Artificial Intelligence

CSE 440/EEE 333/ETE333

Chapter 5
Fall 2017

Mirza Mohammad Lutfe Elahi
Department of Electrical and Computer Engineering
North South University

Generalizing Search Problems

- So far: our search problems have assumed agent has complete control of environment
  - state does not change unless the agent (robot) changes it.
    - All we need to compute is a single path to a goal state.
- Assumption not always reasonable
  - stochastic environment (e.g., the weather, traffic accidents).
  - other agents whose interests conflict with yours
  - Problem: you might not traverse the path you are expecting.

Generalizing Search Problems

- In these cases, we need to generalize our view of search to handle state changes that are not in the control of the agent.

- One generalization yields game tree search
  - agent and some other agents.
  - The other agents are acting to maximize their profits
    - this might not have a positive effect on your profits.

More General Games

- What makes something a game?
  - there are two (or more) agents influencing state change
  - each agent has their own interests
    - e.g., goal states are different; or we assign different values to different paths/states
  - Each agent tries to alter the state so as to best benefit itself.

- What makes games hard?
  - how you should play depends on how you think the other person will play; but how they play depends on how they think you will play; so how you should play depends on how you think they think you will play; but how they play should depend on how they think you think they think you will play; …

Games: Language/Functions

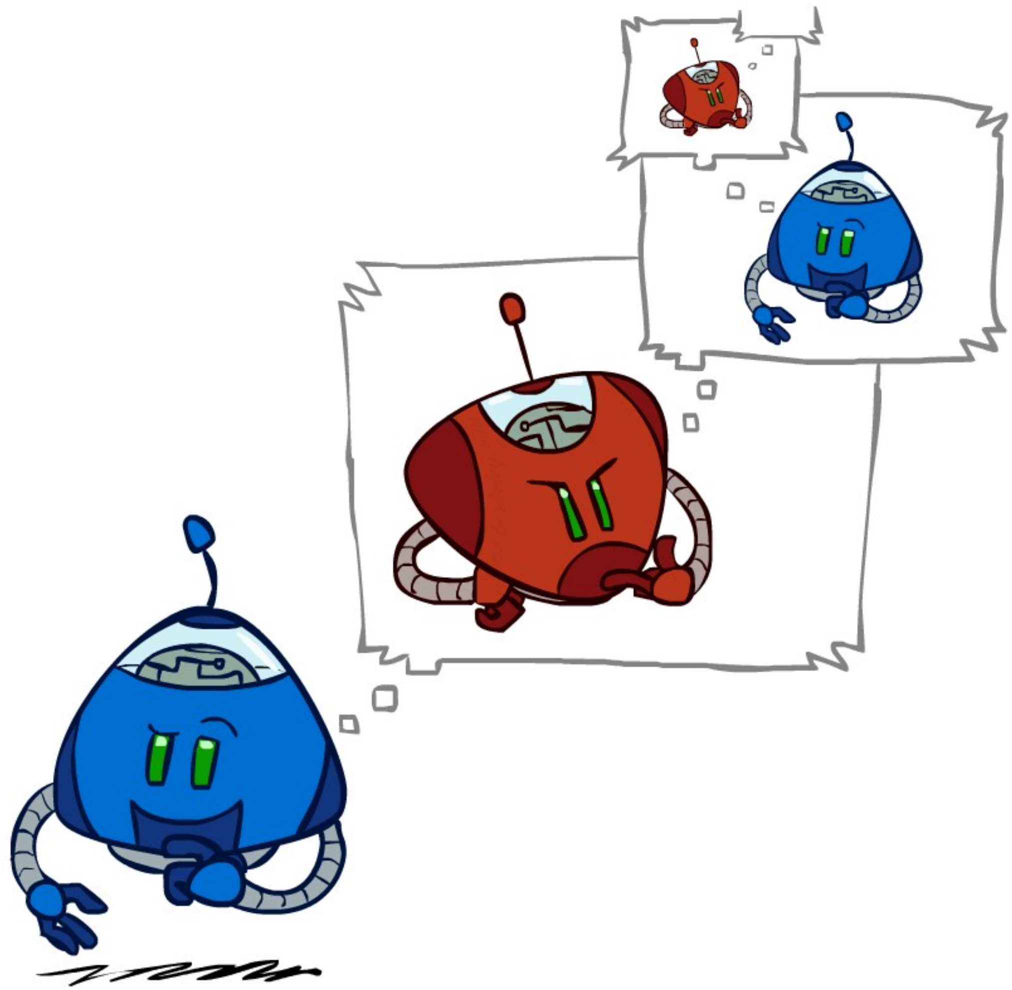
- S_0 //initial state
- **PLAYERS(s)** // who's the player in state ***s***
- **ACTIONS(s)** // possible moves from state ***s***
- **RESULT(s, a)** // the state after action ***a*** is taken on state ***s***
- **TERMINAL-TEST(s)** // returns true if ***s*** is a terminal state
- **UTILITY(s, p)** // the objective function in state ***s*** for player ***p***

Zero-Sum Games

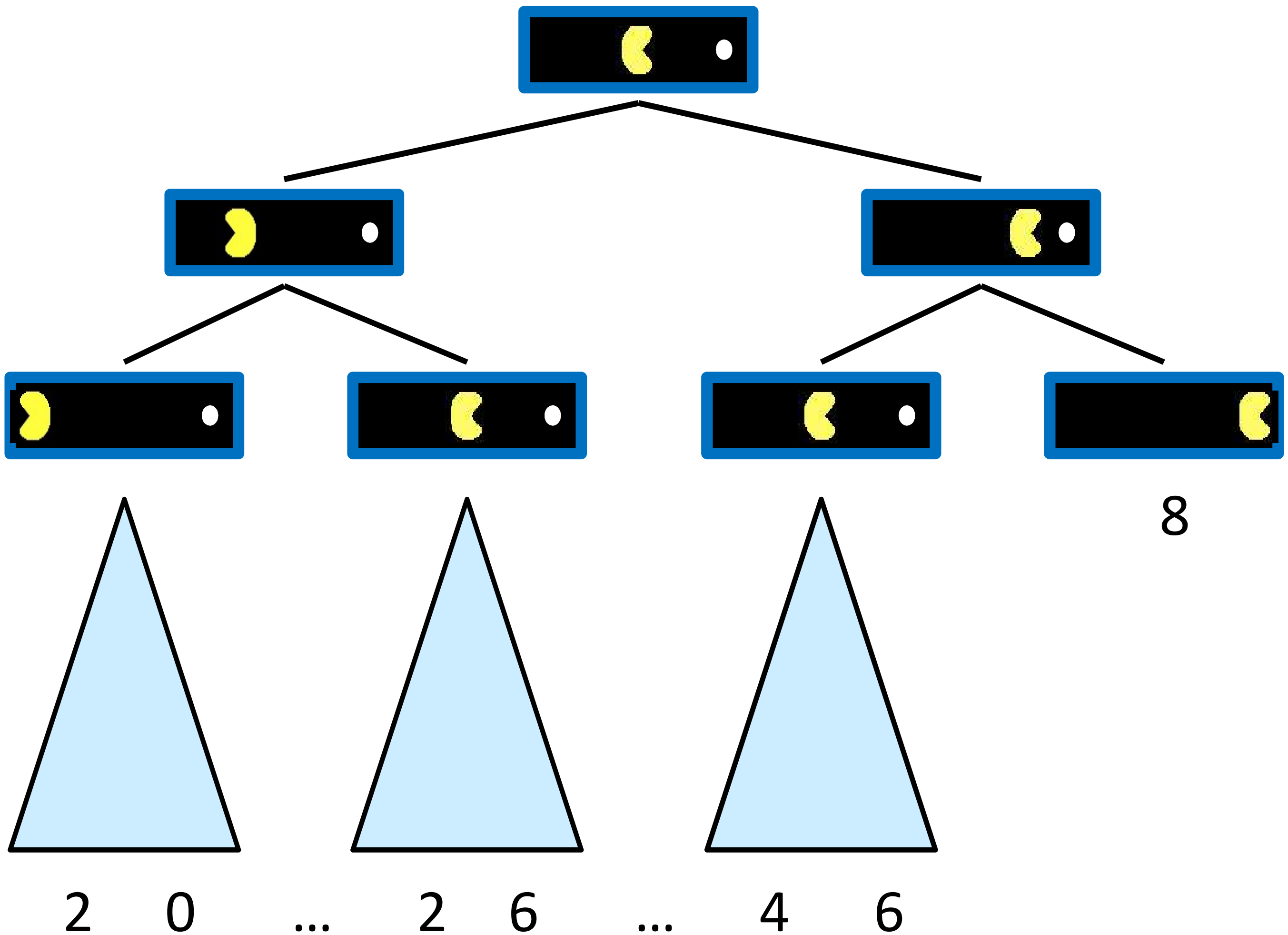
- **Zero-Sum Games**
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

- **General Games**
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, and more are all possible
  - More later on non-zero-sum games

Adversarial Search



Single-Agent Trees

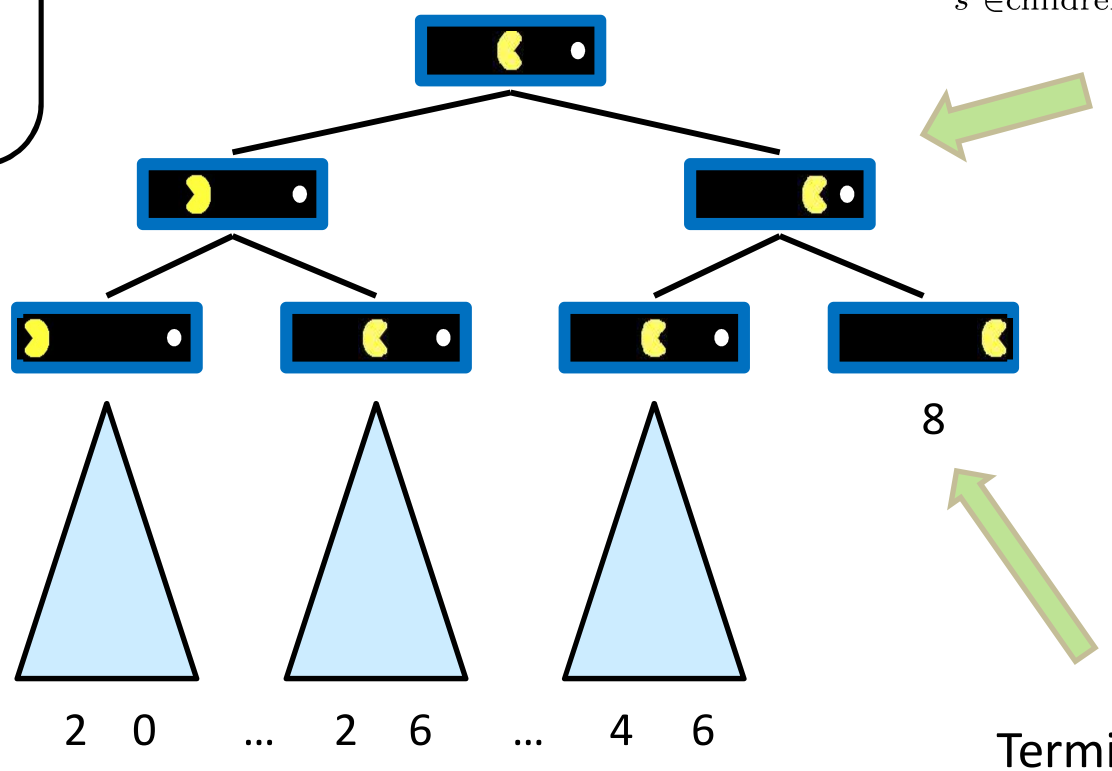


Value of a State

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

Non-Terminal States:

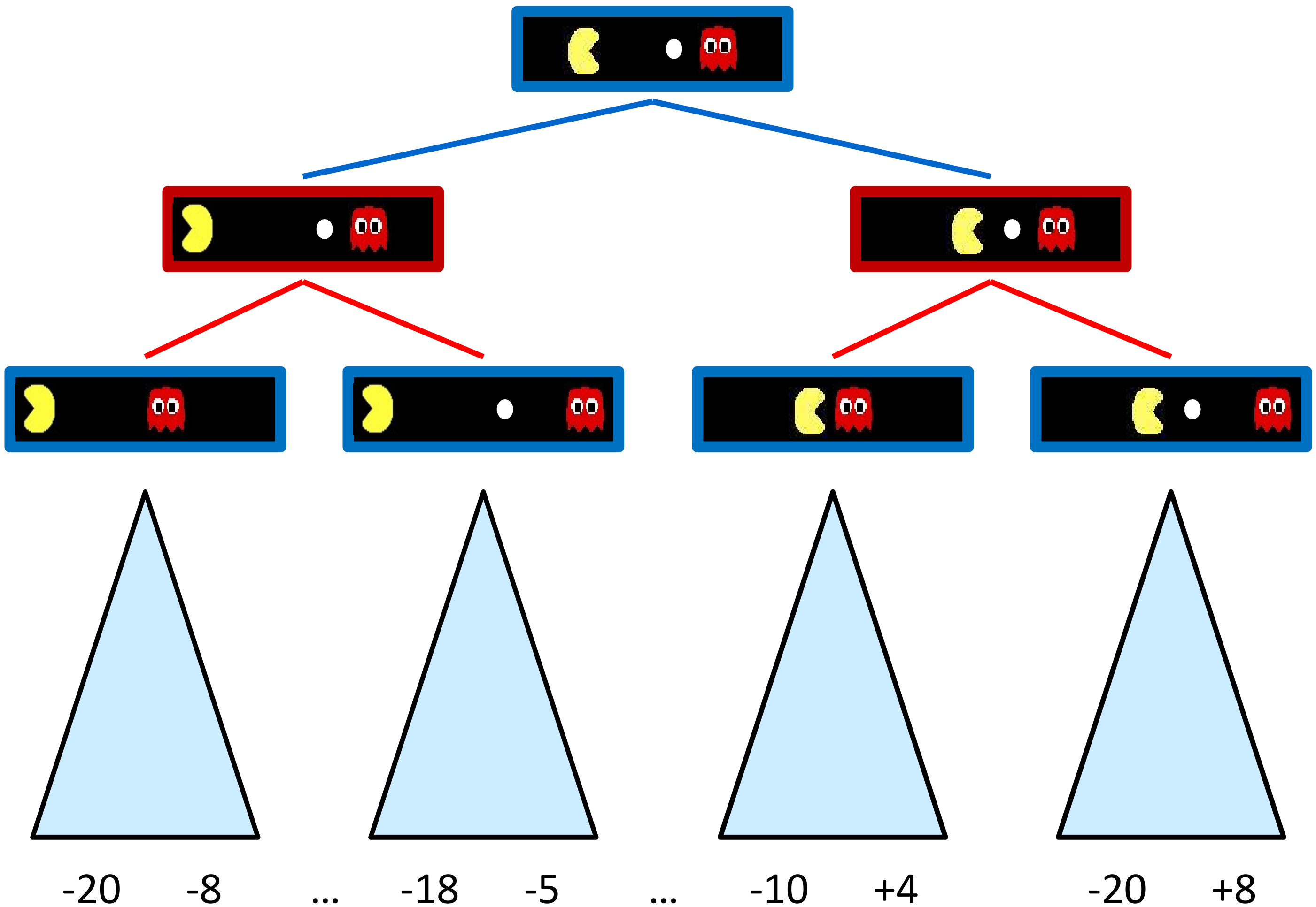
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

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

Adversarial Game Trees



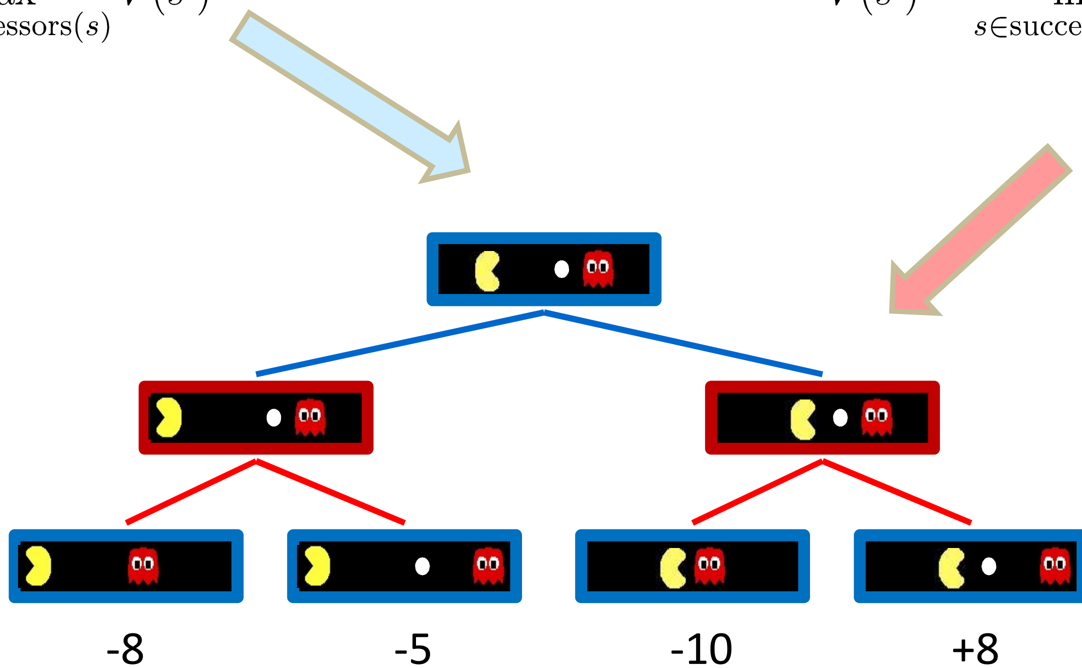
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

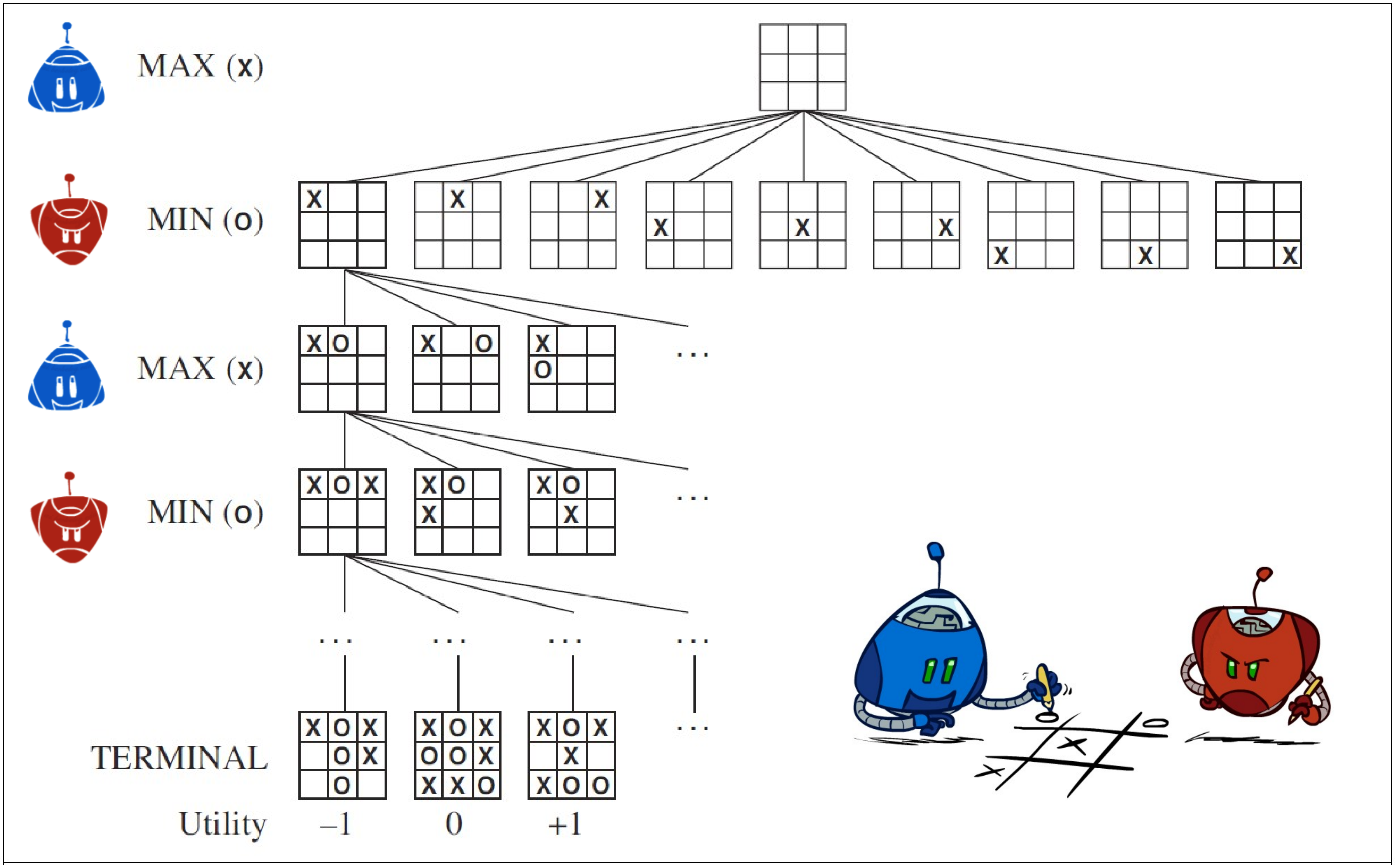
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$



Terminal States:

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

Tic-Tac-toe Game Tree



Game Playing Problem

- Instance of the general search problem.

- States where the game has ended are called **terminal states**.

- A **utility (payoff) function** determines the value of **terminal states**, e.g. win = +1, draw = 0, lose = -1.

- In two-player games, assume one is called **MAX** (tries to maximize utility) and one is called **MIN** (tries to minimize utility).

- In the search tree, first layer is move by **MAX**, next layer by **MIN**, and alternate to terminal states.

- Each layer in the search is called a **ply**

Optimal Decision: Optimal Players

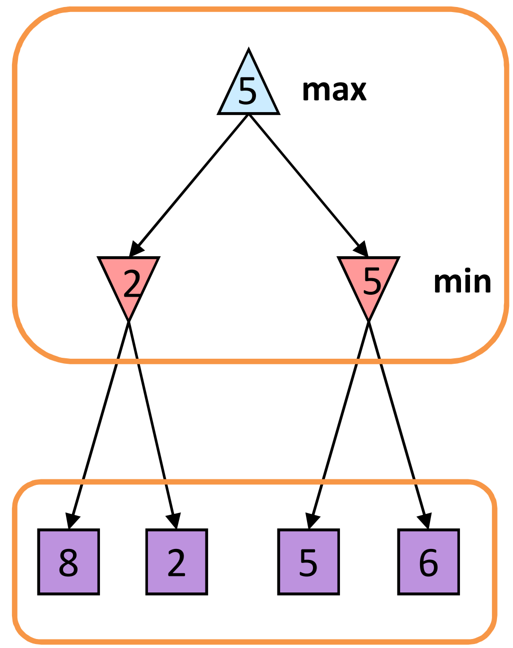
- A game not only finds best way to goal
- The other player has a say
- **Two players:** MAX and MIN
- **ACTIONS(s)** and **RESULTS(s, a)** define a game tree
- **Tic Tac Toe:** Fewer than 9!(3,62,880) terminal nodes
- **Chess:** over 10^{40}
- **Search tree** as theory

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

- General method for determining optimal move.
- Generate complete game tree down to terminal states.
- Compute utility of each node bottom up from leaves toward root.
- At each MAX node, pick the move with maximum utility.
- At each MIN node, pick the move with minimum utility (assumes opponent always acts correctly to minimize utility).
- When reach the root, optimal move is determined.

Minimax algorithm

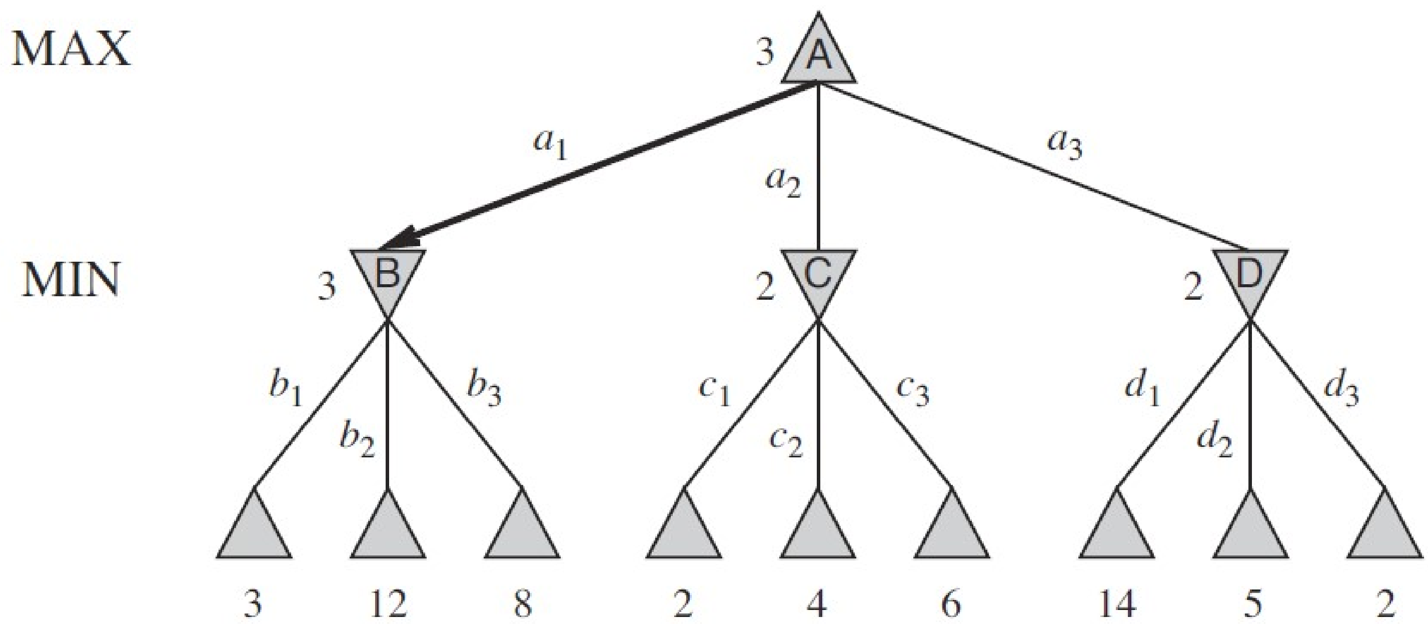
```
function MINIMAX-DECISION(state) returns an action
   inputs: state, current state in game

   return the a in ACTIONS(state) maximizing MIN-VALUE(RESULT(a, state))
```

```
function MAX-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← −∞
   for a, s in SUCCESSORS(state) do v ← MAX(v, MIN-VALUE(s))
   return v
```

```
function MIN-VALUE(state) returns a utility value
   if TERMINAL-TEST(state) then return UTILITY(state)
   v ← ∞
   for a, s in SUCCESSORS(state) do v ← MIN(v, MAX-VALUE(s))
   return v
```

Minimax

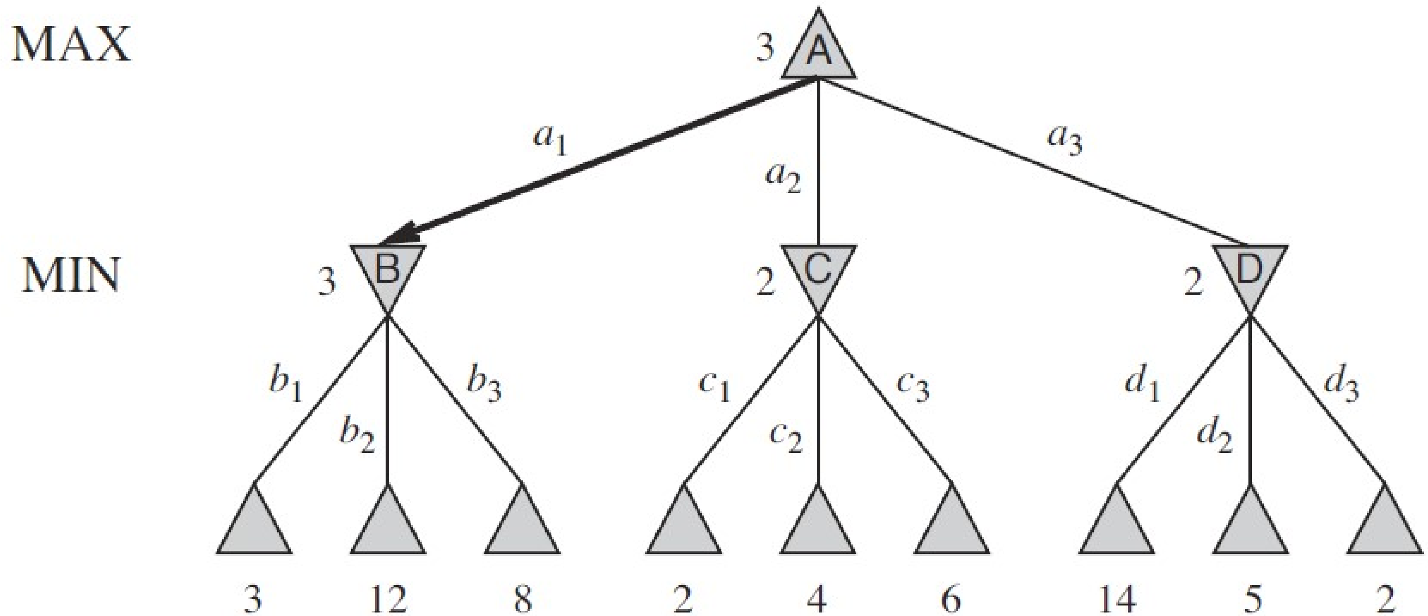
- Small 2-ply game



$$\triangle = MAX; \triangledown = MIN$$

Minimax

- Picking my best move against your best move



$$minimax(s) = \begin{cases} utility(s) & \text{if } terminal(s) \\ \max_{a \in action(s)} minimax(result(s,a)) & \text{if } player(s) = MAX \\ \min_{a \in action(s)} minimax(result(s,a)) & \text{if } player(s) = MIN \end{cases}$$

Properties of Minimax

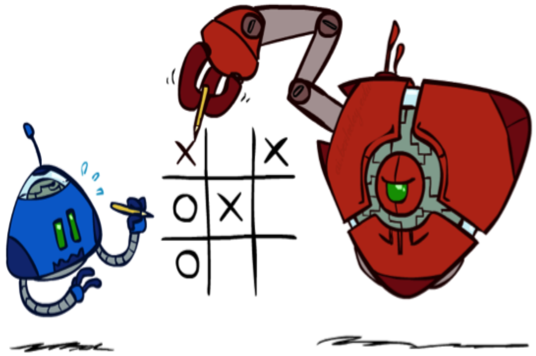
- Complete? Yes, if tree is finite (chess has specific rules for this)
- Optimal? Yes, against an optimal opponent. Otherwise??
- Time? $O(b^m)$
- Space? $O(bm)$ (depth-first exploration)

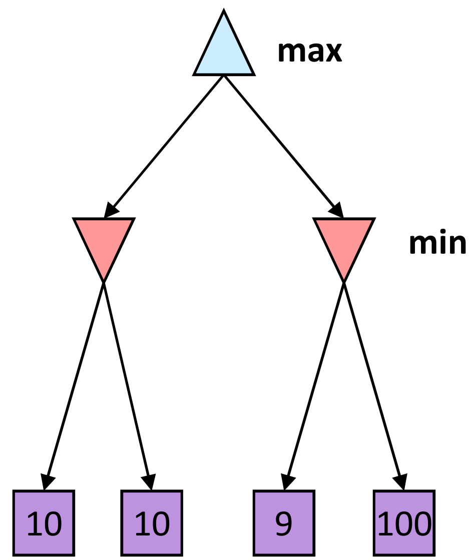
  For chess, $b \approx 35$, $m \approx 100$ for “reasonable” games

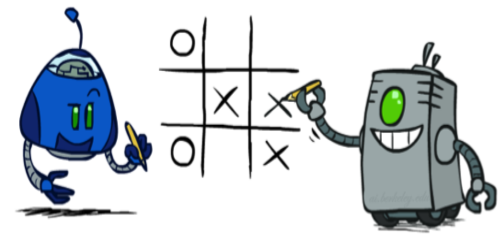
  → exact solution completely infeasible

  But do we need to explore every path?

Minimax Properties



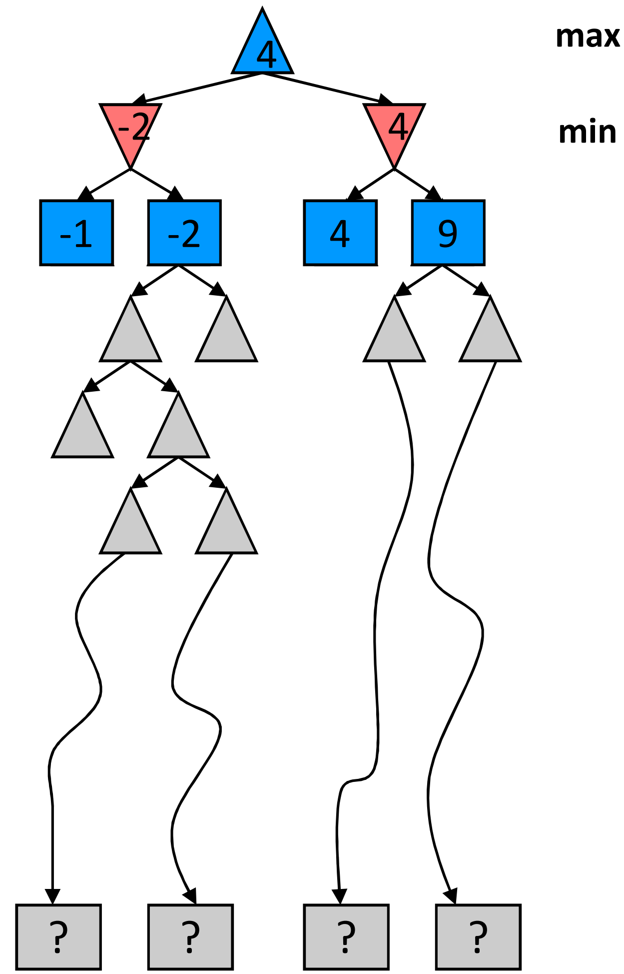




Optimal against a perfect player. Otherwise?

Resources Limits

- **Problem:** In realistic games, cannot search to leaves!
- **Solution:** Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- **Example:**
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - α-β reaches about depth 8 – decent chess program
- **Guarantee of optimal play is gone**
- **More plies makes a BIG difference**
- **Use iterative deepening for an anytime algorithm**

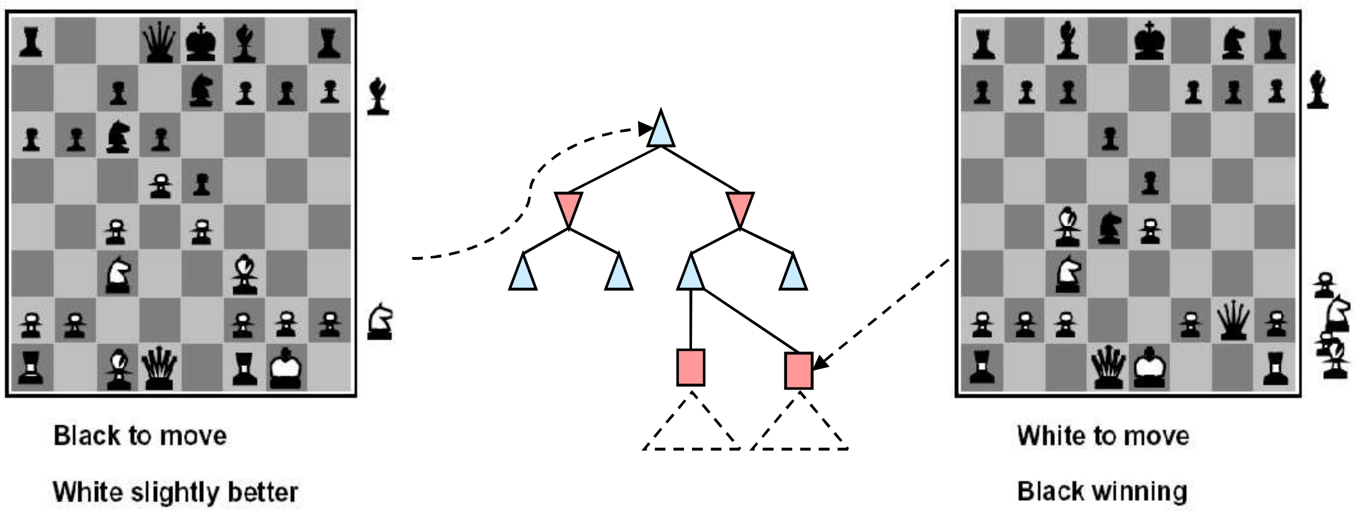


Depth Matters

- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation

Evaluation Function

- Evaluation functions score non-terminals in depth-limited search

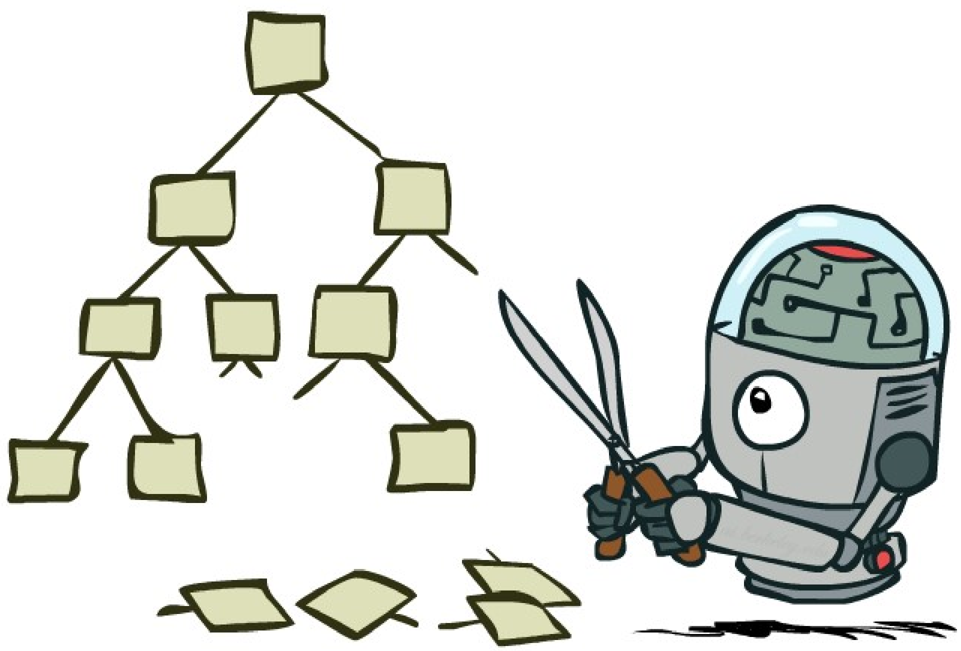


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

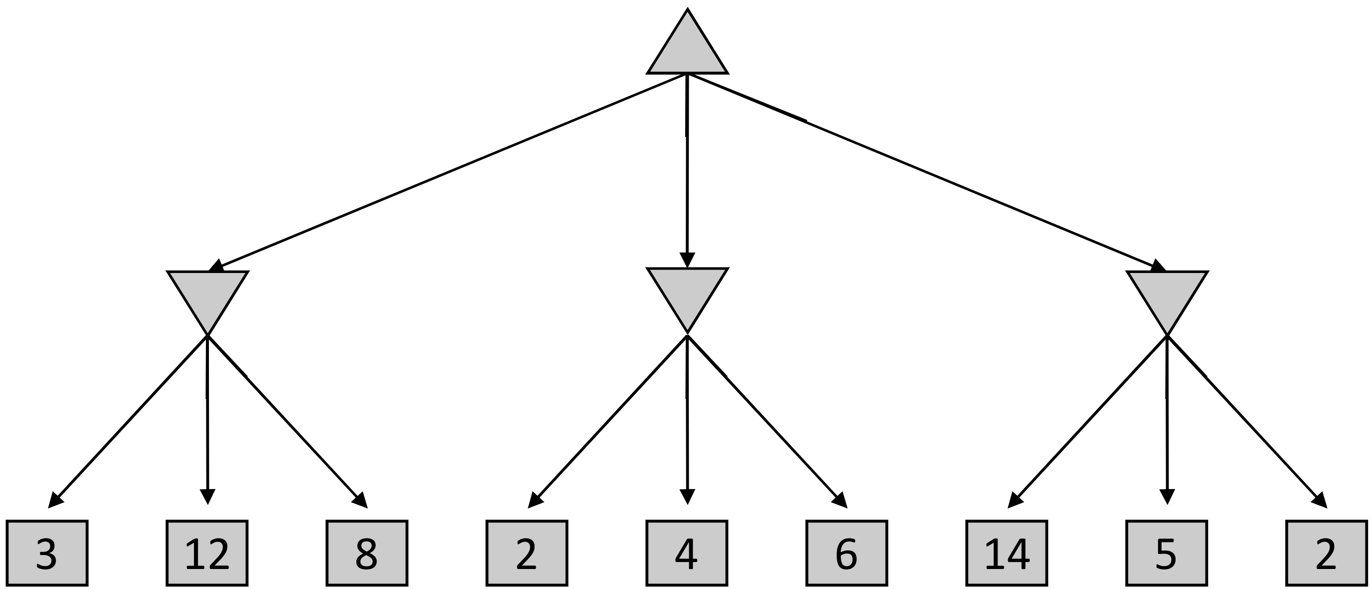
$$Eval(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

- e.g. $f_1(s)$ = (num white queens – num black queens), etc.

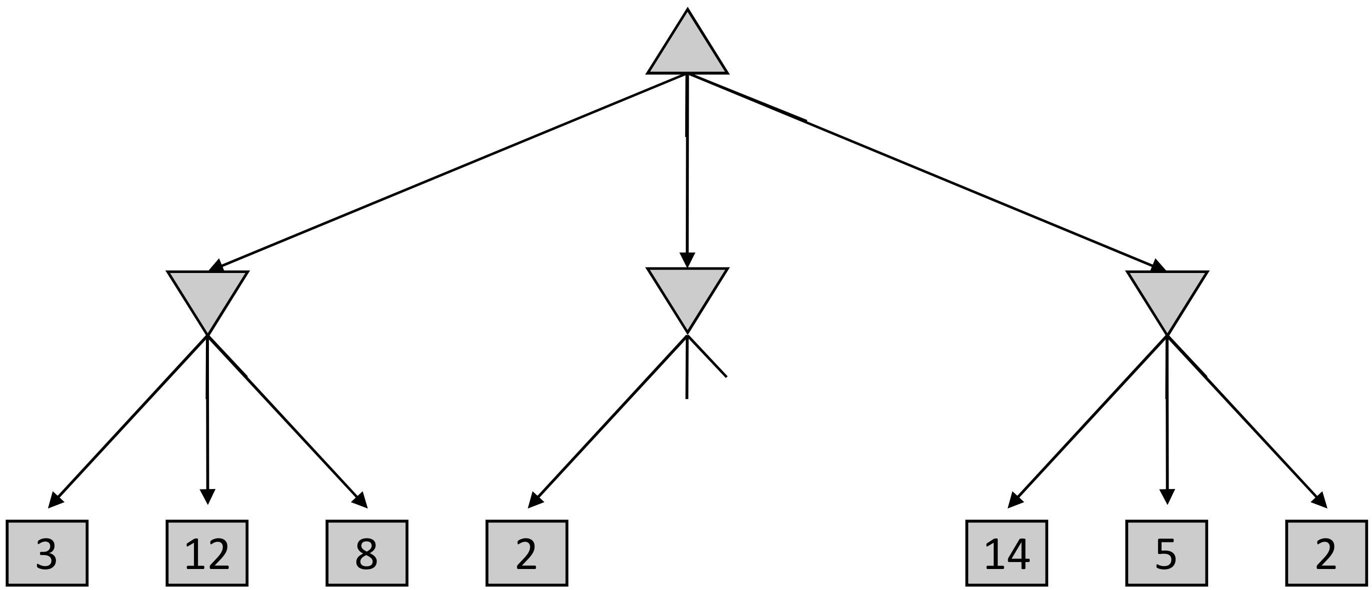
Game Tree Pruning



Minimax Example



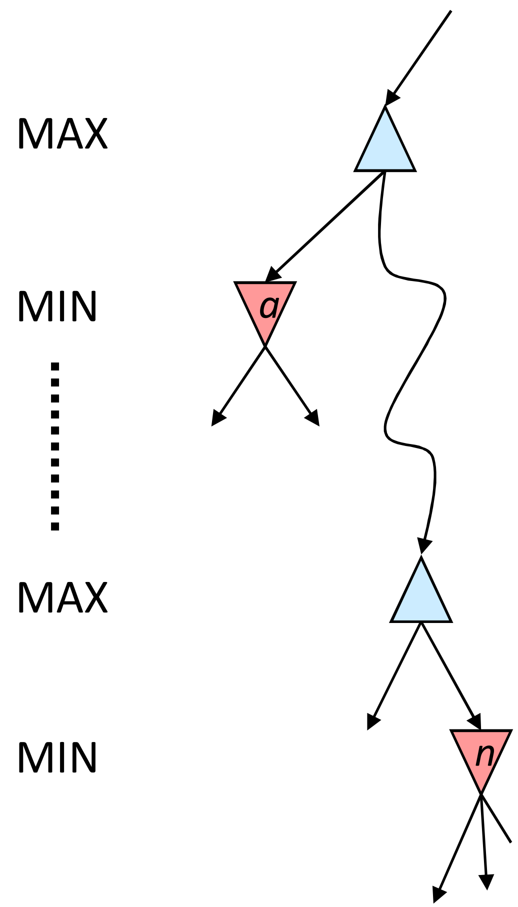
Minimax Pruning



Alpha-Beta 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)

- MAX version is symmetric



Alpha-Beta Pruning - Intuition

Two values:

- α = value of best choice so far for MAX (highest-value)
- β = value of best choice so far for MIN (lowest-value)
- Each node keeps track of its [α, β] values

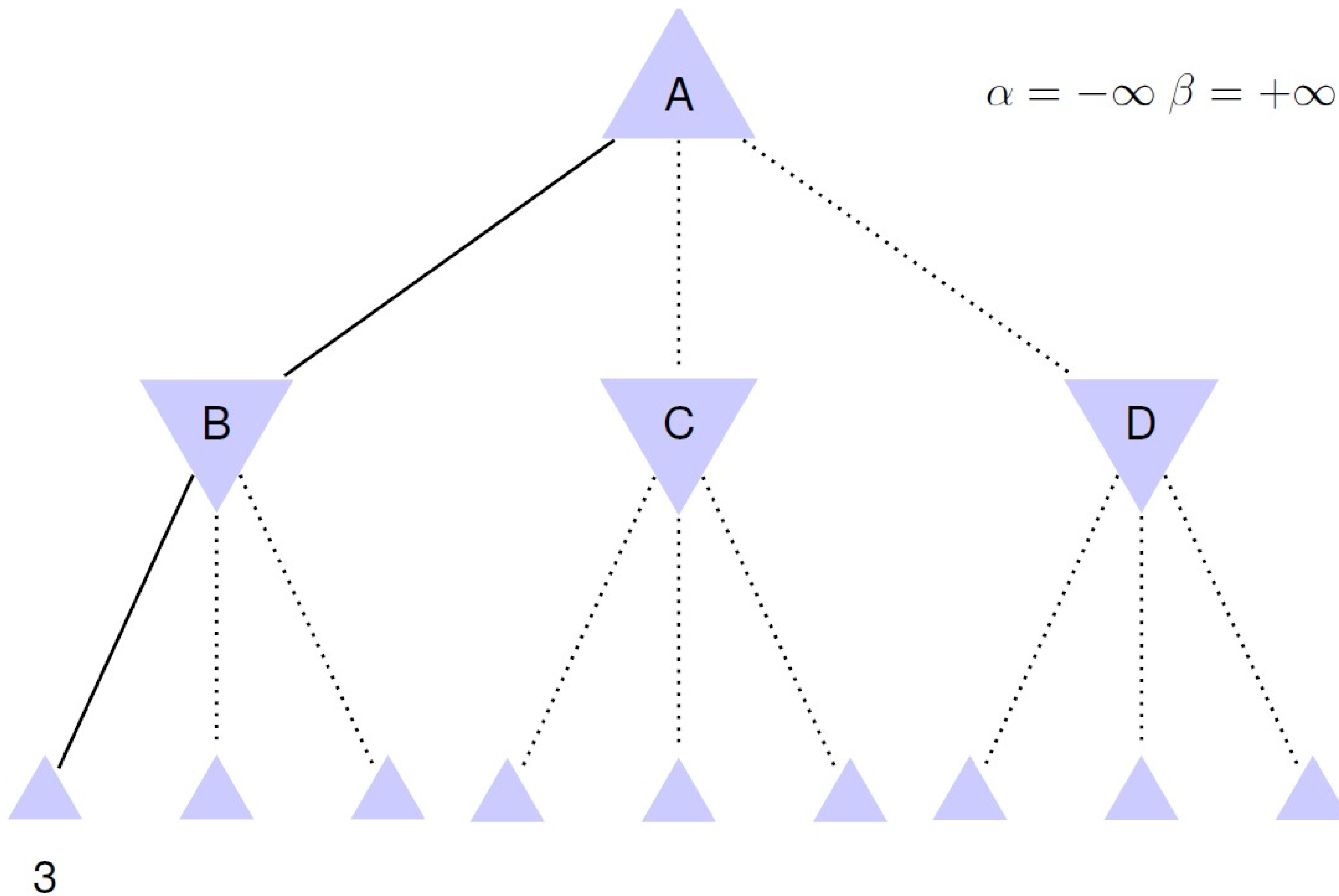
α-β pruning algorithm

```
function ALPHA-BETA-SEARCH(state) returns an action
  v ← MAX-VALUE(state, −∞, +∞)
  return the action in ACTIONS(state) with value v
```

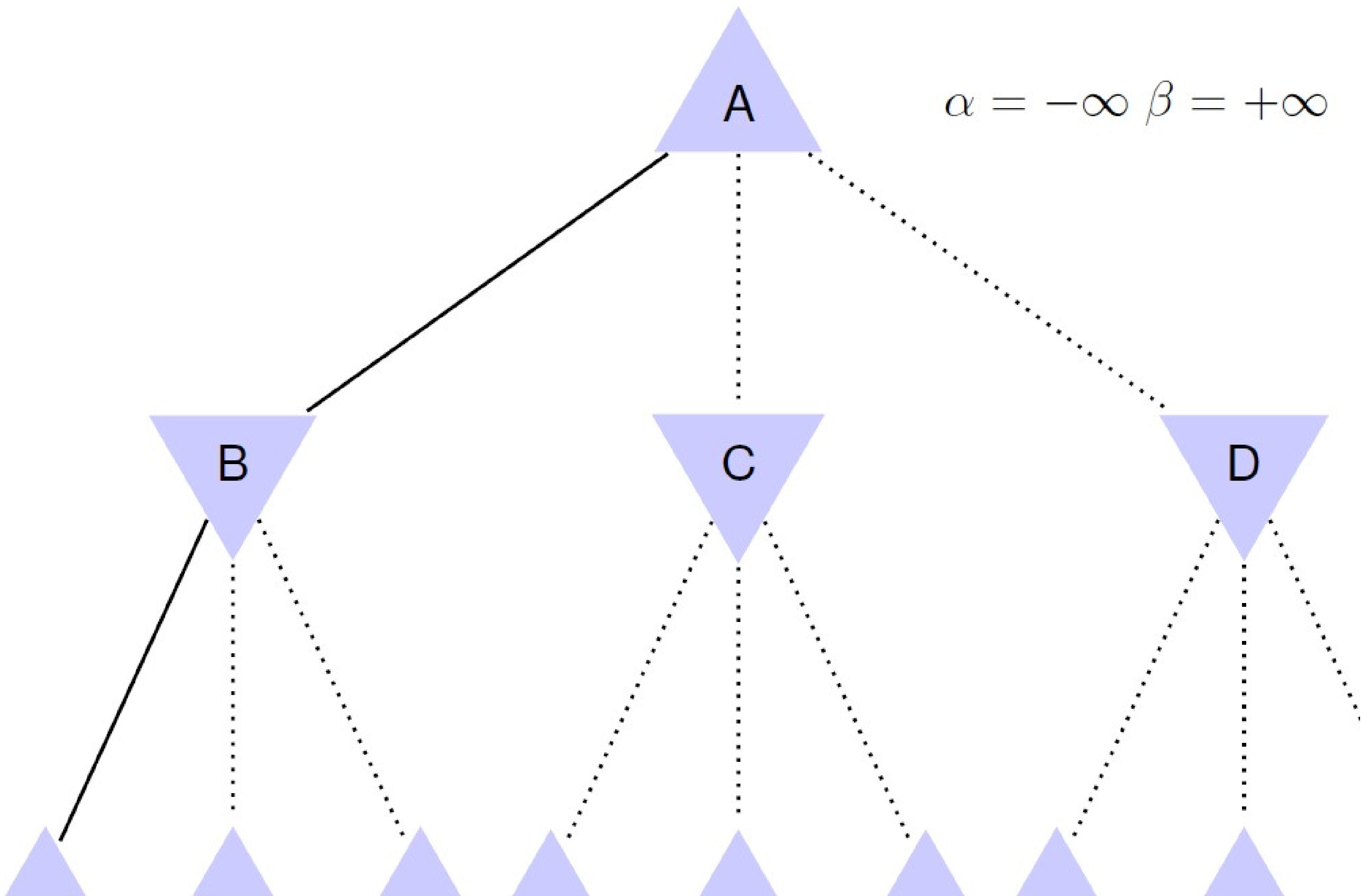
```
function MAX-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← −∞
  for each a in ACTIONS(state) do
    v ← MAX(v, MIN-VALUE(RESULT(s,a), α, β))
    if v ≥ β then return v
    α ← MAX(α, v)
  return v
```

```
function MIN-VALUE(state, α, β) returns a utility value
  if TERMINAL-TEST(state) then return UTILITY(state)
  v ← +∞
  for each a in ACTIONS(state) do
    v ← MIN(v, MAX-VALUE(RESULT(s,a) , α, β))
    if v ≤ α then return v
    β ← MIN(β, v)
  return v
```

α-β pruning example

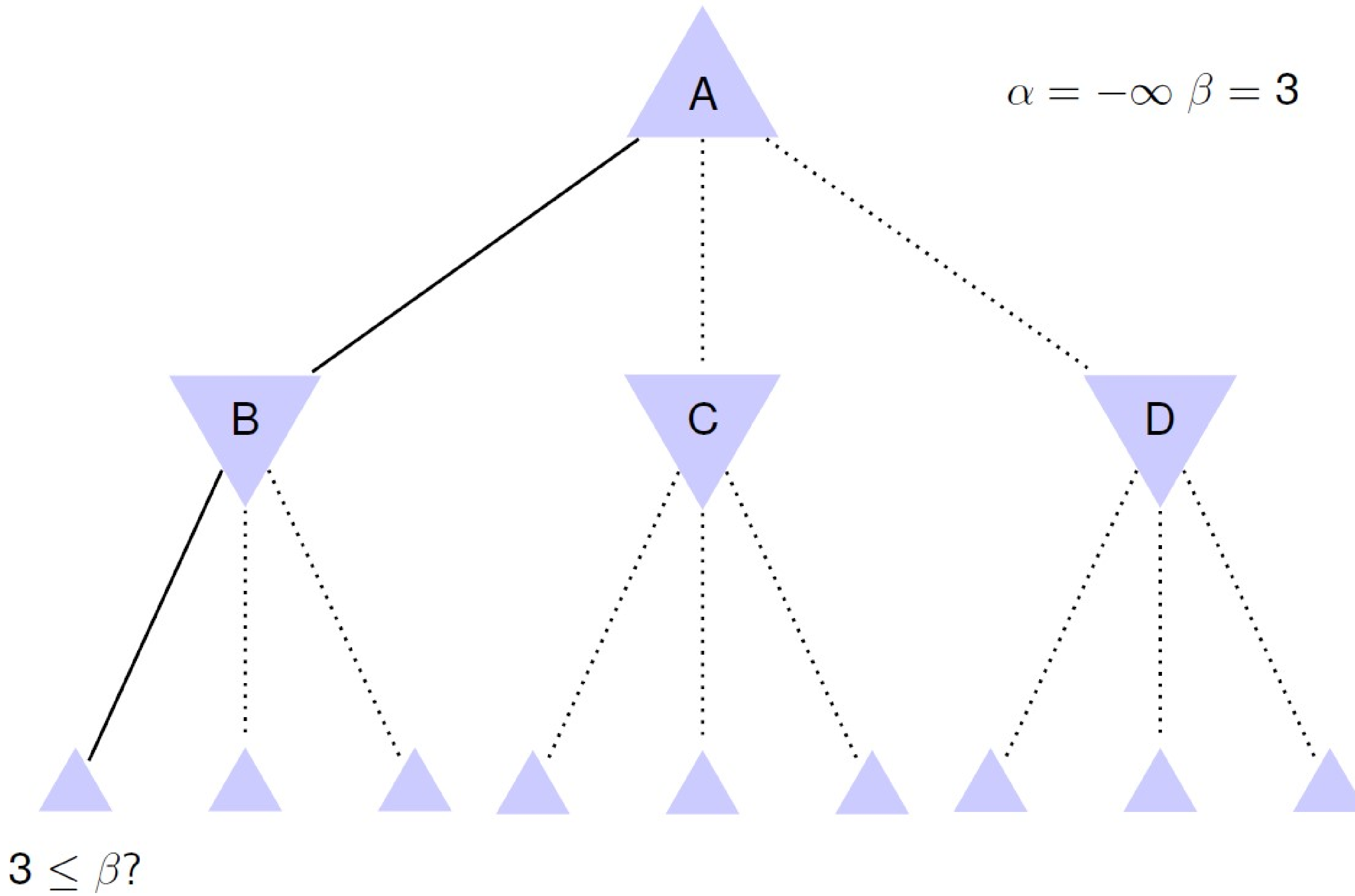


α-β pruning example

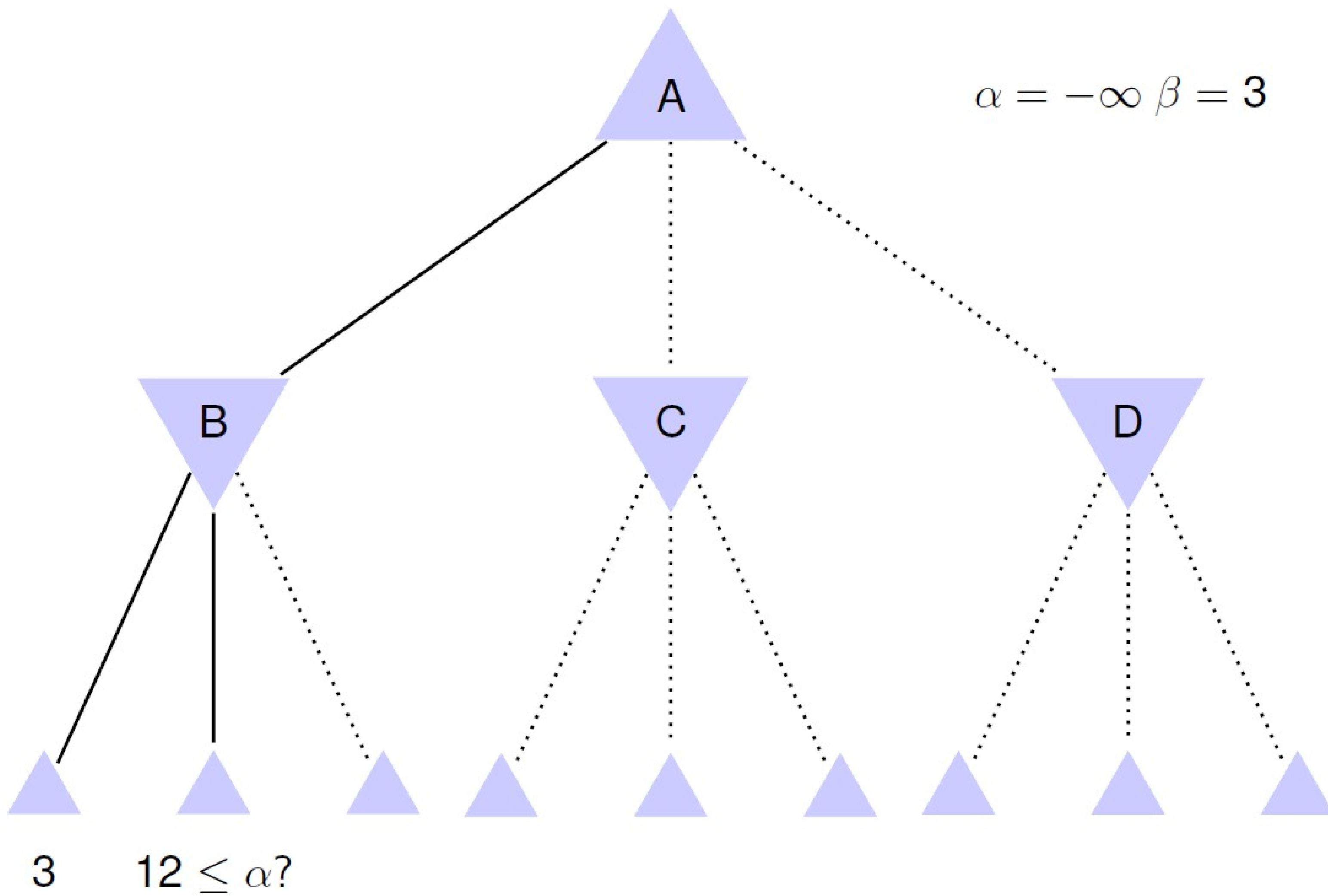


$3 \leq \alpha$?.. continue

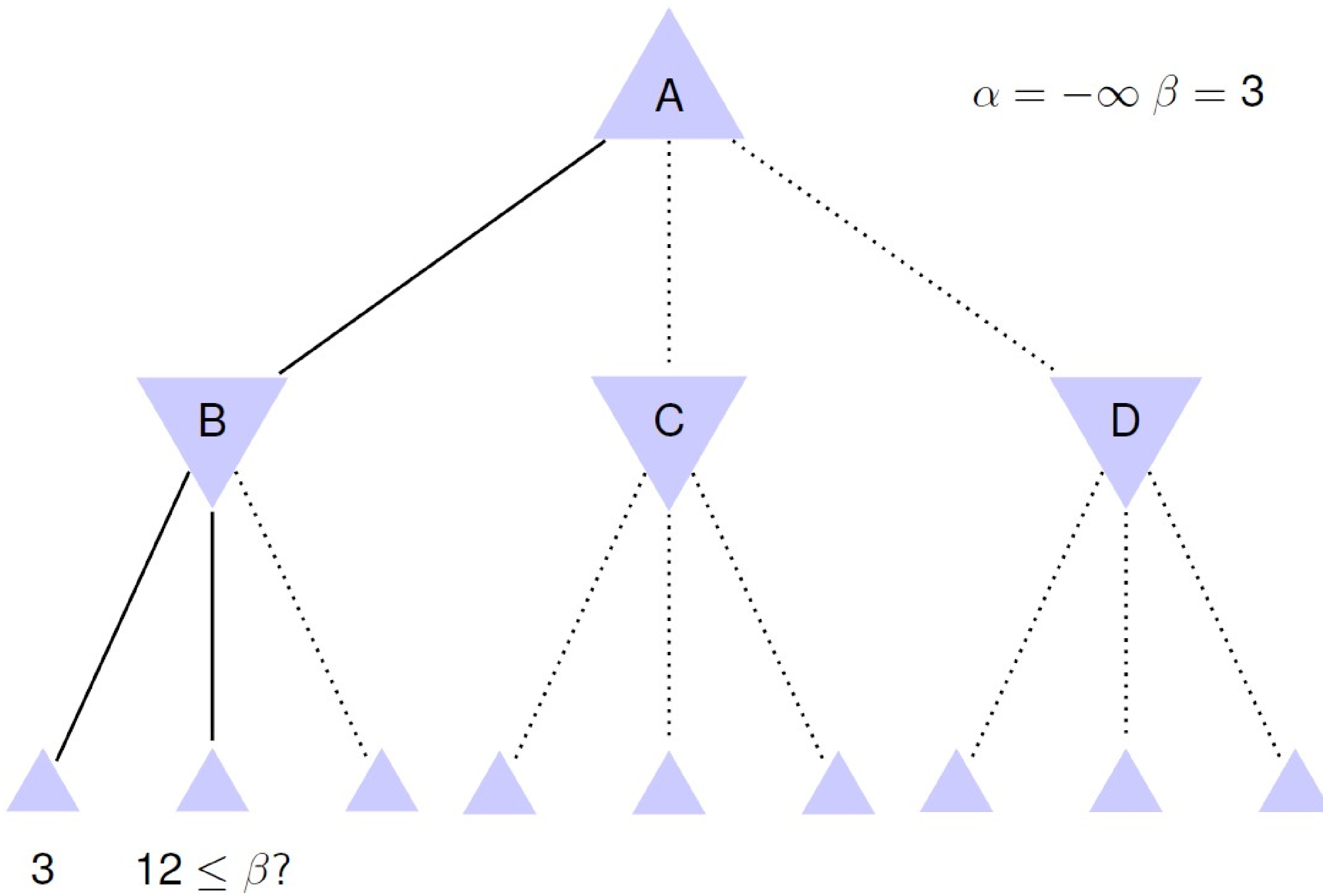
α-β pruning example



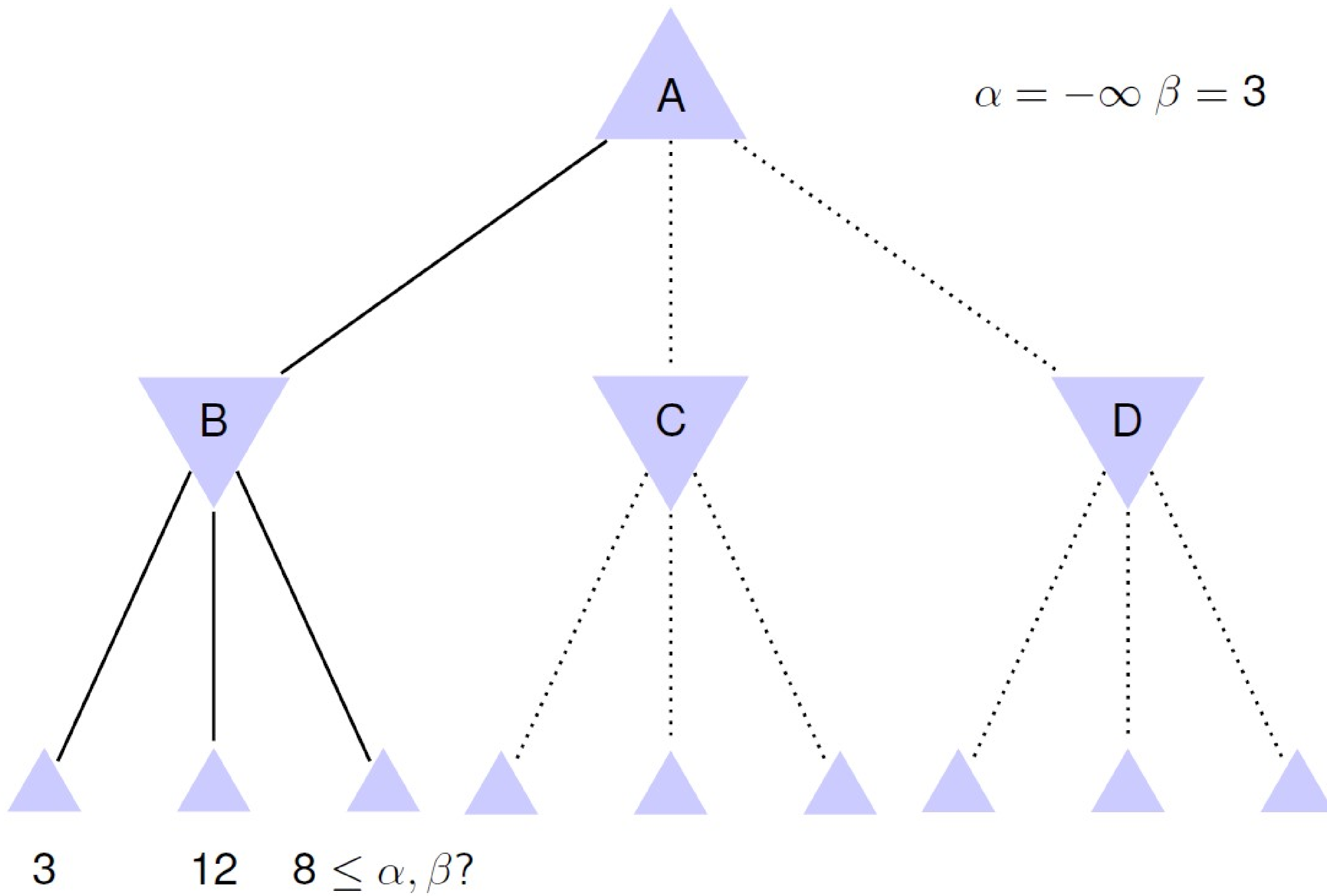
α-β pruning example



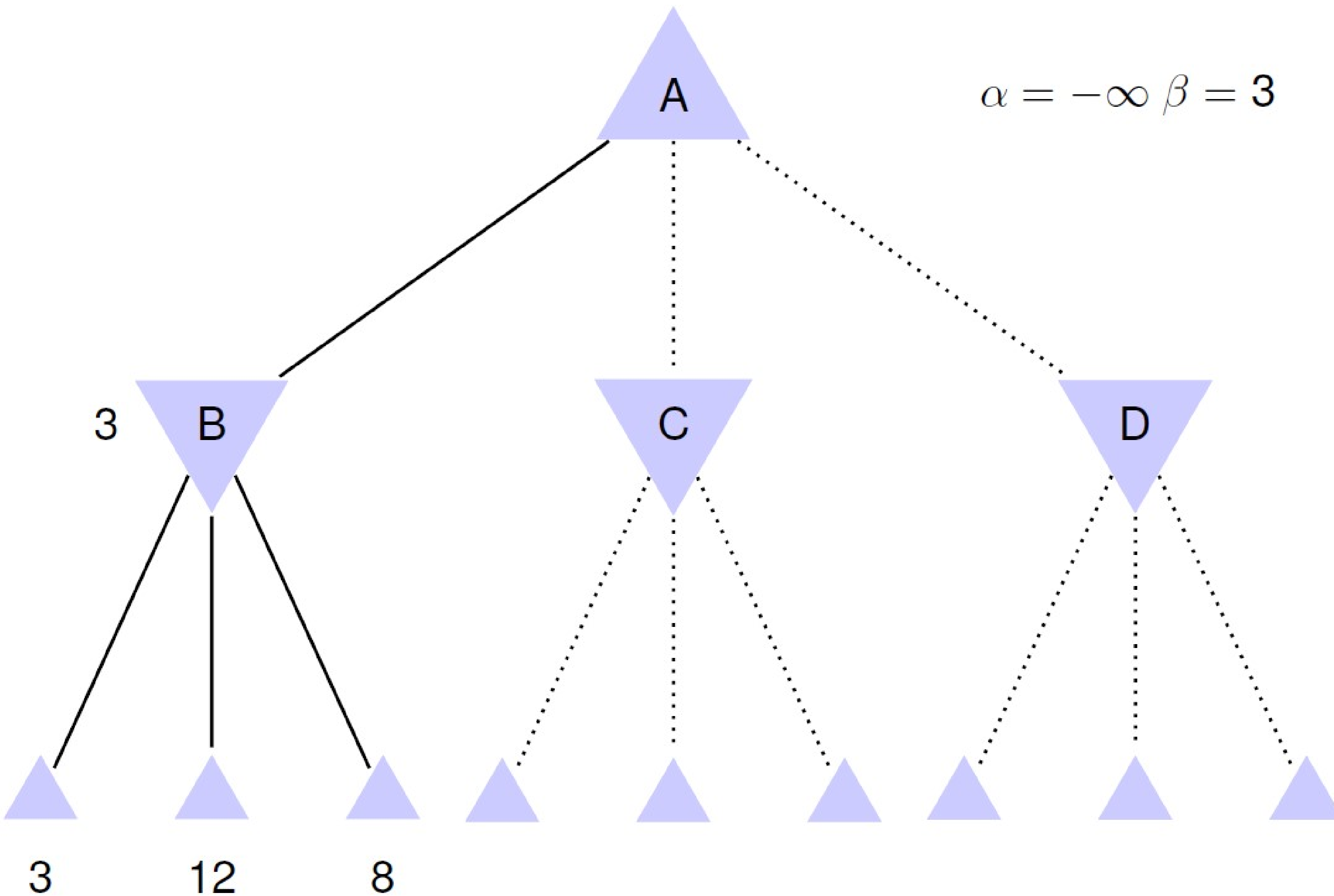
α-β pruning example



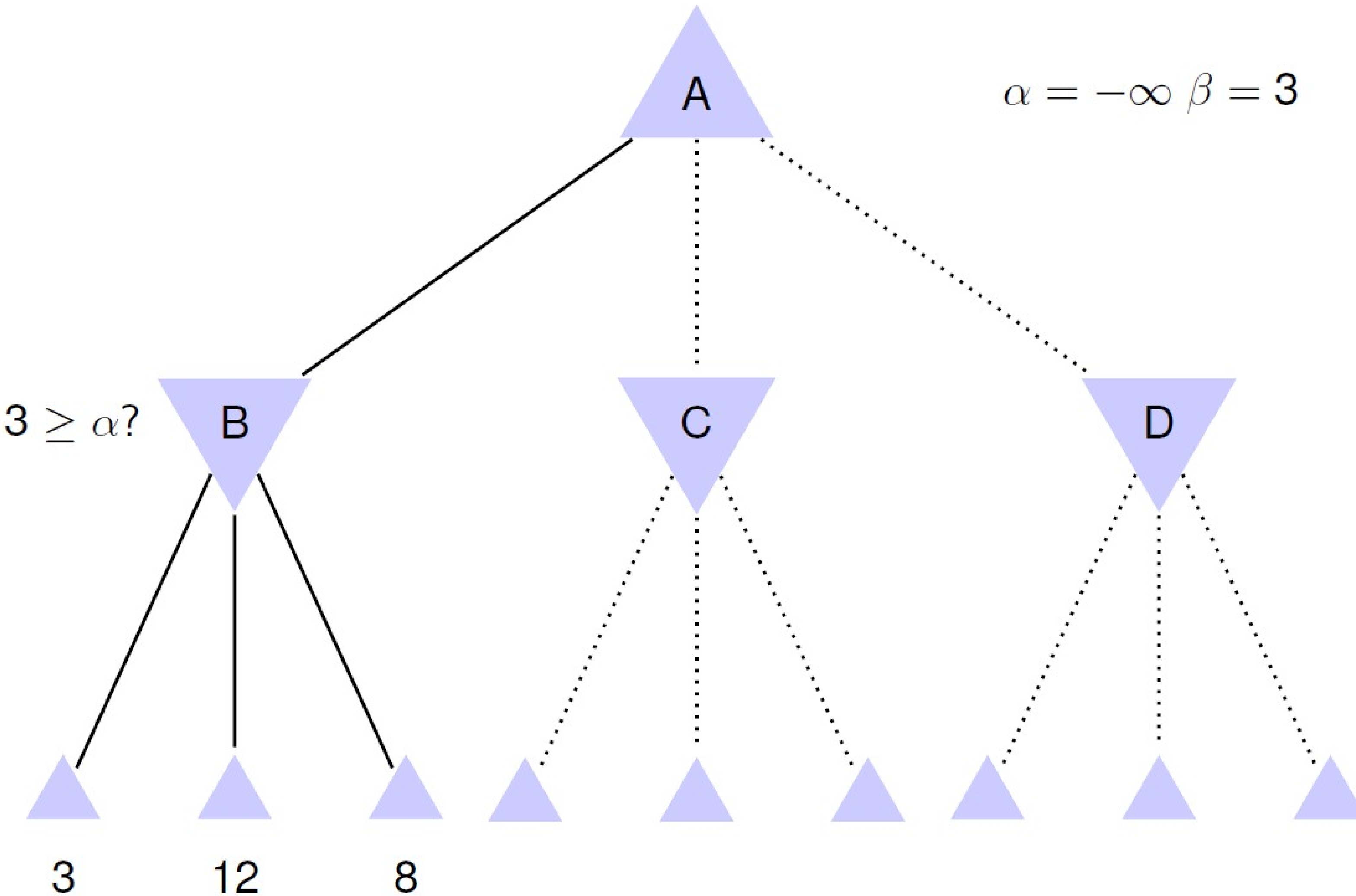
α-β pruning example



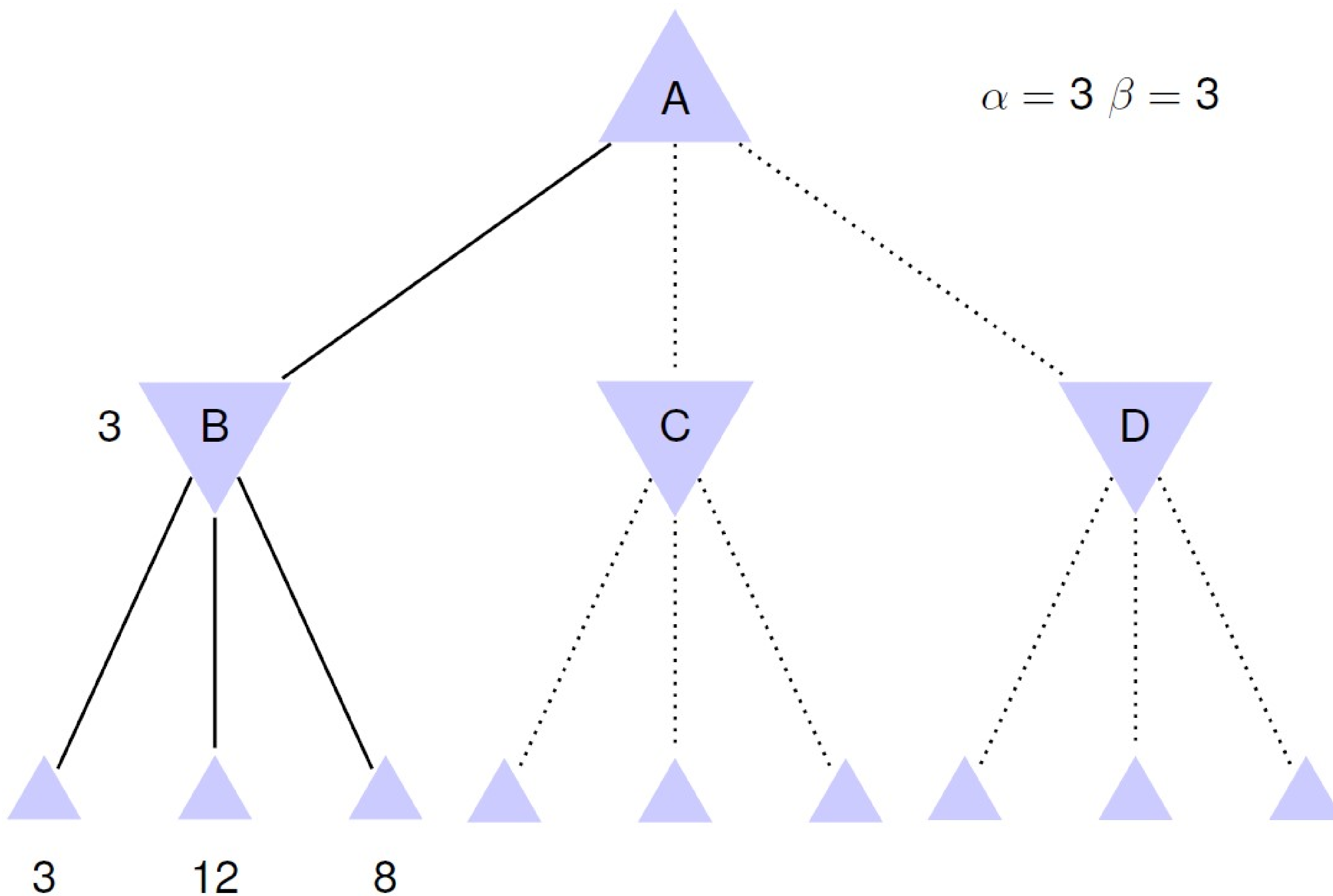
α-β pruning example



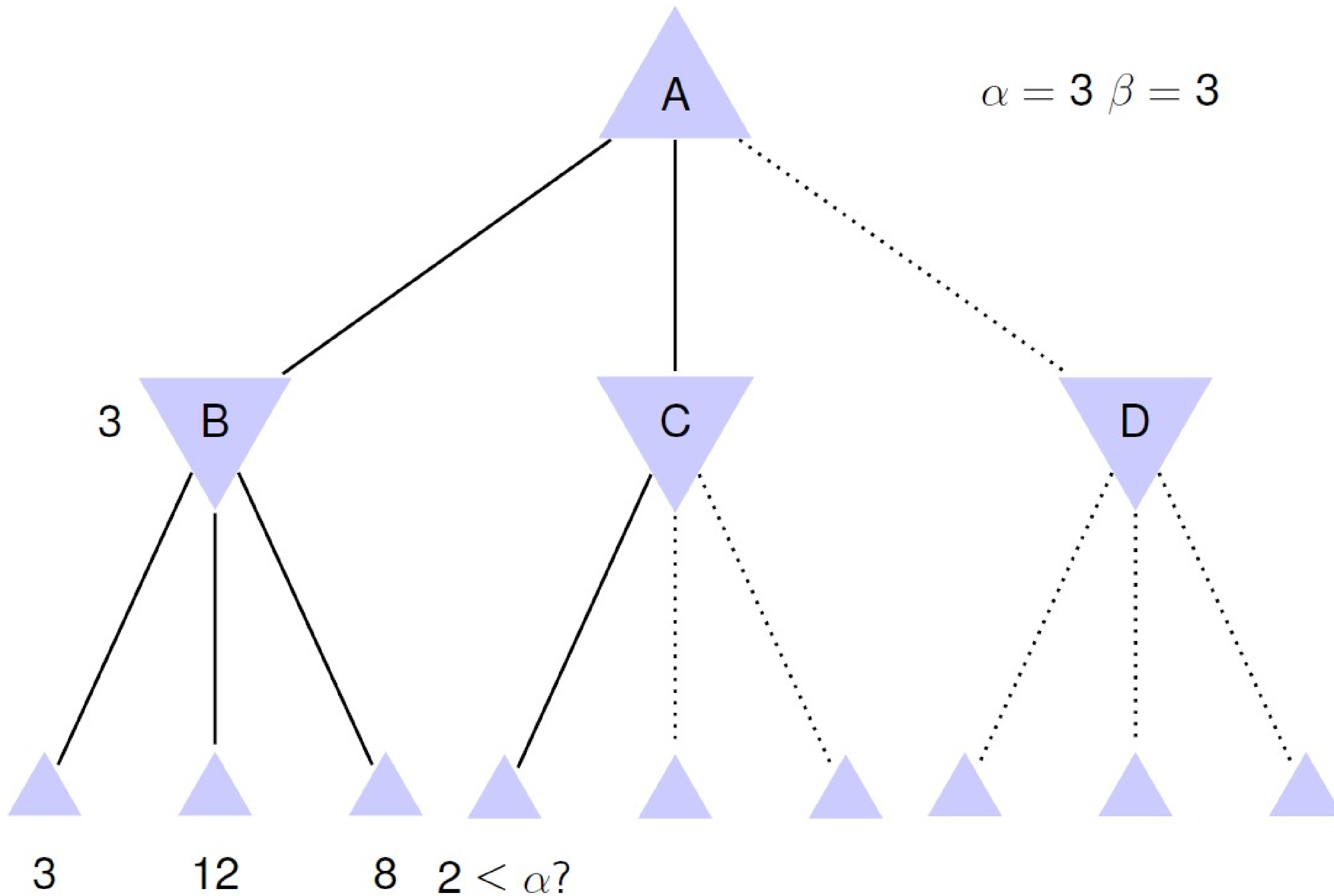
α-β pruning example



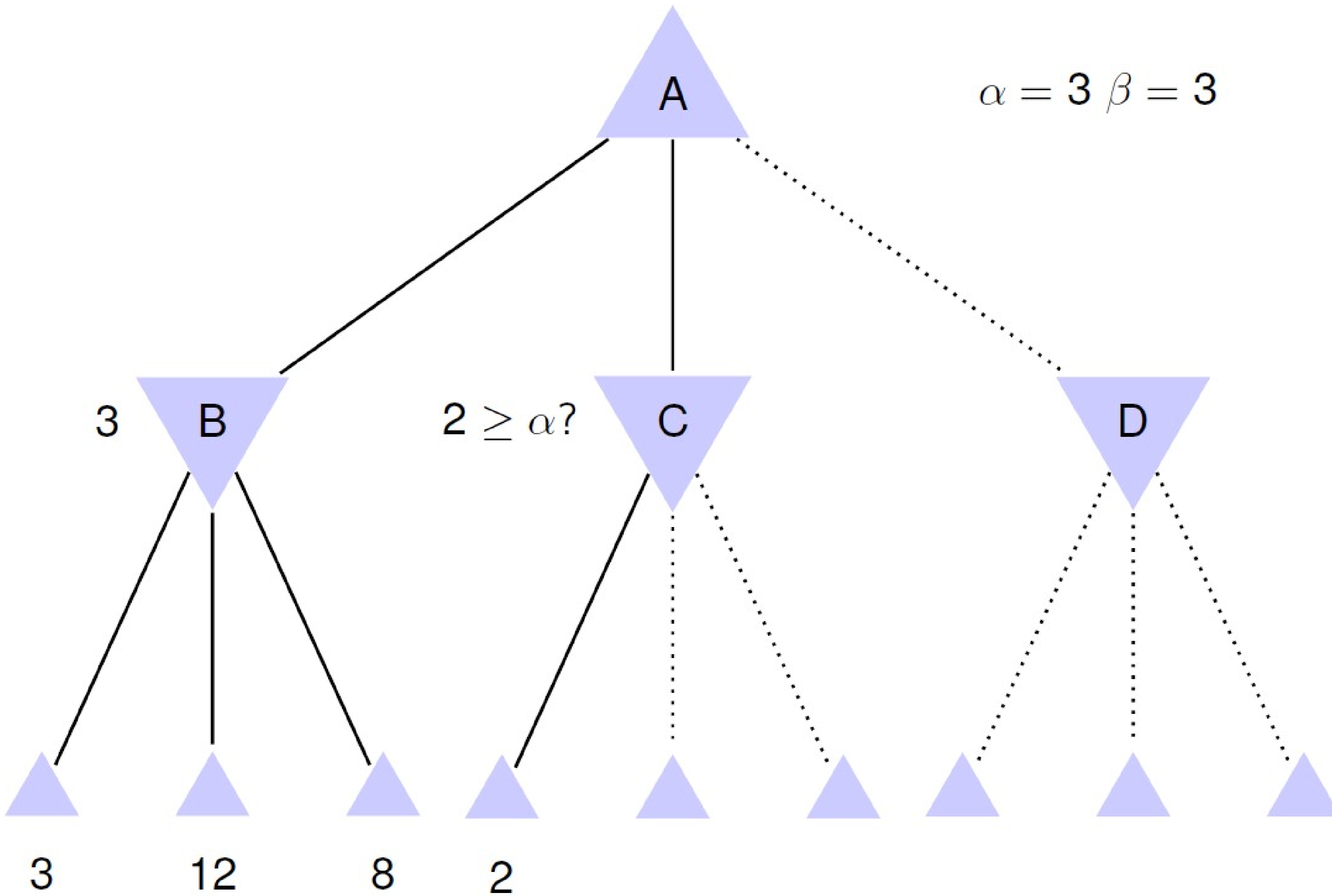
α-β pruning example



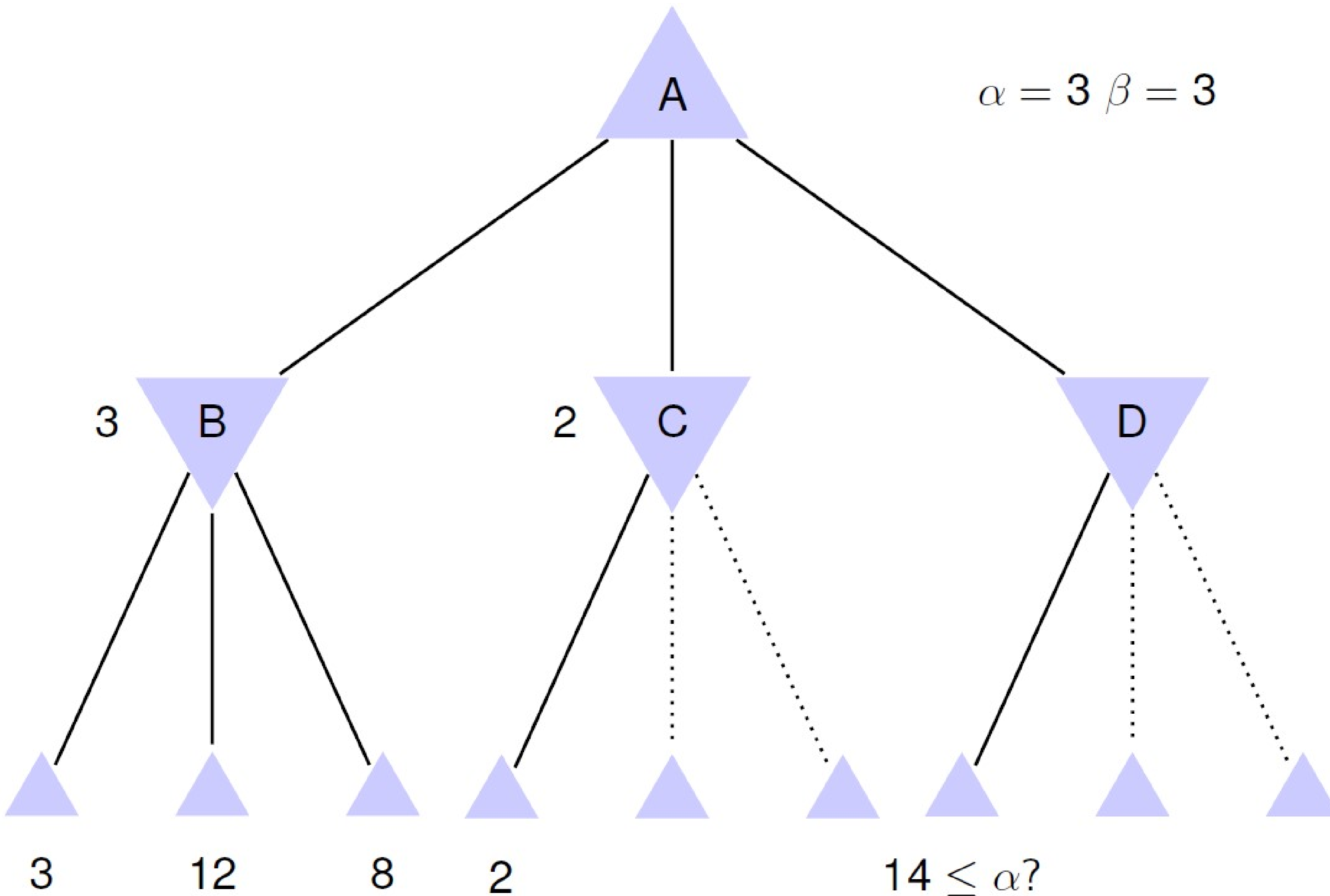
α-β **pruning example**



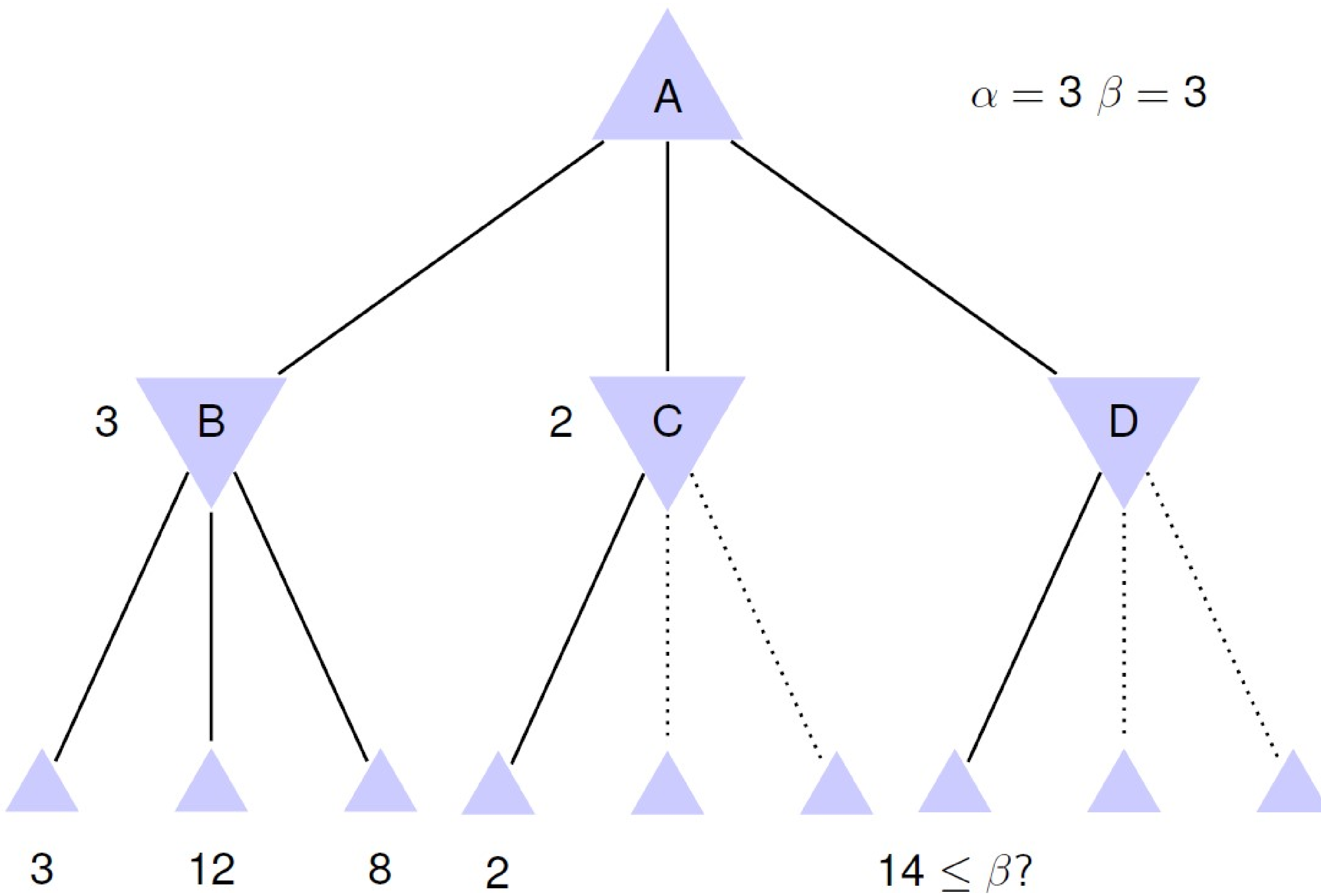
α-β pruning example



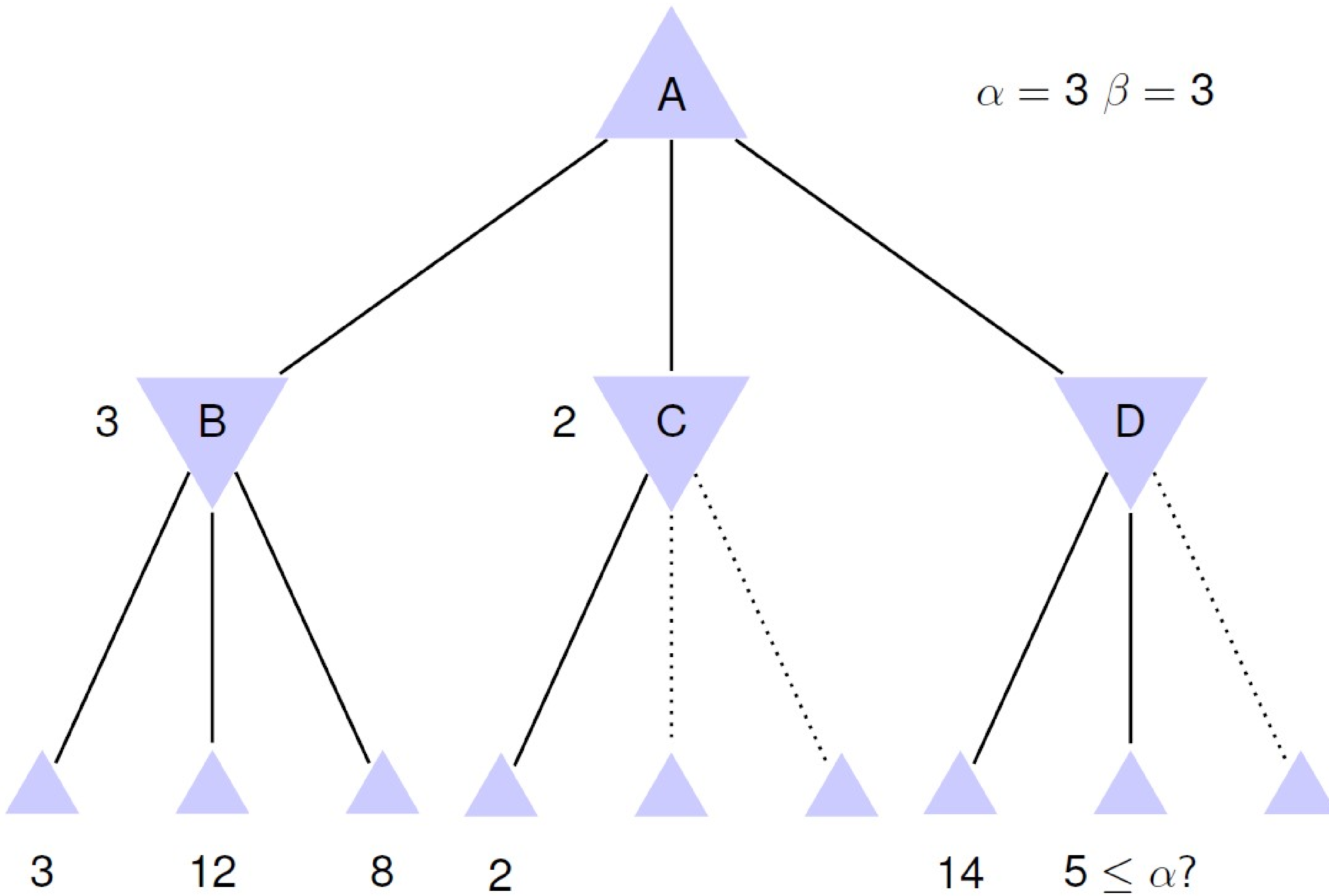
α-β pruning example



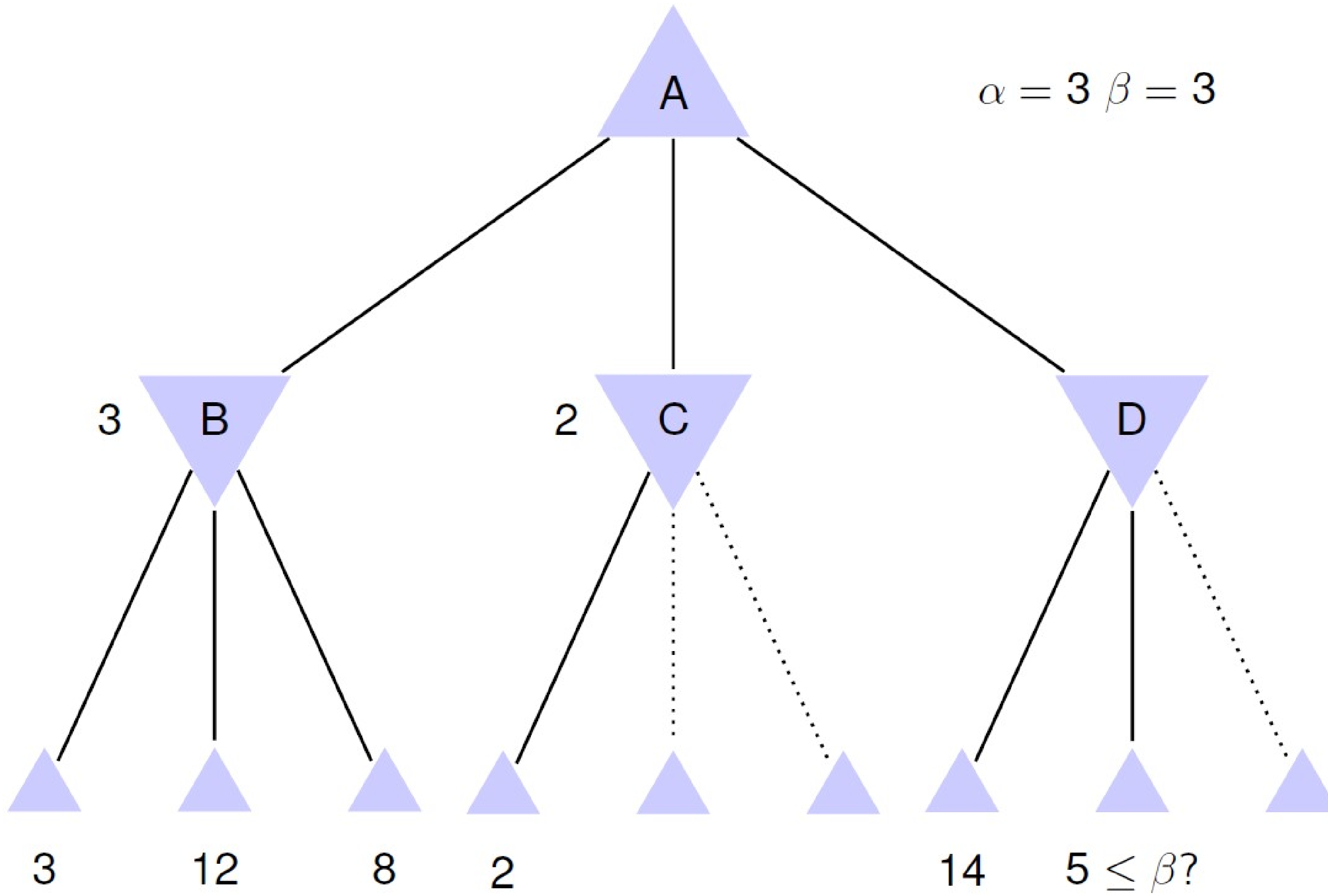
α-β pruning example



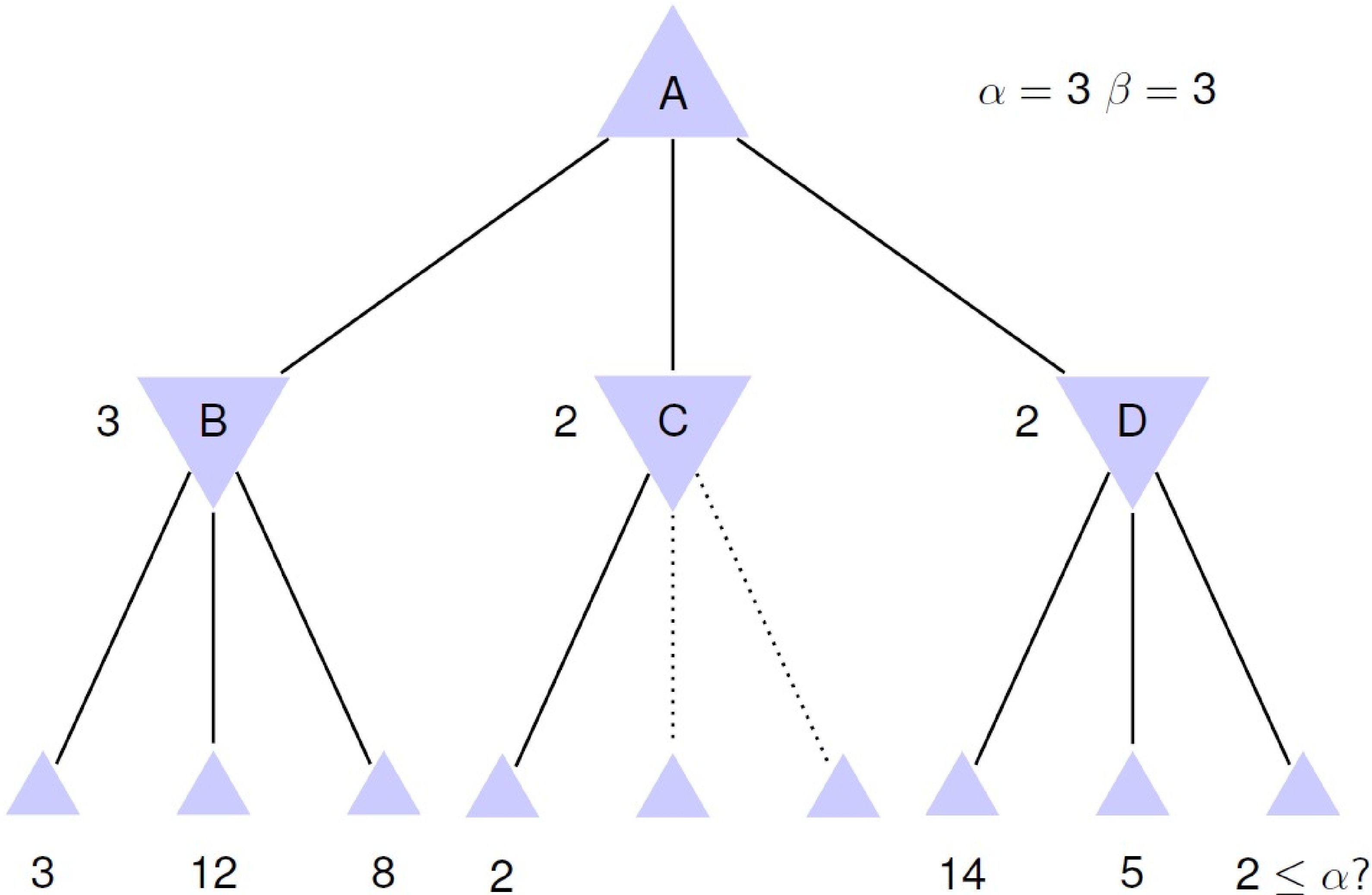
α-β **pruning example**



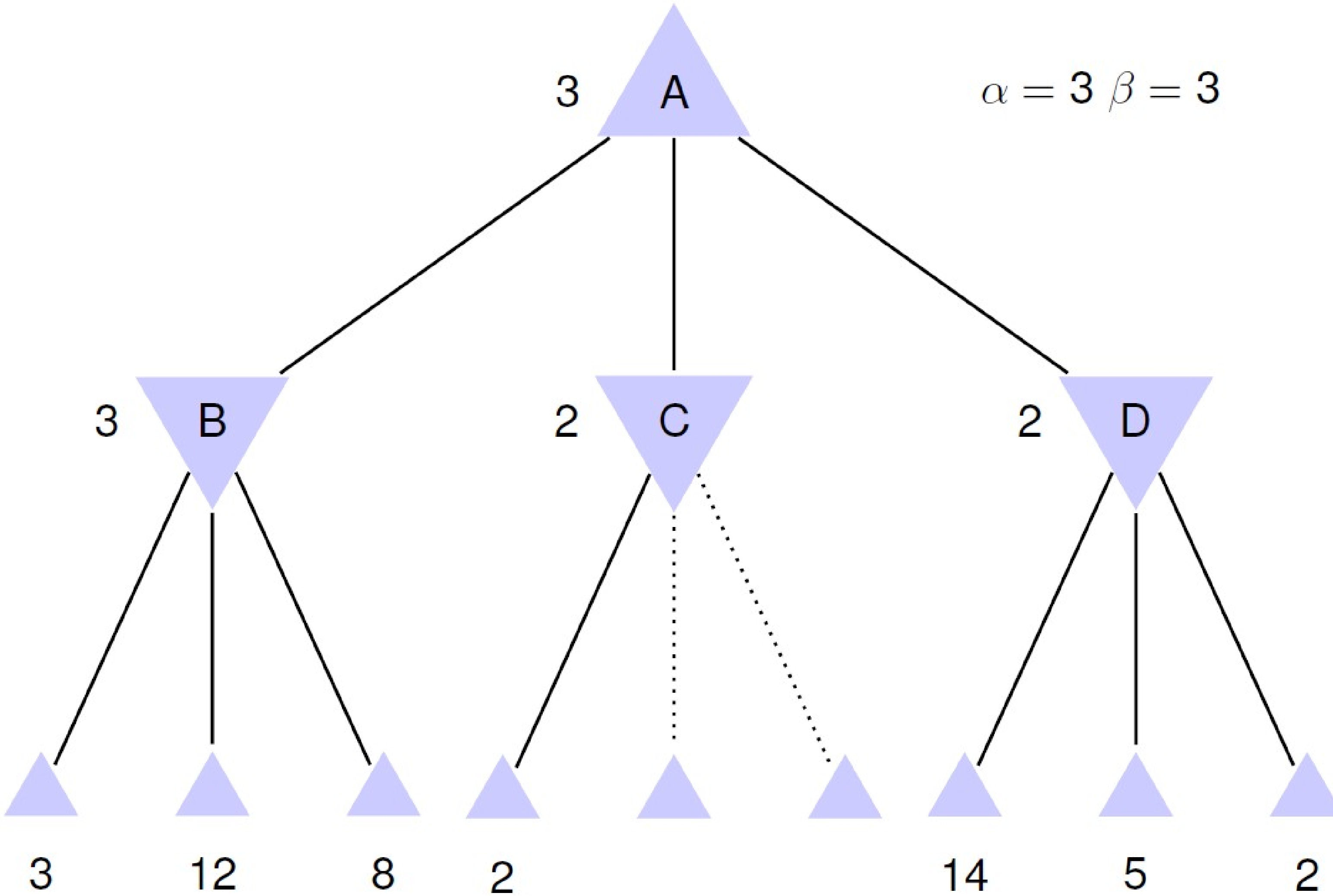
α-β pruning example



α-β pruning example



α-β pruning example



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…