Lecture 13

Games II

Lusi Li

Department of Computer Science

ODU

Reading for This Class:

Chapter 5, Russell and Norvig

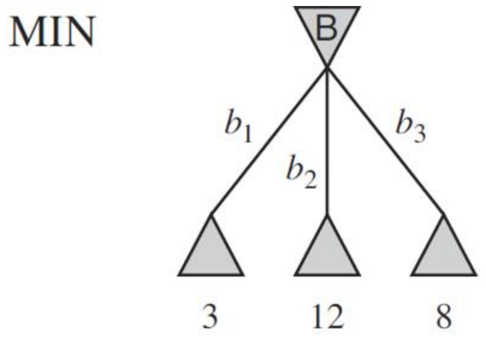
Review

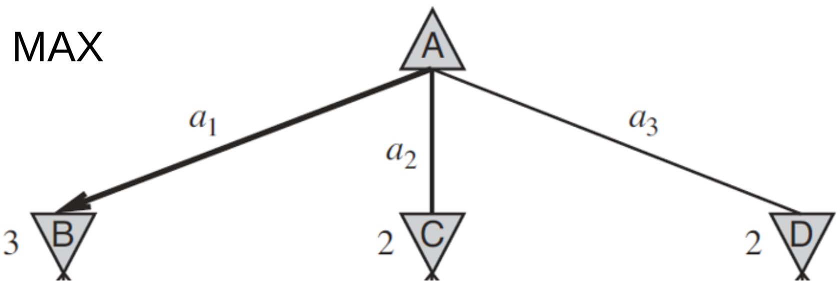
- **Last Class**
  - **Games**
  - **Adversarial Search**
- **This Class**
  - **Pruning a Game Tree**

- **Next**
  - **Midterm Exam (02/29-03/01)**

Review: Optimal Strategy

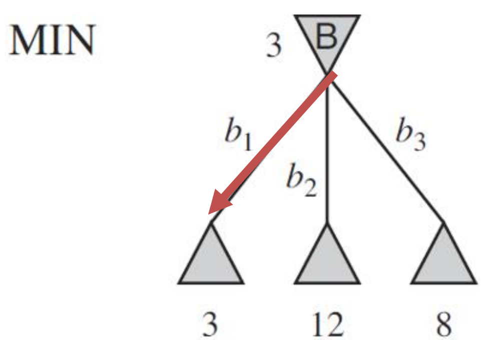
- **MAX moves first and they take turns moving until game is over**
- **Assumption: both players play optimally.**
  - **given a choice, MAX prefers to move to a state of maximum value**
  - **whereas MIN prefers a state of minimum value**
- **Given a game tree, each node will get a MiniMax value.**
- **High value is good for MAX but bad for MIN.**

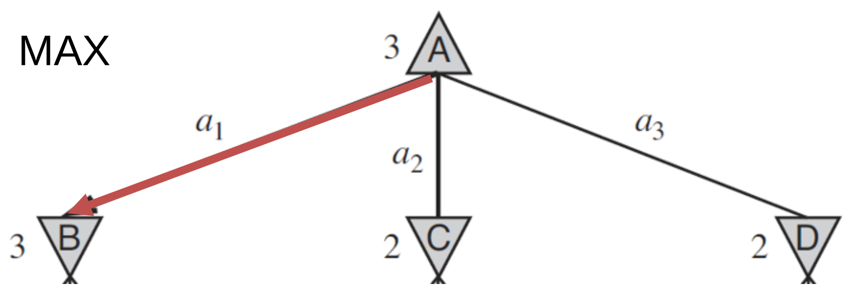




Review: Optimal Strategy

- **MAX moves first and they take turns moving until game is over**
- **Assumption: both players play optimally.**
  - **given a choice, MAX prefers to move to a state of maximum value**
  - **whereas MIN prefers a state of minimum value**
- **Given a game tree, each node will get a MiniMax value.**
- **High value is good for MAX but bad for MIN.**





Review: MiniMax Value

```
MiniMax(s) =

if Terminal-Test(s) then Utility(s)
if Player(s) = MAX then
     max of MiniMax(Result(s, a)) for a in Actions(s)
if Player(s) = MIN then
     min of MiniMax(Result(s, a)) for a in Actions(s)
```

Review: Minimax Algorithm

function MINIMAX-DECISION ($state$) **returns** $an\ action$
return $\arg\min_{a\epsilon\ \mathrm{ACTIONS}(s)}$ MAX − VALUE(RESULT($state, a$))

// choose the best move
// for a MIN node

---

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

// choose the best move
// for a MAX node

---

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

// calculate the value
// for a MAX node

---

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

// calculate the value
// for a MIN node

MiniMax Properties

- Assume all terminal states are at depth m and there are b possible moves at each step
- Minimax explores tree using DFS.
  - recursive implementation
- Time complexity is *$O(b^m)$*
  - exponential in the number of moves
- Space complexity is *$O(bm)$*
  - where b is the branching factor, m the maximum depth of the search tree
- Completeness: Yes, if tree is finite
- Optimality: Yes, if MIN is an optimal opponent

Problem with Minimax Search

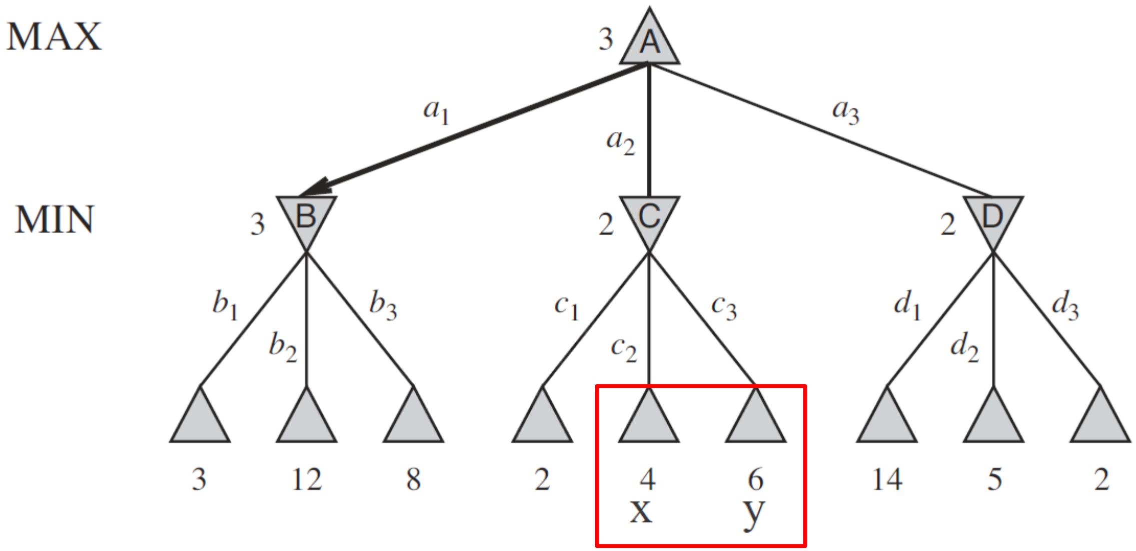
Problems:

- Complete search is impractical for most games
  - Number of game states is exponential in the number of moves

- Solution: Make the same decision without examining every node by pruning

- Alpha-beta pruning
  - an extension of the minimax approach
  - results in the same move as minimax, but with less overhead
  - prunes uninteresting parts of the search tree

Intuition of Alpha-Beta Pruning



x, y unknown

$$
\begin{aligned}
\text{MINIMAX}(root) &= \max(\min(3, 12, 8), \min(2, x, y), \min(14, 5, 2)) \\
&= \max(3, \min(2, x, y), 2) \\
&= \max(3, z, 2) \qquad \text{where } z = \min(2, x, y) \leq 2 \\
&= 3.
\end{aligned}
$$

Therefore, it is possible to compute the correct minimax decision without looking at every node in the tree.

Alpha-Beta Pruning

- α:
  - has initial value $-\infty$
  - the value of the best (i.e., highest-value) choice we have found so far at any choice point along the path for MAX
  - $\alpha \leftarrow$ MAX(α, v) updated only when it's MAX's turn

- β
  - has initial value $+\infty$
  - the value of the best (i.e., lowest-value) choice we have found so far at any choice point along the path for MIN.
  - $\beta \leftarrow$ MIN(β, v) updated only when it's MIN's turn

Alpha-Beta Pruning

Key points:

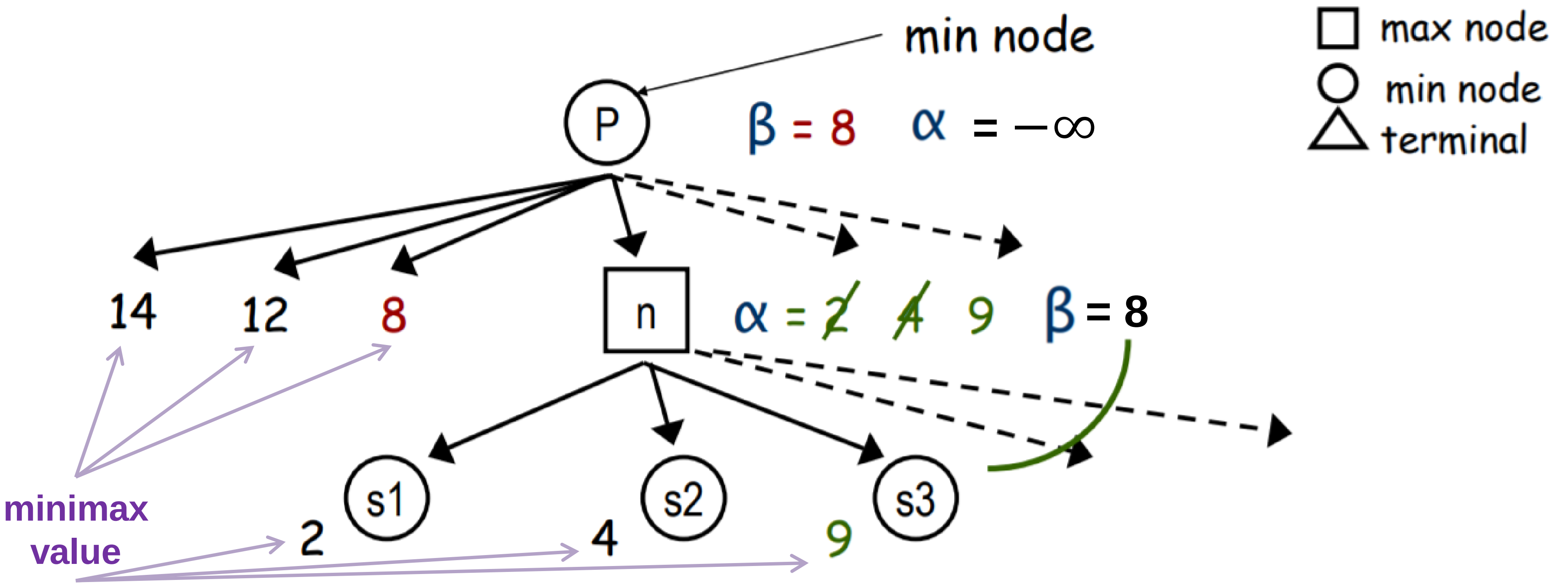
1. **Each node has to keep track of 3 values: α, β, v (minimax value).**
2. **Pruning condition: if α >= β for node n, stop expanding the children of node n and return its current v**
3. **MAX will update only α values and MIN player will update only β values. Both of them will update v values.**
4. **Return v values to parent nodes of the game tree**
5. **Pass α and β values to child nodes.**

Two types of pruning (cuts):

- **pruning of max nodes (α-cuts)**
- **pruning of min nodes (β-cuts)**

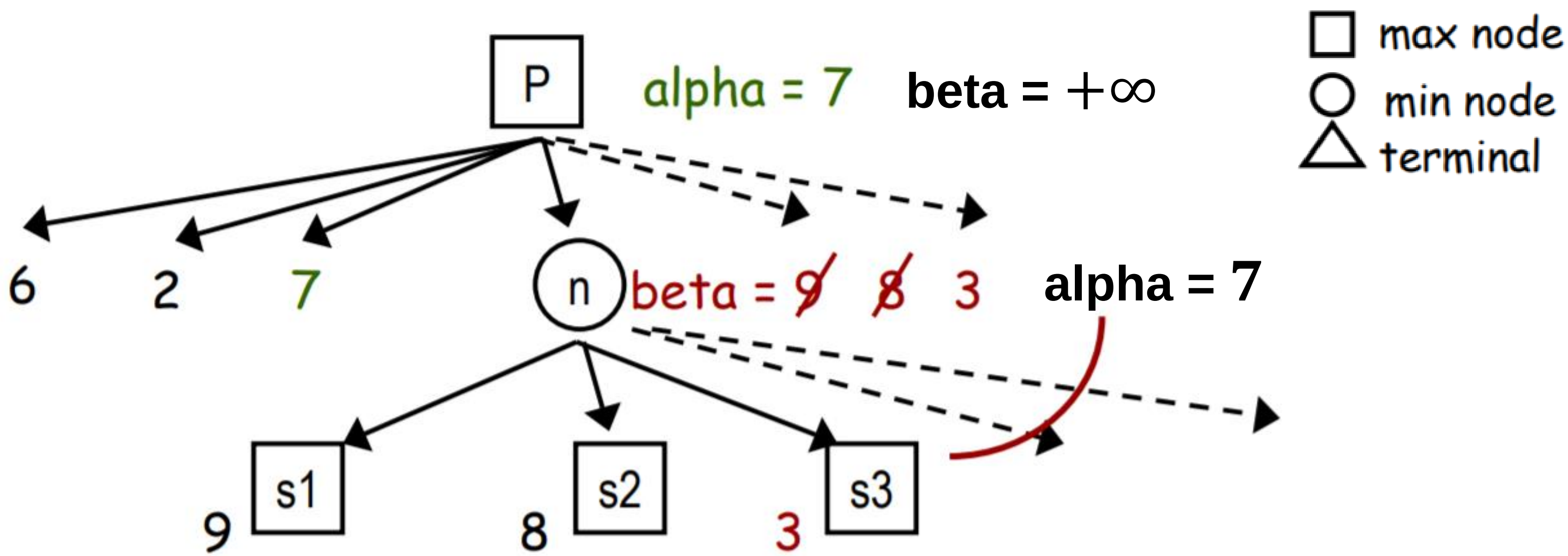
Alpha-Beta Pruning: α-cut

- **If α >= β for node n, stop expanding the children of node n**
- **α-cut: cutting some of the child nodes of a max node**
  - **MIN will never choose to move from n's parent (P) to n since it would choose one of n's lower valued siblings first.**



Alpha-Beta Pruning: β-cut

- **If α >= β for node n, stop expanding the children of node n**
- **β-cut: cutting some of the child nodes of a min node**
  - **Max will never choose to move from n's parent (P) to n since it would choose one of n's higher value siblings first.**



Alpha-Beta Algorithm

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

// choose the best move
// for a MAX node

---

function $\boxed{\text{MAX-VALUE}(state, \alpha, \beta)}$ **returns** *a utility value*
    if TERMINAL-TEST(*state*) **then return** UTILITY(*state*)
    $v \leftarrow -\infty$
    for each a **in** ACTIONS(*state*) **do**
        $v \leftarrow$ MAX(v, MIN-VALUE(RESULT(s,a), α, β))
        if $v \geq \beta$ **then return** v // prune remaining children of this max node
        $\alpha \leftarrow$ MAX(α, v)
    return v // return value of best child

// calculate v and α
// for a MAX node

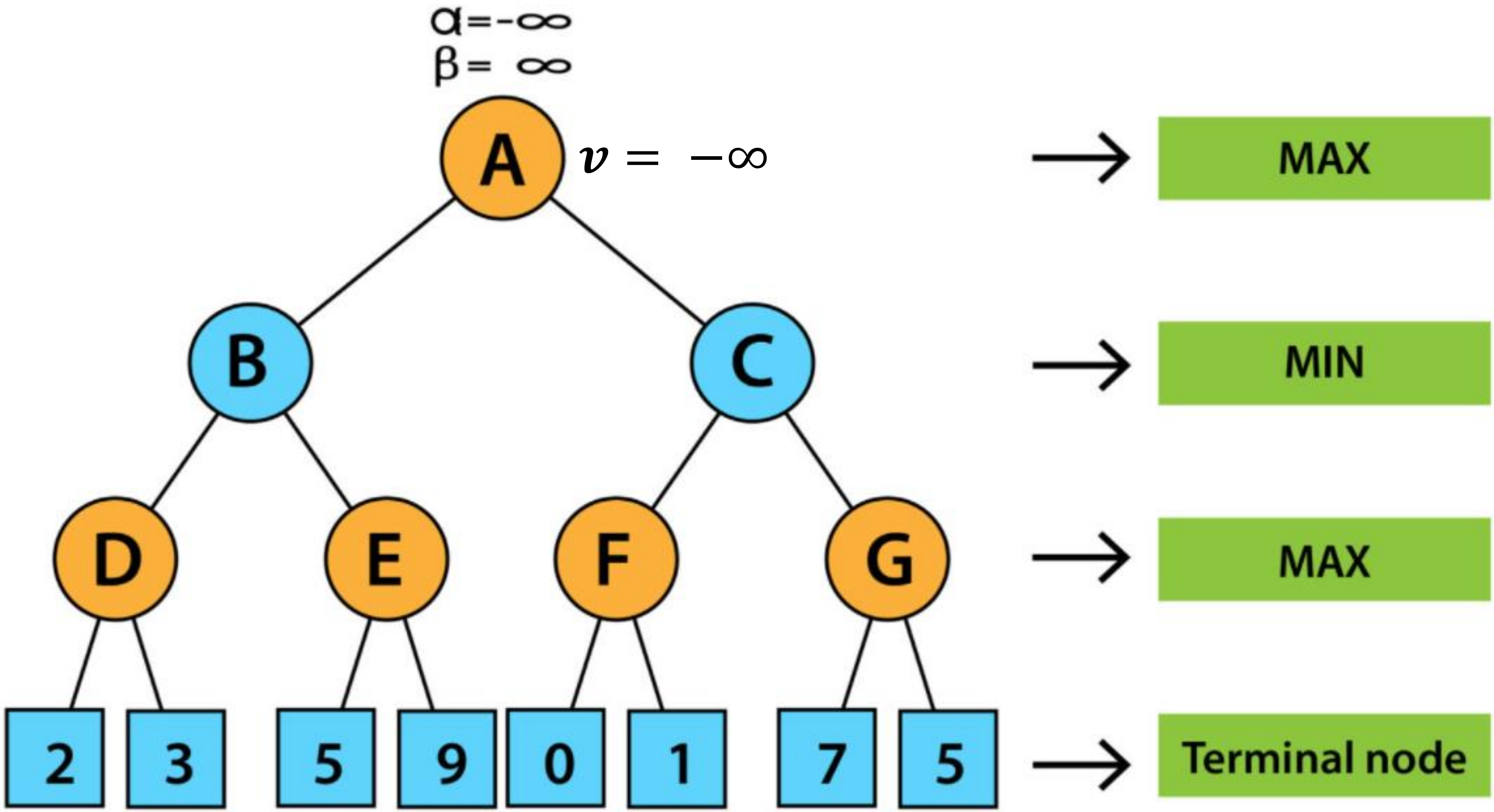
---

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

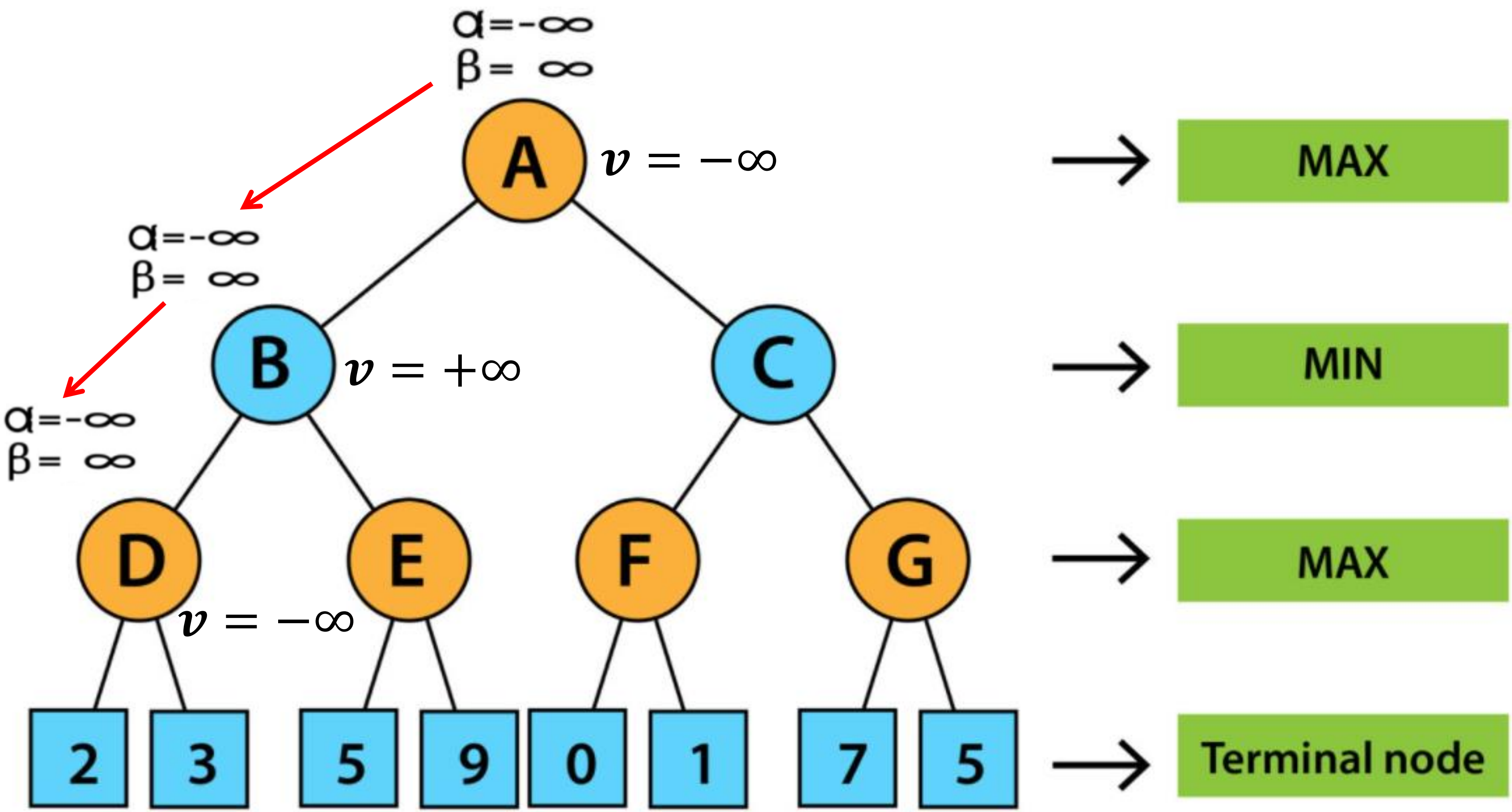
// calculate v and β
// for a MIN node

Alpha-Beta Pruning Example

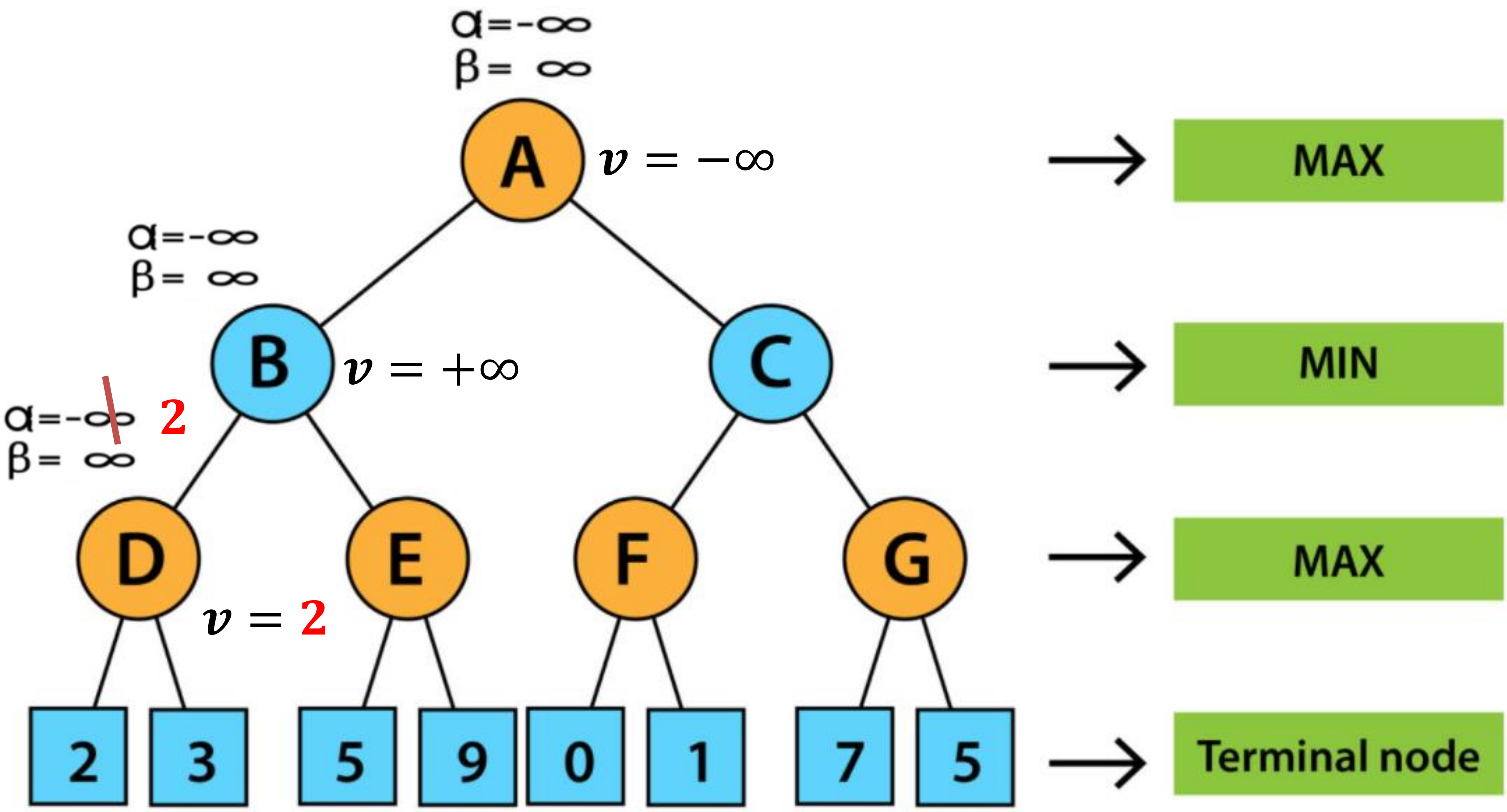
ALPHA-BETA-SEARCH (A) **returns** $an\ action$



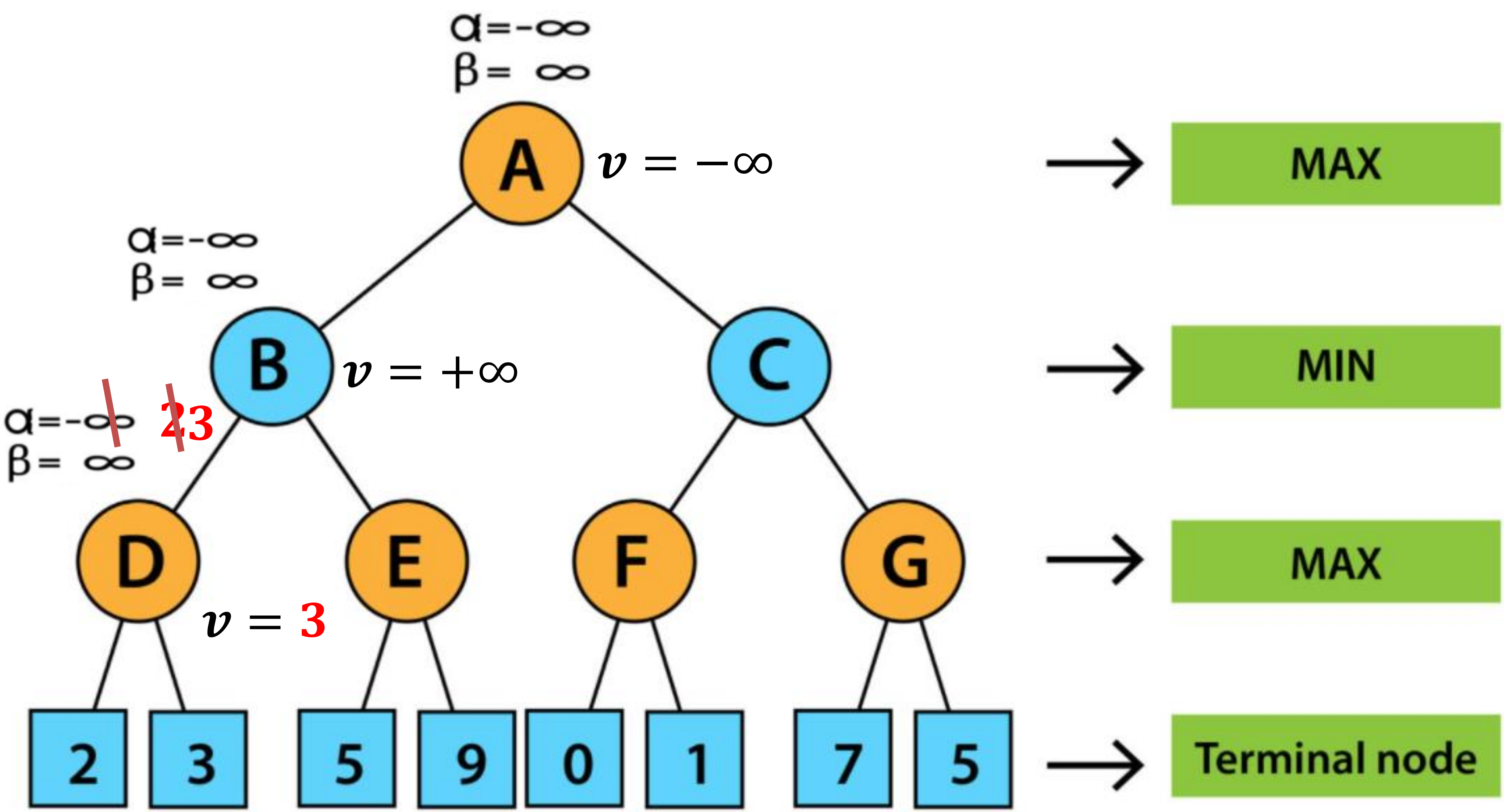
Alpha-Beta Pruning Example



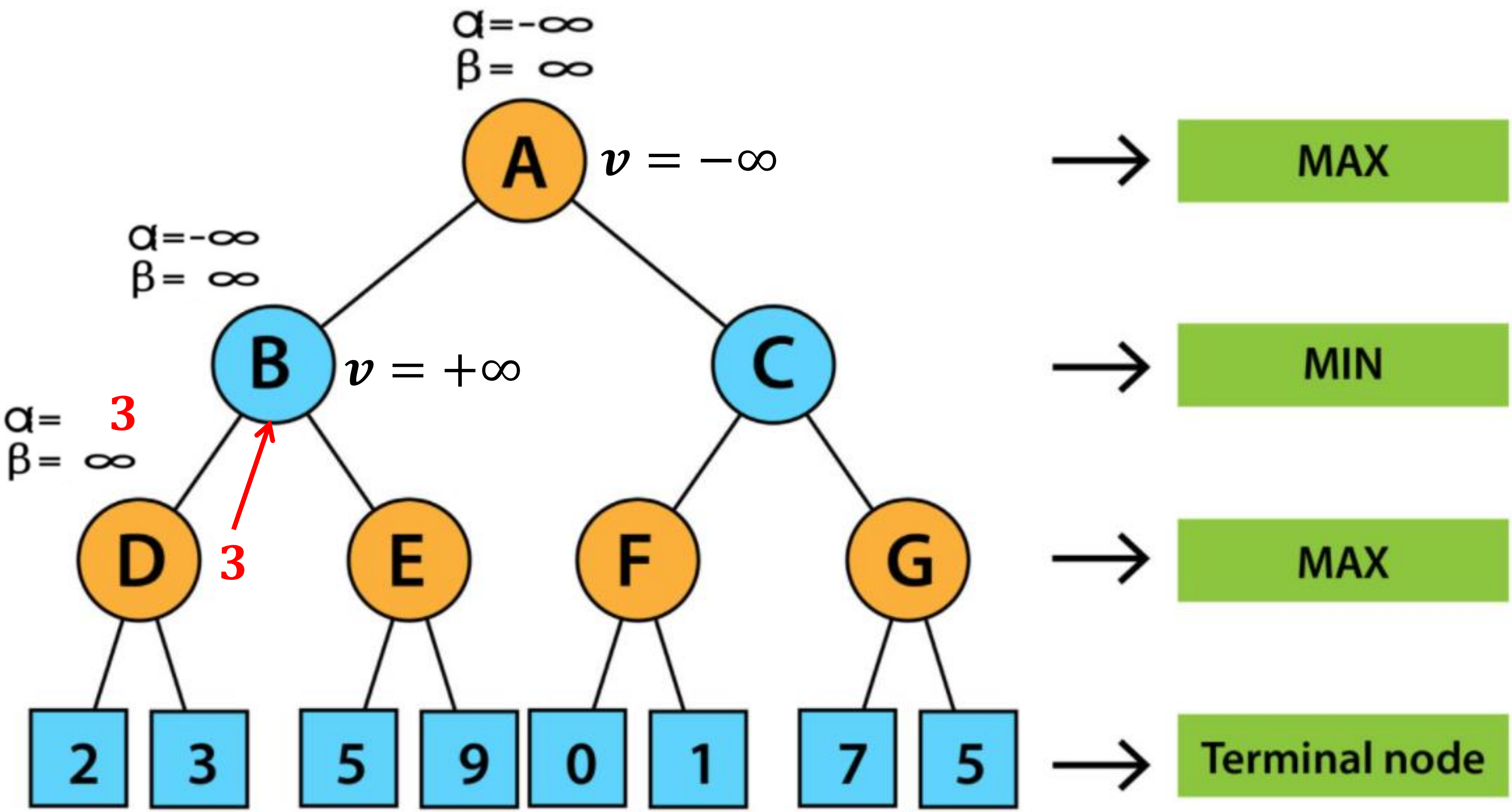
Alpha-Beta Pruning Example



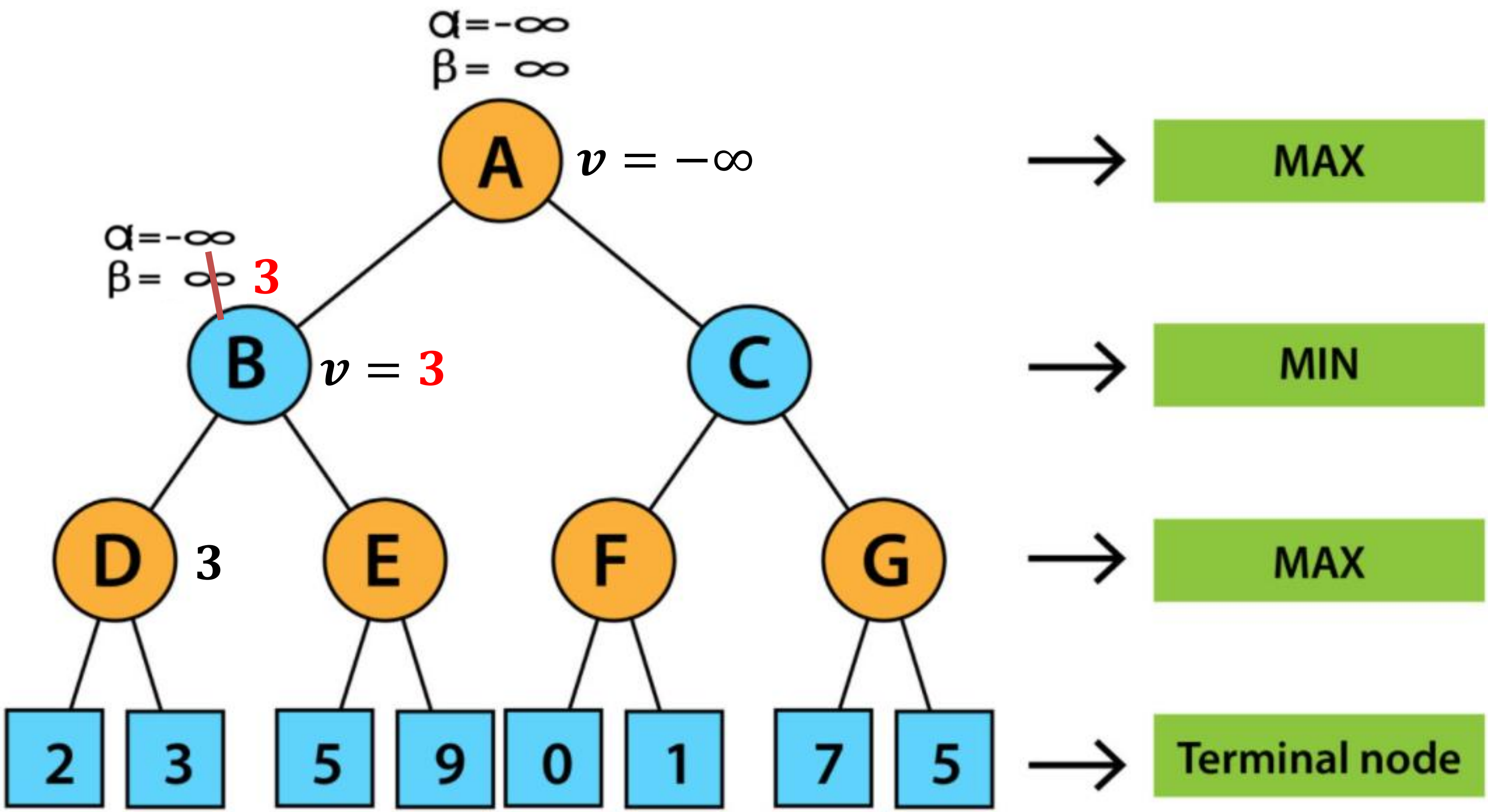
Alpha-Beta Pruning Example



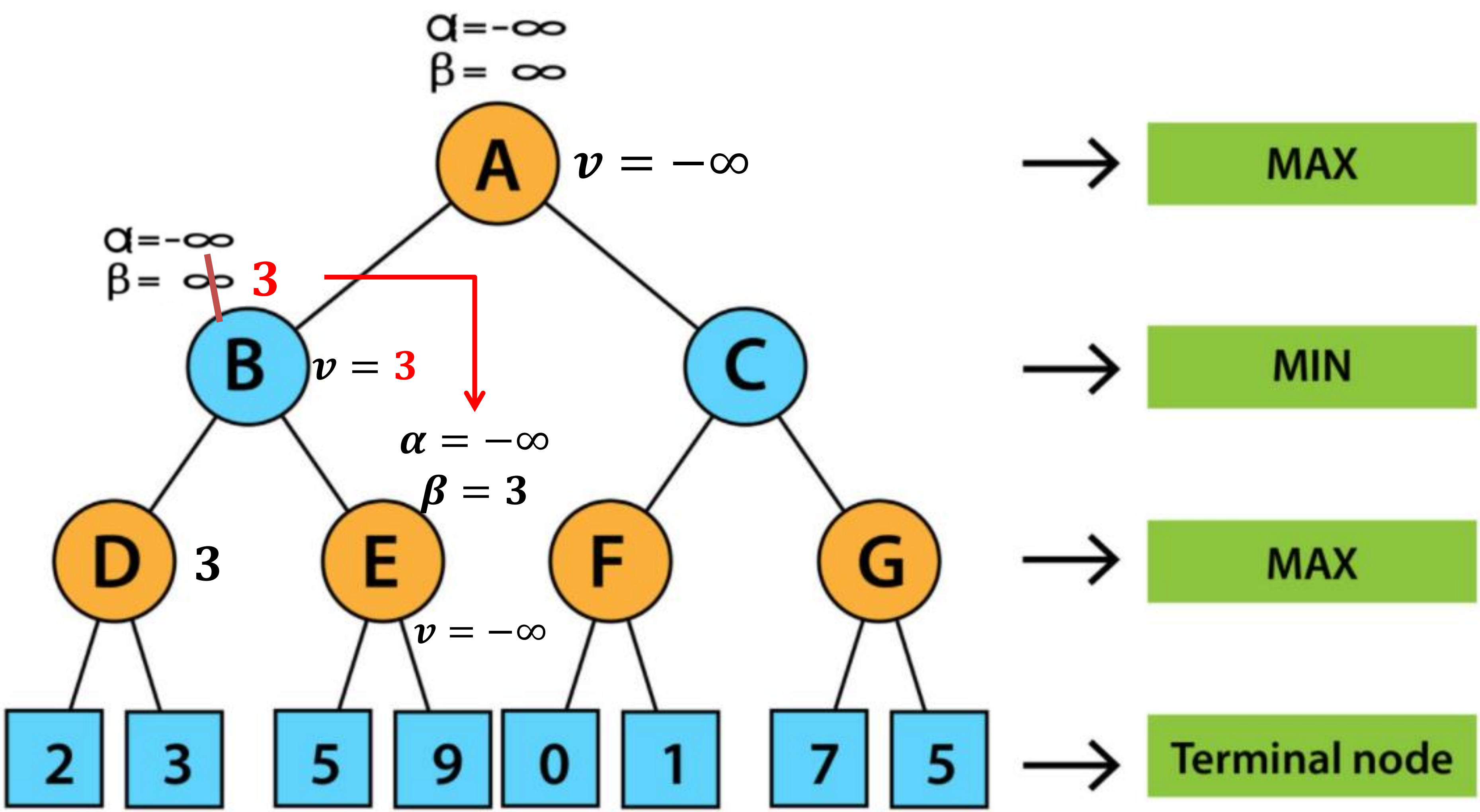
Alpha-Beta Pruning Example



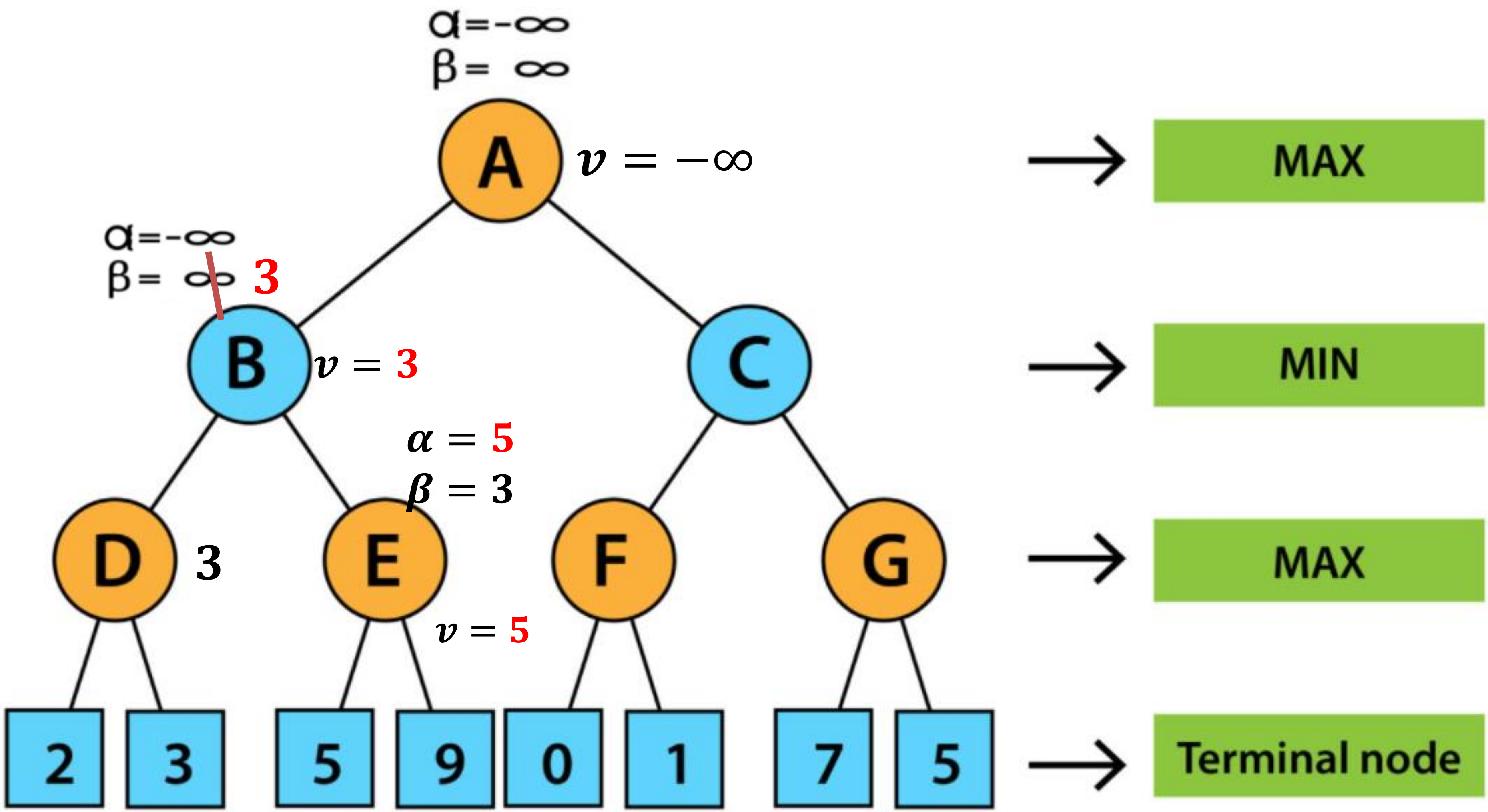
Alpha-Beta Pruning Example



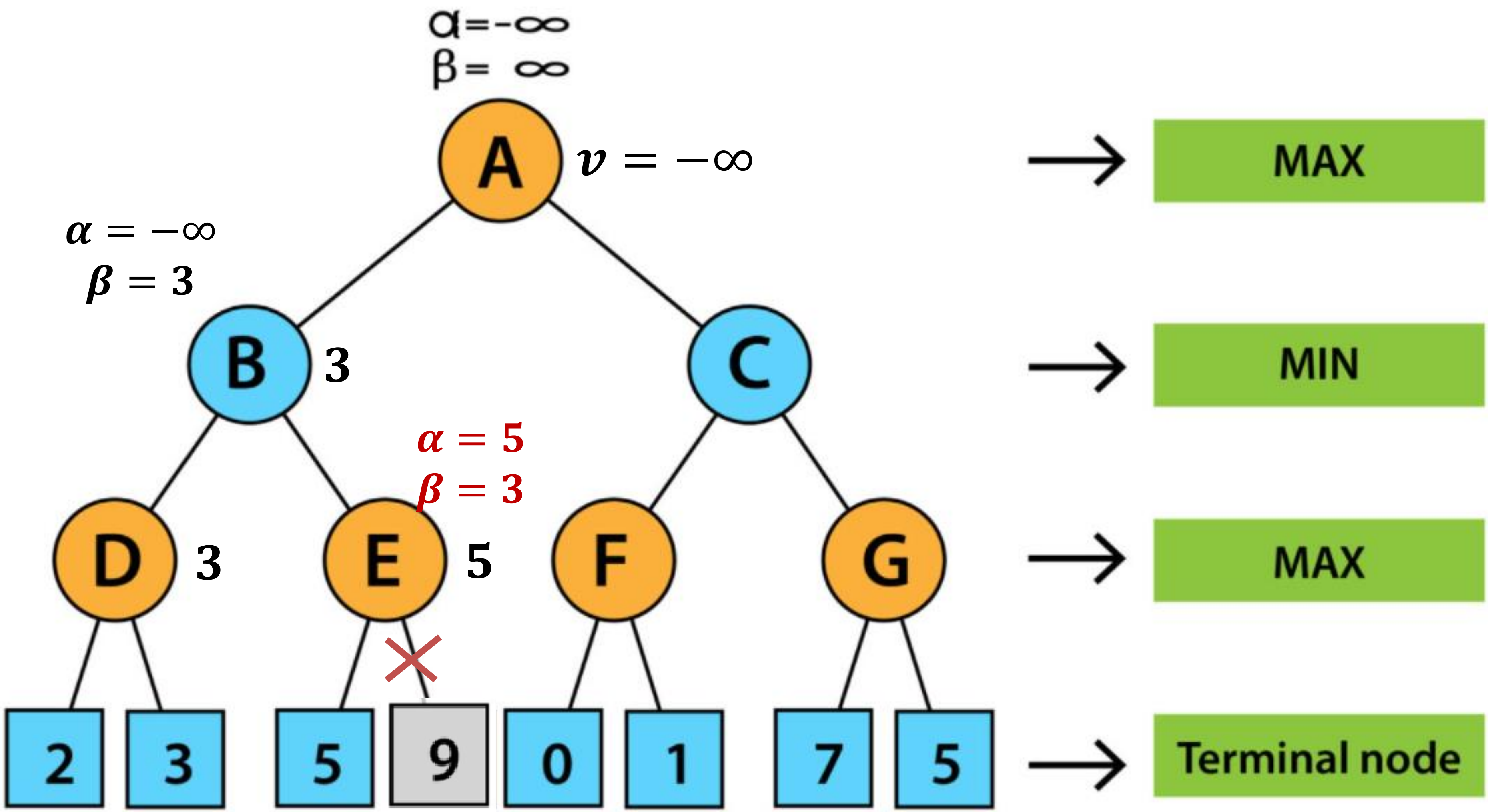
Alpha-Beta Pruning Example



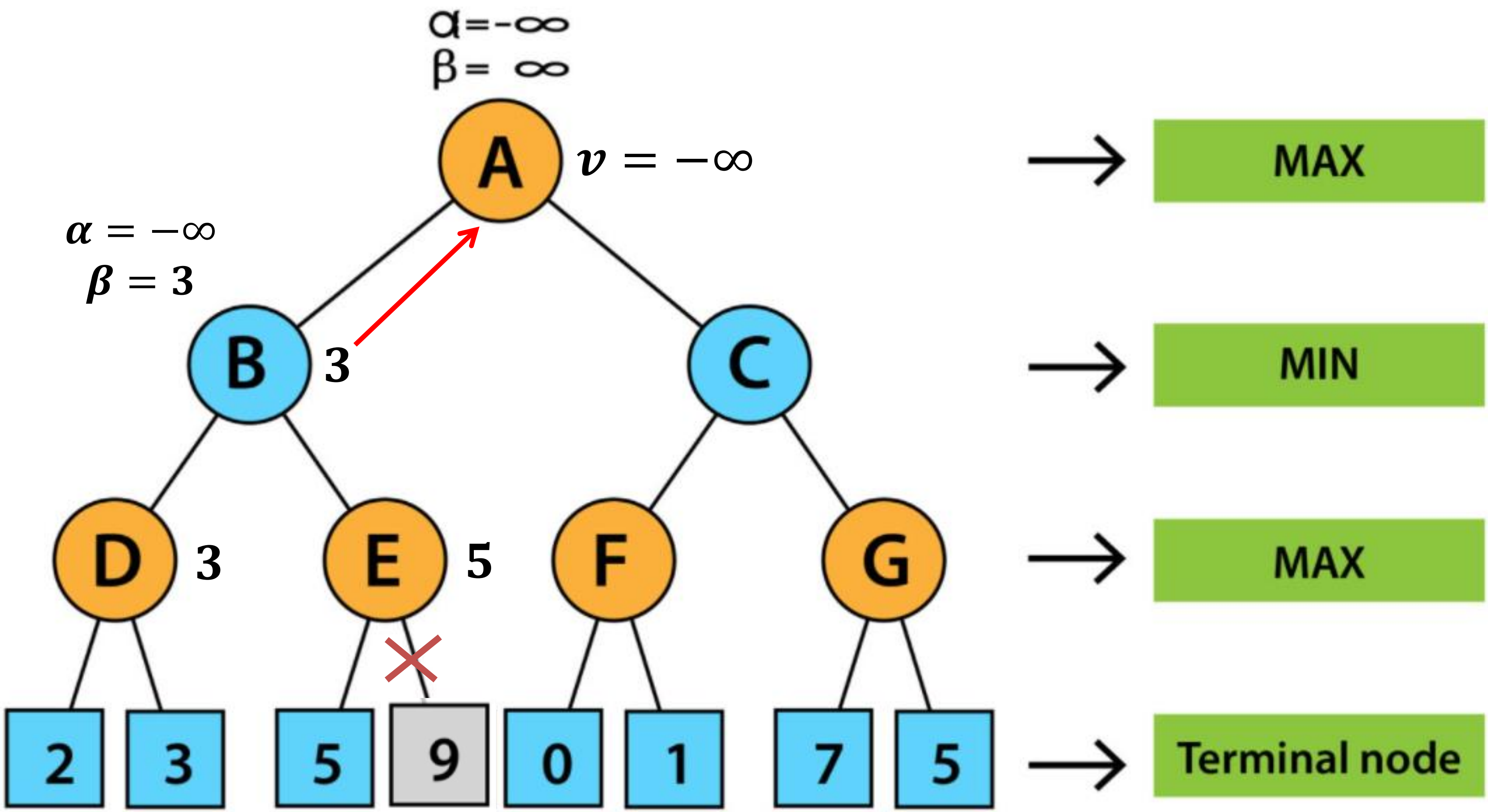
Alpha-Beta Pruning Example



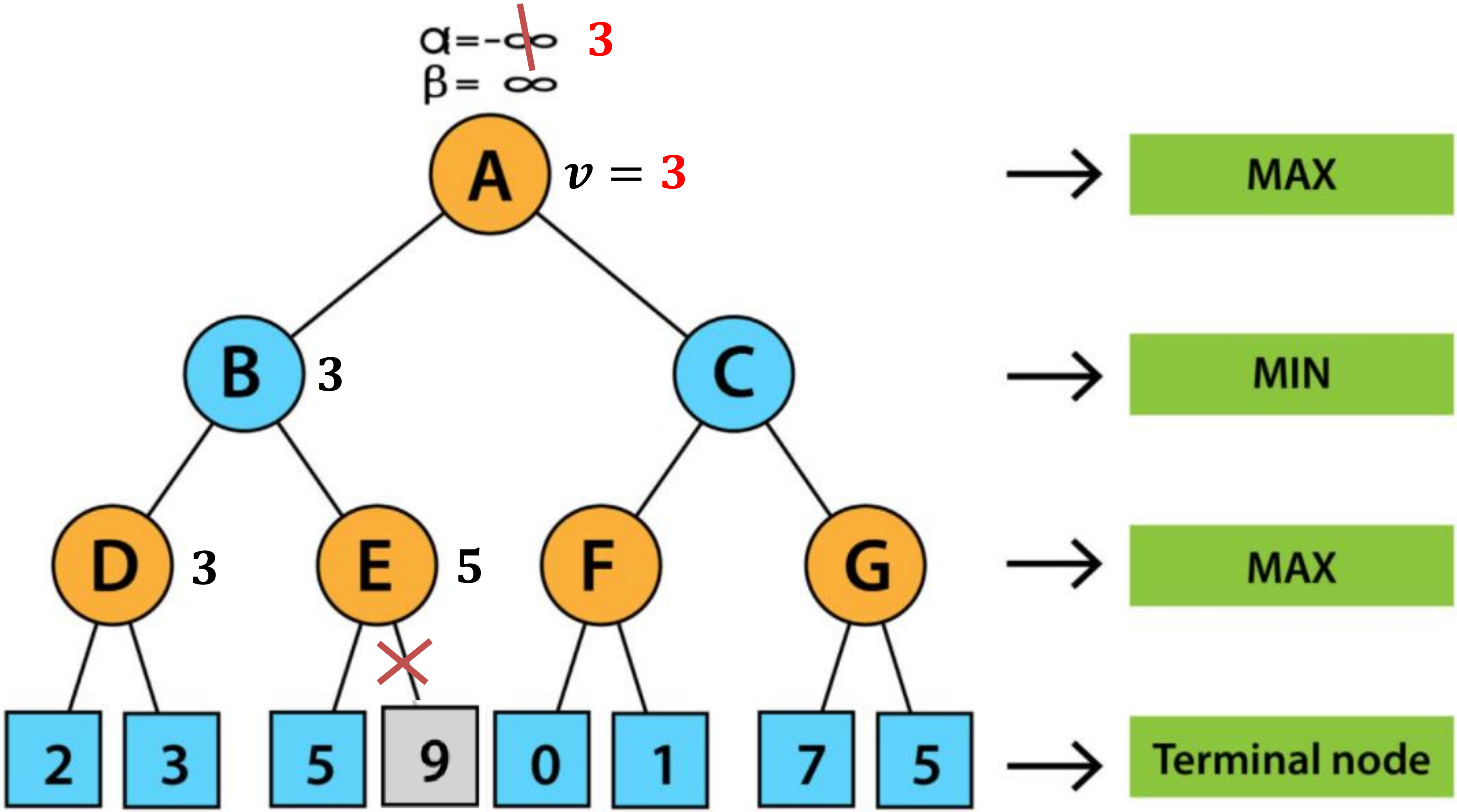
Alpha-Beta Pruning Example



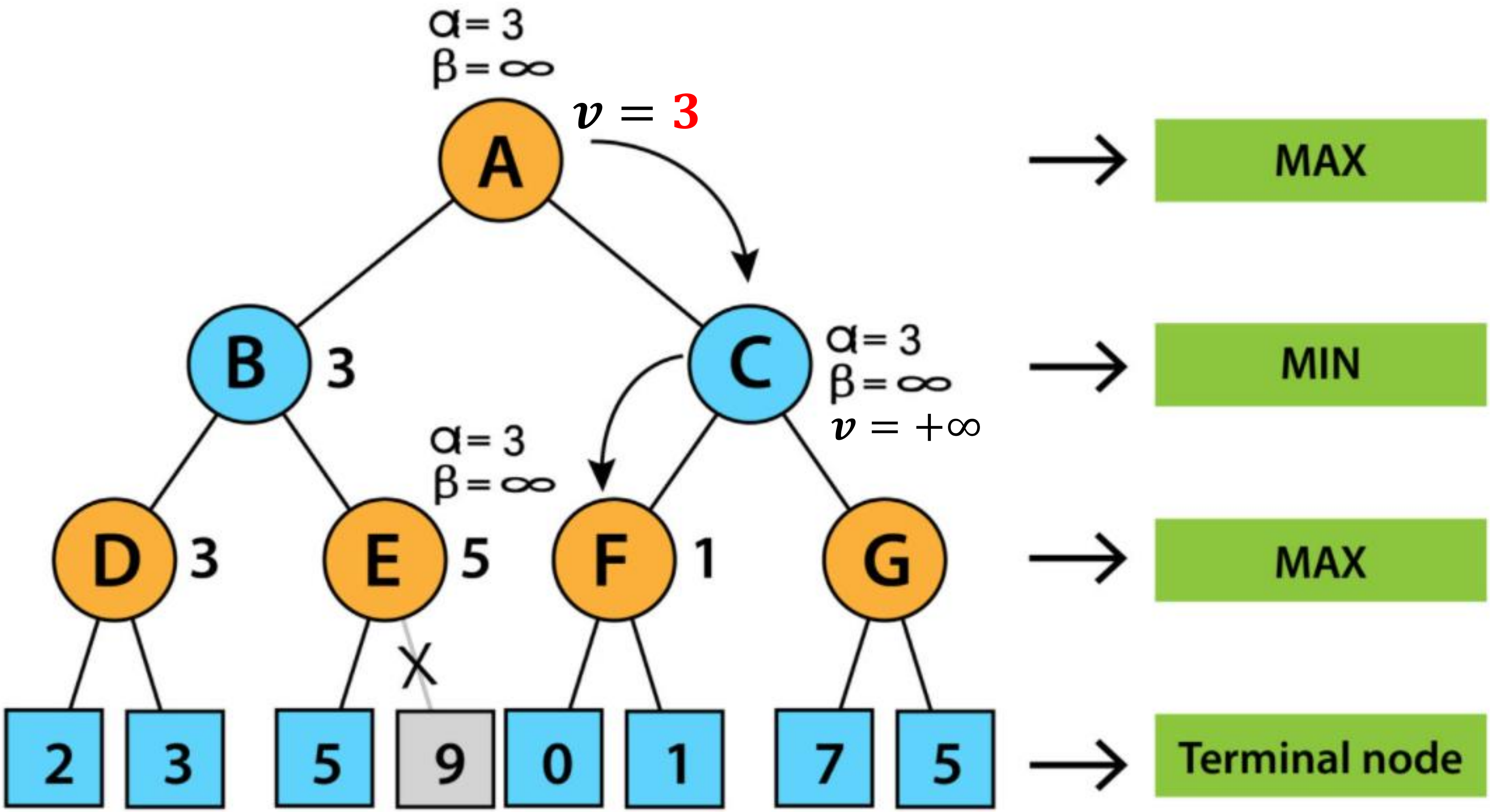
Alpha-Beta Pruning Example



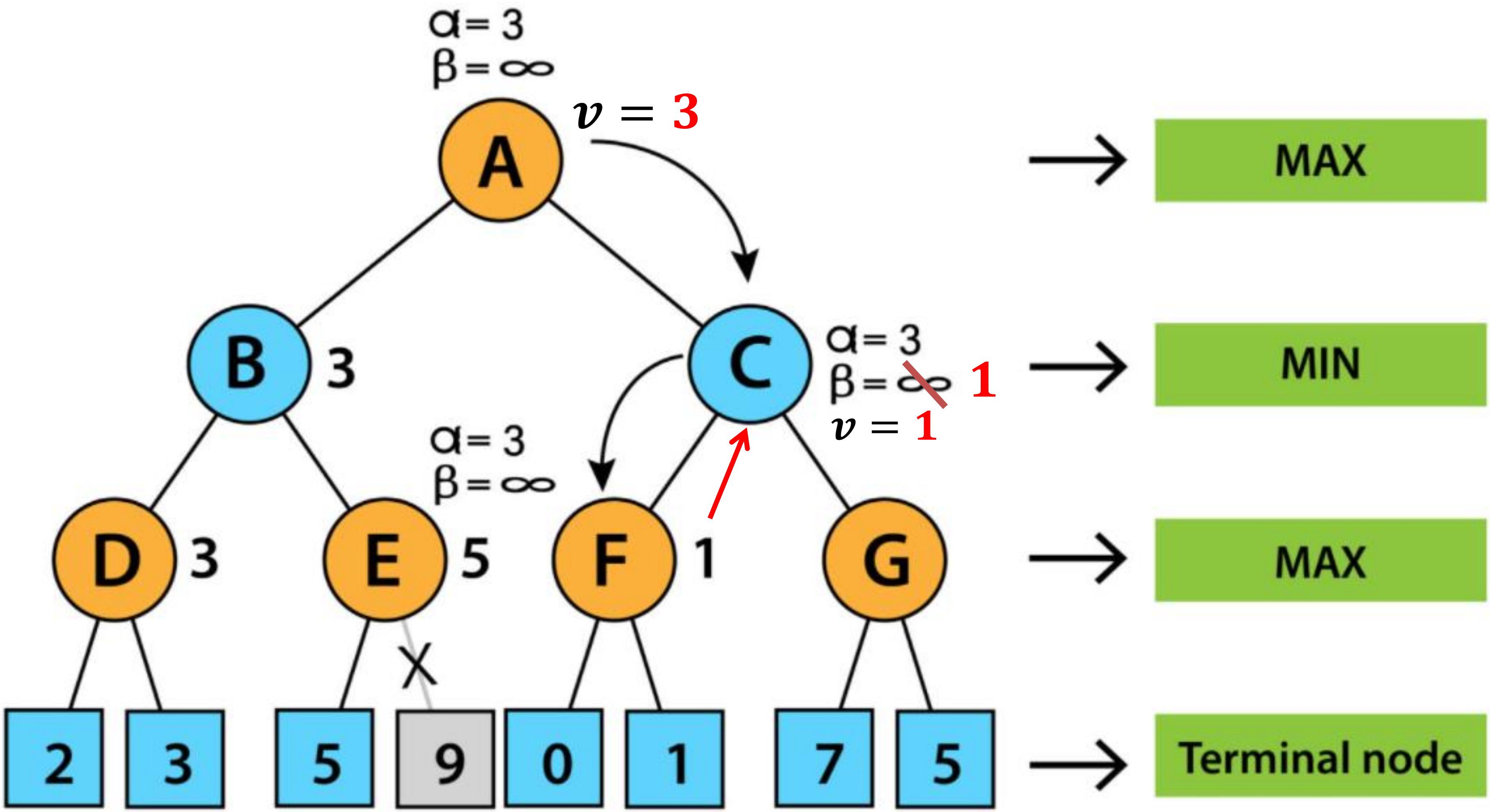
Alpha-Beta Pruning Example



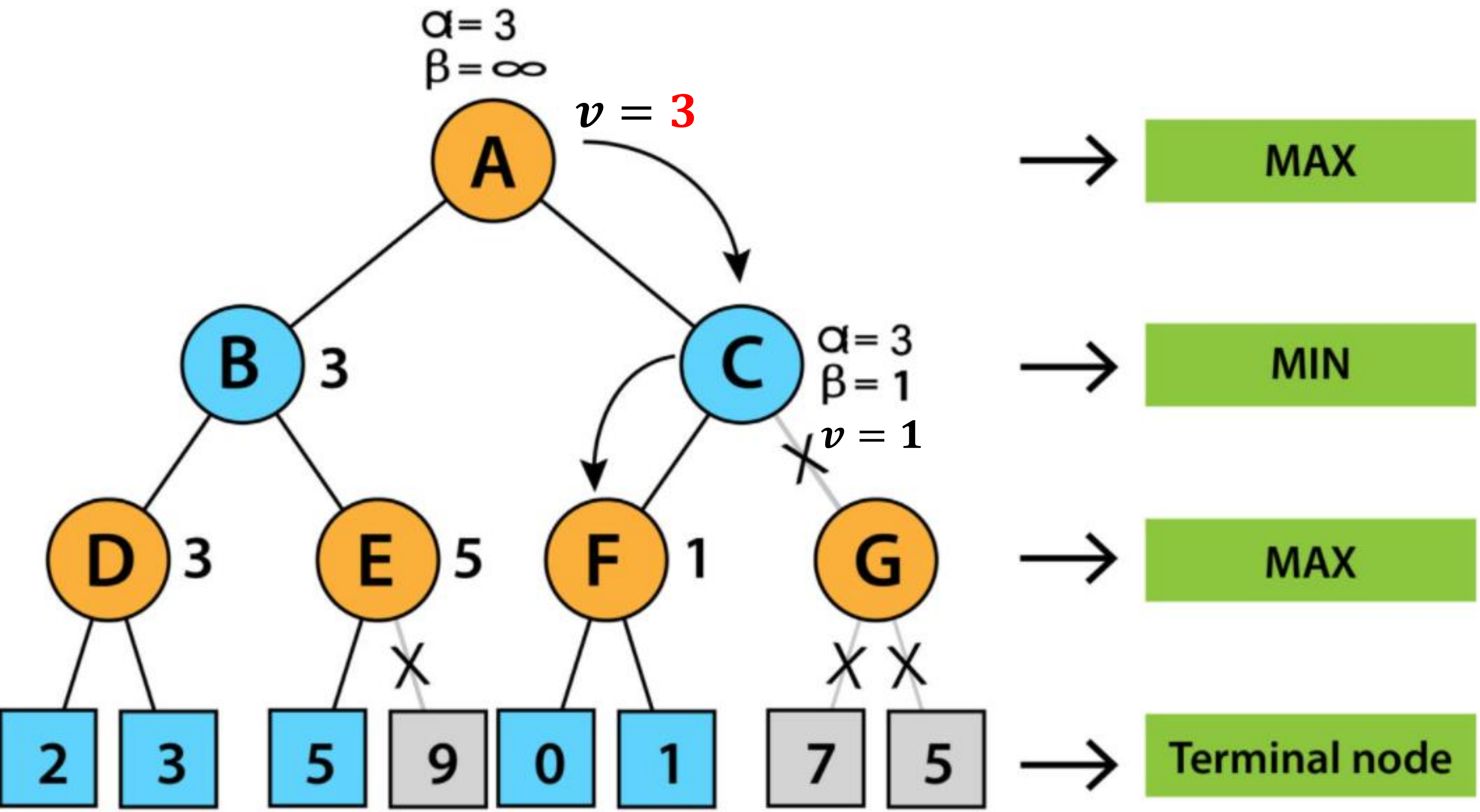
Alpha-Beta Pruning Example



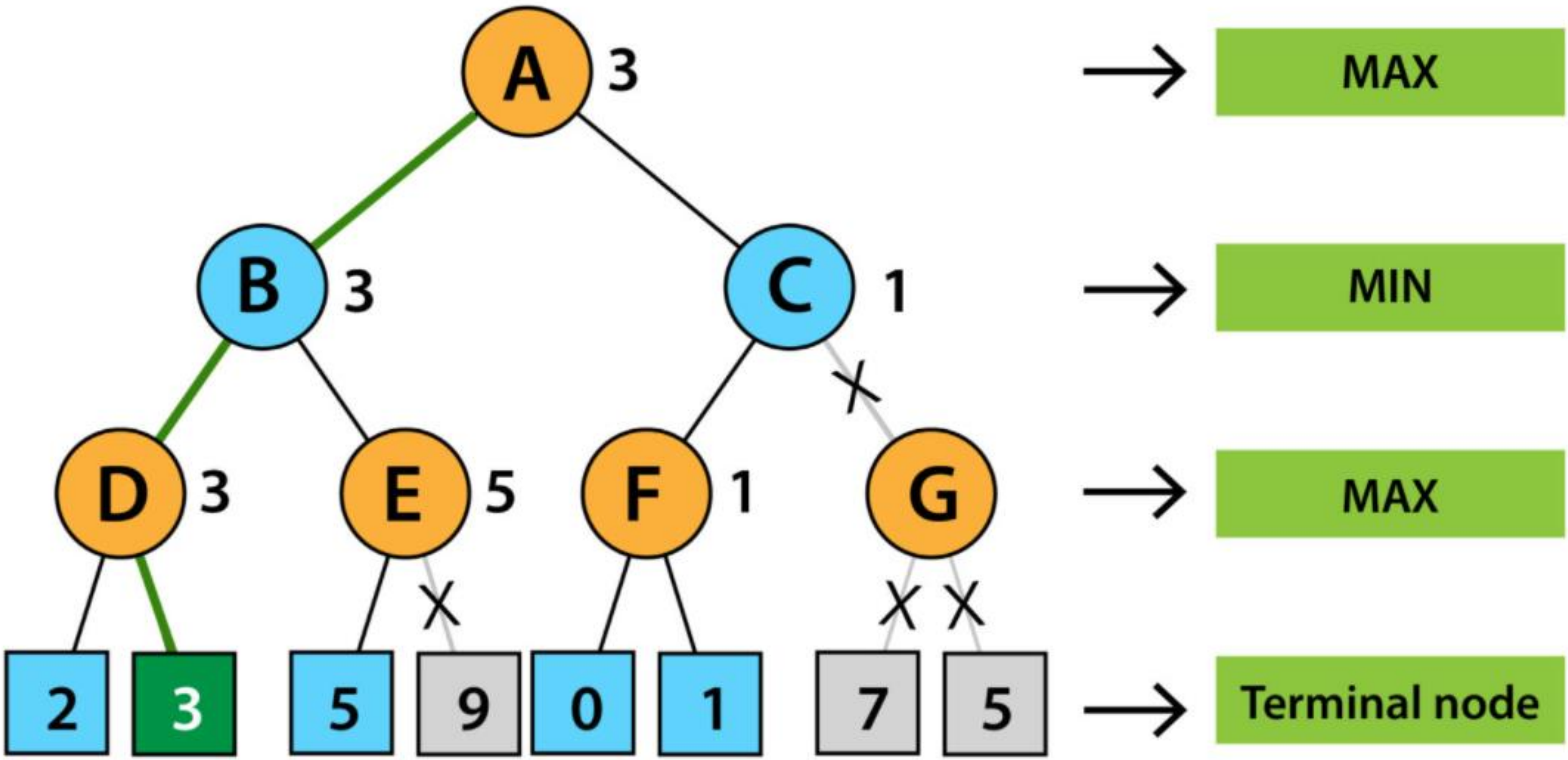
Alpha-Beta Pruning Example



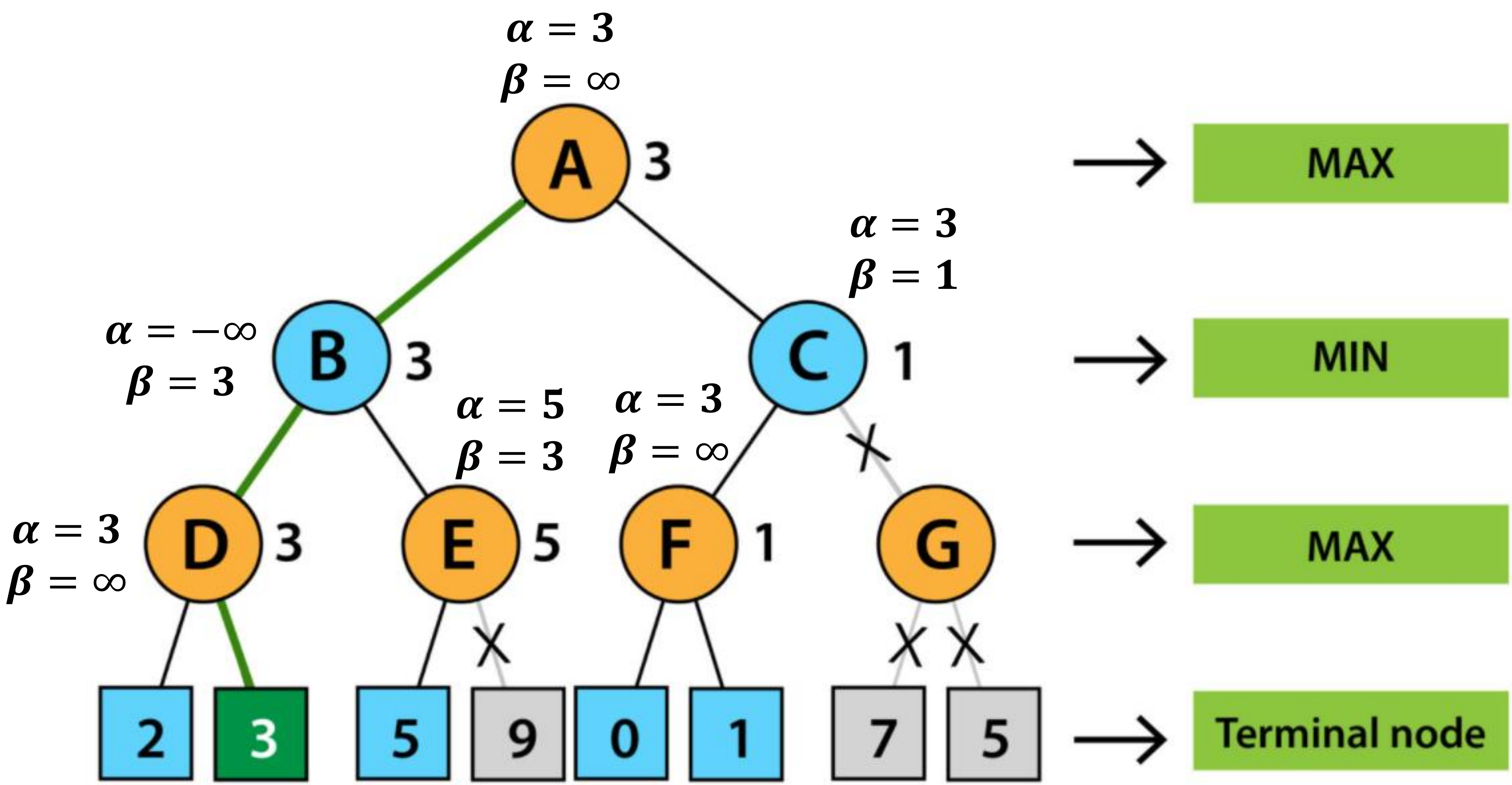
Alpha-Beta Pruning Example



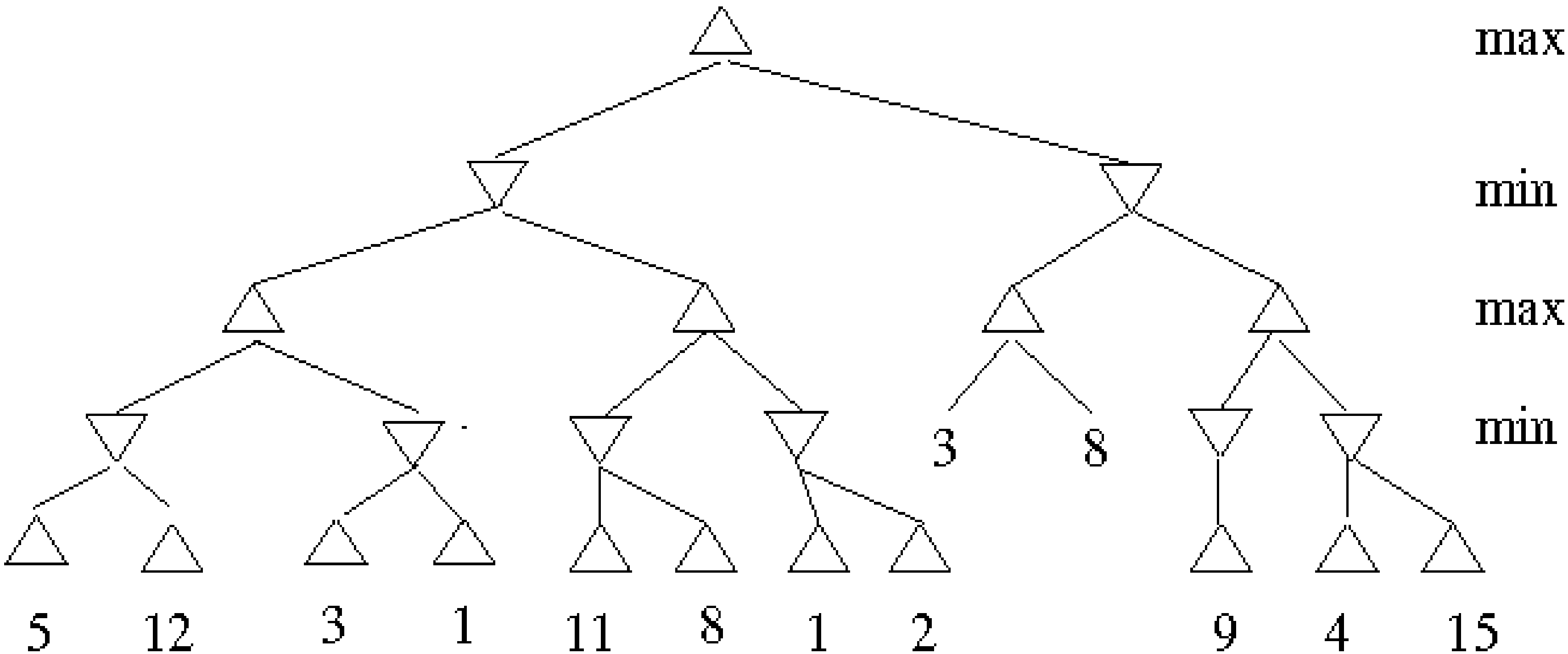
Alpha-Beta Pruning Example



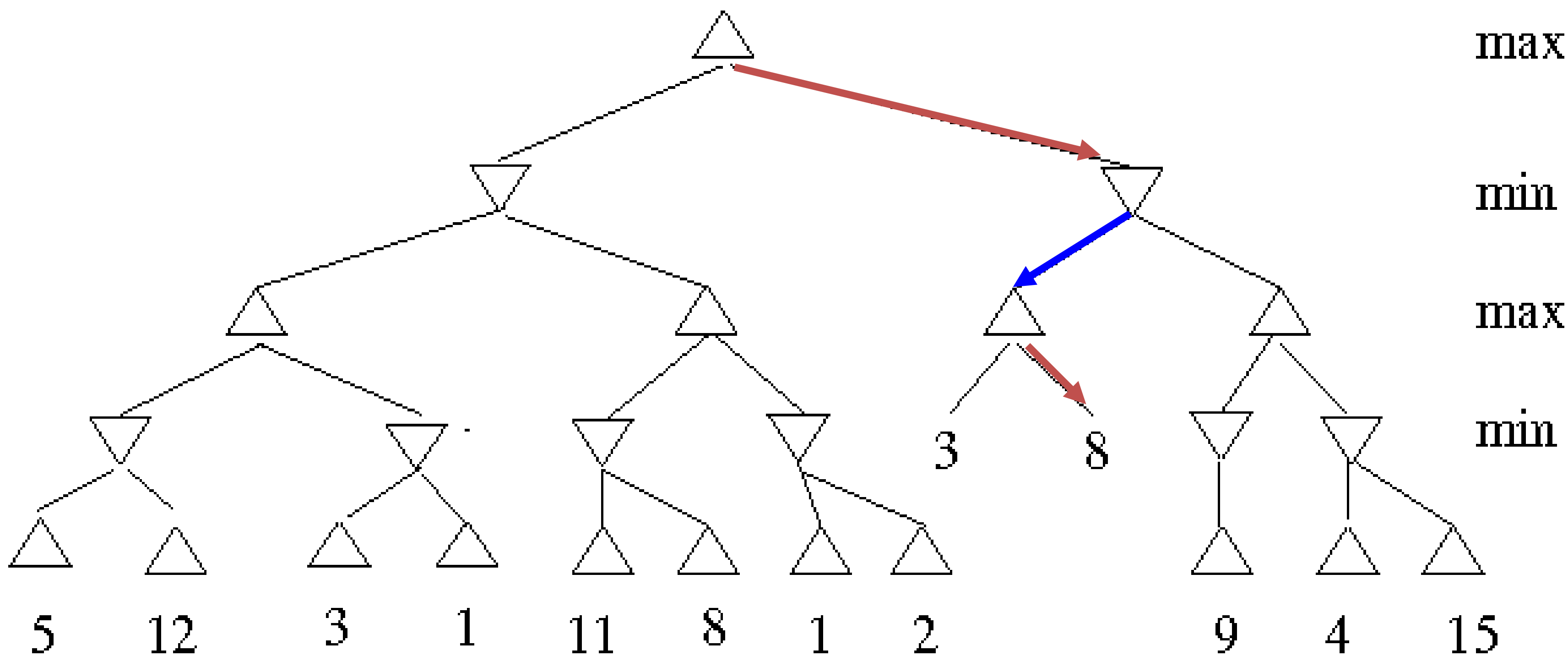
Alpha-Beta Pruning Example



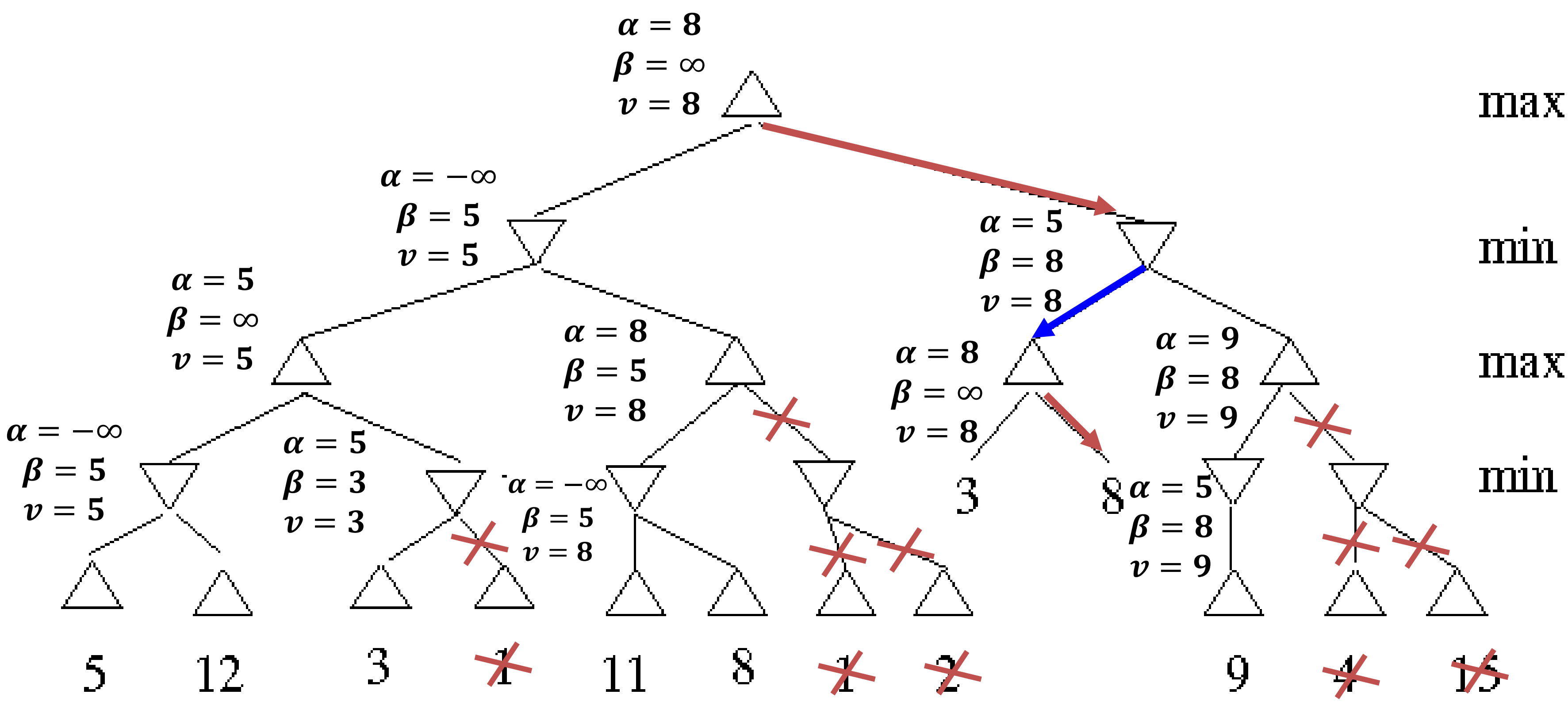
Exercise



Exercise: Solution



Exercise



α = 8
β = ∞
v = 8
max
min
max
min
α = −∞
β = 5
v = 5
α = 5
β = 8
v = 8
α = 5
β = ∞
v = 5
α = 8
β = 5
v = 8
α = 8
β = ∞
v = 8
α = 9
β = 8
v = 9
α = −∞
β = 5
v = 5
α = 5
β = 3
v = 3
α = −∞
β = 5
v = 8
α = 5
β = 8
v = 9
3
8
5
12
3
1
11
8
1
2
9
4
15

Properties of Alpha-Beta Algorithm

- Pruning has no effect on the minimax values.
- Entire subtrees can be pruned, not just leaves.
- Good move ordering improves effectiveness of pruning
- Completeness and optimality are preserved from Minimax.

- Time complexity:
  - The effectiveness depends on the order in which the states are examined.
  - If states could be examined in perfect order, then alpha-beta search examines only $O(b^{m/2})$ nodes to pick the best move, vs. $O(b^m)$ for minimax.
  - If states could be examined in random order rather than perfect order, the total number of nodes examined will be roughly $O(b^{3m/4})$ for moderate b.

- Space complexity: O(m), as for Minimax.

Game Tree Size

E.g., Chess:

- $b \approx 35$ **(approximate average branching factor)**
- $d \approx 100$ **(depth of a game tree for typical games)**
- $b^d \approx 35^{100} \approx 10^{154}$
- **For alpha-beta search and perfect ordering, we still get** $b^{d/2} \approx 35^{50} \approx 10^{77}$

Finding the exact solution is completely infeasible!

Is this practical?

- Minimax and alpha-beta pruning still have exponential complexity.
- May be impractical within a reasonable amount of time.
- Solution: cut the search earlier
  - replace the Utility function with a heuristic evaluation function that estimates the state utility.
  - replace the Terminal-test by a cutoff test that decides when to stop expanding a state.

Cutting off search

- Introduces a fixed-depth limit *d*
  - Selected so that the amount of time will not exceed what the rules of the game allow.
- When cutoff occurs, the evaluation is performed.
- Heuristic MiniMax for state *s* and maximum depth *d*

```
H-MiniMax(s, d) =

if Cutoff-Test(s, d) then Eval(s)
if Player(s) = MAX then
    max of H-MiniMax(Result(s, a), d+1) for a in Actions(s)
if Player(s) = MIN then
    min of H-MiniMax(Result(s, a), d+1) for a in Actions(s)
```

Evaluation Function

- **Idea: produce an estimate of the expected utility of the game from a given position.**
- **Performance depends on quality of Eval.**
- **Requirements:**
  - **must be consistent with the utility function**
    - **states that are wins must evaluate better than draws, which in turn must be better than losses.**
  - **tradeoff between accuracy and time cost**
    - **without time limits, minimax could be used**
  - **should reflect the actual chances of winning for non-terminal states**

- **Frequently weighted linear functions are used**
  - $E = w_1 f_1 + w_2 f_2 + \ldots + w_n f_n$
  - **combination of features ($f_{1,,,}\ f_n$), weighted by their relevance ($w_{1,\ldots,}\ w_n$)**
  - **more important features get more weight**

The horizon effect

- Evaluations functions are always imperfect.

- If not looked deep enough, bad moves may appear as good moves (as estimated by the evaluation function) because their consequences are hidden beyond the search horizon.
  - and vice-versa!

- Often, the deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters.

Summary

- Game Tree
- Alpha-Beta Pruning
- Different Games

Midterm Exam

- 0am 02/29 – 11:59pm 03/01 under “Exams” of Modules on Canvas
- Time duration: 3 hours
- Open-book and open-note

- Total points: 100
- Five Problems
- 1) Search problem: BFS, DFS, UCS, Greedy, A*
- 2) Search Problem
- 3) Local search- simulated annealing
- 4) Constraint satisfaction problem
- 5) Games: minimax

What I want you to do

- **Review Chapter 5**
- **Work on your assignment**