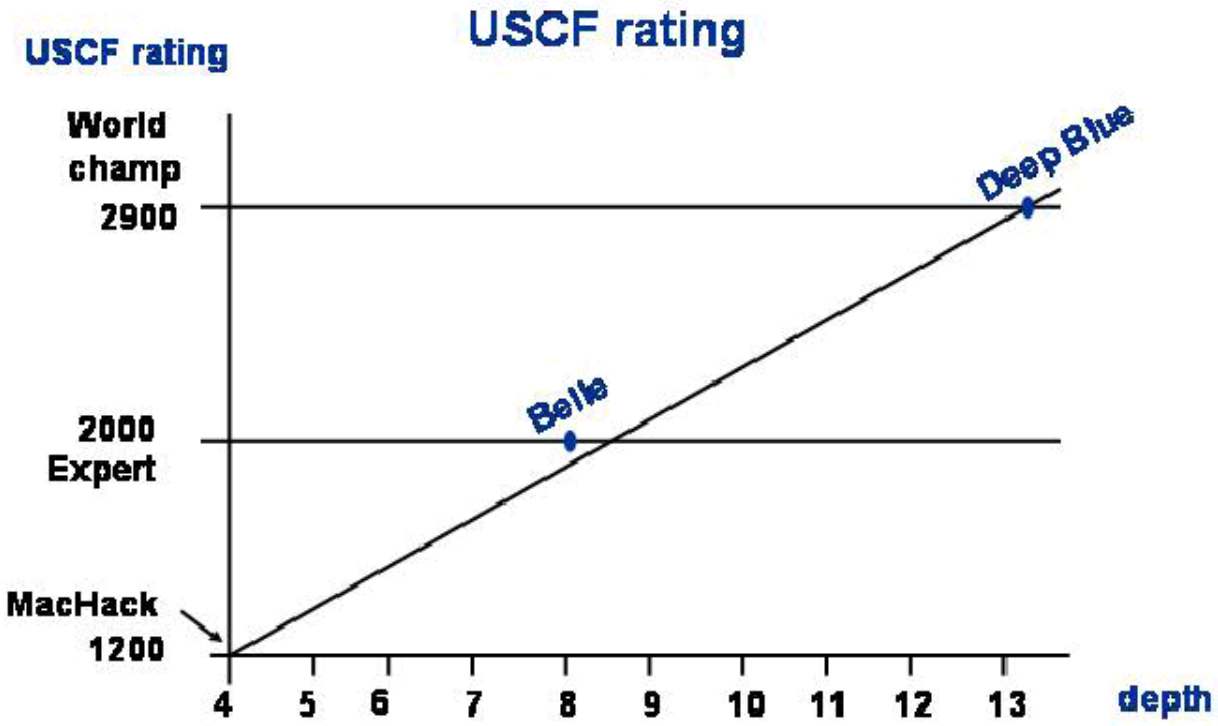
MIN-VALUE (state, depth)

```
if (depth==0) return EVAL (state)
v=∞
for each s in SUCCESSORS (state) do
        v = MIN (v, MAX-VALUE (s, depth-1))
return v
```

Simple recursive **alternation** of maximization and minimization at each layer.

We assume that we **count the depth value down** from the max depth so that when we reach a depth of 0, we apply our static evaluation to the board.

The key idea is that the more **lookahead** we can do, that is, the deeper in the tree we can look, the better our evaluation of a position will be, even with a simple evaluation function.



MacHack6, which had a ranking of 1200, searched on average to a depth of 4.

Belle, which was one of the first hardware-assisted chess programs doubled the depth and gained about 800 points in ranking.

Deep Blue, which searched to an average depth of about 13 beat the world champion with a ranking of about 2900.

Deep Blue:

256 specialized chess processors

30 billion moves per minute.

13-ply typical search depth, but in some dynamic situations it could go as deep as 30.

Games in practice

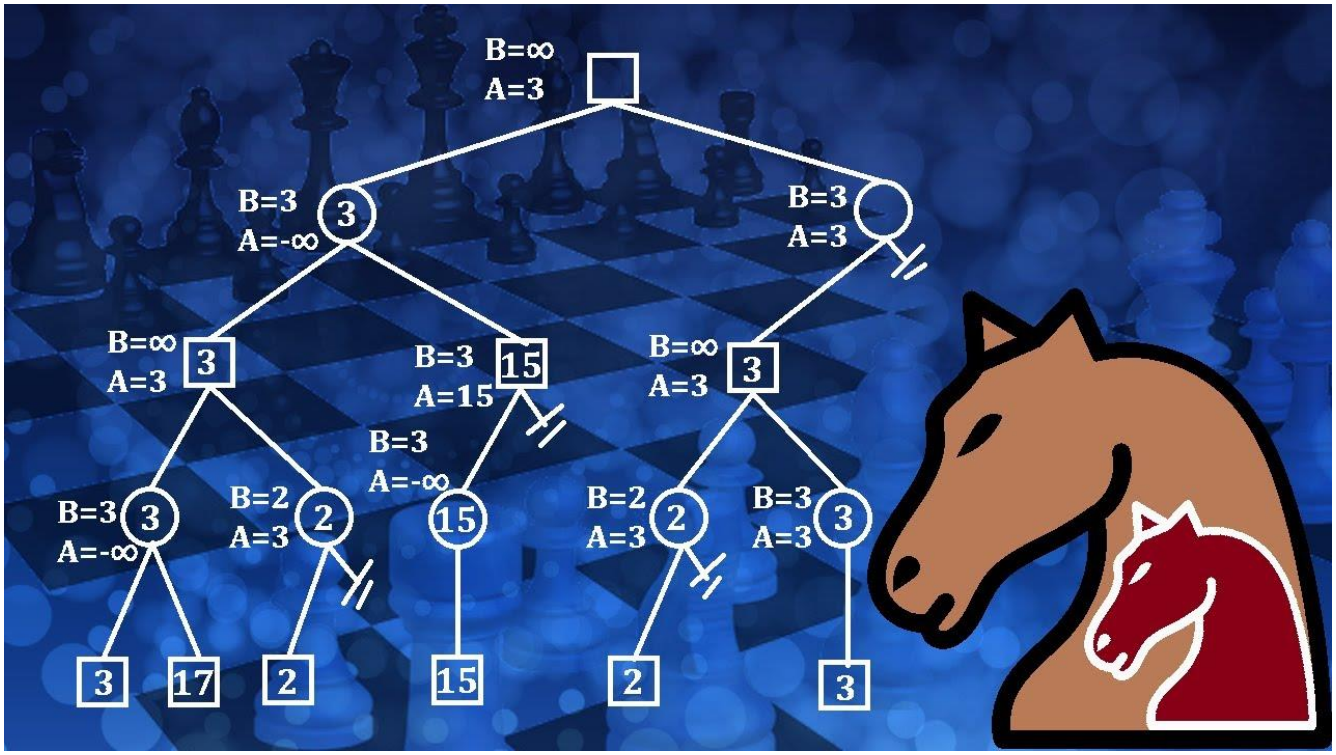
- **Checkers**: Chinook ended the 40-year-reign of human world champion Marion Tinsley in 1994. Used a precomputed endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 444 billion positions.

- **Chess**: Deep Blue defeated human world champion Garry Kasparov in a six-game match in 1997.

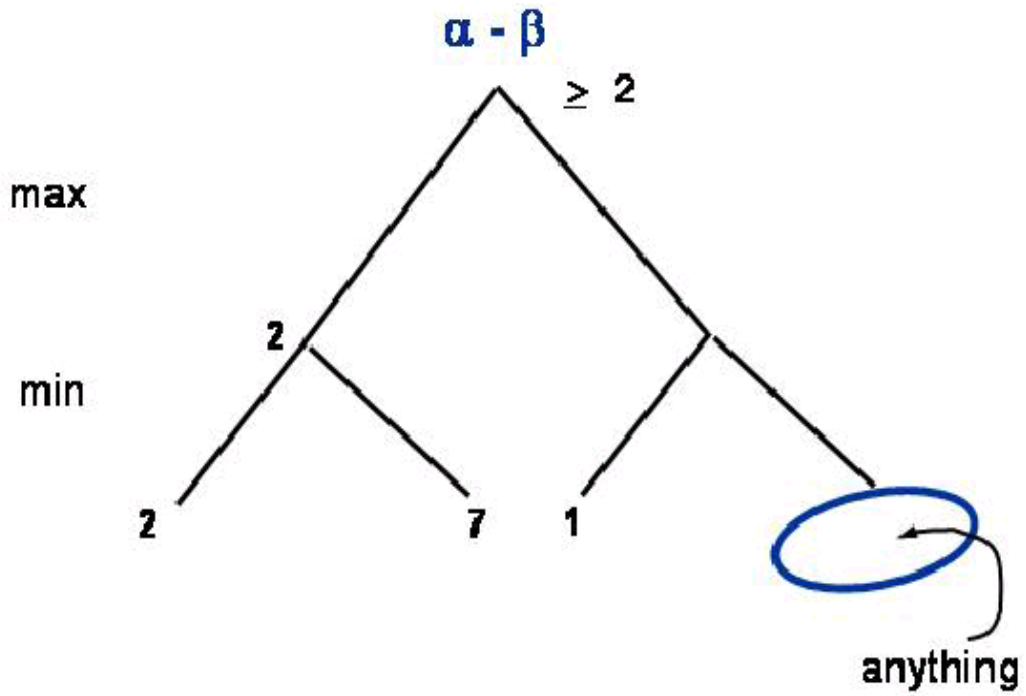
- **Othello**: human champions refuse to compete against computers, who are too good.

- **Go**: human champions used to refuse to compete against computers, who were too bad. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves.
  - AlphaGo, developed by Google DeepMind, made a significant advance by beating a professional human player in October 2015, using techniques that combined deep learning and Monte-Carlo tree search.

alpha-beta pruning



- There's one other idea that has played a crucial role in the development of computer game-playing programs.
- It’s an optimization of Min-Max search, but it is such a powerful and important optimization that it deserves to be understood in detail.
- The technique is called alpha-beta pruning, from the Greek letters traditionally used to represent the **lower** and **upper** bound on the score.



- Suppose that we have evaluated the sub-tree on the left (whose leaves have values 2 and 7).
- Since this is a **minimizing** level, we choose the value 2. So, the **maximizing** player at the top of the tree knows at this point that he can guarantee a score of at least 2 by choosing the move on the left.
- Now, we proceed to look at the subtree on the right. Once we look at the leftmost leaf of that subtree and see a 1, we know that if the maximizing player makes the move to the right then the minimizing player can force him into a position that is worth no more than 1.
- Now, we already know that this move is worse than the one to the left, so why bother looking any further?
- In fact, it may be that this unknown position is a great one for the maximizer, but then the minimizer would never choose it. So, no matter what happens at that leaf, the maximizer's choice will not be affected.

α-β

- **α**– the best outcome for MAX, calculated **so far**
- **β**– the best outcome for MIN, calculated **so far**
- **α and β are variables changing as the game proceeds**

- MAX player changes **α**
- MIN player changes **β**

- **Initialization**: deeply pessimistic
  - MAX presumes losing everything: **$\alpha = -\infty$**
  - MIN presumes that MAX will win everything: **$\beta = +\infty$**
- Note that the game result is always interpreted from MAX point of view.

- Consequently, **α** can go only up, up, up, and **β** can go only down, down, down.

- **Cutoff!** If **$\alpha \geq \beta$**, cutoff!

alpha-beta

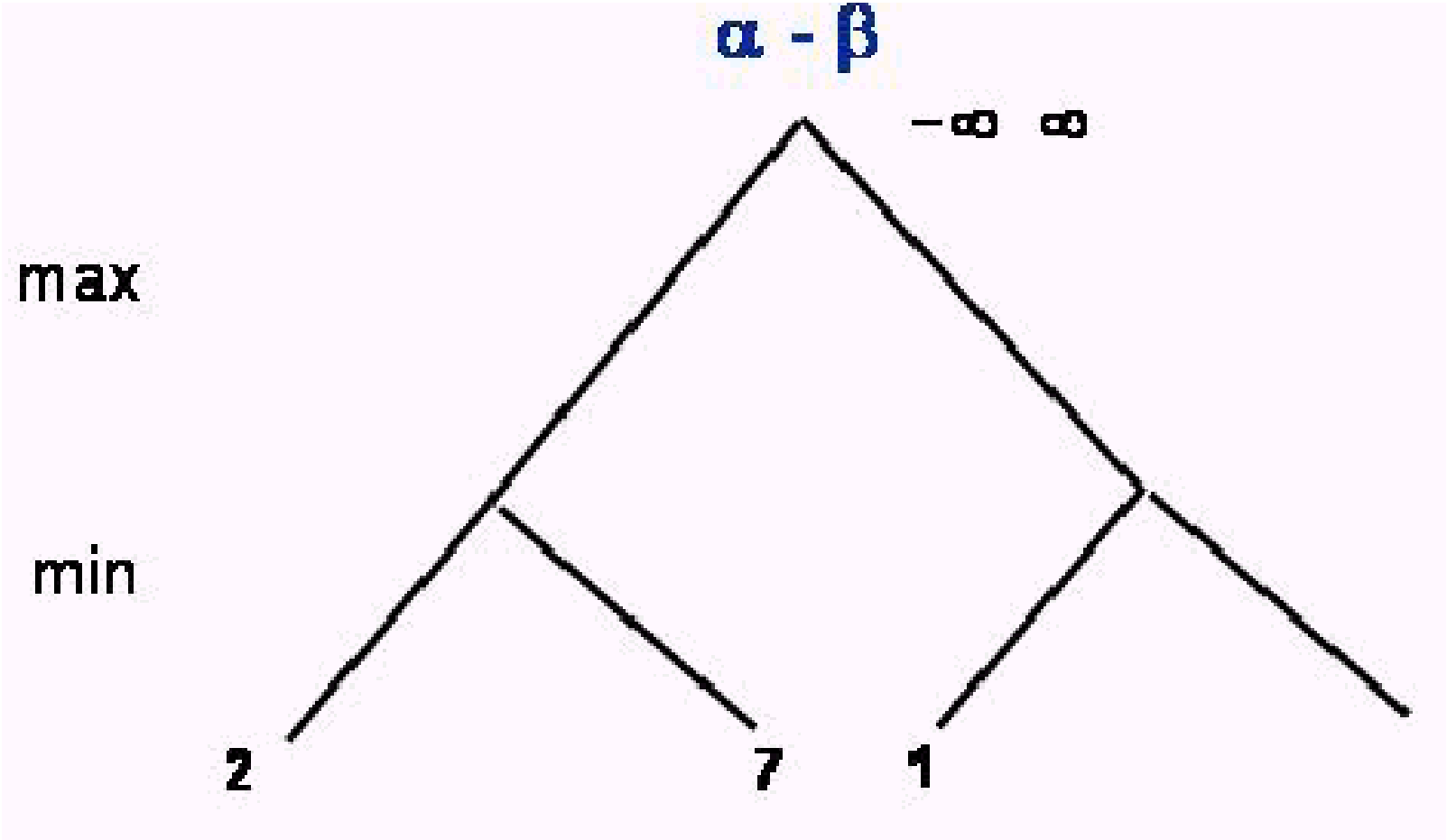
// α is the best score for MAX, β is the best score for MIN
// initial call is **MAX-VALUE (state, -∞, ∞, MAX_DEPTH)**

MAX-VALUE (state, α, β, depth)

```
if (depth == 0) return EVAL(state)
α=-∞
for each s in SUCCESSORS(state) do
        α = MAX (α, MIN-VALUE (s, α, β, depth-1))
        if α≥β return α       // cutoff
return α
```

MIN-VALUE (state, α, β, depth)

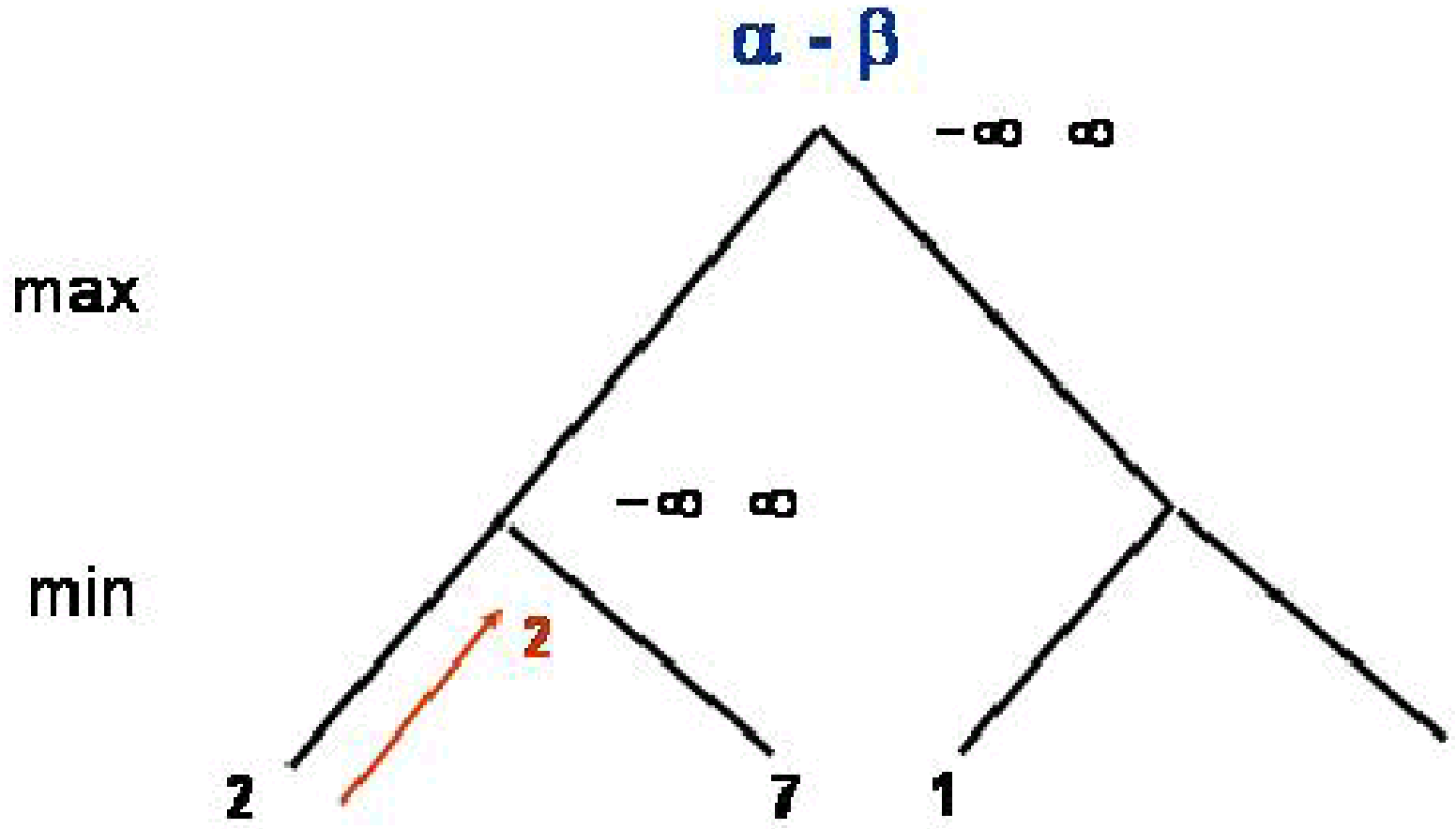
```
if (depth == 0) return EVAL(state)
 β=+∞
for each s in SUCCESSORS(state) do
        β = MIN (β, MAX-VALUE (s, α, β, depth-1))
        if α≥β return β       // cutoff
return β
```



We start with an initial call to **Max-Value** with the initial infinite values of alpha and beta, meaning that we know nothing about what the score is going to be.

Max-Value now calls **Min-Value** on the left successor with the same values of alpha and beta.

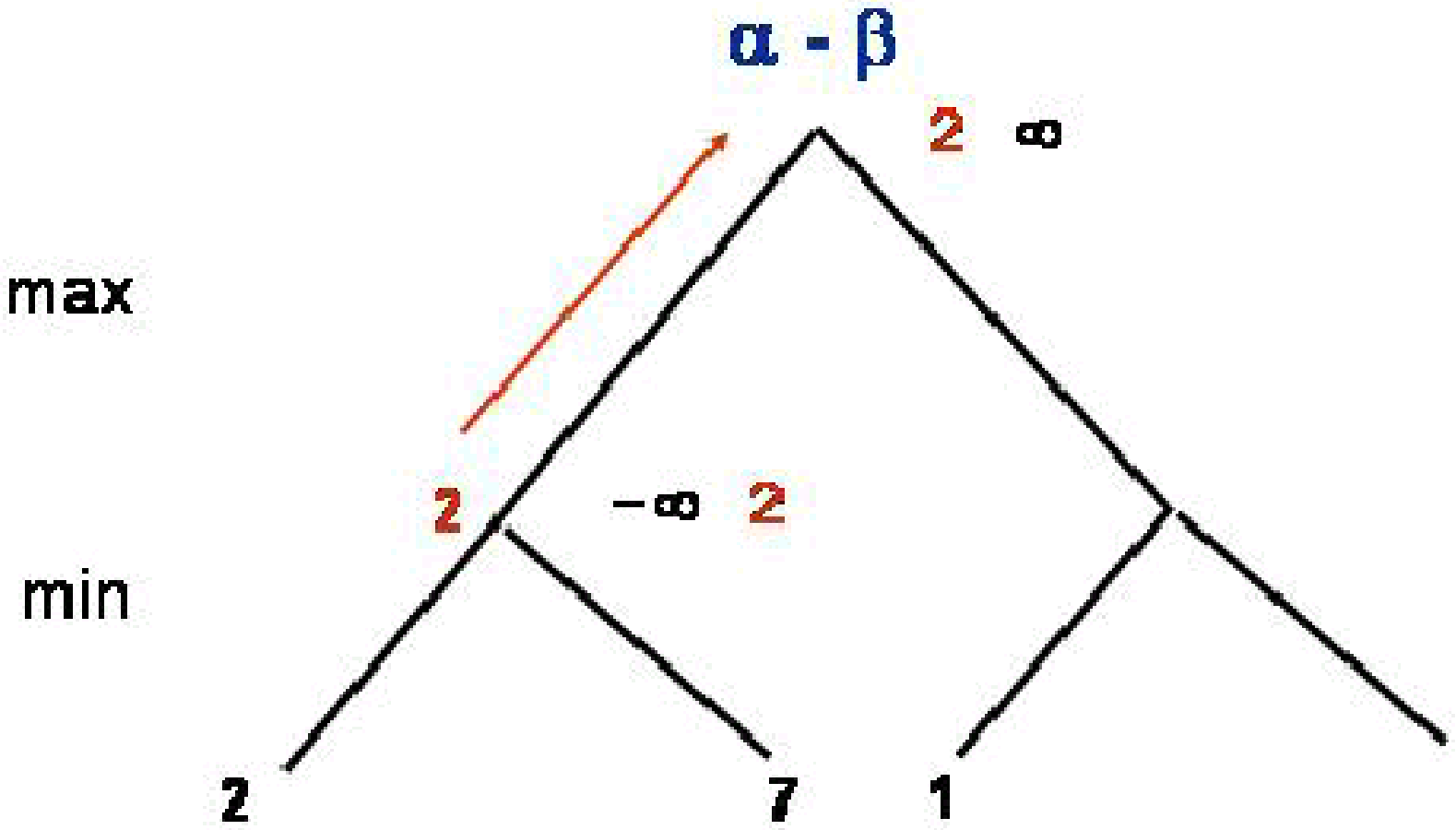
Min-Value now calls **Max-Value** on its leftmost successor.



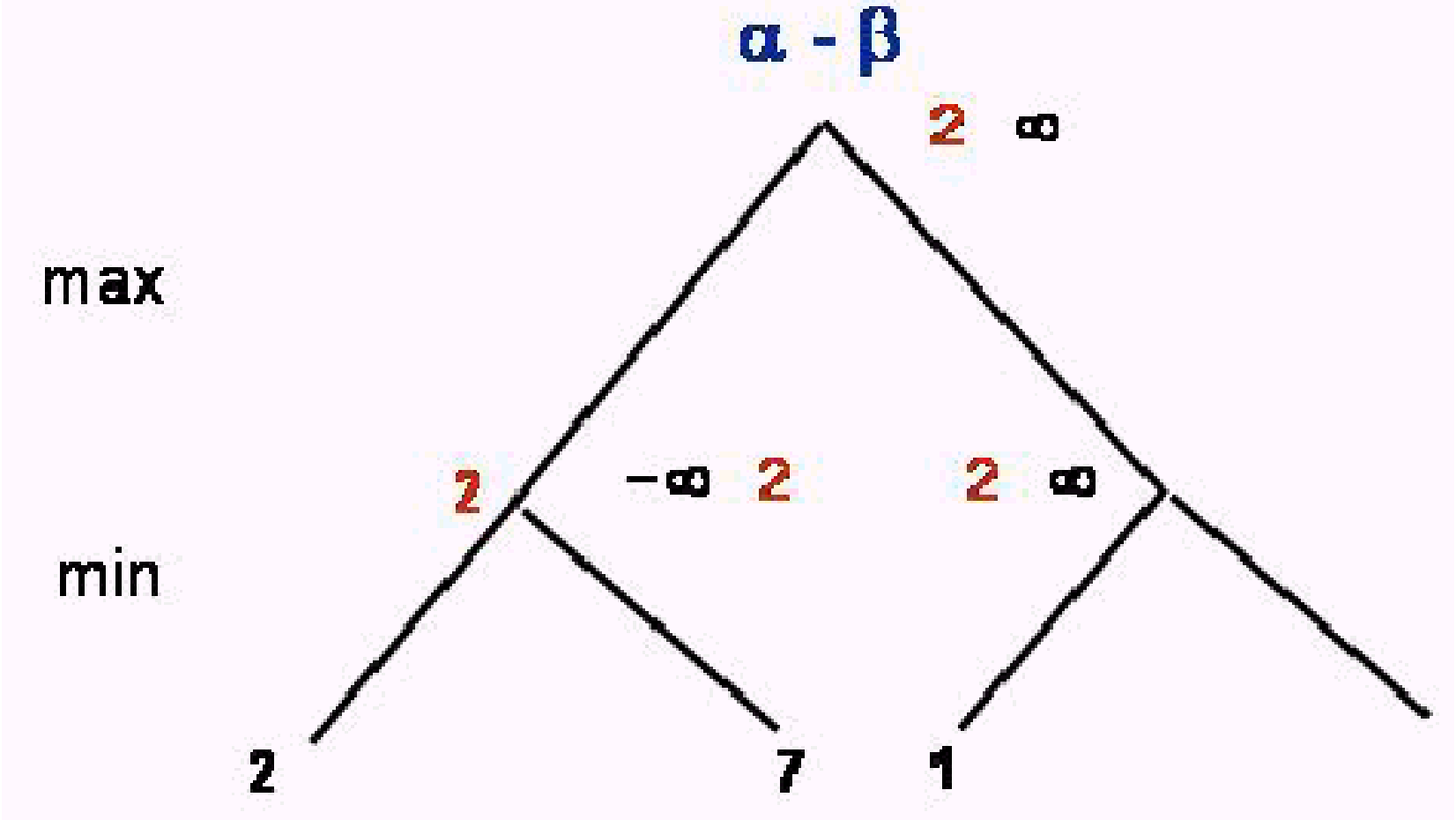
Max-Value is at the leftmost leaf, whose static value is 2 and so it returns that as alpha.

This first value, since it is less than infinity, becomes the new value of beta in **Min-Value**.

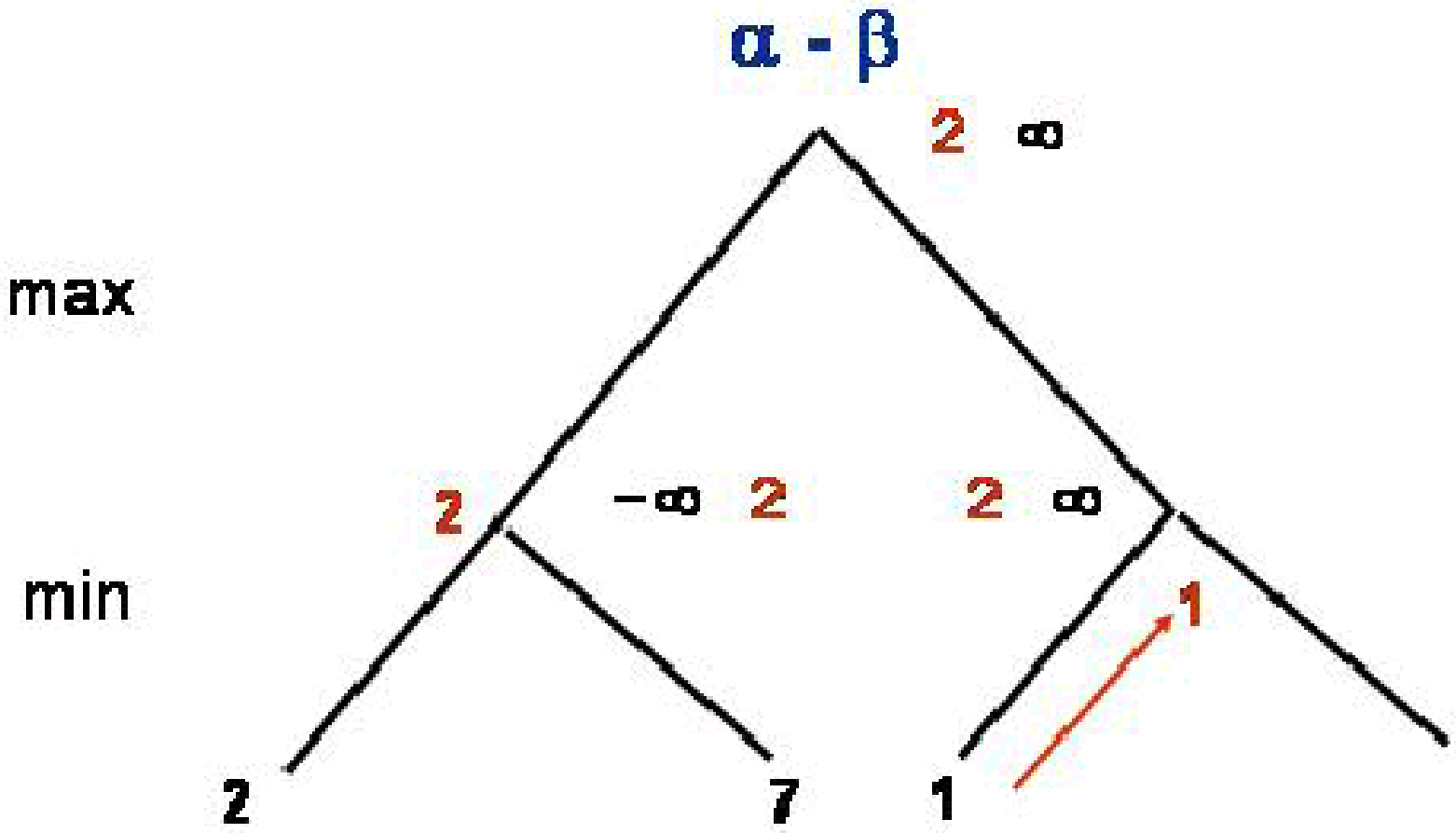
Now we call **Max-Value** with the next successor, which is also a leaf whose value is 7.
7 is not less than 2 and so the final value of beta is 2 for this node.



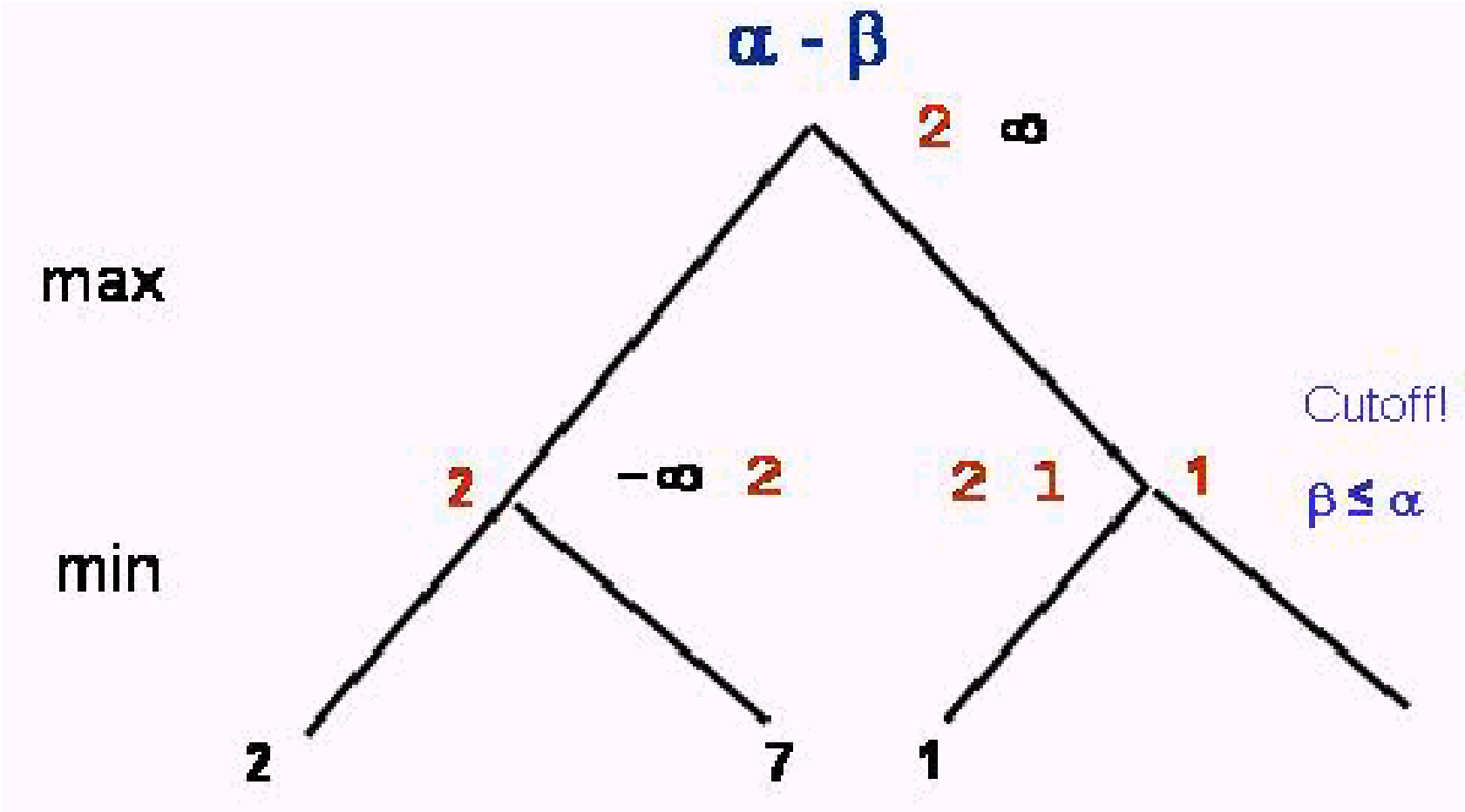
Min-Value now returns this value to its caller.
The calling **Max-Value** now sets alpha to this value, since it is bigger than minus infinity.
Note that the range of [alpha beta] says that the score will be greater or equal to 2 (and less than infinity).



Max-Value now calls **Min-Value** on the right successor with the updated range of [alpha beta].



Min-Value calls **Max-Value** on the left leaf and it returns a value of 1.



This is used to update beta in **Min-Value**, since it is less than infinity.
Note that at this point we have a range where alpha (2) is greater than beta (1).
This situation signals a *cutoff* in **Min-Value** and it returns beta (1), **without looking at the right leaf**.
So, a total of 3 static evaluations were needed instead of the 4 we would have needed under pure **Min-Max**.

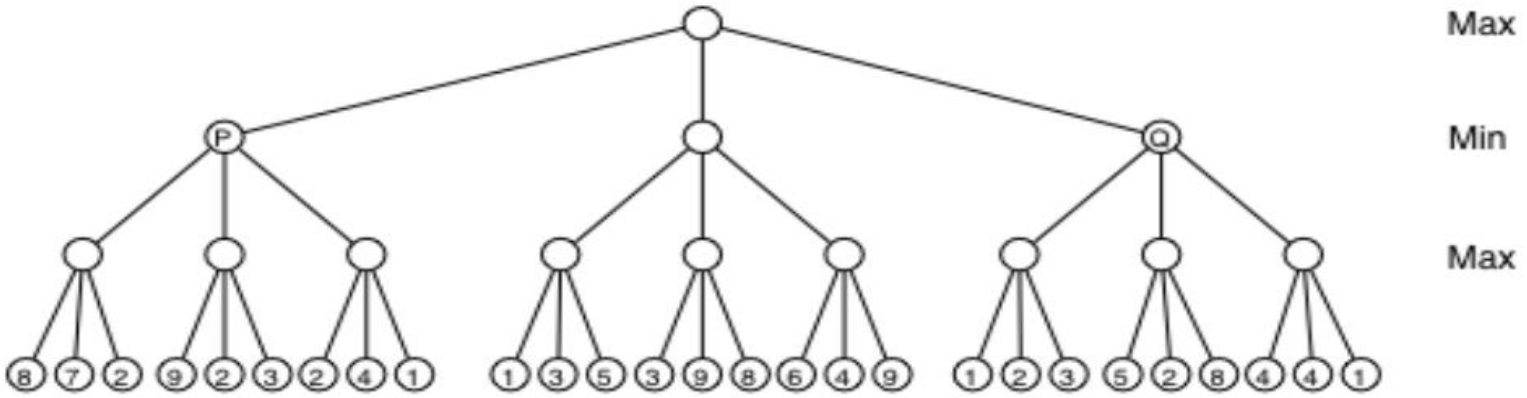
alpha-beta pruning

There are a couple of key points to remember about α-β pruning:

- Guaranteed to return exactly the same value as the Min-Max algorithm.
  - It is a pure optimization without any approximations or tradeoffs.

- In a perfectly ordered tree, with the best moves on the left, alpha beta reduces the cost of the search from order b^d to order $b^{(d/2)}$, that is, we can search **twice** as deep!
  - We already saw the enormous impact of deeper search on performance…
  - Now, this analysis is optimistic, since if we could order moves perfectly, we would not need alpha-beta. But, in practice, performance is close to the optimistic limit.

Example



(a) What is the final value of this game?

Answer: 5, which can be determined from a quick application of the minimax algorithm.

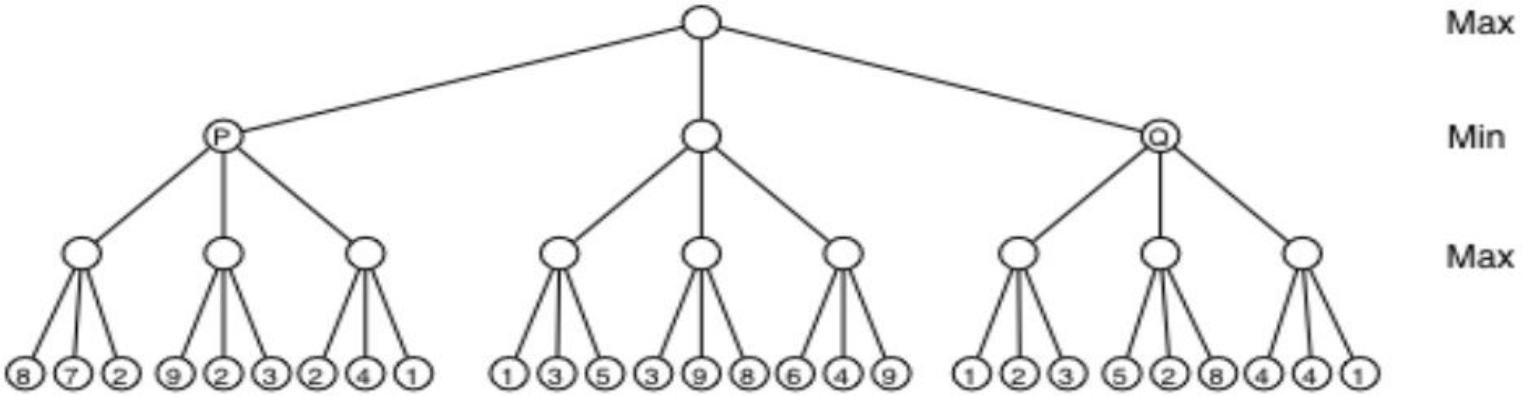
(b) Is the final value of beta at the root node (after all children have been visited) $+\infty$? (T/F)

Answer: True. The root node is a maximizing node, and the value of beta never changes at a maximizing node.

(c) What is the final value of beta at node P (after all of P's children have been visited)?

Answer: Since we are at a minimizing node and alpha is negative infinity, beta will take on the smallest value returned by any of P's children. In this case, P's children would return 8, 9, and 4, so the final value of beta at P will be 4.

Example



(d) Will any nodes be pruned?

Answer: Yes.

Suppose we are in the middle of running the algorithm. The algorithm has just reached the node labeled Q. The value of alpha is 5 and the value of beta is $+\infty$.

If we work out the algorithm on the leftmost subtree of Q, we see that it returns a value of 3. Beta at Q will then become 3, and 3 is less than the alpha value at Q, so Q will immediately return a value to its parent. This means that the other two subtrees of Q are pruned.

(e) What value will Q return to its parent?

Answer: Since its beta value after visiting the leftmost subtree was less than its alpha value of 5, Q will return its alpha value of 5.

Cutting-off search

// α is the best score for MAX, β is the best score for MIN

// initial call is **MAX-VALUE (state, -∞, ∞, MAX_DEPTH)**

MAX-VALUE (state, α, β, depth)

~~if (depth == 0) return EVAL(state)~~

α=-∞

for each s in SUCCESSORS(state) do

α = MAX (α, MIN-VALUE (s, α, β, depth-1))

if α≥β return α // cutoff

return α

MIN-VALUE (state, α, β, depth)

~~if (depth == 0) return EVAL(state)~~

β=+∞

for each s in SUCCESSORS(state) do

β = MIN (β, MAX-VALUE (s, α, β, depth-1))

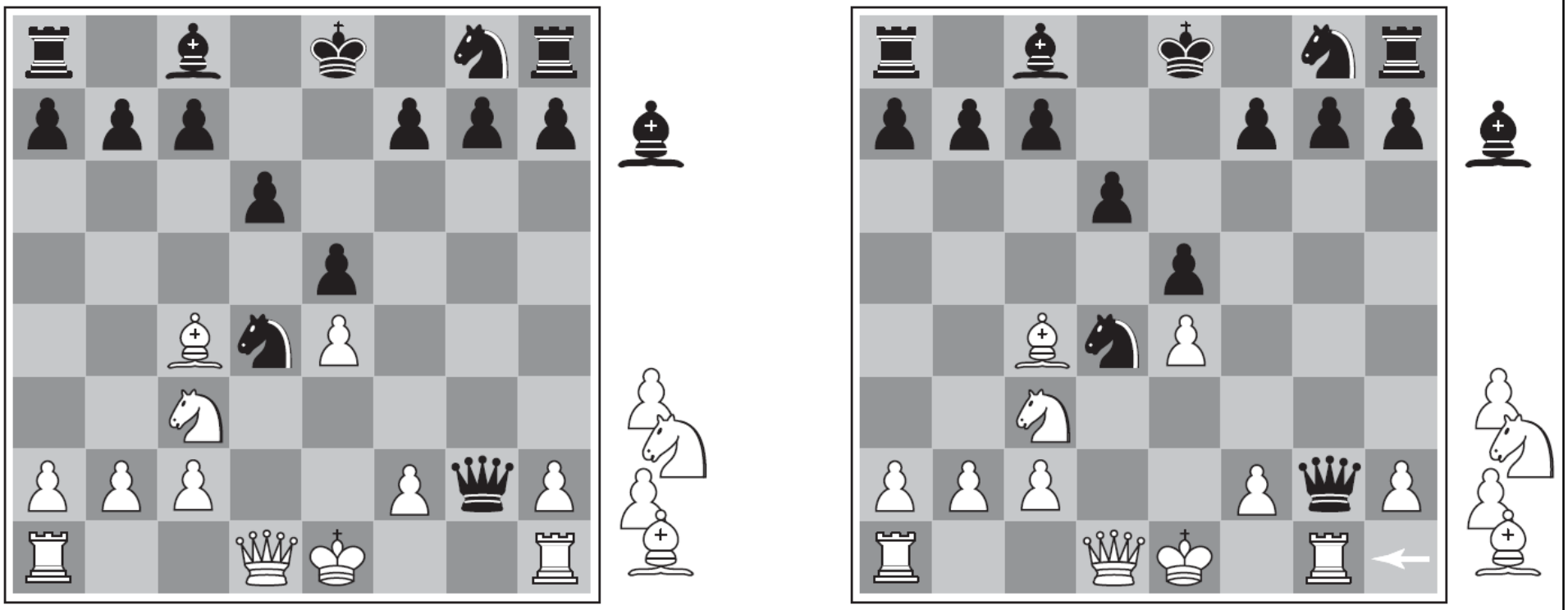
if α≥β return β // cutoff

return β

if CUTOFF-TEST(state, depth)
then return EVAL(state)

Most straightforward approach to controlling the amount of search is for CUTOFF-TEST(state, depth) to return true for depth==0…
but can lead to errors due to the approximate nature of the evaluation function.

Problem 1: non-quiescent states



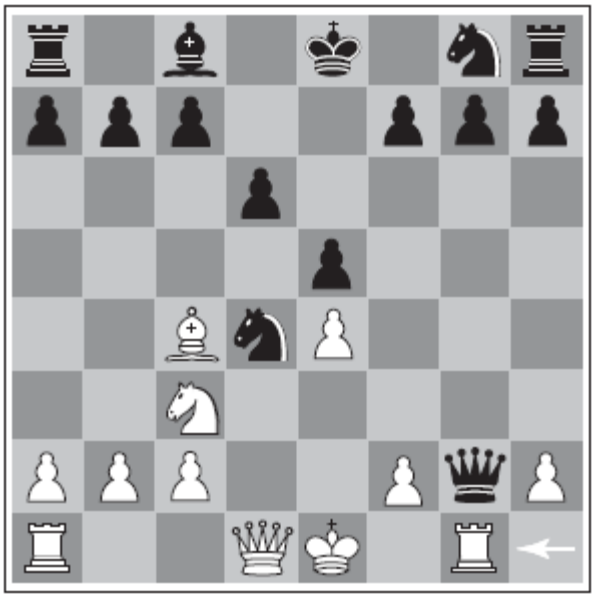
(a) White to move

(b) White to move

Figure 5.8 Two chess positions that differ only in the position of the rook at lower right. In (a), Black has an advantage of a knight and two pawns, which should be enough to win the game. In (b), White will capture the queen, giving it an advantage that should be strong enough to win.

Problem 1: non-quiescent states

- Suppose the program searches to the depth limit, reaching the position in figure (b), where Black is ahead by a knight and two pawns. It would report this as the heuristic value of the state, thereby declaring that the state is a probable win by Black.
- But White's next move captures Black's queen with no compensation. Hence, the position is really won for White, but this can be seen only by looking ahead one more ply.



(b) White to move

non-quiescent state: Wild swings in value in the near future

Problem 1: non-quiescent states

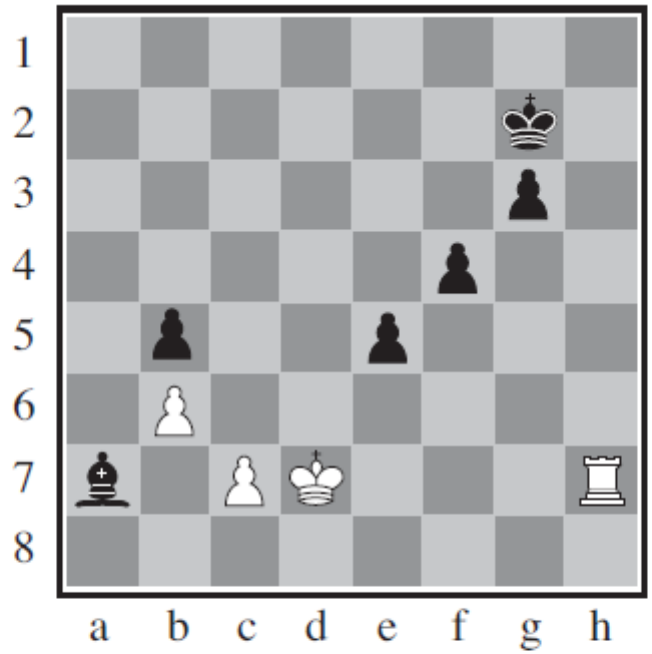
Solution

- **Apply evaluation function** only to positions that are **quiescent**

- **Expand nonquiescent positions** further until quiescent positions are reached.
  - This extra search is called a **quiescence search**;
  - Restricted to consider only certain types of moves, such as capture moves, that will quickly resolve the uncertainties in the position.

Problem 2: Horizon effect

- Arises when the program is facing an opponent's move that causes serious damage and is ultimately unavoidable, but can be temporarily avoided by delaying tactics.



Solution: design a smarter cuttoff function. e.g. one that would discount in this example the existence of the black bishop.

Figure 5.9 The horizon effect. With Black to move, the black bishop is surely doomed. But Black can forestall that event by checking the white king with its pawns, forcing the king to capture the pawns. This pushes the inevitable loss of the bishop over the horizon, and thus the pawn sacrifices are seen by the search algorithm as good moves rather than bad ones.

Search vs Lookup

- It’s overkill for a chess program to start a game by considering a tree of a billion game states, only to conclude that it will move its pawn to e4.
- Books describing good play in the opening and endgame in chess have been available for about a century.
- Many game-playing programs use *table lookup* rather than search for the opening and ending of games.
- Usually after ten moves we end up in a rarely seen position, and the program must switch from table lookup to search.
- Near the end of the game there are again fewer possible positions, and thus more chance to do lookup.
- But here it is the computer that has the expertise: computer analysis of endgames goes far beyond anything achieved by humans.

Endgames

- A human can tell you the general strategy for playing a **king-and-rook-versus-king (KRK)** endgame:
  - reduce the opposing king's mobility by squeezing it toward one edge of the board, using your king to prevent the opponent from escaping the squeeze.

- Other endings, such as **king**, **bishop**, and **knight** versus **king** (**KBNK**), are difficult to master and have no succinct strategy description.

- A computer, on the other hand, can completely solve the endgame by producing a **policy**, which is a mapping from every possible state to the best move in that state. Then we can just look up the best move rather than recompute it anew.

Endgames

- How big will the KBNK lookup table be?
  - It turns out there are 462 ways that two kings can be placed on the board without being adjacent. After the kings are placed, there are 62 empty squares for the bishop, 61 for the knight, and two possible players to move next, so there are $462 \times 62 \times 61 \times 2 = 3{,}494{,}568$ possible positions.

Ken Thomson and others

- Ken Thompson (1986, 1996) and Lewis Stiller (1992, 1996) solved all chess endgames with up to **five pieces** and some with **six pieces**, making them available on the Internet.
- Stiller discovered one case where a forced mate existed but required **262 moves**!!
  - Caused some consternation because the rules of chess require a capture or pawn move to occur within 50 moves.
- Marc Bourzutschky and Yakov Konoval (Bourzutschky, 2006) solved all pawnless six-piece and some seven-piece endgames;
  - there is a KQNKRBN endgame that with best play requires 517 moves until a capture, which then leads to a mate.
- If we could extend the chess endgame tables from 6 pieces to 32, then White would know on the opening move whether it would be a win, loss, or draw.
  - Not happened so far for chess, but has happened for checkers!

Ken Thompson

Hacker

Kenneth Lane Thompson, commonly referred to as ken in hacker circles, is an American pioneer of computer science. Having worked at Bell Labs for most of his career, Thompson designed and implemented the original Unix operating system. Wikipedia

Born: February 4, 1943 (age 75 years), New Orleans, Louisiana, United States

Awards: Turing Award, MORE

Known for: Unix, B, Belle, UTF-8, Endgame tablebase, Go

Education: University of California, Berkeley (1966), University of California, Berkeley (1965)

Quotes View 1+ more

When in doubt, use brute force.

You can't trust code that you did not totally create yourself.

Stochastic Games

- In real life, **unpredictable** external events can put us into **unforeseen situations**.

- Many games mirror this unpredictability by including a **random element**, such as the throwing of **dice**.
  - We call these **stochastic games**.

- **Backgammon** is a typical example; a game that combines luck and skill.
  - Dice are rolled at the beginning of a player's turn to determine the legal moves.

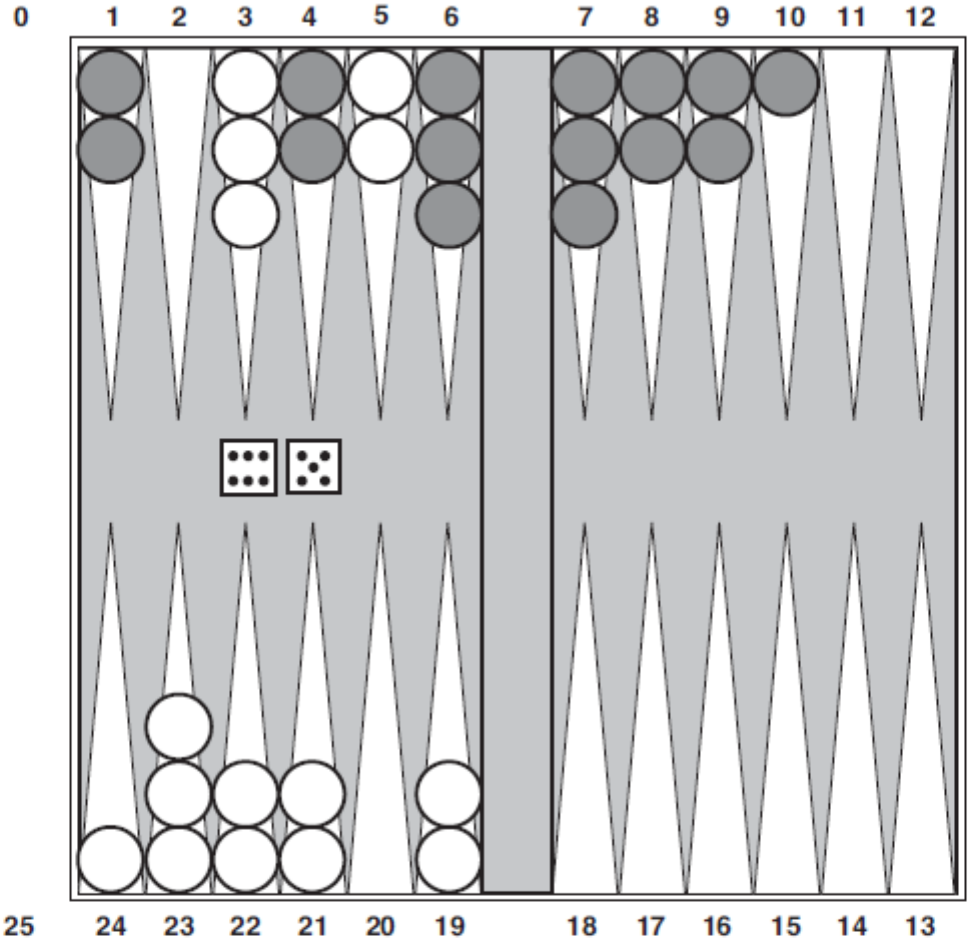


Figure 5.10 A typical backgammon position. The goal of the game is to move all one's pieces off the board. White moves clockwise toward 25, and Black moves counterclockwise toward 0. A piece can move to any position unless multiple opponent pieces are there; if there is one opponent, it is captured and must start over. In the position shown, White has rolled 6–5 and must choose among four legal moves: (5–10,5–11), (5–11,19–24), (5–10,10–16), and (5–11,11–16), where the notation (5–11,11–16) means move one piece from position 5 to 11, and then move a piece from 11 to 16.

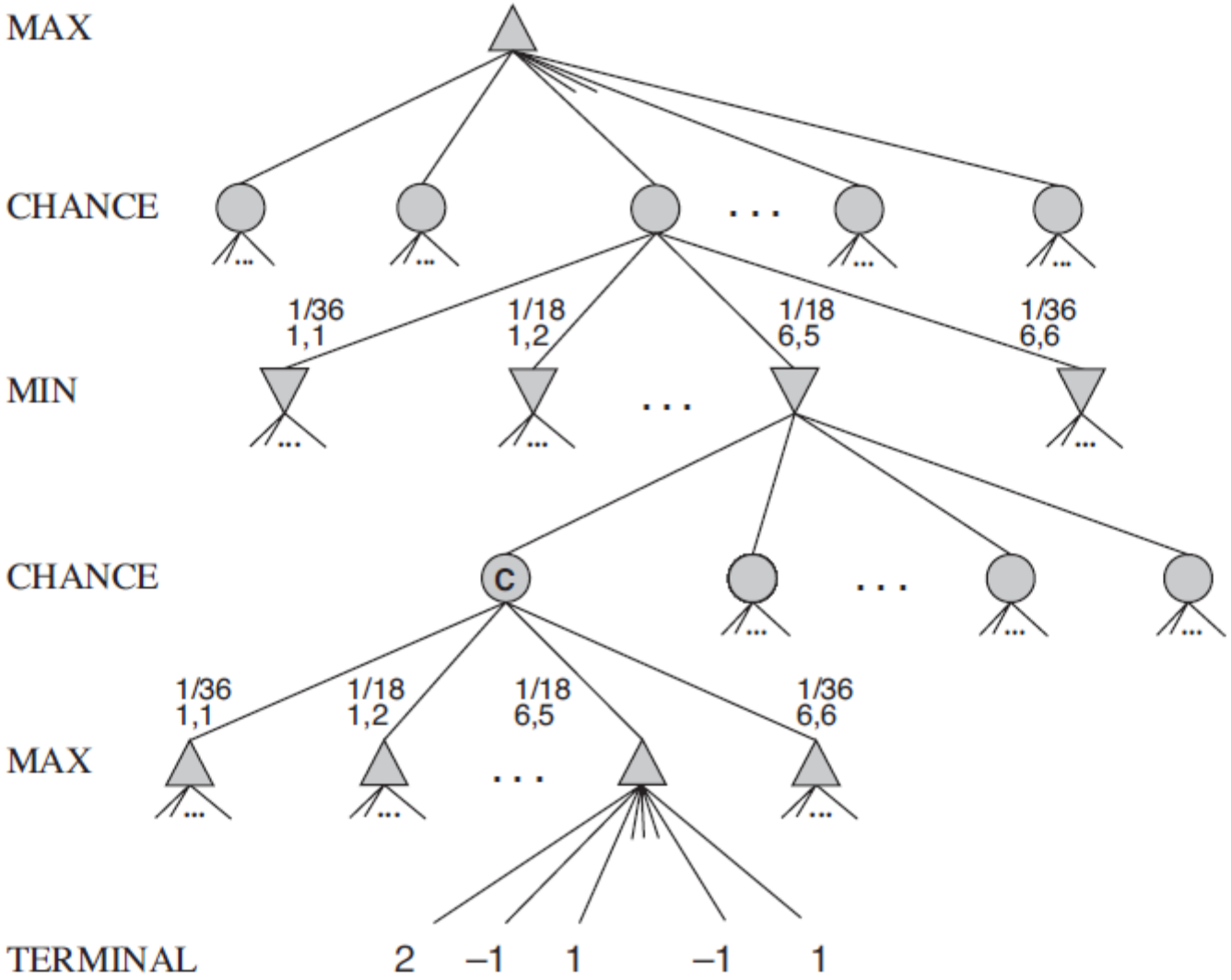


Figure 5.11 Schematic game tree for a backgammon position.

Expectiminimax

$$\textsc{Expectiminimax}(s) = \begin{cases} \textsc{Utility}(s) & \text{if } \textsc{Terminal-Test}(s) \\ \max_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{max} \\ \min_a \textsc{Expectiminimax}(\textsc{Result}(s,a)) & \text{if } \textsc{Player}(s) = \textsc{min} \\ \sum_r P(r)\textsc{Expectiminimax}(\textsc{Result}(s,r)) & \text{if } \textsc{Player}(s) = \textsc{chance} \end{cases}$$

Evaluation functions for games of chance

- As with minimax, the obvious approximation to make with **expectiminimax** is to cut the search off at some point and apply an evaluation function to each leaf.

- We might think that evaluation functions for games such as backgammon should be just like evaluation functions for chess—they **just need to give higher scores to better positions**.

- In fact, the presence of chance nodes means that one has to be **more careful** about what the evaluation values mean.

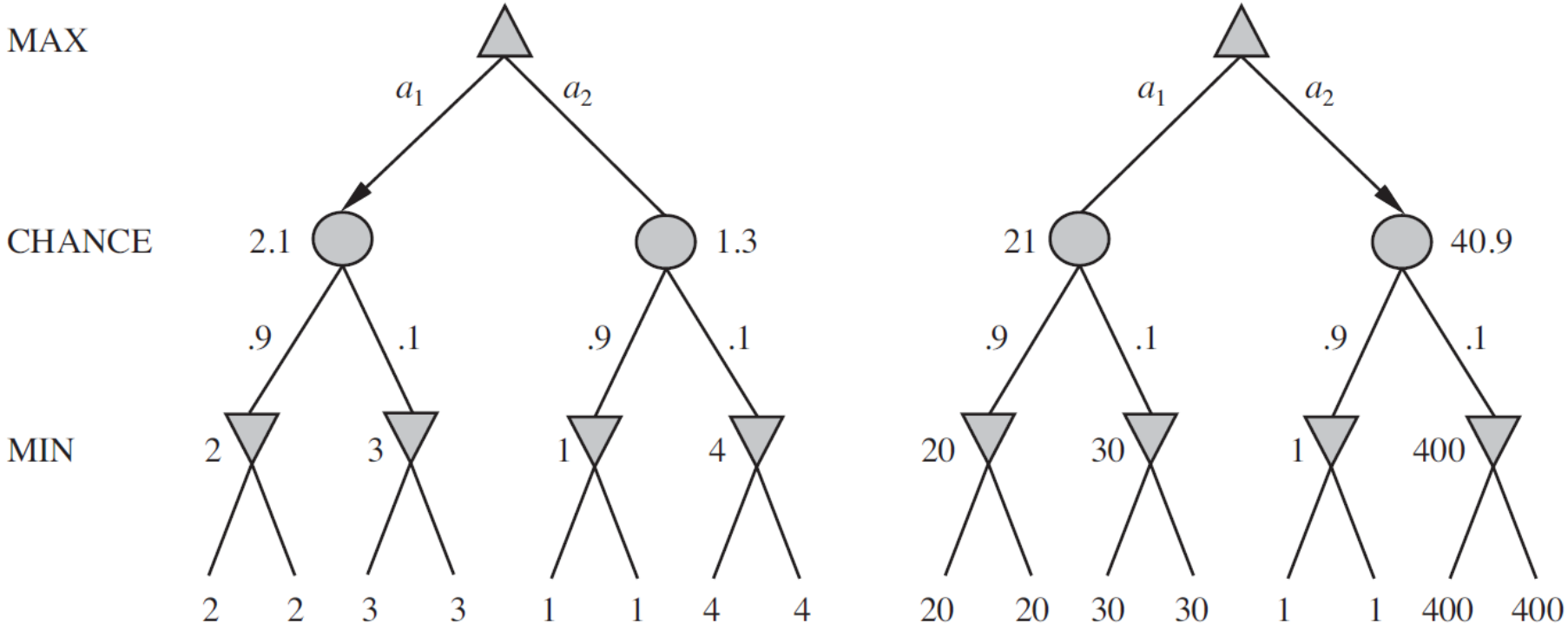


Figure 5.12 An order-preserving transformation on leaf values changes the best move.

With an evaluation function that assigns the values [1, 2, 3, 4] to the leaves, move a_1 is best; with values [1, 20, 30, 400], move a_2 is best.

Hence, the program behaves totally differently if we make a change in the scale of some evaluation values!

To avoid this sensitivity, the evaluation function must be a **positive linear function** of the probability of winning from a position.

Partially Observable Games

- **Chess** described as **war in miniature**, but lacks one major characteristic of real wars, namely, **partial observability**.
- In the "fog of war," the existence and disposition of enemy units is often unknown until revealed by direct contact.
  - As a result, warfare includes the use of scouts and spies to gather information and the use of concealment and bluff to confuse the enemy.
- Partially observable games share these characteristics.
  - Main examples: card games (out of scope for this course)