Artificial Intelligence

CS4365 --- Fall 2022

Adversarial Search

Instructor: Yunhui Guo

Game Playing

An AI Favorite

- Structured task
- Not initially thought to require large amounts of knowledge
- Focus on games of perfect information

Game Playing: State-of-the-Art

- Checkers: 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

- Chess: 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply.

- Go: In 2016, AlphaGo defeats the human champion

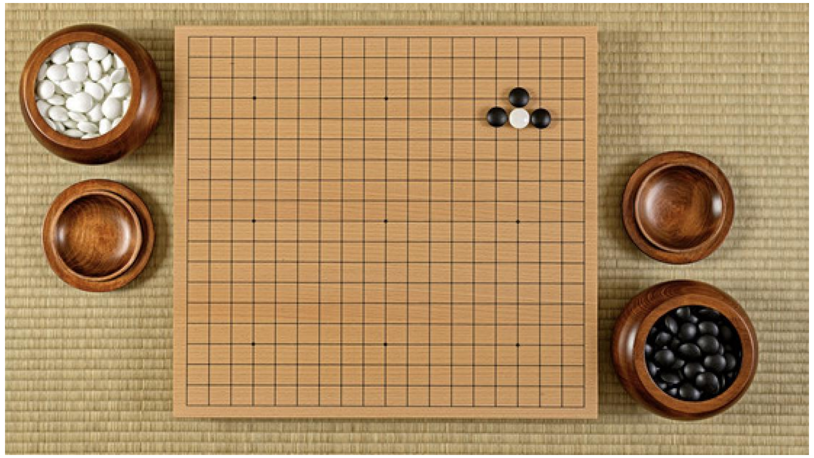
Types of Games

- Many different kinds of games!
  - Stochastic vs. Deterministic
  - One, two or more players
  - Zero sum?
  - Perfect information?

- Want algorithms for calculating a strategy which recommends a move from each state

Deterministic Games

- One possible formulation
  - States: S (start at s_0)
  - Players: P = {1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: $S \times A \rightarrow S$
  - Terminal Test: $S \rightarrow$ {True, False}
  - Terminal Utilities: $S \times P \rightarrow R$



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

Zero-Sum Games

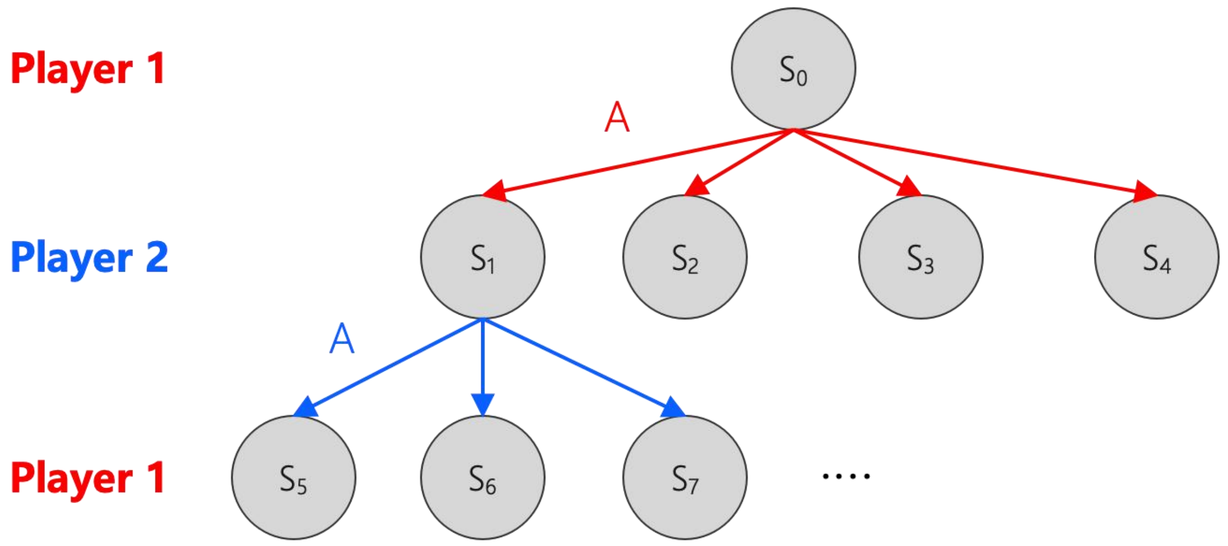
- Agents have opposite utilities (values on outcomes)
- Lets us think of a single value that one (MAX) maximizes and the other minimizes
- Adversarial, pure competition

Games VS. Search Problems

- Unpredictable opponent
  - specifying a move for every possible opponent reply
- Time limits
  - unlikely to find goal, must approximate

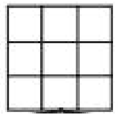
Game Playing as Search

- We can list all the possible actions and states
- In each step, play 1 searches for an action which leads to the maximum utility

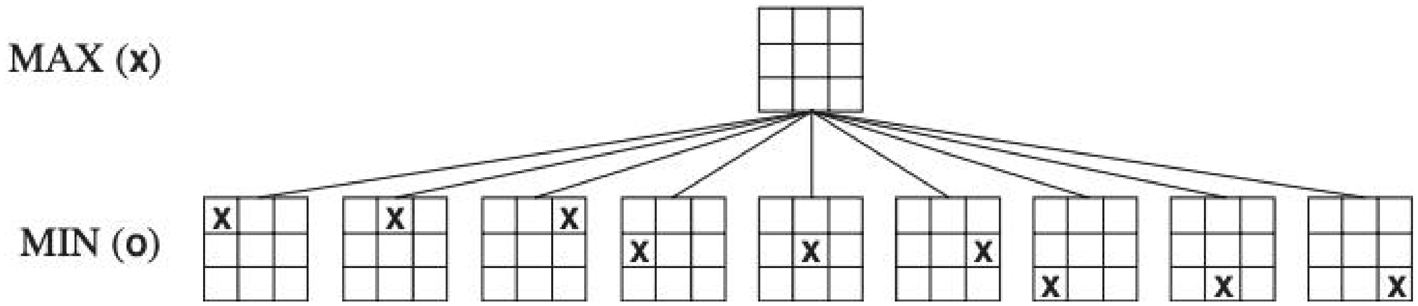


Search Tree for Tic-Tac-Toe

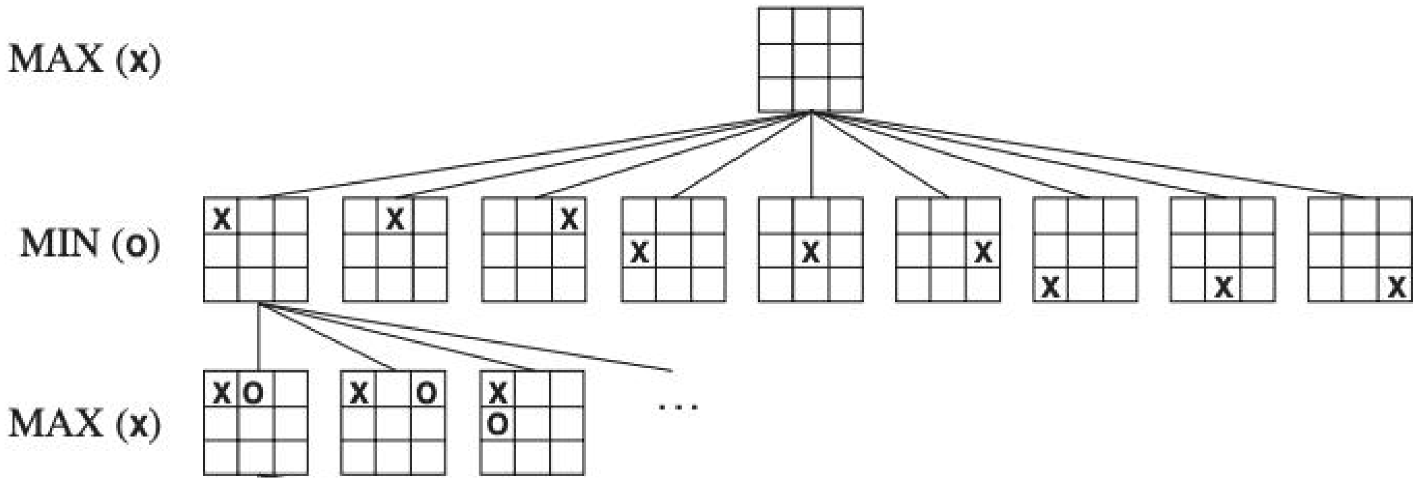
MAX (x)



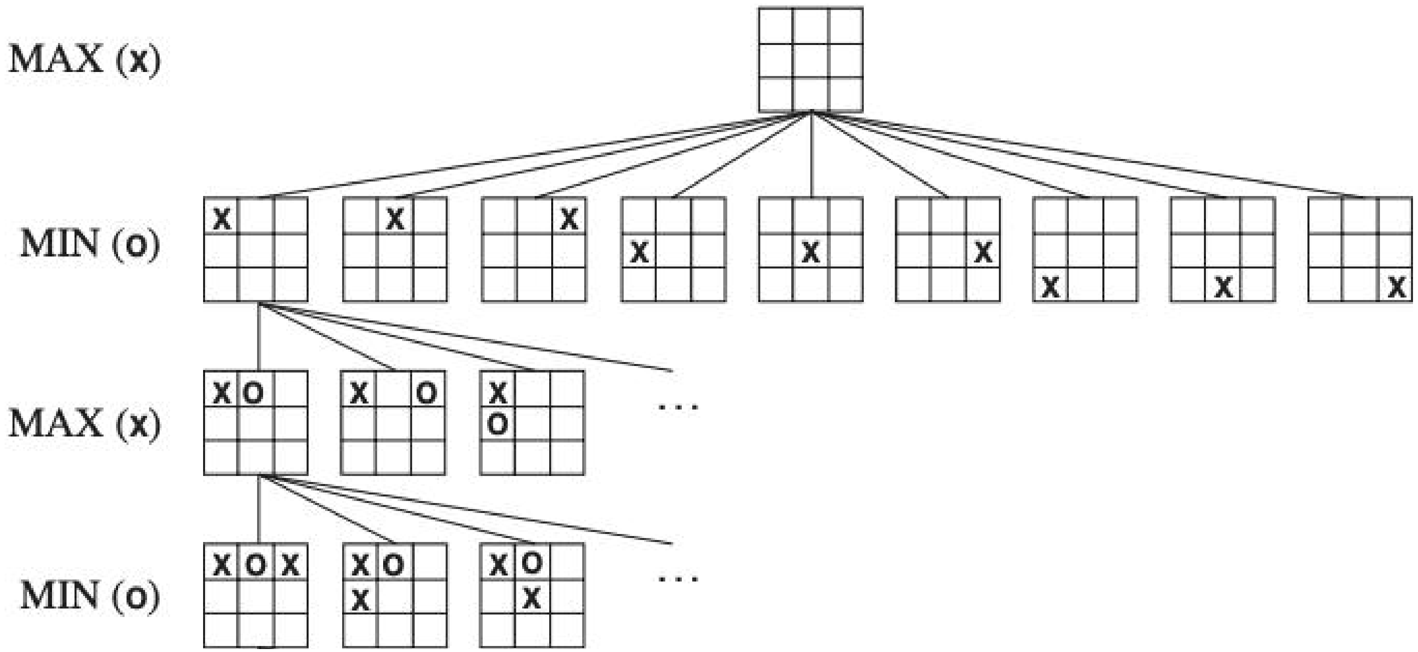
Search Tree for Tic-Tac-Toe



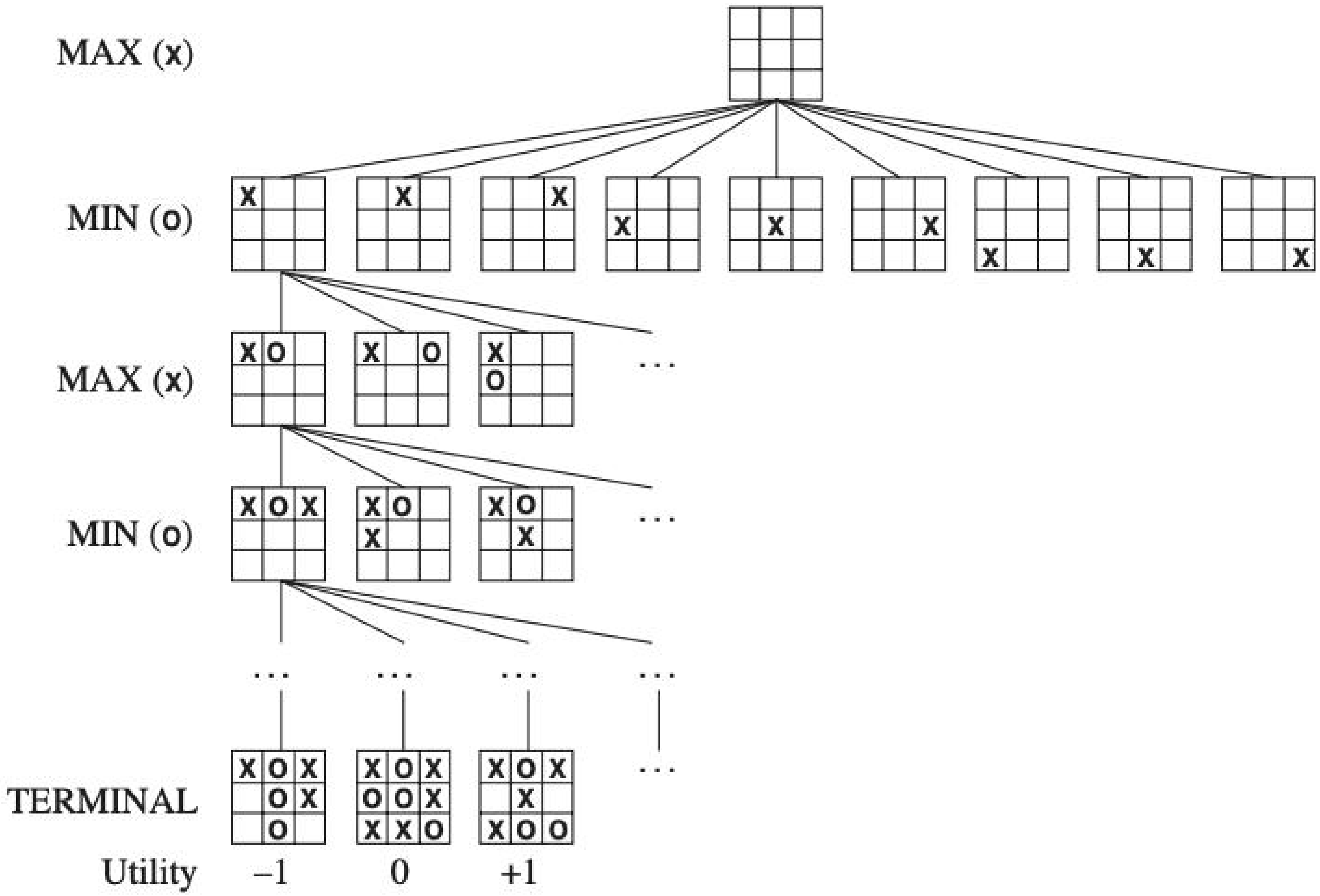
Search Tree for Tic-Tac-Toe



Search Tree for Tic-Tac-Toe



Search Tree for Tic-Tac-Toe



Tic-Tac-Toe

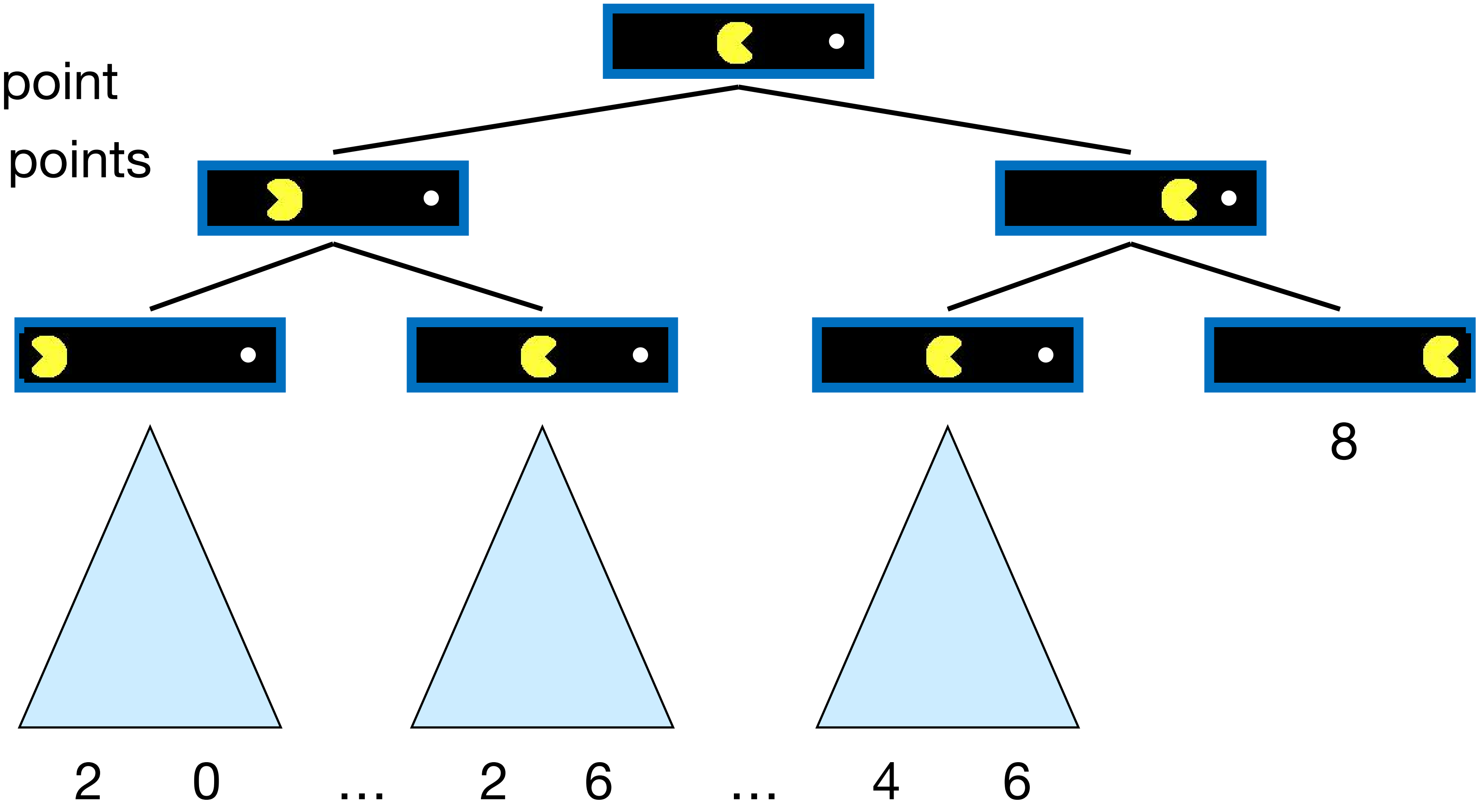
- High values are good for MAX and bad for MIN. It is MAX's job to use the search tree and utility values to determine the best move.

- Root is initial position. Next level are all moves player 1 (MAX) can make; tree is from Max's viewpoint. Next level are all possible responses from player 2 (MIN).

- Max has to find a strategy that will lead to a winning terminal state regardless of what Min does. Strategy has to include the correct move for Max for each possible move by Min.

Pac-Man Trees

- Each step costs 1 point
- The dot worths 10 points

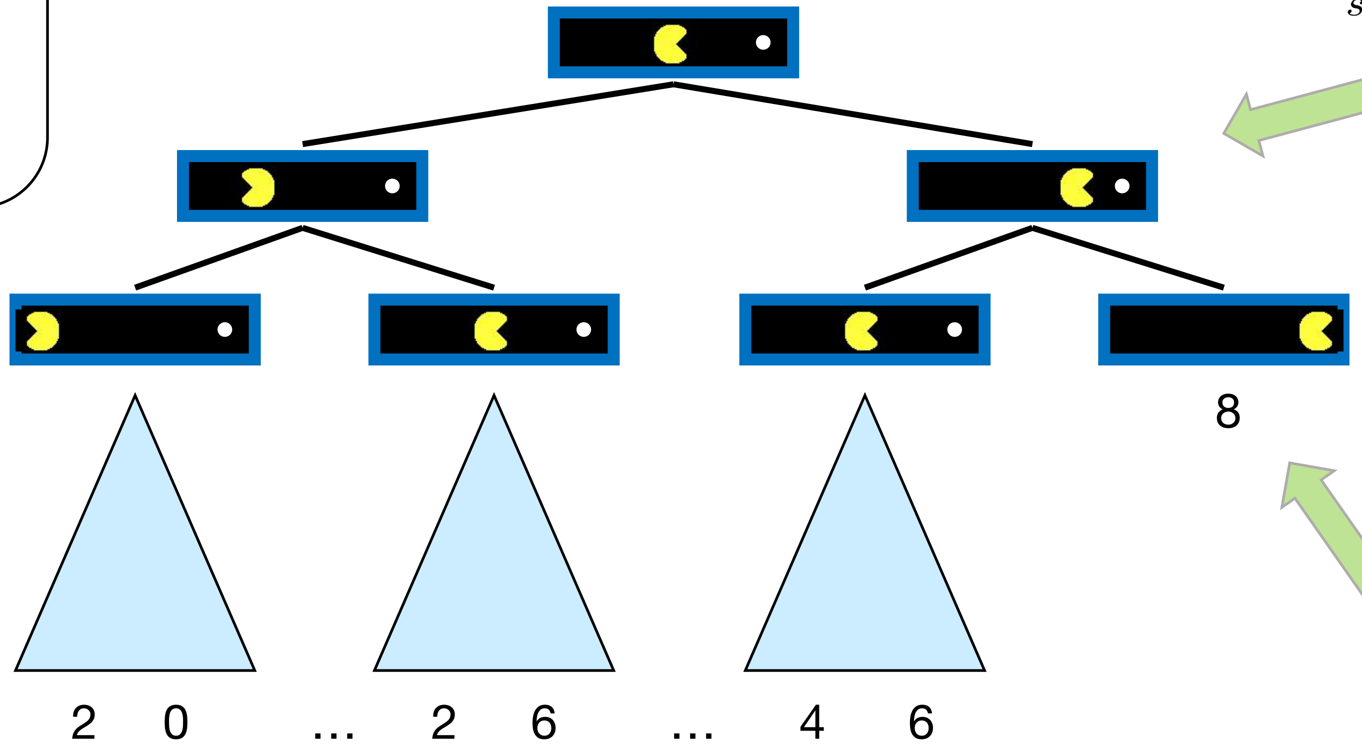


Value of a State

Value of a state: The best achievable outcome (utility) from that state

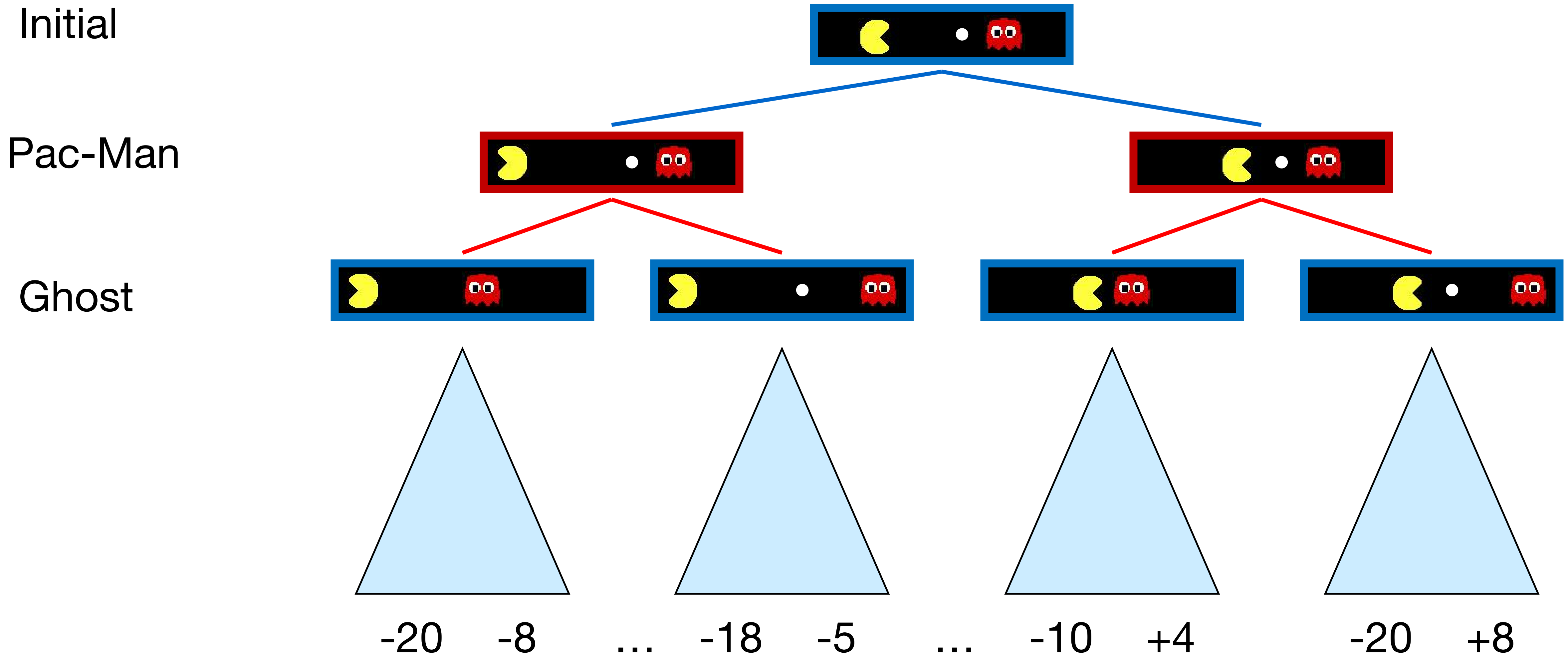
Non-Terminal States:

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



$$V(s) = \text{known}$$

Adversarial Search Tree



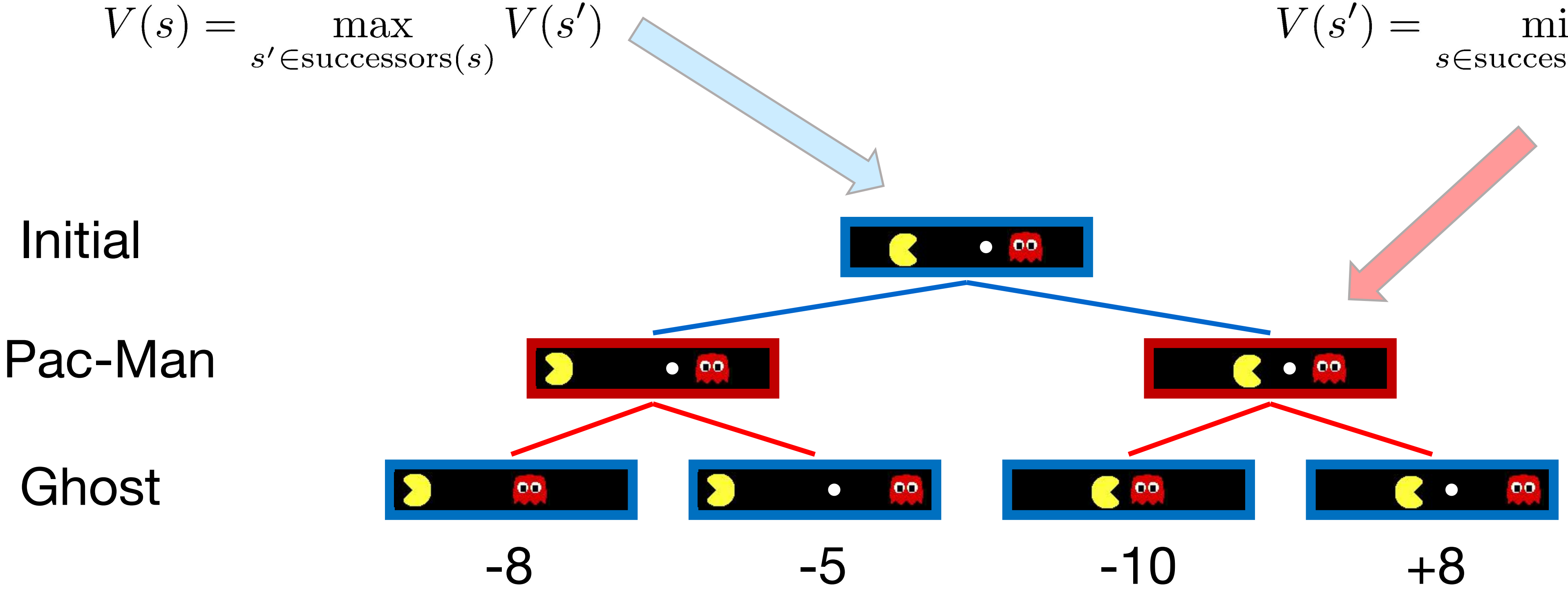
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

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



Terminal States:

$$V(s) = \text{known}$$

Minimax Search

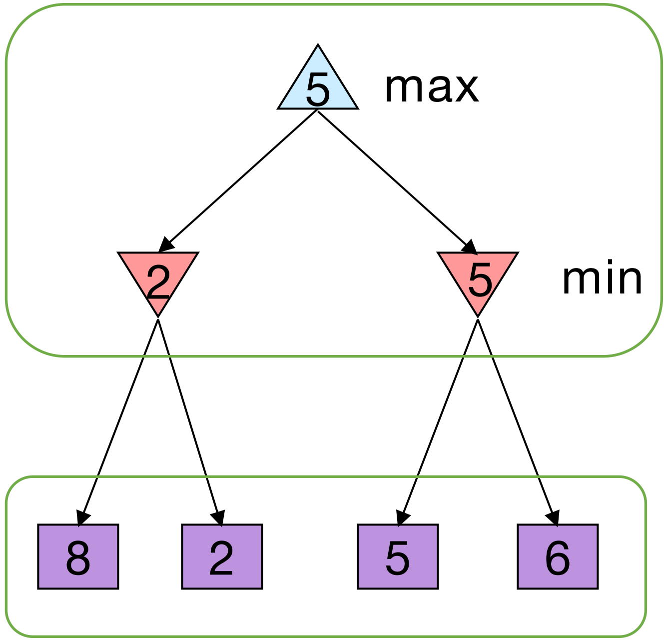
- Why do we take the min value every other level of the tree?
- These nodes represent the opponent’s choice of move.
- The computer assumes that the human will choose that move that is of least value to the computer.

Minimax Search

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess, checkers
  - One player maximizes result
  - The other minimizes result

- 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

Simplifed Minimax Algorithm

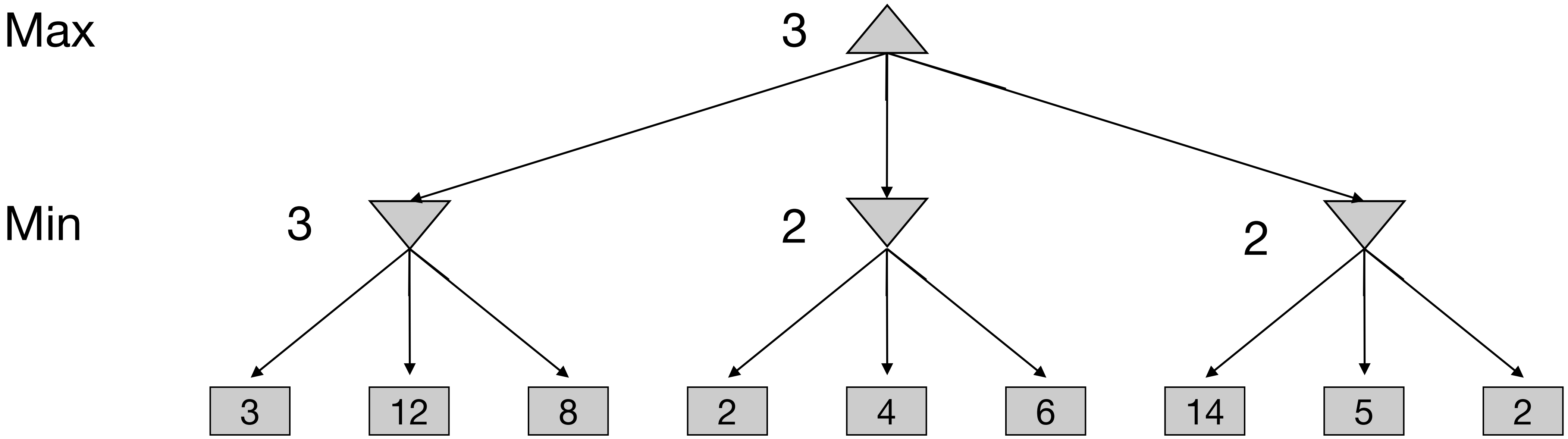
1. Expand the entire tree below the root

2. Evaluate the terminal nodes as wins for the minimizer or maximimizer

3. Select an unlabeled node, n, all of whose children have been assigned values. If there is no such node, we’re done — return the value assigned to the root.

4. If n is a minimizer move, assign it a value that is the minimum of the values of its children. If n is a maximizer move, assign it a value that is the maximum of the values of its children. Return to Step 3.

Another Example



Summary

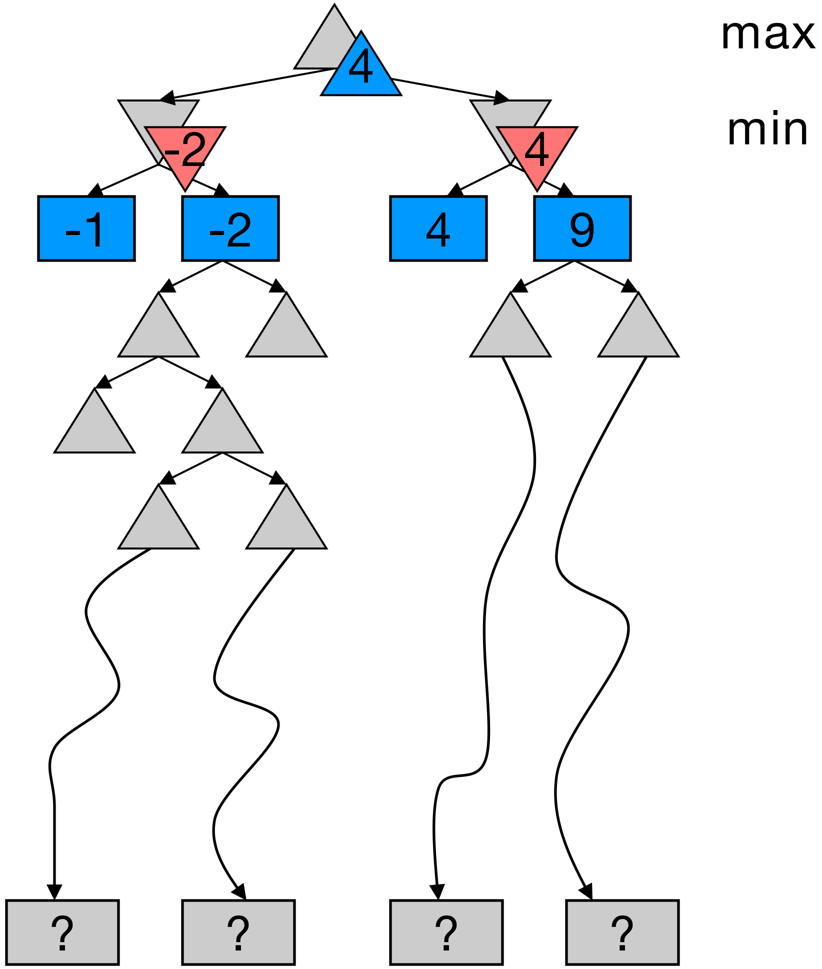
- In game tree search, a move is a pair of actions. one player's action is a ply. 2-ply = one move.
- Called a minimax decision because it maxmizes the utility under the assumption that the opponent will play perfectly to minimize it.
- Time complexity:
  - $O(b^m)$ (m plies and b branching.) Impractical for e.g. chess ($b \approx 30$ to 40). 1000^k for k moves.
- Space complexity: $O(bm)$

Size of Search Space

- Chess:
  - branching factor $b \approx 35$
  - game length $m \approx 100$
  - search space $b^m = 35^{100} \approx 10^{154}$
- The Universe
  - number of atoms $\approx 10^{78}$
- Exact search is infeasible

The Need for Imperfect Decisions

- Problem:
  - Minimax assumes the program has time to search to the terminal nodes.
- In realistic games, cannot search to leaves!
- Solution: Cut off search earlier and apply a heuristic evaluation function to the leaves
- Guarantee of optimal play is gone



Static Evaluation Functions

- Minimax depends on the translation of board quality into a single, summarizing number. Difficult. Expensive

- Evaluation functions score non-terminals in depth-limited search
  - Do you control the center of the board?
  - How well protected is your king?
  - Add up values of pieces each player has (weighted by importance of piece).
  - Mobility

- Strategies change as the game proceeds.

Design Issues for Heuristic Minimax

Evaluation Function:

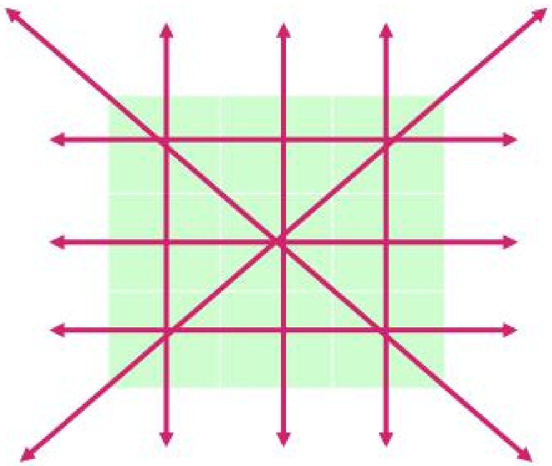
What features should we evaludate and how should we use them?

An evaluation function should:

1. Match utility function on terminal states
2. Not take too long
3. Accurately reflect the chance of winning

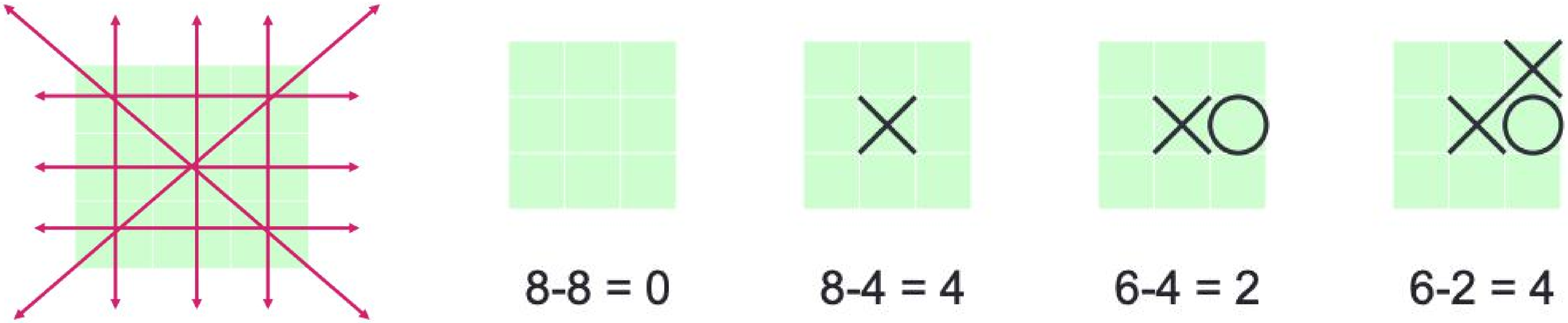
Evaluation Functions for Tic-Tac-Toe

- Let p be a positon in the game
- Define the utility function f(p) by
  - count the number of lines where X can win
  - subtract number of lines where O can win



Evaluation Functions for Tic-Tac-Toe

- f(p) = the number of lines where X can win - the number of lines where O can win



Linear Evaluation Functions

- Let f_i be features and w_i be weights
- Linear evaluation function: $w_1f_1 + w_2f_2 + \ldots + w_nf_n$

  This is what most game playing programs use
- For example: $f = 6 \cdot \text{material} + 4 \cdot \text{mobility} + \text{center control}$

Linear Evaluation Functions

- Let f_i be features and w_i be weights

- Linear evaluation function: $w_1f_1 + w_2f_2 + \ldots + w_nf_n$

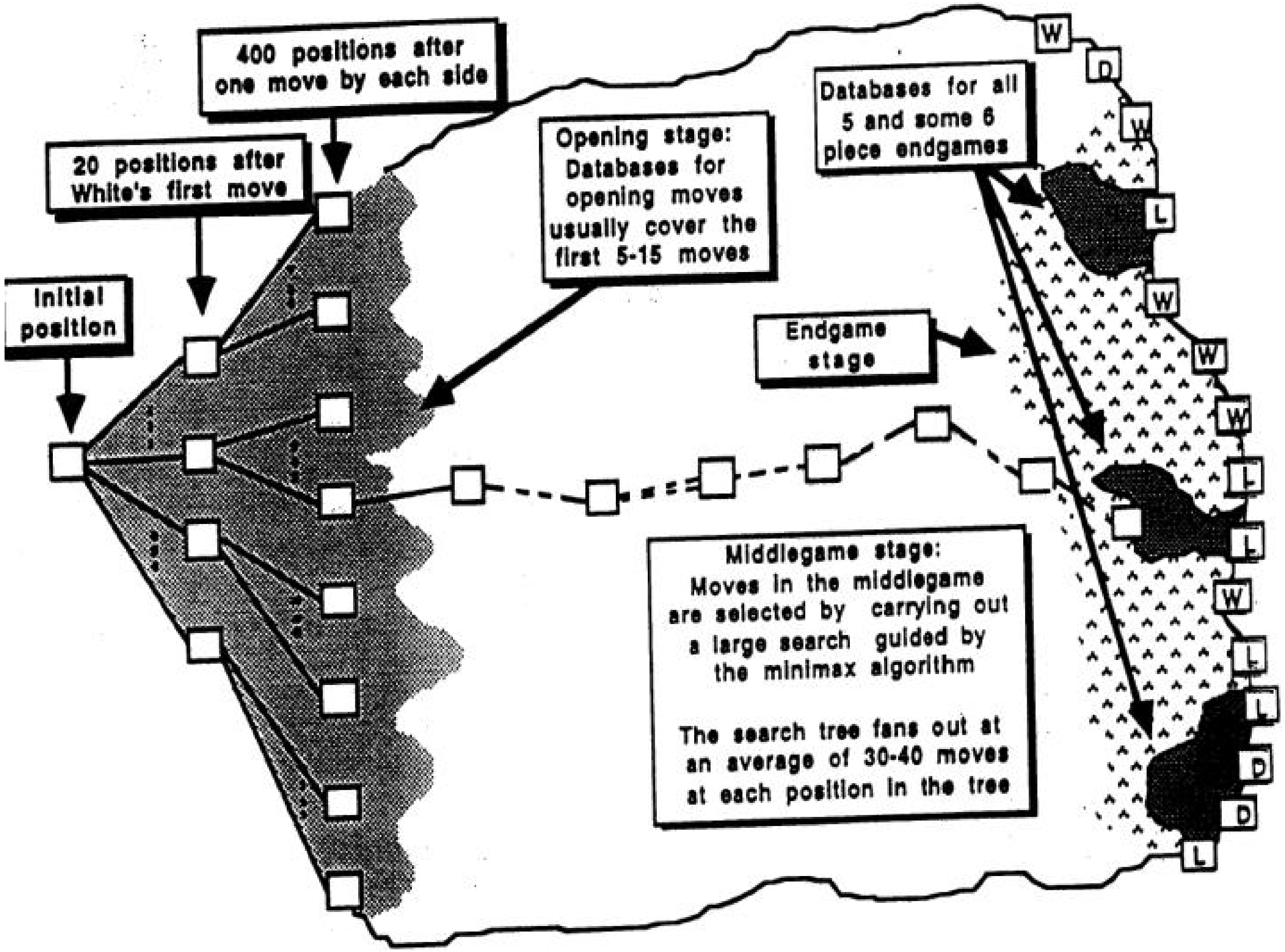
  This is what most game playing programs use

- Steps in designing an evaluation function:

  1. Pick informative features
  2. Find the weights that make the program play well

- Deep Blue used ~6,000 different features!

Minimax Search



Design Issues for Heuristic Minimax

- Depth-limited search:
  - search to a constant depth
- Problems:
  - Some portions of the game tree may be less stable than the others
  - Horizon effect

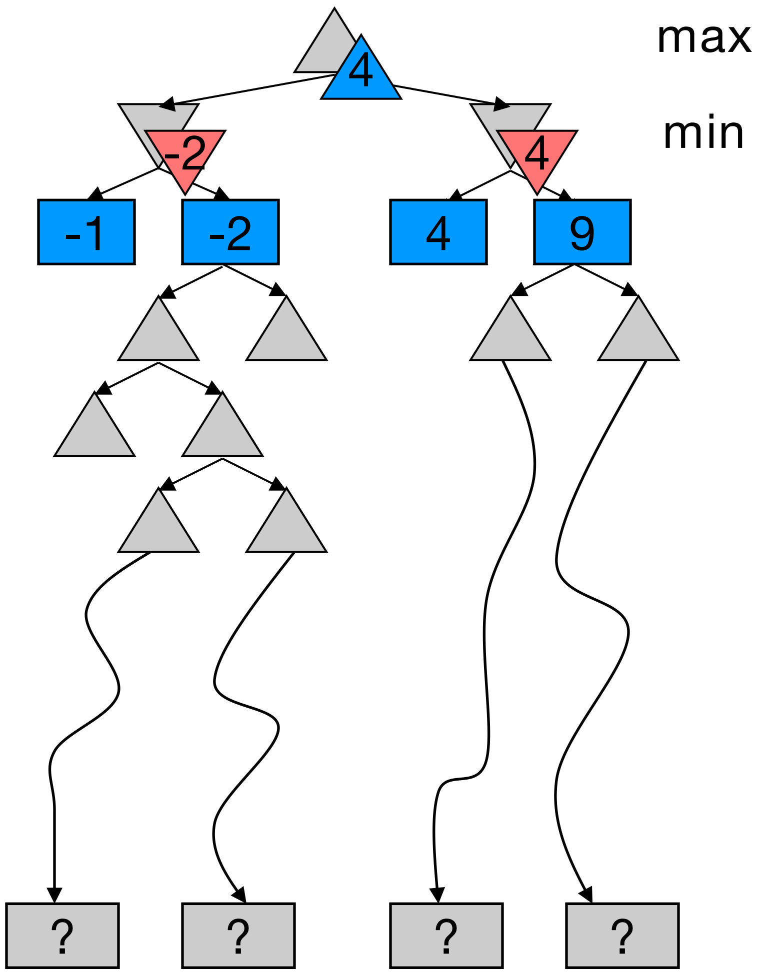
Unstable States

- Unstable state: drastic change from one level to the next
  - A chess evaluation function that counts material gains may evaluate a given state poorly even the play can capture a queen in the next move.
  - Are you about to lose an important piece?
- Evaluation functions can only be trusted when applied to stable board states
- One solution is to extend the normal search to look for stable states

The Horizon Effect

- You may incorrectly estimate the value of a state by overlooking an event that is just beyond the depth limit
  - Opponent moves, move is significant damage and cannot not be avoided

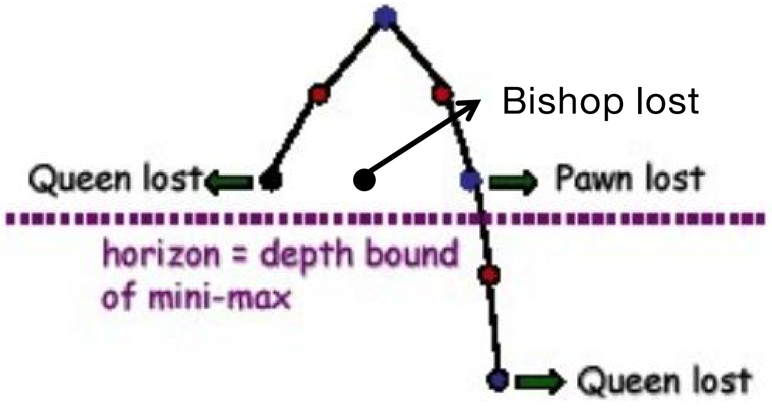
- Fixed-depth searches can be mislead by the fact that these damaging moves can be delayed
  - The damage is beyond the search horizon and so is not seen



The Horizon Effect

- The negative horizon effect
  - MAX may try to avoid a bad situation which is actually inevitable.
  - For example, MAX tries to avoid losing the queen and appears to be able to do so using a lookahead tree of depth 6, but a little deeper it becomes obvious that the queen is going to be lost.

Piece	Value
Pawn	1
Knight	3
Bishop	3
Rook	5
Queen	9



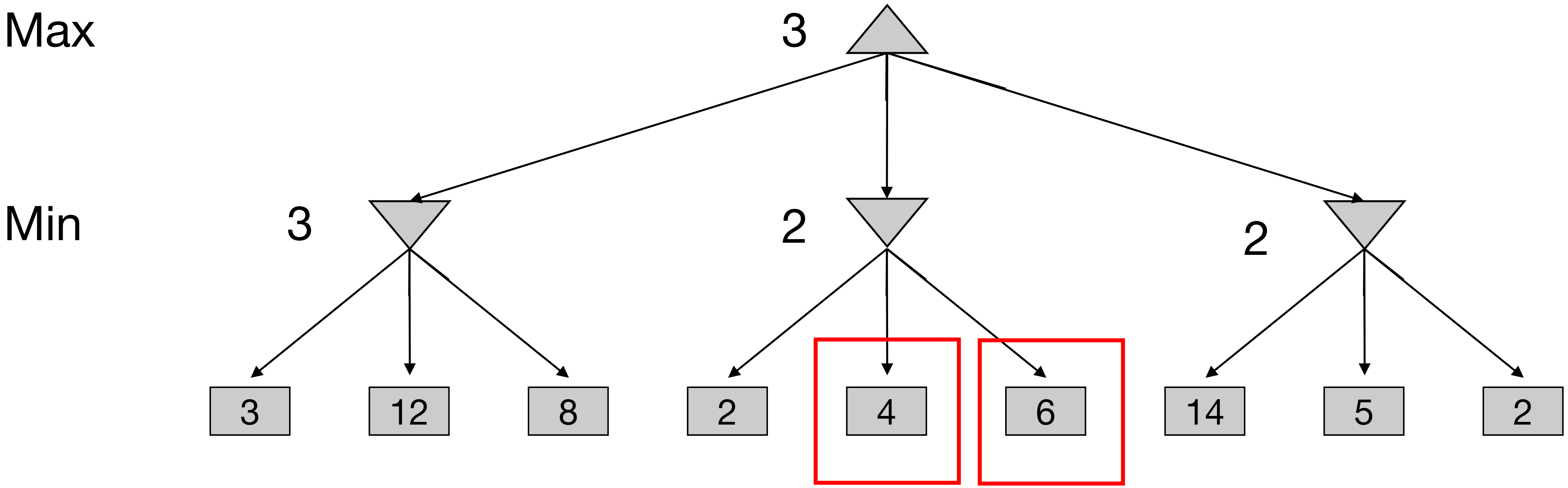
The Horizon Effect

The positive horizon effect:

MAX may not realize that something good is going to be achievable.

For example, MAX would like to take MIN's queen and that can happen
- but the restricted horizon prevents MAX from making the right choices to realize this possibility

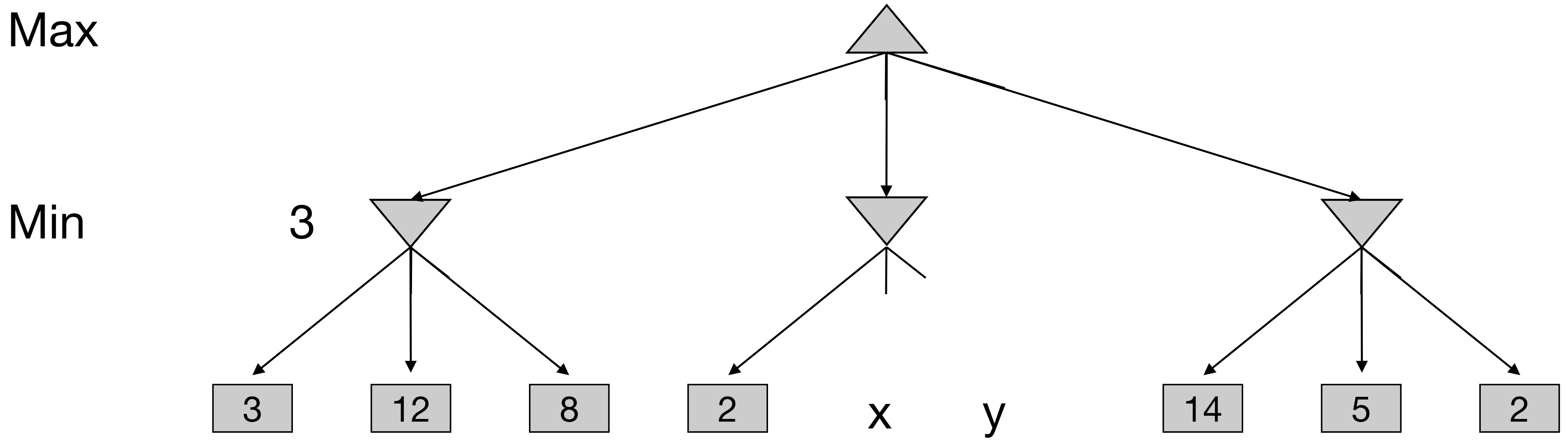
α - β pruning



Improving Minimax -- α - β pruning

- The number of game states is exponential in the depth of the tree.
- It is possible to prune large parts of the tree from consideration.
- Intuition: prune away branches that cannot possibly influence the value of the state.

Improving Minimax -- α - β pruning

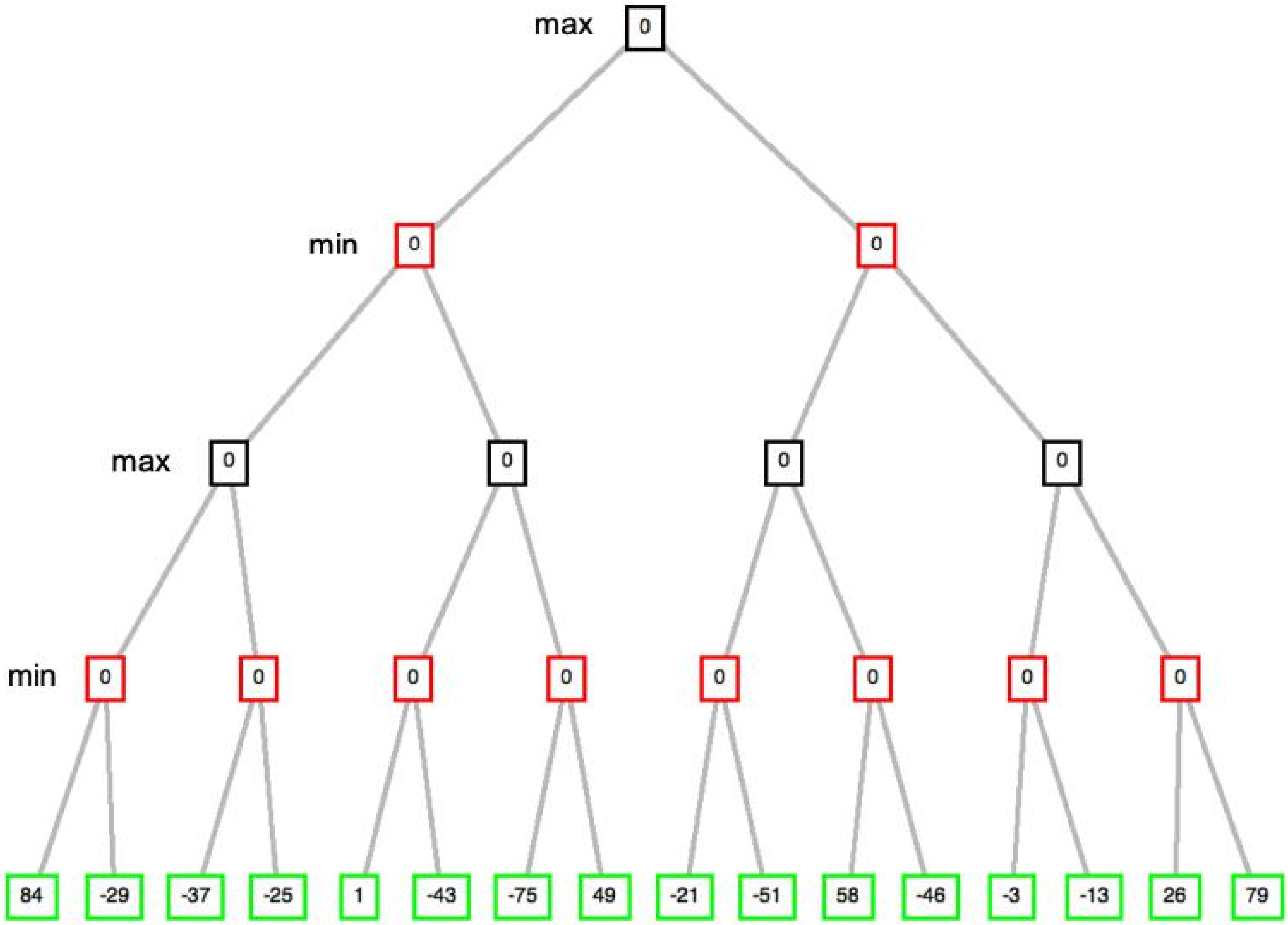


Algebraic Solution

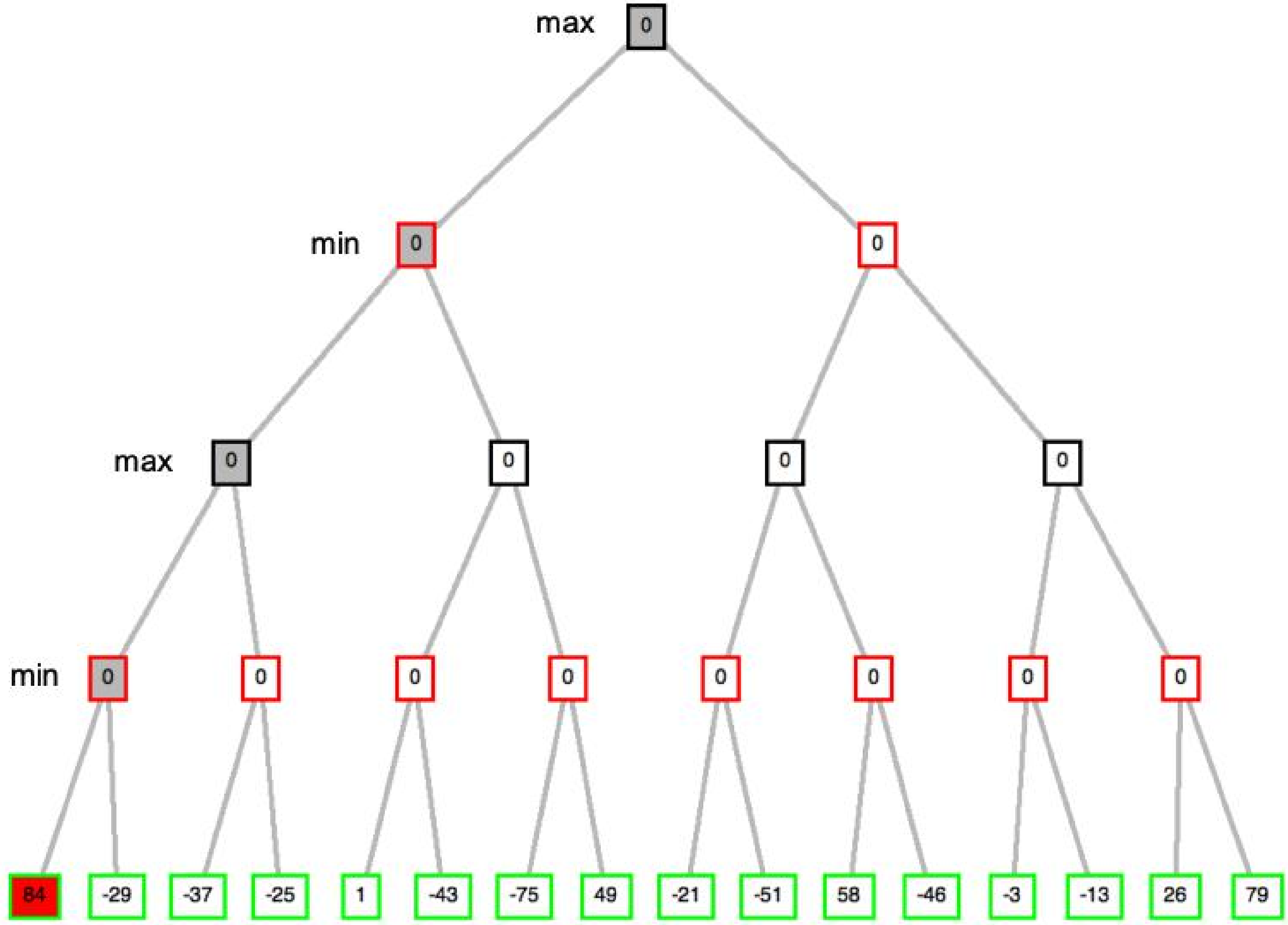
- Minimax(root) = max(min(3, 12, 8), min(2, x, y), min(14, 5, 2))
  = max(3, min(2, x, y), 2)
  = max(3, z, 2) where z = min(2, x, y) <= 2
  = 3

- The value of the root is independent of the values of x and y

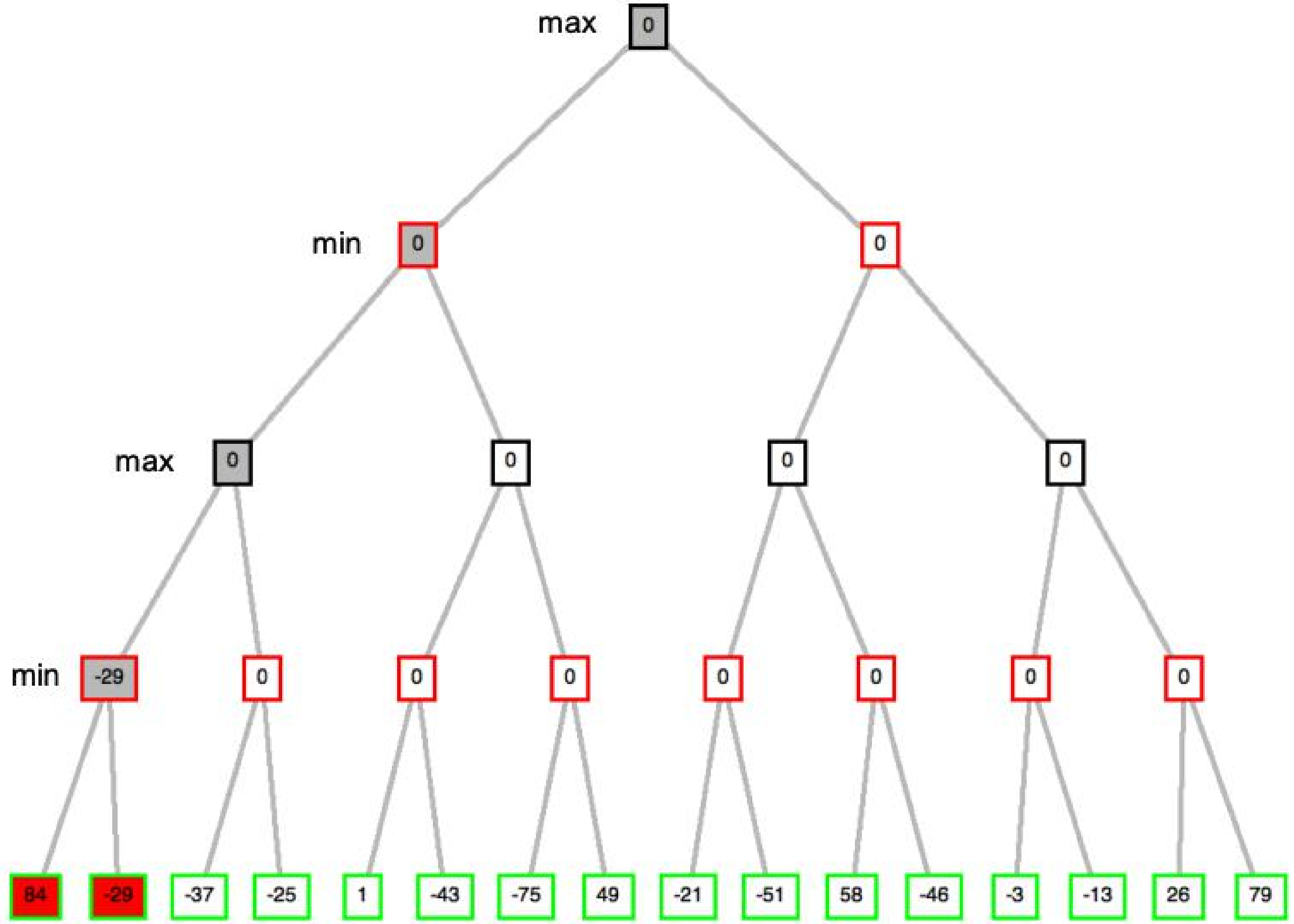
Another Example



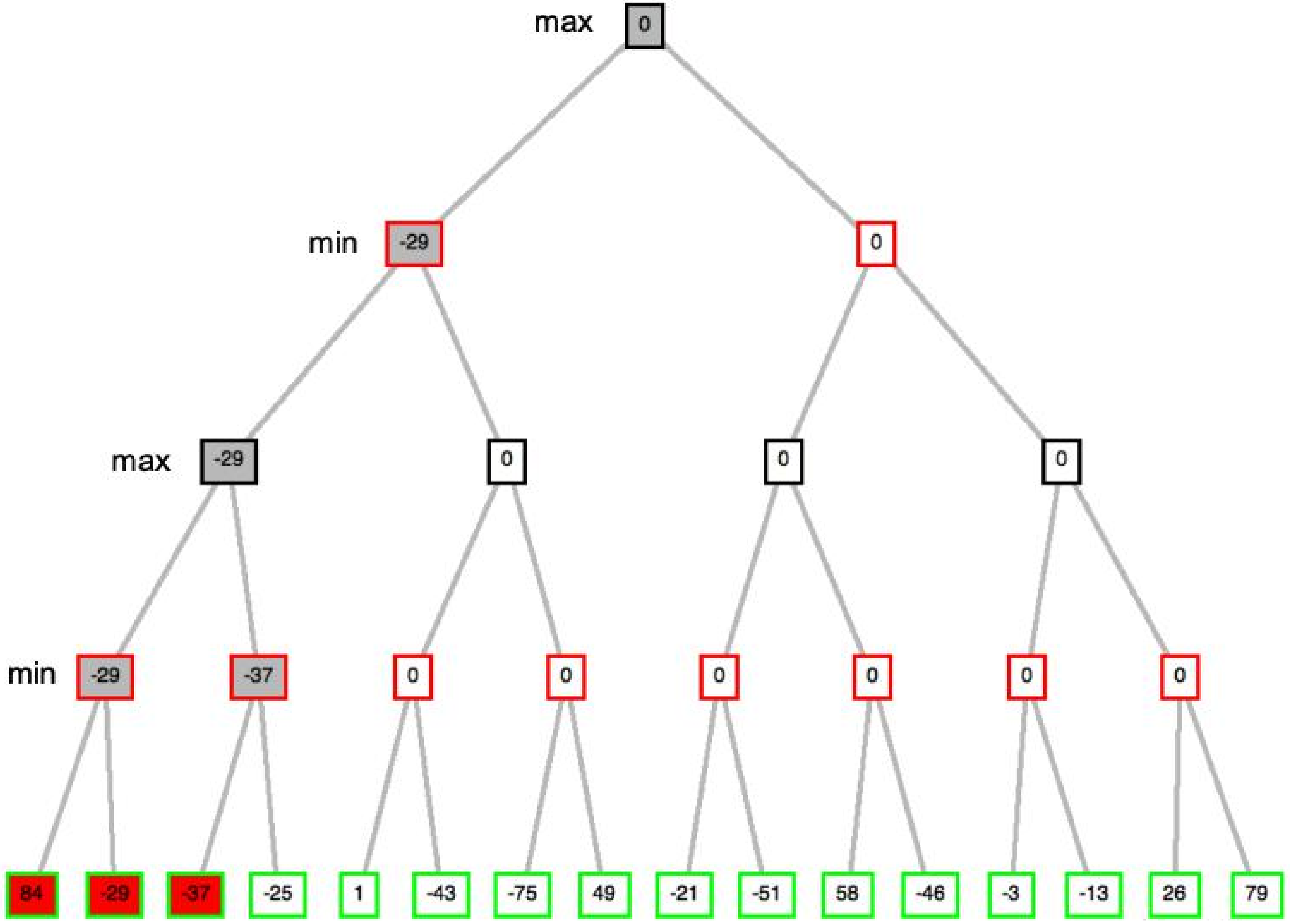
Another Example



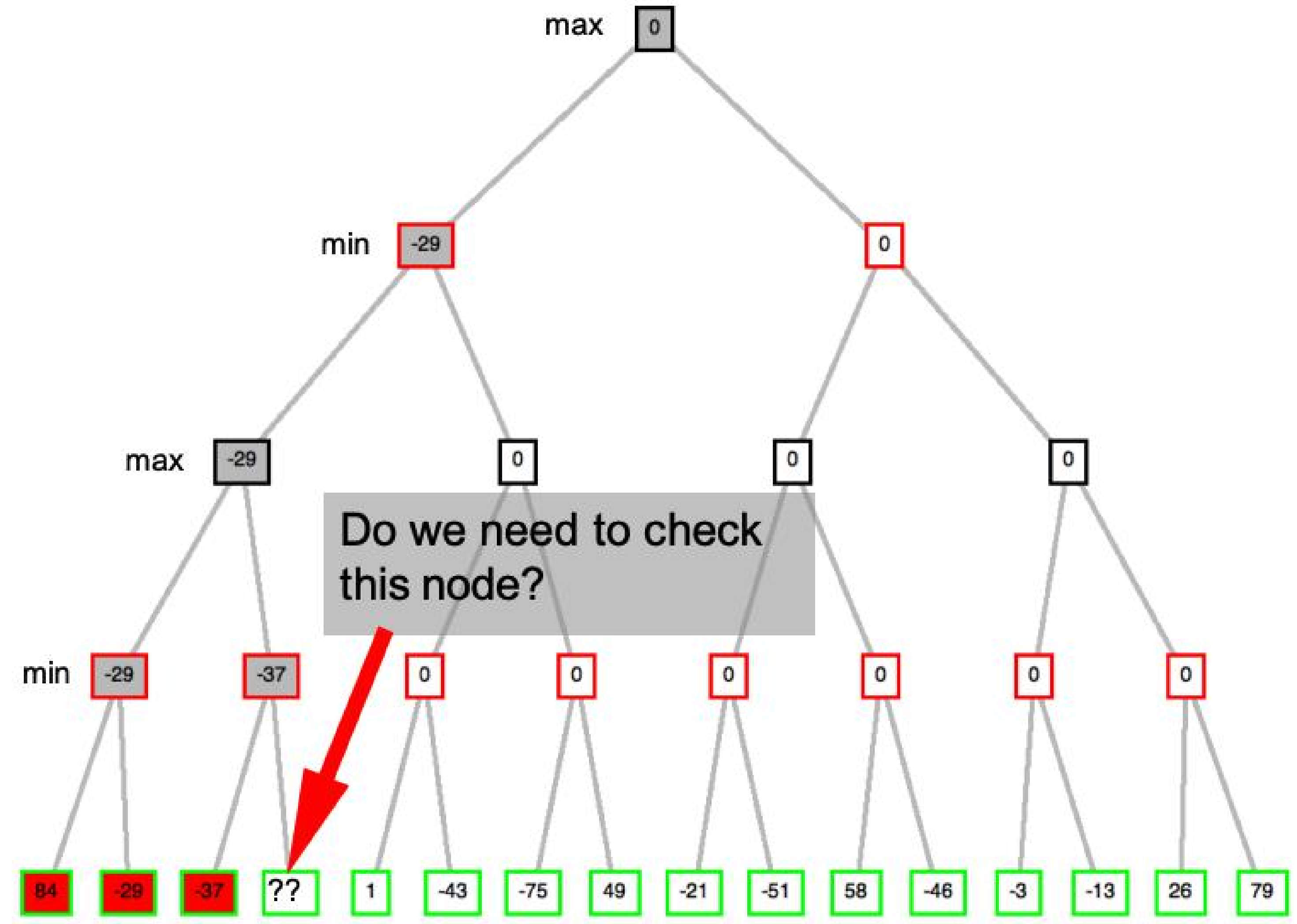
Another Example



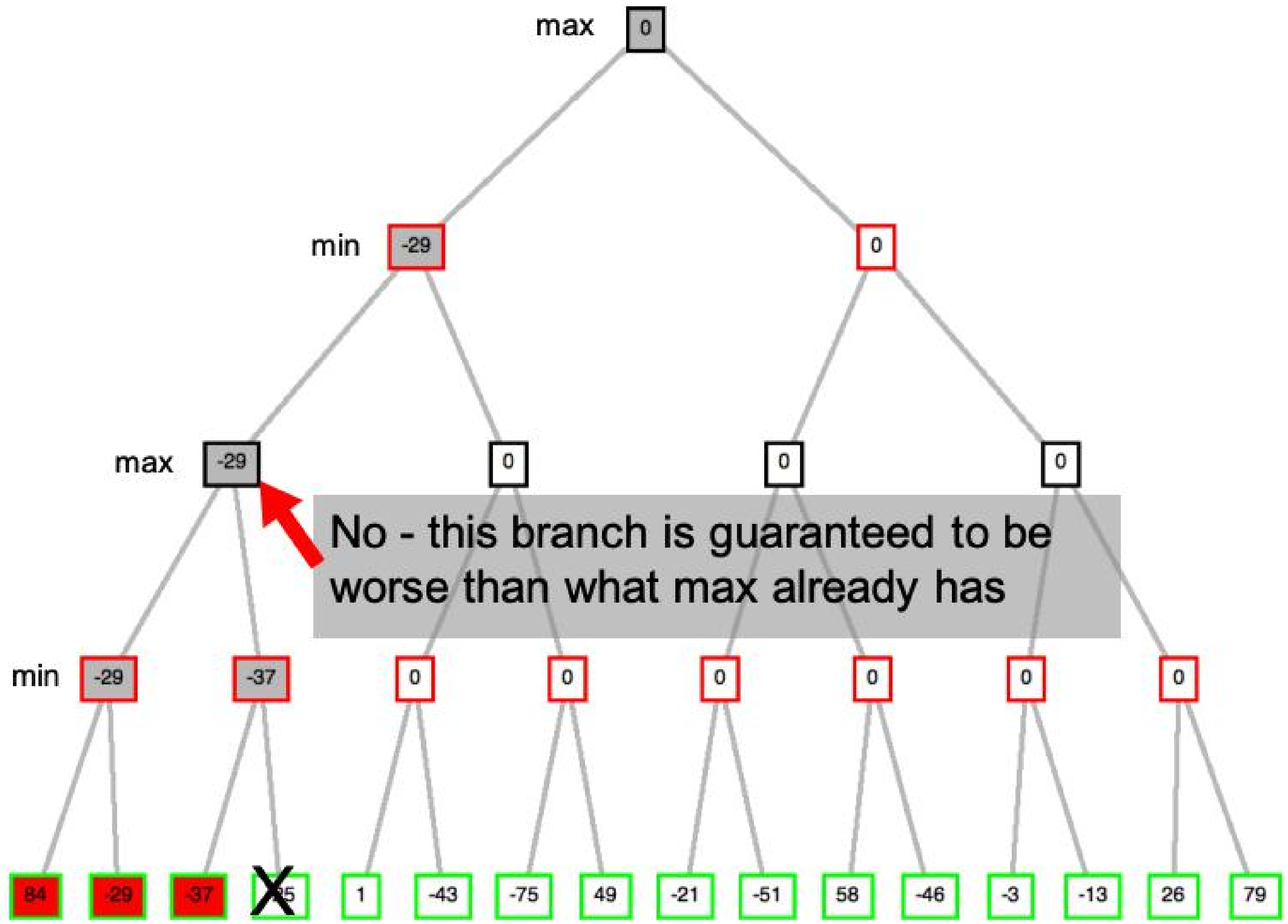
Another Example



Another Example



Another Example



Improving Minimax -- α - β pruning

- The alpha-beta procedure can speed up a depth-first minimax search.

- Alpha: a lower bound on the value that a max node may ultimately be assigned at that level or above

$$v \geq \alpha$$

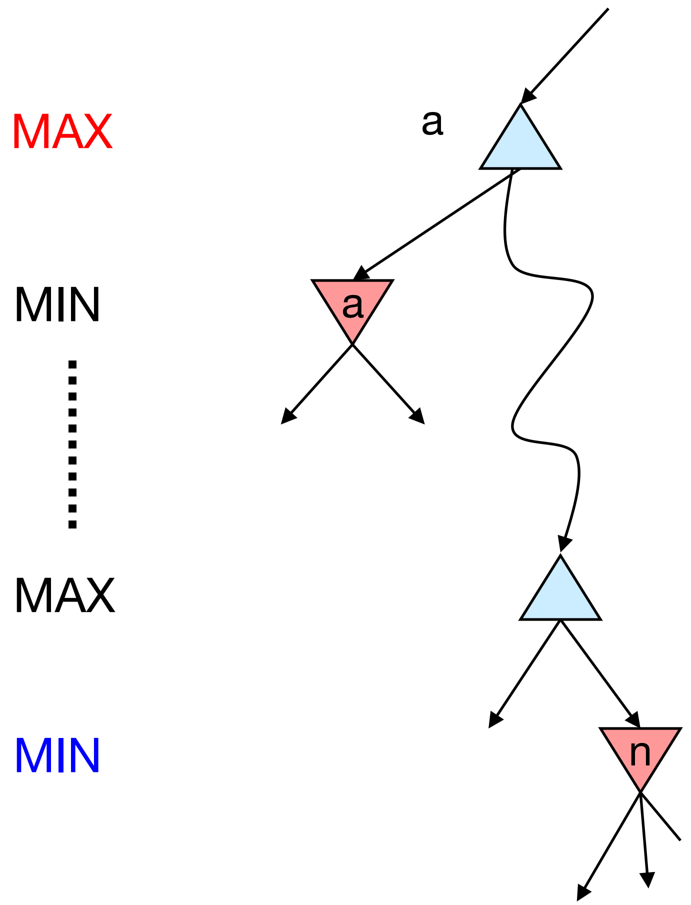
- Beta: an upper bound on the value that a min node may ultimately be assigned at that level or above

$$v \leq \beta$$

α - β Pruning

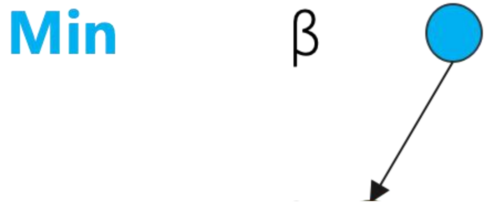
- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node n
  - We're looping over n's children
  - n's estimate of the childrens' min is dropping
  - Who cares about n's value? MAX
  - Let a be the best value that MAX can get at any choice point along the current path from the root
  - If n becomes worse than a, MAX will avoid it, so we can stop considering n's other children

    (it's already bad enough that it won't be played)



α - β Pruning

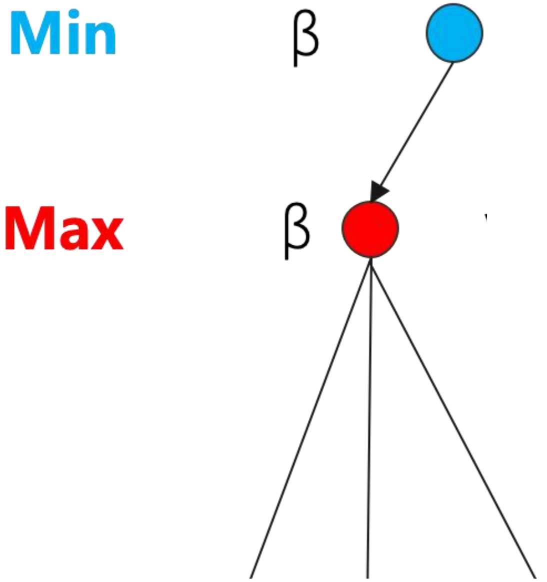
Beta: an upper bound on the value that a min node may ultimately be assigned at that level or above
Alpha: a lower bound on the value that a max node may ultimately be assigned at that level or above



α - β Pruning

Beta: an upper bound on the value that a min node may ultimately be assigned at that level or above

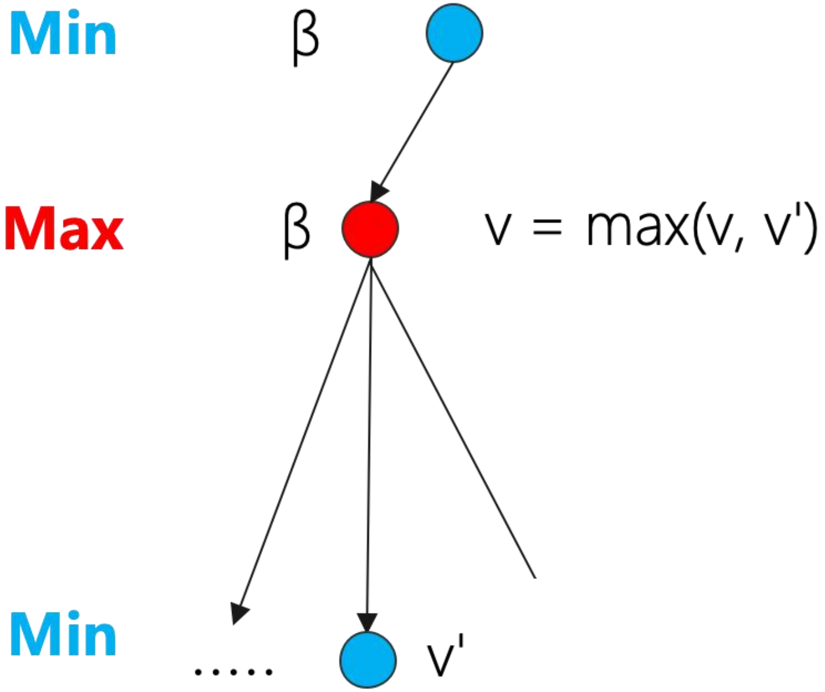
Alpha: a lower bound on the value that a max node may ultimately be assigned at that level or above



α - β Pruning

Beta: an upper bound on the value that a min node may ultimately be assigned at that level or above

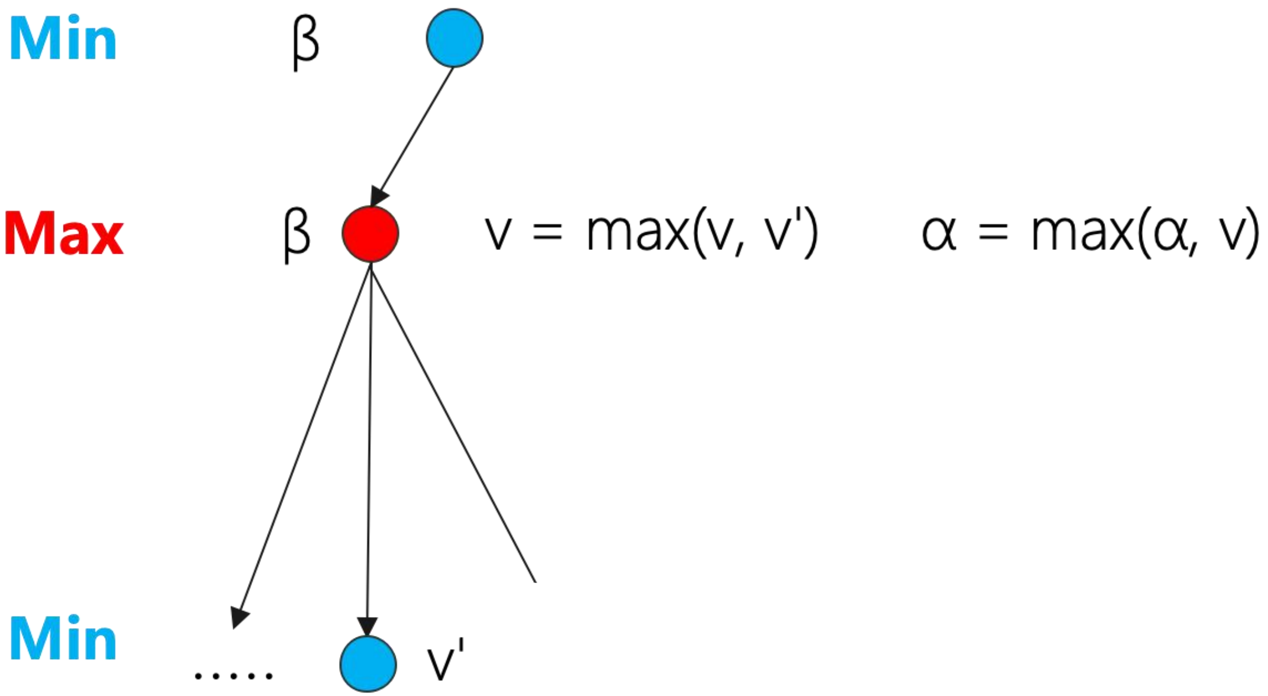
Alpha: a lower bound on the value that a max node may ultimately be assigned at that level or above



α - β Pruning

Beta: an upper bound on the value that a min node may ultimately be assigned at that level or above

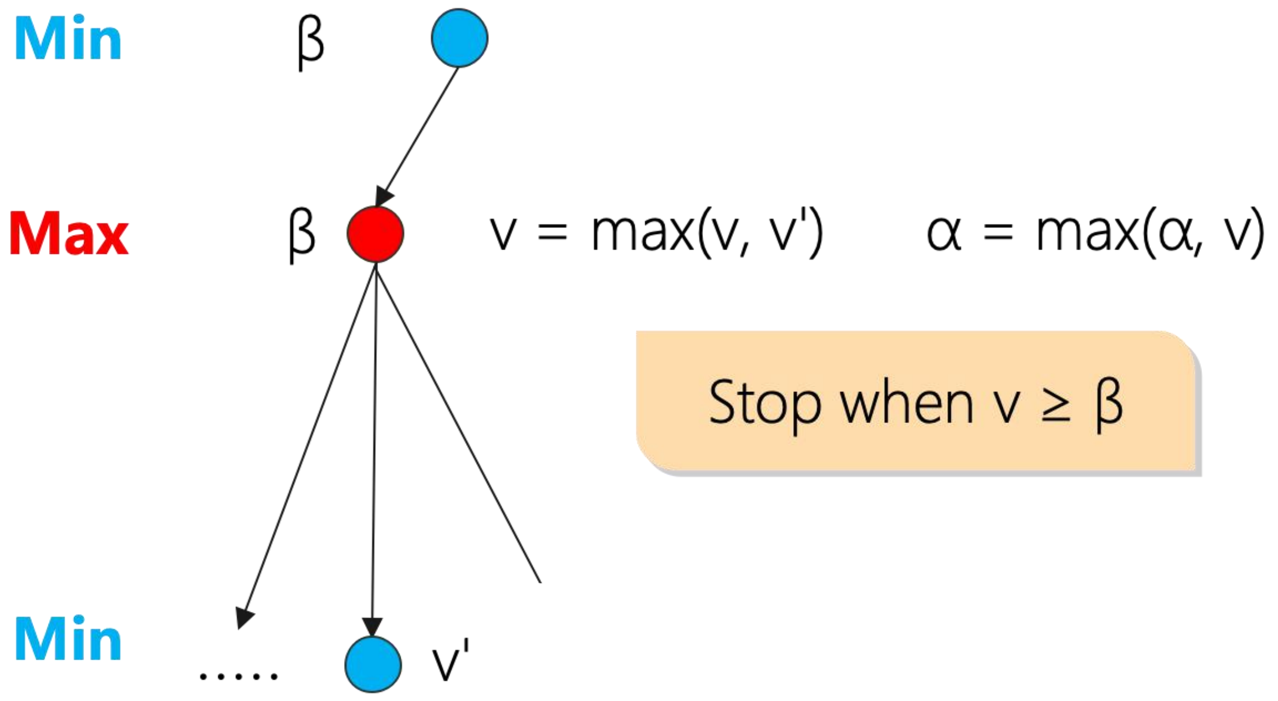
Alpha: a lower bound on the value that a max node may ultimately be assigned at that level or above



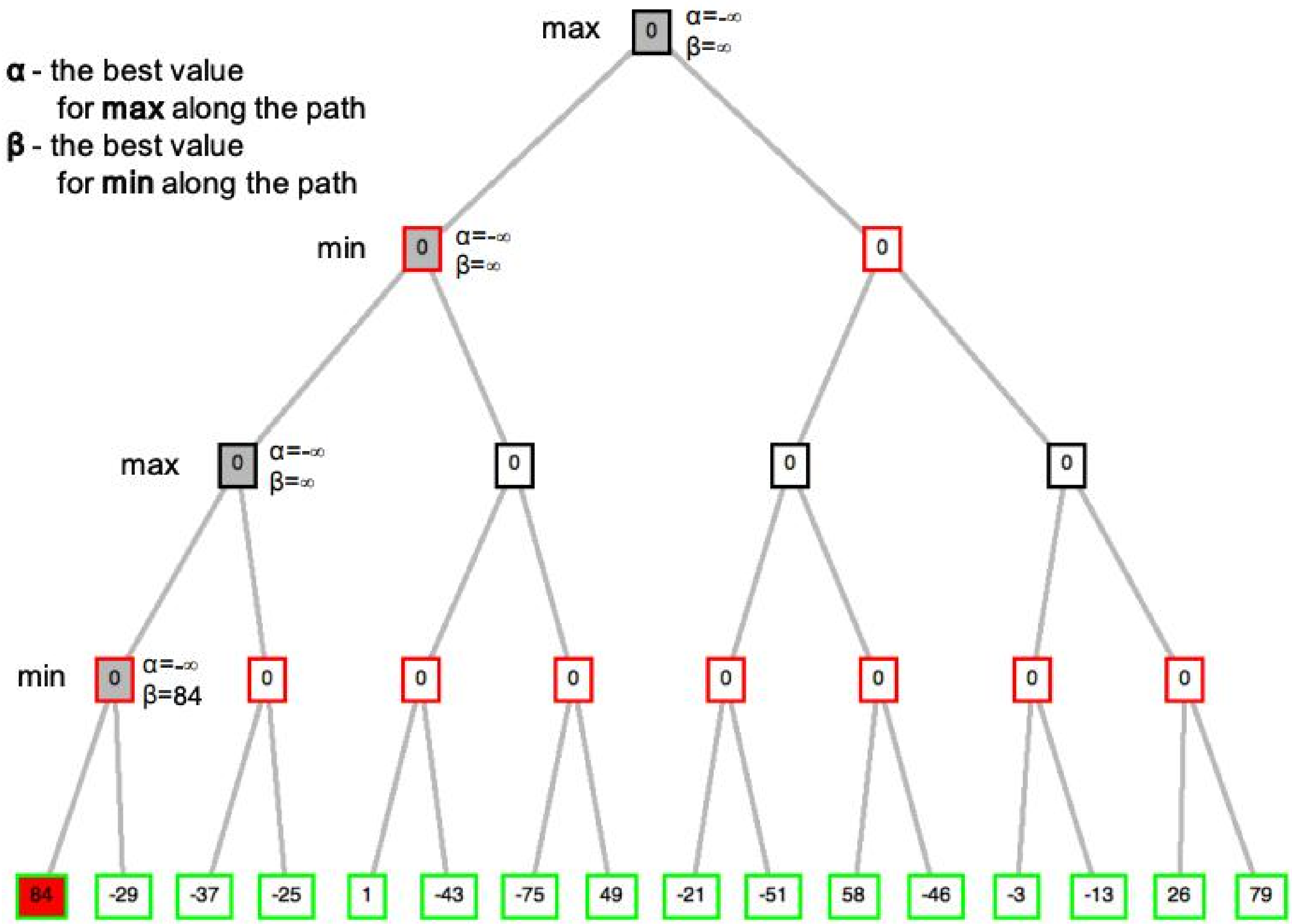
α - β Pruning

Beta: an upper bound on the value that a min node may ultimately be assigned at that level or above

Alpha: a lower bound on the value that a max node may ultimately be assigned at that level or above

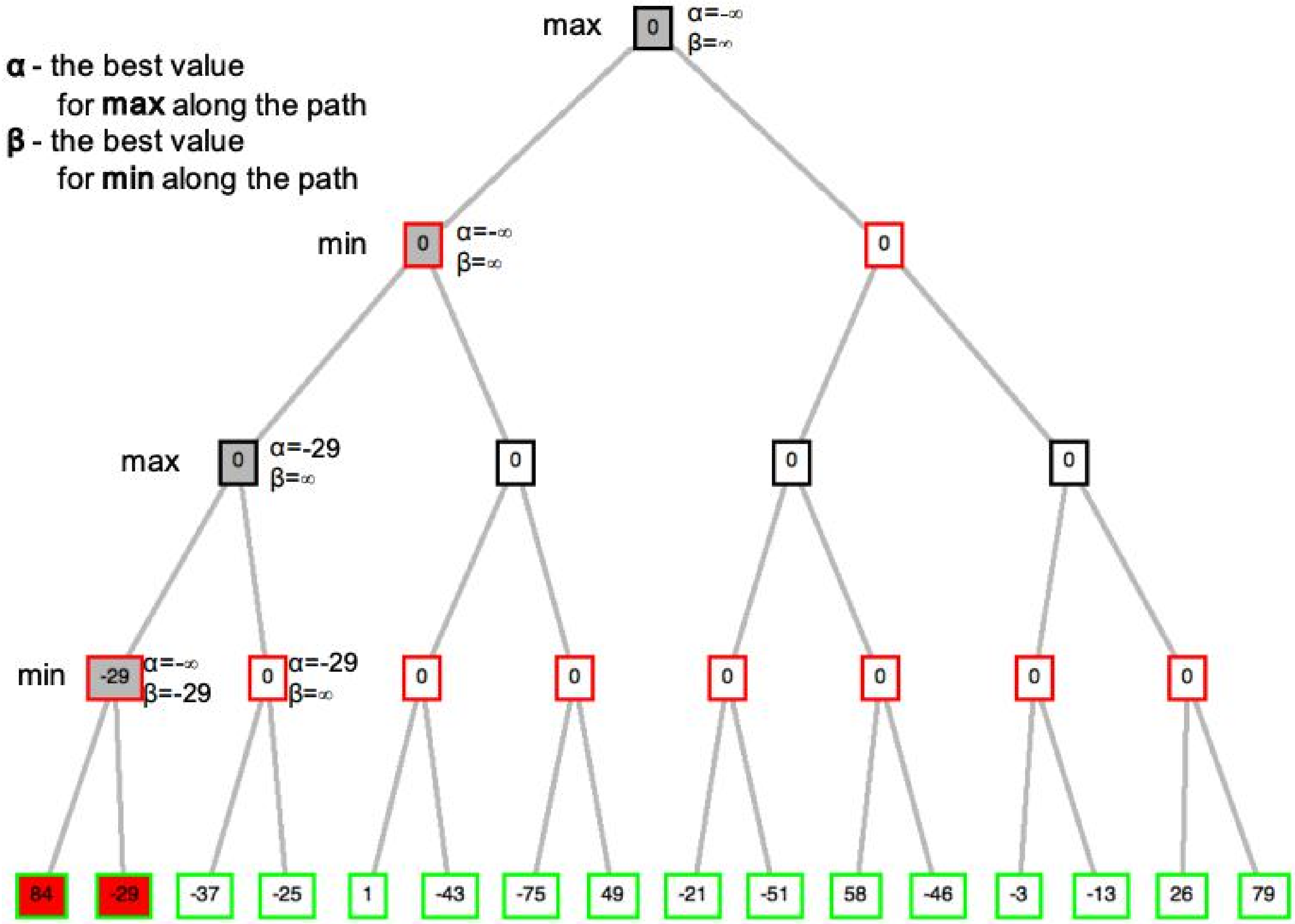


Another Example



update β,
use α to stop

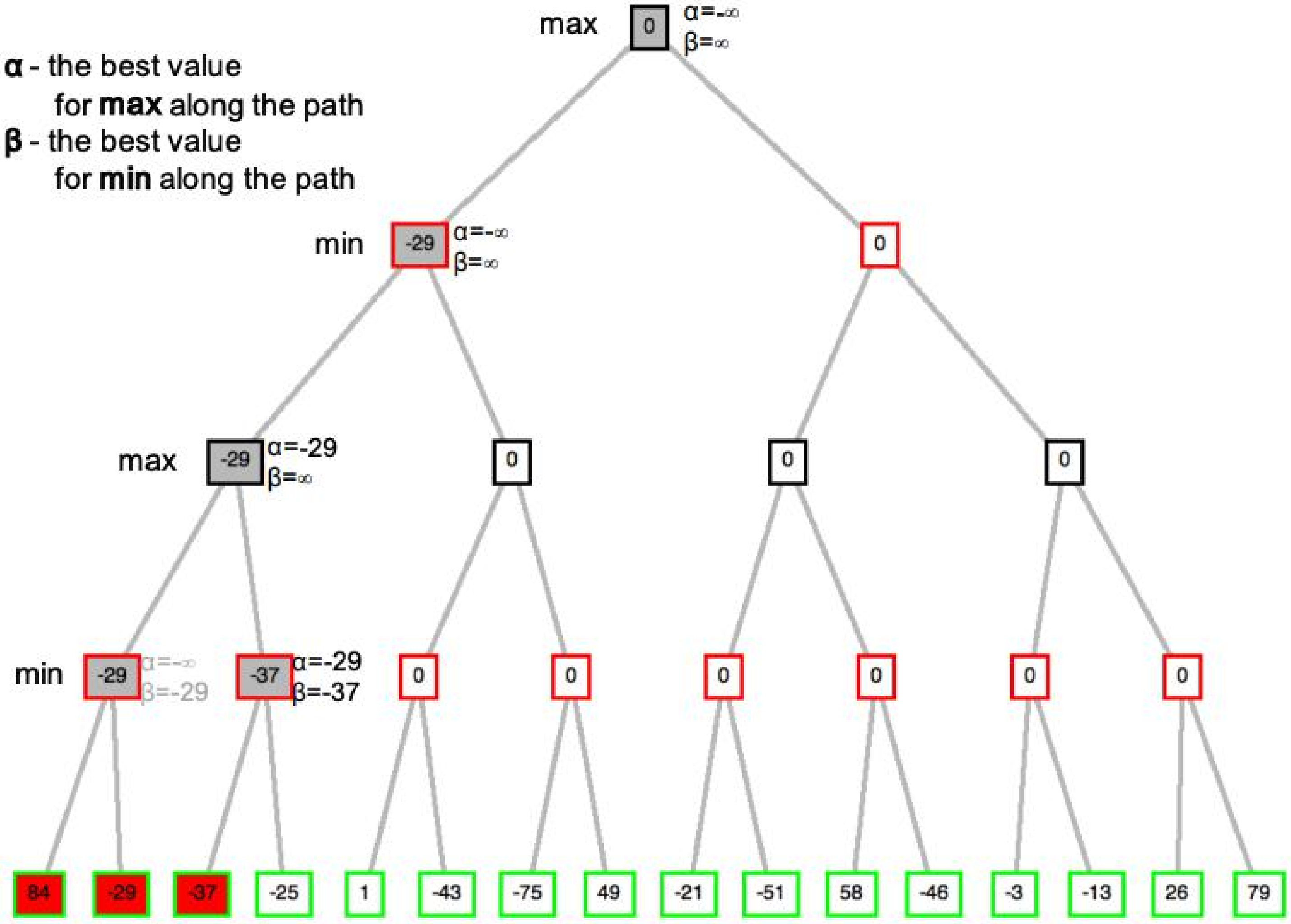
Another Example



update β,
use α to stop

Another Example

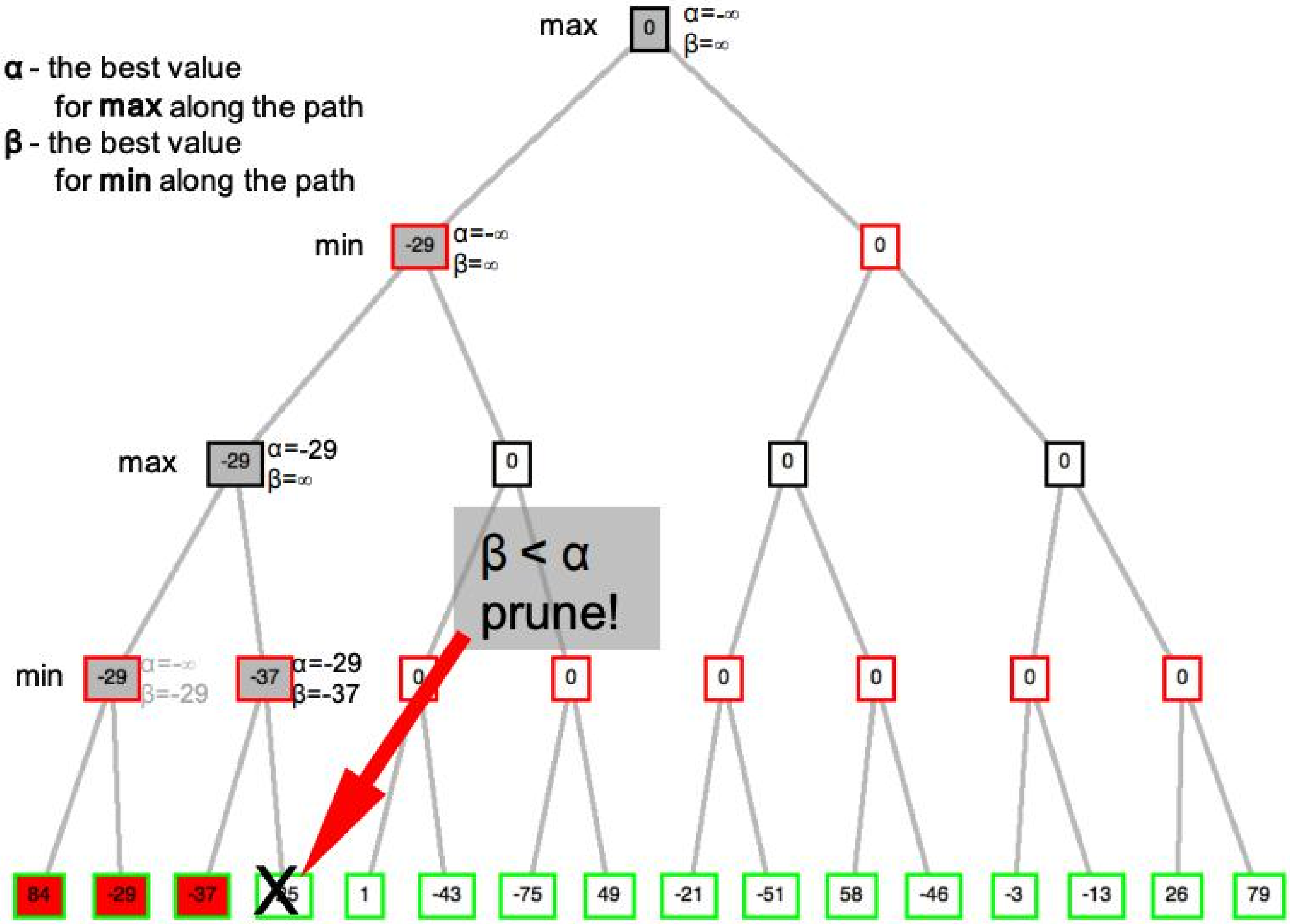
α - the best value
for **max** along the path
β - the best value
for **min** along the path



update β,
use α to stop

Another Example

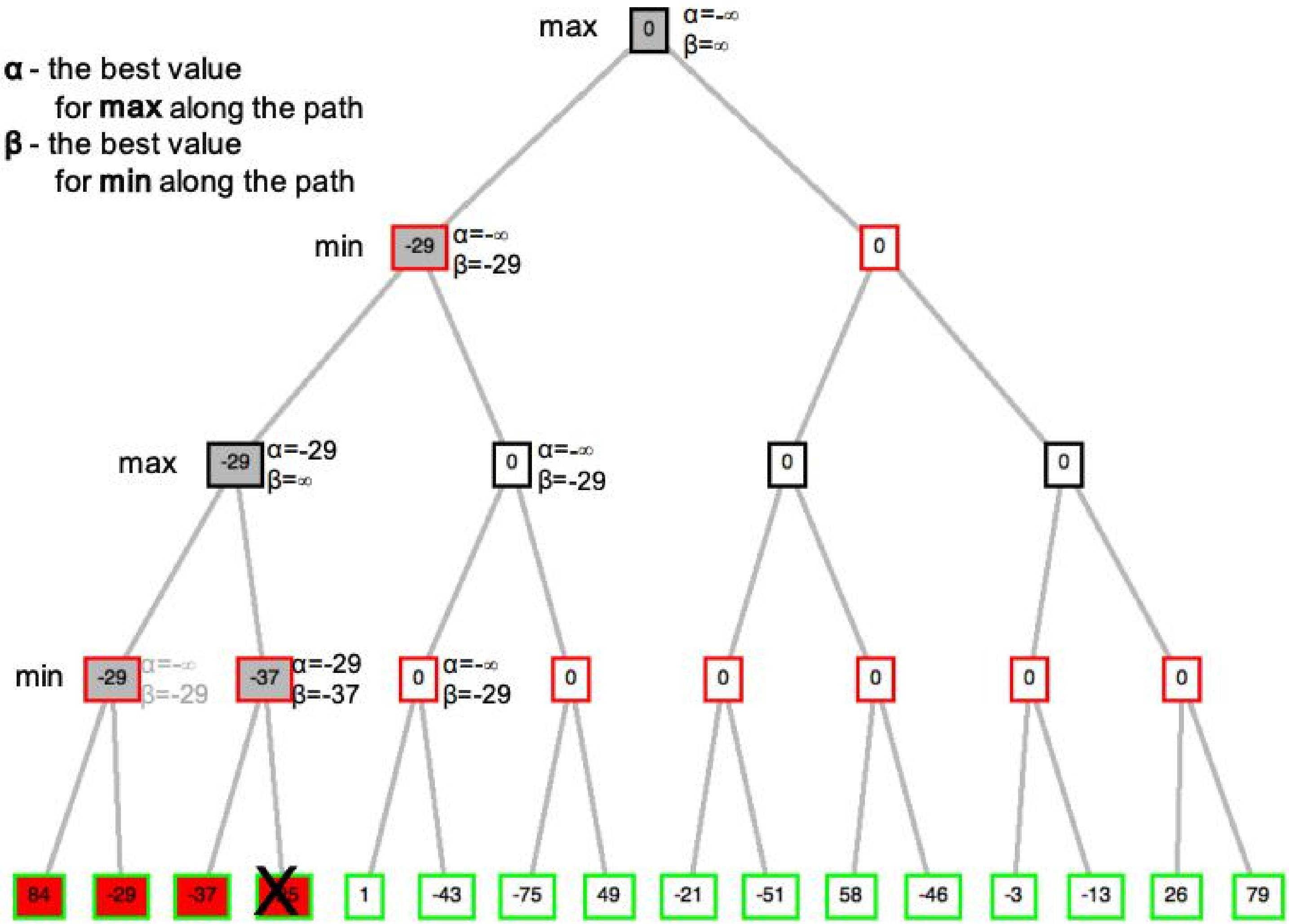
α - the best value
for **max** along the path
β - the best value
for **min** along the path



update β,
use α to stop

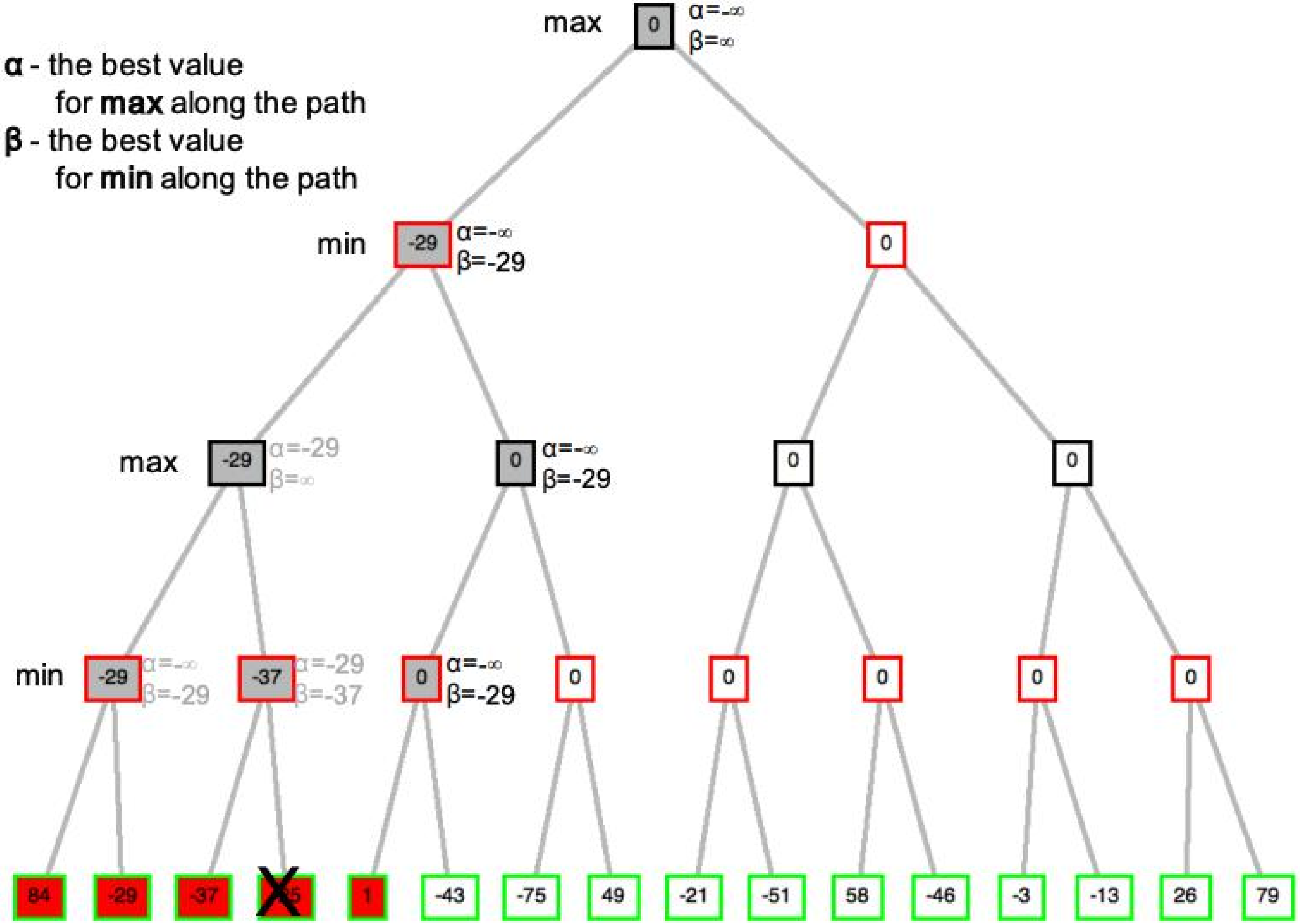
Another Example

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



update β,
use α to stop

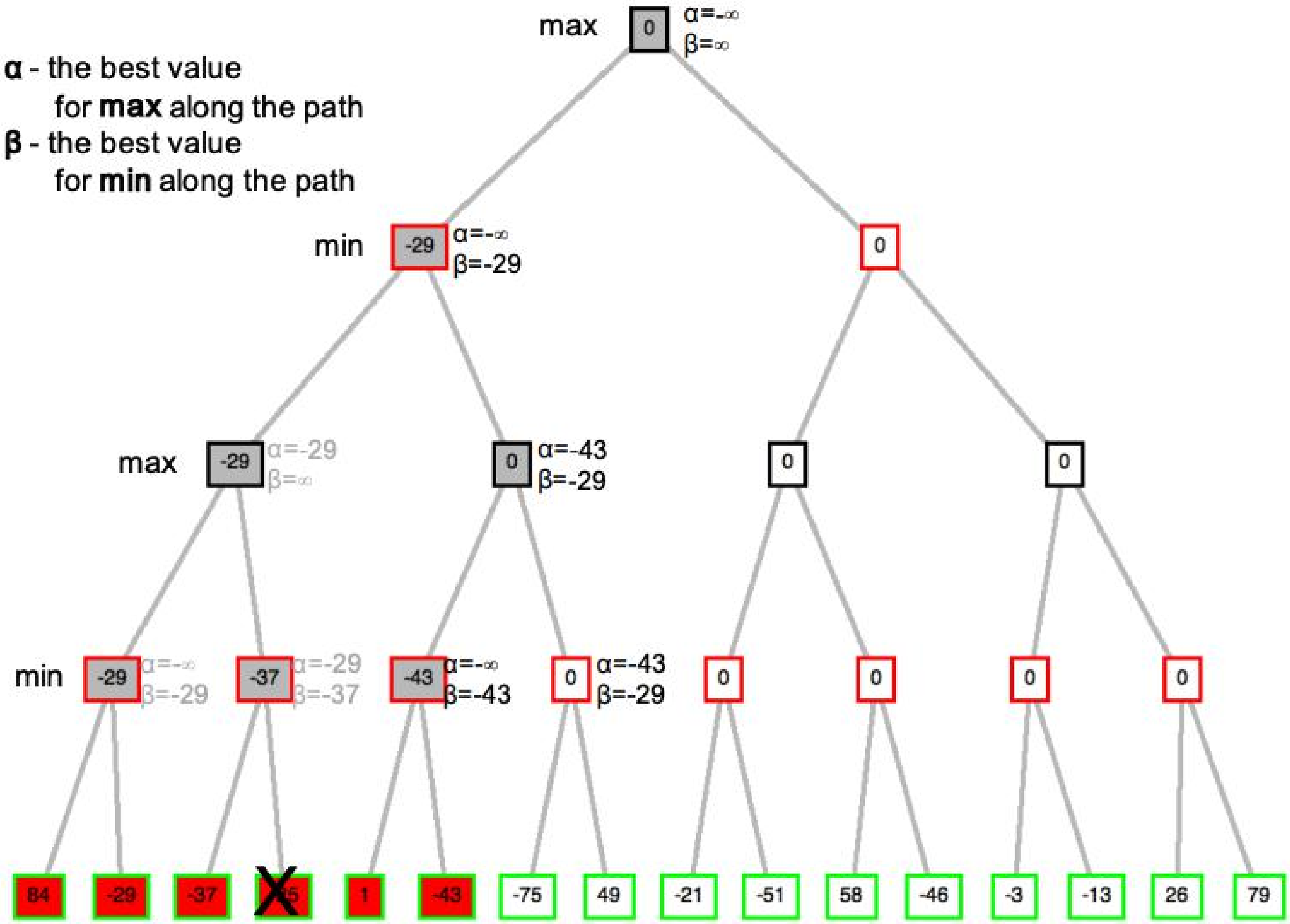
Another Example



update β,
use α to stop

Another Example

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

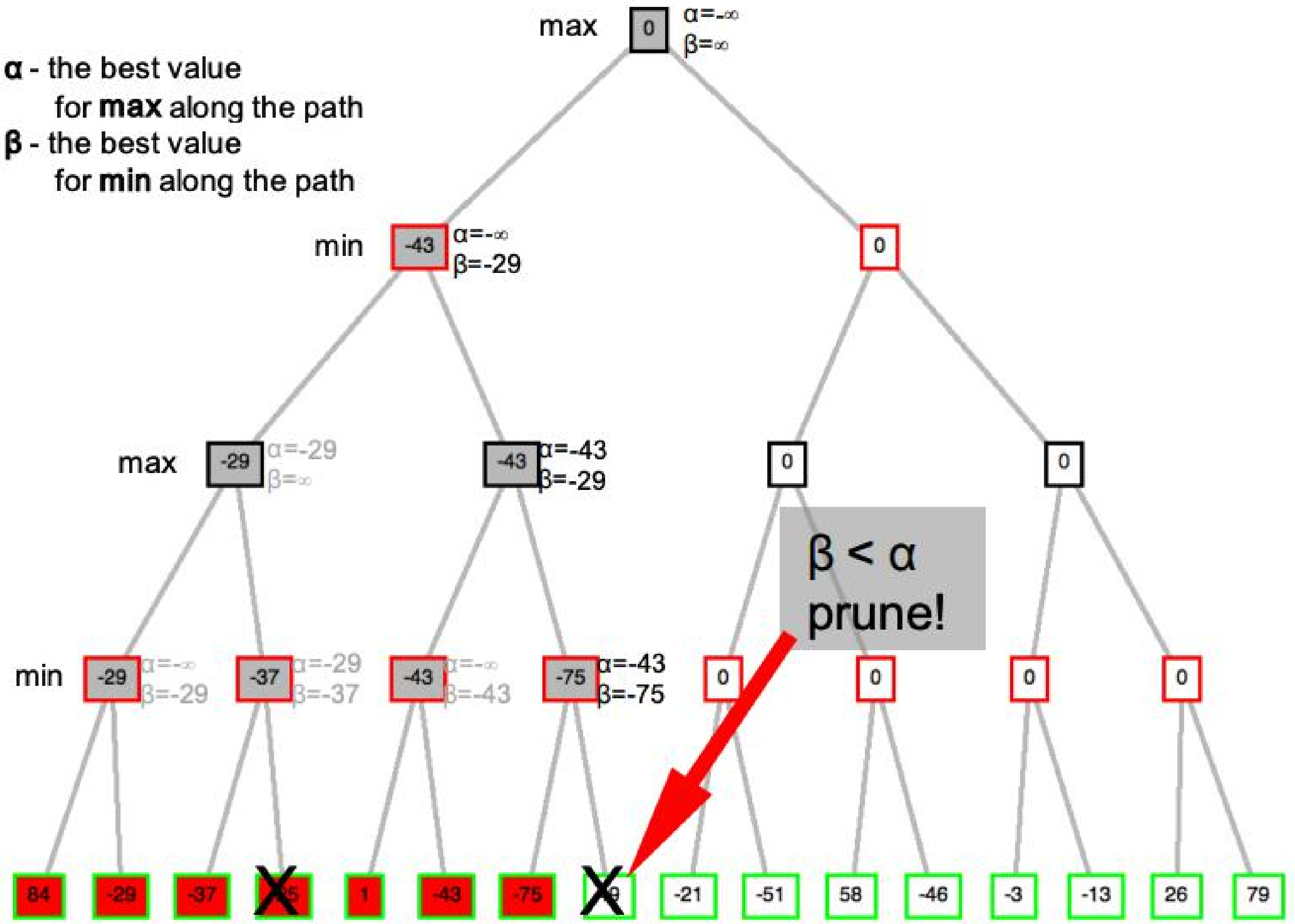


update β,
use α to stop

Another Example

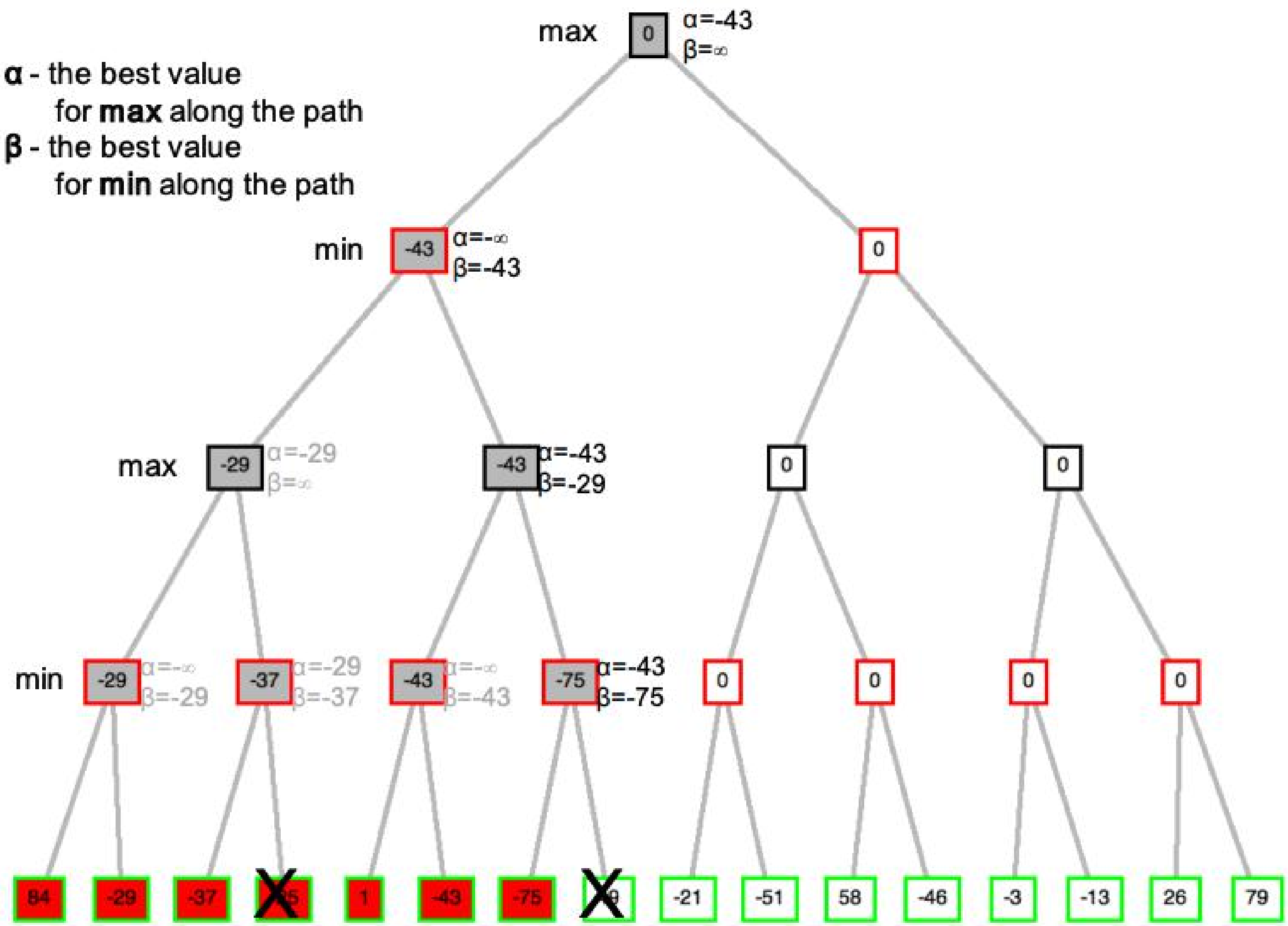
α - the best value for **max** along the path
β - the best value for **min** along the path

update β,
use α to stop



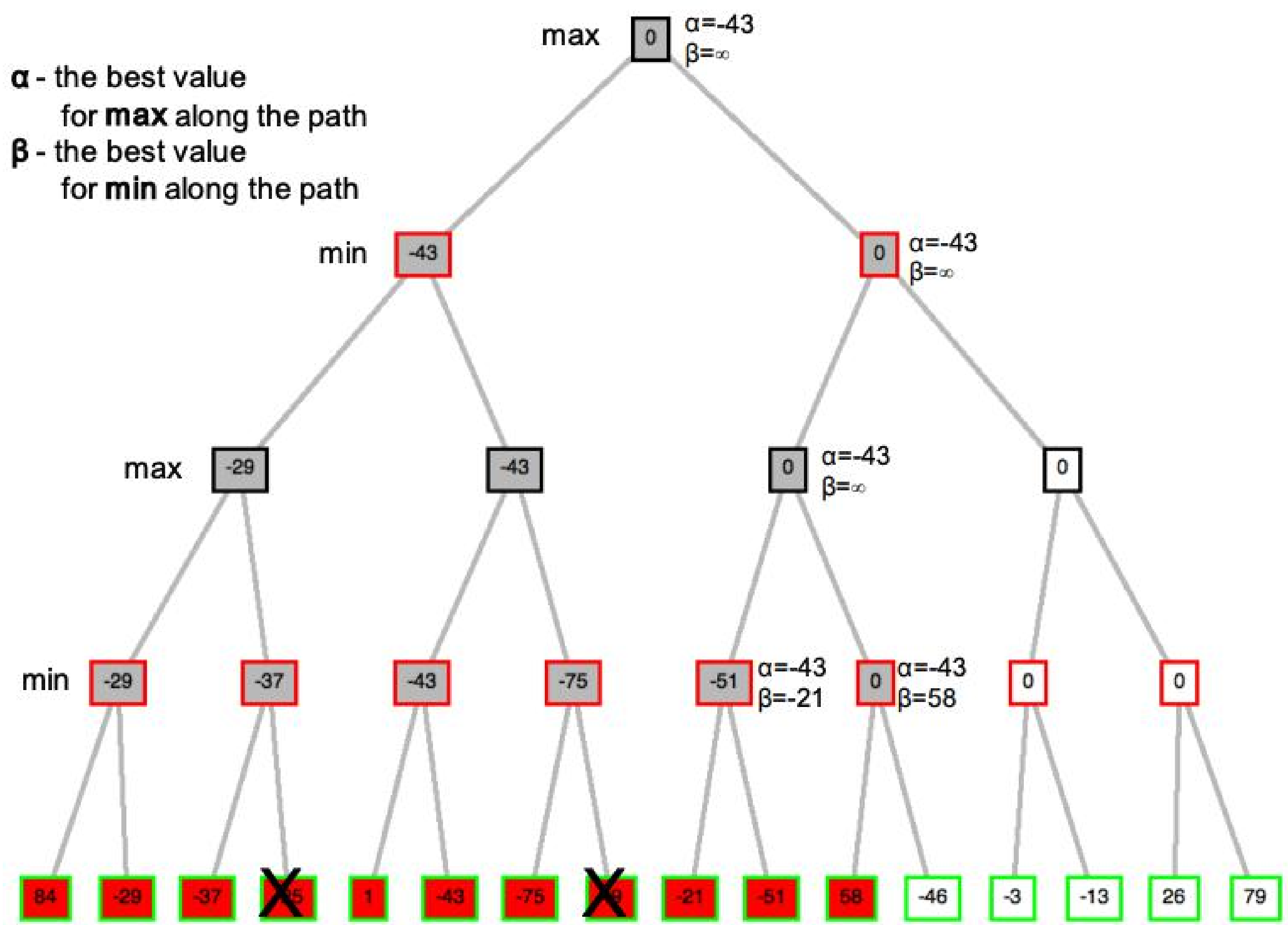
Another Example

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



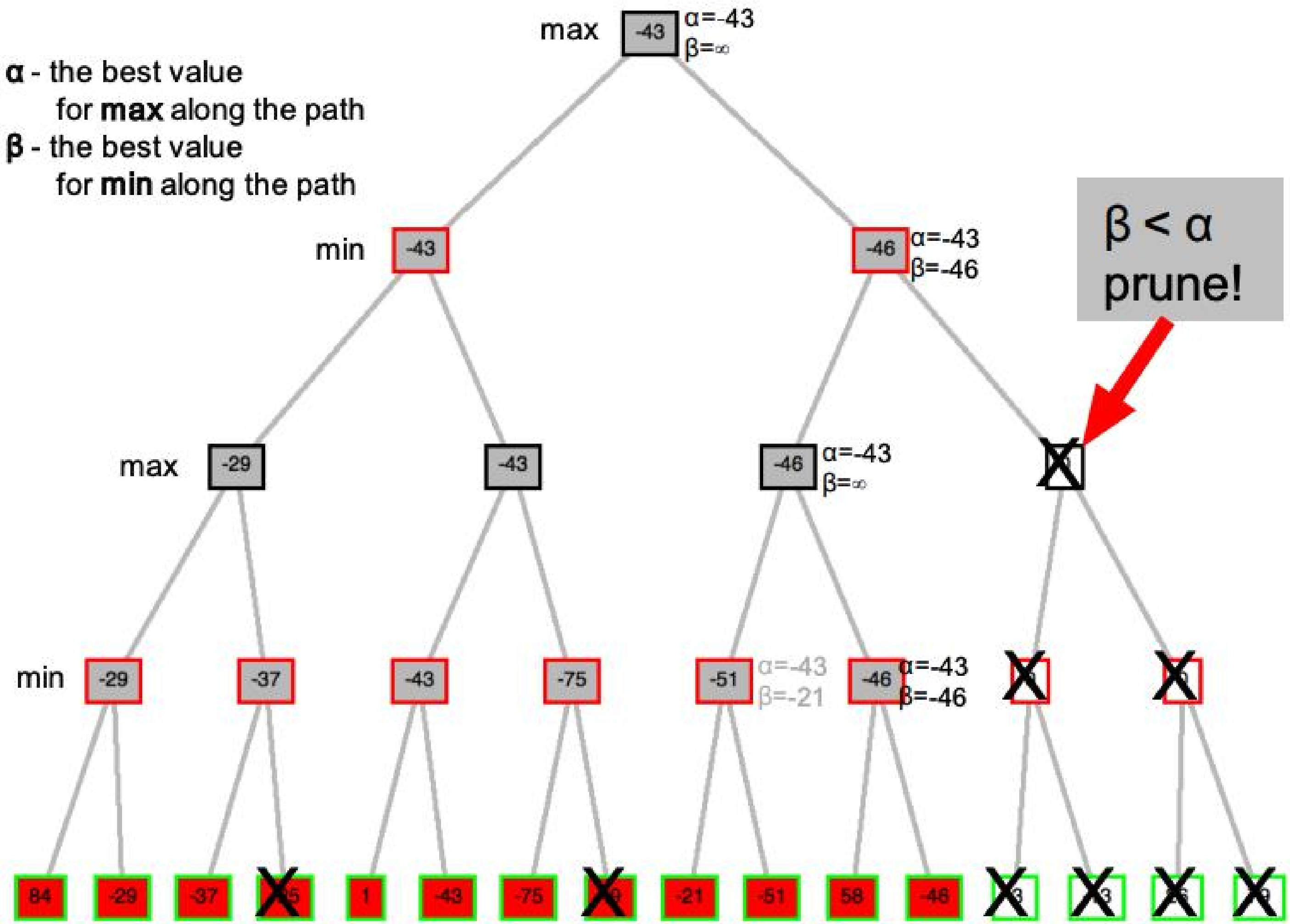
update β,
use α to stop

Another Example



update β,
use α to stop

Another Example



update β,
use α to stop

α - β Search

- c = search cutoff
- α = lower bound on Max's outcome; initially set to $-\infty$
- β = upper bound on Min's outcome; initially set to $+\infty$

- We'll call α - β procedure recursively with a narrowing range between α *and* β

- Maxmizing levels may reset α to a higher value;
- Minimizing levels may reset β to a lower value.

α - β Search Algorithm

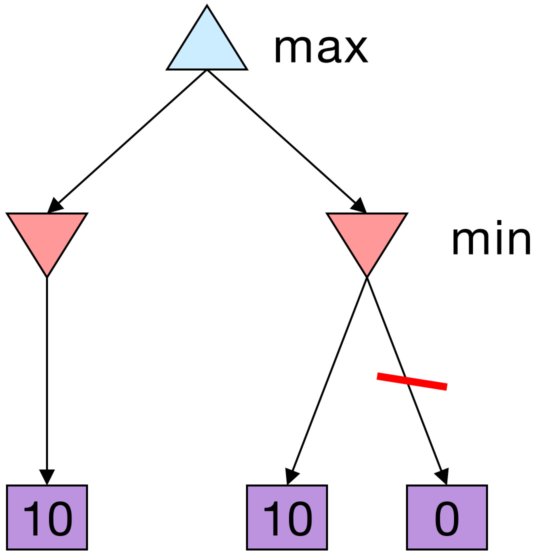
α: MAX's best option on path to root
β: MIN's best option on path to root

```
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
```

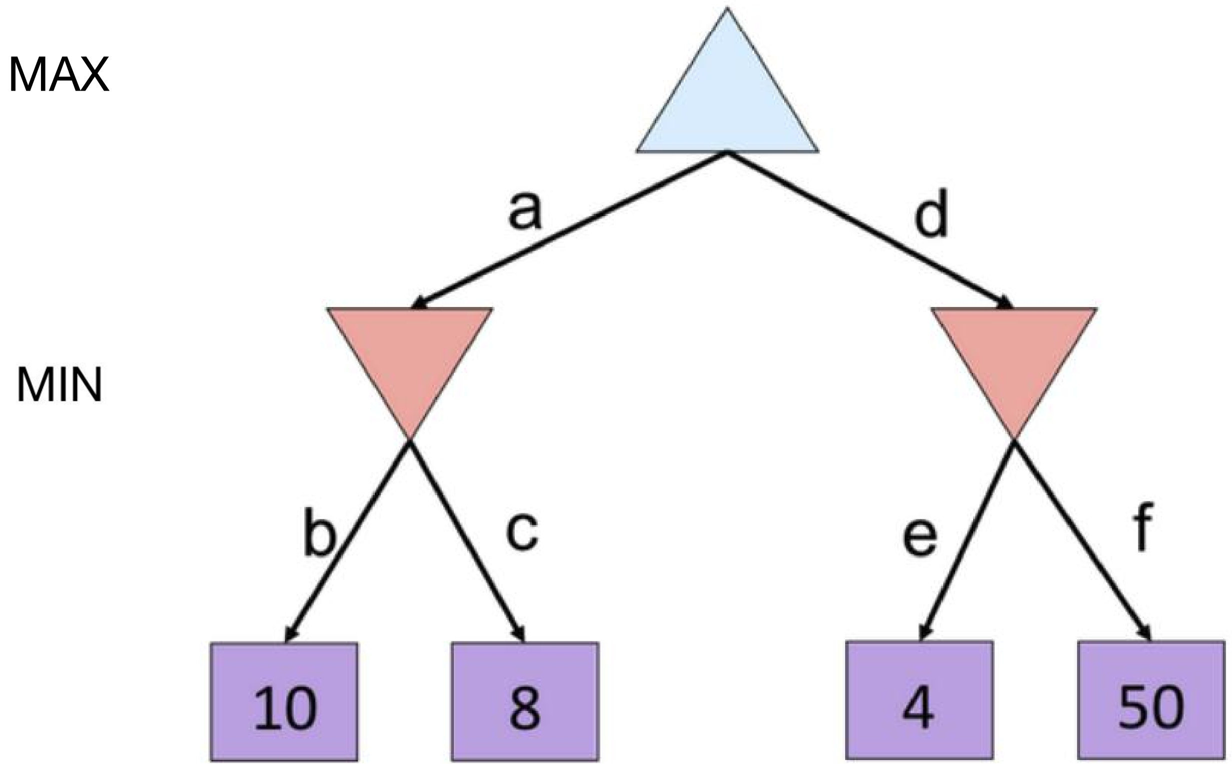
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Pruning Properties

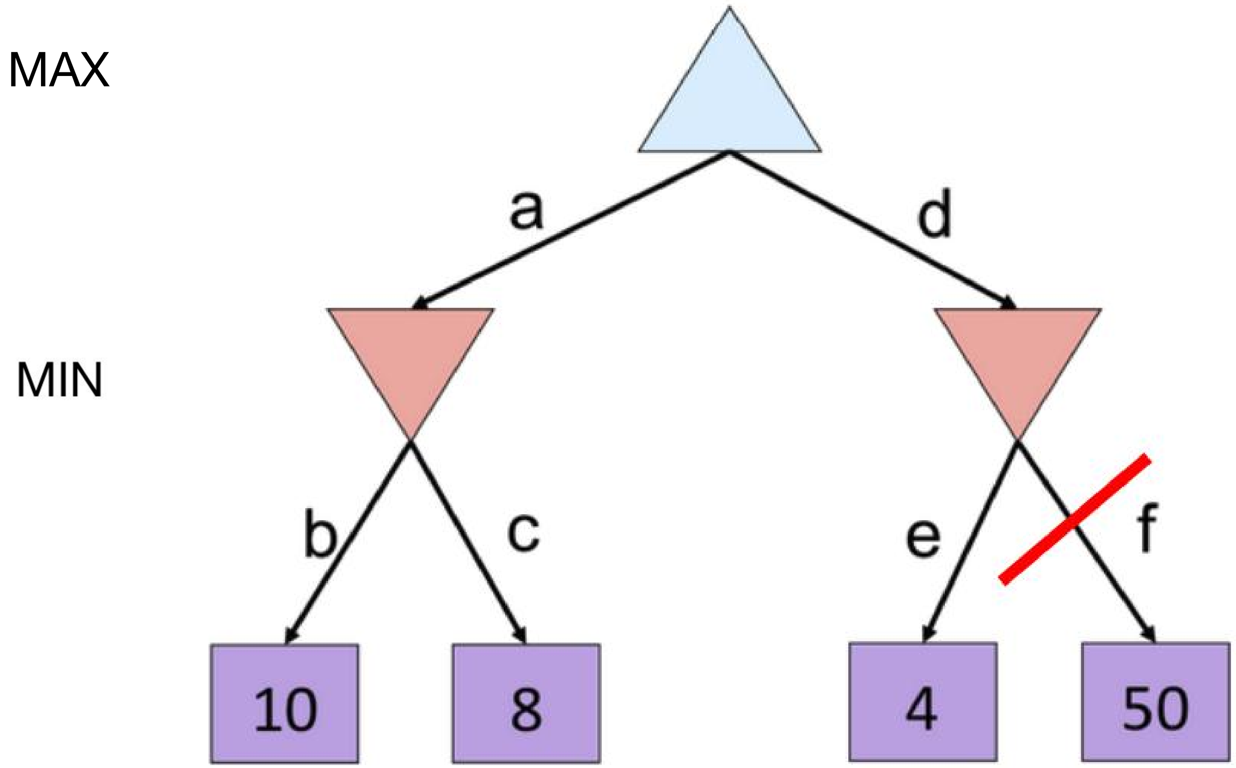
- This pruning has no effect on minimax value computed for the root!
- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…



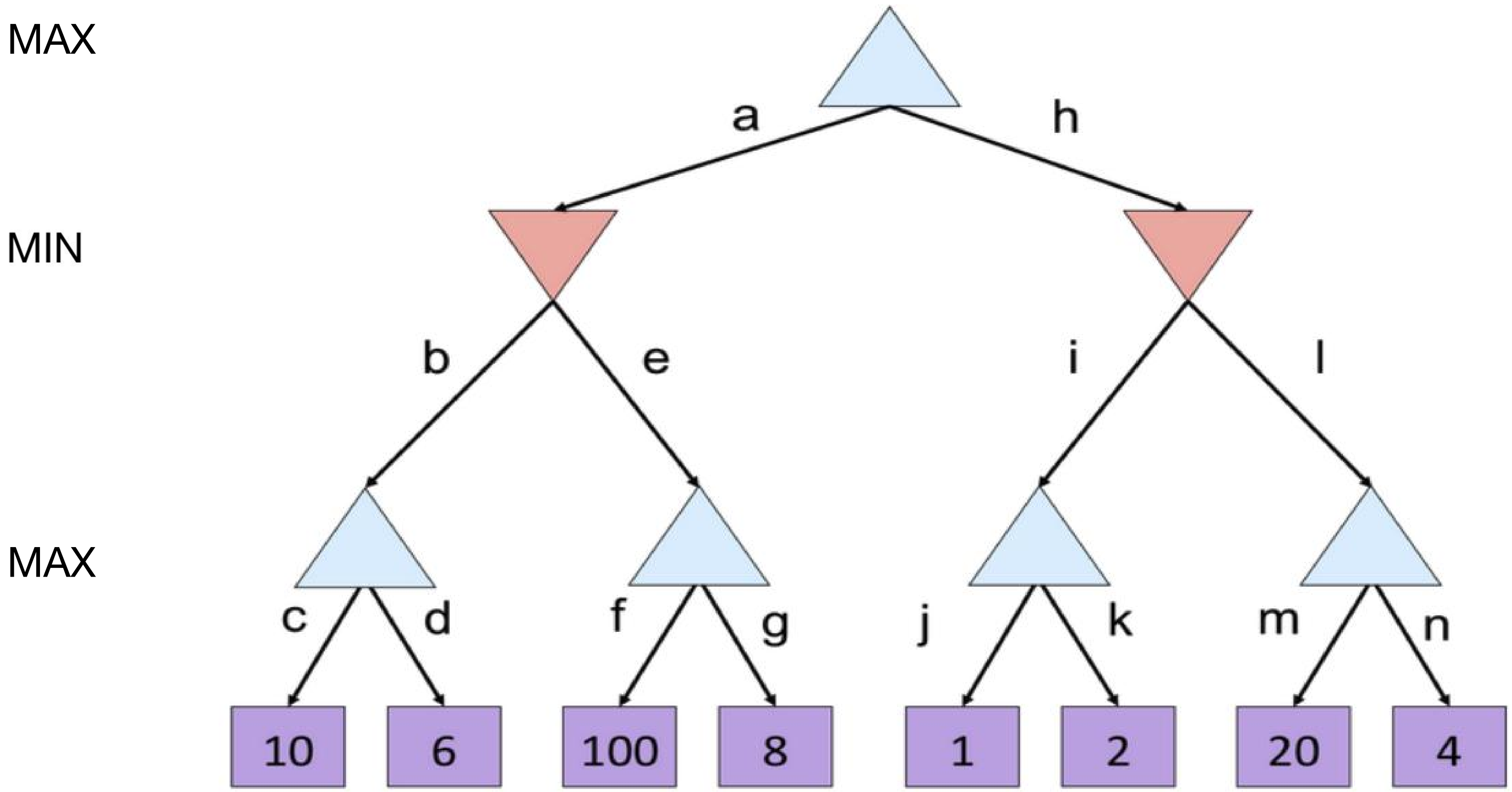
α - β Search Quiz



α - β Search Quiz



α - β Search Quiz



α - β Search Quiz