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

Contents

Additional Materials

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

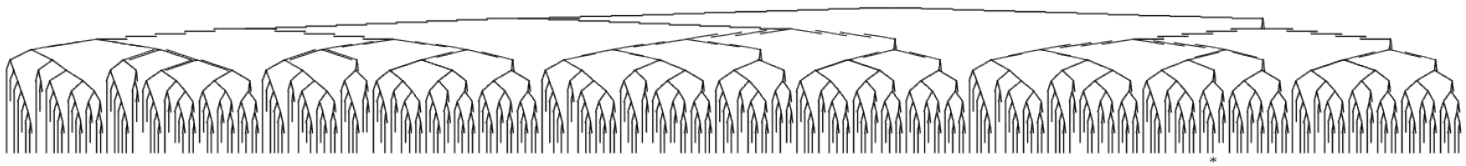
- https://youtu.be/RRO9-QXROss&t=2210 (Introduction)
- https://youtu.be/rwefoi__Fk4 (Uninformed Search: Breadth-First Search, Depth-First Search, Iterative Deepening)
- https://youtu.be/THZ3YxHAwno (Heuristic Search: Greedy Search, A*-Search, IDA*-Search)
- https://youtu.be/IW-HIOPqgsk (Games with Opponents,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

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*.

Introduction

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])

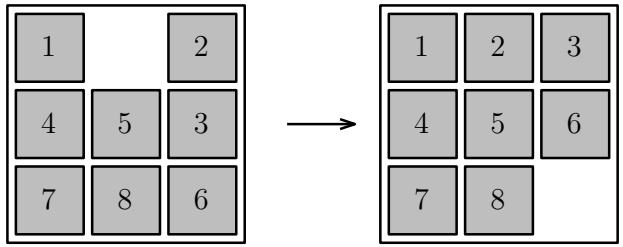


Figure: Possible starting and goal states of the 8-puzzle

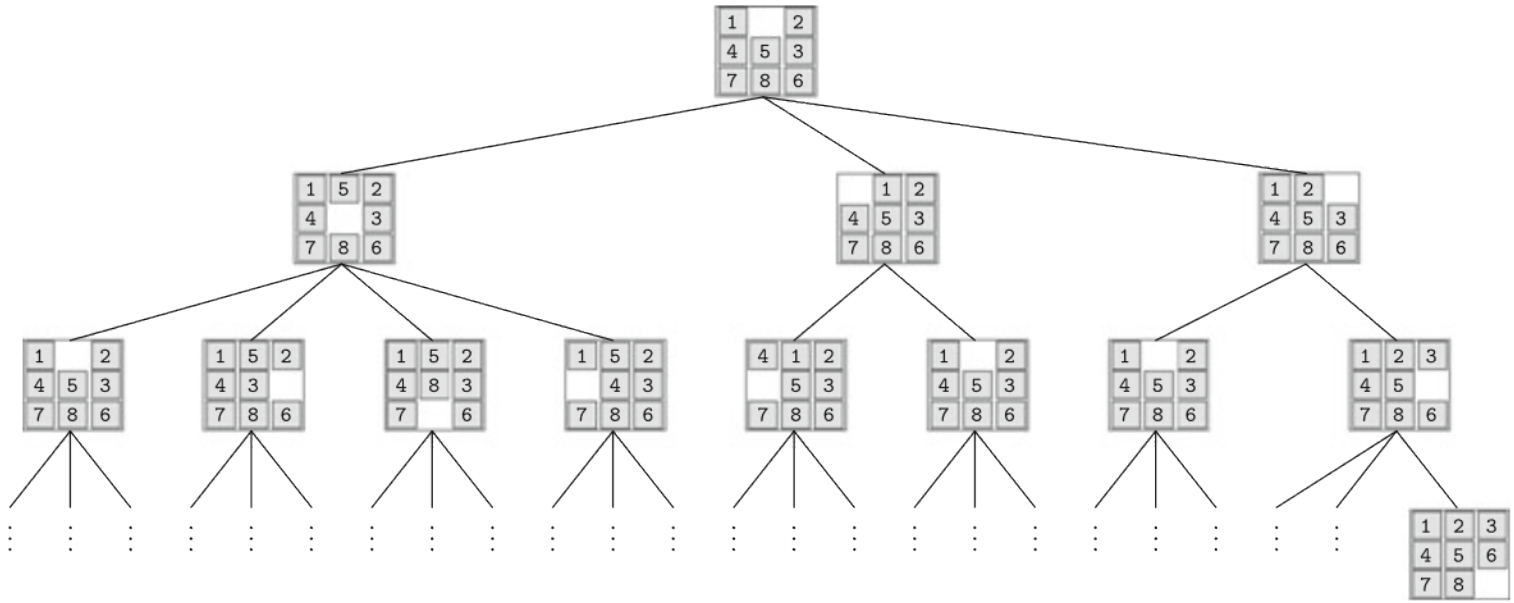


Figure: Search tree for the 8-puzzle. Bottom right a goal state in depth 3 is represented. To save space the other nodes at this level have been omitted

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*.

Introduction

Example 2 (8-Puzzle, cont.)

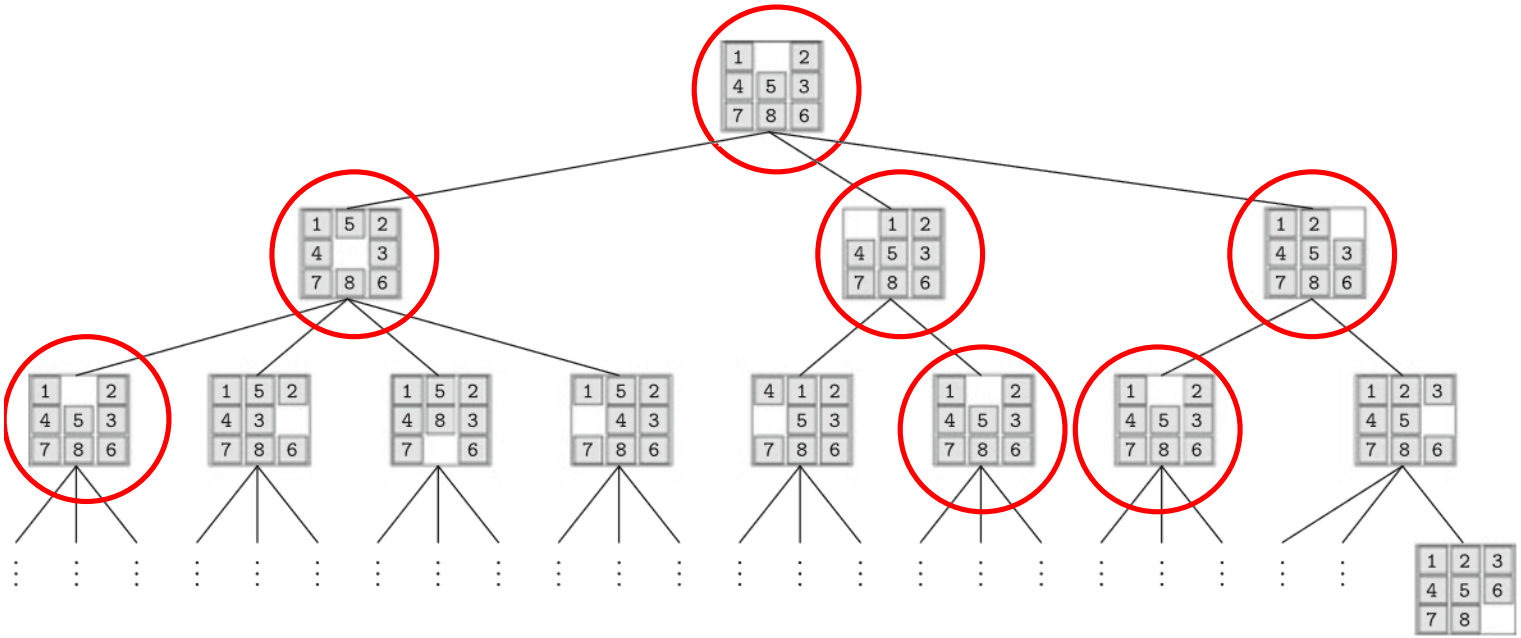


Figure: Cycles of length 2 in the search tree are somewhat redundant.

Introduction

Example 2 (8-Puzzle, cont.)

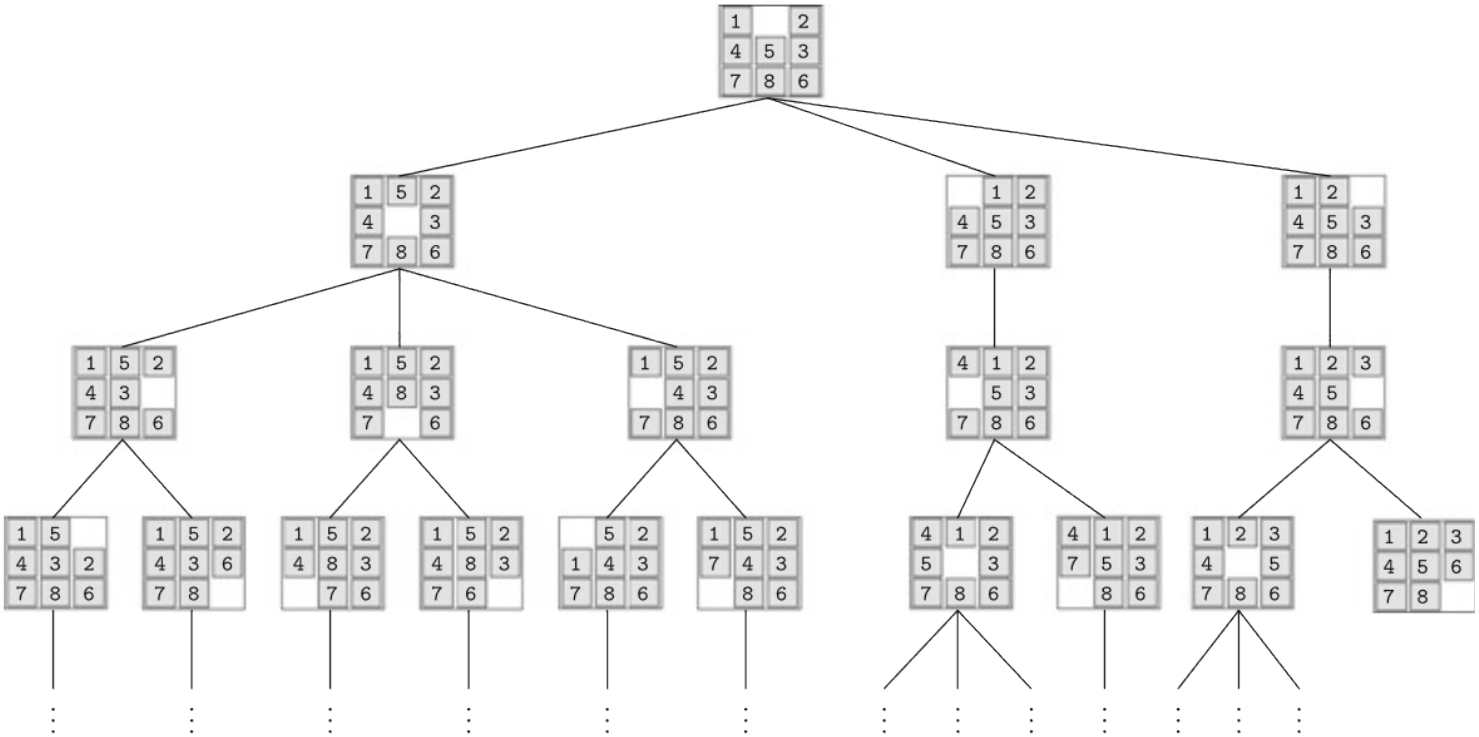


Figure: Search tree for an 8-puzzle without cycles of length 2.

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 state of the world in which the search agent finds itself.

Starting state: The initial state in which the search agent is started.

Goal state: If the agent reaches a goal state, then it terminates and outputs a solution (if desired).

Actions: All of the agents allowed actions.

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

Cost function: Assigns a cost value to every action. Necessary for finding a cost-optimal solution.

State space: Set of all states.

Search tree: States are nodes, actions are edges.

Example 2 (8-Puzzle, cont.)

Apply the definition to the 8-puzzle, we get

State: 3×3 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 degenerate in domains that are mutually unreachable. (Thus there are unsolvable 8-puzzle problems.)

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.

Introduction

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 (nodes) 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)

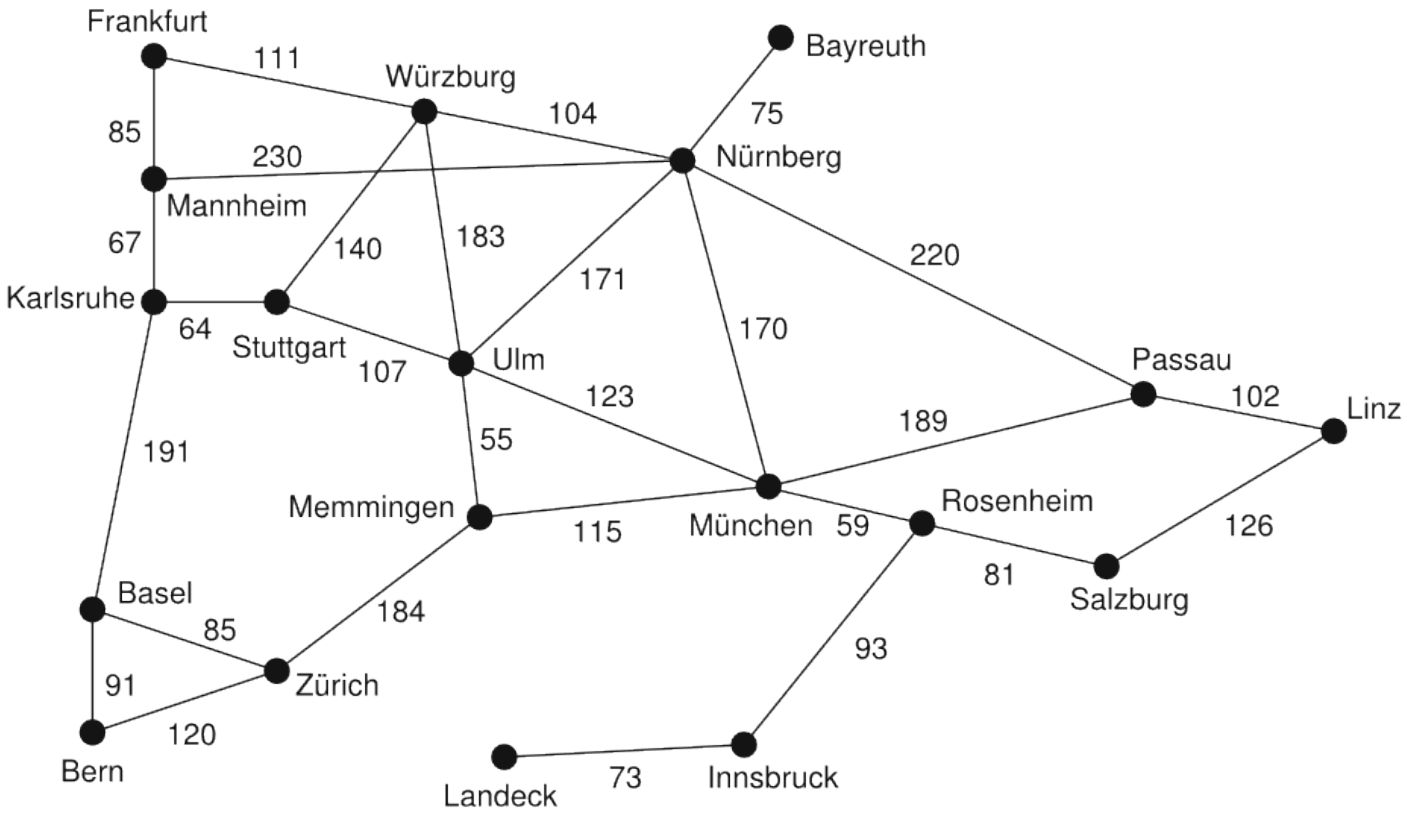


Figure: The graph of southern Germany as an example of a search task with a cost function.

Example 3 (cont.)

State: 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, that is, nodes of the 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 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*.
- Deterministic and observable problems *make action planning relatively simple* because, due to *having an abstract model*, it is possible to *find action sequences for the solution of the problem without actually carrying out the actions in the real world*. $\Rightarrow$ Offline algorithms.
- One faces *much different challenges* when, for example, *building robots that are supposed to play soccer*. *Here there will never be an exact abstract model of the actions.* $\Rightarrow$ Online algorithms (which make decisions based on sensor signals in every situation). (*Reinforcement learning* works toward optimization of these decisions based on experience.)

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) calculates whether Node is a goal node Goal.
- Successors(Node) returns the list of all successor nodes of Node.
- The search tree is explored from top to bottom 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.

Analysis:

- complete.
  - BFS can reach every depth in finite time $\Rightarrow$ If there is a solution, BFS will find it.
- optimal, if all costs of all actions are the same.
  - Otherwise, BFS may not find a solution with the lowest cost.
- 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 all costs are not the same, we use a variant of BFS.

Uniform Cost Search

= BFS + The node with the lowest cost from the list of nodes (which is sorted ascendingly by cost) is always expanded, and the new nodes sorted in. $\Rightarrow$ Always optimal!

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.

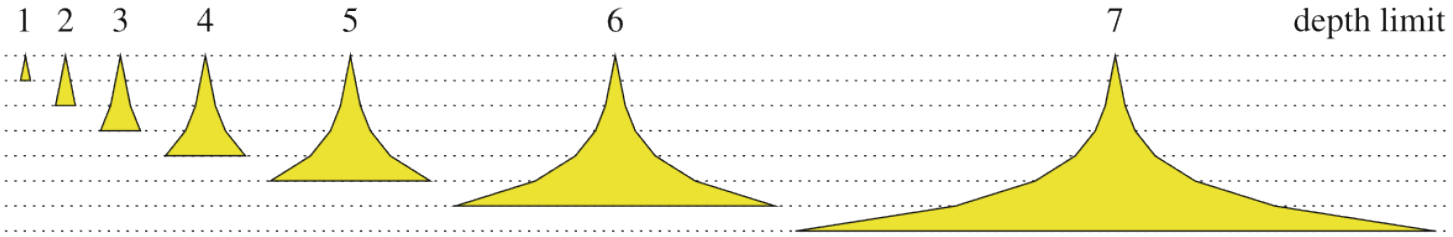
Analysis:

- incomplete (infinite loop when the tree has infinite depth).
- not optimal (the optimal solution may not be found, e.g., when there is no solution in the far left branch and the tree has infinite depth).
  - We can make the search tree finite by setting a depth limit $\Rightarrow$ *Pruned search tree*.
  - However, even when a solution is not found in the pruned search tree, maybe there is a solution outside the limit.
- Computation time = $O(b^d)$.
- Memory requirement = $O(bd)$ (at most b nodes are saved at each depth $\Rightarrow$ need at most $b \cdot d$ memory cells.)

Uninformed Search
Iterative Deepening

Iterative Deepening

We begin the depth-first search with a depth limit of 1. 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")
```

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^d	∞ or b^{d_s}	b^d
Memory use	b^d	b^d	bd	bd

- (*): only true with constant action cost.
- d_s is the maximal depth for a finite search tree.

Uninformed Search
Cycle Check

- As we mentioned before, *nodes may be repeatedly visited during a search*.
  - In the 8-puzzle, every move can be immediately undone, which leads to unnecessary cycles of length two.
- Such cycles can be prevented by *recording within each node all of its predecessors* and, *when expanding a node, comparing the newly created successor nodes with the predecessor nodes*. All of the *duplicates found can be removed from the list of successor nodes*.
- This simple check *costs only a small constant factor of additional memory space* and *increases the constant computation time* c *by an additional constant* δ for the check itself for a total $c + \delta$.
- This overhead for the cycle check is (hopefully) offset by a reduction in the cost of the search. The reduction depends, of course, on the particular application and therefore cannot be given in general terms.

Uninformed Search
Cycle Check

Question

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

- The list of all predecessors must now be stored for each node, which can be done very efficiently (e.g., using linked lists).
- During the search, each newly created node must now be compared with all its predecessors.
- Computation time =

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

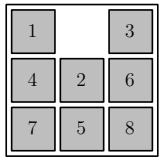
$$c_1 \cdot \sum_{i=0}^{d} b^i + c_2 \cdot \sum_{i=0}^{d} i \cdot b^i$$

cost of generating nodes — cost of the cycle check

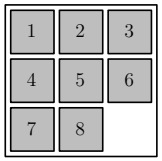
Thus, the complexity of the *search with the full cycle* check therefore only *increases by a factor of d faster* than for the *search without a cycle check*. $\Rightarrow$ Not much overhead, useful for applications with repeatedly occurring nodes.

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

With breadth-first search for the 8-puzzle, find a path

(manually) from the starting node 

to the goal node



.

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

- *Heuristics* are problem-solving strategies which in many cases find a solution faster than uninformed search.
- There is no guarantee!
- In everyday life, heuristic methods are important.
- *Realtime-decisions under limited resources*.
- A good solution found quickly is preferred over a solution that is optimal, but very expensive to derive.

Mathematical Modeling:

- *Heuristic evalutation function* $f(s)$ for states
- **Goal:** Find, with little effort, a *solution* to the stated search problem with *minimal total cost*.
- Node = state + heuristic evaluation + ...

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 heuristic rating is used as the sorting key*. Thus it is guaranteed that *the best node* (that is, the one with the lowest heuristic value) is *always at the beginning of the list*.
  - *When sorting in a new node* from the node list, *it may be advantageous to check whether the node is already available* and, if so, to delete the duplicate.

Heuristic Search

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

Give a heuristic evaluation function for states 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 be *a function that calculates the actual costs from each node to the goal*.

Question

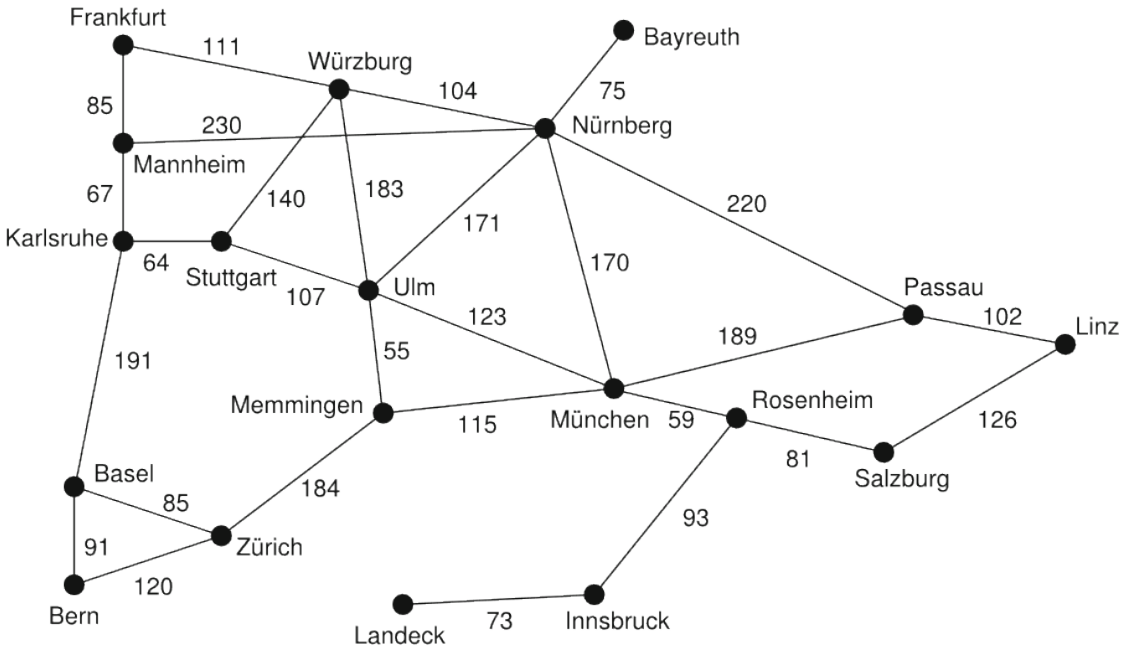
How do we find a heuristic that is fast and simple to compute?

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.

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* $h(s)$ = flying distance from city s to Ulm (given in the figure below).

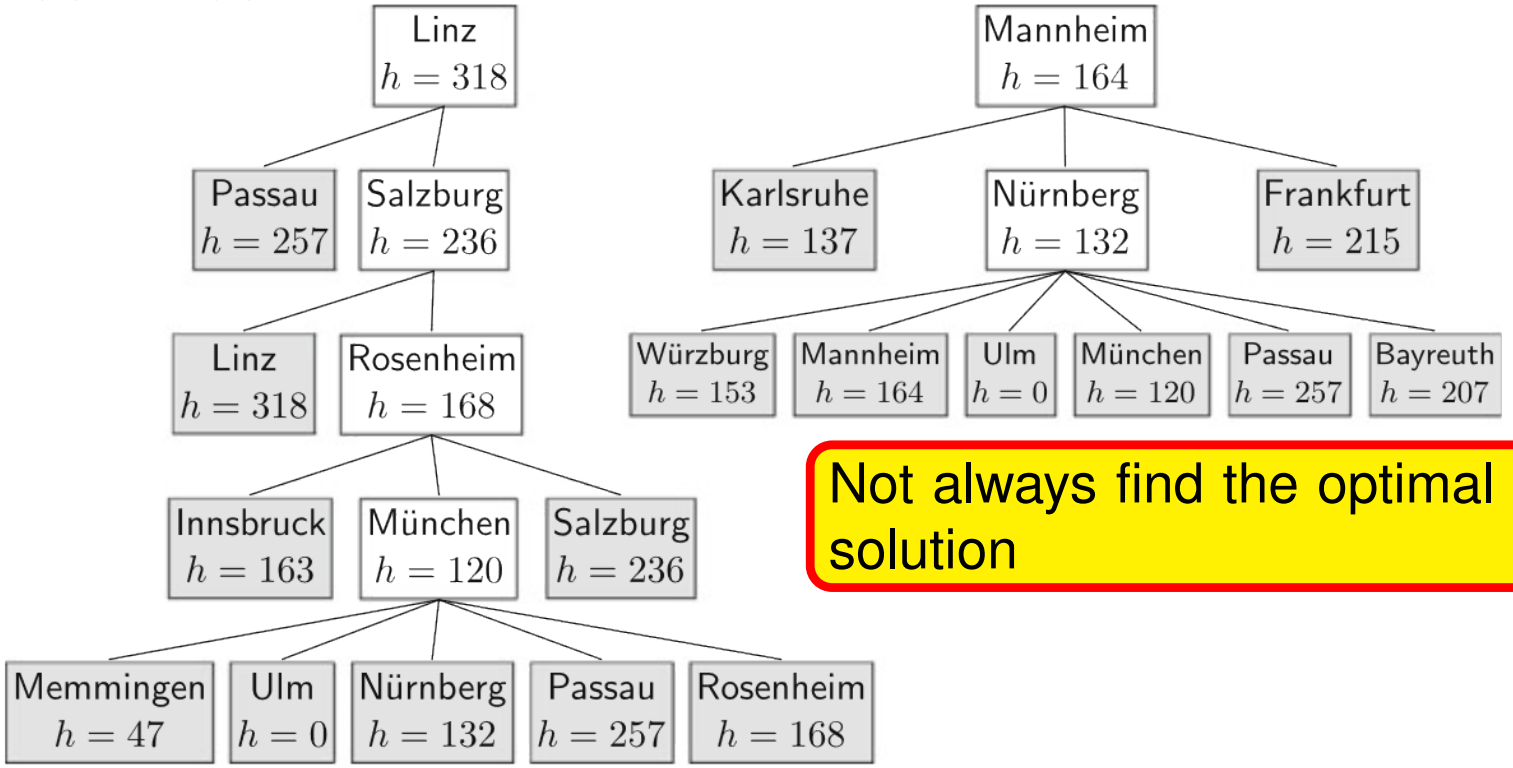


Basel	204
Bayreuth	207
Bern	247
Frankfurt	215
Innsbruck	163
Karlsruhe	137
Landeck	143
Linz	318
München	120
Mannheim	164
Memmingen	47
Nürnberg	132
Passau	257
Rosenheim	168
Stuttgart	75
Salzburg	236
Würzburg	153
Zürich	157

Figure: City graph with flying distances from all cities to Ulm

Example 5 (cont.)

We use the cost estimate function $h(s)$ directly as the evaluation function in the HEURISTICSEARCH, i.e., we set $f(s) = h(s)$.



Not always find the optimal solution

Figure: Greedy search: from Linz to Ulm (left) and from Mannheim to Ulm (right).

Example 5 (cont.)

We use the cost estimate function $h(s)$ directly as the evaluation function in the HEURISTICSEARCH, i.e., we set $f(s) = h(s)$.

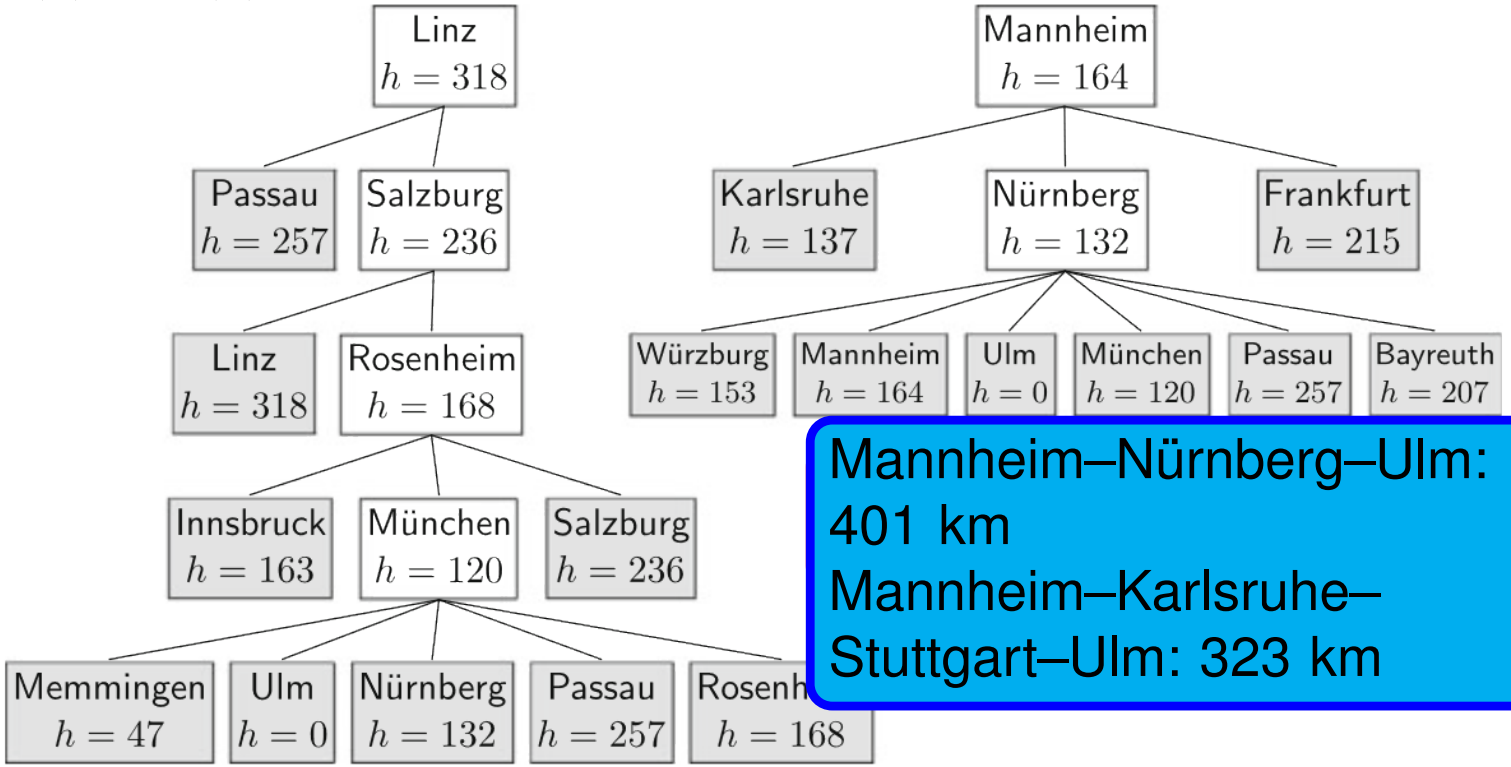


Figure: Greedy search: from Linz to Ulm (left) and from Mannheim to Ulm (right).

Heuristic Search
A*-Search

- *Cost function* $g(s)$ = Sum of accrued costs from root to current node s.
- *Heuristic cost estimate* $h(s)$ = Estimated cost from current node s to goal.
- *Heuristic evaluation function* $f(s) = g(s) + h(s)$.

Admissible heuristic cost estimate function

A heuristic cost estimate function $h(s)$ that *never overestimates the actual cost* from state s to the goal is called *admissible*.

A*-algorithm

= HEURISTICSEARCH + $f(s) = g(s) + h(s)$ + 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?

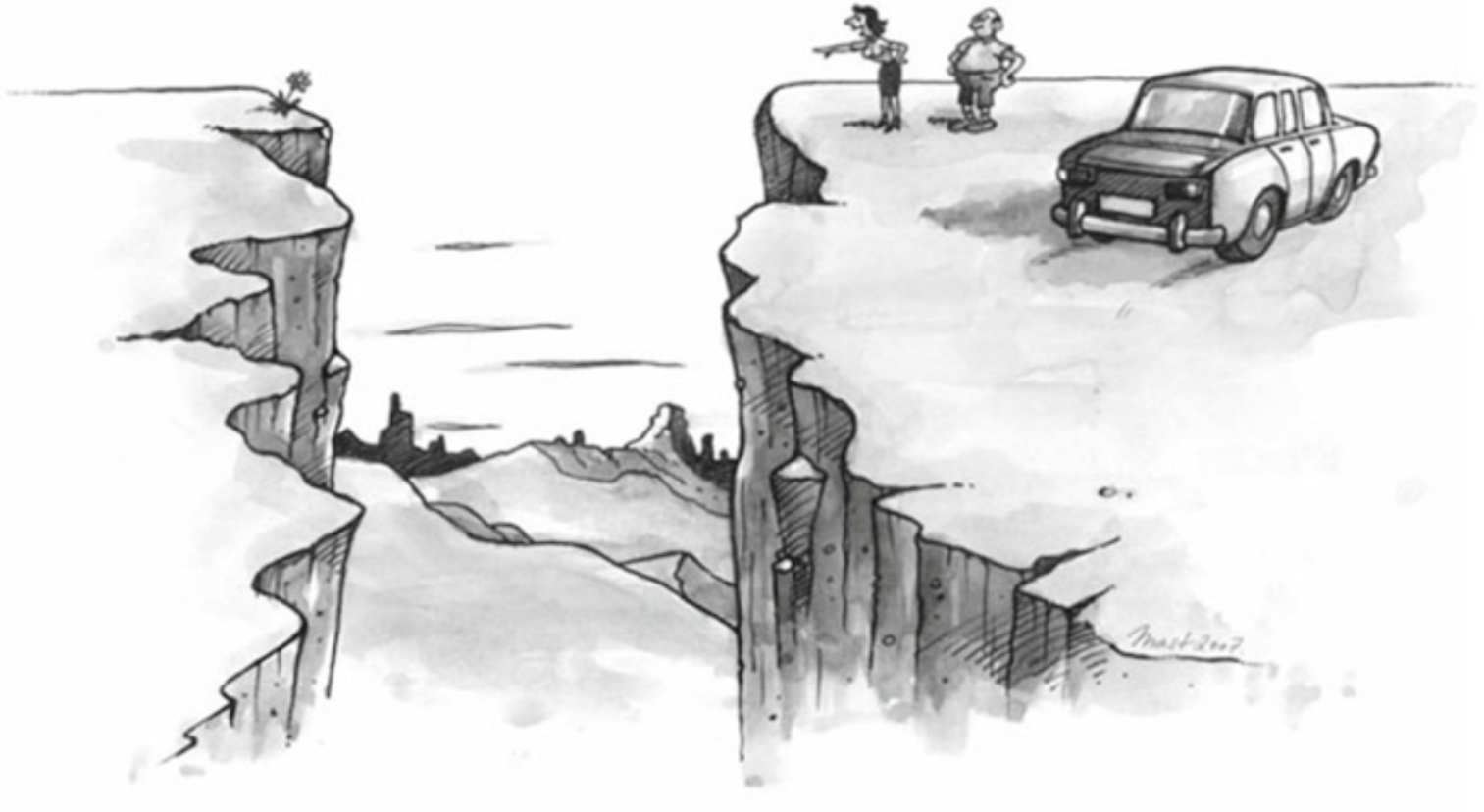


Figure: He: “Dear, think of the fuel cost! I’ll pluck one for you somewhere else.” She: “No, I want that one over there!”

Example 5 (cont.)

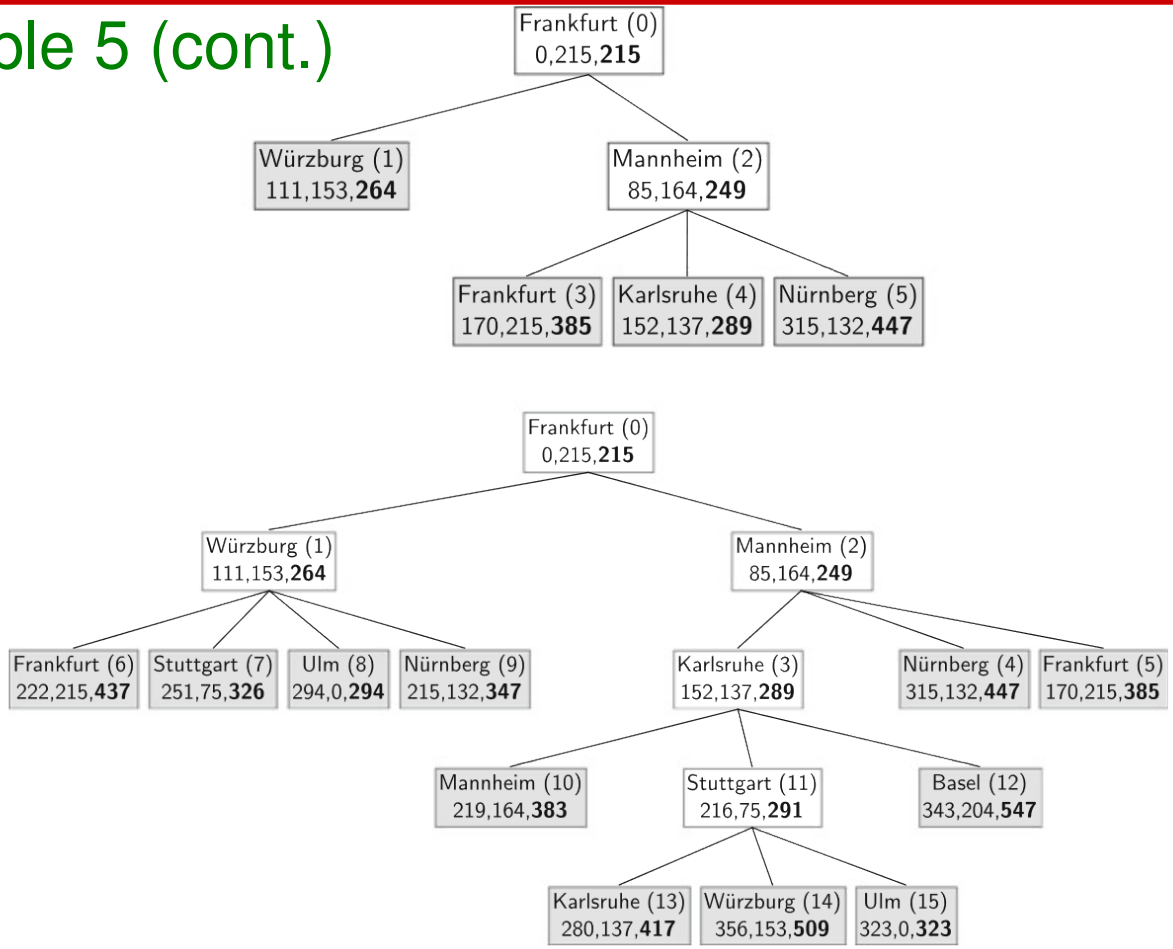


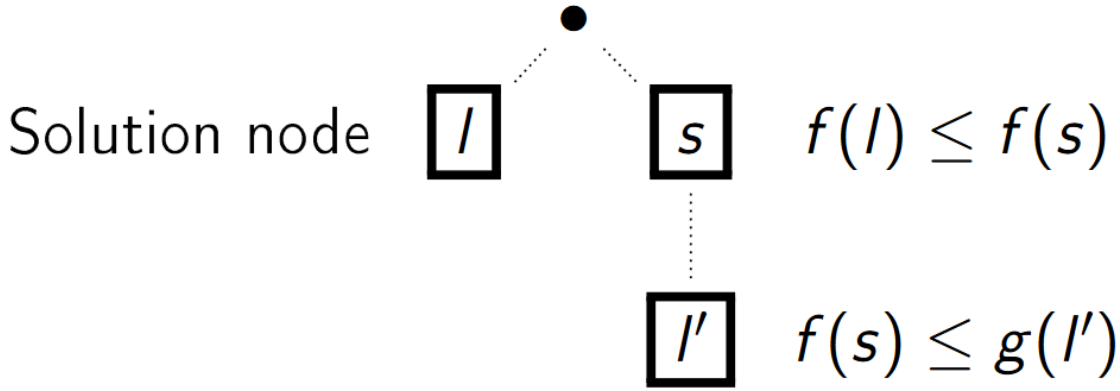
Figure: Two snapshots of the A* search tree for the optimal route from Frankfurt to Ulm. In the boxes below the name of the city s we show $g(s), h(s), \mathbf{f(s)}$. Numbers in parentheses after the city names show the order in which the nodes have been generated by the Successors function

Theorem 2

The A algorithm is optimal. That is, it always finds the solution with lowest total cost if the heuristic h is admissible.*

Proof.

The *first solution node l* found by A* *never has a higher cost than another arbitrary solution node l'*, i.e., $g(l) \leq g(l')$.



Definition of f $\left[f(s) = \text{estimated cost from start to goal via } s \right]$

$$g(l) = g(l) + h(l) = f(l) \leq f(s) = g(s) + h(s) \leq g(l').$$

- l is a solution node $\Leftrightarrow h(l) = 0$
- list of open nodes is sorted in ascending order of f
- the heuristic is admissible $\left(\text{estimated cost to the goal} \leq \text{actual cost to the goal}\right)$

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

$$h(s) = c^*(s, l) - c^*(z, l) \leq c^*(s, z).$$

Heuristic Search

Route Planning with the A* Search Algorithm

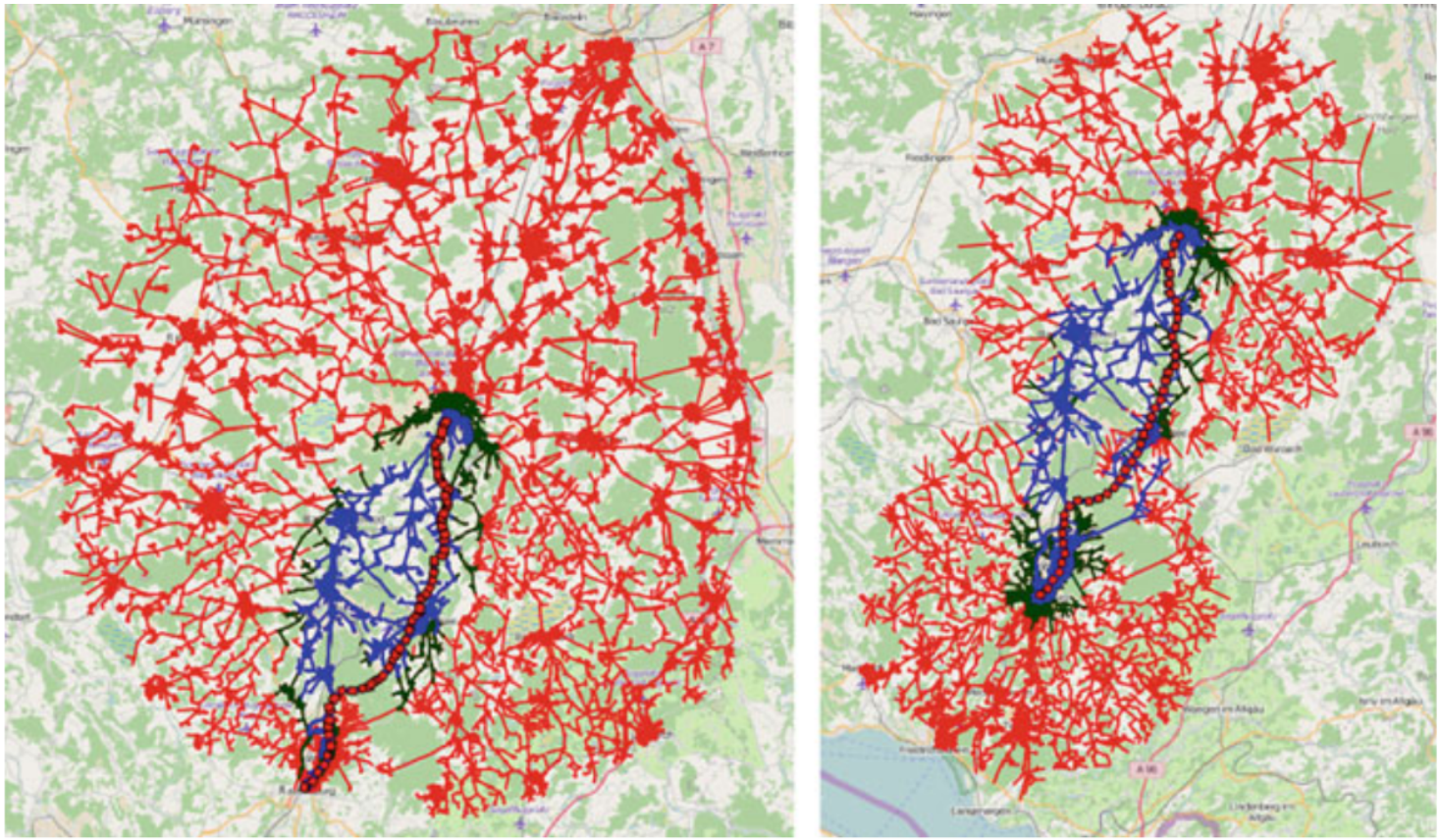


Figure: The search tree of A* search for planning a route from Ravensburg to Biberach with: no heuristic (red), air line distance (dark green), landmark heuristic (blue). **Left:** normal search, **Right:** bidirectional search (plan in parallel a route from Ravensburg to Biberach and from Biberach to 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.

Heuristic Search
Route Planning with the A* Search Algorithm

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]).

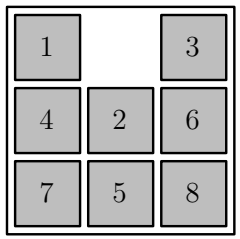
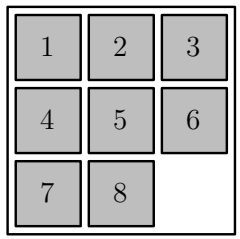
Heuristic Search
IDA*-Search

- Weaknesses of A*:
  - High memory requirements
  - List of open nodes must be sorted $\Rightarrow$ Heapsort (logarithmic time complexity for insertion and removal of nodes)
- Solution: IDA*-algorithm [Korf 1985]
  - Iterative Deepening
  - The same as depth-first search, but limit for heuristic evaluation $f(s)$ (instead of depth limit).
    - Perform a DFS, cut off a branch when $f(s)$ exceeds a given threshold
    - This threshold starts at the estimate of the cost at the initial state, and increases for each iteration of the algorithm.
    - At each iteration, the threshold used for the next iteration is the minimum cost of all values that exceeded the current threshold.

Simple admissible heuristics for 8-puzzle

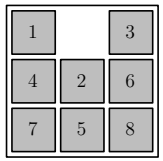
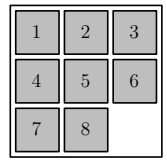
- h_1: counts the number of squares that are not in the right place.
- h_2: *Manhattan distance*
  - For every square the horizontal and vertical distances to that square's location in the goal state are added together.
  - This value is then summed over all squares.

Example 6

Distance between  and  :

- $h_1(s) = 3$.
- $h_2(s) = 0 + 0 + 0 + 1 + 0 + 0 + 1 + 1 = 3$. (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 node  to the goal node  .

(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 direct neighbor nodes x and y, where the 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 the A* algorithm deletes duplicates of identical nodes 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

- Of the various search algorithms for uninformed search, *iterative deepening is the only practical one* because it is complete and can get by with very little memory.
- IDA* is complete, fast and memory efficient.
- *Good* heuristics greatly reduce the effective branching factor.
- *Heuristics have no performance advantage for unsolvable problems* because *the unsolvability of a problem can only be established when the complete search tree has been searched through*.
  - For solvable problems, heuristics often reduce computation time dramatically, but for unsolvable problems the cost can even be higher with heuristics.
  - For example, in the proof of PL1 formulas (which is undecidable), the search tree can be infinitely deep. This means that, in the unsolvable case, the search potentially never ends.

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

- 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
Minimax Search

Characteristics of games:

- The *effective branching factor in chess* is *around* 30 *to* 35.
- 50 moves per player: $30^{100} \approx 10^{148}$ leaf nodes $\Rightarrow$ *no chance to fully explore the search tree*.
- *Real-time requirement* (chess is often played with a time limit) $\Rightarrow$ Limited search depth.
- Among the leaf nodes of this depth-limited tree there are normally no solution nodes (that is, nodes which terminate the game) $\Rightarrow$ *heuristic evaluation function* B *for board positions*.
  - The level of play strongly depends on the quality of the function B.
- Player: Max, Opponent: Min.
- **Assumption:** Opponent Min always makes the best move he can.
- The higher the evaluation $B(s)$ for position s, the better position s is for the player Max and the worse it is for his opponent Min.
  - Max maximizes the evaluation of his moves.
  - Min minimizes evaluation of his moves.

Games with Opponents
Minimax Search

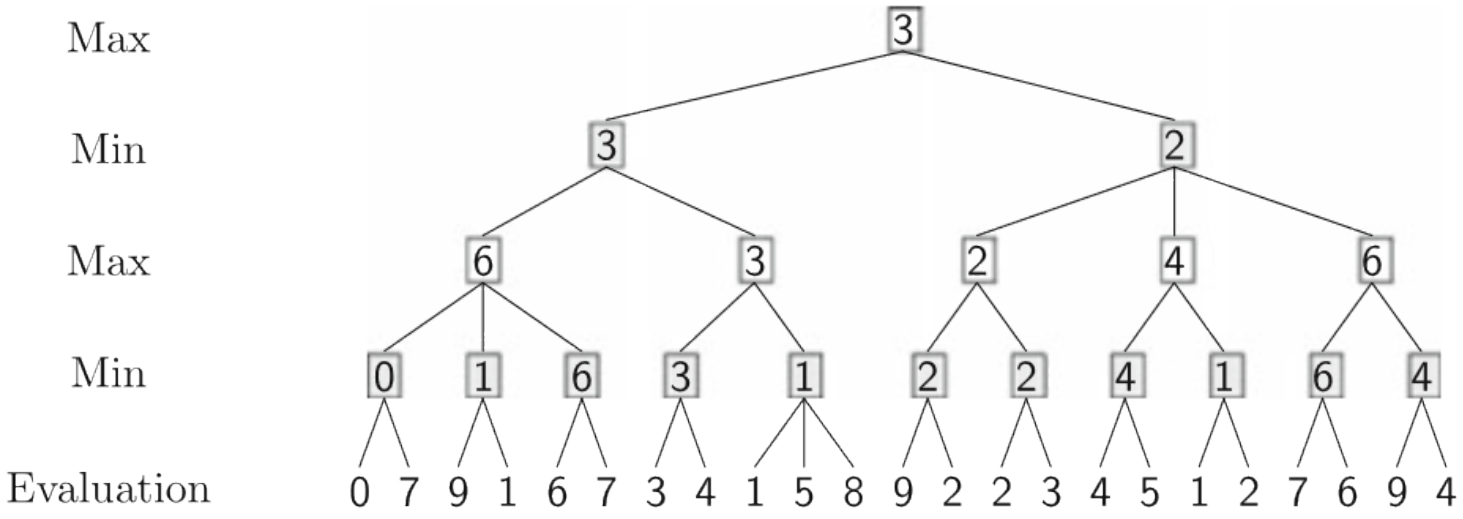


Figure: A minimax game tree with look-ahead of four half-moves. The evaluations of all leaves are given. The evaluation of an inner node is derived recursively as the maximum or minimum of its child nodes, depending on the node's level.

Games with Opponents
Alpha-Beta-Pruning

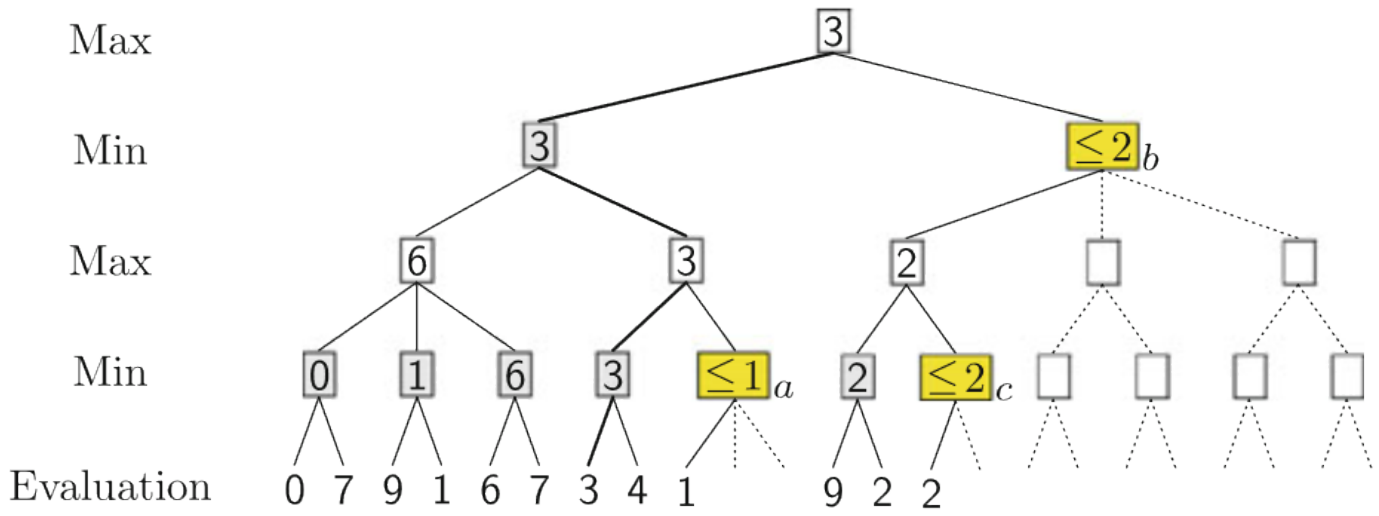


Figure: An alpha-beta game tree with look-ahead of four half-moves. The dotted portions of the tree are not traversed because they have no effect on the end result.

- 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

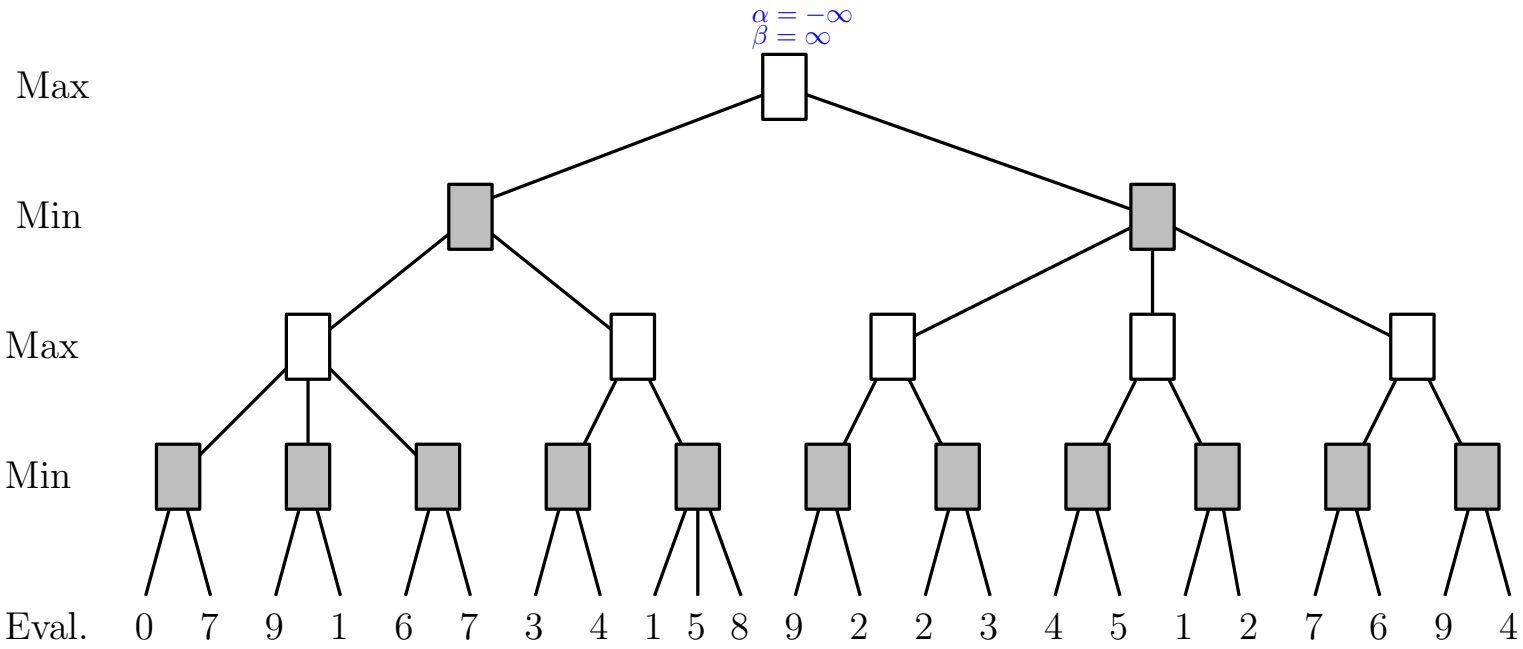


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

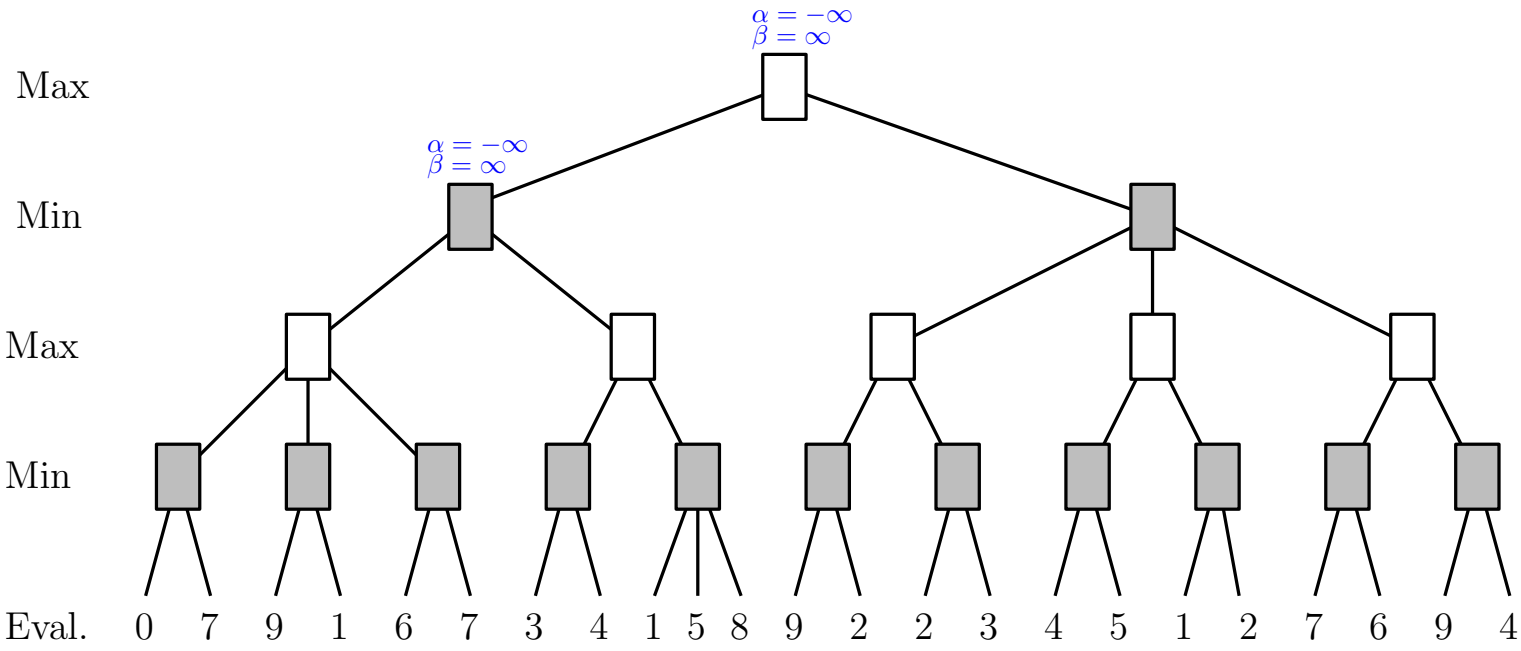


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

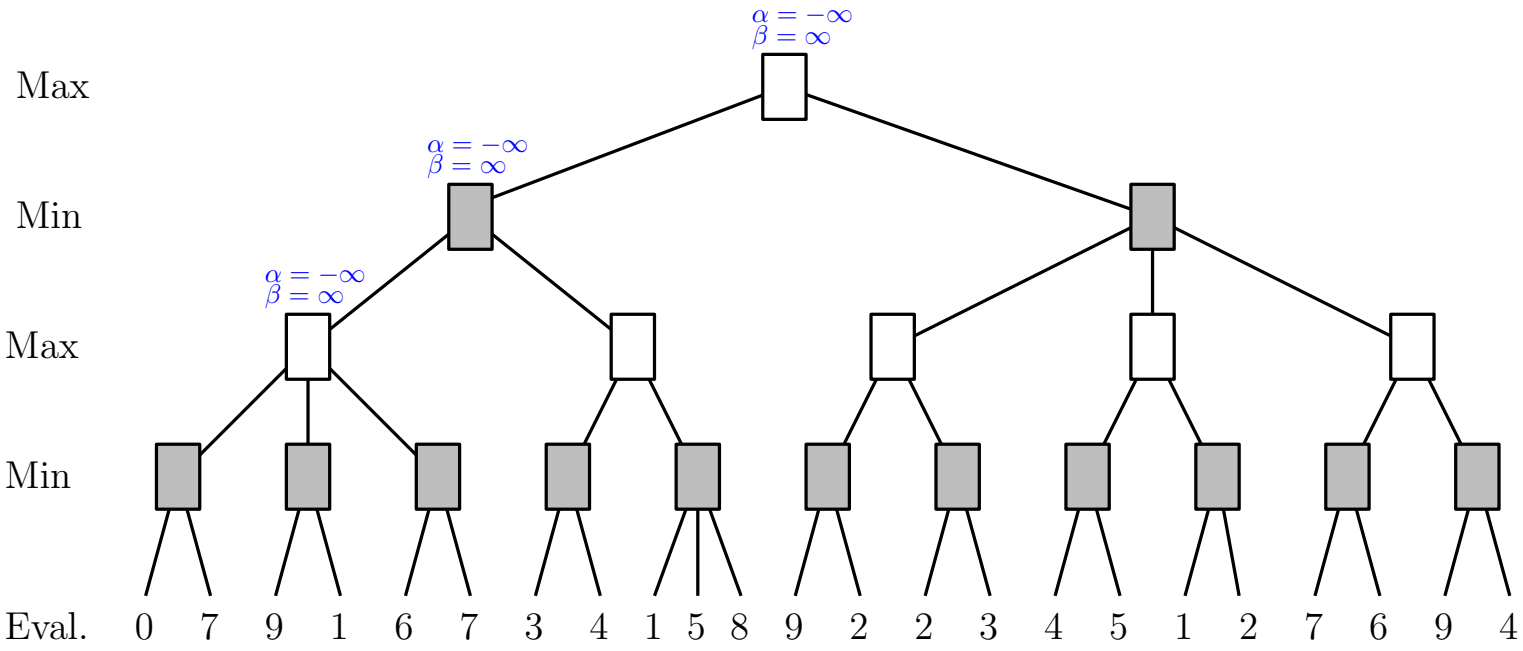


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

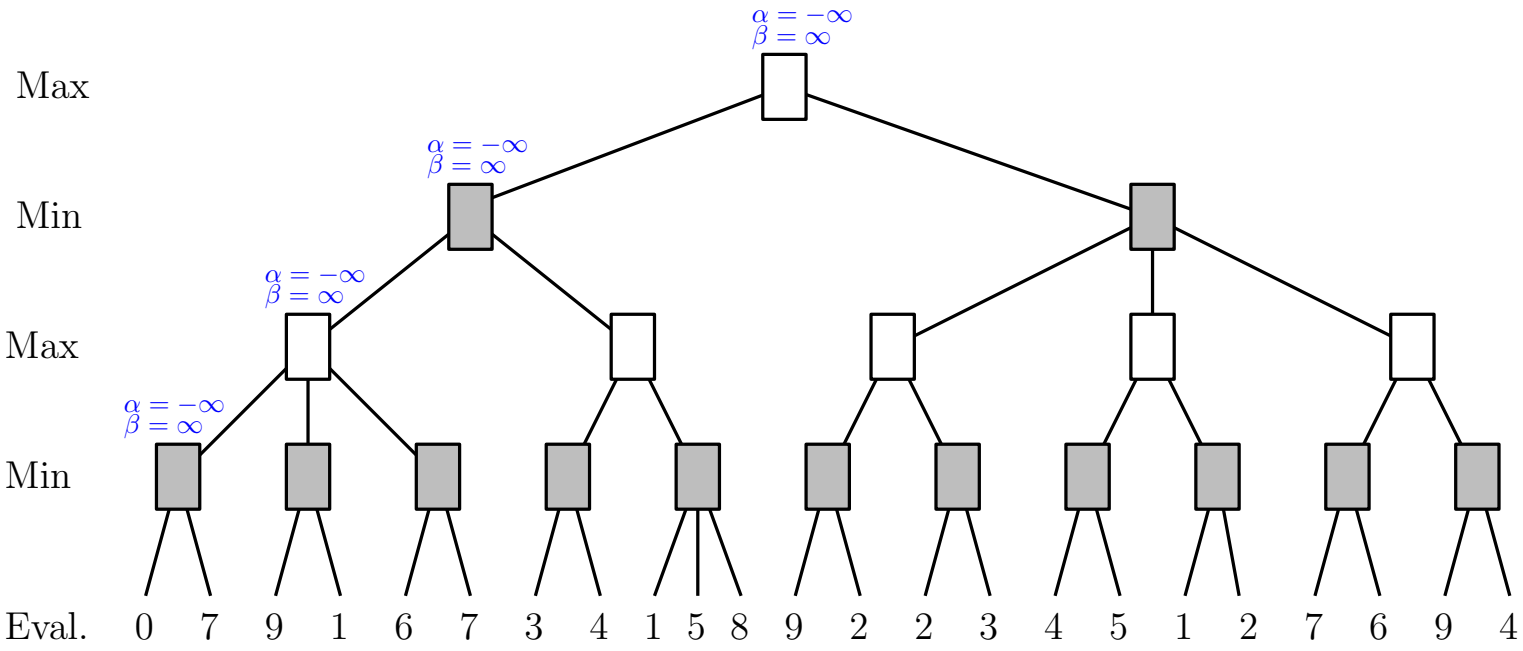


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

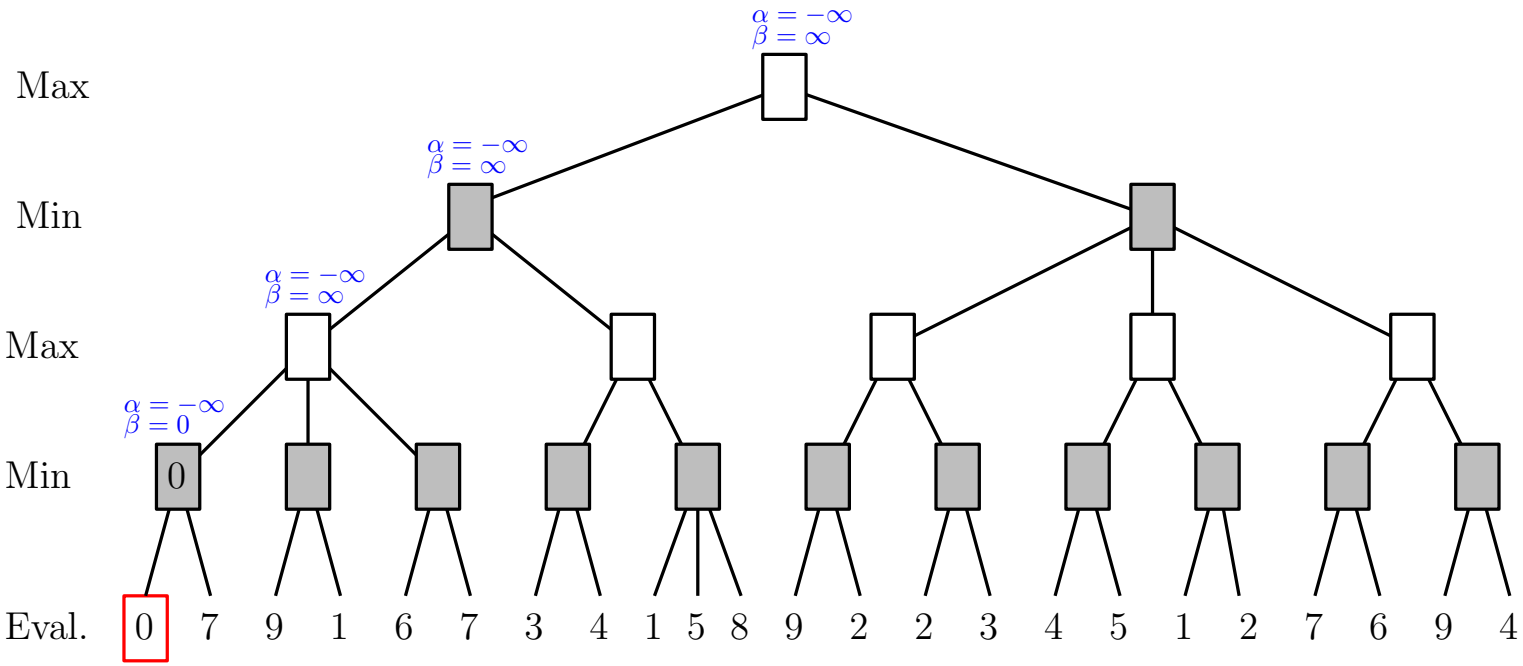


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

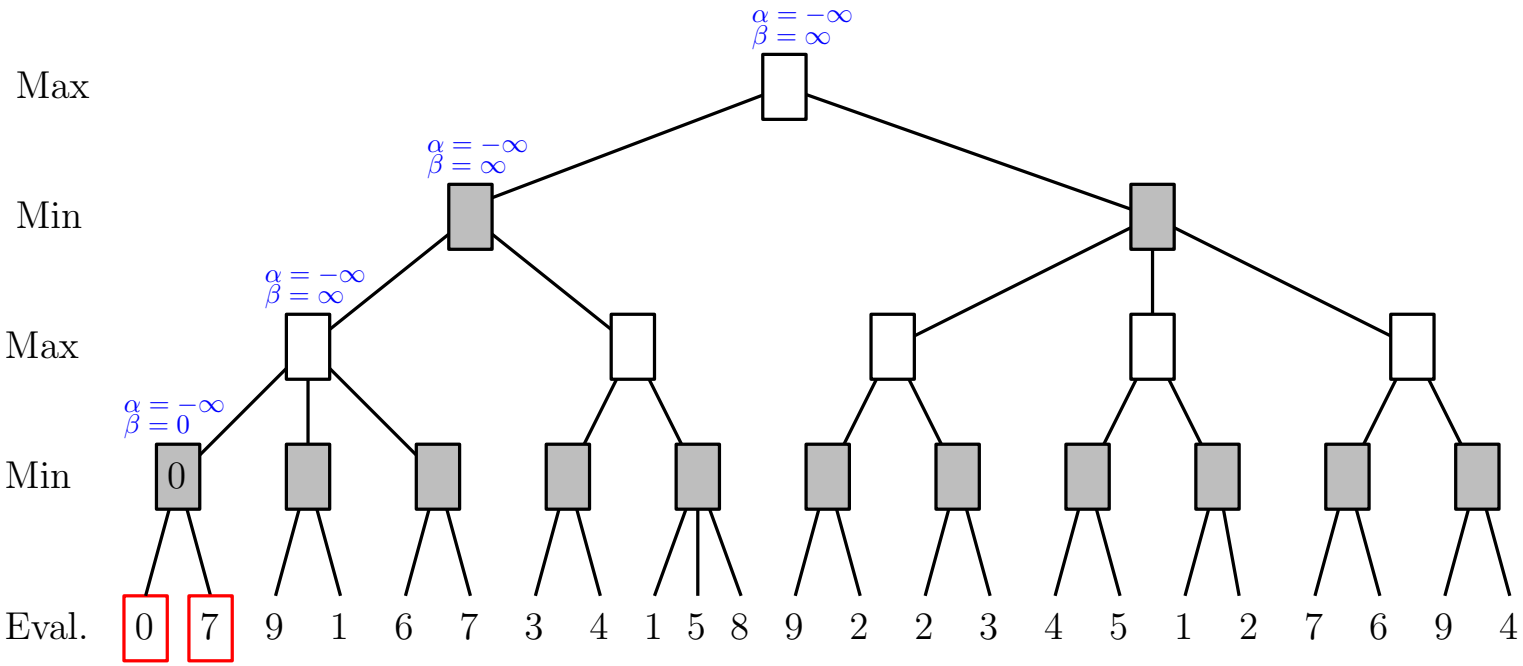


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

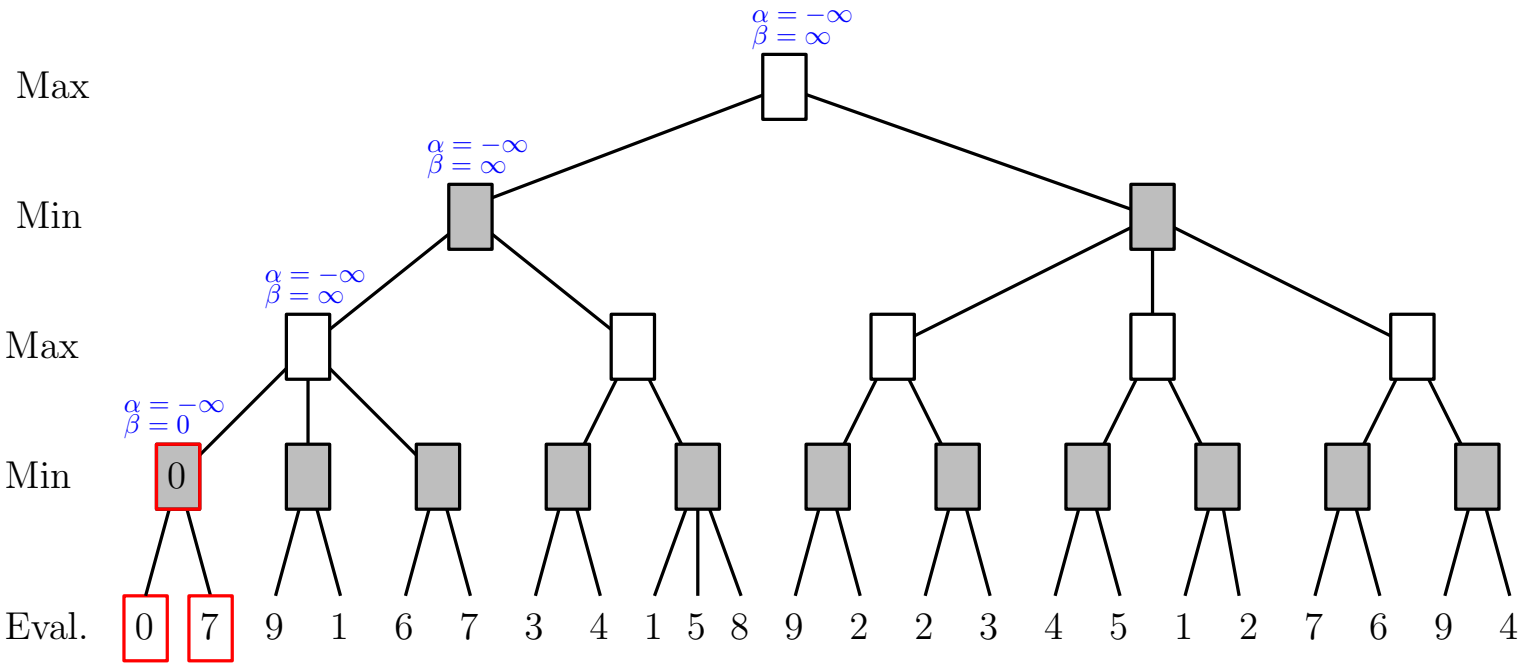


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

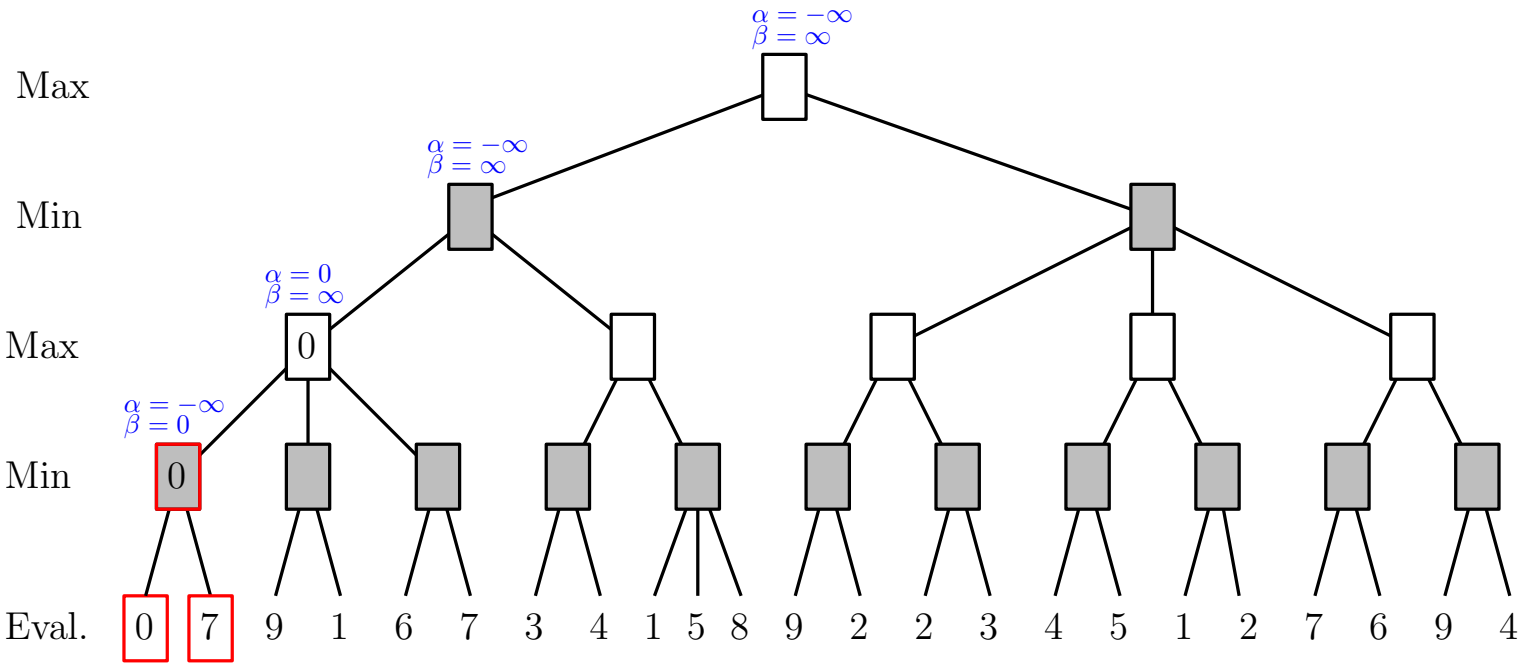


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

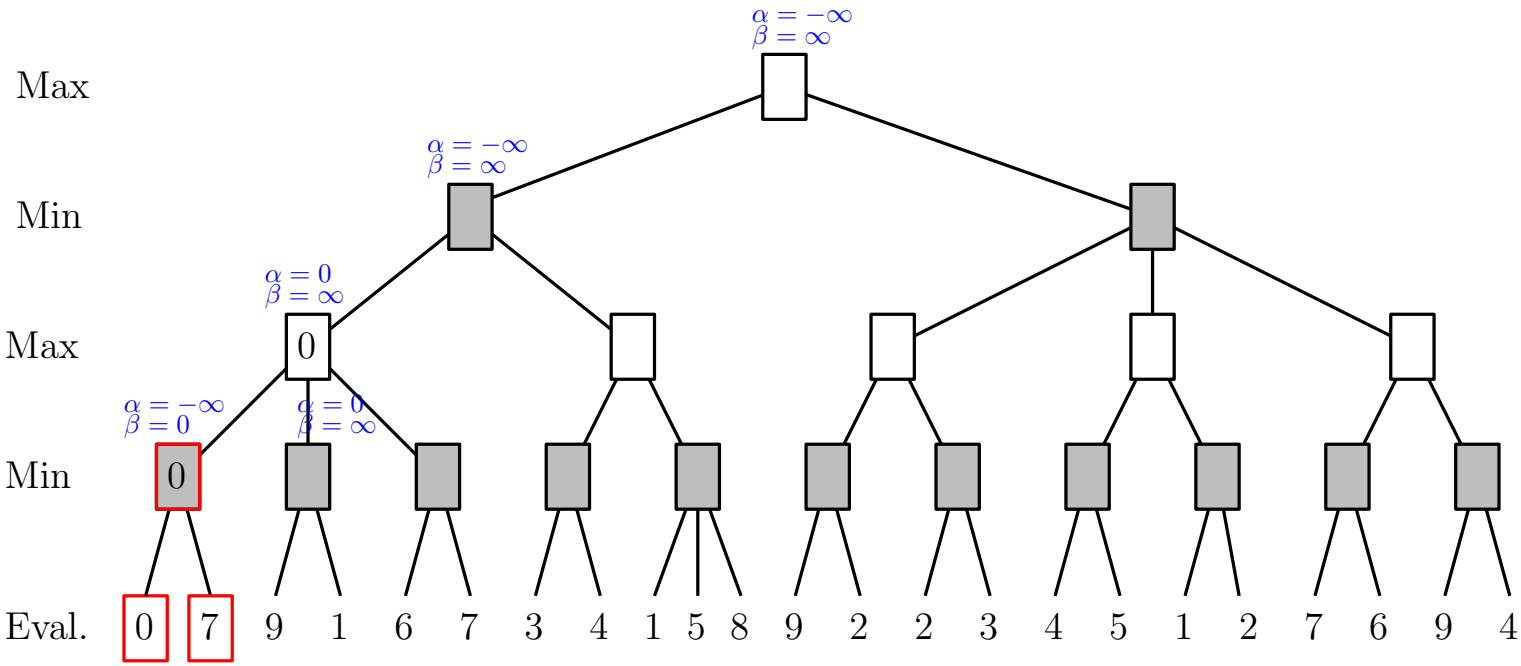


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

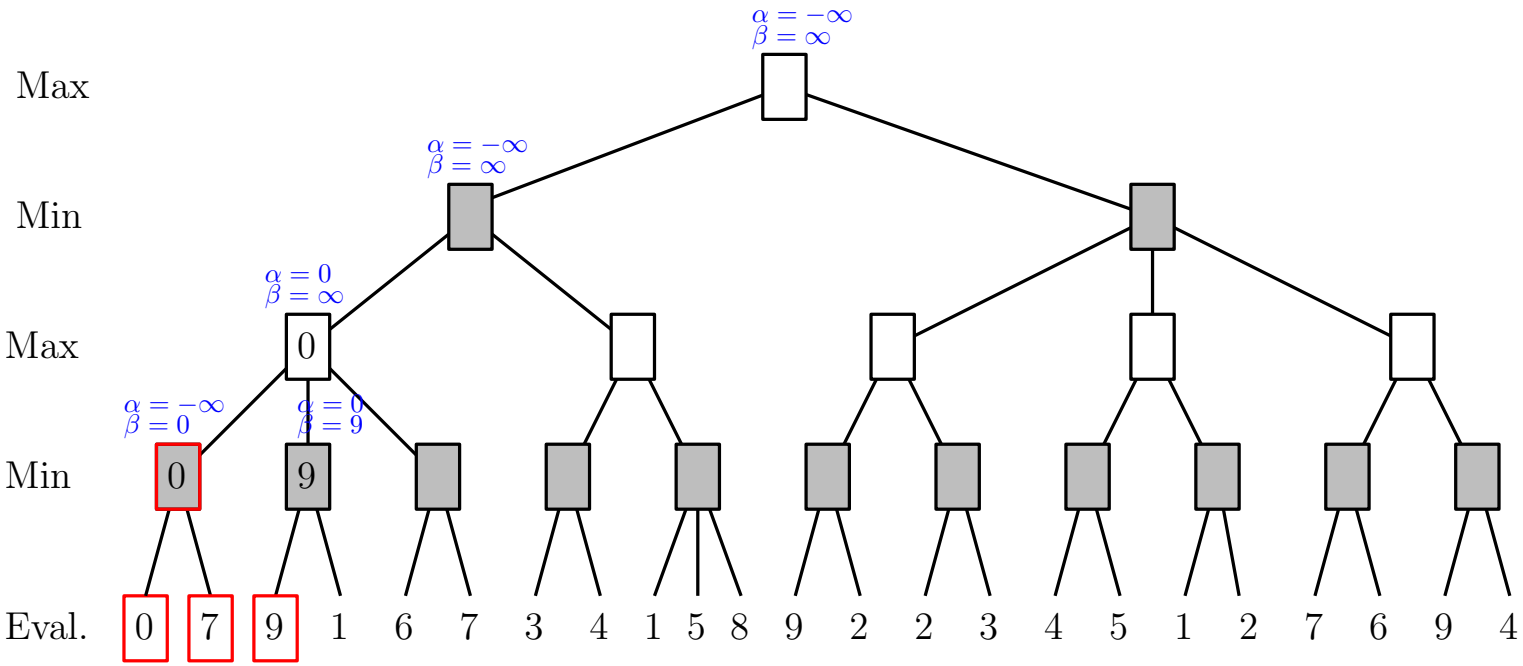


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

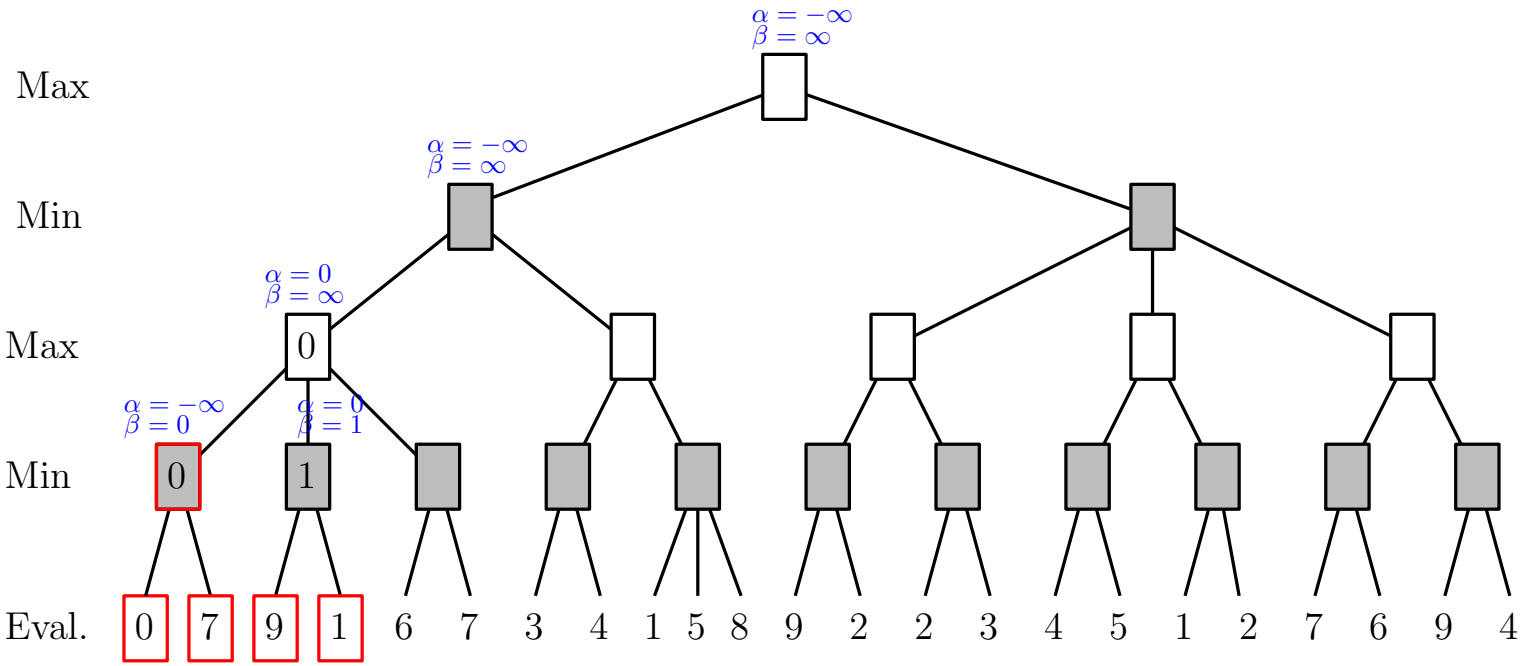


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

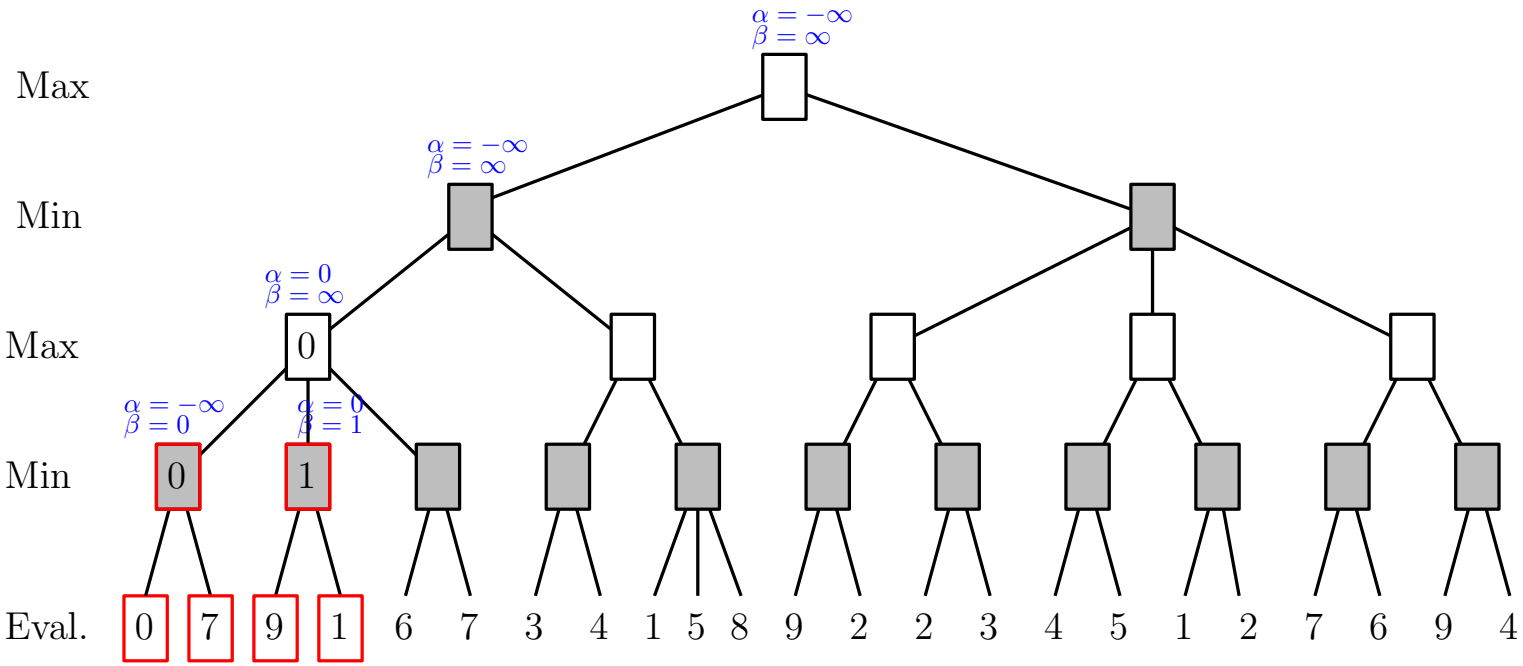


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

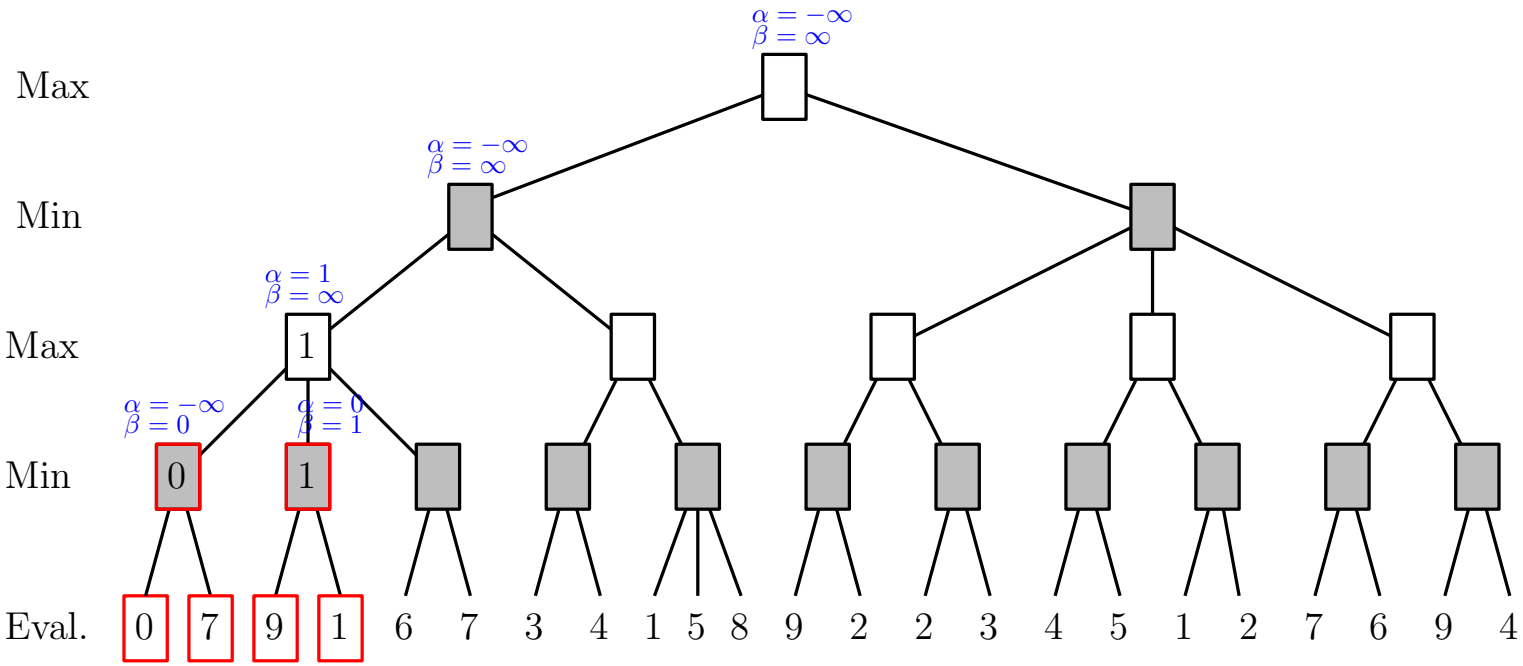


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

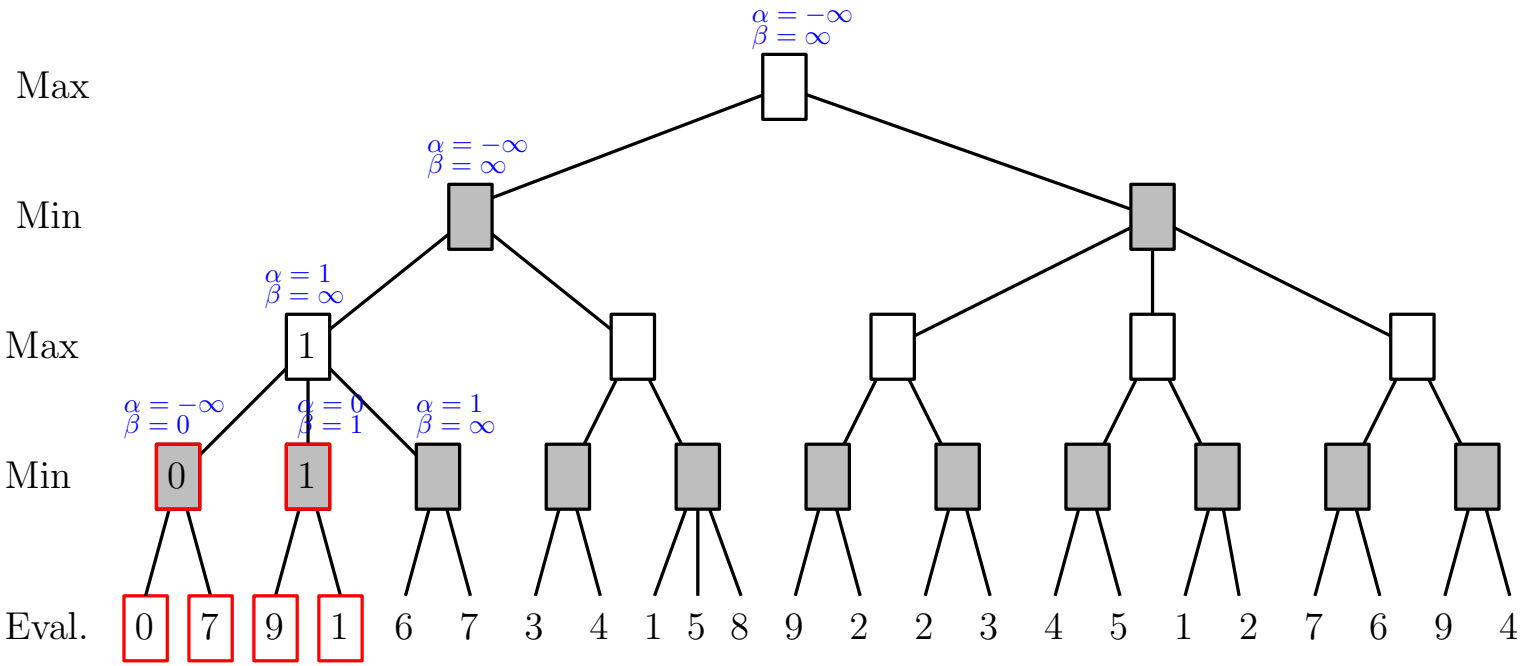


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

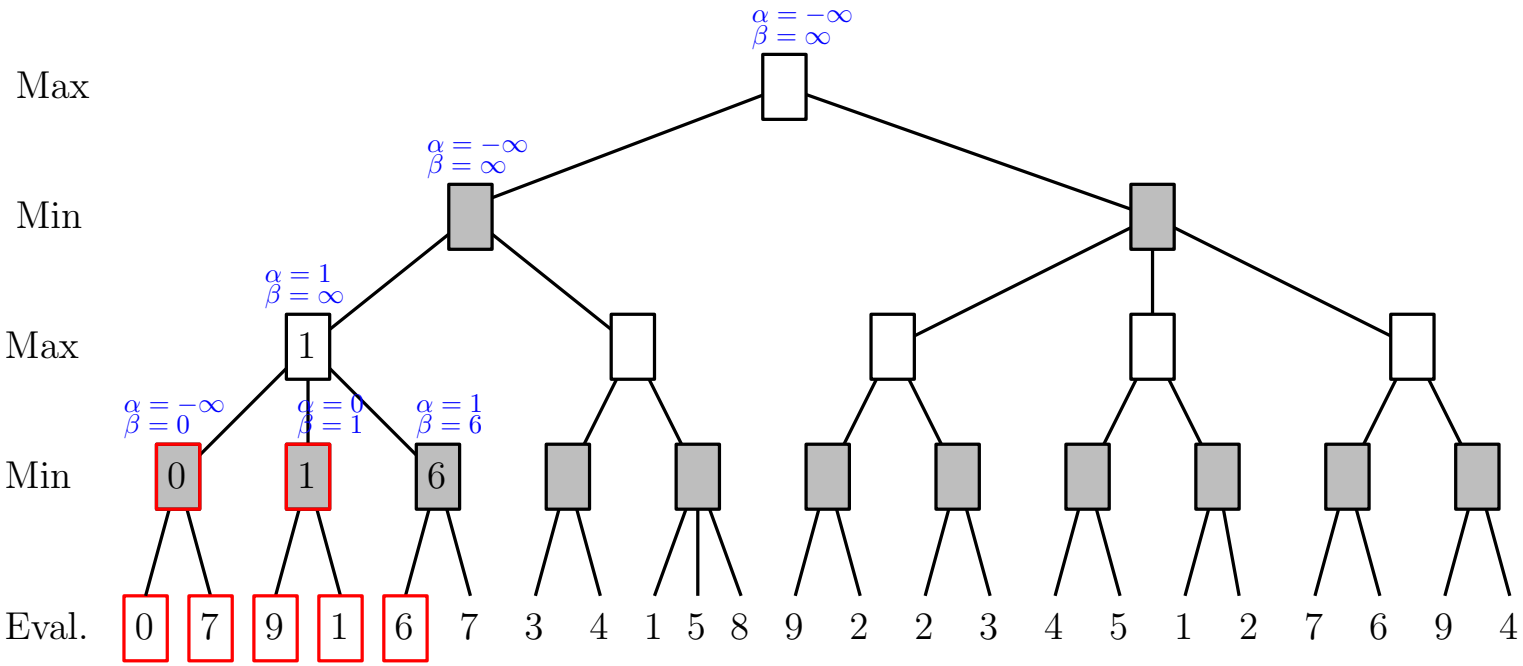


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

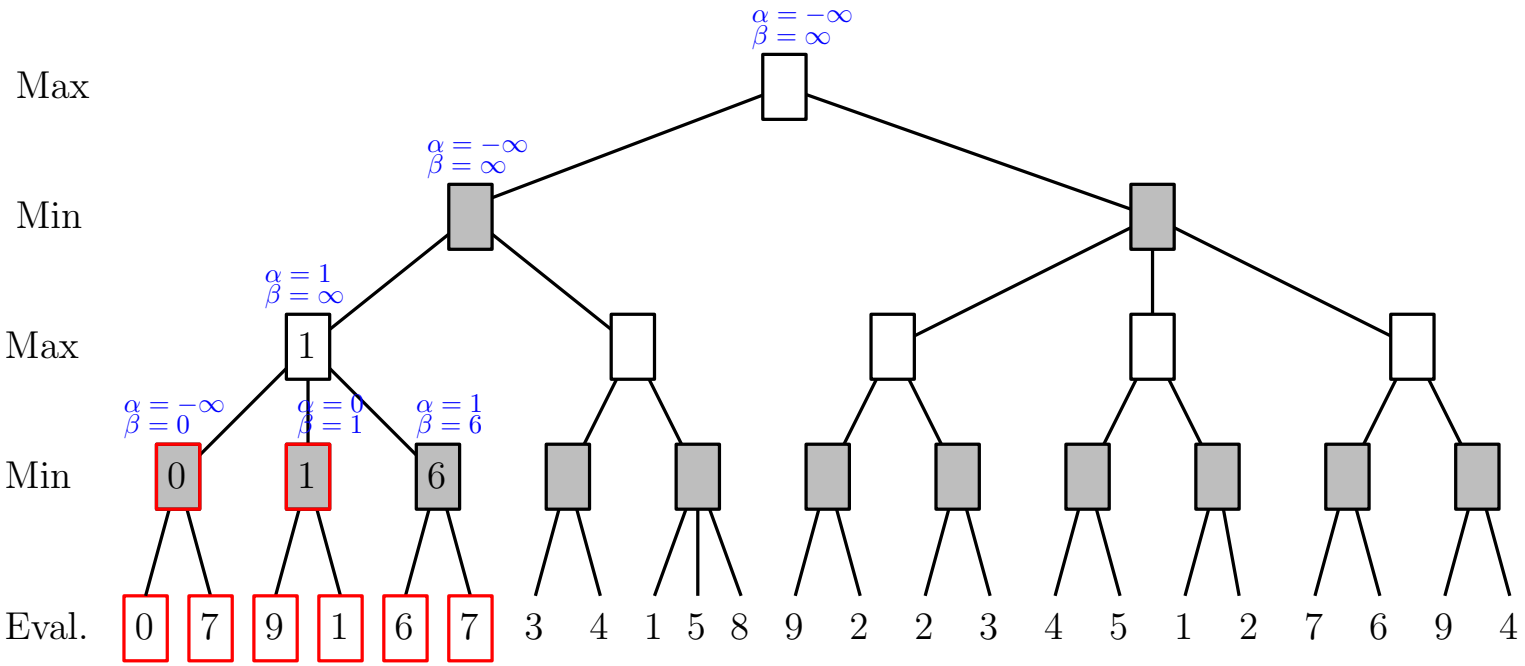


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

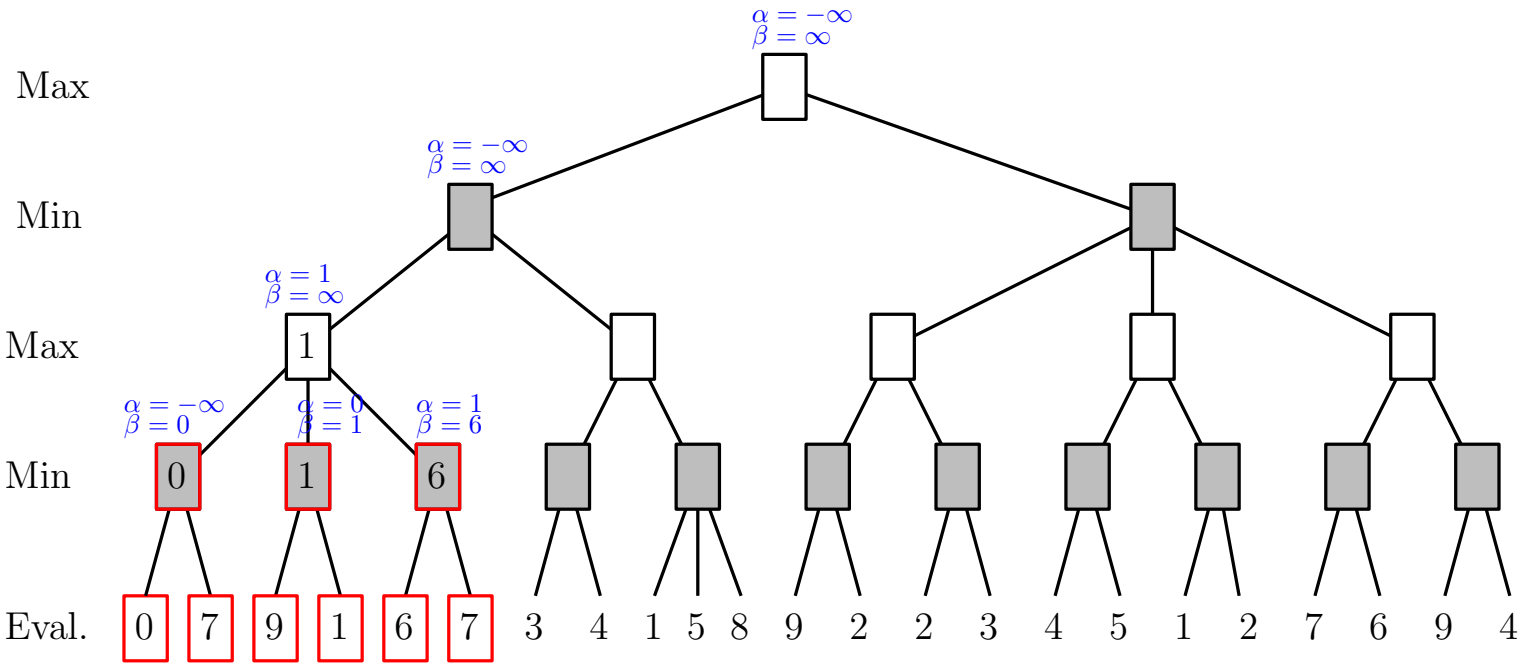


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

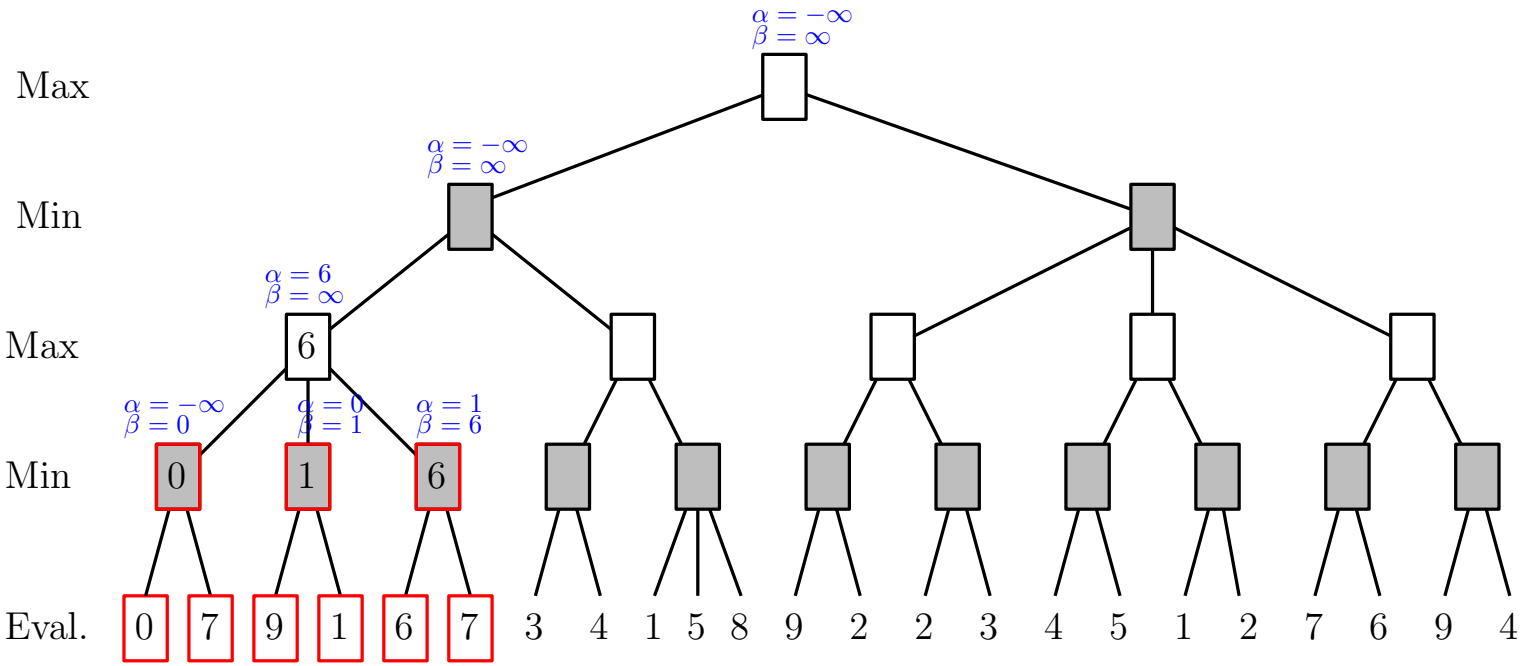


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

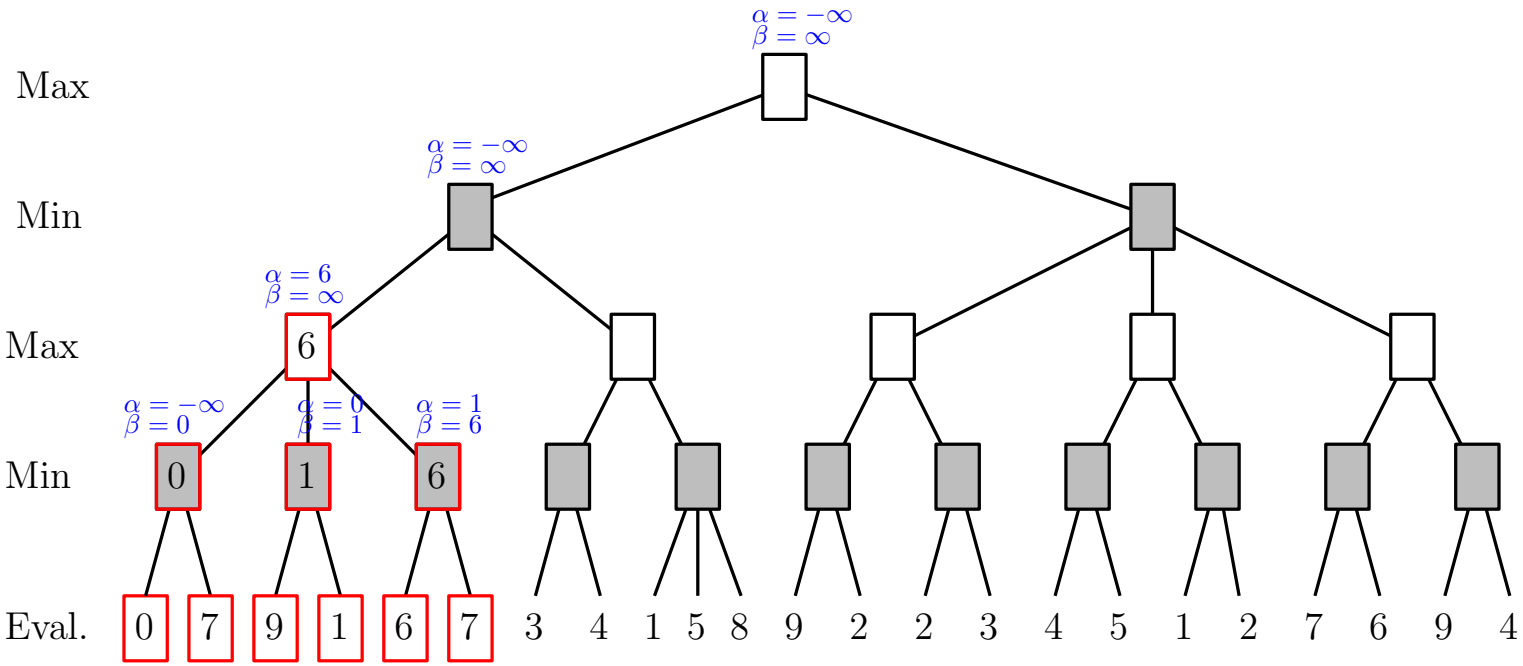


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

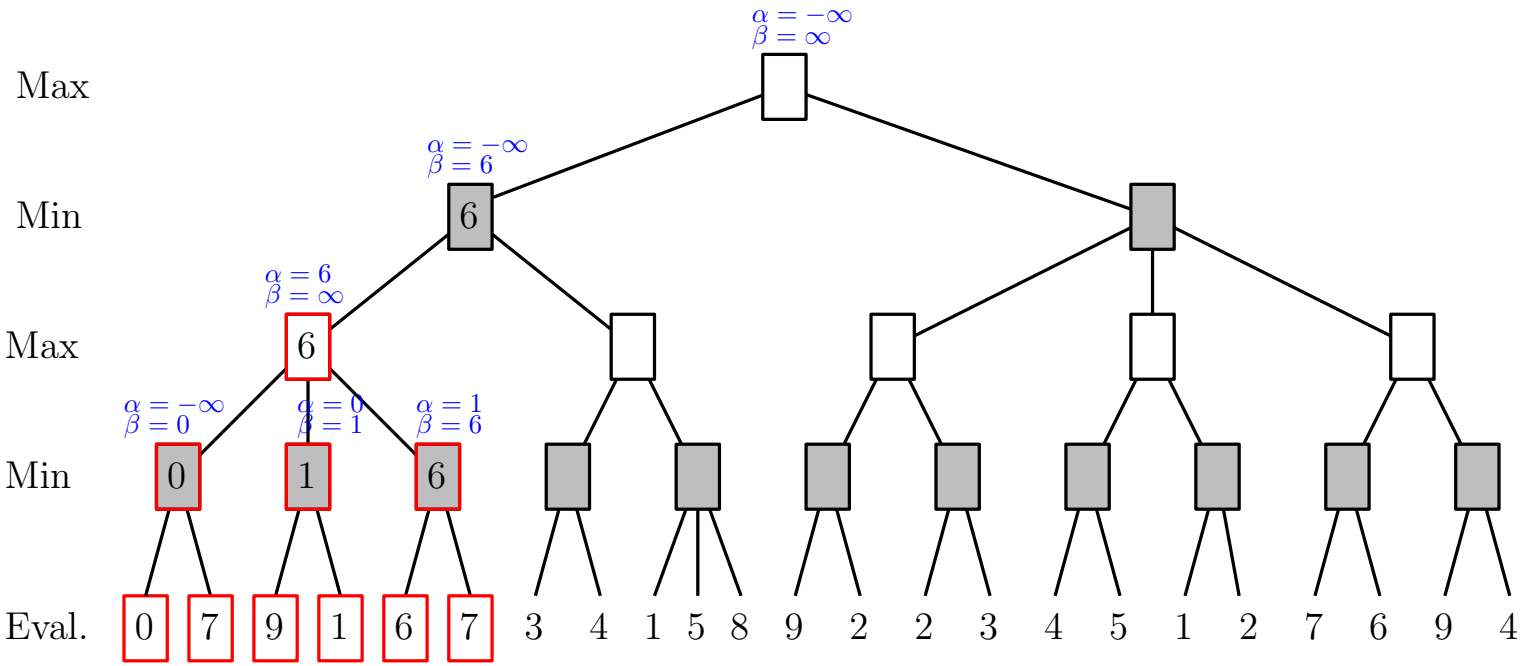


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

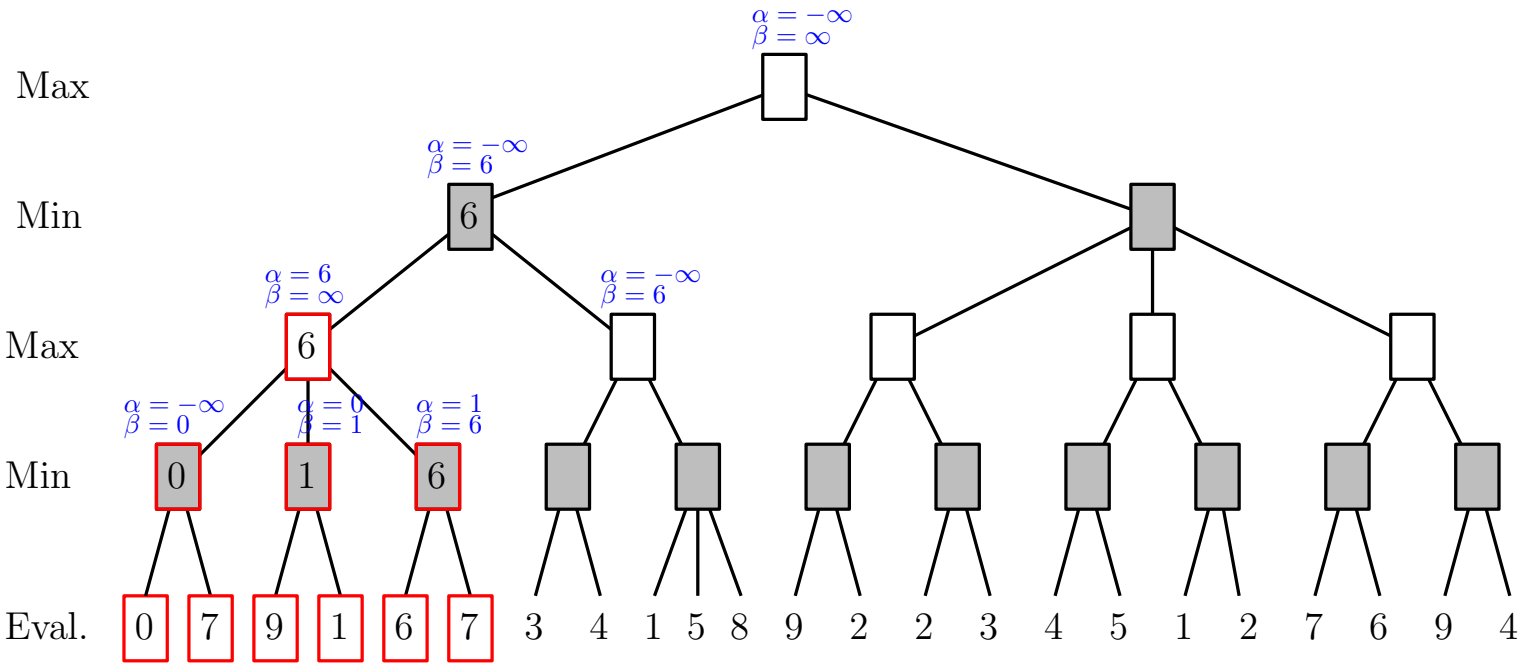


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

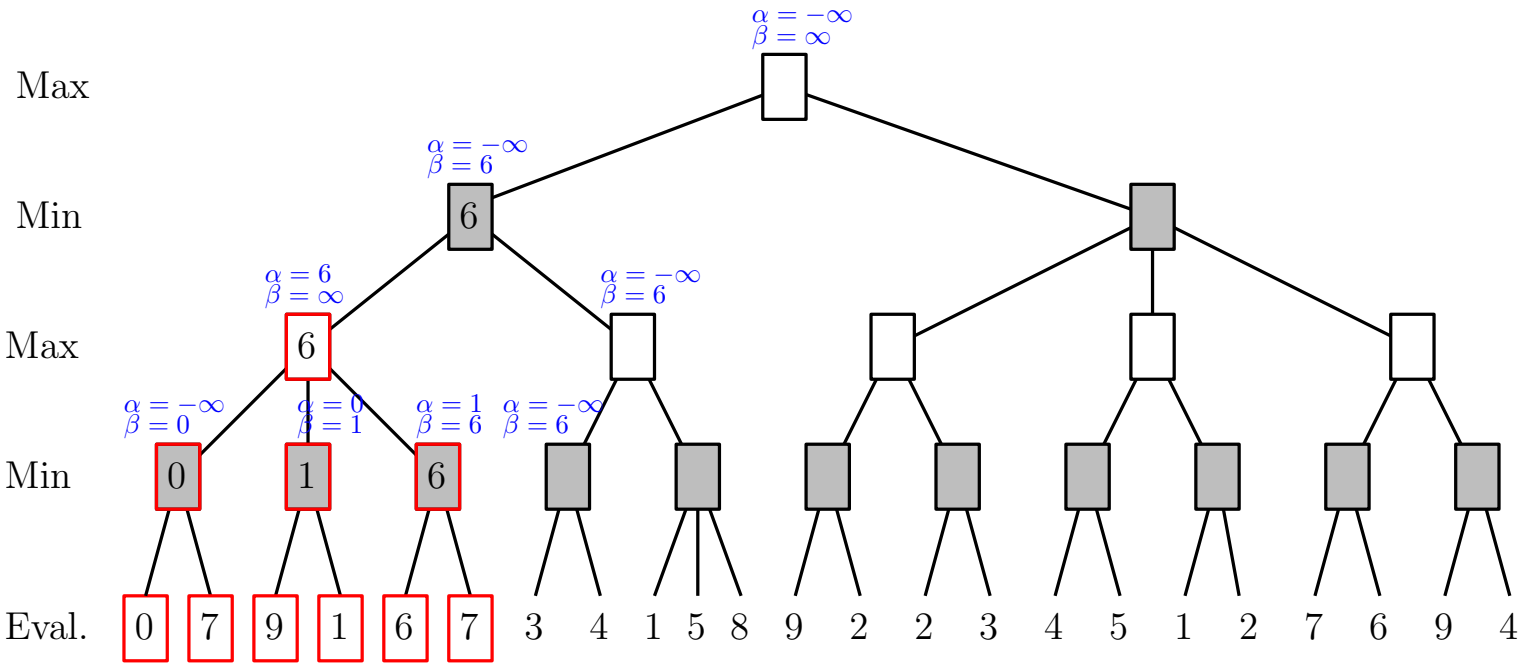


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

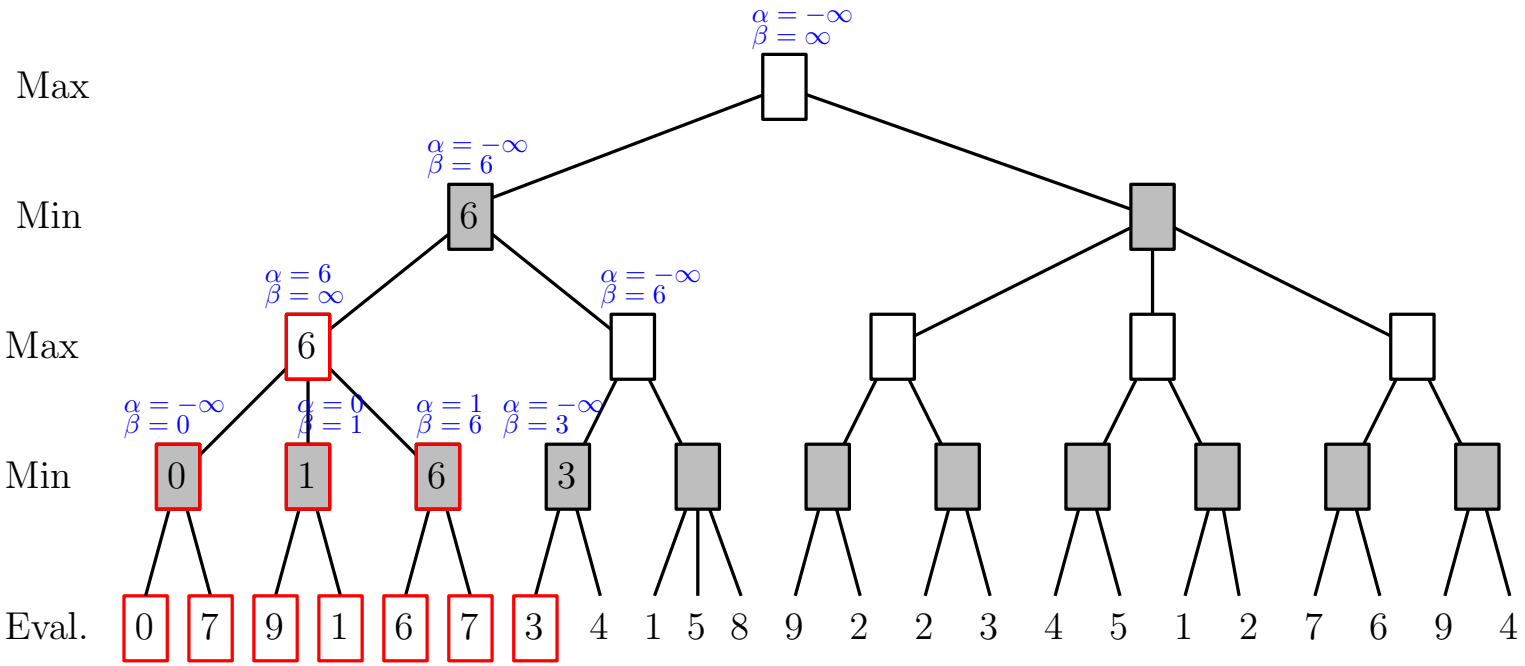


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

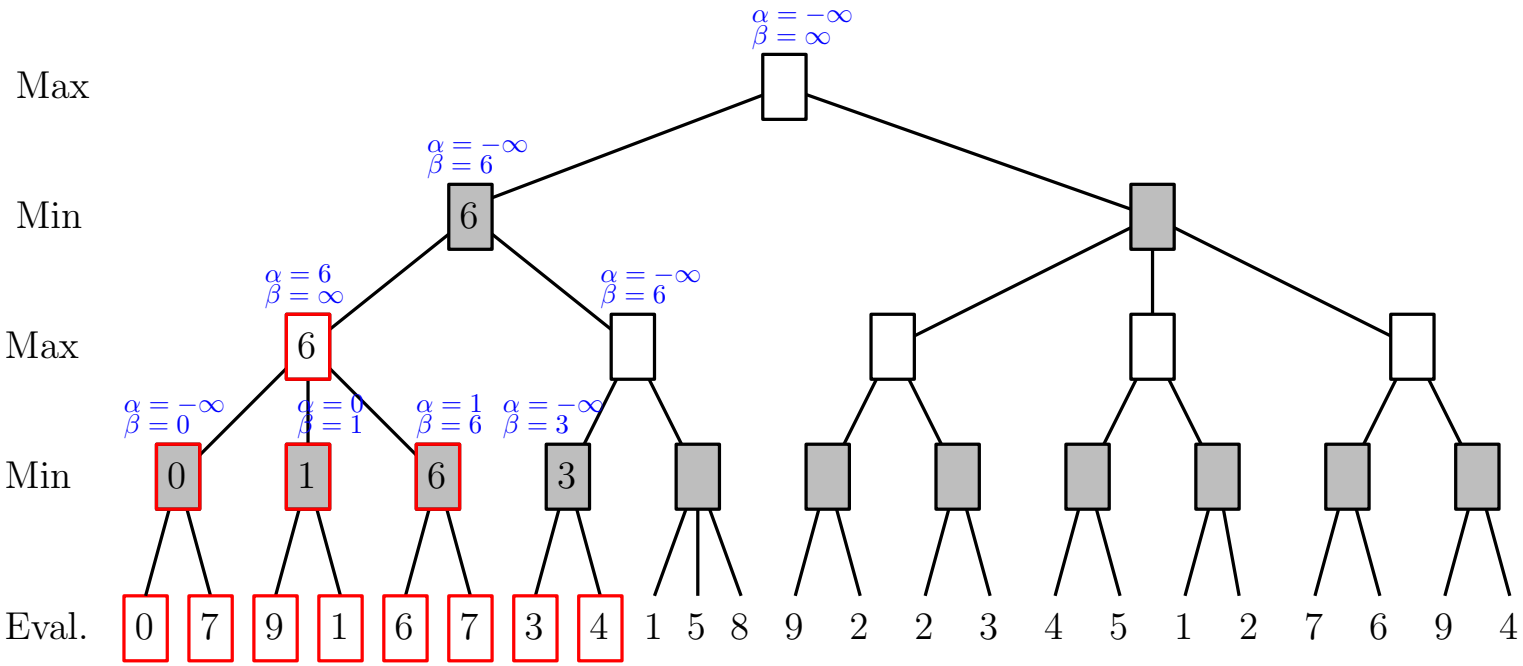


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

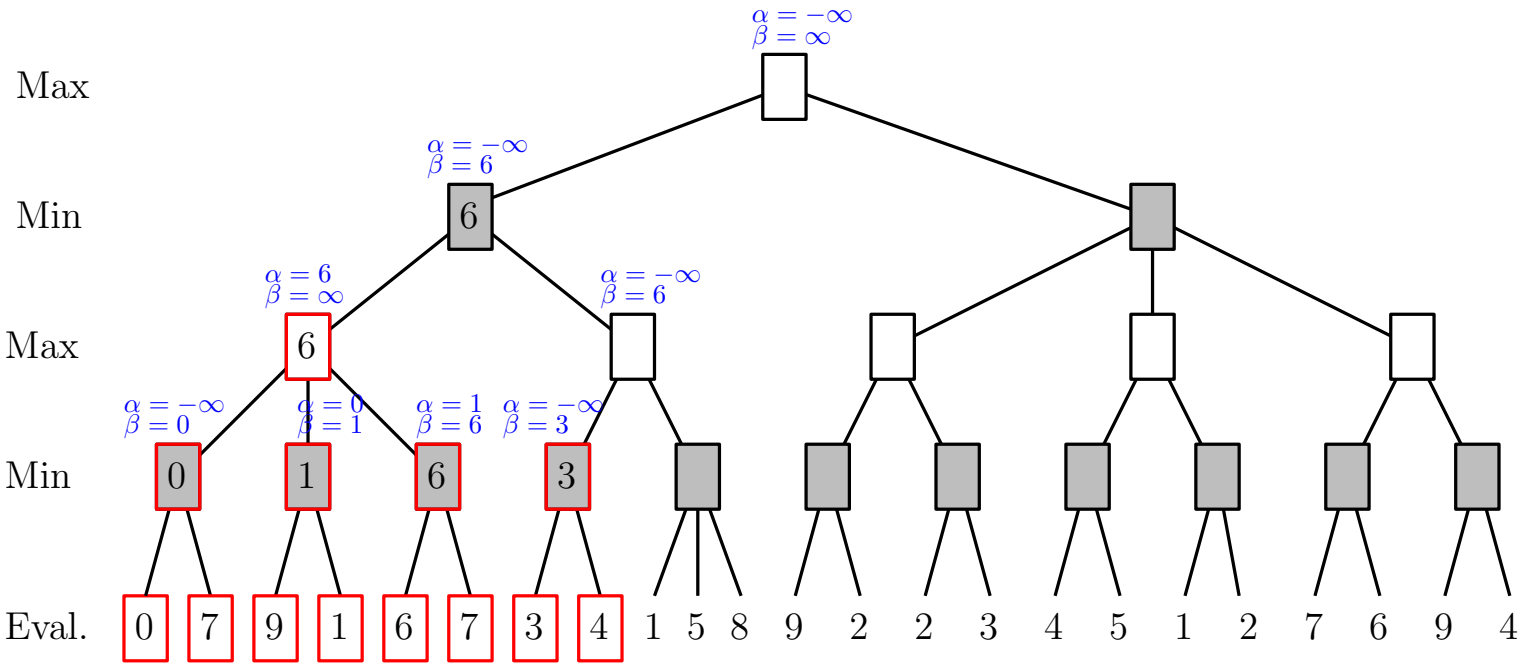


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

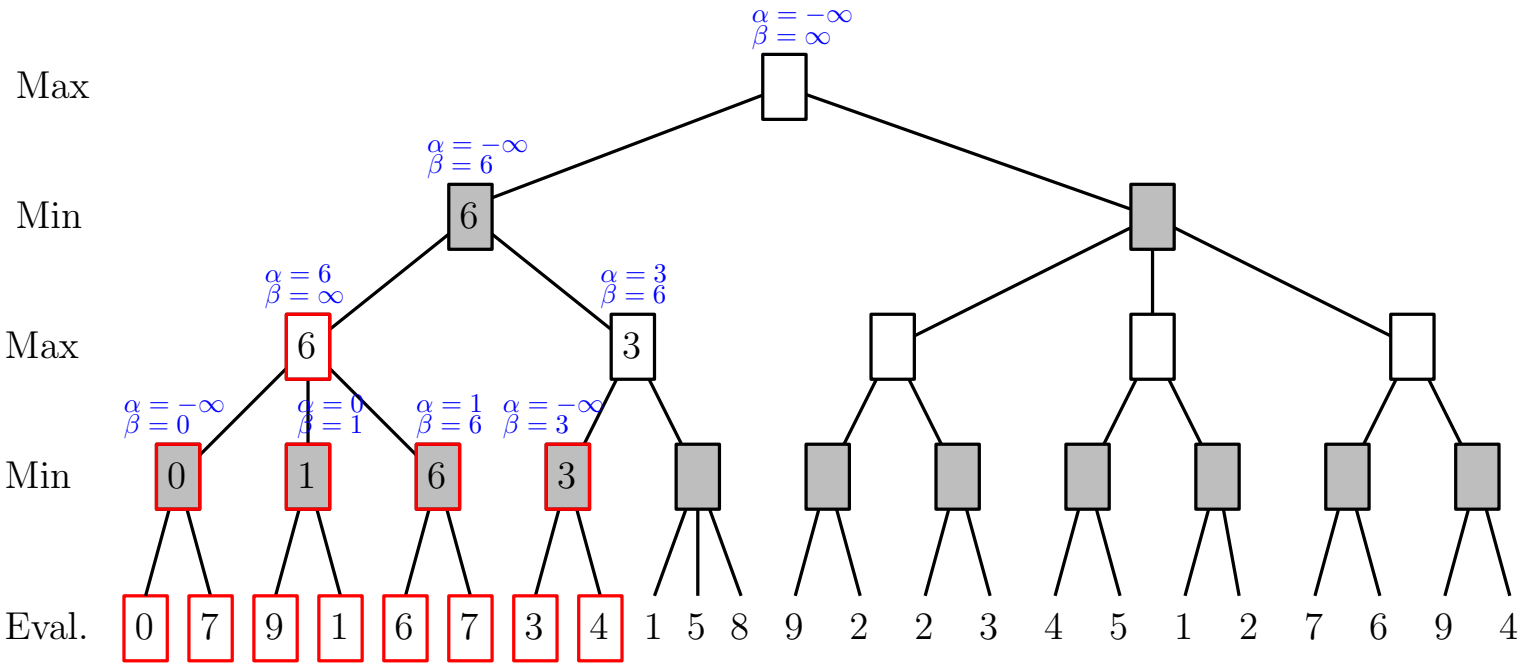


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

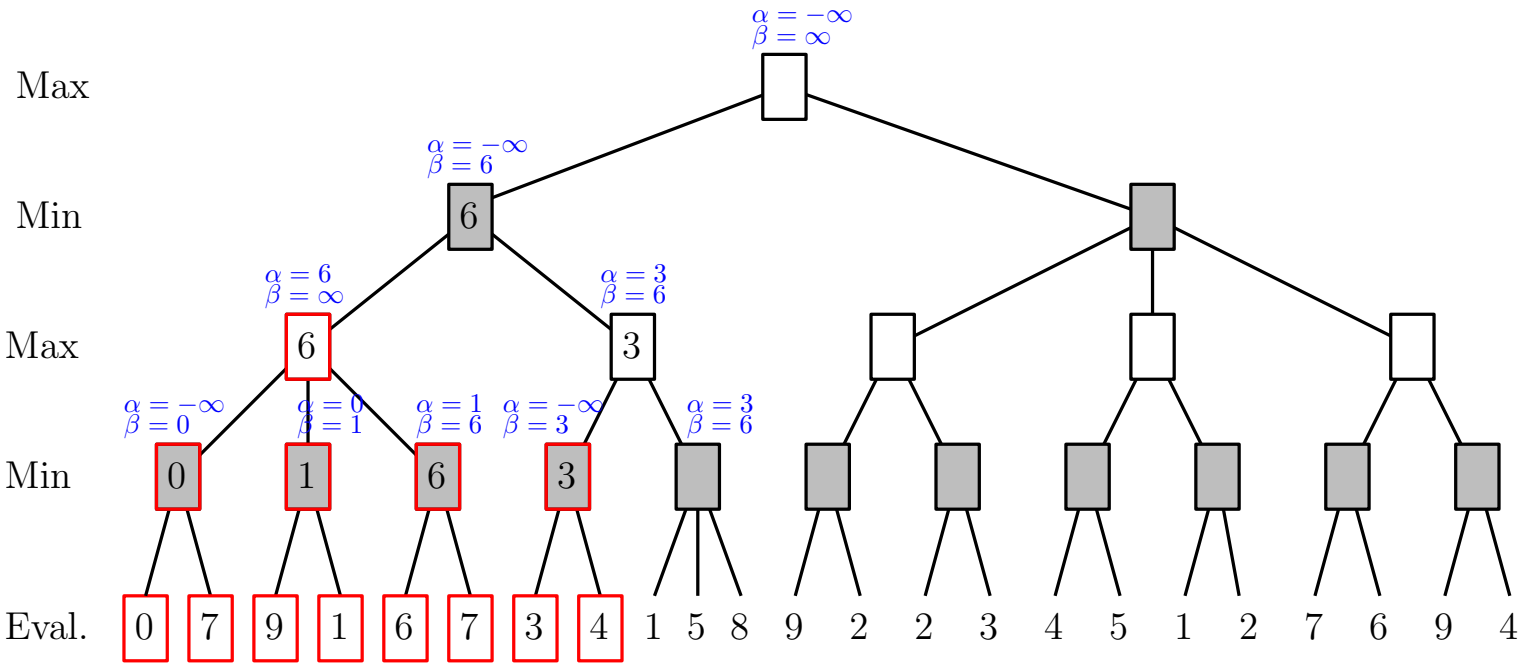


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

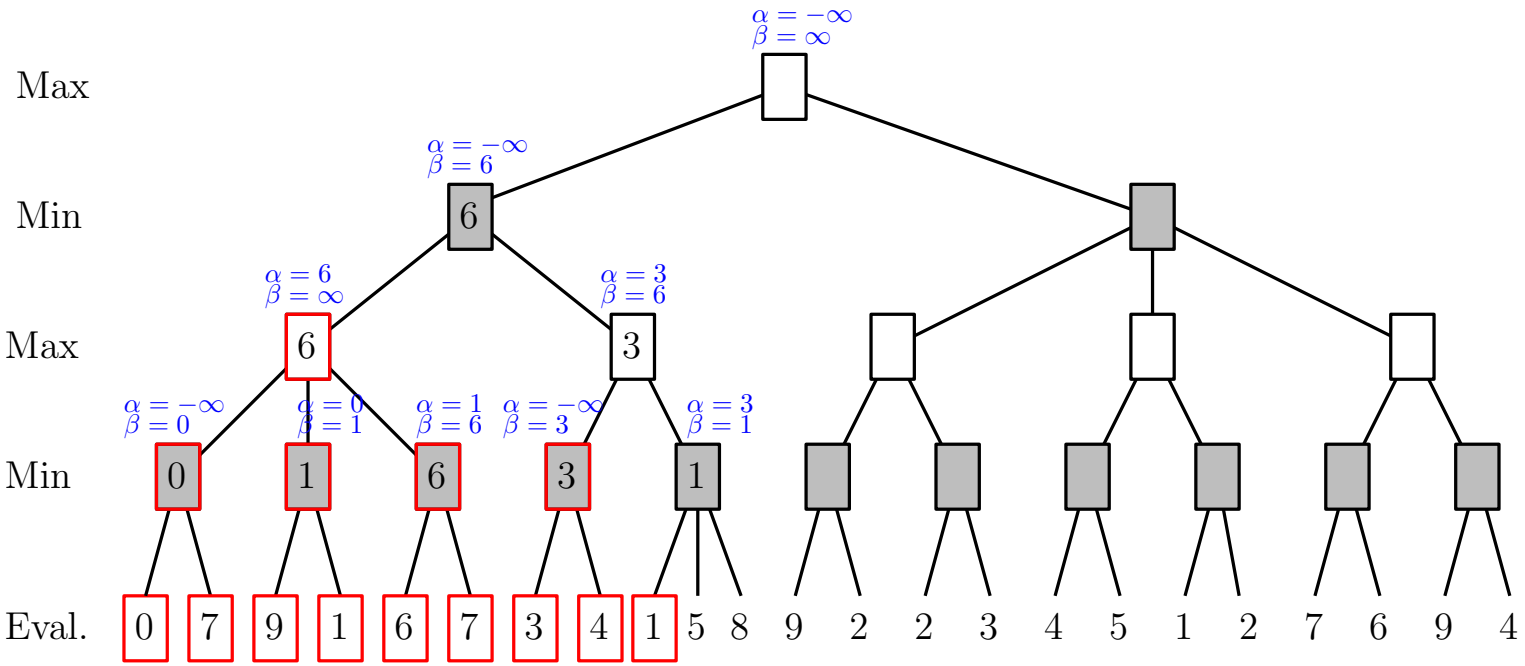


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

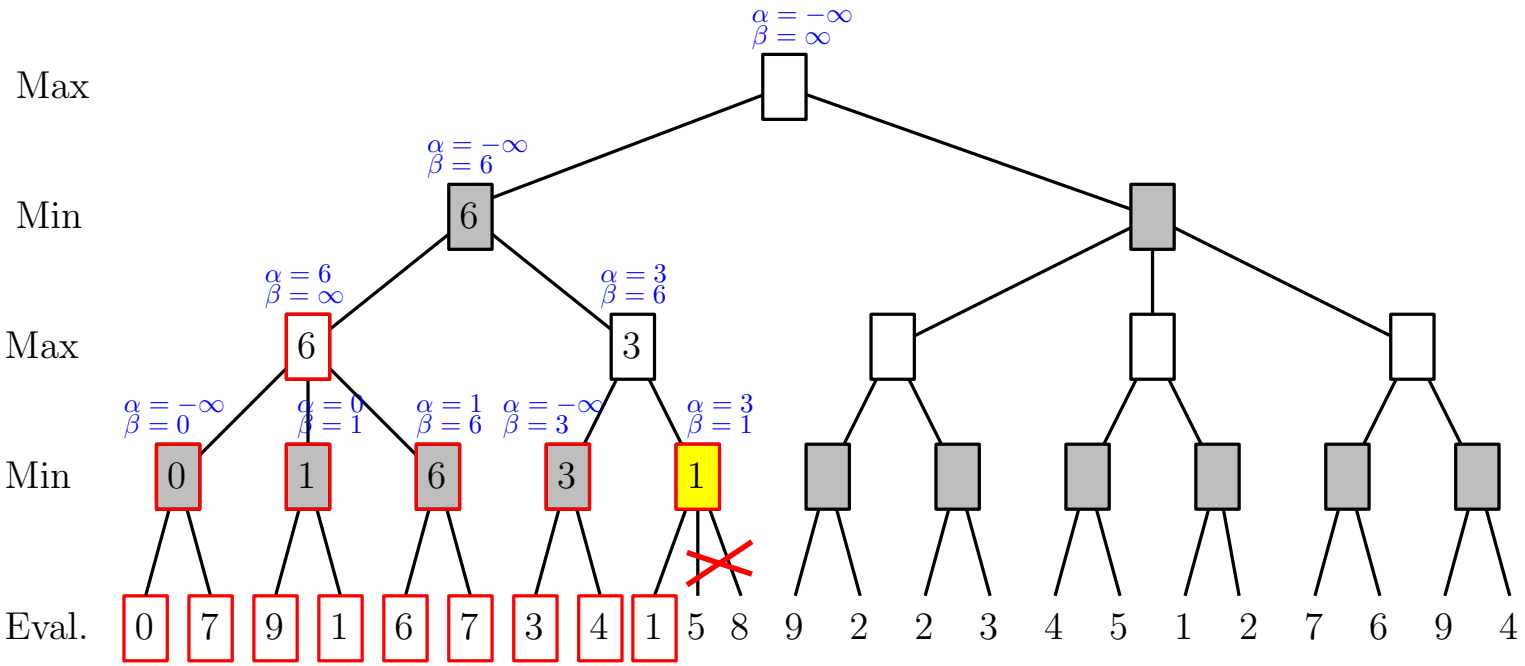


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

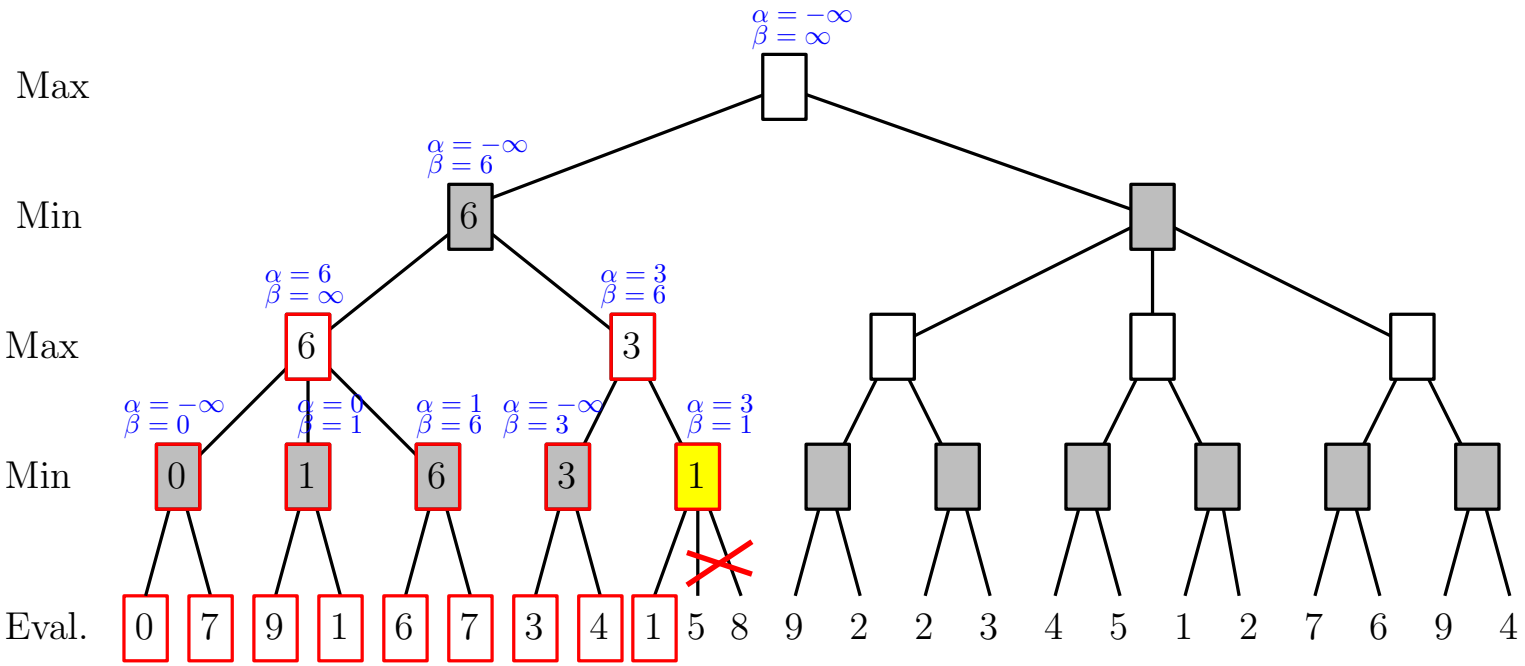


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

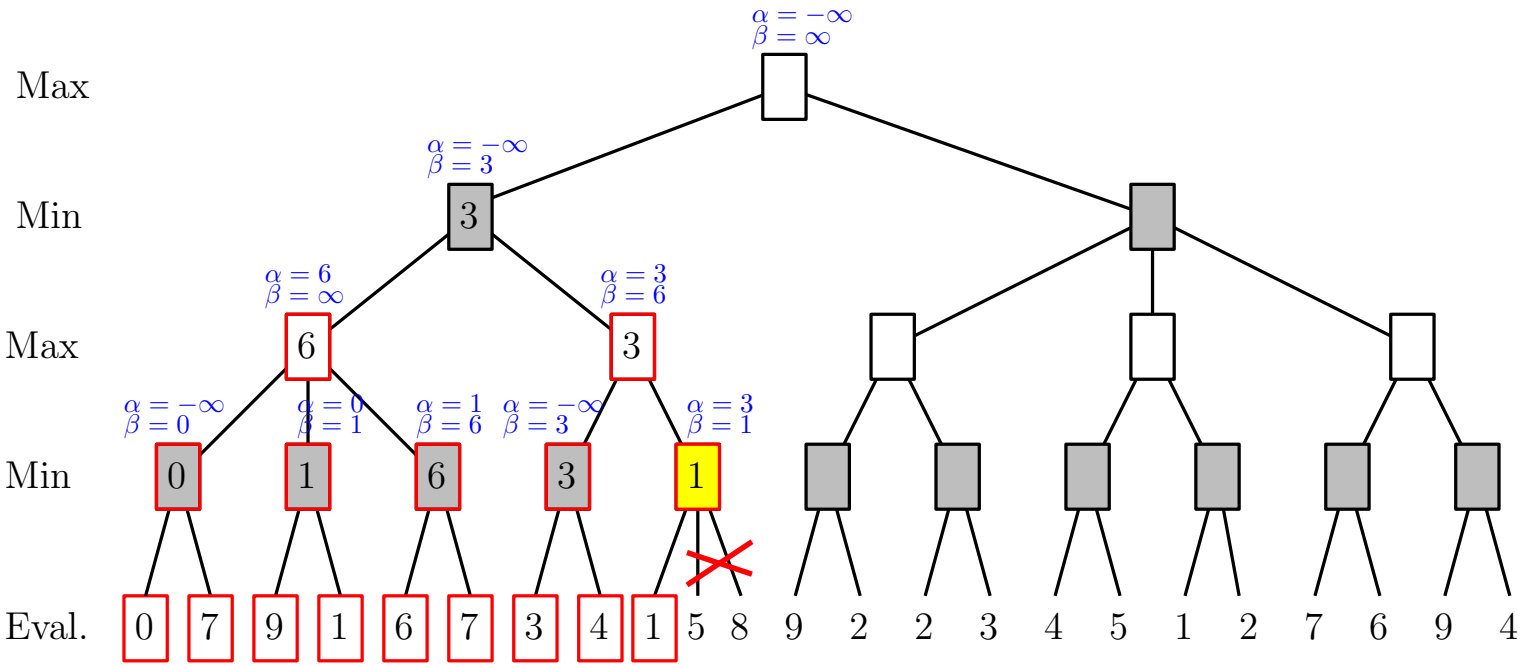


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

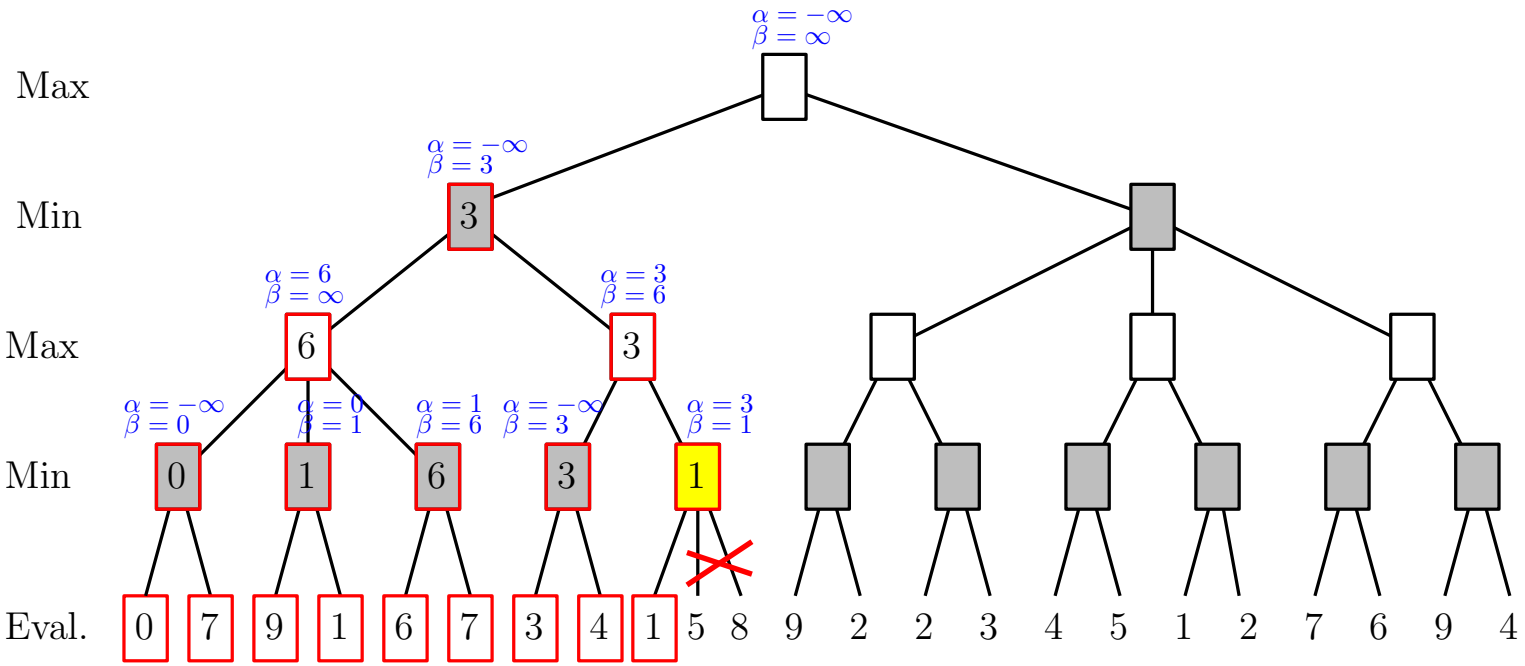


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

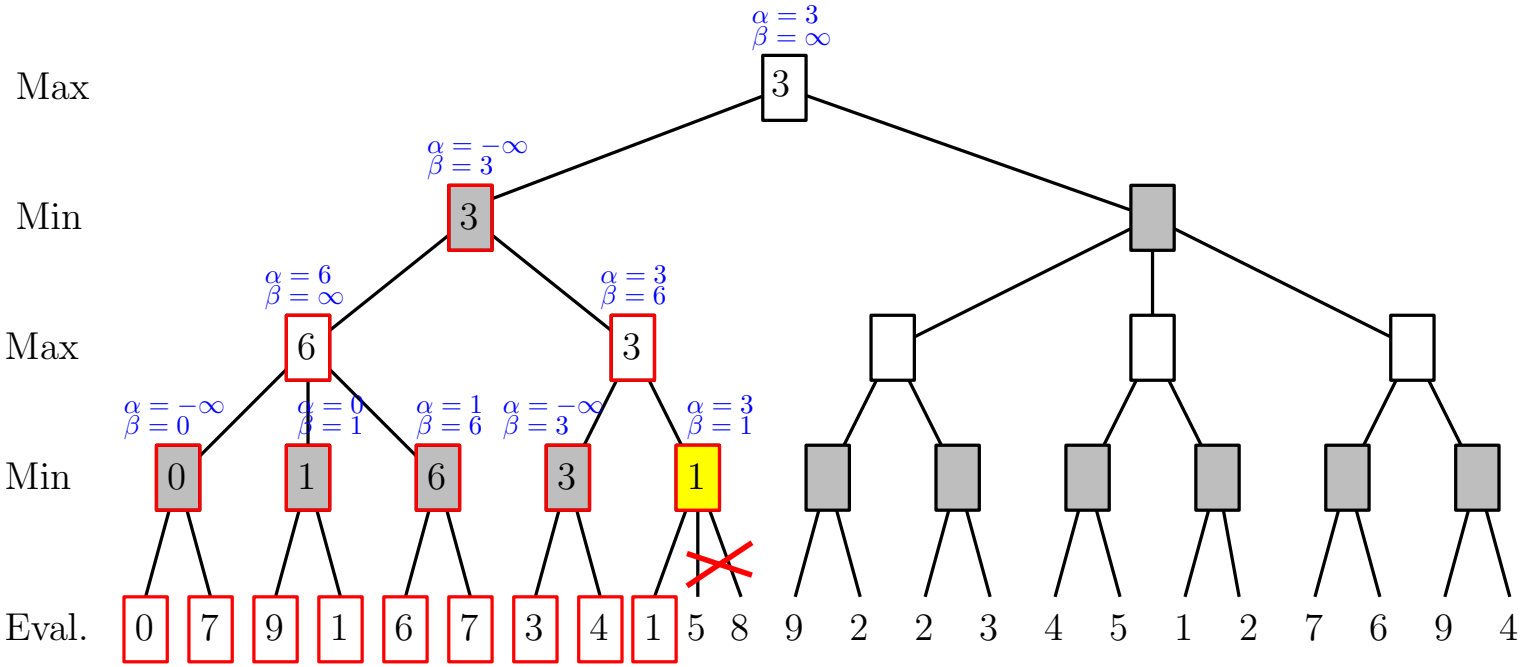


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

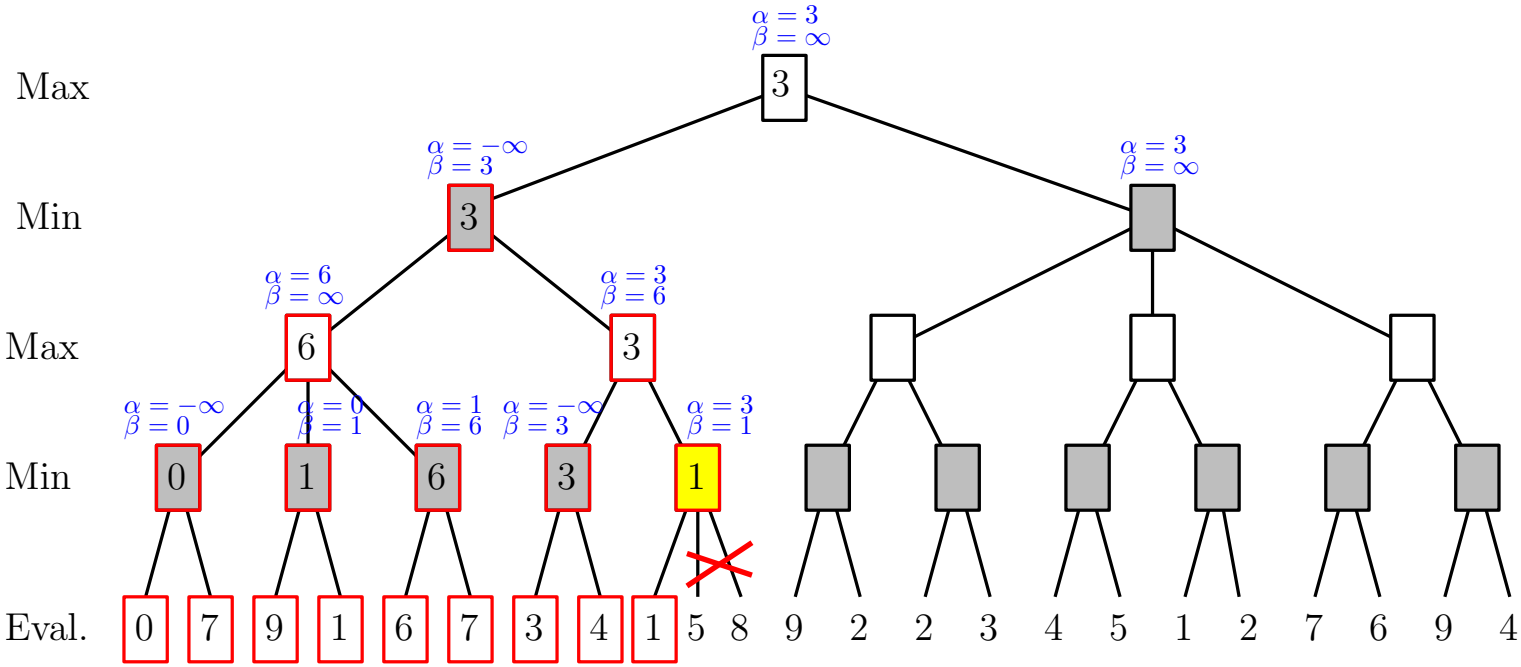


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

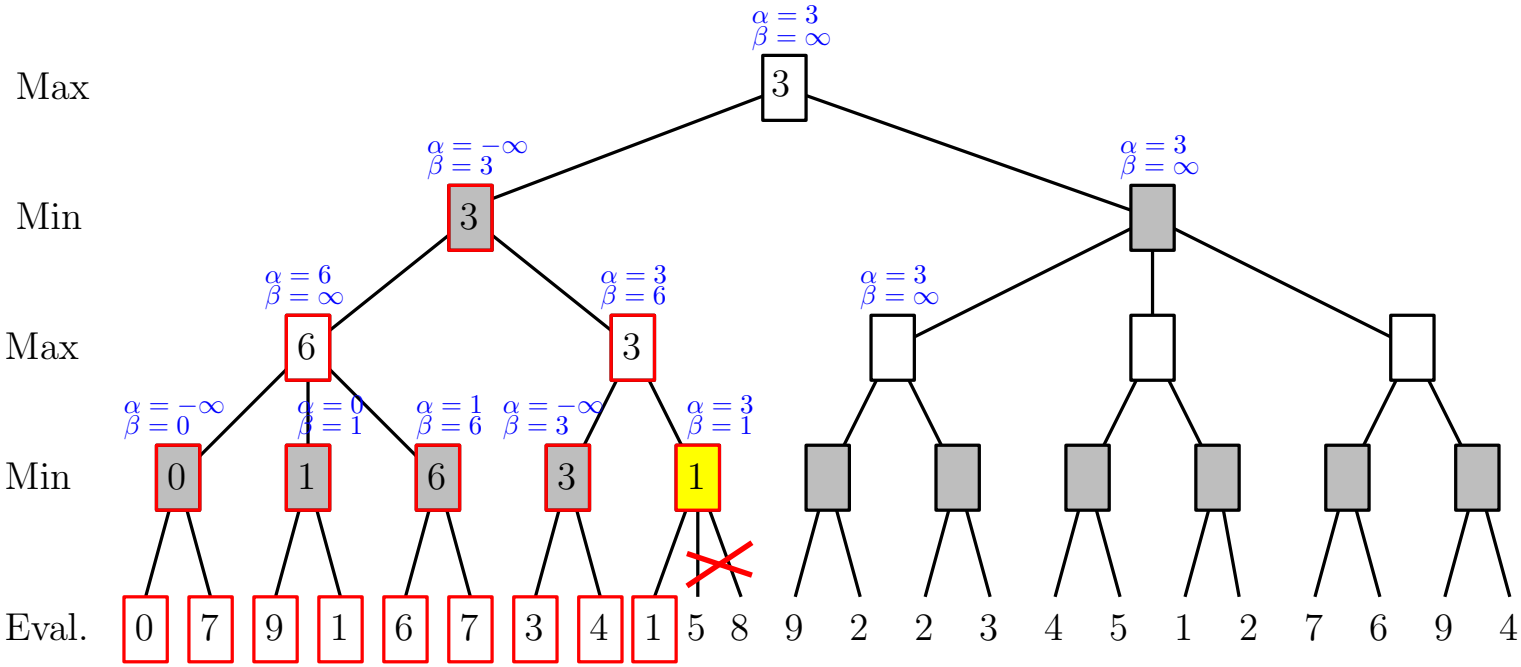


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

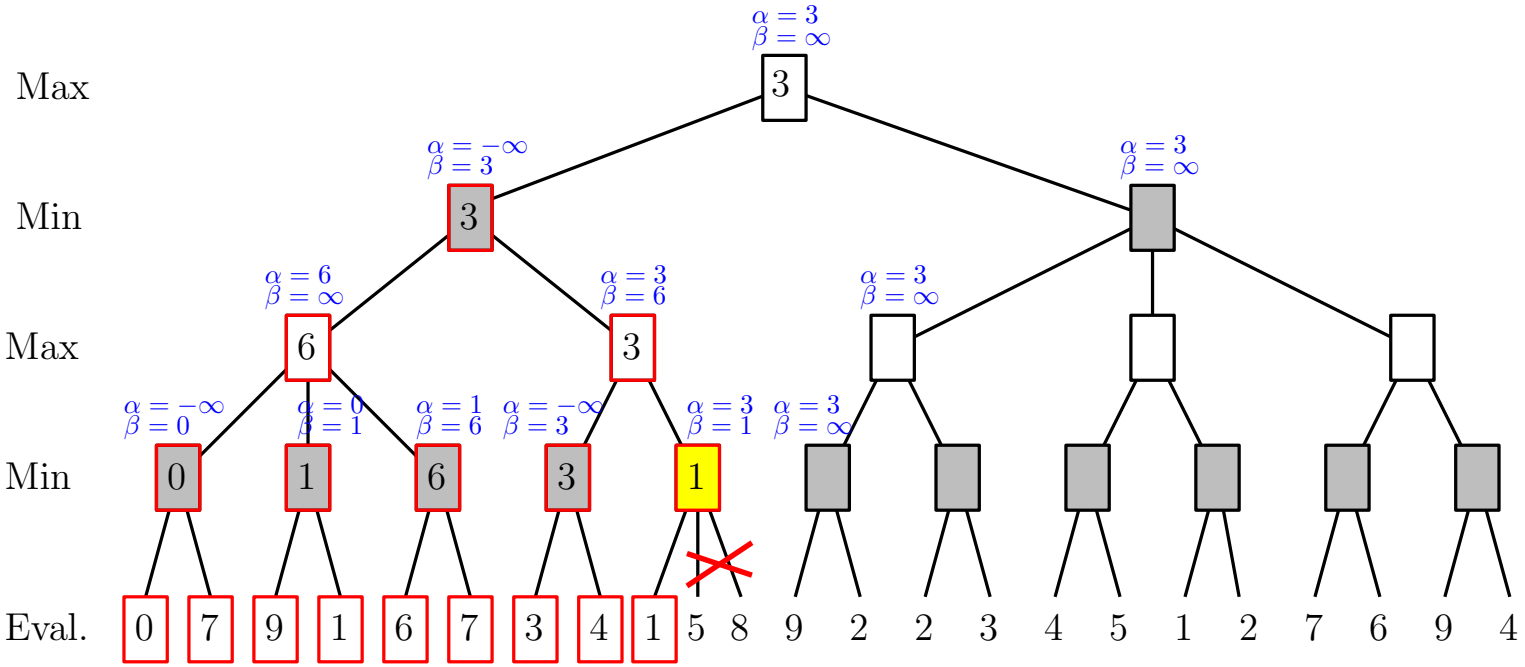


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

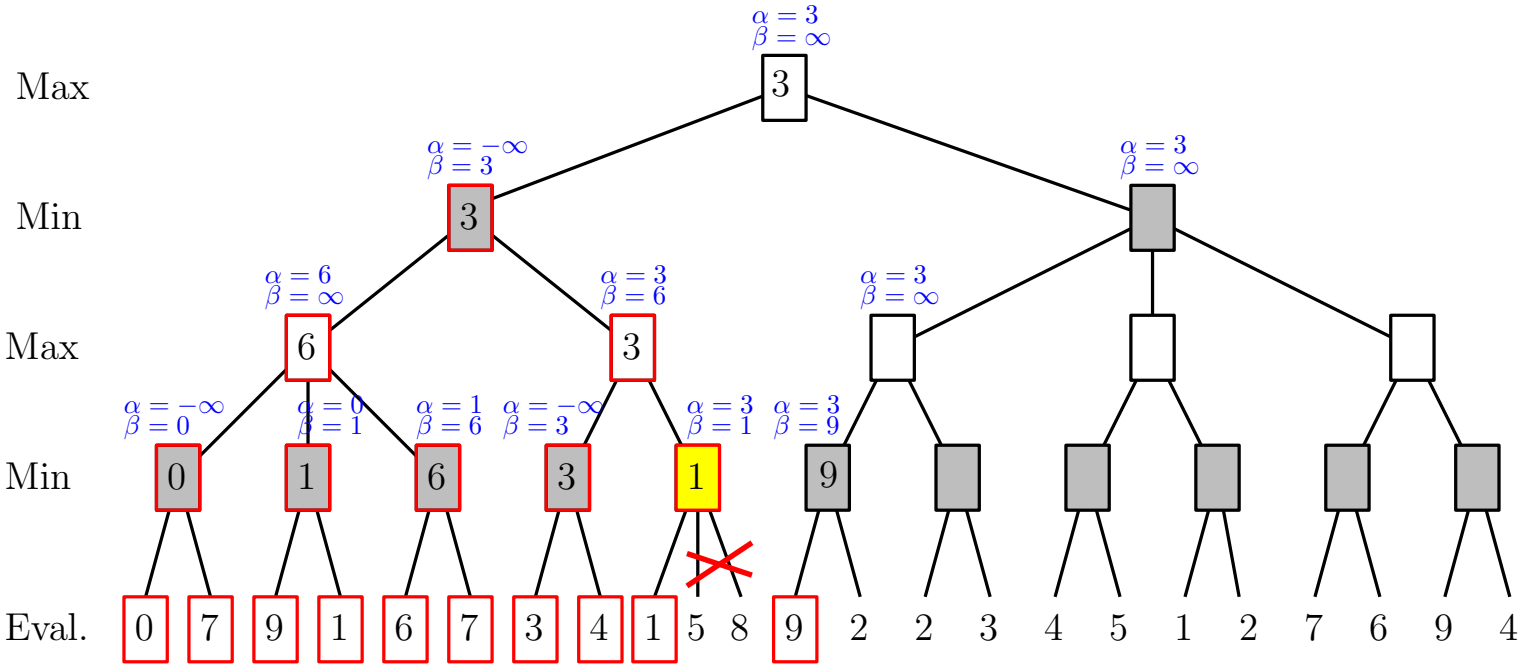


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

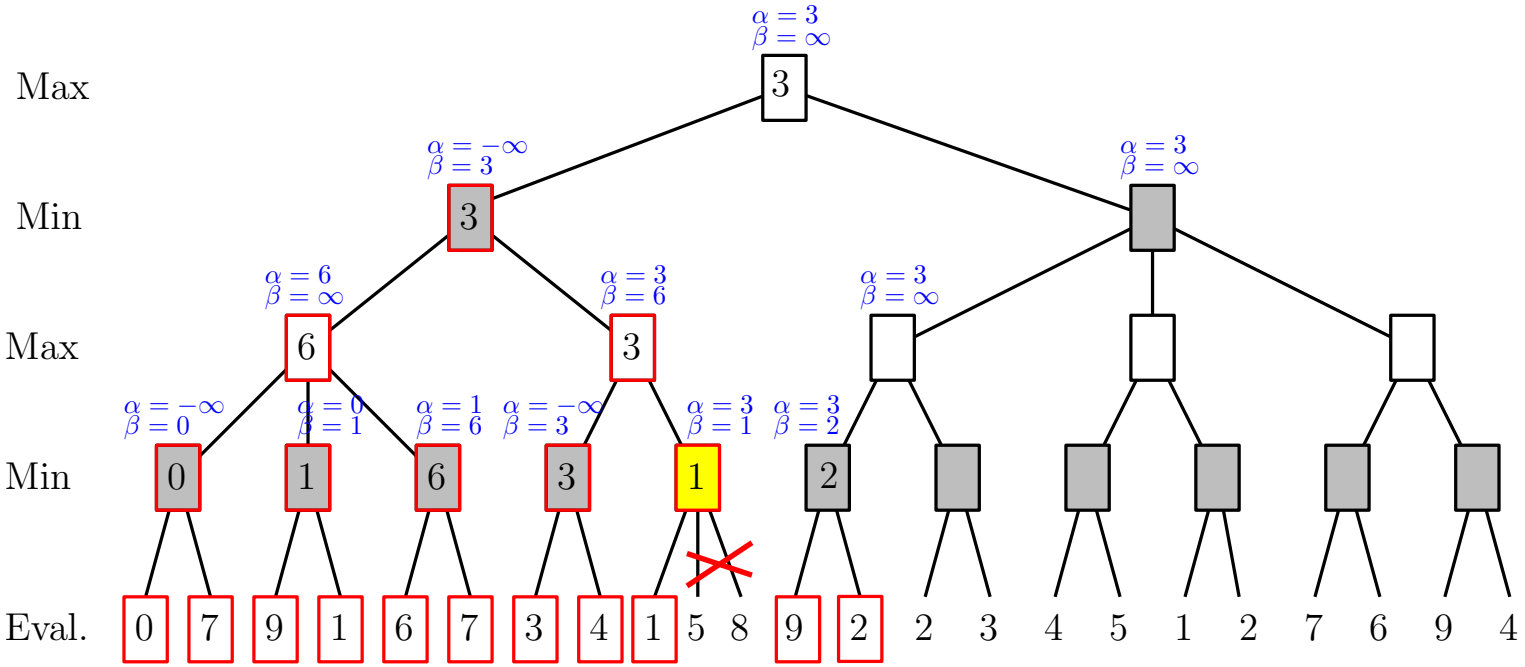


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

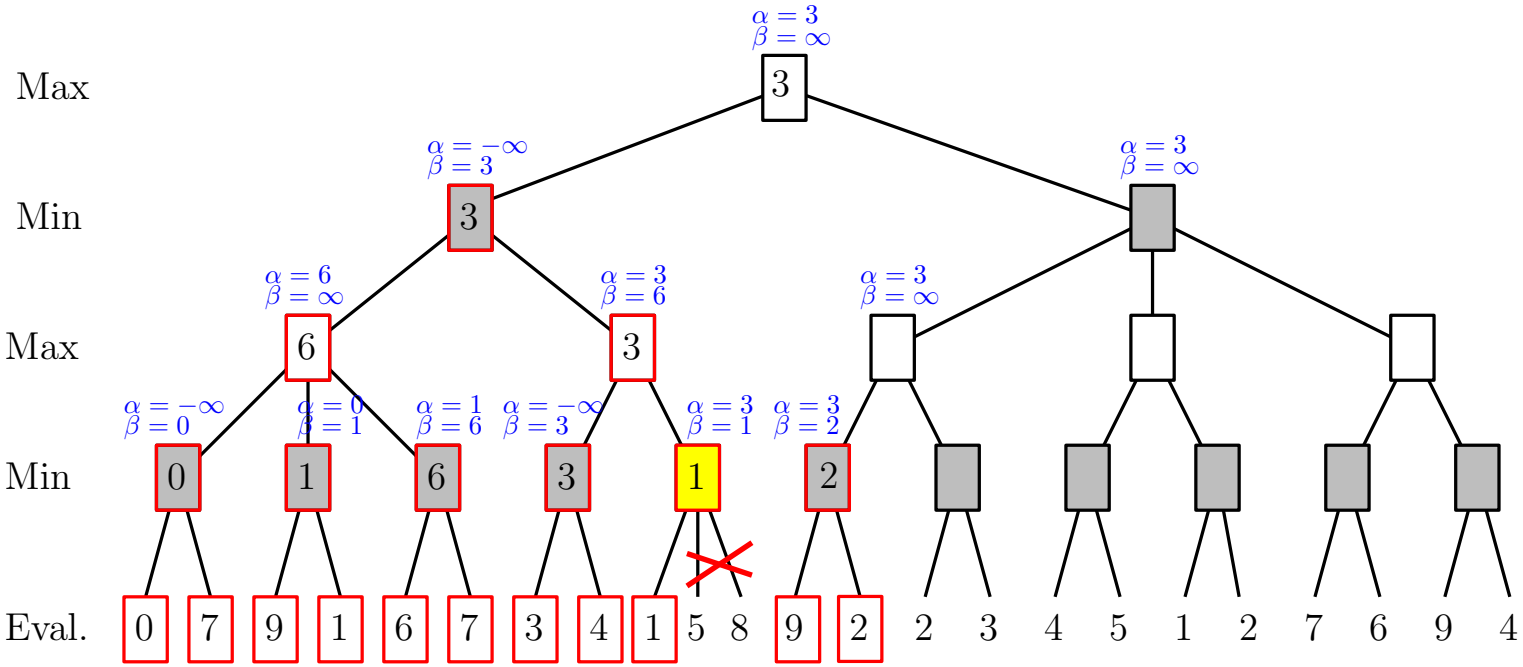


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

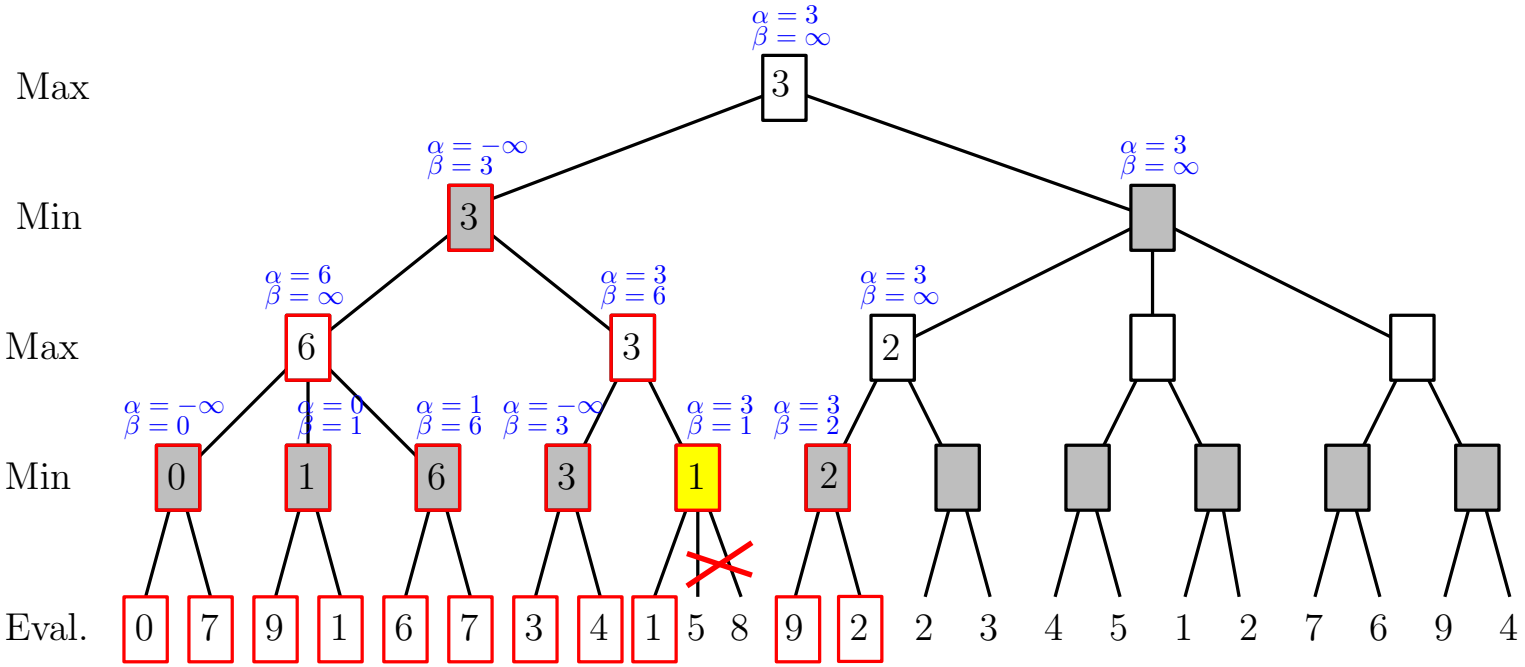


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

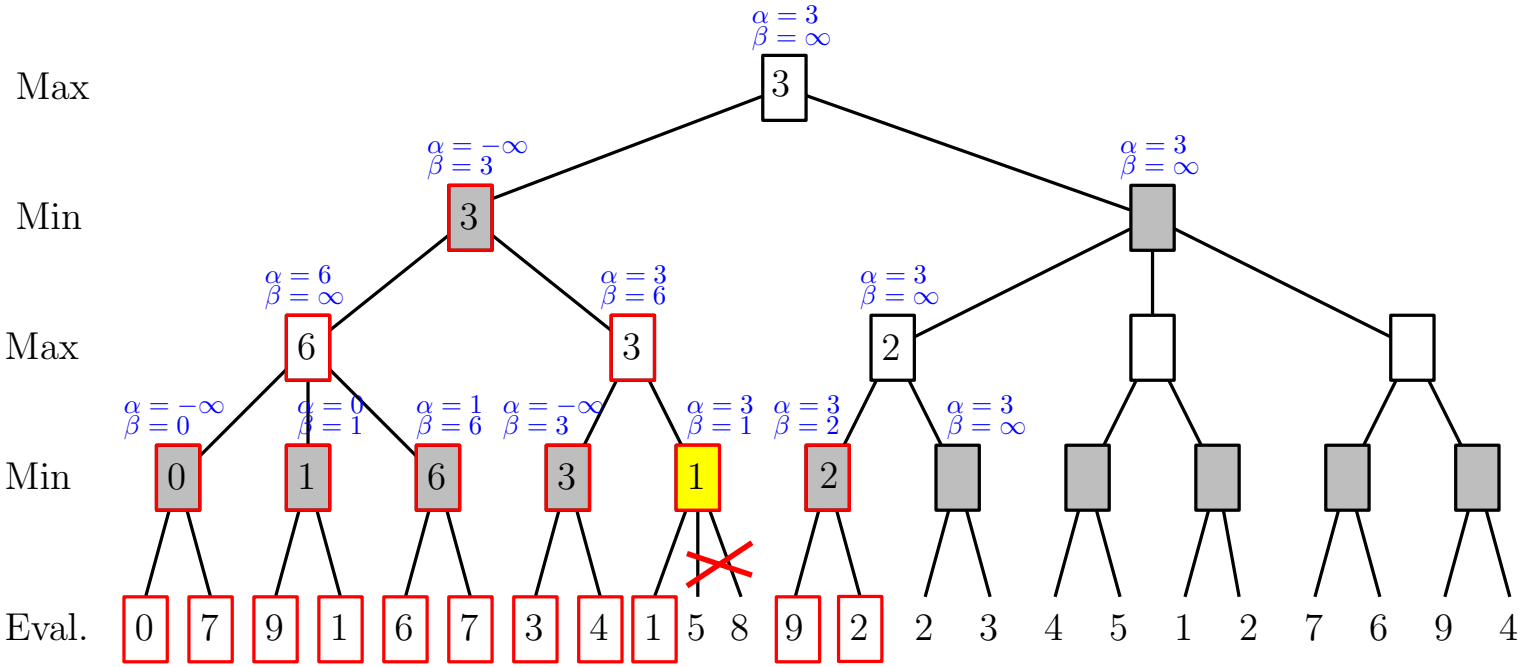


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

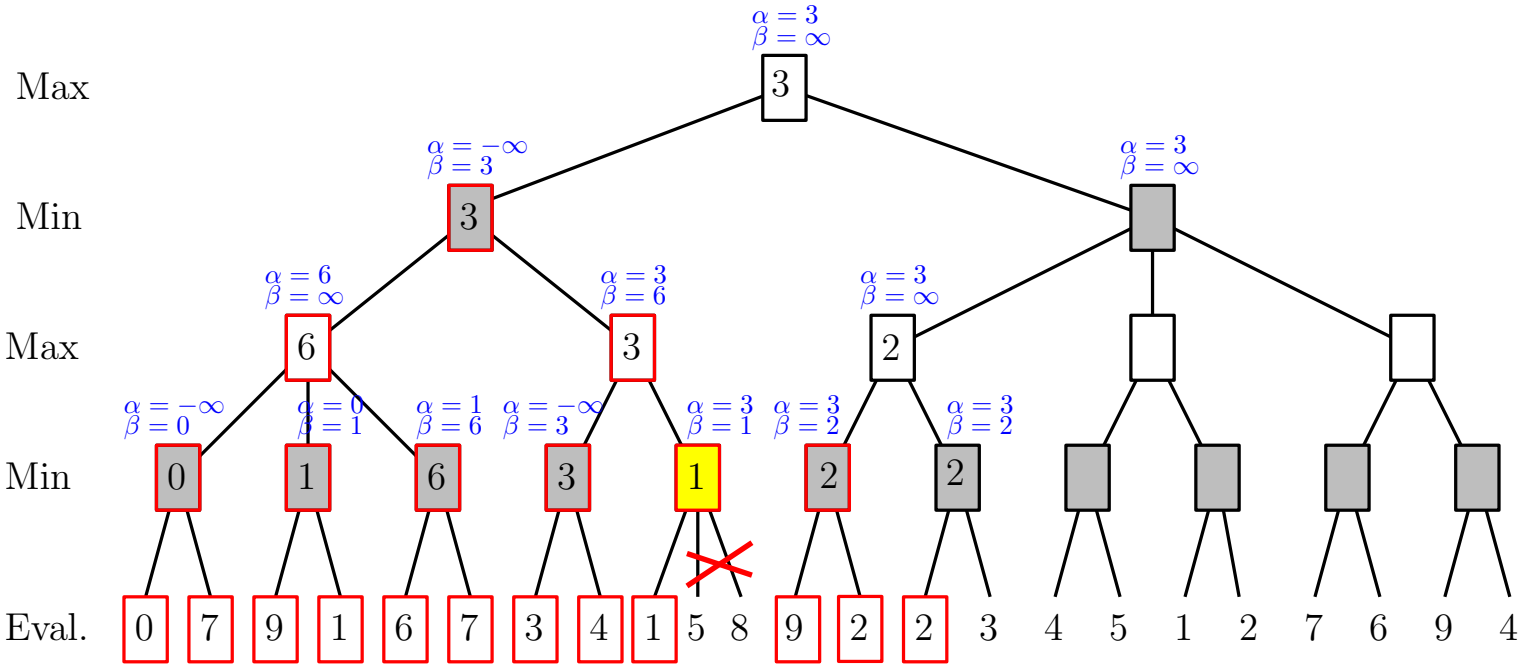


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

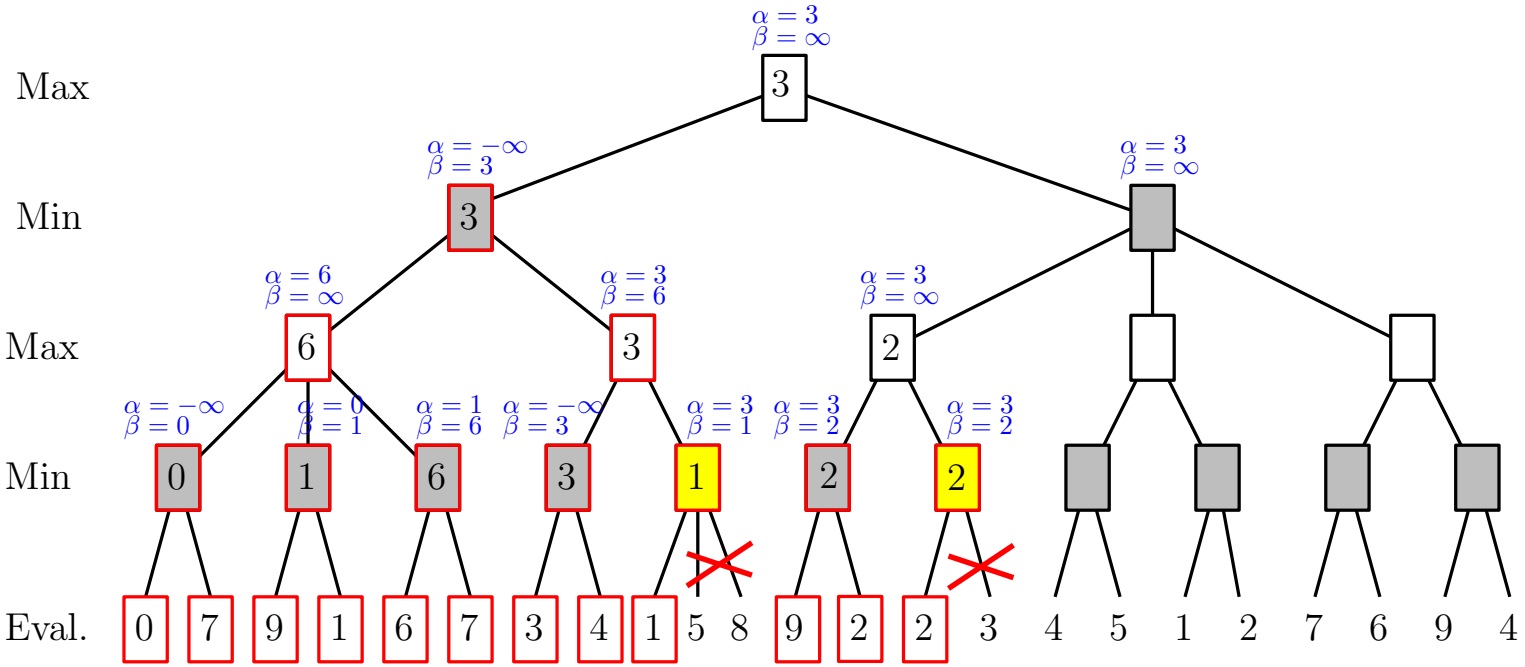


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

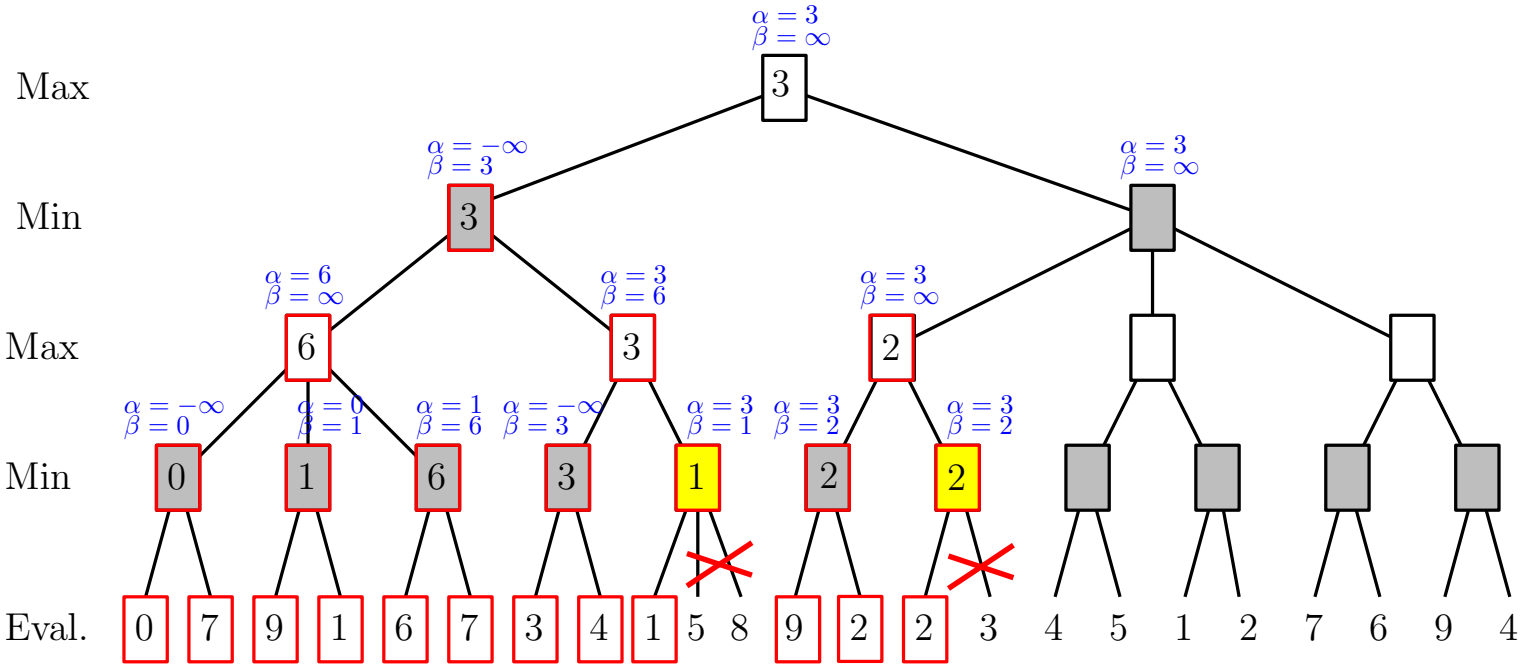


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

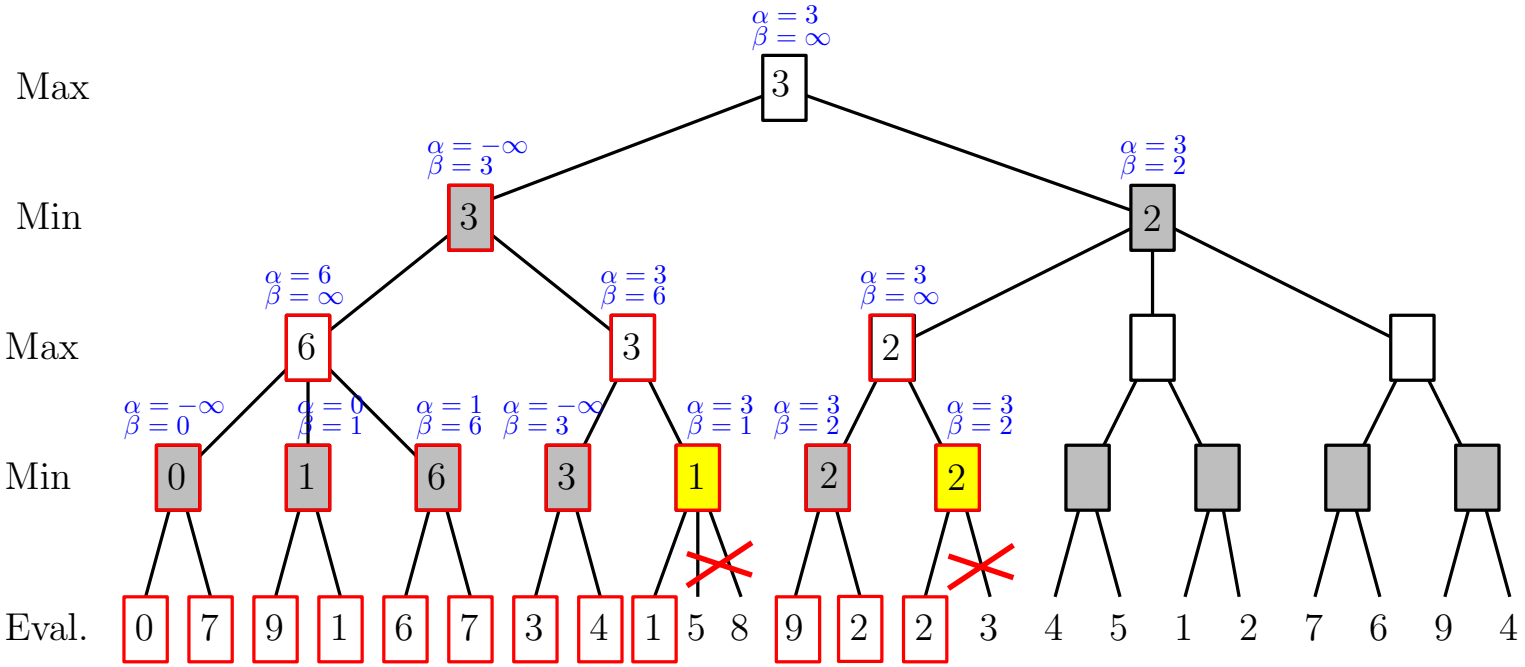


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

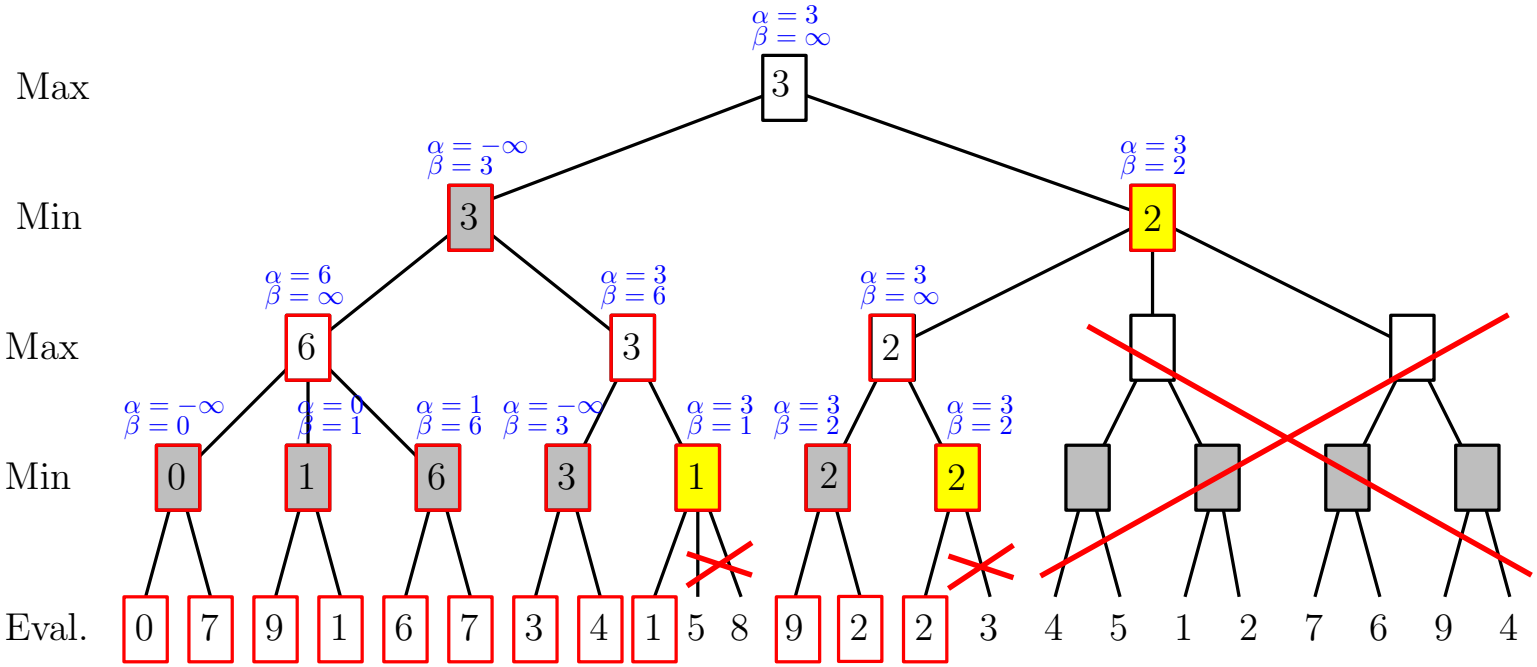


Figure: A more detailed illustration of Alpha-Beta Pruning

Games with Opponents
Alpha-Beta-Pruning

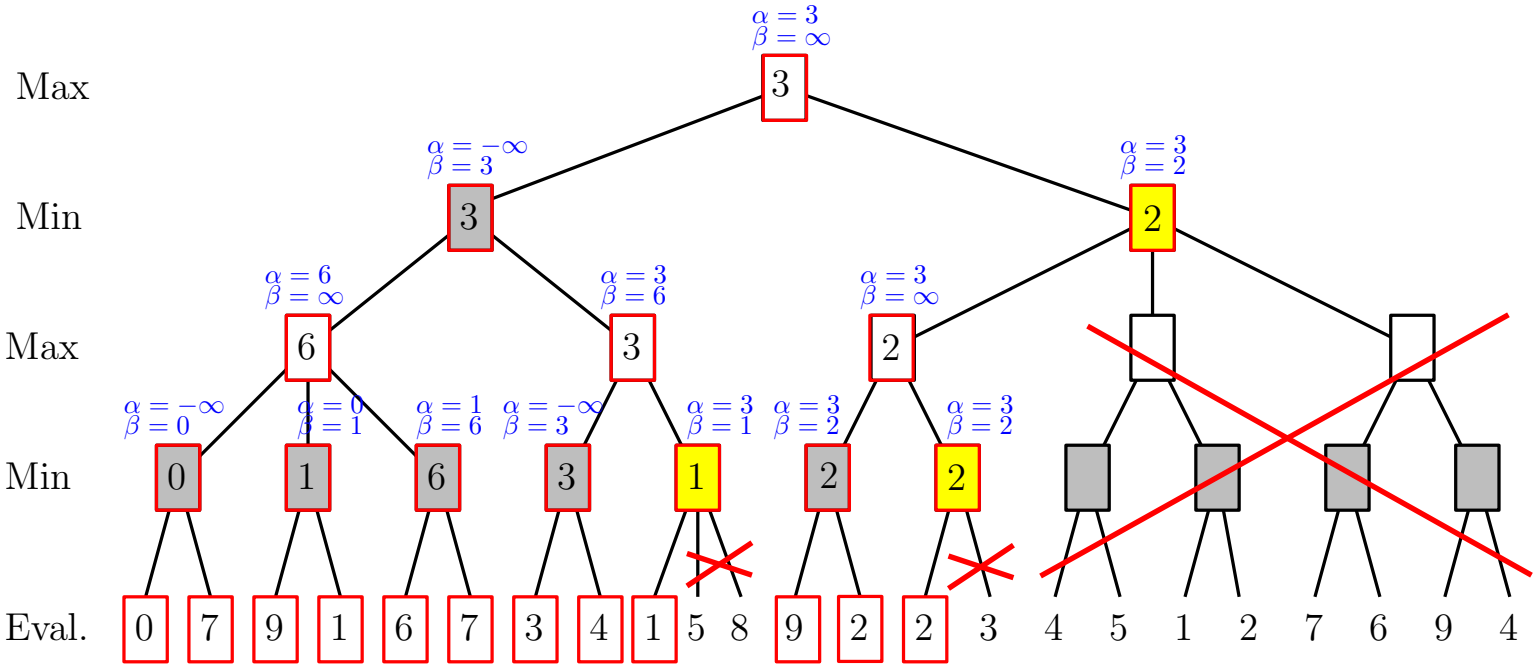


Figure: A more detailed illustration of Alpha-Beta Pruning

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 $6 \approx \sqrt{35}$.

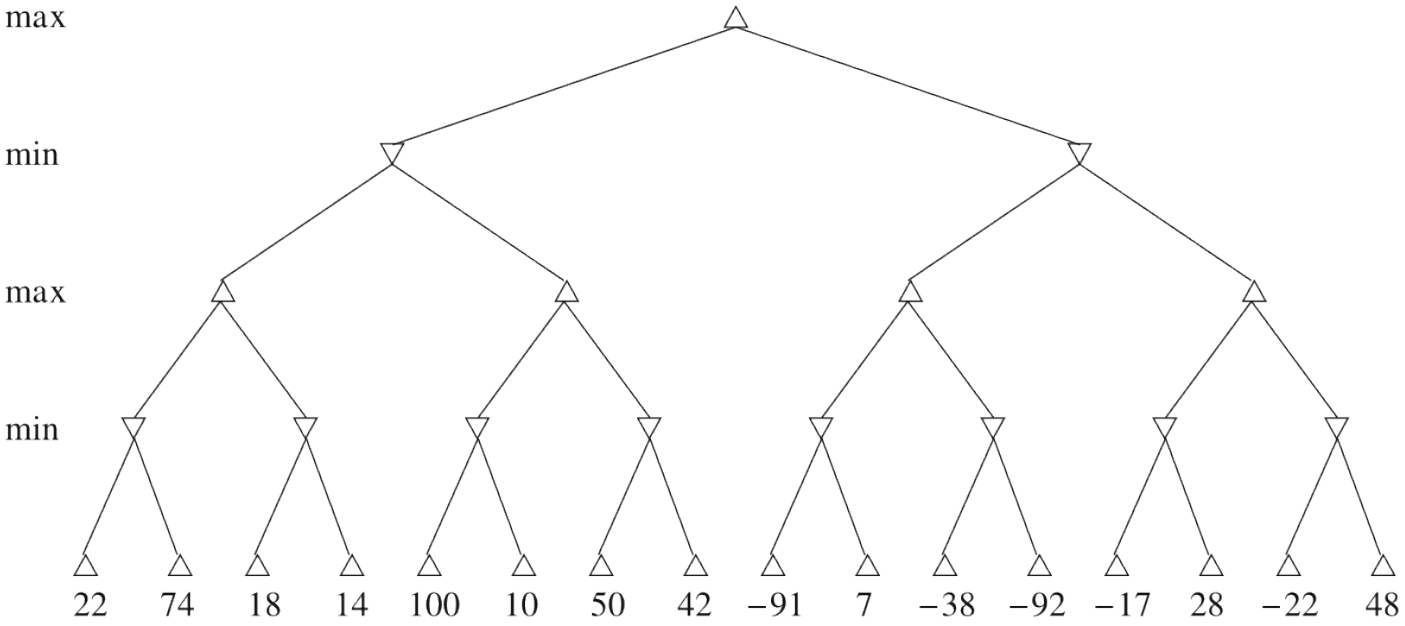
Analysis (cont.): (see [Pearl 1984])

- *Average-Case:*
  - *The child nodes are randomly sorted*.
  - Effective branching factor $\approx b^{3/4} \Rightarrow n_d \approx b^{3d/4}$ leaf nodes.
  - 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 $\sqrt{35} \approx 6$.

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

- e.g. dice games
- In the game tree, there are three types of levels in the sequence Max, dice, Min, dice, ..., where each dice roll node branches six ways.
- Average the values of all rolls

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 7

- 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*.

Heuristic Evaluation Functions

Example 7 (cont.)

Optimizing weights by *machine-learning methods*.

- Expert is only asked about relevant features $f_1(s), f_2(s), \ldots$
- A machine learning process is used to find an evaluation function that is as close to optimal as possible.
  - Start with an initial pre-set evaluation function (determined by the learning process).
  - Let the chess program play.
  - At the end of the game a rating is derived from the result (victory, defeat, or draw).
  - Based on this rating, the evaluation function is changed with the goal of making fewer mistakes next time.
- Problems:
  - *Credit Assignment*
  - positive or negative feedback only at the end
  - no ratings for individual moves
  - Feedback for actions of the past? $\Rightarrow$ *Reinforcement Learning*
- *Most of the world-best chess computers still work without machine-learning techniques*. Reasons:
  - Reinforcement learning has large computation times
  - Manually created heuristics are already heavily optimized.

Latest Research

Definition by Elaine Rich: *Artificial Intelligence is the study of how to make computers do things at which, at the moment, people are better.*

Direct comparison of computer and human in a game

1950 Claude Shannon, Konrad Zuse, John von Neumann: first chess programs

1955 Arthur Samuel: Program that learns to play checkers on a IBM 701.

- archived games, every individual move had been rated by experts.
- Program plays against itself.
- Credit Assignment: For each individual position during a game it compares the evaluation by the function $B(s)$ with the one calculated by alpha-beta pruning and changes $B(s)$ accordingly.

1961 Samuel's checkers program beat the fourth-best checkers player in the USA.

Latest Research

1990 Tesauro: Reinforcement learning.
Learning backgammon program named TD-Gammon, which played at the world champion level.

1997 IBM's Deep Blue defeated the chess world champion Gary Kasparov with a score of 3.5 games to 2.5.
Deep Blue could on average compute 12 half-moves ahead with alpha-beta pruning.

2004 Hydra: Chess computer on parallel machine

- Uses 64 parallel Xeon processors with 3 GHz computing power and 1 GByte memory each.
- Software: Christian Donninger (Austria) and Ulf Lorenz, Christopher Lutz (Germany).
- Position evaluation: FPGA Co-processor (Field Programmable Gate Arrays).
  Evaluates 200 million possitions per second.

Latest Research

2004 Hydra: Chess computer on parallel machine

- Hydra can on average compute about 18 half-moves ahead.
- Hydra often makes moves which grand champions cannot comprehend, but which in the end lead to victory.
- Alpha-beta search with relatively general, well-known heuristics and a good hand-coded position evaluation.
- Hydra works without learning.

2009 Pocket Fritz 4, running on a PDA, won the Copa Mercosur chess tournament in Buenos Aires.

- 9 wins and 1 draw against 10 excellent human chess players, three of them grandmasters.
- search engine HIARCS 13 (Higher Intelligence Auto Response Chess System) searches less than 20,000 positions per second.
- Pocket Fritz 4 is about 10,000 slower than Hydra $\Rightarrow$ HIARCS 13 uses better heuristics to decrease the effective branching factor than Hydra

Latest Research
Go

- square board with 361 squares
- 181 white, 180 black stones
- average branching factor: about 300
- after 4 half-moves: $8 \cdot 10^9$ positions
- classical game tree search processes have no chance!
- Pattern recognition on the board!
- *Deep Learning for pattern recognition, reinforcement learning, and Monte Carlo tree search* lead to successful results.
  - January of 2016: Google [Silver et al. 2016] and Facebook [Tian and Zhu 2015] published the breakthrough concurrently.
  - That same month, the program AlphaGo, developed and presented in [Silver et al. 2016] by Google DeepMind, defeated European Go champion Fan Hui 5:0. Two months later, Korean player Lee Sedol, one of the best in the world, was defeated 4:1.
  - The program plays hundreds of thousands of games against itself and uses the results (win, loss, draw) to learn the best possible heuristic score for a given position. Monte Carlo tree search is used as a replacement for Minimax search, which is not suitable for Go.

References


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


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.


Tian, Yuandong and Yan Zhu (2015). "Better computer go player with neural network and long-term prediction." In: *arXiv preprint*. arXiv: 1511.06410.

References (cont.)


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.


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


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

References (cont.)

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

Korf, Richard E (1985). “Depth-first iterative-deepening: An optimal admissible tree search.” In: *Artificial intelligence* 27.1, pp. 97–109. DOI: 10.1016/0004-3702(85)90084-0.

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