# VNU-HUS MAT1206E/3508: Introduction to AI

## Search, Games and Problem Solving

### In-class Discussion

Hoàng Anh Đức

Bộ môn Tin học, Khoa Toán-Cơ-Tin học
Đại học KHTN, ĐHQG Hà Nội
hoanganhduc@hus.edu.vn

# Contents

# Additional Materials


Prof. Ertel's Lectures at Ravensburg-Weingarten University in 2011

- https://youtu.be/RRO9-QXROss&t=2210 (Introduction)
- https://youtu.be/rwefoi__Fk4 (Uninformed Search: Breadth-First Search, Depth-First Search, Iterative Deepening)
- https://youtu.be/THZ3YxHAwno (Heuristic Search: Greedy Search, A*-Search, IDA*-Search)
- https://youtu.be/IW-HIOPqgsk (Games with Opponents, Heuristic Evaluation Functions)

# Introduction


- By the 1950s, as computers advanced to the point where experimenting with practical AI algorithms became feasible, *games* emerged as one of the most distinctive AI challenges (aside from tasks like cracking Nazi codes).
- Games offered a *well-defined and constrained domain* that could be *easily formalized and studied*.
- Board games such as *checkers*, *chess*, and more recently the highly complex strategy game *Go* (originating in China over 2500 years ago) have inspired countless researchers and continue to drive advancements in AI.
- Closely tied to games, *search and planning techniques* became a *major area of progress in AI* during the 1960s.
- The roots of *Minimax* go back to 1912; *Alpha-Beta Pruning* was conceived in 1956 and published in 1961 [Russell and Norvig 2021], p. 175. Both remain *foundational for game-playing AI*.
- We will explore the fundamental concepts of *search*, *games*, and *problem-solving*.

# Introduction

## Search Problems

A *search problem* is defined by the following values

**State:** Description of the world; denoted by $s$.

**Starting state:** The initial state.

**Goal state:** A state satisfying the goal condition.

**Actions:** The agent's allowed choices.

**Solution:** Search-tree path from the starting state to a goal state.

**Cost function:** Costs of action-induced transitions; needed for cost-optimal solutions.

**State-space graph:** States linked by action-induced transitions.

**Search tree:** Generated from the start; node $n$ stores $\mathrm{state}(n)$ and path data, so a state may reappear.


## Example 1 (8-Puzzle)

Apply the definition to the 8-puzzle, we get

**State:** $3 \times 3$ matrix $S$ with the values $1, 2, 3, 4, 5, 6, 7, 8$ (once each) and one empty square.

**Starting state:** An arbitrary state.

**Goal state:** An arbitrary state.

**Actions:** Movement of the empty square $S_{ij}$ to the left (if $j \neq 1$), right (if $j \neq 3$), up (if $i \neq 1$), down (if $i \neq 3$).

**Solution:** The path in the search tree from the starting state to the goal state.

**Cost function:** The constant function $1$, since all actions have equal cost.

**State space:** The state space is not fully connected: it splits into mutually unreachable components. Therefore, some arbitrary 8-puzzle start/goal pairs are unsolvable.

## Example 2 (8-Puzzle, cont.)

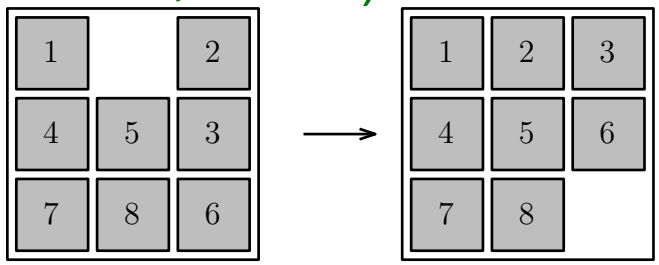


Figure: Possible 8-puzzle start and goal states [Ertel 2025], Fig. 6.2, p. 95

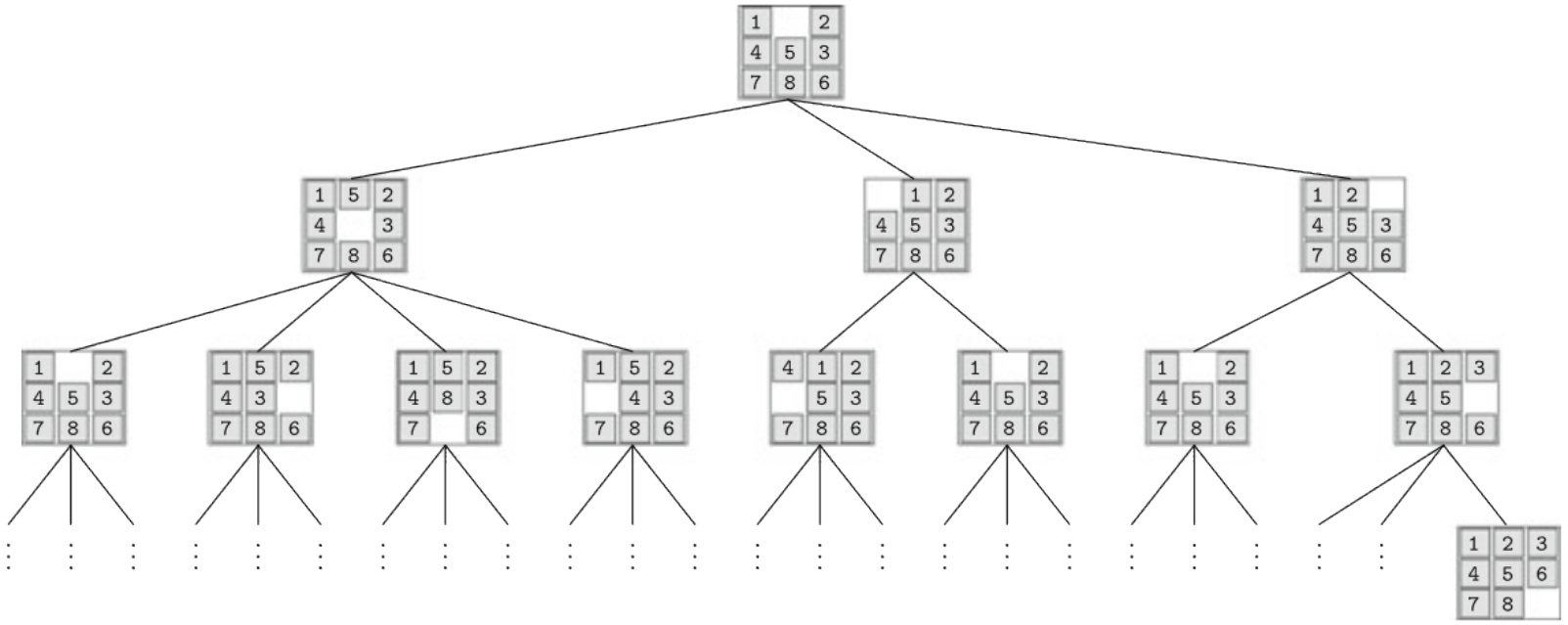


Figure: 8-puzzle search tree; other depth-3 nodes are omitted [Ertel 2025], Fig. 6.3, p. 96




**Branching factor**

The *number of successor states of a state* $s$ is called *the branching factor* $b(s)$, or $b$ if *the branching factor is constant*.

## Quiz

After applying all legal actions to a state $s$, we get exactly three successor states. What is $b(s)$?

(a) $0$

(b) $1$

(c) $3$




### Effective branching factor

For a tree of *depth* $d$ with $N$ *total nodes*, the *effective branching factor* is the *constant branching factor* that would give the same $d$ and $N$.

We count the *root/start node* as depth $0$.

## Quiz

A tree has depth $3$ and $40$ total nodes including the root. Which equation defines its effective branching factor $\bar{b}$?

(a) $\bar{b} = 40/3$

(b) $1 + \bar{b} + \bar{b}^2 + \bar{b}^3 = 40$

(c) $\bar{b} = 40$




## Complete Search Algorithms

A search algorithm is *complete* if it *finds a solution whenever one exists*. If complete search exhausts the space with no solution, then *the problem is unsolvable*.

## Quiz

An algorithm always selects the first child of the current node. In one solvable search tree, the resulting path is infinite and contains no goal, while another path contains a goal. Is the algorithm complete?

(a) Yes: a goal exists.

(b) No: it may never reach the goal.

(c) Yes: it never says “no solution”.

# Introduction



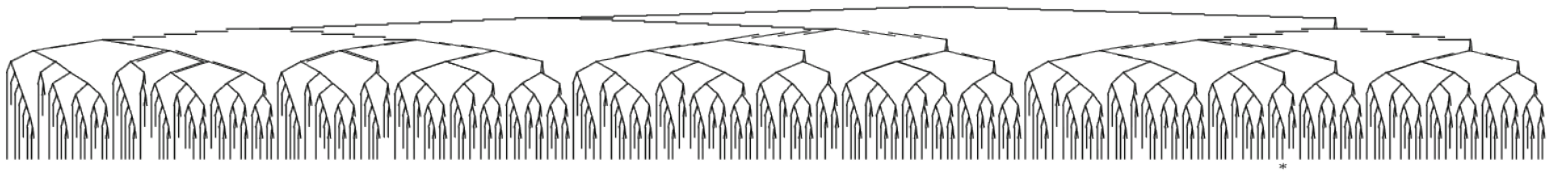



- The *search* for a solution in an *extremely large search tree* presents a problem for *nearly all inference systems*.
  - From the *starting state* there are *many possibilities for the first inference step*.
  - For each of these possibilities there are again *many possibilities in the next step*, and so on.
- Some reference points for *"large" numbers*:
  - The *age of the universe* is about $1.38 \times 10^{10}$ years [Planck Collaboration 2020].
  - A *reference adult body* contains on the order of $10^{13}$ human cells; the estimate depends on the body model and counting method [Hatton et al. 2023].
  - The *human brain* is commonly estimated to contain on the order of $10^{14}$ synapses [Institute of Medicine 1991].



## Example 3

The search tree for SLD resolution proof of *a very simple formula* from [Ertel 1993], as discussed in [Ertel 2025], Sec. 6.1, with *three Horn clauses*, *each with at most three literals*:



- The tree was cut off at a depth of 14 and has a solution in the leaf node marked by $\star$.
- It is only *possible to represent* it at all because of the *small branching factor (= number of children at each node) of at most two* and *a cutoff at depth 14*.






## Quiz

The SLD search tree is cut off at depth $14$, the root is at depth $0$, and every node has at most $2$ children. In the largest possible such tree, how many nodes are at depth $14$?

(a) $14$

(b) $2^{14} = 16{,}384$

(c) $2^{15} - 1$






## Quiz

The displayed SLD tree is cut off at depth $14$, the root is at depth $0$, and every node has at most $2$ children. Why is $2^{14}$ an upper bound, not an exact count, for the nodes at depth $14$?

(a) The branching factor is at most $2$, not exactly $2$ everywhere.

(b) Depth $14$ means there are only $14$ nodes.

(c) A solution leaf removes all other branches.






## Quiz

Following [Ertel 2025], Sec. 6.1, assume a full search tree with constant branching factor $b = 30$ is generated up to depth $50$, with the root at depth $0$. How many nodes are at depth $50$?

(a) $30 \cdot 50$

(b) $30^{50} \approx 7.2 \times 10^{73}$

(c) $50^{30}$






## Quiz

Assume a full search tree with constant branching factor $b = 30$ is generated up to depth $50$, with the root at depth $0$. How many total nodes are generated through depth $50$, including the root?

(a) $30^{50}$

(b) $\sum_{i=0}^{50} 30^i = \frac{30^{51} - 1}{29} \approx 7.4 \times 10^{73}$

(c) $50 \cdot 30^{50}$




## Quiz

Assume each generated search-tree node counts as one inference step. With $10{,}000$ computers, each generating $10^9$ nodes per second, how long would about $7.4 \times 10^{73}$ generated nodes take?

(a) $7.4 \times 10^{60}$ seconds, about $2.3 \times 10^{53}$ years

(b) $7.4 \times 10^{64}$ seconds

(c) About $50$ years






## Quiz

A full search tree with branching factor $30$ and depth $50$ has about $7.2 \times 10^{73}$ nodes at depth $50$ and about $7.4 \times 10^{73}$ total generated nodes. What does this illustrate?

(a) For a large constant branching factor, almost all nodes are on the last level.

(b) Inner nodes dominate the computation.

(c) Complete brute-force search is practical for chess.




**Main lesson**

If every step creates *many new choices*, the *search tree* becomes *huge very quickly*. So we usually *cannot just try everything*. The rest of the lecture is about how to *search more intelligently*: which nodes to try first, what to remember, and which parts of the tree to avoid.

# Introduction


**Questions**

- Why do *good chess players* exist – and nowadays also *good chess computers*?
- Why do mathematicians find proofs for propositions in which the *search space is even larger*?

- Evidently we humans use *intelligent strategies* which *dramatically reduce the search space*.
  - The experienced chess player, just like the experienced mathematician, will, by mere observation of the situation, immediately rule out many actions as senseless. Through his *experience*, he has the *ability to evaluate various actions for their utility in reaching the goal*.
  - Often a person will go by *feel*. If one asks a mathematician how he found a proof, he may answer that the intuition came to him in a dream.
- In everyday problems, *intuition* plays a big role. We will later deal with *heuristic search methods* and with processes by which computers can, similarly to humans, *improve their heuristic search strategies by learning*.

# Introduction

## Example 4 (Shortest Path from City $A$ to City $B$)

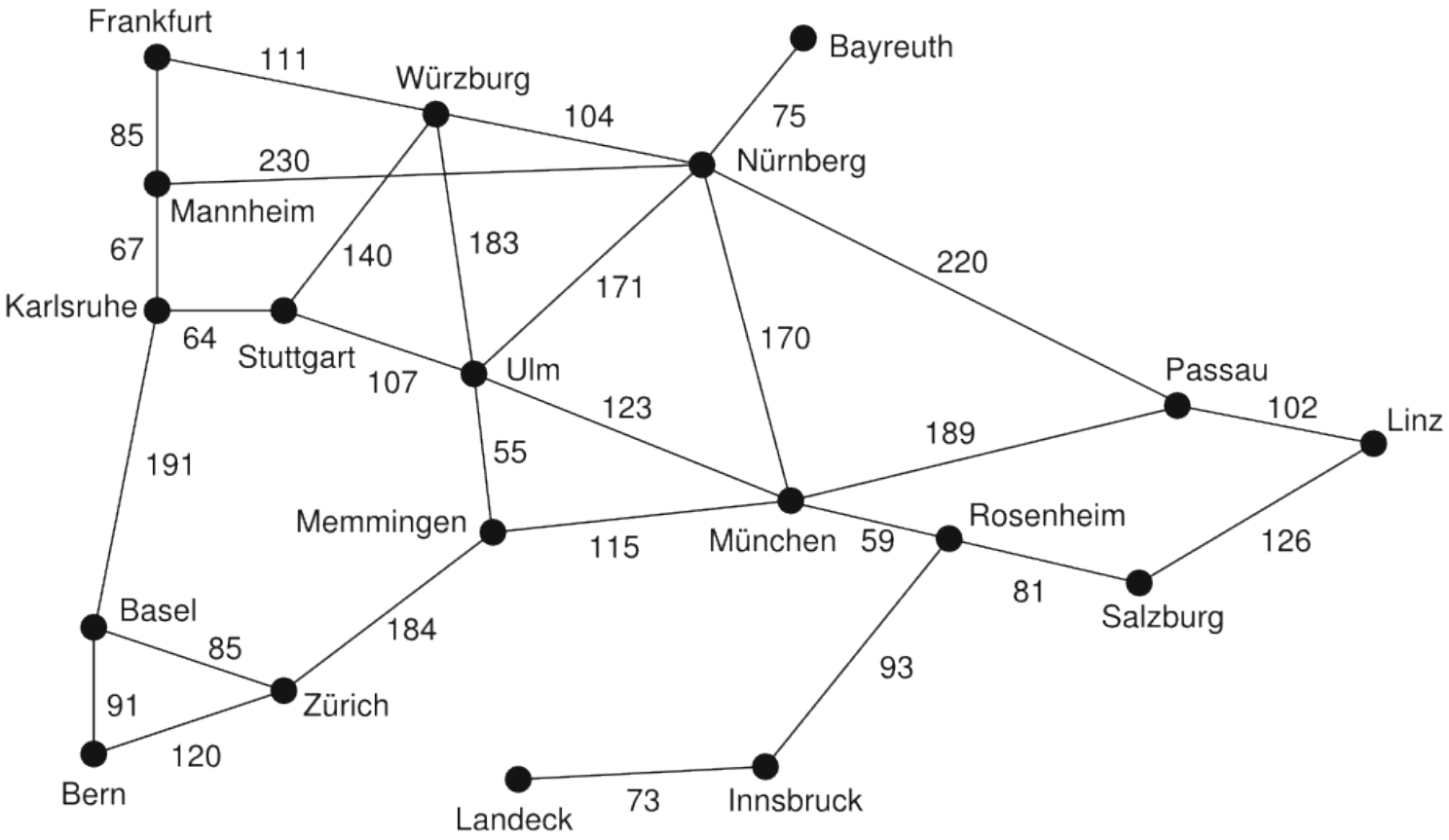



Figure: The graph of southern Germany as an example of a search task with a cost function [Ertel 2025], Fig. 6.5, p. 99.


## Example 4 (cont.)

**State:** A city as the current location of the traveler.

**Starting state:** An arbitrary city $A$.

**Goal state:** An arbitrary city $B$.

**Actions:** Travel from the current city to a neighboring city.

**Cost function:** The distance between the cities. Each action corresponds to an edge in the graph with the distance as the weight.

**State space:** All cities, represented as vertices of the state-space graph.

**Optimal Search Algorithms**

A search algorithm is called *optimal* if it *always finds the solution with the lowest cost*, provided that at least one solution exists.

# Introduction




**Deterministic Problem**

Every action leads from a state to a unique successor state.

**Observable Problem**

The agent always knows which state it is in.

## Example 5

- The 8-puzzle problem is deterministic and observable.
- In route planning in real applications, one or both characteristics may be missing.
  - The action “Drive from Munich to Ulm” may—for example because of an accident—lead to the successor state “Munich”.
  - It can also occur that the traveler no longer knows where he is because he got lost.

# Introduction






- For the search algorithms considered next, we assume a *single-agent problem* that is *deterministic and observable*, and whose *action outcomes are known in advance* [Russell and Norvig 2021], Sec. 2.3.2.
- For the 8-puzzle, this known abstract model lets us *find an action sequence before carrying out the moves*. $\Rightarrow$ Offline algorithms.
- Robot soccer is different: an *exact abstract model of the actions* is not available, and the robot must *choose actions from sensor signals while acting*. $\Rightarrow$ Online algorithms. (*Reinforcement learning* works toward optimization of these decisions based on experience.)

# Uninformed Search
## Introduction


- *Uninformed search* uses only the information given in the search problem. It has *no additional guidance* telling it which current state appears closer to a goal.
- This *does not mean random search*: each method follows a general rule for choosing which node to explore next.
- We study *breadth-first search* and its uniform-cost variant, then *depth-first search*, its depth-limited variant, and *iterative deepening*.
- We compare the principal methods by completeness, cost optimality, time, and memory, then examine cycles along the current path.
- *Heuristic search* later adds *guidance* telling it which current state appears closer to a goal.

# Uninformed Search

## Example Problem: Simplified Sliding-Block Puzzle


Toy marker-grid version: one marker moves; the blocked cell cannot be entered.

**Initial position**

| | | |
|---|---|---|
| A | # | B |
| C | D | E |
| F | G | H |

↓ move down

**Next position**

| | | |
|---|---|---|
| A | # | B |
| C | D | E |
| F | G | H |

**State** Marker location in one cell $A, \ldots, H$.

**Starting state** $B$.

**Goal state** Marker in $G$.

**Actions** Move the marker one cell left, right, up, or down when the destination is inside the grid and not blocked.

**Solution** A legal sequence of moves from $B$ to $G$.

**Cost function** Cost $1$ for each legal move.

**State-space graph** Vertices $A, \ldots, H$ linked by legal moves.

**Search tree** Root $B$; each child represents one legal move from its parent.

**Note.** A state is the whole board, but only the marker's cell changes.

That cell therefore identifies the state and labels each search-tree node.

# Uninformed Search
## Breadth-First Search





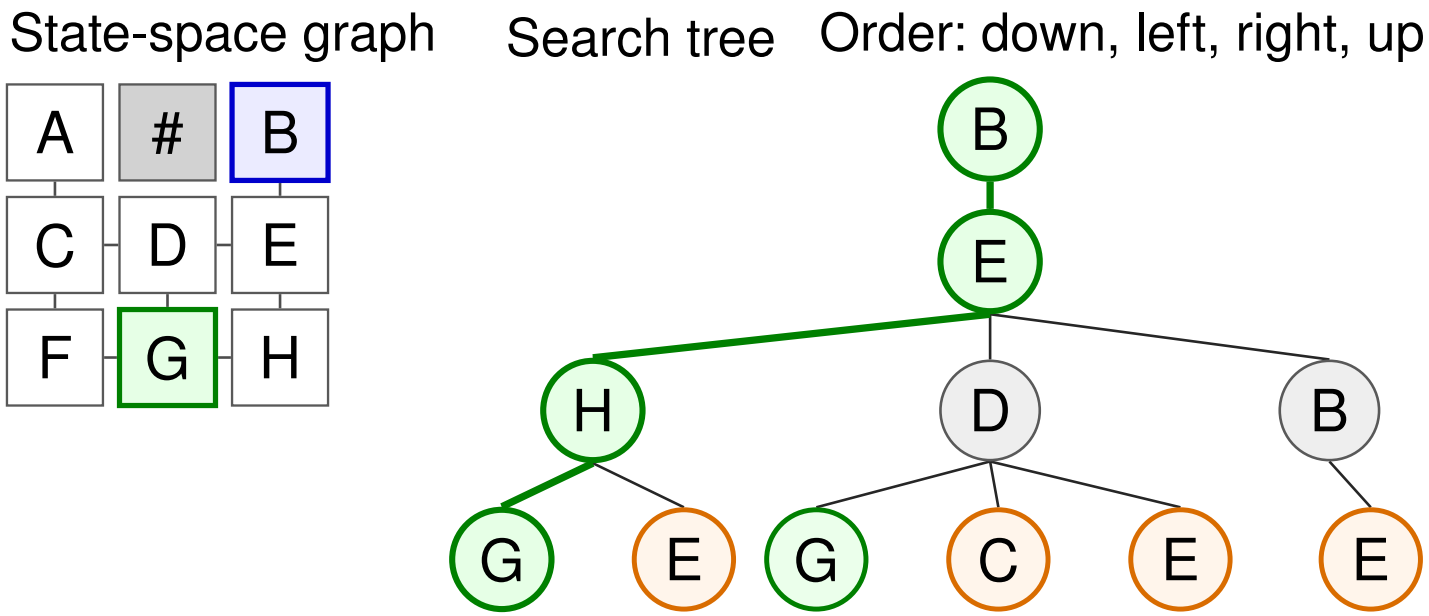

**Core Idea**

BFS explores the search tree *level by level*: it tests every node at depth $i$ before any node at depth $i + 1$.

- BREADTHFIRSTSEARCH(NodeList, Goal) receives the current level in left-to-right order and starts with NewNodes = $\emptyset$.
- For each Node in NodeList:
  - if GOALREACHED(Node, Goal) finds that $\mathrm{state}$(Node) satisfies the goal, return the solution;
  - otherwise append the child search nodes returned by SUCCESSORS(Node) to NewNodes.
- After the current level is processed, call BREADTHFIRSTSEARCH(NewNodes, Goal). Thus generated successors are tested only in the next invocation.
- If NewNodes = $\emptyset$, return “No solution.”

# Uninformed Search

## Simplified Sliding-Block Puzzle Solved by BFS




| NodeList | G | E | G | C | E | E |
|---|---|---|---|---|---|---|

| NewNodes | goal found; NewNodes is not used |
|---|---|

First invocation: BREADTHFIRSTSEARCH($[B], G$). Reset NewNodes to $\emptyset$; test $B.B$ is not $G$. Append SUCCESSORS$(B) = [E]$ to NewNodes.Second invocation: BREADTHFIRSTSEARCH($[E], G$). Reset NewNodes to $\emptyset$; test $E.E$ is not $G$. Append SUCCESSORS$(E) = [H, D, B]$; the next invocation processes

# Uninformed Search
## Breadth-First Search


**Characteristics:** Let $b$ be the branching factor and $d$ the shallowest goal depth.

- *Complete* if *each node has only finitely many successors*.
- *Optimality is guaranteed* only when *all step costs are equal*.
- *Worst-case time:* $O(b^{d+1})$ for the displayed late-test algorithm. It goal-tests $O(b^d)$ nodes but may generate $O(b^{d+1})$ successors before testing the goal.
- *Worst-case space:* $O(b^{d+1})$, because the generated next-level list may be stored.

When *step costs are not all equal*, we use a variant of BFS.

**Uniform-Cost Search**

Select an open node with the *smallest path cost from the start*, and test for a goal *when that node is selected*. With finite branching and every action cost at least *some fixed positive amount*, UCS is complete and returns a lowest-cost solution.

# Uninformed Search
## Depth-First Search



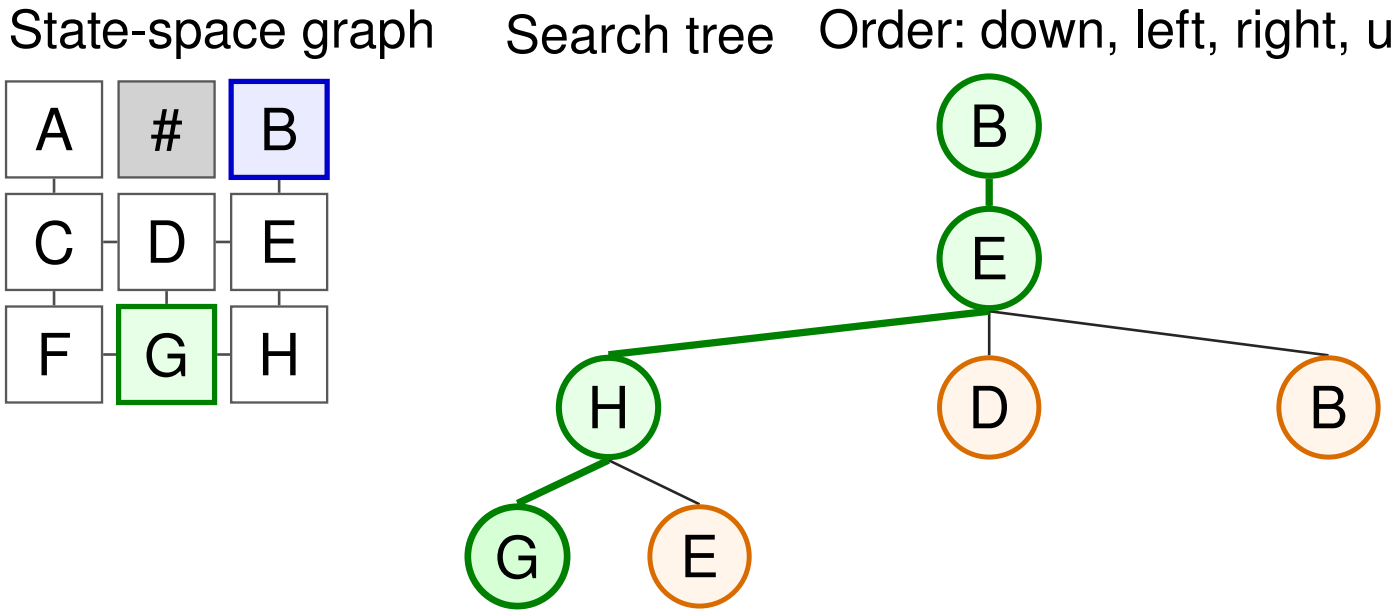



**Core Idea**

DFS follows *one path as deeply as possible*. If that path has no goal, it *backtracks to the most recent node with an untried successor*.

- DEPTHFIRSTSEARCH(Node, Goal) first uses GOALREACHED(Node, Goal).
- If Node is not the goal, NewNodes receives SUCCESSORS(Node).
- Recursively search FIRST(NewNodes). If it succeeds, return the solution; otherwise replace NewNodes by REST(NewNodes) and try the next successor.
- If every successor fails, return “No solution.”

# Uninformed Search
## Simplified Sliding-Block Puzzle Solved by DFS




Current path B E H G

SUCCESSORS GOALREACHED$(G, G)$ succeeds

The call DEPTHFIRSTSEARCH(B, G) sets NewNodes to SUCCESSORS$(B) = [E]$, then recursively calls DEPTHFIRSTSEARCH(E, G).The call DEPTHFIRSTSEARCH(E, G) sets NewNodes to SUCCESSORS$(E) = [H, D, B]$. It recursively calls DEPTHFIRSTSEARCH(H, G); $D$ and $B$ remain untried.The call

# Uninformed Search
## Depth-First Search


**Characteristics:**

- *Incomplete: May follow an infinite branch forever*.
  - For this counterexample, change the move order to left, right, up, down. Without a repeated-state check, DFS follows $B \to E \to D \to C \to D \to C \to \cdots$.
  - Yet $B \to E \to D \to G$ is a solution.
- *Non-optimal:* The first solution found *may be deeper or more costly than another solution*.
- *Time Complexity:* $O(b^{d_s})$ for a finite search tree, where $b$ is the branching factor and $d_s$ is the tree's maximum depth.
- *Space Complexity:* $O(bd_s)$; stores only the current path and its successors.

**Depth-Limited Search**

A variant of DFS with a depth limit ($\Rightarrow$ *Pruned search tree*). It avoids infinite loops but may miss solutions beyond the limit.

# Uninformed Search
## Iterative Deepening

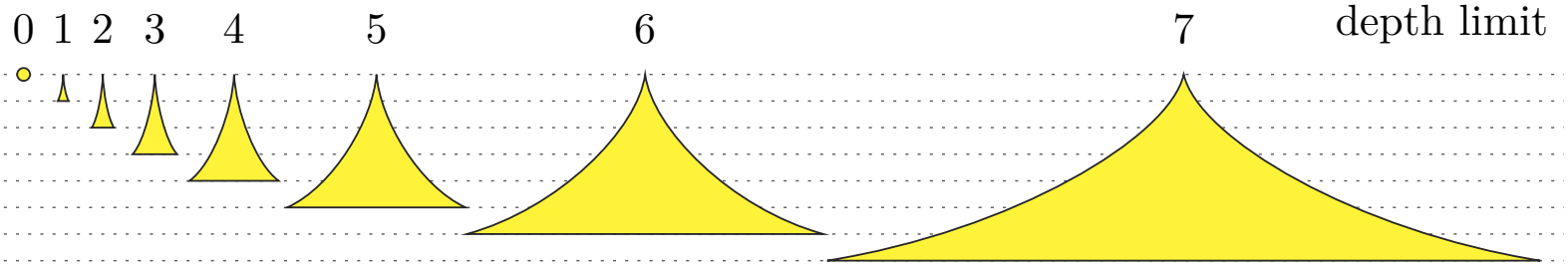





**Core Idea**

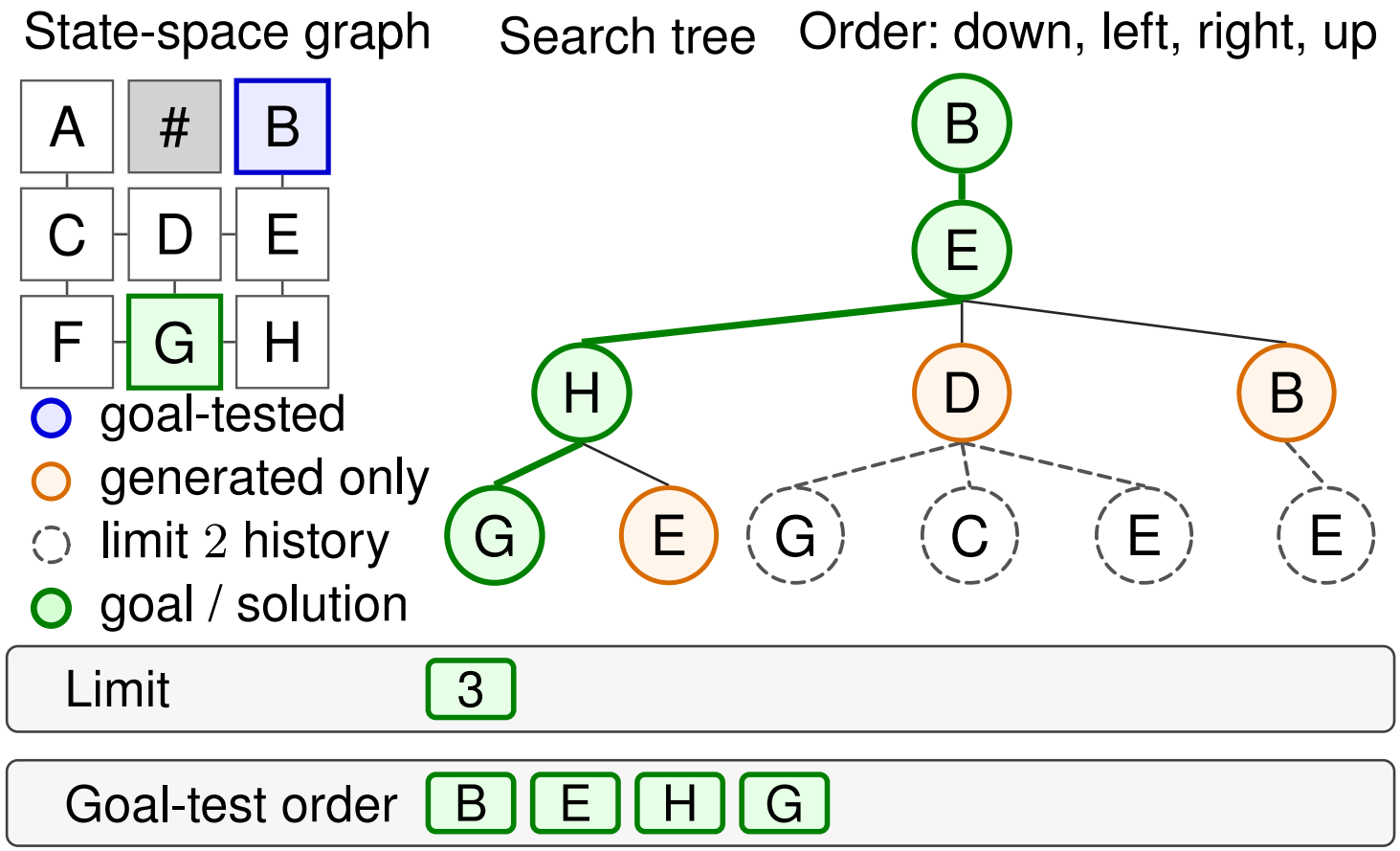
*Iterative deepening* repeats *depth-limited DFS* with *Limit* $= 0, 1, 2, \ldots$. Each iteration *restarts at the starting node* and *stops if it finds a goal*.

- *Depth* is the current node's depth, with the root at depth $0$; *Limit* is the greatest depth searched in the current iteration.
- A reached node is *goal-tested*. If *Depth* $<$ *Limit*, its successors are *searched recursively in DFS order*.
- If *Depth* $=$ *Limit*, successors may be generated, but they are *not recursively visited or goal-tested in that iteration*.



# Uninformed Search

## Simplified Sliding-Block Puzzle Solved by Iterative Deepening




The successful recursion path is $B \to E \to H \to G$. Three unit-cost moves give cost $3$. Limits $0$–$2$ goal-tested every node through depth $2$, so no shallower solution exists.

# Uninformed Search
## Iterative Deepening






**Analysis:** Let $b$ be the branching factor and $d$ the shallowest goal depth.

- *Complete.*
- *Optimal* if all action costs are equal and the limit increases by $1$.
- *Worst-case time:* $O(b^{d+1})$ for the displayed procedure. It goal-tests $O(b^d)$ nodes but may generate $O(b^{d+1})$ successors because it still calls SUCCESSORS at the limit.
- *Memory requirement:* $O(bd)$; the generated fringe is not stored all at once.
- Restarting repeats work, but for large $b$ the *last iteration dominates the geometric sum*. **[Why?]**

# Uninformed Search
## Comparison


| | Breadth-first search | Uniform-cost search (‡) | Depth-first search | Iterative deepening |
|---|---|---|---|---|
| Completeness | Yes | Yes | No | Yes |
| Optimal solution | Yes (*) | Yes | No | Yes (*) |
| Computation time | $b^{d+1}$ (†) | $b^{1+\lfloor C^*/\epsilon \rfloor}$ | $\infty$ or $b^{d_s}$ | $b^{d+1}$ (†) |
| Memory use | $b^{d+1}$ | $b^{1+\lfloor C^*/\epsilon \rfloor}$ | $bd_s$ | $bd$ |

- (*): only with constant action cost. (‡): for UCS, finite branching and costs of at least a fixed $\epsilon > 0$; goal-test on selection.
- (†): these late-test procedures test $O(b^d)$ nodes but can generate $O(b^{d+1})$ successors.
- $C^*$: lowest solution cost; $d$: shallowest goal depth; $d_s$: maximal finite-tree depth.

# Uninformed Search
## Cycle Check: What Is Being Checked?


- A *cycle* occurs when the *current path reaches a state that already appears on that same path*.
  - In the 8-puzzle, one move can often be immediately undone, creating a cycle of length $2$.
- A *full cycle check* is a *path check*: when a successor is generated, compare its state with *all predecessor states on the current path*.
- This is *not* a global "already reached" table: it prevents returning to an earlier state on the same path, but it does not remove every duplicate reached by a different path.




## Quiz

For a full cycle check, each search-tree node stores all predecessor states on its current path. When a successor is newly generated at depth $i$, which states should it be compared with?

(a) All nodes generated earlier by the whole search.

(b) The $i$ predecessor states on its current path.

(c) Only the root node.




## Quiz

Assume a full search tree with constant branching factor $b$. At depth $i$, there are $b^i$ generated nodes. For a full cycle check, each generated node at depth $i$ is compared only with the $i$ predecessor states on its current path. How many predecessor-state comparisons are made for all nodes at depth $i$?

(a) $i$

(b) $b^i$

(c) $i \cdot b^i$




## Quiz

Assume a full search tree with constant branching factor $b$ is generated through depth $d$. Generating one node costs $c_1$. In the full cycle check, each node at depth $i$ is compared with its $i$ path predecessors, and one such comparison costs $c_2$. Which expression counts the generation cost plus the full cycle-check comparison cost?

(a) $c_1 \sum_{i=0}^{d} b^i$

(b) $c_2 \sum_{i=0}^{d} i \cdot b^i$

(c) $c_1 \sum_{i=0}^{d} b^i + c_2 \sum_{i=0}^{d} i \cdot b^i$




## Quiz

Assume a full search tree with constant branching factor $b$ is generated through depth $d$. Without the full cycle check, the generation work is about $\sum_{i=0}^{d} b^i$, dominated by $b^d$ for large $b$ and $d$. With the full cycle check, depth $i$ adds $i \cdot b^i$ predecessor comparisons. What is the main effect on the time complexity?

(a) The extra work is constant.

(b) The extra work is about a factor $d$, because $\sum_{i=0}^{d} i \cdot b^i$ is dominated by $d \cdot b^d$.

(c) The check compares each node with all earlier generated nodes.

# Heuristic Search
## Introduction


- *Heuristics* can find a solution *faster than uninformed search*, but offer *no guarantee*.
- They are useful for *real-time decisions under limited resources*.
- A *good solution found quickly* may be preferred to an *optimal* solution that is very expensive to derive.
- We introduce *greedy search*, *A* search*, and *IDA* search*.

**Mathematical Modeling:**

- A search node $n$ stores $\mathrm{state}(n)$, path information, depth, and an *evaluation* $f(n)$ that determines the search order.
- **Goal:** Find, with *little effort*, a *solution* to the stated search problem with *minimal total cost*.

**Notation**

Ertel sometimes uses $s$ for both states and search nodes. Here $s$ always denotes a state and $n$ a search-tree node [Ertel 2025], pp. 106, 109.

# Heuristic Search
## Core Idea


### Core Idea

Heuristic search keeps NodeList, a list of *generated but not yet expanded nodes*, ordered by *increasing evaluation* $f(n)$.

- *Initialize:* set NodeList to $[n_0]$, the root node for the start.
- *Select:* remove the first node from NodeList; it has the *smallest evaluation*.
- *Test:* if this node is the goal, *return the solution*.
- *Expand:* otherwise generate its successors and *insert them into NodeList in increasing evaluation order*.
- *Failure:* if NodeList becomes empty, *no solution was found*.

### Why the Evaluation Function Matters

The *definition of $f$ determines the search order*; greedy search and A* use different evaluation functions.

# Heuristic Search




Ideally, the *best heuristic* would calculate the actual remaining cost from $\mathrm{state}(n)$ to a goal.

**Question**

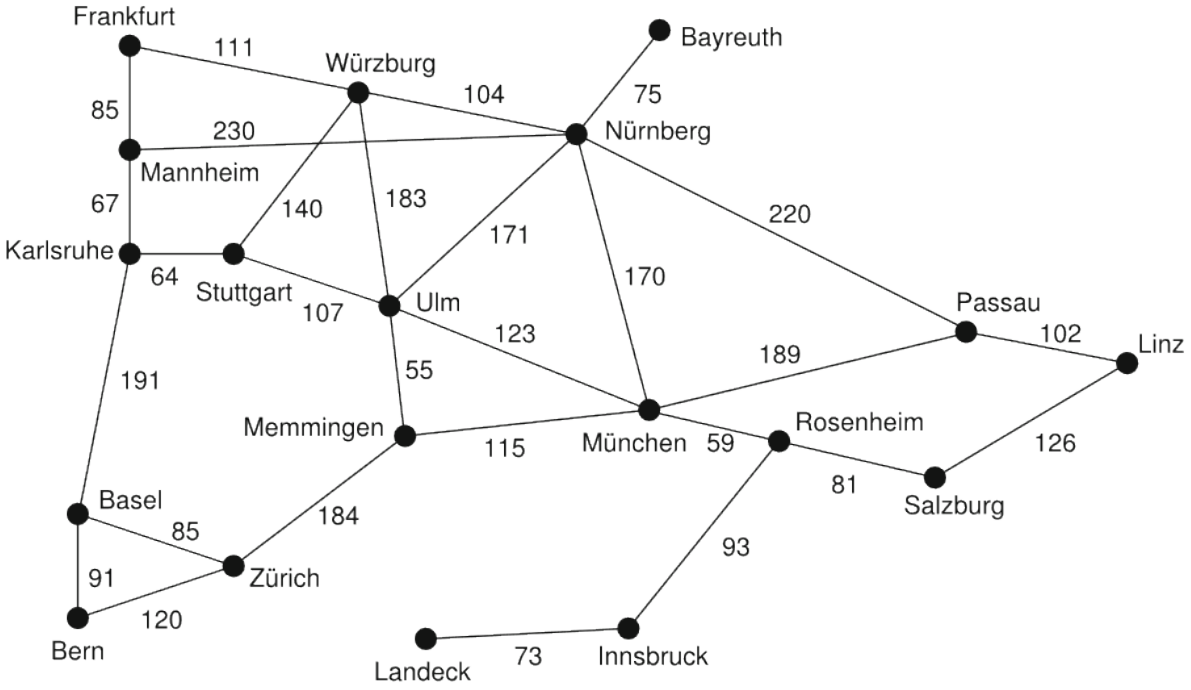
*How do we find a heuristic that is fast and simple to compute?*

An idea for finding a heuristic *simplification of the problem*.

- The *original task* is *simplified enough* that *it can be solved with little computational cost*.
- The *costs from a state to the goal in the simplified problem* then *serve as an estimate for the actual problem*.
  $\Rightarrow$ *cost estimate function* $h(n)$, computed from $\mathrm{state}(n)$.

## Example 6 (Shortest Path from City $A$ to City $B$)

- **Simplified Task:** finding the straight line path from city to city (that is, the flying distance).
- *Cost estimate function* $h(n)$ = flying distance from the city stored in node $n$ to Ulm (given in the figure below).




| | |
|---|---|
| Basel | 204 |
| Bayreuth | 207 |
| Bern | 247 |
| Frankfurt | 215 |
| Innsbruck | 163 |
| Karlsruhe | 137 |
| Landeck | 143 |
| Linz | 318 |
| München | 120 |
| Mannheim | 164 |
| Memmingen | 47 |
| Nürnberg | 132 |
| Passau | 257 |
| Rosenheim | 168 |
| Stuttgart | 75 |
| Salzburg | 236 |
| Würzburg | 153 |
| Zürich | 157 |

Figure: Flying distances to Ulm [Ertel 2025], Fig. 6.14, p. 108

## Example 6 (cont.)

Greedy search uses the cost estimate function directly: $f(n) = h(n)$.

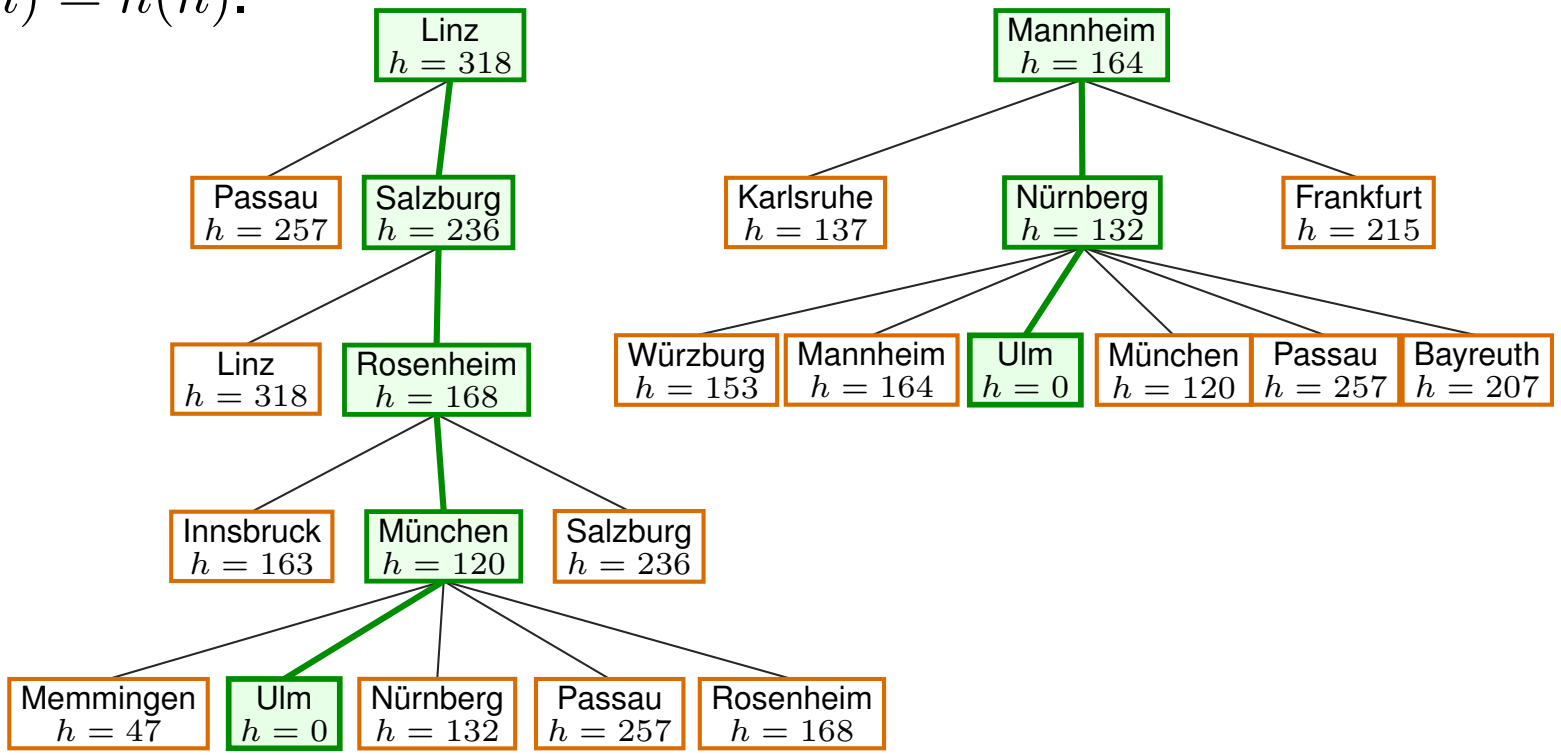



Figure: Start from Linz. Greedy search uses $f(n) = h(n)$ and keeps NODELIST sorted by $h$.Expand Linz. NODELIST: Salzburg(236), Passau(257); select Salzburg.Expand Salzburg. NODELIST: Rosenheim(168), Passau(257), Linz(318); select Rosenheim.Expand Rosenheim. Its successors include München(120), Innsbruck(163),

# Heuristic Search
## A*-Search


- *Path-cost function* $g(n)$ = Cost of the path from the root to node $n$.
- *Heuristic cost estimate* $h(n)$ = Estimated cost from $\mathrm{state}(n)$ to a goal.
- *Heuristic evaluation function* $f(n) = g(n) + h(n)$.

**Admissible heuristic cost estimate function**

A heuristic $h(n)$ is *admissible* if it is nonnegative and *never overestimates the lowest actual cost* from $\mathrm{state}(n)$ to a goal. Nodes with the same state share $h$; their $g$ and $f$ values may differ.

**A*-algorithm**

= HEURISTICSEARCH + $f(n) = g(n) + h(n)$ + admissible $h$

## Example 6 (cont.)

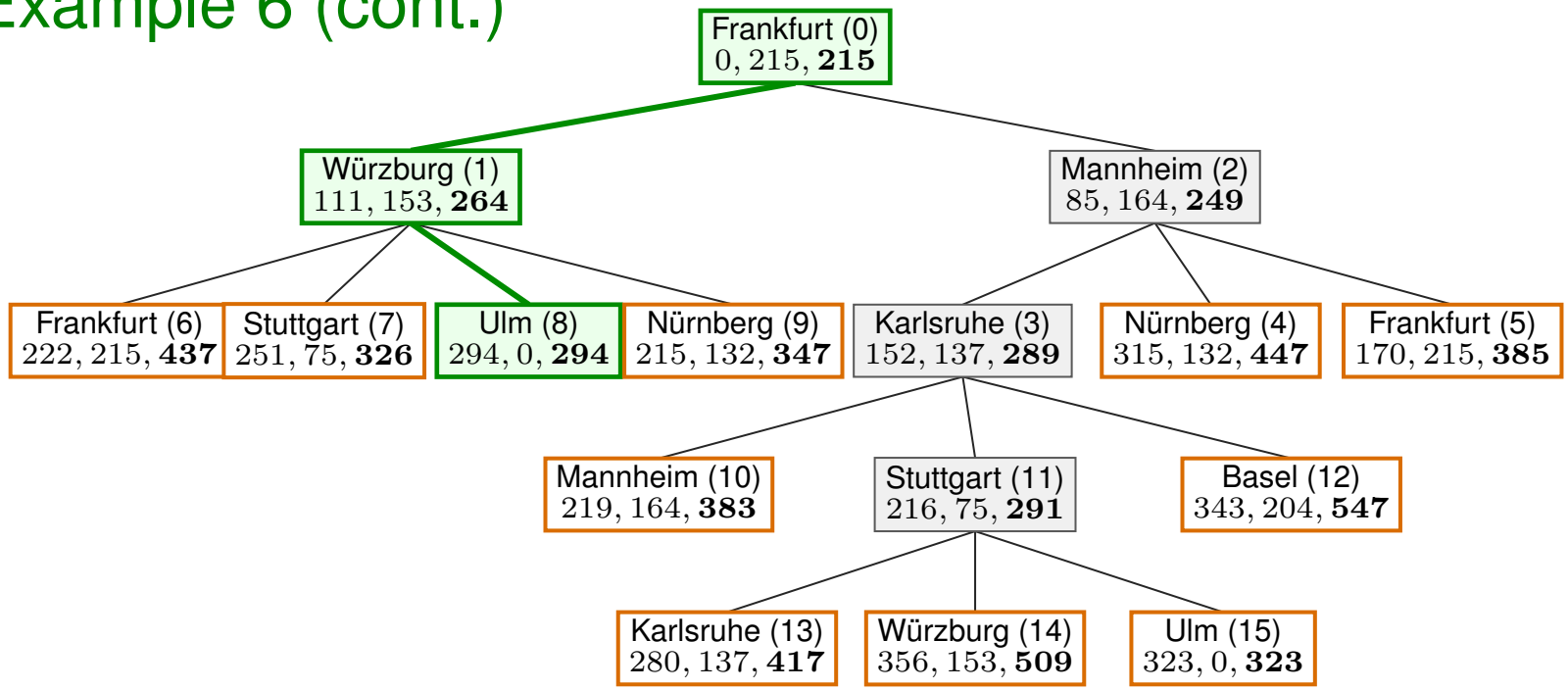



Figure: Start at Frankfurt. Each node shows city (generation number); below it are $g, h, \mathbf{f}$.Expand Frankfurt. NODELIST: Mannheim(249), Würzburg(264); select Mannheim.Expand Mannheim. NODELIST: Würzburg(264), Karlsruhe(289), Frankfurt(385), Nürnberg(447); select Würzburg.Expand Würzburg. Ulm(8) is a goal, but A* does not stop: Karlsruhe(3) has lower $f = 289$ than Ulm's $f = 294$.Expand Karlsruhe. Stuttgart(11) has $f = 291 < 294$, so it is selected before






The following restates Proposition 3.1 of Poole and Mackworth in the notation and terminology of these slides [Poole and Mackworth 2023], Sec. 3.6.1.

### Theorem 1 (Completeness and cost optimality)

*For A* search, suppose that the branching factor is bounded by a finite number, every action cost is greater than some fixed* $\epsilon > 0$*, and* $h$ *is admissible. If a solution exists, A* eventually selects a goal, and the first goal selected represents a minimum-cost solution.*

This is the *tree-search form* (different paths to the same state remain separate nodes; *graph search* instead detects repeated states and retains the lowest-cost path found so far for each state) of the original A* *admissibility result* of Hart, Nilsson, and Raphael [Hart, Nilsson, and Raphael 1968], Thm. 1, pp. 102–103.

# Heuristic Search
## IDA*-Search

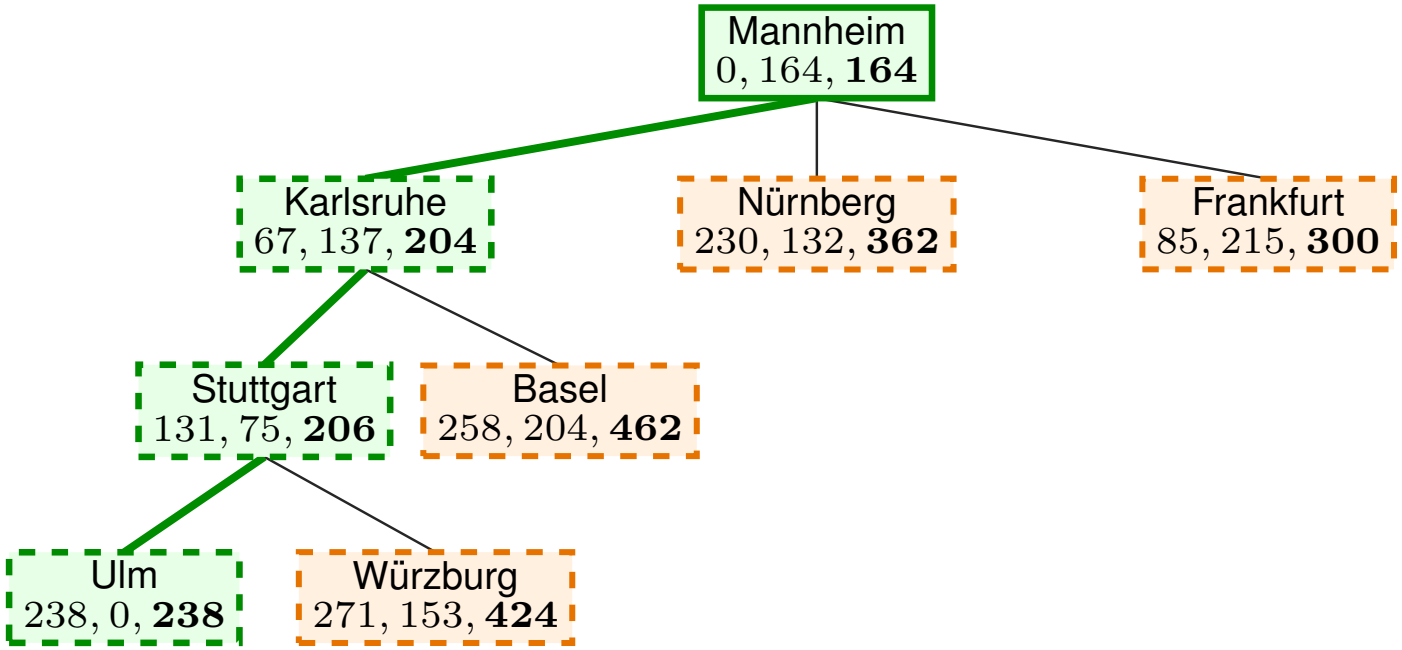

- Weaknesses of A*:
  - High memory requirements
  - A* repeatedly removes the open node with minimum $f(n)$. A binary min-heap priority queue supports insertion and removal of the minimum in logarithmic time.
- Solution: IDA*-algorithm [Korf 1985]
  - Iterative Deepening
  - The same as depth-first search, but limit for heuristic evaluation $f(n)$ (instead of depth limit).
    - Denote the current threshold by $\theta$.
    - Initially, set $\theta = f(n_0)$ for the root node $n_0$.
    - In one DFS iteration, expand a node $n$ only if $f(n) \leq \theta$; cut off its branch for that iteration when $f(n) > \theta$.
    - Stop as soon as a goal node is reached.
    - If no goal is found, set the next $\theta$ to the minimum exceeded $f$ value; if there is no such value, report failure.





## Example 7 (Route from Mannheim to Ulm)

We use IDA* to find a shortest Mannheim–Ulm route. Edge costs are road distances; $h$ is flying distance to the goal. Labels: $g, h, \mathbf{f} = g + h$; at Mannheim, $\theta = f(n_0) = 164$. Search left to right; do not revisit a city already on the current path.




**Iteration 1.** With $\theta = 164$, expand Mannheim. Generated $f$-values $204, 362, 300$ all exceed $\theta$, so they are *cut off*. Next $\theta$ is the smallest exceeded $f$-value, $204$.**Iteration 2.** With $\theta = 204$, expand Mannheim and Karlsruhe. Generated $f$-values $206, 462, 362, 300$ all exceed $\theta$;

# Heuristic Search
## Summary


- Among the *uninformed algorithms compared here*, *iterative deepening* is *complete* and uses *very little memory*. Even it may be impractical when the search space is extremely large.
- *IDA** applies successive $f$*-cost thresholds* using depth-first memory. It saves memory compared with A* but may *regenerate nodes* across iterations.
- *Good heuristics* greatly *reduce the effective branching factor*.
- If a heuristic *only changes the exploration order*, proving that no solution exists still requires *exhausting the relevant reachable search space*.
  - For arbitrary first-order formulas, deciding *logical validity* is undecidable. An unsuccessful proof search may therefore *continue indefinitely*.

# Heuristic Search
## Summary




**Question**

How to find good heuristics?

- *Manually*, e.g. by *simplification of the problem*.
  - *Simplify* the original problem.
  - *Solve* the simplified (easier) version.
  - Optimal solutions from the simplified version $\Rightarrow$ *heuristic functions*.
- *Automatic generation of heuristics* by *machine-learning techniques*.

# Games with Opponents
## Introduction




- *Games for two players*
- Chess, Checkers, Othello, Go
- *deterministic* (= *every action (a move) results in the same child state given the same parent state*), *observable* (= *every player always knows the complete game state*)
- Card games: *only partially observable*.
  - The player *does not know the other players' cards*, or only has partial knowledge about them.
- *Zero-sum games:* Win + Loss = 0
  - *Every gain one player makes means a loss of the same value for the opponent*.

# Games with Opponents

## What the Algorithms Solve


*Goal:* Choose the *next move* that *performs best against the opponent's best replies* [Ertel 2025], Sec. 6.4.1.

*Input:* Current position, player to move, legal moves and resulting positions, and leaf scores [Russell and Norvig 2021], Sec. 5.1.1.

*Output:* A *recommended next move*.

*Why important?* The opponent *chooses against us*, so we must *consider their replies*.

**Algorithms**

- *Minimax search* assumes the opponent's best replies and *chooses our best move*.
- *Alpha-beta pruning* preserves minimax's *root value* and returns *an optimal move*, while skipping subtrees that cannot change that value.

# Games with Opponents
## Minimax Search


- A typical chess position has *about* $30$ *to* $35$ *legal moves*. Using $b = 30$ as a simple model, 50 moves per player gives $30^{100} \approx 10^{148}$ leaves, so *full search is infeasible*.
- A *time limit* means that search must *stop at a chosen depth*.
- Minimax first assigns values to positions where its look-ahead stops:
  - if the *game has finished*, use its exact outcome score;
  - if the game is unfinished but the *search limit has been reached*, estimate its value using $B(s)$.
    - **Why?** Small games may be searched to the end. In large games, searching every continuation is infeasible, so a depth or time limit may stop the search earlier.
- Call the player whose result we optimize *Max* and the opponent *Min*. Assume Min *always chooses their best move*.
- At an internal node, calculate its *minimax value* from its children: Max takes the highest value and Min the lowest. All values are from *Max's perspective*.

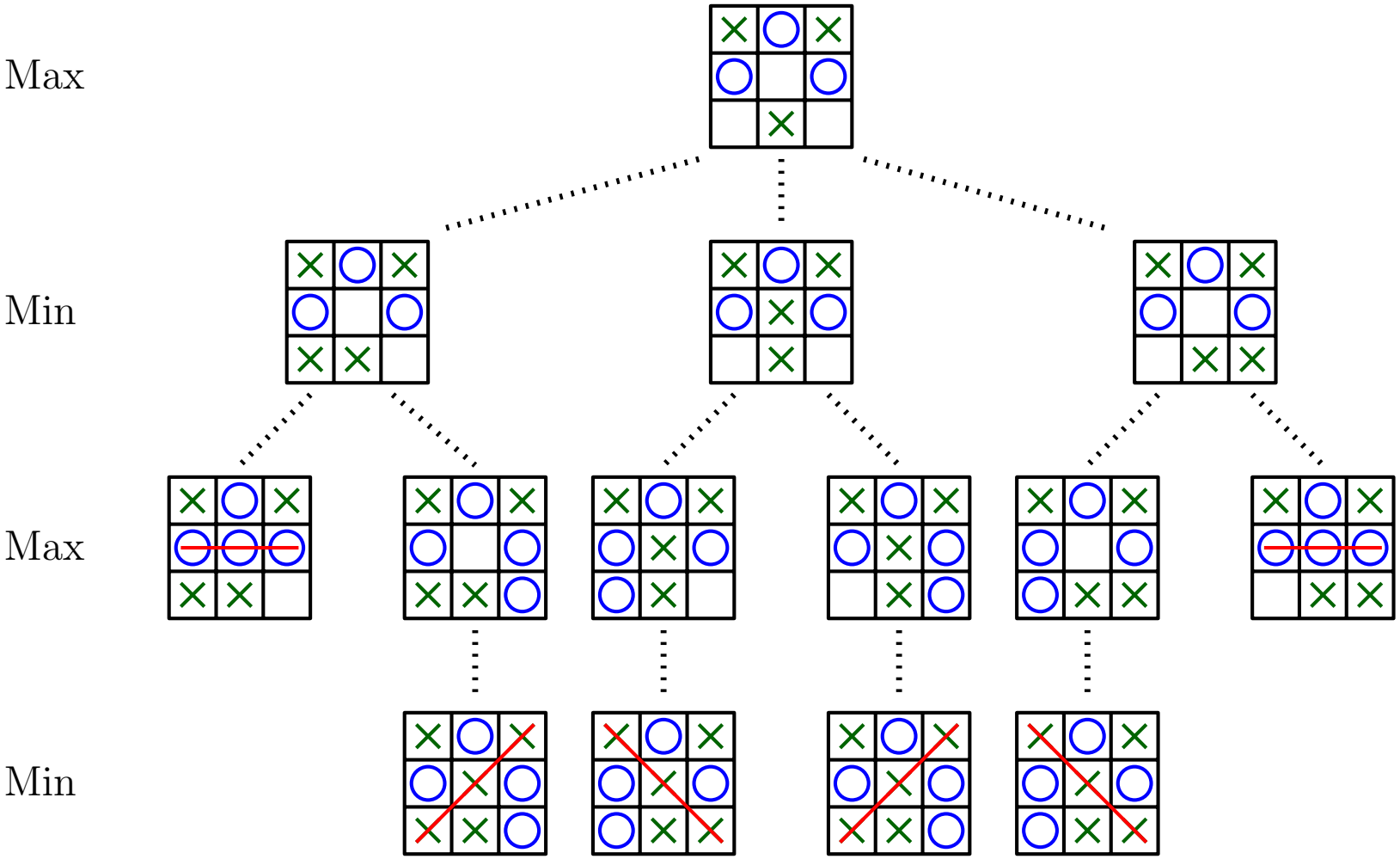
# Games with Opponents
## Minimax Search: Tic-Tac-Toe Example


- **Game Setup:** A 3x3 Tic-Tac-Toe board with two players: Max (X) and Min (O). Players alternate turns, with Max going first.
- **Objective:** The exact score of a finished game is $+1$ for a Max win, $-1$ for a Min win, and $0$ for a draw. A player wins by placing three of their marks in a horizontal, vertical, or diagonal row.
- **Minimax Algorithm:**
  - Starting from the leaf scores, calculate each internal node's minimax value from its children.
  - Max chooses a child with the highest minimax value; Min chooses one with the lowest.

# Games with Opponents

## Minimax Search: Tic-Tac-Toe Example

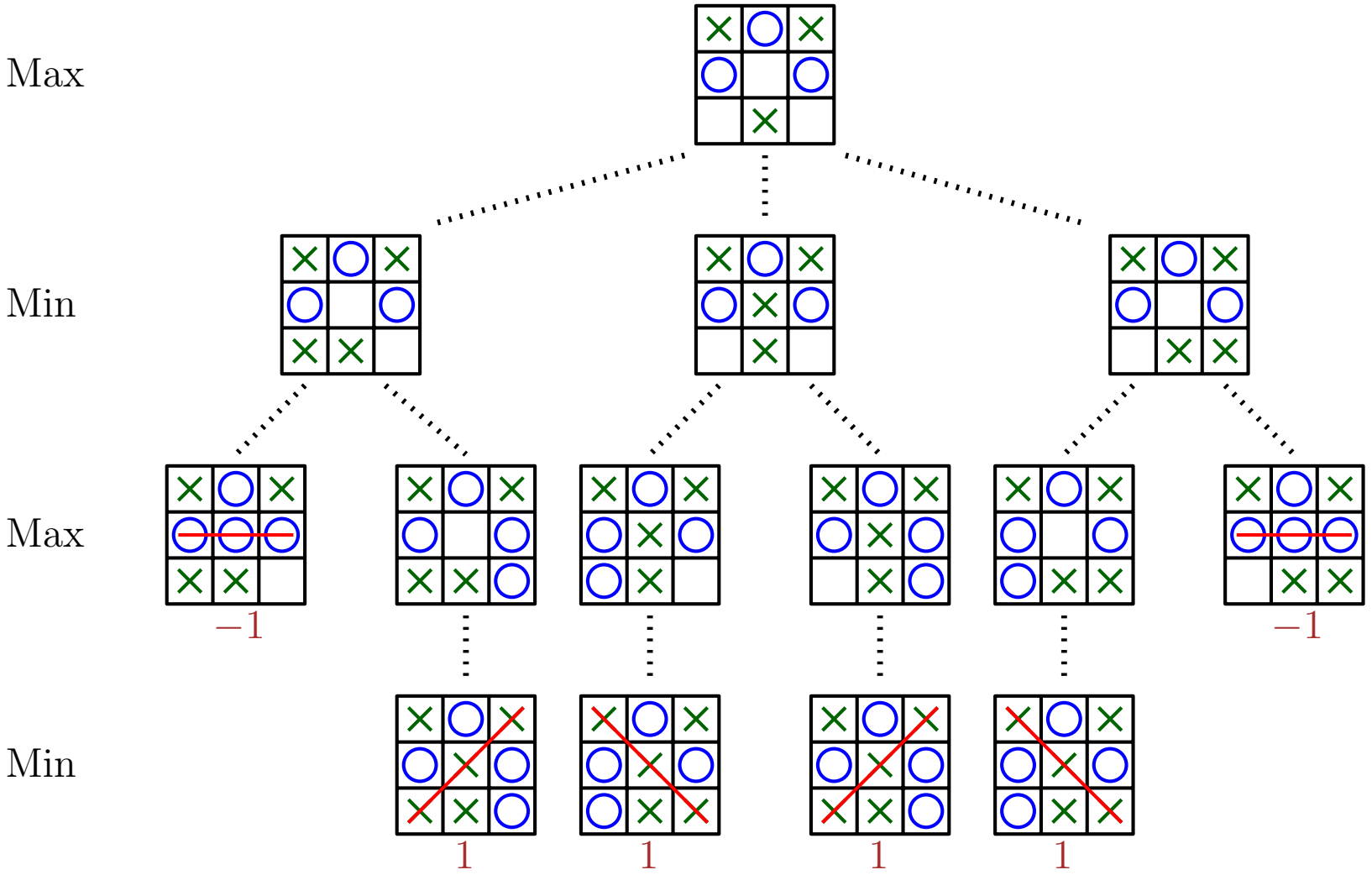
**Example:** A partial game tree starting with Max's turn. The tree alternates between Max placing × and Min placing ○, with the current player shown on the left.



# Games with Opponents

## Minimax Search: Tic-Tac-Toe Example

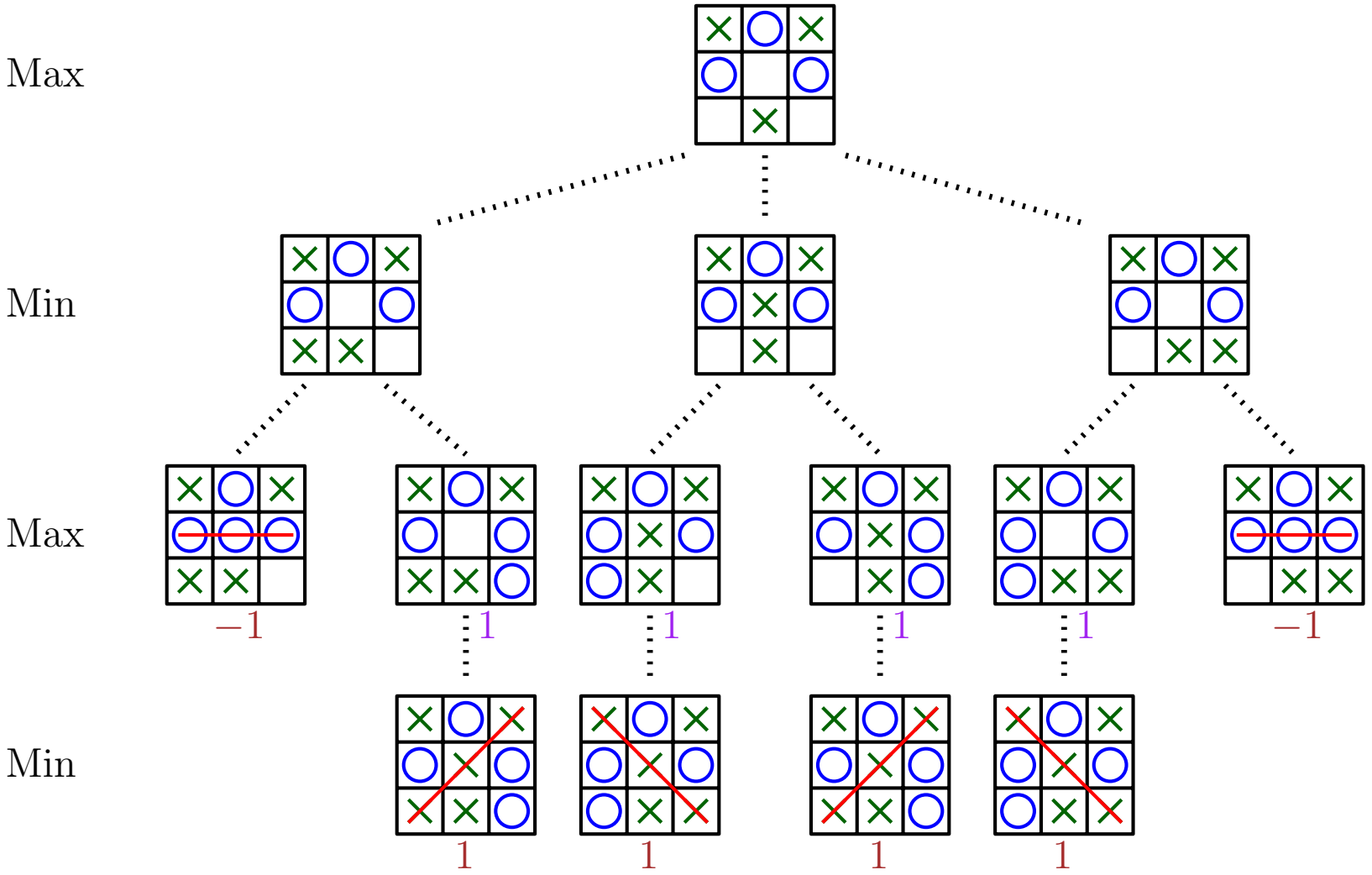
**Example:** A partial game tree starting with Max's turn. The tree alternates between Max placing × and Min placing ○, with the current player shown on the left.



# Games with Opponents

## Minimax Search: Tic-Tac-Toe Example

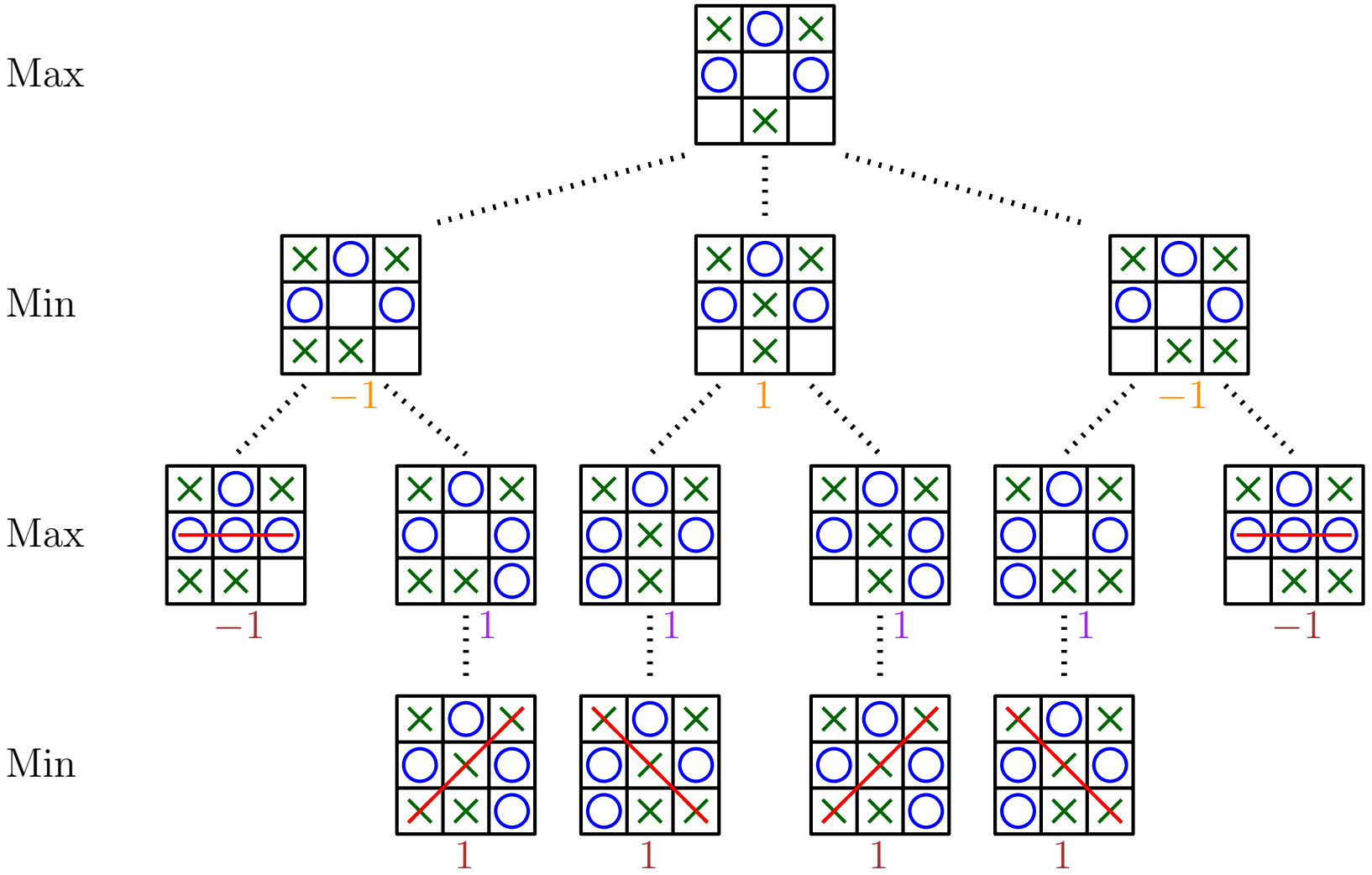
**Example:** A partial game tree starting with Max's turn. The tree alternates between Max placing × and Min placing ○, with the current player shown on the left.



# Games with Opponents

## Minimax Search: Tic-Tac-Toe Example

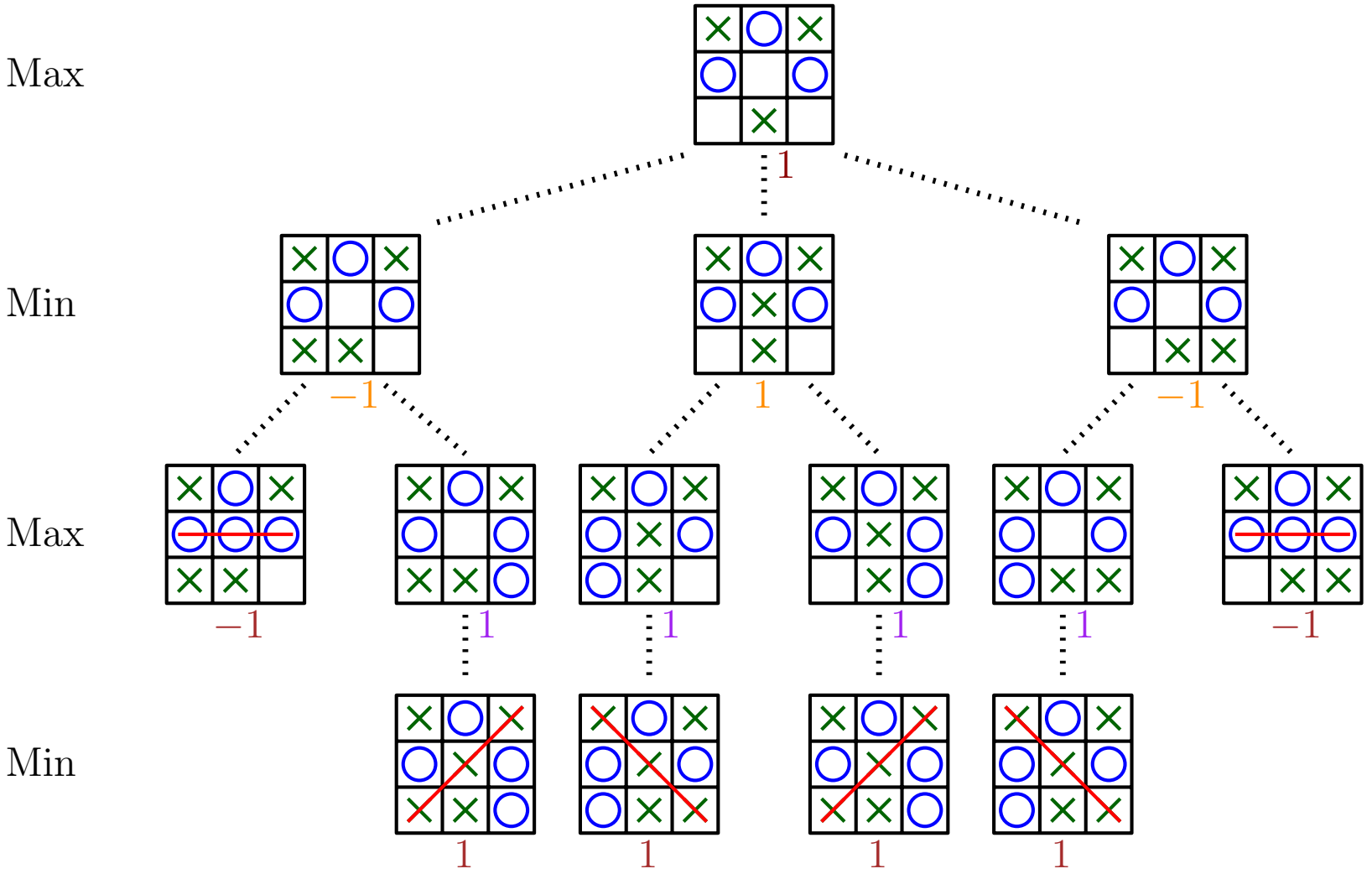
**Example:** A partial game tree starting with Max's turn. The tree alternates between Max placing × and Min placing ○, with the current player shown on the left.



# Games with Opponents

## Minimax Search: Tic-Tac-Toe Example

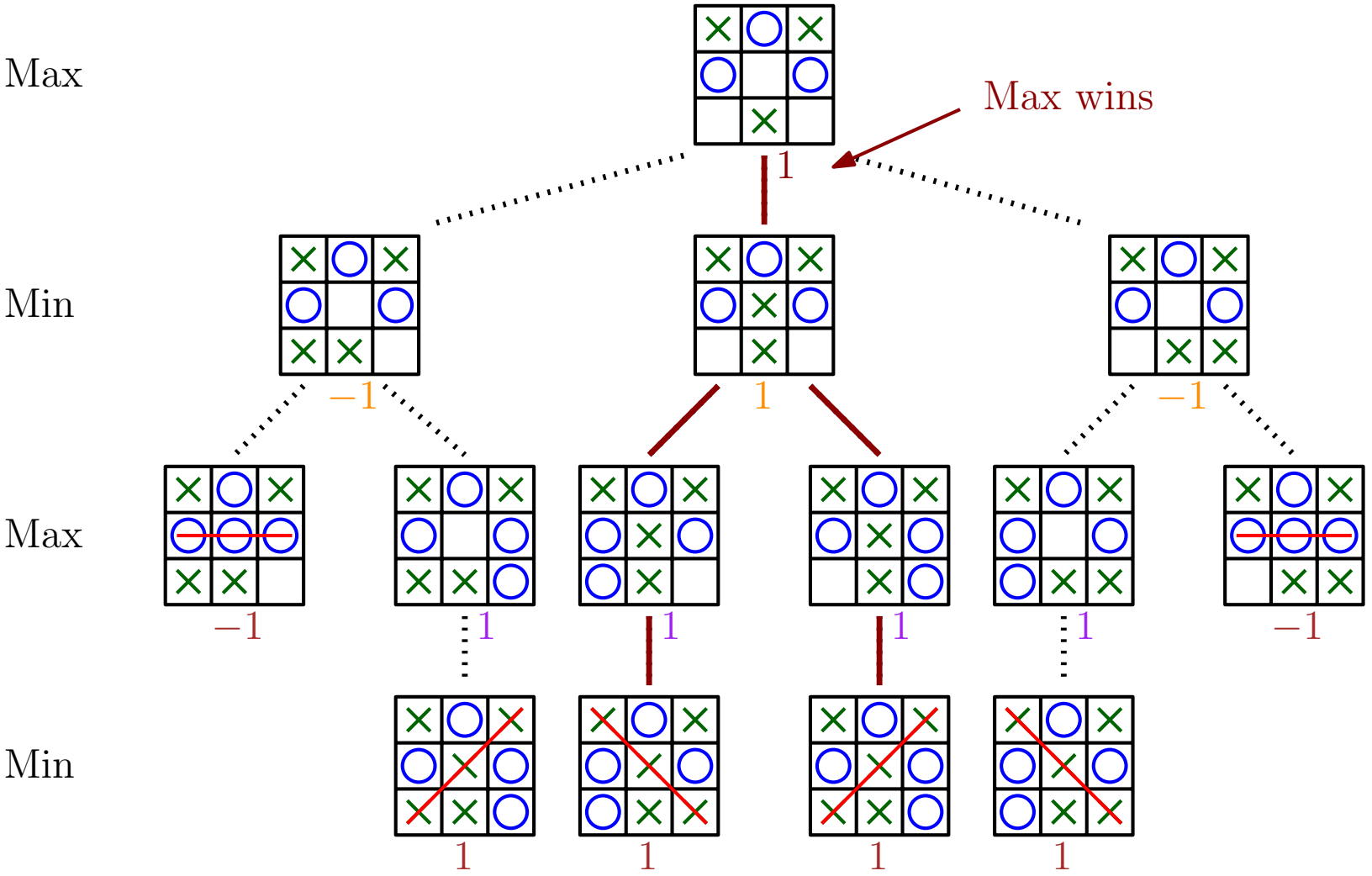
**Example:** A partial game tree starting with Max's turn. The tree alternates between Max placing × and Min placing ○, with the current player shown on the left.



# Games with Opponents

## Minimax Search: Tic-Tac-Toe Example

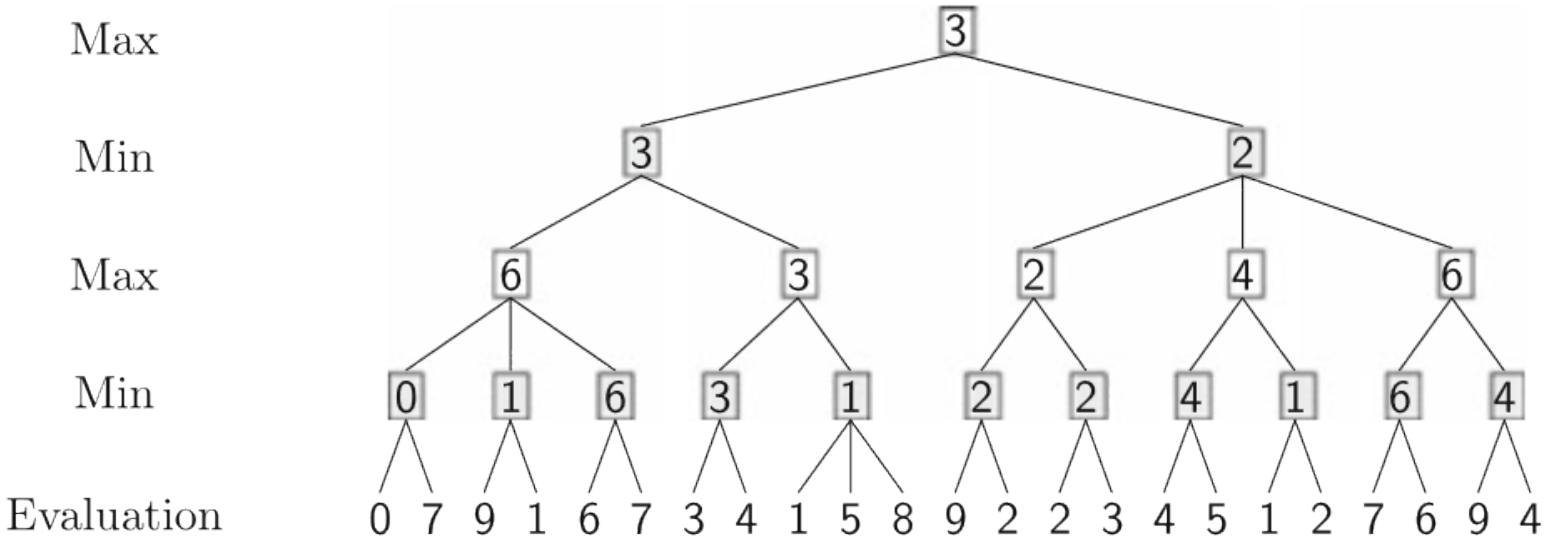
**Example:** A partial game tree starting with Max's turn. The tree alternates between Max placing × and Min placing ○, with the current player shown on the left.





## Exercise 1

The minimax evaluation of a game tree is shown in the figure below [Ertel 2025], Fig. 6.19, p. 117. The look-ahead is four half-moves, and the evaluations of all leaves are given. Explain how the evaluations of the inner nodes are derived.



# Games with Opponents
## Alpha-Beta Pruning






- **Key Idea:** Alpha-Beta Pruning avoids evaluating branches of the game tree that cannot influence the final decision based on the values already discovered during the search.
- It uses two values: *Alpha* $\alpha$ and *Beta* $\beta$.

### Alpha and Beta Values

- **Alpha:** Represents the best (highest-value) score that the maximizing player (usually the AI) can guarantee so far. It acts as a *lower bound*. The initial value of Alpha is $-\infty$.
- **Beta:** Represents the best (lowest-value) score that the minimizing player (the opponent) can guarantee so far. It acts as an *upper bound*. The initial value of Beta is $+\infty$.

# Games with Opponents
## Alpha-Beta Pruning Process

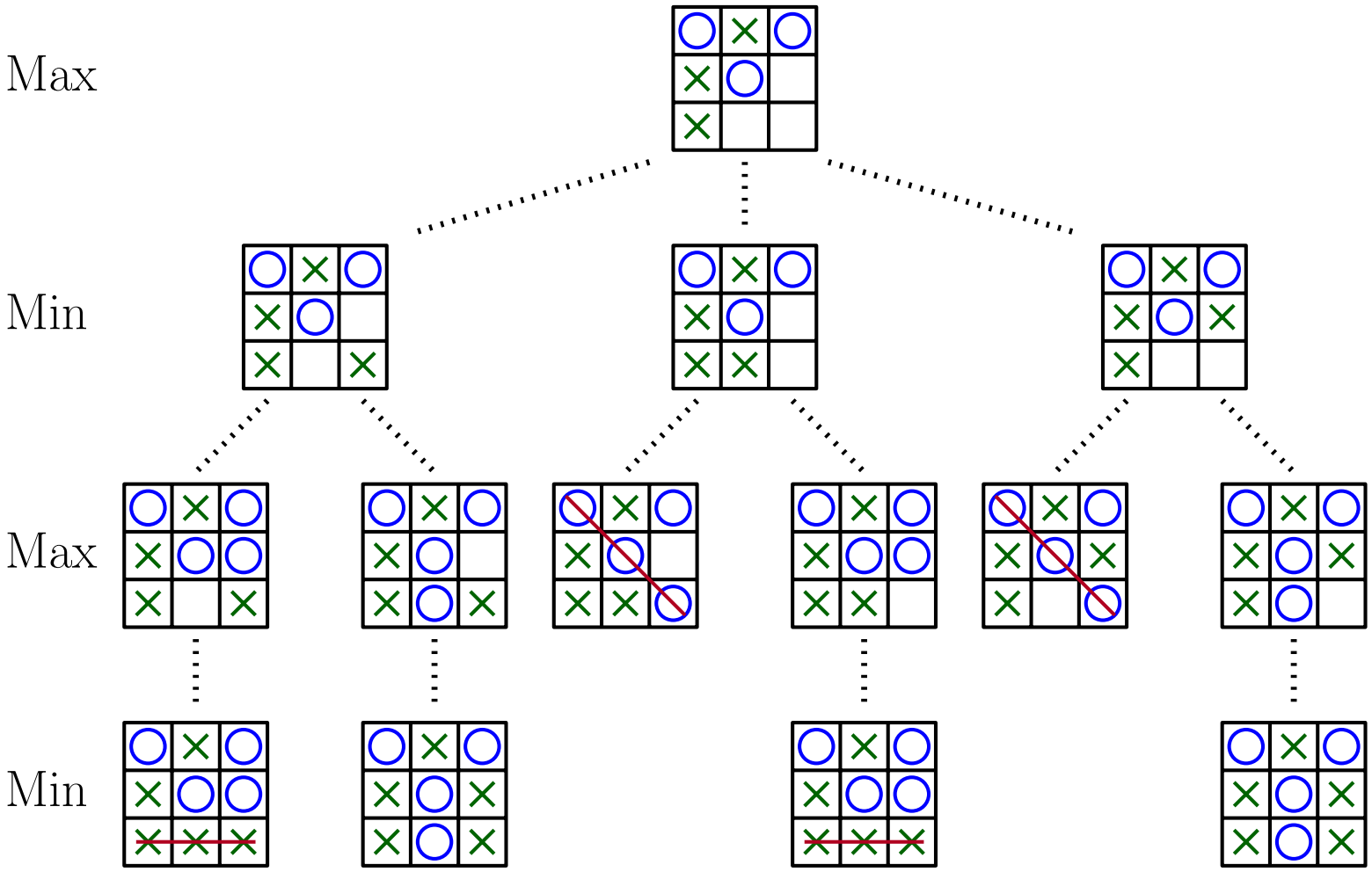

- As the tree is explored, the Alpha and Beta values are updated.
- If *Alpha* $\geq$ *Beta* at a node, its *remaining successors cannot change the root minimax value*.
- Stop examining those successors and *skip their subtrees*. The already examined path is not removed.

**Benefits of Alpha-Beta Pruning**

- Computes the same root minimax value as minimax on the same searched tree.
- Returns an optimal move while often evaluating fewer nodes.
- If several moves tie for best, the two algorithms need not select the same one.

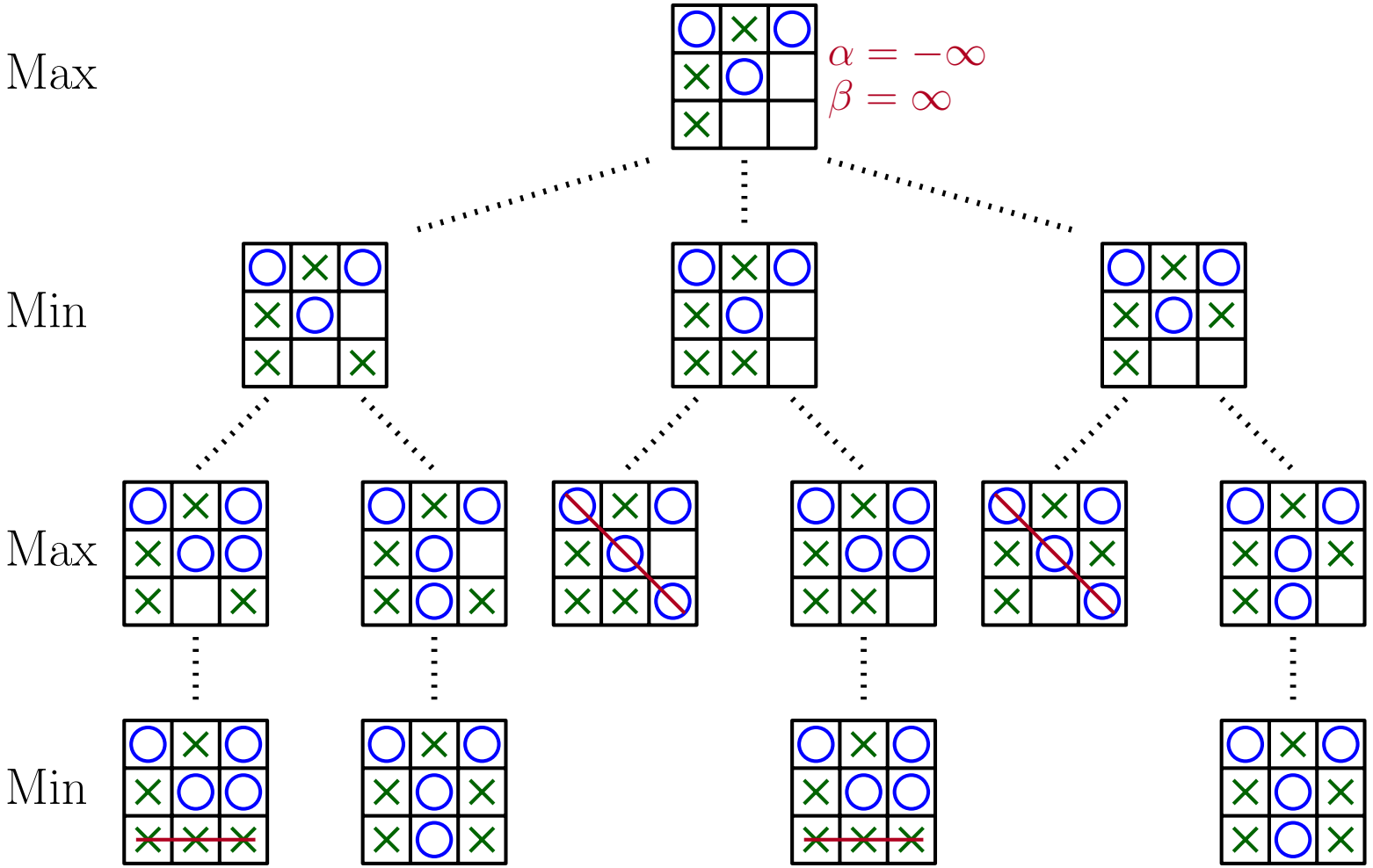
# Games with Opponents

## Alpha-Beta Pruning Process



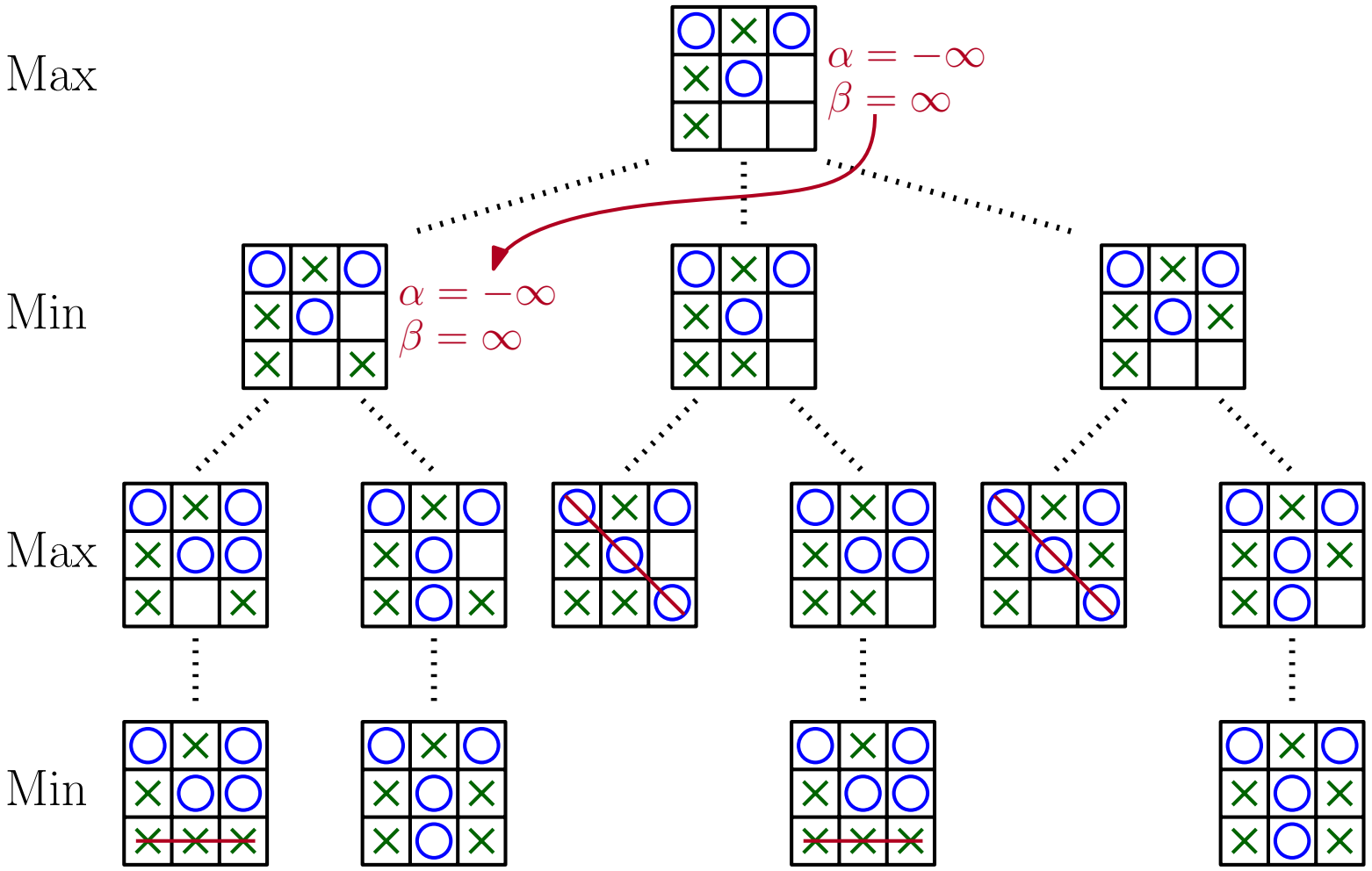

**Step 1.** Start at the root; search depth first from left to right.

# Games with Opponents
## Alpha-Beta Pruning Process




**Step 2.** Initialize the root with $\alpha = -\infty$ and $\beta = \infty$.

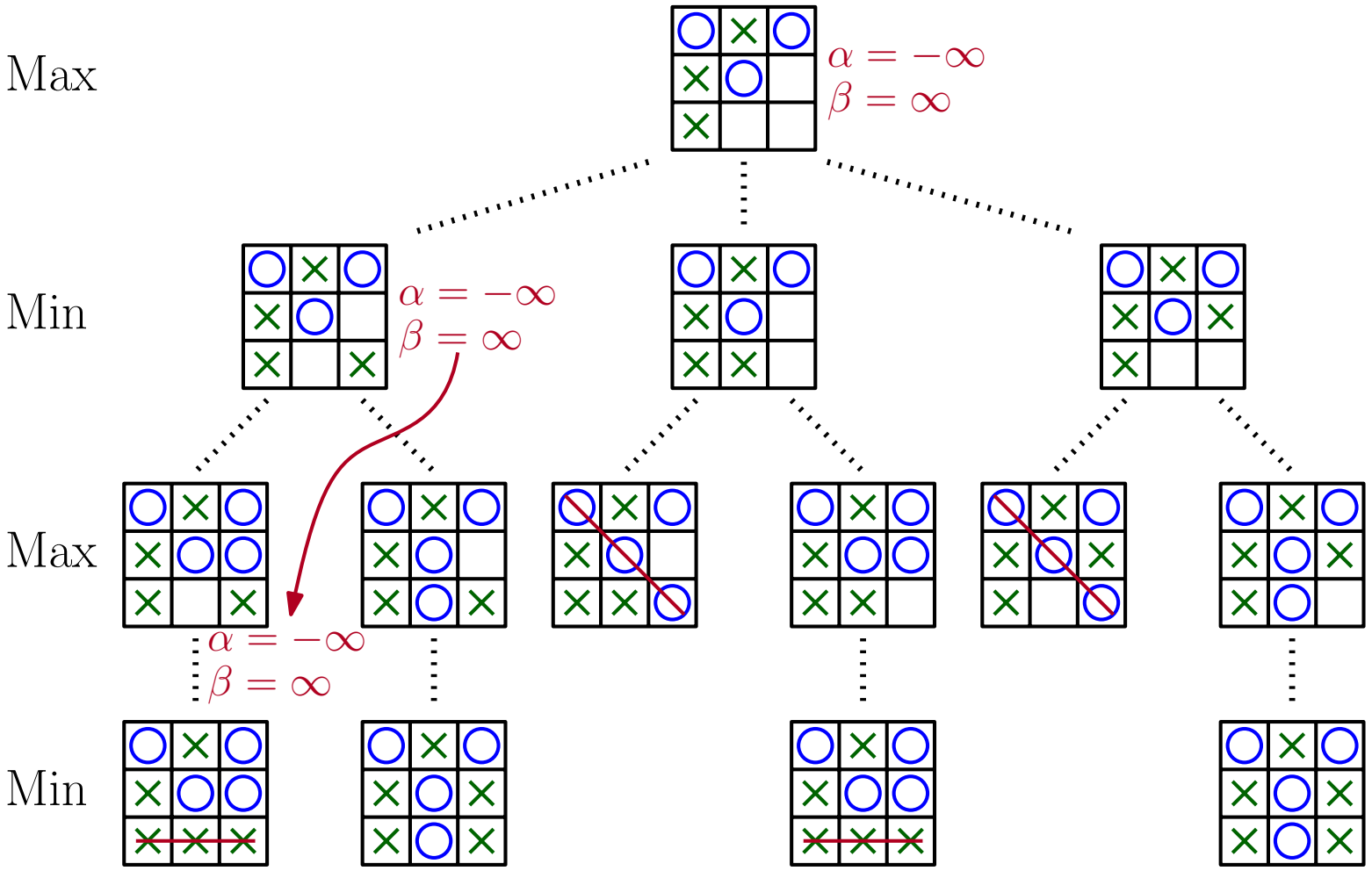
# Games with Opponents
## Alpha-Beta Pruning Process




**Step 3.** Visit the left Min node with $\alpha = -\infty$ and $\beta = \infty$.

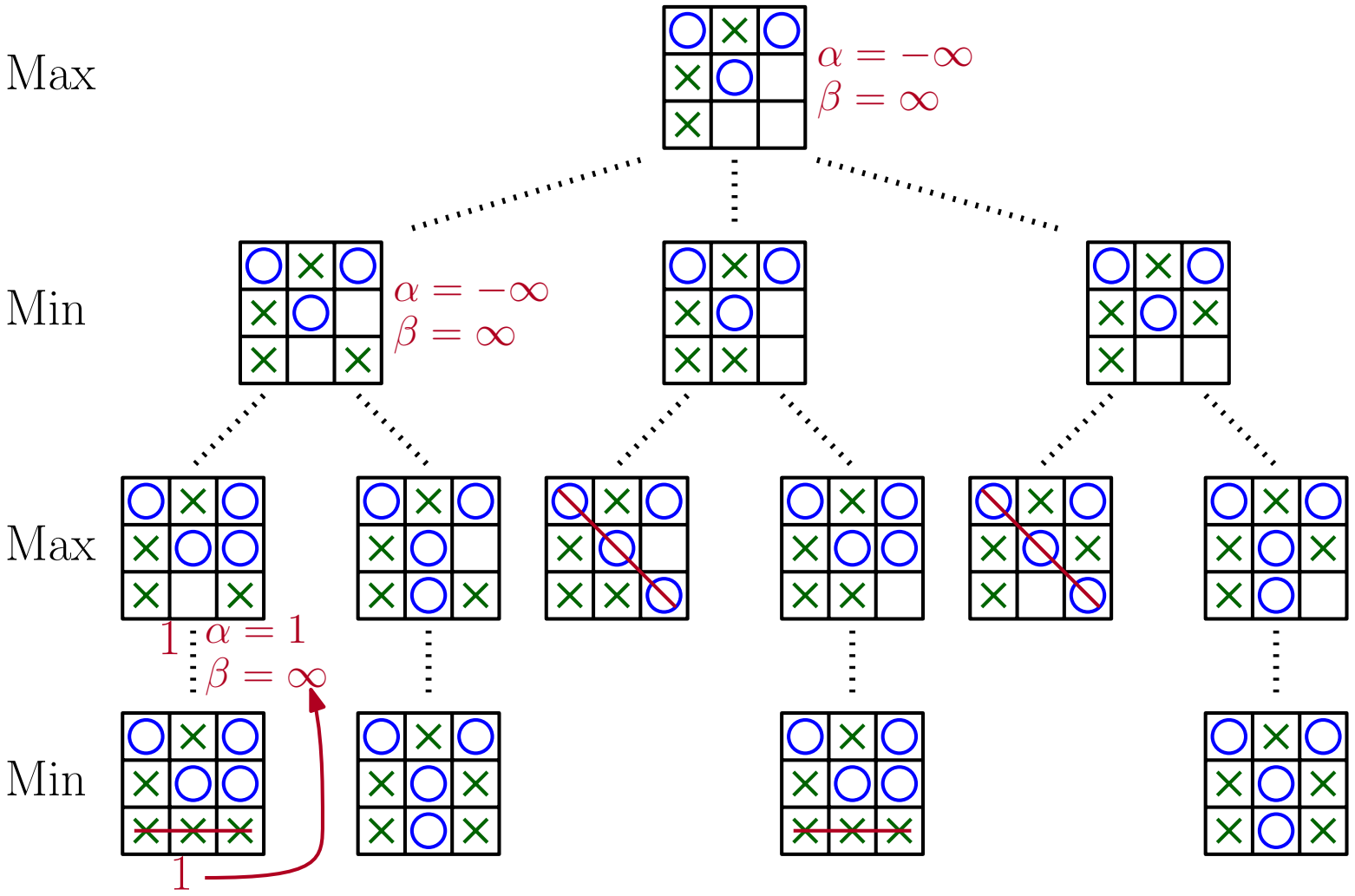
# Games with Opponents

## Alpha-Beta Pruning Process




**Step 4.** Visit its first Max child with the same bounds.

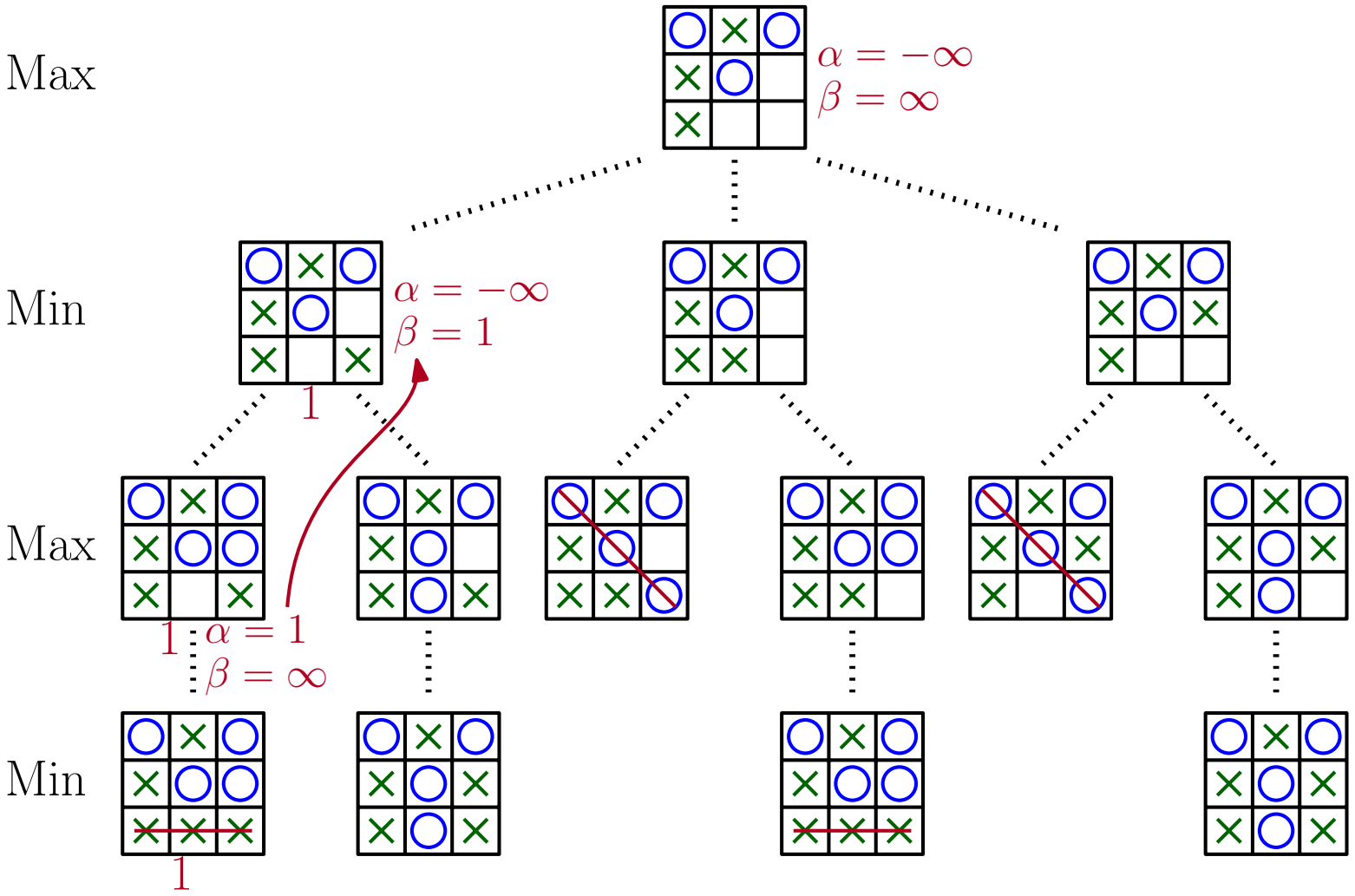
# Games with Opponents
## Alpha-Beta Pruning Process




**Step 5.** Terminal value $1$ sets this Max node's $\alpha = 1$; return $1$.

# Games with Opponents

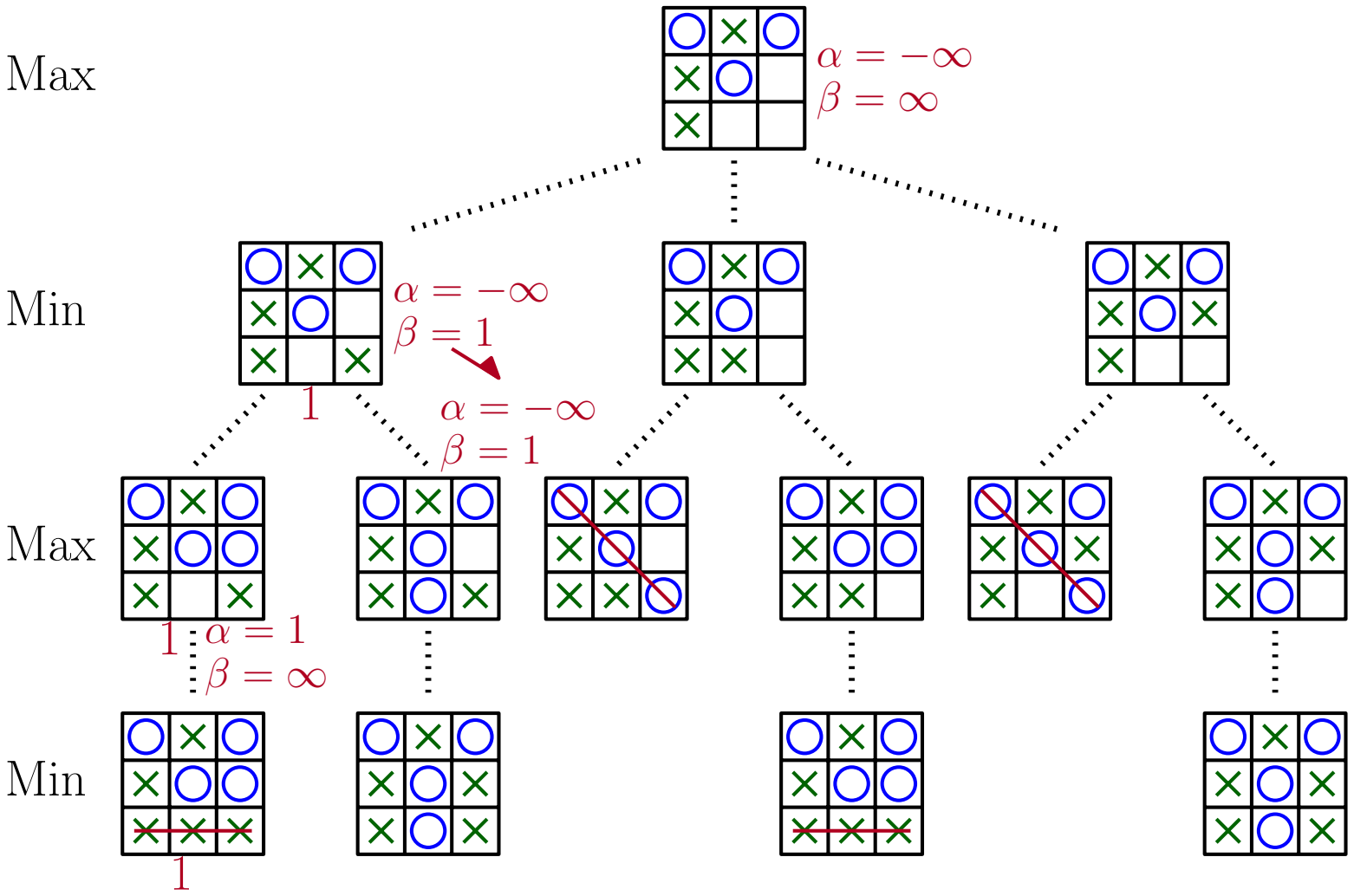
## Alpha-Beta Pruning Process




**Step 6.** Left Min receives $1$ and sets $\beta = \min(\infty, 1) = 1$.

# Games with Opponents

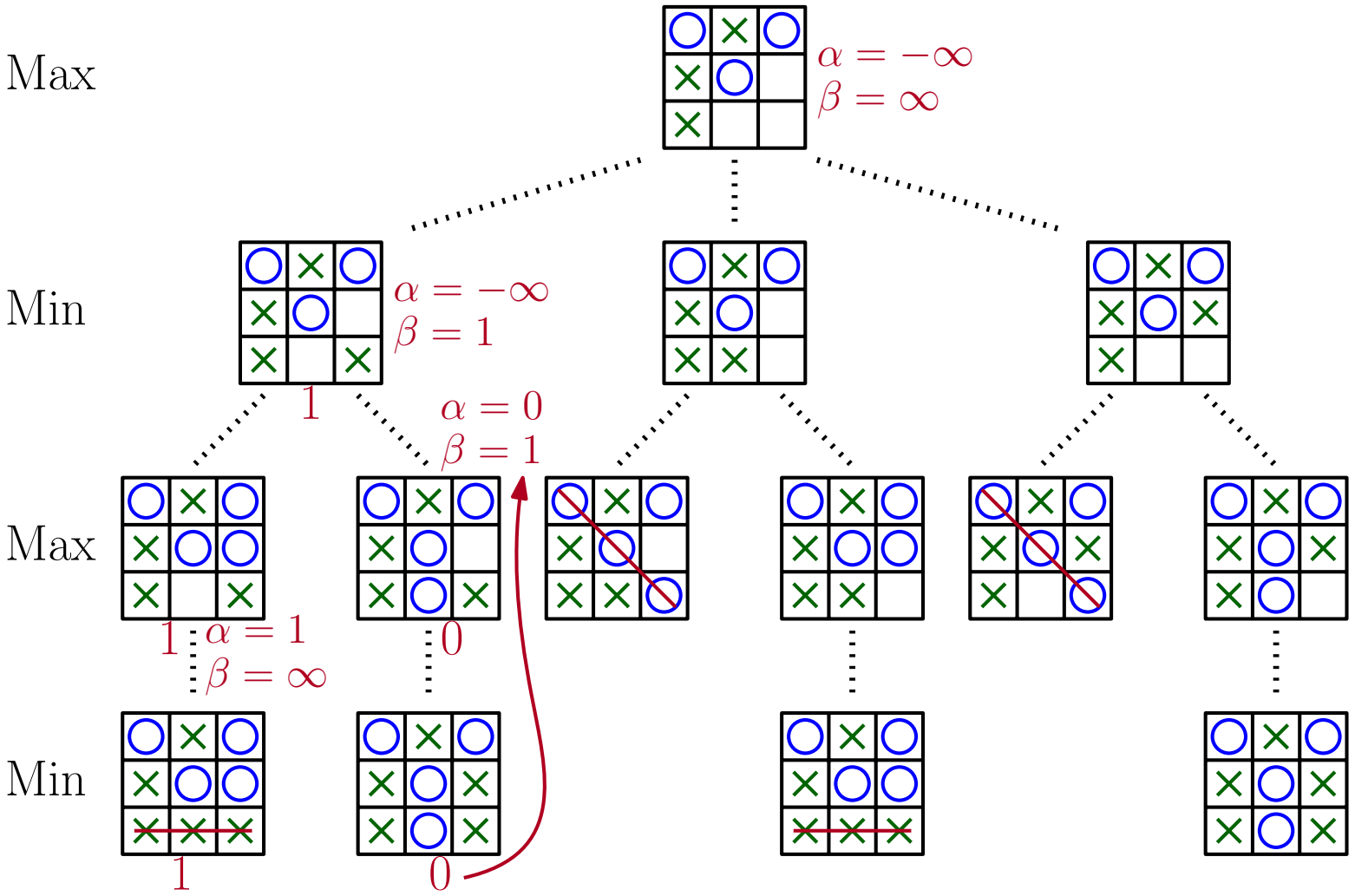
## Alpha-Beta Pruning Process




**Step 7.** Visit its second Max child with $\alpha = -\infty$ and $\beta = 1$.

# Games with Opponents

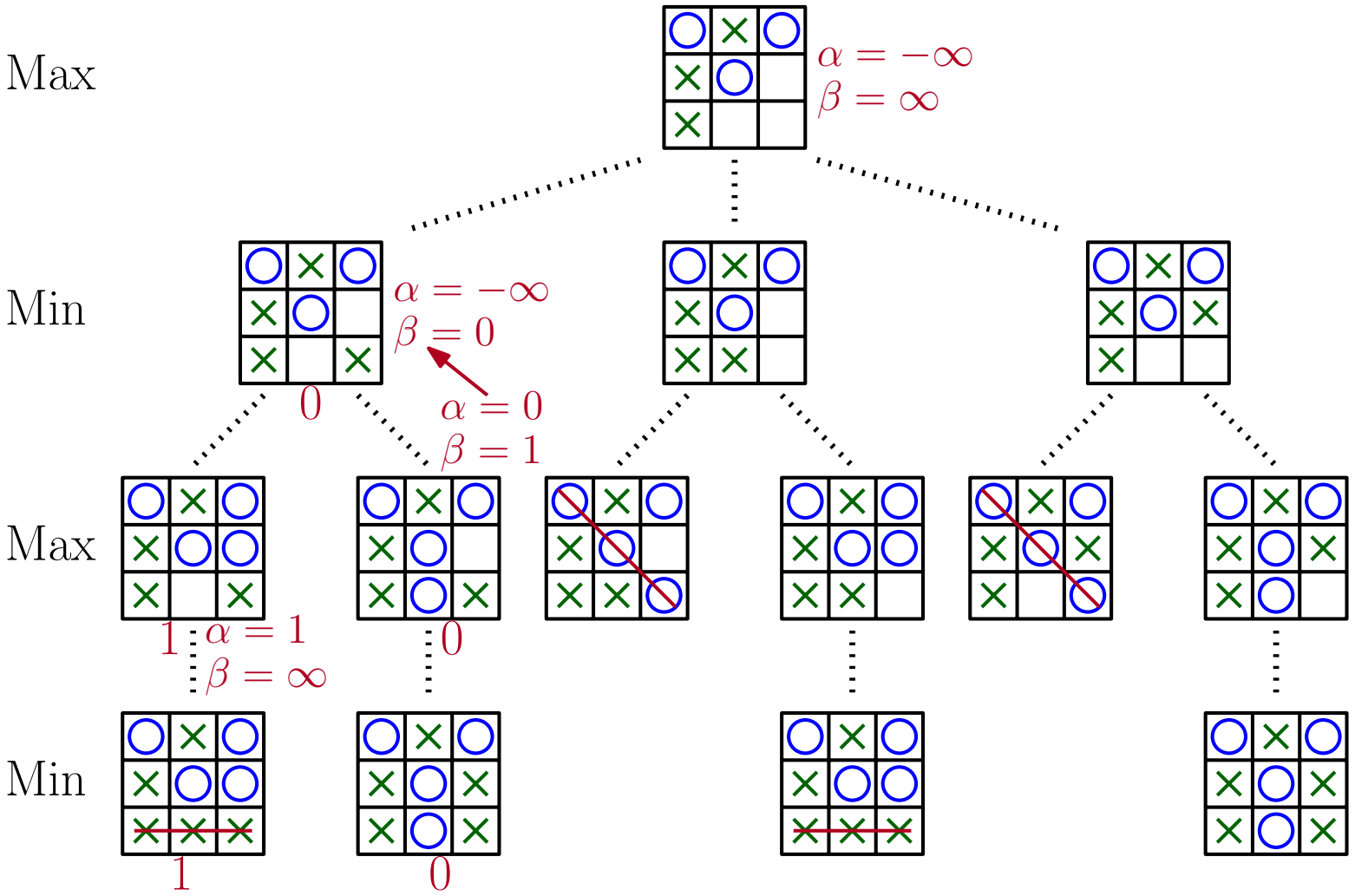
## Alpha-Beta Pruning Process




**Step 8.** Terminal value $0$ sets this Max node's $\alpha = 0$; return $0$.

# Games with Opponents

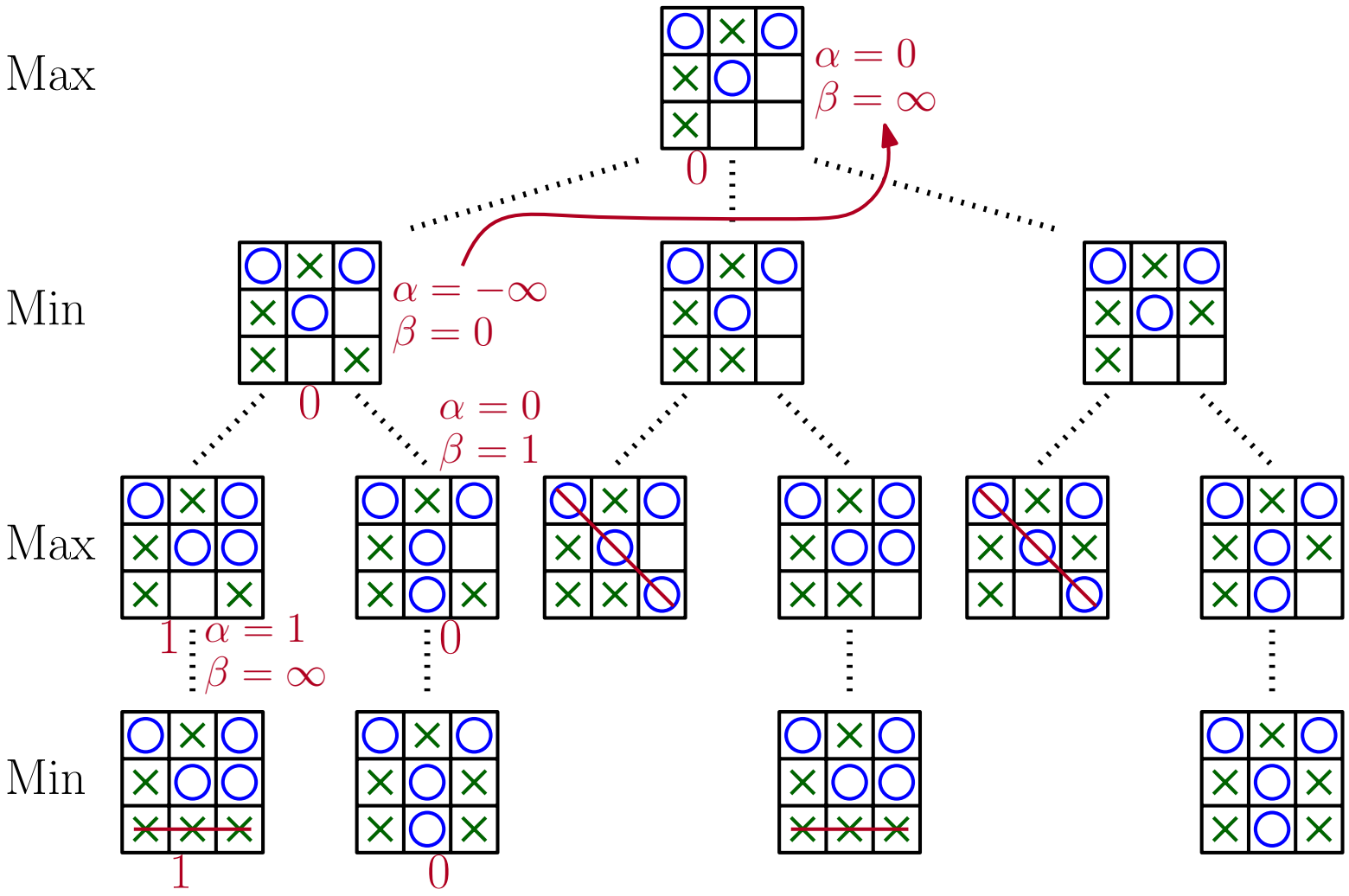
## Alpha-Beta Pruning Process




**Step 9.** Left Min sets $\beta = \min(1, 0) = 0$; return $0$.

# Games with Opponents

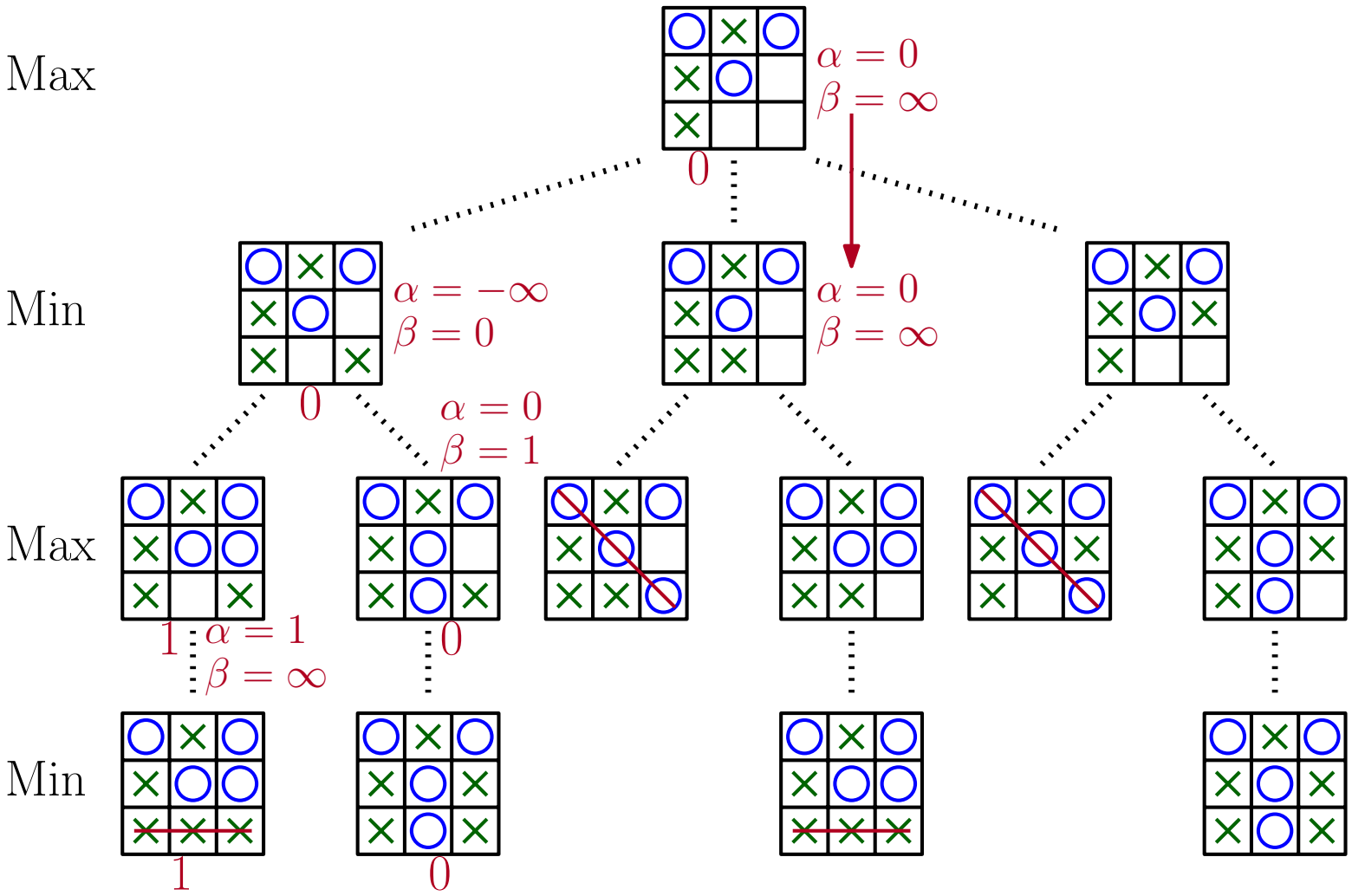
## Alpha-Beta Pruning Process




**Step 10.** The root sets $\alpha = \max(-\infty, 0) = 0$.

# Games with Opponents

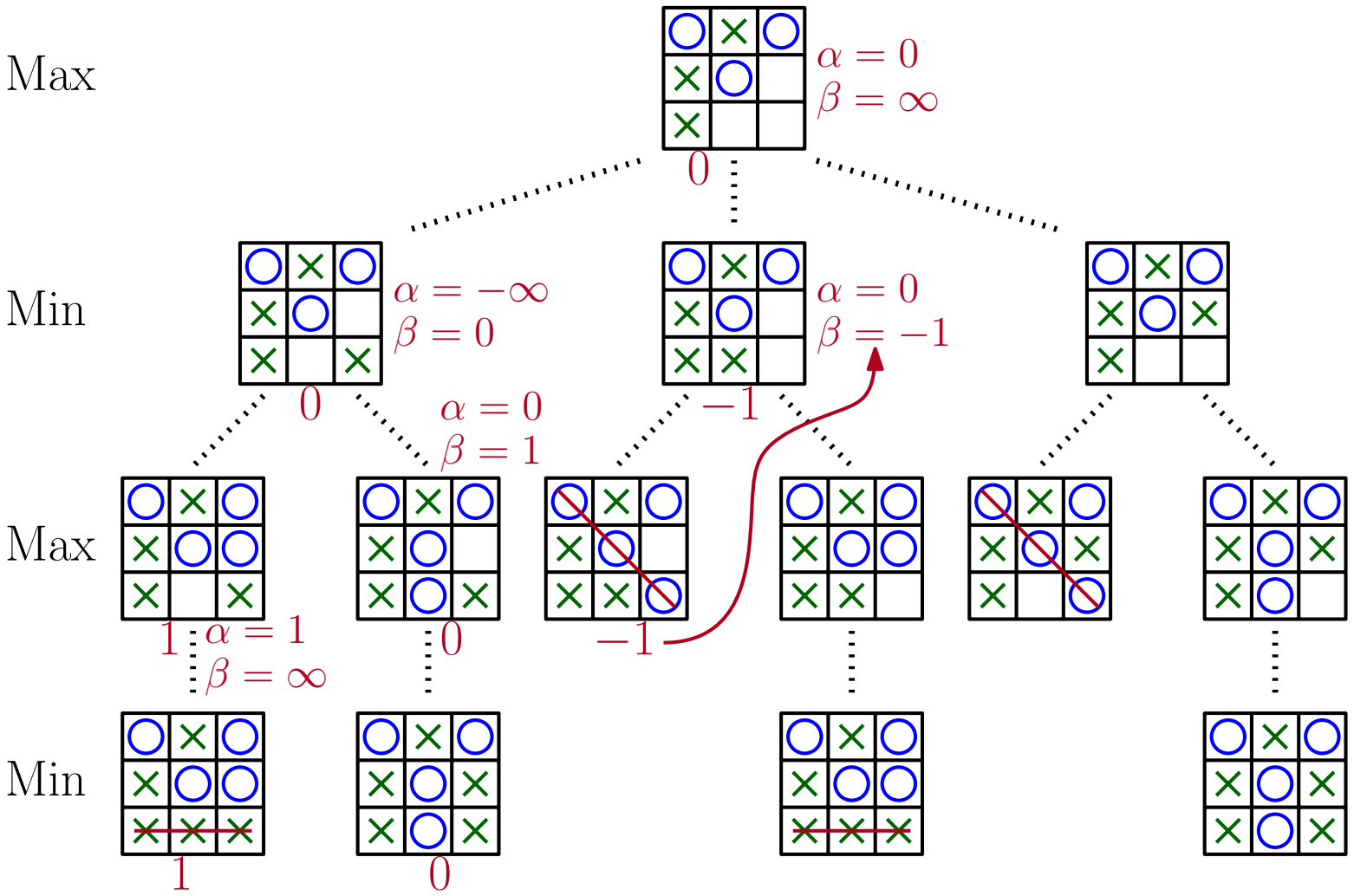
## Alpha-Beta Pruning Process




**Step 11.** Visit the middle Min node with $\alpha = 0$ and $\beta = \infty$.

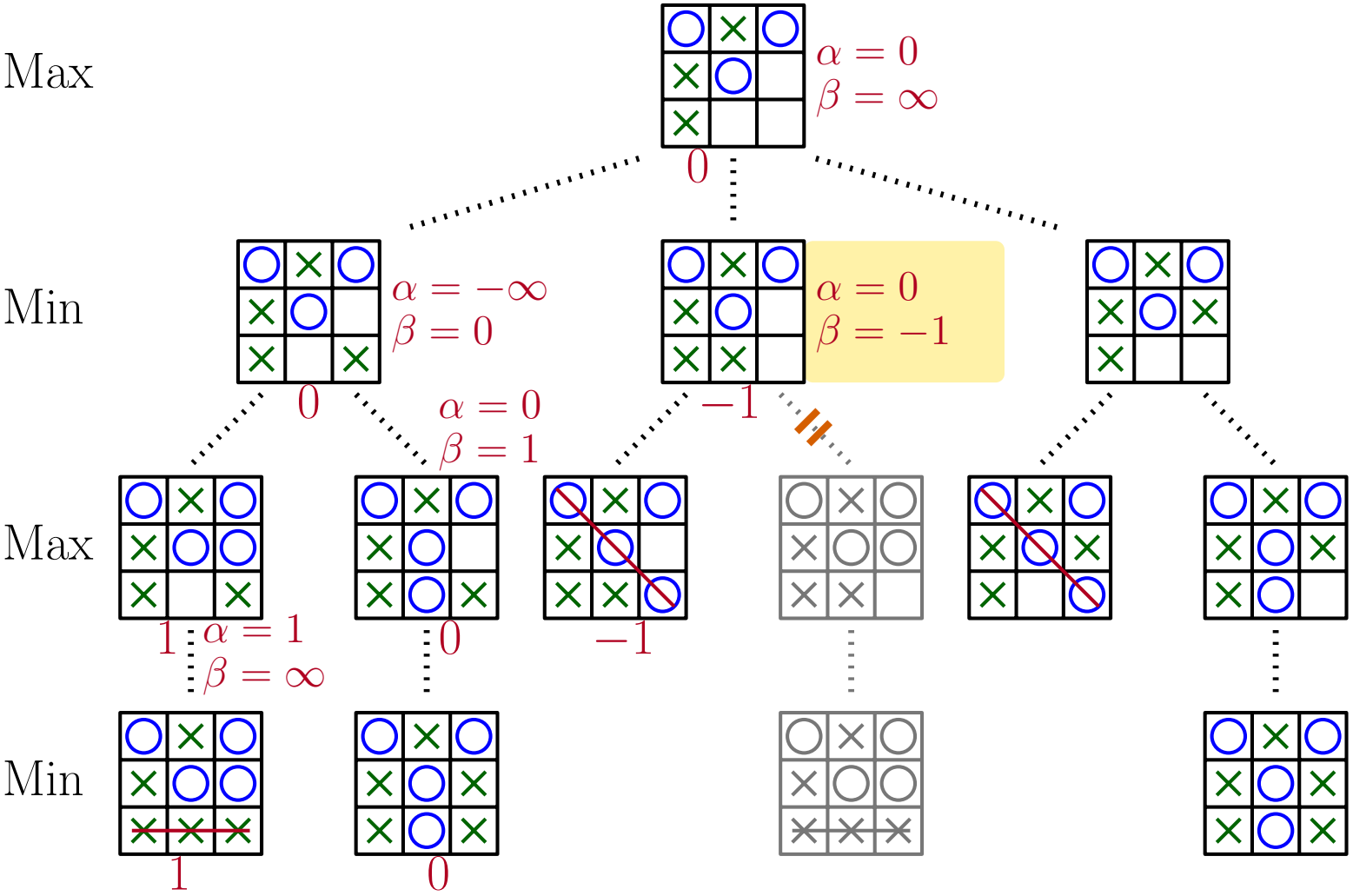
# Games with Opponents

## Alpha-Beta Pruning Process




**Step 12.** First value $-1$ sets $\beta = \min(\infty, -1) = -1$.

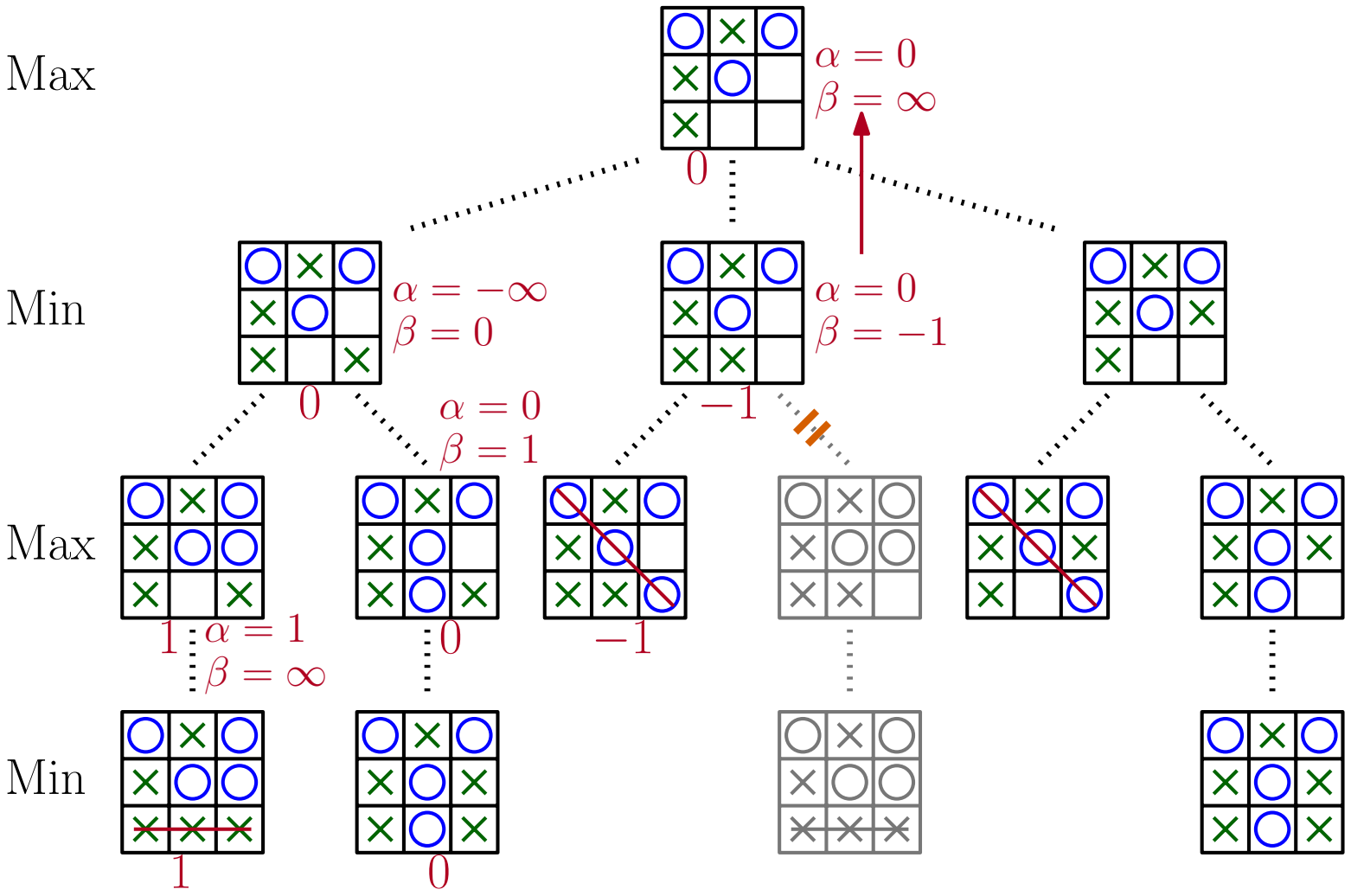
# Games with Opponents
## Alpha-Beta Pruning Process




**Step 13.** $\alpha = 0 \geq \beta = -1$, so prune the unvisited successor.

# Games with Opponents

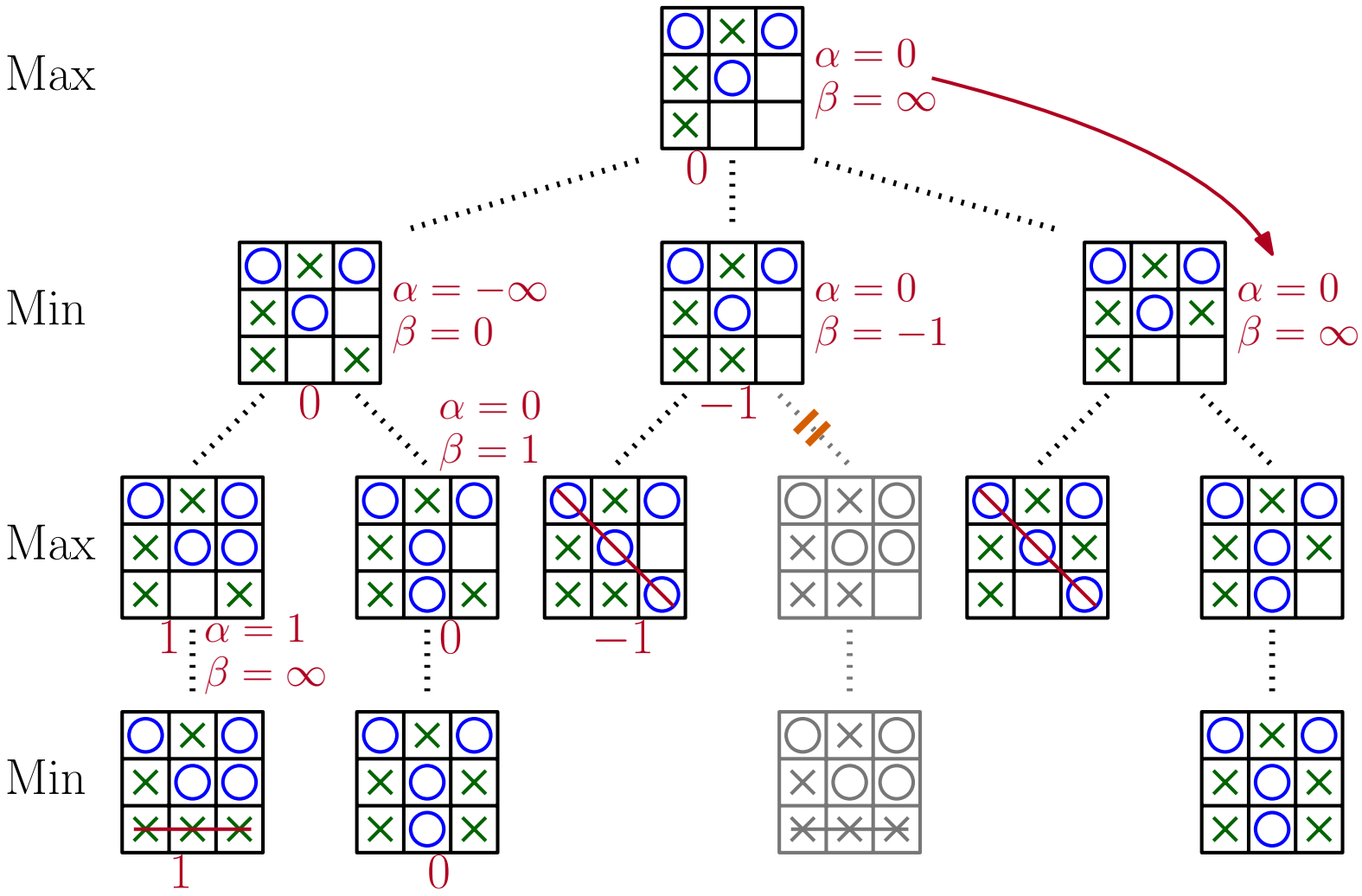
## Alpha-Beta Pruning Process




**Step 14.** Middle Min returns $-1$; the root keeps $0$.

# Games with Opponents

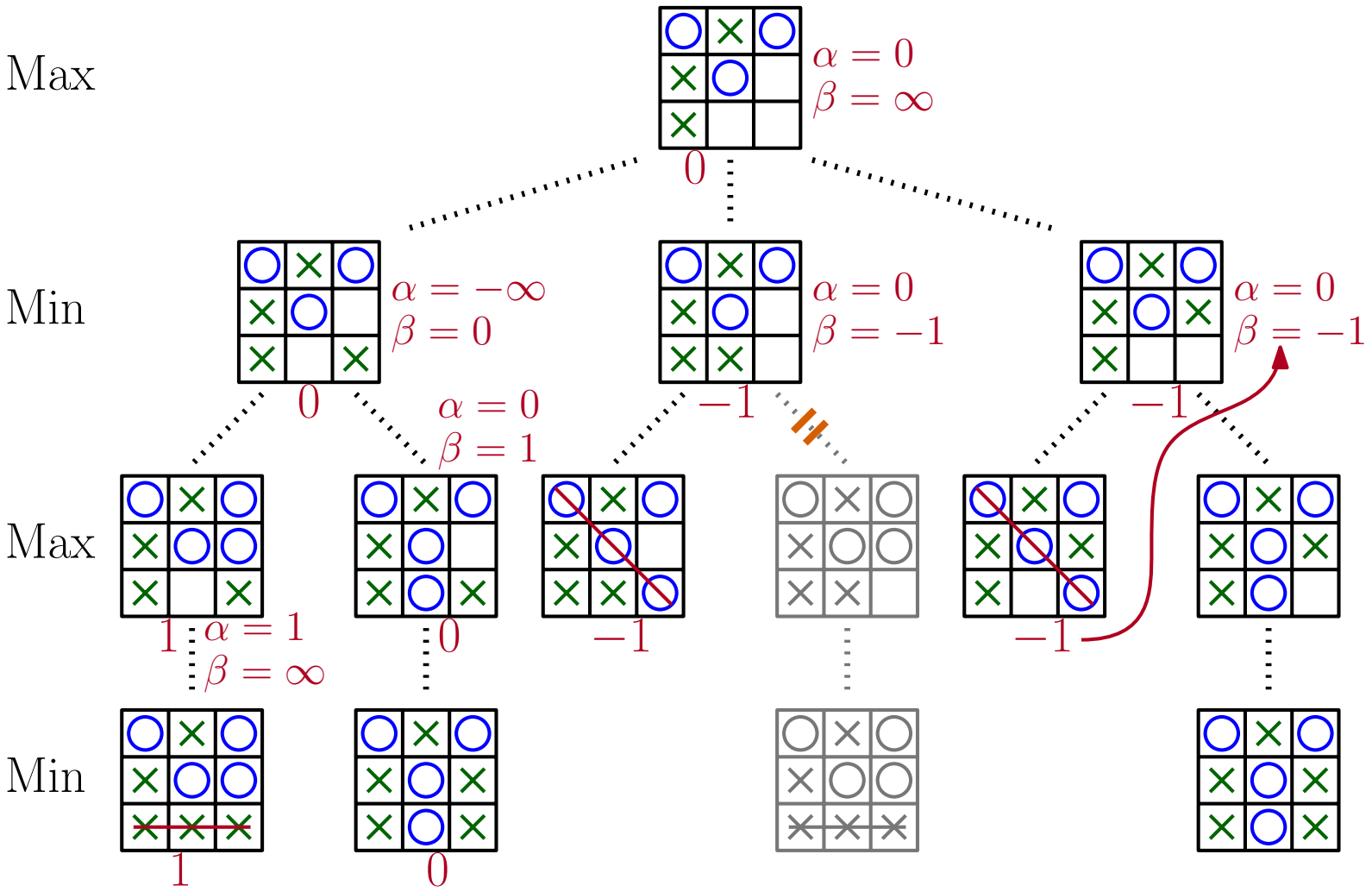
## Alpha-Beta Pruning Process




**Step 15.** Visit the right Min node with $\alpha = 0$ and $\beta = \infty$.

# Games with Opponents

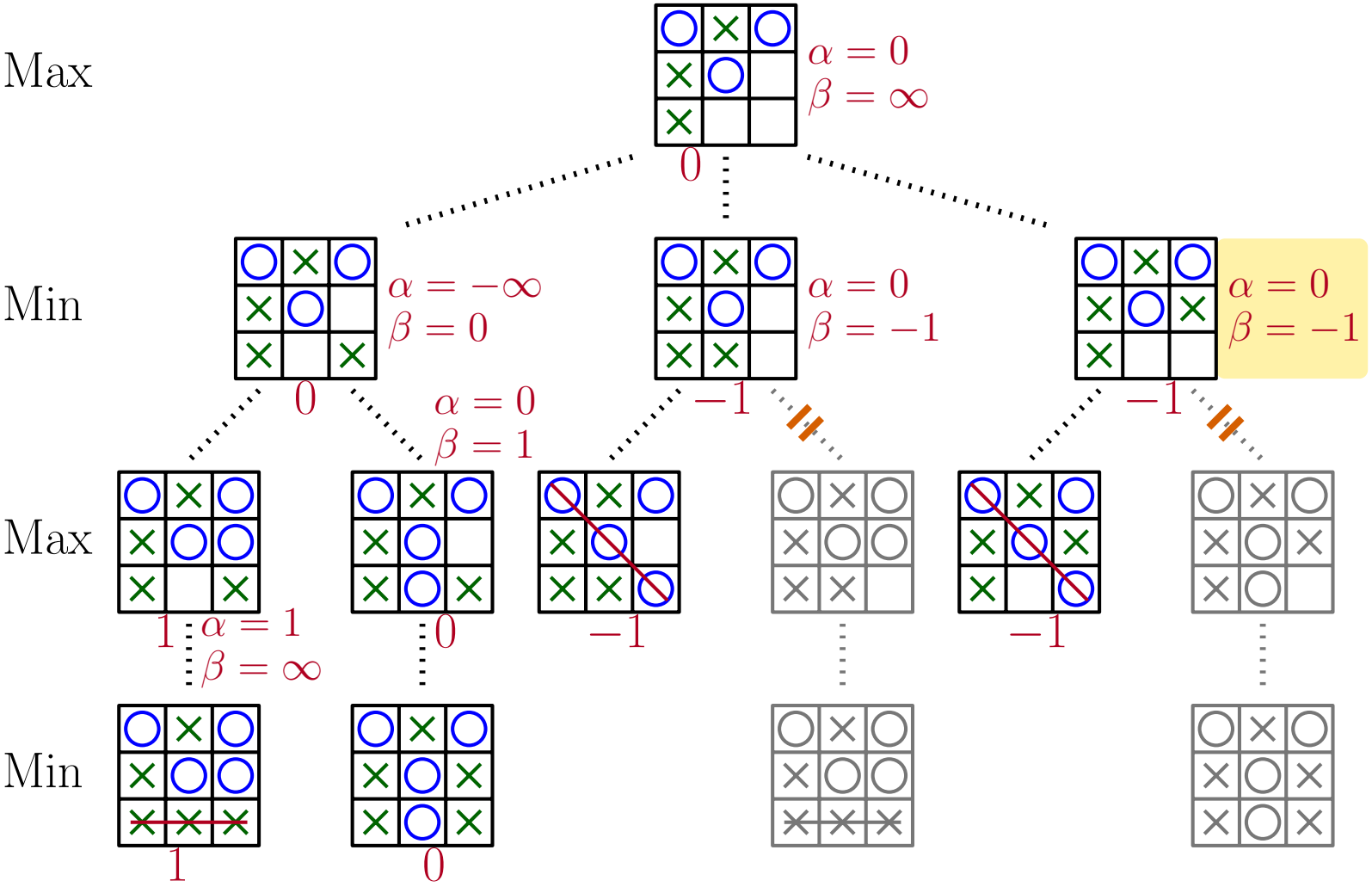
## Alpha-Beta Pruning Process




**Step 16.** First value $-1$ sets $\beta = \min(\infty, -1) = -1$.

# Games with Opponents

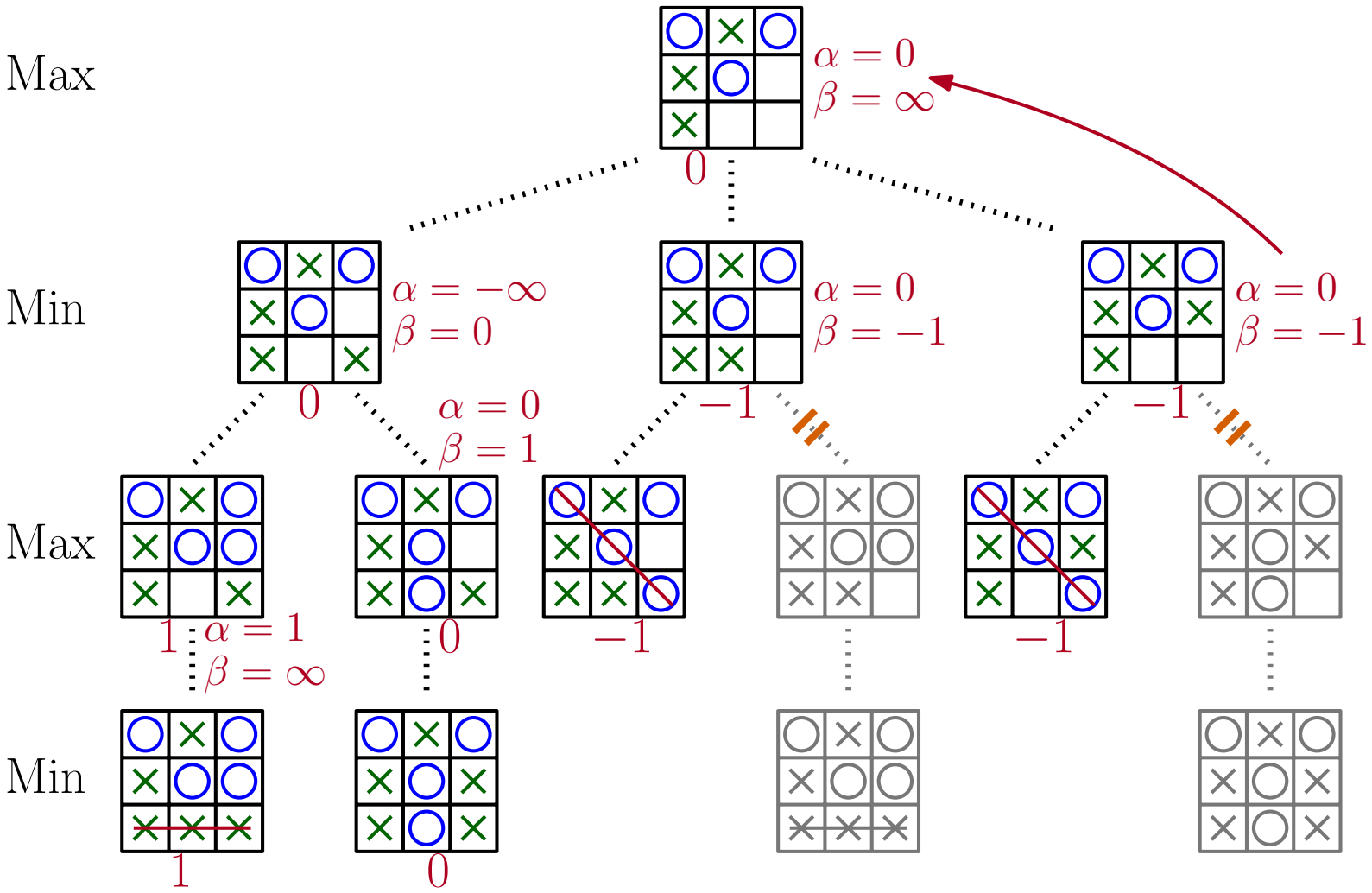
## Alpha-Beta Pruning Process




**Step 17.** $\alpha = 0 \geq \beta = -1$, so prune the unvisited successor.

# Games with Opponents

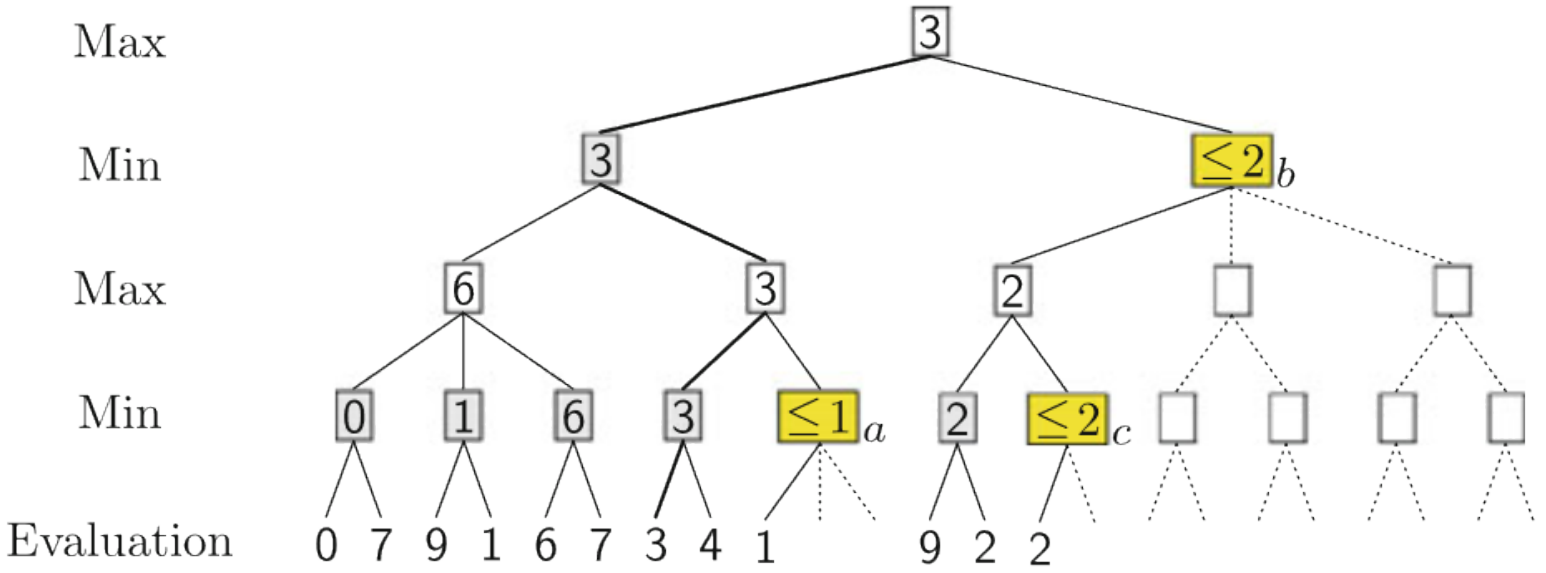
## Alpha-Beta Pruning Process




**Step 18.** Right Min returns $-1$; the root keeps its value $0$.


## Exercise 2

Exercise 1 gave the full game tree and all leaf evaluations from [Ertel 2025], Fig. 6.19, p. 117. The figure below shows alpha-beta pruning applied to the same tree [Ertel 2025], Fig. 6.20, p. 118. Assuming depth-first, left-to-right traversal, explain how the displayed inner-node values are obtained and why the dotted branches are not traversed.




**Analysis:** (see [Pearl 1984])

- *Computation time* depends heavily on *the order in which child nodes are traversed*.
- *Worst case:*
  - Visit *lower-valued successors first at Max nodes* and *higher-valued successors first at Min nodes*.
  - Alpha-beta evaluates $N_d = b^d$ leaves, the same as minimax.
- *Best case:*
  - Visit *higher-valued successors first at Max nodes* and *lower-valued successors first at Min nodes*.
  - Alpha-beta evaluates about $N_d = (\sqrt{b})^d = b^{d/2}$ leaves.
  - With the same leaf-evaluation budget, it can search *roughly twice as deep*.
  - For $b \approx 35$ in chess, the effective branching factor is about $\sqrt{35} \approx 6$.




## Analysis (cont.):

- *Average-Case:*
  - *The child nodes are randomly sorted*.
  - Effective branching factor $\approx b^{3/4} \Rightarrow N_d \approx b^{3d/4}$ leaf nodes [Pearl 1984].
  - In chess, this means effective branching factor reduces from $35$ to about $14$.
- *Heuristic node order:*
  - Connect alpha-beta pruning with iterative deepening over the depth limit $\Rightarrow$ *At every new depth limit we can access the ratings of all nodes of previous levels and order the successors at every branch*.
  - Effective branching factor of roughly $7$ to $8$, which is close to the optimal value $\sqrt{35} \approx 6$ [Ertel 2025], p. 119.

# Games with Opponents
## Non-deterministic Games




- Consider a game in which *one fair six-sided die* is rolled before each move.
- The game tree contains *Max*, *Min*, and *chance nodes*, in the repeating order Max, chance, Min, chance, . . .
- A chance node has *one successor for each possible die outcome*.
- Its value is the *probability-weighted average* of the resulting position values. Because all six outcomes have probability $1/6$ here, this is the ordinary average of the six successor values [Russell and Norvig 2021], Sec. 5.5, pp. 165–166.

# Heuristic Evaluation Functions

## From Expert Knowledge to Features






The following example illustrates how to construct a heuristic evaluation function using human expertise [Ertel 2025], Sec. 6.5, p. 120.

### Example 8

- Experts identify the *most important factors for selecting a move*. These factors are quantified as *features or attributes*.
- In the simplest case, the features are combined into a *linear evaluation function* $B(s)$ for a position $s$:

$$\begin{aligned} B(s) = a_1 \cdot \text{material} + a_2 \cdot \text{pawn\_structure} \\ + a_3 \cdot \text{king\_safety} + a_4 \cdot \text{knight\_in\_center} \\ + a_5 \cdot \text{bishop\_diagonal\_coverage} + \cdots \end{aligned}$$




## Example 8 (cont.)

- One important feature is the *material advantage*, measured from *Max's perspective*:

$$\begin{aligned}\text{material} &= \text{material}(\text{Max}) - \text{material}(\text{Min}),\\ \text{material}(P) &= 100 \cdot \text{num\_pawns}(P) + 300 \cdot \text{num\_knights}(P)\\ &\quad + 300 \cdot \text{num\_bishops}(P) + 500 \cdot \text{num\_rooks}(P)\\ &\quad + 900 \cdot \text{num\_queens}(P).\end{aligned}$$

- Initially, the weights $a_i$ can be set using *expert judgment* and adjusted after games. A *machine-learning method* can optimize them more systematically.


## Example 8 (cont.)

- Human experts may identify relevant features $f_1(s), f_2(s), \ldots$, while a *learning algorithm adjusts how the features are combined*.
- A simple learning loop is:
  - play games using the current evaluation function;
  - observe the delayed result: victory, defeat, or draw;
  - adjust the evaluation function to improve future decisions.
- However, the final result does not reveal *which earlier decisions contributed to it* or how the evaluation function should change. This is the *credit-assignment problem*.
- *Reinforcement learning* addresses such delayed feedback.
- *Stockfish combines learning and search*: a *trained neural network evaluates chess positions*; during a game, alpha-beta search uses these scores to choose a move [Stockfish developers 2020].

# Search, Evaluation, and Learning in Games

## Take-Home Messages


- Early game-playing programs combined *limited search* with *human-designed evaluation*.
- Strong classical chess systems used *larger alpha-beta searches* and increasingly refined evaluation functions.
- *Monte Carlo tree search* introduced a different approach: spend more computation on promising continuations.
- Modern systems learn *evaluation or search guidance* from games or self-play and combine it with search; *search remains central*.


## Exercise 3

Choose one chess engine or research system released or substantially updated within the last five years. In 300–500 words:

- identify the system, version, and release or publication date;
- explain how it uses search, position evaluation, and learning;
- report one performance result and name the metric or benchmark;
- compare one design choice with an earlier system discussed in this section.

Use at least two reliable sources, including one primary source such as a research paper, official technical document, or official source repository. Give complete citations and state when the information was last checked.

# References



Ertel, Wolfgang (2025). *Introduction to Artificial Intelligence*. 3rd. Springer. DOI: 10.1007/978-3-658-43102-0.


Hatton, Ian A., Eric D. Galbraith, Nono S. C. Merleau, Teemu P. Miettinen, Benjamin McDonald Smith, and Jeffery A. Shander (2023). "The Human Cell Count and Size Distribution." In: *Proceedings of the National Academy of Sciences* 120.39, e2303077120. DOI: 10.1073/pnas.2303077120.


Poole, David L. and Alan K. Mackworth (2023). *Artificial Intelligence: Foundations of Computational Agents*. 3rd. Cambridge University Press. ISBN: 9781009258197. URL: https://artint.info/3e/html/ArtInt3e.html.


Russell, Stuart J. and Peter Norvig (2021). *Artificial Intelligence: A Modern Approach*. 4th. Pearson.

# References (cont.)


Planck Collaboration (2020). “Planck 2018 Results. VI. Cosmological Parameters.” In: *Astronomy & Astrophysics* 641, A6. DOI: 10.1051/0004-6361/201833910.

Stockfish developers (2020). *Introducing NNUE Evaluation*. URL: https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/ (visited on 07/12/2026).

Ertel, Wolfgang (1993). “Parallele Suche mit randomisiertem Wettbewerb in Inferenzsystemen.” PhD thesis. Technische Universität München.

# References (cont.)


Institute of Medicine Committee on a National Neural Circuitry Database (1991). *Mapping the Brain and Its Functions: Integrating Enabling Technologies into Neuroscience Research*. Ed. by Constance M. Pechura and Joseph B. Martin. Washington, DC: National Academies Press. URL: https://www.ncbi.nlm.nih.gov/books/NBK234381/ (visited on 07/14/2026).

Korf, Richard E. (1985). "Iterative-deepening-A*: An optimal admissible tree search." In: *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, pp. 1034–1035.

Pearl, Judea (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Series in Artificial Intelligence. Addison-Wesley Publishing Company.

# References (cont.)

Hart, Peter E., Nils J. Nilsson, and Bertram Raphael (1968). “A Formal Basis for the Heuristic Determination of Minimum Cost Paths.” In: *IEEE Transactions on Systems Science and Cybernetics* 4.2, pp. 100–107. DOI: 10.1109/TSSC.1968.300136.