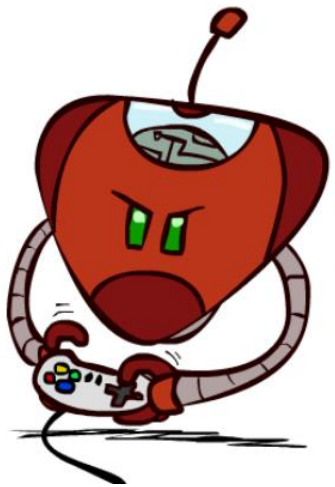
Grado en ingeniería informática
Artificial Intelligence 2021/2022

2.3 Adversarial Search: Games

Lara Quijano Sánchez



UAM
Universidad Autónoma
de Madrid

Problem solving using informed search

- Informed/ heuristic search
  - Introduction
  - Best-first search
  - Iterative improvement algorithms
  - Adversarial Search
    - Introduction
    - Minimax
    - Alpha-beta

Readings

- CHAPTER 6 of Russell & Norvig
- CHAPTER 12 of Nilsson

Some figures and code from http://aima.cs.berkeley.edu/

Introduction

- **History:**
  - "The theory of Games and Economic Behavior", 1944 John von Neumann, Oskar Morgenstern.
  - Ideas:
    - Optimal player strategy: Zermelo (1912), Von Neumann (1928)
    - Limitations in resources (approximate evaluation of the utility function): Konrad Zuse (1945), Norbert Wiener (1948), Claude Shannon (1950)
    - First chess program: Alan Turing (1951).
    - Learning: Arthur Samuel (1952-57)
    - Pruning of the game tree: MacCarthy (1956)
- **Multiagent environment + competition**

  Agents have different goals ⇒ adversarial search.
  - **Idealized setting** in which players with **divergent objectives** take **turns** to perform actions.
  - Agents can only perform "**legal moves**" (as defined by the game rules).
  - Moves are chosen according to a **strategy** that specifies a move for every possible move of the opponent.
  - The game ends when one of the agents reaches its goal (as measured by a **utility function**).

Types of games

- Classification criteria:
  - **Number of players**
    - Two players (e.g. backgammon, chess, checkers)
    - Multiplayer: mixed competitive / cooperative strategies (e.g. temporary alliances).
  - Properties of the **utility function**
    - **Zero-sum**: The sum of the utilities of the agents is zero regardless of the outcome of the game (e.g. chess, checkers)
    - **Constant sum**: The sum of the utilities of the agents is constant, regardless of the outcome of the game (equivalent to zero-sum games: utility normalization)
    - **Variable sum**: They are not zero sum. They can have complex optimal strategies where sometimes collaboration is involved (e.g. monopoly, backgammon)
  - **Information** available to the players
    - **Perfect information** (e.g. chess, checkers, go)
    - **Partial information** (e.g. almost all card games)
  - **Chance** elements
    - **Deterministic** (e.g. chess, checkers, go)
    - **Stochastic** (e.g. backgammon)
  - Unlimited / limited time (e.g. chess with clock).
  - Unlimited / limited movements (e.g. chess with a limit of 40 movements).

Adversarial search

- **Search problem:**
  - **Initial state.**
  - **Successor function**:
    (move current-state) → successor-state
  - **Terminal test**: determines if the state of the game is a terminal state (i.e. end of game)
  - **Utility (objective or payoff function)**: Numerical evaluation of **terminal** states.

  Game tree: Initial state + interleaved legal moves.

- Consider the following simple game:

  Two players make alternate moves until one of them wins (the other loses) or there is a tie.

  Each player has a perfect model of the deterministic environment and the effects produced by legal moves.

  There may be computational / temporal limitations to the movements of the agents.
  - Two agents
  - Perfect information
  - Deterministic
  - Zero-sum game or constant sum

Adversarial search

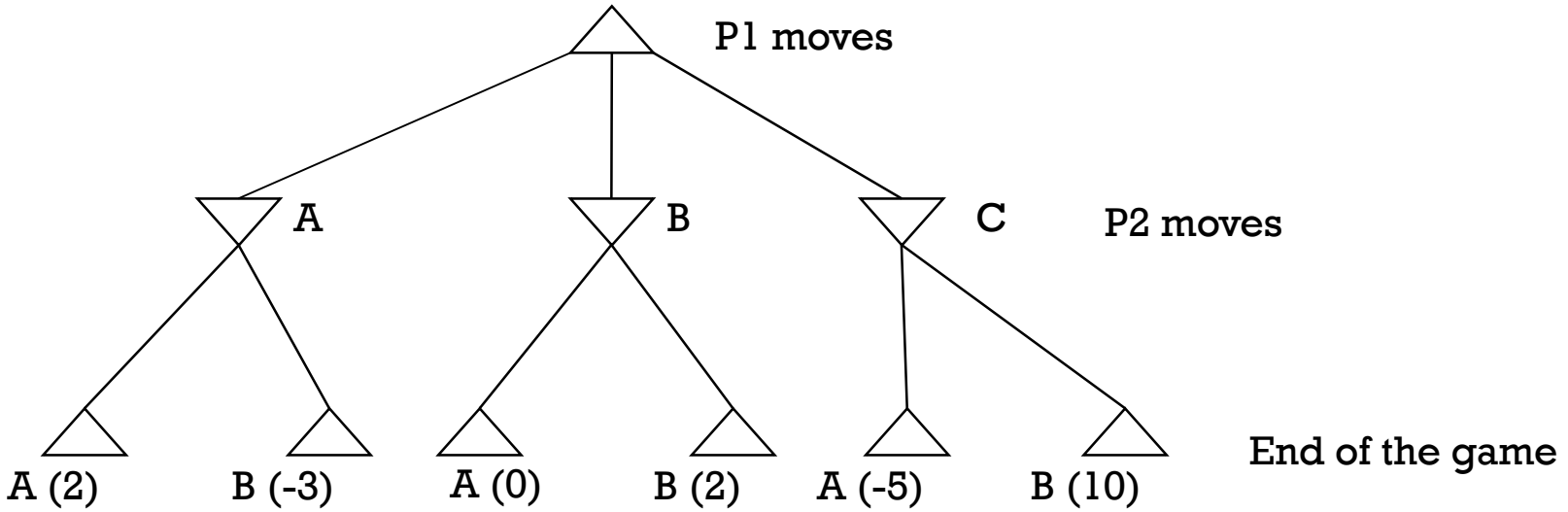
- A game tree is an explicit representation of all possible sequences of moves in a game
  - The root node corresponds to the initial state of the game.
  - Players alternate their movements
  - To generate the next level from a given node, we generate as many child nodes as there are possible movements for the player who has the turn at the node considered.
  - The leaves correspond to terminal states (end of game)
  - A path from the root (the initial state of the game) to a leaf represents a complete game
- Search algorithms seen so far do not work
- The problem is no longer to find a way in the game tree (since this depends on the future moves that the opponent will make), but to decide the best move given the current state of the game

Representation of games

- Matrix of final balances: Value of the utility function for each player given the actions of the players

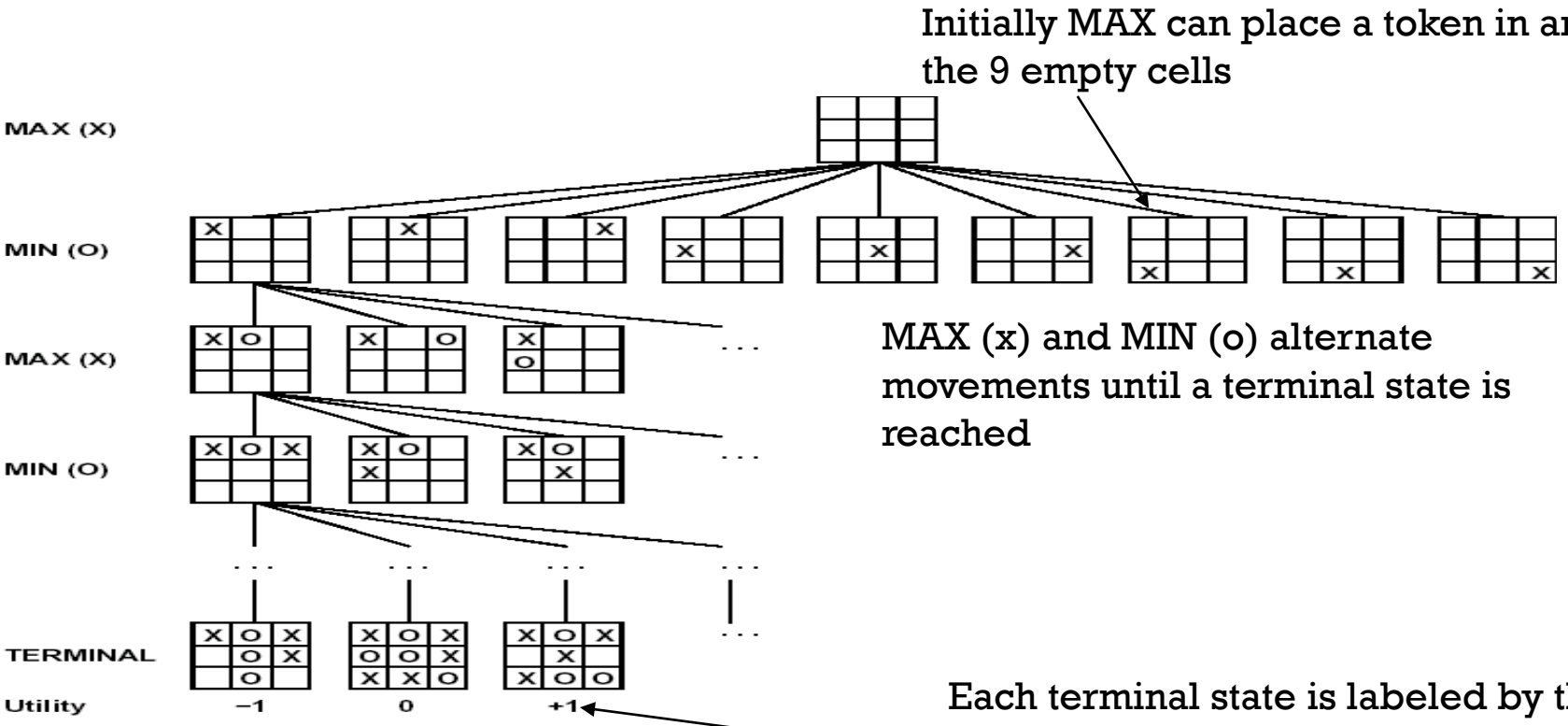
		P2	
		A	B
P1	A	2	-3
	B	0	2
	C	-5	10

- Game tree



Game tree for tic-tac-toe

- Initial state: Empty 3x3 board
- Movements: Place a token (MAX: x, MIN: o) in one of the empty cells.
- Terminal test: 3 tokens of the same player are aligned



Optimal strategies

- Let us consider a game with two players: **MAX** and **MIN**.
  - **MAX moves first**.
  - **MAX and MIN alternate their moves**: In the game tree, the nodes of even depth correspond to MAX. The nodes of odd depth correspond to MIN.
  - A **ply** of depth **k** in the game tree corresponds to the tree nodes of depths 2k and 2k+1.
- Formal description of the game:
  - Initial state: Initial board configuration + identity of first player.
  - Successor function: Successors(n)
  - Terminal test: Terminal(n).
  - Utility function: Utility (n), only if n is a terminal node.
- **Optimal strategy for MAX**: Strategy that performs at least as well as any other, assuming MIN is an infallible opponent.
- **Minimax strategy**:

  Use the **minimax value** of a node to guide the search: **Utility of a node** (from the point of view of MAX) **assuming that both players play optimally** from there until the end of the game

$$\text{minimax}(n)\begin{cases} \text{Utility}(n) & \text{if n is a terminal node.} \\ \max\{\text{minimax}(s); s \in \text{successors}(n)\} & \text{if n is a MAX node.} \\ \min\{\text{minimax}(s); s \in \text{sucesores}(n)\} & \text{if n is a MIN node.} \end{cases}$$

The minimax algorithm

```
function MINIMAX-DECISION(state) returns an action
    inputs: state, current state in game
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
    return v

function MIN-VALUE(state) returns a utility value
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← ∞
    for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
    return v
```

- **Complete** only if the **tree game is finite** (note that there may be optimal finite strategies for infinite trees)
- **Optimal only if the opponent is optimal** (if the opponent is suboptimal, we can use their weaknesses to find better strategies. Dangerous).
- **Exponential temporal complexity** $O(b^m)$;

  m = maximum depth of the game tree
- **Spatial complexity**: **Linear** if **depth-first search** is used $O(b \cdot m)$

Minimax

- The player who has the turn is MAX. The opponent is MIN
- MAX (MIN) nodes are those in which MAX (MIN) has the turn
  - The root node is a MAX node (depth 0).
  - Even depth nodes are MAX nodes
  - The nodes of odd depth are MIN nodes
- A terminal node is assigned a utility value from the point of view of MAX.
  - Eg. Chess, coded as a game of

    zero sum: win MAX (+1); ties (0); Win MIN (-1)

    constant sum: wins MAX (2); ties(1); win MIN (0)
- The MINIMAX value of a node is obtained by propagating the values of the utility function corresponding to the terminal nodes towards the root of the tree:
  - MINIMAX value of a node **MAX = maximum of the MINIMAX values of its children**.

    MAX tries to maximize his advantage by choosing the best possible move on his turn.
  - MINIMAX value of a node **MIN = minimum of the MINIMAX values of its children**.

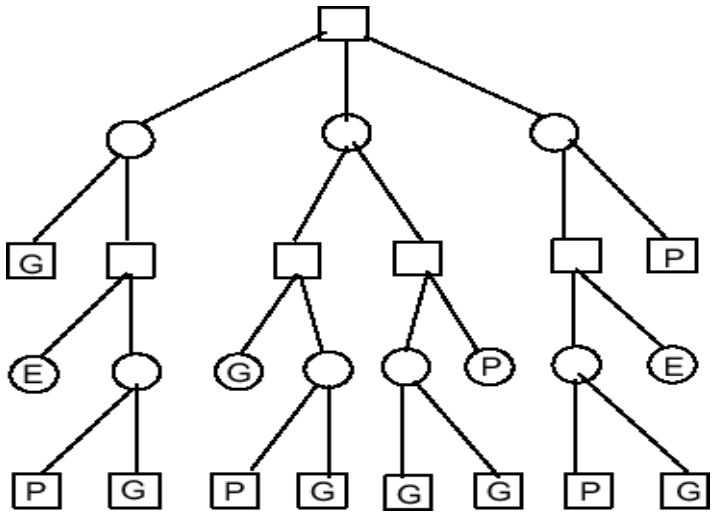
    MIN tries to minimize the MAX score by choosing the movement that hurts the most MAX
- The resulting sequence of plays is the MINIMAX strategy.

Minimax

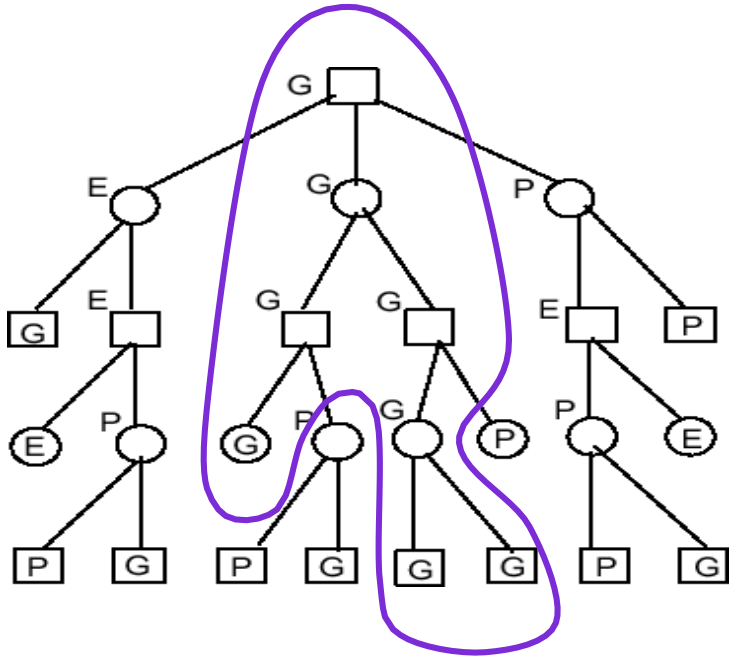
- The MINIMAX value of the root node corresponds to the best value of the utility function that MAX can achieve in the game assuming that the opponent is optimal.
  - Even if the result is optimal, MAX may not be able to win the game.
- Solving a game tree means finding the MINIMAX value of the root node using the MINIMAX algorithm.

Example

Unsolved game tree



Solved game tree



G(WIN) = 1, E(TIE) = 0, P(LOSE) = -1

Nodes MAX: squares

Nodes MIN: circles

Winning tree for MAX

Minimax

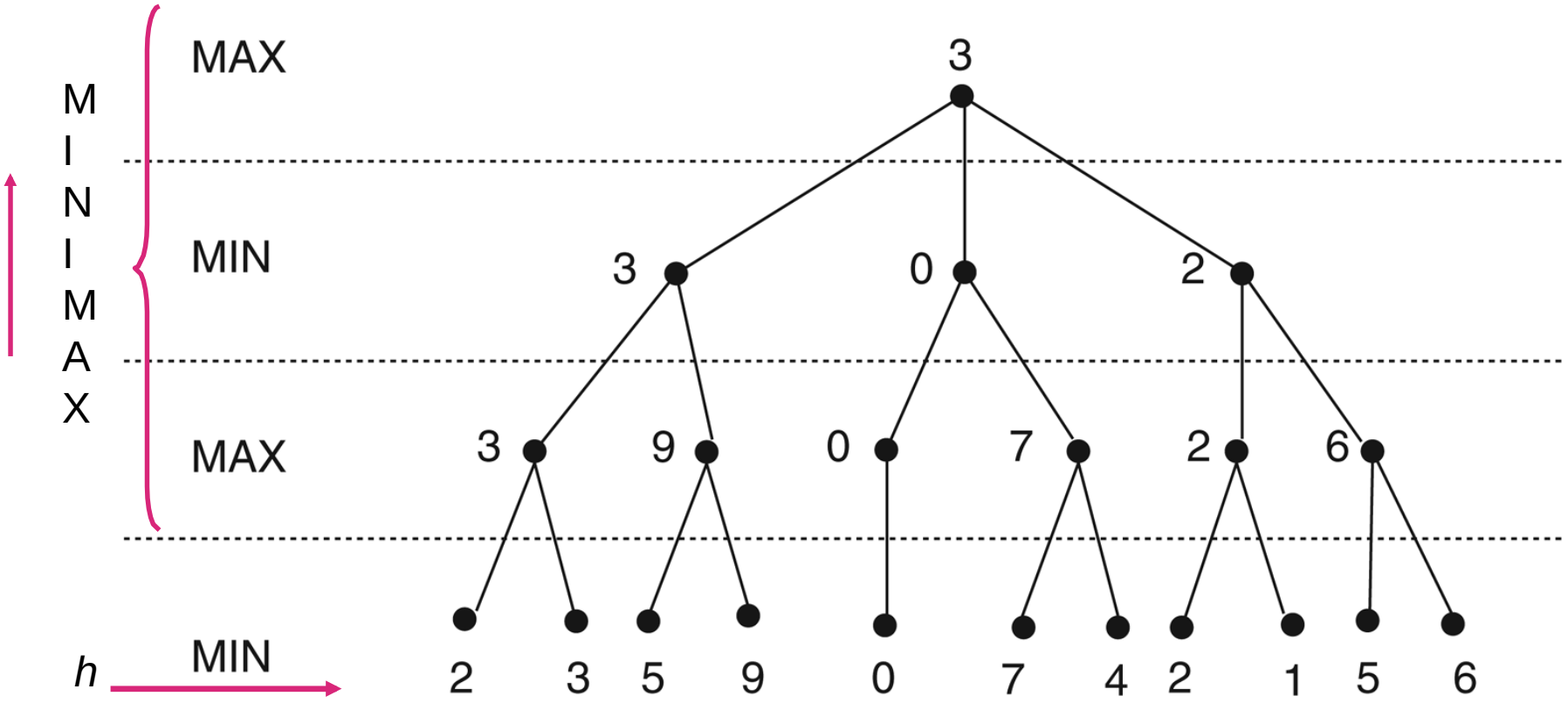
- The game tree is generated by depth- first search
  - The spatial complexity of the MINIMAX algorithm is linear in the maximum depth of the tree (m) and in the branching factor (b): O (bm)

  It is not necessary to keep the entire tree in memory.
  - The temporal complexity of the MINIMAX algorithm is exponential: O (b^m)
    - The labeling method described requires generating the entire game tree.
    - In practice, for most games, fully developing the game tree is an impractical task

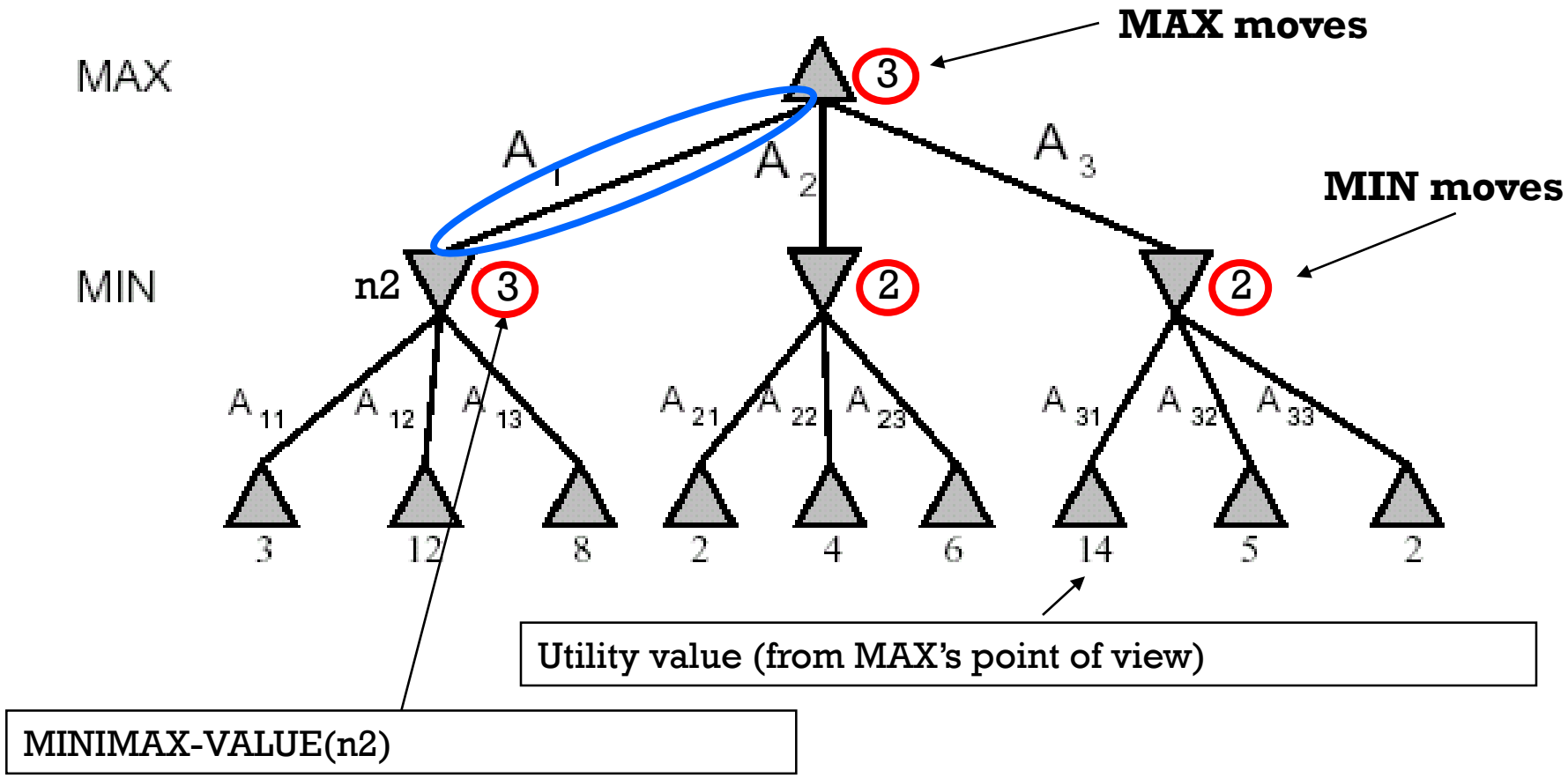
    - Checkers: The entire tree has approximately 10^{40} nodes
      - Assuming they are generated. Generating the entire tree would require 10^{21} centuries (3 billion nodes / sec.)
    - Chess ~ about 10^{120} nodes and about 10^{101} centuries
      - Even if a fortune teller were to provide us with a tagged tree, it would be very expensive to store or walk through it to find a solution
- In practice, a pre-pruning of the tree is carried out: A search depth limit is established and the terminal nodes (not necessarily corresponding to the end of the game) are labeled by the value of an evaluation function (ideally, monotonously related to the minimax value)

Example 1: 3 layer minimax value

Calculation of the minimax value looking forward 3 layers (levels)



The minimax procedure: example 2



- The best move for MAX is A_1 (maximizes utility)
- The best reply for MIN is A_{11} (minimizes utility)

Imperfect decisions

- **Problem**:
  - In many real-world games the **utility function** is **too expensive** to compute.
  - With **limited resources** or games with infinite trees it is infeasible to carry out a complete search.

- **Solution:** Limited horizon search
  - Define a **heuristic evaluation function, eval(n)**, which is an **estimation** of the actual utility function **utility(n)**.
  - Use a **cutoff test** to determine when to stop the search and compute **eval(n).**
  - **Properties of eval(n):**
    - **eval(n)** should order terminal nodes in the same manner as **utility(n).**
    - **eval(n)** should be strongly correlated with **utility(n)** in non terminal nodes.
    - **eval(n)** should be easy to compute.

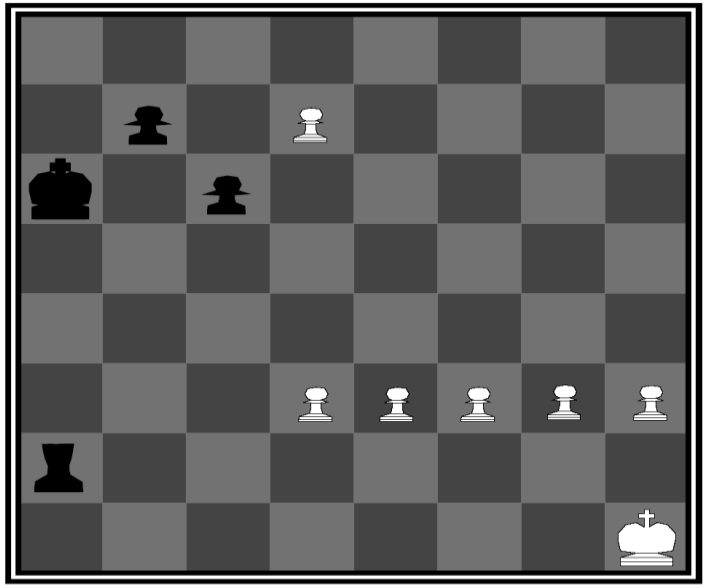
Limited horizon search

Change in minimax search:

"If terminal-test(state) then return utility(state)"

"If cutoff-test(state, depth) then return eval(state)"

- Depth-limited search: Stop search when depth reaches fixed limit.
- Iterative deepening depth-first search: More robust if there are time limitations.
- **Horizon effect**: Disaster / success may lie right behind the search horizon.
  - Do not stop the search in "live" positions. Stop only if the state is **quiescent** .
  - **Singular extensions**:
    Explore deeper positions that are clearly advantageous.
  - **Forward pruning**:
    Remove search branches that are clearly inferior
    (danger: we can miss a masterly sacrifice).



Black to move

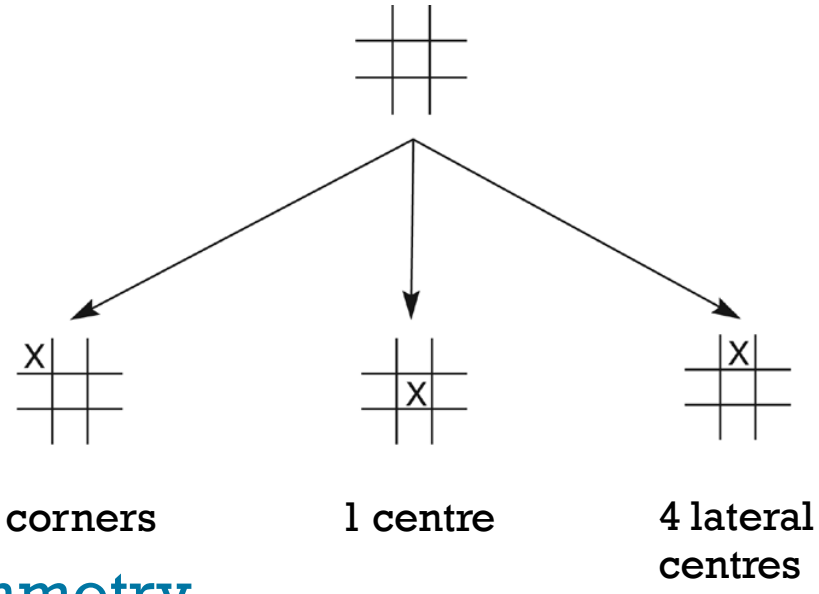
The evaluation function

- At the terminal nodes (end-of-game), the value of the evaluation function must match that of the utility function.
- In non-terminal nodes (eg. if a depth limit has been established in the search), it is an estimation of the minimax value in that node.
- The evaluation function (heuristic) takes into account different characteristics of the game state:
  - Number of own and opponent's pieces (advantage of one over the other)
  - Weighting of the importance of pieces in chess
  - Positions of the pieces on the board
  - Weak points (isolated pawn, etc.)
  - Positional characteristics (king protection, maneuverability, control of the center of the board, etc.)
- The value of the evaluation function must be able to be calculated efficiently: The more time spent calculating this value, the less time can be spent searching.

It is necessary to reach a compromise between the quality of the estimation and the computational cost necessary to obtain said estimation.

Example tic-tac-toe

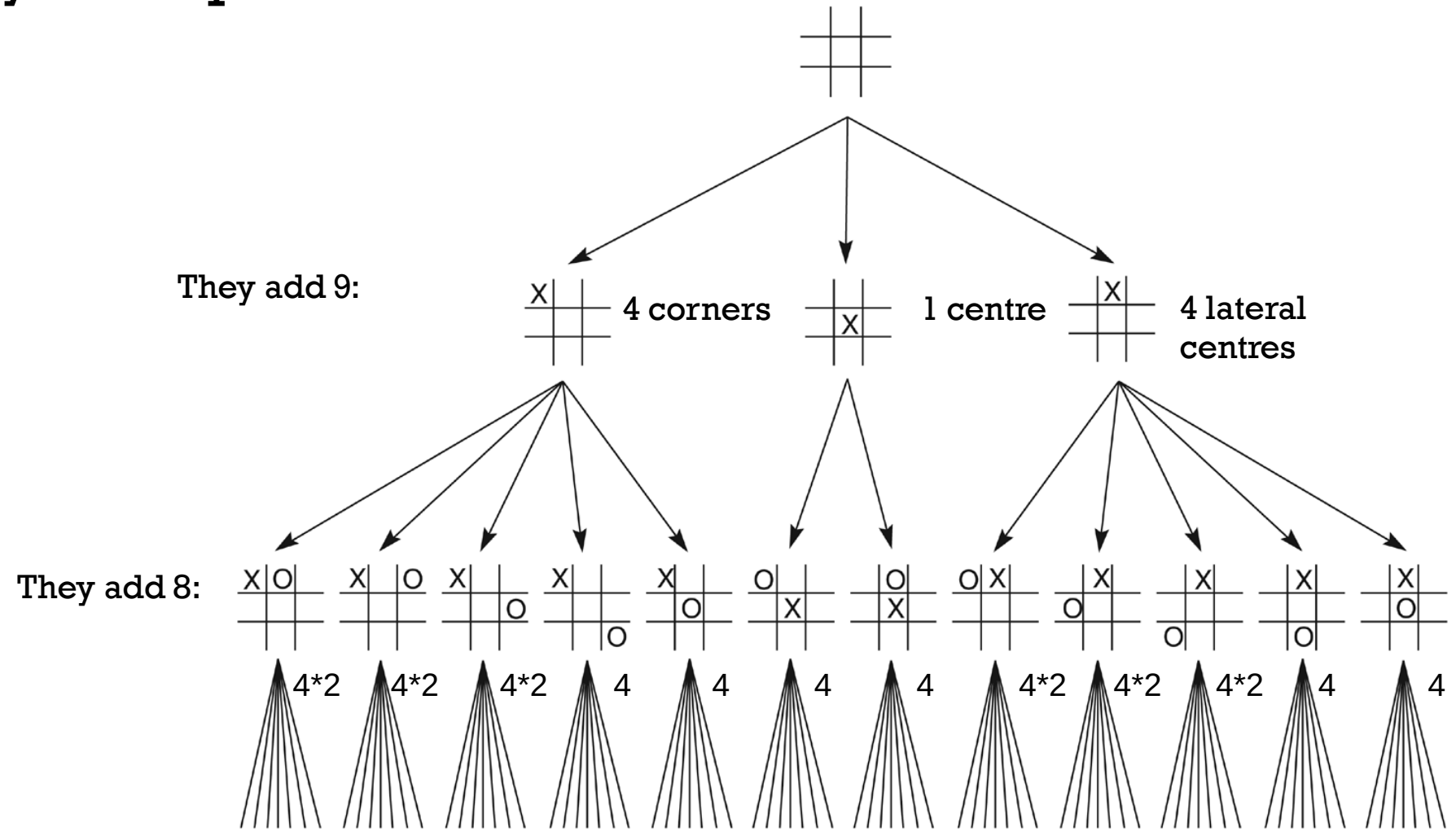
- Direct representation:
- 9 initial movements, 8 possible second movements, …
- Many different operators and "states" (board configurations)
- Representation that takes symmetries into account
  - 3 initial moves
  - Corner
  - center of the board
  - center of a side



- The remaining 6 configurations are equivalent to any of those considered by symmetry.
- Reduces the state space and therefore the complexity of the search

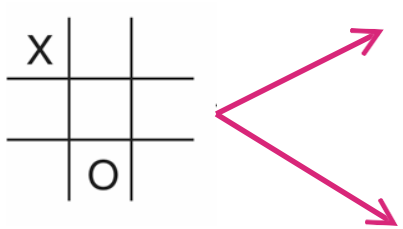
Example symmetric reduction of tic-tac-toe

- They are equivalence classes



Example evaluation function for tic-tac-toe

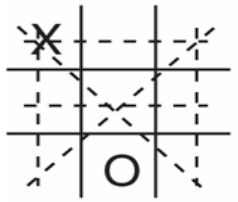
- Utility function:
  - -∞ if in terminal state MIN wins(and therefore MAX looses)
  - +∞ if in the terminal state MAX wins (and therefore MIN looses)
- Evaluation function: eval (n) = M (n) - O (n)
  - M (n) = n° of possible MAX winning lines (counting empty ones)
  - O (n) = n° of possible MIN winning lines (counting empty ones)

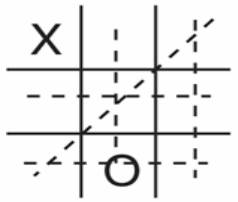


X has 6 possible win lines

O X has 5 possible win lines

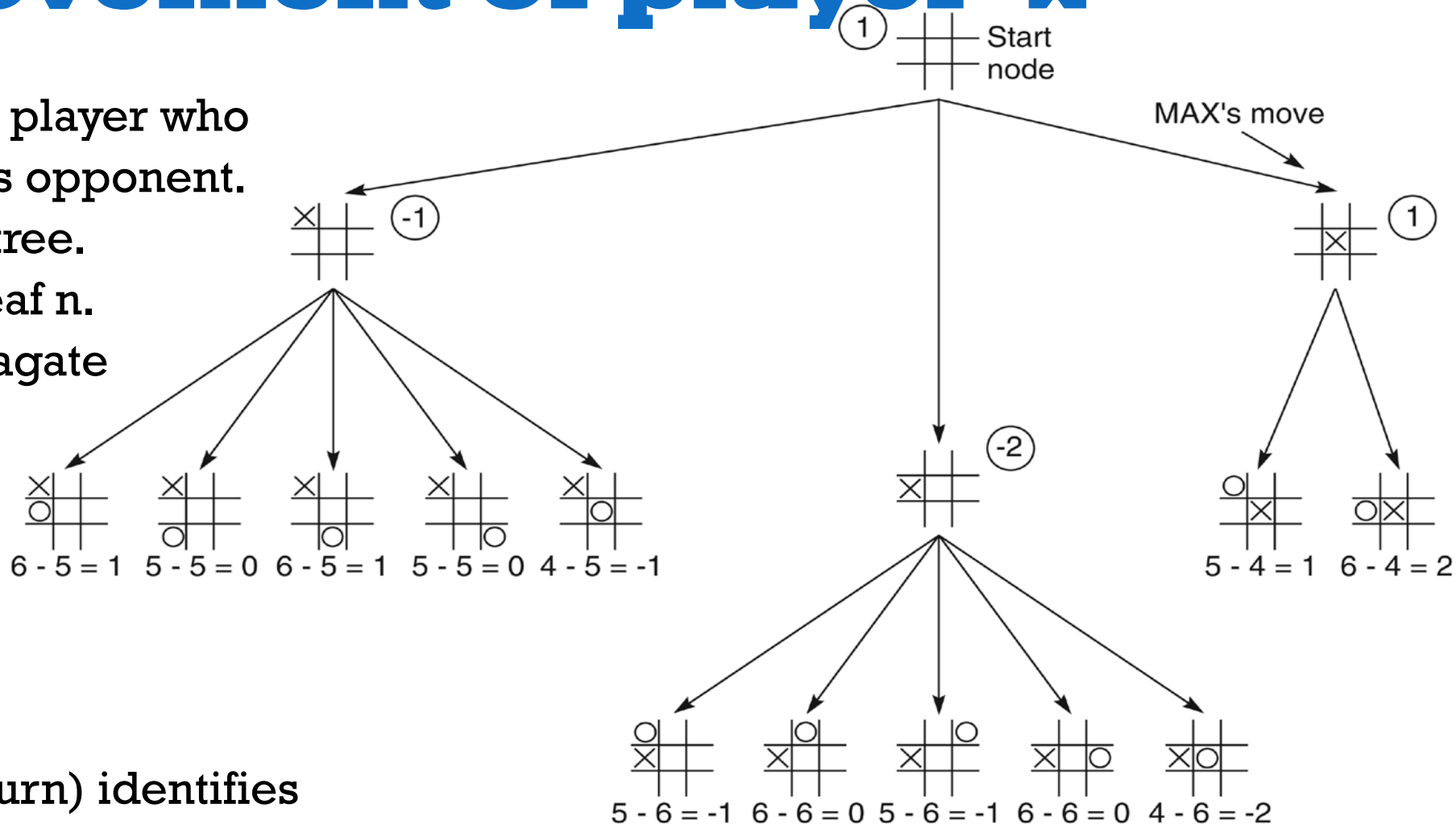
$h(n) = 6 - 5 = 1$





MINIMAX with depth limit 2: Initial movement of player 'x'

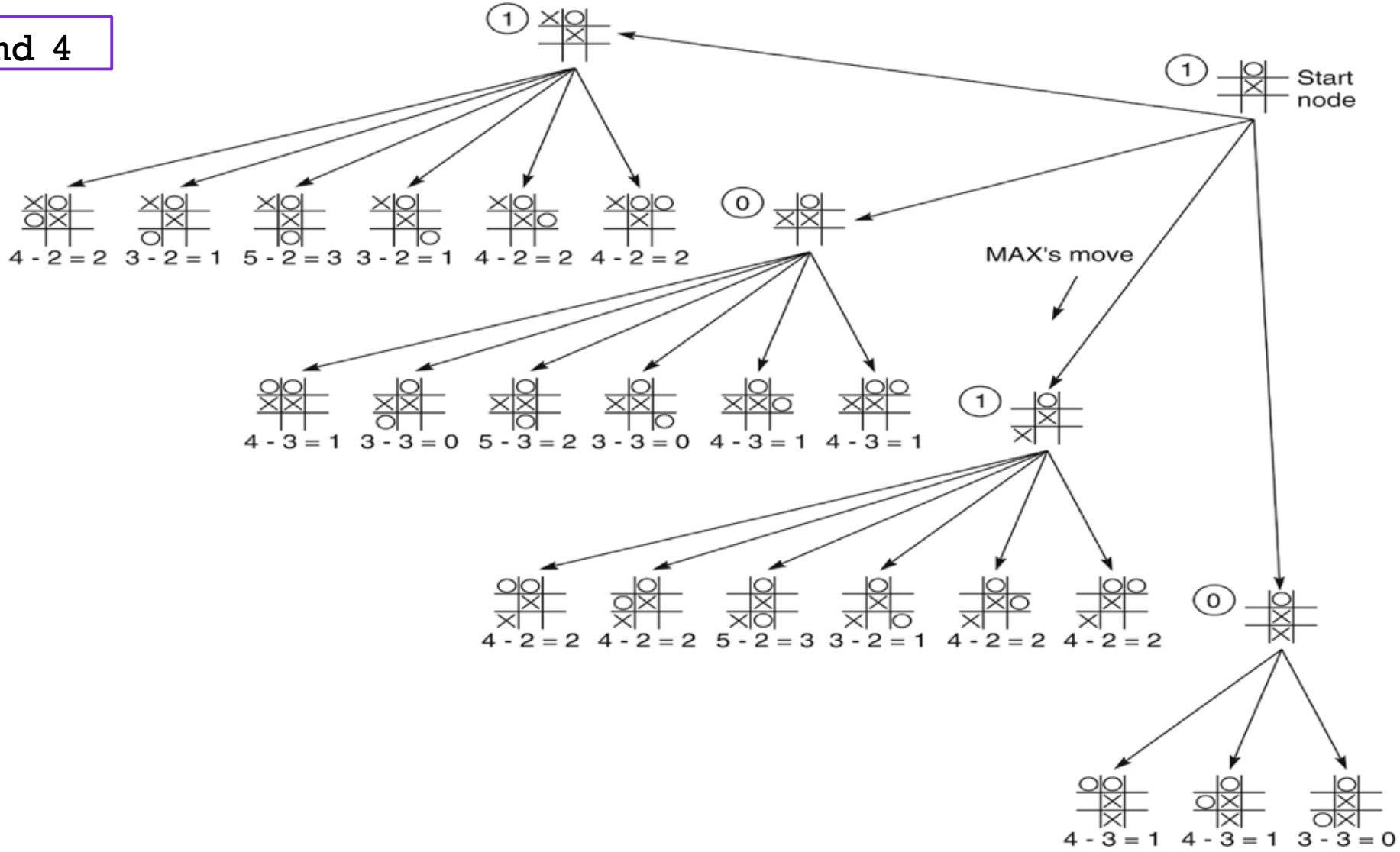
1. Identify as MAX as the player who has the turn and MIN as his opponent.
2. Generate the 2 levels tree.
3. Estimate eval (n) for leaf n.
4. Use MINIMAX to propagate this value up the tree.



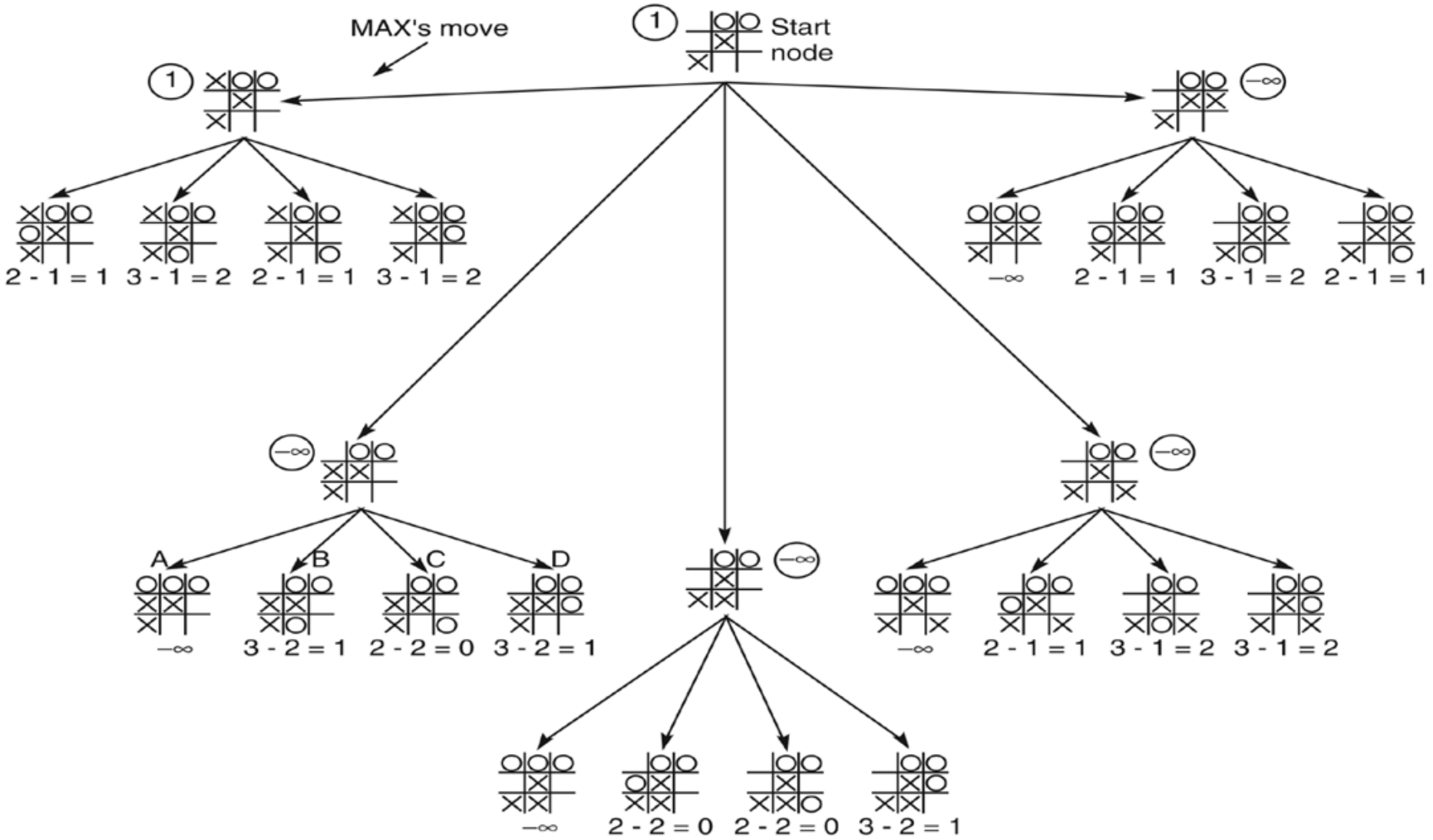
- MAX (or MIN if it is her turn) identifies the best move and performs it (game state is modified).
- Repeat step 1 alternating players (there is a shift change) until the end of the game.

MINIMAX with depth limit 2: second movement of player 'x'

Repeat steps: 1, 2, 3 and 4



MINIMAX with depth limit 2: third movement of player 'x'



Evaluation functions

How do we build good evaluation functions?

- Calculate the **expected value** of the utility by estimating the "probabilities" of different final results from current position.

$$\text{eval}(n) = \sum_{\text{outcomes}} \text{Probability}(n \rightarrow \text{outcome}) \times \text{utility}(\text{outcome})$$

E.g., 50% winning chances (utility 1)
25% losing chances (utility -1)
25% tying chances (utility 0)

f(n) =0.50*1 + 0.25*(-1) + 0.25*0 = 0.25

- **Feature-based evaluation**:
  Characterize the current state by a set of **features** $\{ f_1(n), f_2(n), \ldots, f_K(n) \}$.

$$\text{eval}(n) = F[f_1(n), f_2(n), \ldots, f_K(n)]$$

$$e.g., \text{eval}(n) = \sum_{i=1}^{K} w_i f_i(n)$$

- F contains expert knowledge:
  - Given by experts (eg chess Deep Blue de IBM, 1997)
  - F can be **learnt** (machine learning) from experience. For example, it can be a neural network trained by reinforcement learning from games that the computer plays against itself (e.g. in chess alphaZero de DeepMind, 2018)

Chess

Approximately 10^{40} nodes, b = 35.

- Let us assume that each successor can be generated in 1 μs (10^{-6} s)
  - A full exploration would take $3 \cdot 10^{26}$ years (the age of the universe is ~ 10^{10} years).
  - If we assume a time limit of 1 minute per movement, we can only perform a full exploration up to depth 5.
  - With alpha-beta pruning + perfect order we can perform a full scan up to depth 10.

  - Simple evaluation function (weighted linear function) based on the **material value**:

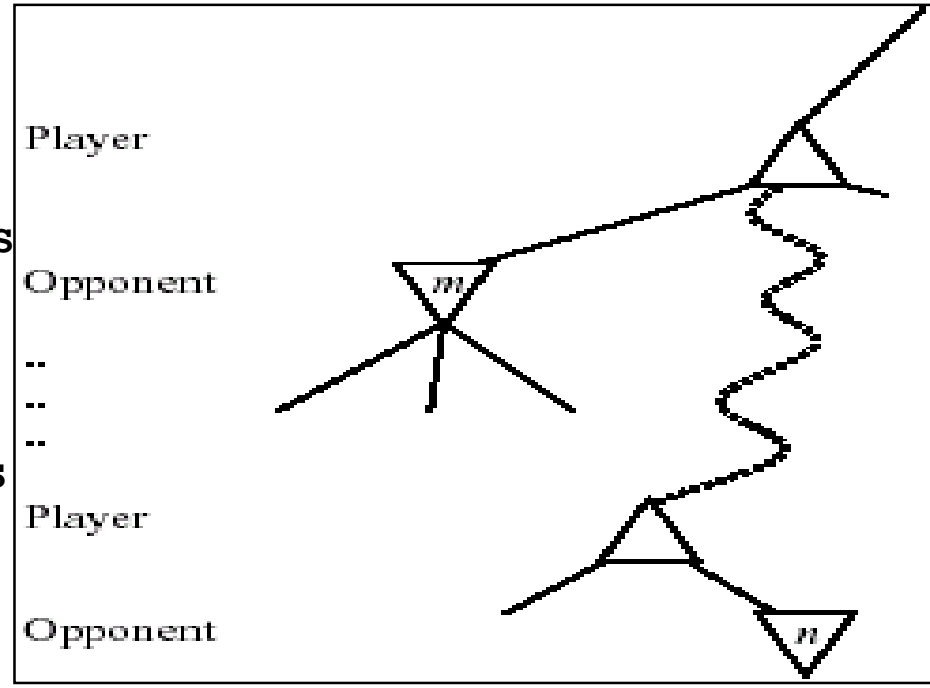
    initially MAX = white;

    pawn=1; knight and bishop=3; rook = 5; queen = 9.

    f = (n-white-pawns)*1 + (n-white-bishops)*3 +...

    - (n-black-pawns)*1 - (n-black-bishops)*3 -...
  - More complex evaluation functions take into account qualitative features such as "control of the center" , "good position of the king", "good structure of pawns".
  - Use of libraries of moves (openings, end of game,)

Alpha-beta pruning



- Observation: To calculate the minimax value of a node many times it is necessary to explore exhaustively.

  The algorithm performs a depth-first search from a given node.

  If during the search the nodes **m** and **n** are found in different subtrees and node **m** is better than **n** then, assuming optimal decisions, current game will never get to node **n**.

- Take into account **in each node an interval** $[\alpha, \beta]$ that contains the minimax value of the node, and **update** the interval limits **as the search progresses**
  - α is the value of the **best** alternative **for MAX** found so far (i.e. with **higher** value) $\Rightarrow$ α values never decrease in a MAX node
  - β is the value of the **best** alternative **for MIN** found so far (i.e. with **lowest** value) $\Rightarrow$ β values never increase in a MIN node.
- **Rules to stop the search**:
  - **α-cutoff**: Stop the search on a MIN node whose β-value $\leq$ α-value of any of its MAX ancestors.
  - **β -cutoff**: Stop the search on a MAX node whose α-value $\geq$ β-value of any of its MIN ancestors.

Minimax + alpha-beta pruning + depth-first search

The **value** α represents a **lower bound for the MINIMAX value at a MAX** node (the worst that MAX could do). It is initialized to Min-utility. It never decreases.

The **value** β represents an **upper bound for the MINIMAX value at a MIN** node (the best MIN could do). It is initialized to Max-utility. It never increases.

1. Initialization at root node: $[\alpha, \beta] \leftarrow$ [min-utility, max-utility]

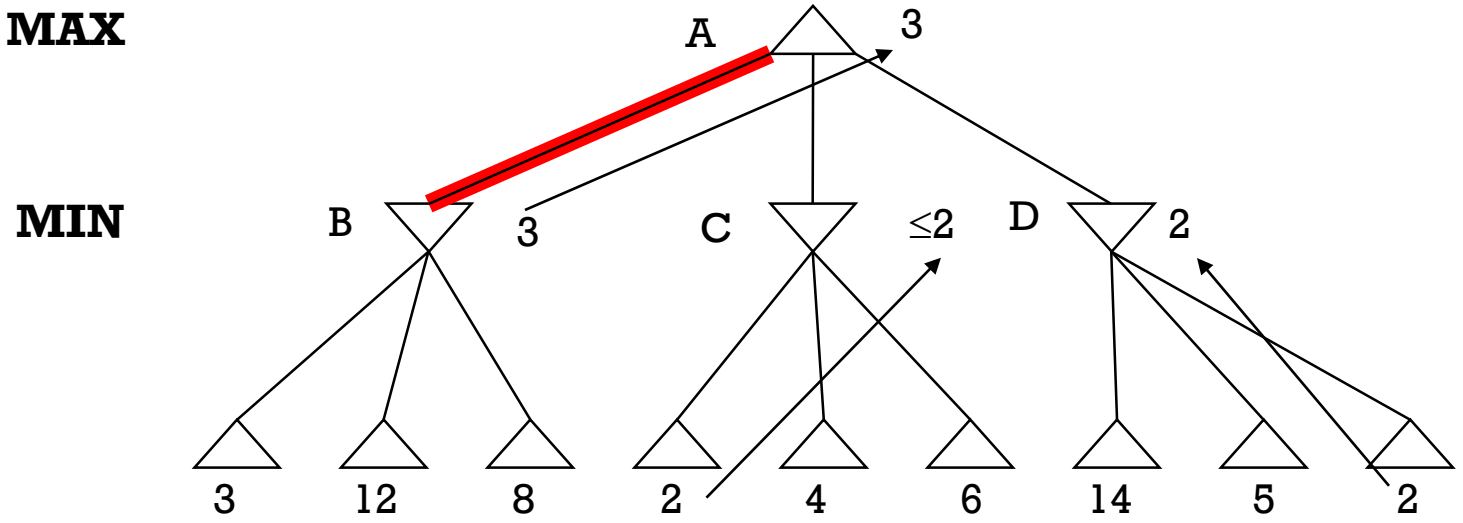
In case the minimum (min-utility) and maximum (max-utility) values that the utility function can reach for the game under consideration are not known, the initialization $[\alpha, \beta] \leftarrow [-\infty, +\infty]$ will be used

2. From the root, the values of $[\alpha, \beta]$ are propagated from the parent node to each child node by traversing the tree first in depth until the child node:
   - ❑ is terminal: α o**r** β = utility (n)
   - ❑ is at the present depth limit: α o**r** β = eval (n)
3. In the backtracking process (propagation of information from the leaves to the root in the search tree, the intervals of the internal nodes are updated as follows:
   - ❑ In a MAX node the value α is updated:

   $\alpha \leftarrow$ maximum (α , β's of successors of MAX generated up to that moment).
   - ❑ In a MIN node the value β is updated:

   $\beta \leftarrow$ minimum (β , α's of MIN successors generated up to that point)
4. **Pruning**: If, as a result of **updating the values** of α, β, , $\alpha \geq \beta$ is fulfilled, **pruning is performed in that node and the search is not continued** for the rest of the successors not yet explored of that node.

Example: alpha-beta pruning



Alpha-beta pruning

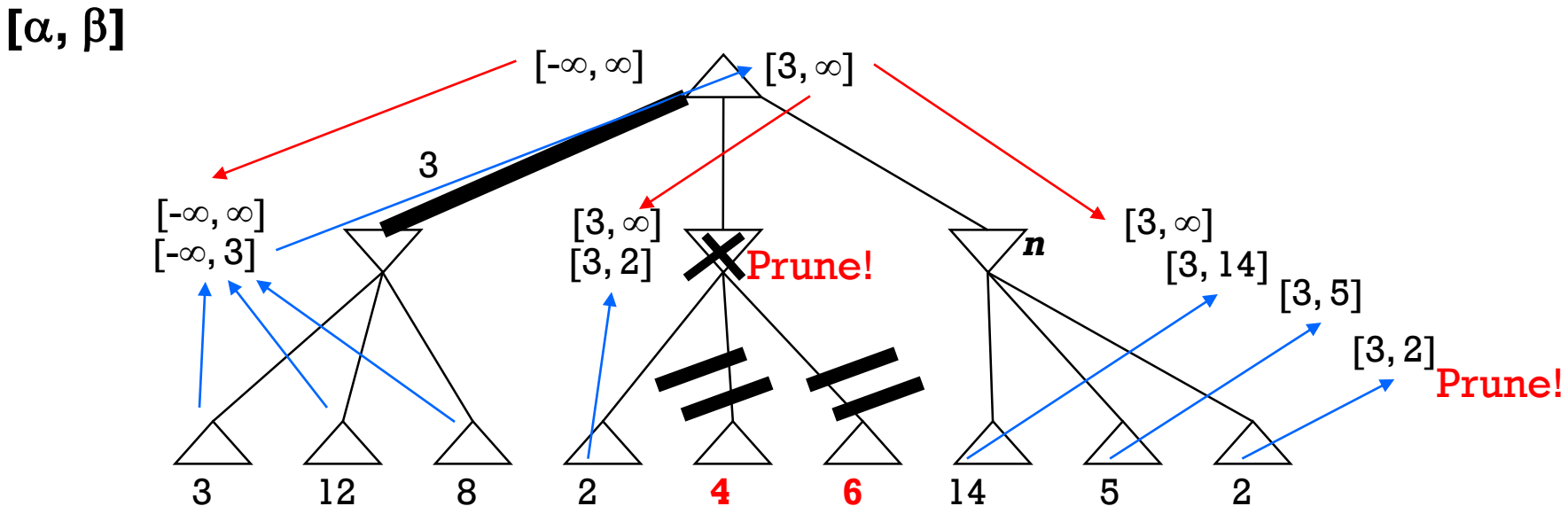
```
function ALPHA-BETA-DECISION(state) returns an action
    return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))

function MAX-VALUE(state, α, β) returns a utility value
    inputs: state, current state in game
            α, the value of the best alternative for MAX along the path to state
            β, the value of the best alternative for MIN along the path to state
    if TERMINAL-TEST(state) then return UTILITY(state)
    v ← −∞
    for a, s in SUCCESSORS(state) do
        v ← MAX(v, MIN-VALUE(s, α, β))
        if v ≥ β then return v
        α ← MAX(α, v)
    return v

function MIN-VALUE(state, α, β) returns a utility value
    same as MAX-VALUE but with roles of α, β reversed
```

- **Pruning does not affect the final result.**
- **The efficiency of pruning depends on the order** in the search: It is better if good movements are explored first.
  - **Worst case: There is no improvement**
  - **Random order: $O(b^{3d/4})$** $\Rightarrow b^* = b^{3/4}$ (Pearl, 1984)
  - **Perfect order: $O(b^{d/2})$** $\Rightarrow b^* = b^{1/2}$.

    Donald E. Knuth; Ronald W. Moore; *An analysis of alpha-beta pruning*. Artificial Intelligence 6(4); 293-326 (1975)

    The use of simple heuristics often leads to b* close to the optimum (e.g. first examine the movements that were found to be good in the previous move)

Alpha-beta pruning: Example I



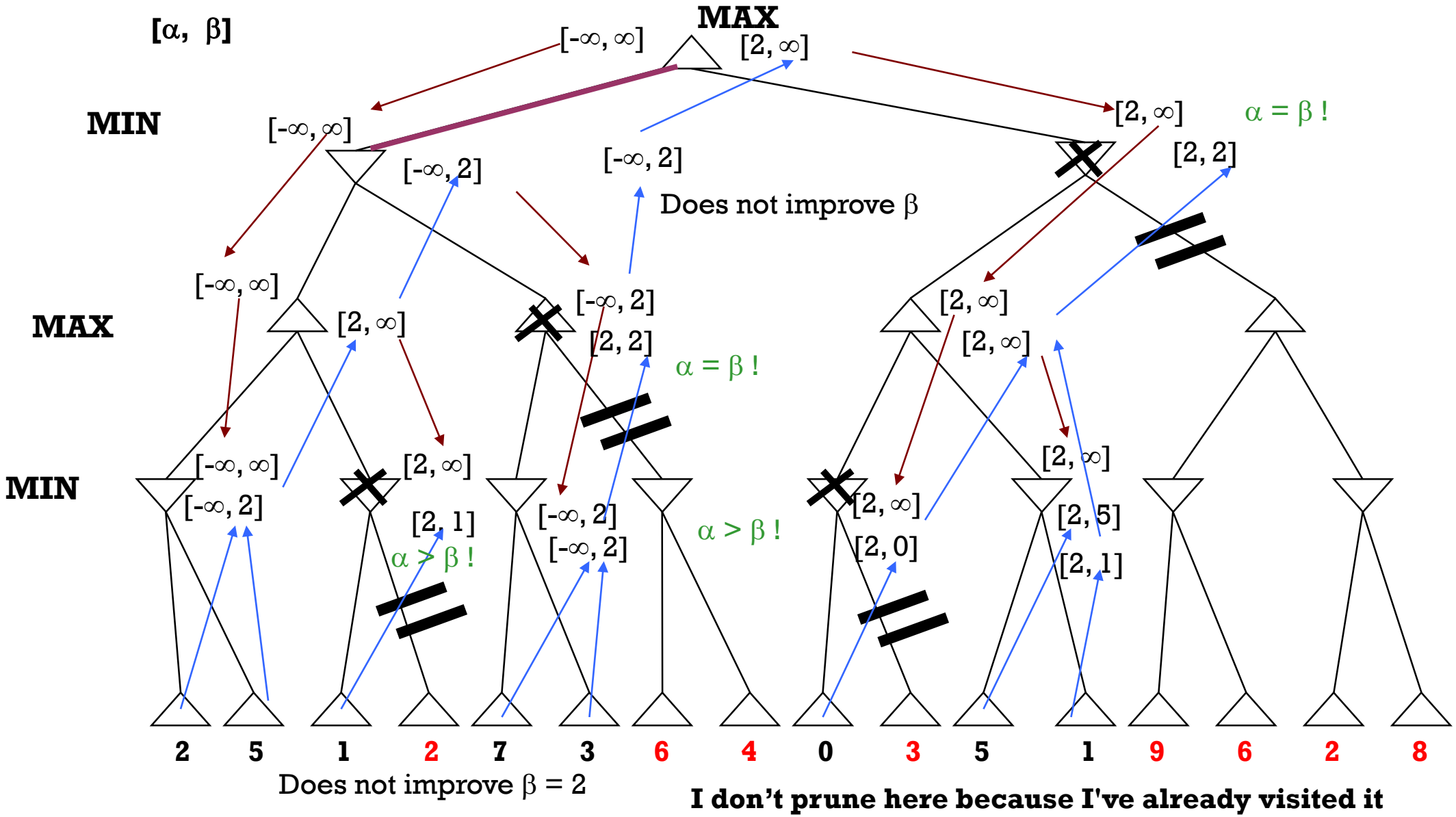
The order in the search is important for the efficiency of pruning

Pruning algorithm

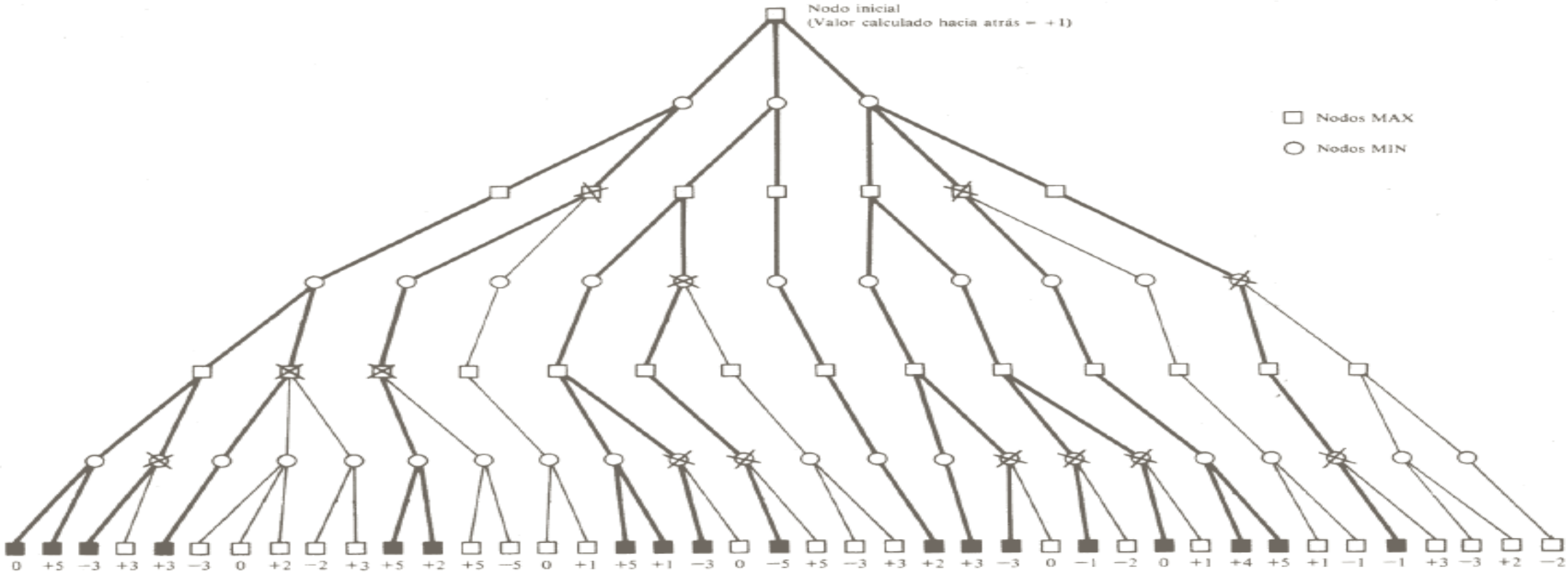
In MAX node: α ← max(α, β's of successors)

In MIN node: β ← min(β, α's of successors)

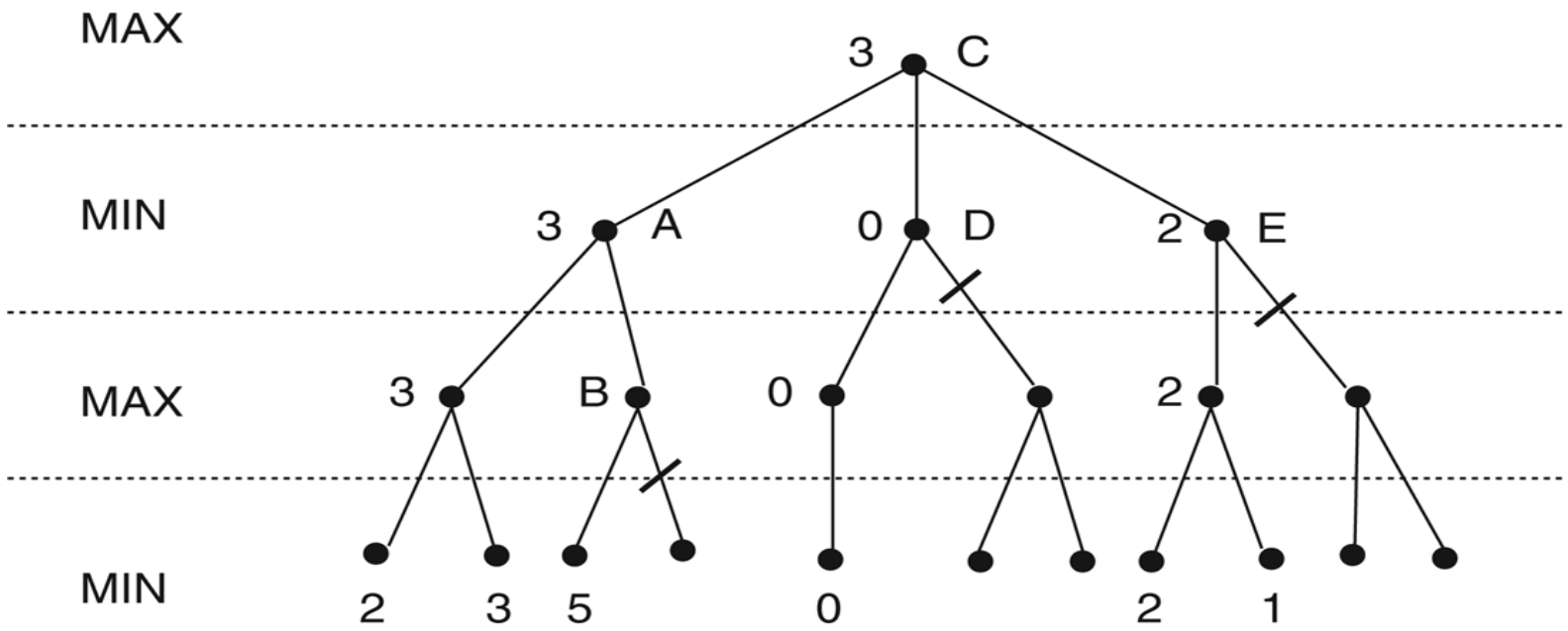
Alpha-beta pruning: Example II



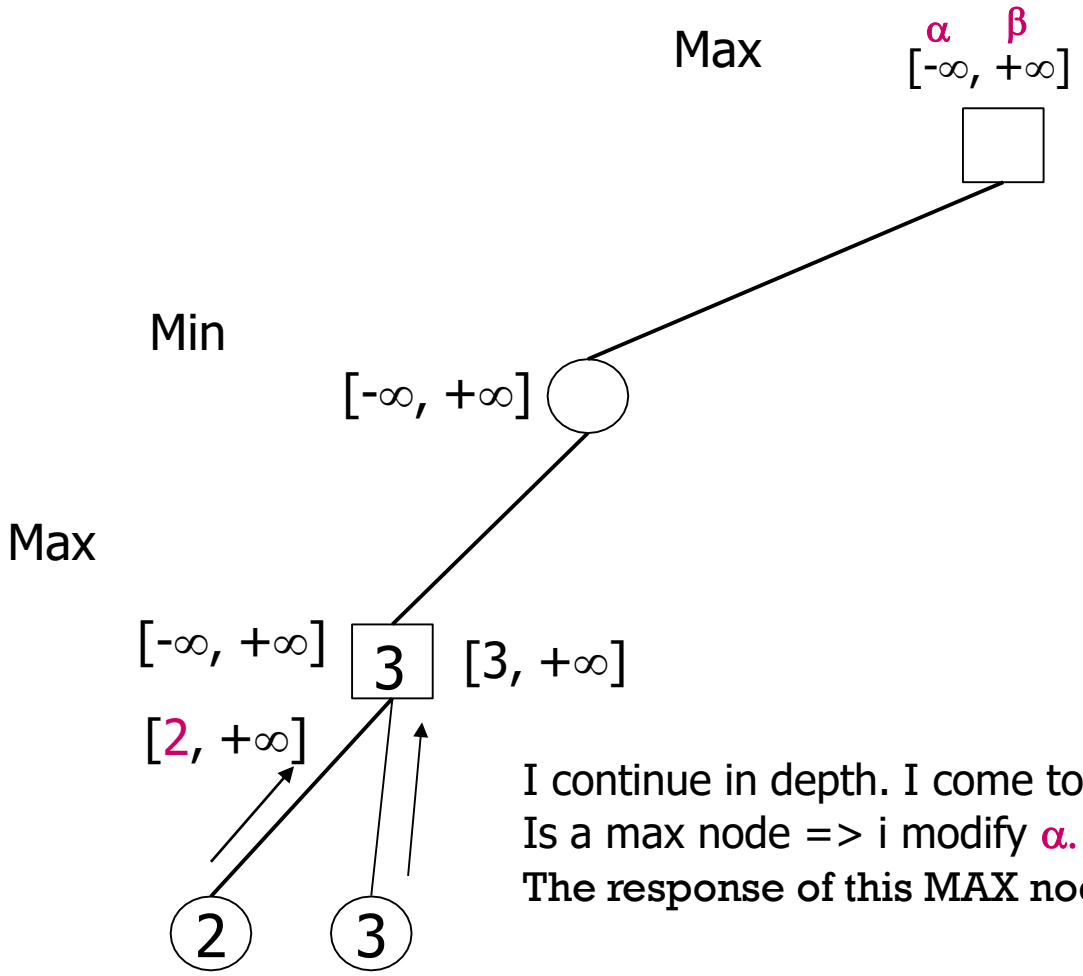
Alpha-beta pruning: Example III



Example IV



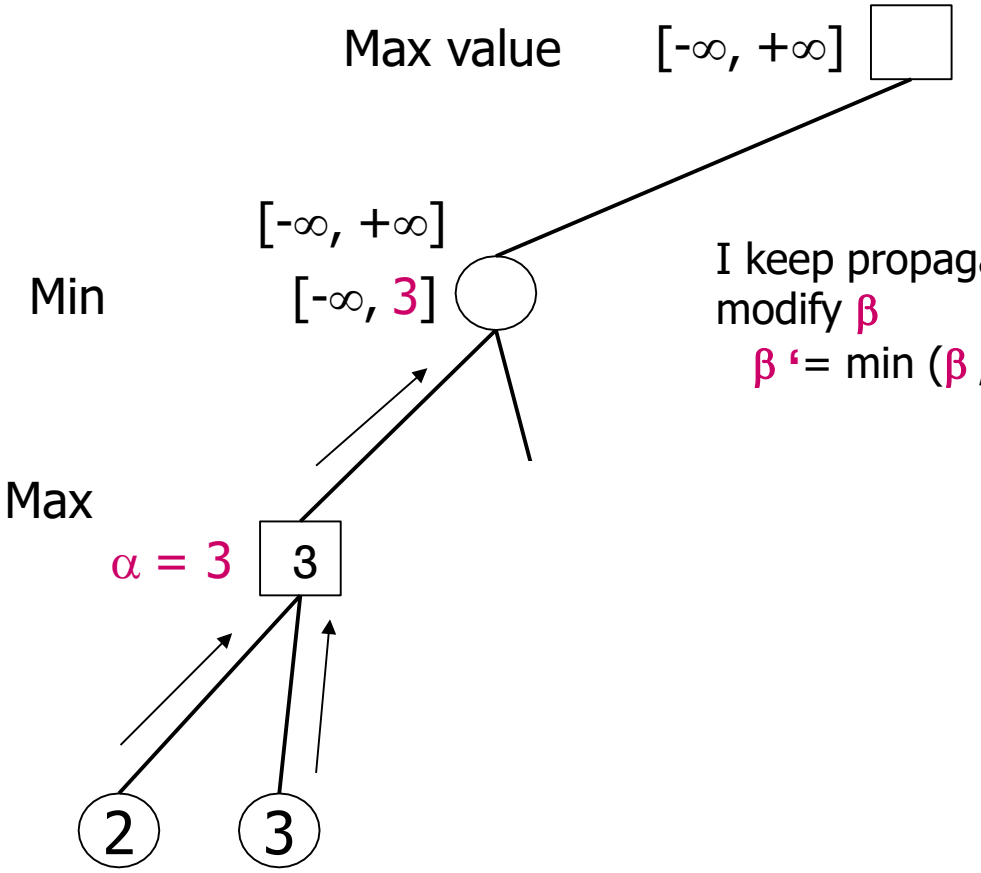
Step by step example



I continue in depth. I come to a value. Upward propagation.
Is a max node => i modify α. α ' = max $(2,3) = 3$.
The response of this MAX node is 3

I come to a value. Propagation upwards. It is node
max => I modify α. α ' = max $(-\infty,2) = 2$.

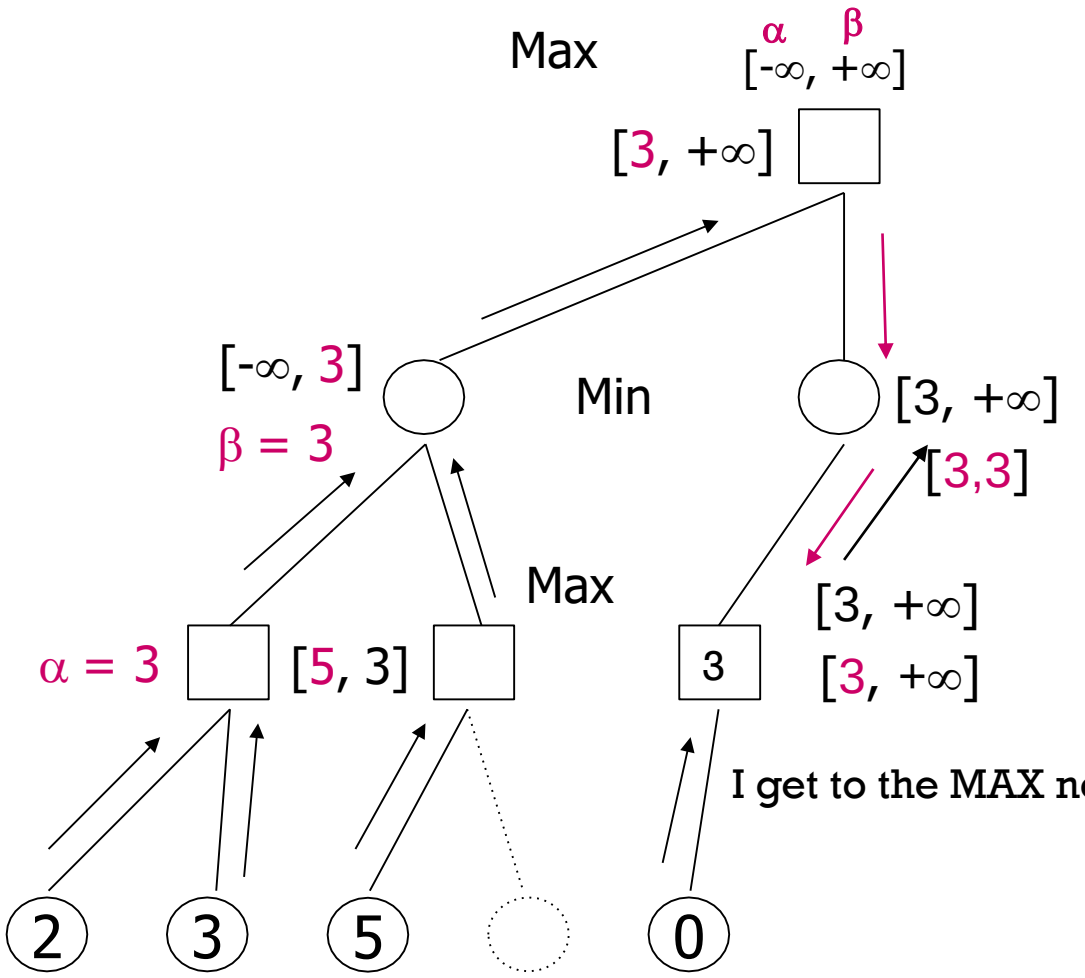
Step by step example



I keep propagating upwards. I get to node min => I modify β

β ‘= min (β ,successor value) = min (+∞, 3)

Step by step example

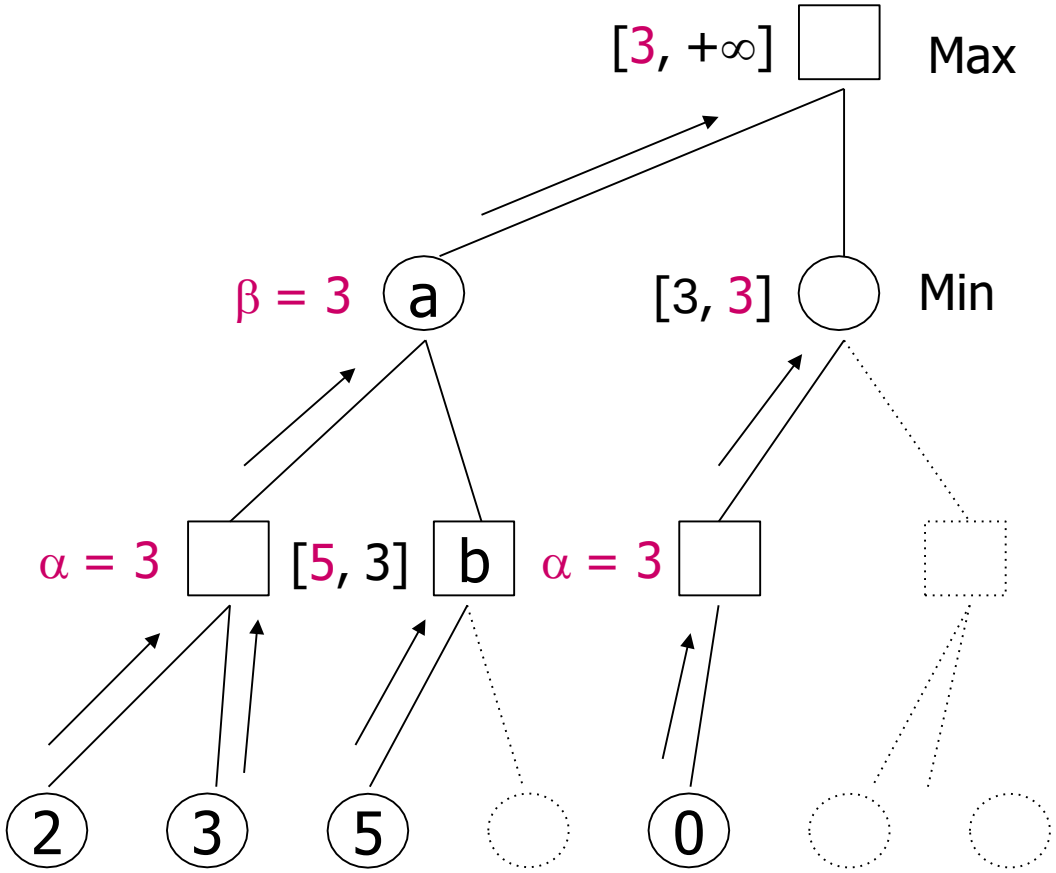


I get to the MAX node. I modify α. α ' = max (-∞,3) =3
I start Downward propagation

I get to node min => I modify β
β '= min (β , successor value)
= min (+∞, 3) = 3

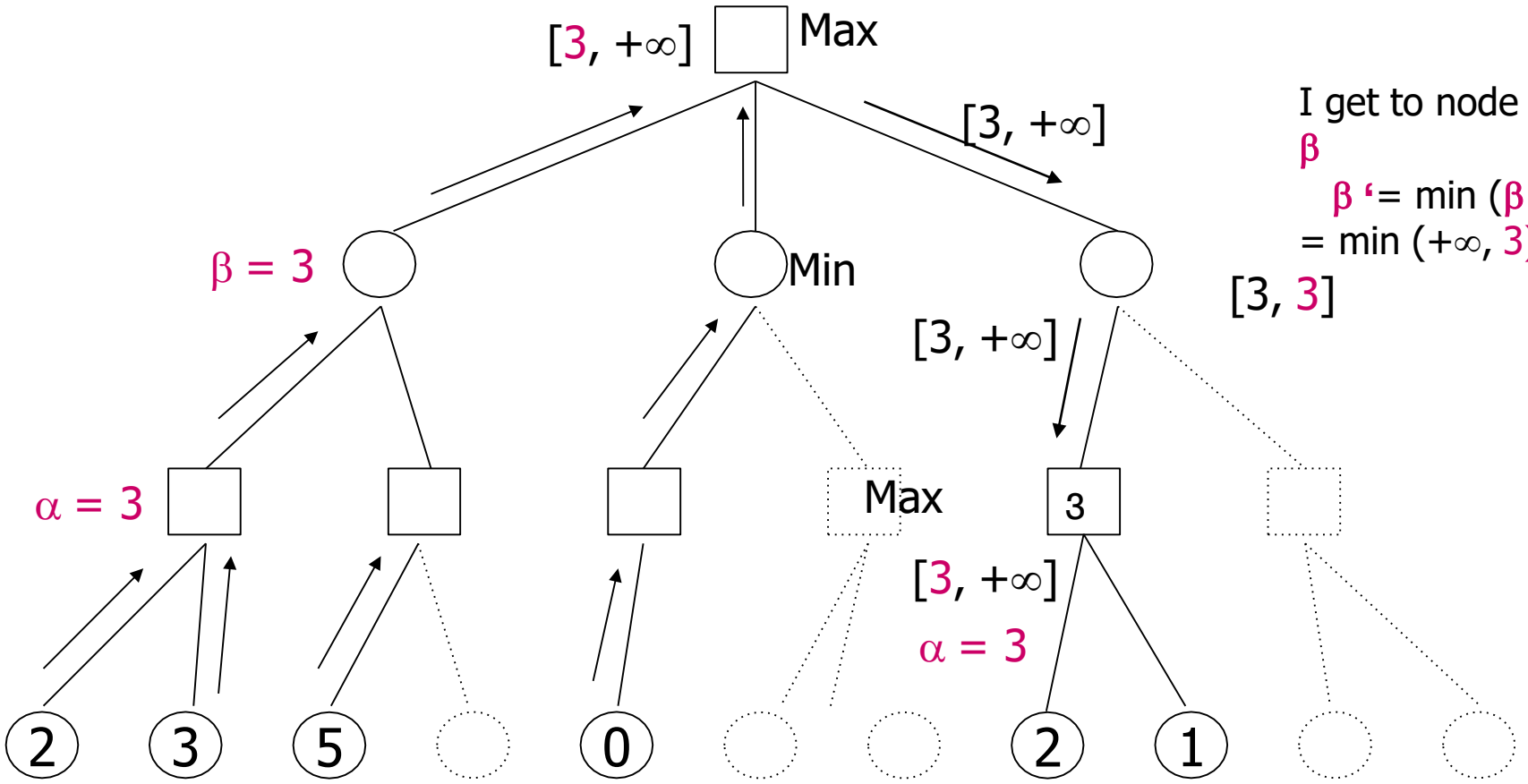
I get to the MAX node. I modify α. α ' = max (3,0) =3

Step by step example



alpha pruning. I am in min node and $\beta <= \alpha$

Step by step example



I get to node min => I modify β

β ‘= min (β ,successor value)

= min (+∞, 3) = 3

[3, 3]

I start Downward propagation I get to the MAX node. I modify α. α ‘ = max (3,2) =3. I continue max (3,1)= 3

alpha pruning. I am in min node and β <= α

Properties of alpha-beta pruning

- This algorithm is of the branching and pruning type:

Guarantees to find the same best movement (or another equivalent with the same value) as minimax, but more efficiently

- If the best limit node is the one generated first, the pruning will be maximum and the minimum number of nodes will have to be generated and evaluated
  - The generation order of the successors greatly influences
  - Assuming a depth search with depth limit l, Using alpha-beta optimal sorting you have to examine only $O(b^{l/2})$ nodes to choose the best move, instead of $O(b^{l})$ with minimax
  - The effective branching factor would be $b^{1/2}$ instead of b
  - With the same computational resources, it allows searches with the same time cost with a depth limit that is twice the corresponding search without pruning (= time)

Games of chance

- Introduce the **random process** as an additional **agent**
  - MAX's turn = RAND's move + MAX's move
  - MIN's turn = RAND's move + MIN's move

  RAND's move = flip a coin, toss a dice, etc.

- The game tree has MAX nodes + MIN nodes + RAND nodes
- Expectiminimax value:

$$\text{expectiminimax(n)} = \begin{cases} \text{utility(n)} & \text{if n is a terminal node} \\ \max\{\text{expectiminimax(s)};\ \text{s} \in \text{successors (n)}\} & \text{if n is a MAX node} \\ \min\{\text{expectiminimax(s)};\ \text{s} \in \text{successors (n)}\} & \text{if n is a MIN node} \\ \sum_{\text{s}\in\text{successors (n)}} p(s)\ \text{expectiminimax(s)} & \text{if n is a RAND node} \end{cases}$$

  - IMPORTANT: For expectiminimax to work, the evaluation function should be a **positive linear transformation** of the expected utility of the state.
  - Temporal complexity is exponential: $O(b^d \cdot n^d)$.
    b = branching factor of MAX / MIN nodes.
    n = branching factor of RAND nodes.
    d = limit of depth-first search.

  - RAND nodes can be pruned.