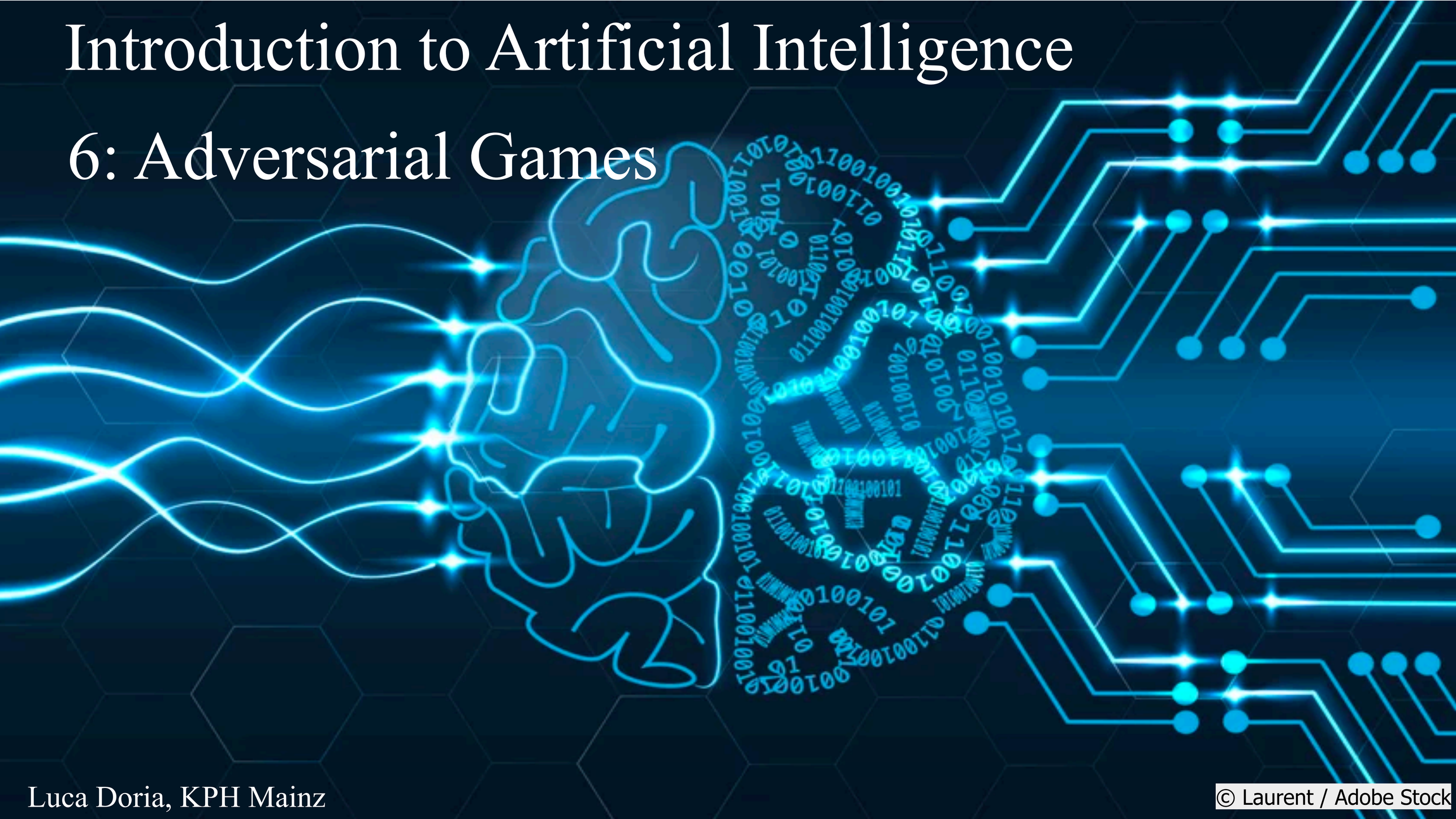
Introduction to Artificial Intelligence

6: Adversarial Games



Luca Doria, KPH Mainz

Introduction

- Playing board (adversarial) games is one of the oldest research areas in AI (since 1950s).
- Board games present an abstract form of competition between two agents requiring a form of “intelligence”.
- The states of a game are easy to represent (map to a data structure).
- The possible actions of the players are well-defined (rules).
- The implementation of a game maps to a search problem.
- The individual states are fully accessible (every agent knows them).
- It is a contingency problem, because the actions of one agent are not decided by the others.

Introduction

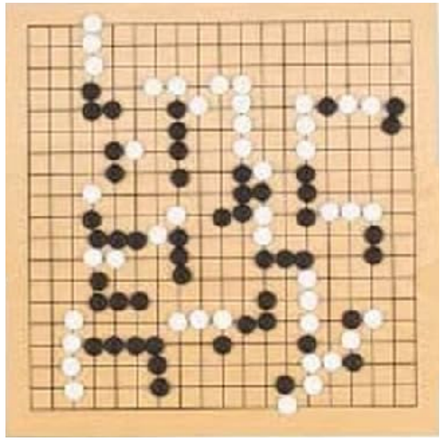
Board games can be very complex: they are contingency games + a very large state space.

Examples:

chess.com

- Chess: On average 35 possible actions possible from a position.
  Games have O(50) moves per player, resulting in a search depth of O(100):
  → $35^{100} \approx 10^{150}$ nodes in the search tree (with ~10^{40} legal chess positions).

- Go: On average O(200) possible actions with
  O(300) moves → $200^{300} \approx 10^{700}$ nodes.



amazon.com

- Efficient programs delete irrelevant branches of the tree (pruning), use good evaluation functions for in-between states, and look ahead as many moves as possible.

Definitions

Players are **max** and **min**, where max begins.

Initial position (e.g., board arrangement)

Operators (= legal moves)

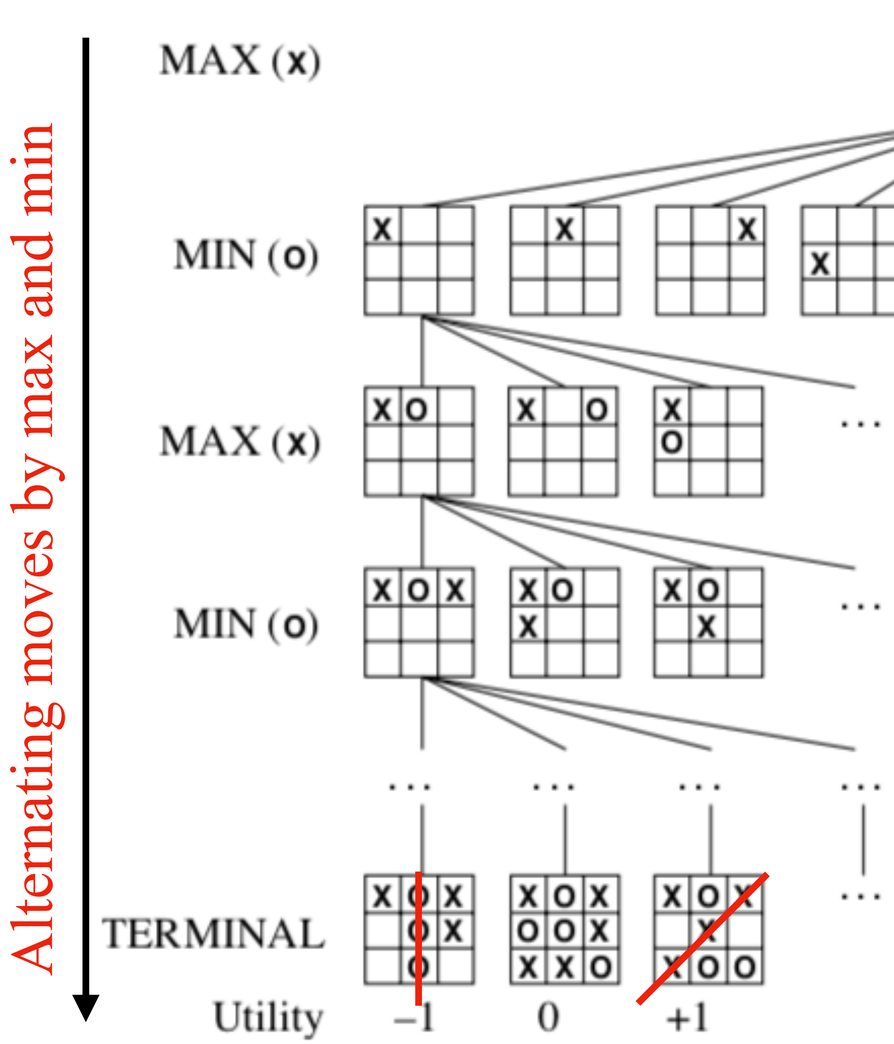
Game tree is the search tree generated from the possible (alternate) moves.

Termination test, determines when the game is over and what the value of the final state is.

Strategy. In contrast to regular searches, where a path from beginning to end is a solution, max must come up with a strategy to reach a favorable terminal state regardless of what min does.

Thus, all of min's moves must be considered and reactions to them must be computed.

Example: Tic-Tac-Toe (and imagine the Chess tree…)



Initial state

9 possible first moves

8 possible second moves —> 9*8 nodes

Possible terminal states

9! = 362880 different games

As many know, Max has one elementary strategy for reducing the tree by a factor 9…

And Min by another factor 2…

The Minimax algorithm

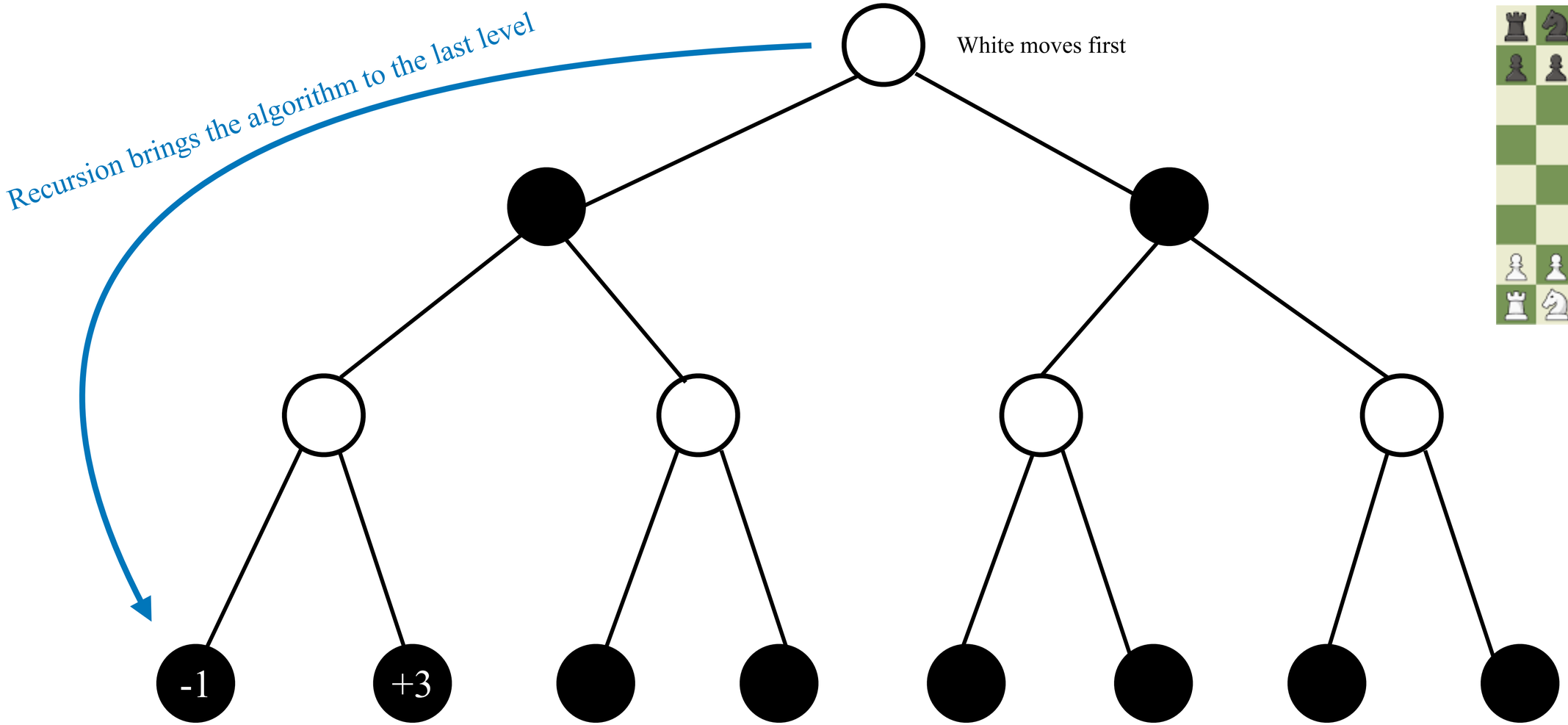
1. Generate the complete game tree using depth-first search (we do not need the full tree in memory)

2. Apply the utility function to each terminal state.

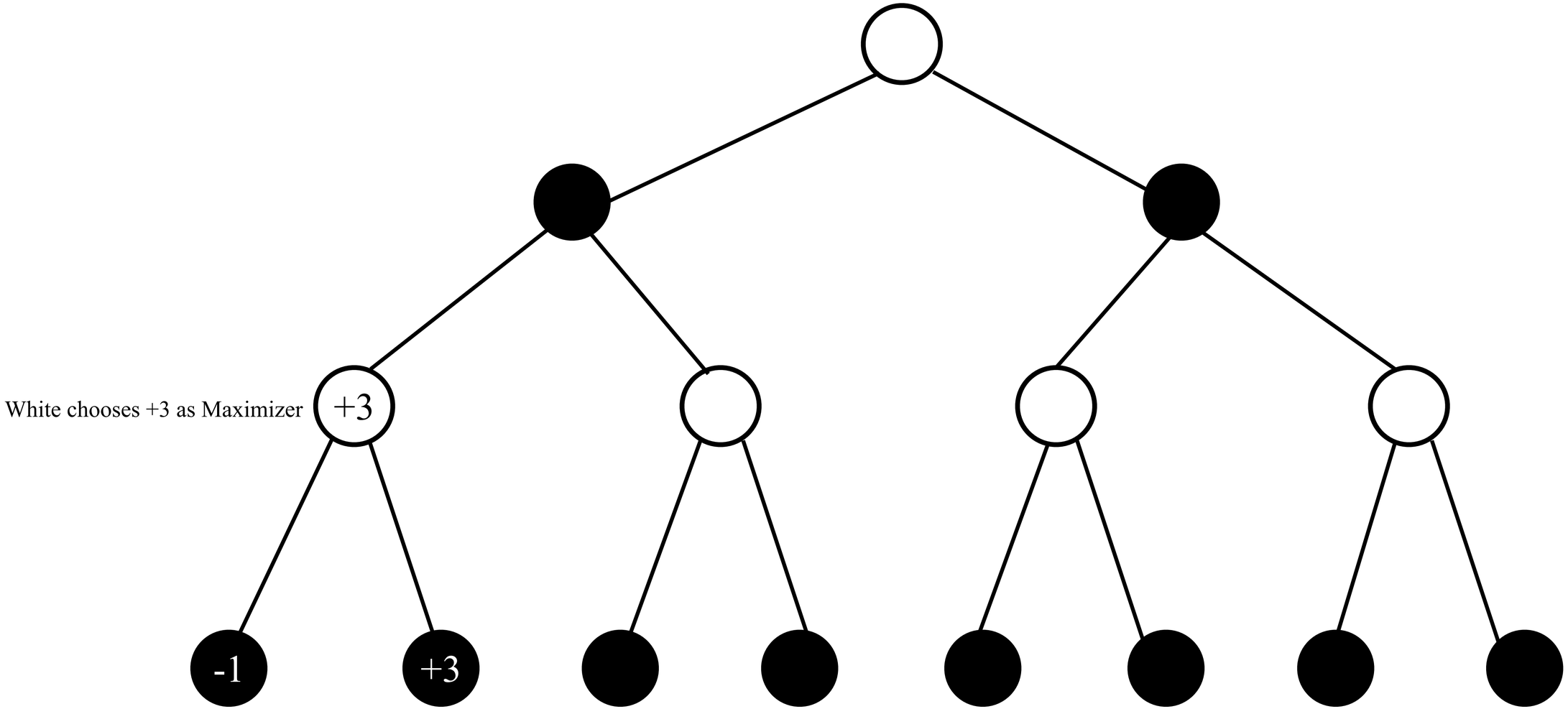
3. Beginning with the terminal states, determine the utility of the predecessor nodes with:
   - Node is a min-node: Value is the minimum of the child nodes
   - Node is a max-node: Value is the maximum of the child nodes
   - From the initial state (root of the game tree), Max chooses the move that leads to the highest value (minimax decision).

Minimax assumes that min does not make any mistakes. Even if he did, Max would only end up even better.

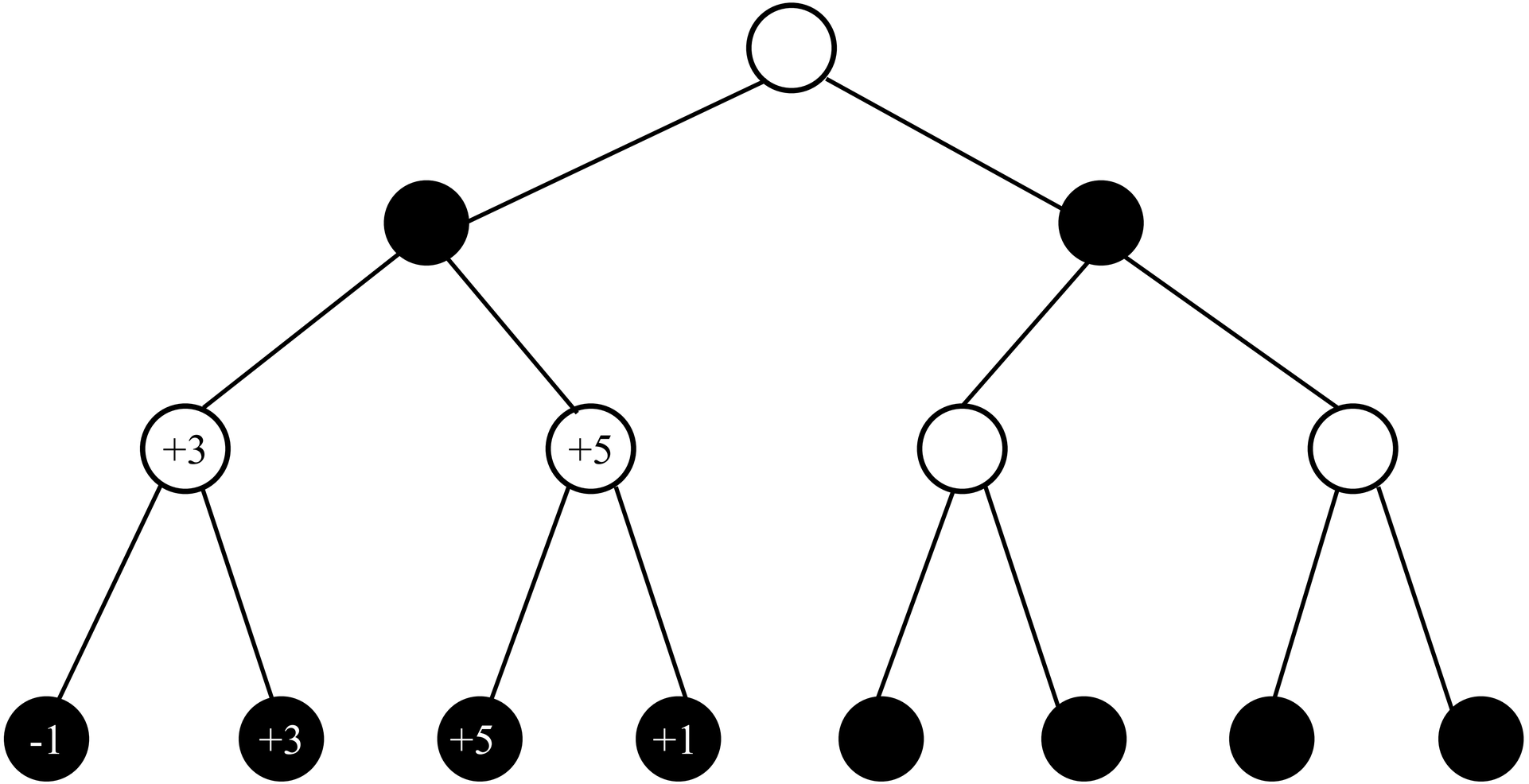
Minimax example



Minimax example

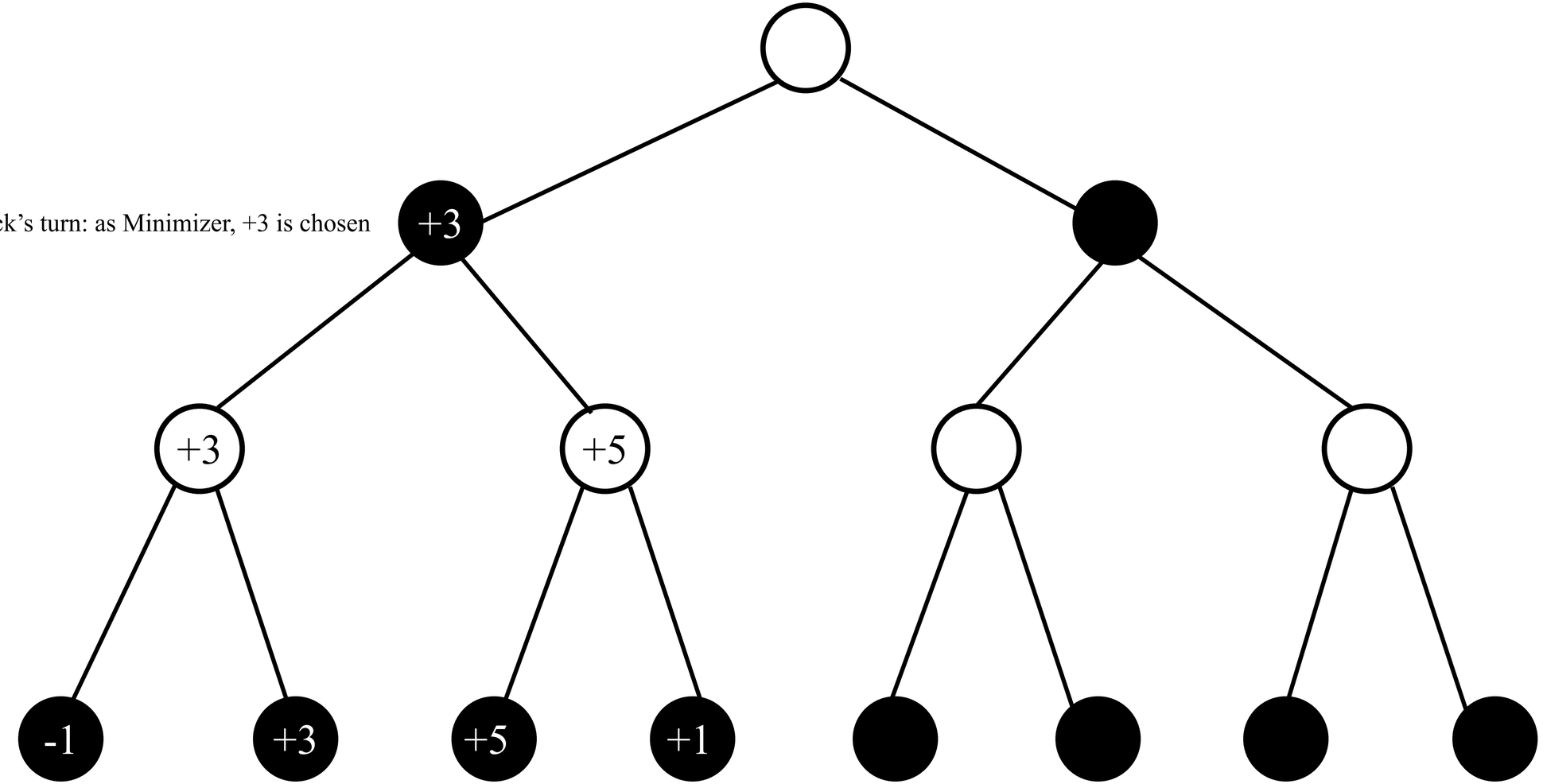


Minimax example

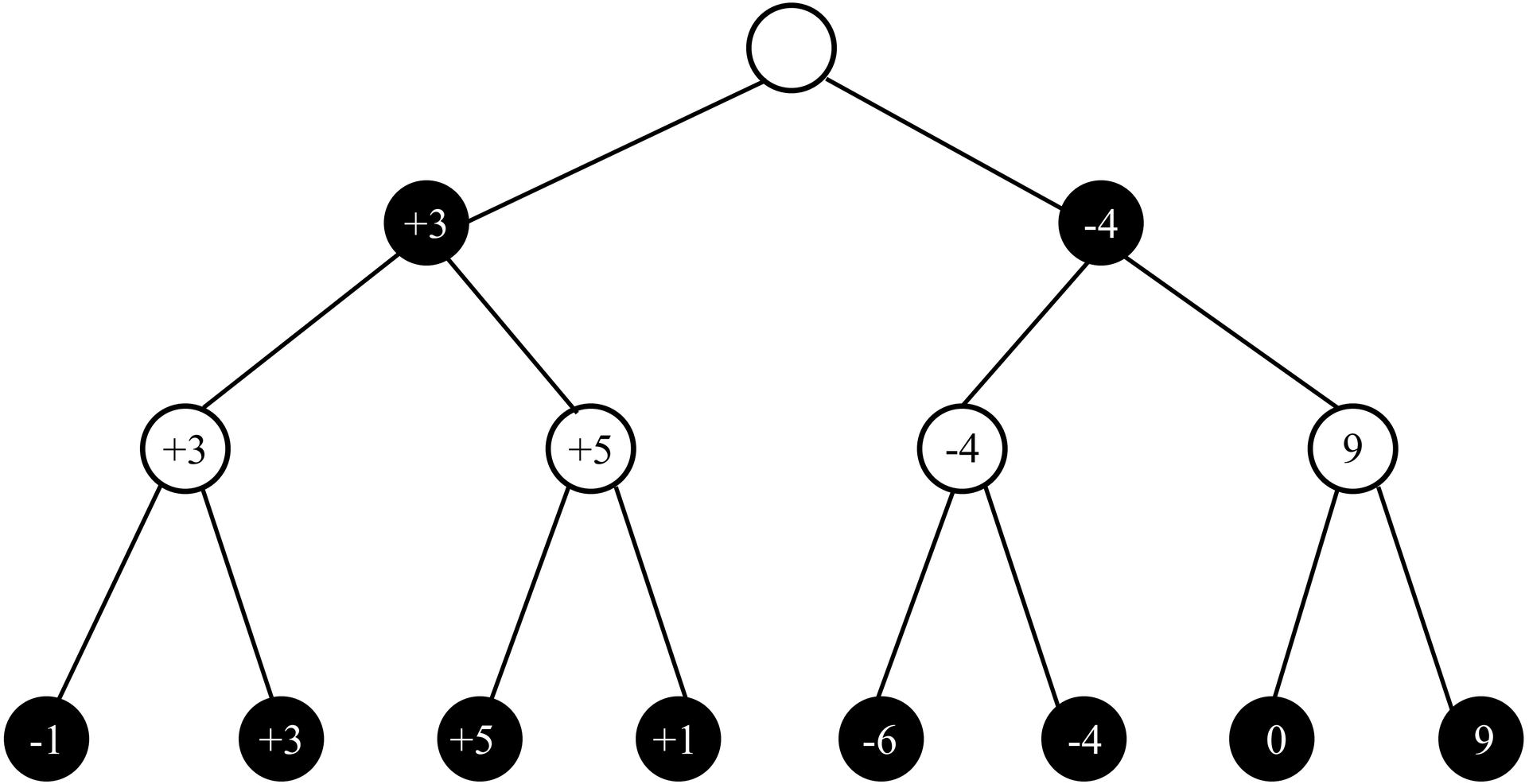


Minimax example

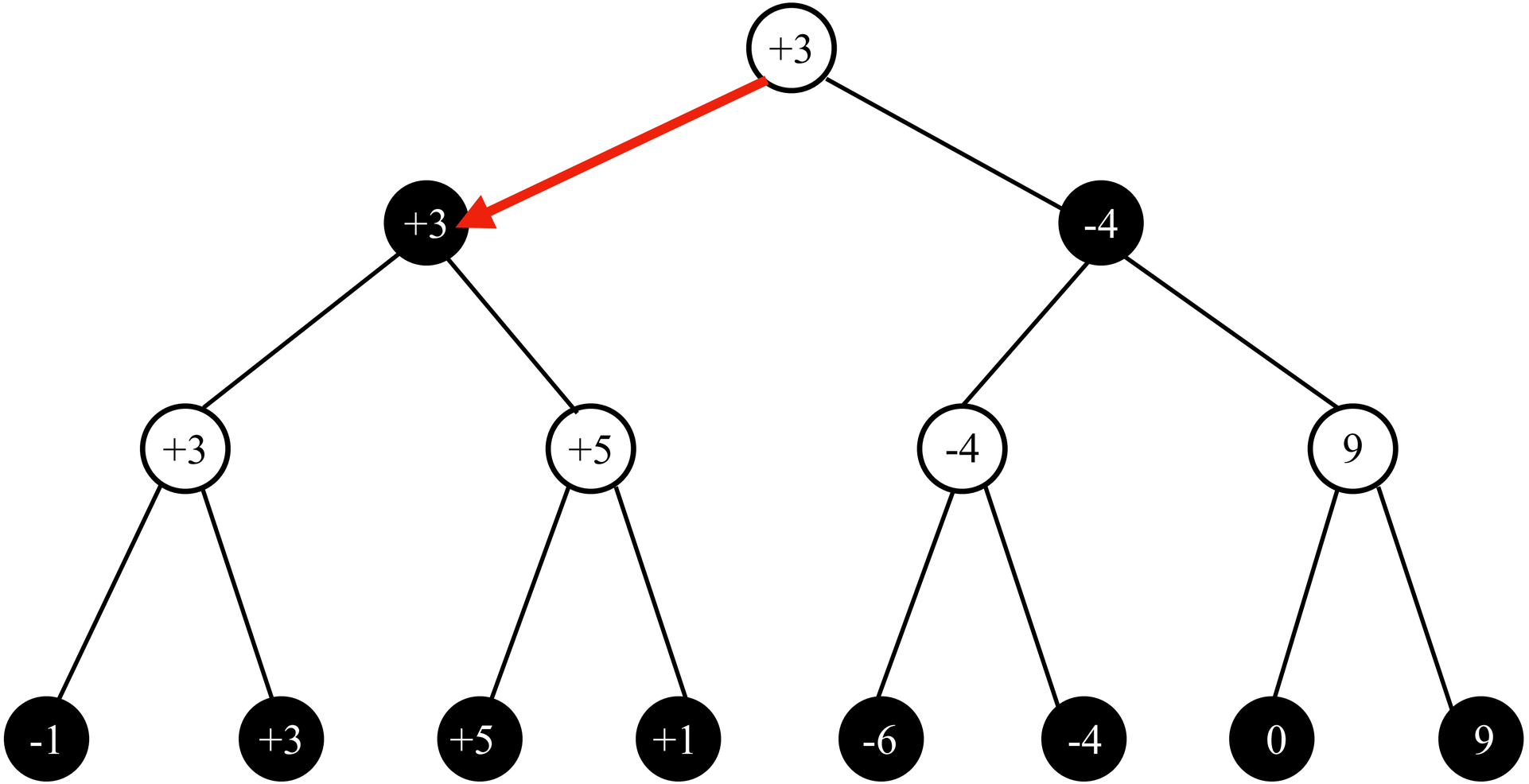
It’s black’s turn: as Minimizer, +3 is chosen



Minimax example



Minimax example



Minimax Algorithm

Recursive algorithm.

Max must maximise starting from -∞

Min must minimise starting from +∞

Function Evaluation must be specified

```
function MiniMax(position, depth, MaxPlayer)
    if depth = 0 or game_over in position
        return Evaluation(position)
    if MaxPlayer
        maxEval = - Infinity
        for child in position
            eval = MiniMax(child, depth-1, false)
            maxEval = max(maxEval, eval)
        return maxEval
    else
        minEval = + Infinity
        for child in position
            eval = MiniMax(child, depth-1, true)
            maxEval = min(minEval, eval)
        return minEval
```

Evaluation Function

When the search tree is very large, only a certain depth can be practically reached.

Correctly evaluating the playing position at a given depth is fundamental.

Example of a simple evaluation function for chess:

Assign the scores: pawn 1, knight/bishop 3, rook 5, queen 9

Additional heuristic criteria: king safety, good pawn structure, bishops on long diagonals,…

Rule of thumb: three-point advantage = certain victory

Consistency of the evaluation function: the chance of winning with a one-point advantage should be less than with a two-point advantage.

Evaluation Function

In general, preferred evaluation functions are weighted, linear functions:

$$f = \omega_1 f_1 + \omega_2 f_2 + \ldots + \omega_n f_n$$

where the ω_i are weights, and the f_i are the features (e.g. $\omega = 10$, f="Queen present").

The above linear sum makes the (rather strong) assumption that the contributions of all features are independent. (not true in general: e.g., bishops in the endgame are more powerful, when there is more space. Or bishops are stronger when both present).

The weights can be learned via some optimising procedure. The features, however, are often designed by human intuition and understanding (heuristic criteria).

Maximum Tree Depth and other limiting methods

Motivation: return an answer within a given time limit. Fixed-depth search.
Even better: **iterative deepening search** (stop, when time is over).

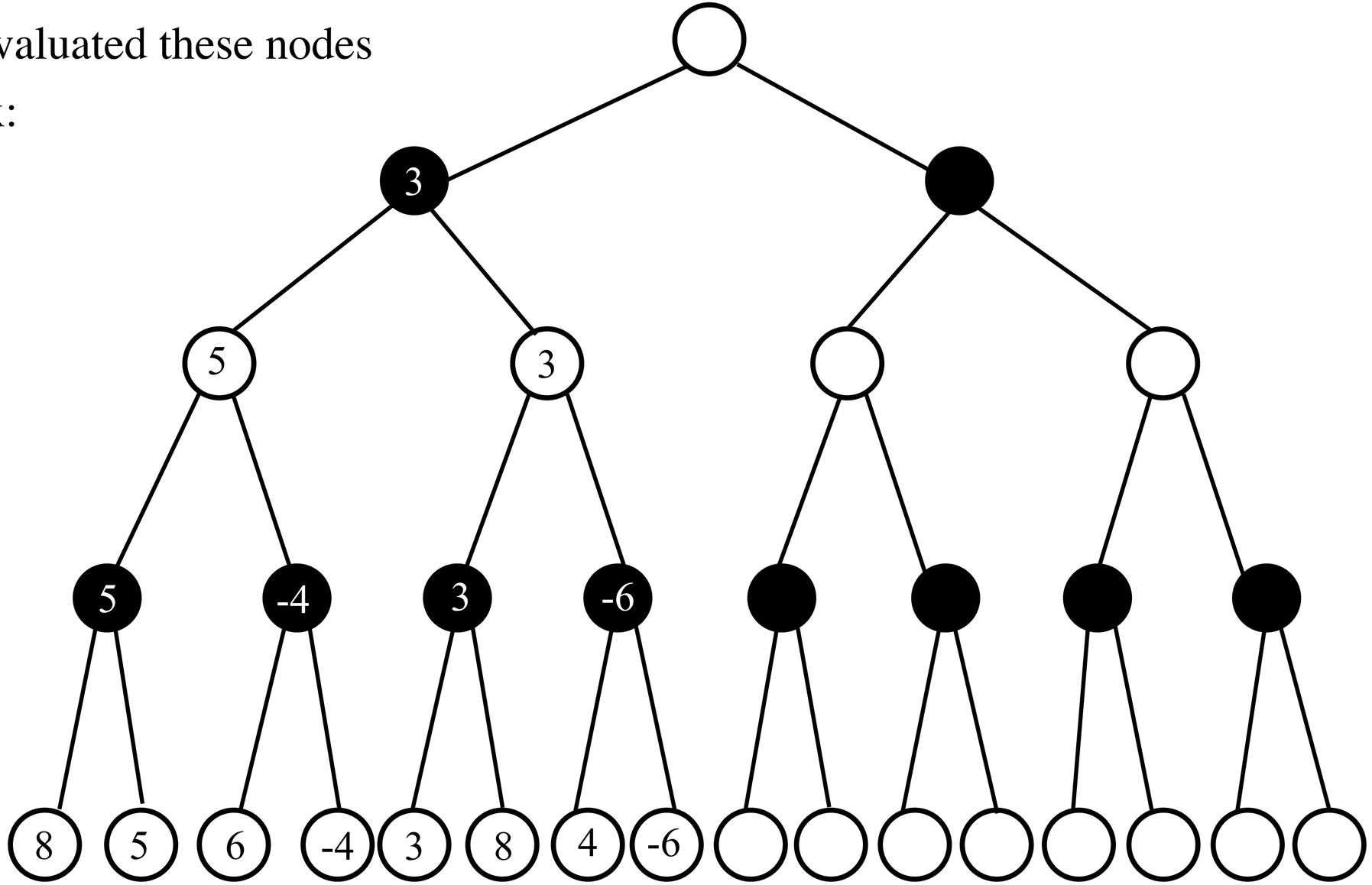
It is better to reach in "**quiescent**" positions that do not cause large fluctuations in the evaluation function in the following moves. Chess example: you can have material advantage, but if on the next (non evaluated) move the opponent captures the queen, then the evaluation changes dramatically.

Limited-depth search can lead also to the **horizon effect.**

This happens when a very dangerous move can be delayed by the opponent to the point that the move "falls beyond the horizon", i.e. it is beyond the maximum search depth.

Alpha-Beta Pruning: General Idea

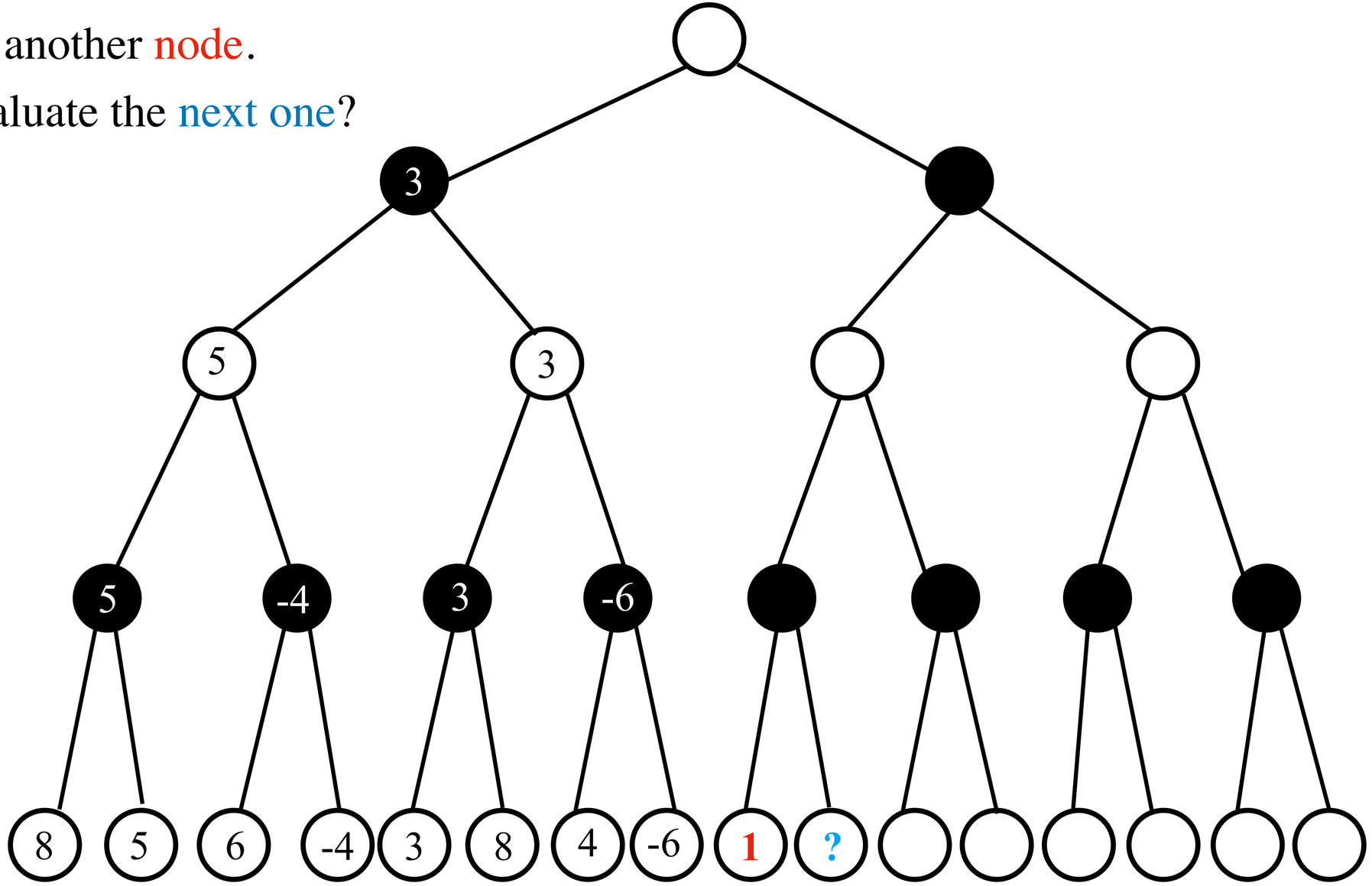
Suppose we evaluated these nodes with MiniMax:



Alpha-Beta Pruning: General Idea

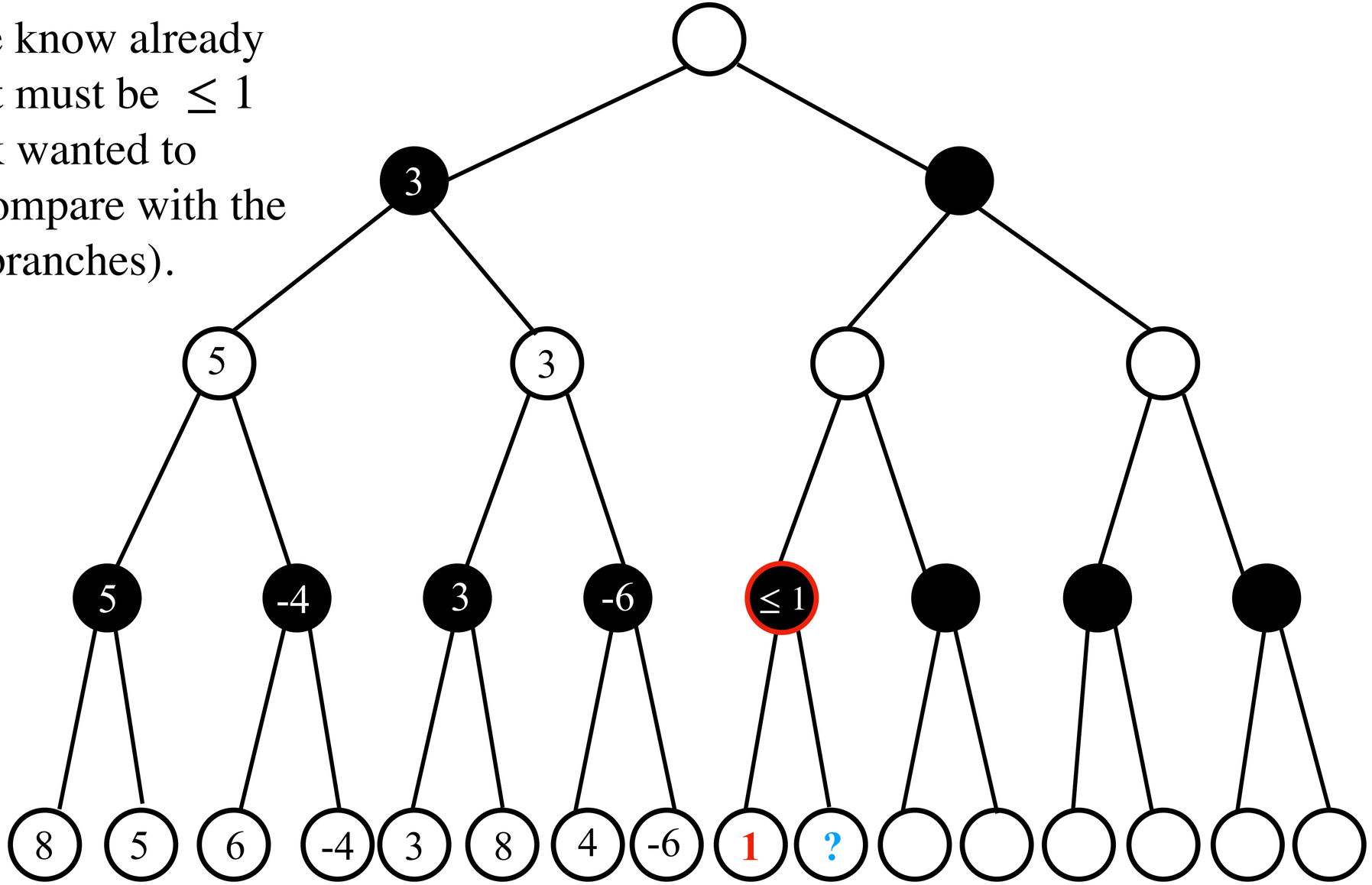
Now evaluate another node.

Should we evaluate the next one?



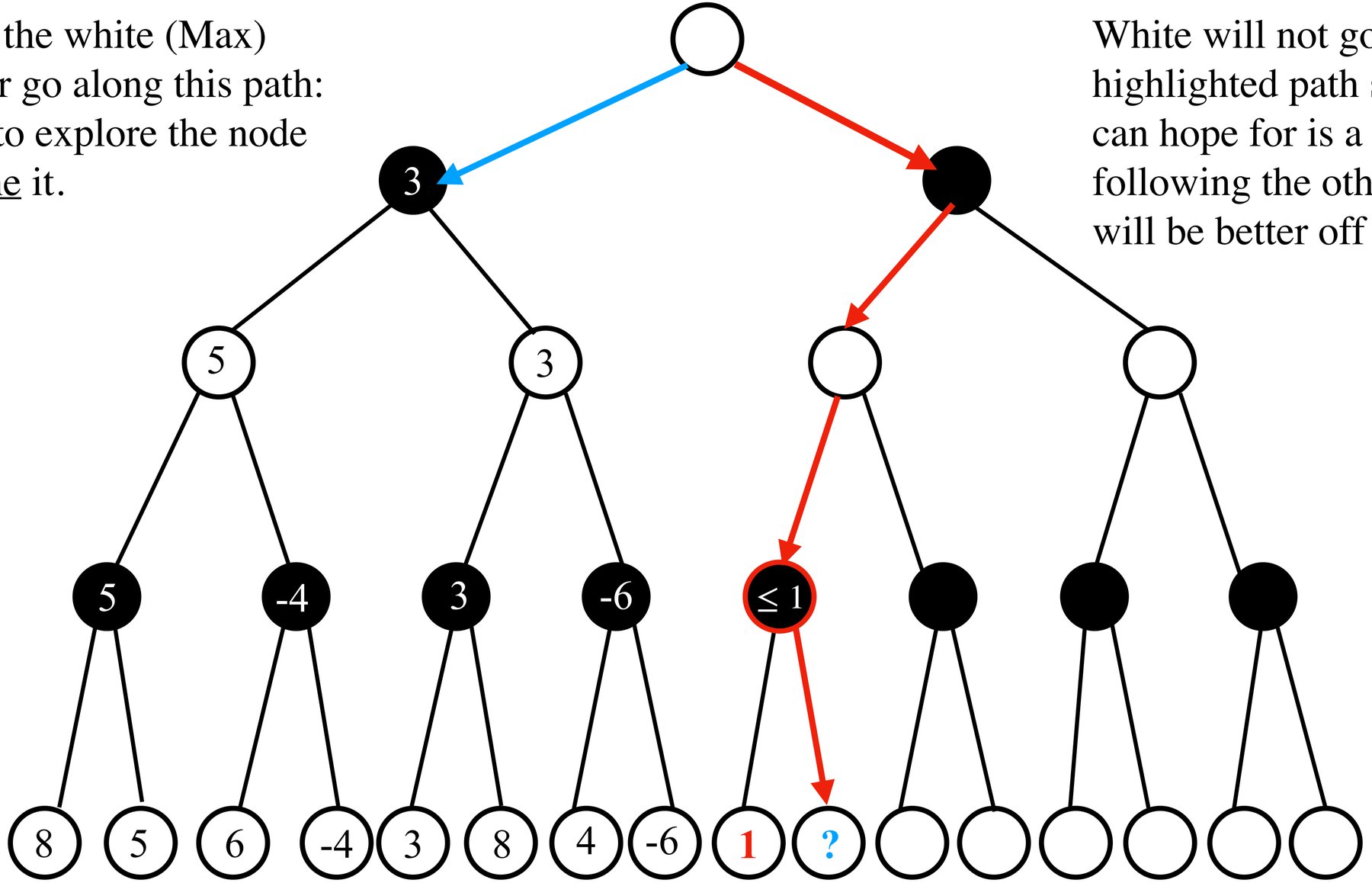
Alpha-Beta Pruning: General Idea

Not really: we know already that the parent must be ≤ 1 because Black wanted to minimize! (Compare with the other known branches).



Alpha-Beta Pruning: General Idea

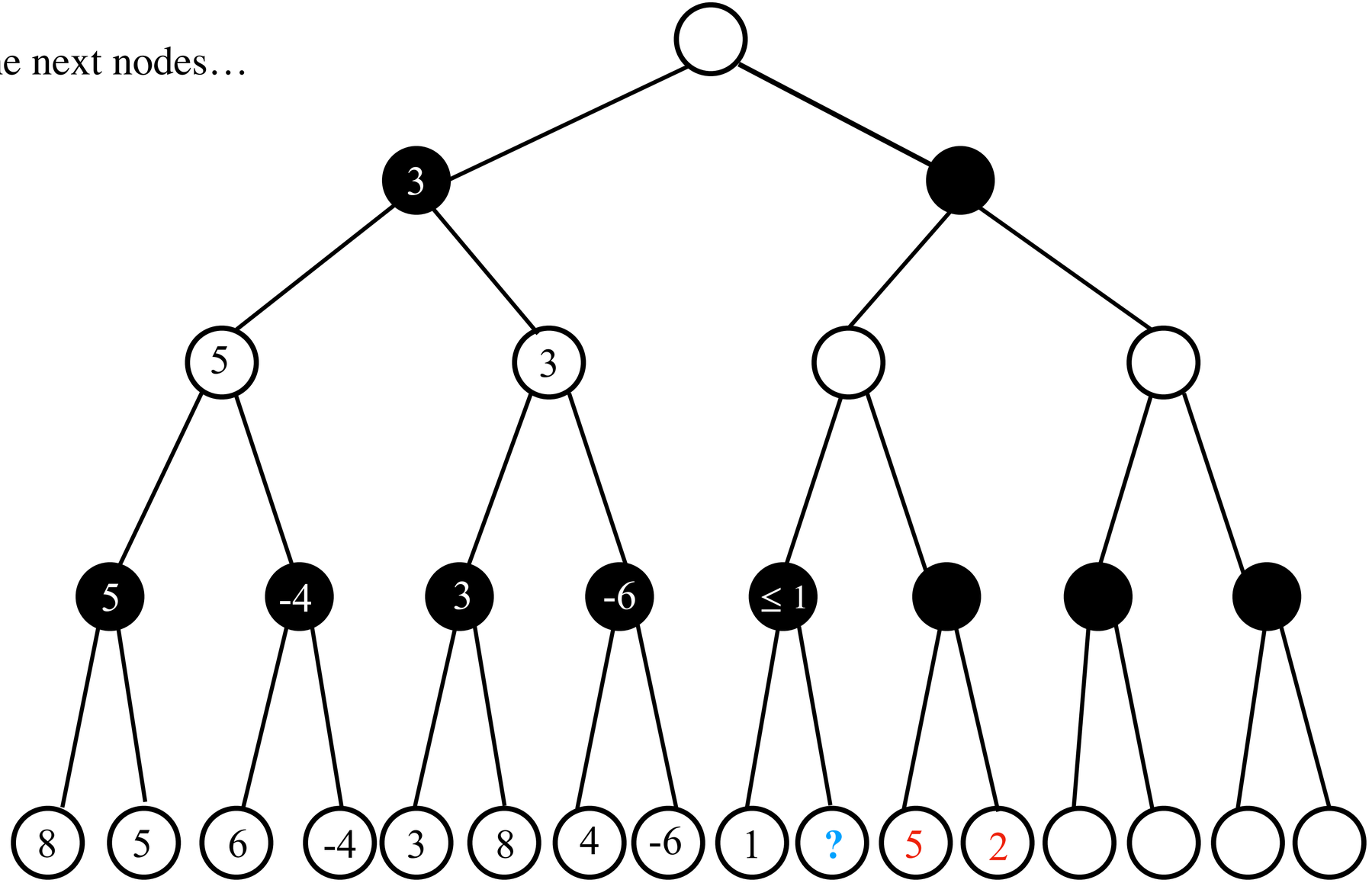
This means that the white (Max) player will never go along this path: we do not need to explore the node "?" and we prune it.



White will not go down the (red) highlighted path since the best he can hope for is a "+1", while following the other (blue) path it will be better off (getting a 3>1).

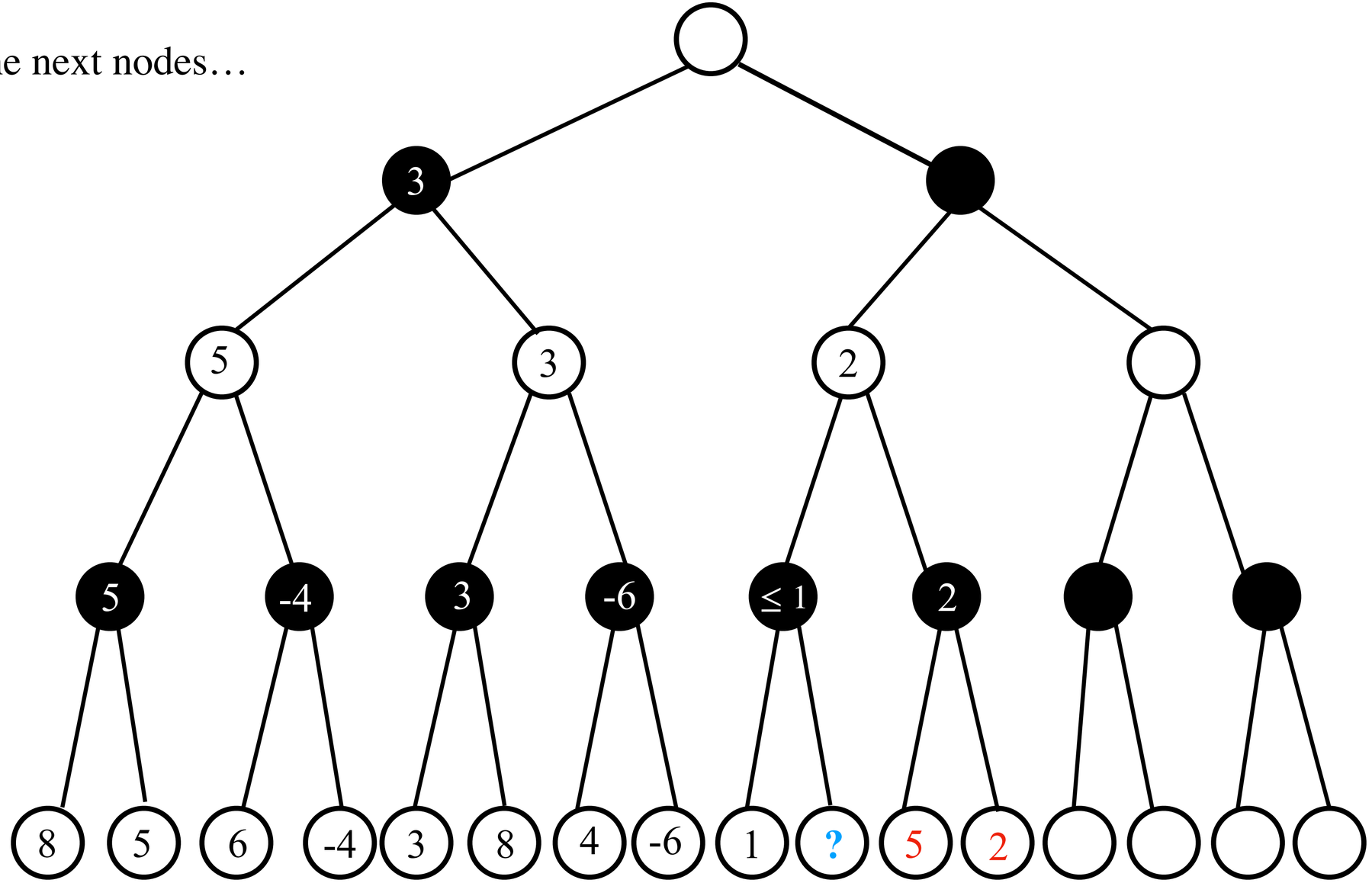
Alpha-Beta Pruning: General Idea

Continuing to the next nodes…



Alpha-Beta Pruning: General Idea

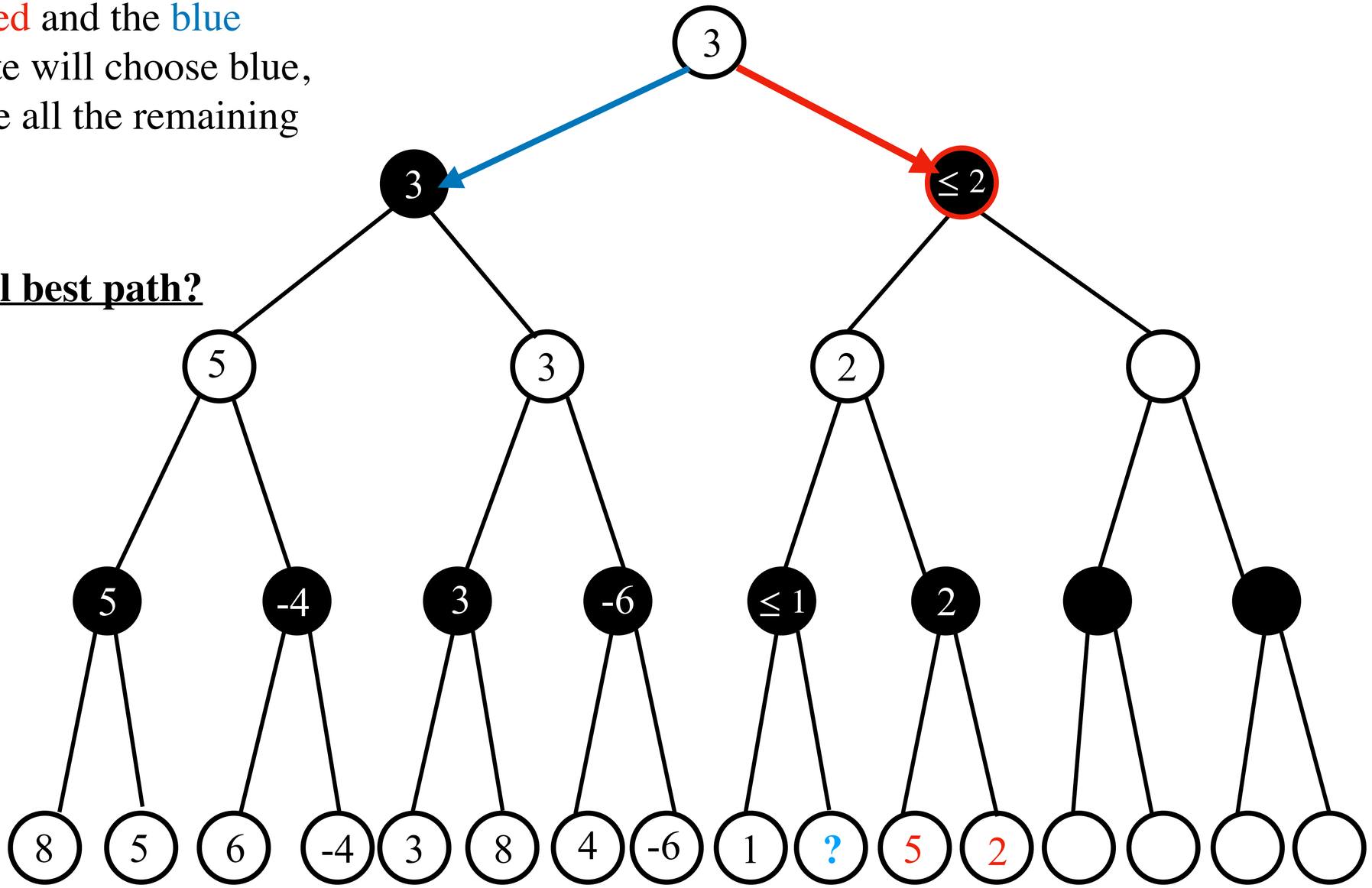
Continuing to the next nodes…



Alpha-Beta Pruning: General Idea

Comparing the red and the blue paths, again white will choose blue, and we can prune all the remaining nodes.

What is the final best path?



Alpha-Beta Pruning

Idea: keep track of the best score you can achieve (assuming a "perfect" play of the other player).

For this, we add two mode parameters, conventionally called **alpha** and **beta** to the MiniMax function:

function MiniMax(position, depth, MaxPlayer)

function MiniMax(position, depth, alpha, beta, MaxPlayer)

We will decide if to prune a tree branch based on the values of alpha and beta.

For Max: alpha will be the max(alpha,MiniMax) : alpha carries the best (=maximum) choice we found so far along a path. We can think at alpha as "at least this value".

For Min: beta will be the min(beta,MiniMax): beta carries the best (=minumun) choice we found so far along a path. We can think at beta as "at most this value".

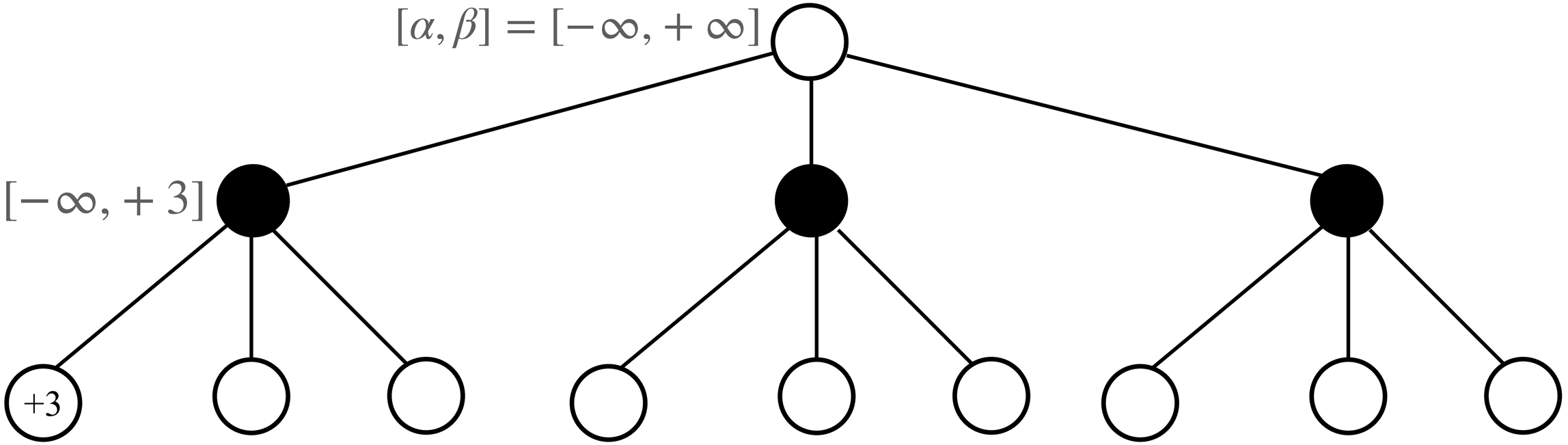
Alpha-Beta Pruning Algorithm

```
function MiniMax(position, depth, alpha, beta, MaxPlayer)
    if depth = 0 or game_over in position
        return Evaluation(position)
    if MaxPlayer
        maxEval = - Infinity
        for child in position
            eval = MiniMax(child, depth-1, alpha, beta, false)
            maxEval = max(maxEval, eval)
            alpha = max(alpha, eval)
            if beta <= alpha: break
        return maxEval
    else
        minEval = + Infinity
        for child in position
            eval = MiniMax(child, depth-1, alpha, beta, true)
            maxEval = min(minEval, eval)
            beta = min(beta, eval)
            if beta <= alpha: break
        return minEval
```

At the beginning, the function is called with:

$\alpha = -\infty \quad ; \quad \beta = +\infty$

Visualization of the Algorithm

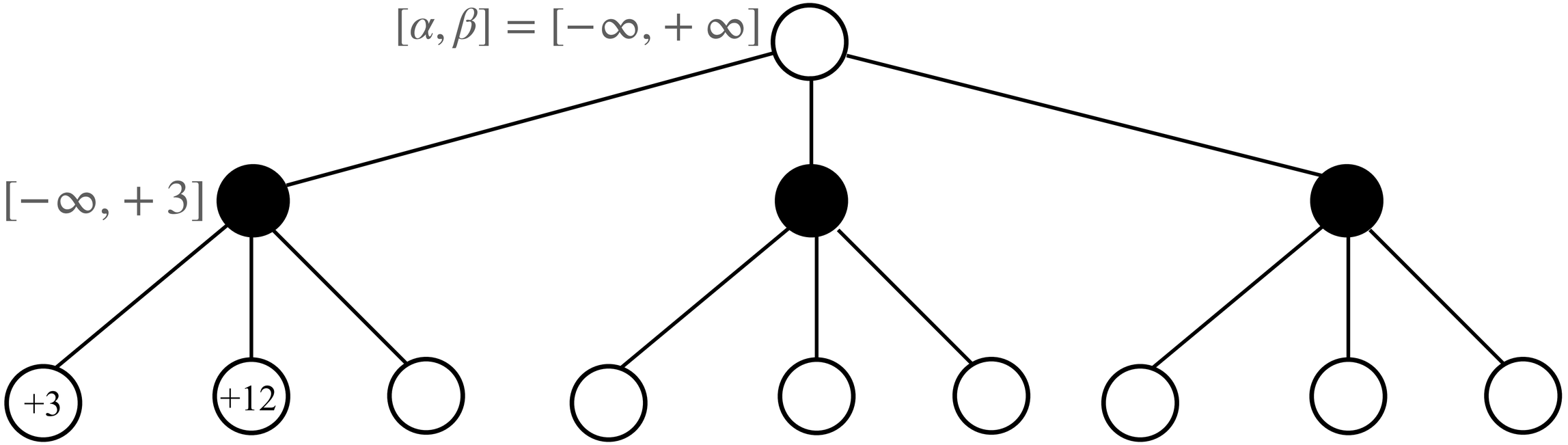


First depth-first pass: the final node is evaluated as +3.

This means that the parent black node is AT MOST 3 (or, ≤ 3).

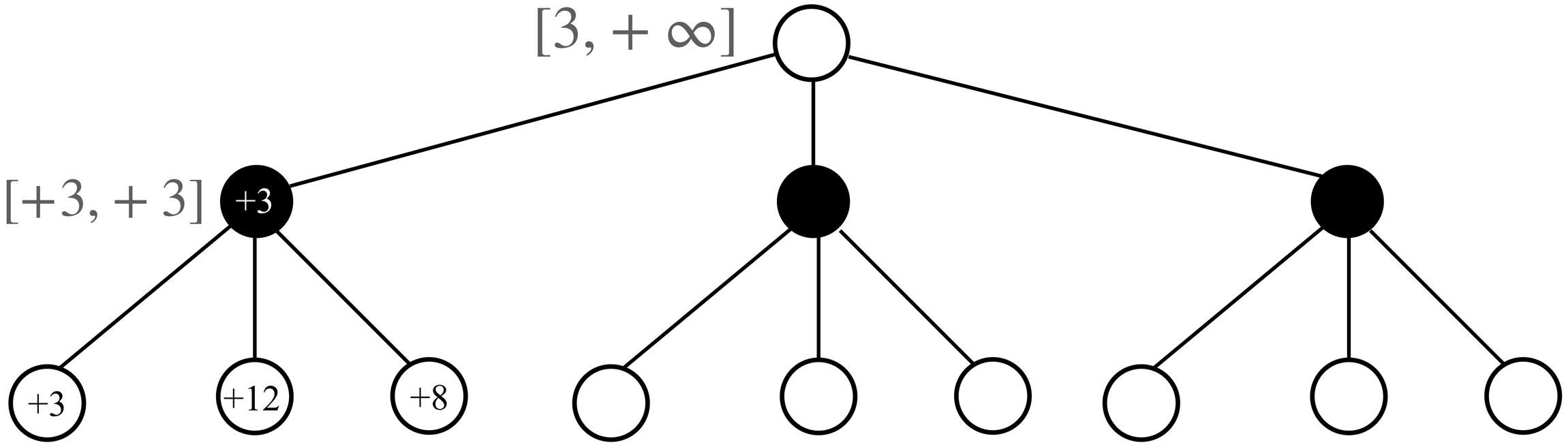
Beta is updated accordingly: min(+∞,+3) = +3

Visualization of the Algorithm



The second node is evaluated as +12: Min (Black) will avoid this move and beta stays +3.

Visualization of the Algorithm

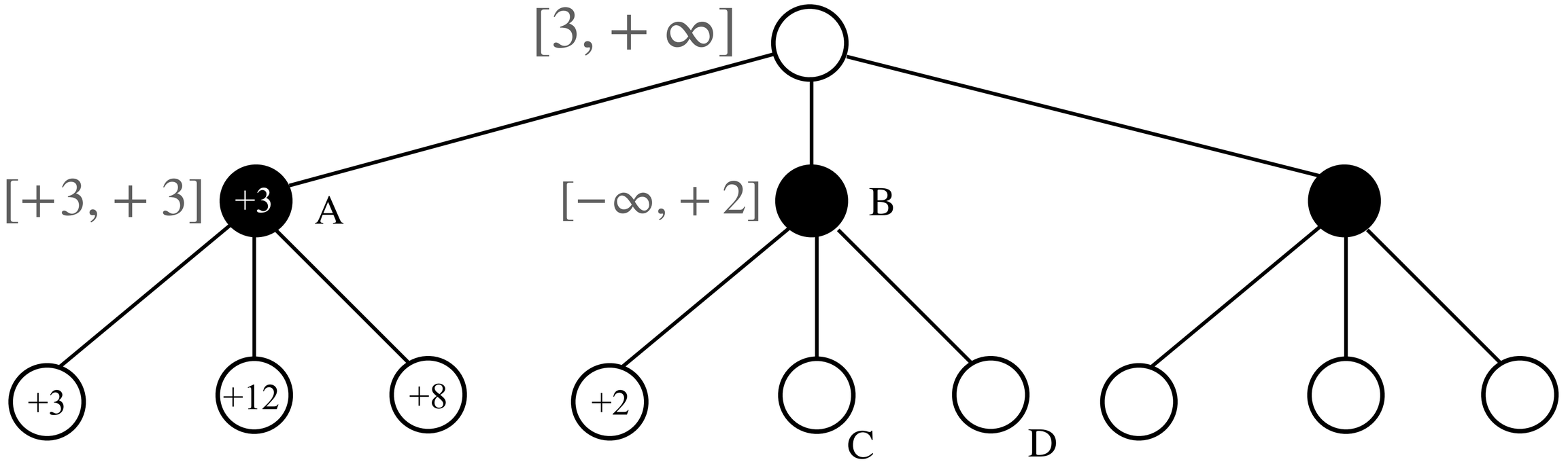


Now we evaluated all the successors of the black node and discover that it has the value 3.

This means that the value of the ROOT node is AT LEAST 3 (≥ 3) and we update alpha accordingly:

$\max(-\infty, +3) = +3$.

Visualization of the Algorithm

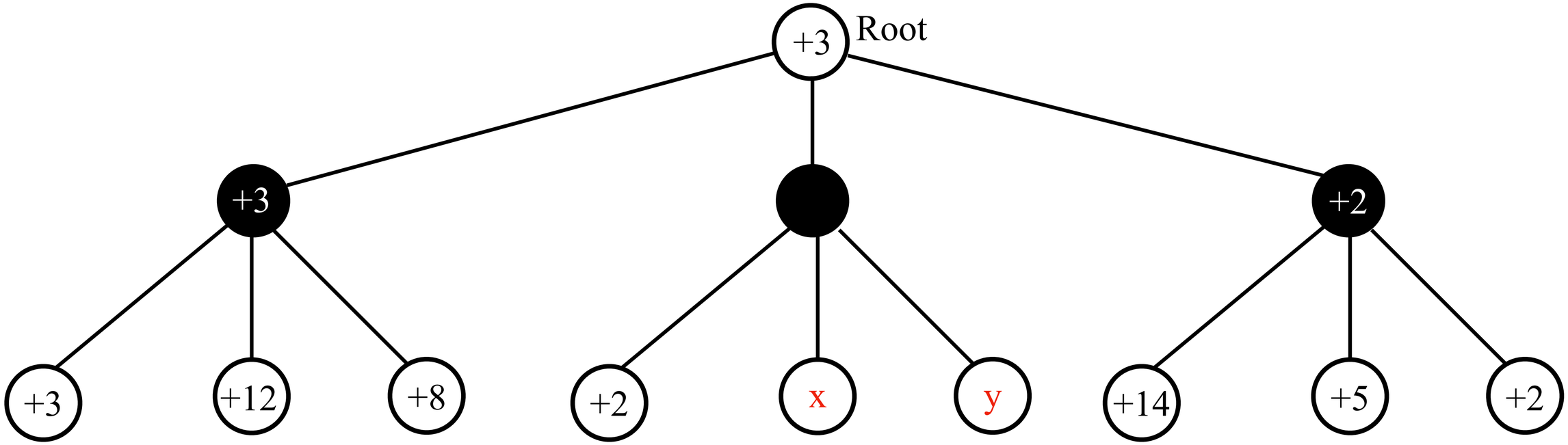


A new terminal node is evaluated with the result +2.

As in the previous branch, beta is updated the the "at most" value beta = min(+∞,2).

BUT: we know that node A has a value +3 and node B has ≤ 2. This means that the Max (white player) will never go down the path through B. This means that we can prune nodes C and D and avoid their evaluation (the **`break`** command in the code) …. …. …

Another view of the recursion



MiniMax(Root) = max(min(3,12,8) , min(2,x,y) , min(14,5,2)) =
= max(3, min(2,x,y), 2) = 3

min(2,x,y) ≤ 2 → The final result is independent from x,y

Pruning Efficiency increase

- The alpha-beta pruning eliminates the largest amount of nodes/branches when we examining the best move first.

- In the best case (always the best move first), the time complexity is reduced to $O(b^{d/2})$. This means that we can search twice as deep in the same amount of time.

- In the average case (randomly distributed moves), for *moderate* b ($b < 100$), we roughly have $O(b^{3d/4})$.

- Unfortunately, the best move is not known in general. An ordering heuristic can bring the performance close to the optimal case. In chess, a depth of about 6-7 moves can be reached.

- Can we establish a good ordering for chess? Heuristics: try captures first, threats, forward moves, backward moves, control the center, no knights at the edges, …

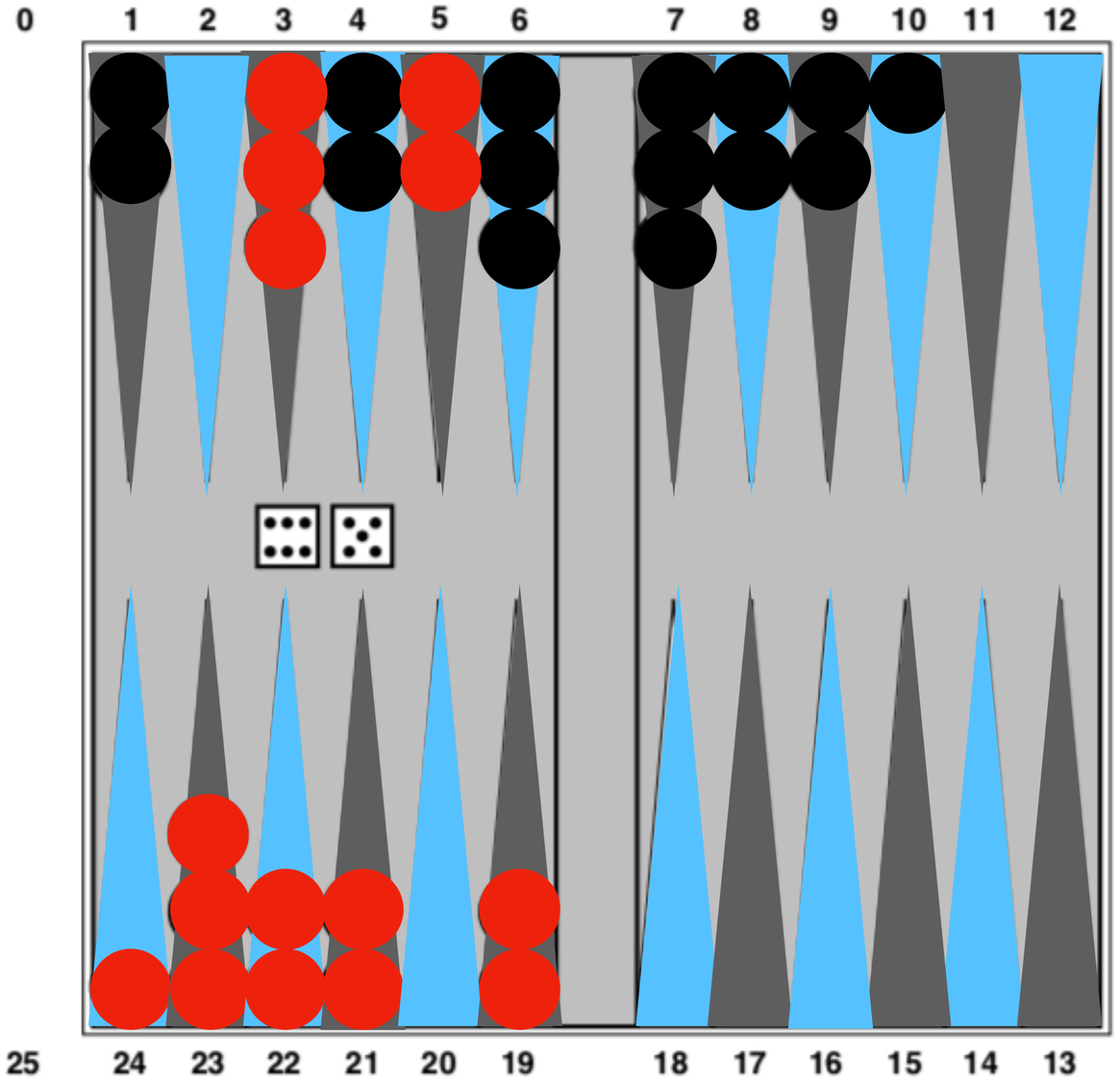
Games with Random Components

Some games involve probabilistic outcomes,

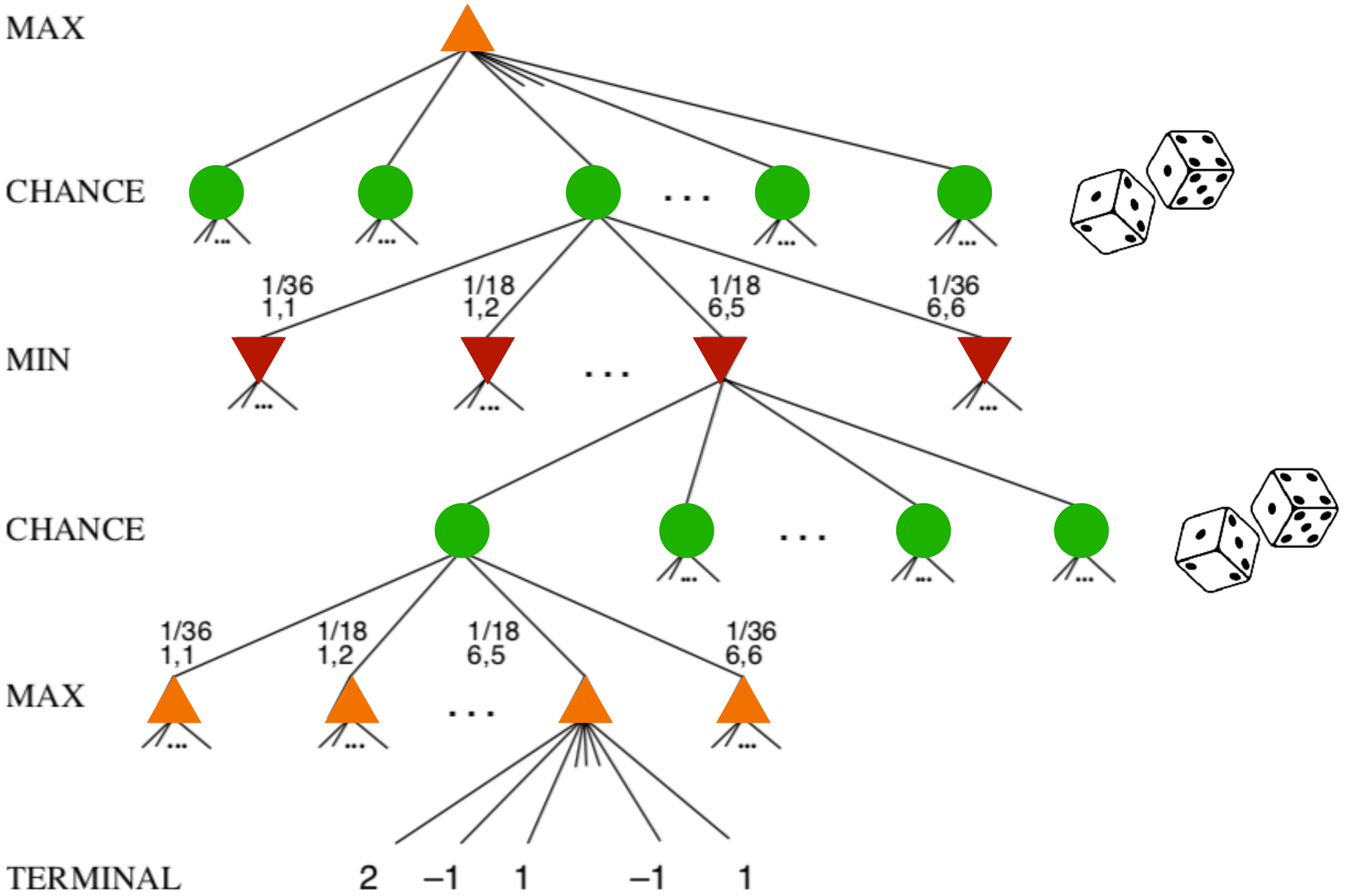
like e.g. the throw of two dice at backgammon.

For the example on the right, a player rolled a 6 and a 5 which lead to 4 legal moves.

The MinMax tree must contain also “random nodes”.



Games with Random Components



Creator: siridhata | Credit: Getty Images/iStockphoto

Expected Value Calculation

The Min and Max nodes are treated like before.

The "chance" nodes are treated calculating the expected value of a state.

The new algorithm is called expectiminimax:

$$\texttt{ExpectiMiniMax(s)} =$$

$$\begin{cases} \texttt{Utility(s, Max)} & \text{if s is a terminal state.} \\ \max_a \texttt{ExpectiMiniMax(Result(s,a))} & \text{if Max moves} \\ \min_a \texttt{ExpectiMiniMax(Result(s,a))} & \text{if Min moves} \\ \sum_r P(r)\ \texttt{ExpectiMiniMax(Result(s,r))} & \text{if we are in a chance node} \end{cases}$$

Corresponds to the average over the chance nodes

r = possible value of a dice roll

Summary

- A game can be defined by an initial state, the operators (or legal moves), a terminal test and a utility function (outcome of the game).
- In two-player games which are deterministic, zero-sum, discrete, turn-taking with perfect information, the MiniMax algorithm is optimal in selecting the moves adopting a depth-first search.
- An improvement is achieved through pruning techniques like alpha-beta pruning.
- There are alternatives, like the Monte Carlo random search coupled with some guidance for trying randomly only “good” moves.
- Often search is combined with look-up tables (e.g. the openings in chess)
- Games with imperfect information (poker, kriegspiel,…) can employ strategies consisting in averaging over possible states.
- Computer programs achieve human-like or better performance in newly all cases with humans retaining a little advantage in games with imperfect information.