CSE373: Data Structures & Algorithms Lecture 23: Intro to Artificial Intelligence and Game Theory

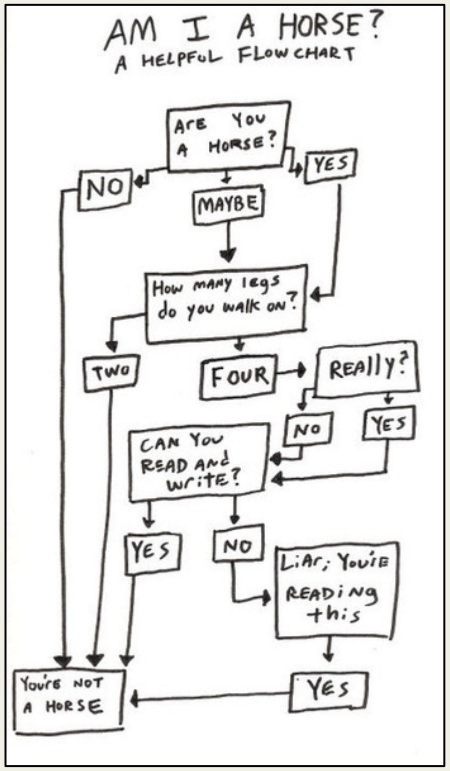
Based on slides adapted Luke Zettlemoyer, Dan Klein, Pieter Abbeel, Dan Weld, Stuart Russell or Andrew Moore

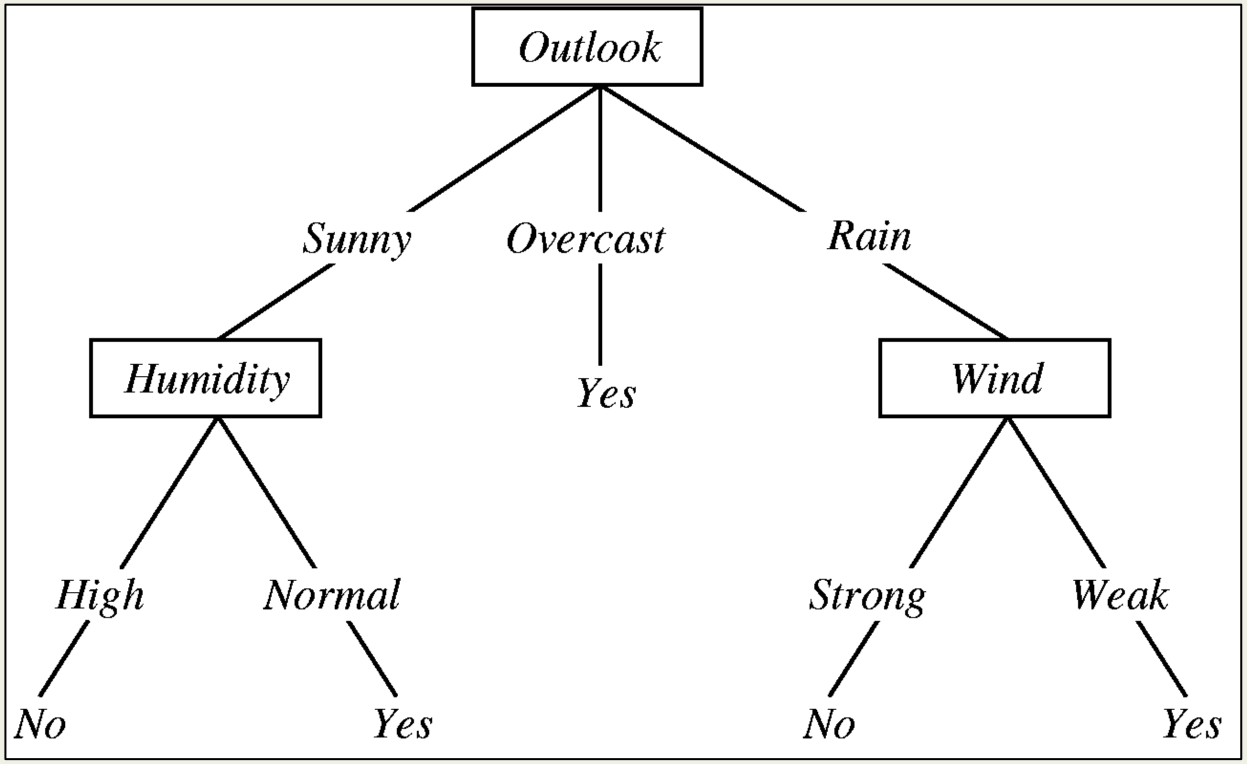
Game Playing State-of-the-Art

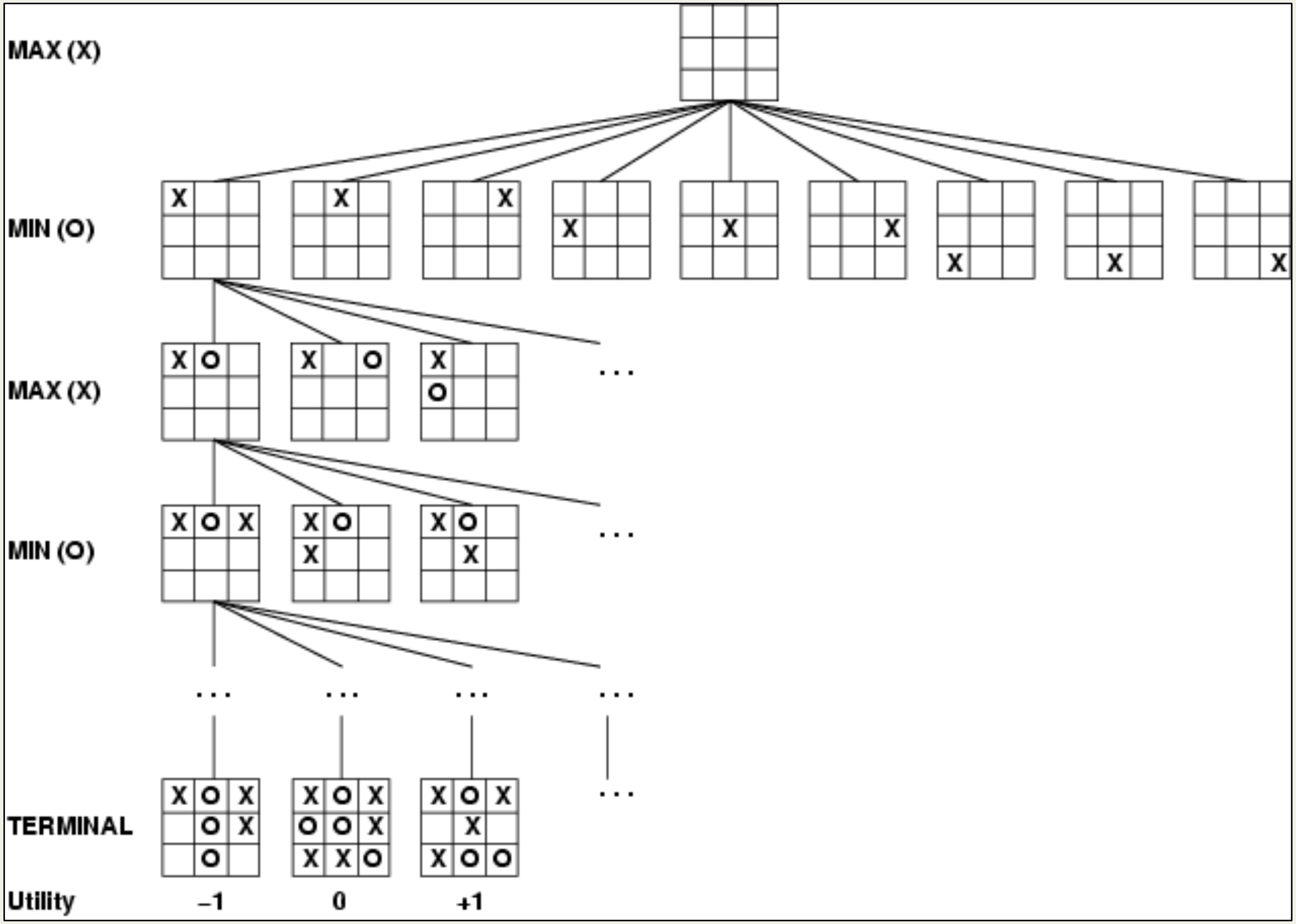
- **Checkers:** Chinook ended 40-year-reign of human world champion Marion Tinsley in 1994. Used an endgame database defining perfect play for all positions involving 8 or fewer pieces on the board, a total of 443,748,401,247 positions. Checkers is now solved!
- **Chess:** Deep Blue defeated human world champion Gary Kasparov in a six-game match in 1997. Deep Blue examined 200 million positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.
- **Othello:** Human champions refuse to compete against computers, which are too good.
- **Go:** Human champions are beginning to be challenged by machines, though the best humans still beat the best machines. In go, $b > 300$, so most programs use pattern knowledge bases to suggest plausible moves, along with aggressive pruning.

Decision Trees

- Graph like computational structure used to assess potential decisions.






MAX (X)
MIN (O)
MAX (X)
MIN (O)
TERMINAL
Utility
–1
0
+1

Game Playing

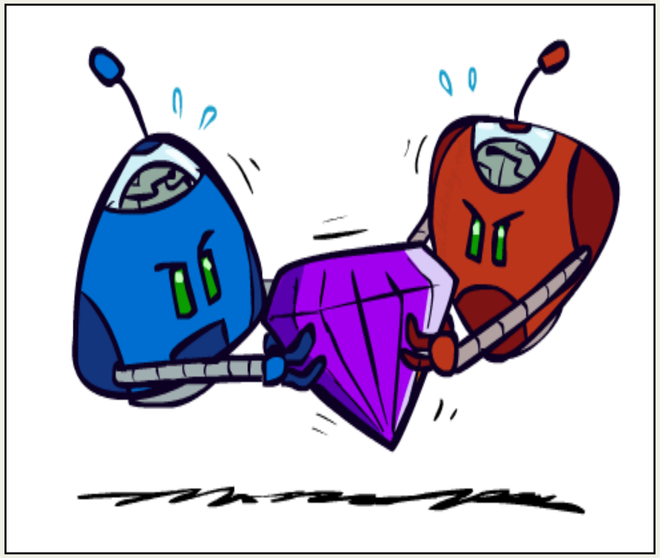
- Many different kinds of games
- Choices:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Perfect information (can you see the state)?
- Want algorithms for calculating a strategy which recommends a move in each state. We will try to combine the algorithms and structures we have seen thus far.

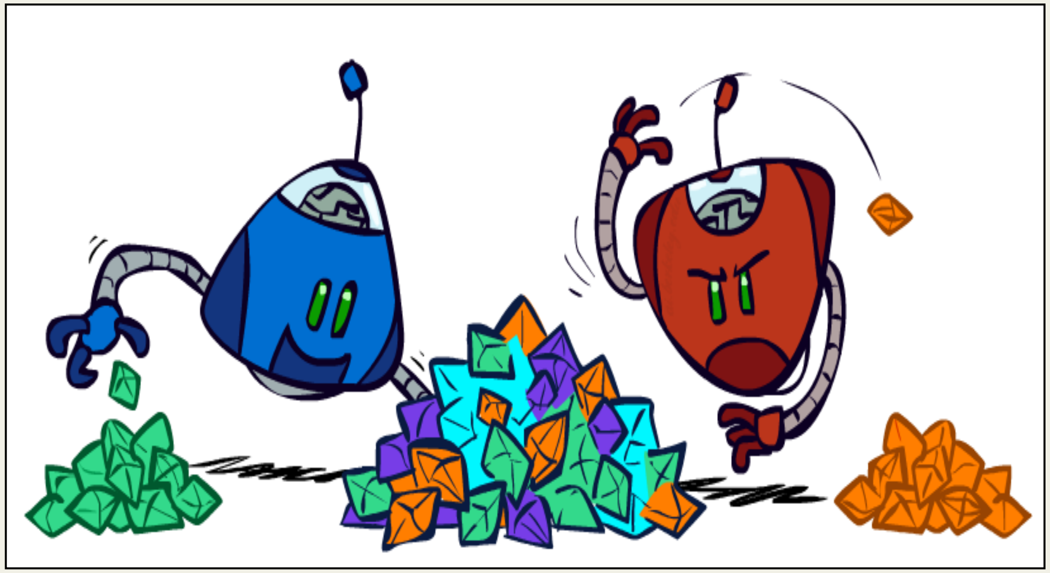
Deterministic Games

- Many possible formalizations, one is:
  - States: S (start at s_0)
  - Players: P = {1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: $S \times A \rightarrow S$

- Solution for a player is a policy: $S \rightarrow A$

Zero-Sum Games





- Zero-Sum Games
  - Players have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

- General Games
  - Players have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, & more are possible

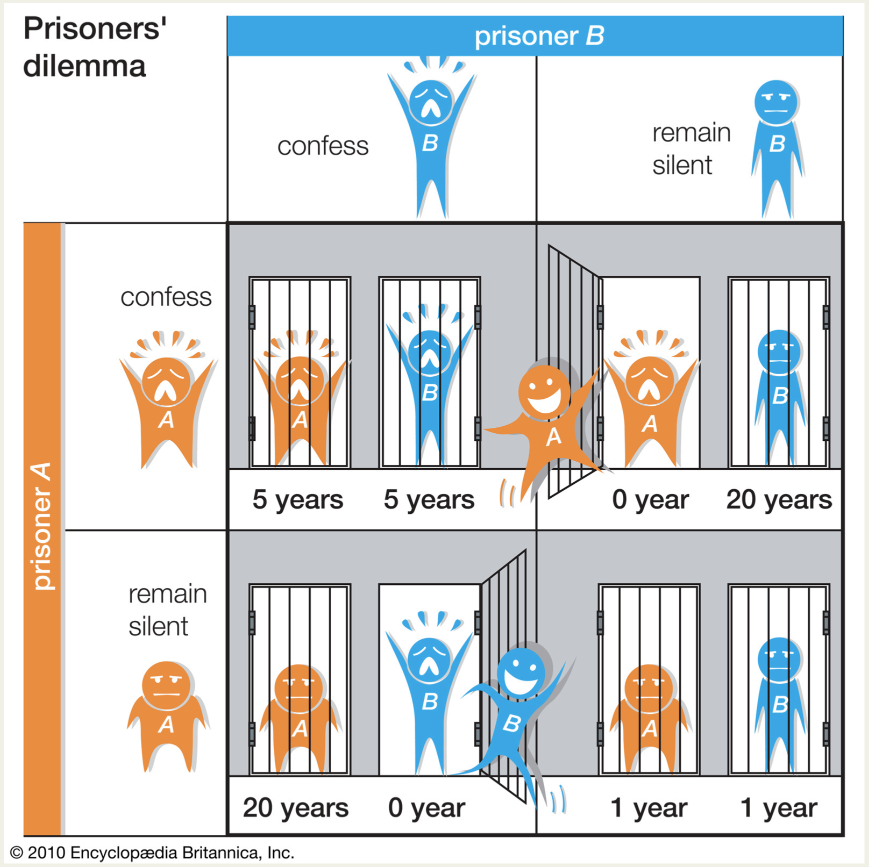
Non Zero-Sum Games

Two criminals are taken in, each prisoner is in solitary confinement with no means of communicating with the other.

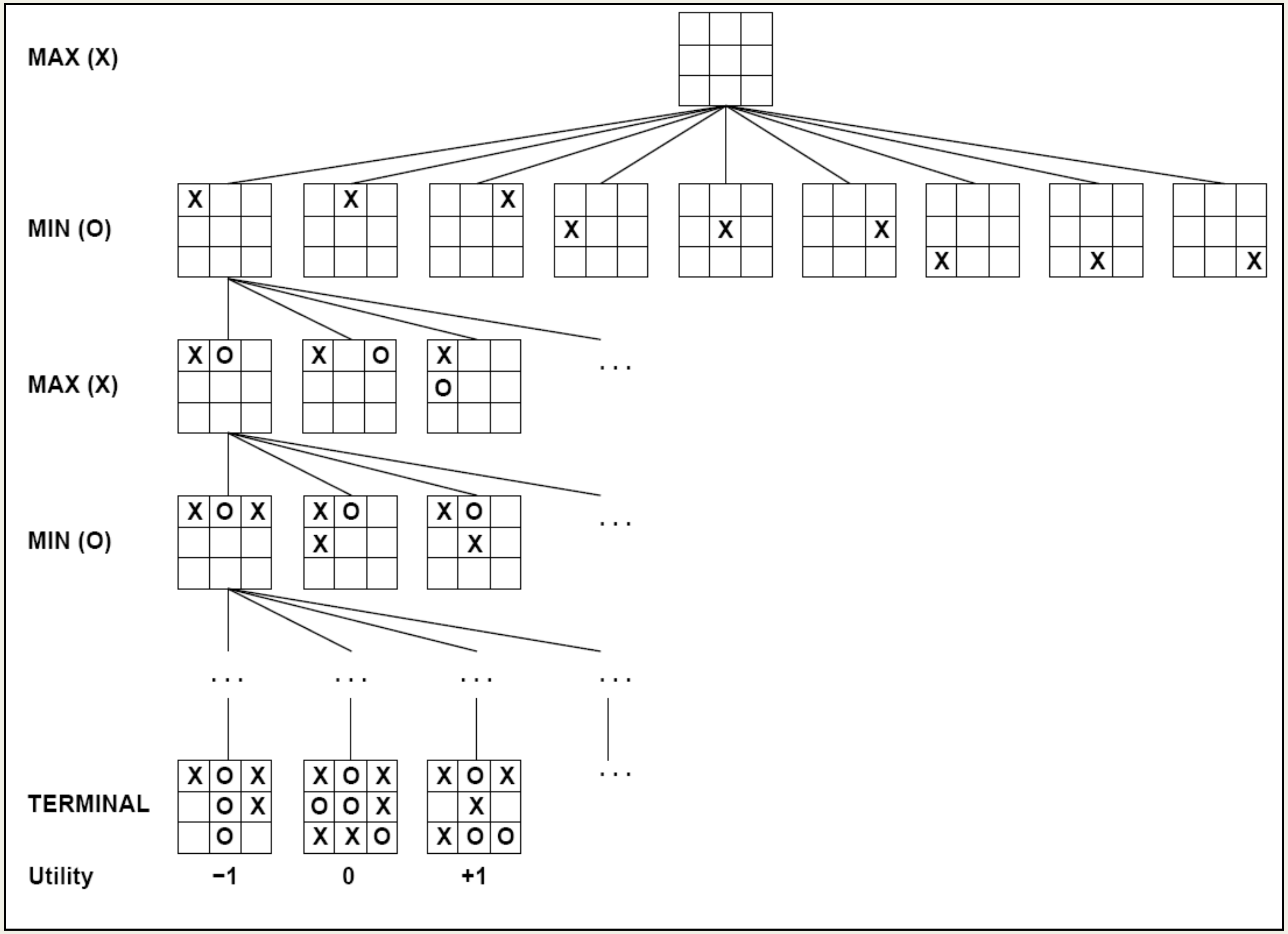
The prosecutors lack sufficient evidence to convict the pair on the principal charge. They hope to get both sentenced to a year in prison on a lesser charge.

Each prisoner may either **betray** the other by testifying that the other committed the crime, or to **cooperate** with the other by remaining silent.

What should you do?



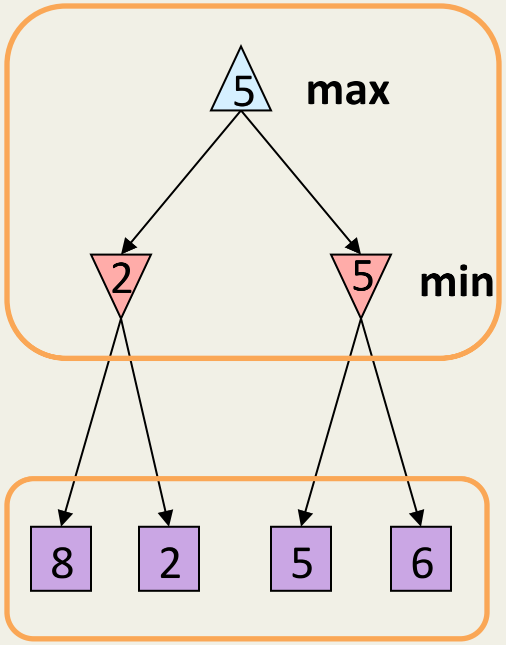
Tic-tac-toe Game Tree



Adversarial Search (Minimax)

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result
- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

Minimax values: computed recursively



Terminal values: part of the game

Minimax Implementation

```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

Minimax Implementation (Dispatch)

```
def value(state):
    if the state is a terminal state: return the state’s value
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

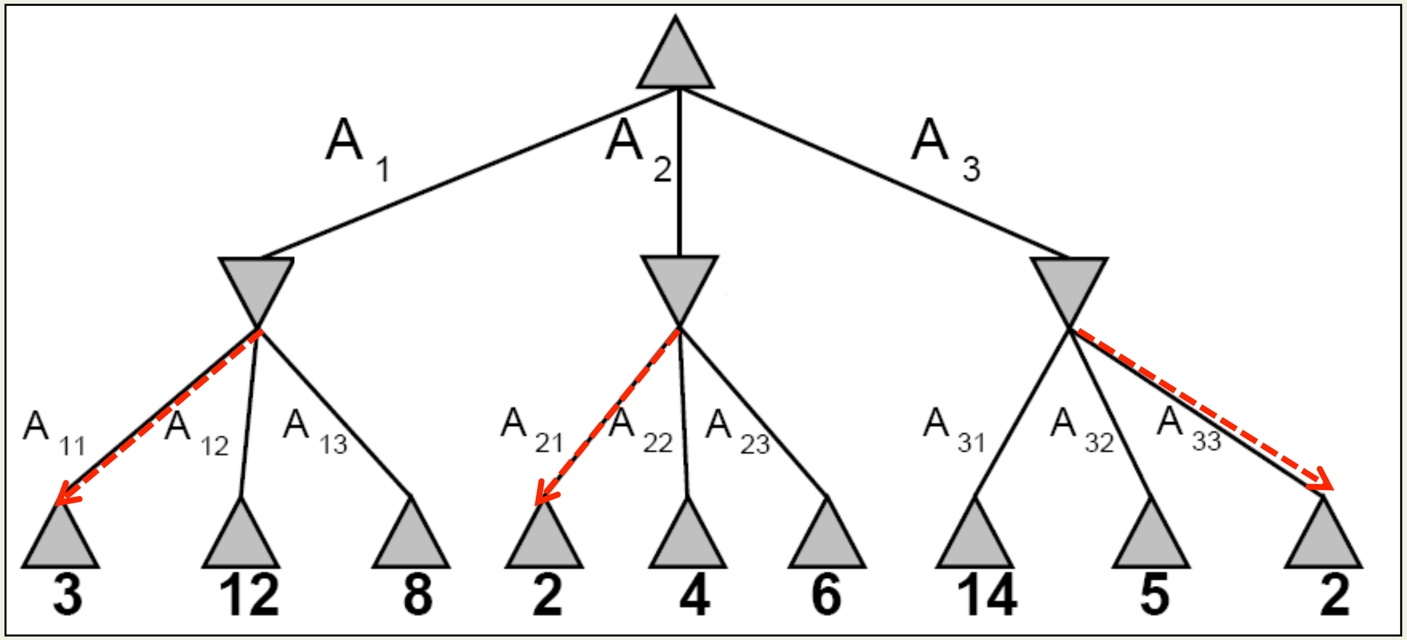
```
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v
```

```
def min-value(state):
    initialize v = +∞
    for each successor of state:
        v = min(v, max-value(successor))
    return v
```

Concrete Minimax Example

max

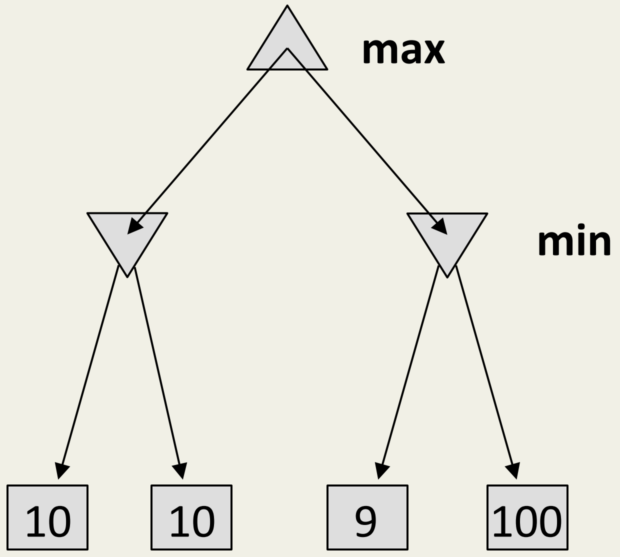
min



Minimax Properties

- Optimal?
  - Yes, against perfect player. Otherwise?
- Time complexity
  - $O(b^m)$
- Space complexity?
  - $O(bm)$

- b = branching factor
- m = moves (depth)



- For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?

Tic-Tac-Toe is easy, what about Chess?

- We know chess is NP-Hard (EXP-complete to be precise), but can we still write a good chess solver?

- Assuming 50-move-rule, games are of finite length: average game is about 40 moves, with about 30 possible moves per turn.
    - Some estimate about 10^{120} reasonable games of chess (called the Shannon Number)
    - This decision tree is **way too big** to create

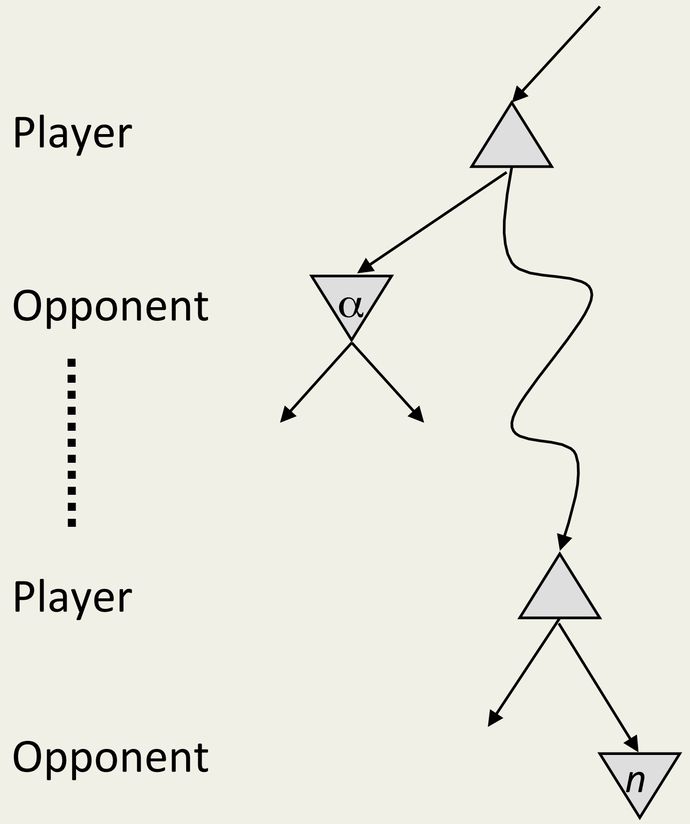
- We must use better heuristics than just decision trees to successfully play chess

A Few Optimizations

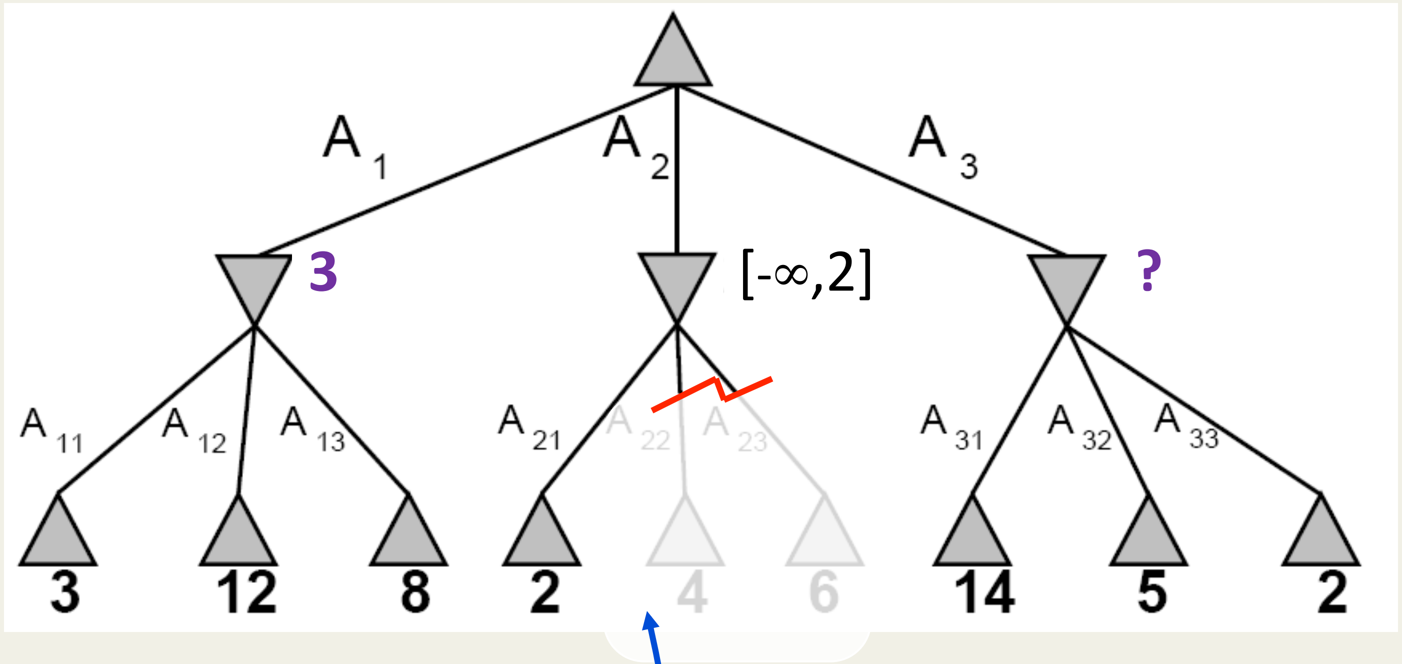
- Transposition Table
  - Store lookup table for similar looking parts of the table
- Iterative Deepening
  - Don't search entire tree all at once, incrementally increase search space
- Aspiration Windows
  - Make educated guesses about future search space
- Alpha-Beta Pruning
  - Prune branches that cannot result in the optimal solution

α-β Pruning

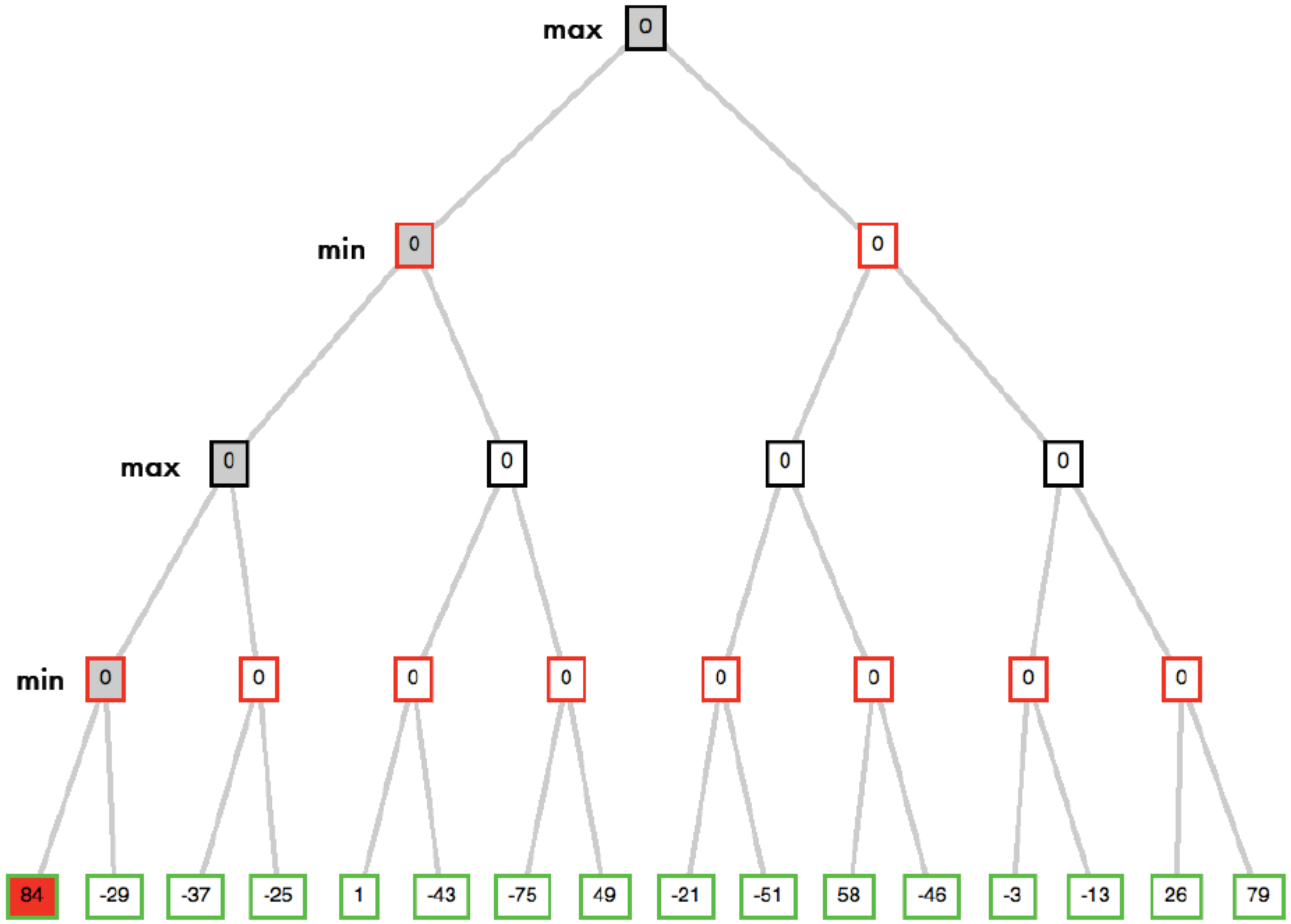
- General configuration
  - α is the best value that MAX can get at any choice point along the current path
  - If *n* becomes worse than α, MAX will avoid it, so can stop considering *n*'s other children
  - Define β similarly for MIN

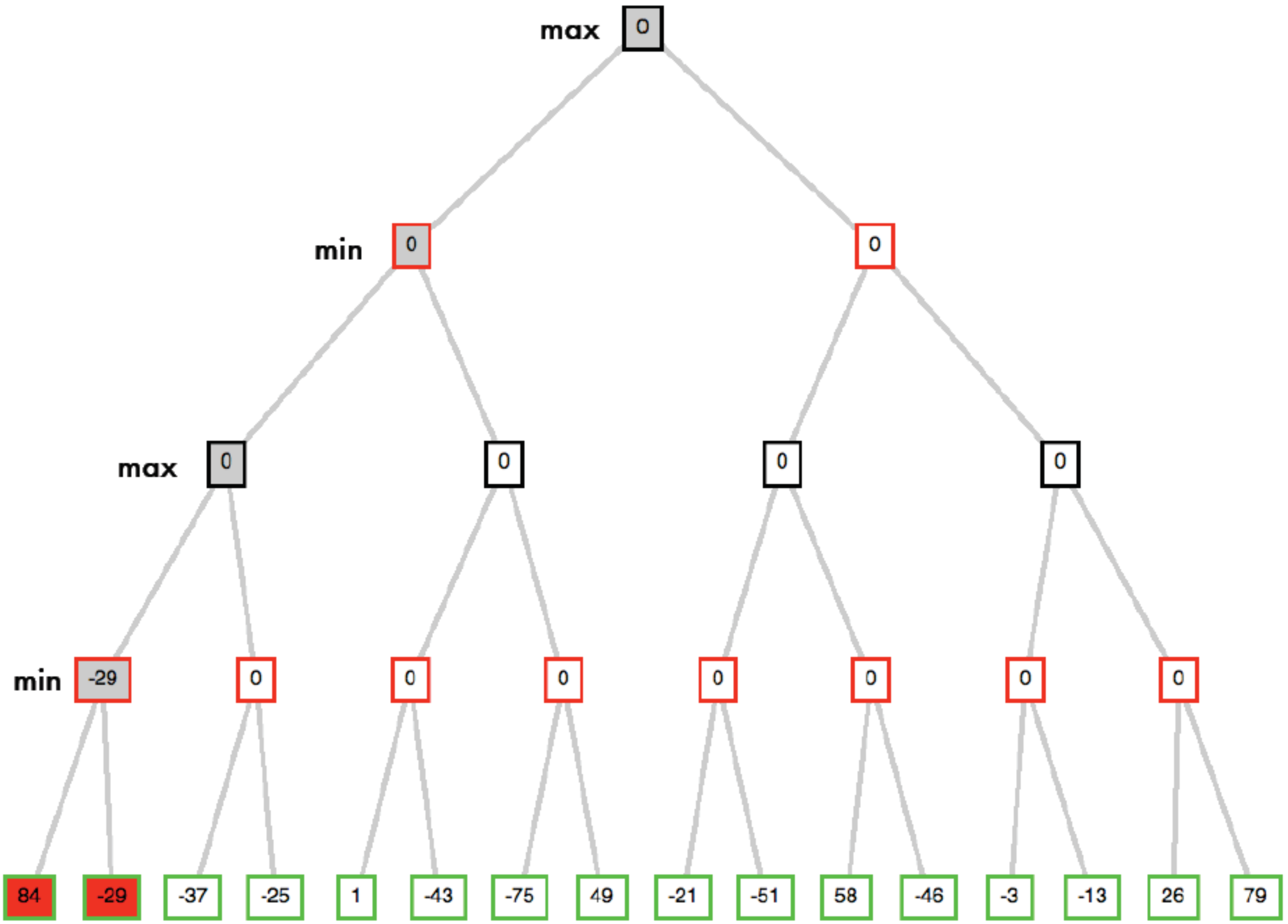


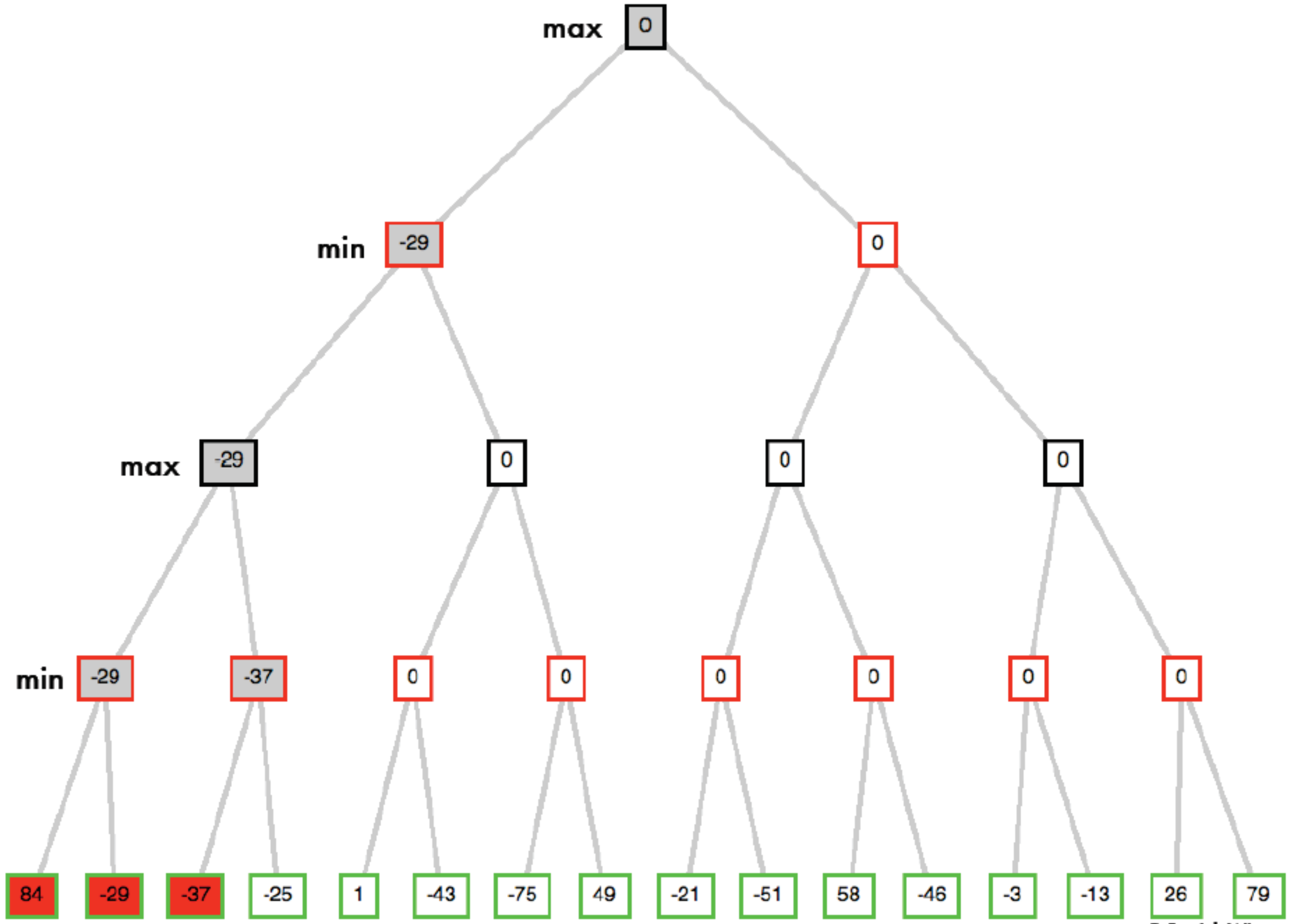
Pruning Example

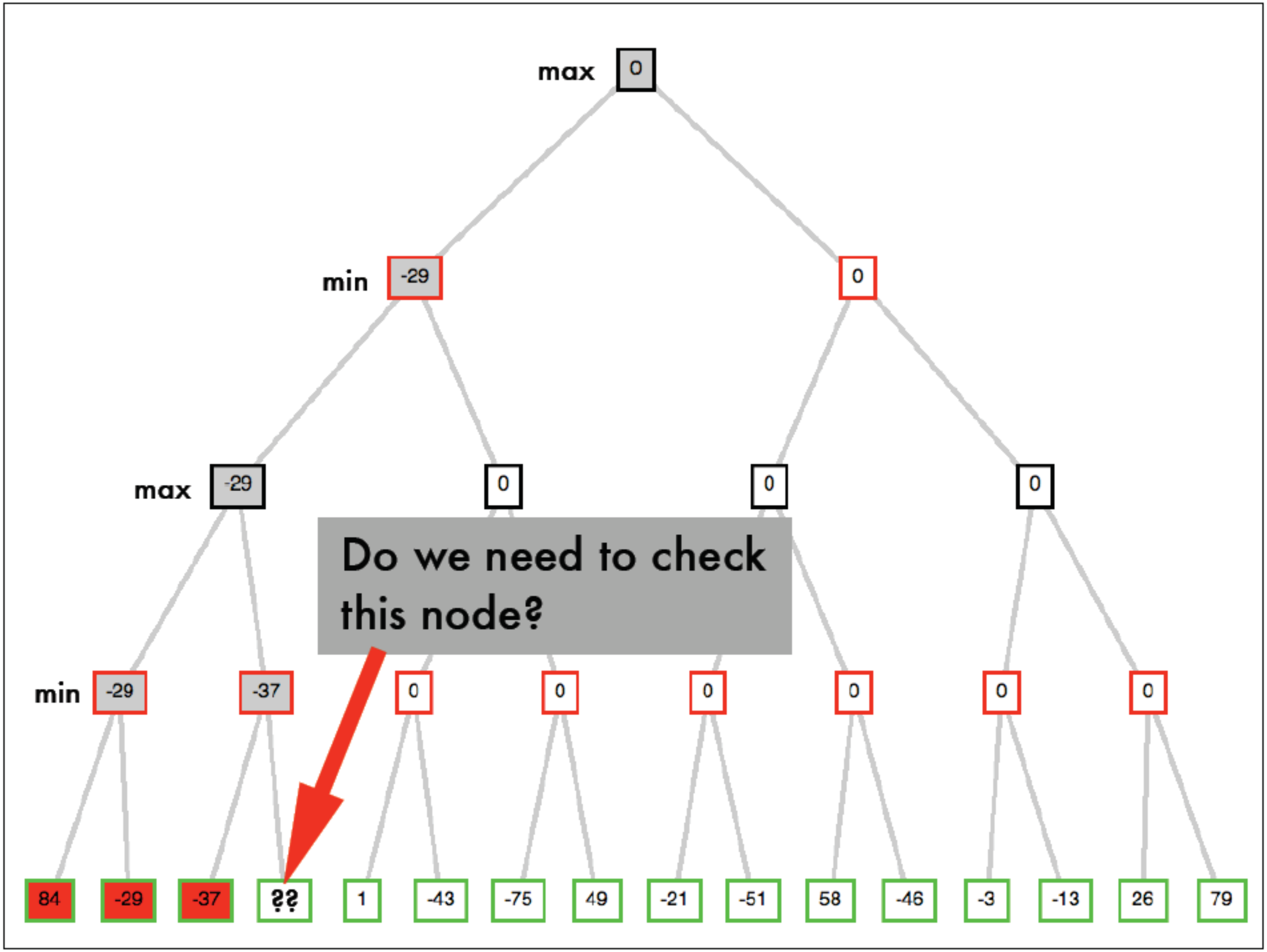


Progress of search…

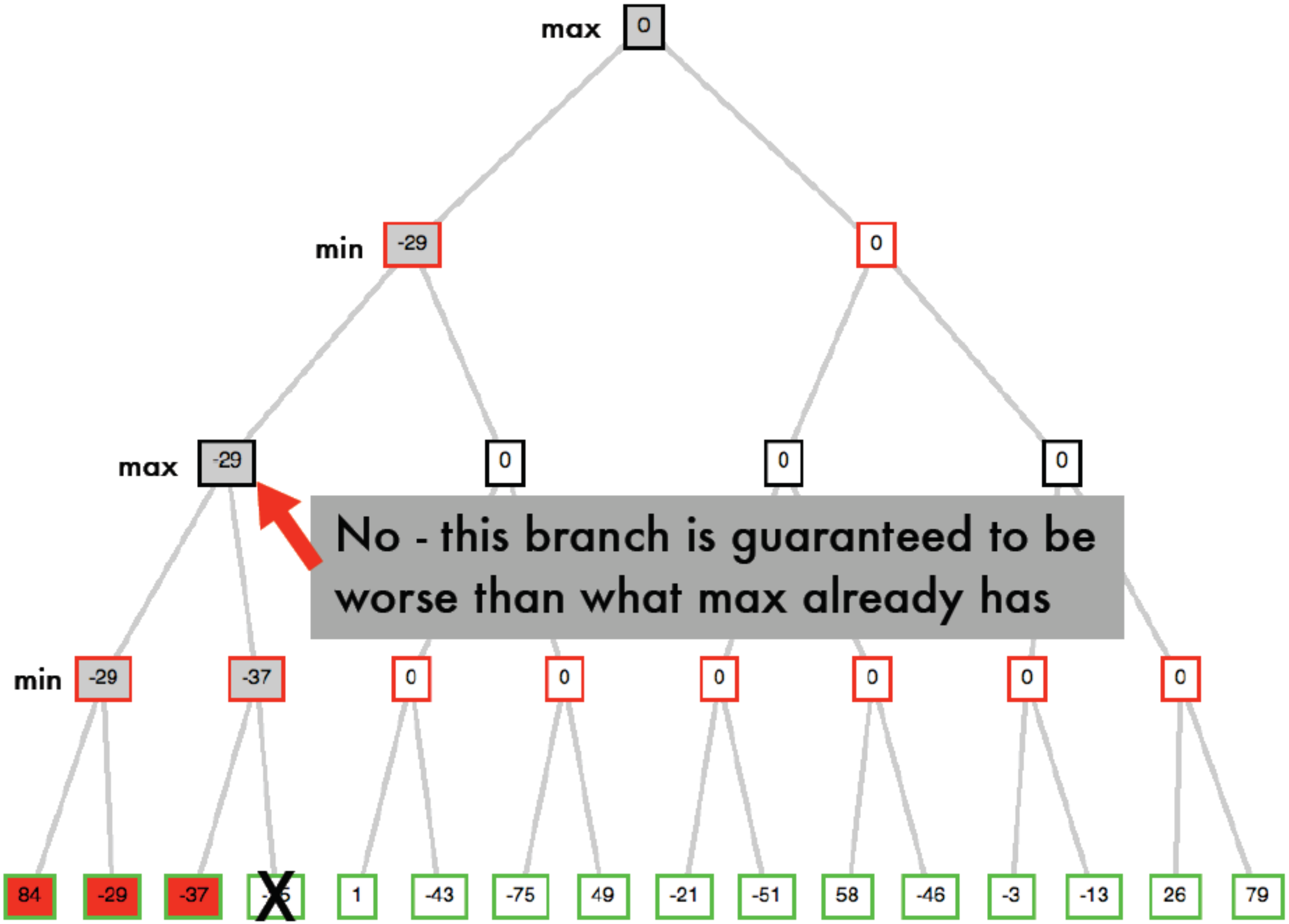

max
min
max
min
0
84
-29
-37
-25
1
-43
-75
49
-21
-51
58
-46
-3
-13
26
79


max
0
min
0
0
max
0
0
0
0
min
-29
0
0
0
0
0
0
0
84
-29
-37
-25
1
-43
-75
49
-21
-51
58
-46
-3
-13
26
79


max
0
min
-29
0
max
-29
0
0
0
min
-29
-37
0
0
0
0
0
0
84
-29
-37
-25
1
-43
-75
49
-21
-51
58
-46
-3
-13
26
79



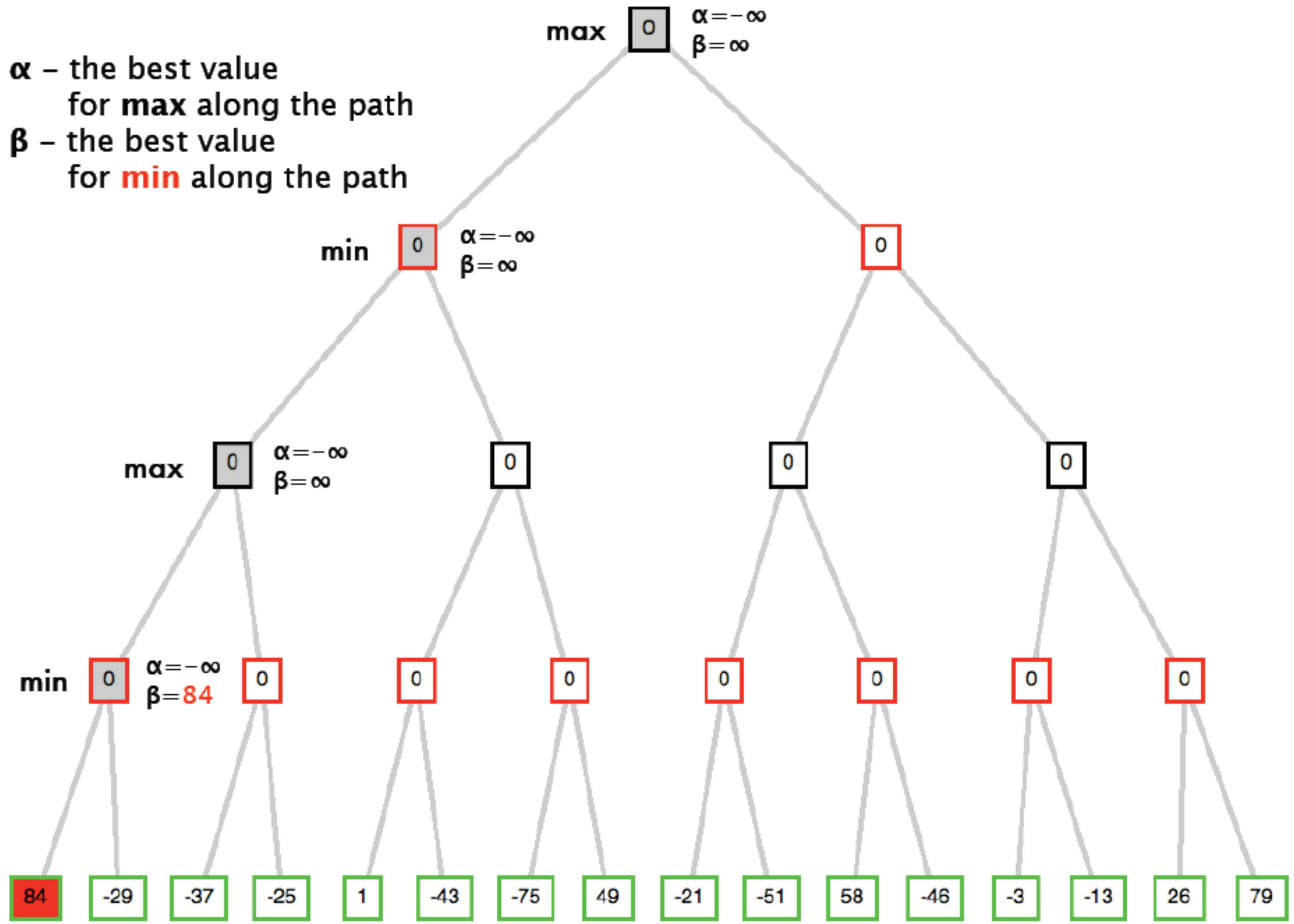
Do we need to check this node?



No - this branch is guaranteed to be worse than what max already has

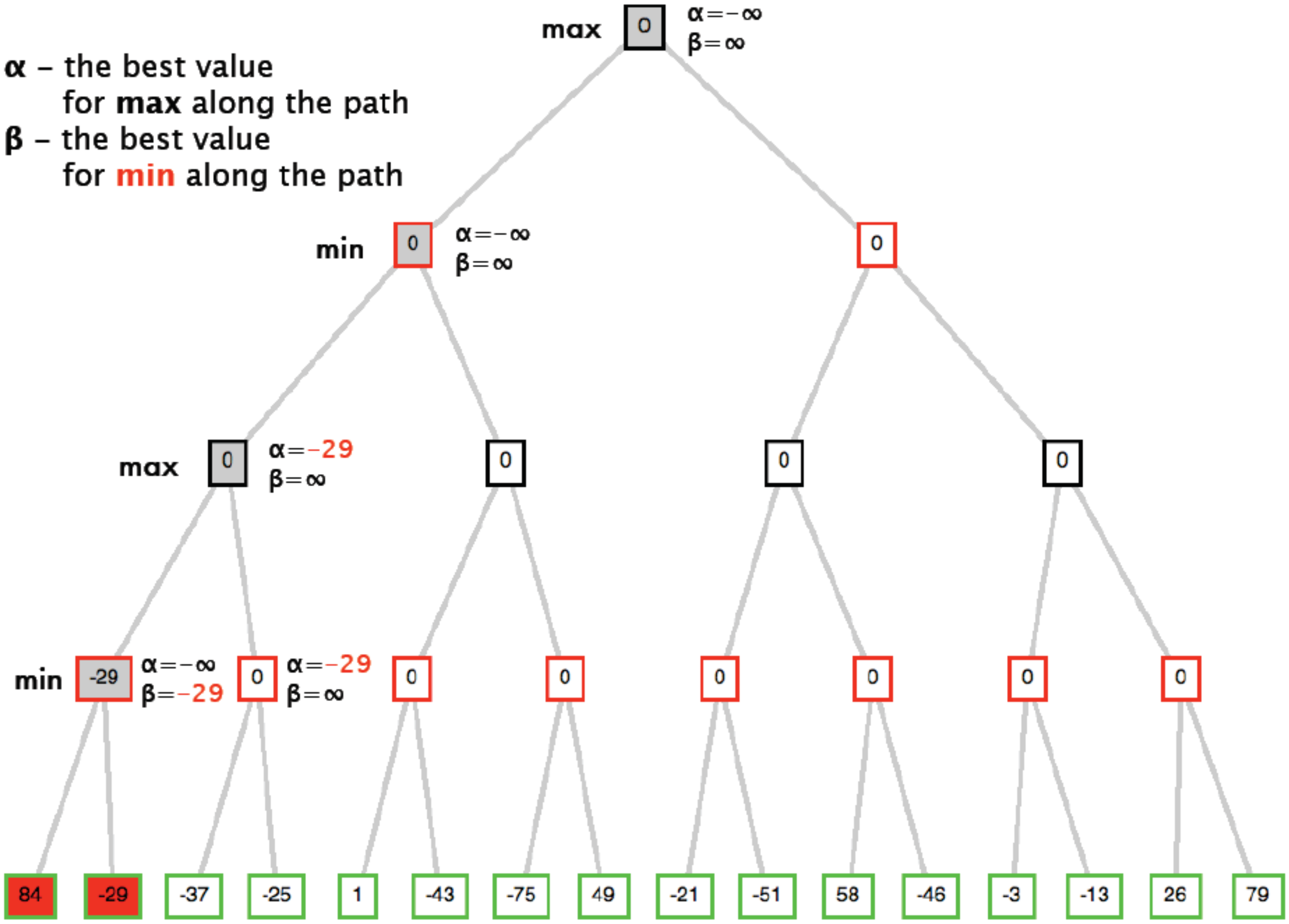
α – the best value for **max** along the path

β – the best value for **min** along the path



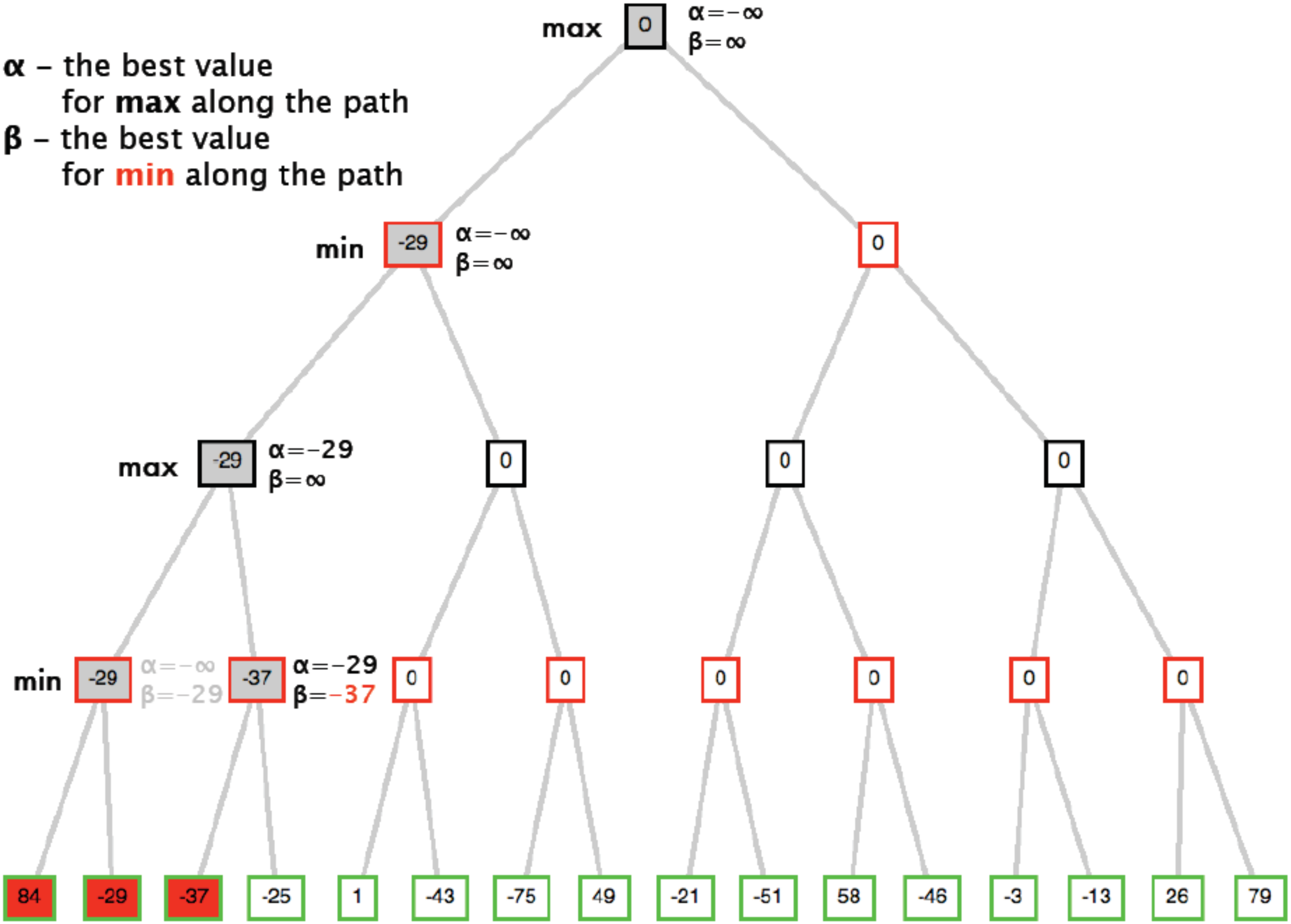
α – the best value
for **max** along the path

β – the best value
for **min** along the path

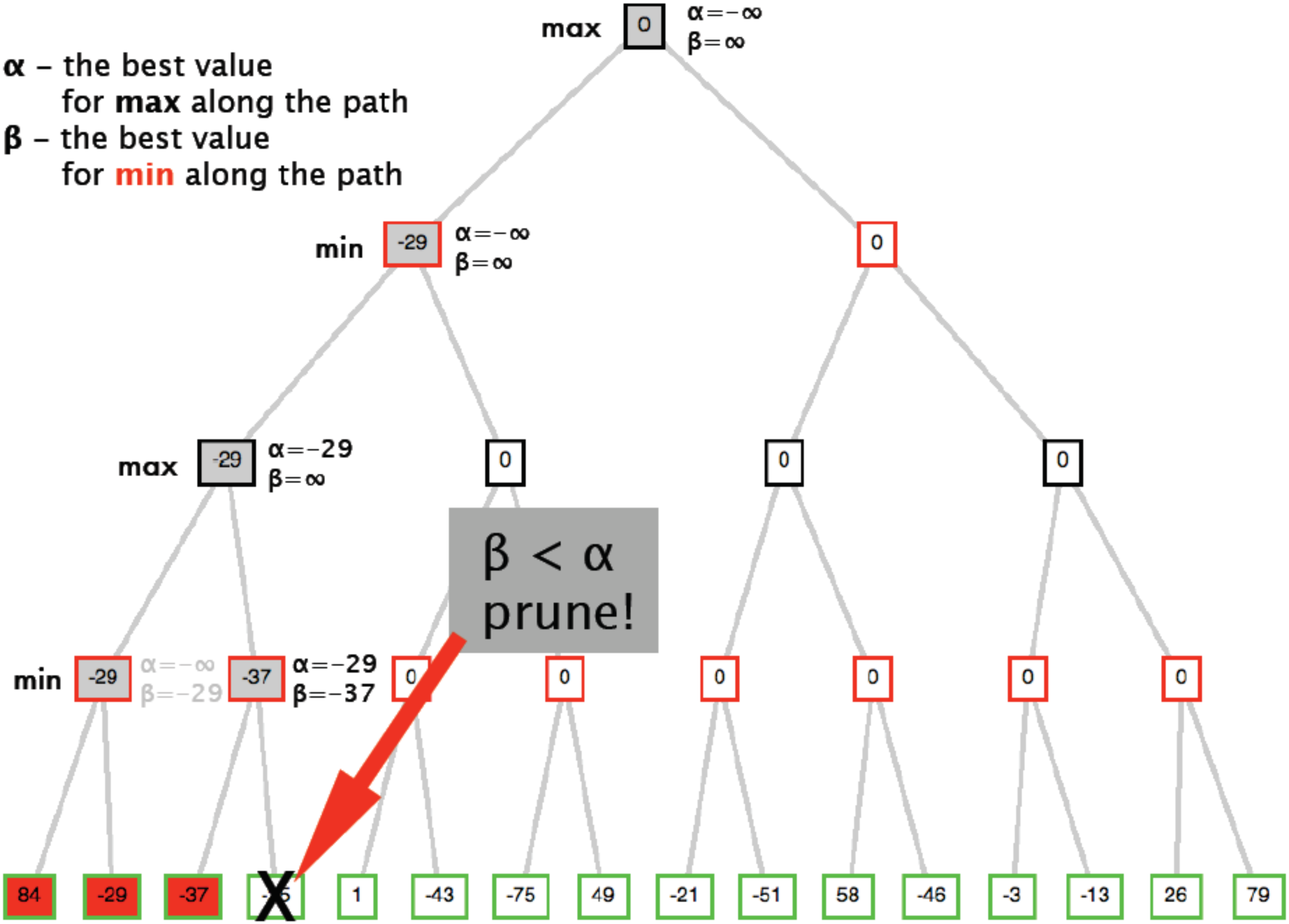


α – the best value for **max** along the path

β – the best value for **min** along the path

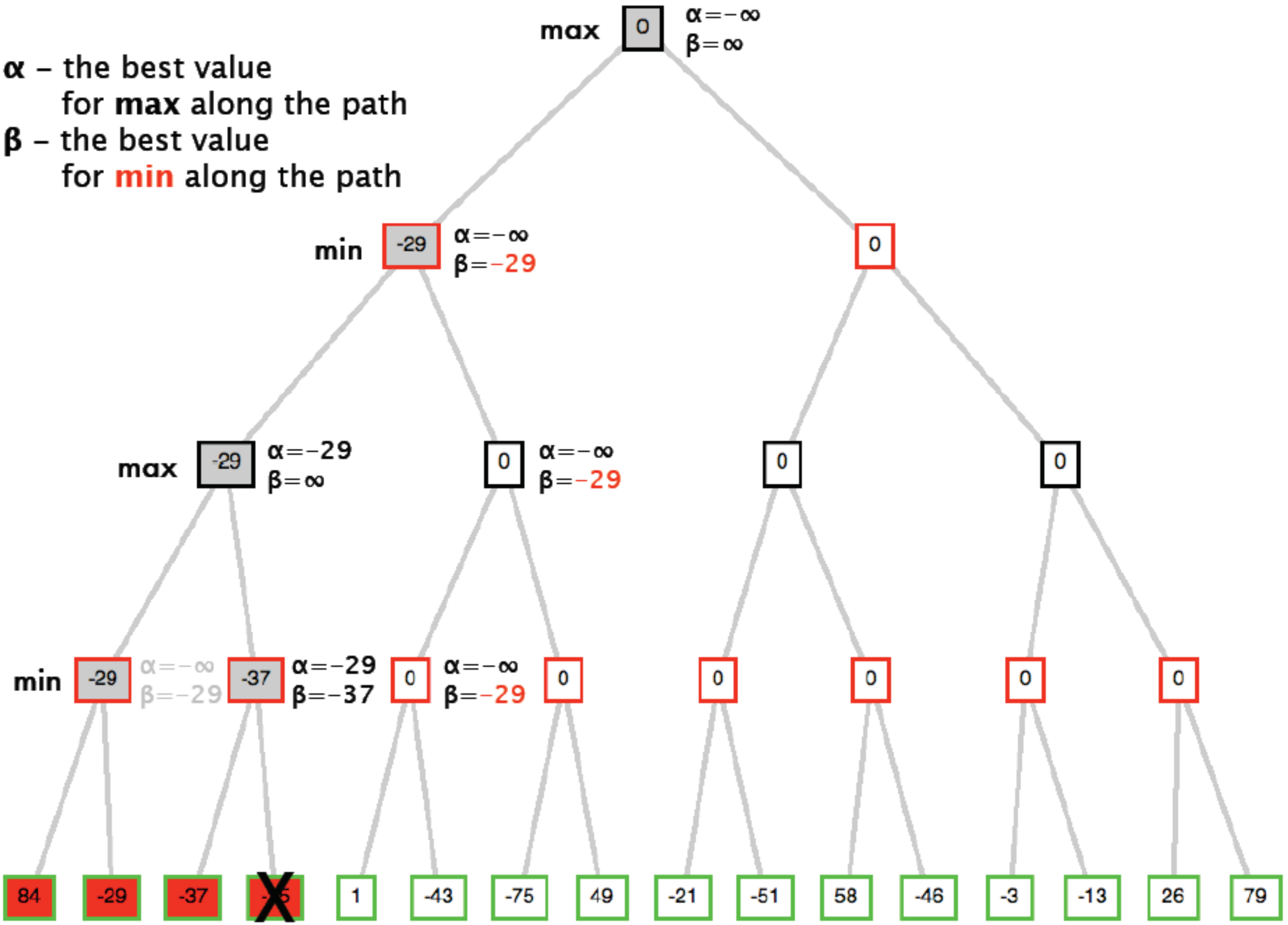


α – the best value
for **max** along the path
β – the best value
for **min** along the path



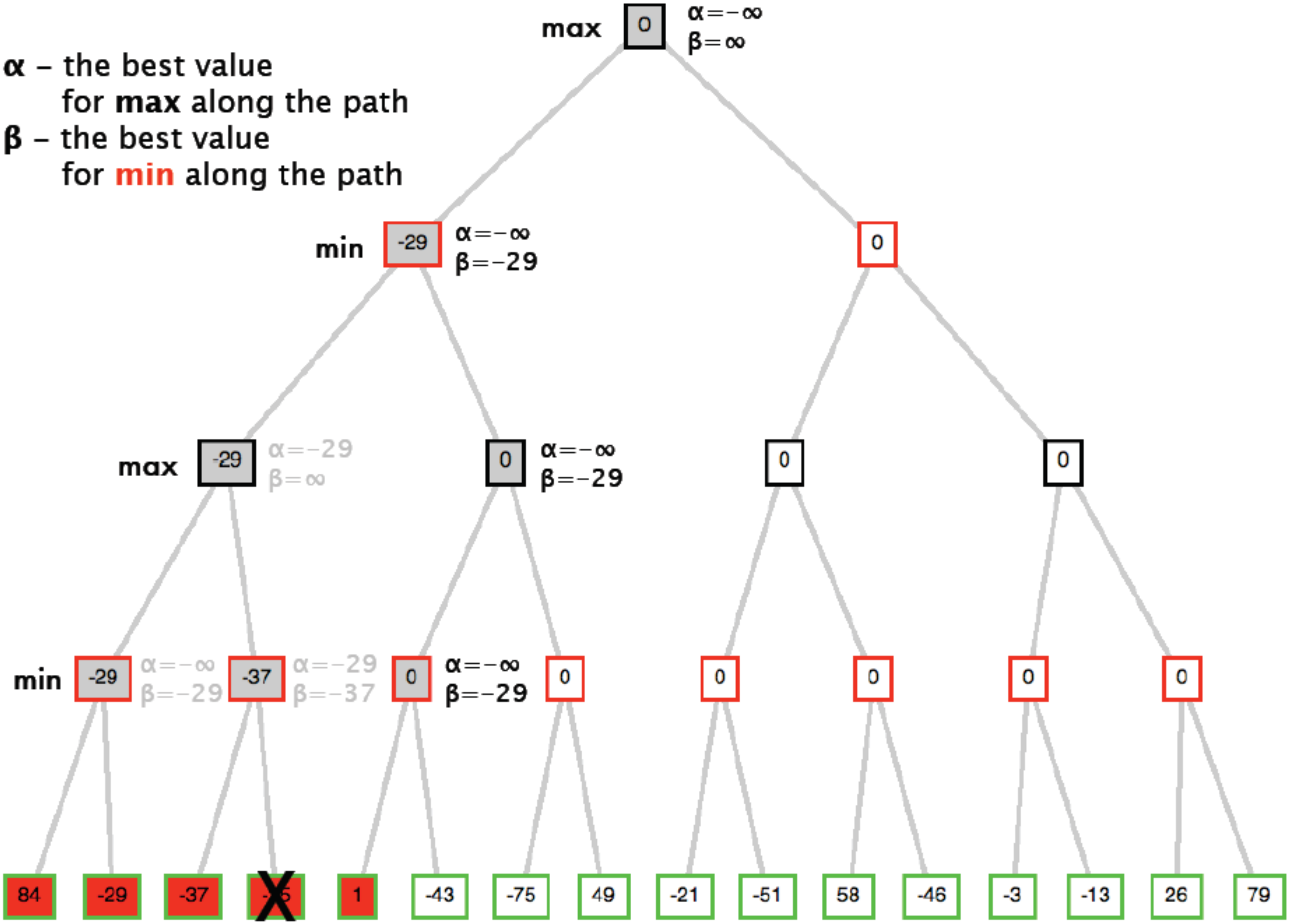
α – the best value for **max** along the path

β – the best value for **min** along the path



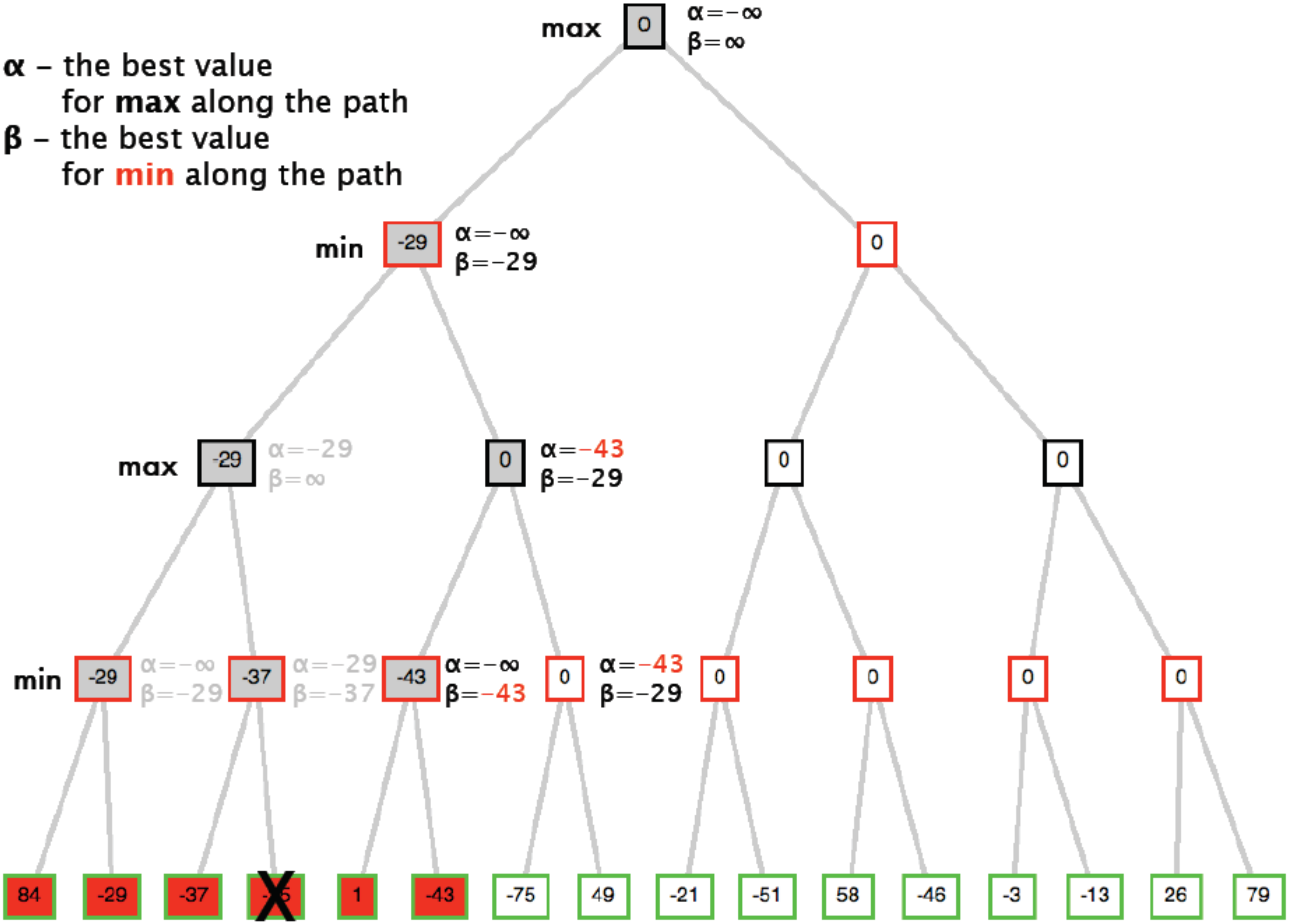
α – the best value for **max** along the path

β – the best value for **min** along the path



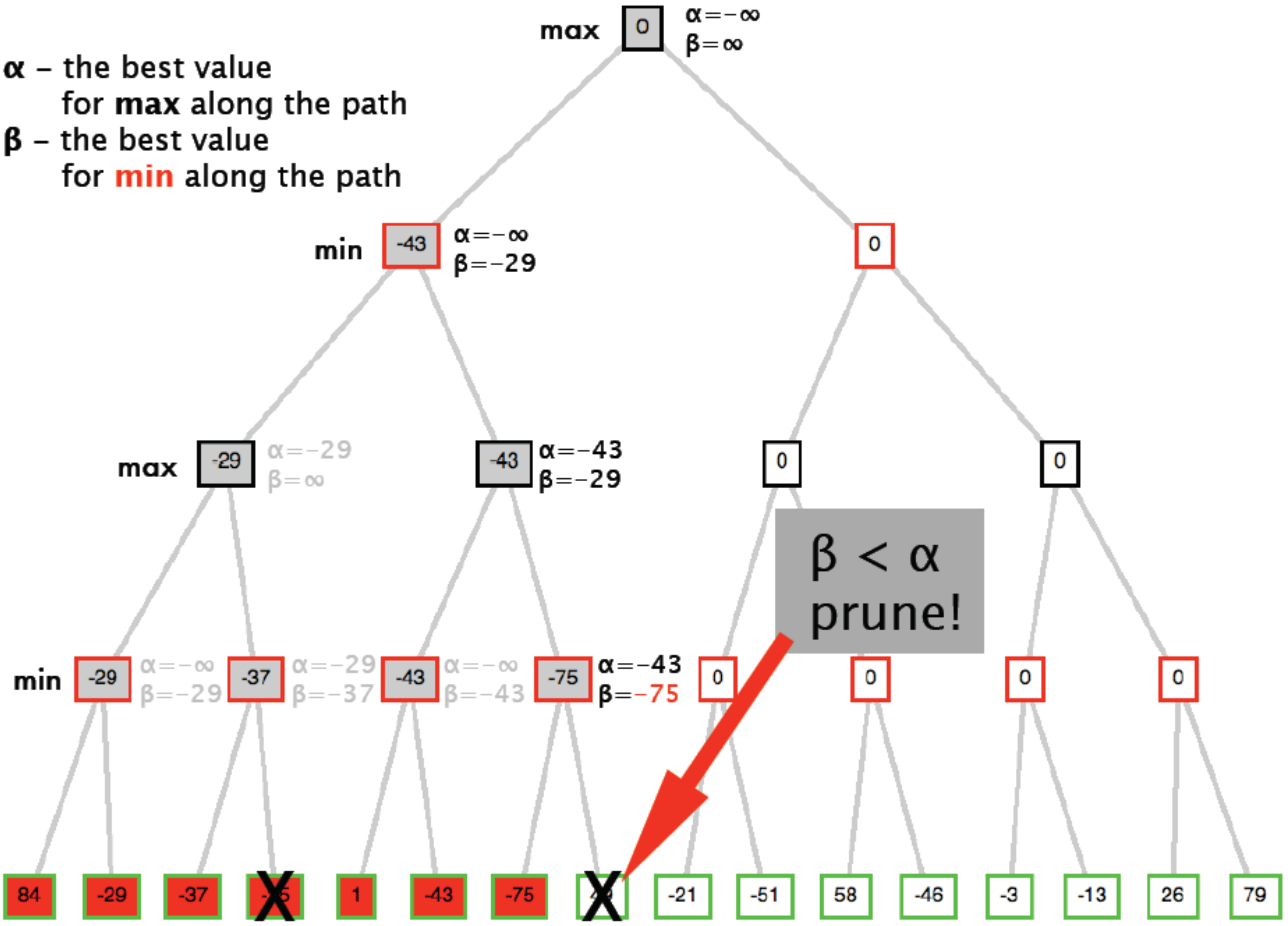
α – the best value
for **max** along the path

β – the best value
for **min** along the path



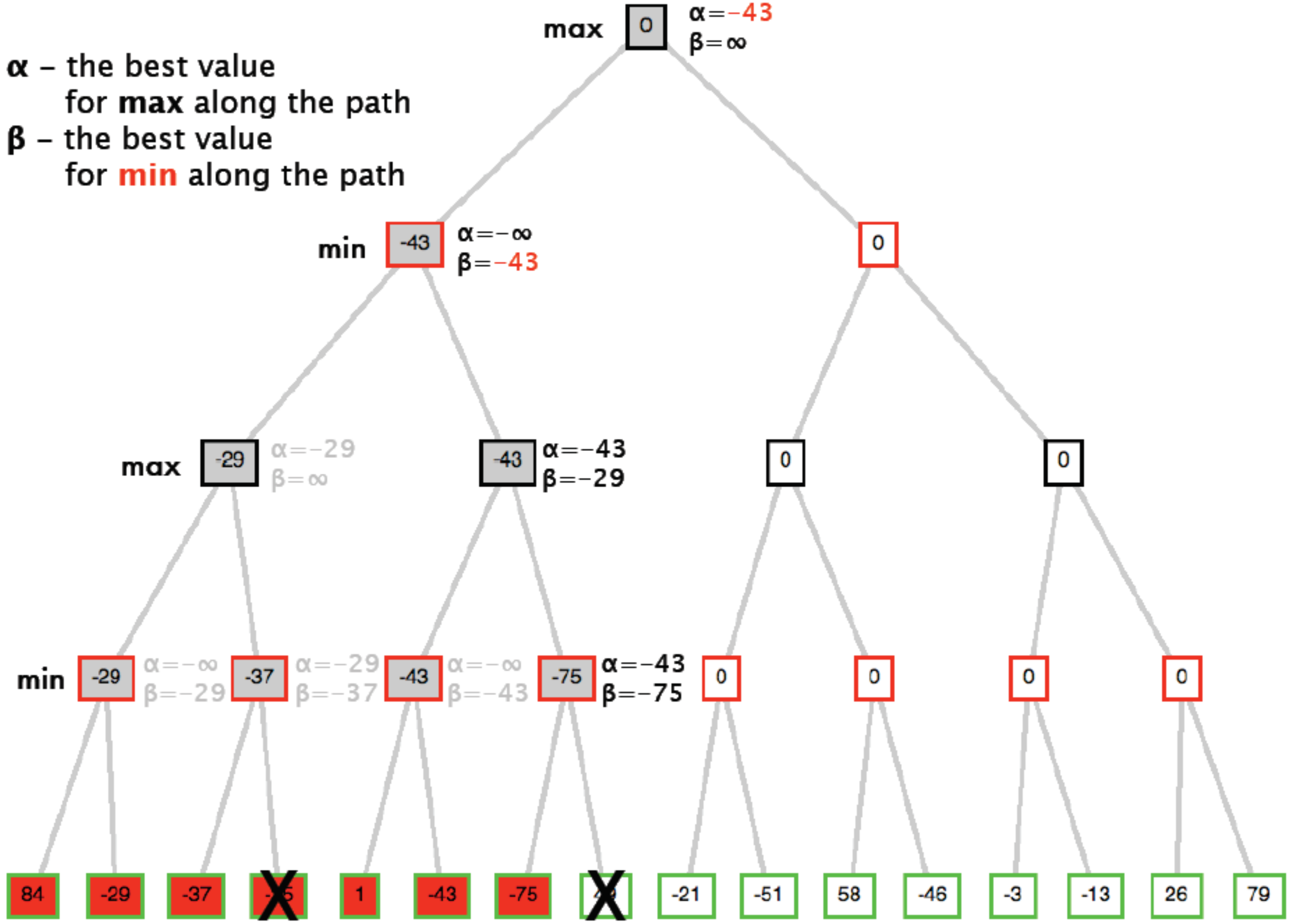
α – the best value for **max** along the path

β – the best value for **min** along the path



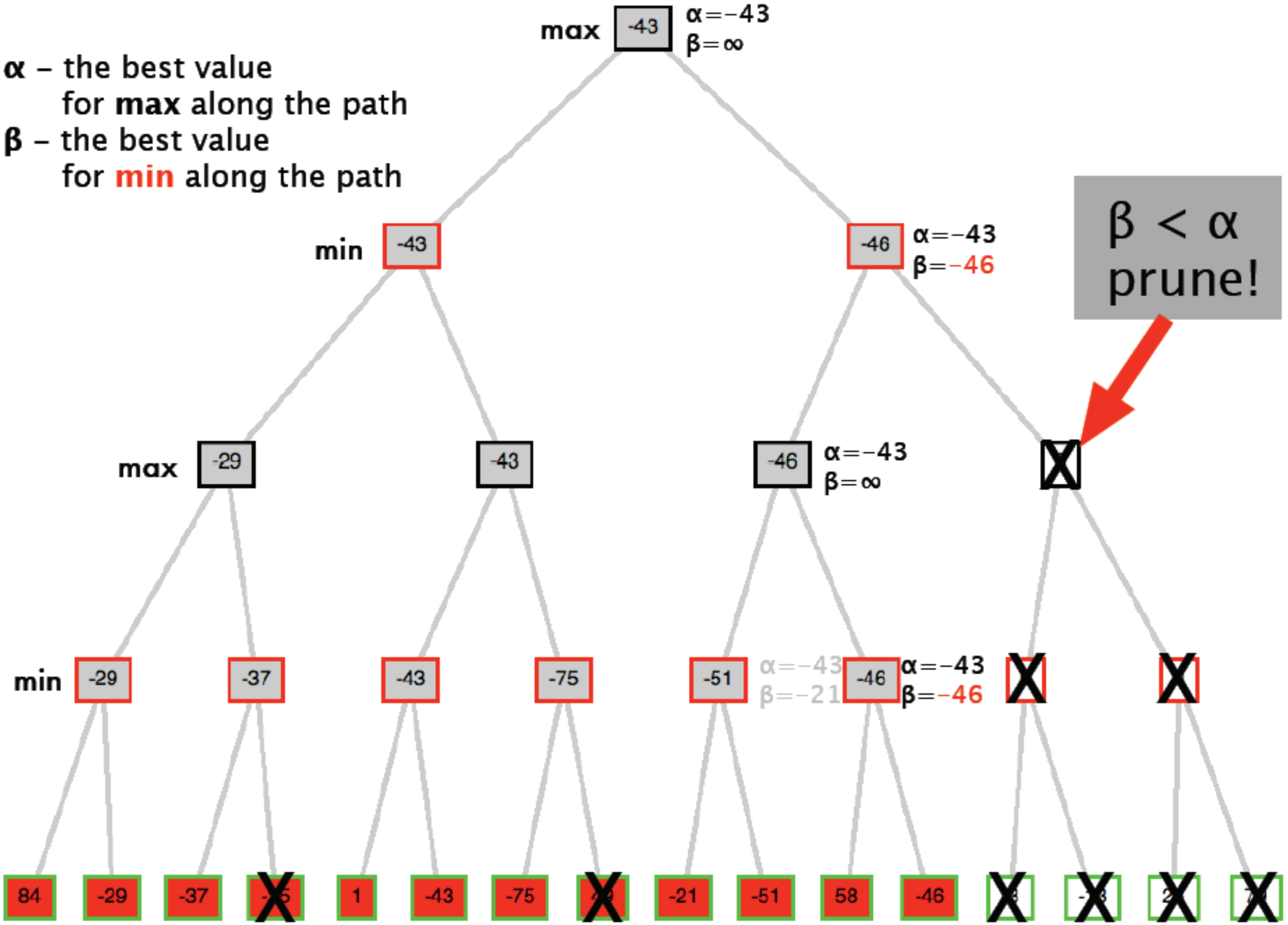
α – the best value for **max** along the path

β – the best value for **min** along the path



α – the best value
for **max** along the path

β – the best value
for **min** along the path



Alpha-Beta Pruning Properties

- This pruning has no effect on final result at the root
- Values of intermediate nodes might be wrong!
  - but, they are bounds
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…

Alpha-Beta Implementation

α: MAX's best option on path to root
β: MIN's best option on path to root

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v,
          value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v,
          value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```