Game Playing

Today's class

- Game playing
  - State of the art and resources
  - Framework
- Game trees
  - Minimax
  - Alpha-beta pruning
  - Adding randomness

Why study games?

- Clear criteria for success

- Offer an opportunity to study problems involving {hostile, adversarial, competing} agents.

- Historical reasons

- Fun

- Interesting, hard problems which require minimal "initial structure"

- Games often define very large search spaces
  - chess 35^{100} nodes in search tree, 10^{40} legal states

State of the art

- How good are computer game players?
  - **Chess**:
    - Deep Blue beat Gary Kasparov in 1997
    - Garry Kasparav vs. Deep Junior (Feb 2003): tie!
    - Kasparov vs. X3D Fritz (November 2003): tie!
      http://www.thechessdrum.net/tournaments/Kasparov-X3DFritz/index.html
    - Deep Fritz beat world champion Vladimir Kramnik (2006)
  - **Checkers**: Chinook (an AI program with a *very large* endgame database) is the world champion and can provably never be beaten. Retired in 1995
  - **Go**: Computer players have finally reached tournament-level play
  - **Bridge**: "Expert-level" computer players exist (but no world champions yet!)
- Good places to learn more:
  - http://www.cs.ualberta.ca/~games/
  - http://www.cs.unimass.nl/icga

Typical case

- 2-person game
- Players alternate moves
- **Zero-sum**: one player's loss is the other's gain
- **Perfect information**: both players have access to complete information about the state of the game. No information is hidden from either player.
- No chance (e.g., using dice) involved
- Examples: Tic-Tac-Toe, Checkers, Chess, Go, Nim, Othello
- Not: Bridge, Solitaire, Backgammon, ...

How to play a game

- A way to play such a game is to:
  - Consider all the legal moves you can make
  - Compute the new position resulting from each move
  - Evaluate each resulting position and determine which is best
  - Make that move
  - Wait for your opponent to move and repeat
- Key problems are:
  - Representing the "board"
  - Generating all legal next boards
  - Evaluating a position

Evaluation function

- **Evaluation function** or **static evaluator** is used to evaluate the "goodness" of a game position.
  - Contrast with heuristic search where the evaluation function was a non-negative estimate of the cost from the start node to a goal and passing through the given node
- The zero-sum assumption allows us to use a single evaluation function to describe the goodness of a board with respect to both players.
  - **f(n) >> 0**: position n good for me and bad for you
  - **f(n) << 0**: position n bad for me and good for you
  - **f(n) near 0**: position n is a neutral position
  - **f(n) = +infinity**: win for me
  - **f(n) = -infinity**: win for you

Evaluation function examples

- Example of an evaluation function for Tic-Tac-Toe:

  f(n) = [# of 3-lengths open for me] - [# of 3-lengths open for you]
  where a 3-length is a complete row, column, or diagonal

- Alan Turing's function for chess

  - **f(n) = w(n)/b(n)** where w(n) = sum of the point value of white's pieces and b(n) = sum of black's

- Most evaluation functions are specified as a weighted sum of position features:

  $f(n) = w_1 * \text{feat}_1(n) + w_2 * \text{feat}_2(n) + ... + w_n * \text{feat}_k(n)$

- Example features for chess are piece count, piece placement, squares controlled, etc.

- Deep Blue had over 8000 features in its evaluation function

Game trees

- Problem spaces for typical games are represented as trees

- Root node represents the current board configuration; player must decide the best single move to make next

- **Static evaluator function** rates a board position. f(board) = real number with f>0 "white" (me), f<0 for black (you)

- Arcs represent the possible legal moves for a player

- If it is **my turn** to move, then the root is labeled a "**MAX**" node; otherwise it is labeled a "**MIN**" node, indicating **my opponent's turn**.

- Each level of the tree has nodes that are all MAX or all MIN; nodes at level i are of the opposite kind from those at level i+1
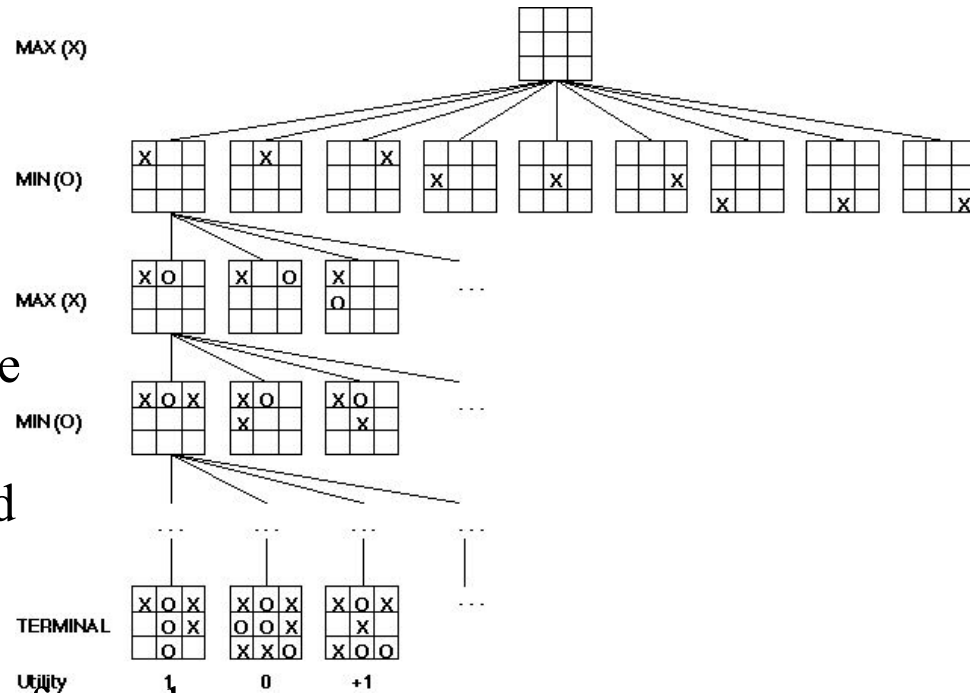
MinMax - Overview

- Search tree
  - *Squares* represent decision states (ie- after a move)
  - *Branches* are decisions (ie- the move)
  - Start at root
  - Nodes at end are leaf nodes
  - Ex: Tic-Tac-Toe (symmetrical positions removed)
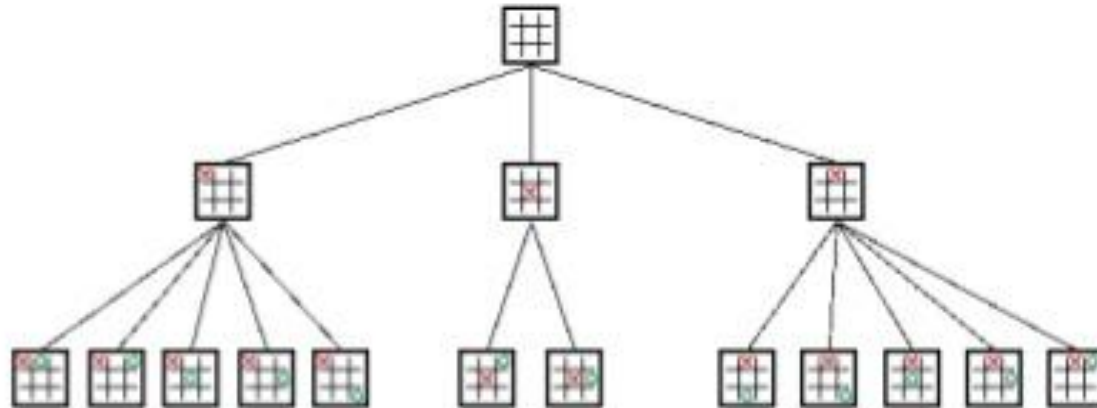
- Unlike binary trees can have any number of children
  - Depends on the game situation
- Levels usually called *plies* (a *ply* is one level)
  - Each ply is where "turn" switches to other player
- Players called *Min* and *Max* (next)

Minimax procedure

- Create start node as a MAX node with current board configuration

- Expand nodes down to some **depth** (a.k.a. **ply**) of lookahead in the game

- Apply the evaluation function at each of the leaf nodes

- "Back up" values for each of the non-leaf nodes until a value is computed for the root node
  - At MIN nodes, the backed-up value is the **minimum** of the values associated with its children.
  - At MAX nodes, the backed-up value is the **maximum** of the values associated with its children.

- Pick the operator associated with the child node whose backed-up value determined the value at the root
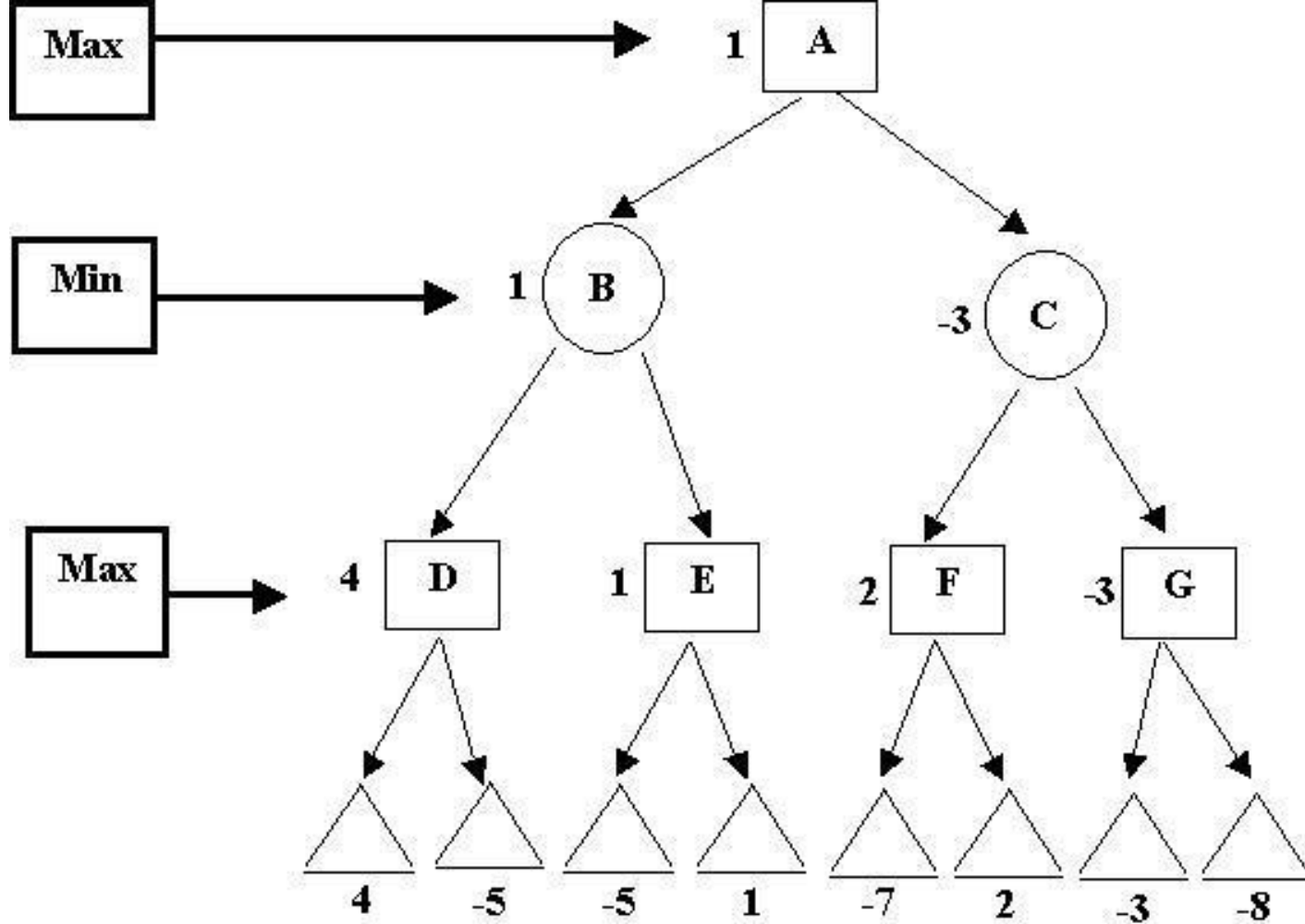
Minimax
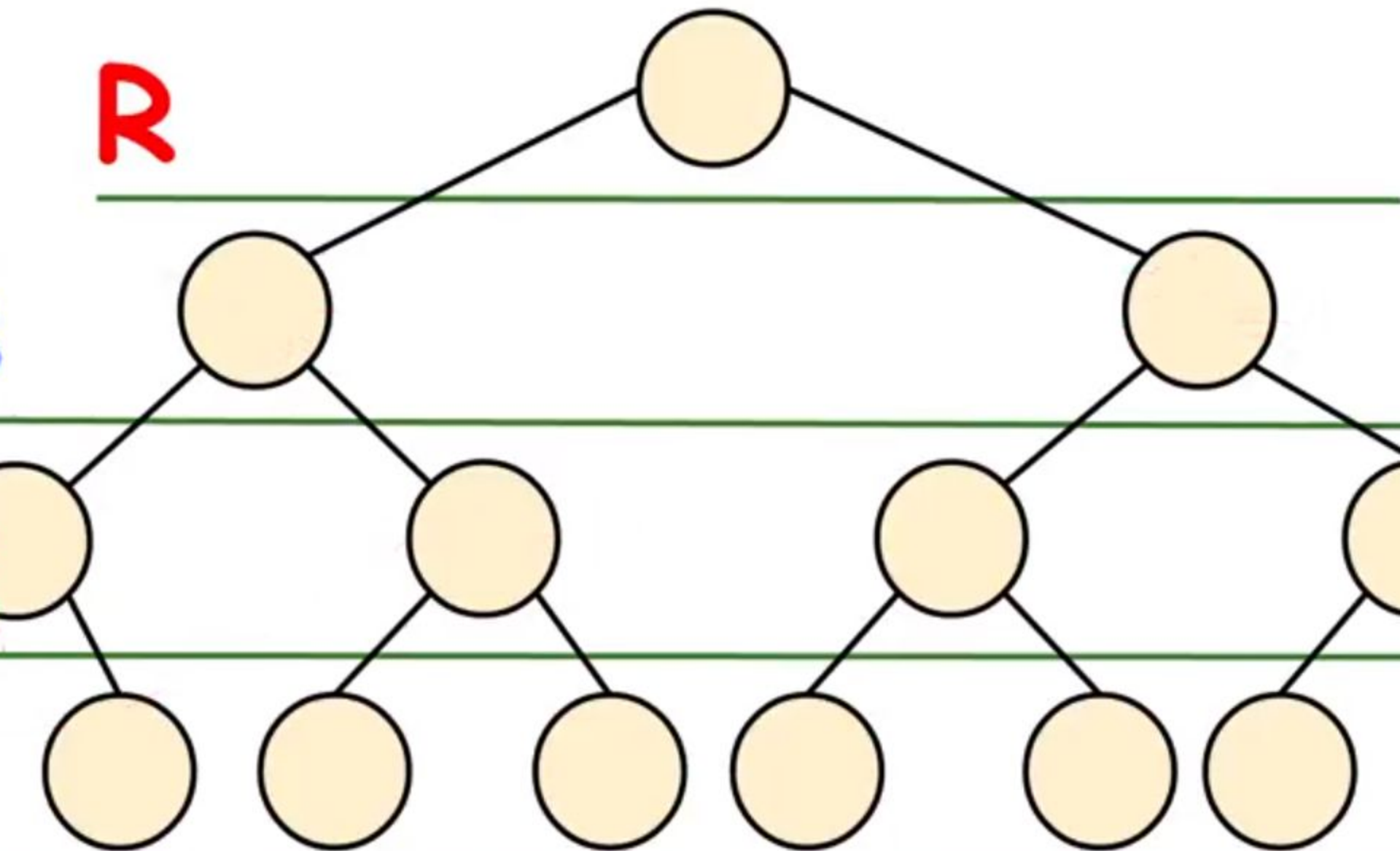
R

Minimax

R

Minimax

Minimax Algorithm

Static evaluator value

This is the move selected by minimax

MAX

MIN

Max ──────────────▶ 1 A

Min ──────────────▶ 1 B   -3 C

Max ──────────▶ 4 D   1 E   2 F   -3 G
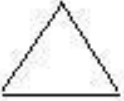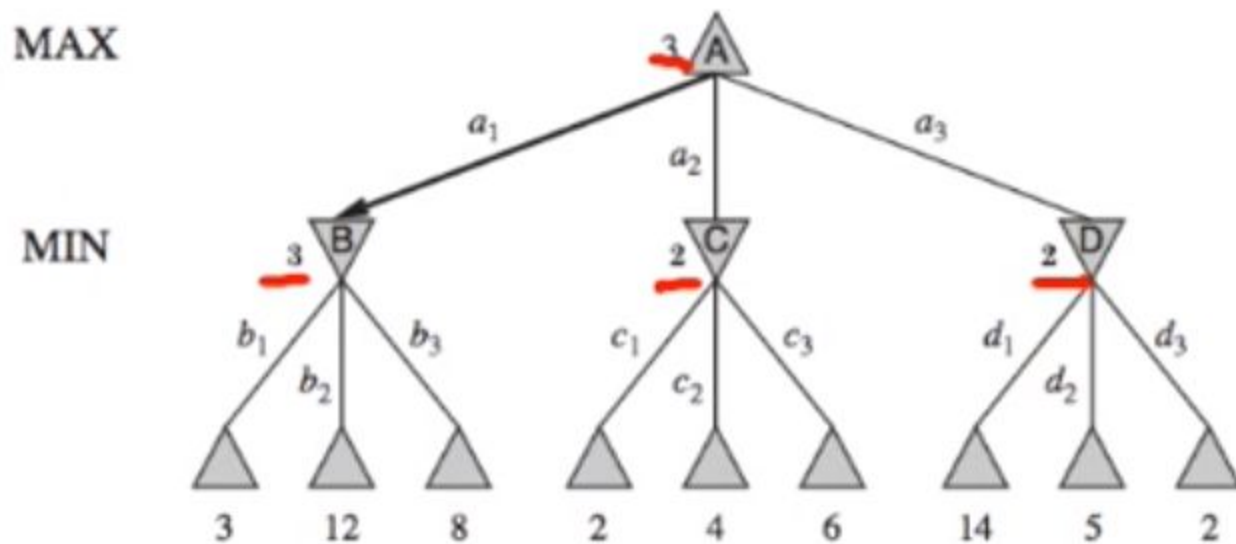
4   -5   -5   1   -7   2   -3   -8
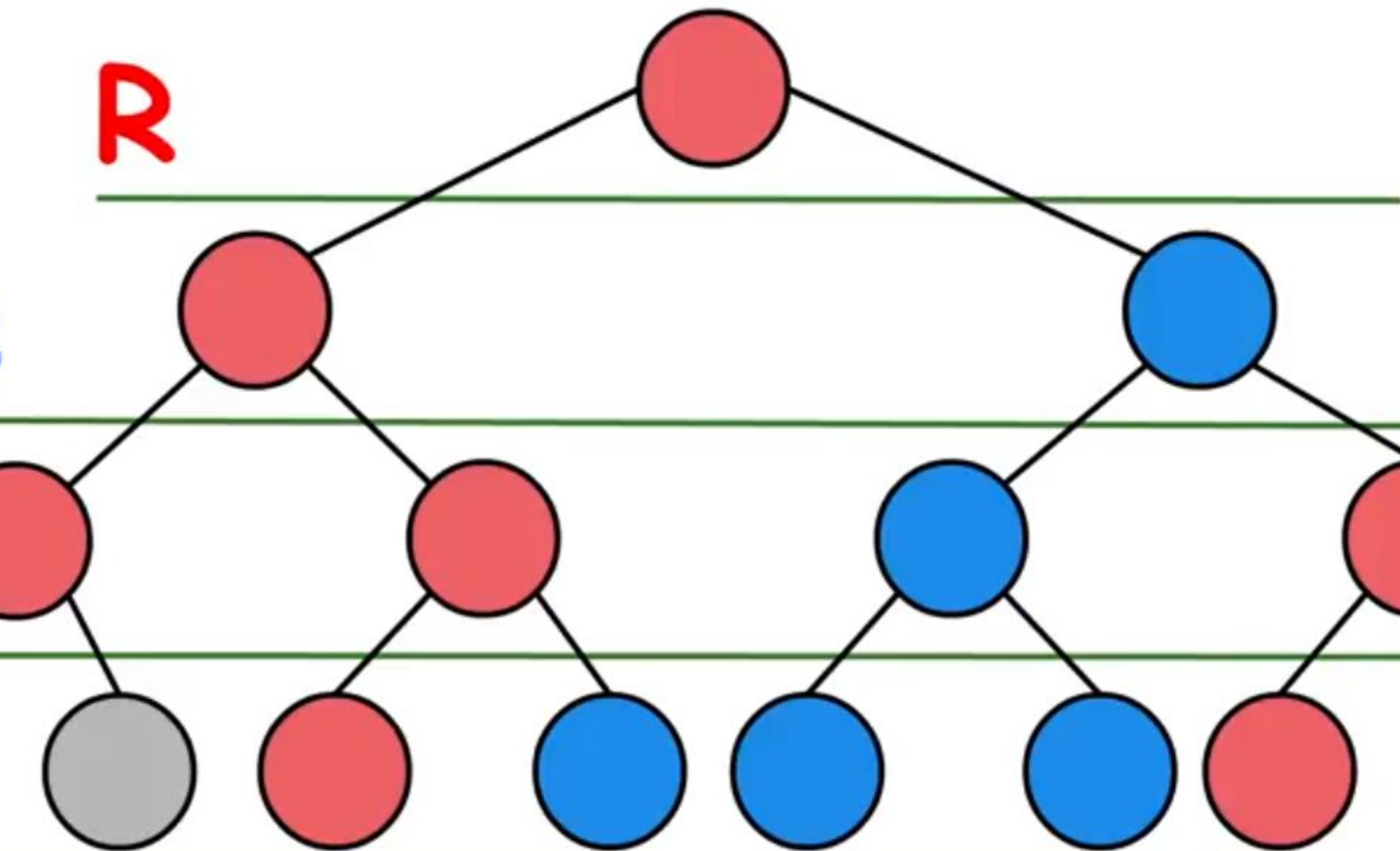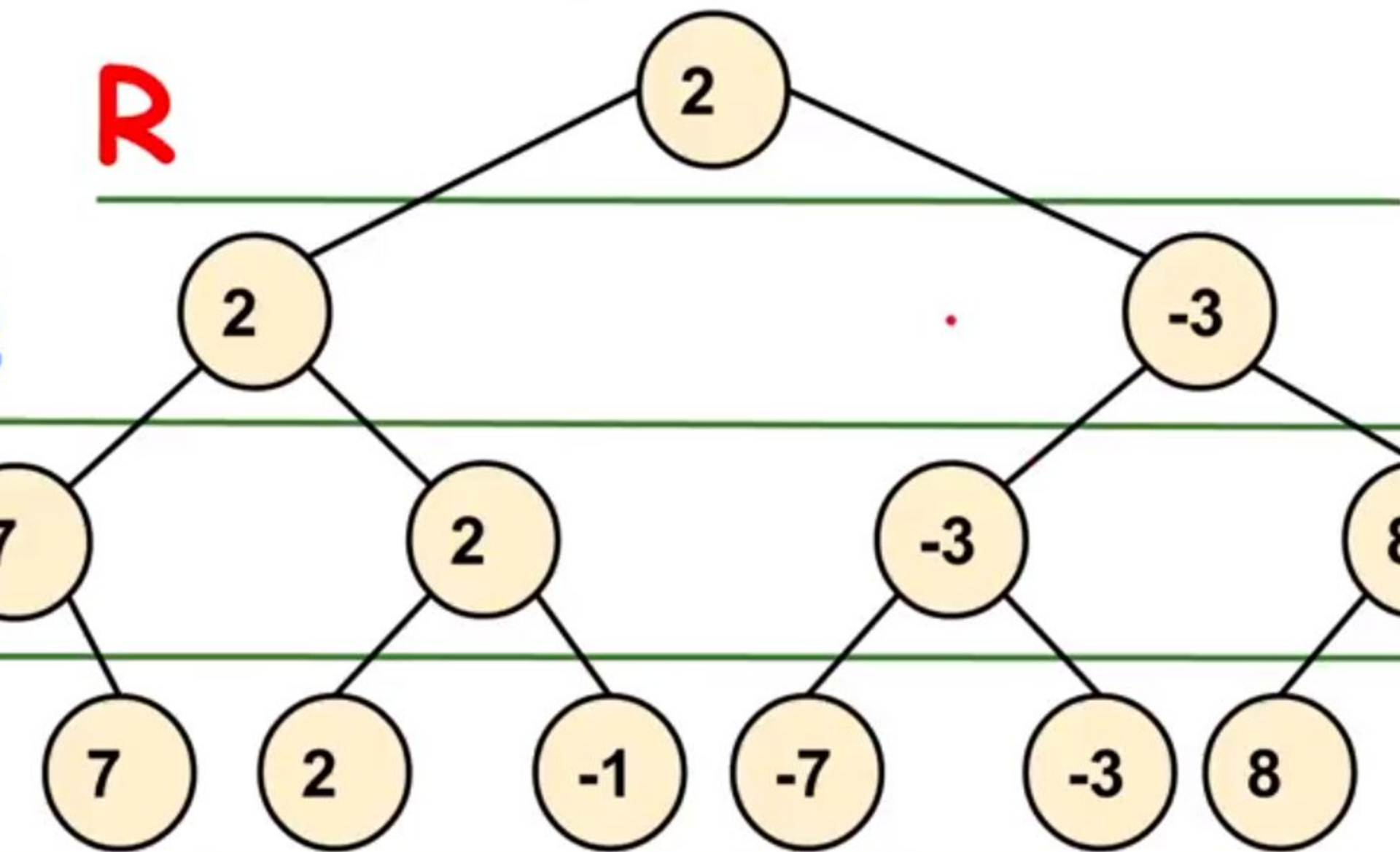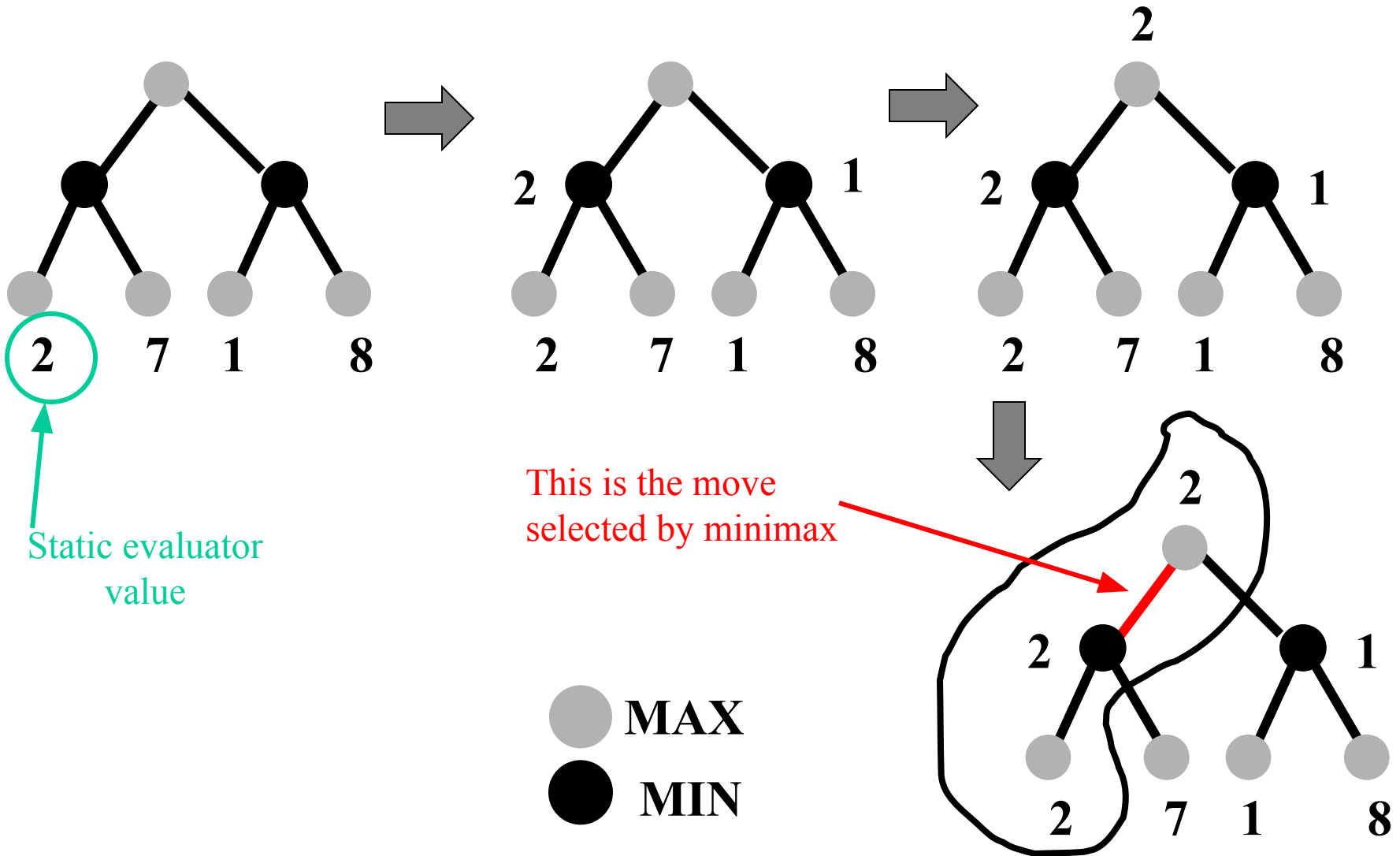
Terminal
Nodes

Computer
Move

Opponent
Move

Minimax
Picking my best move against your best move

$minimax(s) =$

$$
\begin{cases}
\underline{utility(s)} & \text{if } \underline{terminal(s)} \\
\max_{a \in action(s)} minimax(result(s, a)) & \text{if } \underline{player(s)} = MAX \\
\min_{a \in action(s)} minimax(result(s, a)) & \text{if } player(s) = MIN
\end{cases}
$$

Partial Game Tree for Tic-Tac-Toe
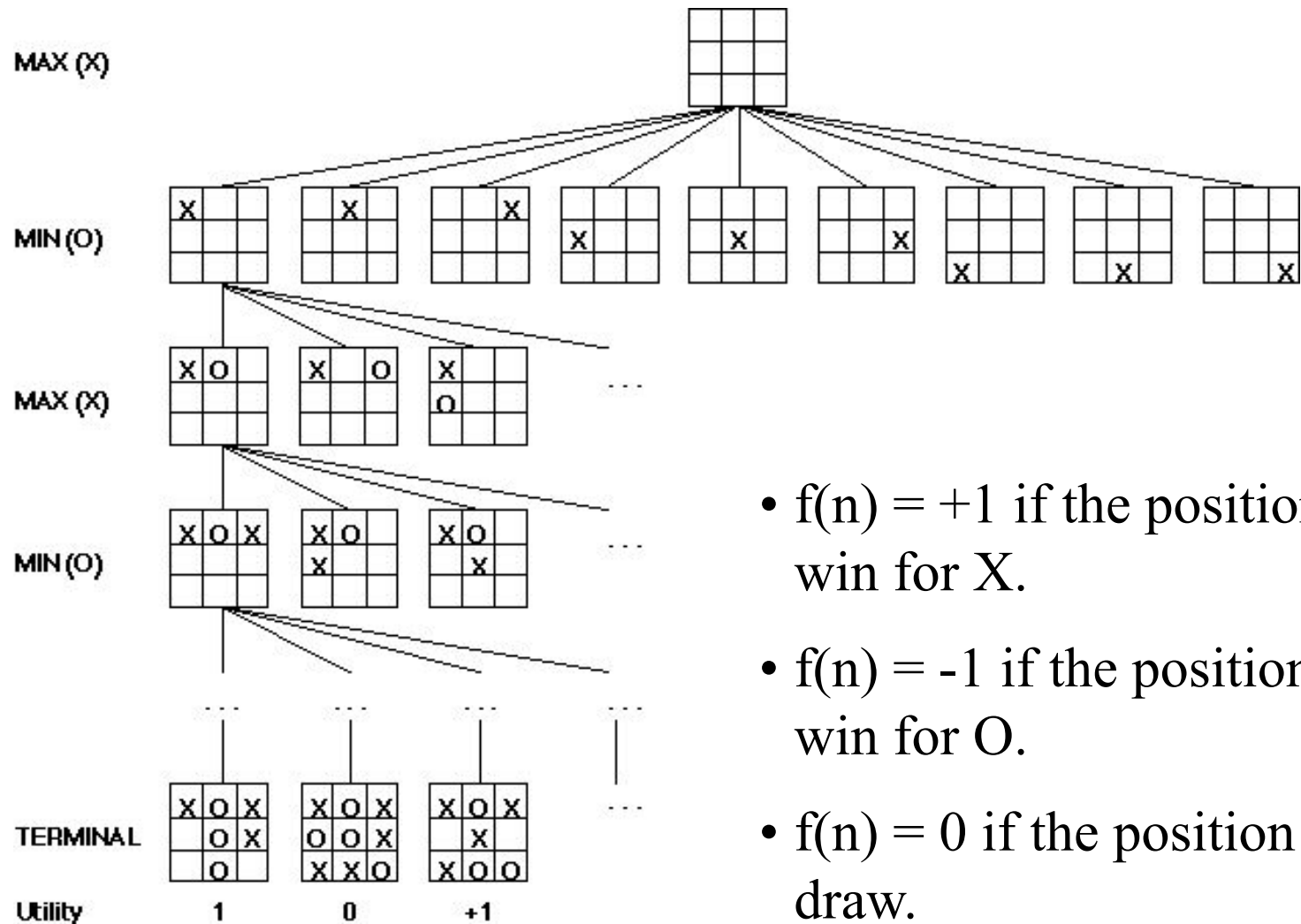
- f(n) = +1 if the position is a win for X.

- f(n) = -1 if the position is a win for O.

- f(n) = 0 if the position is a draw.

Minimax Tree
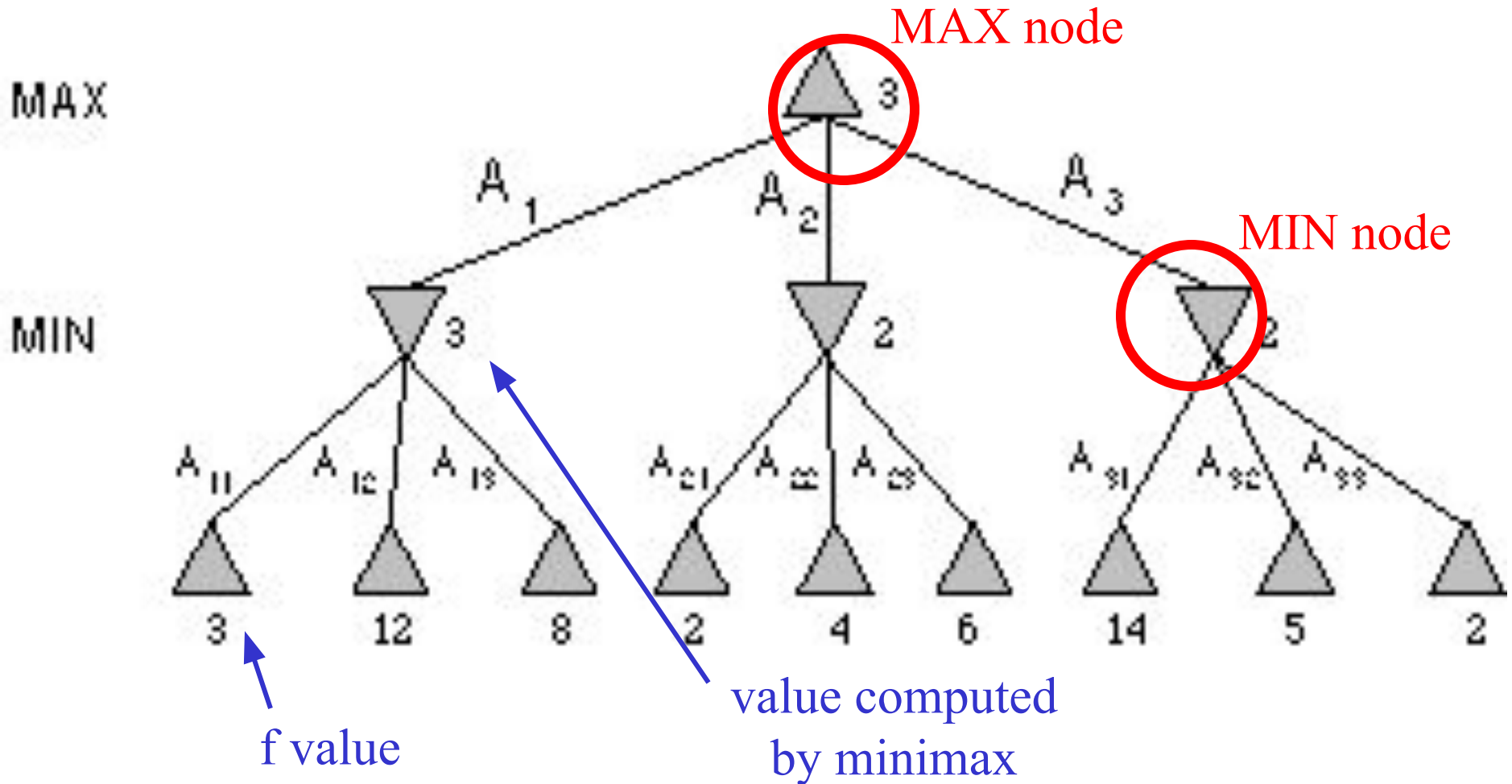
Minimax
Discussion

- ▶ Complete depth first exploration
- ▶ Depth m with b legal moves. $O(b^m)$
- ▶ Space complexity (memory) $O(bm)$
- ▶ Chess: $m \approx 35$; on average: $50 \leq b \leq 100$
- ▶ Impractical for most games, but basis of other algs.

Alpha-beta pruning

- We can improve on the performance of the minimax algorithm through **alpha-beta pruning**

- Basic idea: *"If you have an idea that is surely bad, don't take the time to see how truly awful it is."* -- Pat Winston
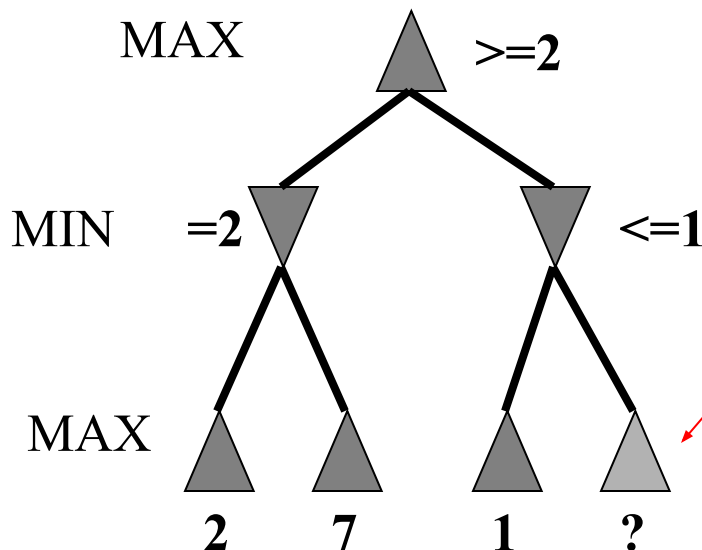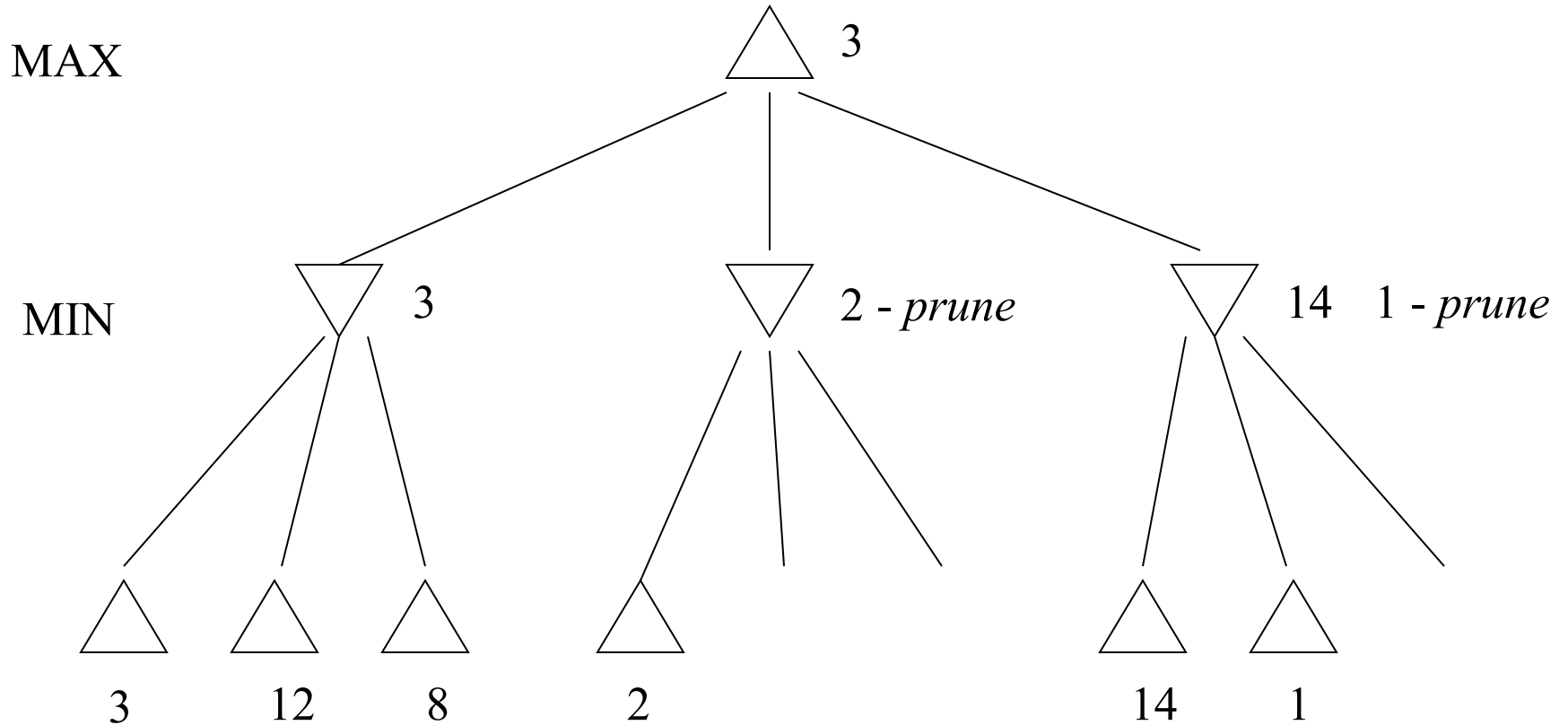

- We don't need to compute the value at this node.

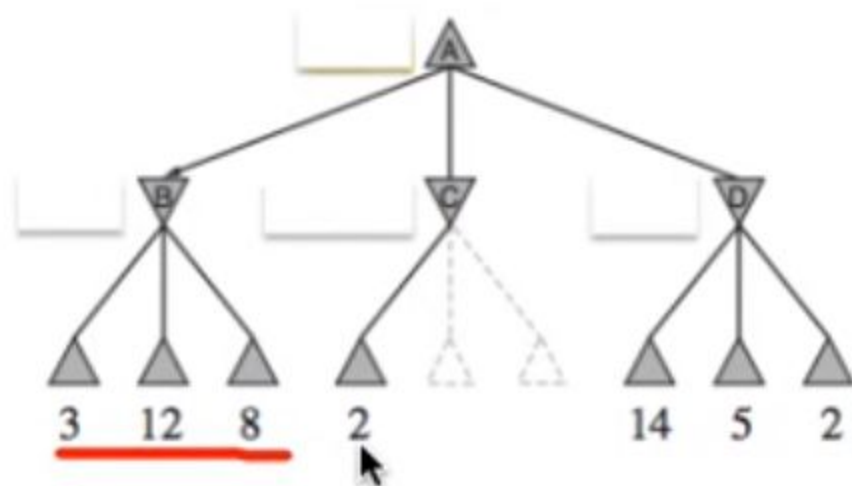- No matter what it is, it can't affect the value of the root node.

Alpha-beta pruning

- Traverse the search tree in depth-first order
- At each **MAX** node n, **alpha(n)** = maximum value found so far
- At each **MIN** node n, **beta(n)** = minimum value found so far
  - Note: The alpha values start at -infinity and only increase, while beta values start at +infinity and only decrease.
- **Beta cutoff**: Given a MAX node n, cut off the search below n (i.e., don't generate or examine any more of n's children) if alpha(n) >= beta(i) for some MIN node ancestor i of n.
- **Alpha cutoff:** stop searching below MIN node n if beta(n) <= alpha(i) for some MAX node ancestor i of n.

Alpha-beta example

MAX

MIN

3

3   2 - *prune*   14   1 - *prune*

3   12   8   2   14   1

Do we need to expand all nodes?

$$minimax(root) = max(min(\underline{3, 12, 8}), min(2, x, y), min(14, 5, 2))$$
$$= max(3, min(2, x, y), 2)$$
$$= max(3, z, 2)$$
$$= 3$$

Do we need z?

Alpha-beta algorithm

```
function MAX-VALUE (state, α, β)
    ;; α = best MAX so far; β = best MIN
if TERMINAL-TEST (state) then return UTILITY(state)
v := -∞
for each s in SUCCESSORS (state) do
    v := MAX (v, MIN-VALUE (s, α, β))
    if v >= β then return v
    α := MAX (α, v)
end
return v

function MIN-VALUE (state, α, β)
if TERMINAL-TEST (state) then return UTILITY(state)
v := ∞
for each s in SUCCESSORS (state) do
    v := MIN (v, MAX-VALUE (s, α, β))
    if v <= α then return v
    β := MIN (β, v)
end
return v
```

Two values:

- ▶ α = value of best choice so far for MAX (highest-value)
- ▶ β = value of best choice so far for MIN (lowest-value)
- ▶ Each node keeps track of its $[\alpha, \beta]$ values
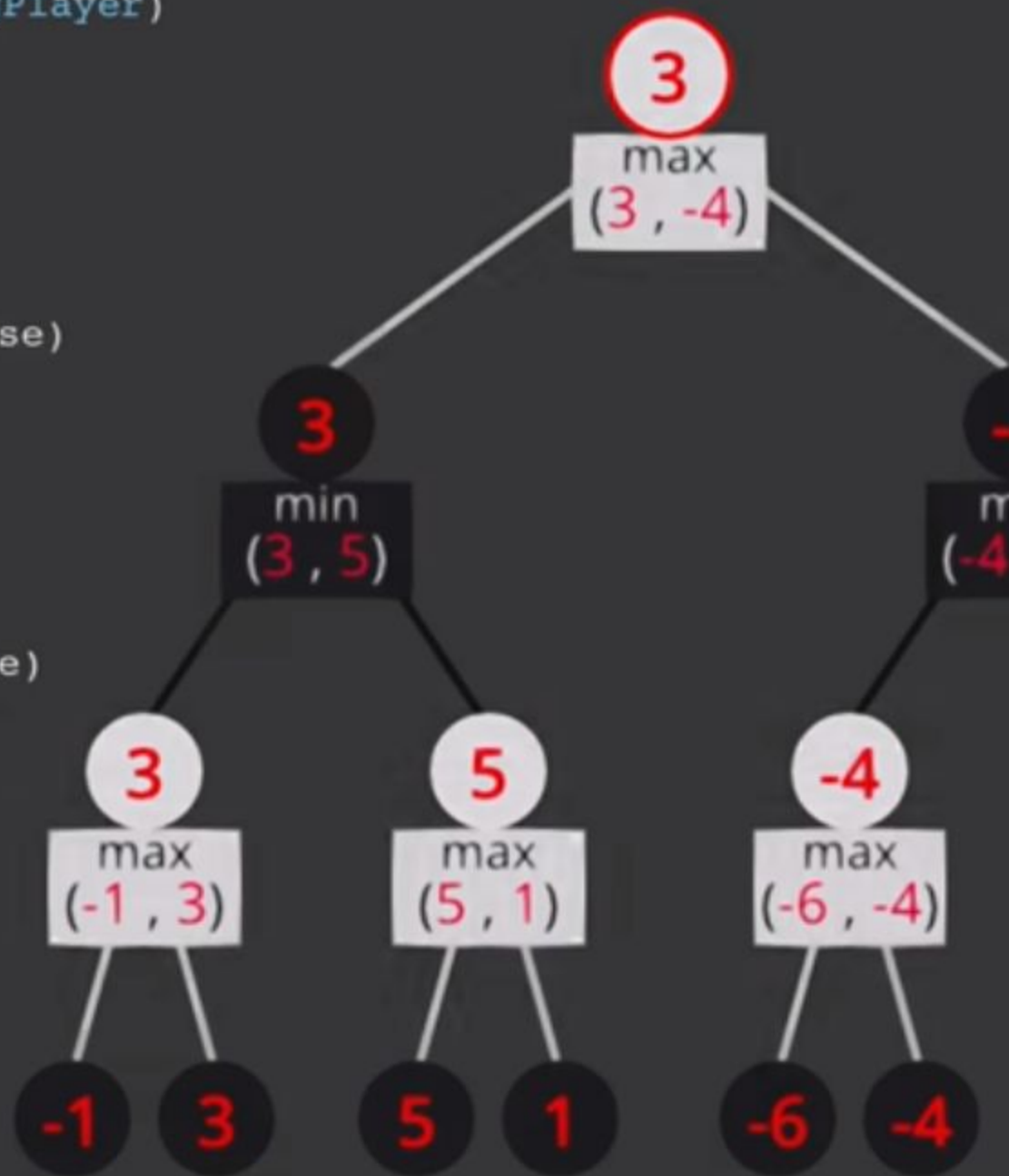
► Prunning does not affect final outcome

► Sorting moves by result improves $\alpha - \beta$ performance

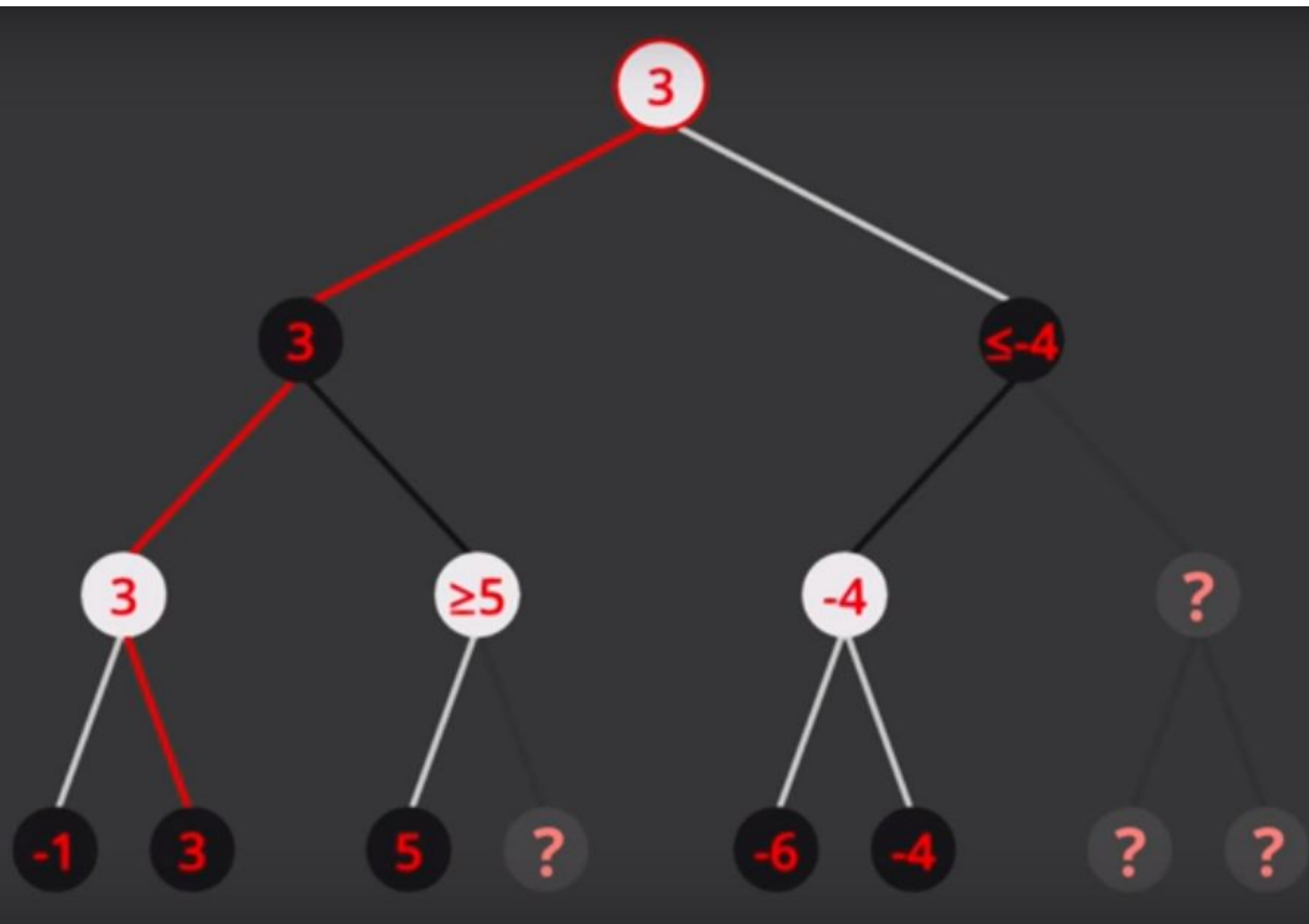► Perfect ordering: $O(b^{\frac{m}{2}})$

► An exercise on **metareasoning**

```
max(position, depth, maximizingPlayer)
= 0 or game over in position
static evaluation of position


ingPlayer
= -infinity
 child of position
= minimax(child, depth - 1, false)
al = max(maxEval, eval)
axEval


= +infinity
 child of position
= minimax(child, depth - 1, true)
al = min(minEval, eval)
inEval


ll
ntPosition, 3, true)
```
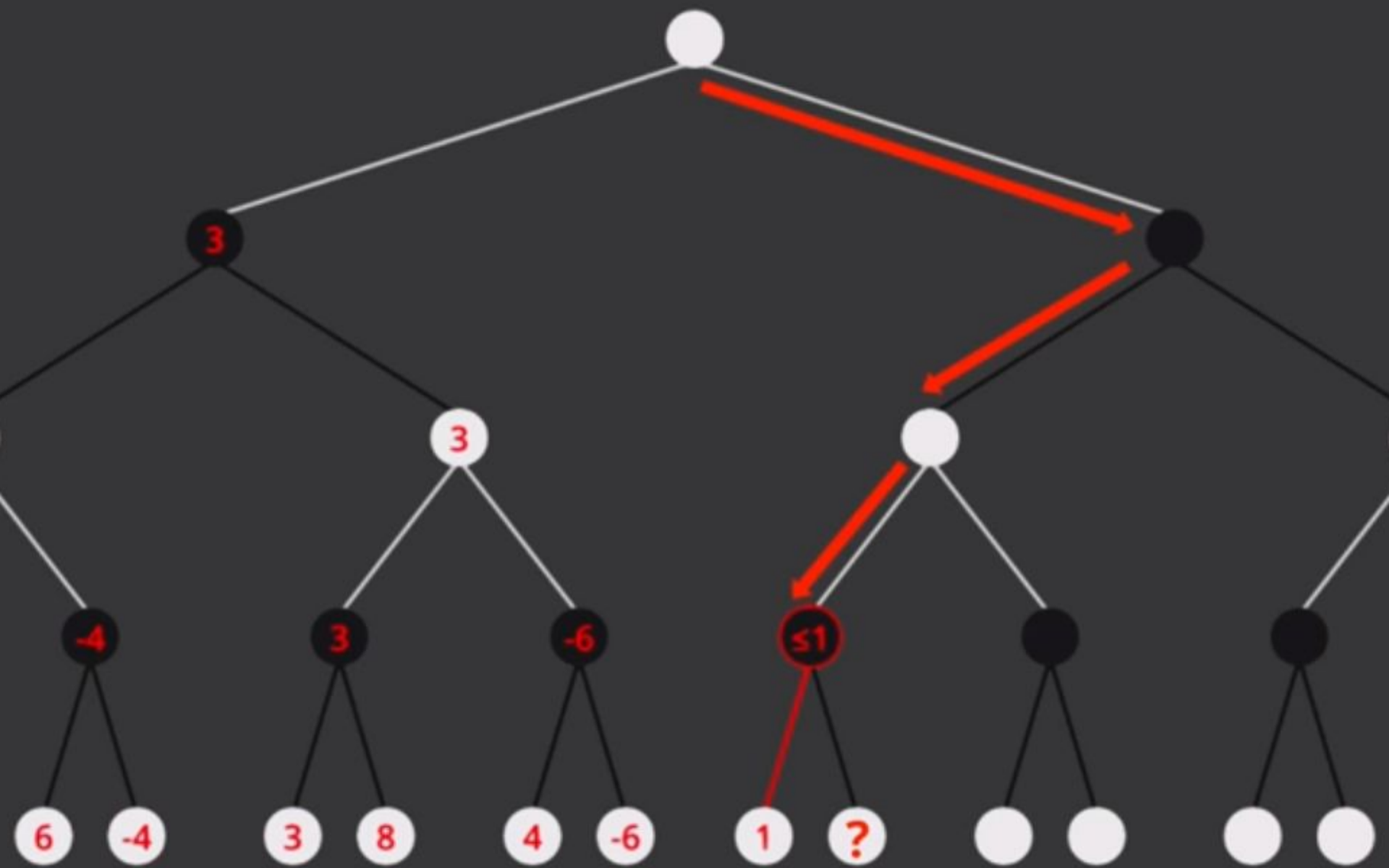

3 max (3 , -4)

3 min (3 , 5)

m (-4

3 max (-1 , 3)

5 max (5 , 1)

-4 max (-6 , -4)

-1 3 5 1 -6 -4
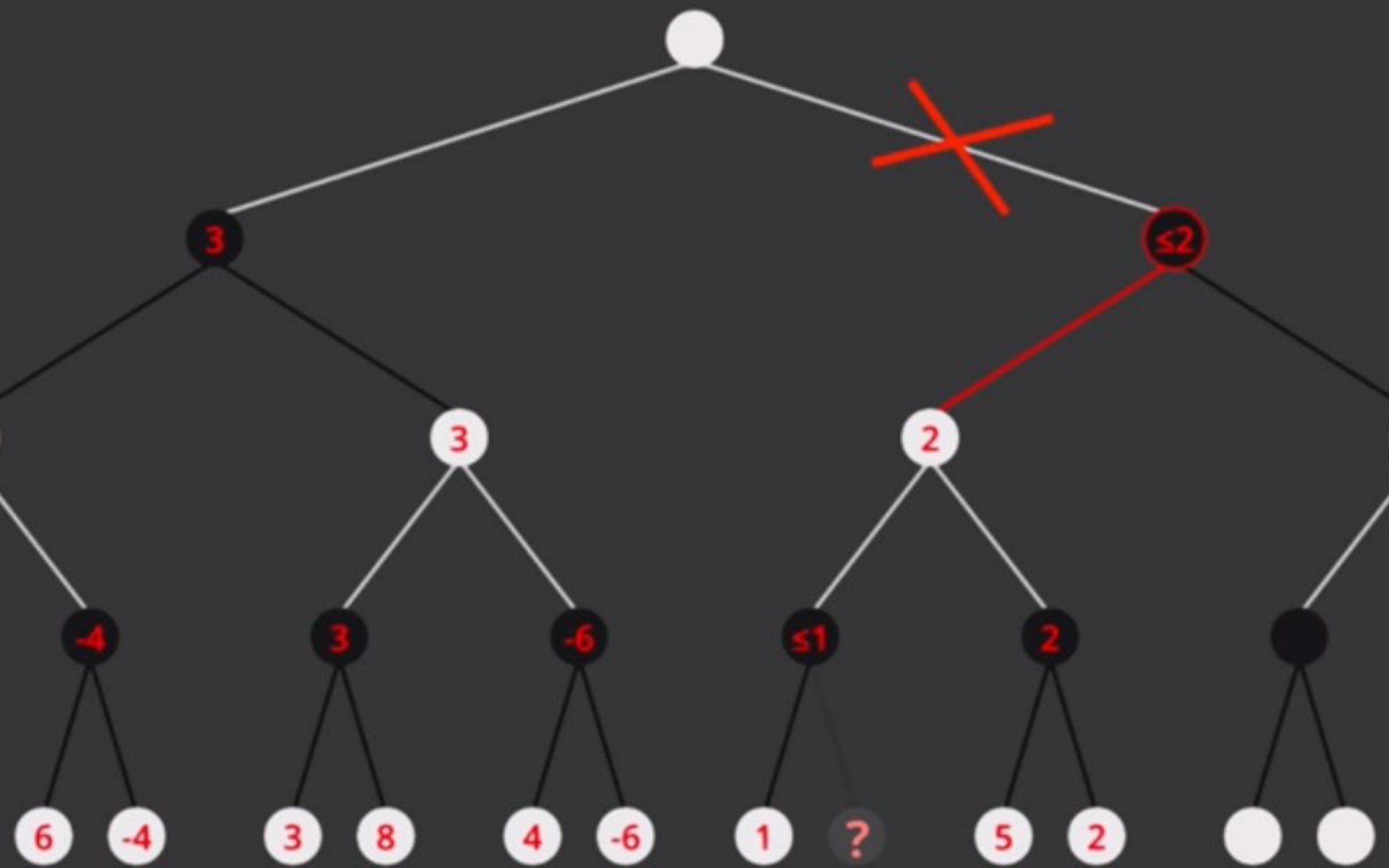
```
(position, depth, alpha, beta, maximizingPlayer)
 or game over in position
ic evaluation of position

Player
-infinity
hild of position
inimax(child, depth - 1, alpha, beta, false)
 = max(maxEval, eval)
max(alpha, eval)
 <= alpha

Eval

+infinity
hild of position
inimax(child, depth - 1, alpha, beta, true)
 = min(minEval, eval)
in(beta, eval)
 <= alpha

Eval

ll
ntPosition, 3, -∞, +∞, true)
```
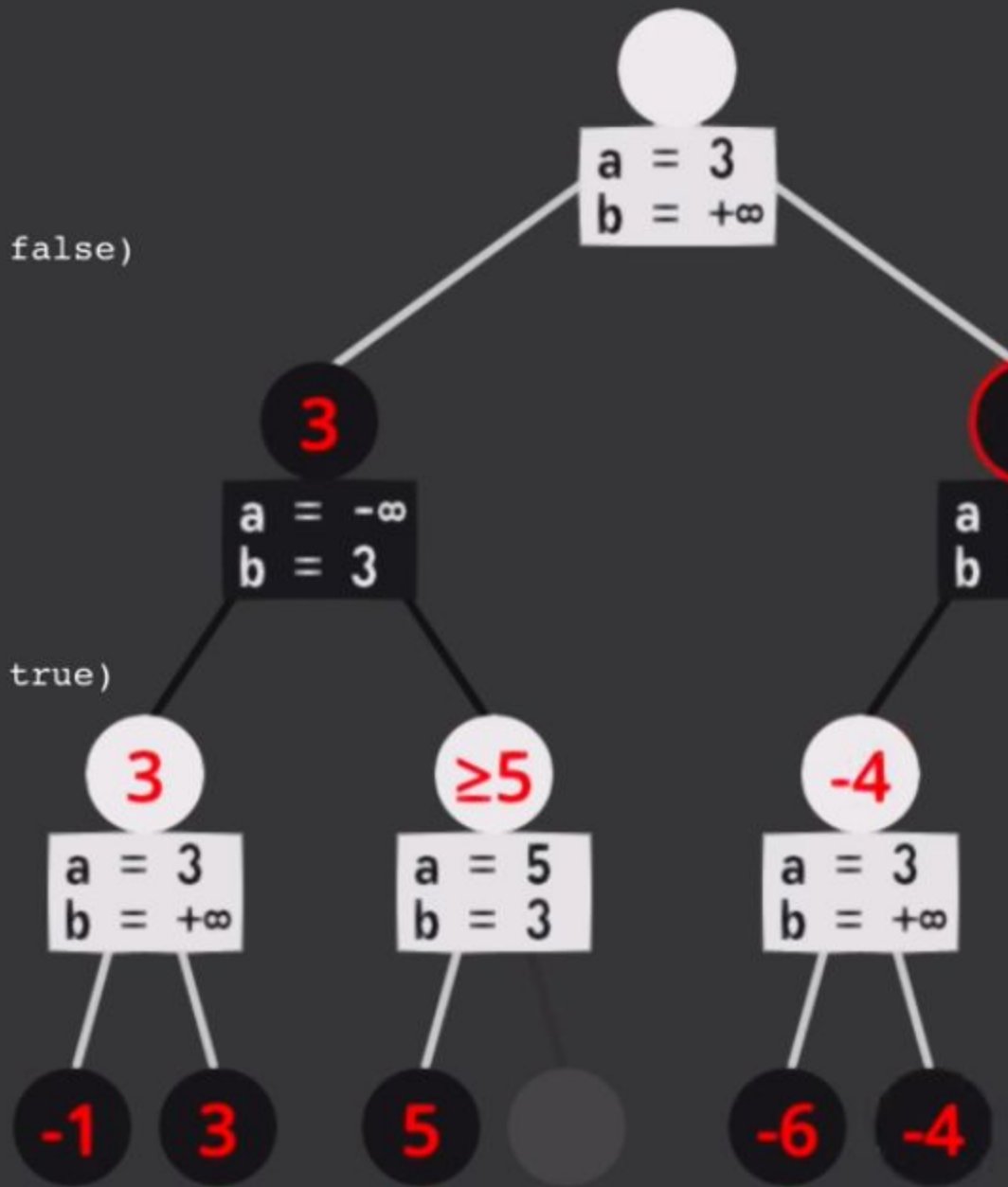
Effectiveness of alpha-beta

- Alpha-beta is guaranteed to compute the same value for the root node as computed by minimax, with less or equal computation
- **Worst case:** no pruning, examining b^d leaf nodes, where each node has b children and a d-ply search is performed
- **Best case:** examine only $(2b)^{d/2}$ leaf nodes.
  - Result is you can search twice as deep as minimax!
- **Best case** is when each player's best move is the first alternative generated
- In Deep Blue, they found empirically that alpha-beta pruning meant that the average branching factor at each node was about 6 instead of about 35!

Games of chance

- Backgammon is a two-player game with **uncertainty**.

- Players roll dice to determine what moves to make.

- White has just rolled *5 and 6* and has four legal moves:
  - 5-10, 5-11
  - 5-11, 19-24
  - 5-10, 10-16
  - 5-11, 11-16

- Such games are good for exploring decision making in adversarial problems involving skill and luck.