VNU-HUS MAT1206E/3508: Introduction to AI

Search, Games and Problem Solving

Hoàng Anh Đức

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

Textbook Basis

These lecture slides were constructed based primarily on the contents of:

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

Contents

Additional Materials

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

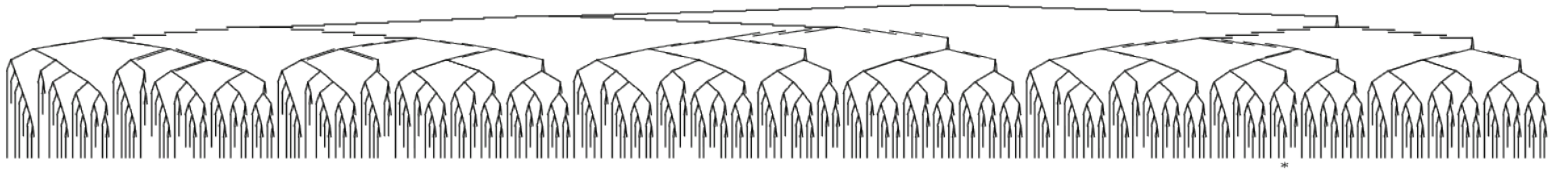
- https://youtu.be/RRO9-QXROss&t=2210 (Introduction)
- https://youtu.be/rwefoi__Fk4 (Uninformed Search: Breadth-First Search, Depth-First Search, Iterative Deepening)
- https://youtu.be/THZ3YxHAwno (Heuristic Search: Greedy Search, A*-Search, IDA*-Search)
- https://youtu.be/IW-HIOPqgsk (Games with Opponents,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

Introduction

- 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.

Example 1

Even in the proof of *a very simple formula* from [Ertel 1993] with *three Horn clauses*, *each with at most three literals*, the search tree for SLD resolution has the following shape:



- 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*.

Introduction

Example 1 (cont.)

- For realistic problems, the *branching factor* and *depth of the first solution* may become *significantly bigger*.
- Assume the branching factor is a constant equal to 30 and the first solution is at depth 50. The search tree then has $30^{50} \approx 7.2 \times 10^{73}$ leaf nodes.
  - *These assumptions are completely realistic*. In chess for example, there are over 30 possible moves for a typical situation, and a game lasting 50 half-turns is relatively short. (Each player has 25 moves.)
- But the *number of inference steps is even bigger* because not only every leaf node, but also every inner node of the tree corresponds to an inference step. Therefore we must add up the nodes over all levels and obtain the total number of nodes of the search tree

$$\sum_{d=0}^{50} 30^d = \frac{1 - 30^{51}}{1 - 30} = 7.4 \times 10^{73},$$

which does not change the node count by much. Evidently, *nearly all of the nodes of this search tree are on the last level*. As we will see, *this is generally the case*.

Example 1 (cont.)

- Assume we had 10,000 computers which can each perform a billion inferences per second, and that we could distribute the work over all of the computers with no cost.
- The total computation time for all 7.4×10^{73} inferences would be approximately

$$\frac{7.4 \times 10^{73} \text{ inferences}}{10000 \times 10^{9} \text{ inferences/sec}} = 7.4 \times 10^{60} \text{ sec}$$

$$\approx 2.3 \times 10^{53} \text{ years},$$

  which is about 10^{43} times as much time as the age of our universe.
- *There is no realistic chance of searching this kind of search space completely with the means available to us in this world*.

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 this kind of *heuristic search method* and additionally *describe processes with which computers can*, similarly to humans, *improve their heuristic search strategies by learning*.

Introduction

Example 2 (8-Puzzle [Nilsson 1998]; [Russell and Norvig 2010])

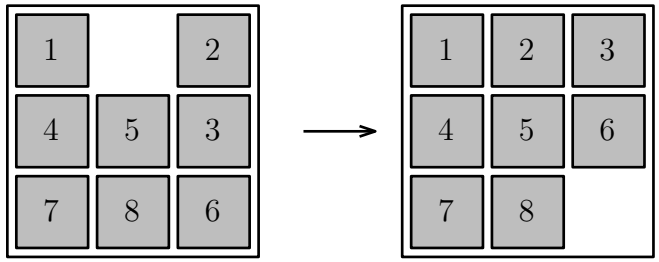


Figure: Possible starting and goal states of the 8-puzzle

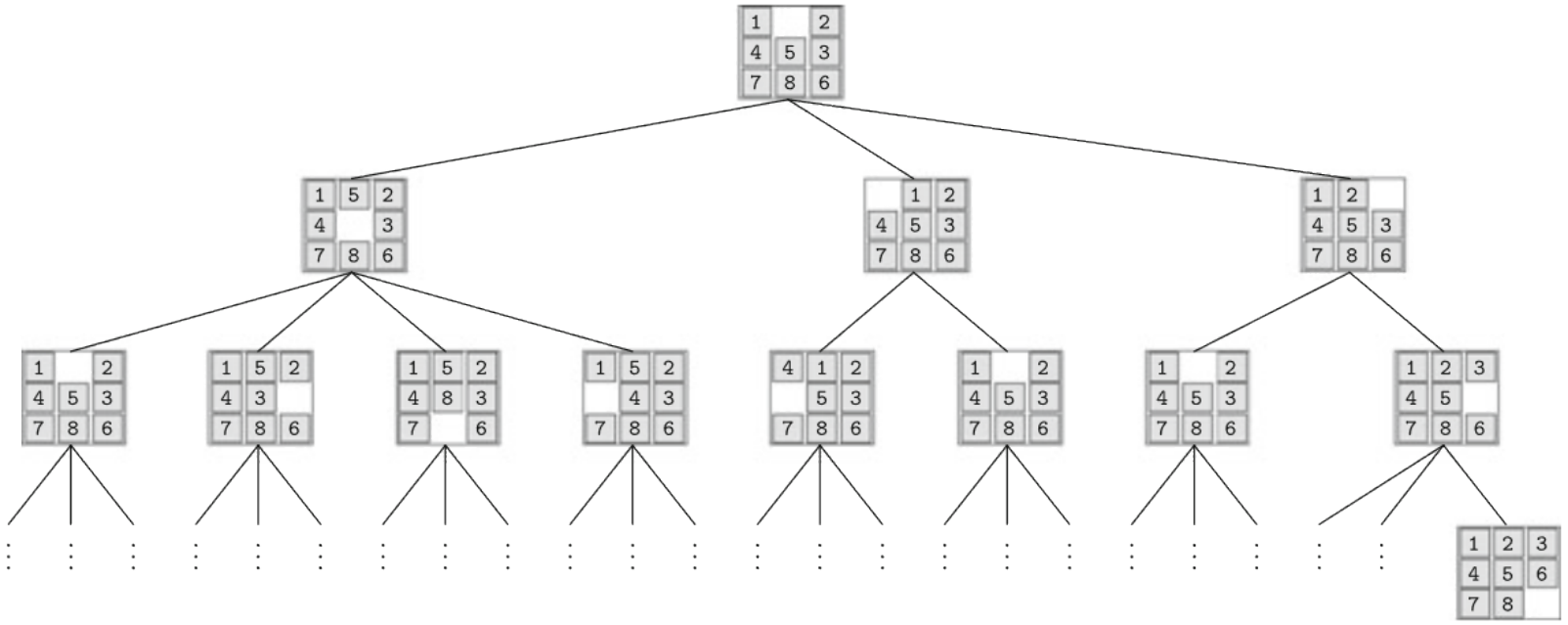


Figure: Search tree for the 8-puzzle. Bottom right a goal state in depth 3 is represented. To save space the other nodes at this level have been omitted

Example 2 (8-Puzzle, cont.)

- The *branching factor* of the search tree for 8-puzzle *alternates between two, three, and four*.
- The *average branching factor* of a tree is *the branching factor that a tree with a constant branching factor, equal depth, and an equal amount of leaf nodes* would have.
  - If we *consider only the subtrees induced by the first three levels* of the search tree. To calculate the *average branching factor* of this subtree, we take a tree T with constant branching factor b, depth 2, and 8 leaf nodes, and calculate b. Note that a tree T with constant branching factor b and depth d has b^d leaf nodes. Therefore, we have $b^2 = 8$ and thus $b = \sqrt{8} \approx 2.83$.
- We also observe that *each state is repeated multiple times two levels deeper* because *in a simple uninformed search, every action can be reversed in the next step*.

Example 2 (8-Puzzle, cont.)

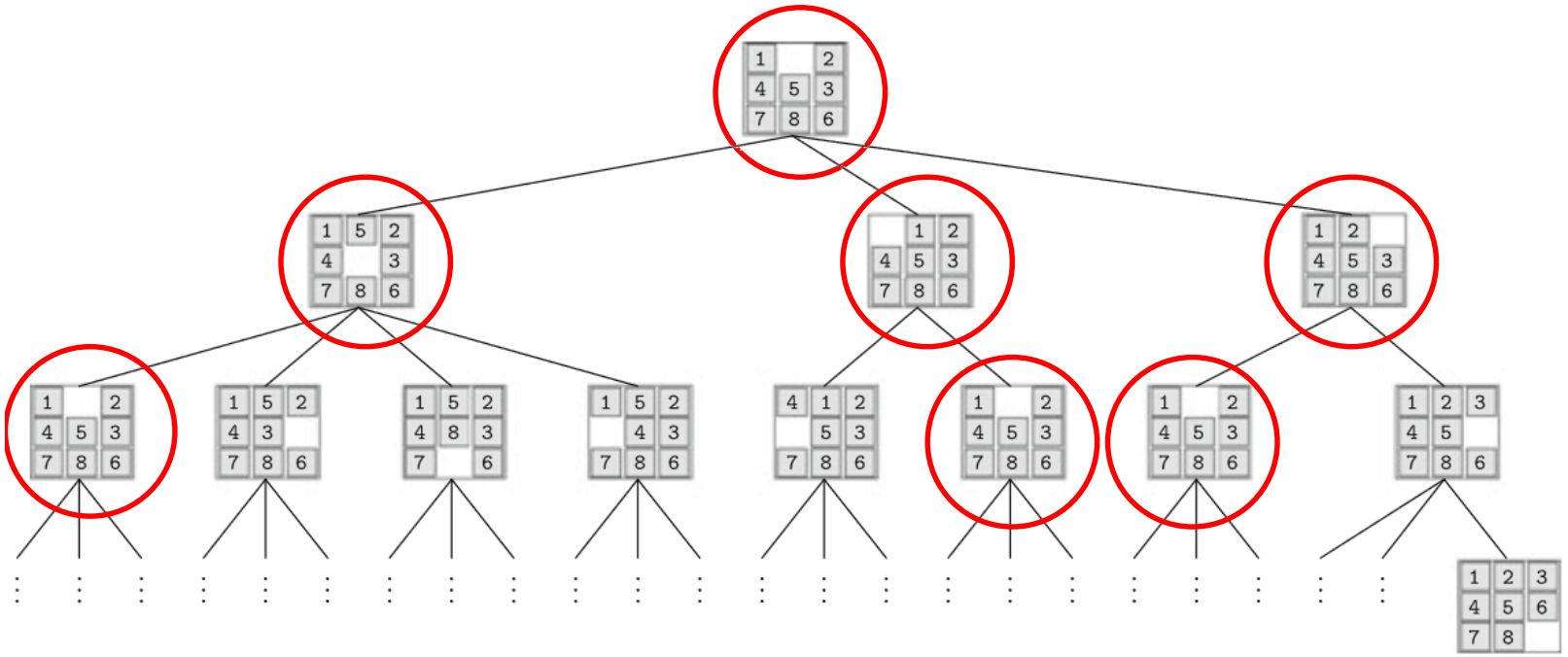


Figure: Cycles of length 2 in the search tree are somewhat redundant.

Example 2 (8-Puzzle, cont.)

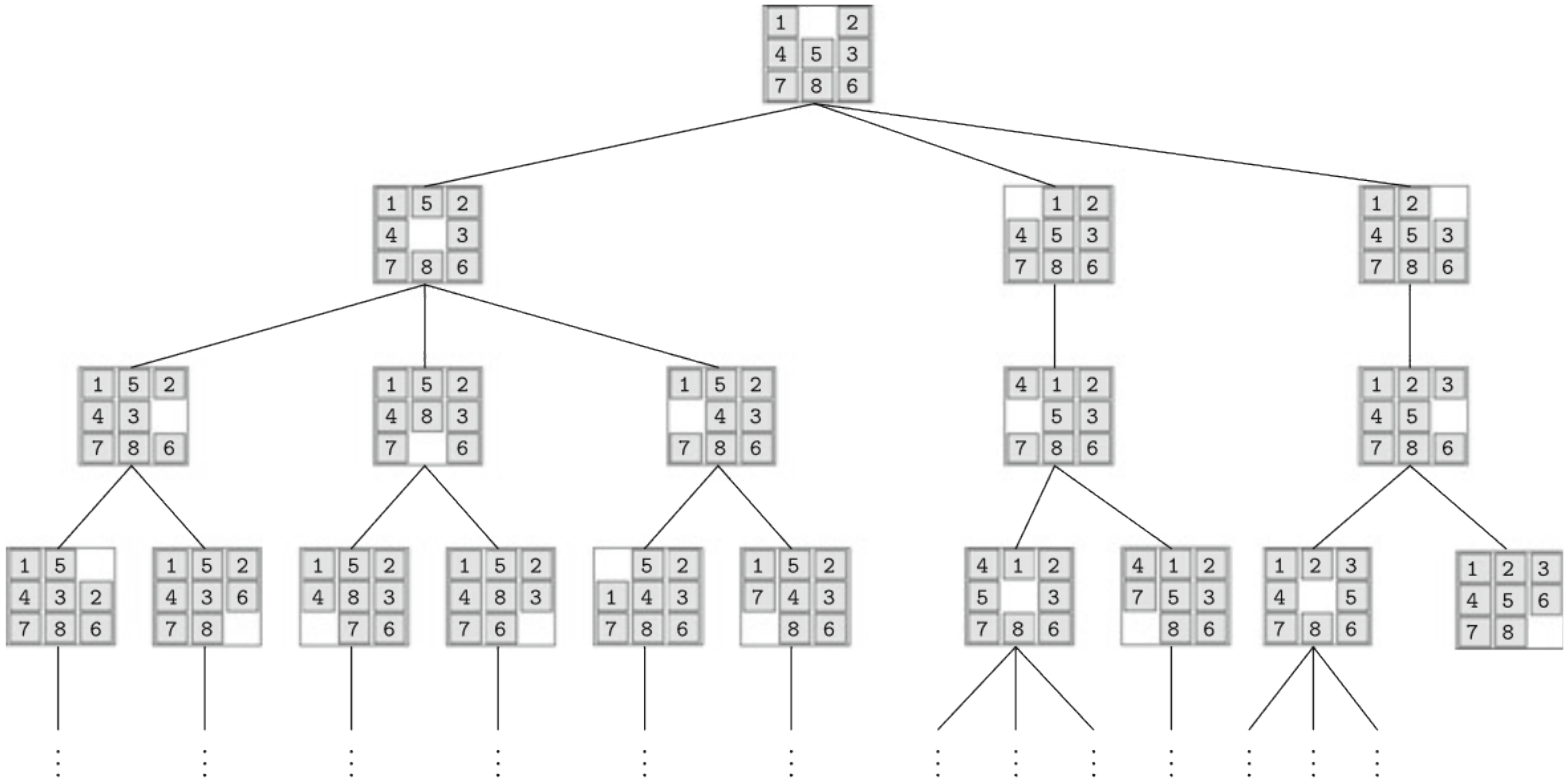


Figure: Search tree for an 8-puzzle without cycles of length 2.

Observe that *for any node excepting the root, the number of its children decreases by 1* in the search tree without cycles of length 2 $\Rightarrow$ *average branching factor* ≈ 1.8.

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 2 (8-Puzzle, cont.)

Apply the definition to the 8-puzzle, we get

State: Arrangement of tiles $1, \ldots, 8$ and one empty square in a 3×3 grid.

Starting state: The chosen initial arrangement.

Goal state: The chosen target arrangement.

Actions: Move the empty square one cell left, right, up, or down when legal.

Solution: A search-tree path from the starting state to the goal state.

Cost function: Cost 1 per legal move.

State-space graph: Arrangements linked by legal moves; two disconnected components make some start/goal pairs unsolvable.

Search tree: Rooted at the starting state; children apply legal moves, and states may reappear.

Introduction

For analysis of the search algorithms, the following terms are needed:

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.

Effective branching factor

The *effective branching factor* of a tree of depth d with N total nodes is defined as *the branching factor that a tree with constant branching factor, equal depth, and equal* N *would have*.

Complete Search Algorithms

A search algorithm is called *complete* if it *finds a solution for every solvable problem*. If a complete search algorithm terminates without finding a solution, then the problem is unsolvable.

Introduction

- A tree with constant branching factor b and depth d has total

$$n = \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1}$$

nodes.

Theorem 1

For heavily branching finite search trees with a large constant branching factor, almost all nodes are on the last level.

Exercise 1 ([Ertel 2025], Exercise 6.1, p. 124)

(a) Prove Theorem 1, which says that for a tree with large constant branching factor b, almost all nodes are on the last level at depth d.

(b) Show that this is not always true when the effective branching factor is large and not constant.

Exercise 2 ([Ertel 2025], Exercise 6.2, p. 124)

(a) Calculate the average branching factor for the 8-puzzle without a check for cycles.

(b) Calculate the average branching factor for the 8-puzzle for uninformed search while avoiding cycles of length 2.

Exercise 3 ([Ertel 2025], Exercise 6.3, p. 125)

(a) What is the difference between the average and the effective branching factor?

(b) Why is the effective branching factor better suited to analysis and comparison of the computation time of search algorithms than the average branching factor?

(c) Show that for a heavily branching tree with N nodes and depth d the effective branching factor $\overline{b}$ is approximately equal to the average branching factor and thus equal to $\sqrt[d]{N}$.

Exercise 4 ([Ertel 2025], Exercise 6.4, p. 125)

(a) Calculate the size of the state space for the 8-puzzle, for the analogous 3-puzzle (2×2-matrix), as well as for the 15-puzzle (4×4-matrix).

(b) Prove that the state graph consisting of the states (vertices) and the actions (edges) for the 3-puzzle falls into two connected subgraphs, between which there are no connections.

Introduction

Example 3 (Shortest Path from City A to City B)

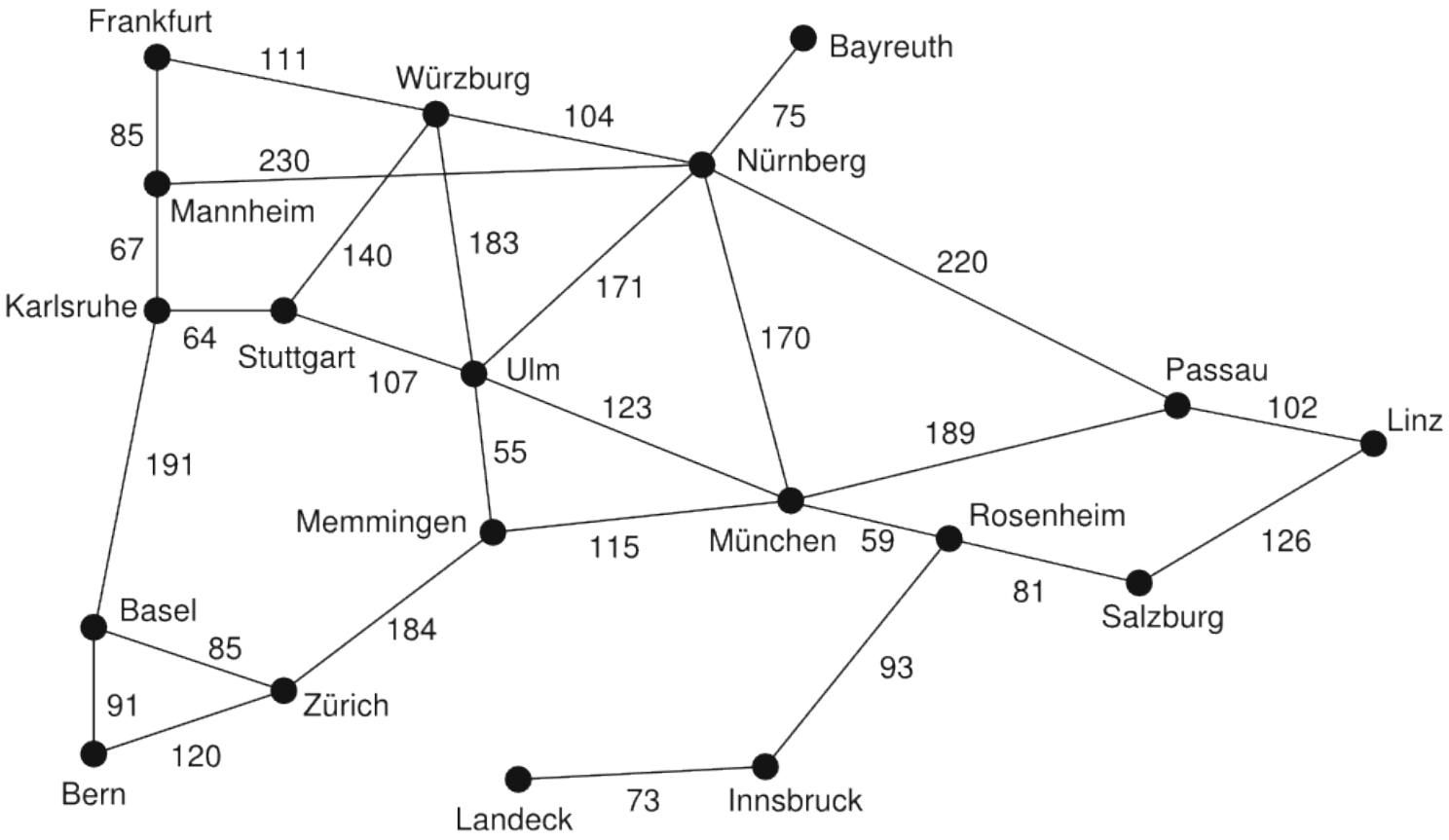


Figure: The graph of southern Germany as an example of a search task with a cost function.

Example 3 (cont.)

State: Current city.
Starting state: City A.
Goal state: City B.
Actions: Travel to a neighboring city.
Solution: A search-tree path from city A to city B.
Cost function: Road distance for each travel action.
State-space graph: Cities linked by weighted road edges.
Search tree: Rooted at city A; children follow road edges, and a city may reappear along another route.

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 4

- The 8-puzzle problem is deterministic and observable.
- In route planning in real applications, both characteristics are not always given.
  - 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

- We want to *ignore all kinds of complications* similar to those in the route planning problems. Therefore, we will *only look at problems that are deterministic and observable*.
- For deterministic and observable problems such as the 8-puzzle, a *usable abstract model* is available. Therefore, one can *find an action sequence for the solution without actually carrying out the actions in the real world*. $\Rightarrow$ 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

BREADTHFIRSTSEARCH(NodeList, Goal)

```
1  NewNodes = ∅
2  for all Node ∈ NodeList
3        if GOALREACHED(Node, Goal)
4              return ("Solution found", Node)
5        NewNodes = APPEND(NewNodes,
         SUCCESSORS(Node))
6  if NewNodes ≠ ∅
7        return BREADTHFIRSTSEARCH(NewNodes, Goal)
8  else
9        return ("No solution")
```

- GOALREACHED(Node, Goal) checks whether the state $\mathrm{state}(\mathrm{Node})$ satisfies Goal.
- SUCCESSORS(Node) returns the child search nodes generated from the successor states of $\mathrm{state}(\mathrm{Node})$.
- The search tree is explored level by level until a solution is found.
  - Every node in the node list is tested for whether it is a goal node. If so, the program is stopped.
  - Otherwise all successors of the node are generated.
  - The search is then continued recursively on the list of all newly generated nodes.
  - The whole thing repeats until no more successors are generated.

Uninformed Search
Breadth-First Search

Analysis:

- complete if each node has only finitely many successors: BFS then reaches every depth in finite time.
- optimal if all action costs are equal; otherwise, BFS may not return a lowest-cost solution.
- Computation time =

$$c \cdot \sum_{i=0}^{d} b^i = \frac{b^{d+1} - 1}{b - 1} = O(b^d).$$

- Memory space = $O(b^d)$.

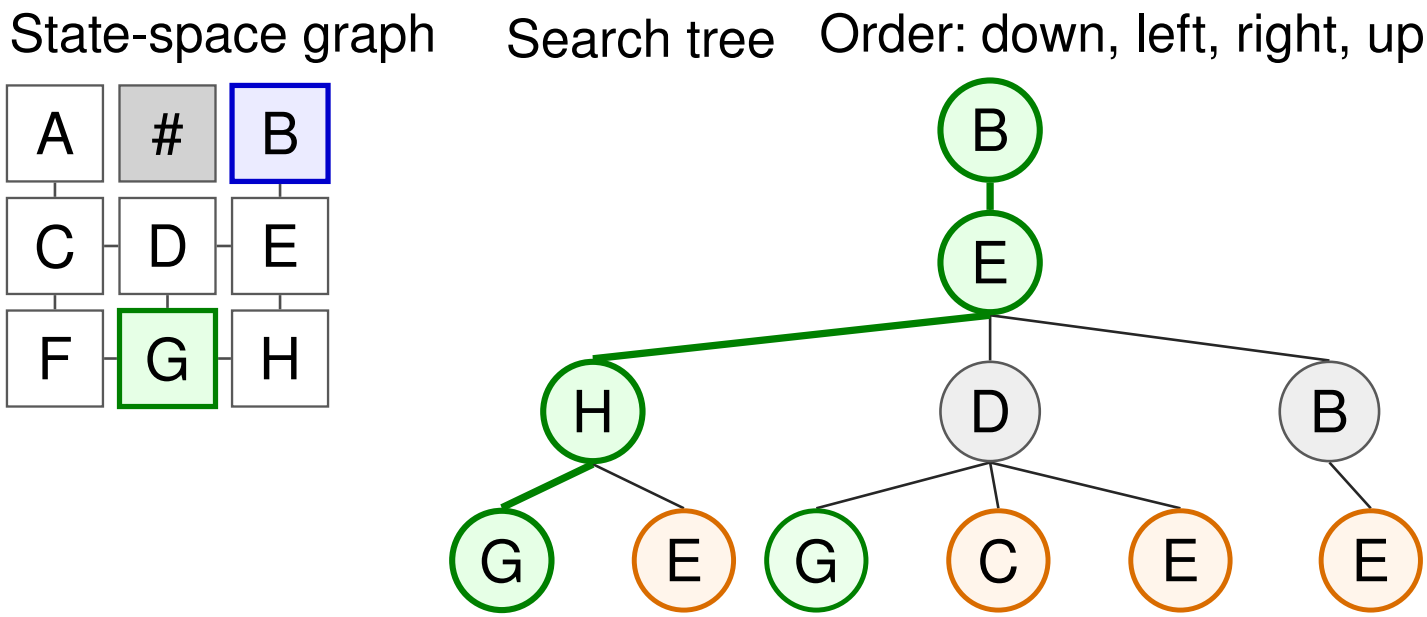
When action costs are unequal, use:

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

Simplified Sliding-Block Puzzle Solved by BFS



NodeList	G	E	G	C	E	E

NewNodes	goal found; NewNodes is not used

Following parent links gives $B \to E \to H \to G$, with cost 3. The G generated from D is never tested.

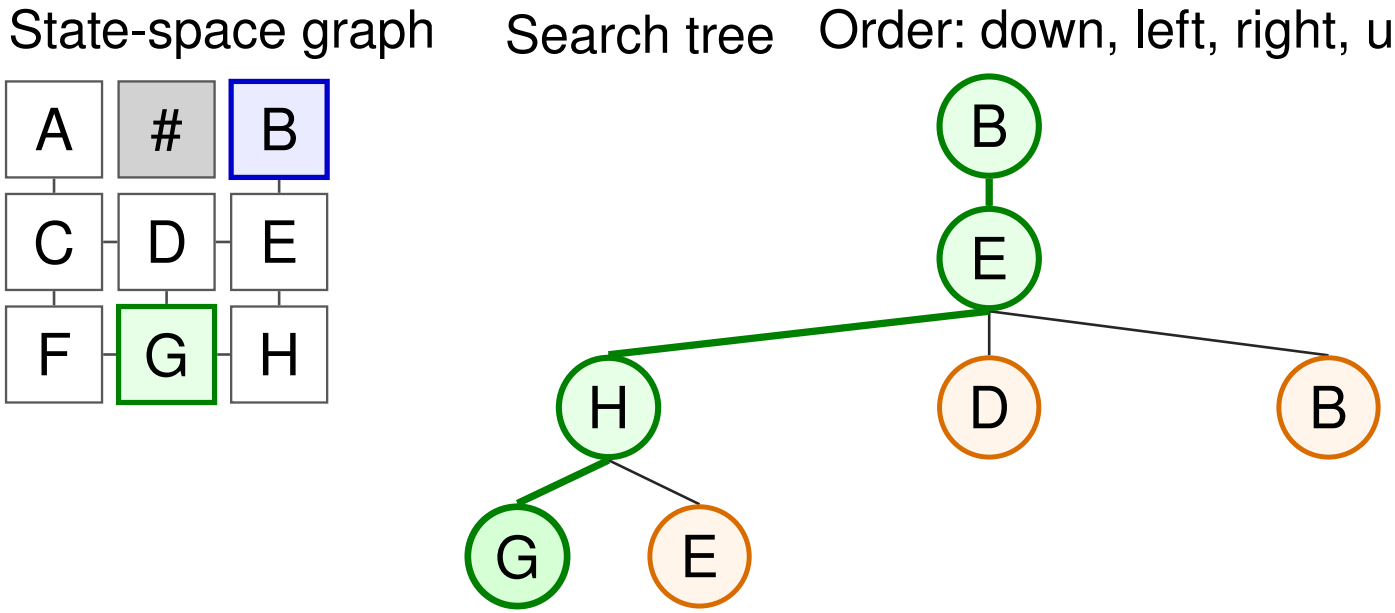
Uninformed Search
Depth-First Search


```
DepthFirstSearch(Node, Goal)
1   if GoalReached(Node, Goal)
2         return ("Solution found")
3   NewNodes = Successors(Node)
4   while NewNodes ≠ ∅
5         Result = DepthFirstSearch(First(NewNodes), Goal)
6         if Result = "Solution found"
7               return ("Solution found")
8         NewNodes = Rest(NewNodes)
9   return ("No solution")
```

- The function First returns the first element of a list, and Rest the rest of the list.
- After the expansion of a node only its successors are saved, and the first successor node is immediately expanded.
- The search quickly becomes very deep. Only when a node has no successors and the search fails at that depth is the next open node expanded via *backtracking* to the last branch, and so on.

Uninformed Search

Simplified Sliding-Block Puzzle Solved by DFS



Current path B E H G

SUCCESSORS GOALREACHED(G, G) succeeds

The call DEPTHFIRSTSEARCH(G, G) finds that GOALREACHED(G, G) succeeds and returns “Solution found.” The recursive call chain gives the path $B \to E \to H \to G$, with cost 3.

Analysis:

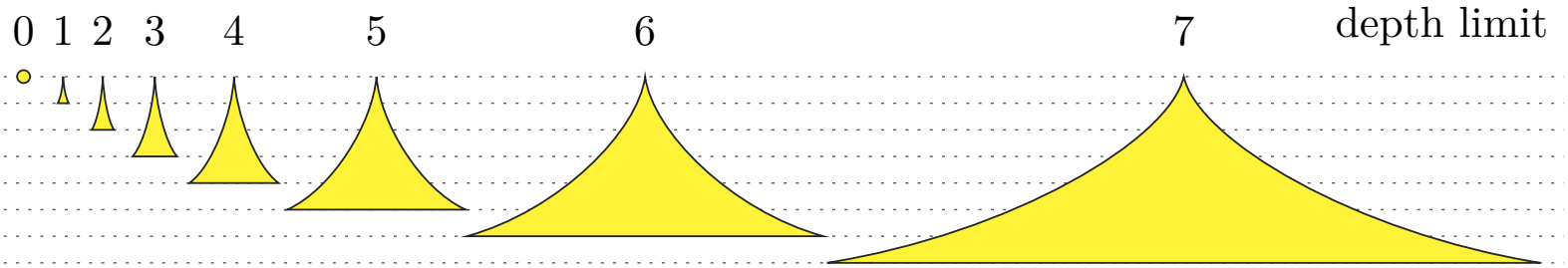
- incomplete (may follow an infinite branch forever).
  - For the Simplified Sliding-Block Puzzle (start B, goal G), move order left, right, up, down and no repeated-state check make DFS follow $B \to E \to D \to C \to D \to C \to \cdots$.
  - Yet $B \to E \to D \to G$ is a solution.
- not optimal (the first solution found may be deeper or more costly than another solution).
- Computation time = $O(b^{d_s})$ for a finite search tree, where d_s is the tree's maximum depth.
- Memory requirement = $O(bd_s)$; 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.

Iterative Deepening

We begin the depth-first search with a depth limit of 0. If no solution is found, we raise the limit by 1 and start searching from the beginning, and so on.



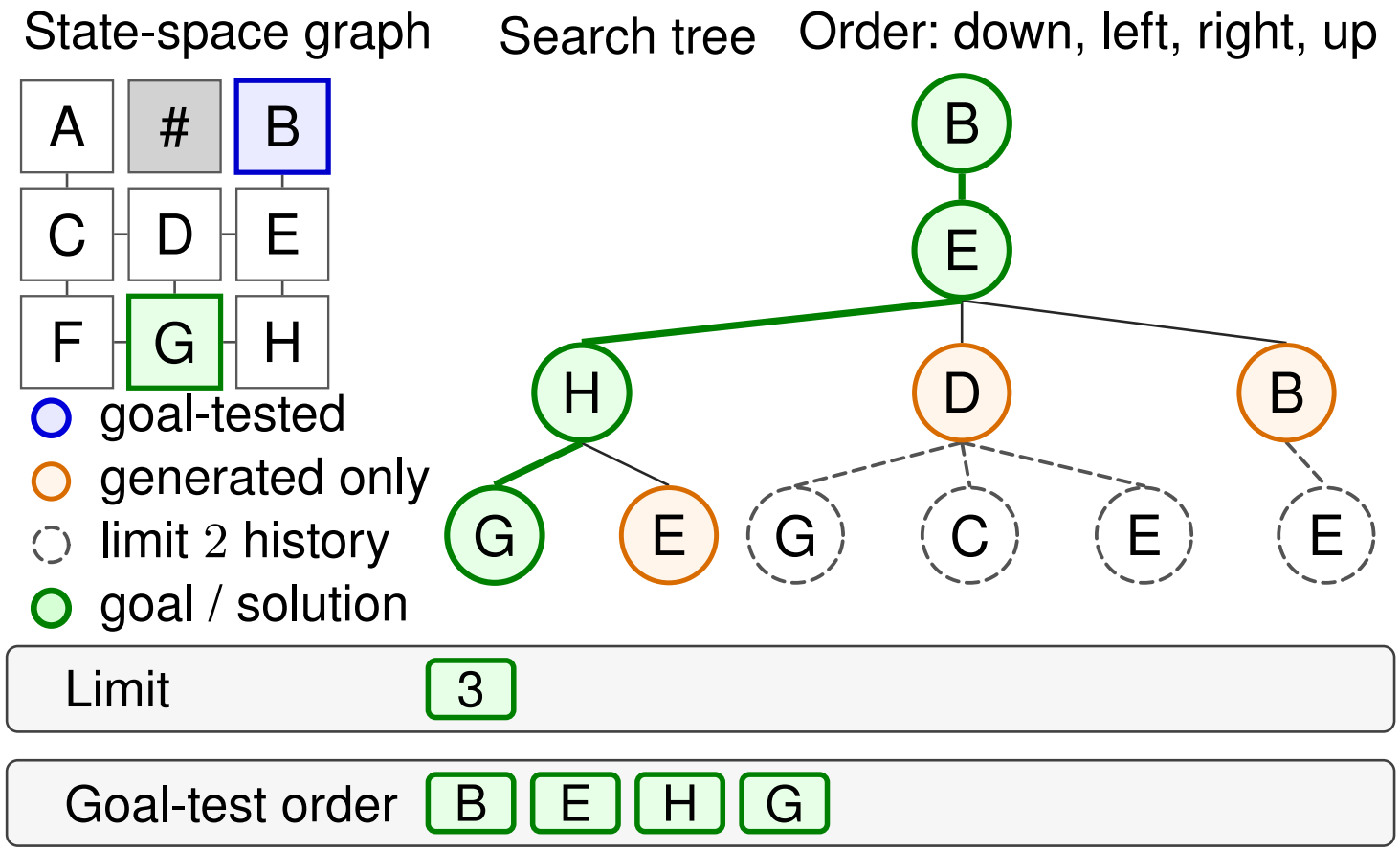
Uninformed Search
Iterative Deepening


```
ITERATIVEDEEPENING(Node, Goal)
1   DepthLimit = 0
2   repeat
3       Result = DEPTHFIRSTSEARCH-B(Node, Goal, 0, DepthLimit)
4       DepthLimit = DepthLimit + 1
5   until Result = "Solution found"

DEPTHFIRSTSEARCH-B(Node, Goal, Depth, Limit)
1   if GOALREACHED(Node, Goal)
2       return ("Solution found")
3   NewNodes = SUCCESSORS(Node)
4   while NewNodes ≠ ∅ and Depth < Limit
5       Result =
        DEPTHFIRSTSEARCH-B(FIRST(NewNodes),
        Goal, Depth+1, Limit)
6       if Result = "Solution found"
7           return ("Solution found")
8       NewNodes = REST(NewNodes)
9   return ("No solution")
```

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:

- complete.
- optimal, if costs are constant and increment = 1
- Computation time = $O(b^d)$
- Memory requirement = $O(bd)$.
- One could argue that *repeatedly re-starting depth-first search at depth zero causes a lot of redundant work*. *For large branching factors this is not the case*. Indeed, *the computation time for all iterations besides the last can be ignored*.

Uninformed Search
Iterative Deepening

The sum of the number of nodes of all depths up to the one before last $d_{\max} - 1$ in all trees searched is much smaller than the number of nodes in the last tree searched.

- Let $N_b(d)$ be the number of nodes of a search tree with branching factor b and depth d and $d_{\max}$ be the last depth searched.
- The last search tree has $N_b(d_{\max}) = \sum_{i=0}^{d_{\max}} b^i = \frac{b^{d_{\max}+1} - 1}{b - 1}$ nodes.
- All trees searched beforehand together have

$$\sum_{d=1}^{d_{\max}-1} N_b(d) = \sum_{d=1}^{d_{\max}-1} \frac{b^{d+1} - 1}{b - 1} = \frac{1}{b-1}\left(\left(\sum_{d=1}^{d_{\max}-1} b^{d+1}\right) - d_{\max} + 1\right)$$

$$= \frac{1}{b-1}\left(\frac{b^{d_{\max}+1} - 1}{b-1} - 1 - b - d_{\max} + 1\right)$$

$$\approx \frac{1}{b-1}\left(\frac{b^{d_{\max}+1} - 1}{b-1}\right) = \frac{1}{b-1} N_b(d_{\max})$$

nodes. For $b > 2$, this is less than $N_b(d_{\max})$.

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	$b^{1+\lfloor C^*/\epsilon \rfloor}$	∞ or b^{d_s}	b^d
Memory use	b^d	$b^{1+\lfloor C^*/\epsilon \rfloor}$	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.
- C^*: lowest solution cost; 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.

Uninformed Search
Cycle Check

Question

How would a check on *cycles of arbitrary length* affect *search performance*?

- Store the *predecessor states on the current path* for each node, for example with a linked list.
- At depth i, there are b^i generated nodes; each is compared with its i path predecessors.
- If generating one node costs c_1 and one predecessor comparison costs c_2, the *total work through depth* d is

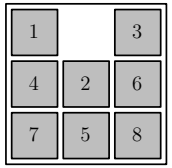
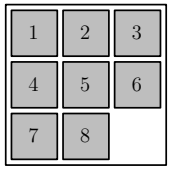
$$\approx c_2 \cdot d \cdot b^d$$

$$\underbrace{c_1 \cdot \sum_{i=0}^{d} b^i}_{\text{cost of generating nodes}} + \underbrace{c_2 \cdot \sum_{i=0}^{d} i \cdot b^i}_{\text{cost of the cycle check}}$$

Since $\sum_{i=0}^{d} i\, b^i$ is dominated by $d\, b^d$, the full cycle check adds about a *factor of* d, not a new exponential factor.

Uninformed Search
Exercises

Exercise 5 ([Ertel 2025], Exercise 6.5, p. 125)

With breadth-first search for the 8-puzzle, find a path (manually) from the starting state  to the goal state  .

Exercise 6 ([Ertel 2025], Exercise 6.6, p. 125)

(a) Program breadth-first search, depth-first search, and iterative deepening in the language of your choice and test them on the 8-puzzle example.

(b) Why does it make little sense to use depth-first search on the 8-puzzle?

Exercise 7 ([Ertel 2025], Exercise 6.7, p. 125)

(a) Show that breadth-first search given constant cost for all actions is guaranteed to find the shortest solution.

(b) Show that this is not the case for varying costs.

Exercise 8 ([Ertel 2025], Exercise 6.8, p. 125)

The predecessors of all nodes must be stored to check for cycles during depth-first search.

(a) For depth-first search develop a data structure (not a hash table) that is as efficient as possible for storing all nodes in the search path of a search tree.

(b) For constant branching factor b and depth d, give a formula for the storage space needed by depth-first search with and without storing predecessors.

(c) Show that for large b and d, we have $\sum_{k=0}^{d} k \cdot b^k \approx d \cdot b^d$.

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

HEURISTICSEARCH(Start, Goal)

```
1   NodeList = [Start]
2   while True
3       if NodeList = ∅
4           return ("No solution")
5       Node = First(NodeList)
6       NodeList = Rest(NodeList)
7       if GoalReached(Node, Goal)
8           return ("Solution found", Node)
9       NodeList = SortIn(Successors(Node), NodeList)
```

With appropriate evaluation functions, one can generate BFS and DFS from HEURISTICSEARCH

- SortIn(X,Y) inserts the elements from the unsorted list X into the ascendingly sorted list Y.
  - *The evaluation $f(n)$ is used as the sorting key*. Thus the node with the lowest evaluation is *always at the beginning of the list*.
  - When inserting a new node, the implementation may check whether NodeList already contains a node representing the same state and avoid storing a redundant duplicate.

Exercise 9 ([Ertel 2025], Exercise 6.13, p. 126)

Give an evaluation function for search nodes with which HEURISTICSEARCH can be implemented as depth-first search, and one for a breadth-first search implementation.

Heuristic Search

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

Question

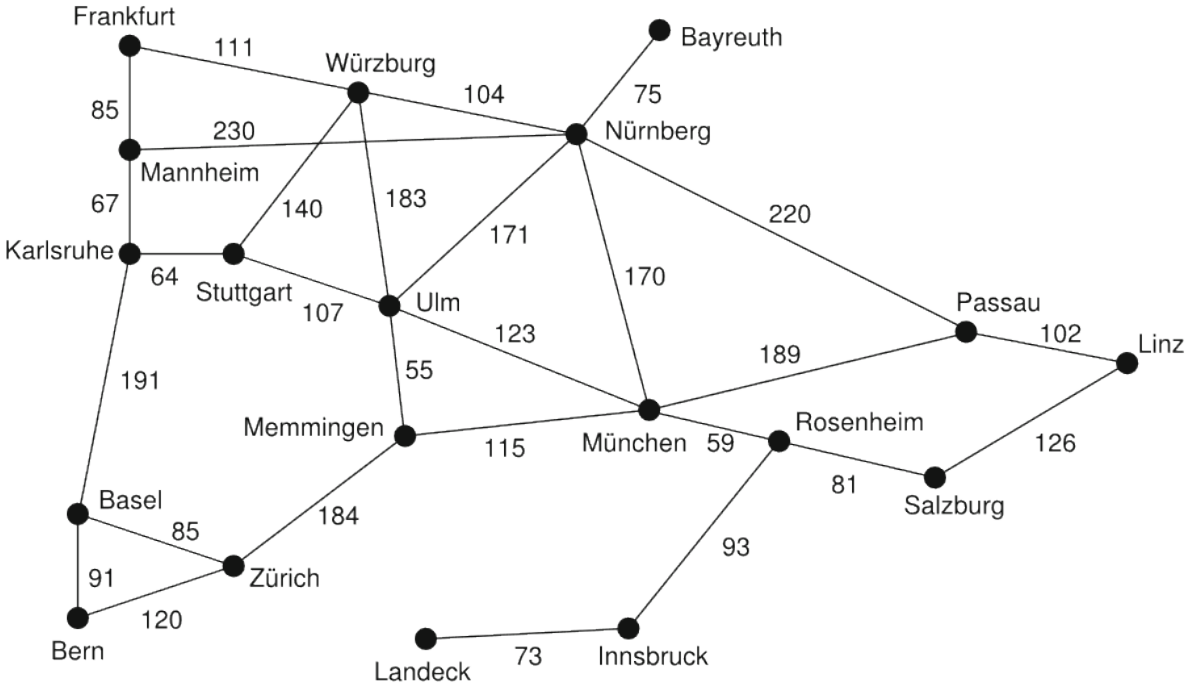
How do we find a heuristic that is fast and simple to compute?

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 5 (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: City graph with flying distances from all cities to Ulm

Example 6 (cont.)

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

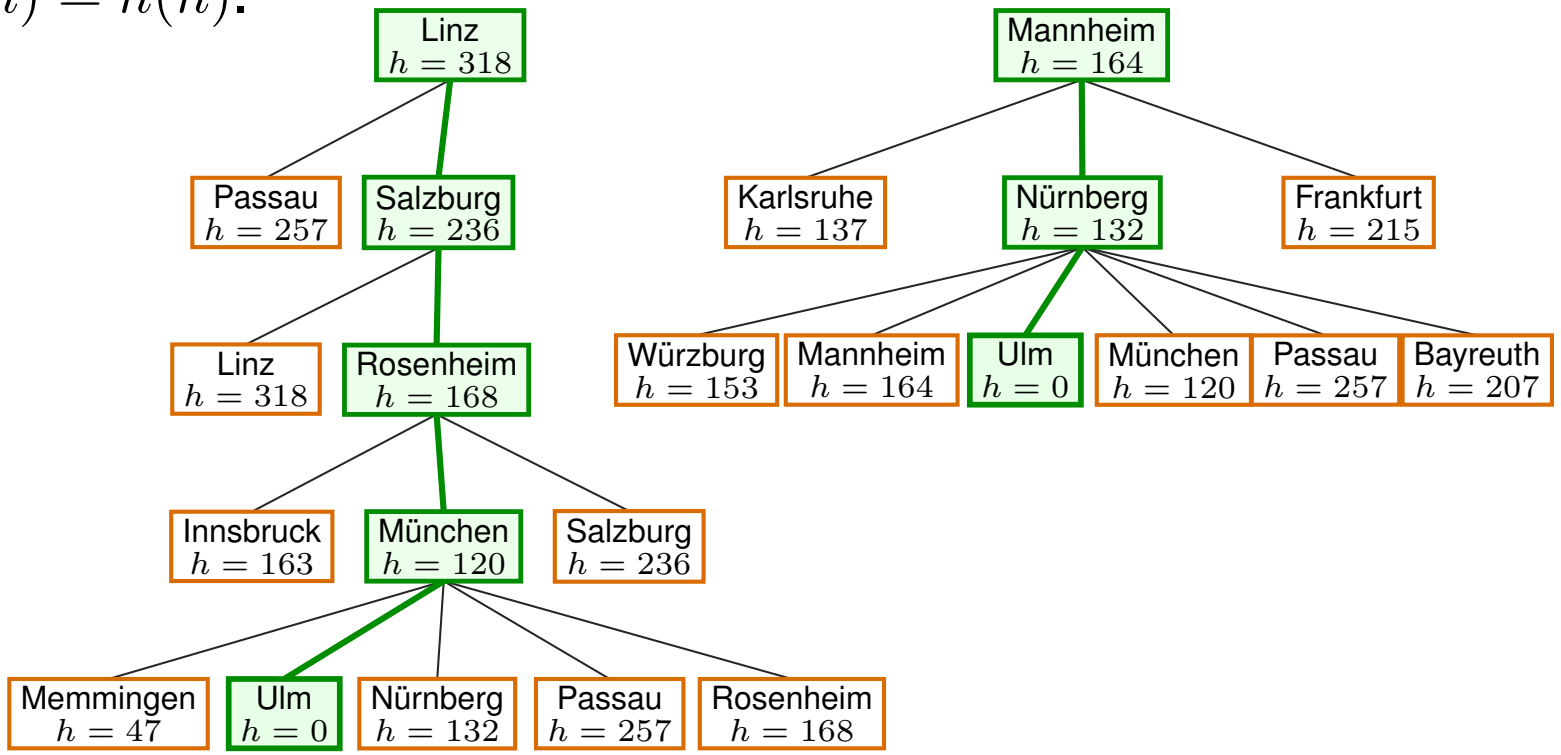


Figure: Start from Linz. Greedy search uses $f(n) = h(n)$ and keeps NODELIST sorted by h.Expand Linz. NODELIST: Salzburg(236), Passau(257); select Salzburg.Expand Salzburg. NODELIST: Rosenheim(168), Passau(257), Linz(318); select Rosenheim.Expand Rosenheim. Its successors include München(120), Innsbruck(163),

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

Exercise 10 ([Ertel 2025], Exercise 6.14, p. 126)

What is the relationship between the picture of the couple at the canyon below and admissible heuristics?

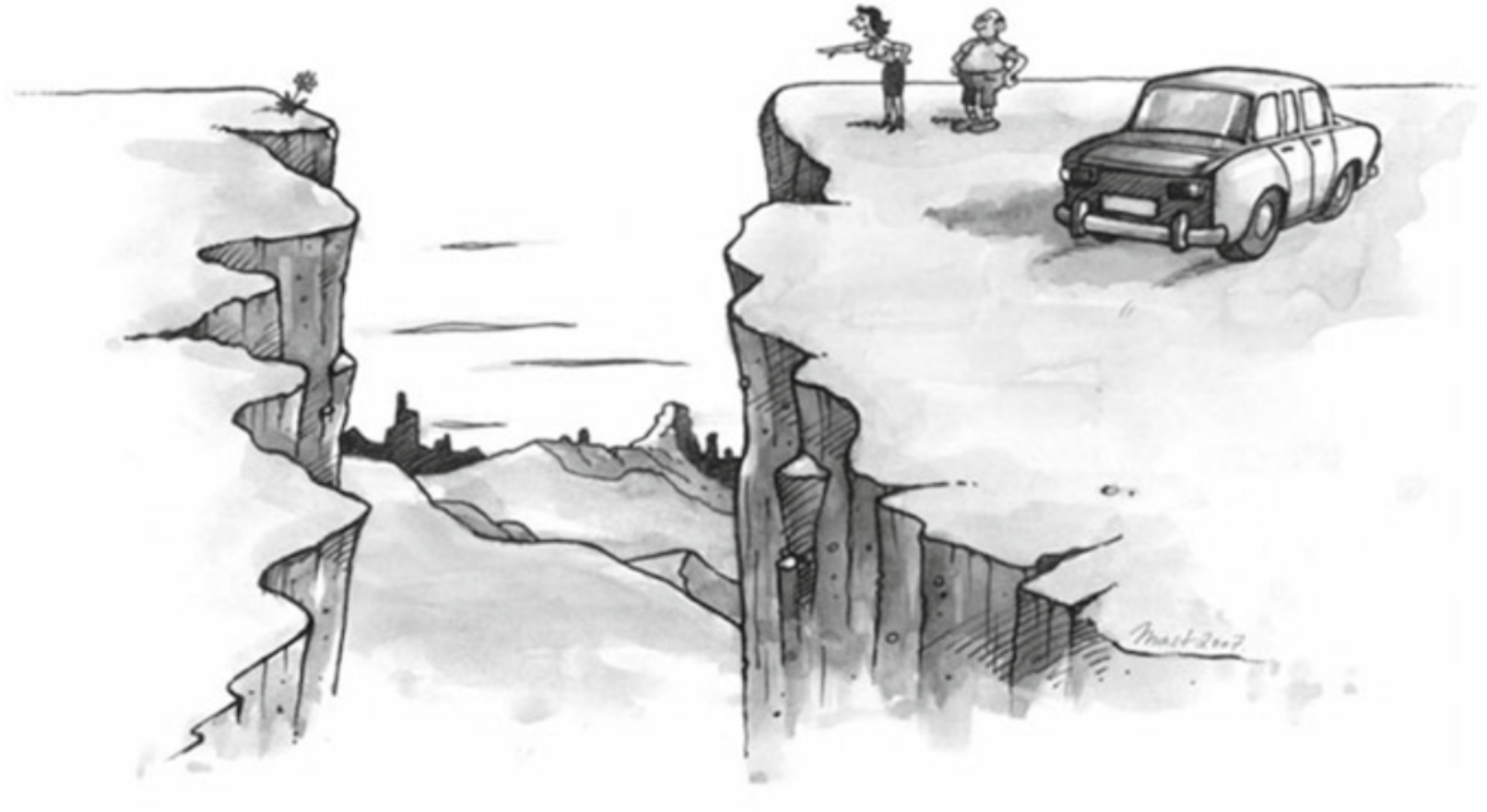


Figure: He: “Dear, think of the fuel cost! I’ll pluck one for you somewhere else.” She: “No, I want that one over there!”

Example 6 (cont.)

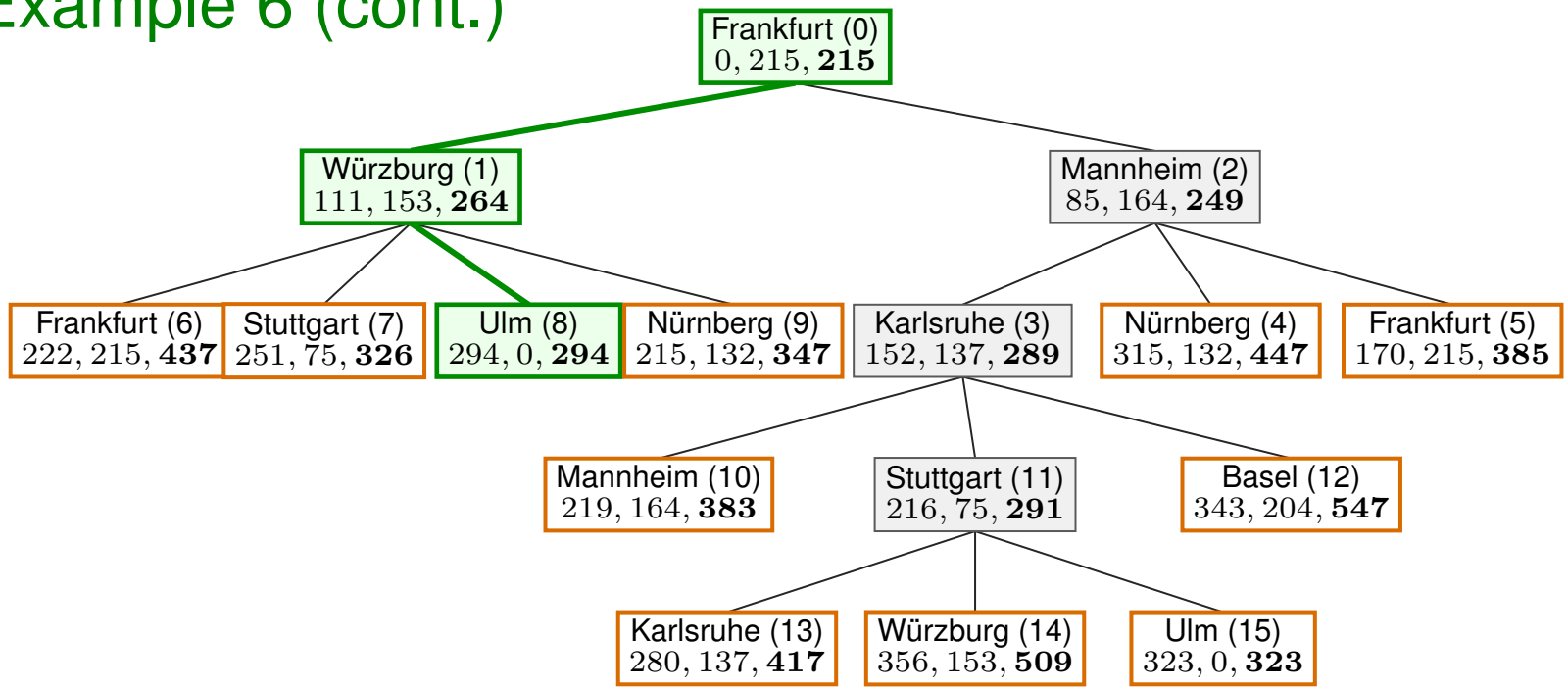


Figure: Start at Frankfurt. Each node shows city (generation number); below it are $g, h, \mathbf{f}$.Expand Frankfurt. NODELIST: Mannheim(249), Würzburg(264); select Mannheim.Expand Mannheim. NODELIST: Würzburg(264), Karlsruhe(289), Frankfurt(385), Nürnberg(447); select Würzburg.Expand Würzburg. Ulm(8) 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 2 (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
A*-Search: Comparison with Ertel

- In [Ertel 2025], Thm. 6.2, p. 111, the result is stated as: “The A* algorithm is *optimal*. That is, it always finds the solution with the *lowest total cost* if the heuristic h is *admissible*.”
- **The guarantee has two parts.** The preceding theorem states them separately: *completeness* means that A* *eventually selects a goal*, and *cost optimality* means that the *selected goal represents a minimum-cost solution*.
- **The assumptions are explicit.** Following Poole and Mackworth, the preceding theorem states *bounded branching*, *action costs bounded away from zero*, and an *admissible heuristic*. Ertel’s Theorem 6.2 *does not list the first two conditions explicitly*.

Proof, Part 1.

Let C^* be the lowest solution cost and fix an optimal path. Until A* selects a goal, that path has a *first unexpanded node* n in NODELIST:

root $\longrightarrow \cdots \longrightarrow n$ in NODELIST $\longrightarrow \cdots \longrightarrow$ optimal goal (C^*).

Admissibility says that $h(n)$ is no greater than the remaining cost along this path. Therefore,

$$f(n) = g(n) + h(n) \leq g(n) + \text{remaining path cost} = C^*.$$

If m is the next node selected by A*, its f-value is lowest in NODELIST. Hence

$$f(m) \leq f(n) \leq C^*.$$

Proof, Part 1, continued.

Before the first goal is selected, Part 1 (1/2) shows that every selected node m satisfies $f(m) \leq C^*$. If m is at depth d, then

$$\underbrace{d\epsilon \leq g(m)}_{\substack{d \text{ actions, each costing at least } \epsilon}} \leq \underbrace{f(m)}_{\substack{g(m)+h(m),\ h(m)\geq 0}} \leq C^*.$$

Thus every such node has bounded depth:

$$d \leq C^*/\epsilon.$$

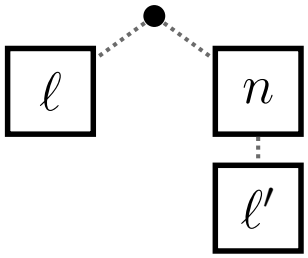
finitely many depths + **finite branching** $\Longrightarrow$ *finitely many nodes with* $f \leq C^*$.

A* selects one of these nodes at each iteration. It therefore cannot continue forever without selecting a goal. Hence A* is *complete*.

Proof, Part 2.

If the root is a goal, its zero-cost path is optimal. Otherwise, by Part 1, a first selected goal exists; call it ℓ. For any other goal node ℓ', let n be the *first unexpanded node* on the root-to-ℓ' path. Its parent has already been expanded, so n is open and remains in NODELIST.

Goal node



$$f(\ell) \le f(n)$$

$$f(n) \le g(\ell')$$

$$\Rightarrow g(\ell) \le g(\ell')$$

Definition of f: path cost to a node + estimated cost from its state to a goal

$$g(\ell) = g(\ell) + h(\ell) = f(\ell) \le f(n) = g(n) + h(n) \le g(\ell')$$

$\text{state}(\ell)$ is a goal state, so $h(\ell) = 0$

open nodes are sorted by increasing f

admissibility: $h(n) \le$ remaining cost on the path from n to ℓ'

- **Simple heuristic:** air line distance (= straight-line distance)
- **Better:** Landmarks (e.g., see [Batzill 2016]) (shrinking the search space more)
  - 5 to 60 landmark cities.
  - Preprocessing: For all landmarks the shortest paths to all cities are stored.
  - Let p be a landmark city, n the current search node, z the goal city, and $c^*(x, y)$ the cost of the shortest path from city x to city y.
    Triangle inequality:
    $c^*(\text{state}(n), p) \leq c^*(\text{state}(n), z) + c^*(z, p)$.
    Solving for $c^*(\text{state}(n), z)$ yields an admissible heuristic h:

$$h(n) = c^*(\text{state}(n), p) - c^*(z, p) \leq c^*(\text{state}(n), z).$$

Heuristic Search

Route Planning with the A* Search Algorithm

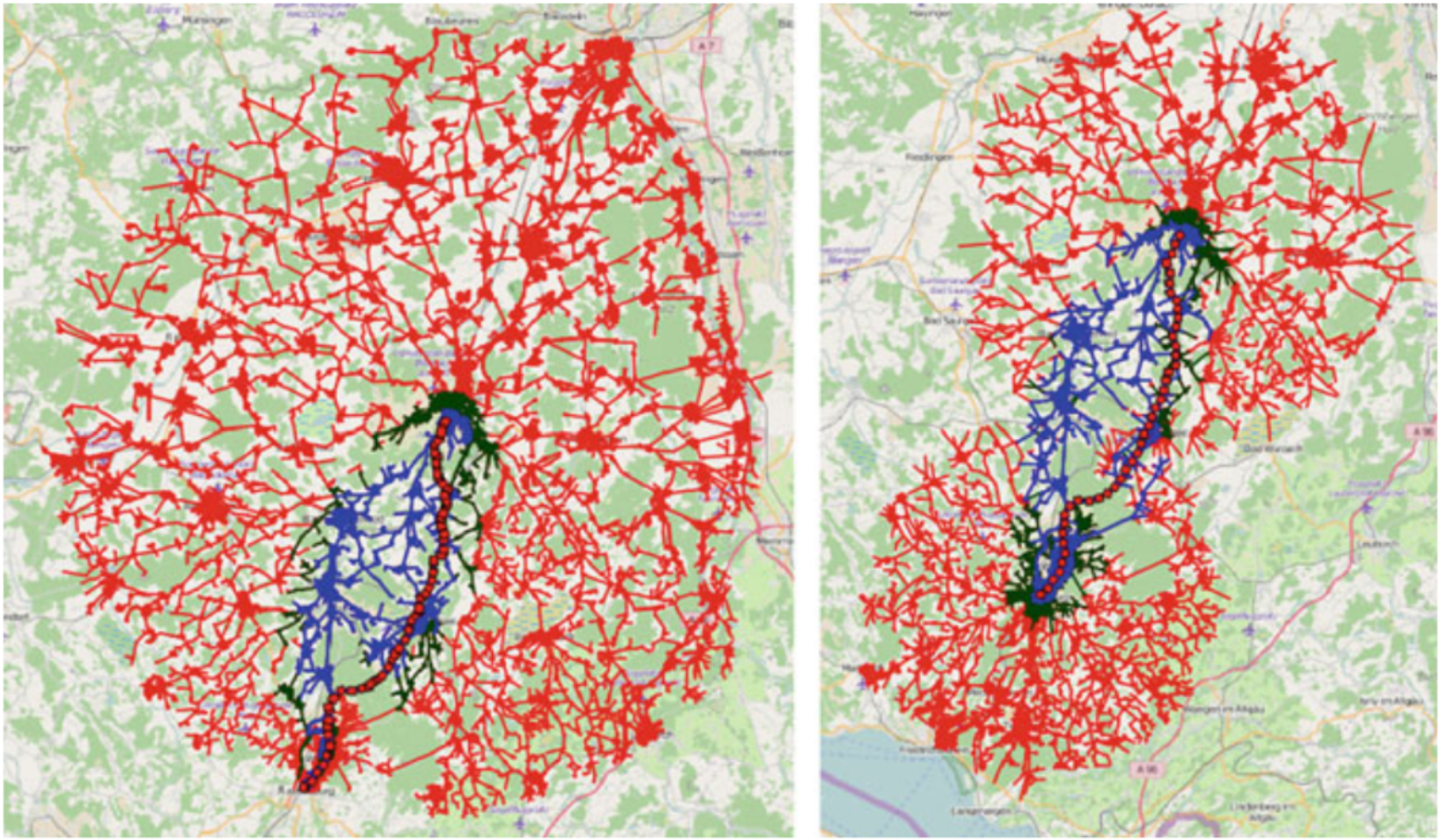


Figure: The search tree of A* search for planning a route from Ravensburg to Biberach with: no heuristic (red), air line distance (dark green), landmark heuristic (blue). **Left:** normal search, **Right:** bidirectional search (plan in parallel a route from Ravensburg to Biberach and from Biberach to Ravensburg. If the routes meet, given certain conditions of the heuristic, an optimal route has been found.)

Heuristic Search

Route Planning with the A* Search Algorithm

	Unidirectional		Bidirectional	
	Tree Size [nodes]	Comp. time [msec.]	Tree Size [nodes]	Comp. time [msec.]
No heuristic	62000	192	41850	122
Straight-line distance	9380	86	12193	84
Landmark heuristic	5260	16	7290	16

Figure: Comparison of search tree size and computation time for route planning with and without each of the two heuristics. The landmark heuristic is the clear winner.

- Both heuristics clearly reduce the search space.
- In the case of the landmark heuristic, we see the computation time and the size of the search space reduced by a factor of about 12. The cost of computing the heuristic is thus insignificant.
- The straight-line distance, however, results in a search space reduction of a factor of 6.6, but only an improvement of a factor of 2.2 in run time due to the overhead of computing the euclidean distance.
- In the case of bidirectional search, in contrast to unidirectional search, we see a significant reduction of the search space even without heuristic.

Optimization of time, not distance $\Rightarrow$ adjust the heuristic accordingly

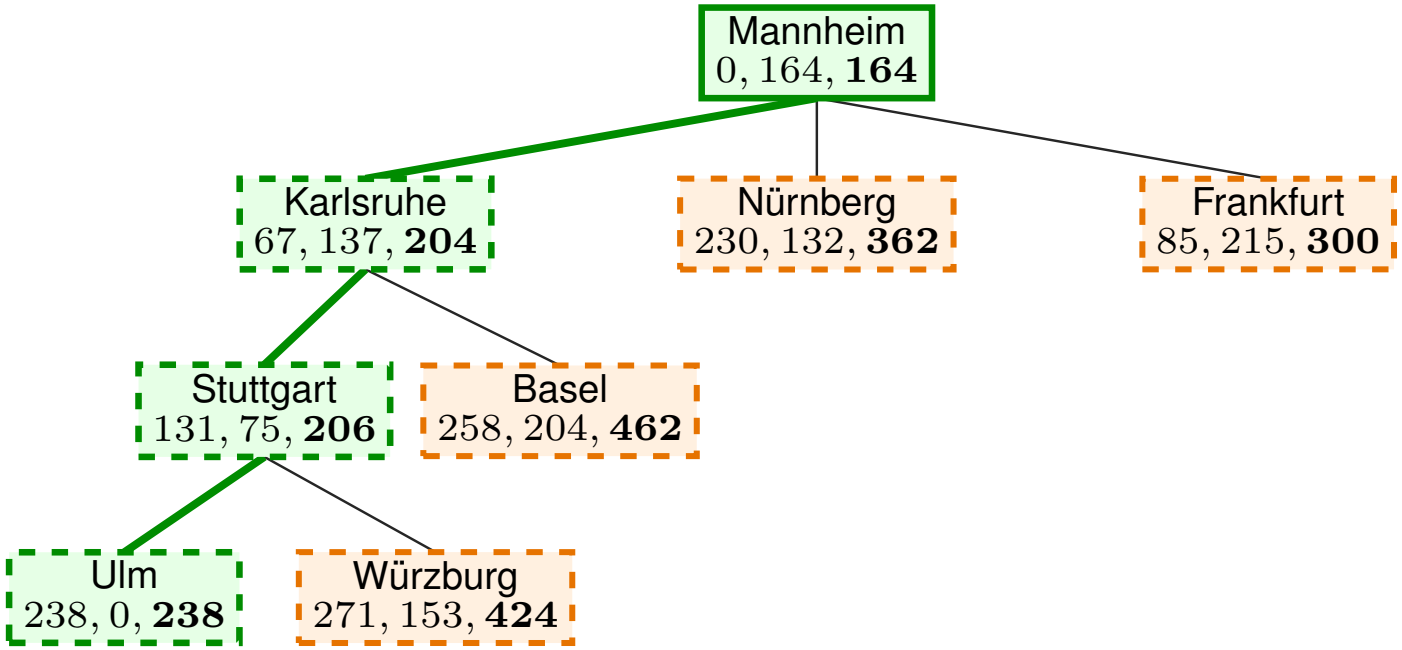
- straight-line distance $d(s, z)$ is replaced by time $t(s, z) = d(s, z)/v_{\max}$ where $v_{\max}$ is the maximum average velocity.
- air line heuristic estimate becomes worse (dividing by $v_{\max}$ causes $t(s, z)$ to be much too small).
- landmark heuristic is still very good.
- **Further improvement:** *contraction hierarchies* (combining, in a precomputation step, several edges into so-called *shortcuts*, which are then used to reduce the search space [Batzill 2016]; [Geisberger, Sanders, Schultes, and Delling 2008]).

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 θ.
    - 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 θ 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.

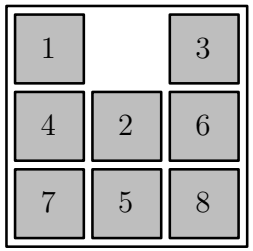


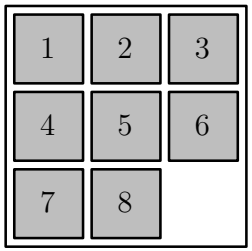
Iteration 1. With $\theta = 164$, expand Mannheim. Generated f-values $204, 362, 300$ all exceed θ, so they are *cut off*. Next θ is the smallest exceeded f-value, 204.**Iteration 2.** With $\theta = 204$, expand Mannheim

Simple admissible heuristics for 8-puzzle

- h_1: counts the number of squares that are not in the right place.
- h_2: *Manhattan distance*
  - For every square the horizontal and vertical distances to that square's location in the goal state are added together.
  - This value is then summed over all squares.

Example 8

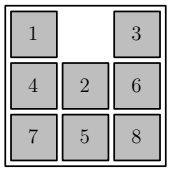
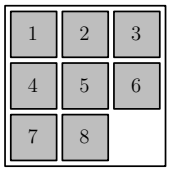
Let node n store the state 

, and let 

be the goal state:

- $h_1(n) = 3$.
- $h_2(n) = 0 + 0 + 0 + 1 + 0 + 0 + 1 + 1 = 3$. (The values respectively correspond to the squares numbered $1, 3, 4, 2, 6, 7, 5, 8$.)

Exercise 11 ([Ertel 2025], Exercise 6.9, p. 126)

Using A* search for the 8-puzzle, search (manually) for a path from the starting state  to the goal state  .

(a) using the heuristic h_1

(b) using the heuristic h_2

Exercise 12 ([Ertel 2025], Exercise 6.10, p. 126)

Construct the A* search tree for the city graph from the map in Example 5 and use the flying distance to Ulm as the heuristic. Start in Bern with Ulm as the destination. Take care that each city only appears once per path.

Exercise 13 ([Ertel 2025], Exercise 6.15, p. 126)

Show that the heuristics h_1 and h_2 for the 8-puzzle are admissible.

Exercise 14 ([Ertel 2025], Exercise 6.11, p. 126)

(a) Show that the triangle inequality is valid for shortest distances on maps.

(b) Using an example, show that it is not always the case that the triangle inequality holds for neighboring vertices x and y of a road graph, where the direct-edge distance is $d(x, y)$. That is, it is not the case that $d(x, y) \leq d(x, z) + d(z, y)$.

Exercise 15 ([Ertel 2025], Exercise 6.12, p. 126)

Program A* search in the programming language of your choice using the heuristics h_1 and h_2 and test these on the 8-puzzle example.

Heuristic Search

Empirical Comparison of the Search Algorithms

- The described algorithms are implemented in Mathematica.
- Averaged over 132 randomly generated 8-puzzle problems.

	Iterative Deepening		A*-Algorithm				
			Heuristic h_1		Heuristic h_2		Num
Depth	Steps	Time [sec]	Steps	Time [sec]	Steps	Time [sec]	Runs
2	20	0.003	3.0	0.0010	3.0	0.0010	10
4	81	0.013	5.2	0.0015	5.0	0.0022	24
6	806	0.13	10.2	0.0034	8.3	0.0039	19
8	6455	1.0	17.3	0.0060	12.2	0.0063	14
10	50512	7.9	48.1	0.018	22.1	0.011	15
12	486751	75.7	162.2	0.074	56.0	0.031	12
			IDA*				
14	–	–	10079.2	2.6	855.6	0.25	16
16	–	–	69386.6	19.0	3806.5	1.3	13
18	–	–	708780.0	161.6	53941.5	14.1	4

Figure: Comparison of the computation cost of uninformed search and heuristic search for solvable 8-puzzle problems with various depths. Measurements are in steps and seconds. All values are averages over multiple runs (see last column).

Heuristic Search
Empirical Comparison of the Search Algorithms

- The *heuristics significantly reduce the search cost* compared to uninformed search.
- If we compare iterative deepening to A* with h_1 at depth 12, for example, it becomes evident that h_1 *reduces the number of steps by a factor of about* 3000, but the computation time by only *a factor of* 1023. This is due to *the higher cost per step for the computation of the heuristic*.
- Closer examination reveals *a jump in the number of steps between depth 12 and depth 14* in the column for h_1.
  - This jump cannot be explained solely by the repeated work done by IDA*.
  - It comes about because *the implementation of A* removes duplicate search nodes that represent an already stored puzzle state and thereby shrinks the search space*. This is not possible with IDA* because *it saves almost no nodes*.
  - Despite this, *A* can no longer compete with IDA* beyond depth 14* because *the cost of sorting in new nodes pushes up the time per step* so much.
- *Effective branching factor*
  - Uninformed search: 2.8 *Heuristic* h_1*: 1.5* *Heuristic* h_2*: 1.3*

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 terminal or cutoff 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*, not only our immediate move.

Algorithms

- *Minimax search* explores alternating moves, assumes the opponent's best replies, and *chooses our best move*.
- *Alpha-beta pruning* returns the *same move* while skipping branches that *cannot change it*.

Games with Opponents
Minimax Search

- Chess has an *effective branching factor* of *about* 30 *to* 35. With 50 moves per player, the tree can have $30^{100} \approx 10^{148}$ leaves, so *full search is infeasible*.
- A *time limit* means that search must *stop at a chosen depth*.
- At that depth, most leaves are *not finished games*. A *heuristic evaluation* $B(s)$ estimates how good position s is; its quality strongly affects play.
- Call the player whose result we optimize *Max* and the opponent *Min*. Assume Min *always chooses their best move*.
- *Higher* $B(s)$ means that s is *better for Max and worse for Min*: Max chooses the highest child value; Min chooses the lowest.

Games with Opponents
Minimax Search

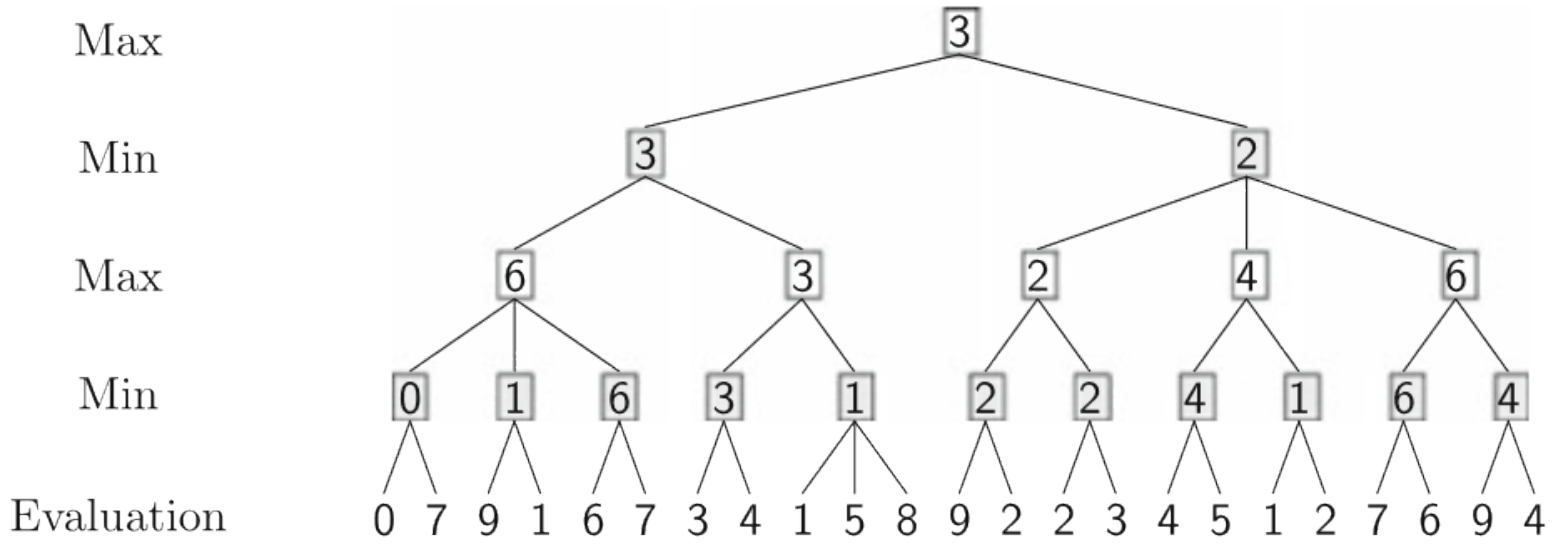


Figure: A minimax game tree with look-ahead of four half-moves. The evaluations of all leaves are given. The evaluation of an inner node is derived recursively as the maximum or minimum of its child nodes, depending on the node's level.

Games with Opponents
Alpha-Beta-Pruning

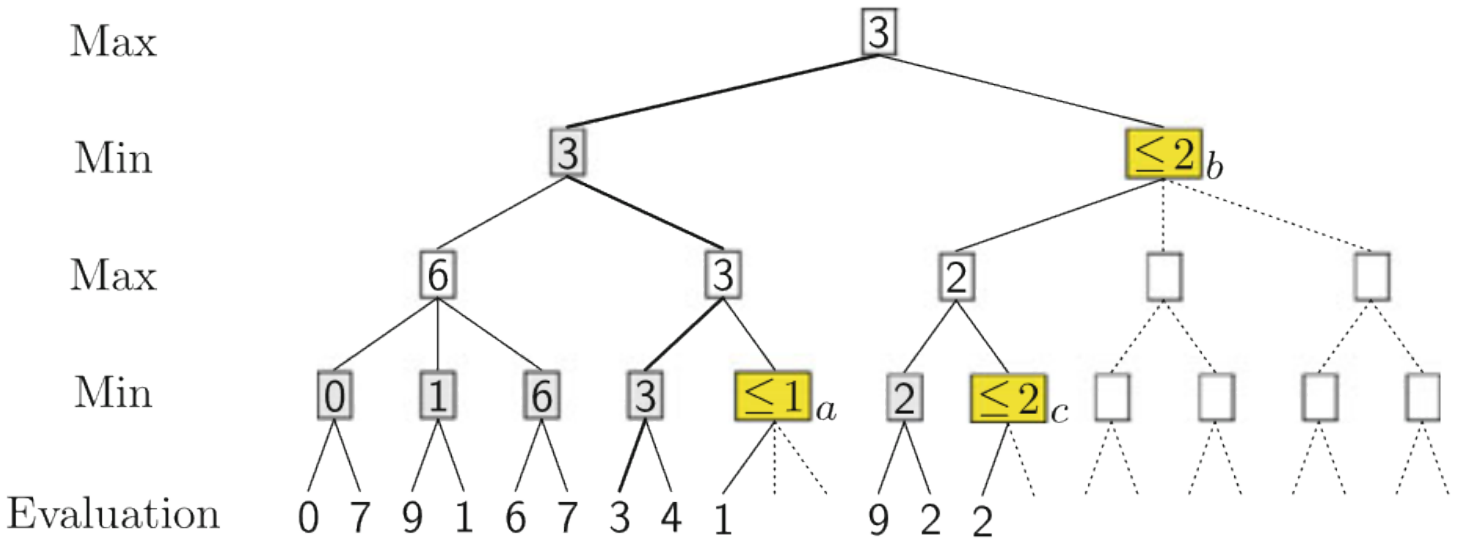


Figure: An alpha-beta game tree with look-ahead of four half-moves. The dotted portions of the tree are not traversed because they have no effect on the end result.

- At every leaf node the evaluation function is calculated.
- For every maximum node the current largest child value is saved in α. For every minimum node the current smallest child value is saved in β.
- If at a minimum node k the current value $\beta \leq \alpha$, then the search under k can end. Here α is the largest value of a maximum node in the path from the root to k.
- If at a maximum node l the current value $\alpha \geq \beta$, then the search under l can end. Here β is the smallest value of a minimum node in the path from the root to l.

Games with Opponents
Alpha-Beta-Pruning

AlphaBetaMax(Node, α, β)

```
1  if DepthLimitReached(Node)
2        return (Rating(Node))
3  NewNodes = Successors(Node)
4  while NewNodes ≠ ∅
5        α = Maximum(α, AlphaBetaMin(First(NewNodes), α, β))
6        if α ≥ β
7              return (β)
8        NewNodes = Rest(NewNodes)
9  return (α)
```

AlphaBetaMin(Node, α, β)

```
1  if DepthLimitReached(Node)
2        return (Rating(Node))
3  NewNodes = Successors(Node)
4  while NewNodes ≠ ∅
5        β = Minimum(β, AlphaBetaMax(First(NewNodes), α, β))
6        if β ≤ α
7              return (α)
8        NewNodes = Rest(NewNodes)
9  return (β)
```

AlphaBetaMin and AlphaBetaMax are extensions of DFS which call themselves mutually.

Initial call:
AlphaBetaMax(RootNode, $-\infty$, ∞).

Games with Opponents
Alpha-Beta-Pruning

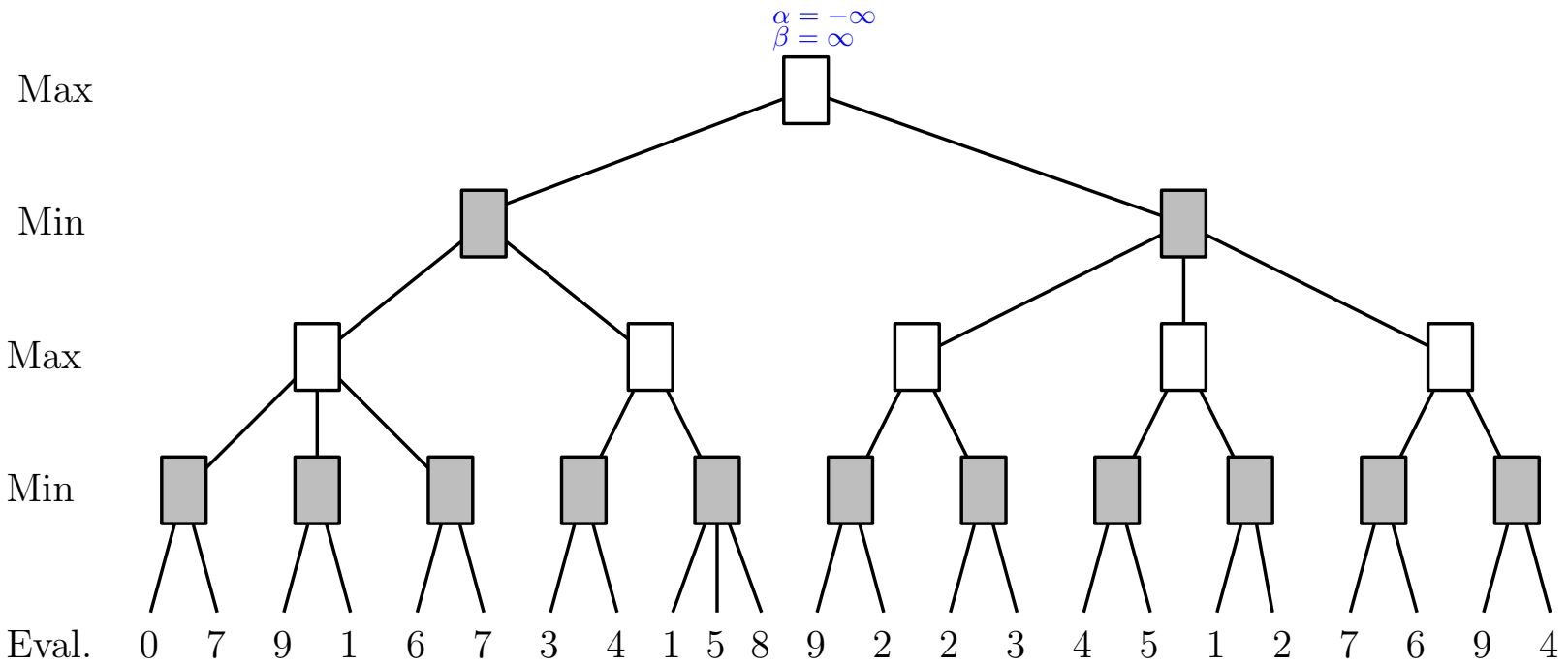


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

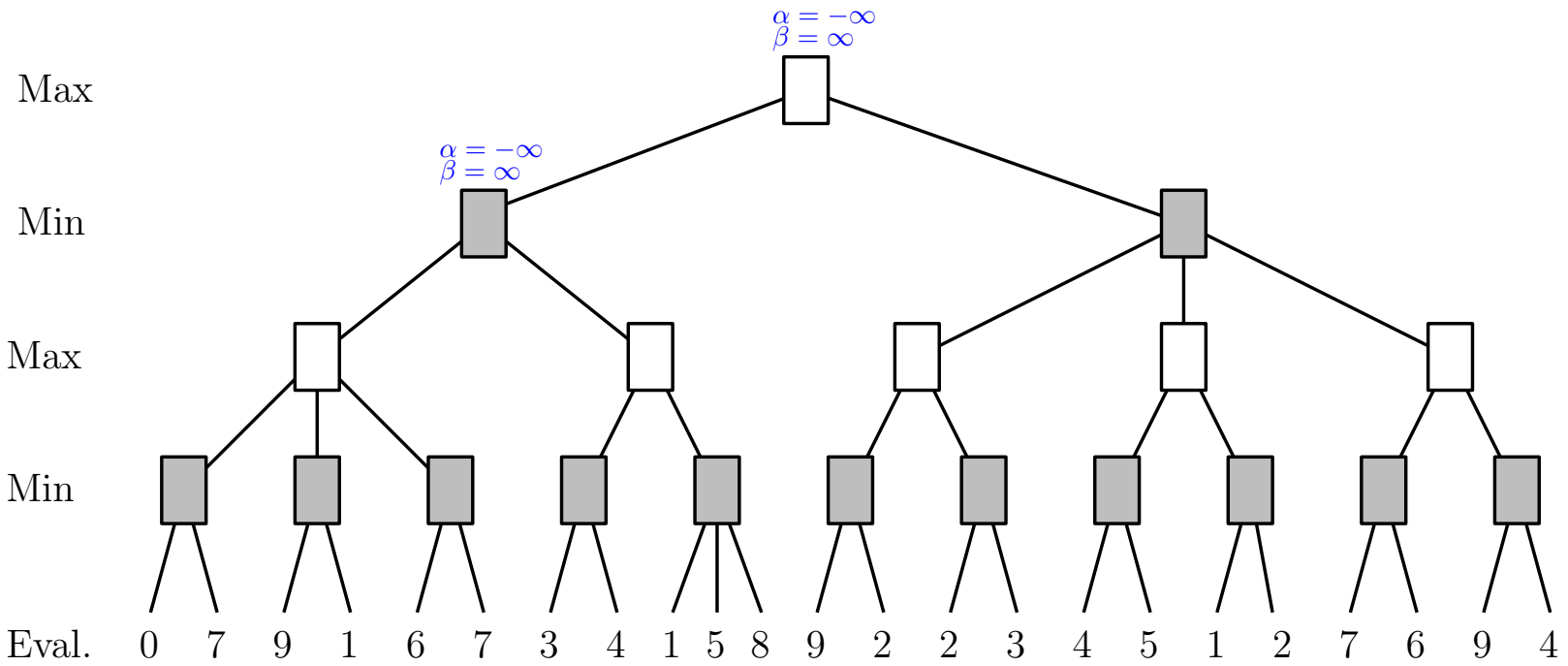


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

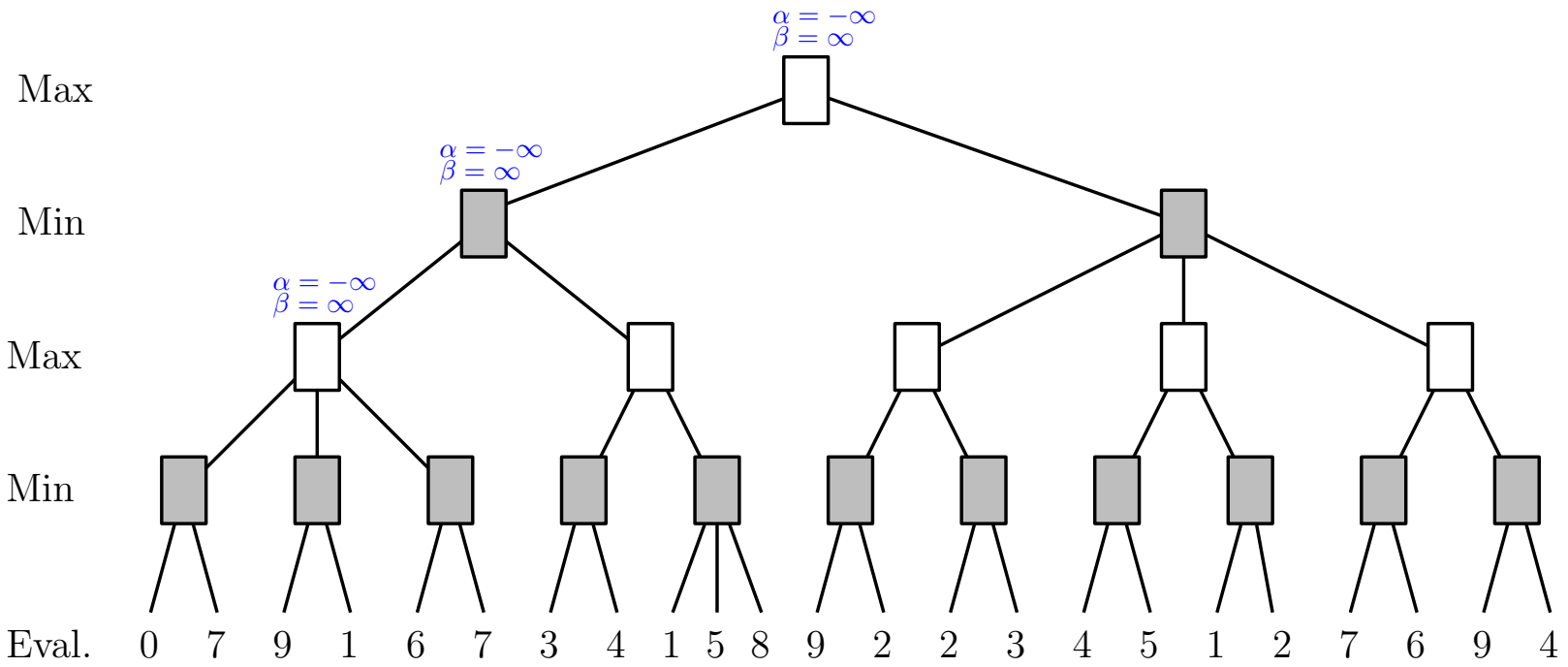


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

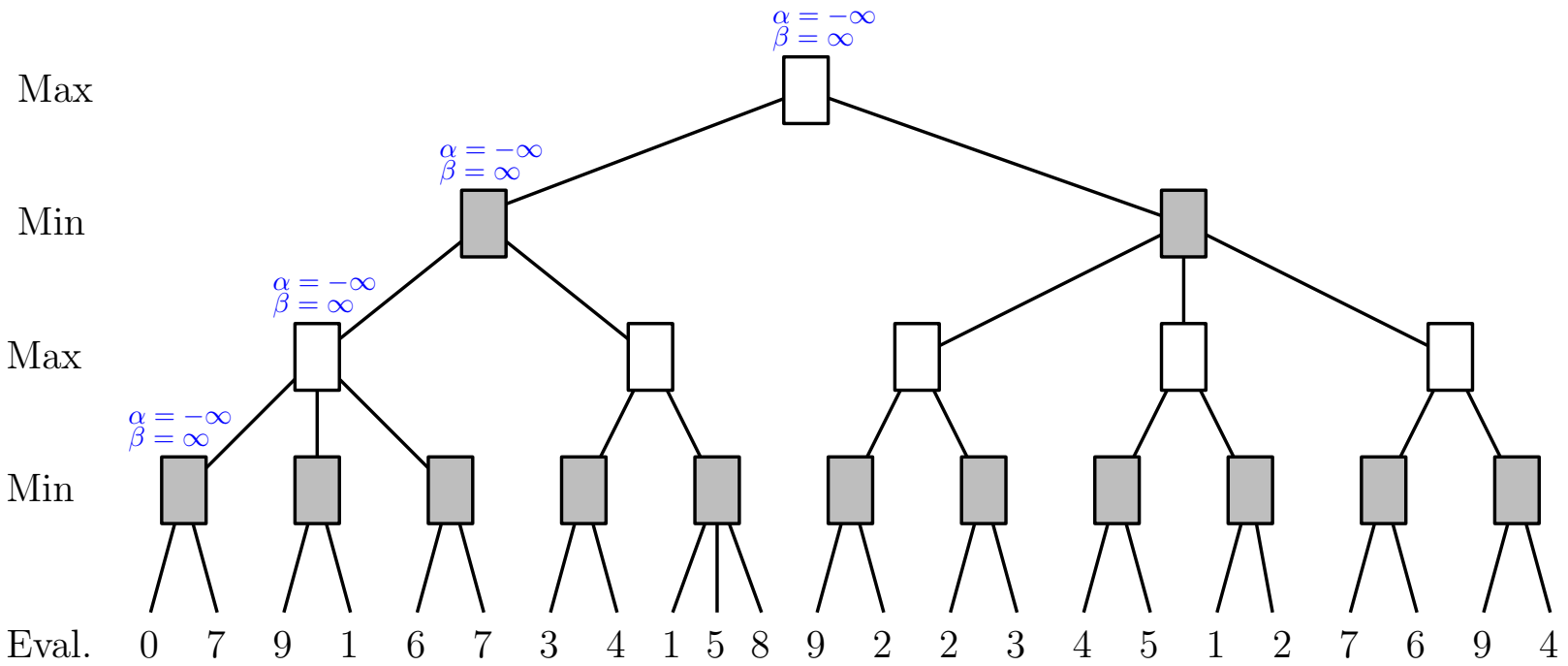


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

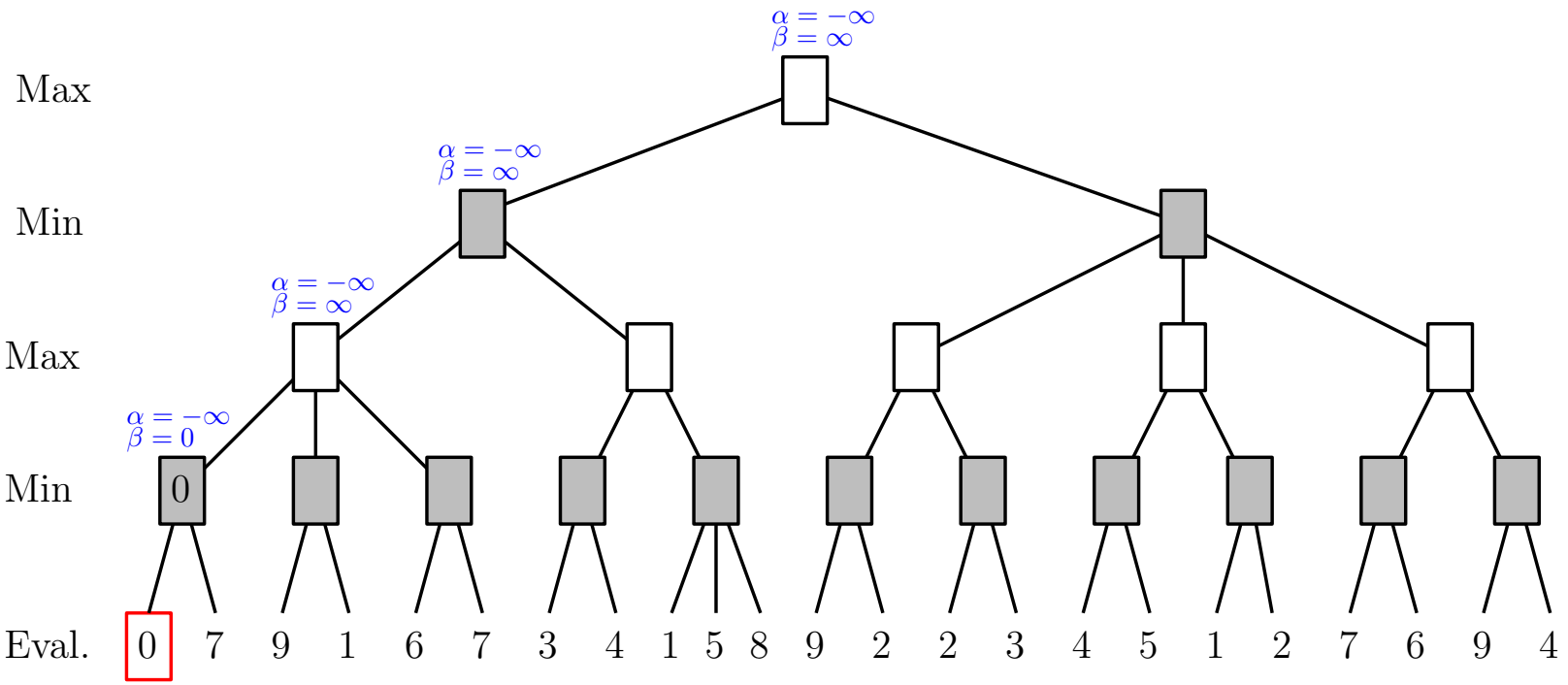


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

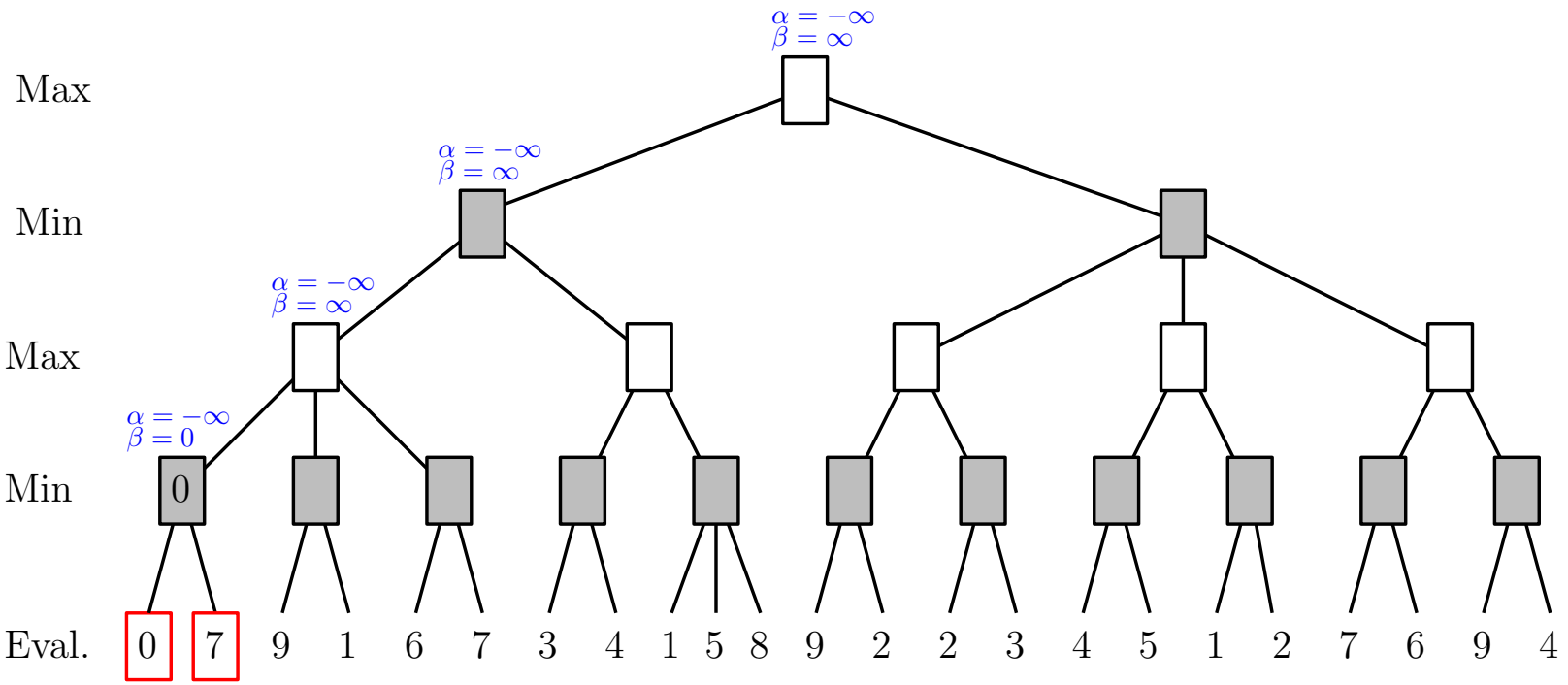


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

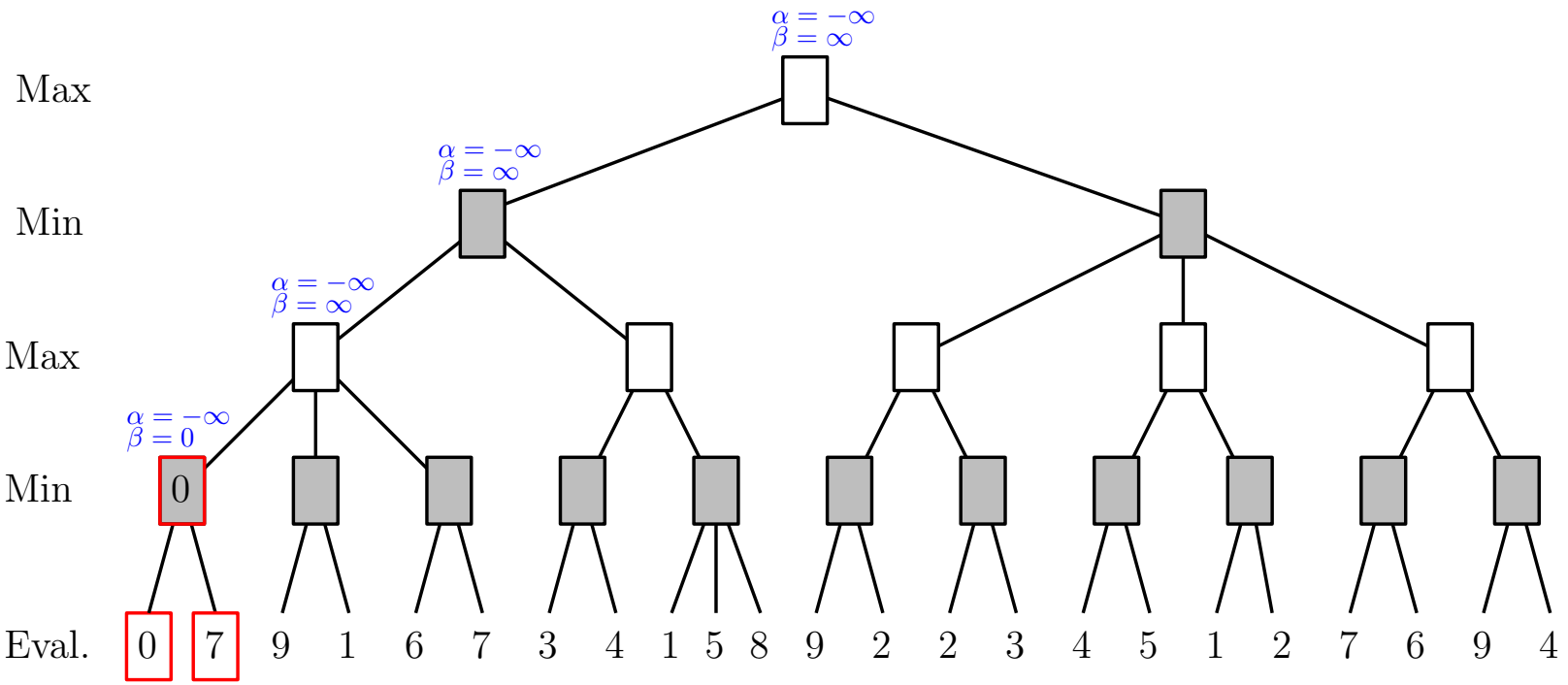


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

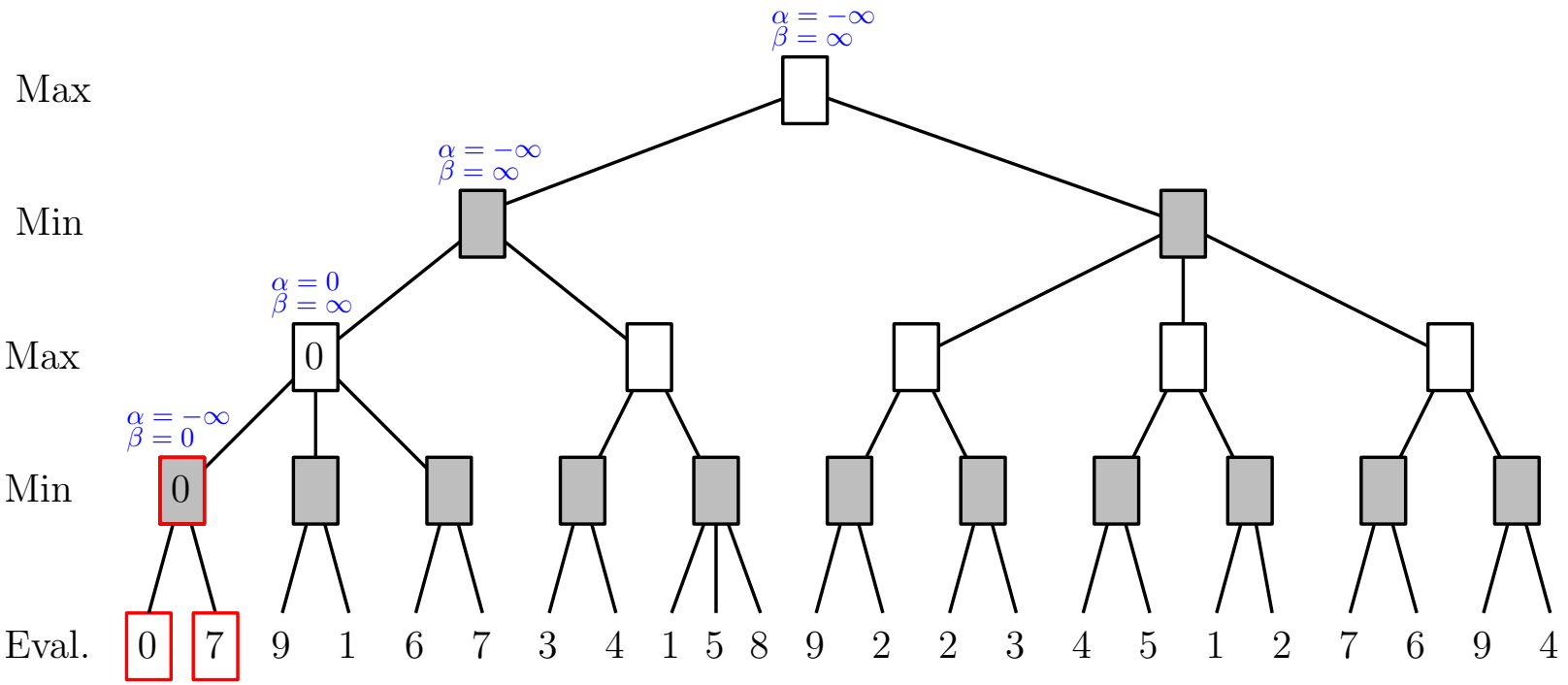


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

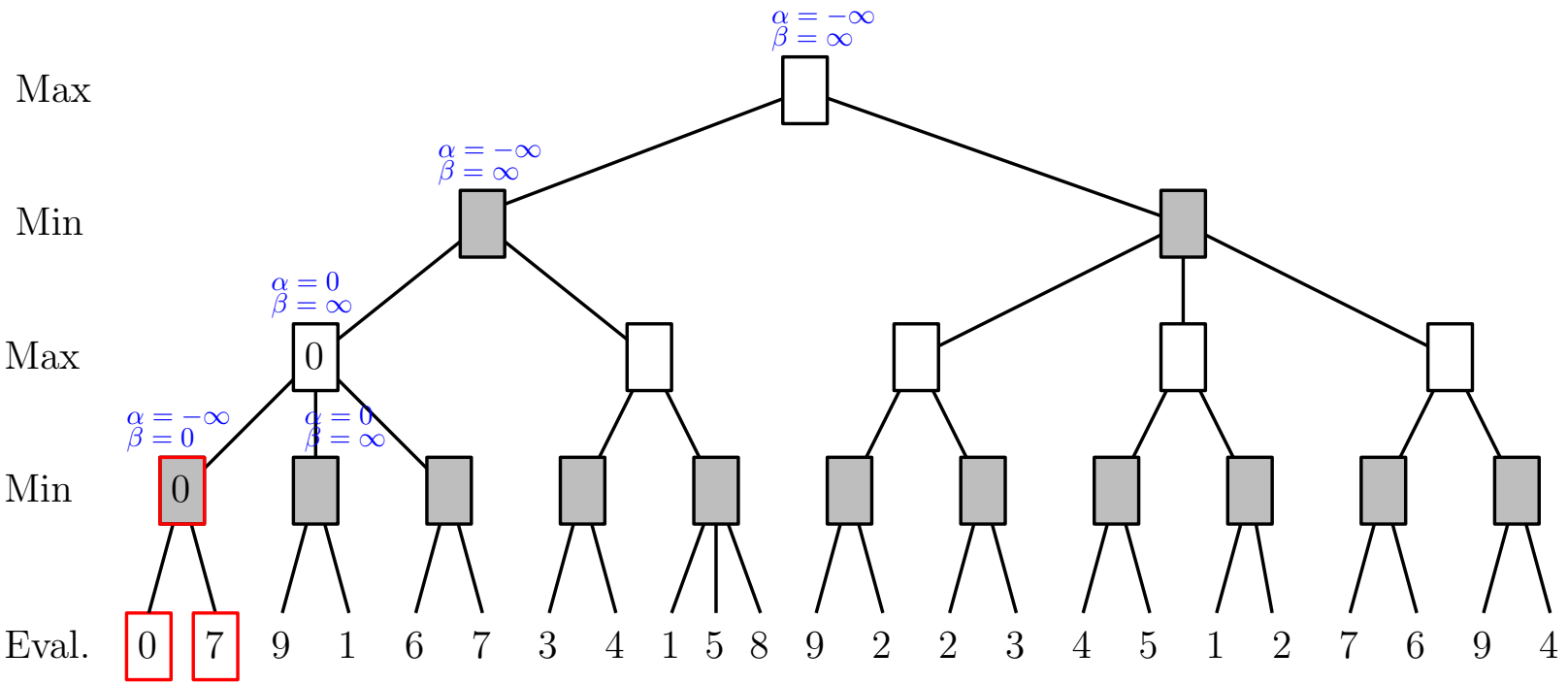


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

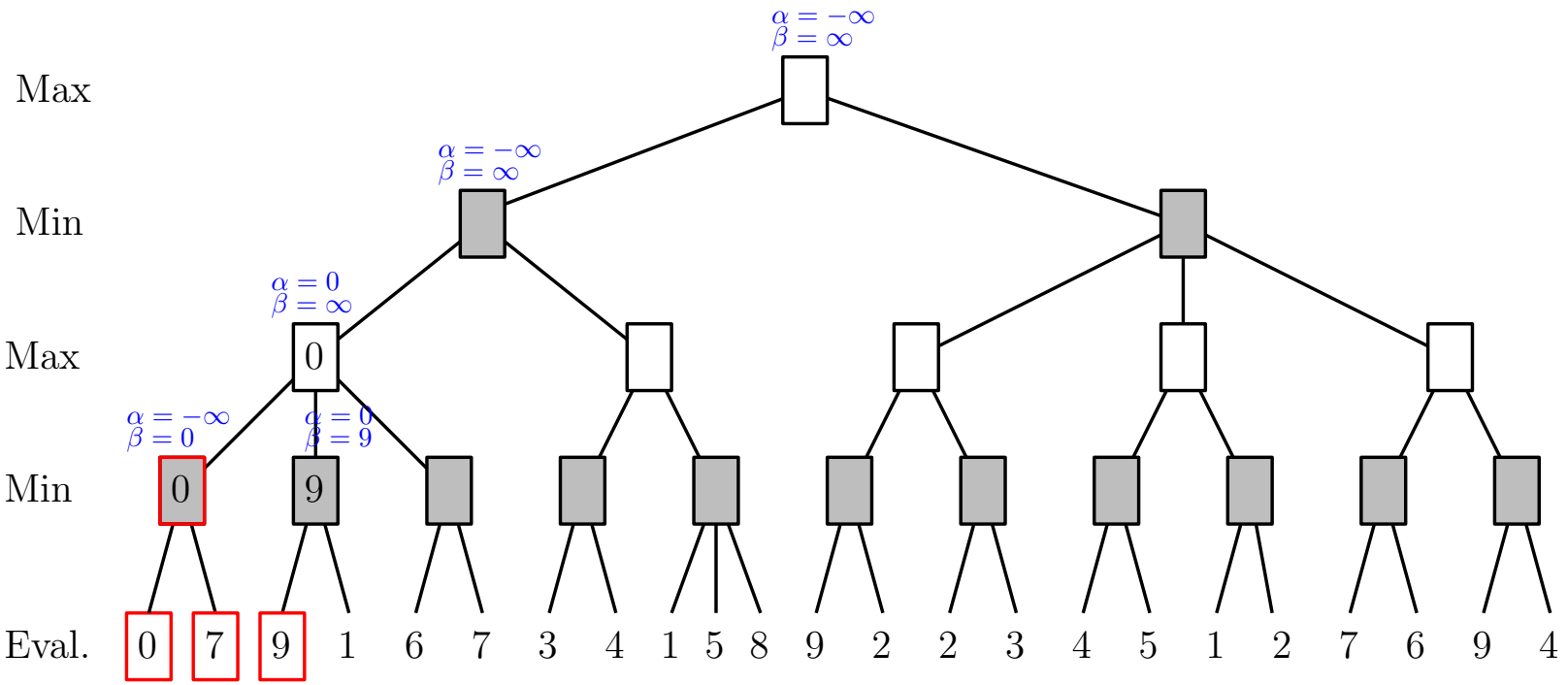


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

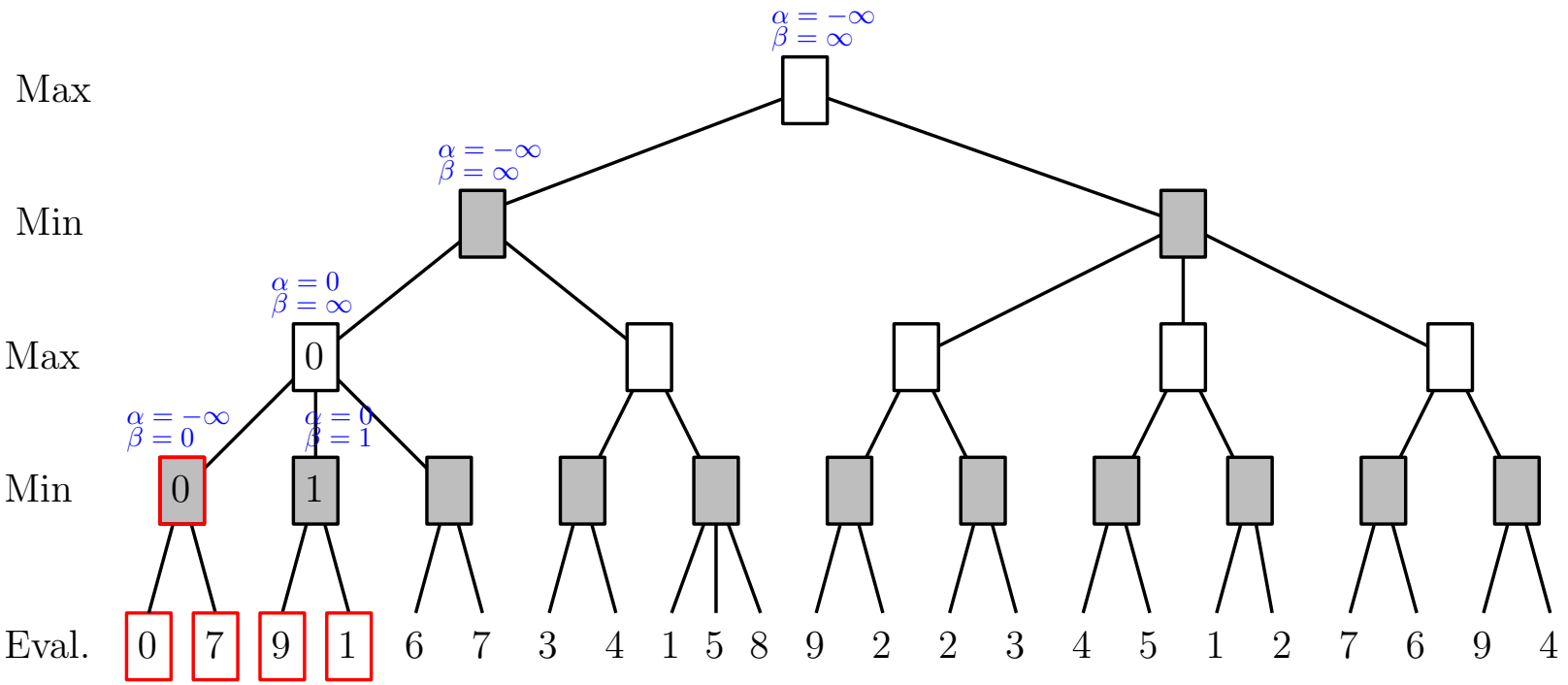


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

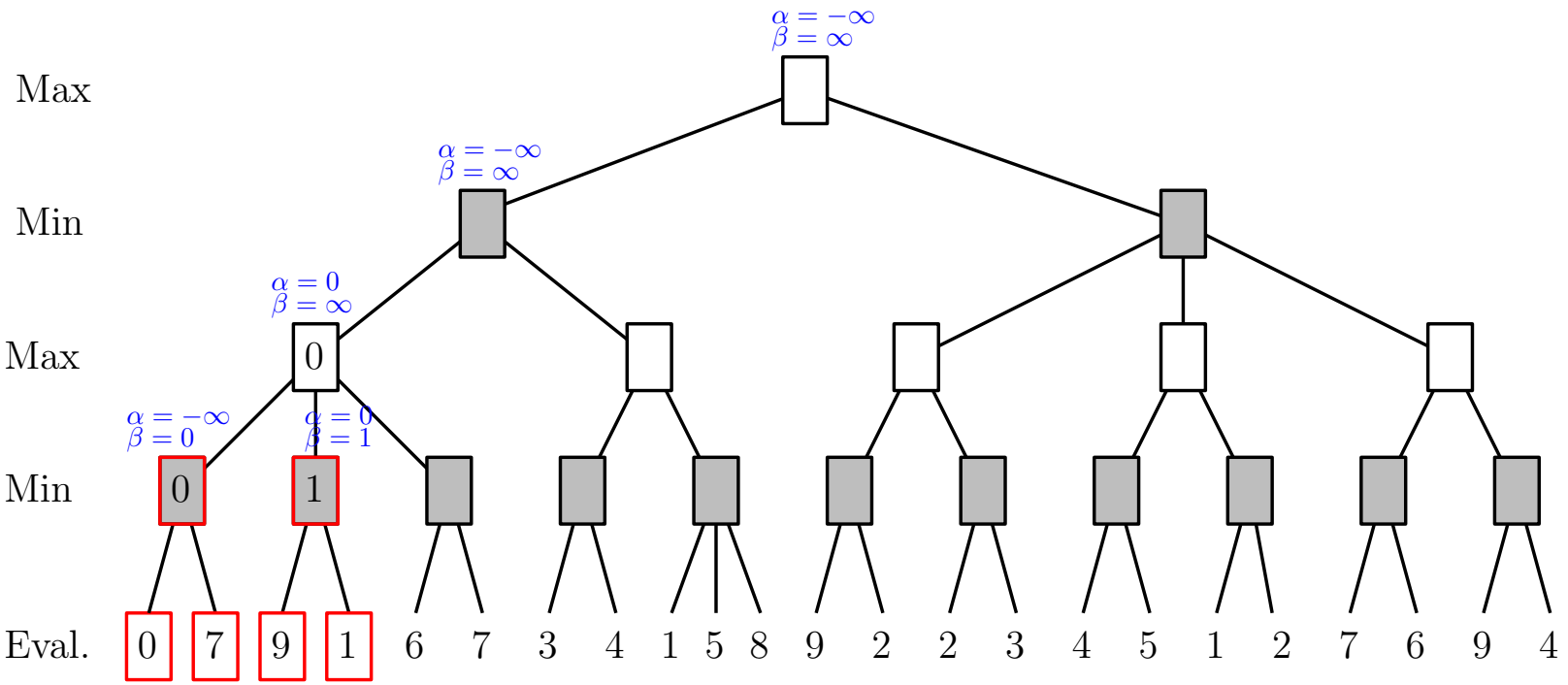


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

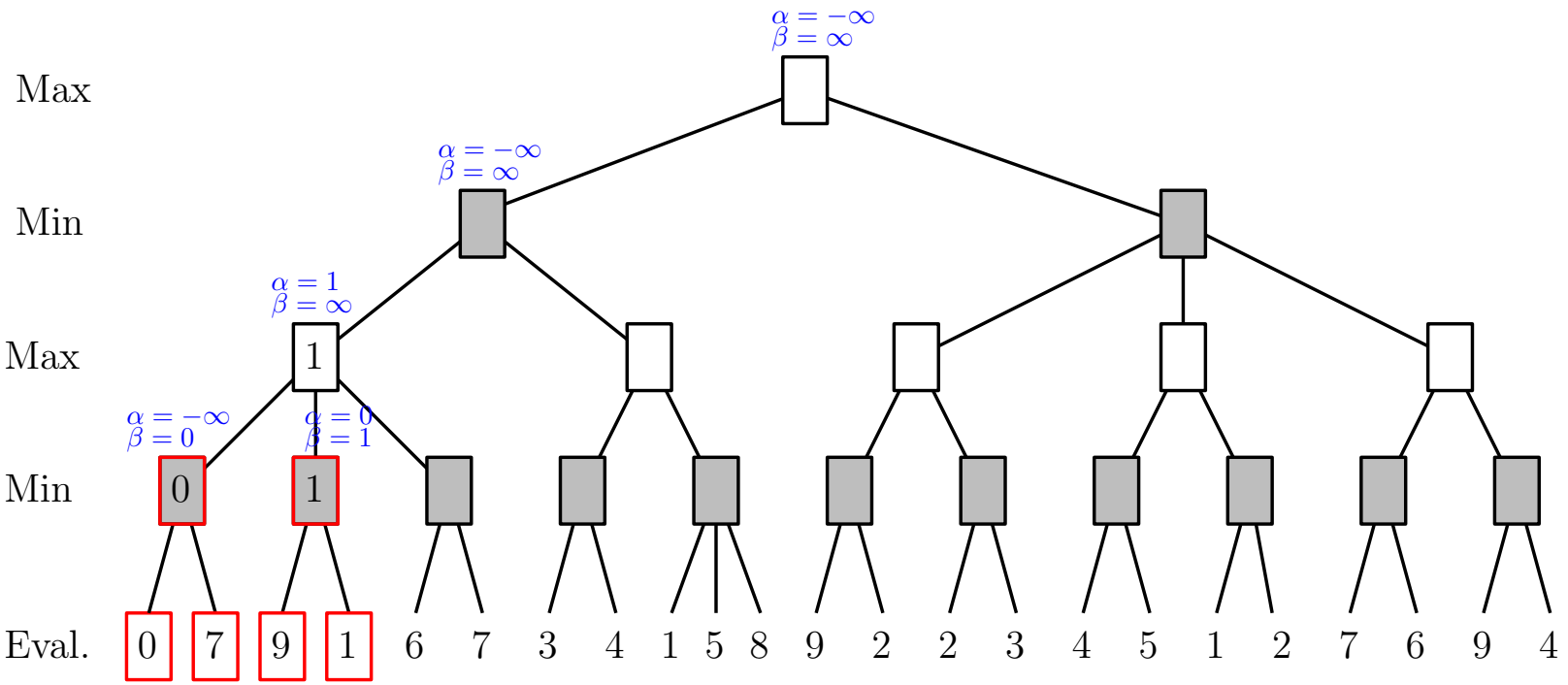


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

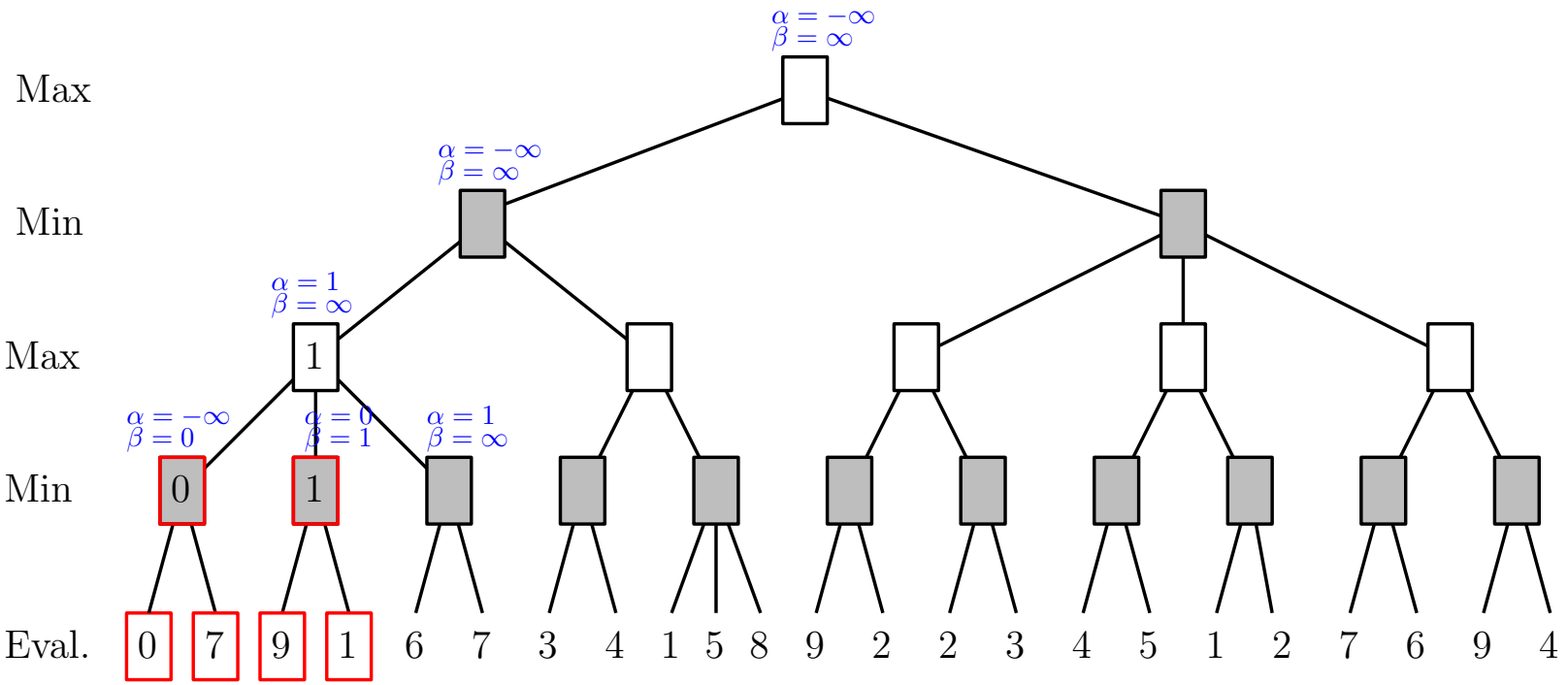


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

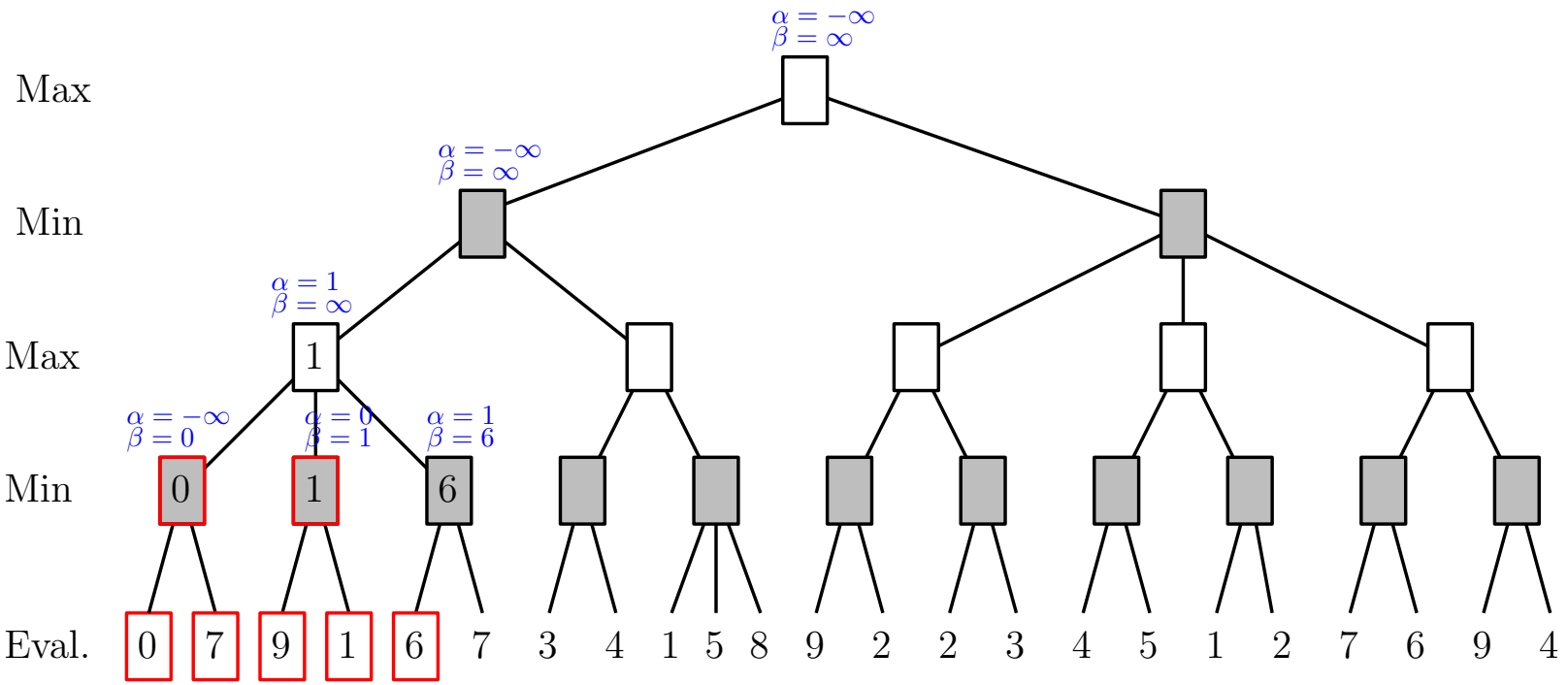


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

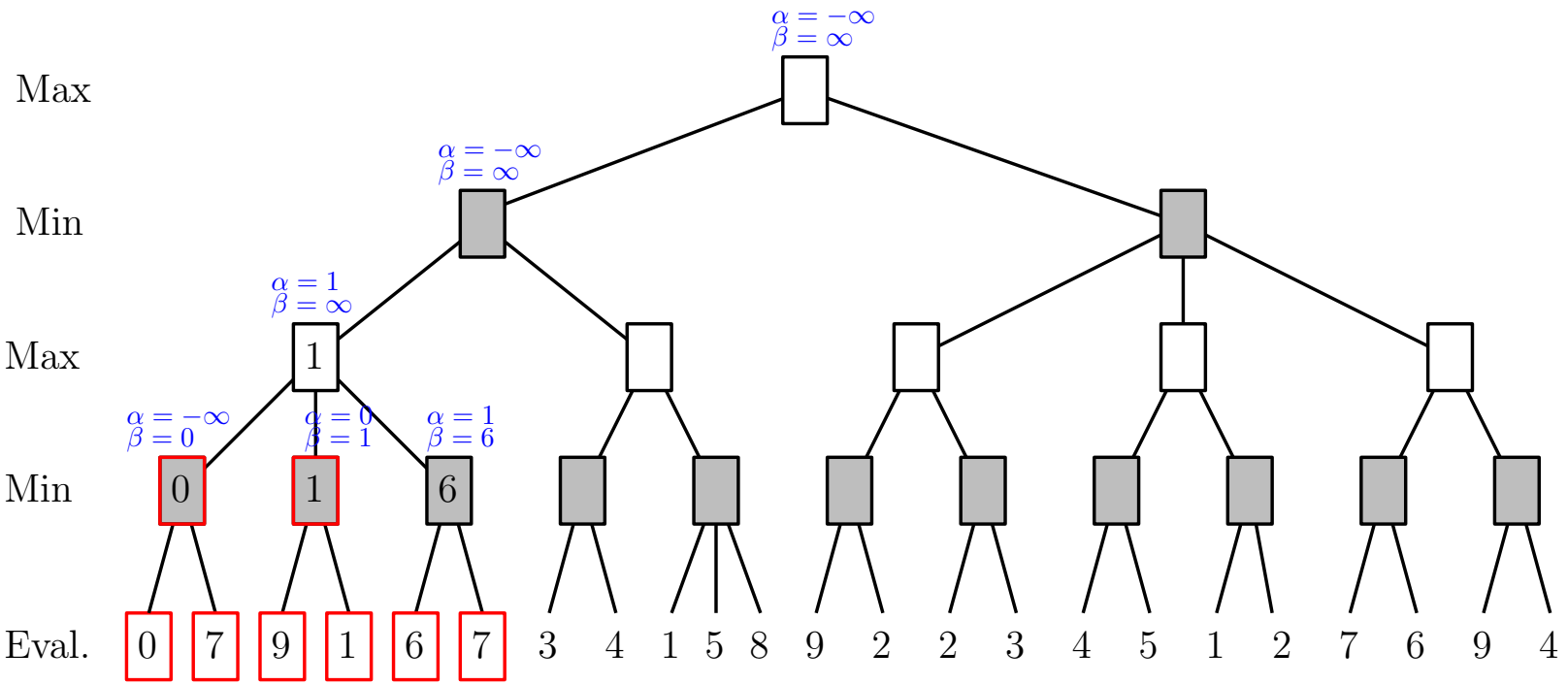


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

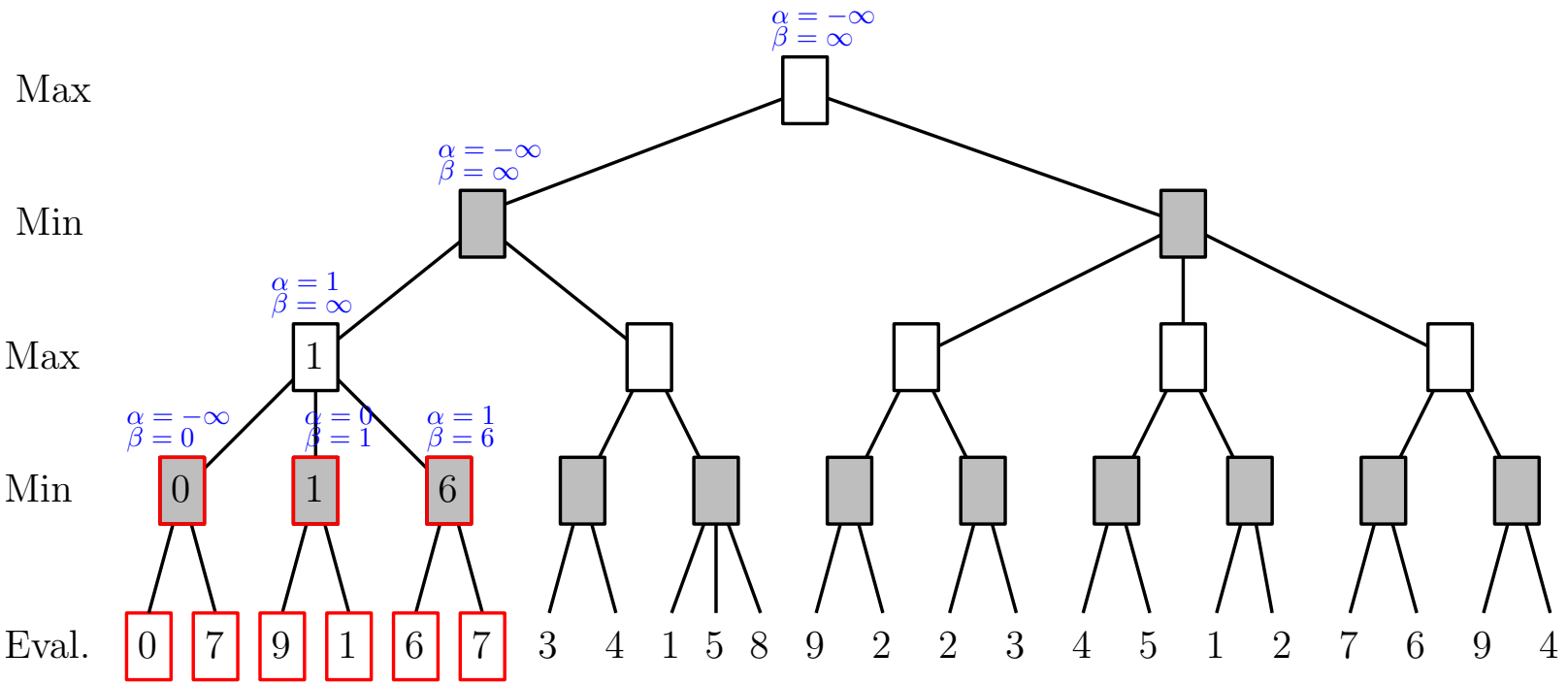


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

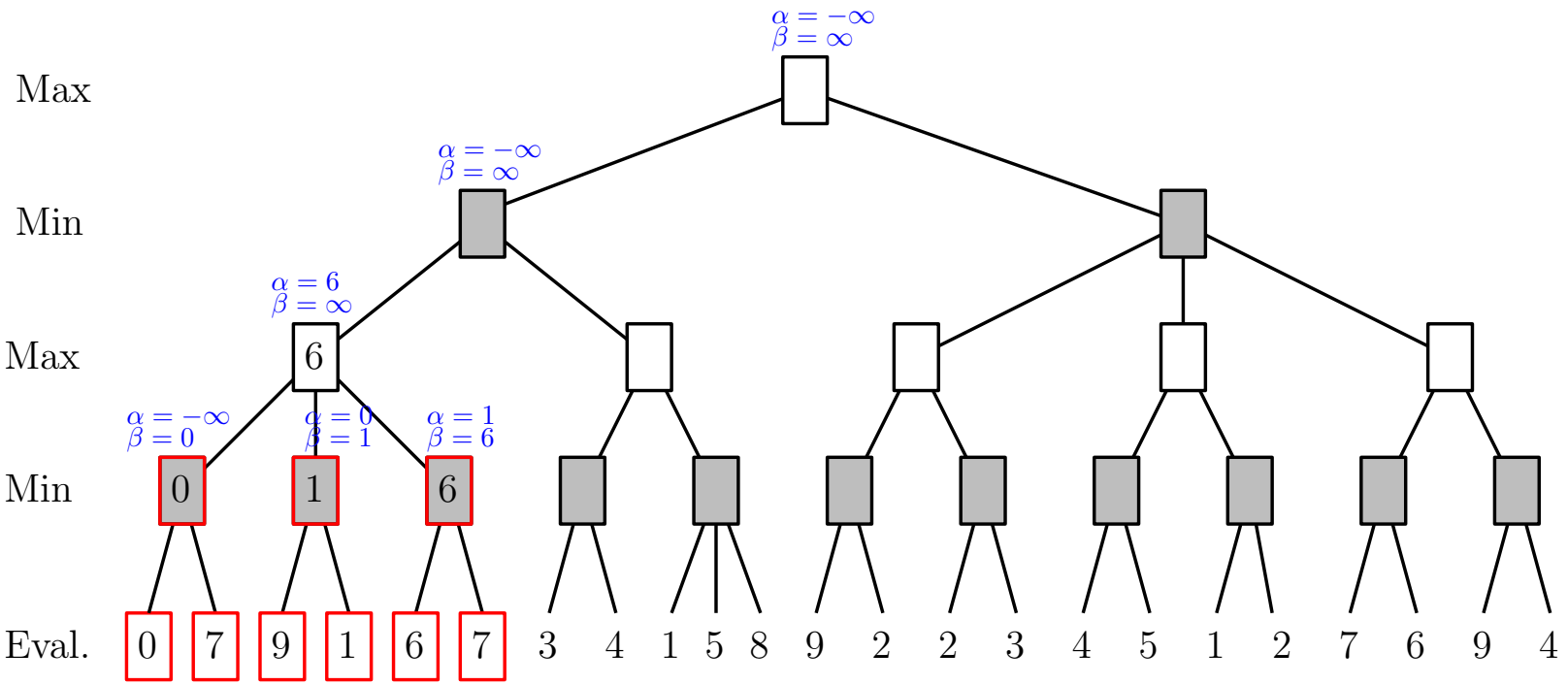


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

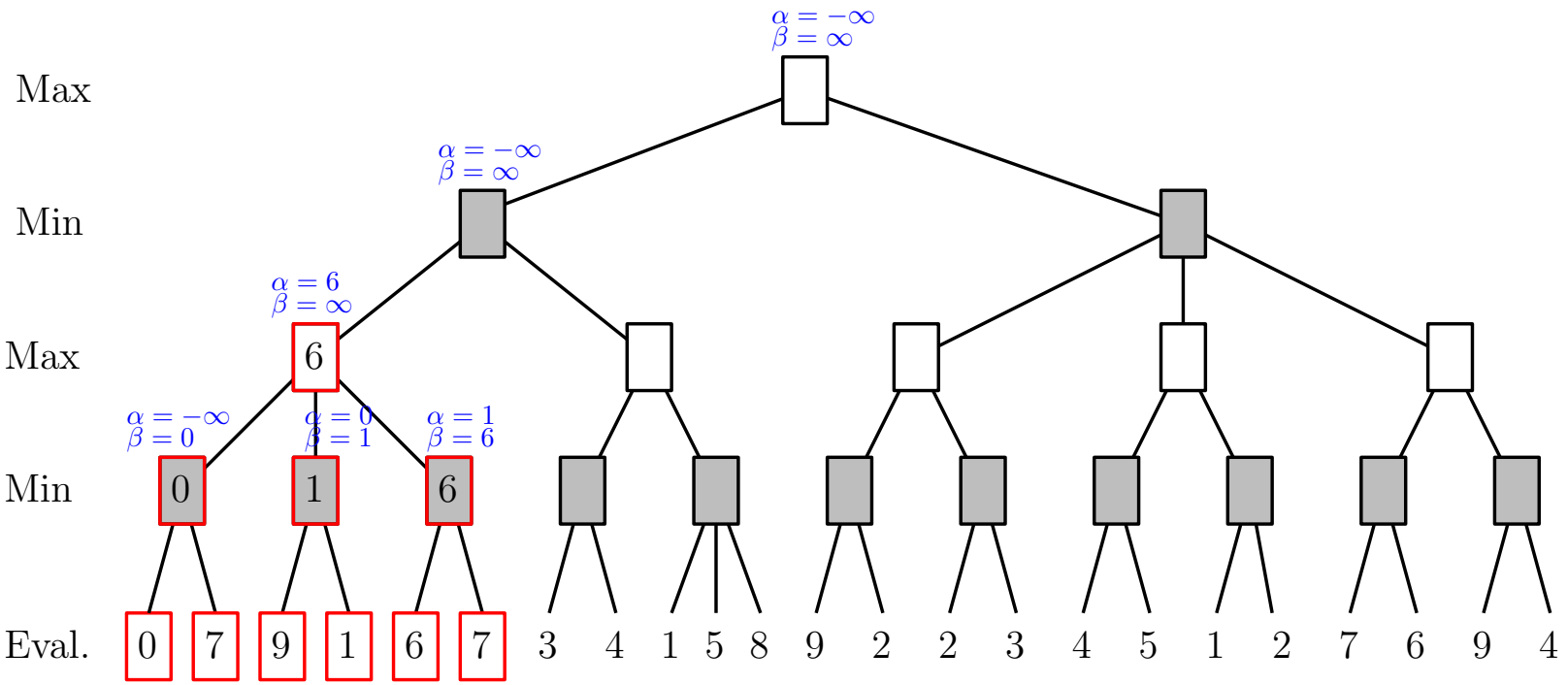


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

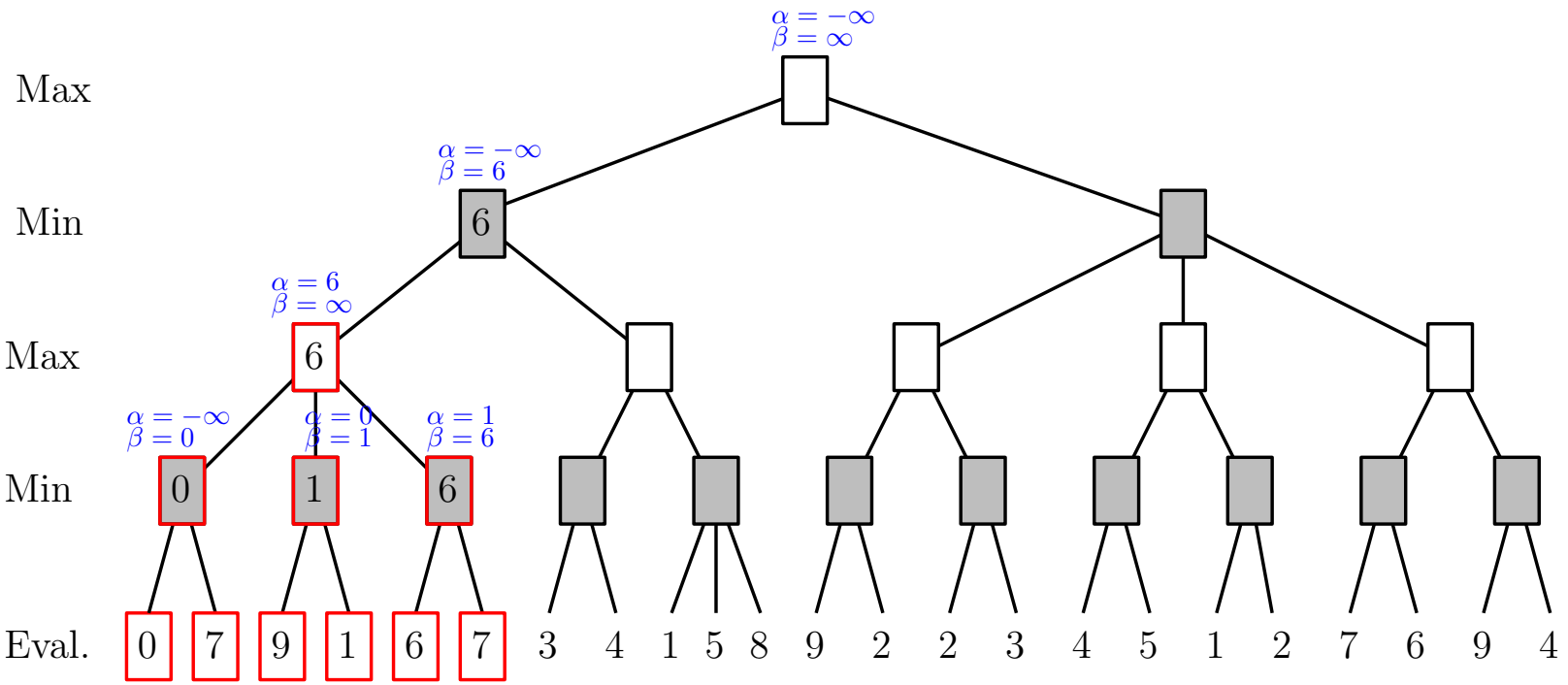


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

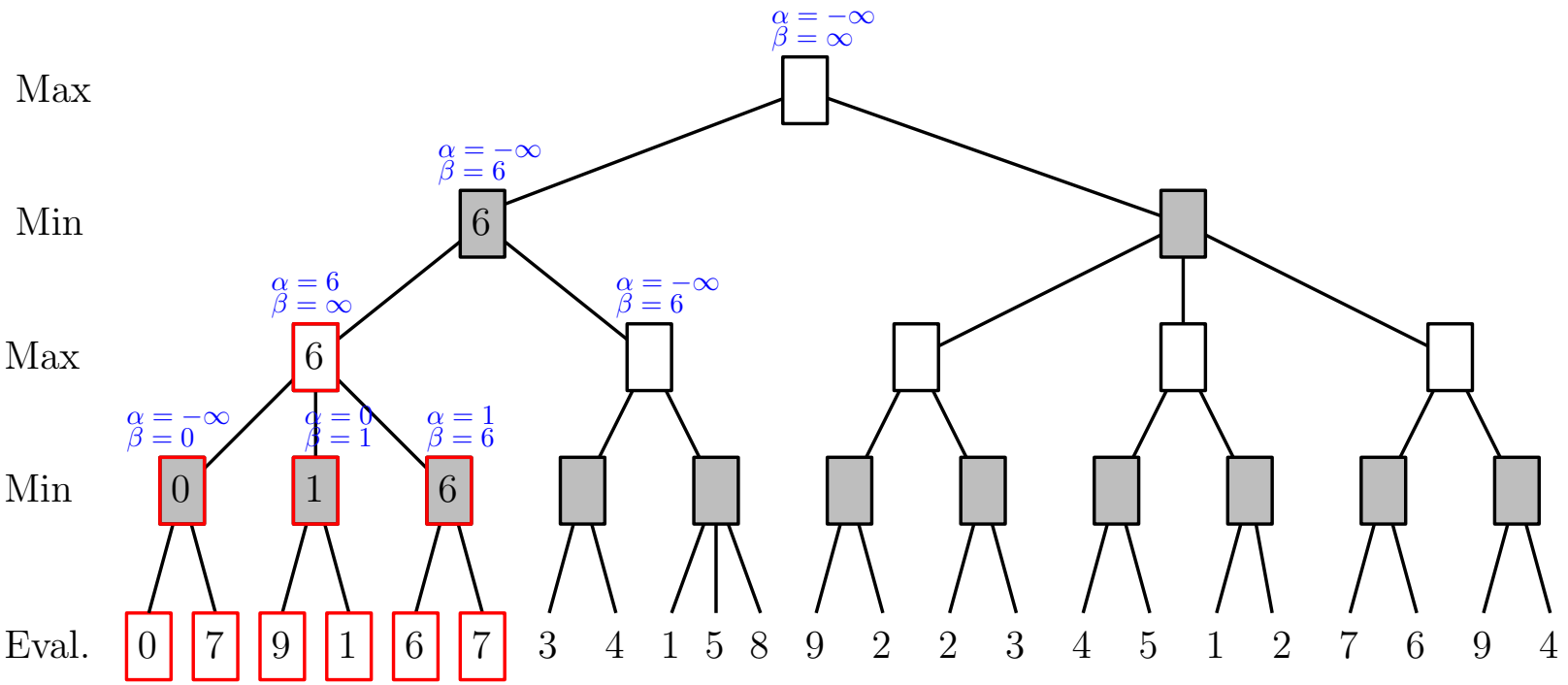


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

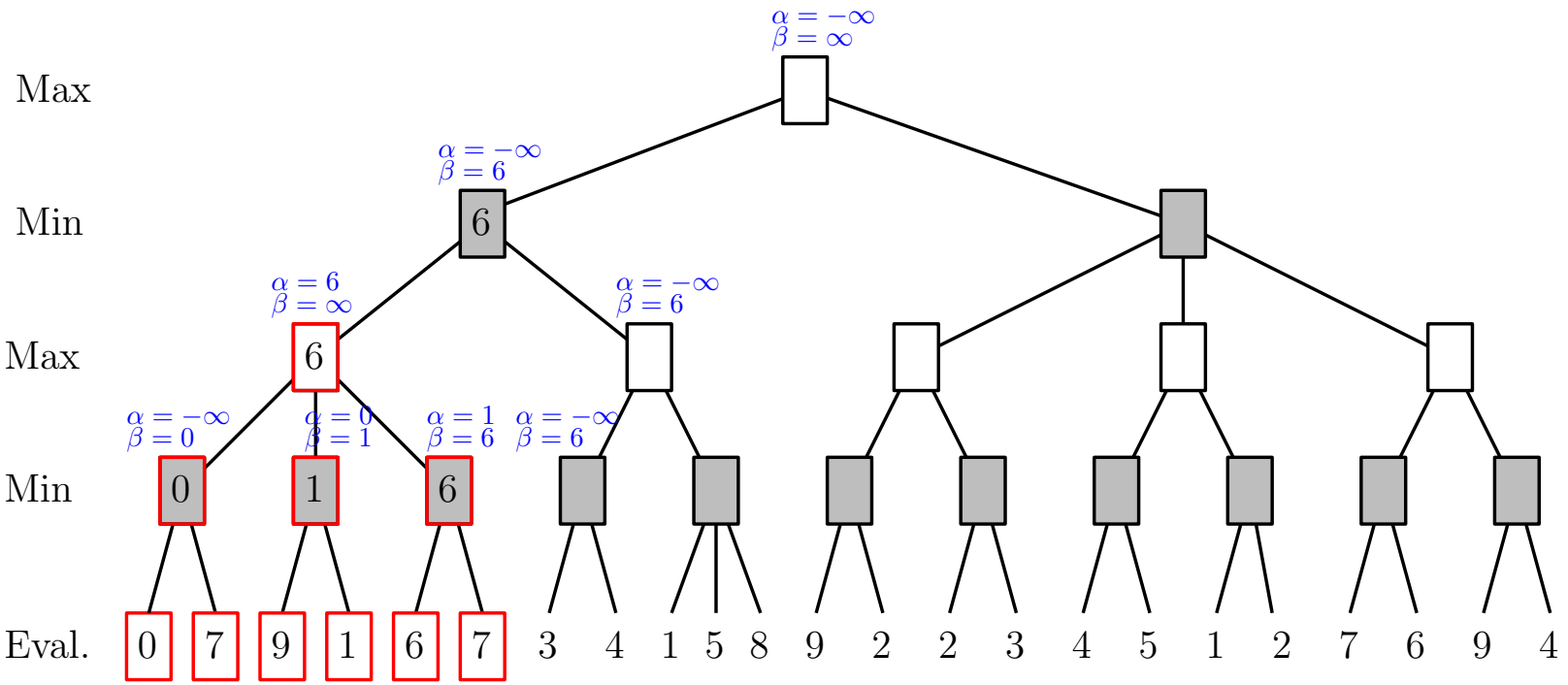


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

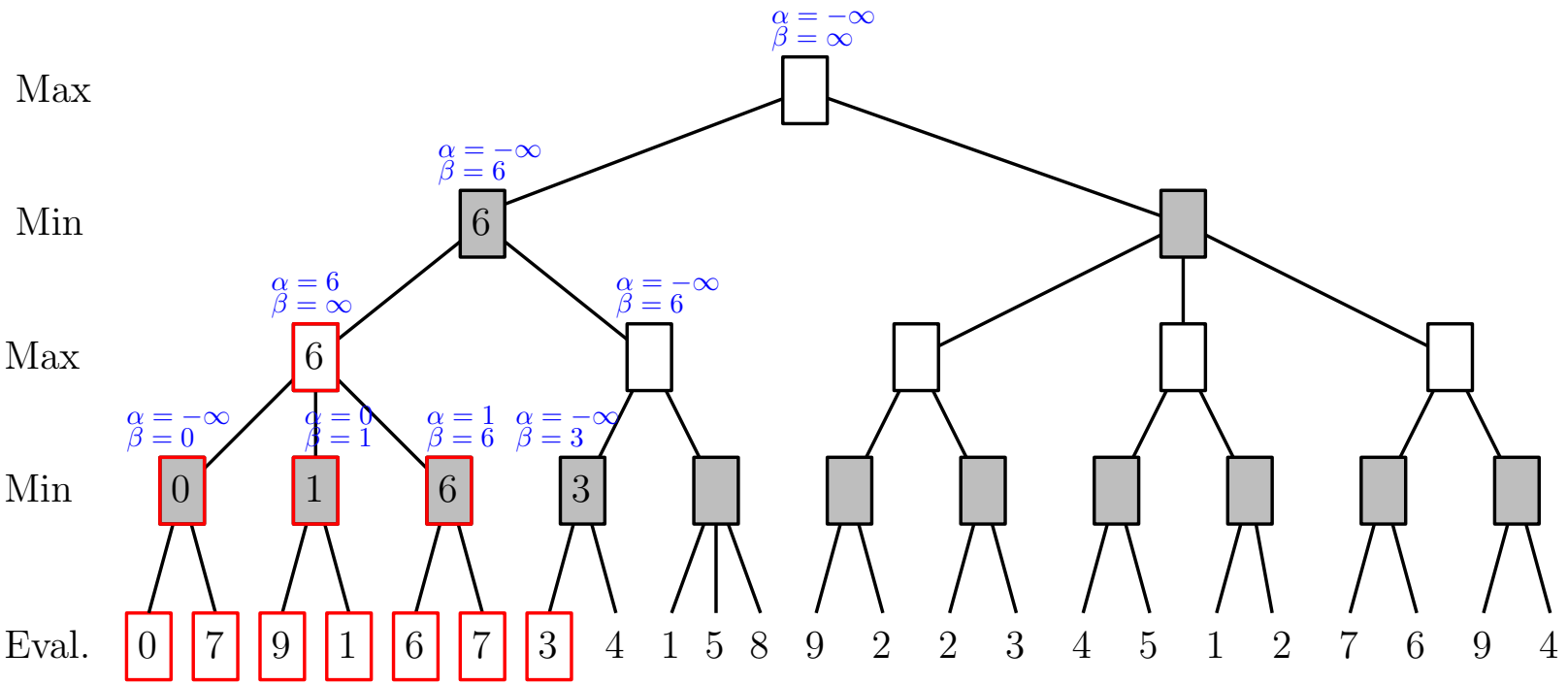


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

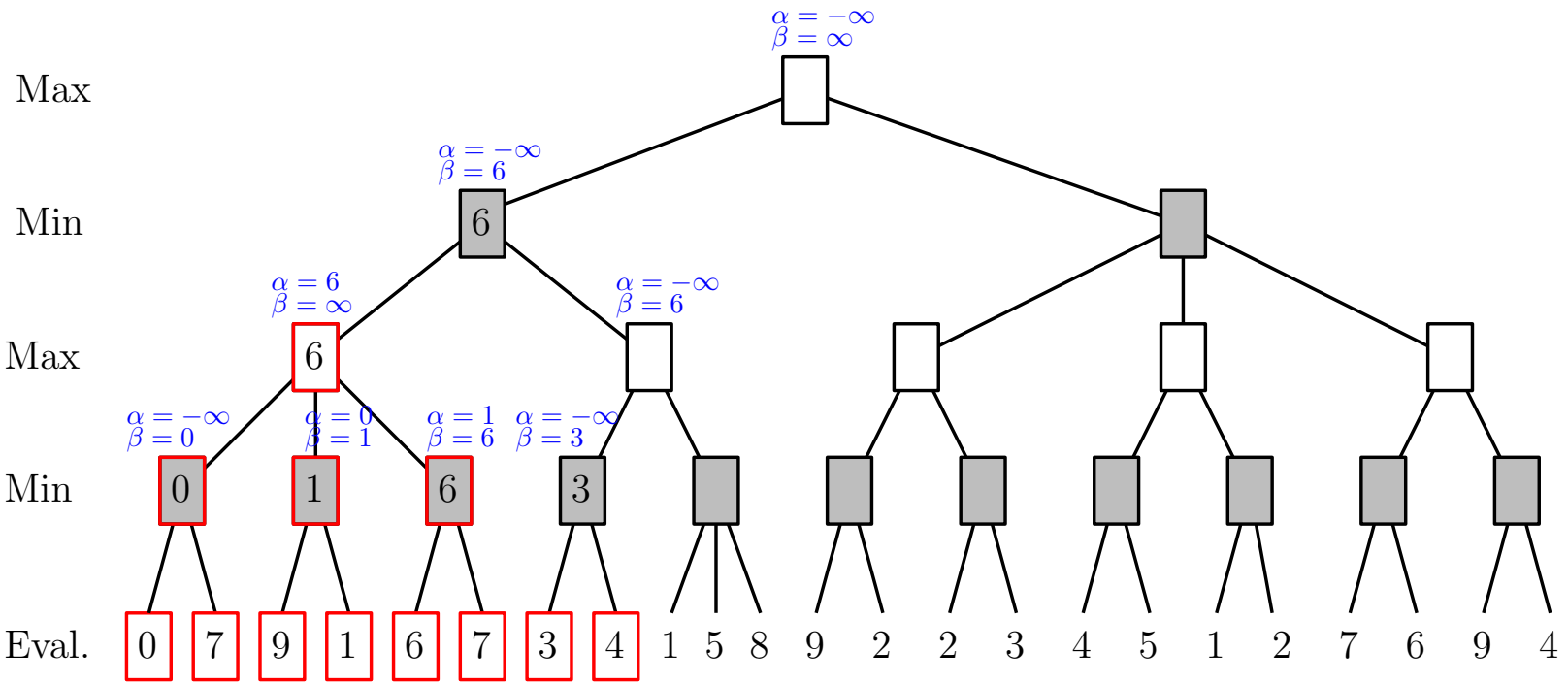


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

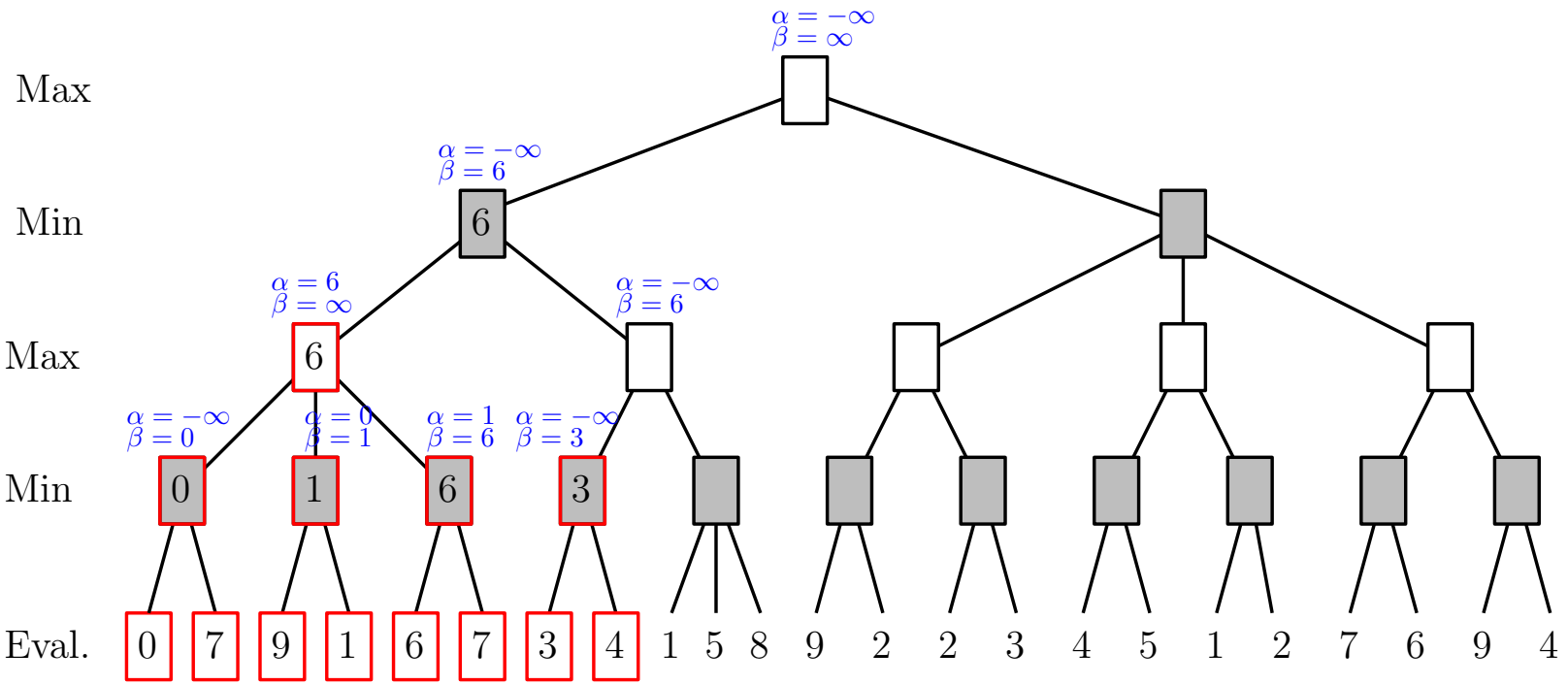


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

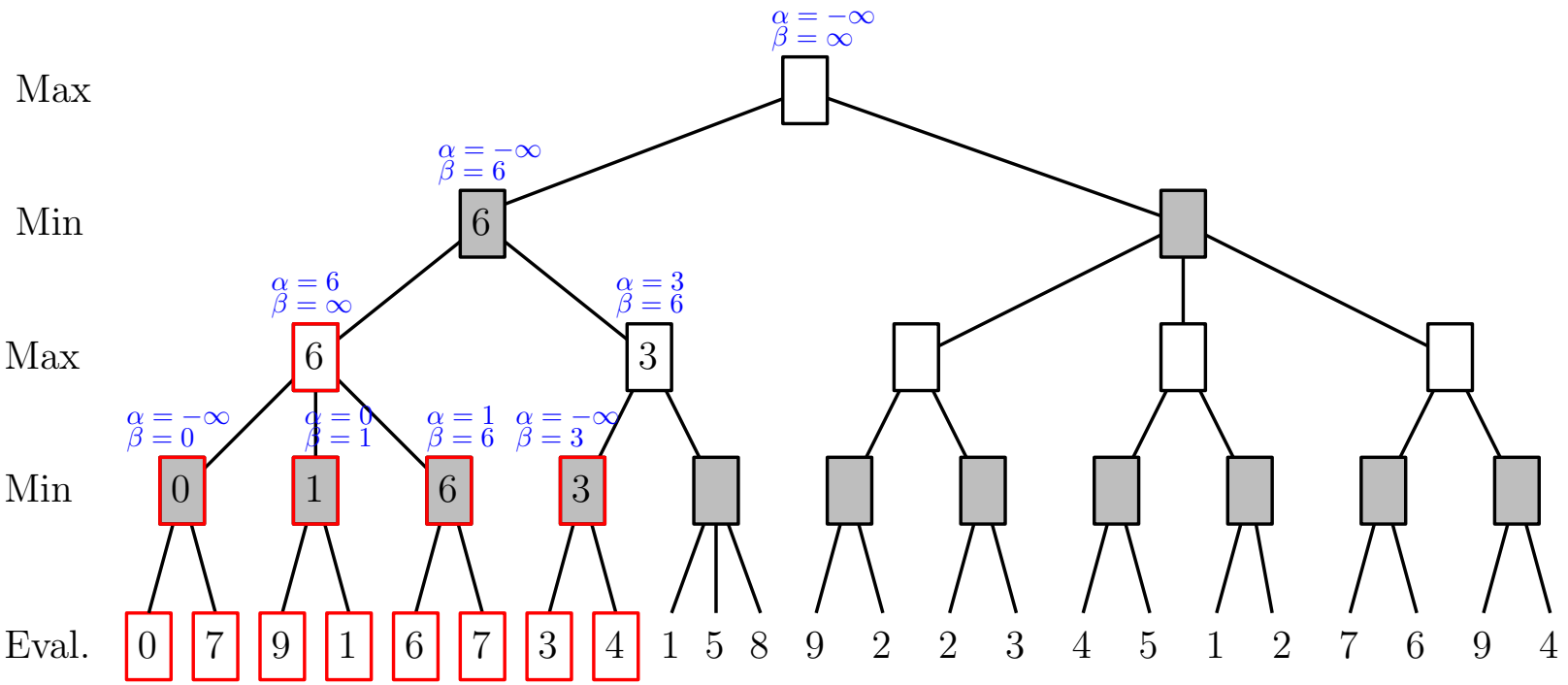


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

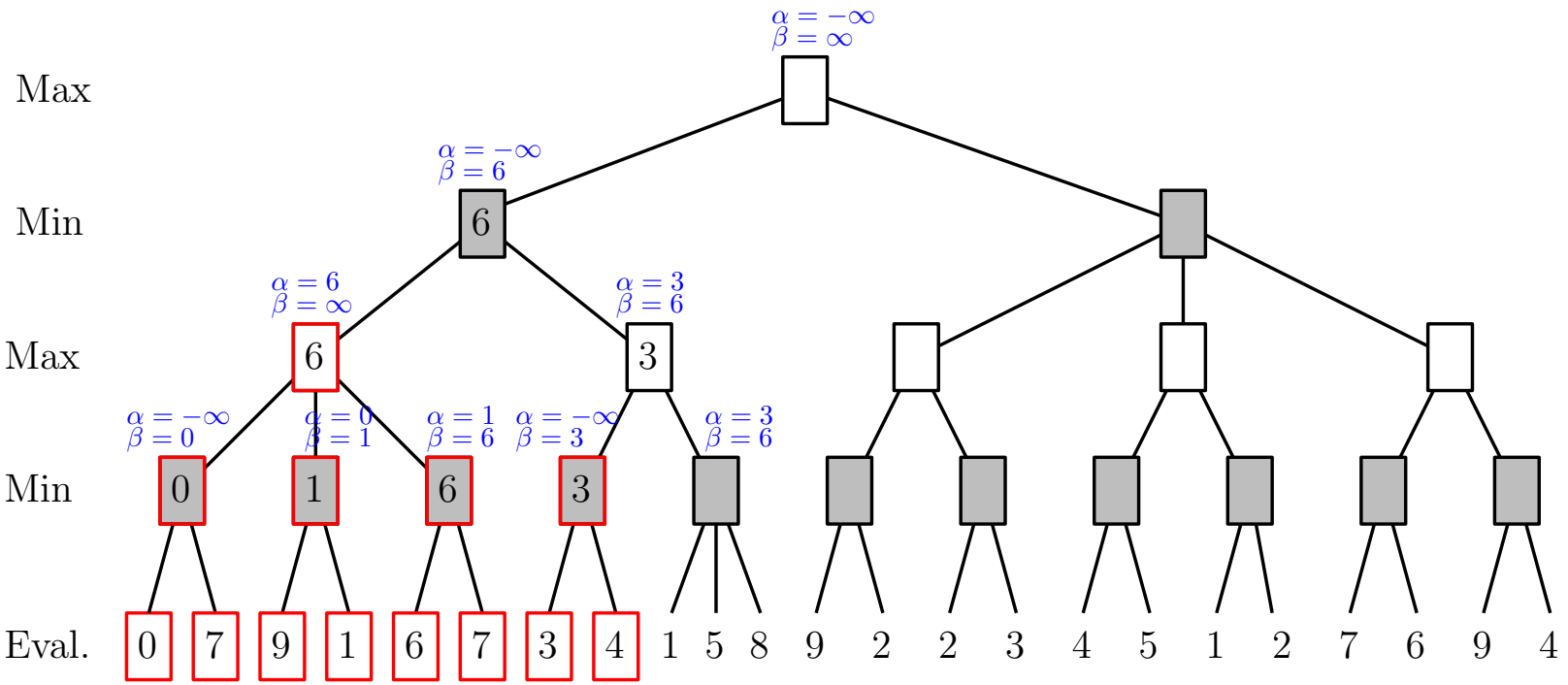


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

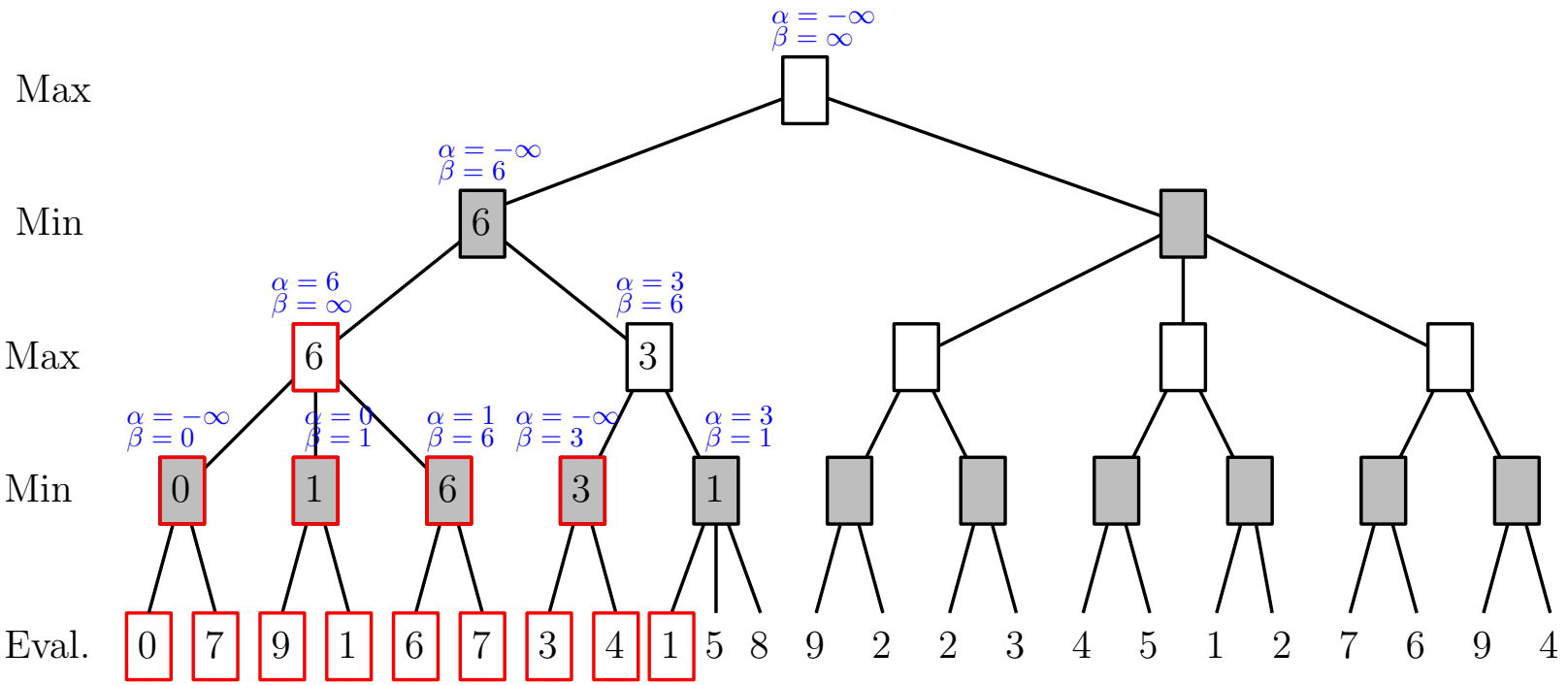


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

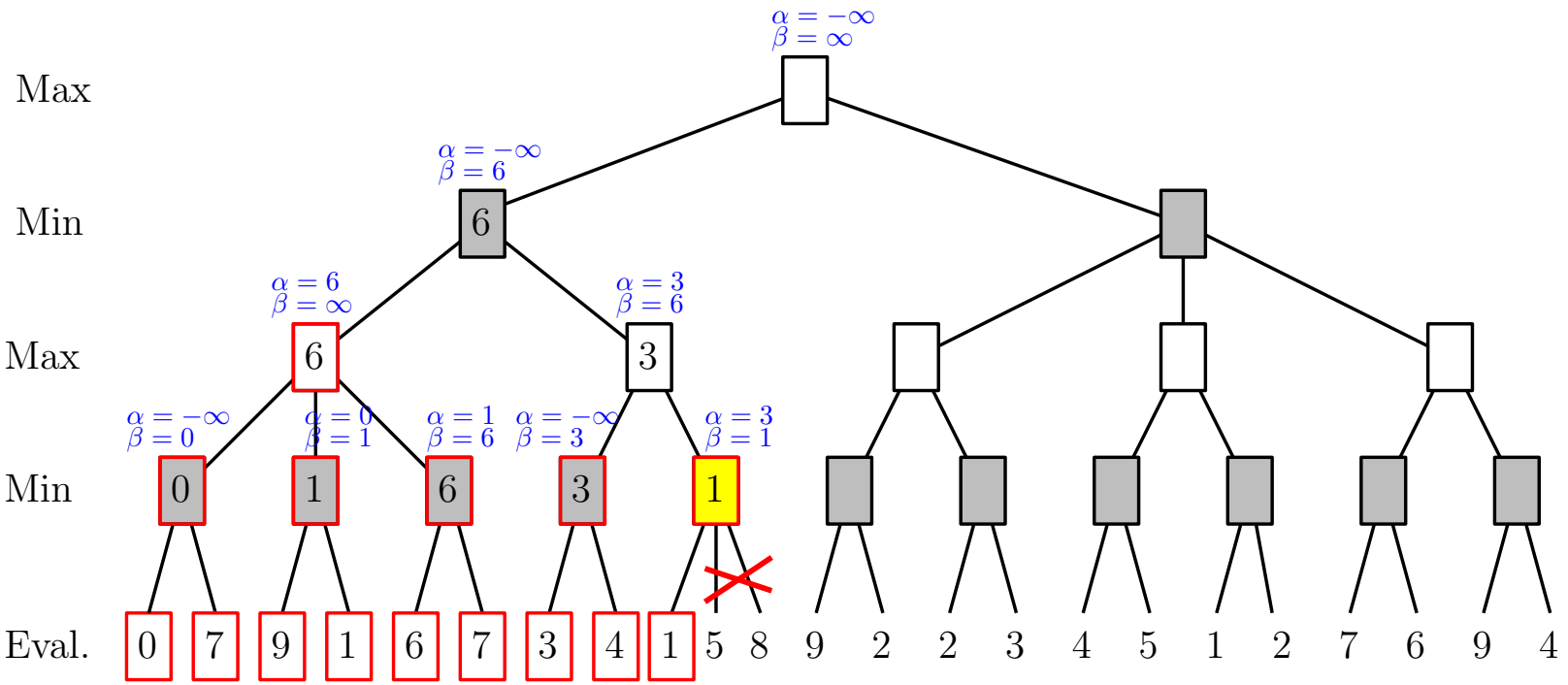


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

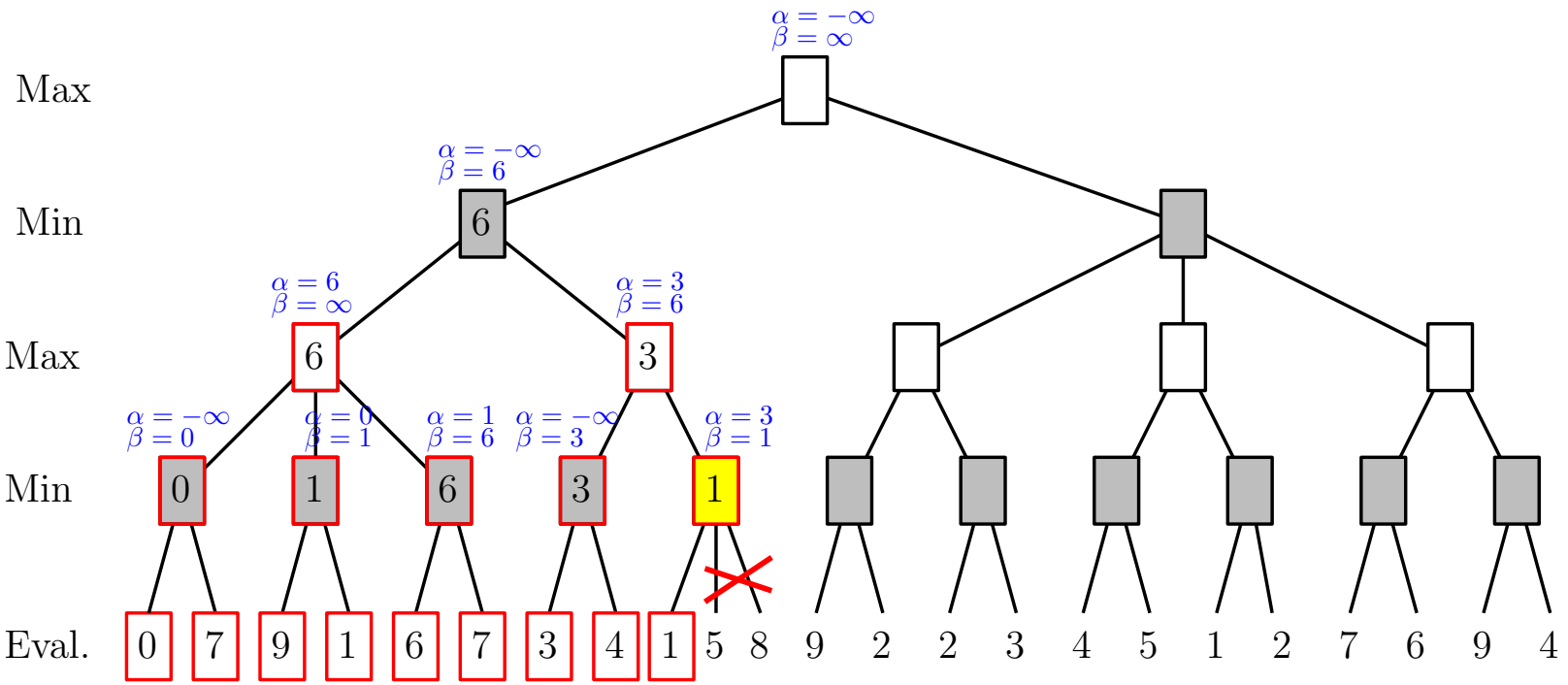


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

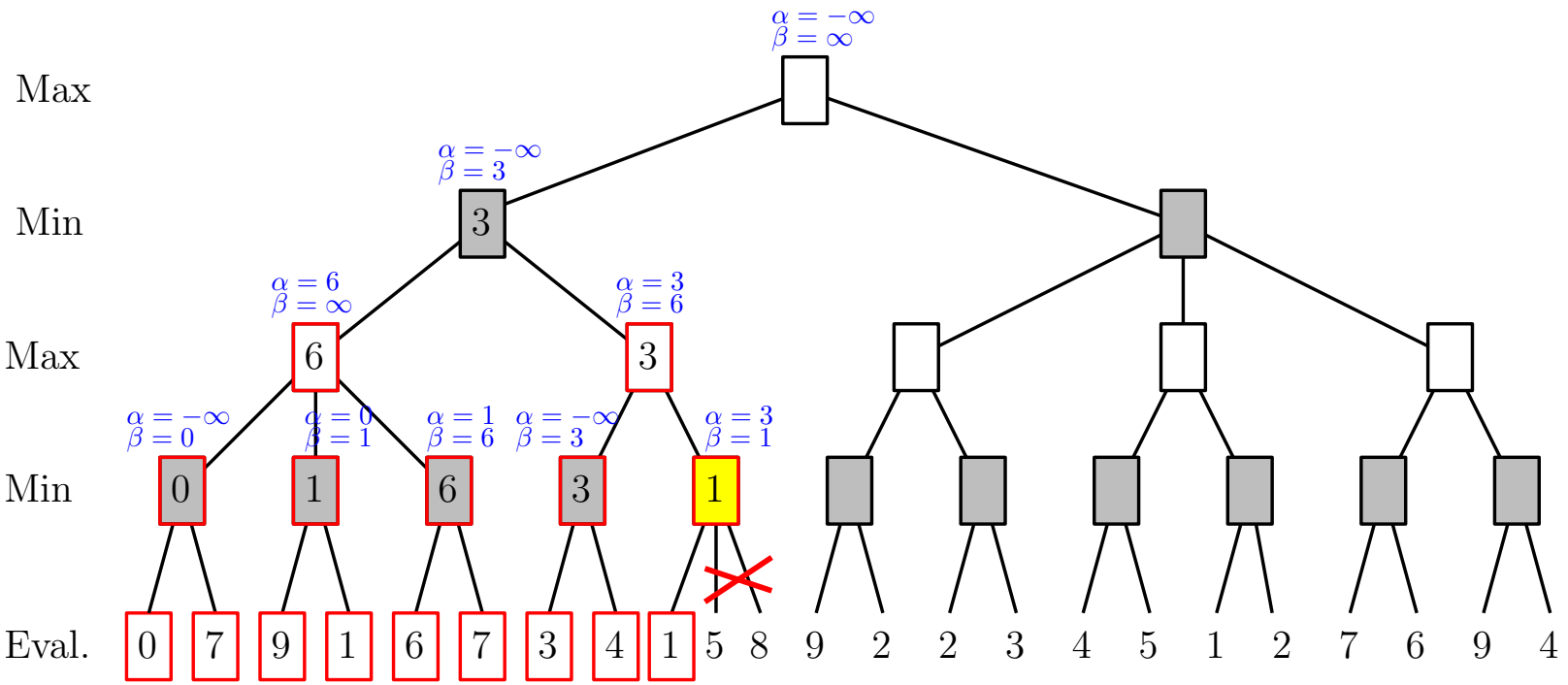


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

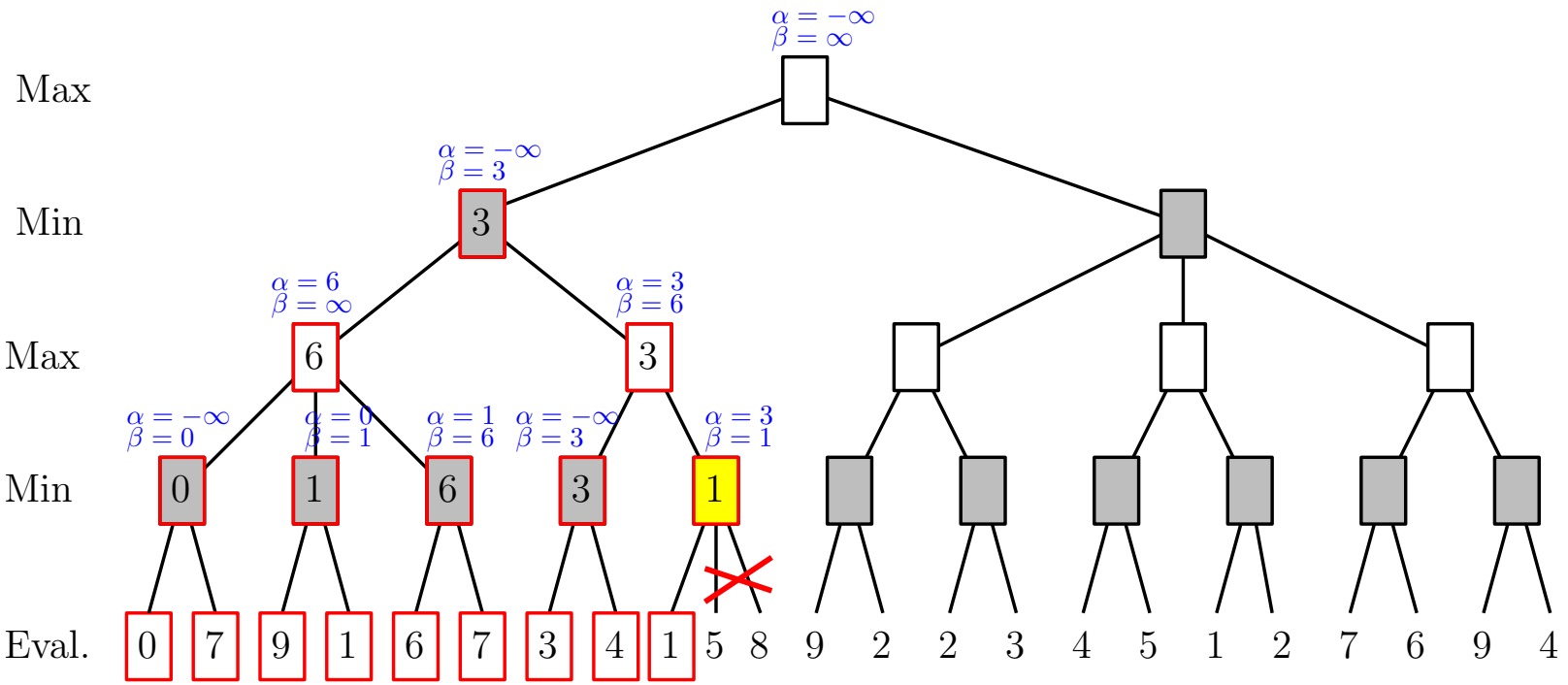


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

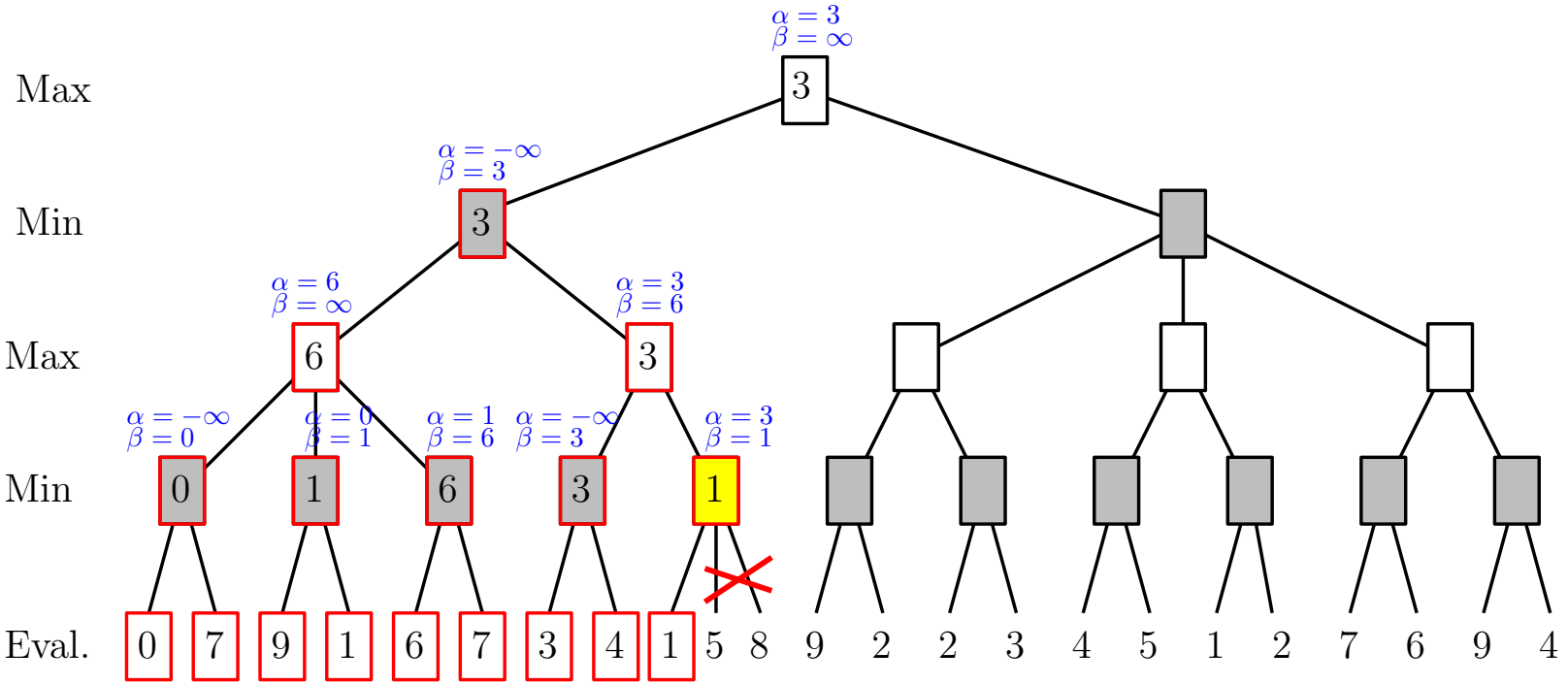


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

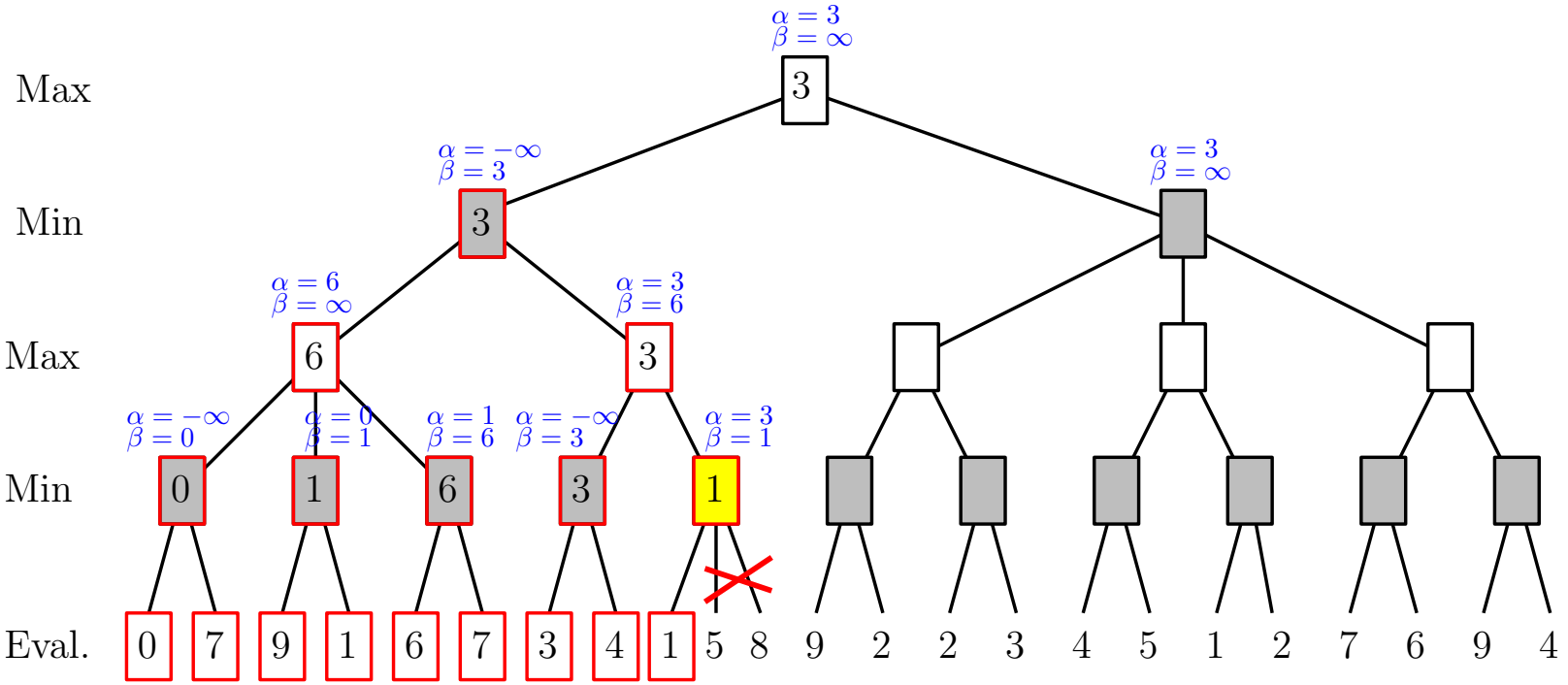


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

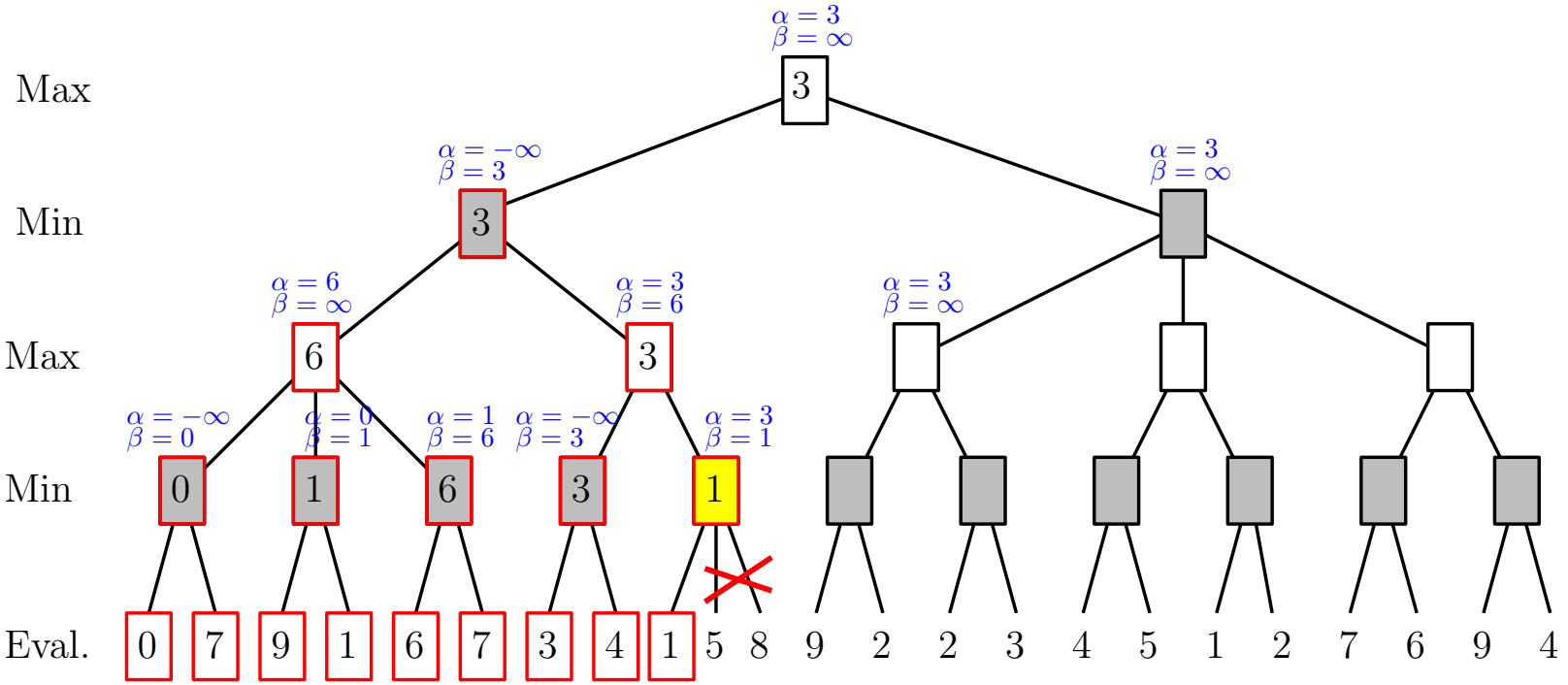


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

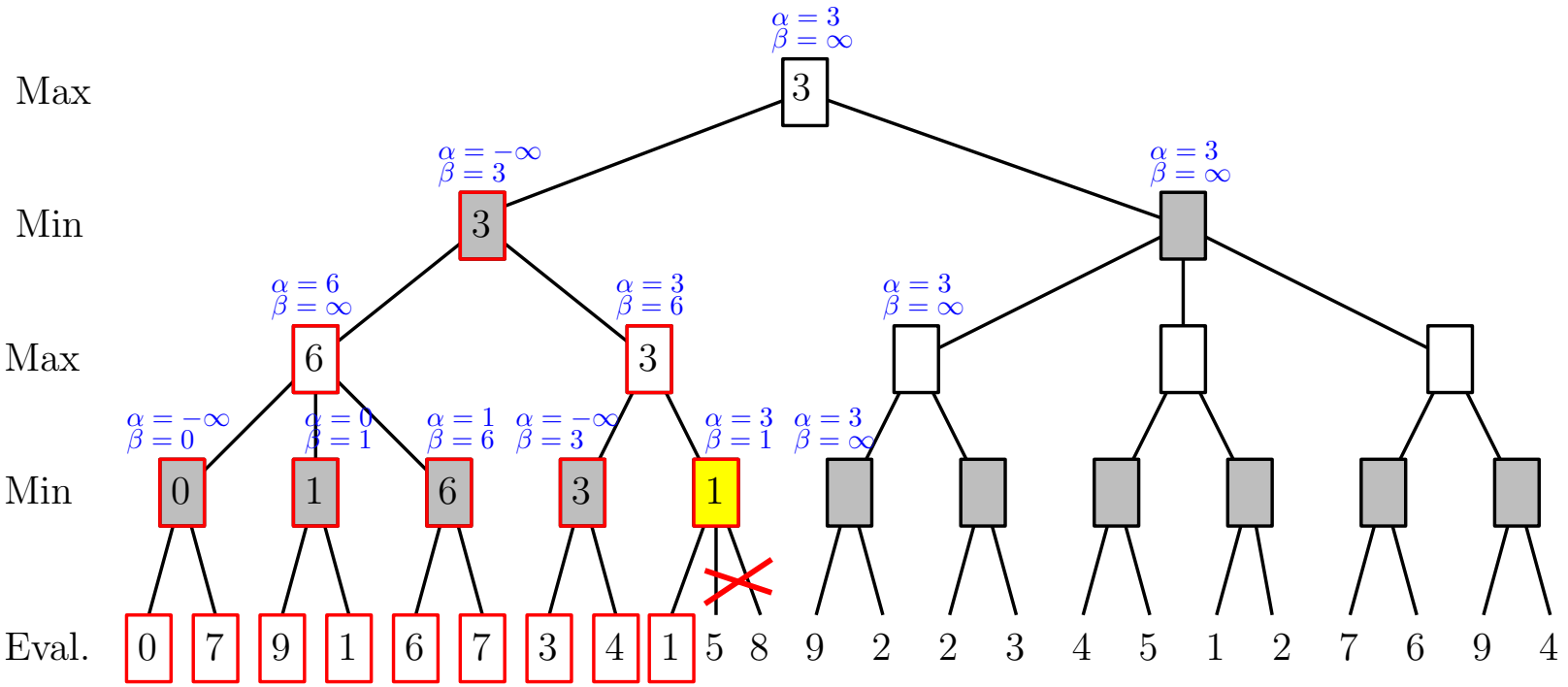


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

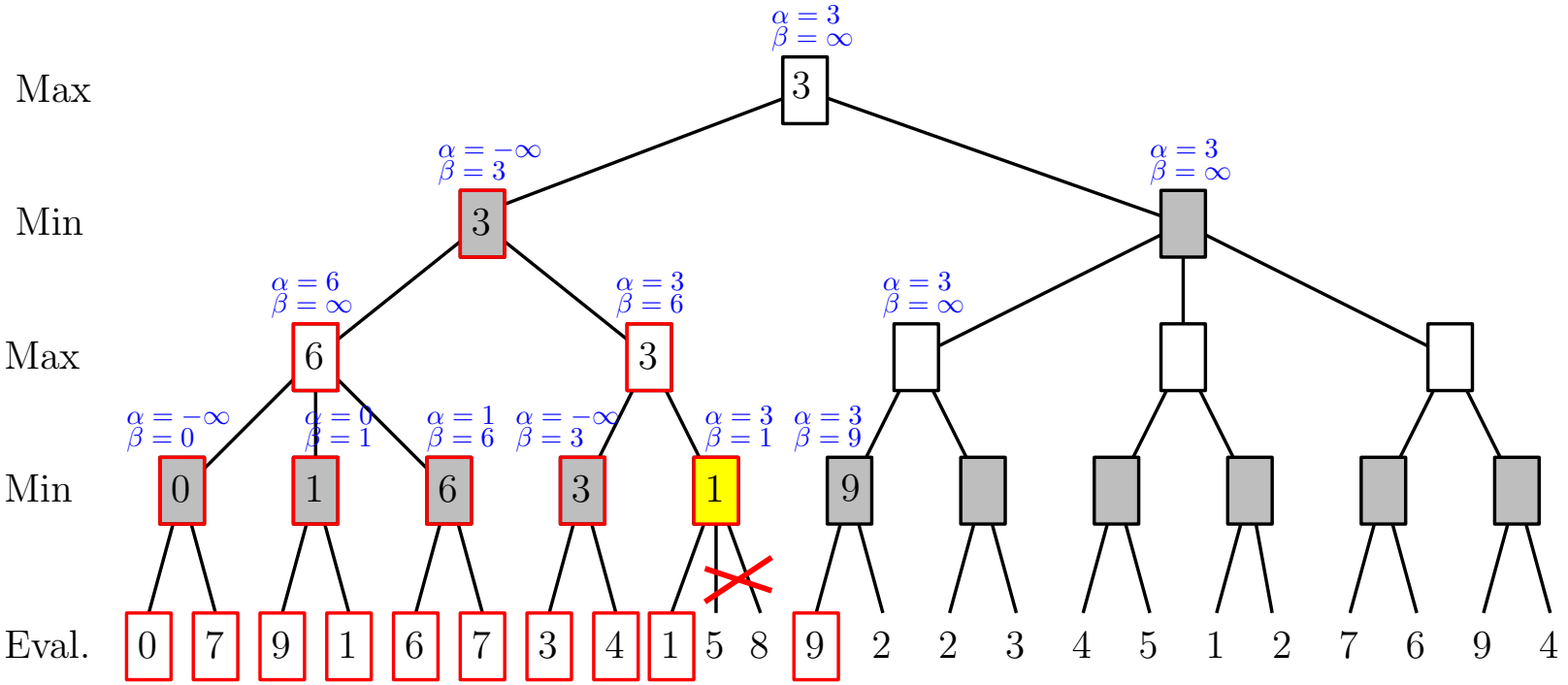


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

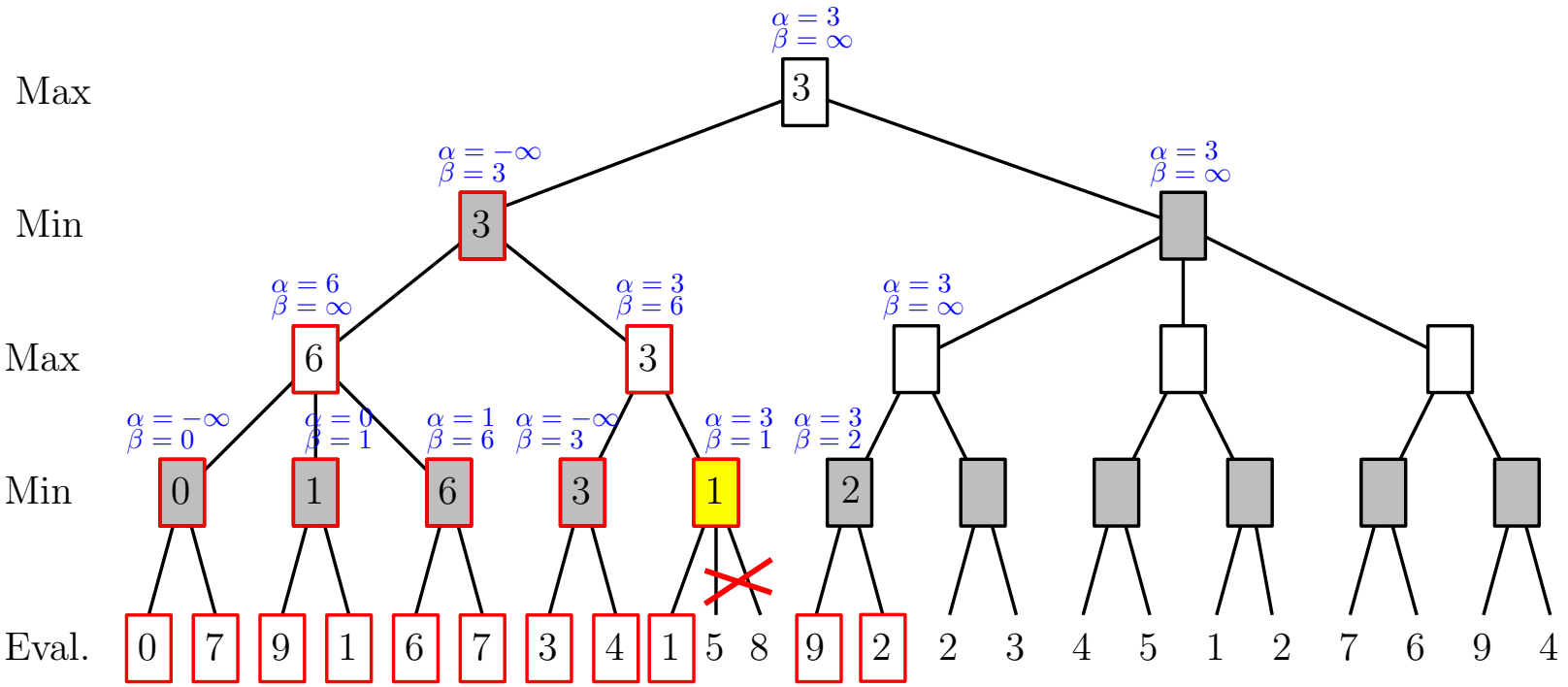


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

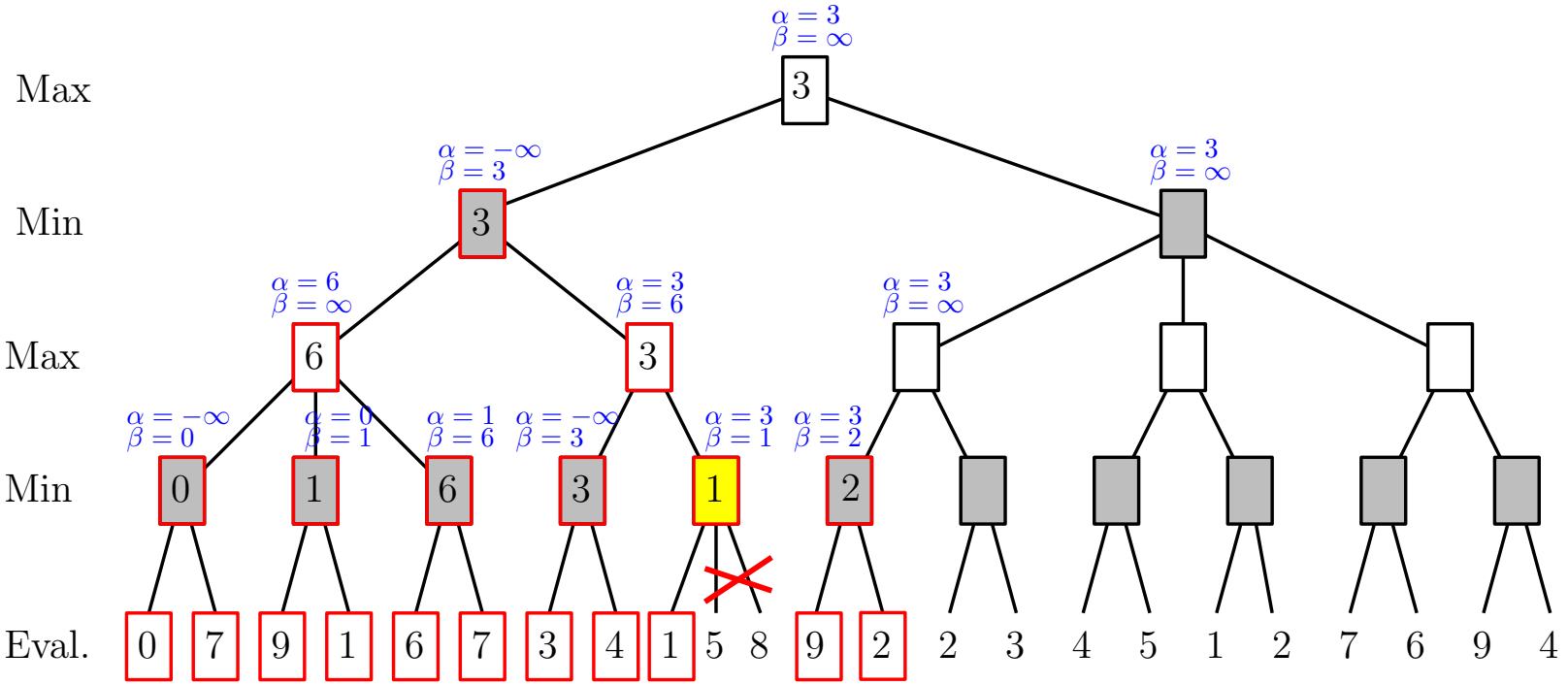


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

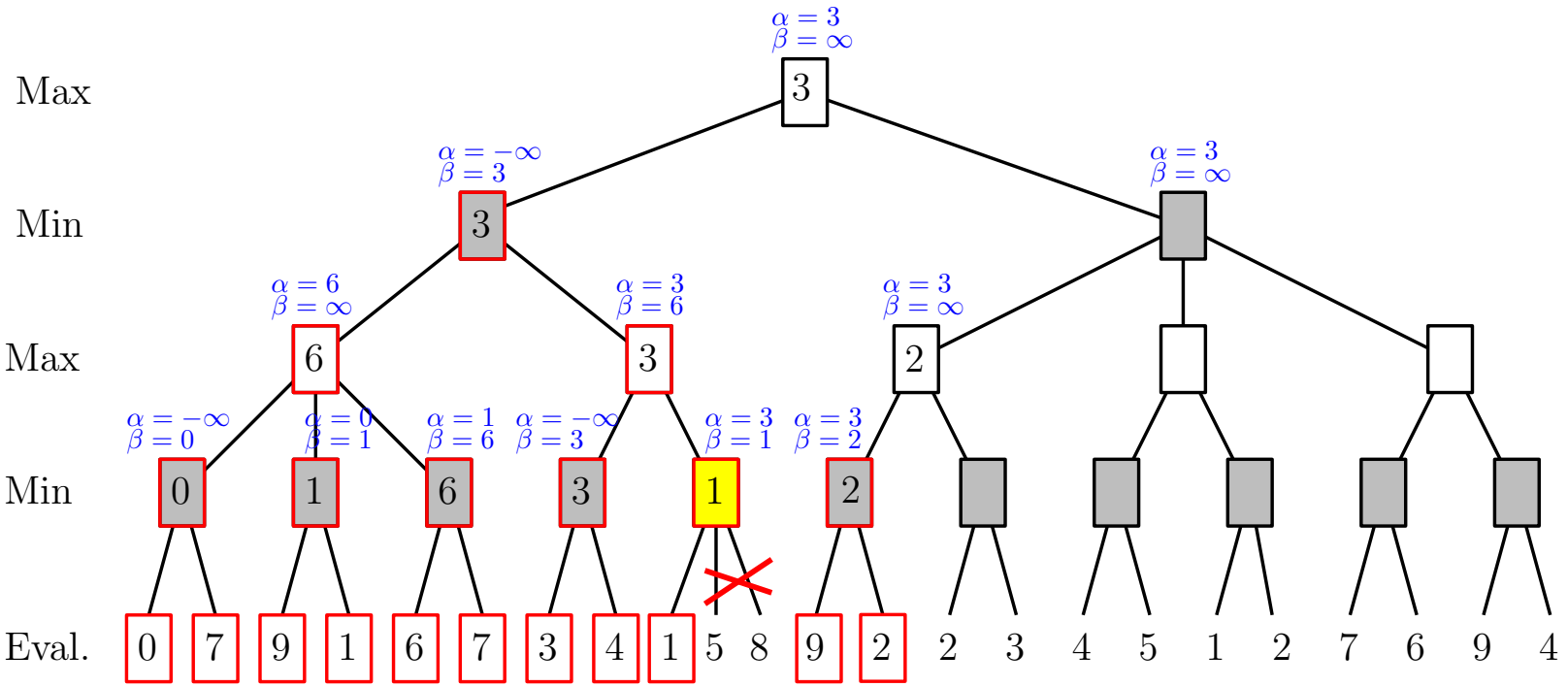


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

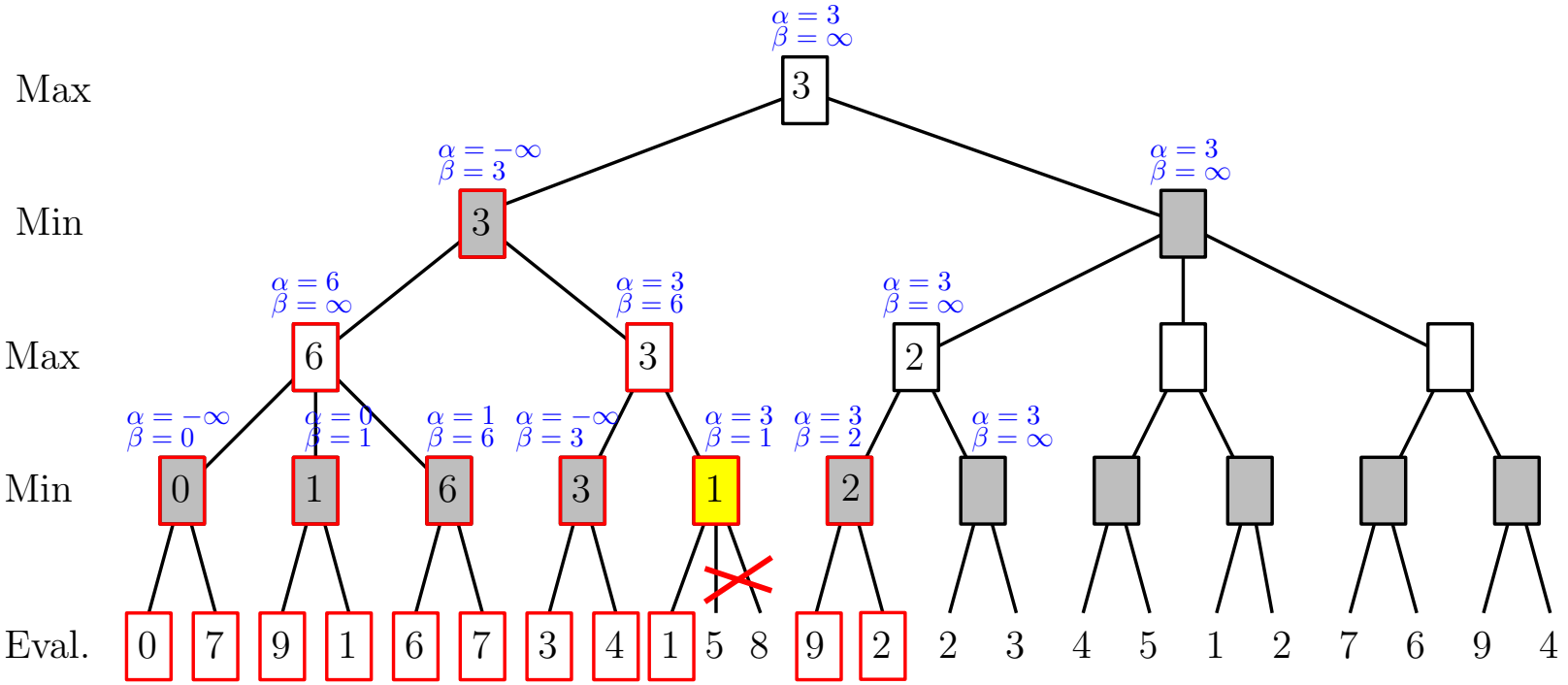


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

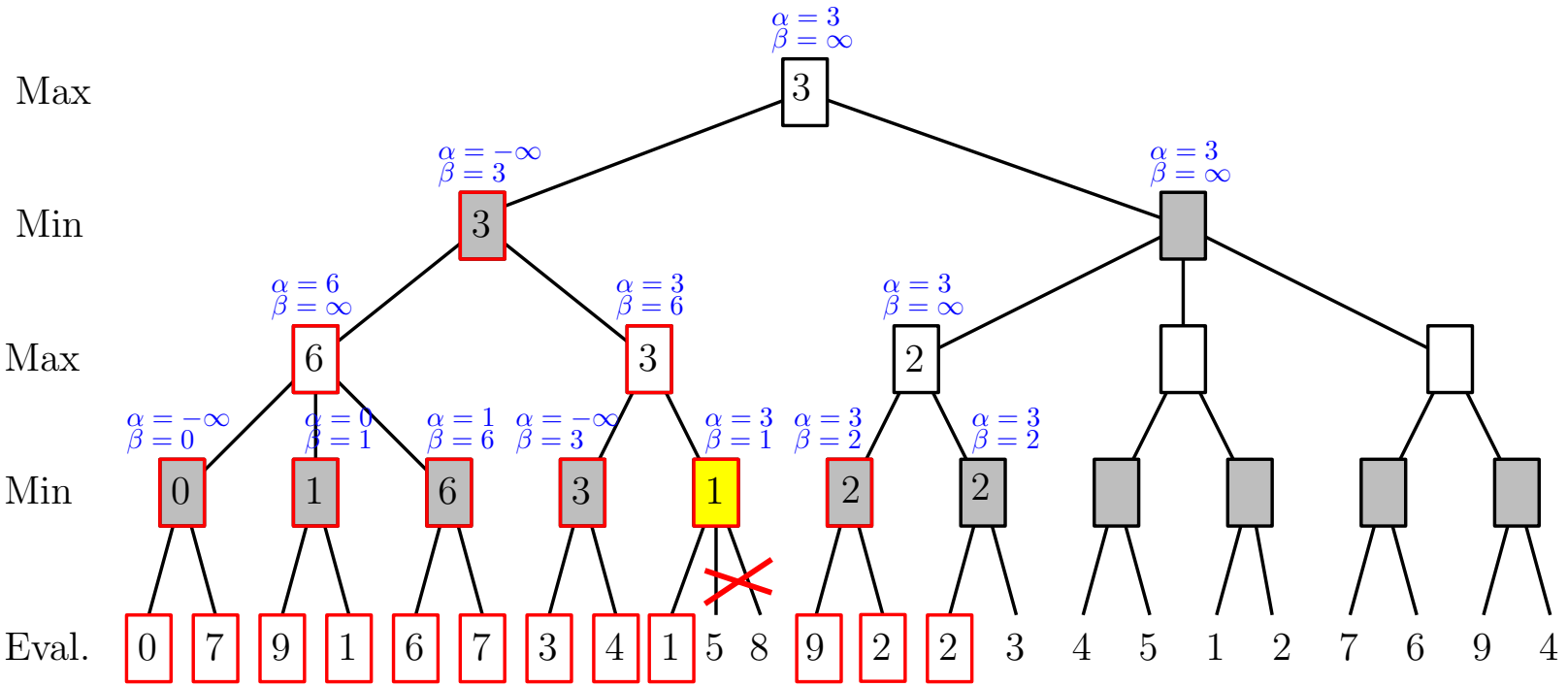


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

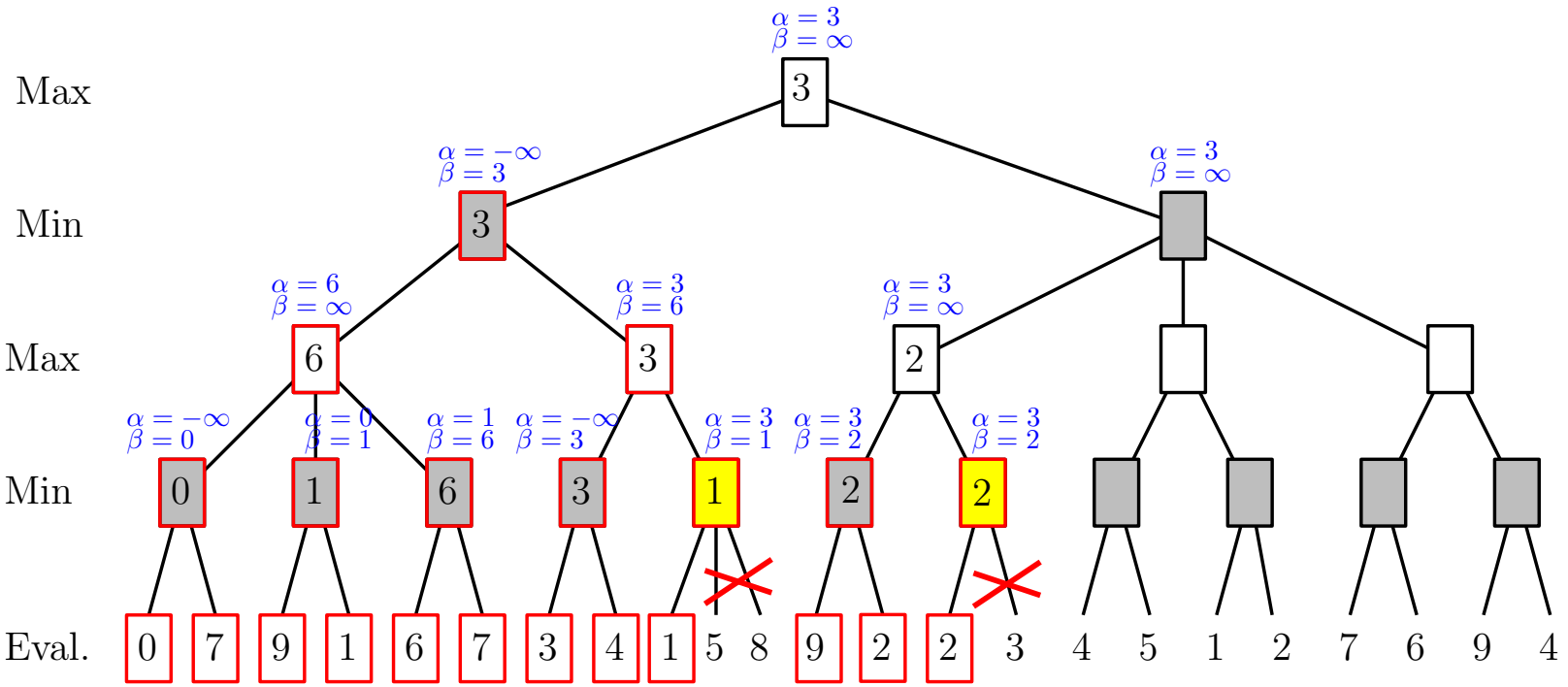


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

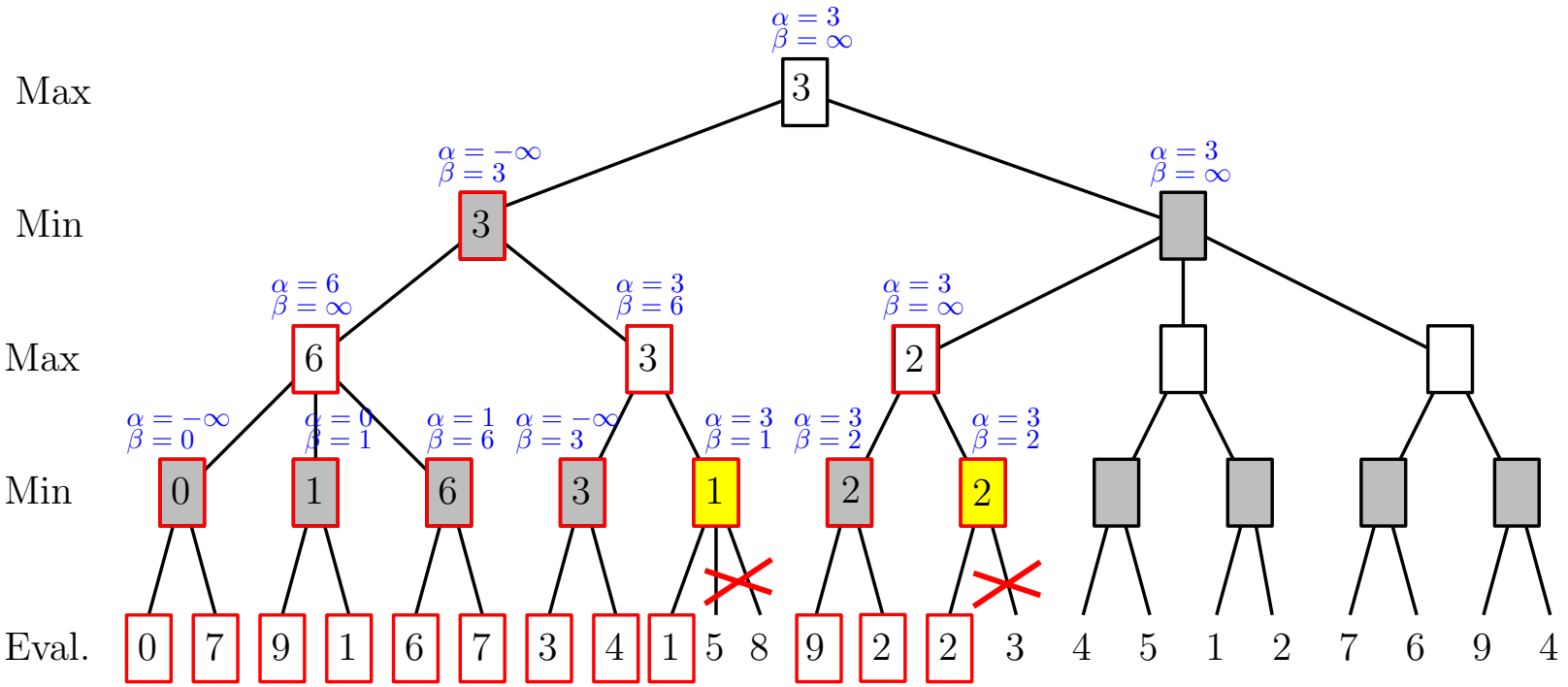


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

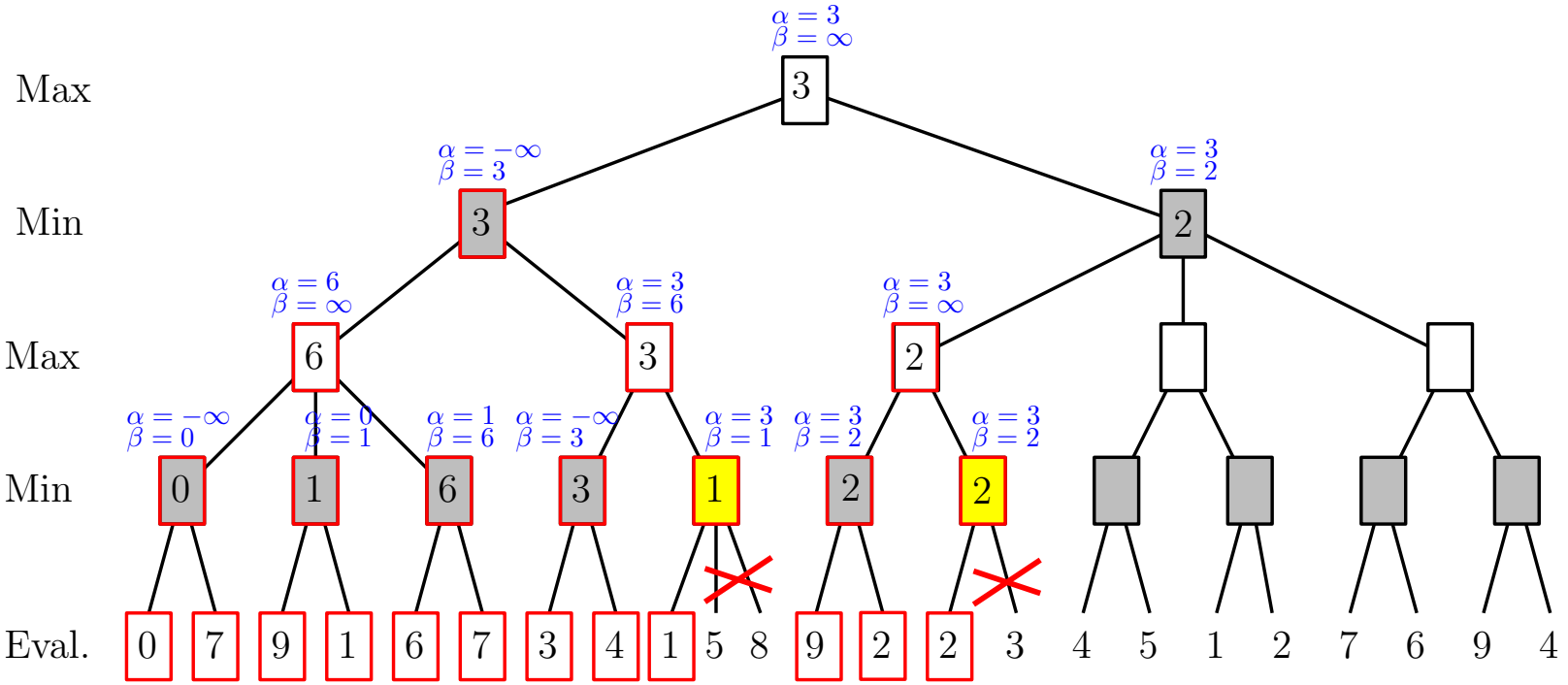


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

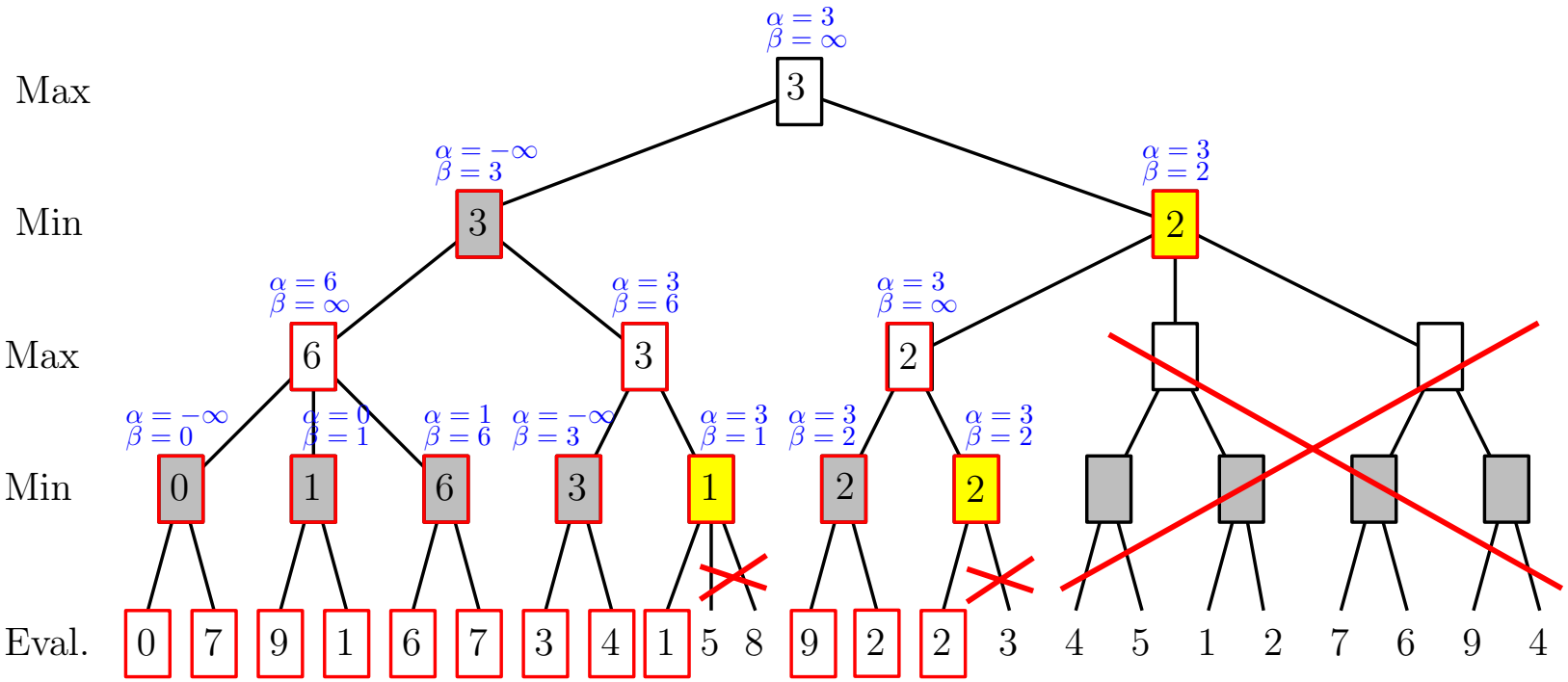


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

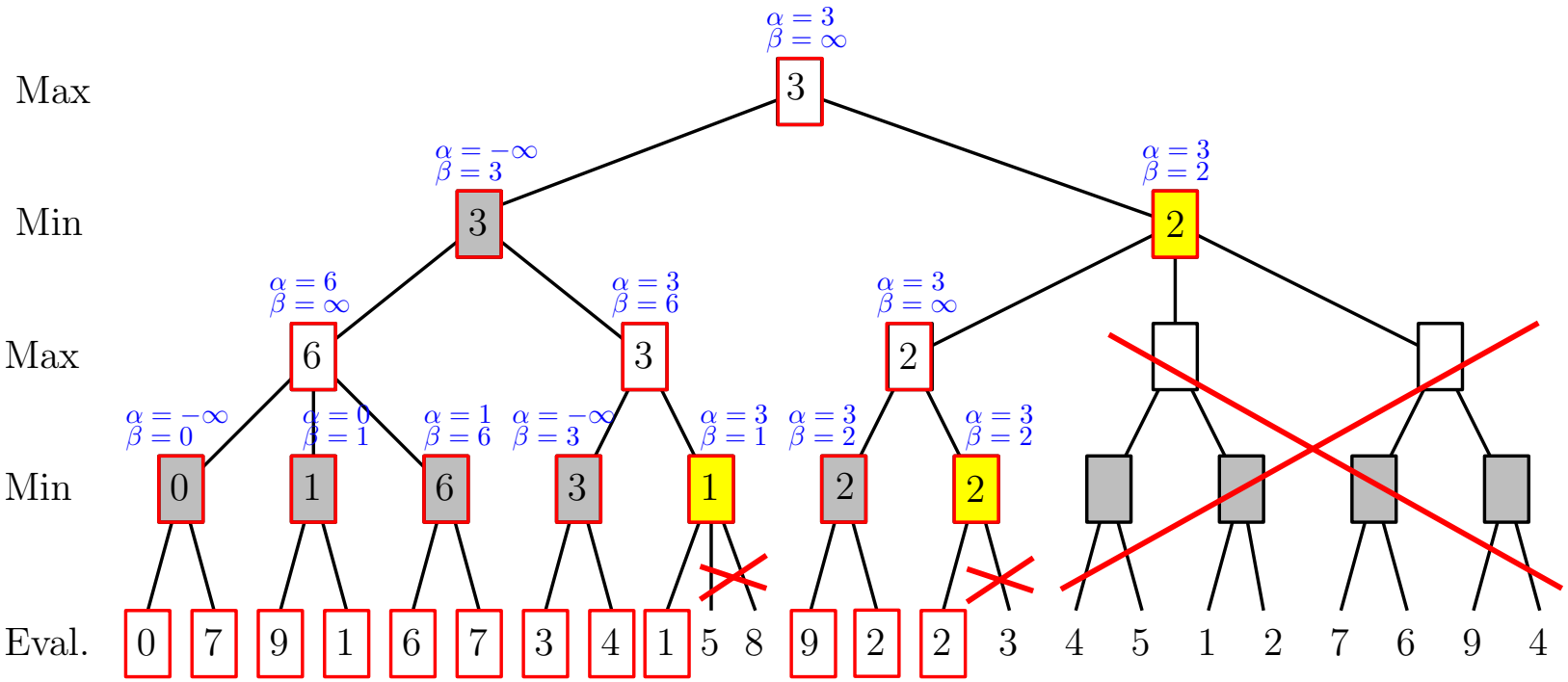


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

Analysis: (see [Pearl 1984])

- *Computation time* heavily depends on *the order in which child nodes are traversed*
- *Worst-Case:* does not offer any advantage.
  - *Successors of maximum nodes are sorted in ascending order, successors of minimum nodes are sorted in descending order*.
  - With constant branching factor b, the number N_d of leaf nodes to evaluate at depth d is $N_d = b^d$.
- *Best-Case:*
  - *Successors of maximum nodes are sorted in descending order, successors of minimum nodes are sorted in ascending order*.
  - Effective branching factor $\approx \sqrt{b} \Rightarrow N_d \approx \sqrt{b}^d = b^{d/2}$ leaf nodes. $\Rightarrow$ Search horizon is doubled. (E.g., if you have computational resources to do a full search with depth d_1 then in the best case, with that same resources, you can search with depth $2d_1$ using alpha-beta pruning.)
  - In chess, this means effective branching factor reduces from 35 to about $6 \approx \sqrt{35}$.

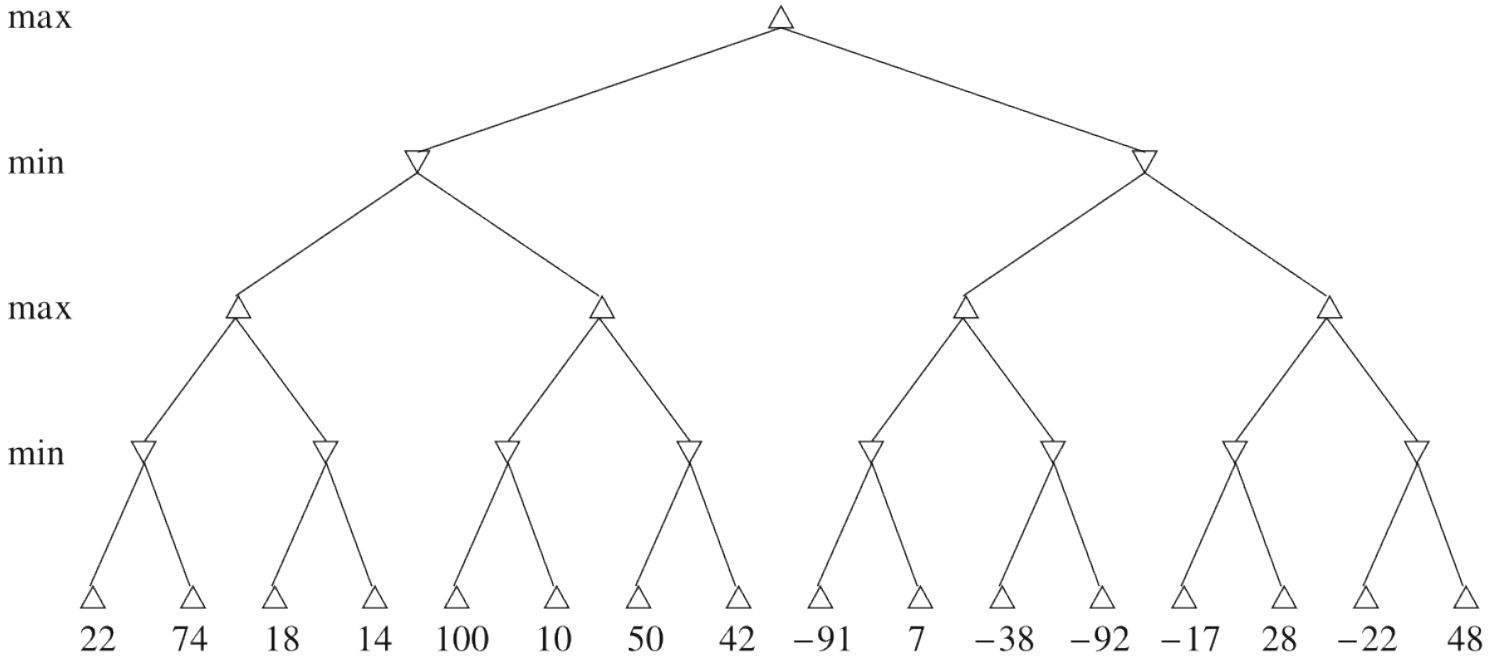
Analysis (cont.):

- *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.

Exercise 16 ([Ertel 2025], Exercise 6.16, p. 127)

(a) The search tree for a two-player game is given in the below figure with the ratings of all leaf nodes. Use minimax search with α-β pruning from left to right. Cross out all nodes that are not visited and give the optimal resulting rating for each inner node. Mark the chosen path.

(b) Test yourself using the following applets:
`http://homepage.ufp.pt/jtorres/ensino/ia/alfabeta.html` and
`https://raphsilva.github.io/utilities/minimax_simulator/`



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

The following example illustrates how to *find good heuristic evaluation functions using the knowledge of human experts* in a chess program [Frayn 2005].

Example 9

- Experts are questioned about the *most important factors in the selection of a move* $\Rightarrow$ these factors are *quantified* $\Rightarrow$ *list of relevant features or attributes*.
- These are then (in the simplest case) combined into a *linear evaluation function* $B(s)$ *for positions*, which could look like

$$\begin{aligned} B(s) &= a_1 \cdot \text{material} + a_2 \cdot \text{pawn\_structure} + \\ &\quad + a_3 \cdot \text{king\_safety} + a_4 \cdot \text{knight\_in\_center} + \\ &\quad + a_5 \cdot \text{bishop\_diagonal\_coverage} + \cdots \\ \text{material} &= \text{material}(\text{own\_team}) - \text{material}(\text{opponent}) \\ \text{material}(\text{team}) &= \text{num\_pawns}(\text{team}) \cdot 100 + \text{num\_knights}(\text{team}) \cdot 300 + \\ &\quad + \text{num\_bishops}(\text{team}) \cdot 300 + \text{num\_rooks}(\text{team}) \cdot 500 + \\ &\quad + \text{num\_queens}(\text{team}) \cdot 900 \end{aligned}$$

- *Weights* a_i *are set intuitively* after discussion with experts + *changed after each game* based on positive and negative experience. *Better:* optimizing weights by *machine-learning methods*.

Example 9 (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 uses both learning and search*: learning trains *NNUE to score chess positions*; during a game, alpha-beta/PVS search uses these scores to choose a move [Stockfish developers 2020].

Search, Evaluation, and Learning in Games

Key Milestones

1950 *Claude Shannon* proposed limited game-tree search guided by position evaluation [Shannon 1950].

1955–59 *Arthur Samuel's* checkers program improved its evaluation through learning and self-play [Samuel 1959].

1992 *TD-Gammon* learned a strong backgammon evaluation through temporal-difference learning and self-play [Tesauro 1992].

1997 *Deep Blue* combined large-scale search with complex hand-designed evaluation to defeat world chess champion Garry Kasparov [Campbell, Hoane, and Hsu 2002].

Progression

The progression is from *hand-designed evaluation* (humans define the score) to *learned evaluation* (a model learns it from games); search remains central.

Search, Evaluation, and Learning in Games

Why Go Is Difficult for Full-Width Search

- Go is played on a 19×19 grid with *361 intersections*, where players place black and white stones [New Zealand Go Society 2017].
- A typical position has about *250 legal moves*, and a typical game lasts about *150 moves* [Silver et al. 2016].
- Expanding every continuation to that depth is infeasible: the full game tree grows roughly like 250^{150}.
- Position evaluation is also difficult: local patterns can affect distant parts of the board.

Consequence

A practical algorithm must *allocate computation selectively*: explore promising moves more often instead of expanding the full tree uniformly.

Search, Evaluation, and Learning in Games

Minimax, Alpha-Beta, and MCTS

Minimax and alpha-beta

- They choose a move by assuming both players make their best choices.
- Minimax backs up alternating *maximum and minimum values*.
- Alpha-beta skips *branches irrelevant to the minimax result*.

Monte Carlo tree search

- MCTS chooses a move by sampling possible future plays.
- MCTS *grows the tree selectively* through repeated trials.
- It balances *promising choices* with less-explored choices.

One MCTS iteration

Select a path ⇒ *expand* one child ⇒ estimate its outcome ⇒ *update* the path's statistics.

MCTS is *selective tree search*: it chooses where to spend computation [Browne et al. 2012].

Search, Evaluation, and Learning in Games

Neural-Guided Search

- *AlphaGo (2015–16)* first learned from expert human games and then improved through self-play. Its policy and value networks guided MCTS. It defeated Fan Hui 5–0 in October 2015 and Lee Sedol 4–1 in March 2016 [Silver et al. 2016]; [Google DeepMind n.d.]
- *AlphaGo Zero (2017)* learned Go solely through self-play, without human game data [Silver et al. 2017].
- *AlphaZero (2018)* applied the same general approach in separately trained systems for chess, shogi, and Go [Silver et al. 2018].
- *Stockfish* illustrates a different modern combination: alpha-beta/PVS search with a learned NNUE evaluator [Stockfish developers 2020].

Main lesson

Learning supplies evaluation or search guidance; *search allocates computation among possible futures*.

References

- Ertel, Wolfgang (2025). *Introduction to Artificial Intelligence*. 3rd. Springer. DOI: 10.1007/978-3-658-43102-0.
- 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.
- Stockfish developers (2020). *Introducing NNUE Evaluation*. URL: https://stockfishchess.org/blog/2020/introducing-nnue-evaluation/ (visited on 07/12/2026).

References (cont.)

Silver, David, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. (2018). "A General Reinforcement Learning Algorithm that Masters Chess, Shogi, and Go through Self-Play." In: *Science* 362.6419, pp. 1140–1144. DOI: 10.1126/science.aar6404.

New Zealand Go Society (2017). *New Zealand Rules of Go*. URL: https://go.org.nz/index.php/about-go/new-zealand-rules-of-go (visited on 07/12/2026).

Silver, David, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. (2017). "Mastering the Game of Go without Human Knowledge." In: *Nature* 550.7676, pp. 354–359. DOI: 10.1038/nature24270.

References (cont.)

Batzill, Adrian (2016). “Optimal route planning on mobile systems.” MA thesis. Masterarbeit, Hochschule Ravensburg Weingarten.

Silver, David, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. (2016). “Mastering the game of Go with deep neural networks and tree search.” In: *nature* 529.7587, pp. 484–489. DOI: 10.1038/nature16961.

References (cont.)

Browne, Cameron B., Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton (2012). "A Survey of Monte Carlo Tree Search Methods." In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1, pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

Russell, Stuart J. and Peter Norvig (2010). *Artificial Intelligence: A Modern Approach*. 3rd. Pearson.

Geisberger, Robert, Peter Sanders, Dominik Schultes, and Daniel Delling (2008). "Contraction hierarchies: Faster and simpler hierarchical routing in road networks." In: *Experimental Algorithms: 7th International Workshop, WEA 2008 Provincetown, MA, USA, May 30-June 1, 2008 Proceedings 7*. Springer, pp. 319–333. DOI: 10.1007/978-3-540-68552-4_24.

References (cont.)

Frayn, Colin (2005). *Computer Chess Programming Theory*. URL: http://www.frayn.net/beowulf/theory.html.

Campbell, Murray, A. Joseph Hoane Jr., and Feng-hsiung Hsu (2002). "Deep Blue." In: *Artificial Intelligence* 134.1–2, pp. 57–83. DOI: 10.1016/S0004-3702(01)00129-1.

Nilsson, Nils J. (1998). *Artificial Intelligence: A New Synthesis*. Morgan Kaufmann.

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

References (cont.)


Tesauro, Gerald (1992). “Practical Issues in Temporal Difference Learning.” In: *Advances in Neural Information Processing Systems 4*. Morgan Kaufmann, pp. 259–266. URL: https://proceedings.neurips.cc/paper/1991/hash/68ce199ec2c5517597ce0a4d89620f55-Abstract.html.


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.

Samuel, Arthur L. (1959). "Some Studies in Machine Learning Using the Game of Checkers." In: *IBM Journal of Research and Development* 3.3, pp. 210–229. DOI: 10.1147/rd.33.0210.

Shannon, Claude E. (1950). "Programming a Computer for Playing Chess." In: *Philosophical Magazine* 41.314, pp. 256–275. DOI: 10.1080/14786445008521796.

Google DeepMind (n.d.). *AlphaGo*. URL: https://deepmind.google/research/alphago/ (visited on 07/12/2026).