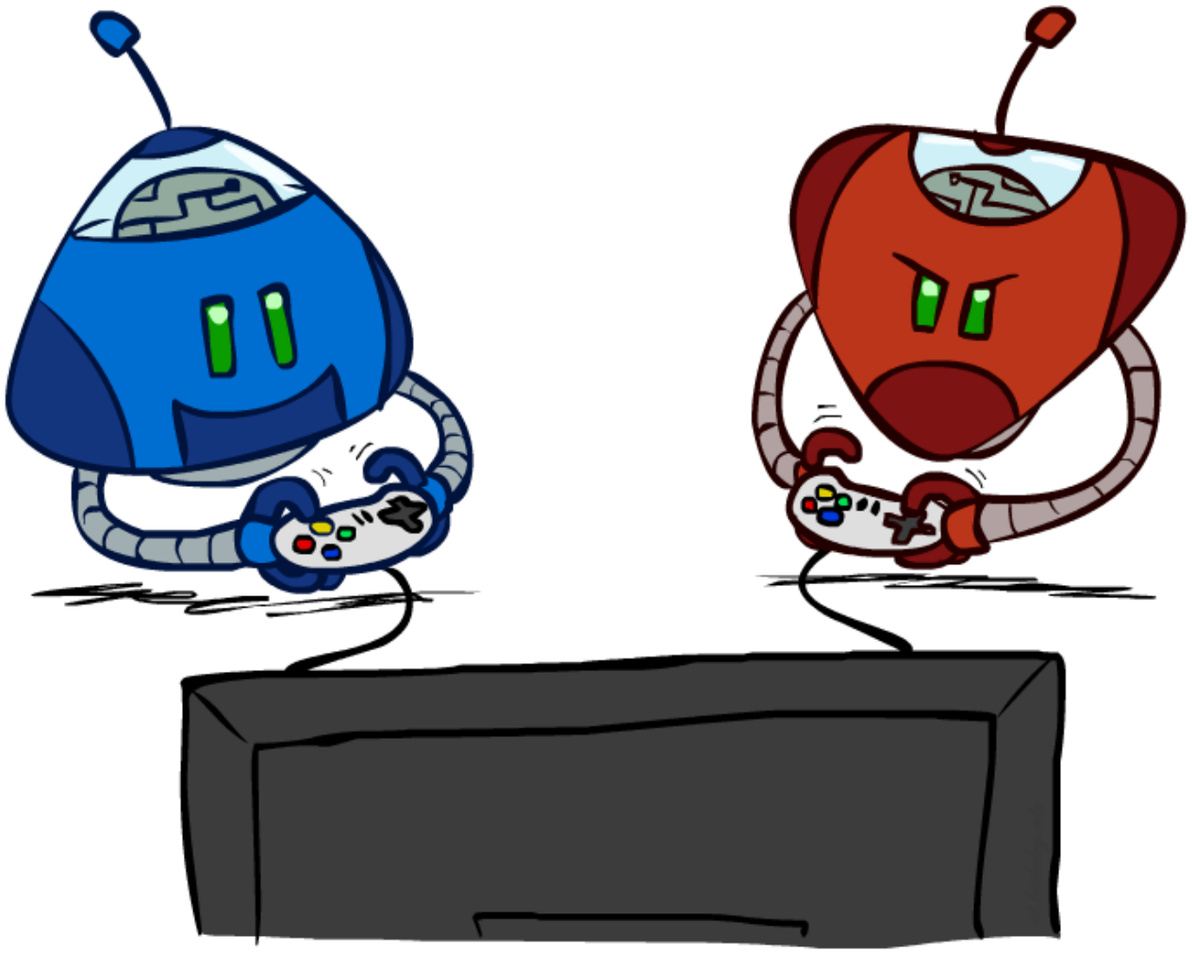
Ve492: Introduction to Artificial Intelligence

Search in Games; Adversarial Search

multi-agent



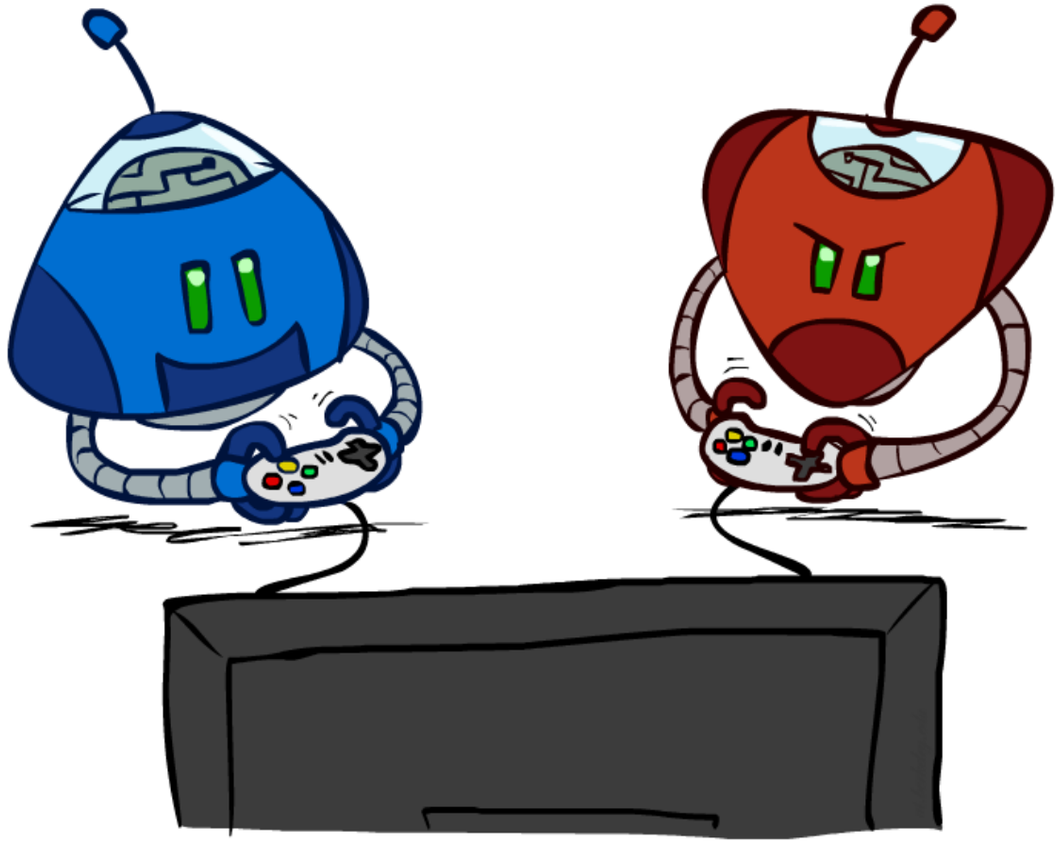
Paul Weng

UM-SJTU Joint Institute

Slides adapted from http://ai.berkeley.edu, AIMA, UM, CMU

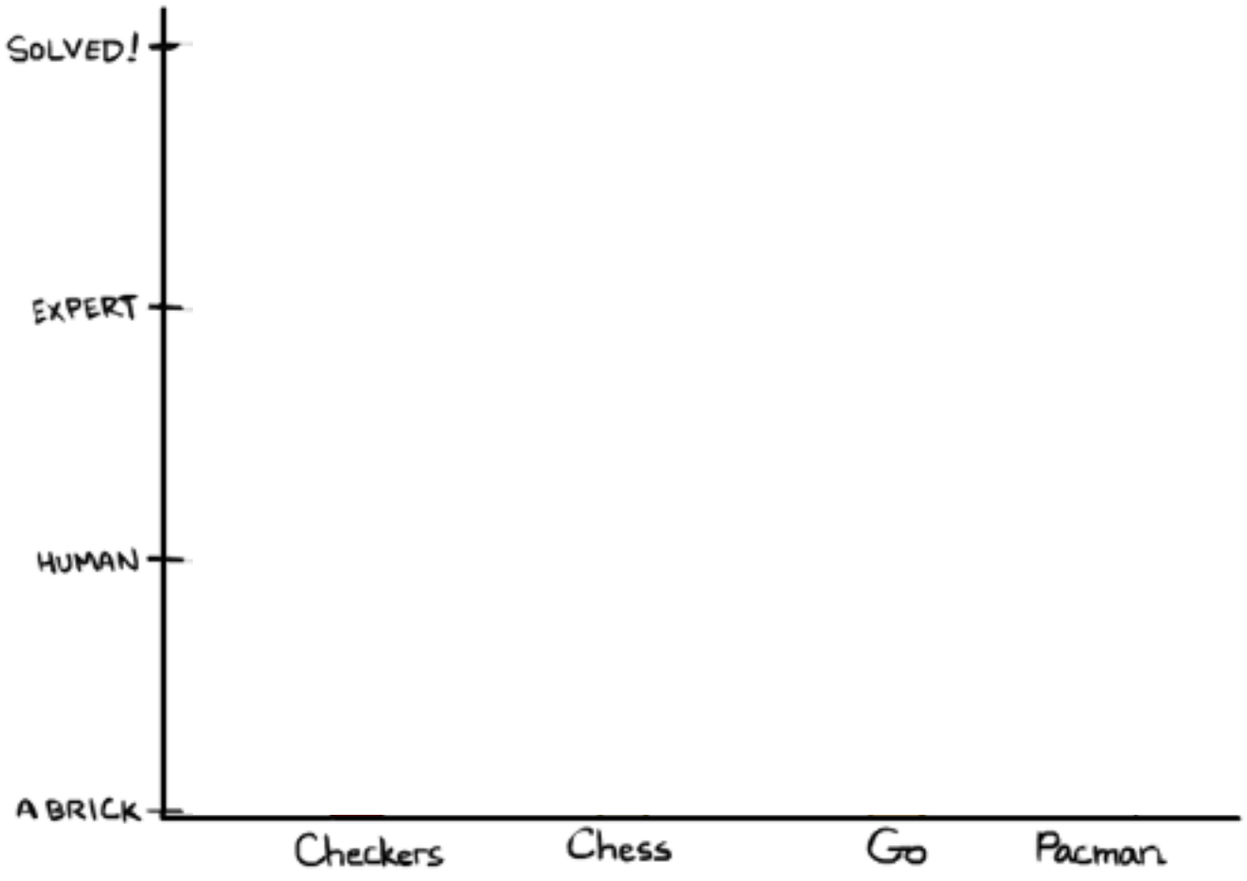
Outline

- Introduction
- Game Formalization
- Games with an adversary

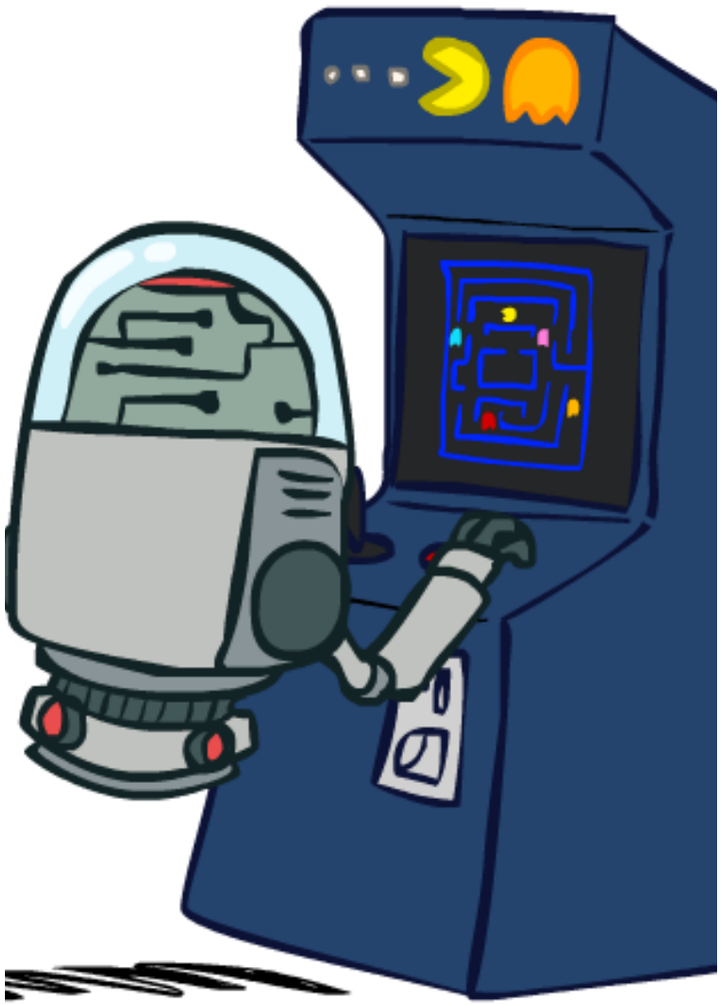


Game Playing State-of-the-Art

- 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. Current programs are even better, if less historic.
- Go: 2016: Human champions are now losing against machines. In go, b > 300! Classic programs use pattern knowledge bases, but recent advances use Monte Carlo (randomized) expansion methods and learned evaluation function.
- Pacman

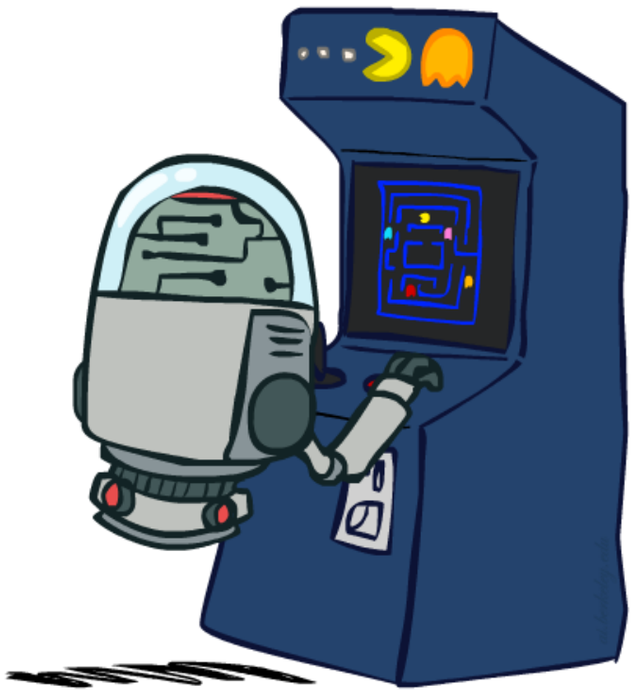


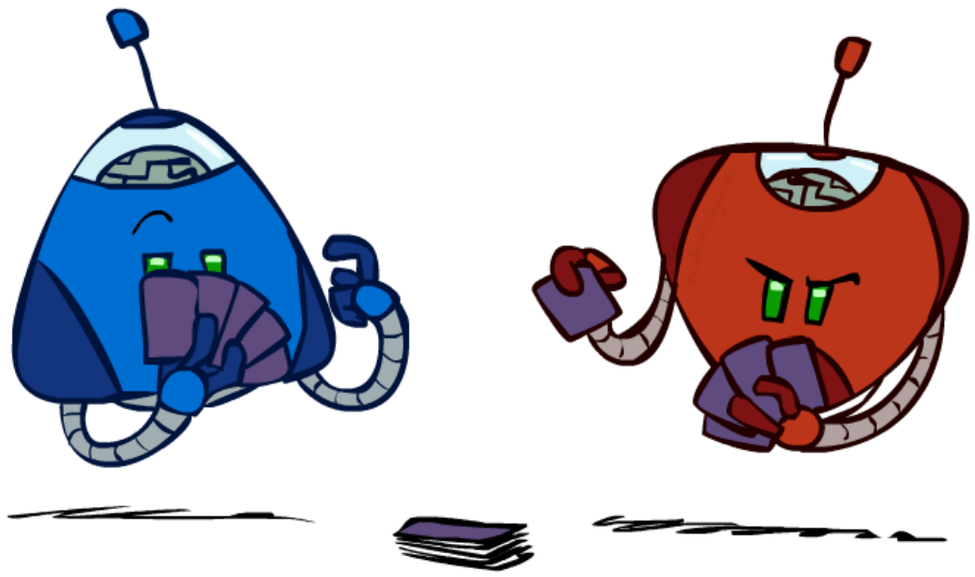
Game Formalization



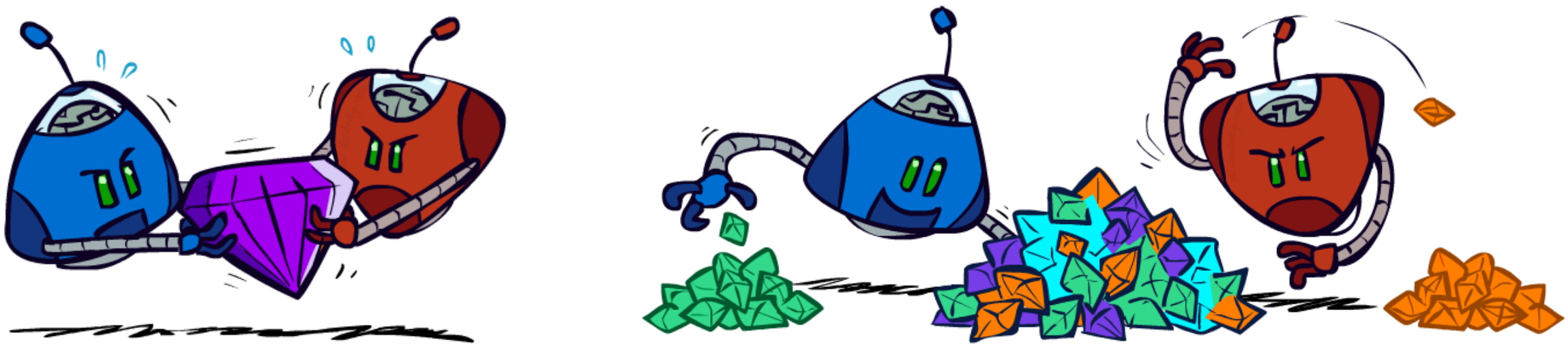
Games

- Many variety of games
- Properties:
  - Deterministic or stochastic?
  - Perfect information (fully observable)?
  - One, two, or more players?
  - Turn-taking or simultaneous?
  - Zero sum?
- Goal: which strategy to play?





Zero-Sum Games



Zero-Sum Games

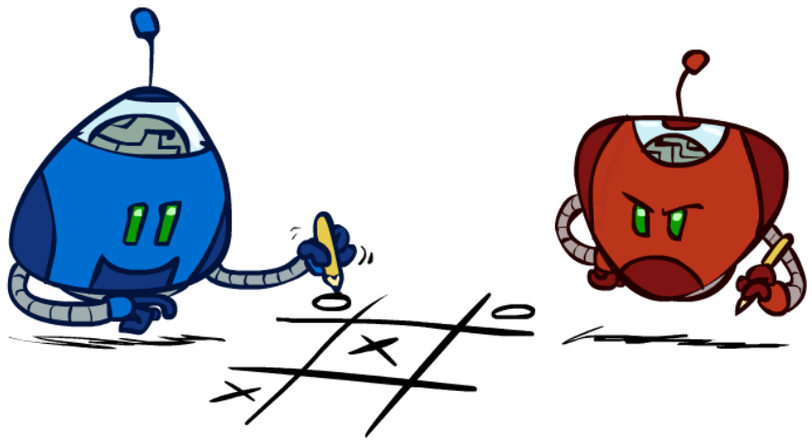
- Agents have opposite utilities
- Pure competition:
  - One maximizes
  - The other minimizes

General Games

- Agents have independent utilities
- Cooperation, indifference, competition, shifting alliances, and more are all possible

Classic Games

- Classic games are deterministic, observable, two-player, turn-taking, zero-sum
- Game formulation:
  - Initial state and States: $s_0 \in S$ 2 in the picture's case
  - Players: $Player(s) \in \{1, \ldots, N\}$ indicates whose move it is
  - Actions: $a \in Action(s)$ for player on move
  - Transition model: $Succ(s, a) \in S$
  - Terminal test: *Terminal-Test*(s) boolean
  - Terminal values: *Utility*(s, p) for player p

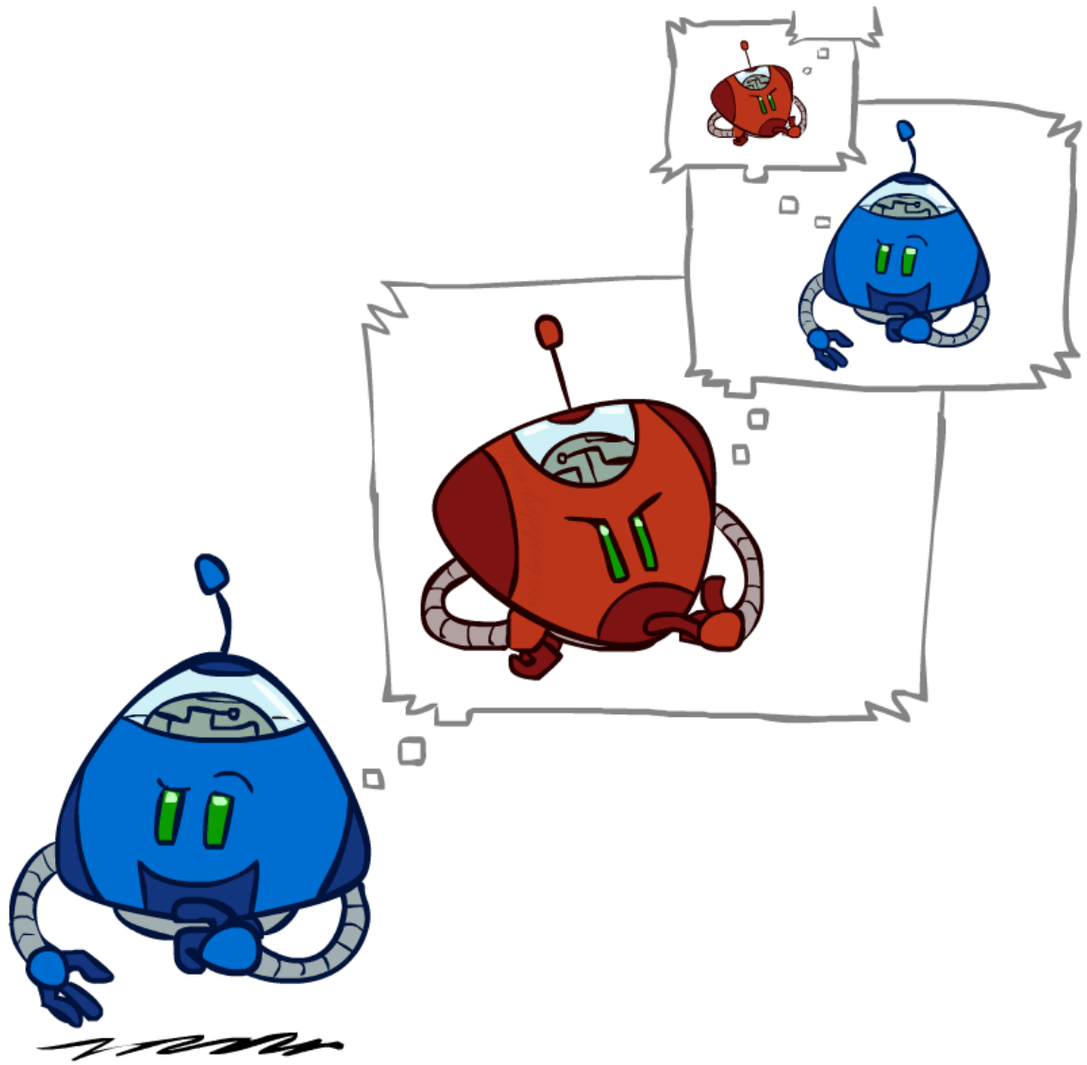


Or just *Utility*(s) for player making the decision at root

- Solution for a player: strategy or policy defines how to choose an action in a state.

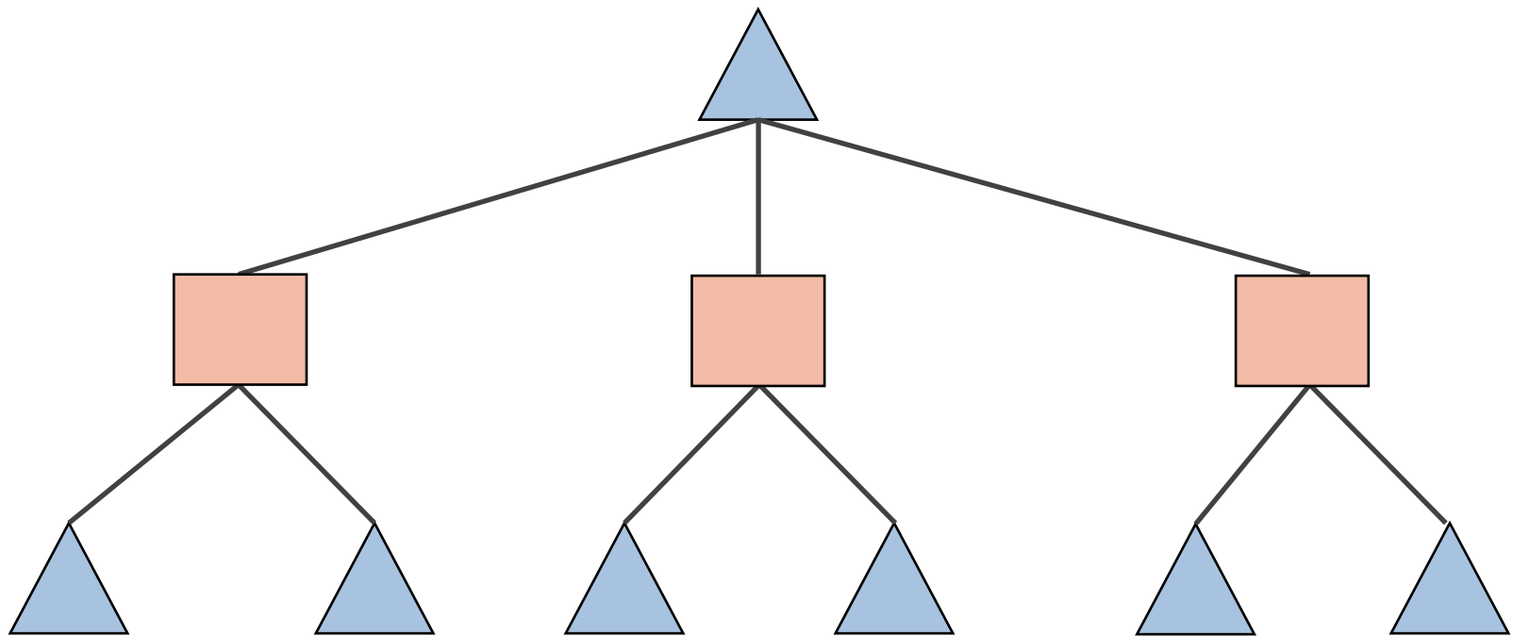
Games with an Adversary

Adversarial Search



Game Trees

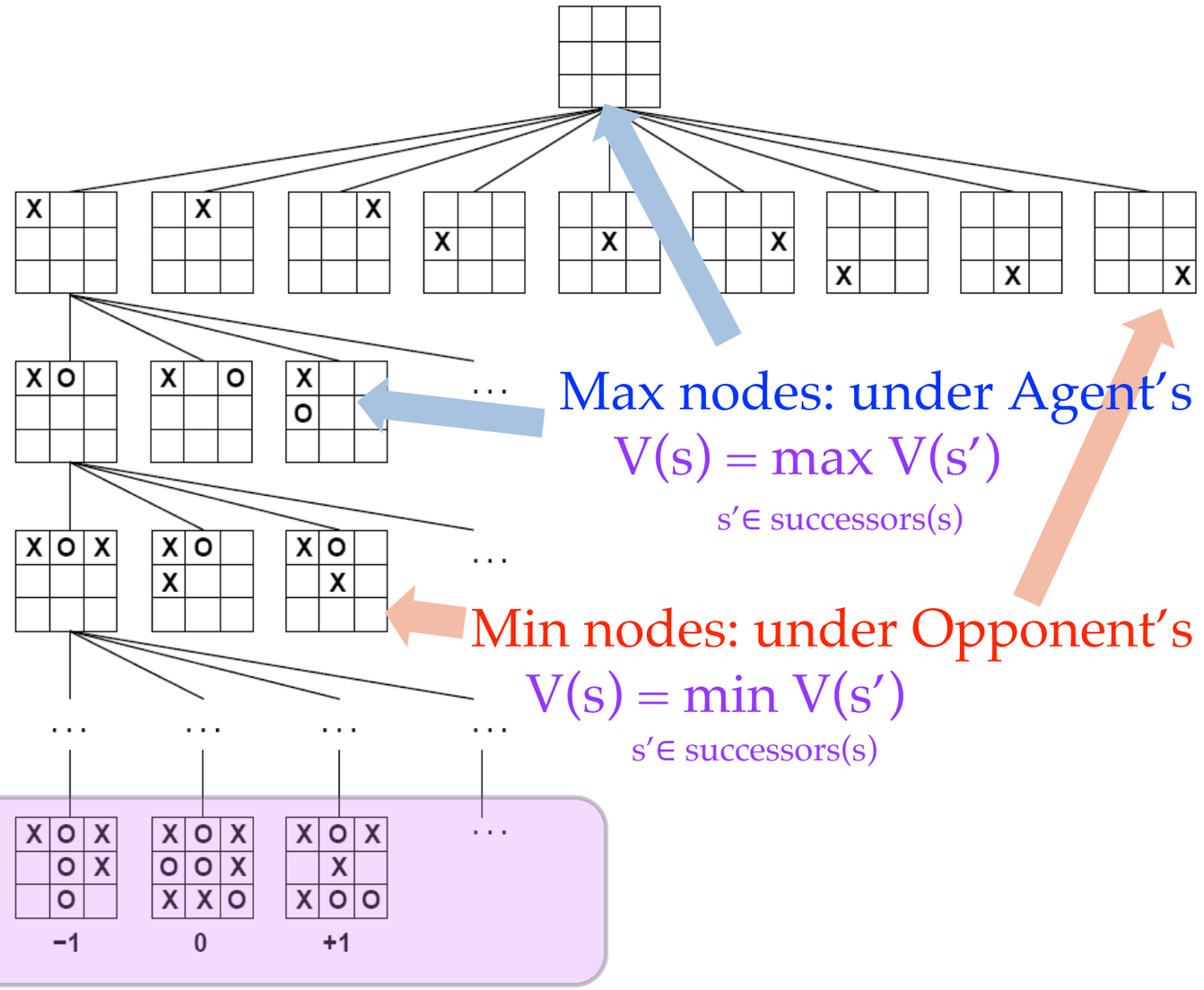
- Search one agent's actions from a state, search the competitor's actions from those resulting states , etc…



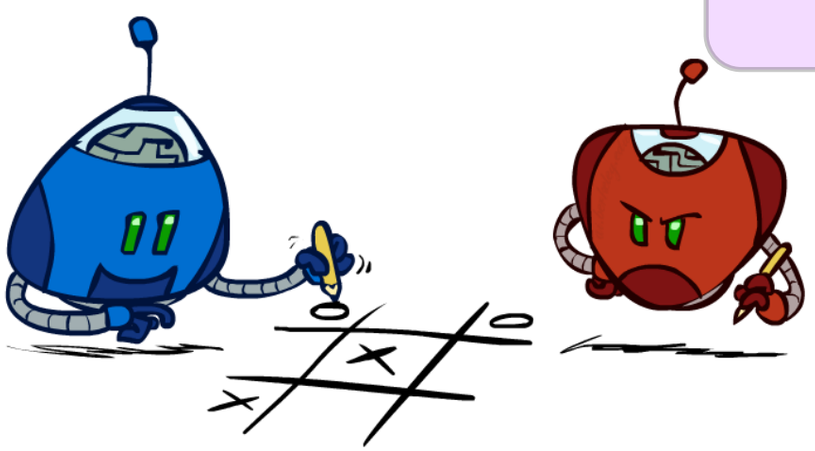
Game Tree of Tic-Tac-Toe

This is a zero-sum game, the best action for X is the worst action for O and vice versa

How do we define best and worst?



Instead of taking the max utility at every level, alternate max and min



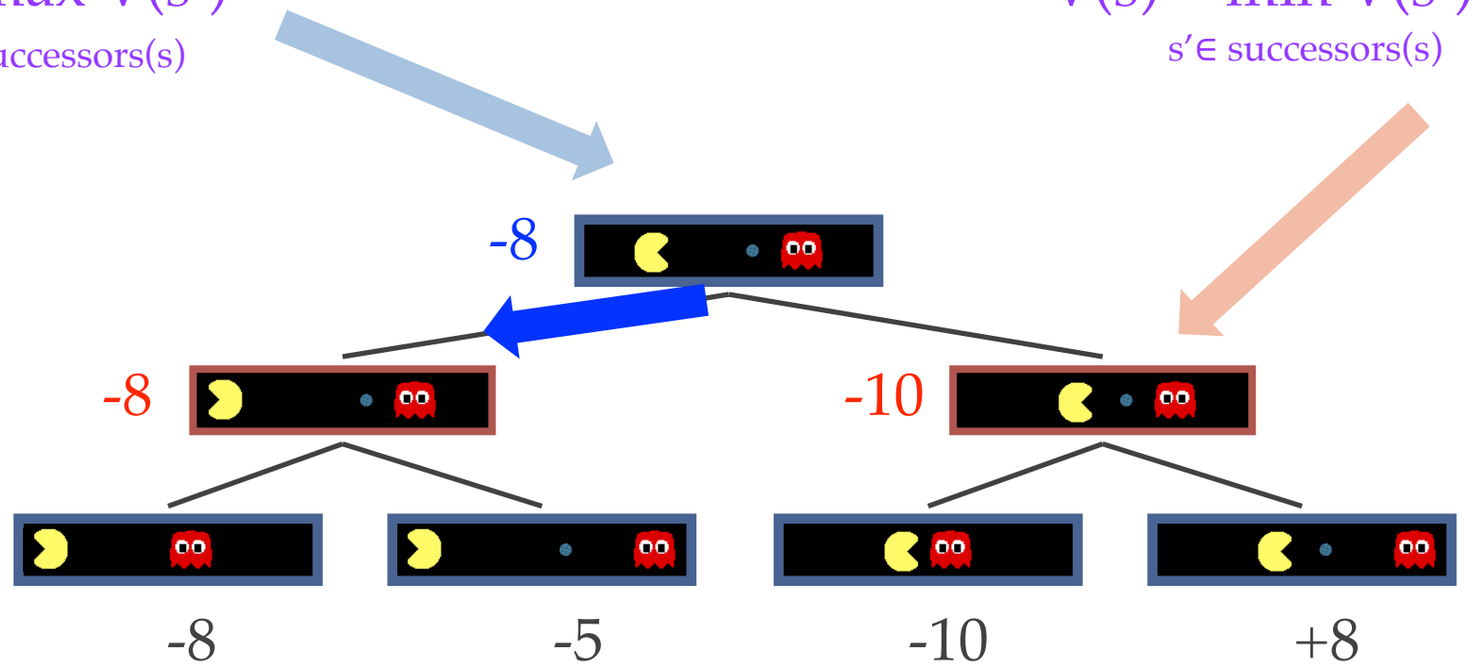
Example: Small Pacman

Max nodes: under Agent's control

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

Min nodes: under Opponent's control

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



Terminal States:

$V(s)$ = known

Adversarial Search with Minimax

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

Minimax Implementation

```
function minimax-decision(s) returns action
    return an action a in Actions(s) with the highest
                min-value(Succ(s,a))
```

Succ(s,a)→s′

```
function max-value(s) returns value
if Terminal-Test(s) then return Utility(s)
initialize v = -∞
for each a in Actions(s):
      v = max(v, min-value(Succ(s,a)))
return v
```

```
function min-value(s) returns value
if Terminal-Test(s) then return Utility(s)
initialize v = +∞
for each a in Actions(s):
      v = min(v, max-value(Succ(s,a)))
return v
```

Max nodes: under Agent's control

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

Min nodes: under Opponent's control

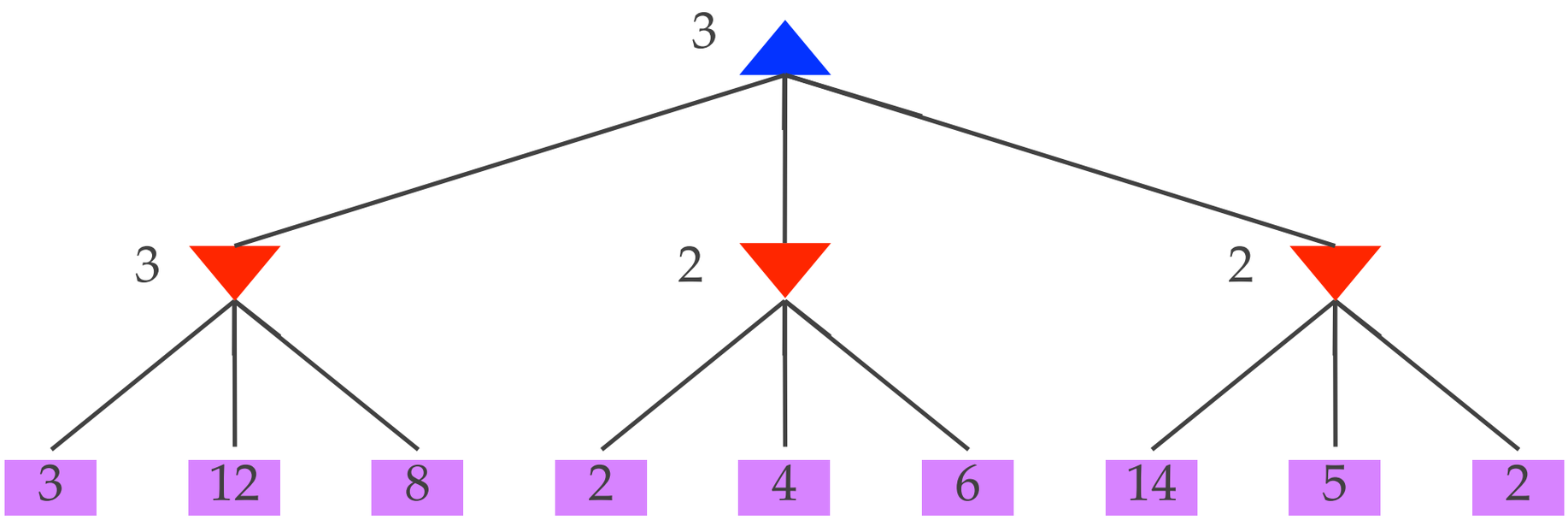
$V(s) = \min_{s' \in \text{successors}(s)} V(s')$

Alternative Implementation

function minimax-decision(s) returns an action
    return an action a in Actions(s) with the highest
            value(Succ(s,a))

function value(s) returns a value
if Terminal-Test(s) then return Utility(s)
if Player(s) = MAX then return $max_{a \in Action(s)}$ value(Succ(s,a))
if Player(s) = MIN then return $min_{a \in Action(s)}$ value(Succ(s,a))

Minimax Example

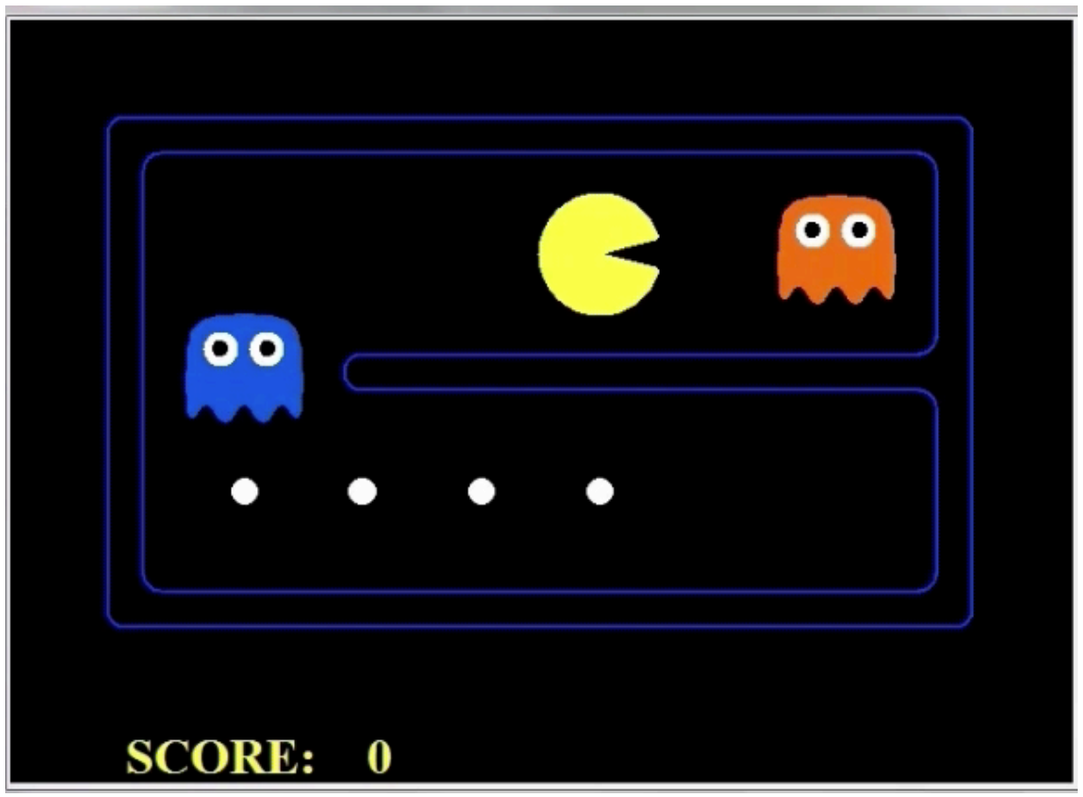


- What kind of search is Minimax Search?

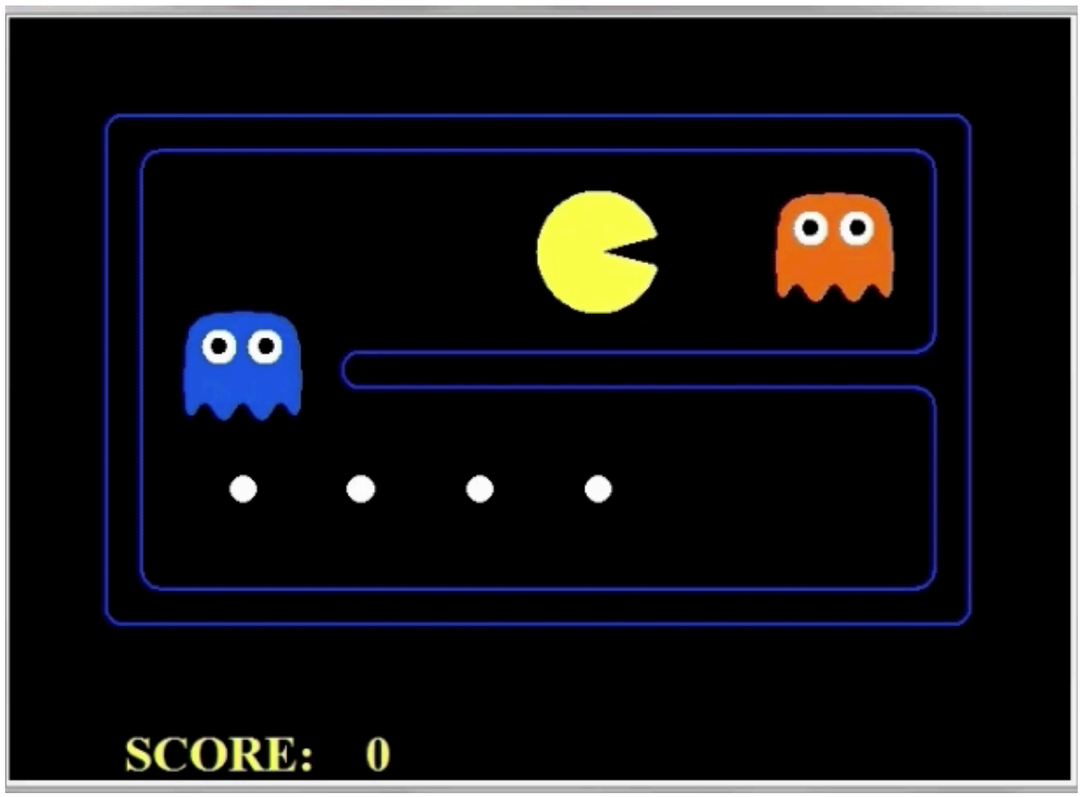
Minimax Properties

- Multi-agent decision-making requires a model of other agents
- Here, adversary assumed to be rational
- Minimax is optimal against perfect opponent
- Is Minimax still optimal otherwise?

Example: Adversarial Opponents

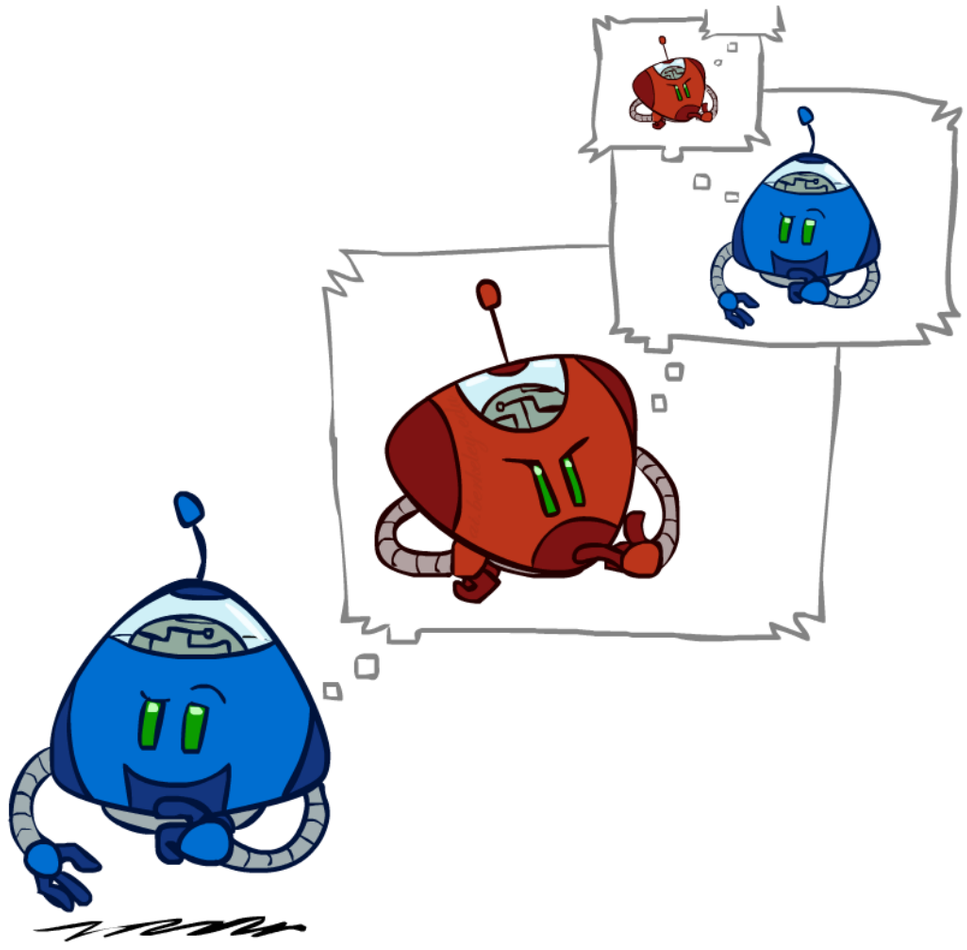


Example: Random Opponents

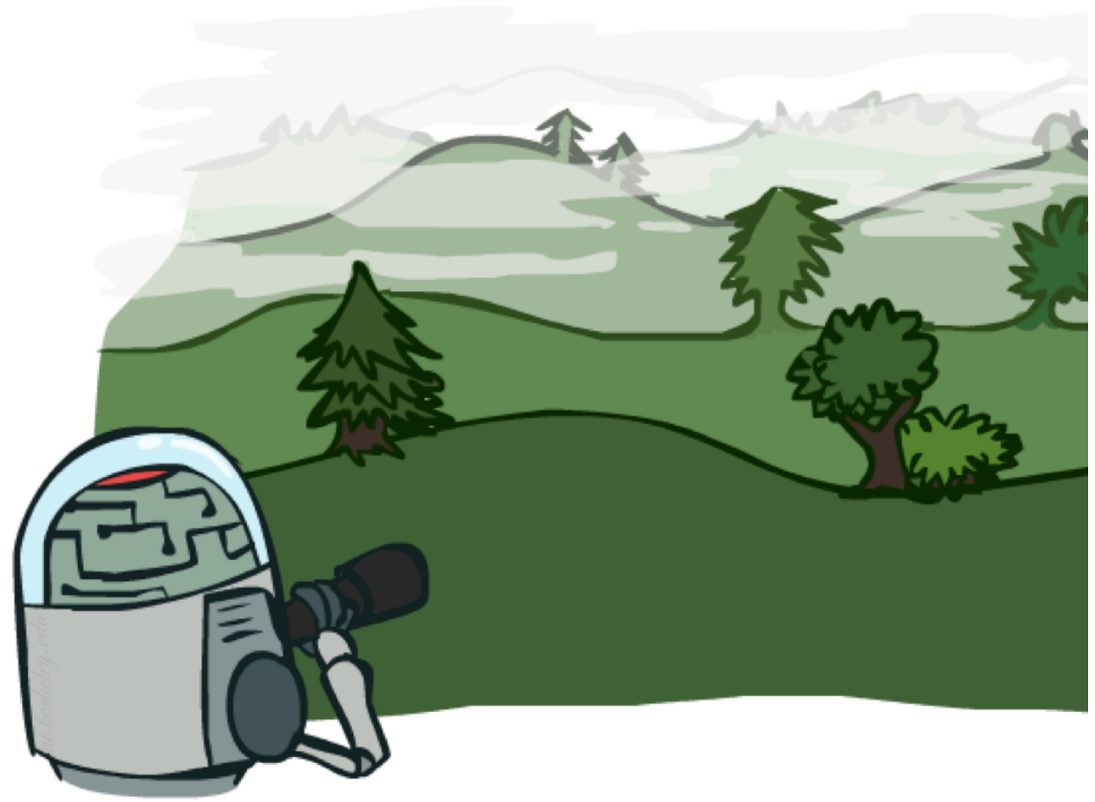


Minimax Efficiency

- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$
- E.g., for chess, b ≈ 35, m ≈ 100
  - Exact solution is completely infeasible
  - Humans can't do this either, so how do we play chess?



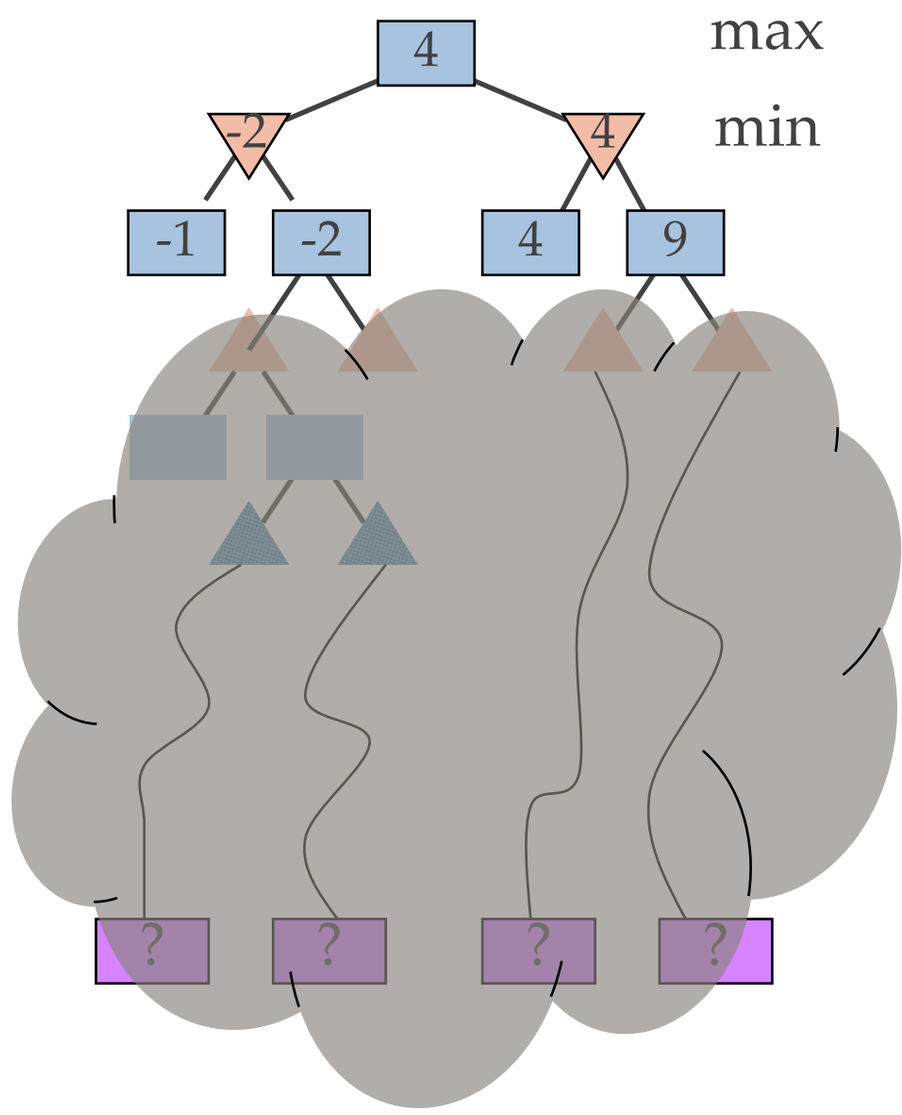
Resource Limits



Problem: In realistic games, we cannot search to leaves!

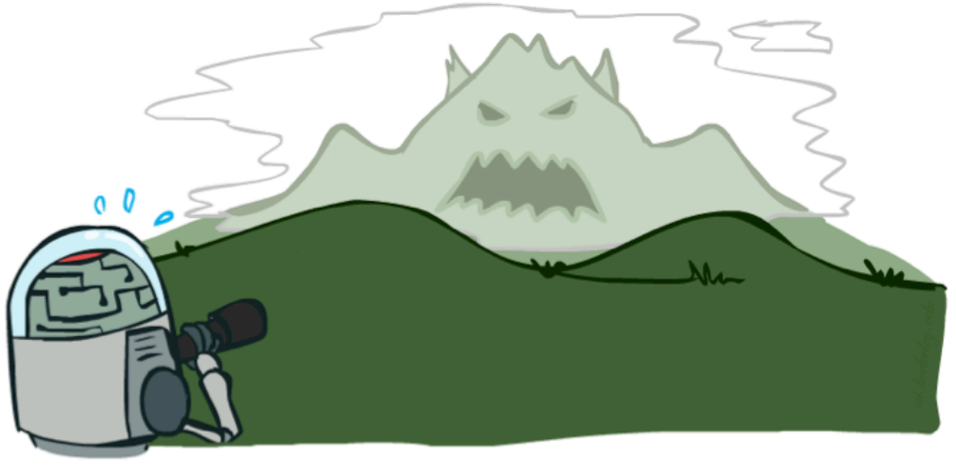
Solution 1: Bounded lookahead

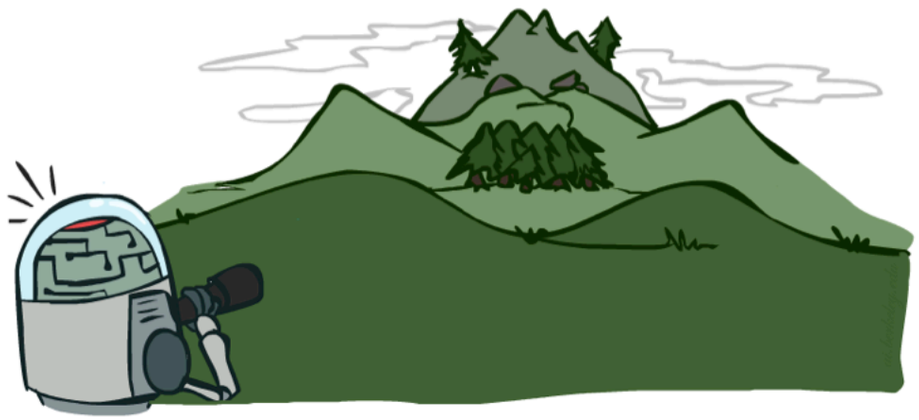
- Idea:
  - Search only to a preset depth limit or horizon
  - Use an evaluation function for non-terminal positions
- Guarantee of optimal play is gone
- More plies make a BIG difference
- Use iterative deepening for an anytime algorithm
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - For chess, b≈35 so reaches about depth 4 – not so good



Depth Matters

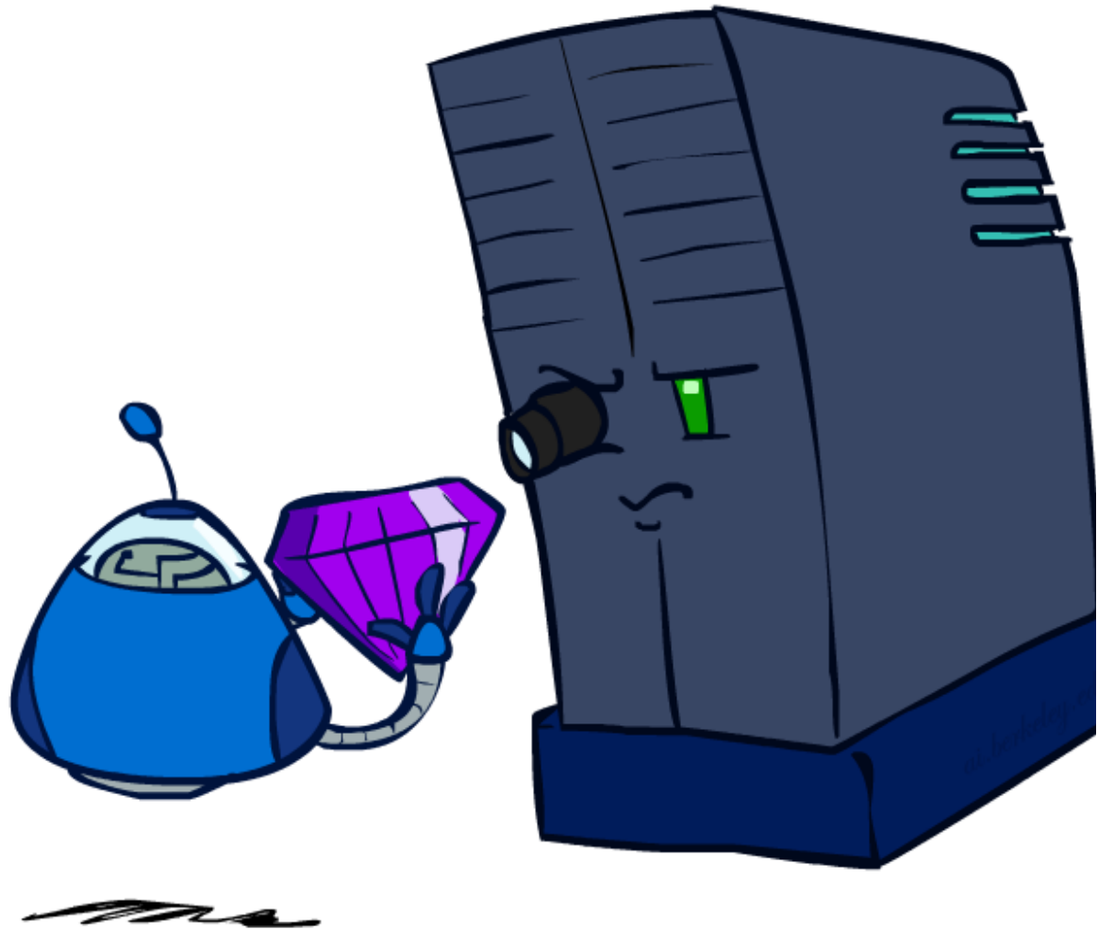
- Evaluation functions are always imperfect
- Deeper search => better play (usually)
- Deeper search gives same quality of play with a less accurate evaluation function
- An important example of the tradeoff between complexity of features and complexity of computation





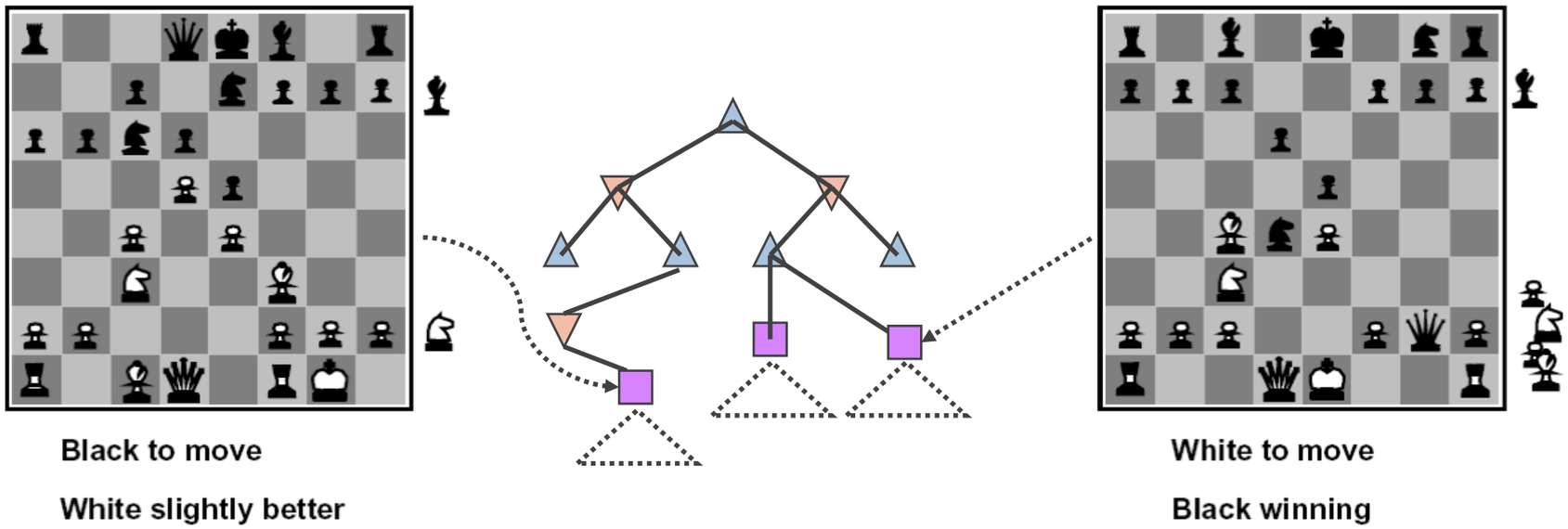
Adversarial Search

Evaluation Functions



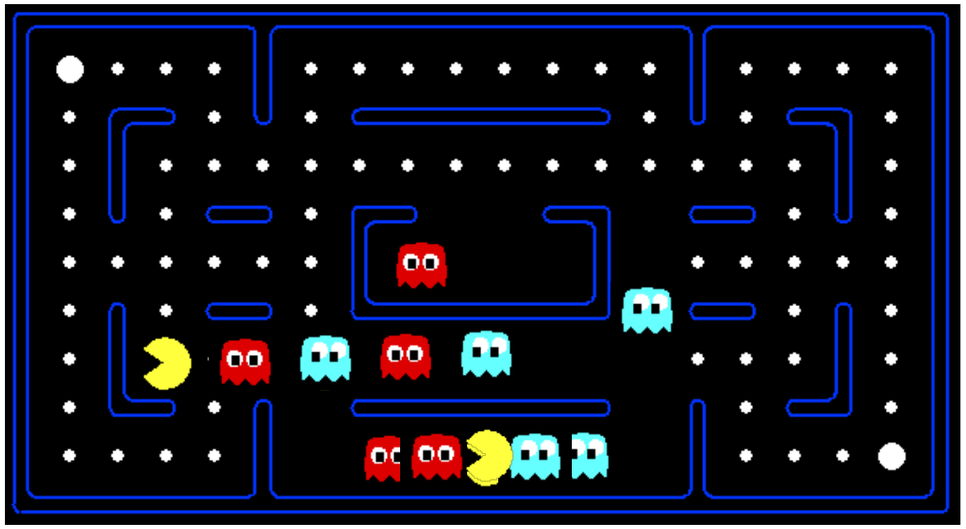
Evaluation Functions

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

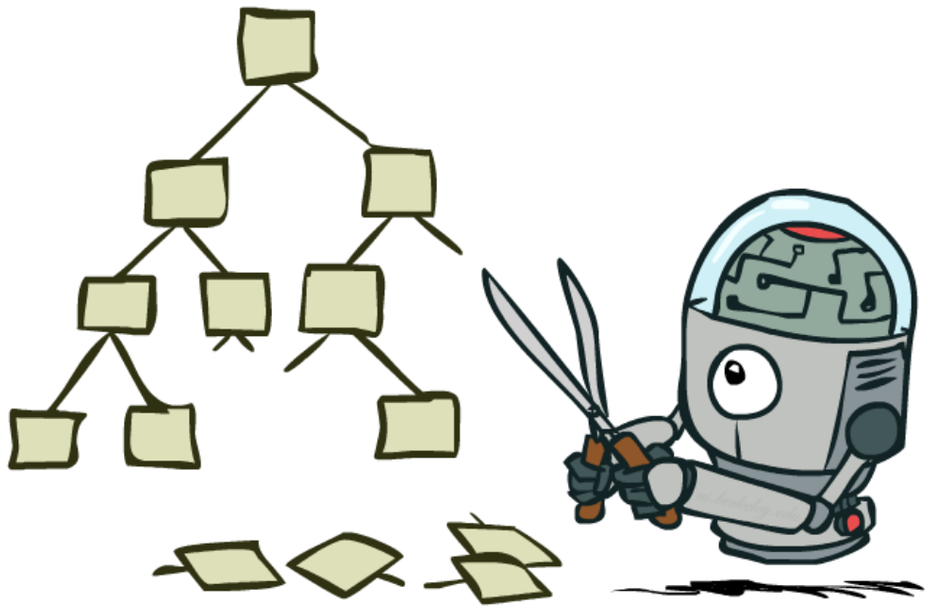


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:
  - EVAL(s) = $w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$
  - E.g., $w_1 = 9$, $f_1(s)$ = (num white queens – num black queens), etc.
- Terminate search only in quiescent positions, i.e., no major changes expected in feature values

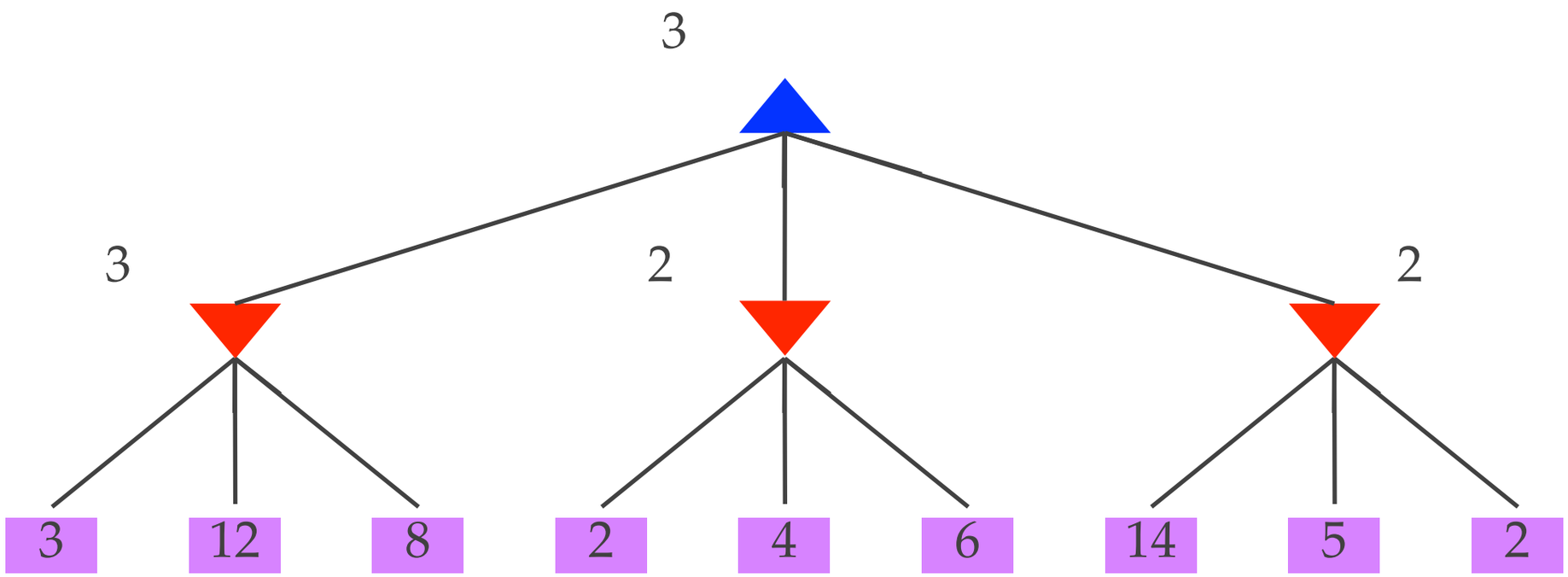
Evaluation for Pacman



Solution 2: Game Tree Pruning

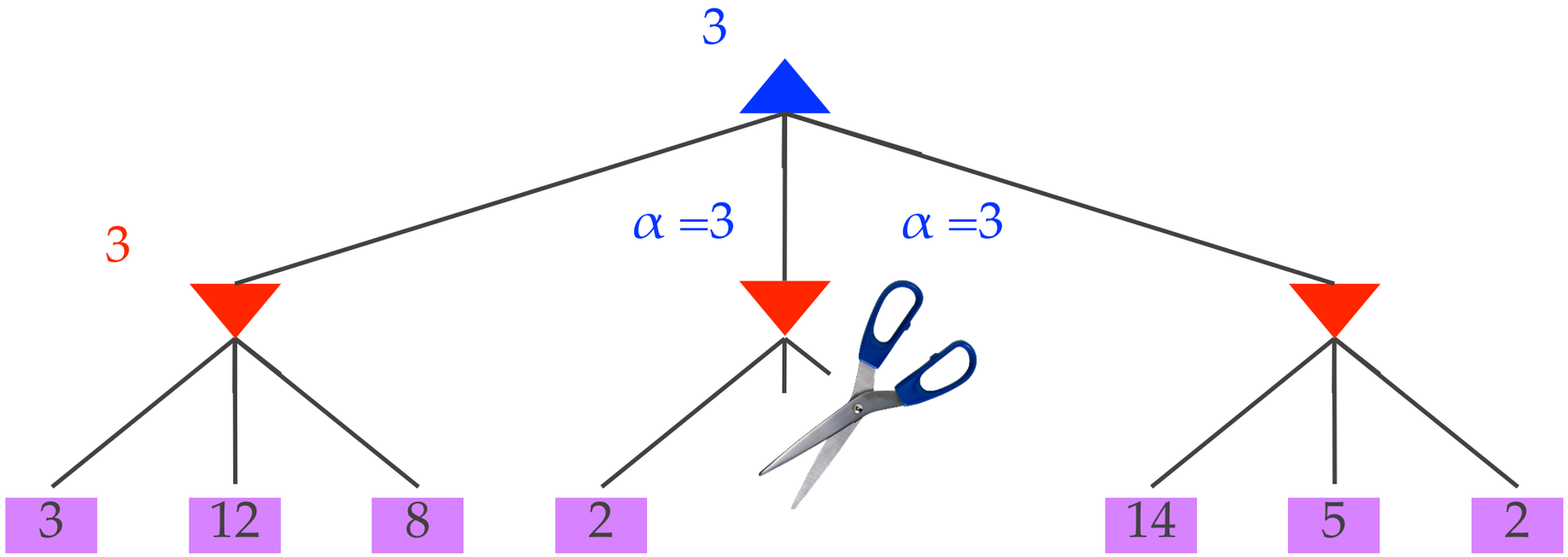


Intuition: prune the branches that can't be chosen



Alpha-Beta Pruning Example

α = best option so far from any MAX node on this path

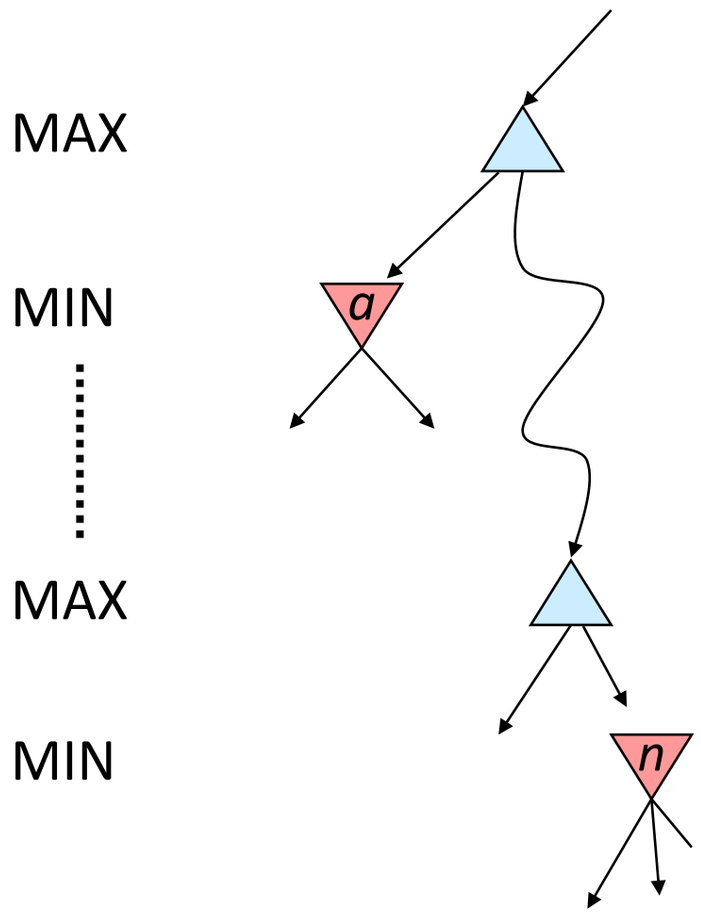


We can prune when: min node won't be higher than 2, while parent max has seen something larger in another branch

The order of generation matters: more pruning is possible if good moves come first

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 α be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than α, 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 Implementation

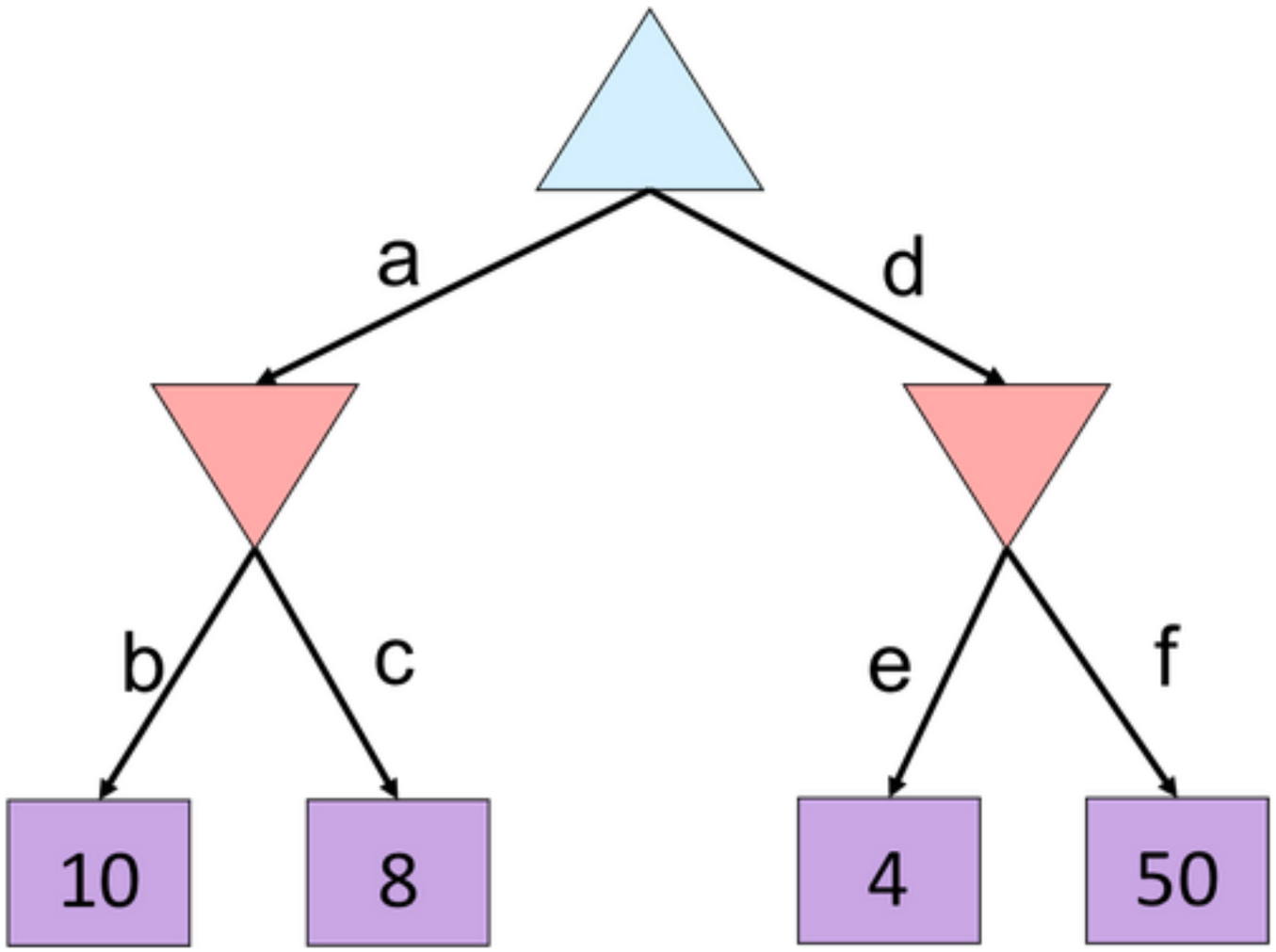
α: MAX's best option on path to root
β: MIN's best option on path to root

```
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
```

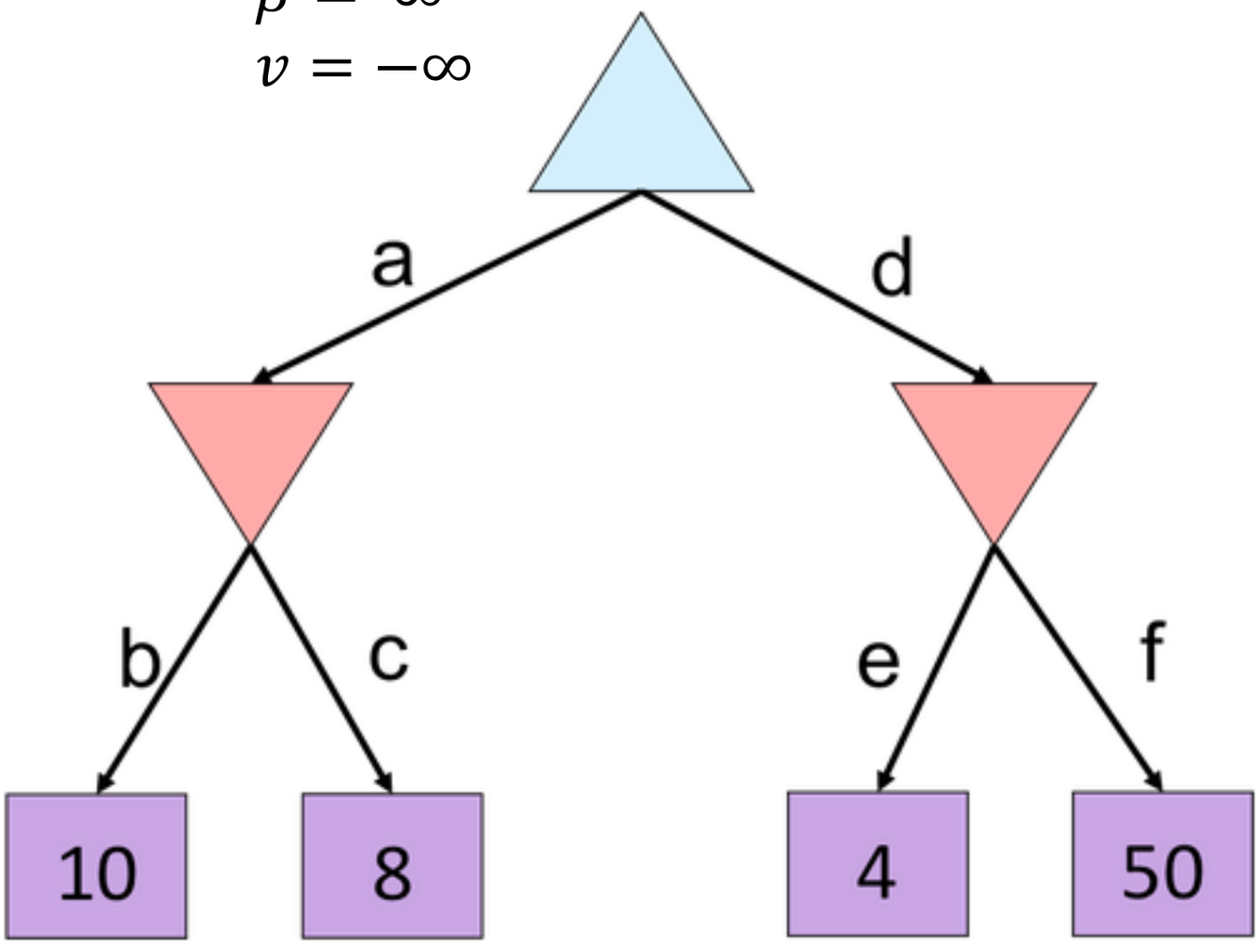
Minimax Example

- What is the value of the blue triangle?



Alpha-Beta Small Example

$\alpha = -\infty$
$\beta = \infty$
$v = -\infty$



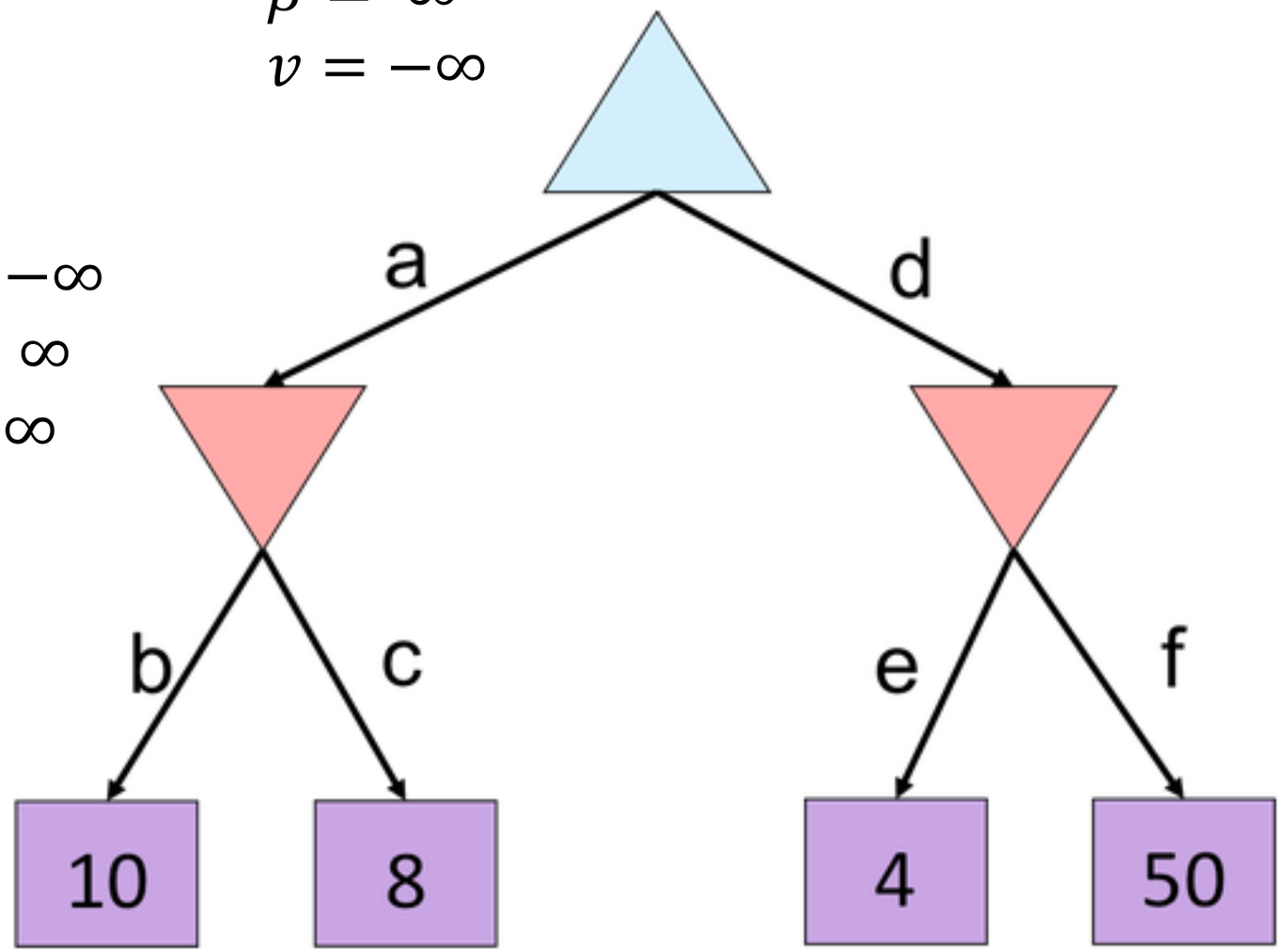

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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 Small Example

$\alpha = -\infty$
$\beta = \infty$
$v = -\infty$

$\alpha = -\infty$
$\beta = \infty$
$v = \infty$



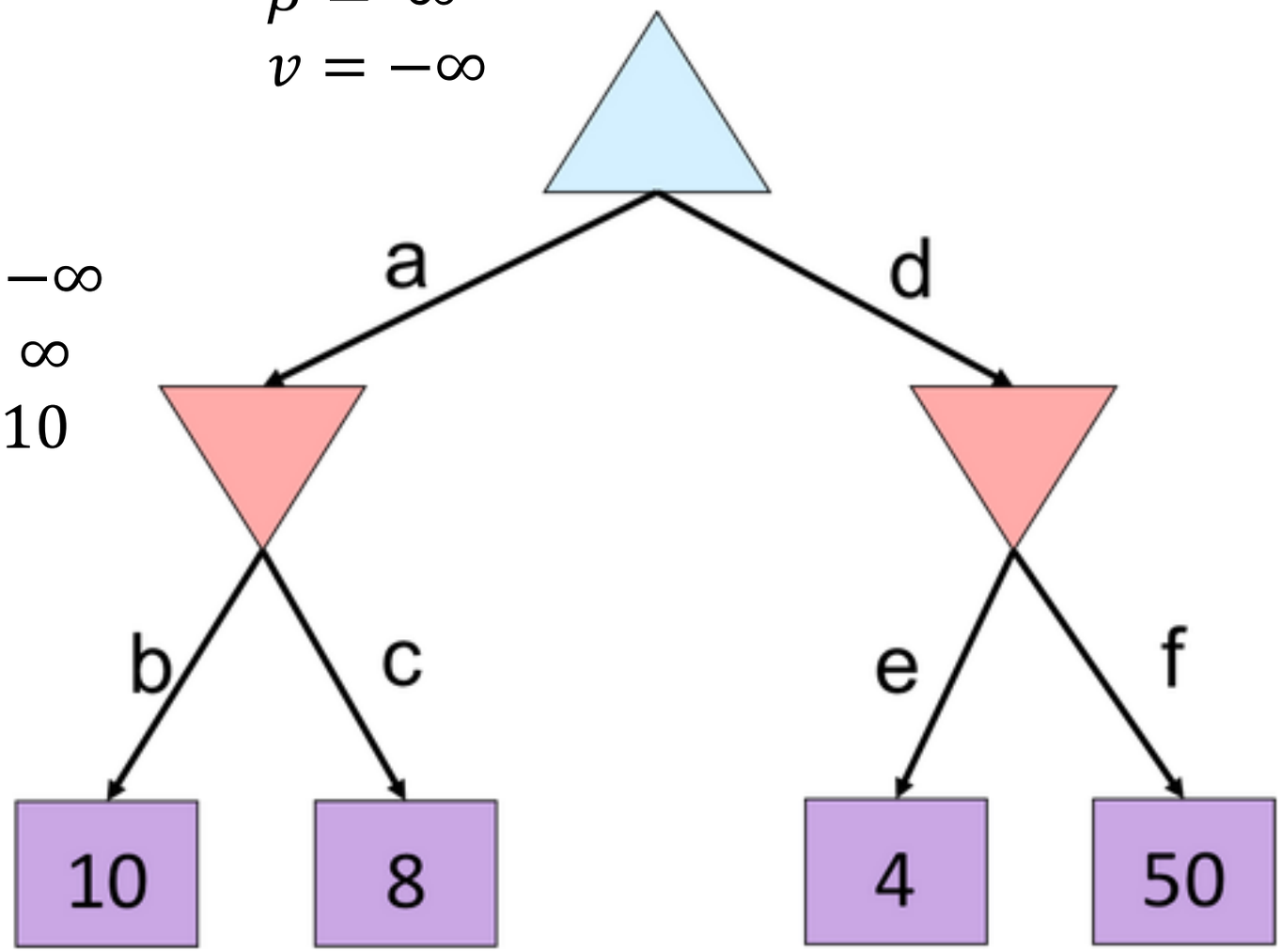

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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 Small Example

$\alpha = -\infty$
$\beta = \infty$
$v = -\infty$

$\alpha = -\infty$
$\beta = \infty$
$v = 10$

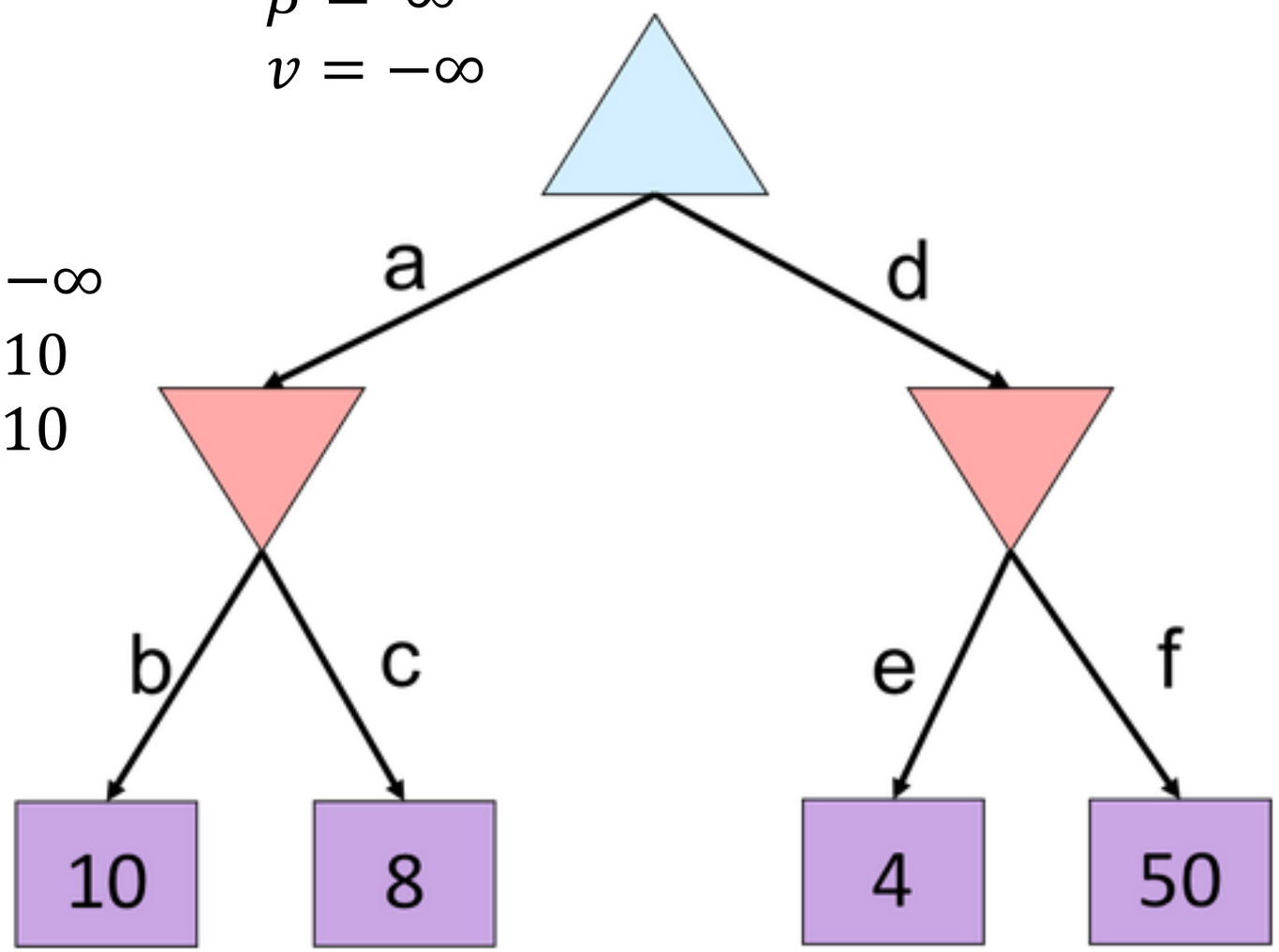



```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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 Small Example

$\alpha = -\infty$
$\beta = \infty$
$v = -\infty$



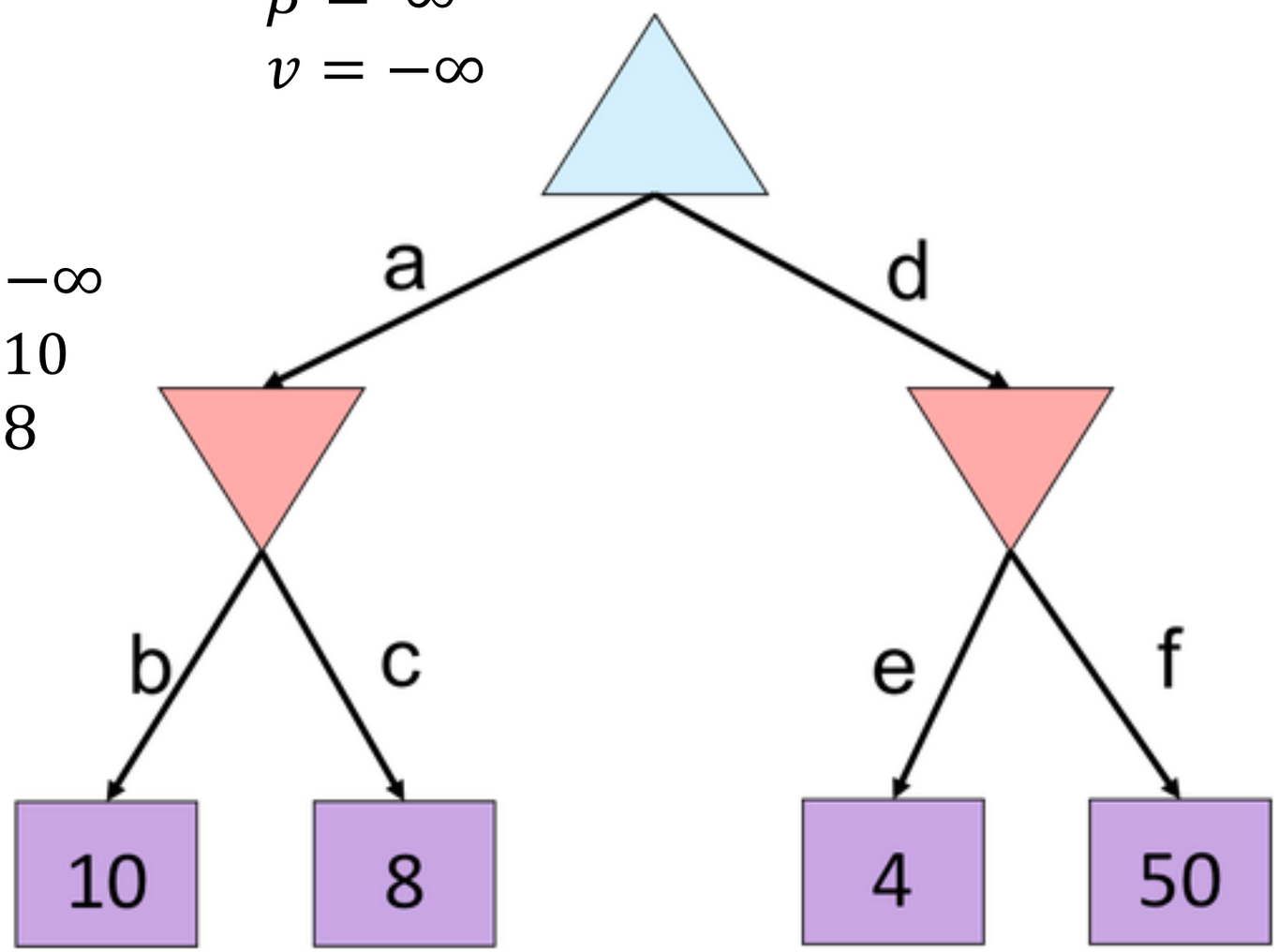
$\alpha = -\infty$
$\beta = 10$
$v = 10$

```
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 Small Example

$\alpha = -\infty$
$\beta = \infty$
$v = -\infty$



$\alpha = -\infty$
$\beta = 10$
$v = 8$

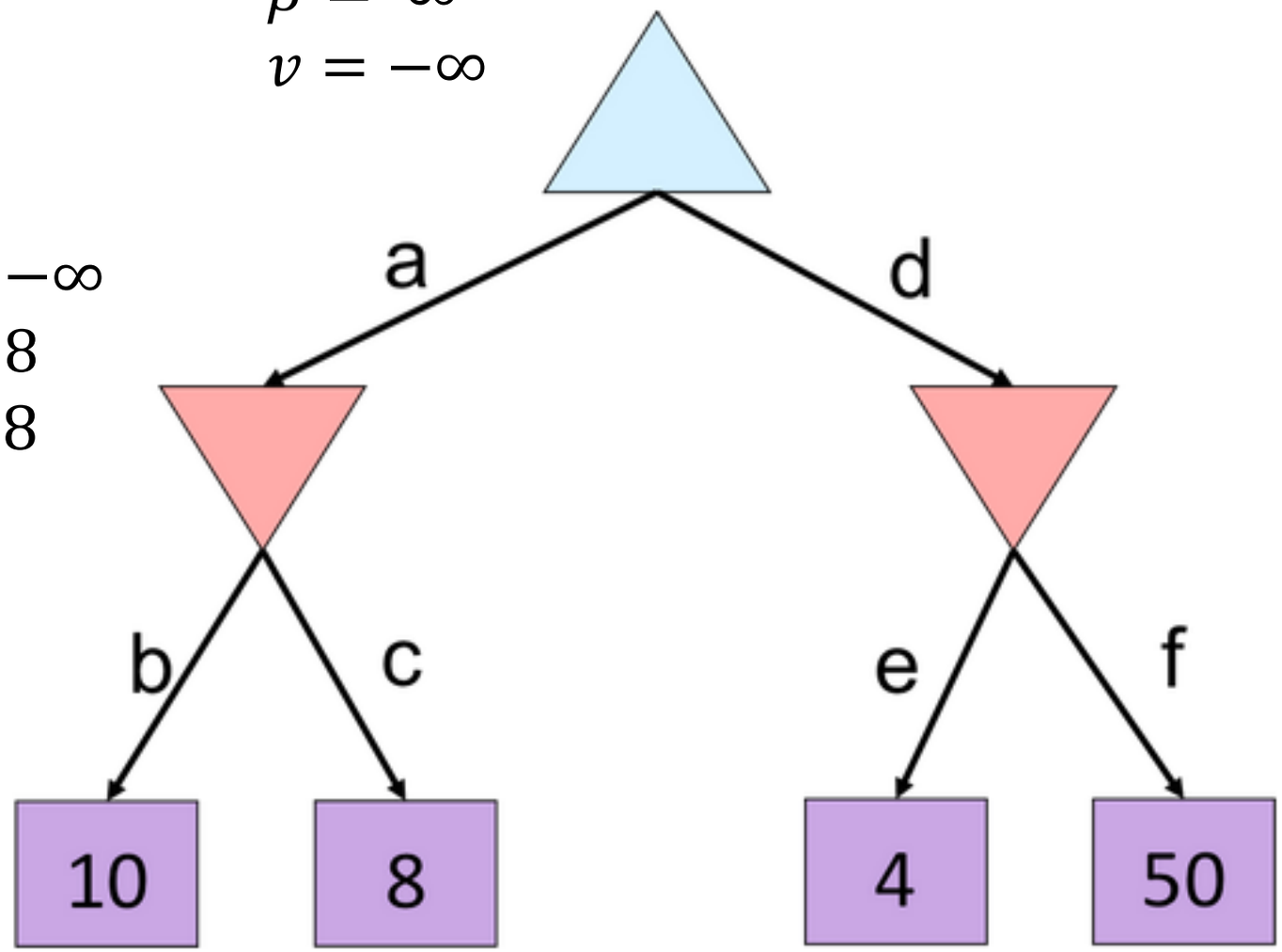
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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 Small Example

$\alpha = -\infty$
$\beta = \infty$
$v = -\infty$

$\alpha = -\infty$
$\beta = 8$
$v = 8$

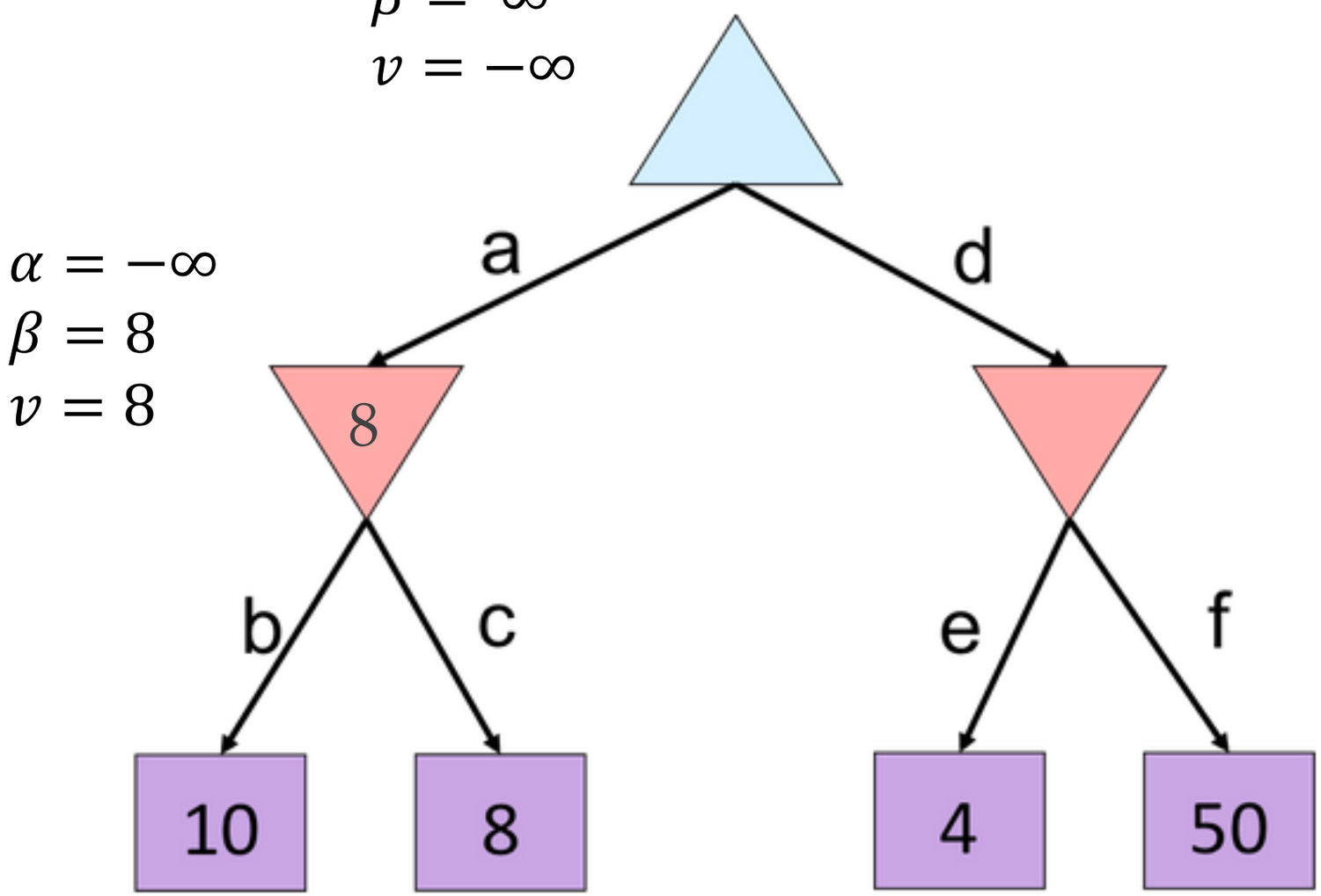



```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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 Small Example

$\alpha = -\infty$
$\beta = \infty$
$v = -\infty$



$\alpha = -\infty$
$\beta = 8$
$v = 8$

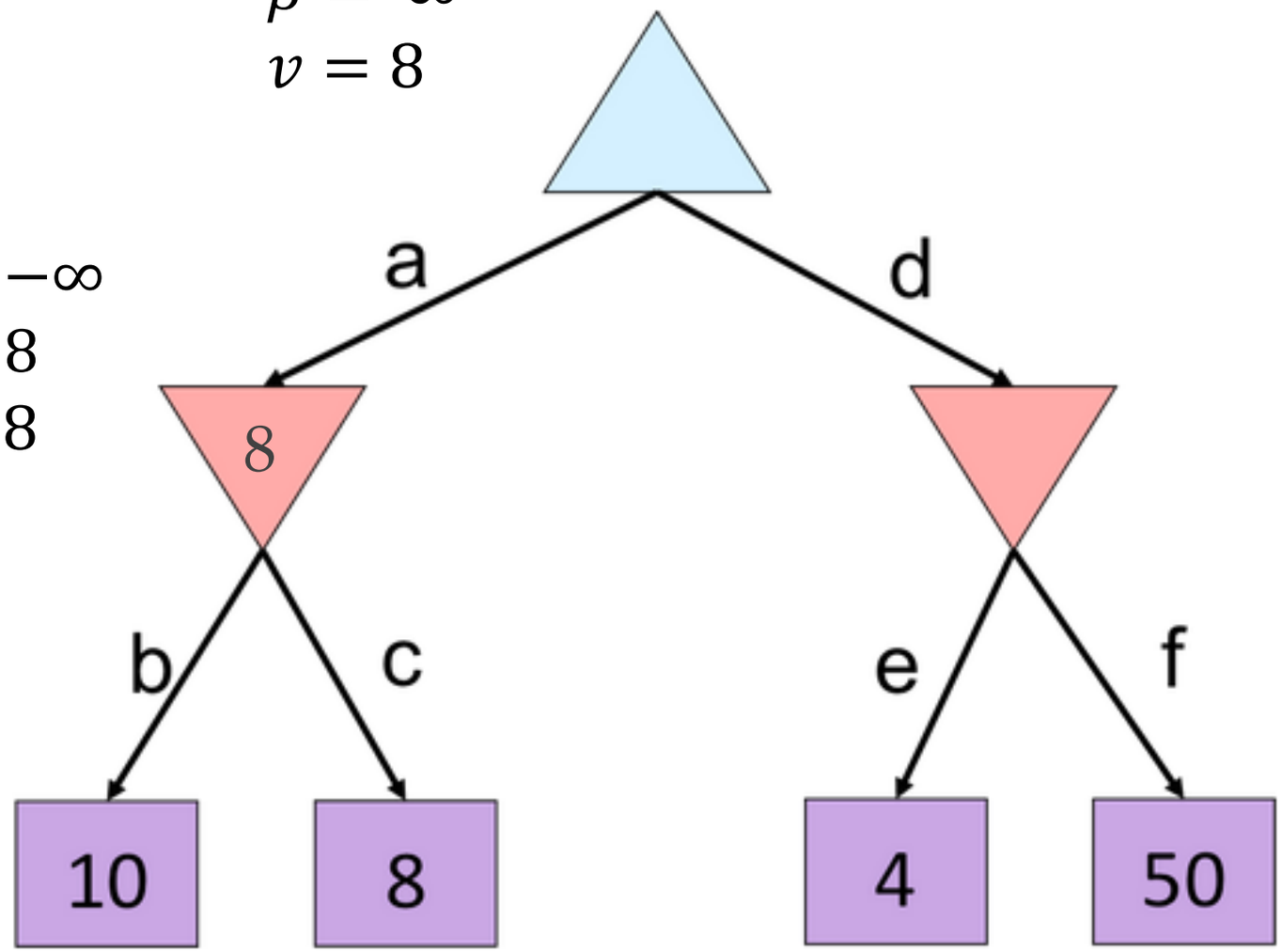
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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 Small Example

$\alpha = -\infty$
$\beta = \infty$
$v = 8$

$\alpha = -\infty$
$\beta = 8$
$v = 8$



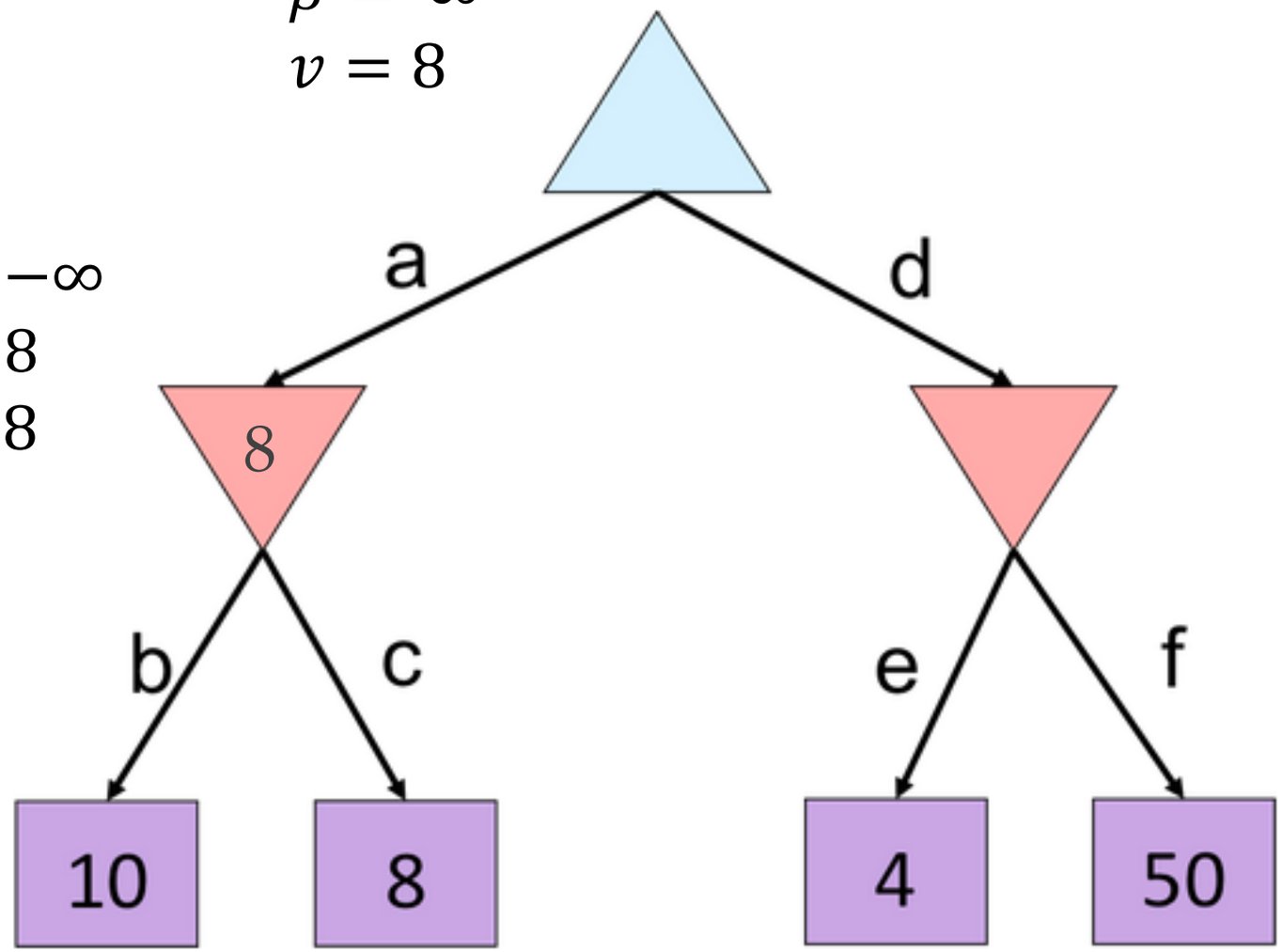

```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

```
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 Small Example

$\alpha = 8$
$\beta = \infty$
$v = 8$

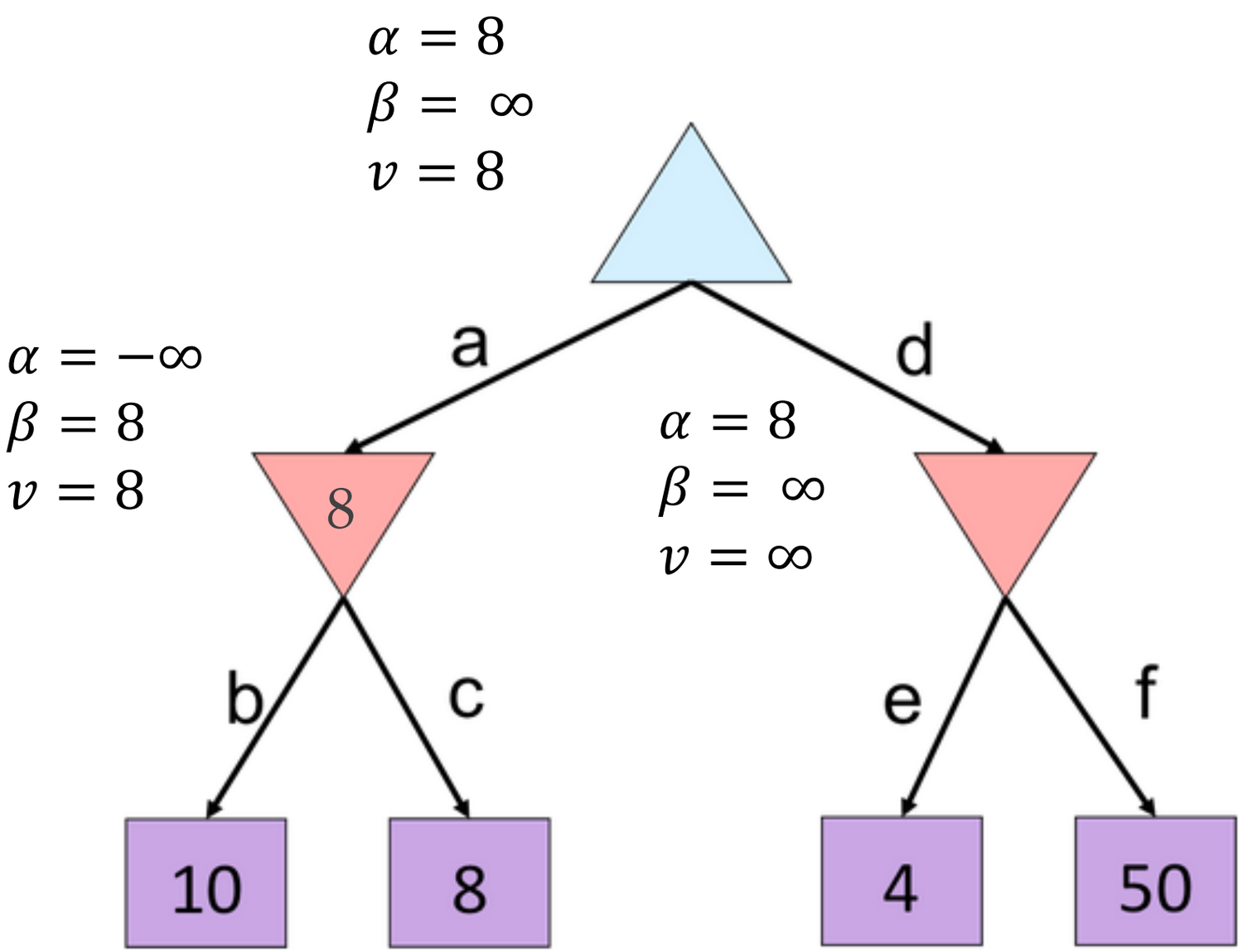
$\alpha = -\infty$
$\beta = 8$
$v = 8$




```
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
```

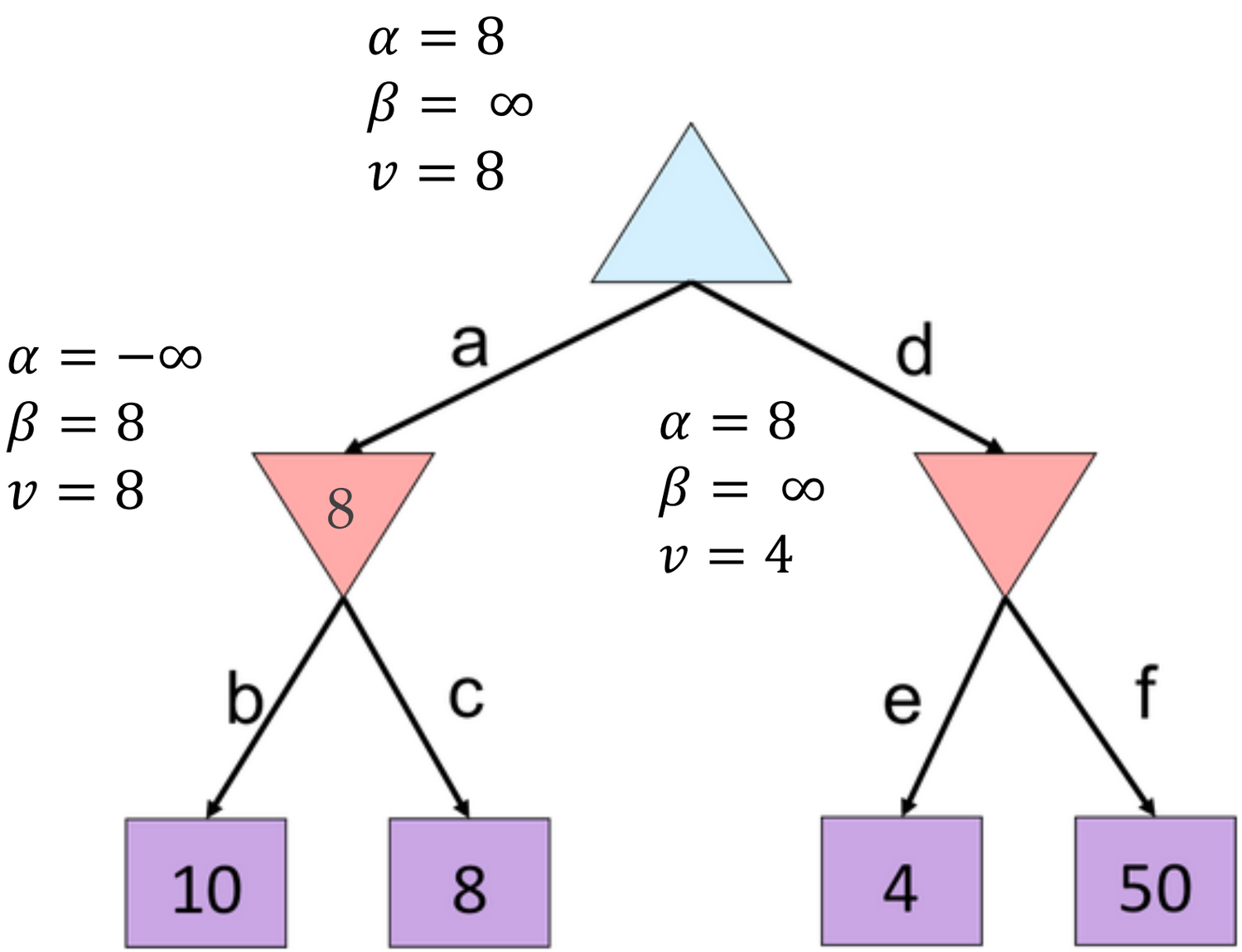
Alpha-Beta Small Example




```
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
```

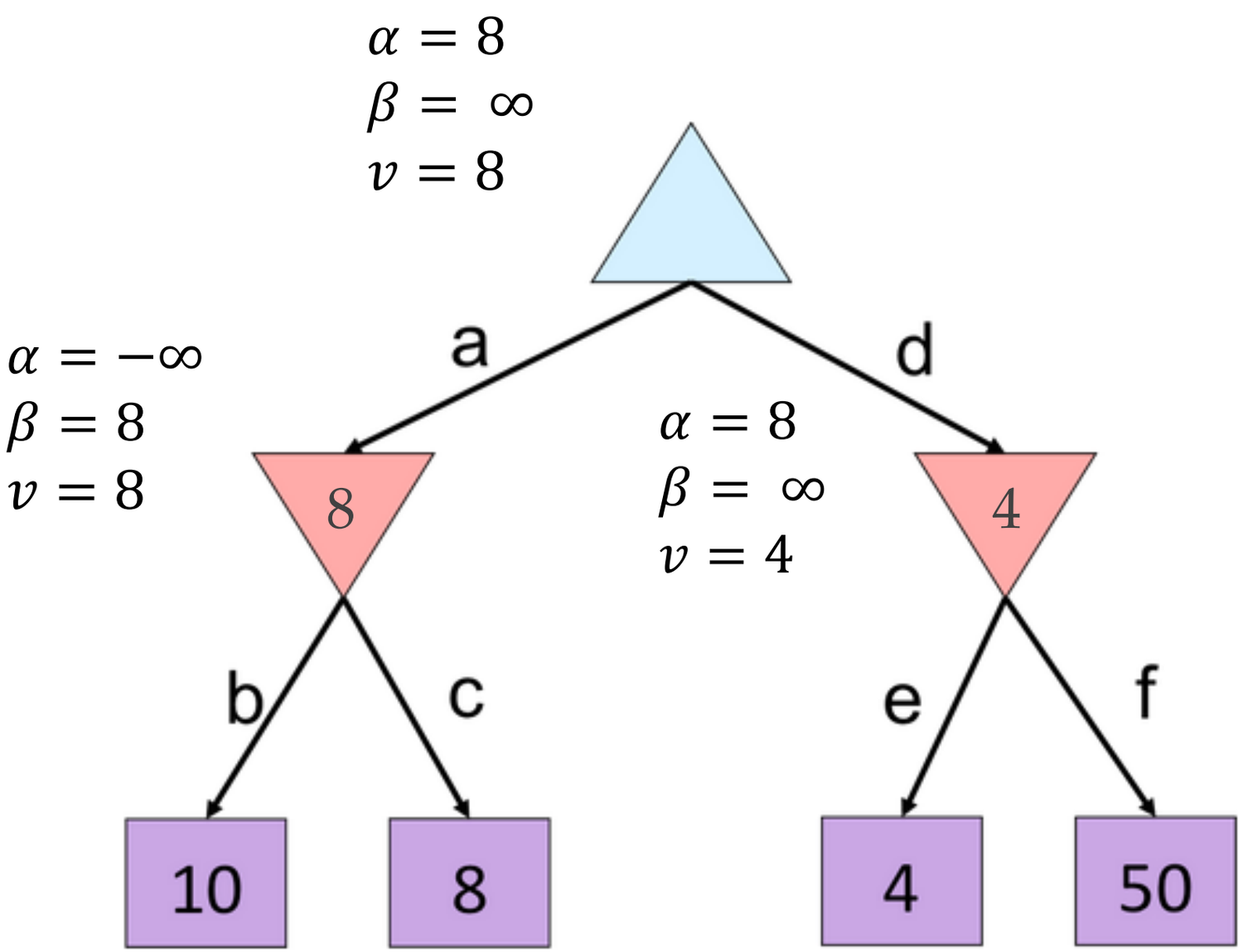
Alpha-Beta Small Example




```
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 Small Example




```
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
```

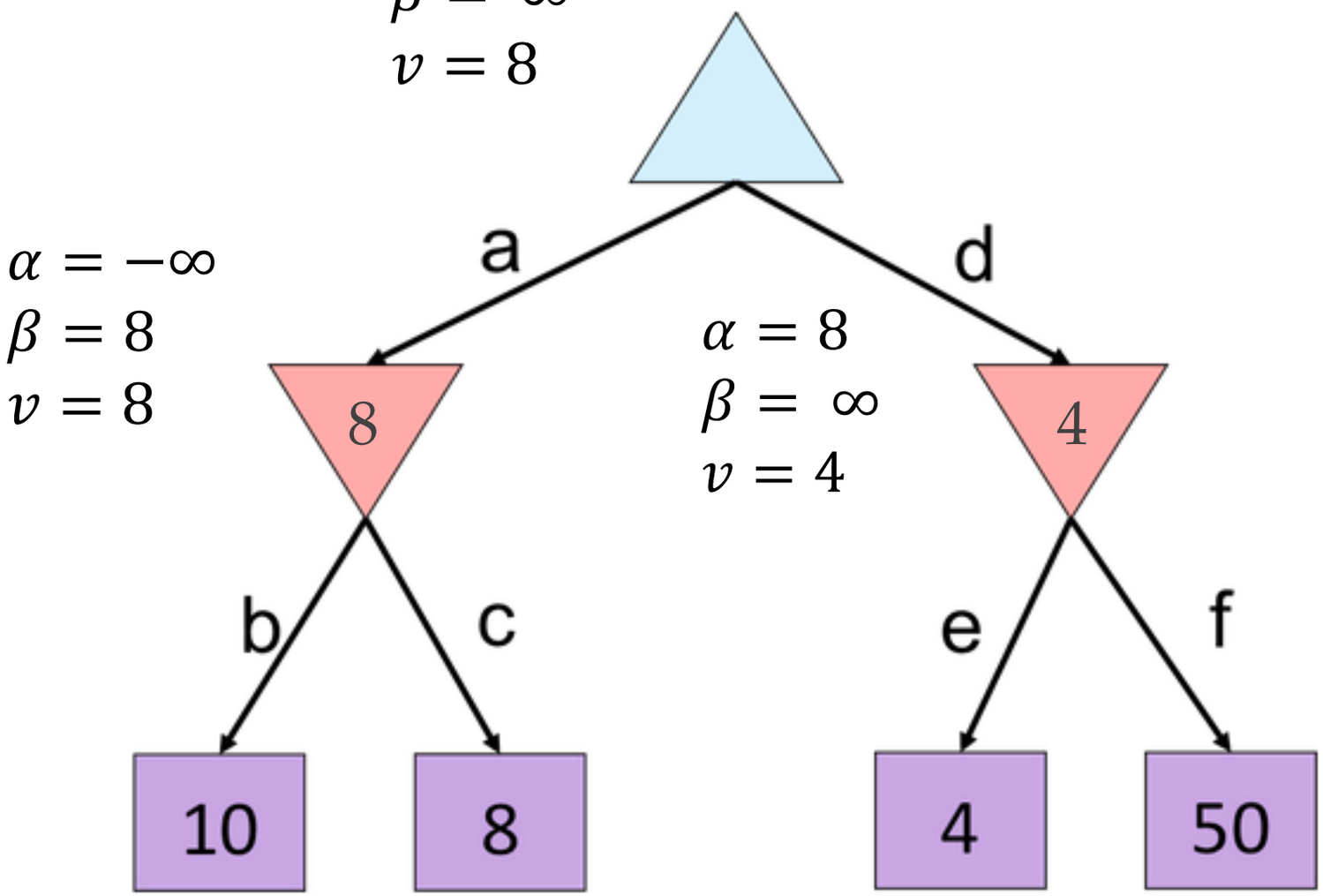
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Small Example

$\alpha = 8$
$\beta = \infty$
$v = 8$

$\alpha = -\infty$
$\beta = 8$
$v = 8$

$\alpha = 8$
$\beta = \infty$
$v = 4$




```
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
```

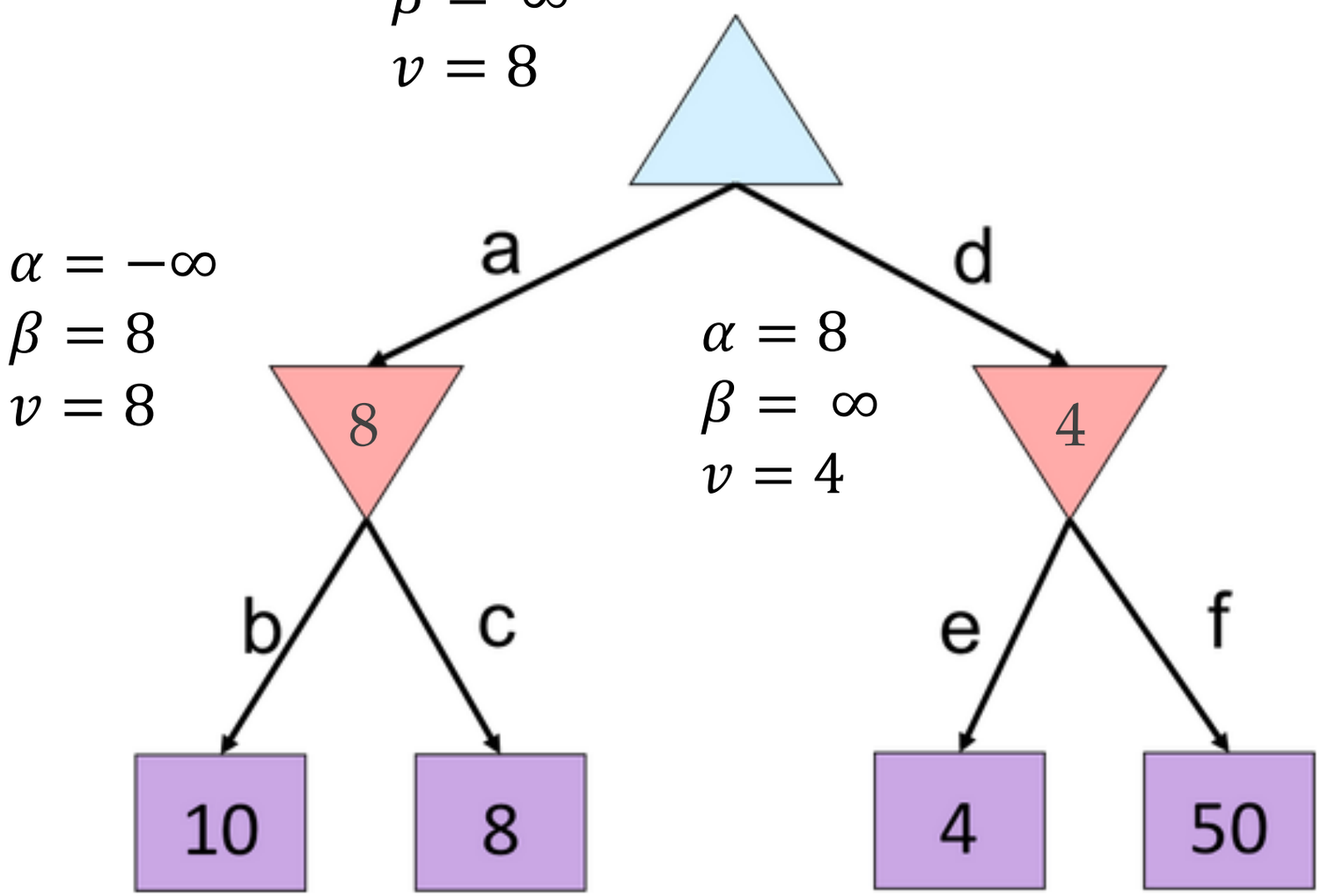
```
def min-value(state , α, β):
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        if v ≤ α return v
        β = min(β, v)
    return v
```

Alpha-Beta Small Example

$\alpha = 8$
$\beta = \infty$
$v = 8$

$\alpha = -\infty$
$\beta = 8$
$v = 8$

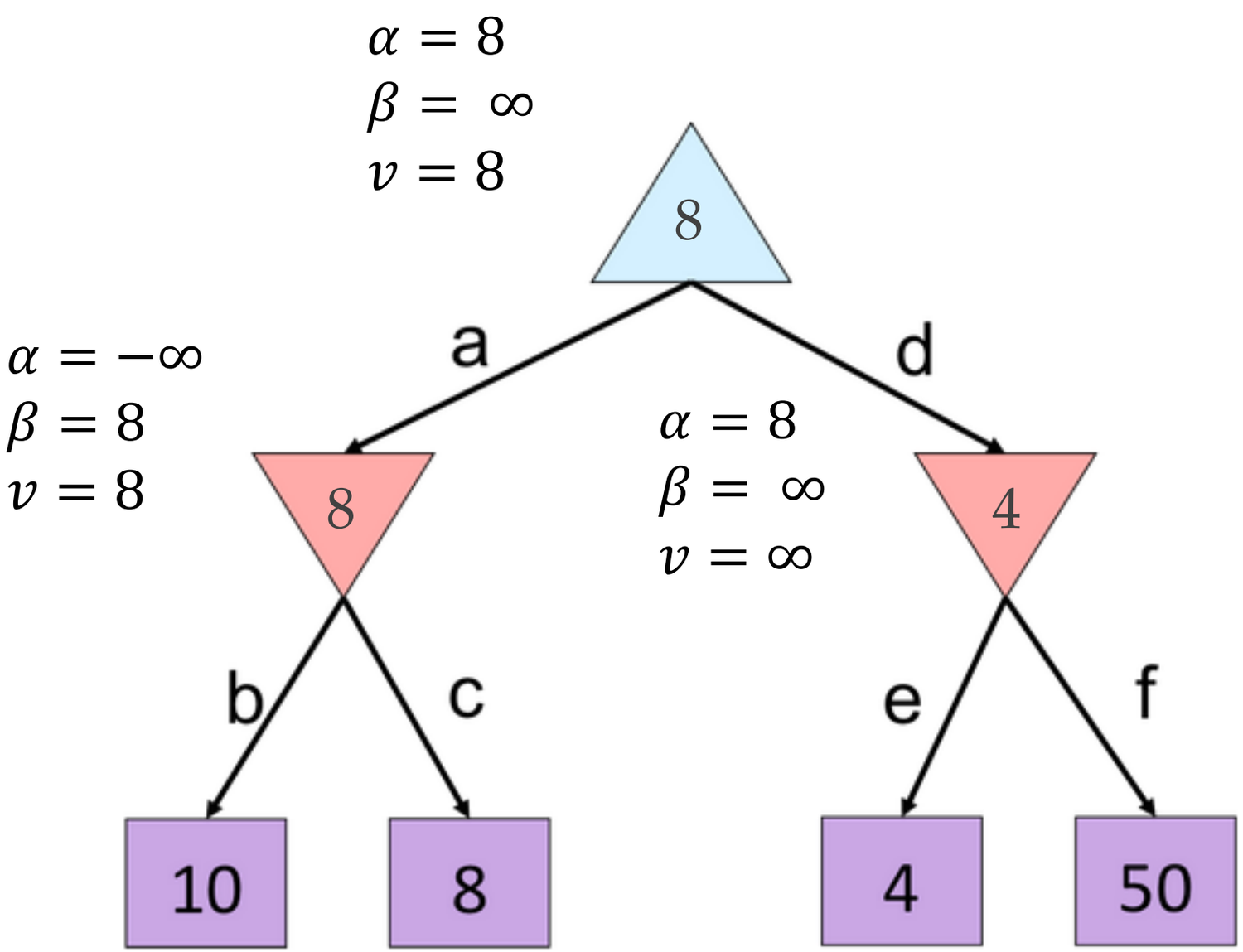
$\alpha = 8$
$\beta = \infty$
$v = 4$




```
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 Small Example




```
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
```

Quiz: Minimax

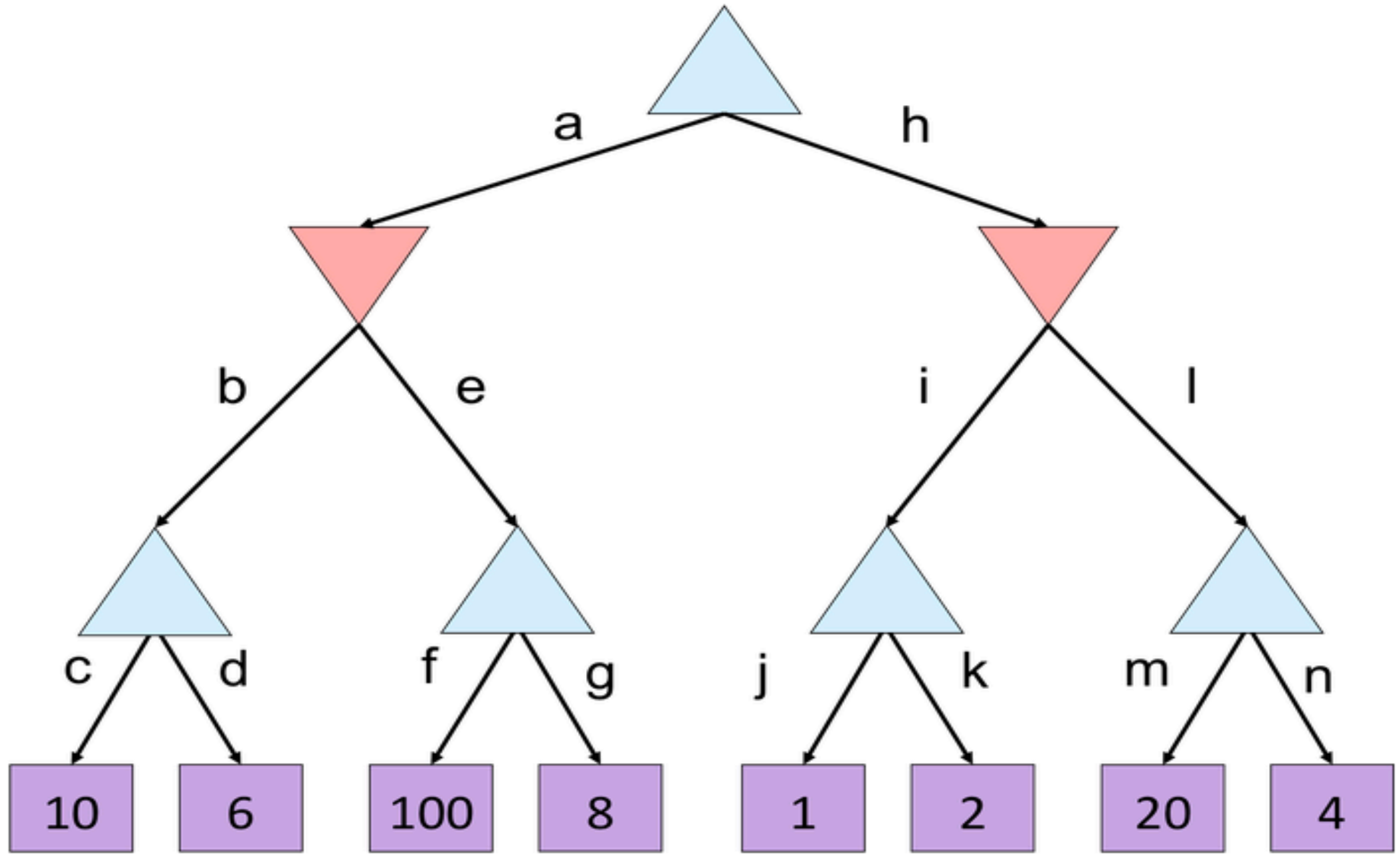
- What is the value of the root node?

A. 10

B. 100

C. 2

D. 4



Quiz: Alpha Beta

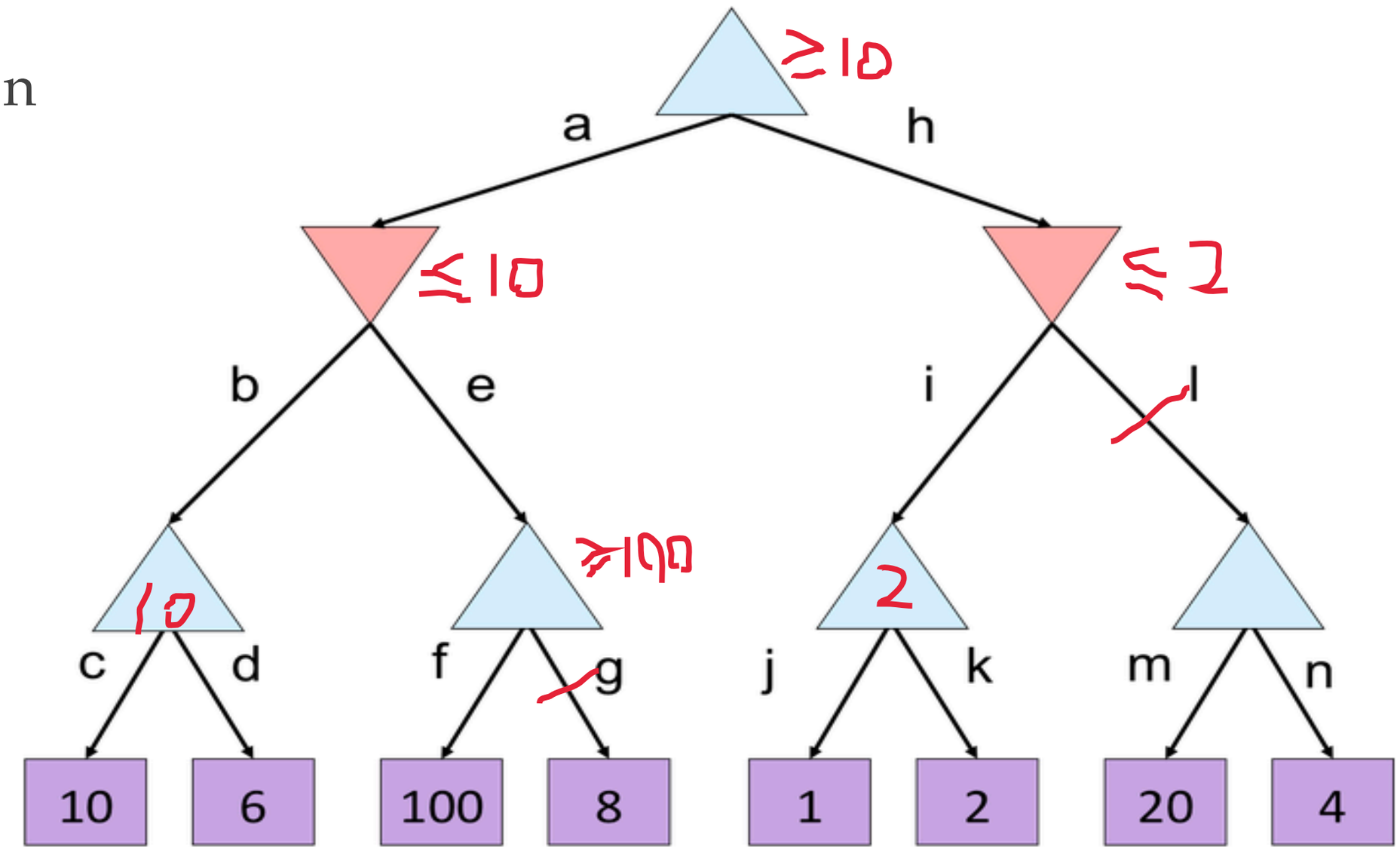
- Which branches are pruned?

A. e, l

B. g, l, m, n

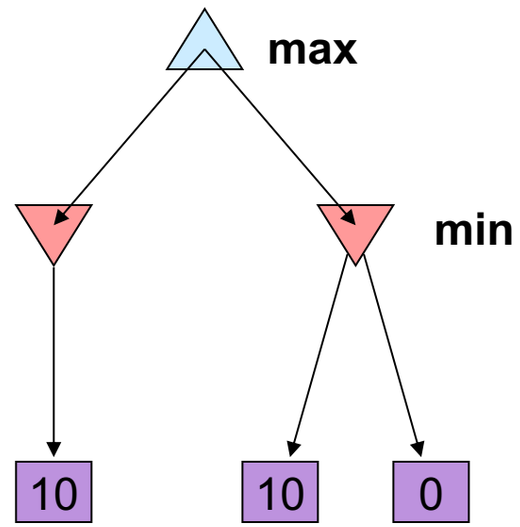
C. g, k, n

D. g, n



Alpha-Beta Pruning Properties

- **Theorem**: This pruning has no effect on minimax value computed for the root!
- Good child ordering improves effectiveness of pruning
  - Iterative deepening helps with this
- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - 1M nodes/move => depth=8, respectable



- This is a simple example of metareasoning (computing about what to compute)