



# Adversarial Search

---

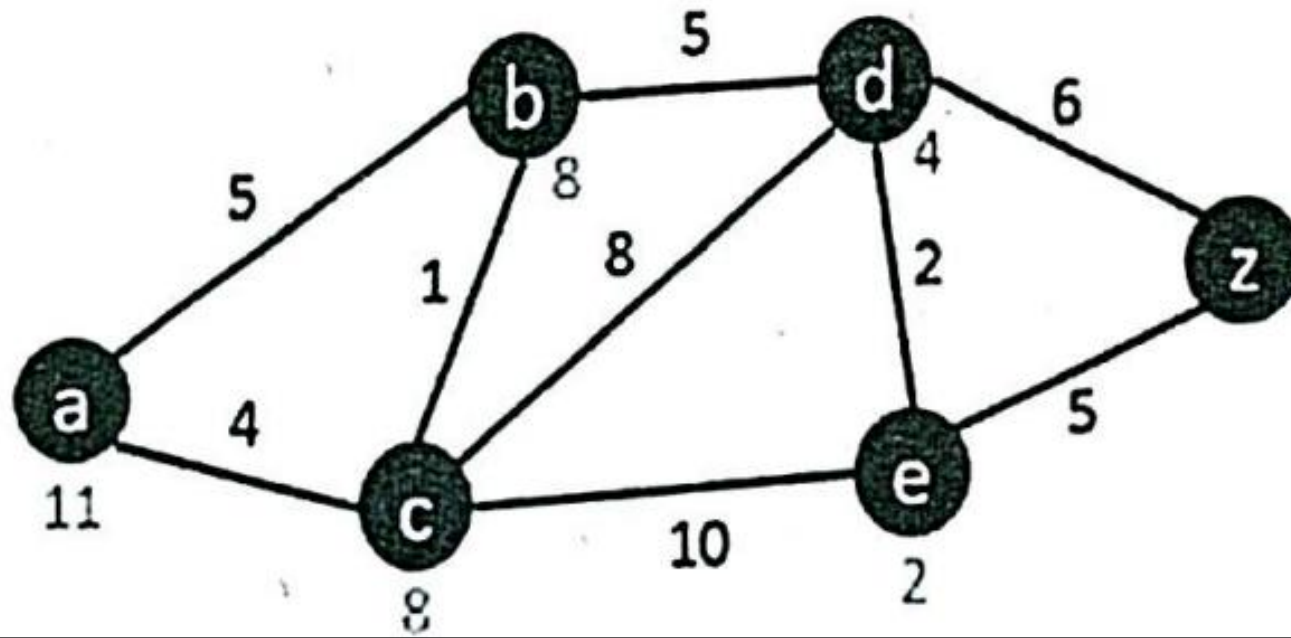
Vijay Kumar Meena  
Assistant Professor  
KIIT University

# Acknowledgement

- These slides are taken (as it is or with some modifications) from various resources including -
  - Artificial Intelligence: A Modern Approach (AIMA) by Russell and Norvig.
  - Slides of Prof. Mausam (IITD)
  - Slides of Prof. Rohan Paul (IITD)
  - Slides of Prof. Brian Yu (Harvard University)

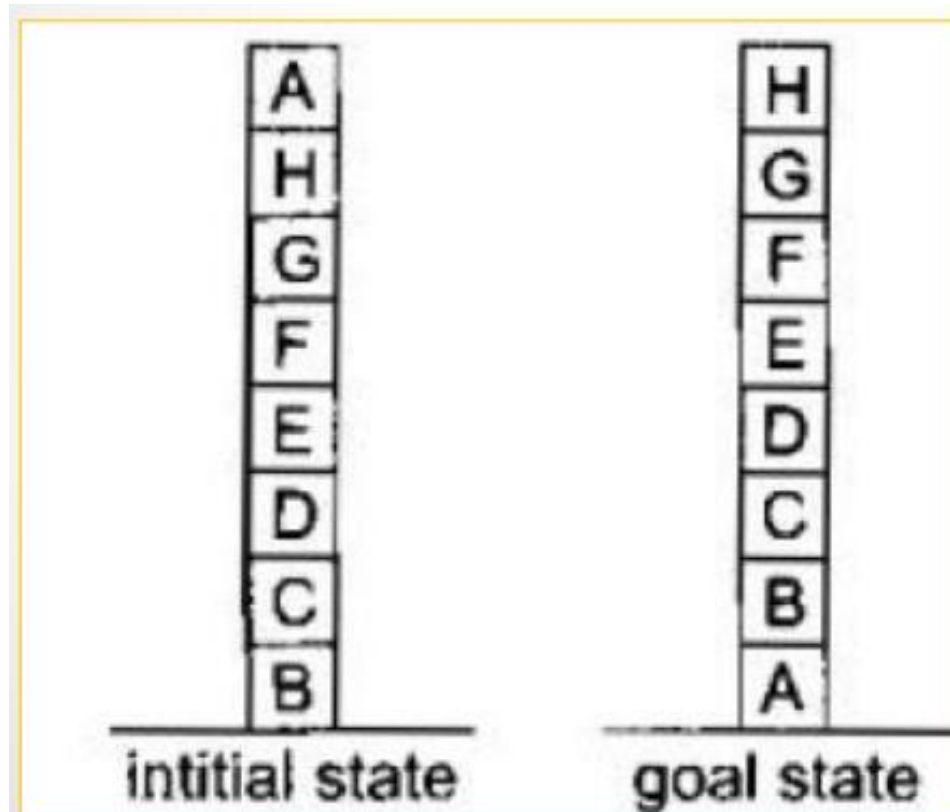
## Exercises

- What's the difference between rationality and omniscience?
- For following state space graph, where **a** is start node and **z** is goal node, which of the following algorithms will generate less number of nodes -
  - BFS
  - DFS
  - UCS
  - Greedy BFS
  - A\*



## Example: Hill-Climbing

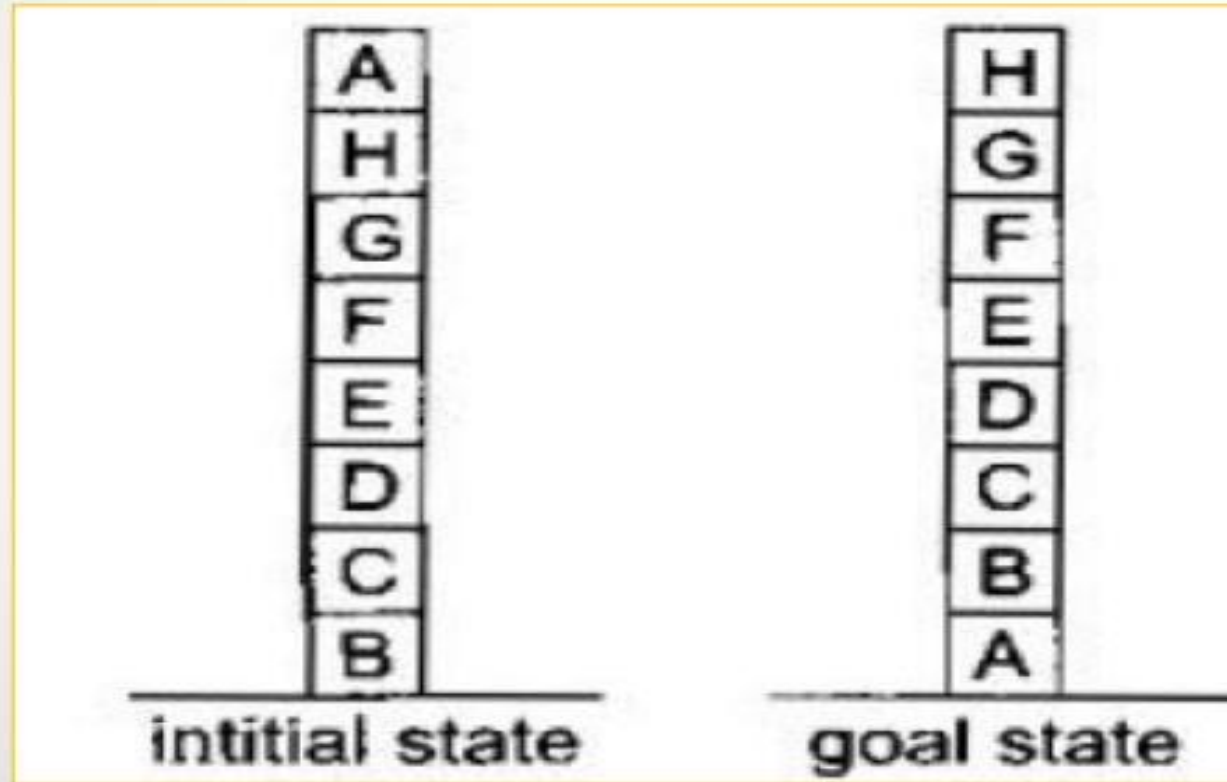
- Apply the hill climbing algorithm to solve the blocks world problem shown in Figure.



# Examples

## Example 1

Apply the hill climbing algorithm to solve the blocks world problem shown in Figure.



## Solution

To use the hill climbing algorithm we need an evaluation function or a heuristic function.

We consider the following evaluation function:

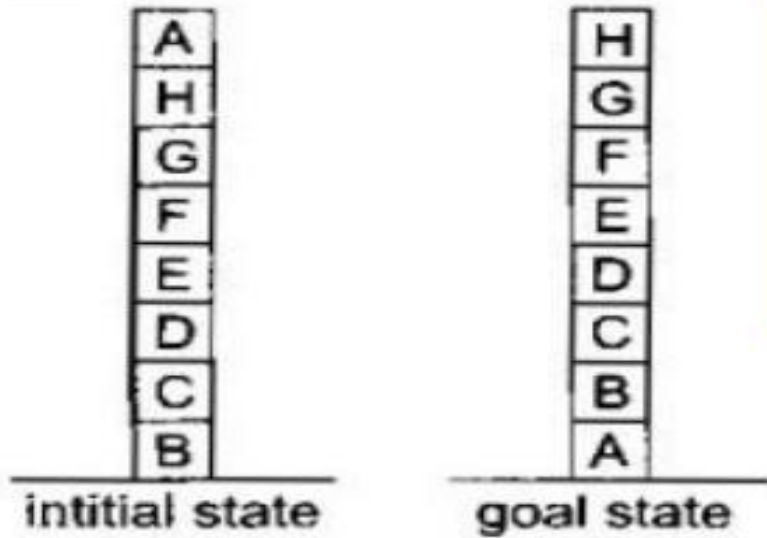
**$h(n)$  = Add one point for every block that is resting on the thing it is supposed to be resting on. Subtract one point for every block that is sitting on the wrong thing.**



# Examples

## Example 1

We call "initial state" "State 0" and "goal state" "Goal". Then considering the blocks A, B, C, D, E, F, G, H in that order we have



$$h(\text{State 0}) = -1 - 1 + 1 + 1 + 1 + 1 + 1 + 1 \\ = 4$$

$$h(\text{Goal}) = +1 + 1 + 1 + 1 + 1 + 1 + 1 + 1 \\ = 8$$

There is only one move from State 0, namely, to move block A to the table. This produces the State 1 with a score of 6:

$$h(\text{State 1}) = 1 - 1 + 1 + 1 + 1 + 1 + 1 + 1 = 6.$$

**$h(n)$  = Add one point for every block that is resting on the thing it is supposed to be resting on. Subtract one point for every block that is sitting on the wrong thing.**

# Examples

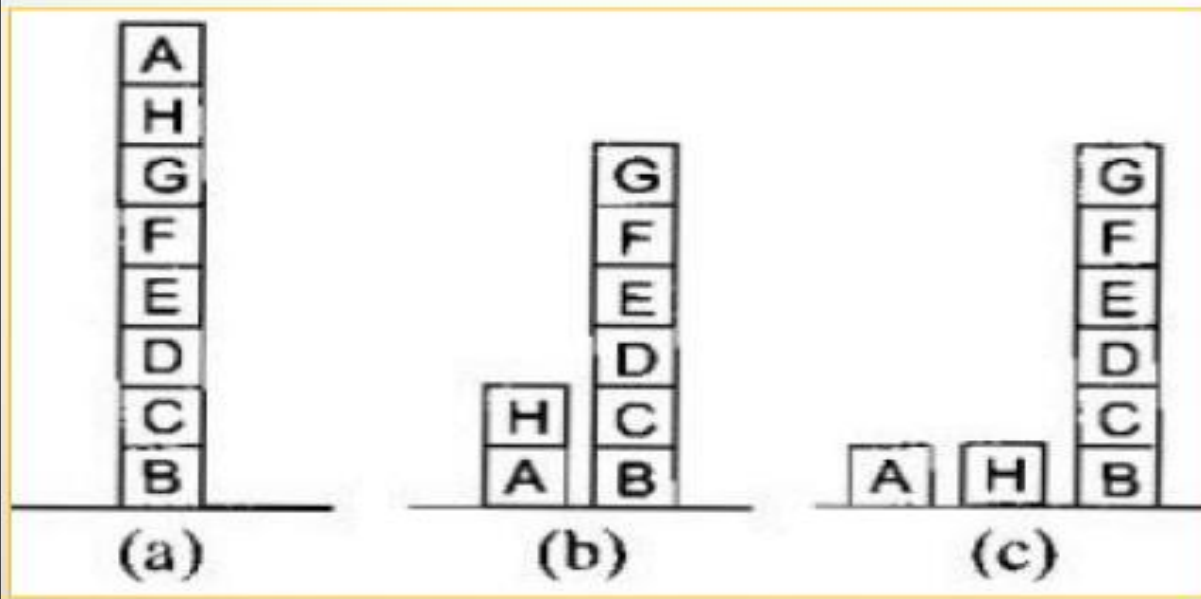
## Example 1

There are three possible moves from State 1 as shown in Figure. We denote these three states State 2(a), State 2(b) and State 2(c). We also have

$$h(\text{State 2(a)}) = -1 - 1 + 1 + 1 + 1 + 1 + 1 + 1 = 4$$

$$h(\text{State 2(b)}) = +1 - 1 + 1 + 1 + 1 + 1 + 1 - 1 = 4$$

$$h(\text{State 2(c)}) = +1 - 1 + 1 + 1 + 1 + 1 + 1 - 1 = 4$$



Hill climbing will halt because all these states have lower scores than the current state. The process has reached a local maximum that is not a global maximum. We have reached such a situation because of the particular choice of the heuristic function. A different choice of heuristic function may not produce such a situation.

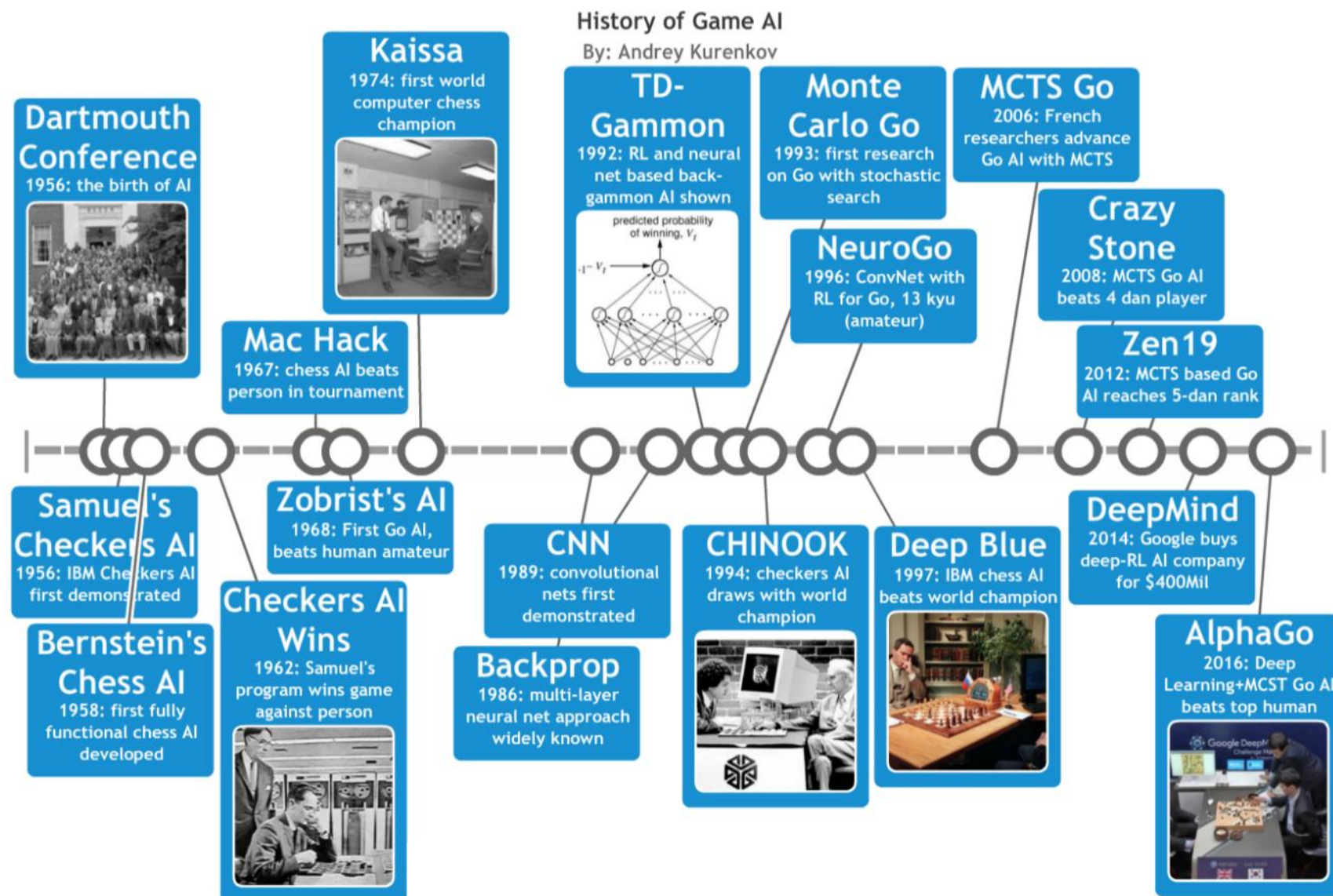


## Adversarial Search Problems

- So far we have been studying about environments where only one agent was active. How would things change when there are multiple agents?
- Each agent needs to consider the actions of other agents and how they affect its own performance measure.
- The unpredictability of these other agents can introduce possible contingencies into the agent's problem solving process.
- There might be different types of agents like **cooperative multi-agents** or **competitive multi-agents**.
- Having to deal with competitive multi-agents which have conflicting goals give rise to different category of problems called **adversarial search problems**. Best example of these problems are **Games**.



# AI & Games



# Games vs search problems

## ➤ Unpredictable Opponent

- Search strategy of AI agent is not independent anymore, it's affected by other agent's actions.
- Need to specify a move for every possible reply by opponent.

## ➤ Time Limits

- Games have a time limit in which each player should make their move.
- It is unlikely to find best move within that time limit, we need some way of approximating so that agent can be best possible move found within that time limit.
- For example, chess has an average branching factor of about 35, and games often go to 50 moves by each player, so the search tree has about  $35^{100}$  or about  $10^{154}$  nodes.

## Games & AI

- In AI, games have a special format –
  - Deterministic
  - Two player - usually called MAX and MIN
  - Turn taking - MAX moves first, and then they take turns moving until the game is over.
  - Zero sum games – It means at the end of the game, utility values are equal and opposite for each player. Like At the end of the game, points are awarded to the winning player and equal penalties are given to the loser.

## Game as a Search Problem

- **So** - The **Initial state**, which specifies how the game is set up at the start.
- **Player(s)** - Defines which player has the move in a state.
- **Actions(s)** - Returns the set of legal moves in a state.
- **Result(s, a)** - The **Transition model**, which defines the result of a move.
- **Terminal-Test(s)** - A **terminal test**, which is true when the game is over and false otherwise. States where the game has ended are called **terminal states**.
- **Utility(s, p)** - A **utility function** (also called an objective function) defines the final numeric value for a game that ends in terminal state **s** for a player **p**.



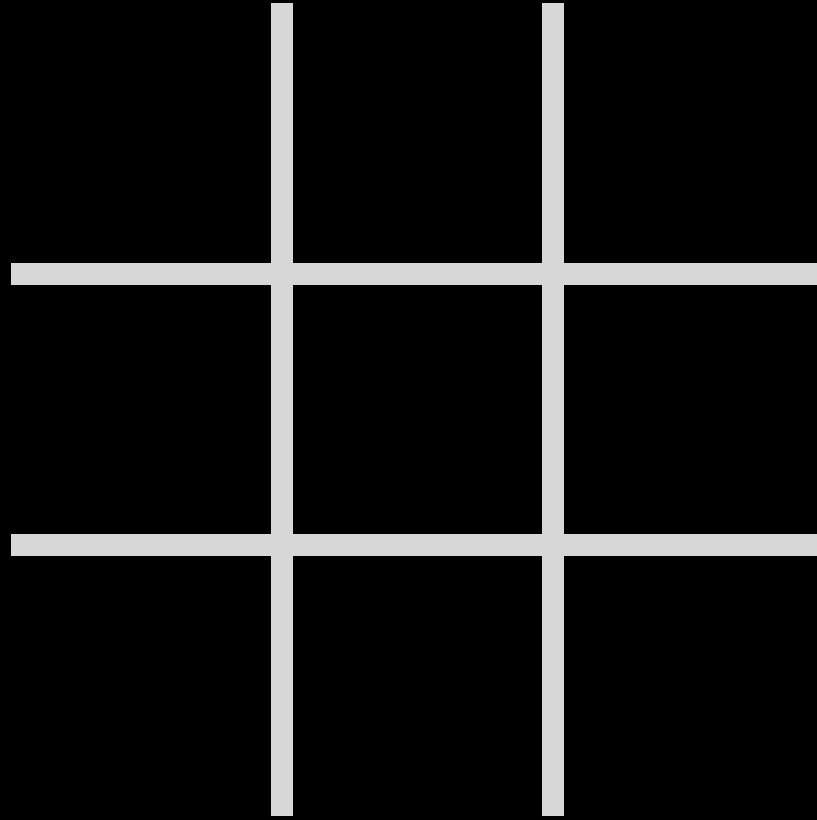
## Game as a Search Problem

- In chess, the outcome is a win, loss, or draw, with values  $+1$ ,  $0$ , or  $1/2$ . Some games have a wider variety of possible outcomes; the utility values in backgammon range from  $0$  to  $+192$ .
- A **zero-sum game** is defined as one where the total payoff (utility values) to all players is the same for every instance of the game.
- Chess is zero-sum because every game has payoff of either  $0 + 1$ ,  $1 + 0$  or  $1/2 + 1/2$ . “Constant-sum” would have been a better term, but zero-sum is traditional and makes sense if you imagine each player is charged an entry fee of  $1/2$ .

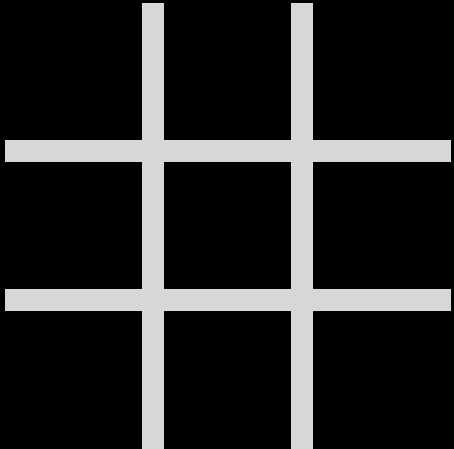
# Game as a Search problem

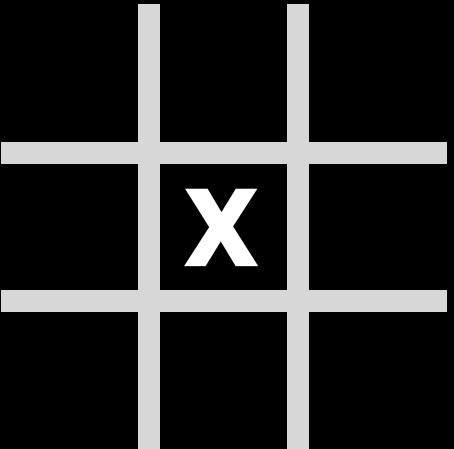
- $S_0$  : initial state
- $\text{PLAYER}(s)$  : returns which player to move in state  $s$
- $\text{ACTIONS}(s)$  : returns legal moves in state  $s$
- $\text{RESULT}(s, a)$  : returns state after action  $a$  taken in state  $s$
- $\text{TERMINAL}(s)$  : checks if state  $s$  is a terminal state. States where the game has ended are called terminal states.
- $\text{UTILITY}(s)$  : final numerical value for terminal state  $s$

# Initial State



PLAYER(*s*)

PLAYER() = **X**

PLAYER() = **O**



# ACTIONS( $s$ )

$$\text{ACTIONS}\left( \begin{array}{c|c|c} & \mathbf{x} & \mathbf{o} \\ \hline \mathbf{o} & \mathbf{x} & \mathbf{x} \\ \hline \mathbf{x} & & \mathbf{o} \end{array} \right) = \left\{ \begin{array}{c|c|c} \mathbf{o} & & \\ \hline \hline \hline \end{array}, \begin{array}{c|c|c} & & \\ \hline \hline \hline \mathbf{o} \end{array} \right\}$$

**RESULT(*s*, *a*)**

$$\text{RESULT}\left( \begin{array}{c|c|c} & \mathbf{x} & \mathbf{o} \\ \hline \mathbf{o} & \mathbf{x} & \mathbf{x} \\ \hline \mathbf{x} & & \mathbf{o} \end{array}, \begin{array}{c} \mathbf{o} \\ \hline \# \\ \hline \end{array} \right) = \begin{array}{c|c|c} \mathbf{o} & \mathbf{x} & \mathbf{o} \\ \hline \mathbf{o} & \mathbf{x} & \mathbf{x} \\ \hline \mathbf{x} & & \mathbf{o} \end{array}$$

# TERMINAL(*s*)

TERMINAL(

o		
o	x	
x	o	x

) = false

TERMINAL(

o		x
o	x	
x	o	x

) = true

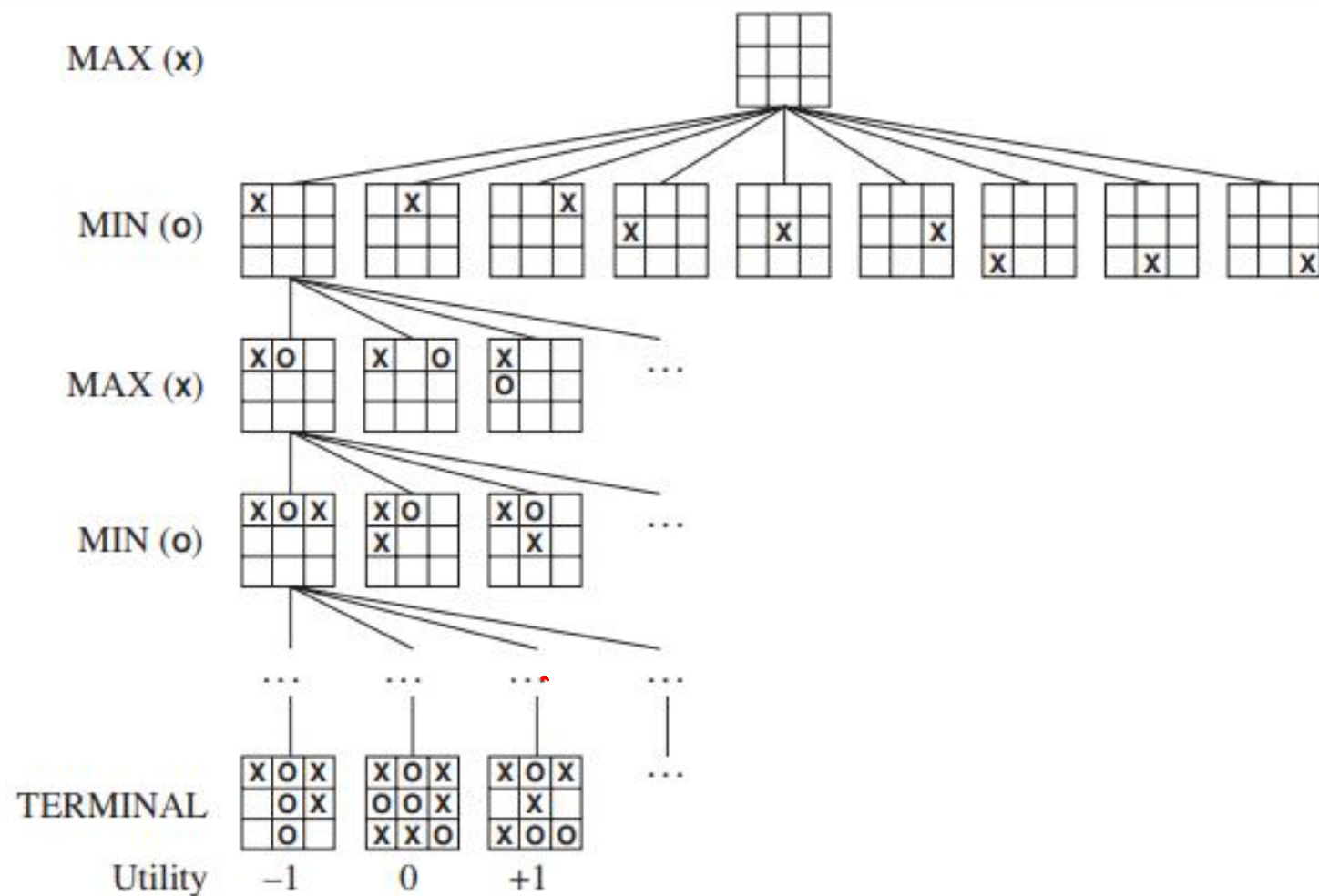
UTILITY(*s*)

$$\text{UTILITY}\left( \begin{array}{c|c|c} \mathbf{o} & & \mathbf{x} \\ \hline \mathbf{o} & \mathbf{x} & \\ \hline \mathbf{x} & \mathbf{o} & \mathbf{x} \end{array} \right) = 1$$

$$\text{UTILITY}\left( \begin{array}{c|c|c} \mathbf{o} & \mathbf{x} & \mathbf{x} \\ \hline \mathbf{x} & \mathbf{o} & \\ \hline \mathbf{o} & \mathbf{x} & \mathbf{o} \end{array} \right) = -1$$



# SEARCH TREE



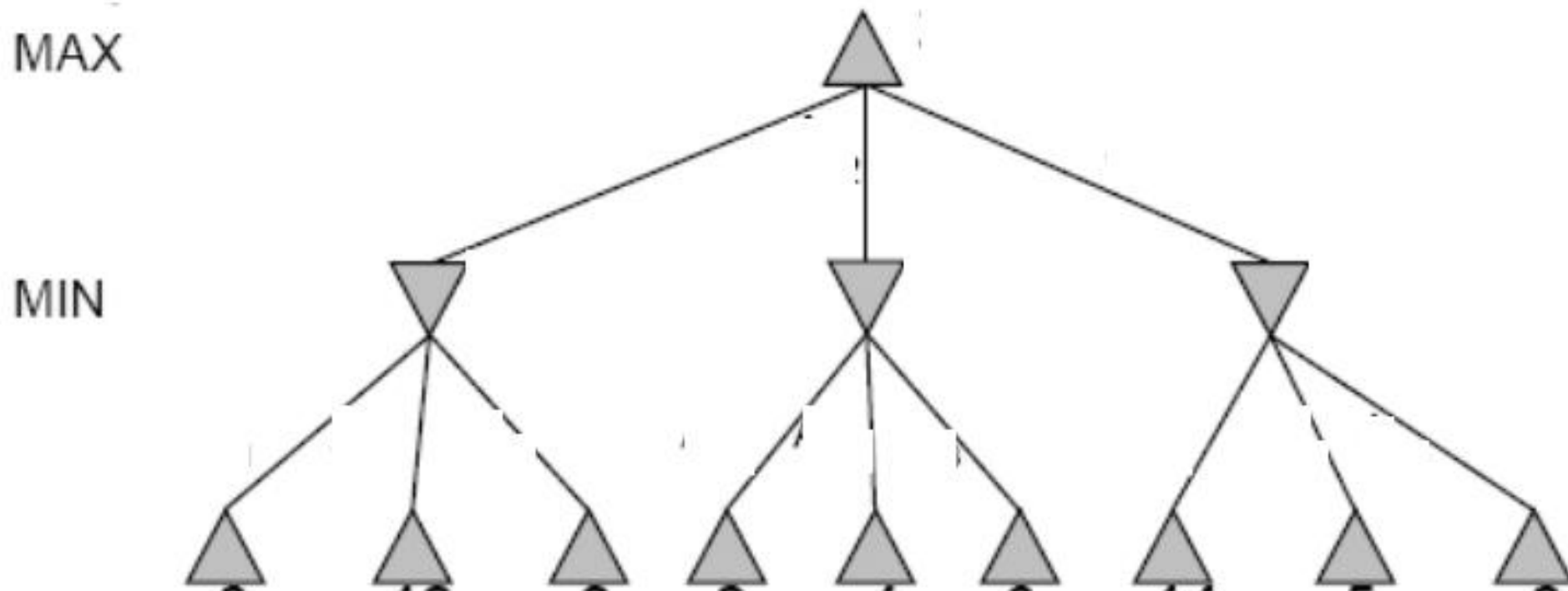
**Figure 5.1** A (partial) game tree for the game of tic-tac-toe. The top node is the initial state, and MAX moves first, placing an X in an empty square. We show part of the tree, giving alternating moves by MIN (O) and MAX (X), until we eventually reach terminal states, which can be assigned utilities according to the rules of the game.

## Optimal Decisions in Games

- In a normal search problem, the optimal solution would be a sequence of actions leading to a goal state – a terminal state that is a win.
- In adversarial search, MIN and MAX players take alternative turns. Therefore MAX player has to find move which maximizes the score for all actions taken by MIN player.

## Optimal Decisions in Game

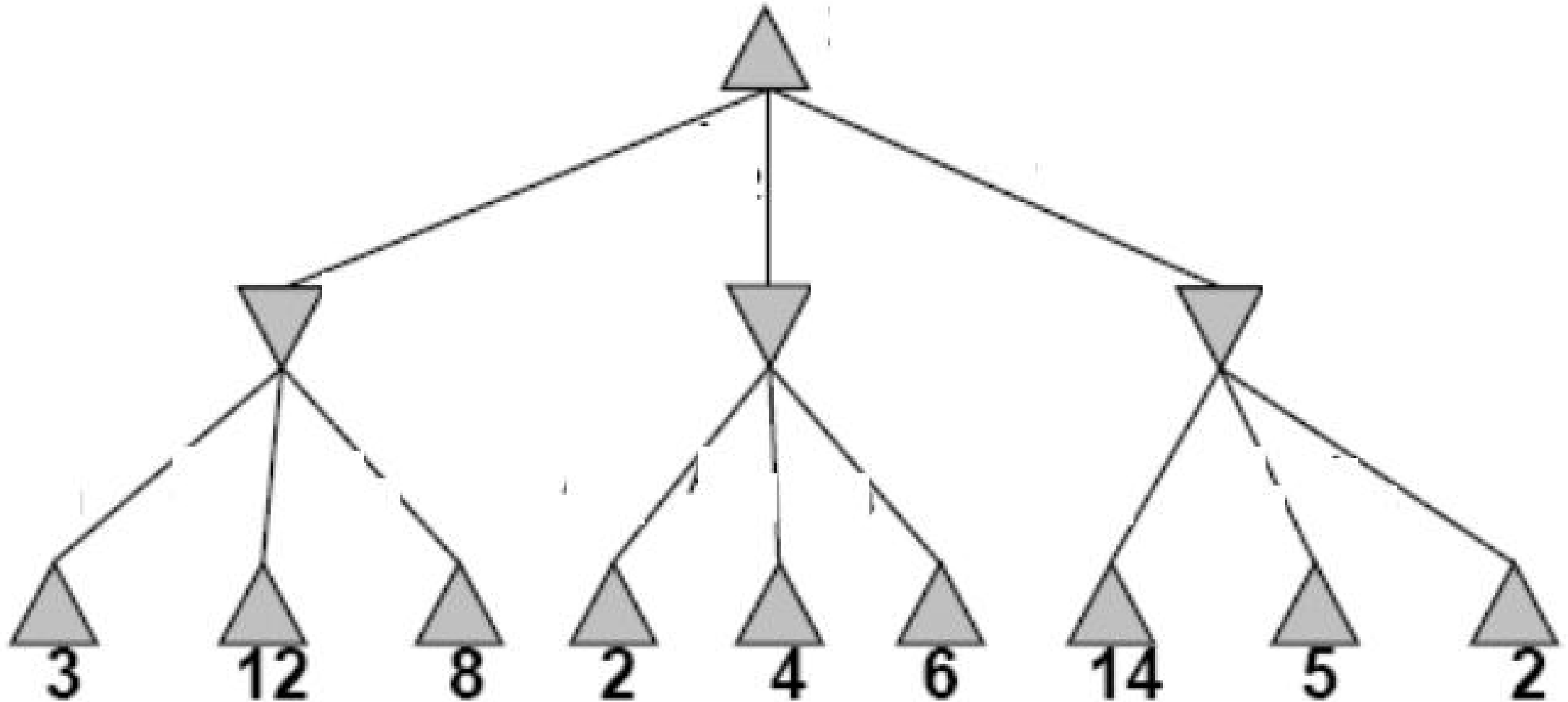
- Let's suppose you are given a search tree for a very simple game. Can you tell which action should be taken by MAX player and MIN player.



## Optimal Decisions in Game

MAX

MIN

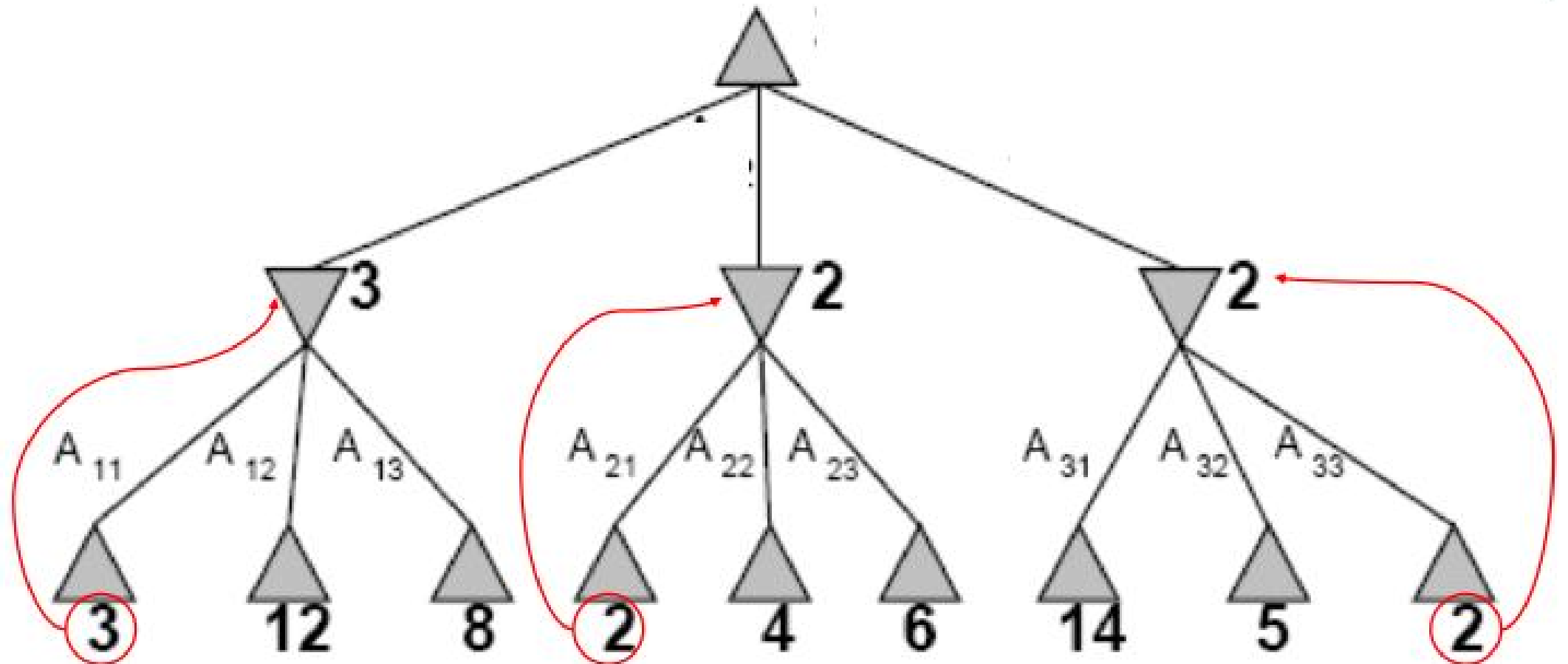




## Optimal Decisions in Game

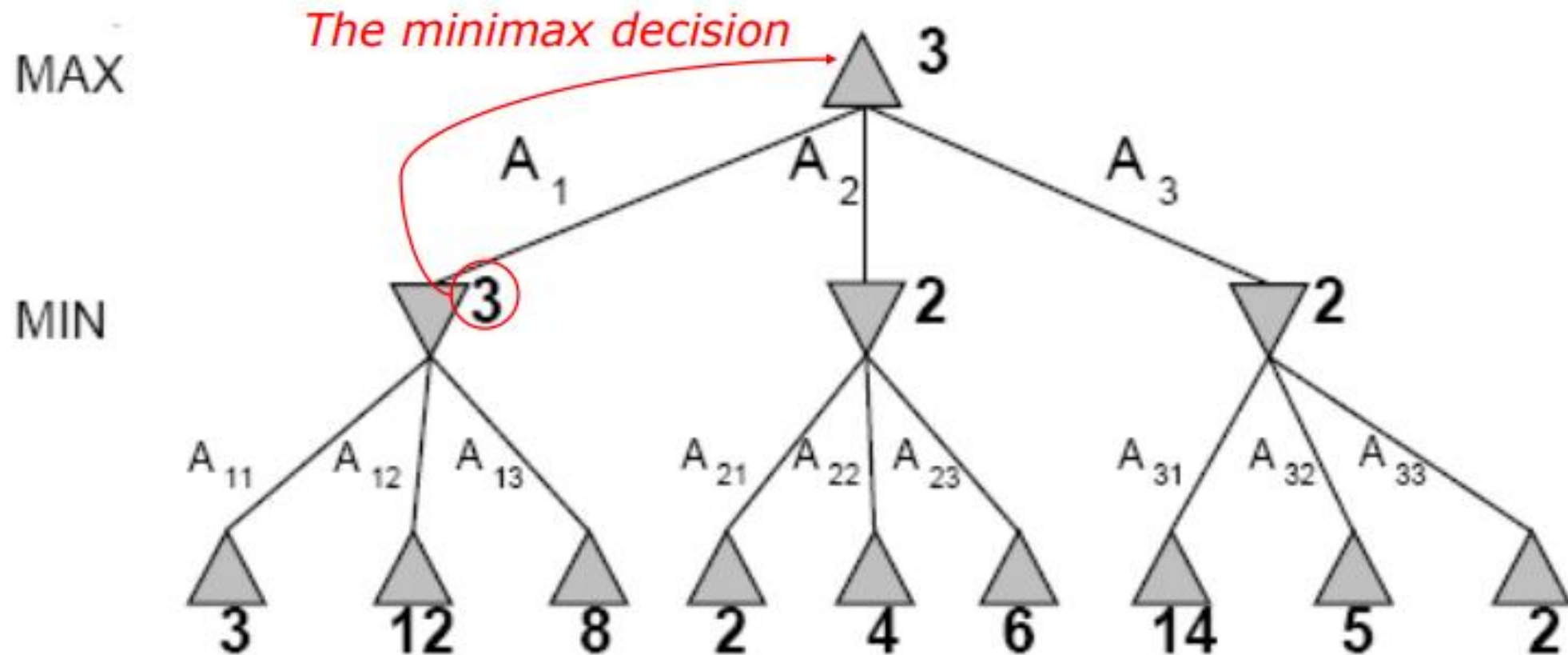
MAX

MIN

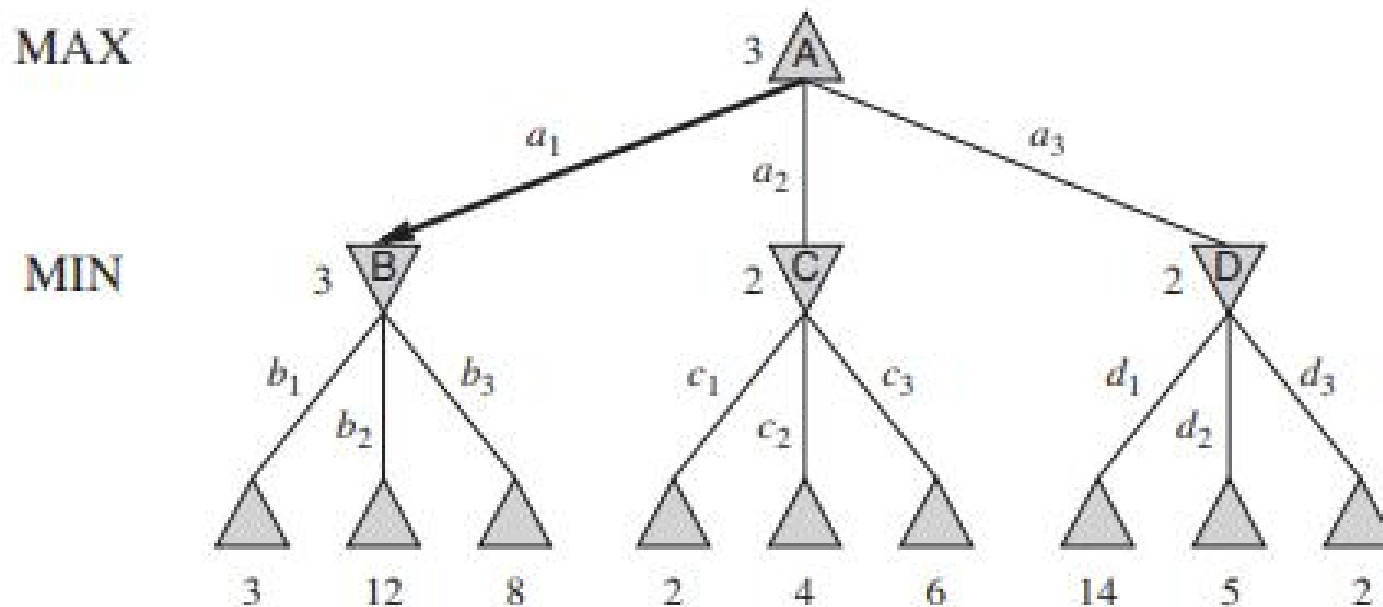


## Optimal Decisions in Game

- Minimax maximizes the utility for the worst-case outcome for max.



# OPTIMAL DECISIONS IN GAMES



**Figure 5.2** A two-ply game tree. The  $\triangle$  nodes are “MAX nodes,” in which it is MAX’s turn to move, and the  $\nabla$  nodes are “MIN nodes.” The terminal nodes show the utility values for MAX; the other nodes are labeled with their minimax values. MAX’s best move at the root is  $a_1$ , because it leads to the state with the highest minimax value, and MIN’s best reply is  $b_1$ , because it leads to the state with the lowest minimax value.

# MINIMAX SEARCH ALGORITHM

- Given a game tree, the optimal strategy can be determined from the **minimax value** of each node.
- The minimax value of a node is the utility of being in the corresponding state, *assuming that both players play optimally*. Minimax value of a terminal state is just the utility value.
- Given a choice, MAX prefers to move to a state of maximum value, whereas MIN prefers a state of minimum value.

$$\text{MINIMAX}(s) = \begin{cases} \text{UTILITY}(s) & \text{if } \text{TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{if } \text{PLAYER}(s) = \text{MIN} \end{cases}$$

# GAME TREE FOR TIC-TAC-TOE

computer's  
turn

opponent's  
turn

computer's  
turn

opponent's  
turn

leaf nodes  
are evaluated

MAX (X)

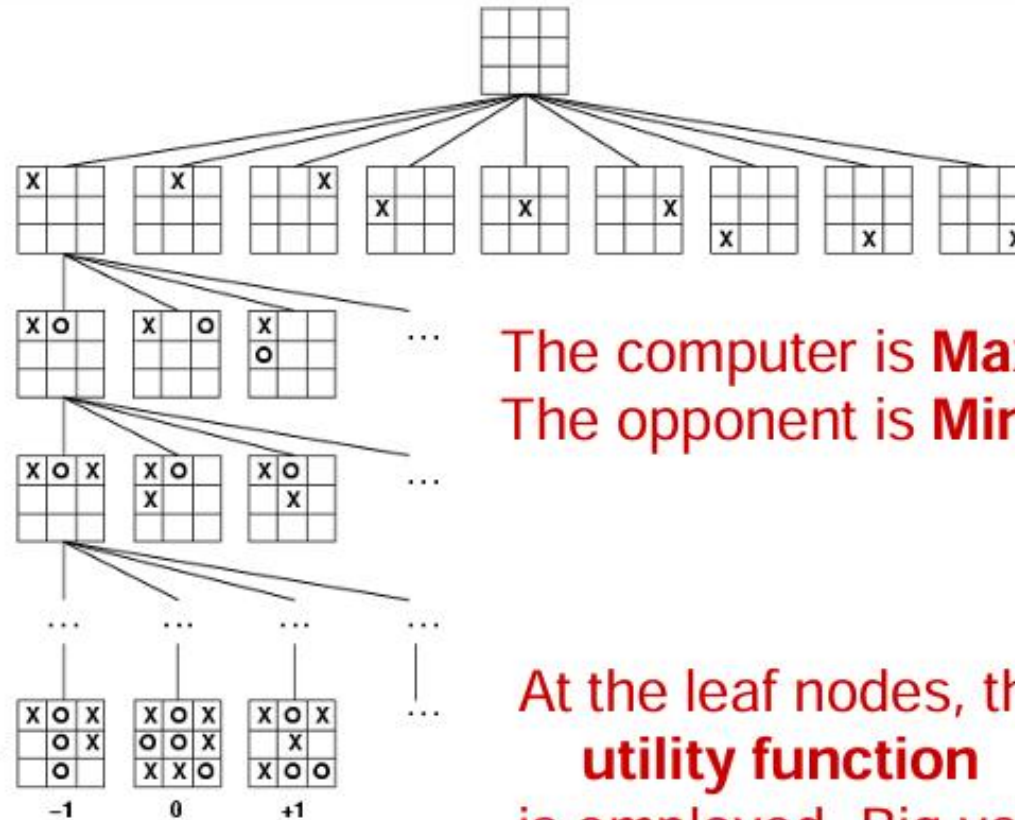
MIN (O)

MAX (X)

MIN (O)

TERMINAL

Utility



The computer is **Max**.  
The opponent is **Min**.

At the leaf nodes, the  
**utility function**  
is employed. Big value  
means good, small is bad.

## MINI-MAX TERMINOLOGY

- **Move** – a move by both players
- **Ply** – a half move i.e. decision of a player
- **Utility Function** – The function applied to leaf nodes to get utility values of nodes
- **Backed-up value**
  - **Of a max position** – the value of its largest successor
  - **Of a min position** – the value of its smallest successor
- **Minimax Strategy** – Search down several levels; at the bottom level apply the utility function, back-up values all the way up to the root node, and that node selects the move.

# MINIMAX SEARCH ALGORITHM

```
function MINIMAX-DECISION(state) returns an action  
  return  $\arg \max_{a \in \text{ACTIONS}(s)} \text{MIN-VALUE}(\text{RESULT}(\text{state}, a))$ 
```

---

```
function MAX-VALUE(state) returns a utility value  
  if TERMINAL-TEST(state) then return UTILITY(state)  
   $v \leftarrow -\infty$   
  for each a in ACTIONS(state) do  
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a)))$   
  return v
```

---

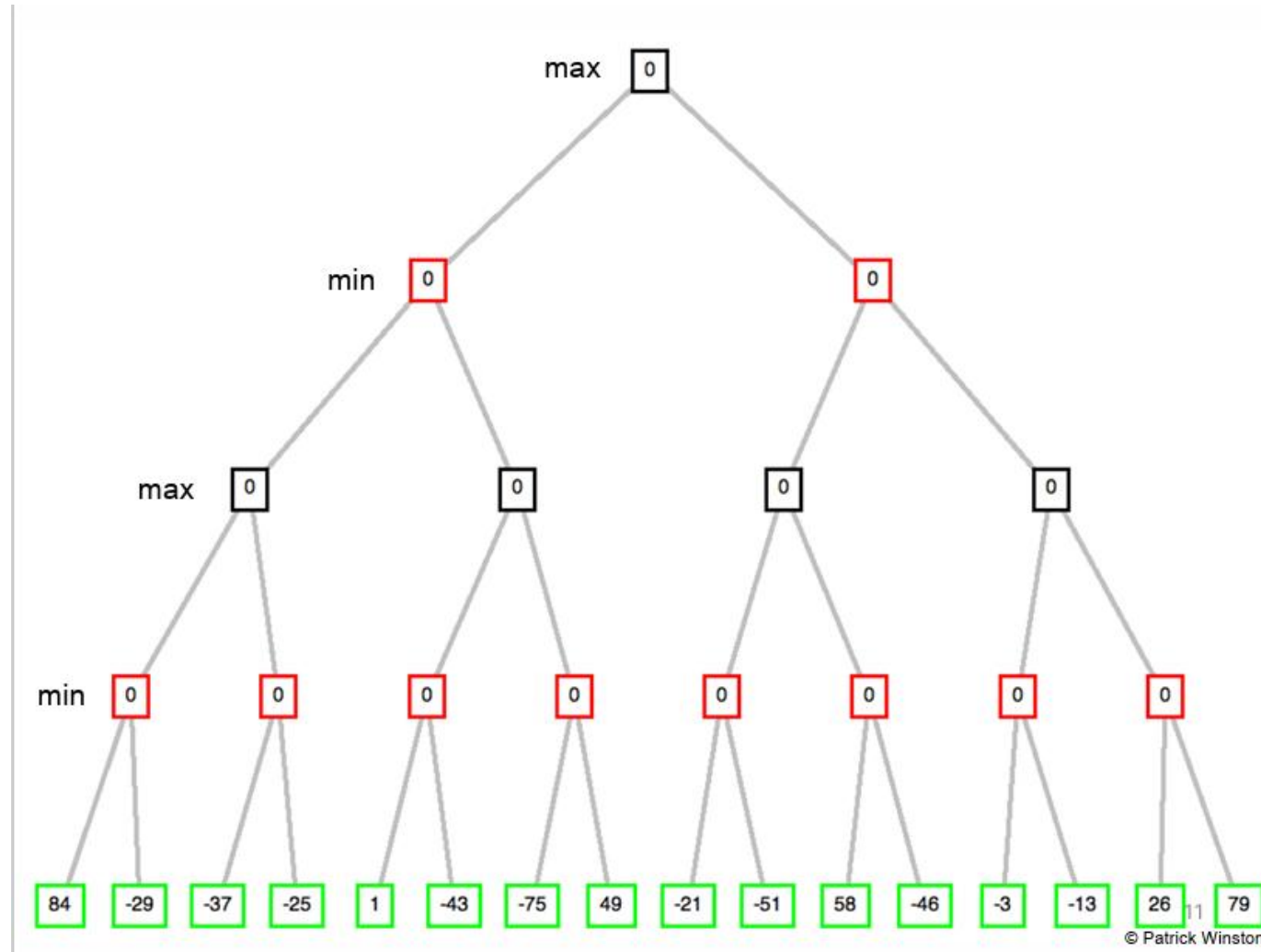
```
function MIN-VALUE(state) returns a utility value  
  if TERMINAL-TEST(state) then return UTILITY(state)  
   $v \leftarrow \infty$   
  for each a in ACTIONS(state) do  
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a)))$   
  return v
```

---

**Figure 5.3** An algorithm for calculating minimax decisions. It returns the action corresponding to the best possible move, that is, the move that leads to the outcome with the best utility, under the assumption that the opponent plays to minimize utility. The functions MAX-VALUE and MIN-VALUE go through the whole game tree, all the way to the leaves, to determine the backed-up value of a state. The notation  $\arg \max_{a \in S} f(a)$  computes the element *a* of set *S* that has the maximum value of *f(a)*.



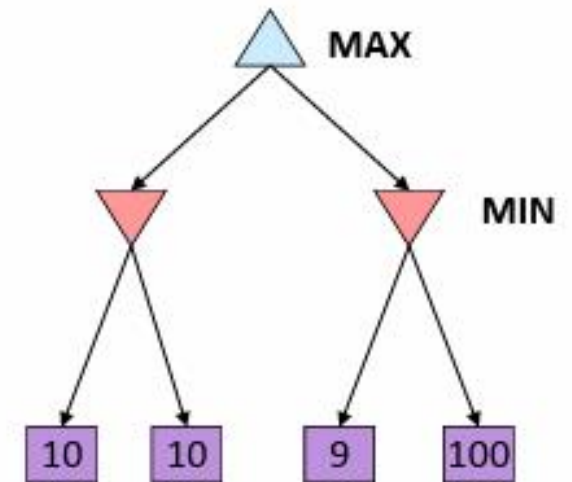
# MINIMAX SEARCH ALGORITHM



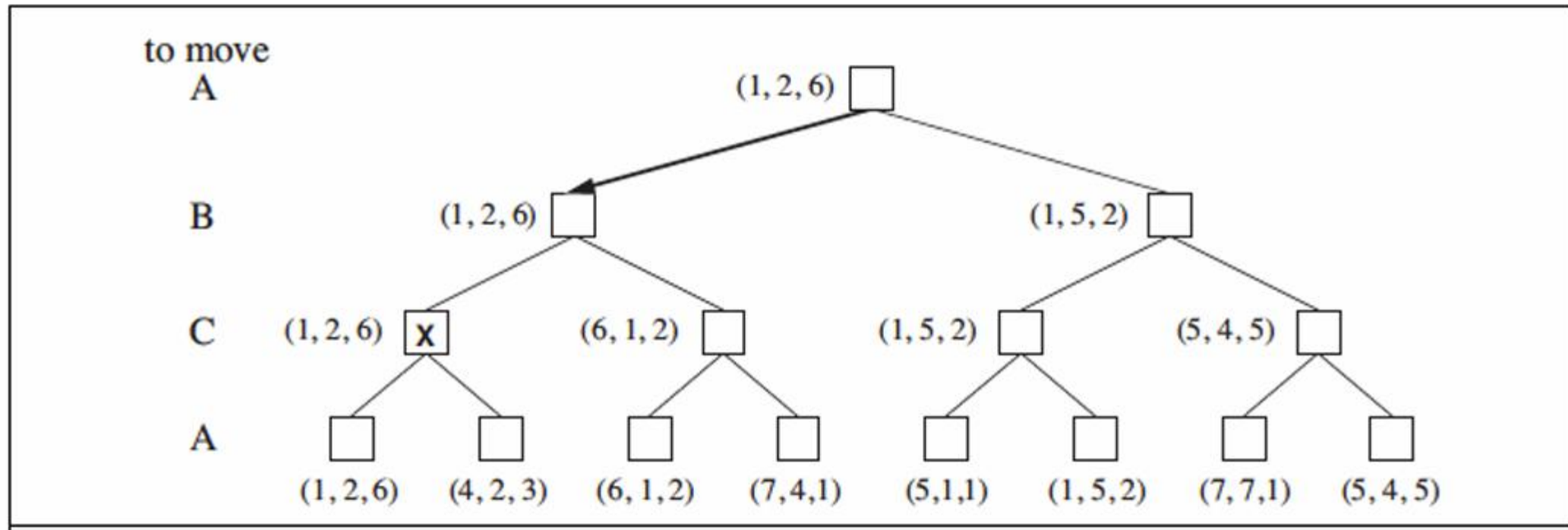


# MINIMAX SEARCH PROPERTIES

- Complete?
  - Yes (if tree is finite)
- Optimal?
  - Yes (against an optimal opponent)
  - No (does not exploit opponent weakness against suboptimal opponent)
- Time complexity?
  - $O(b^m)$
- Space complexity?
  - $O(bm)$  (depth-first exploration)



# MINIMAX FOR MULTIPLAYER



- Terminals have utility tuples
- Node values are also utility tuples
- Each player maximizes its own component

# Minimax: Recap

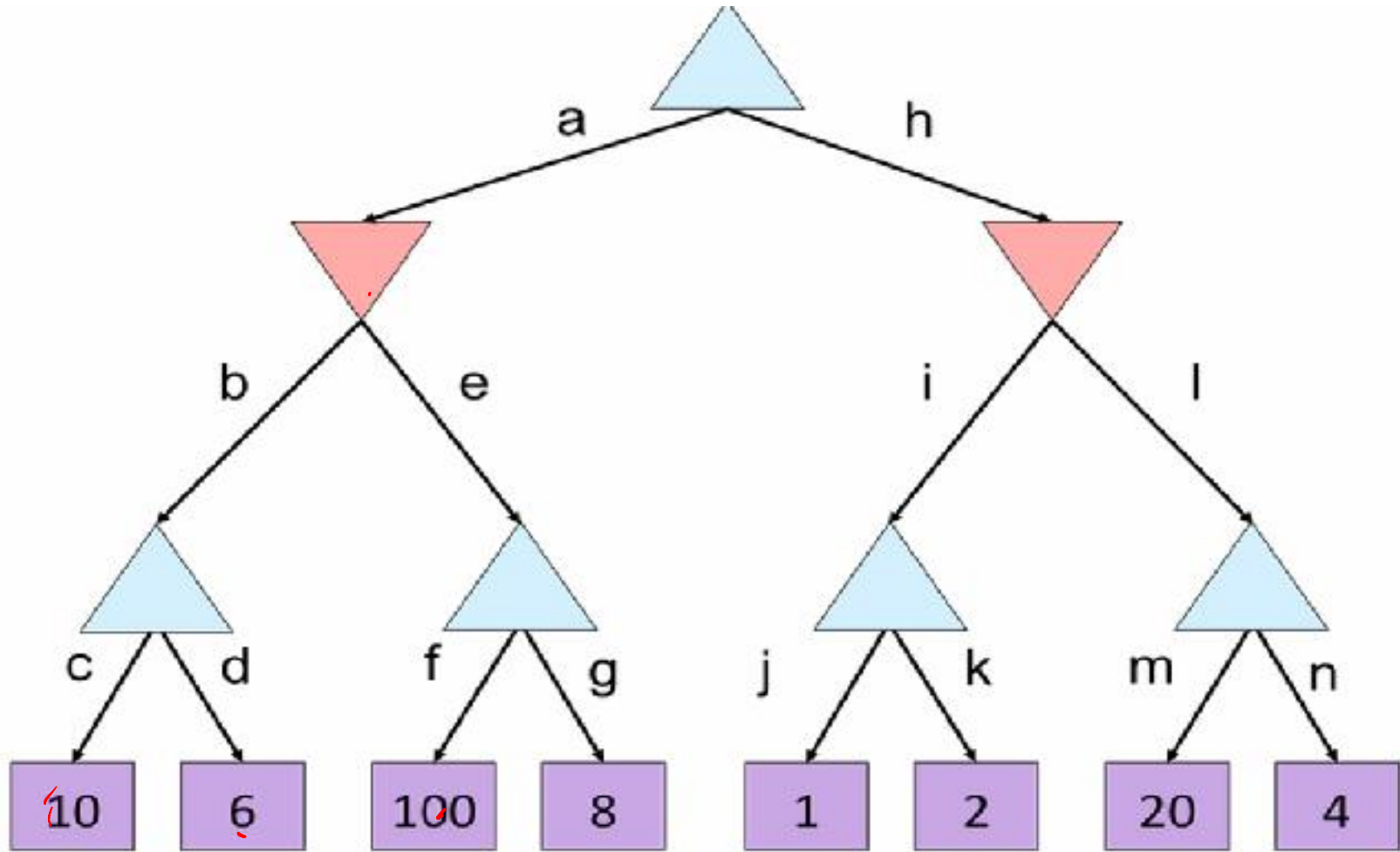
True or False for Minimax?

- Uses Breadth First Search traversal
- Every weakness of MIN can only improve the result for MAX.
- Must search till the end of the game.
- Multiple adversaries can move together.
- Utility values are provided only for internal nodes.
- Is incomplete.
- Space complexity is polynomial.

# Minimax for Chess

- Chess:
  - branching factor  $b \approx 35$
  - game length  $m \approx 100$
  - search space  $b^m \approx 35^{100} \approx 10^{154}$
- Requires growing the tree till the terminal nodes.
- Not feasible in practice for a game like Chess.

# ALPHA-BETA PRUNING



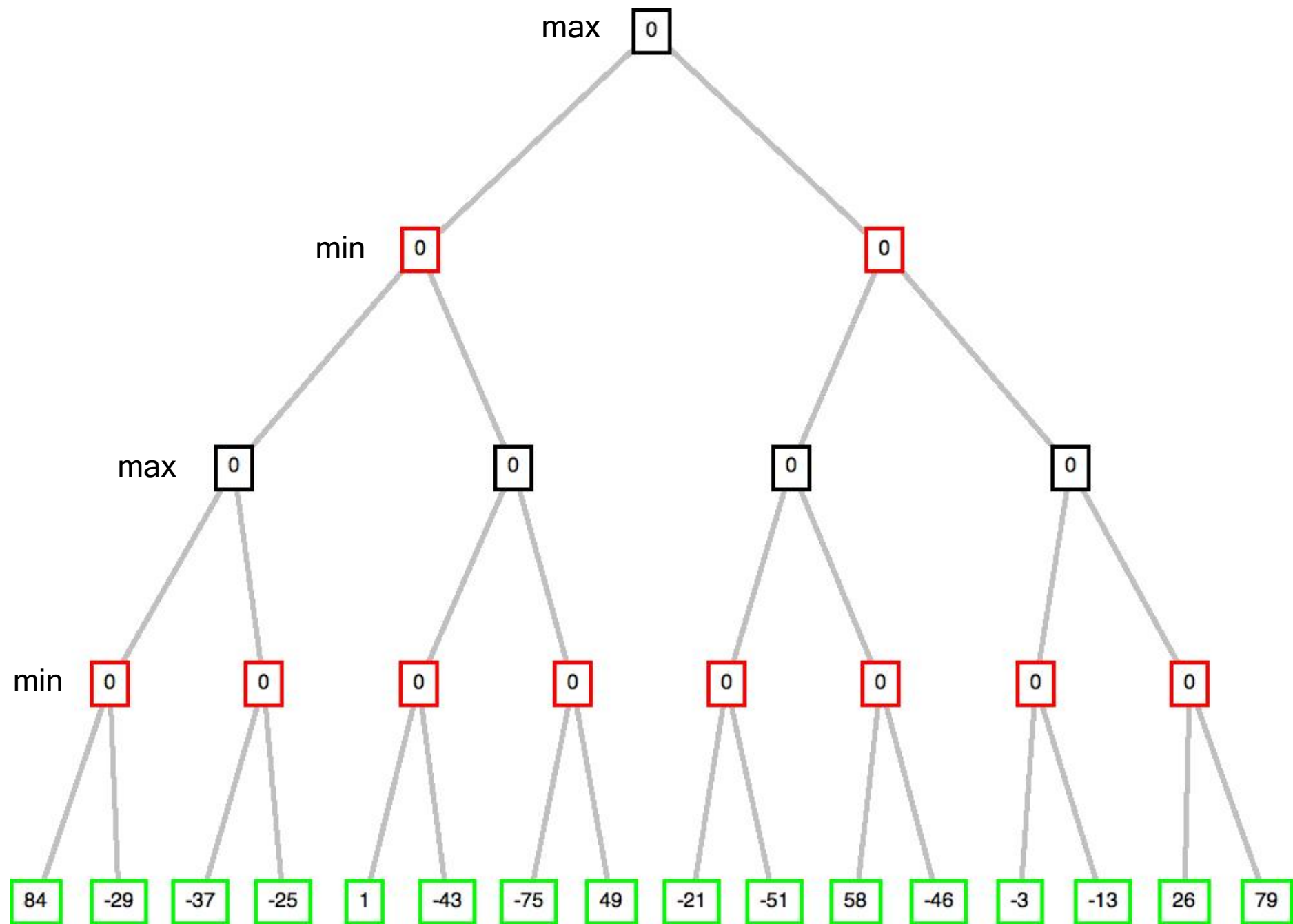
## ALPHA-BETA PRUNING

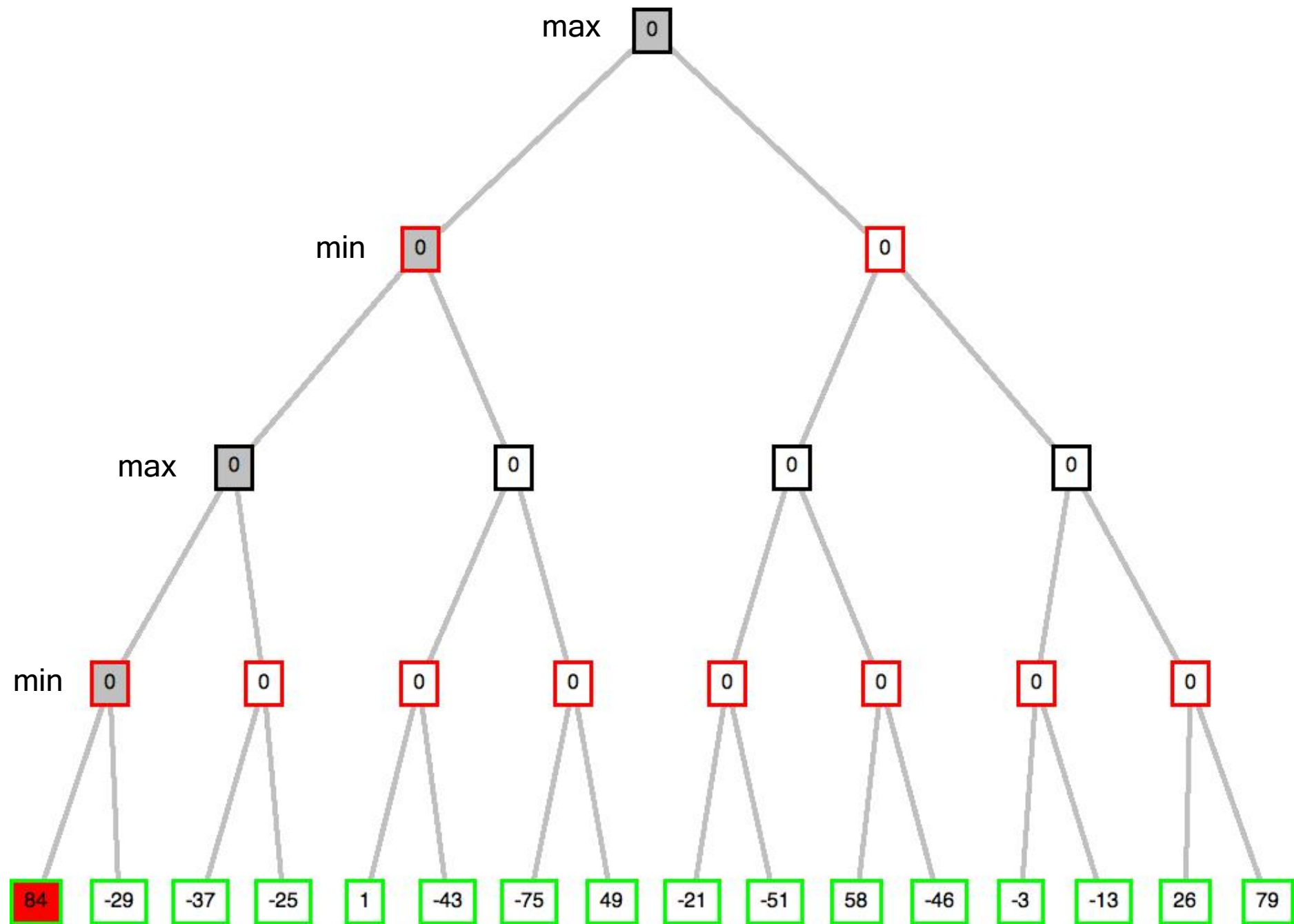
- The alpha-beta procedure can speed up a depth-first minimax search.
- Alpha: a lower bound on the value that a max node may ultimately be assigned

$$v \geq \alpha$$

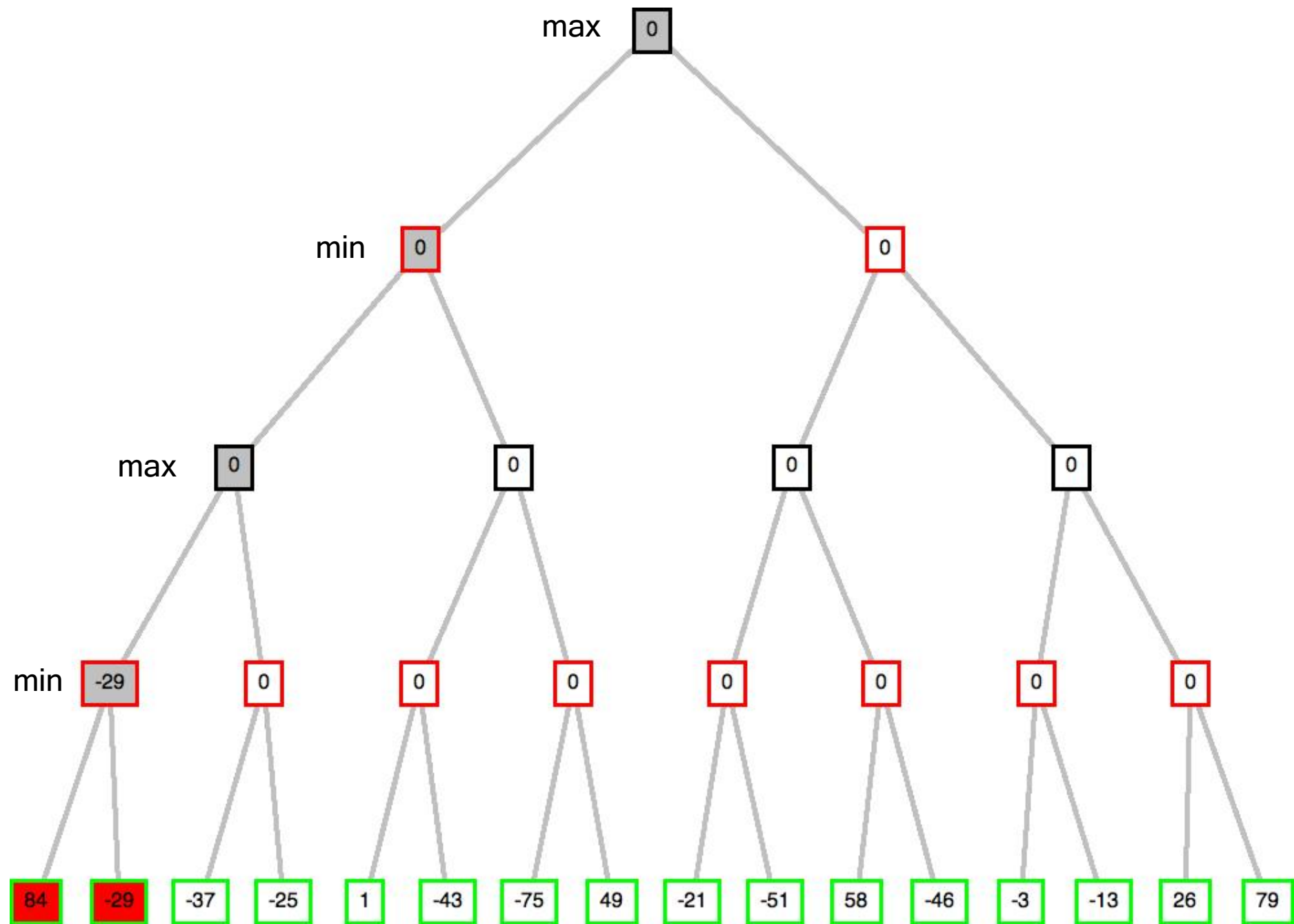
- Beta: an upper bound on the value that a minimizing node may ultimately be assigned

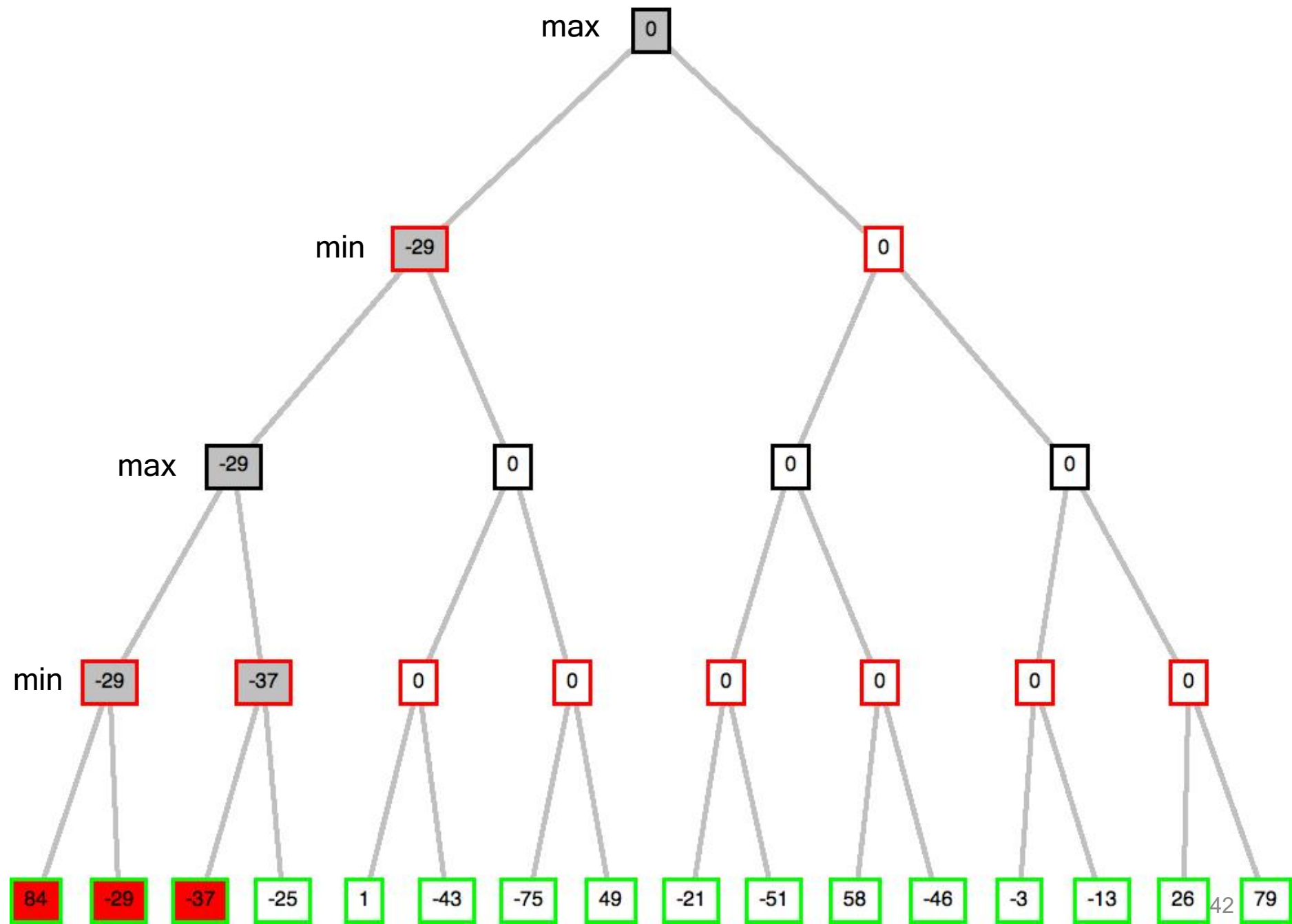
$$v \leq \beta$$

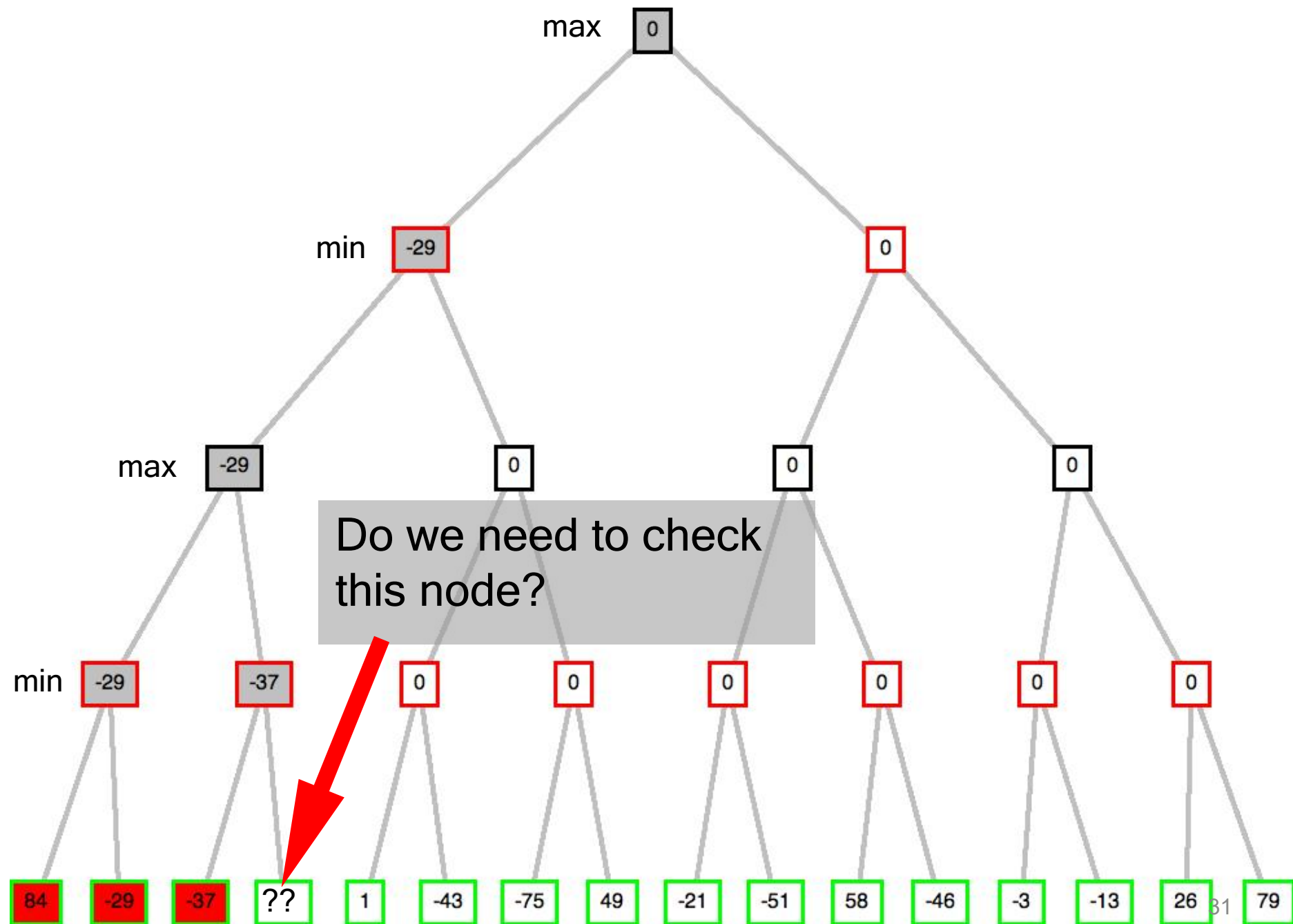


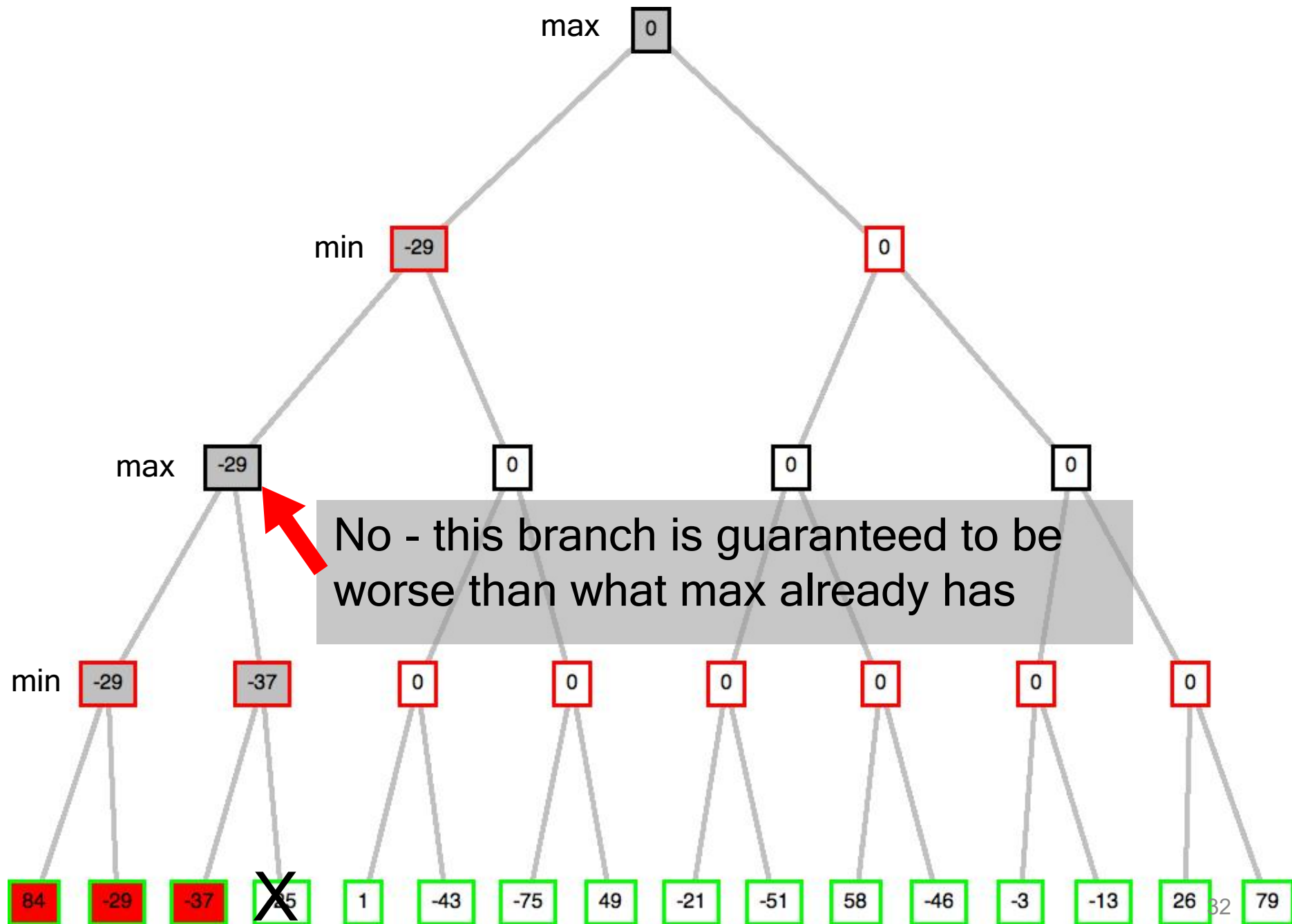












# ALPHA-BETA PRUNING

```
function ALPHA-BETA-SEARCH(state) returns an action  
   $v \leftarrow \text{MAX-VALUE}(\text{state}, -\infty, +\infty)$   
  return the action in  $\text{ACTIONS}(\text{state})$  with value  $v$ 
```

---

```
function MAX-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value  
  if  $\text{TERMINAL-TEST}(\text{state})$  then return  $\text{UTILITY}(\text{state})$   
   $v \leftarrow -\infty$   
  for each  $a$  in  $\text{ACTIONS}(\text{state})$  do  
     $v \leftarrow \text{MAX}(v, \text{MIN-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$   
    if  $v \geq \beta$  then return  $v$   
     $\alpha \leftarrow \text{MAX}(\alpha, v)$   
  return  $v$ 
```

---

```
function MIN-VALUE(state,  $\alpha$ ,  $\beta$ ) returns a utility value  
  if  $\text{TERMINAL-TEST}(\text{state})$  then return  $\text{UTILITY}(\text{state})$   
   $v \leftarrow +\infty$   
  for each  $a$  in  $\text{ACTIONS}(\text{state})$  do  
     $v \leftarrow \text{MIN}(v, \text{MAX-VALUE}(\text{RESULT}(s, a), \alpha, \beta))$   
    if  $v \leq \alpha$  then return  $v$   
     $\beta \leftarrow \text{MIN}(\beta, v)$   
  return  $v$ 
```

**Figure 5.7** The alpha-beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain  $\alpha$  and  $\beta$  (and the bookkeeping to pass these parameters along).

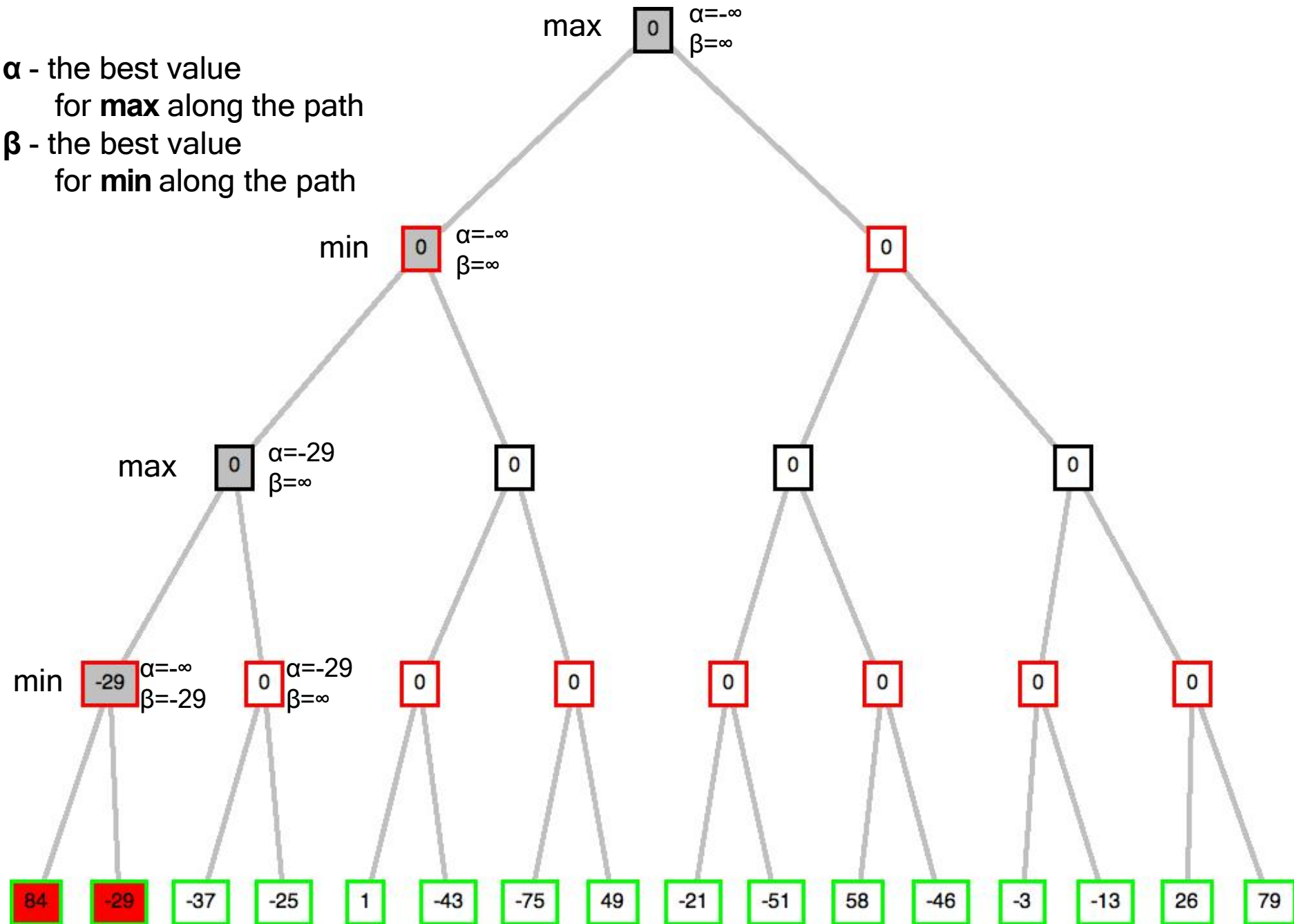
$\alpha$  - the best value for **max** along the path  
 $\beta$  - the best value for **min** along the path

```

graph TD
    Root["max 0 α=-∞ β=∞"] --> L1L["min 0 α=-∞ β=∞"]
    Root --> L1R["min 0"]
    L1L --> L2L1["max 0 α=-∞ β=∞"]
    L1L --> L2L2["max 0"]
    L1R --> L2R1["max 0"]
    L1R --> L2R2["max 0"]
    L2L1 --> L3L1["min 0 α=-∞ β=84"]
    L2L1 --> L3L2["min 0"]
    L2L2 --> L3L3["min 0"]
    L2L2 --> L3L4["min 0"]
    L2R1 --> L3R1["min 0"]
    L2R1 --> L3R2["min 0"]
    L2R2 --> L3R3["min 0"]
    L2R2 --> L3R4["min 0"]
    L3L1 --> L4L1["84"]
    L3L1 --> L4L2["-29"]
    L3L2 --> L4L3["-37"]
    L3L2 --> L4L4["-25"]
    L3L3 --> L4L5["1"]
    L3L3 --> L4L6["-43"]
    L3L4 --> L4L7["-75"]
    L3L4 --> L4L8["49"]
    L3R1 --> L4R1["-21"]
    L3R1 --> L4R2["-51"]
    L3R2 --> L4R3["58"]
    L3R2 --> L4R4["-46"]
    L3R3 --> L4R5["-3"]
    L3R3 --> L4R6["-13"]
    L3R4 --> L4R7["26"]
    L3R4 --> L4R8["79"]
  
```

$\alpha$  - the best value  
for **max** along the path

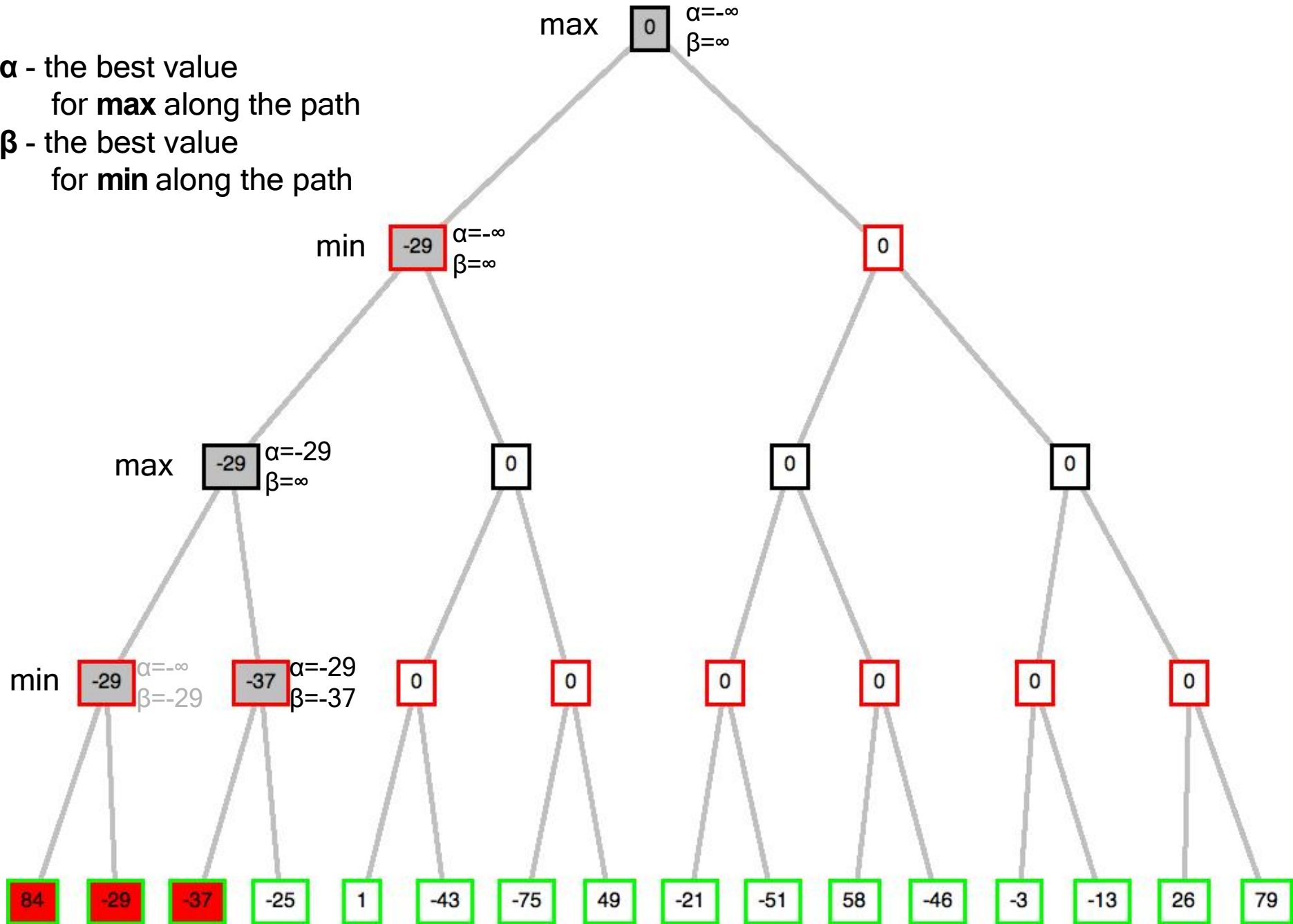
$\beta$  - the best value  
for **min** along the path





$\alpha$  - the best value  
for **max** along the path

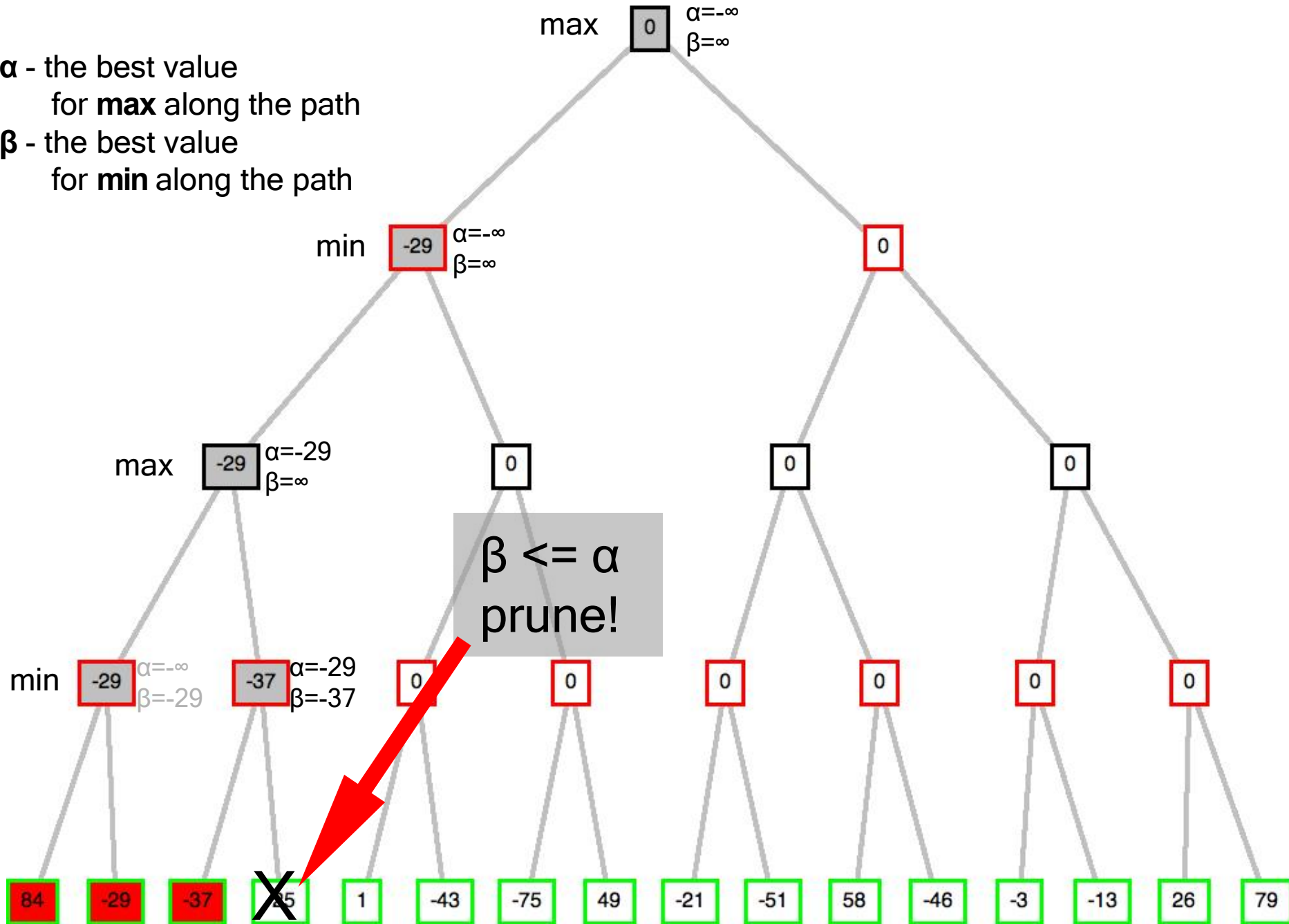
$\beta$  - the best value  
for **min** along the path





**$\alpha$**  - the best value  
for **max** along the path

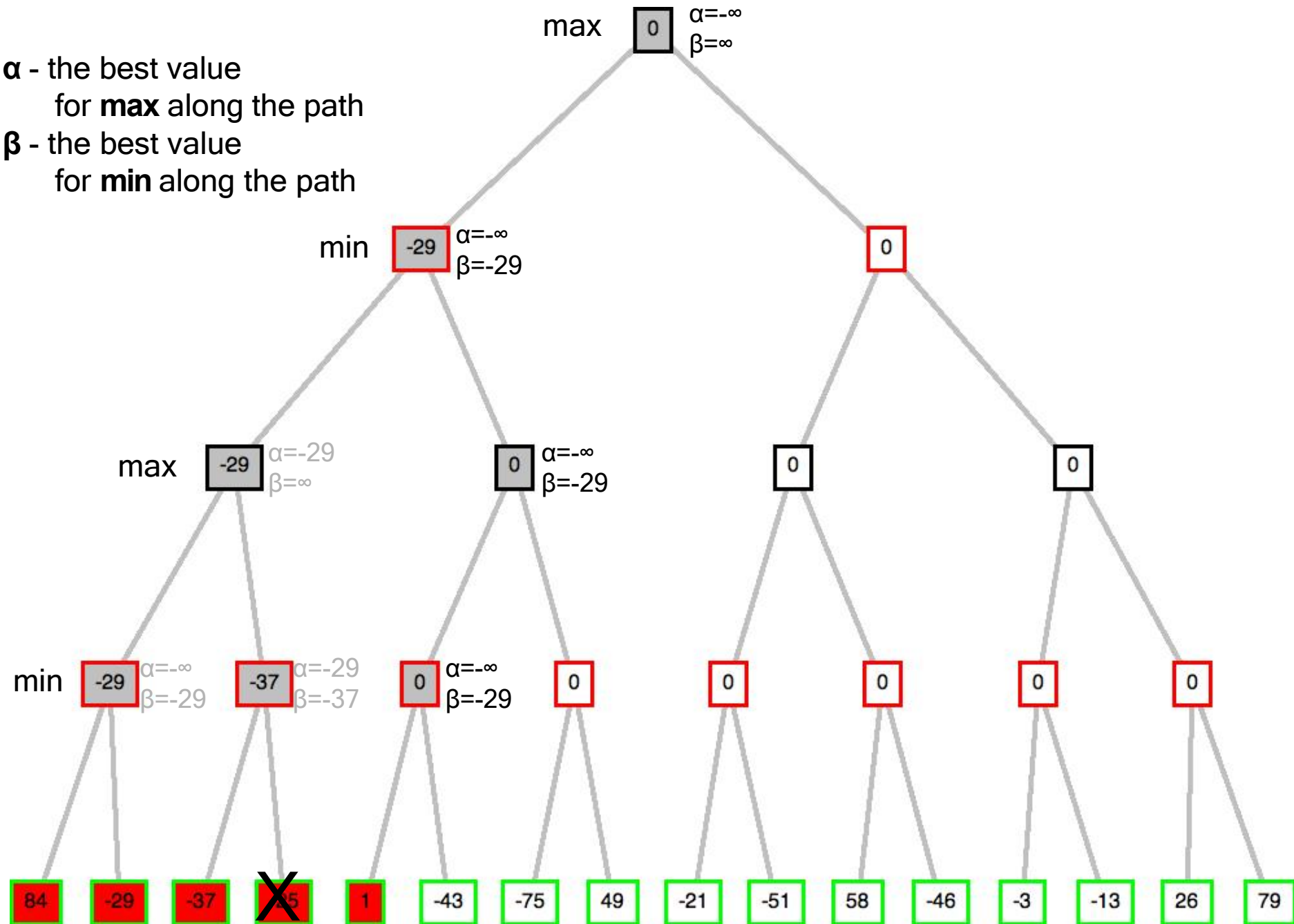
**$\beta$**  - the best value  
for **min** along the path



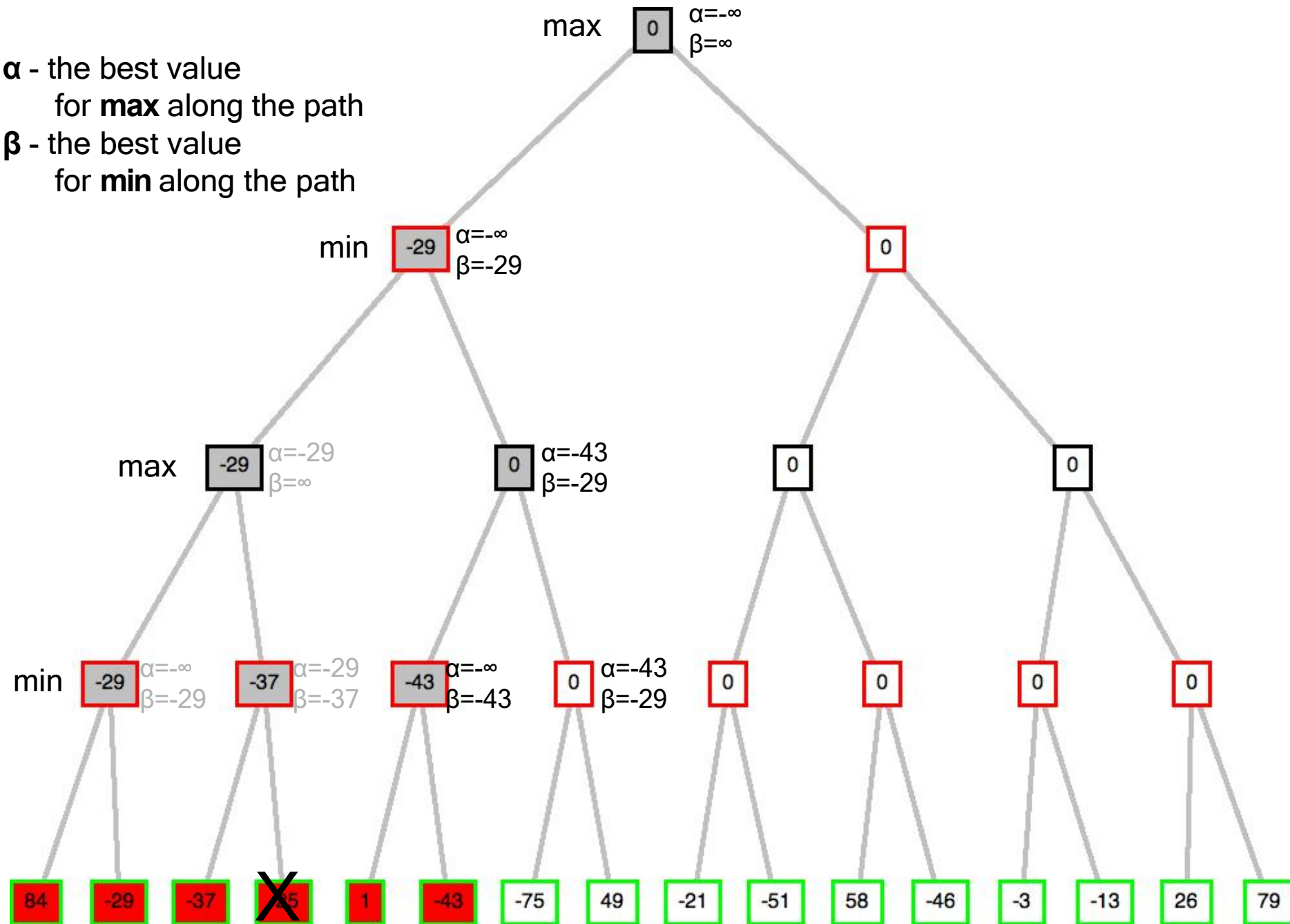
[illegible]

$\alpha$  - the best value  
for **max** along the path

$\beta$  - the best value  
for **min** along the path

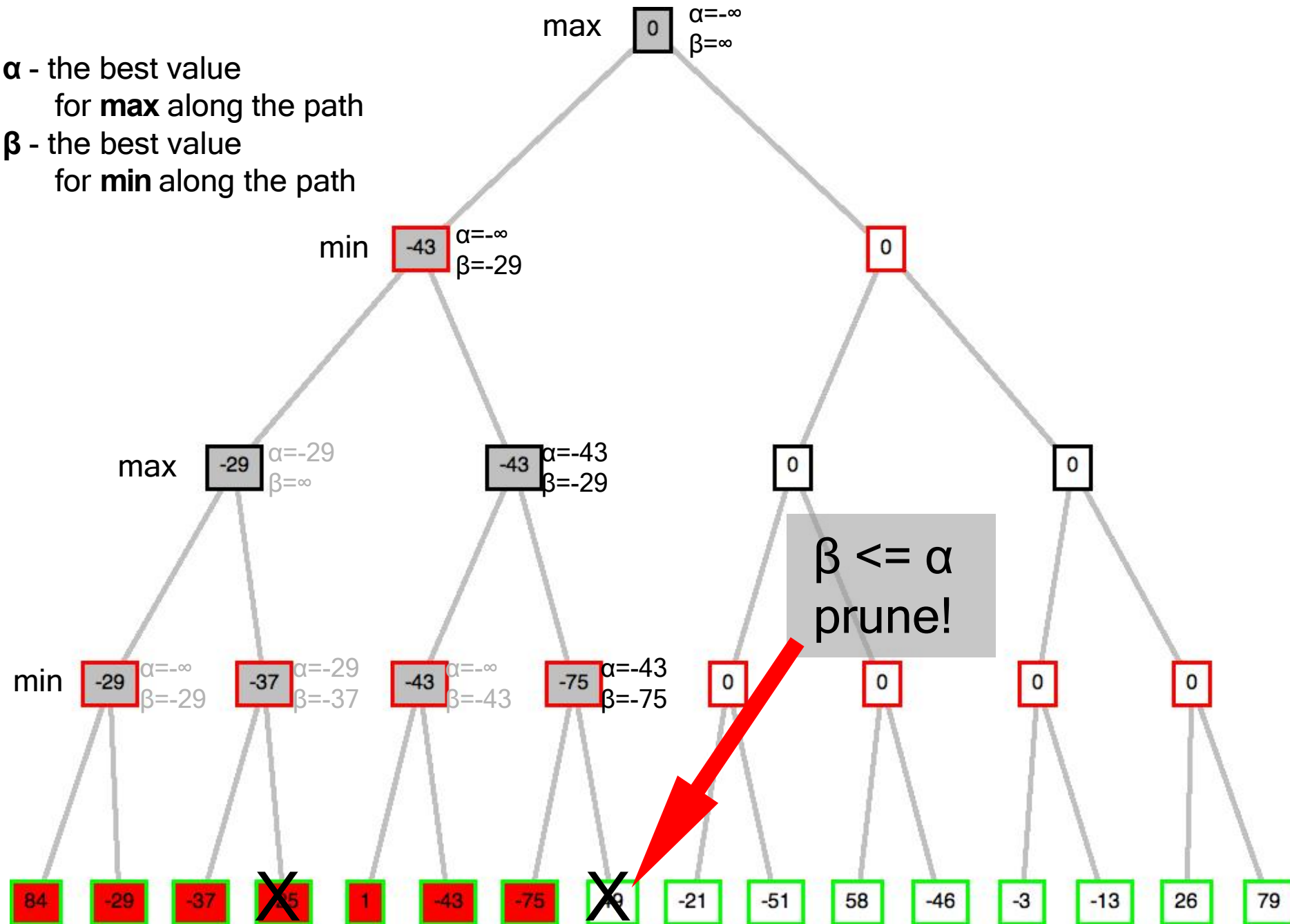


$\alpha$  - the best value  
for **max** along the path  
 $\beta$  - the best value  
for **min** along the path



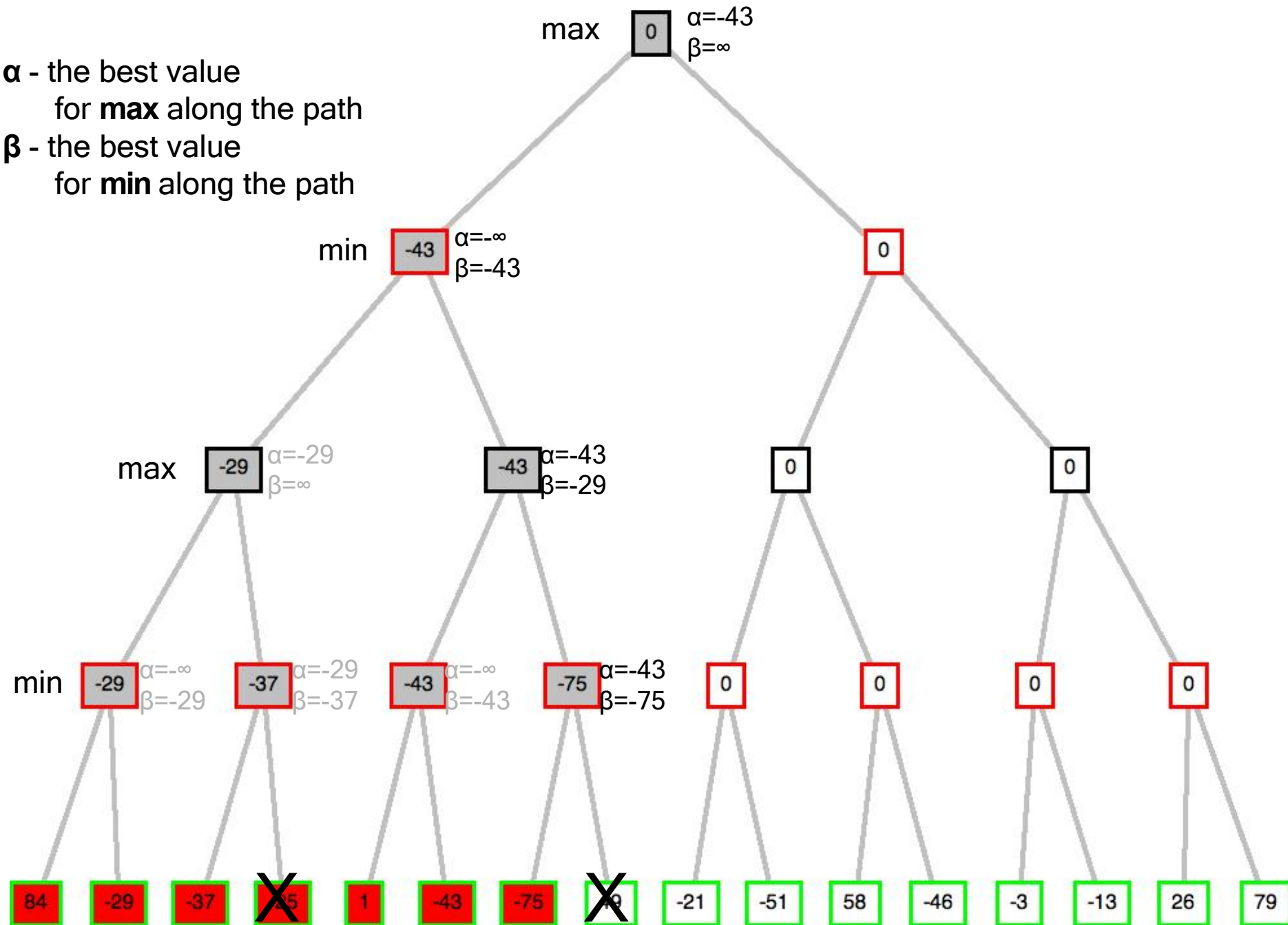
$\alpha$  - the best value  
for **max** along the path

$\beta$  - the best value  
for **min** along the path

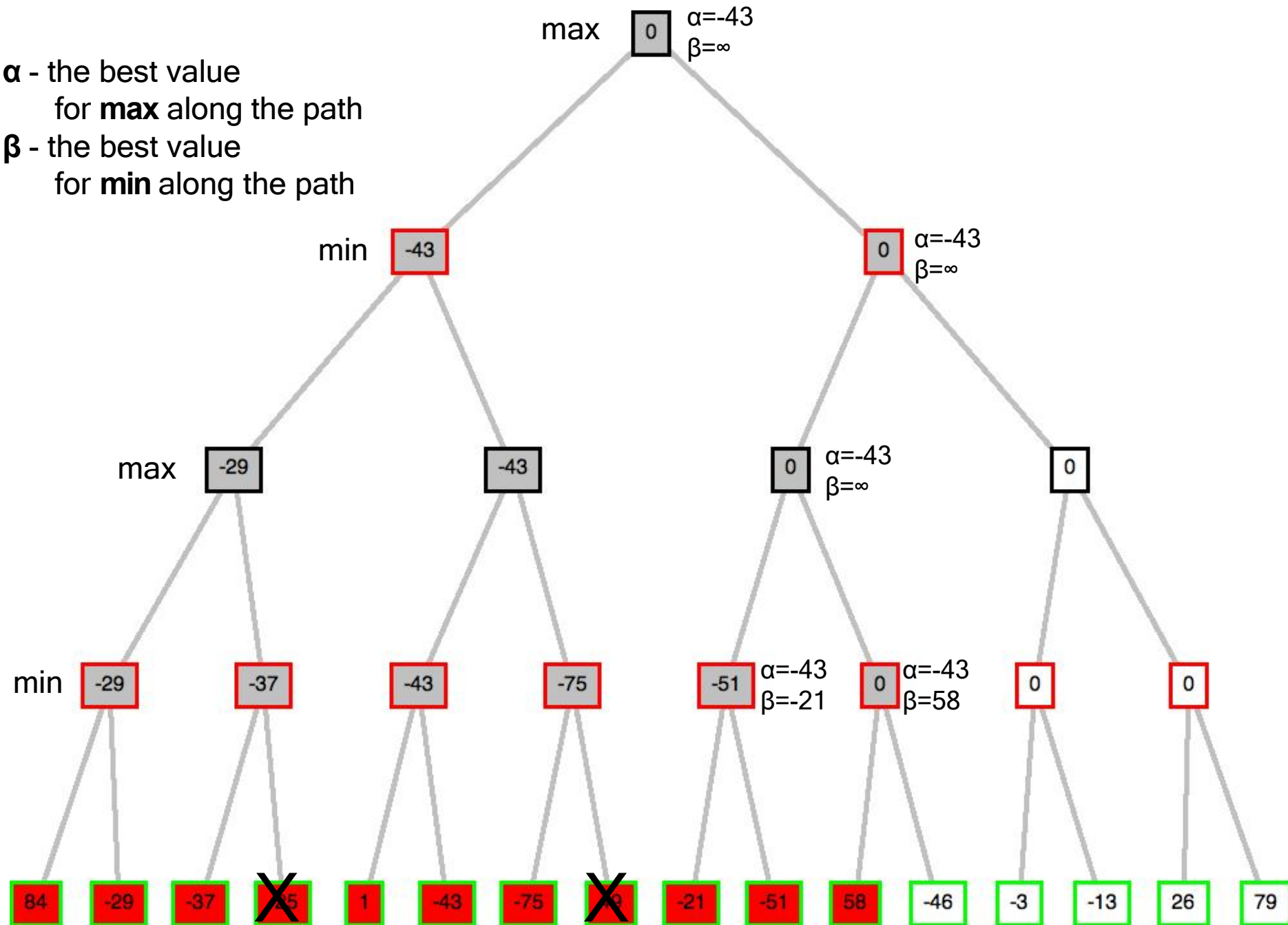


$\alpha$  - the best value  
for **max** along the path

$\beta$  - the best value  
for **min** along the path



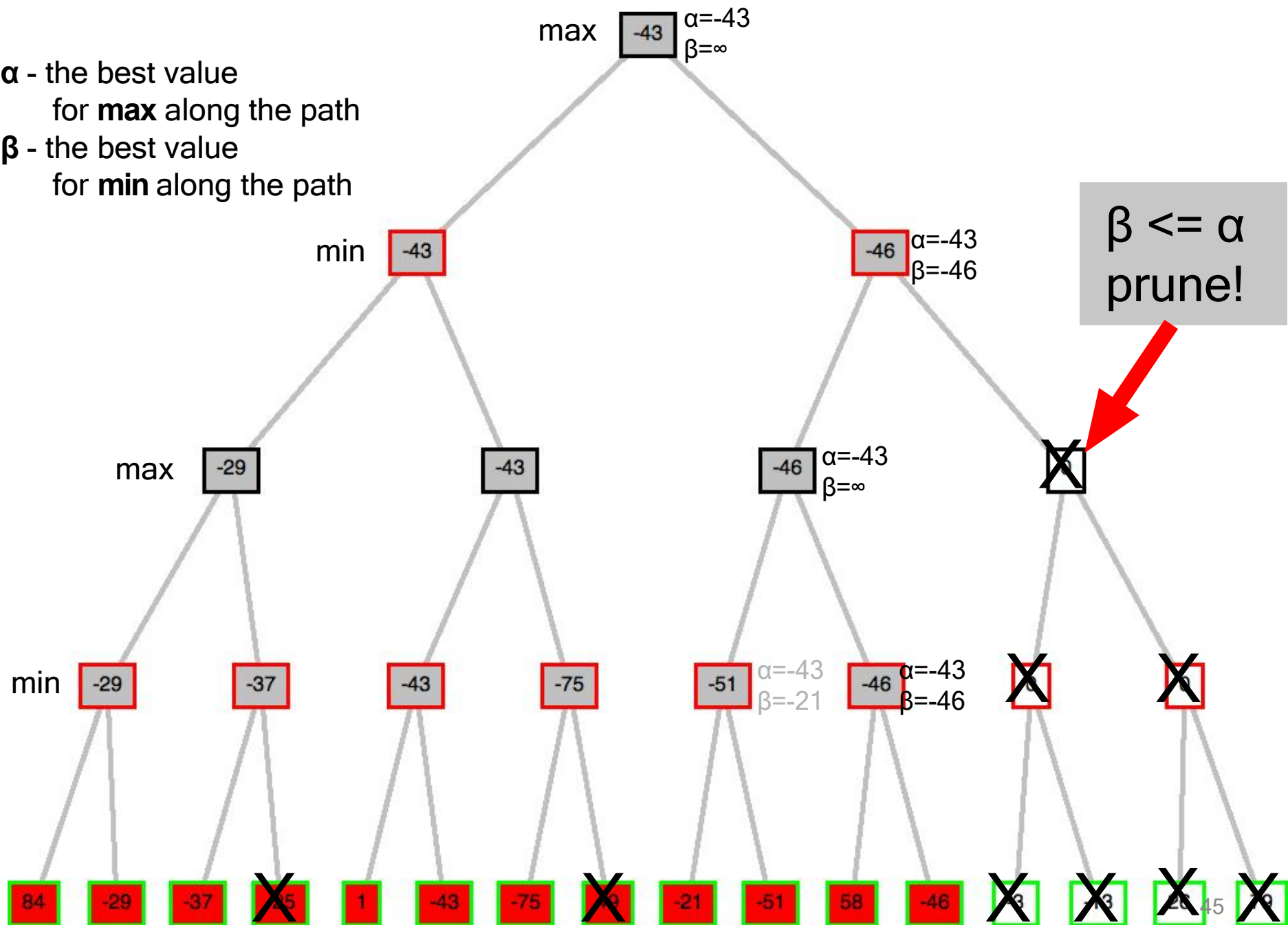
$\alpha$  - the best value  
for **max** along the path  
 $\beta$  - the best value  
for **min** along the path





$\alpha$  - the best value for **max** along the path  
 $\beta$  - the best value for **min** along the path

$\beta \leq \alpha$   
 prune!

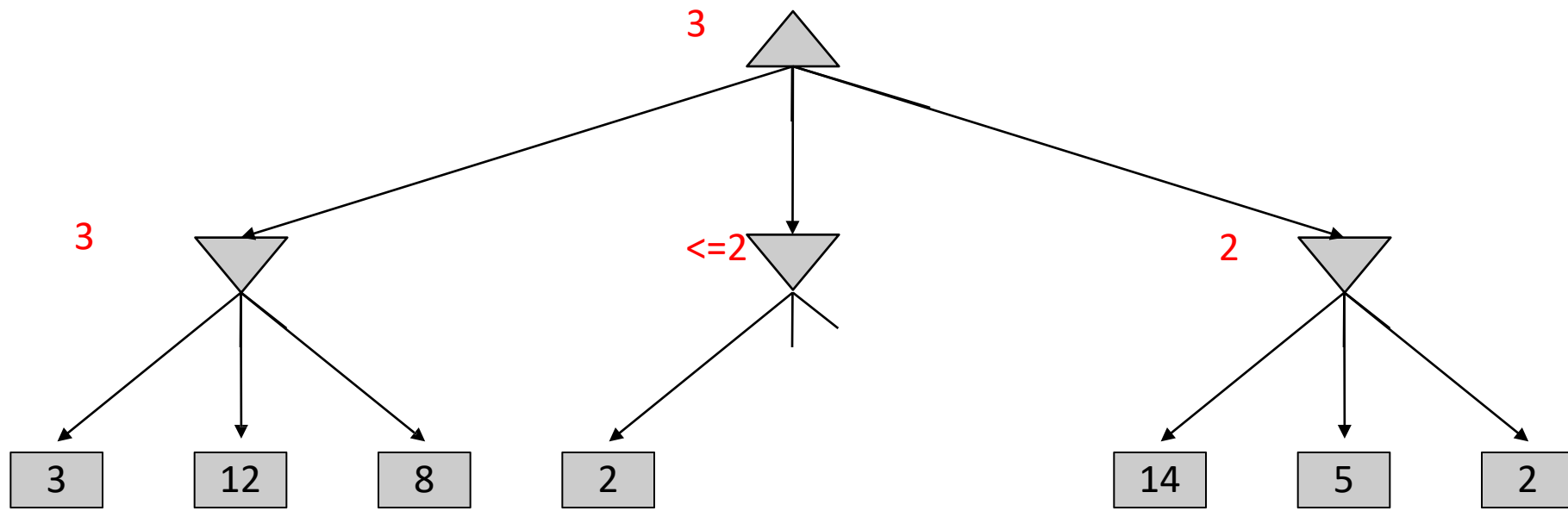




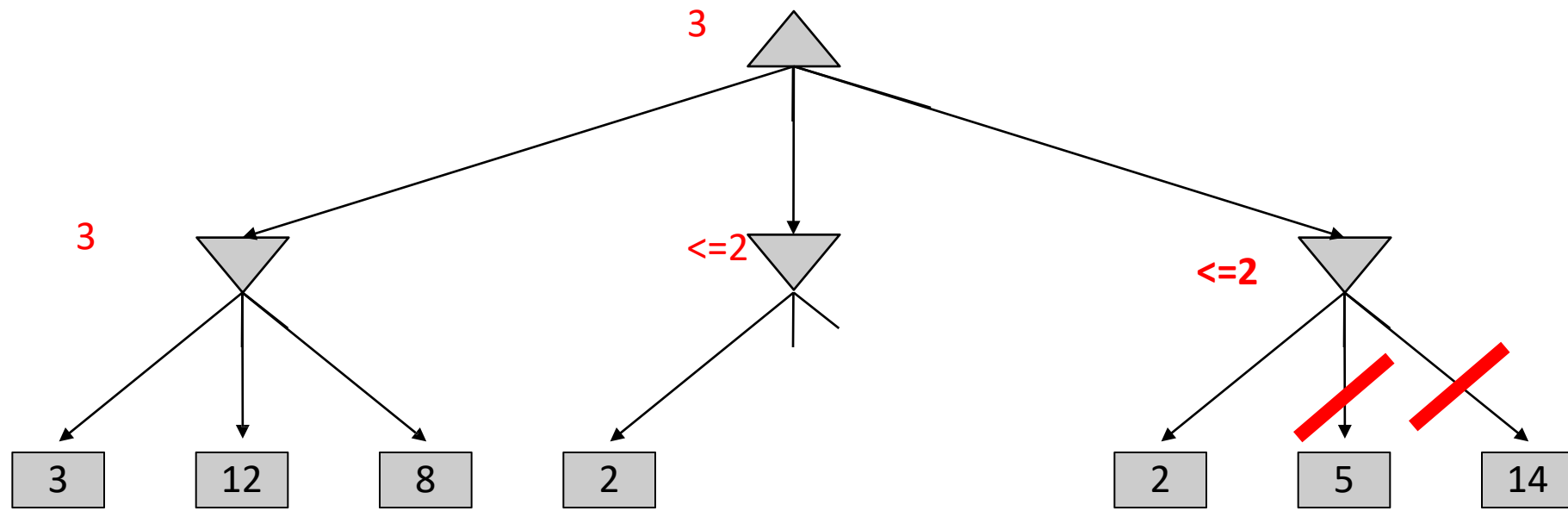
# Alpha-Beta Pruning - Properties

1. Pruning has **no effect** on the minimax value at the root.
  - Pruning does not affect the final action selected at the root.
2. A form of **meta-reasoning** (computing what to compute)
  - Eliminates nodes that are irrelevant for the final decision.

# Alpha-Beta Pruning – Order of nodes matters



# Alpha-Beta Pruning – Order of nodes matters



# Alpha-Beta Pruning - Properties

1. Pruning has **no effect** on the minimax value at the root.
  - Pruning does not affect the final action selected at the root.
2. A form of **meta-reasoning** (computing what to compute)
  - Eliminates nodes that are irrelevant for the final decision.
3. The alpha-beta search cuts the largest amount off the tree when we examine the **best move first**
  - Problem: However, best moves are typically **not** known.
  - Solution: Perform iterative deepening search and evaluate the states.
4. Time Complexity
  - **Best ordering** -  $O(b^{m/2})$ . Can double the search depth for the same resources.
  - On average –  $O(b^{3m/4})$  if we expect to find the min or max after  $b/2$  expansions.

# Minimax for Chess

- Chess:
  - branching factor  $b \approx 35$
  - game length  $m \approx 100$
  - search space  $b^m \approx 35^{100} \approx 10^{154}$

# Alpha-Beta for Chess

- Chess:
  - branching factor  $b \approx 35$
  - game length  $m \approx 100$
  - search space  $b^{m/2} \approx 35^{50} \approx 10^{77}$

# Cutting-off Search

- Problem:
  - Minimax search: full tree till the terminal nodes.
  - Alpha-beta prunes the tree but still searches till the terminal nodes.
  - Still difficult to search till the leaves.
- Solution:
  - Depth-limited Search (H-Minimax)
  - Search only to a limited depth (cutoff) in the tree
  - Replace the terminal utilities with an evaluation function for non-terminal positions

H-MINIMAX( $s, d$ ) =

$$\begin{cases} \text{EVAL}(s) & \text{if CUTOFF-TEST}(s, d) \\ \max_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{H-MINIMAX}(\text{RESULT}(s, a), d + 1) & \text{if PLAYER}(s) = \text{MIN.} \end{cases}$$

