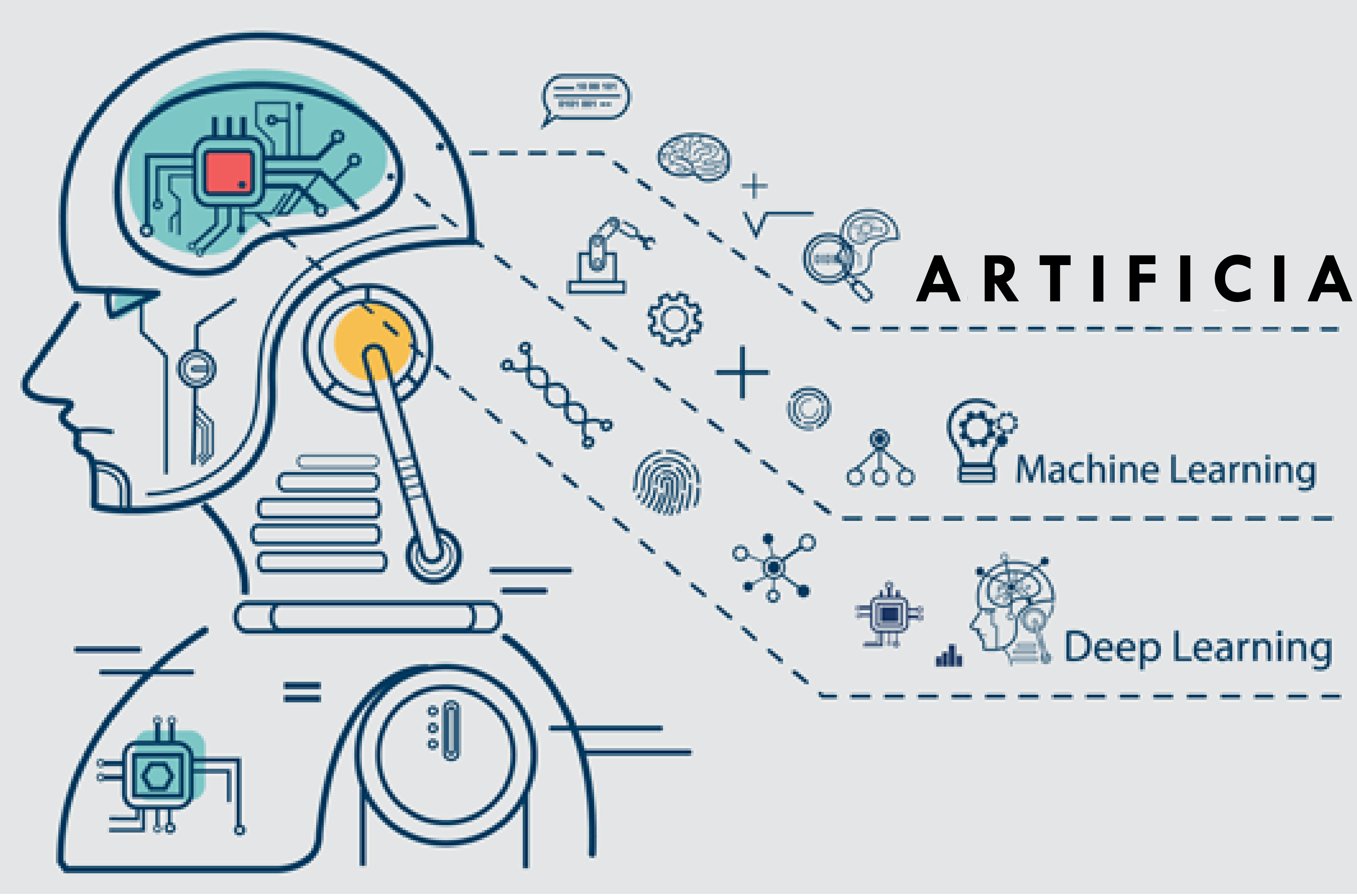


ARTIFICIAL INTELLIGENCE

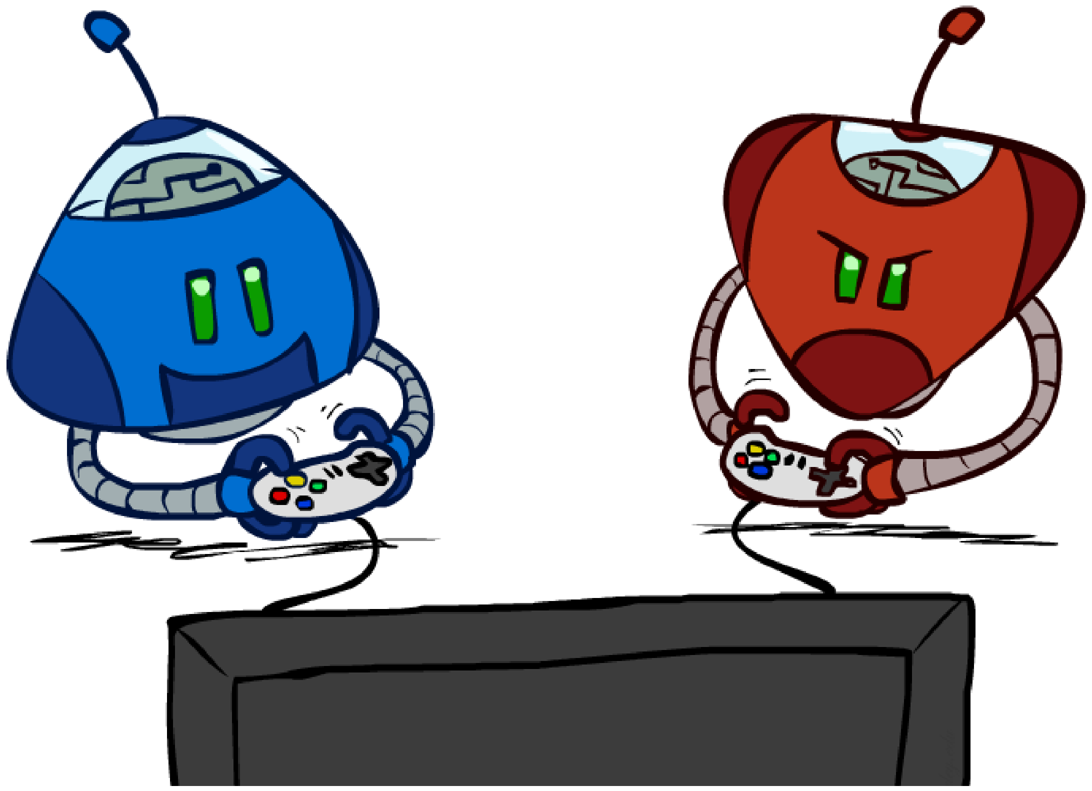
Dr. Seemab latif

Lecture 5

9th Oct 2023

AI: Representation and Problem Solving

Adversarial Search



Slide credits: Pat Virtue, http://ai.berkeley.edu

Outline

- History / Overview
- Zero-Sum Games (Minimax)
- Evaluation Functions
- Search Efficiency (α-β Pruning)
- Games of Chance (Expectimax)

Game Playing State-of-the-Art

Checkers:

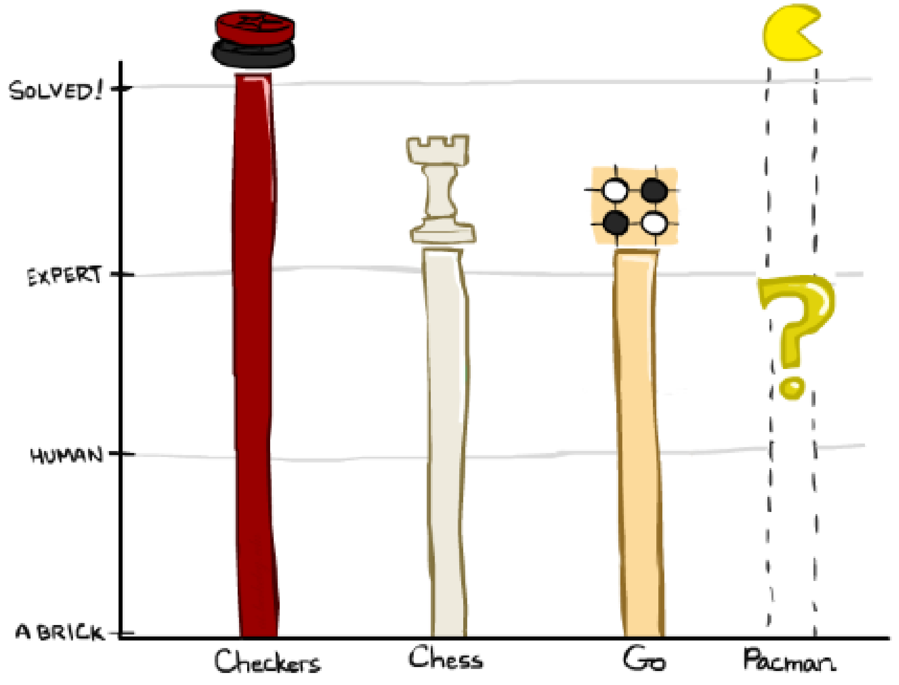
- 1950: First computer player.
- 1959: Samuel's self-taught program.
- 1994: First computer world champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame.
- 2007: Checkers solved! Endgame database of 39 trillion states

Chess:

- 1945-1960: Zuse, Wiener, Shannon, Turing, Newell & Simon, McCarthy.
- 1960s onward: gradual improvement under "standard model"
- 1997: special-purpose chess machine Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second and extended some lines of search up to 40 ply. Current programs running on a PC rate > 3200 (vs 2870 for Magnus Carlsen).

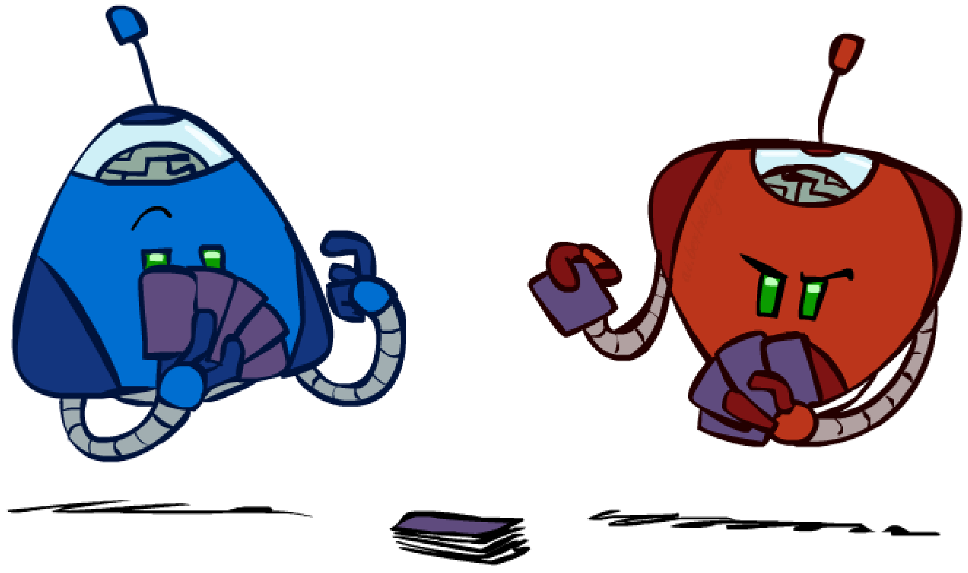
Go:

- 1968: Zobrist's program plays legal Go, barely (b>300!)
- 2005-2014: Monte Carlo tree search enables rapid advances: current programs beat strong amateurs, and professionals with a 3-4 stone handicap.
- 2015: AlphaGo from DeepMind beats Lee Sedol

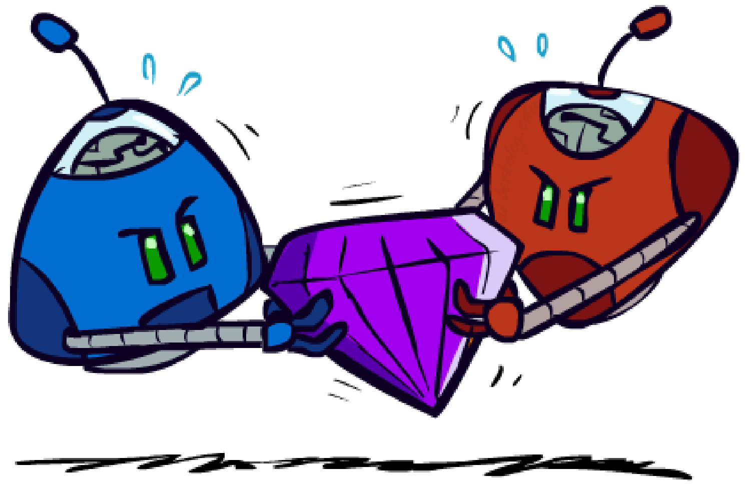


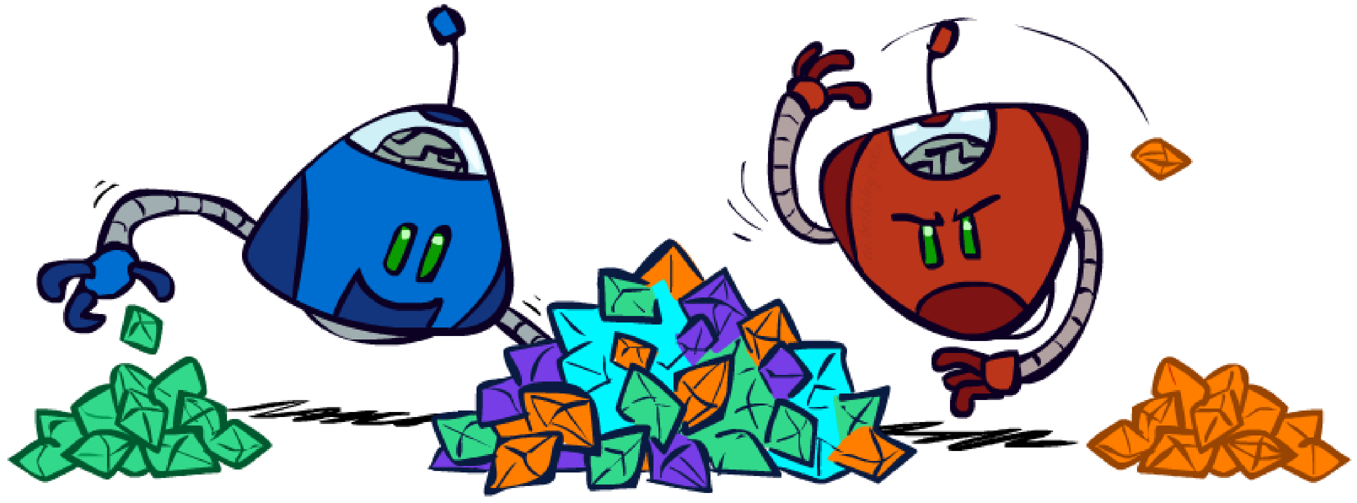
Types of Games

- Deterministic or stochastic?
- Perfect information (fully observable)?
- One, two, or more players?
- Turn-taking or simultaneous?
- Zero sum?



Zero-Sum Games





Zero-Sum Games

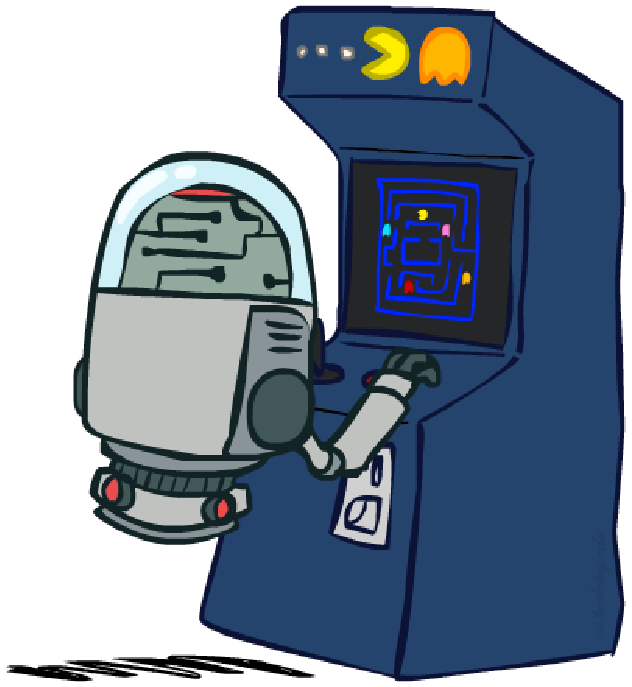
- Agents have ***opposite*** utilities
- Pure competition:
  - One ***maximizes***, the other ***minimizes***

General Games

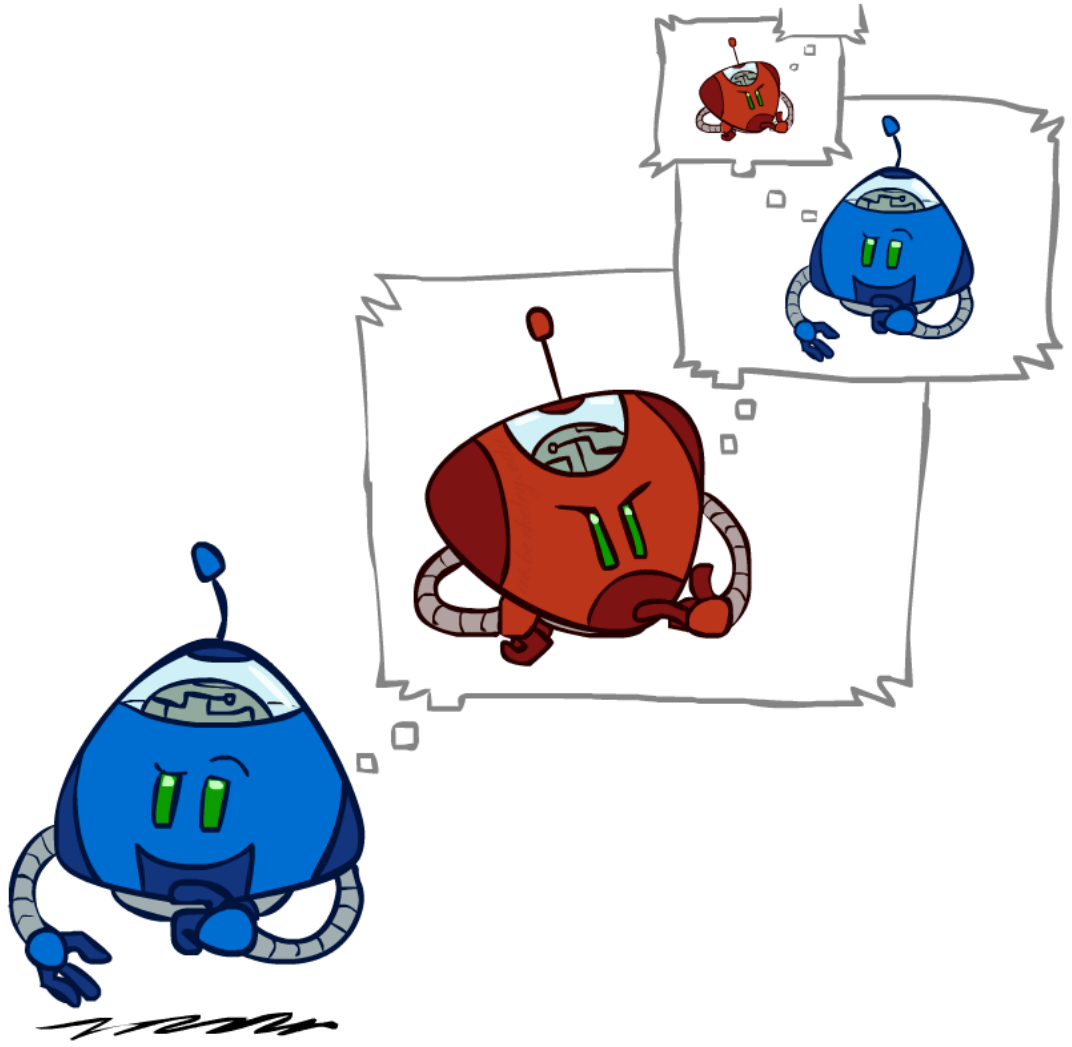
- Agents have ***independent*** utilities
- Cooperation, indifference, competition, shifting alliances, and more are all possible

Standard Games

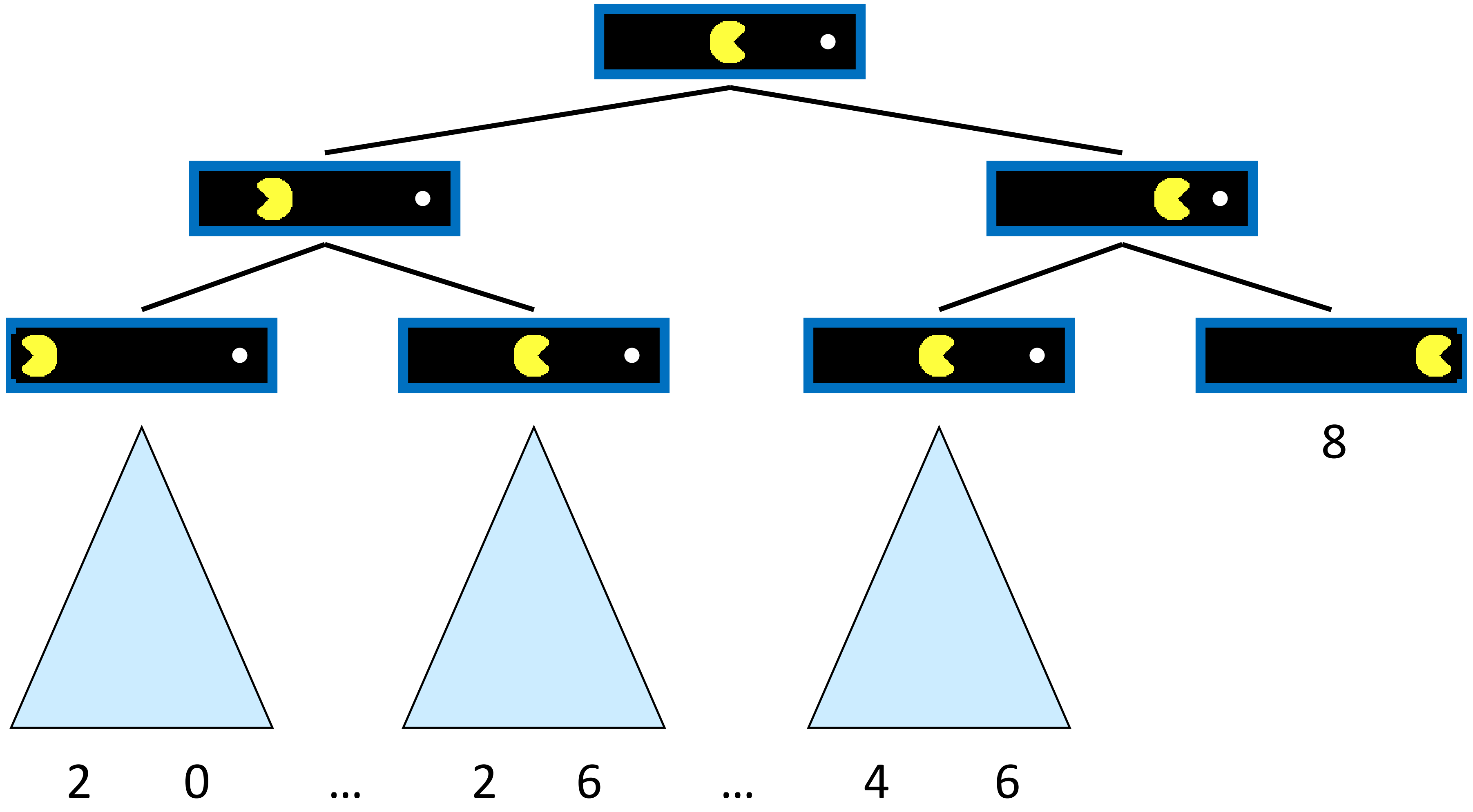
- Standard games are deterministic, observable, two-player, turn-taking, zero-sum
- Game formulation:
  - Initial state: s_0
  - Players: Player(s) indicates whose move it is
  - Actions: Actions(s) for player on move
  - Transition model: Result(s,a)
  - Terminal test: Terminal-Test(s)
  - Terminal values: Utility(s,p) for player p
    - Or just Utility(s) for player making the decision at root



Adversarial Search

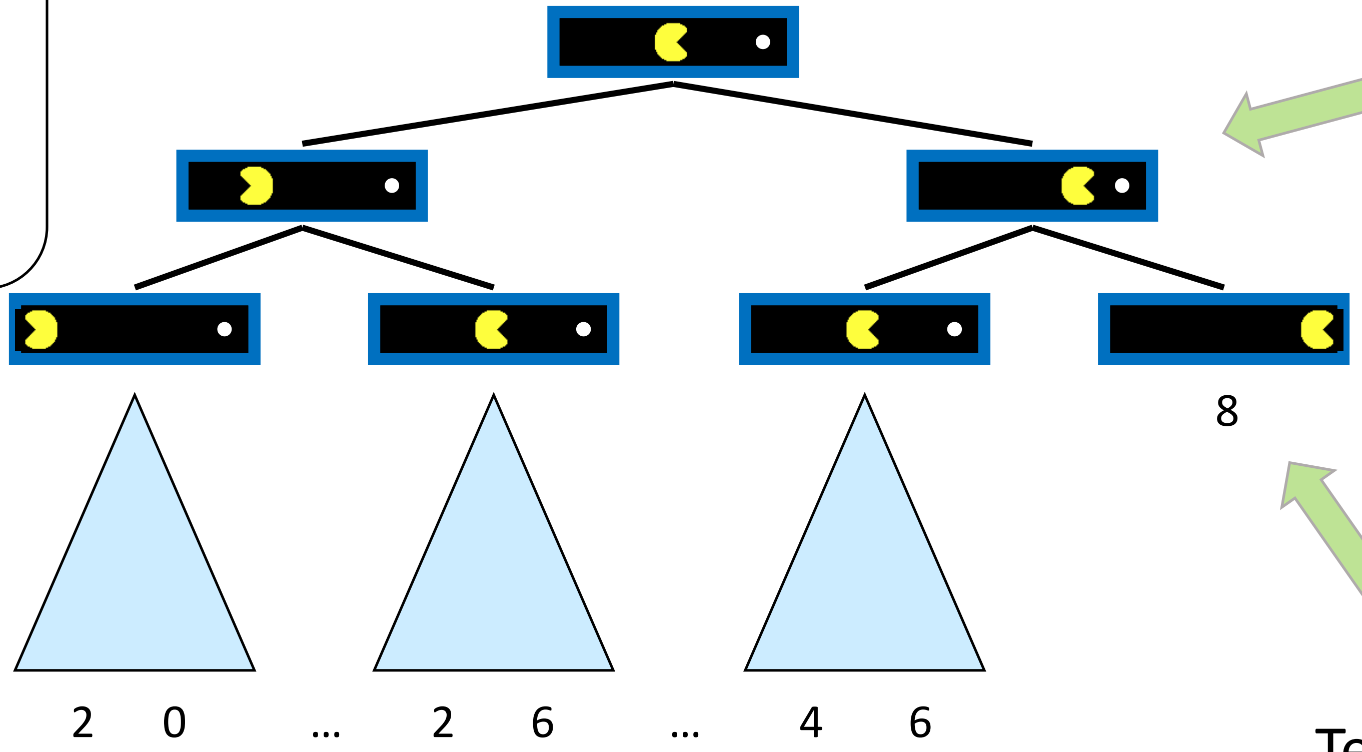


Single-Agent Trees



Value of a State

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



Non-Terminal States:

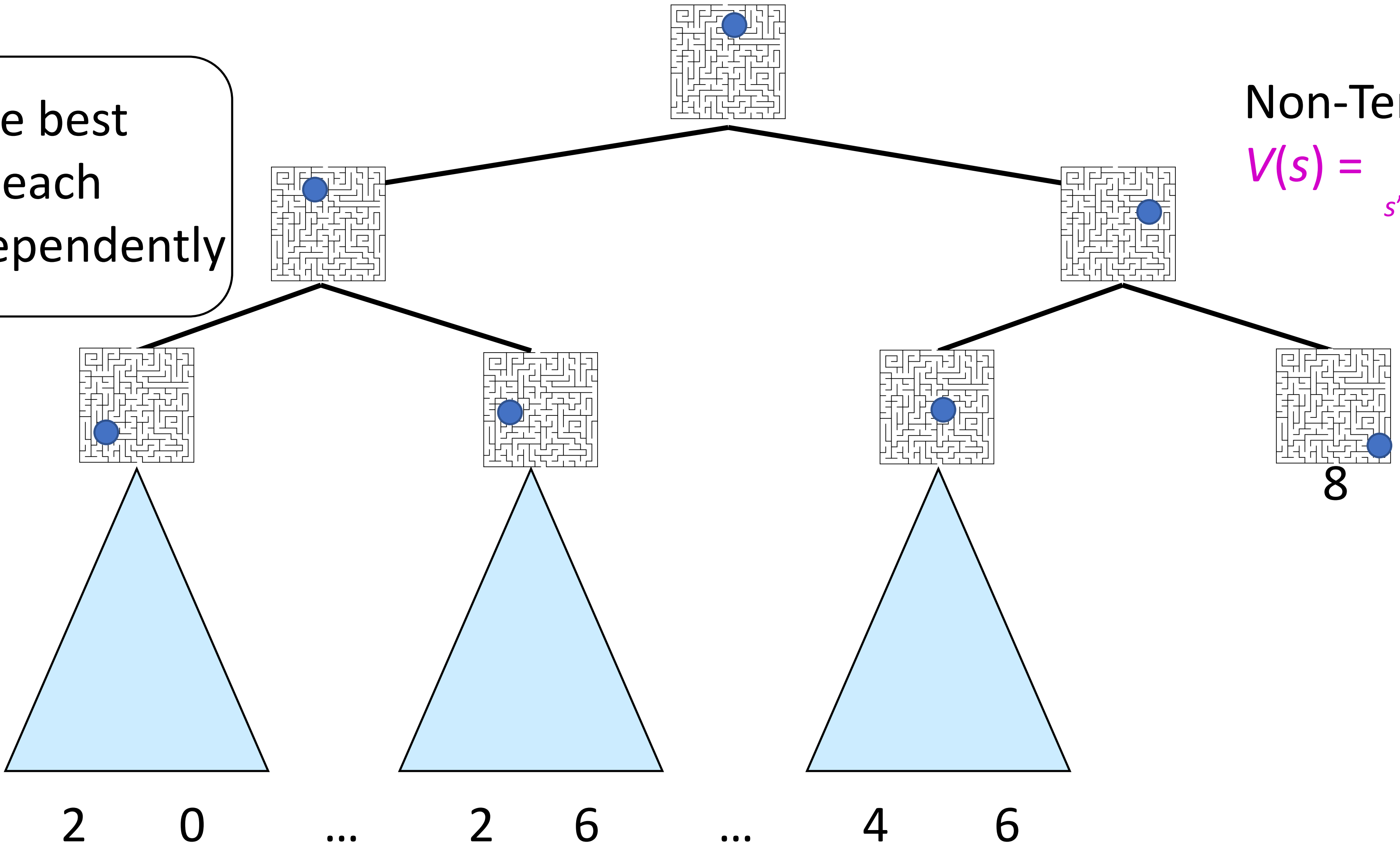
$V(s) = \max_{s' \in \text{successors}(s)} V(s')$

Terminal States:

$V(s)$ = known

Idea 1: Many Single-Agent Trees

Choose the best action for each agent independently



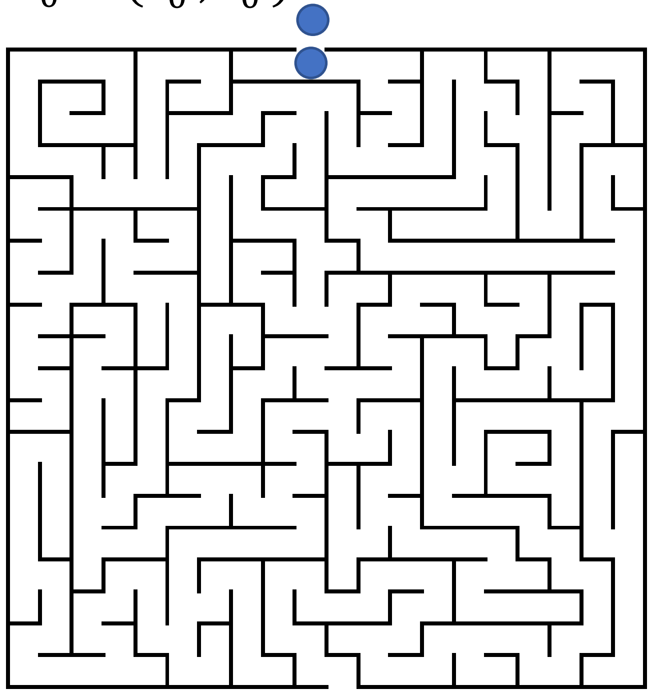
Non-Terminal States:

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

Idea 2: Joint State/Action Spaces

Combine the states and actions of the N agents

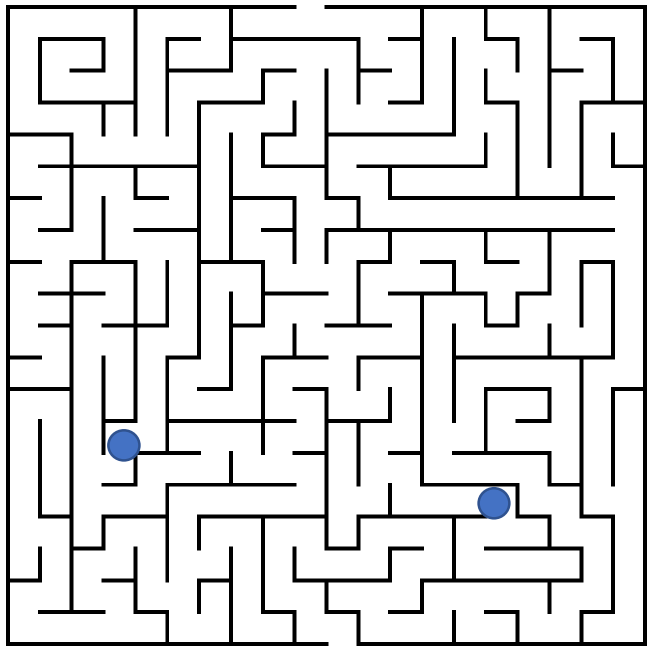
$S_0 = (S_0^A, S_0^B)$



Idea 2: Joint State/Action Spaces

Combine the states and actions of the N agents

$S_K = (S_K^A, S_K^B)$

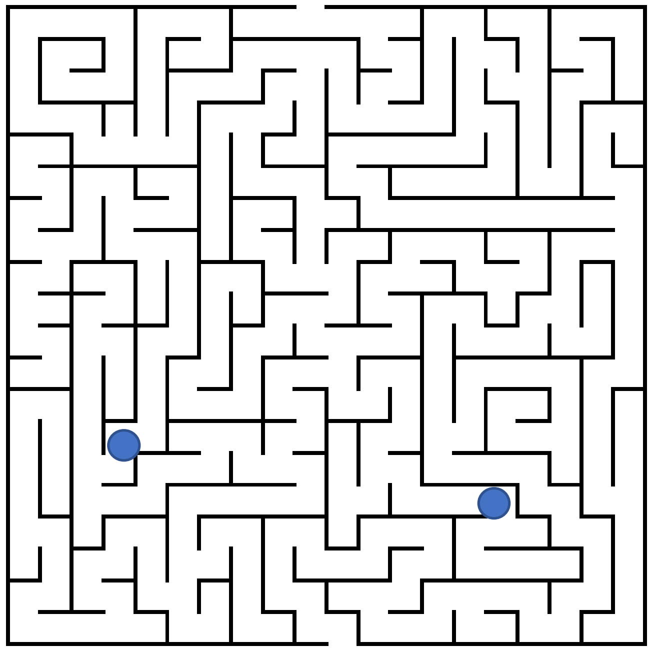


Idea 2: Joint State/Action Spaces

Search looks through all combinations of all agents' states and actions

Think of one brain controlling many agents

$S_K = (S_K^A, S_K^B)$



Idea 2: Joint State/Action Spaces

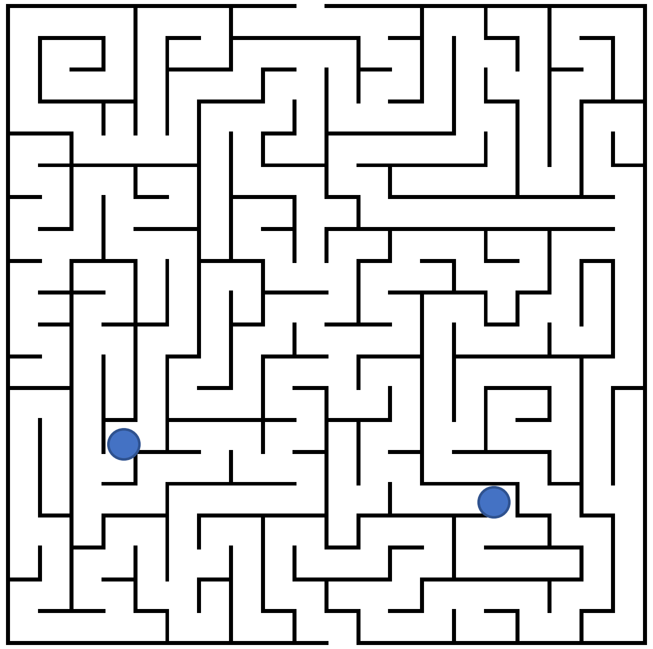
Search looks through all combinations of all agents' states and actions

Think of one brain controlling many agents

What is the size of the state space?

What is the size of the action space?

What is the size of the search tree?



Idea 3: Centralized Decision Making

Each agent proposes their actions and computer confirms the joint plan

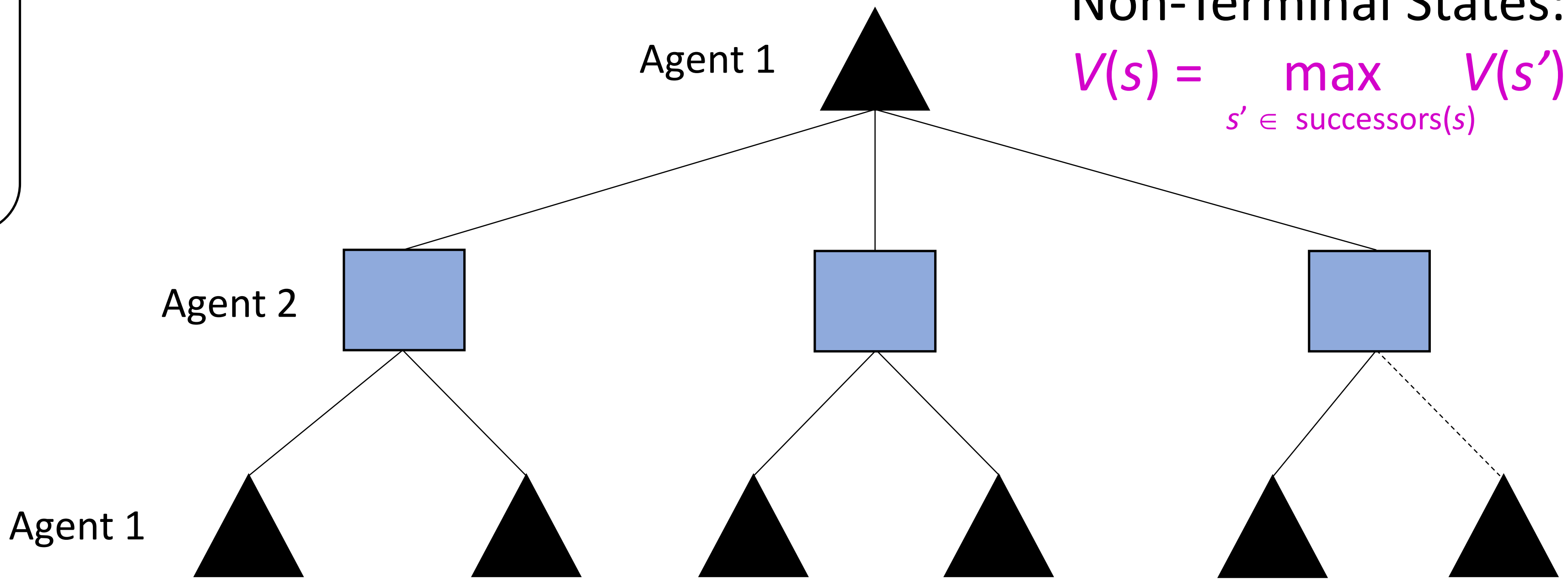
Example: Autonomous driving through intersections

https://www.youtube.com/watch?v=4pbAI40dK0A

Idea 4: Alternate Searching One Agent at a Time

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

Choose the best cascading combination of actions



Non-Terminal States:

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

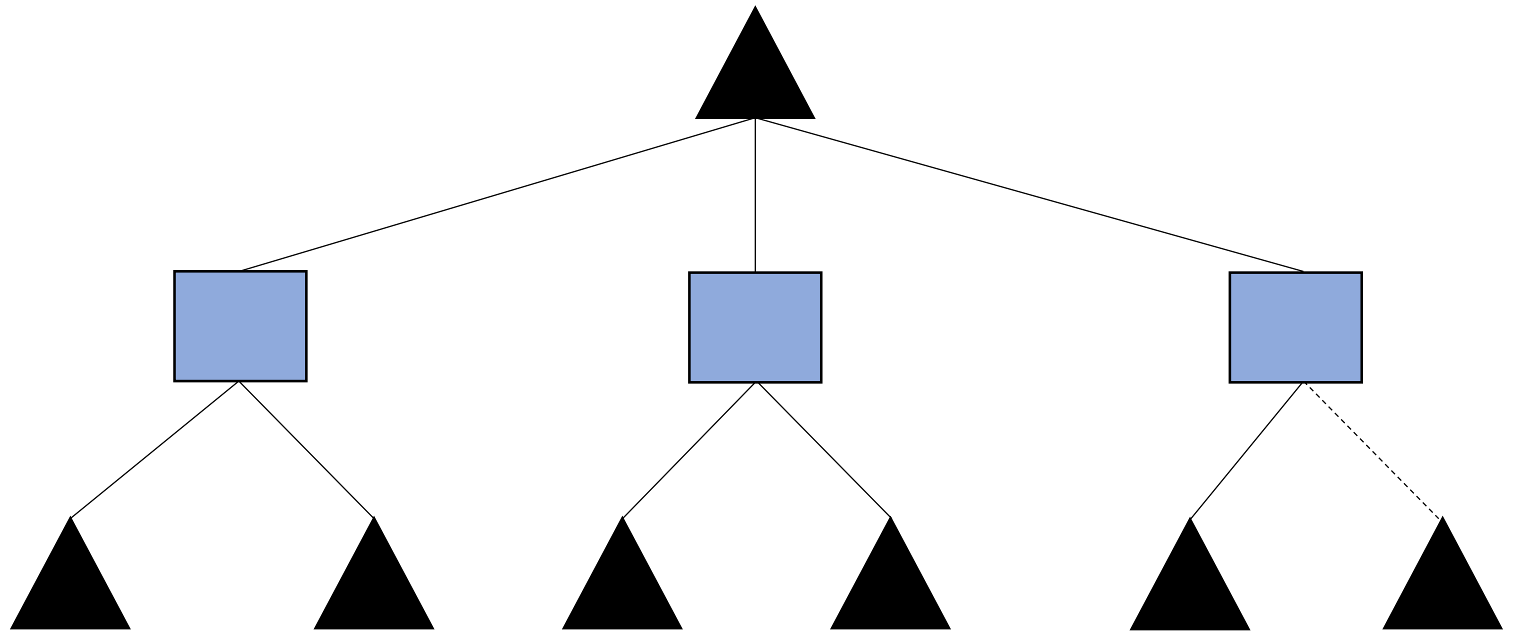
Idea 4: Alternate Searching One Agent at a Time

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

What is the size of the state space?

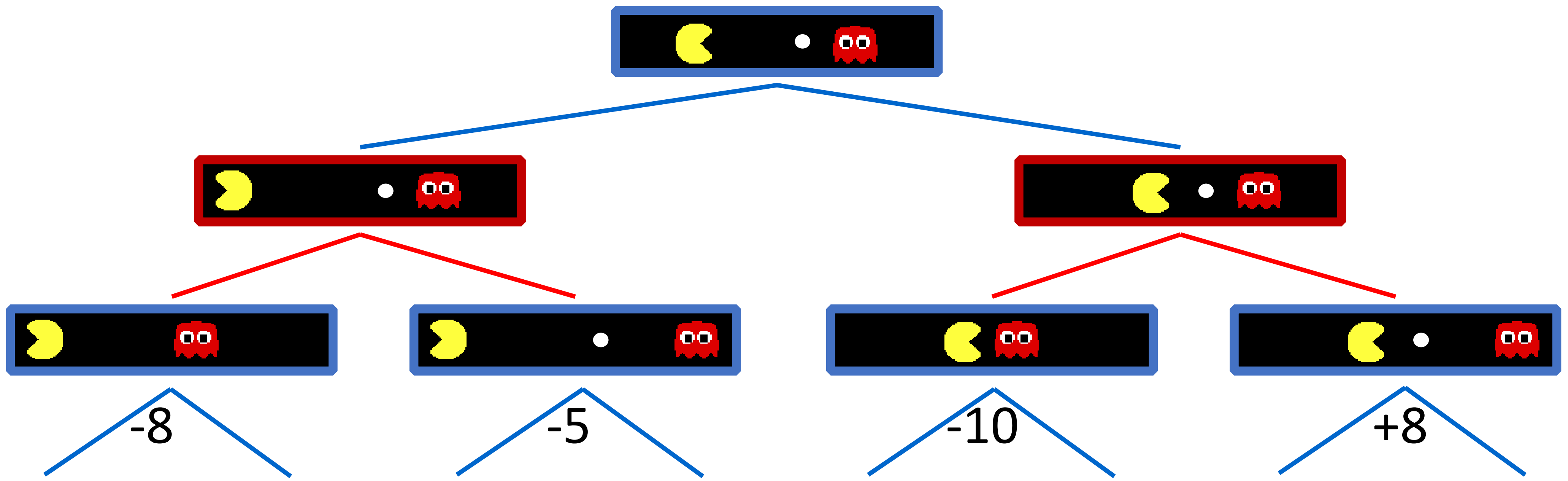
What is the size of the action space?

What is the size of the search tree?

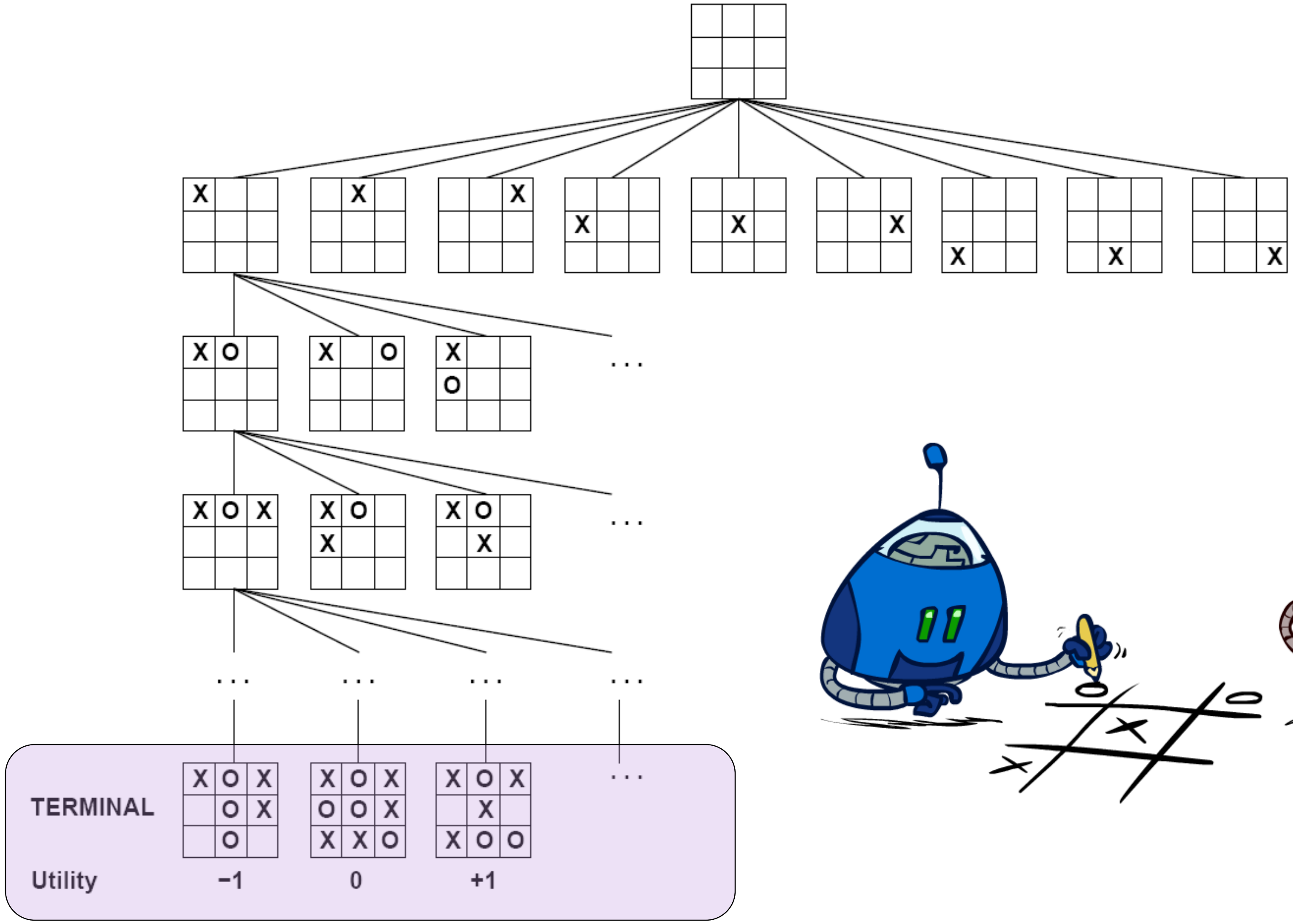


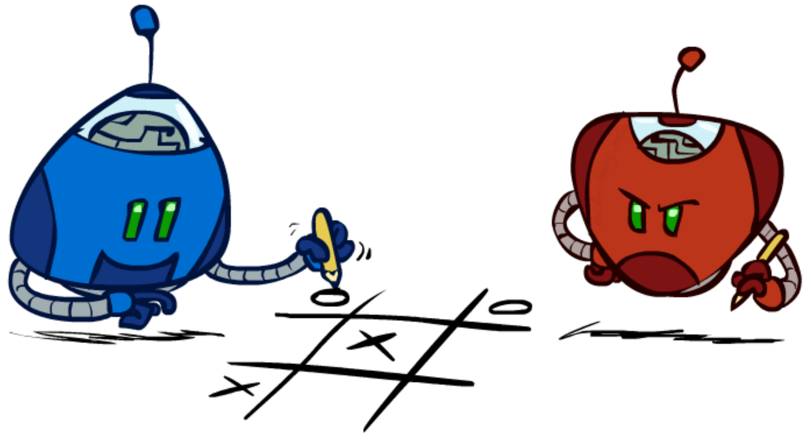
Minimax

States
Actions
Values



Tic-Tac-Toe Game Tree

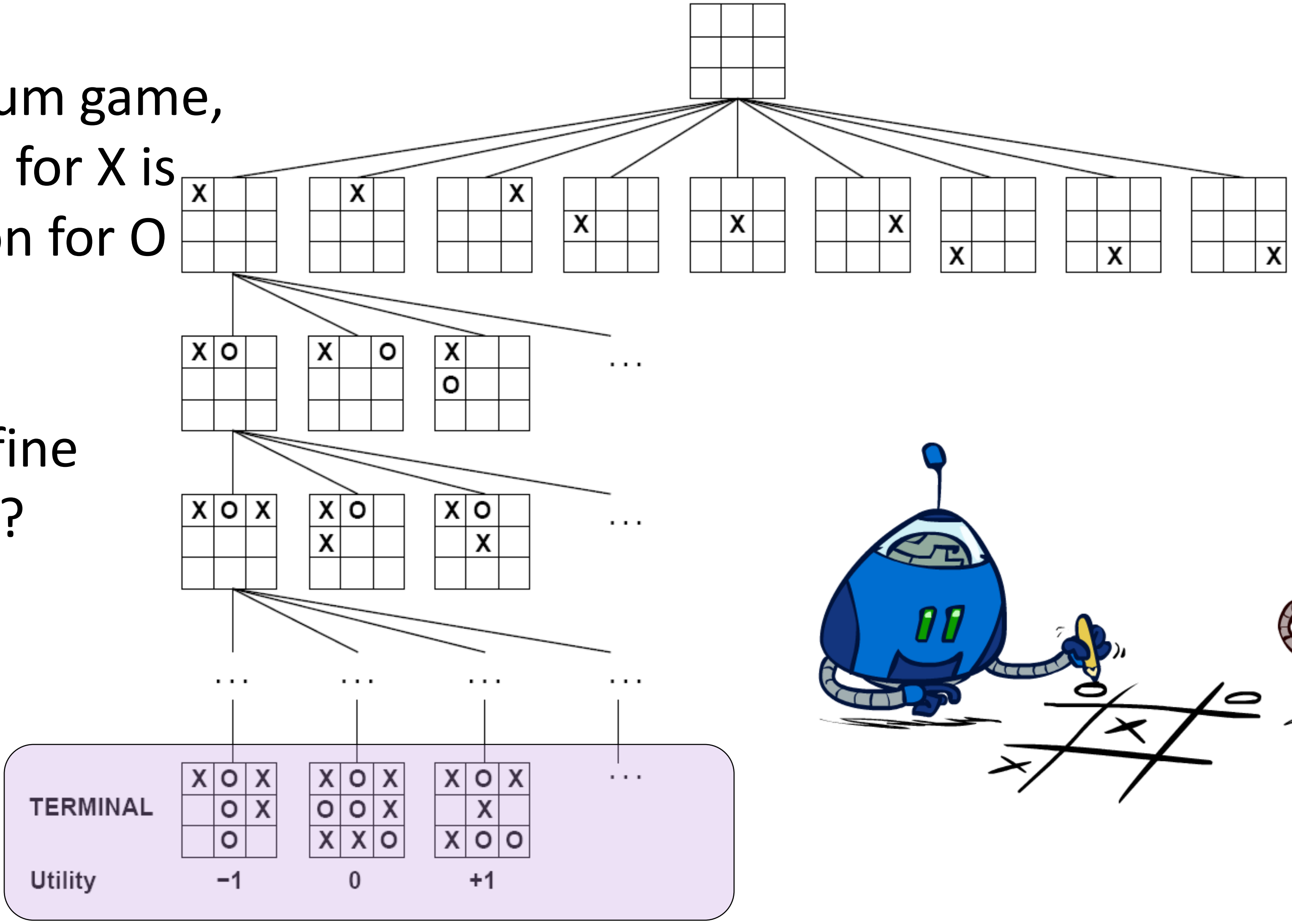


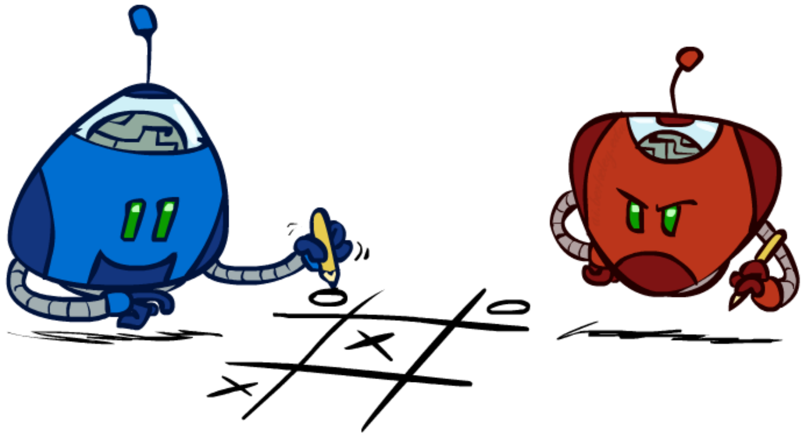


Tic-Tac-Toe Game Tree

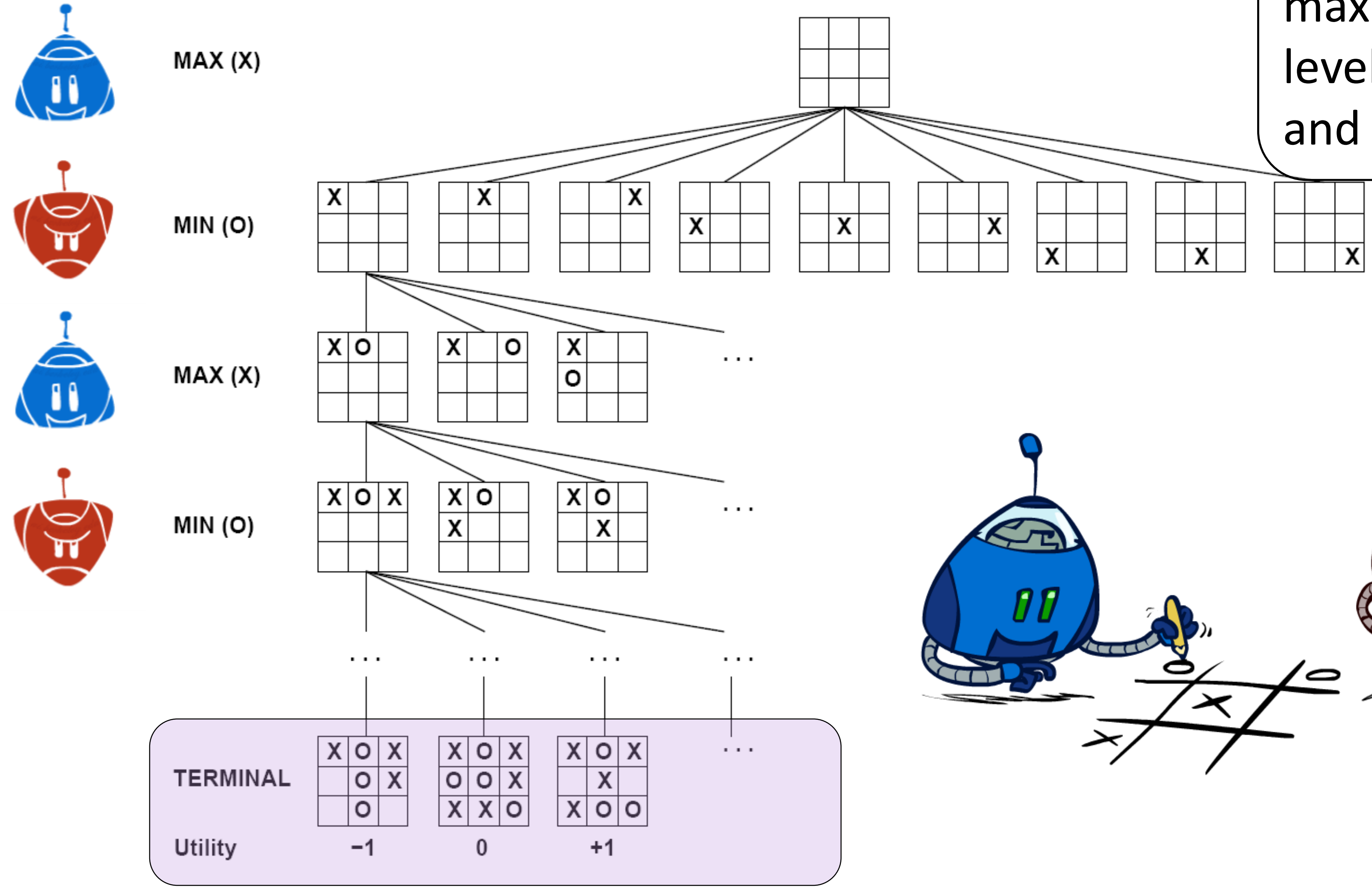
This is a zero-sum game, the best action for X is the worst action for O and vice versa

How do we define best and worst?

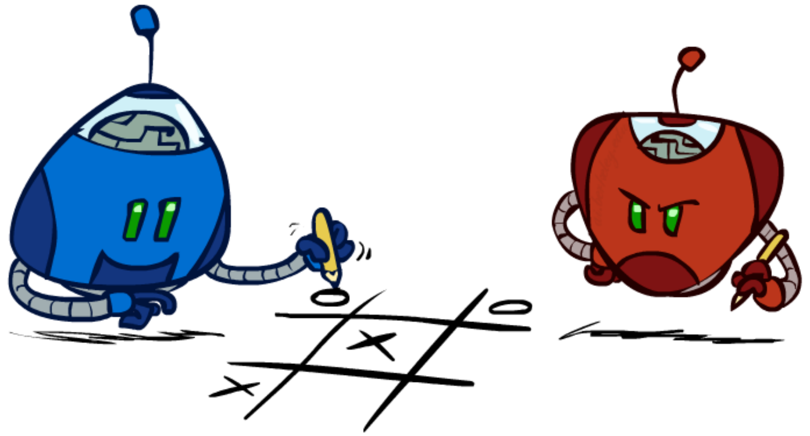




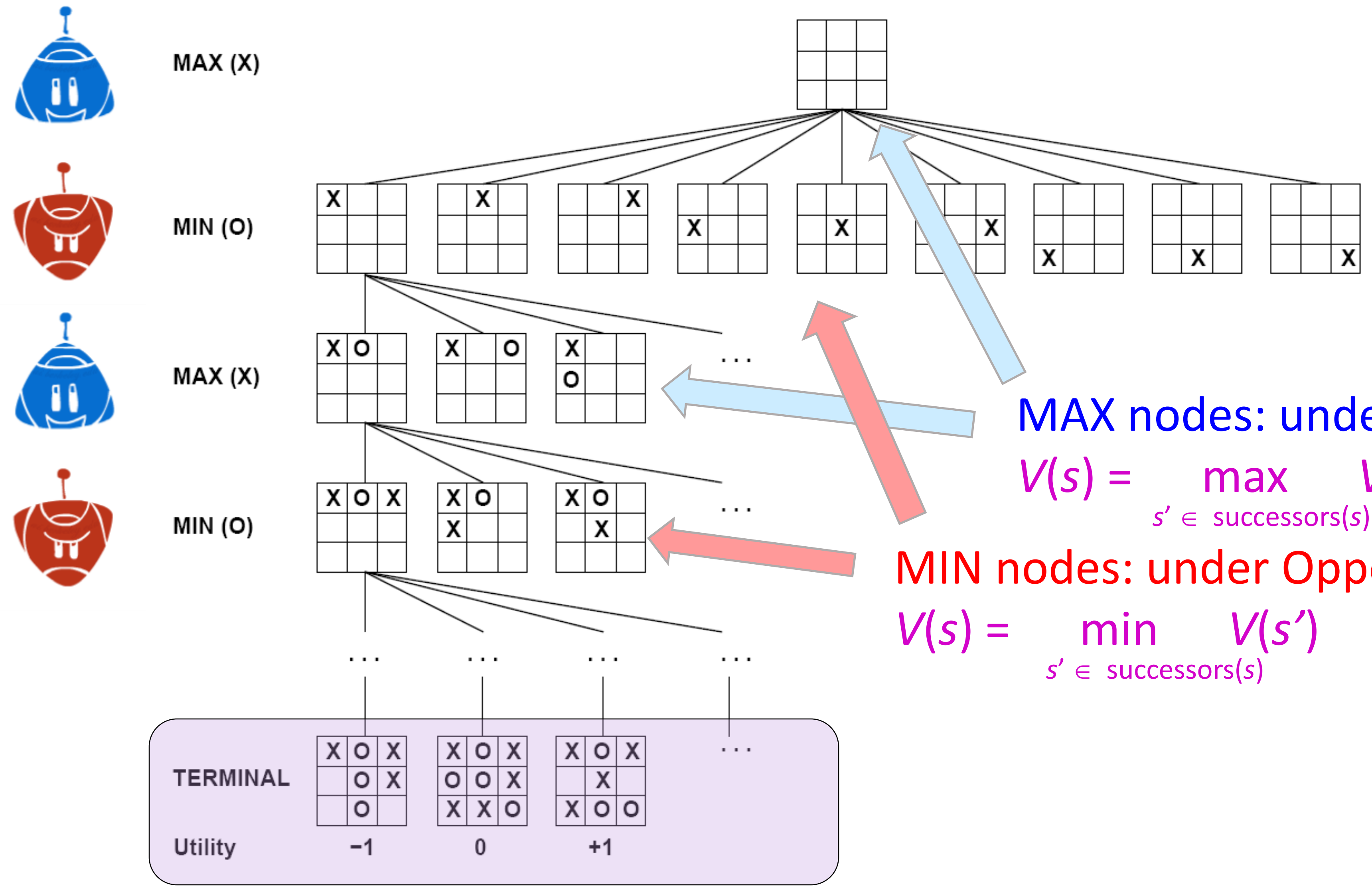
Tic-Tac-Toe Game Tree



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



Tic-Tac-Toe Minimax



MAX nodes: under Agent's control

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

MIN nodes: under Opponent's control

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

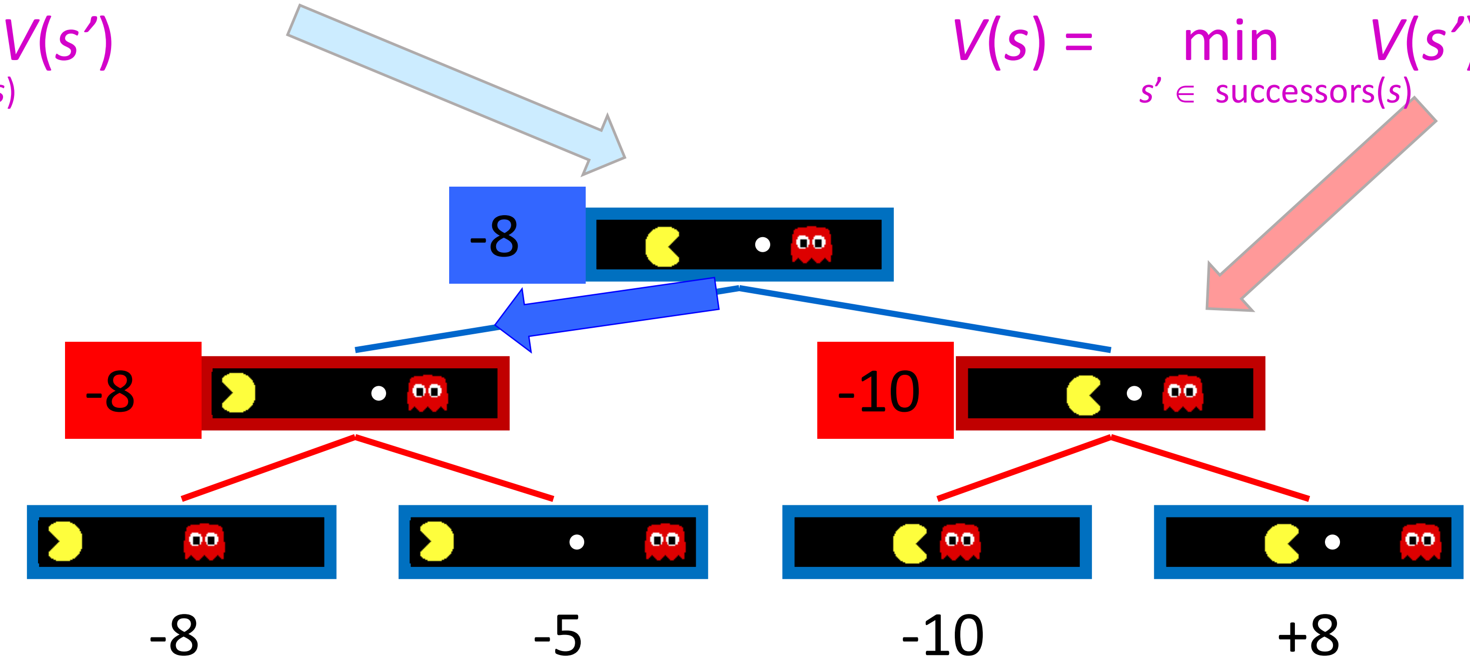
Small Pacman Example

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

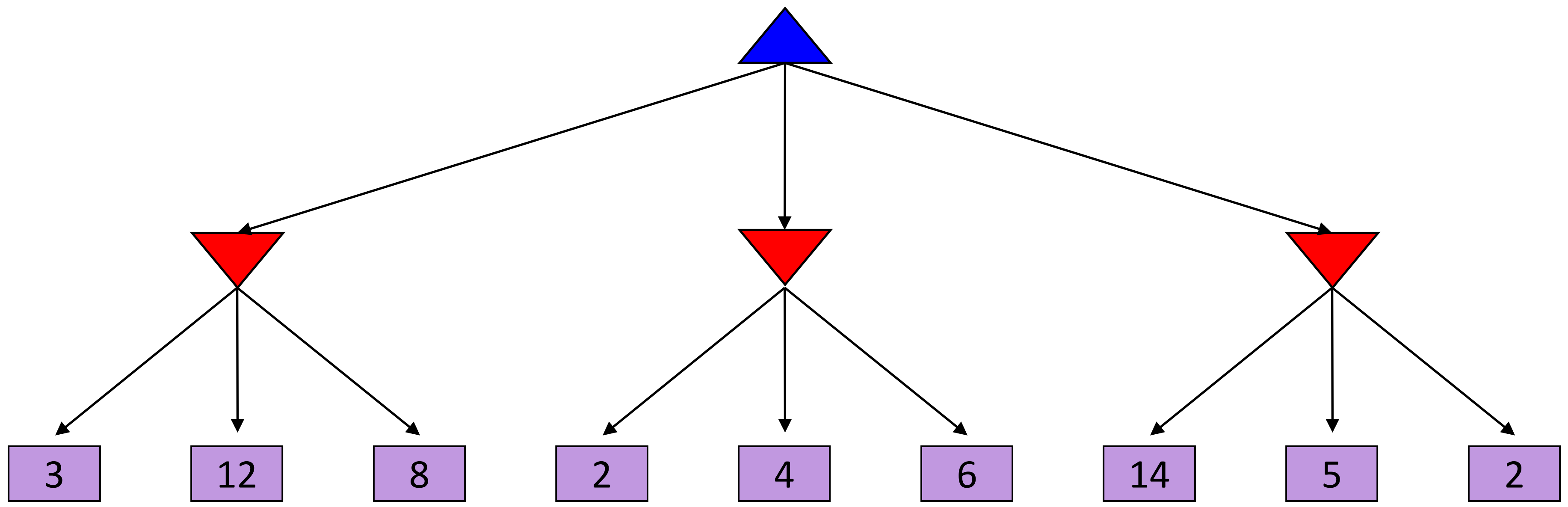
Minimax Code

```
def max_value(state):

    if state.is_leaf:
        return state.value
    # TODO Also handle depth limit

    best_value = -10000000

    for action in state.actions:
        next_state = state.result(action)

        next_value = min_value(next_state)

        if next_value > best_value:
            best_value = next_value

    return best_value

def min_value(state):
```

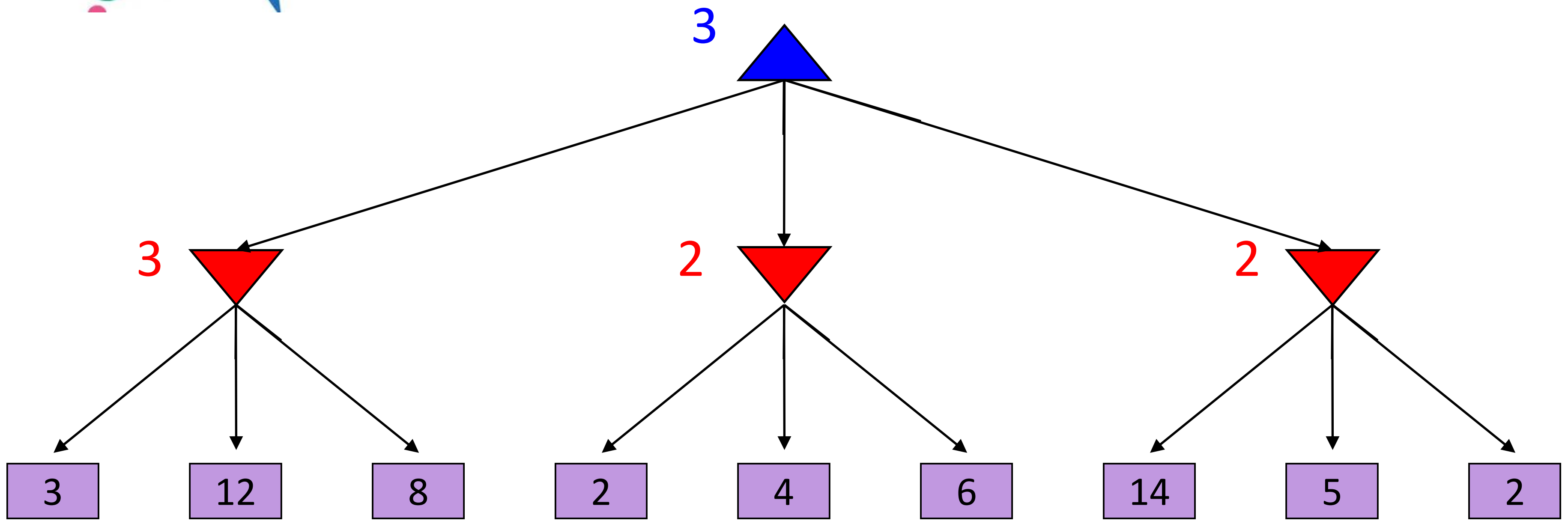
1

What is the minimax value at the root

1. 2
2. 3
3. 6
4. 12
5. 14



1-answer



2

What kind of search is Minimax Search?

A) BFS
B) DFS
C) UCS
D) A*

2-answer

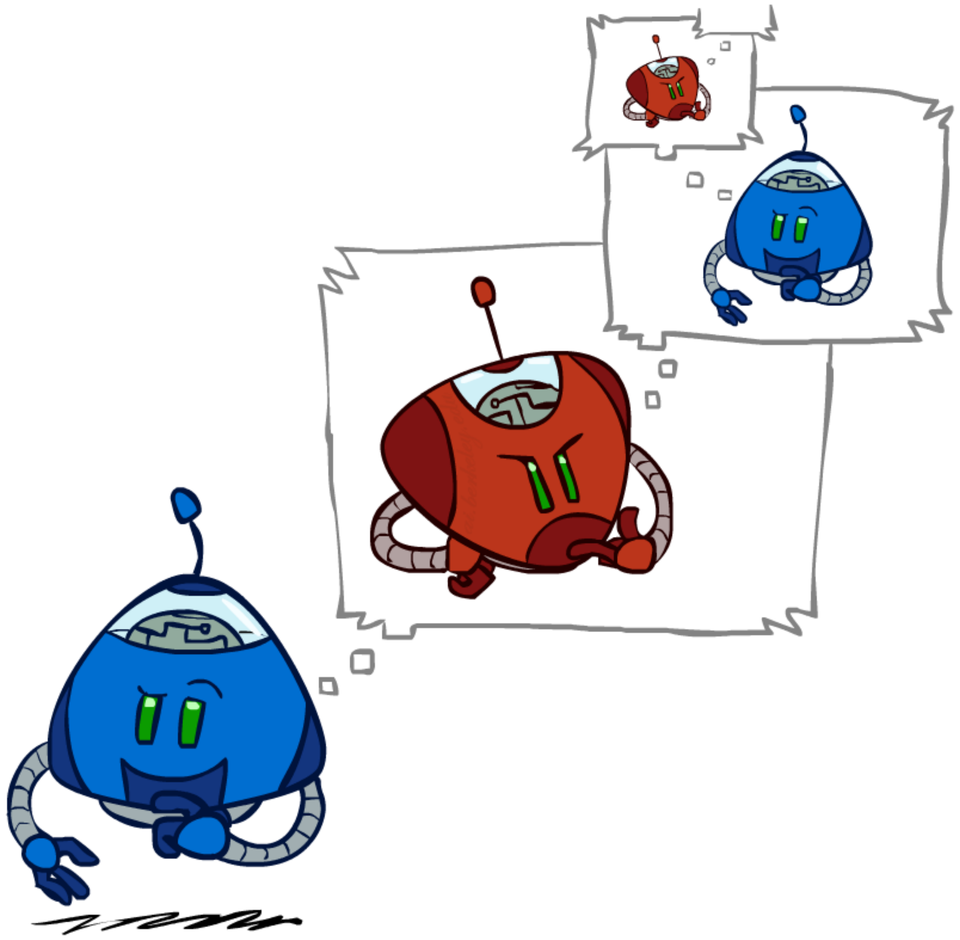
What kind of search is Minimax Search?

A) BFS
B) DFS
C) UCS
D) A*

Minimax Efficiency

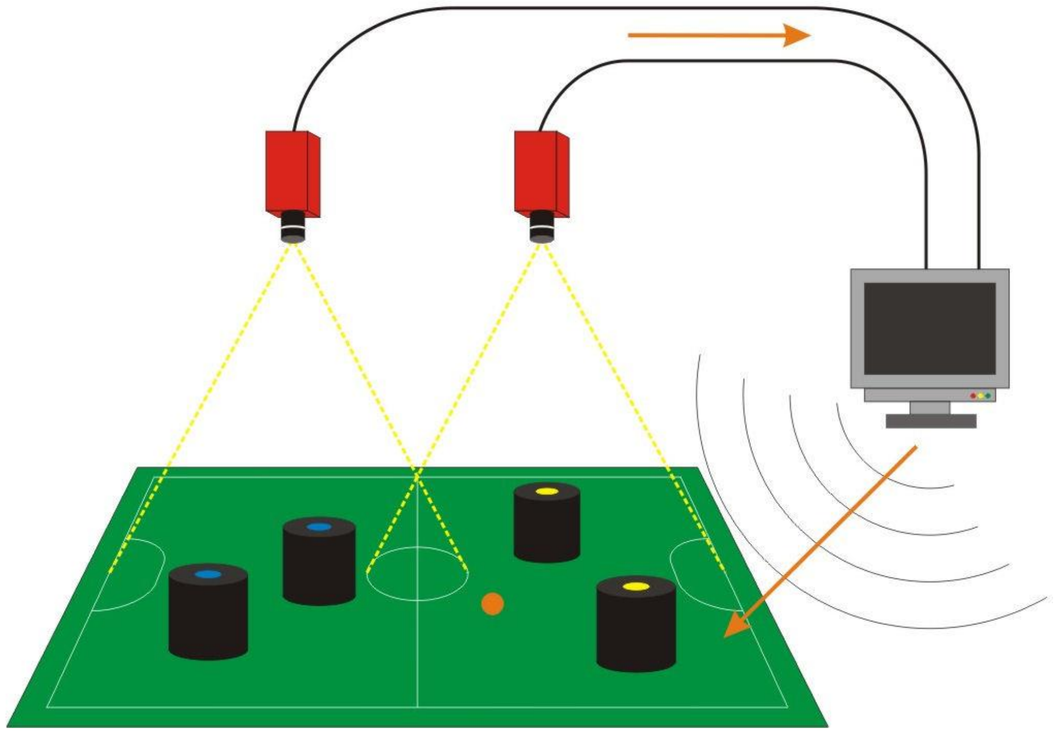
- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - Humans can't do this either, so how do we play chess?



Small Size Robot Soccer

- Joint State/Action space and search for our team
- Adversarial search to predict the opponent team

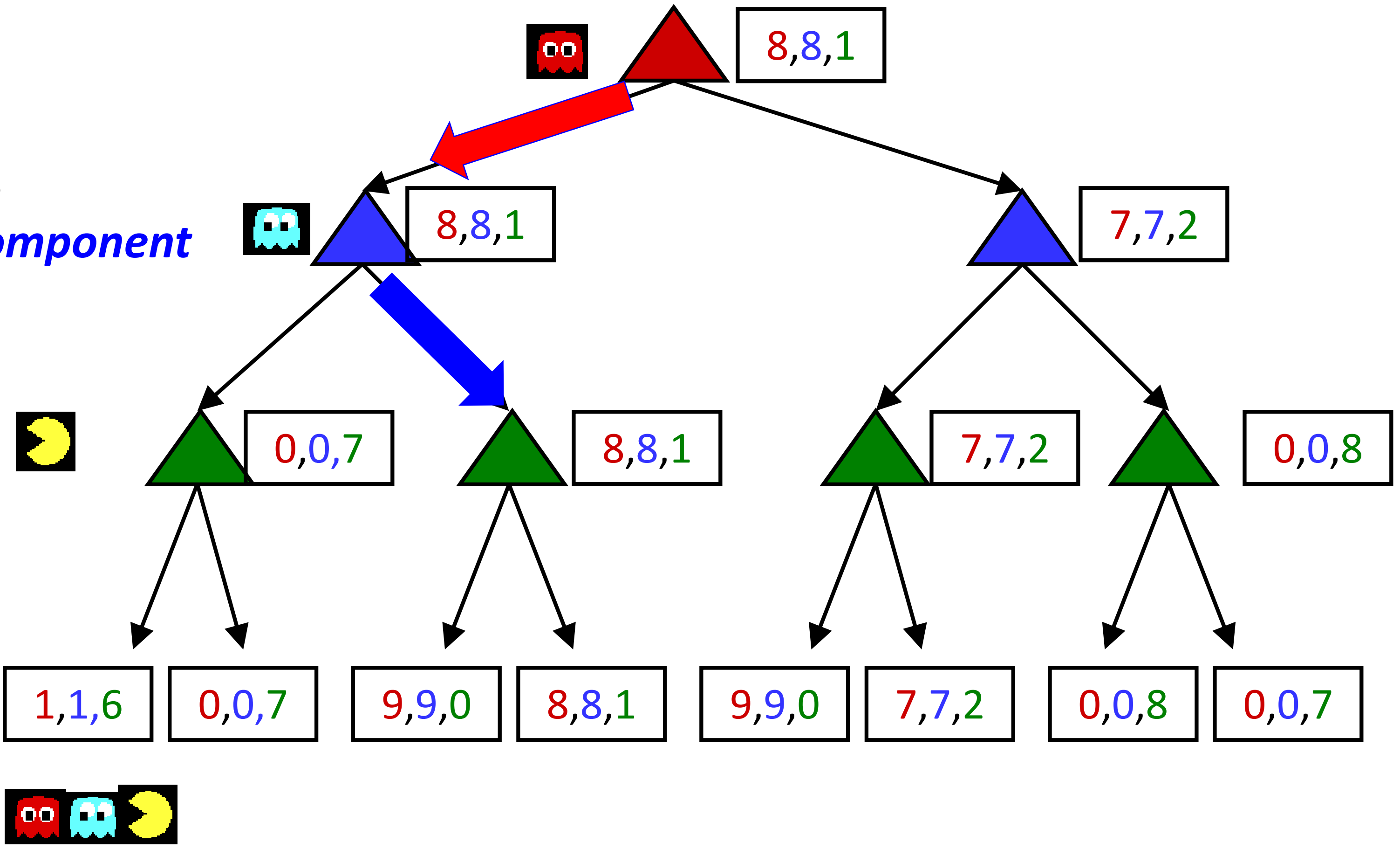


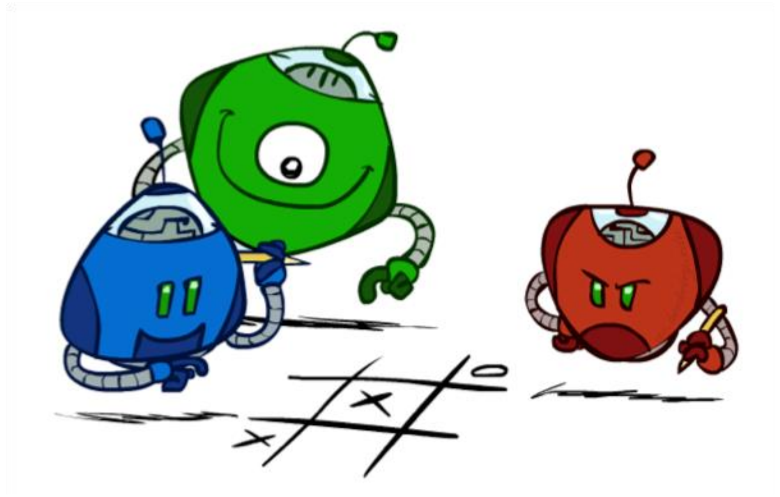
https://www.youtube.com/watch?v=YihJguq26ek

Generalized minimax

- What if the game is not zero-sum, or has multiple players?

- Generalization of minimax:
  - Terminals have ***utility tuples***
  - Node values are also utility tuples
  - ***Each player maximizes its own component***
  - Can give rise to cooperation and competition dynamically…





Resource Limits

Resource Limits

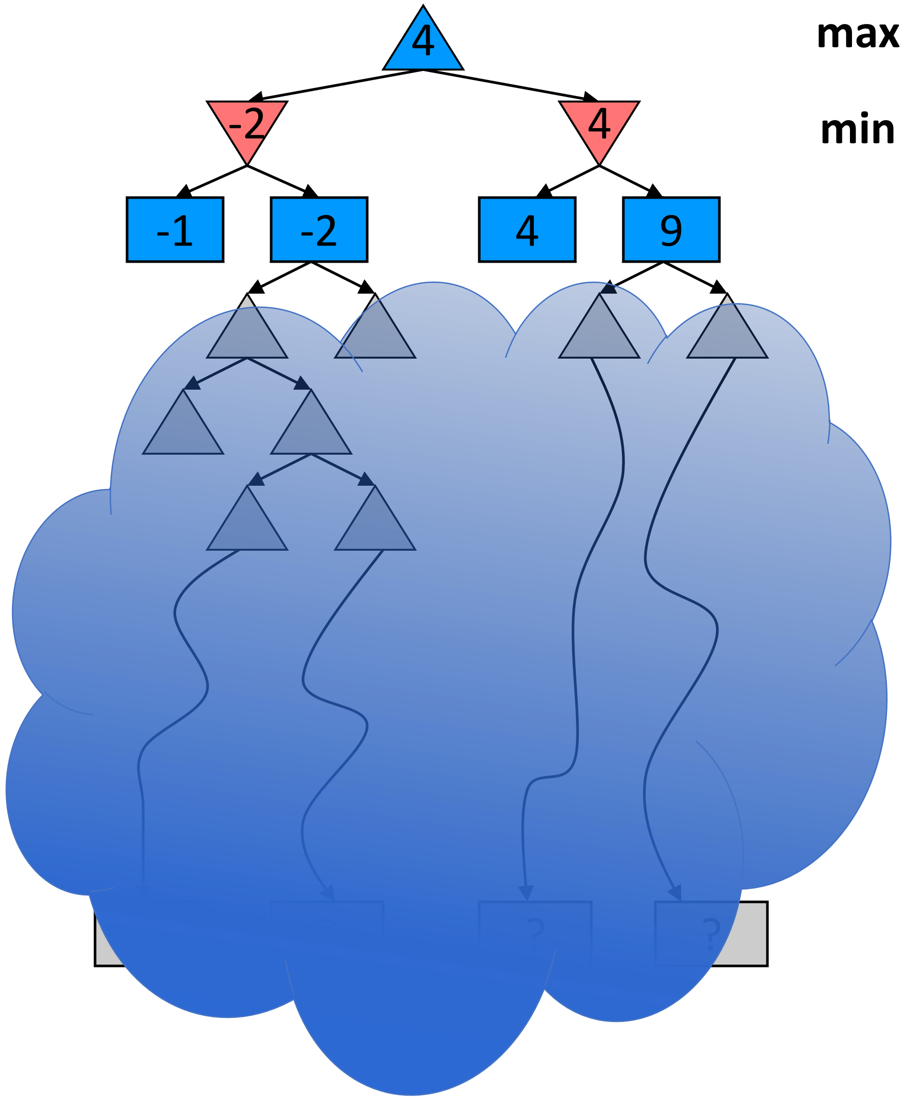
- Problem: In realistic games, cannot search to leaves!

- Solution 1: Bounded lookahead
  - Search only to a preset ***depth limit*** or ***horizon***
  - Use an ***evaluation function*** for non-terminal positions

- Guarantee of optimal play is gone

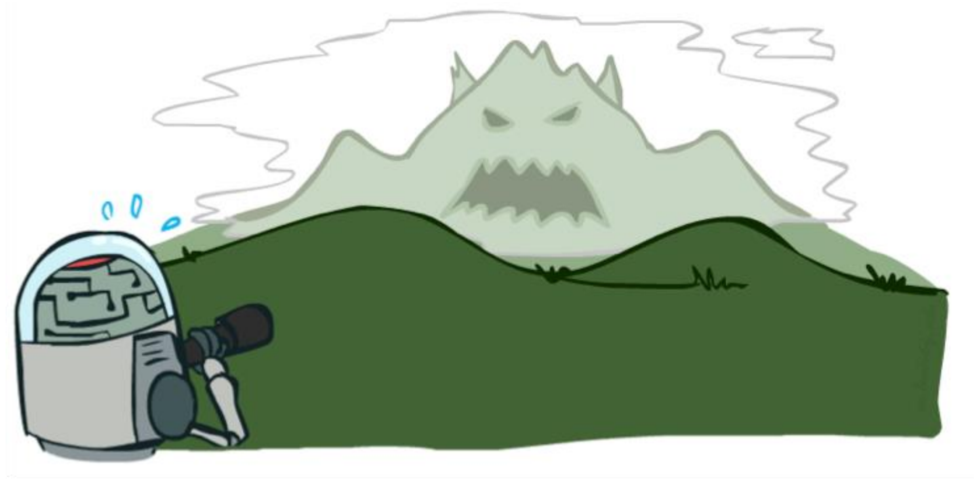
- More plies make a BIG difference

- 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

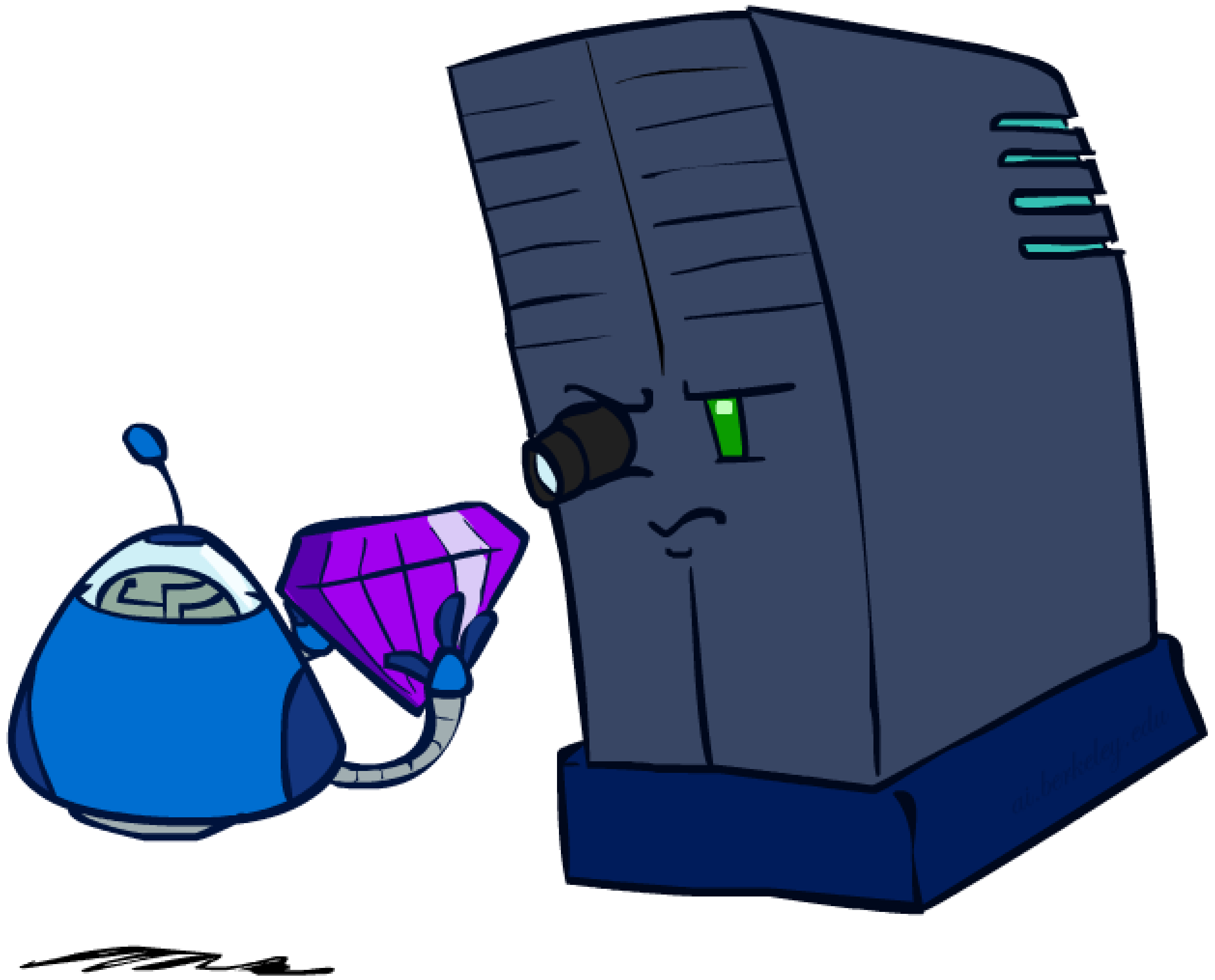


Depth Matters

- Evaluation functions are always imperfect
- Deeper search => better play (usually)
- Or, 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

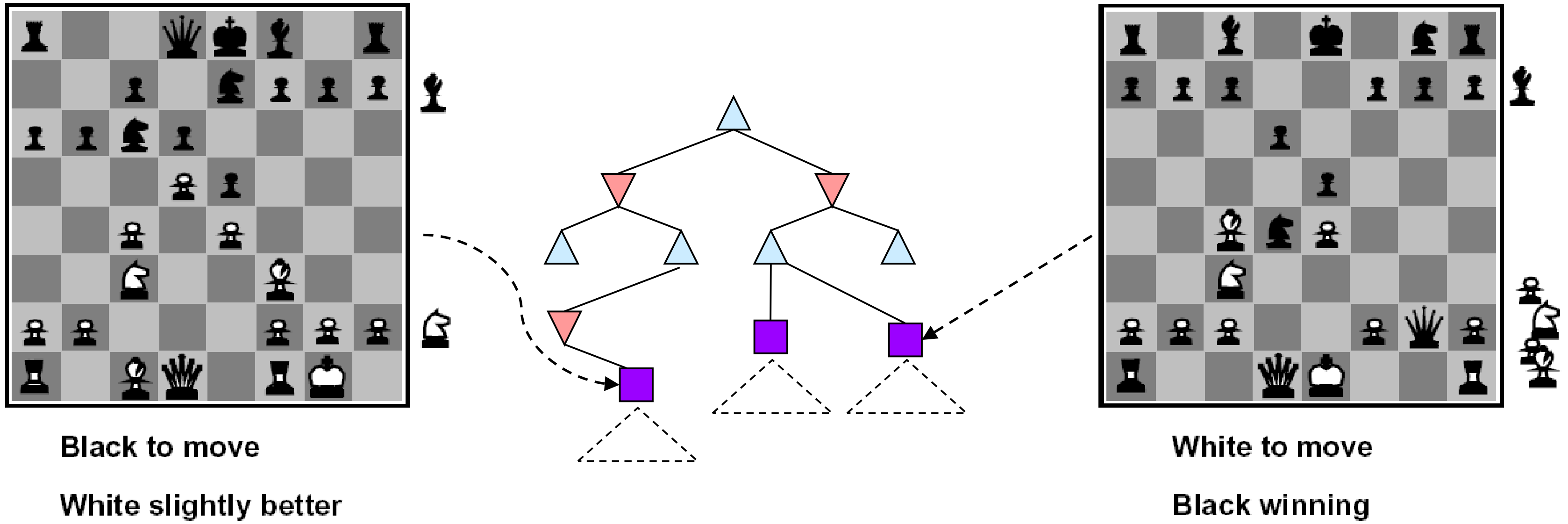


Evaluation Functions



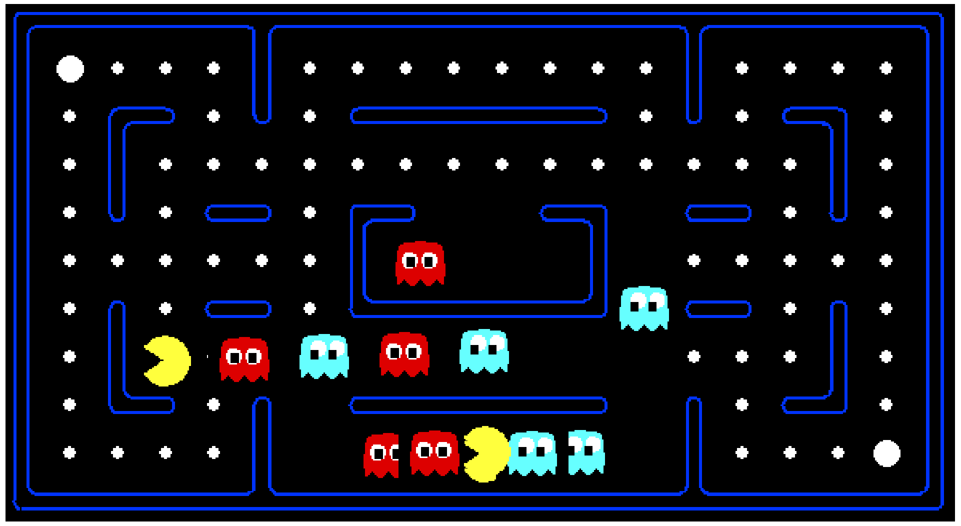
Evaluation Functions

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

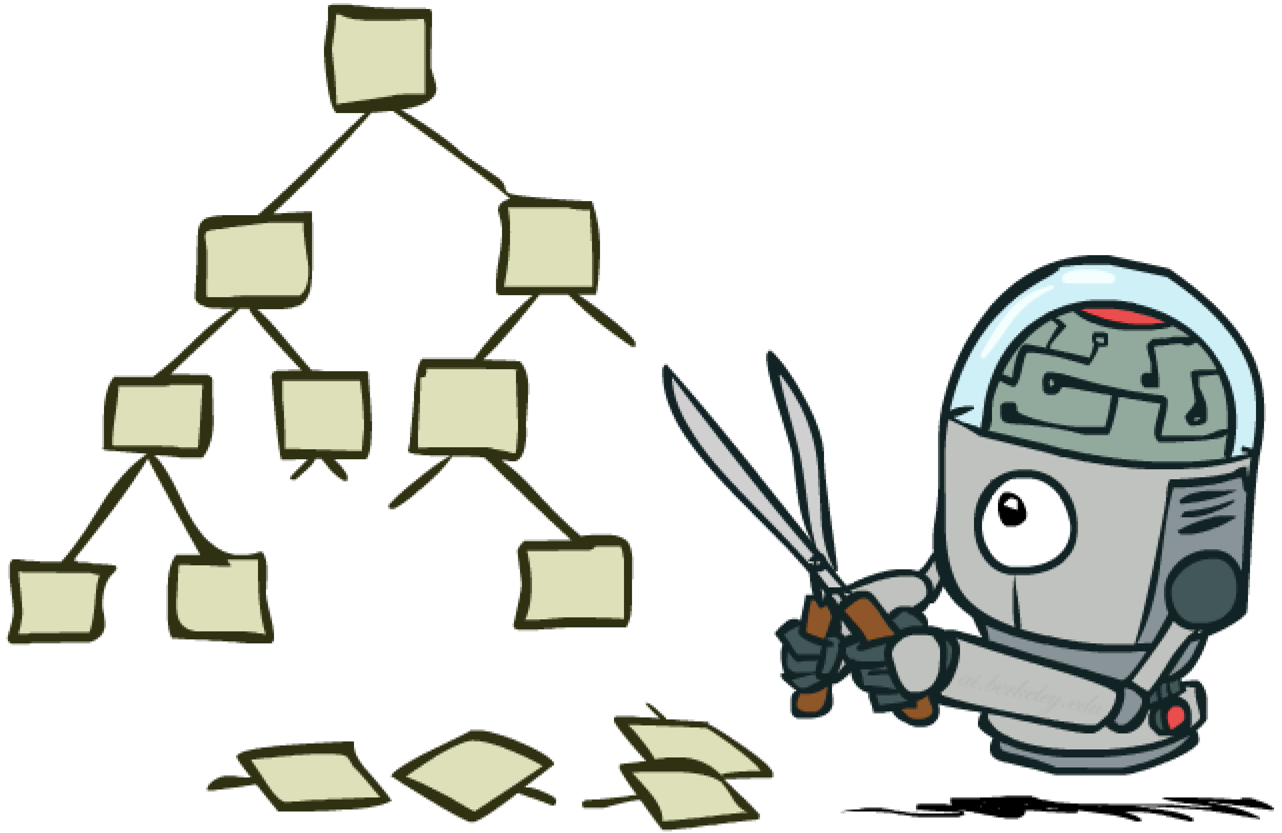


- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:
  - EVAL(s) = $w_1 f_1(s) + w_2 f_2(s) + ... + w_n f_n(s)$
  - E.g., $w_1 = 9$, $f_1(s)$ = (num white queens – num black queens), etc.
- Terminate search only in ***quiescent***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

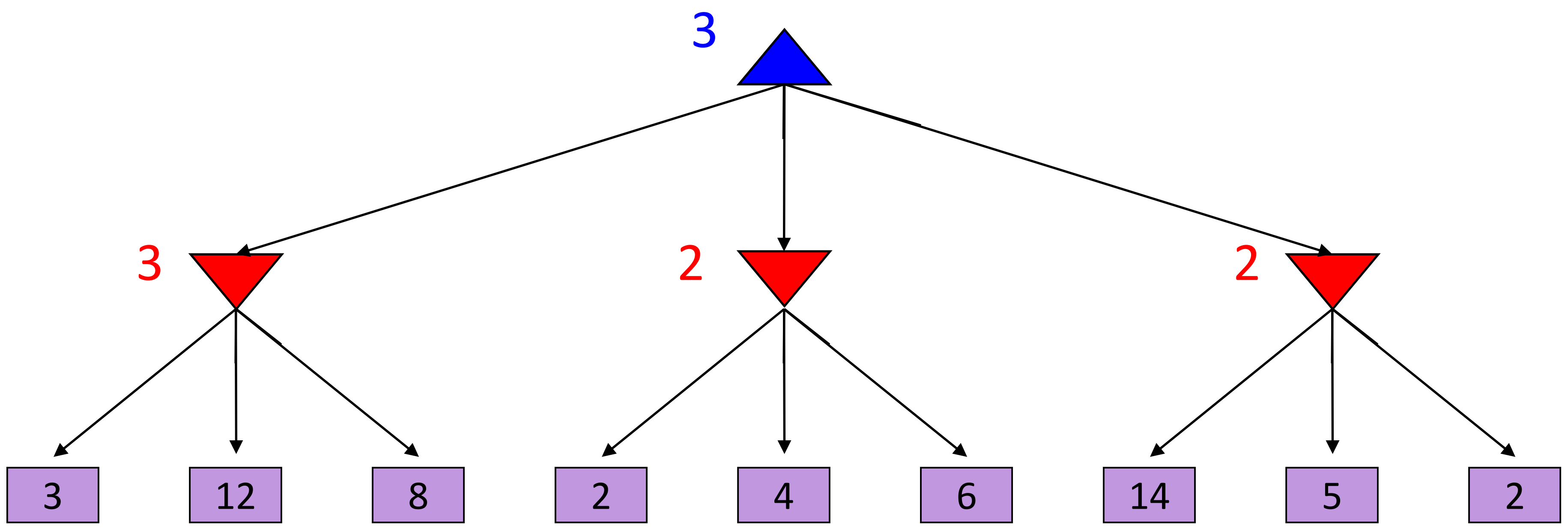
Evaluation for Pacman



Solution 2: Game Tree Pruning

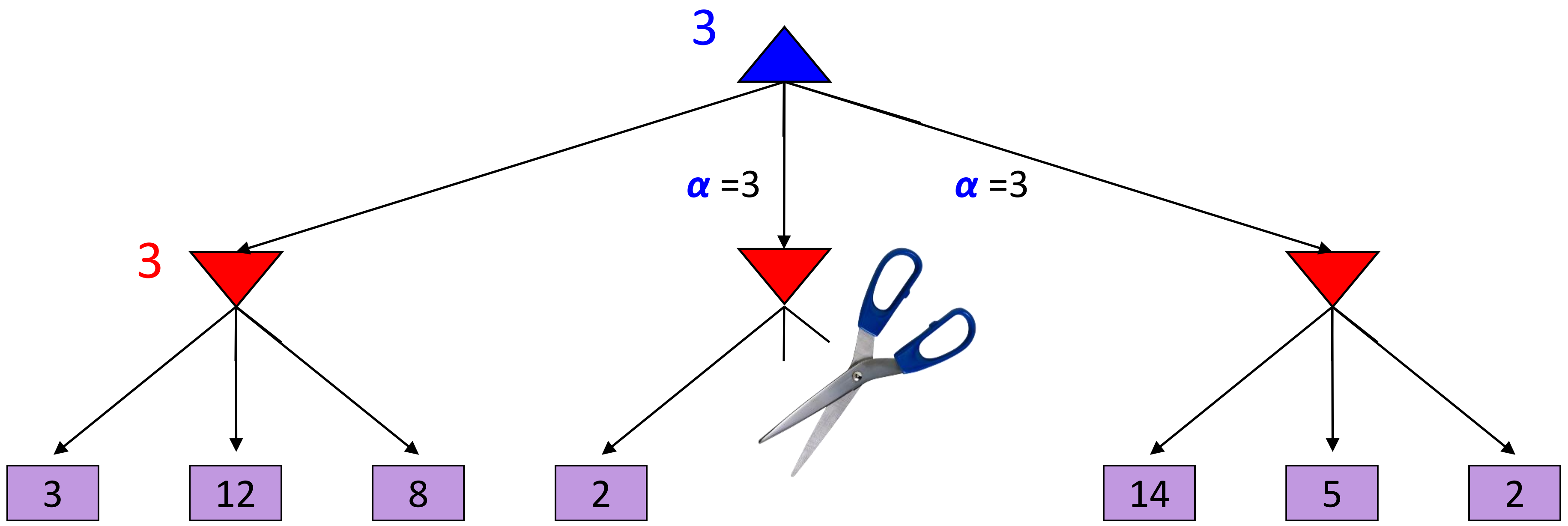


Intuition: prune the branches that can't be chosen



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

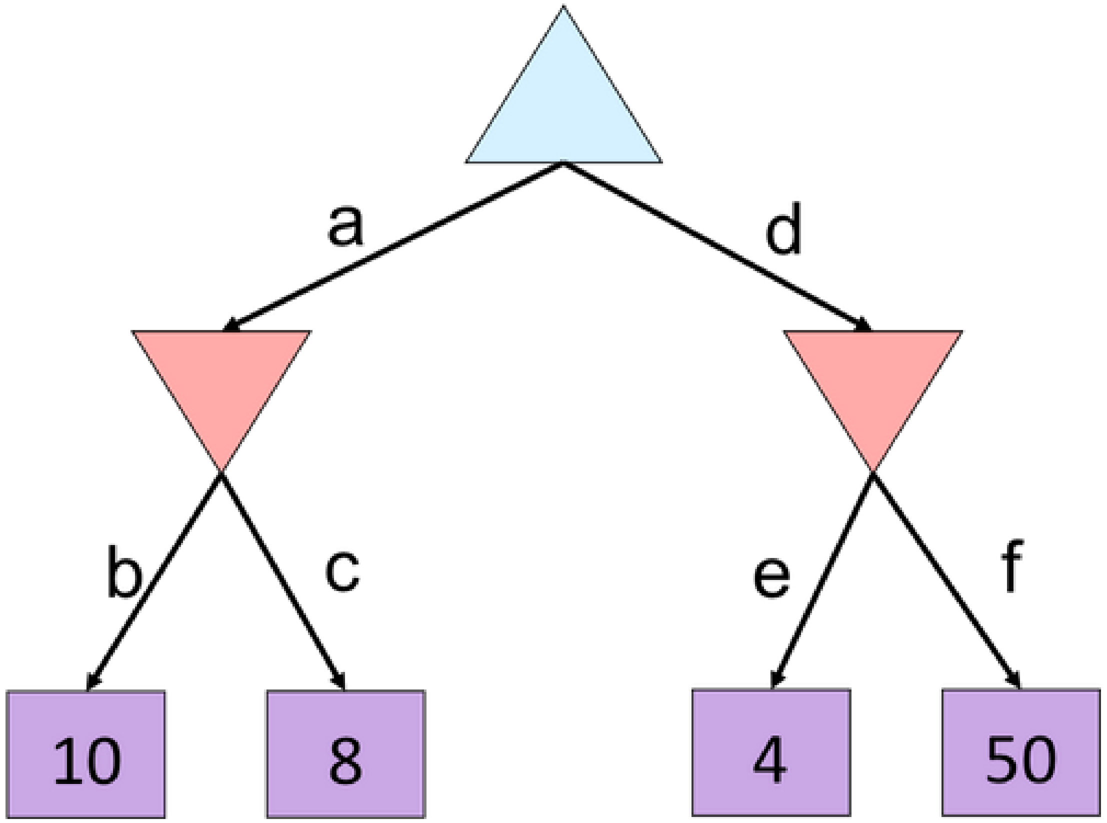
The order of generation matters: more pruning is possible if good moves come first

Alpha-Beta Implementation

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

```
def max-value(state, $\alpha$, $\beta$):
    initialize v = $-\infty$
    for each successor of state:
        v = max(v, value(successor, $\alpha$, $\beta$))
        if $v \geq \beta$
            return v
        $\alpha$ = max($\alpha$, v)
    return v
```

```
def min-value(state , $\alpha$, $\beta$):
    initialize v = $+\infty$
    for each successor of state:
        v = min(v, value(successor, $\alpha$, $\beta$))
        if $v \leq \alpha$
            return v
        $\beta$ = min($\beta$, v)
    return v
```



What is the value of the blue triangle?

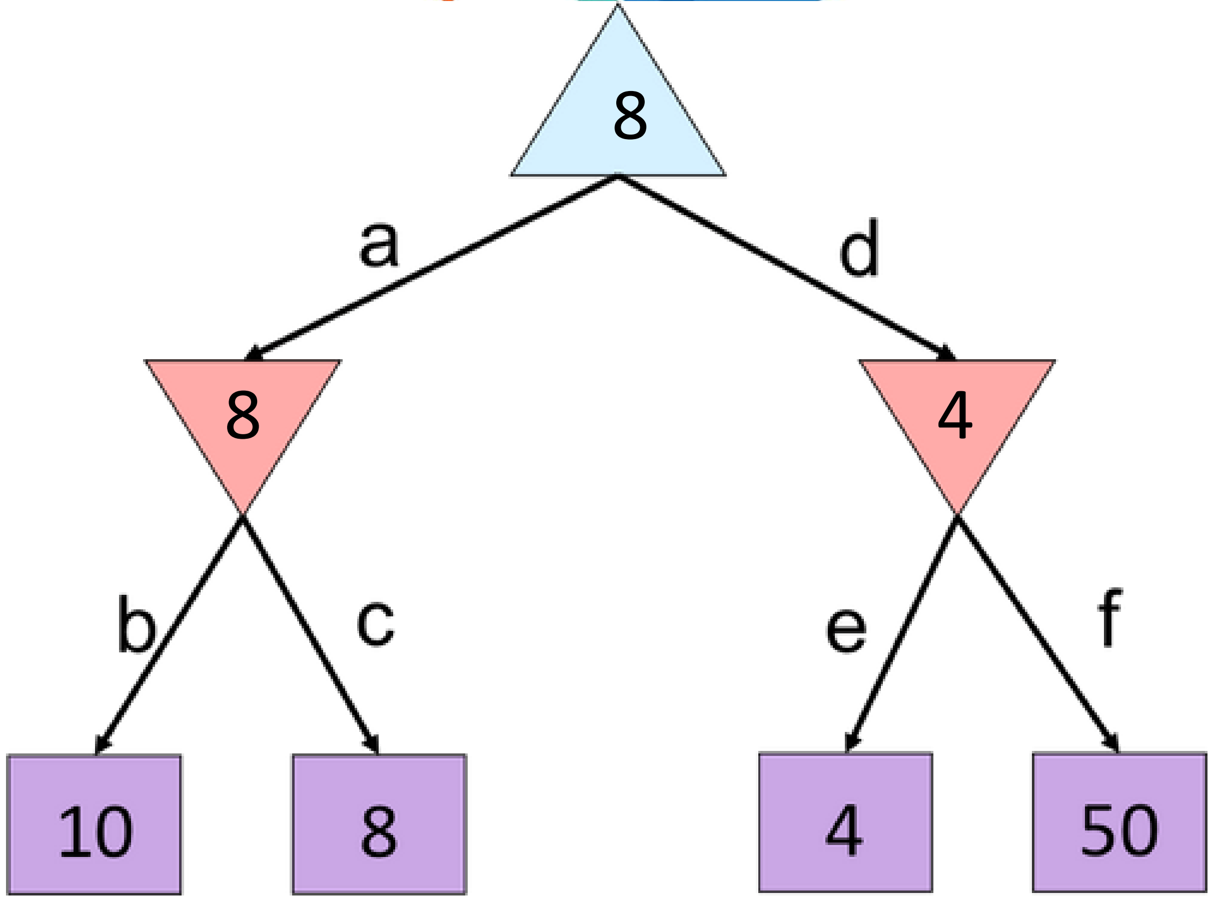
A) 10

B) 8

C) 4

D) 50

3-answer



What is the value of the blue triangle?

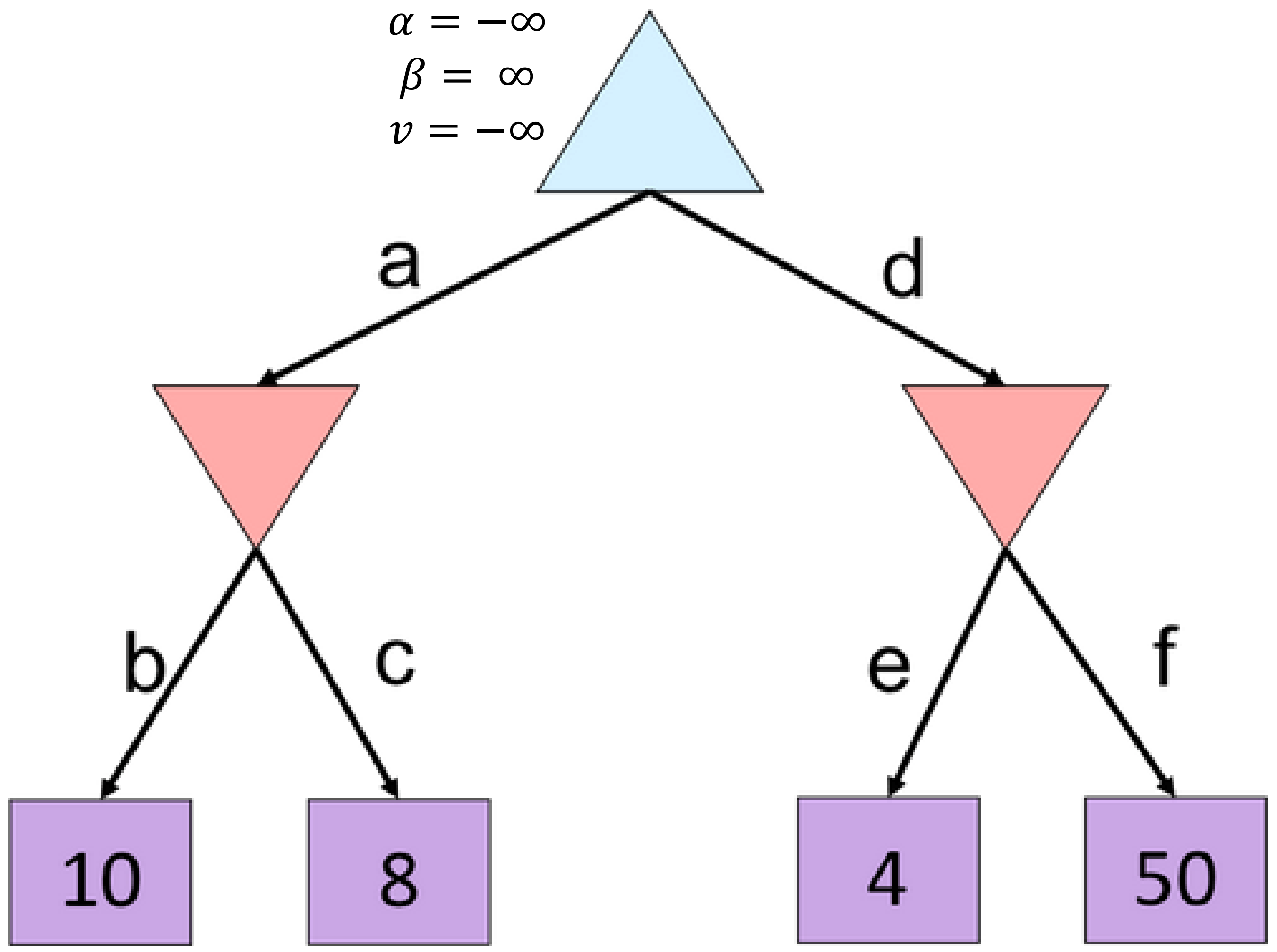
A) 10

B) 8

C) 4

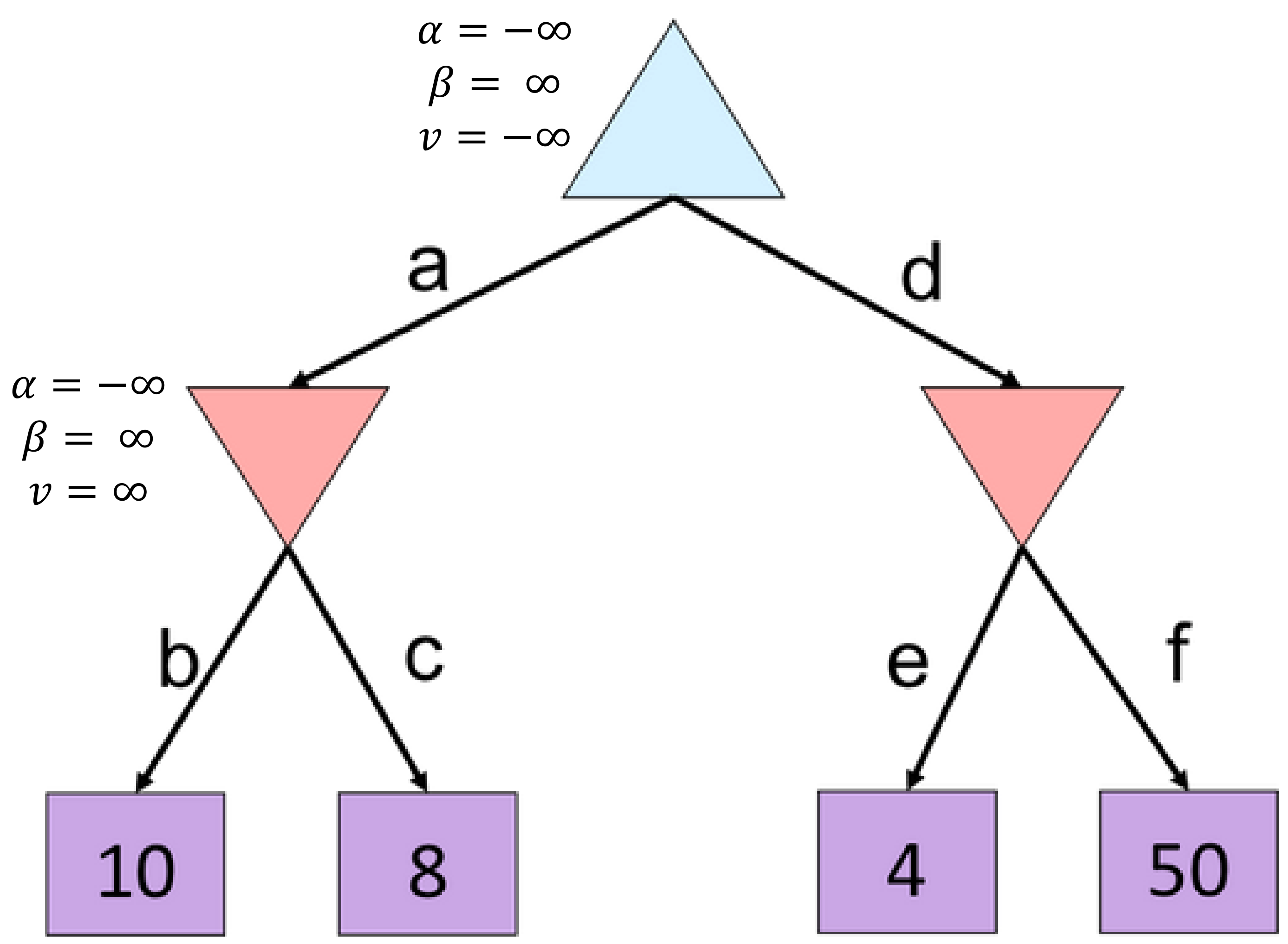
D) 50

Alpha-Beta Small Example



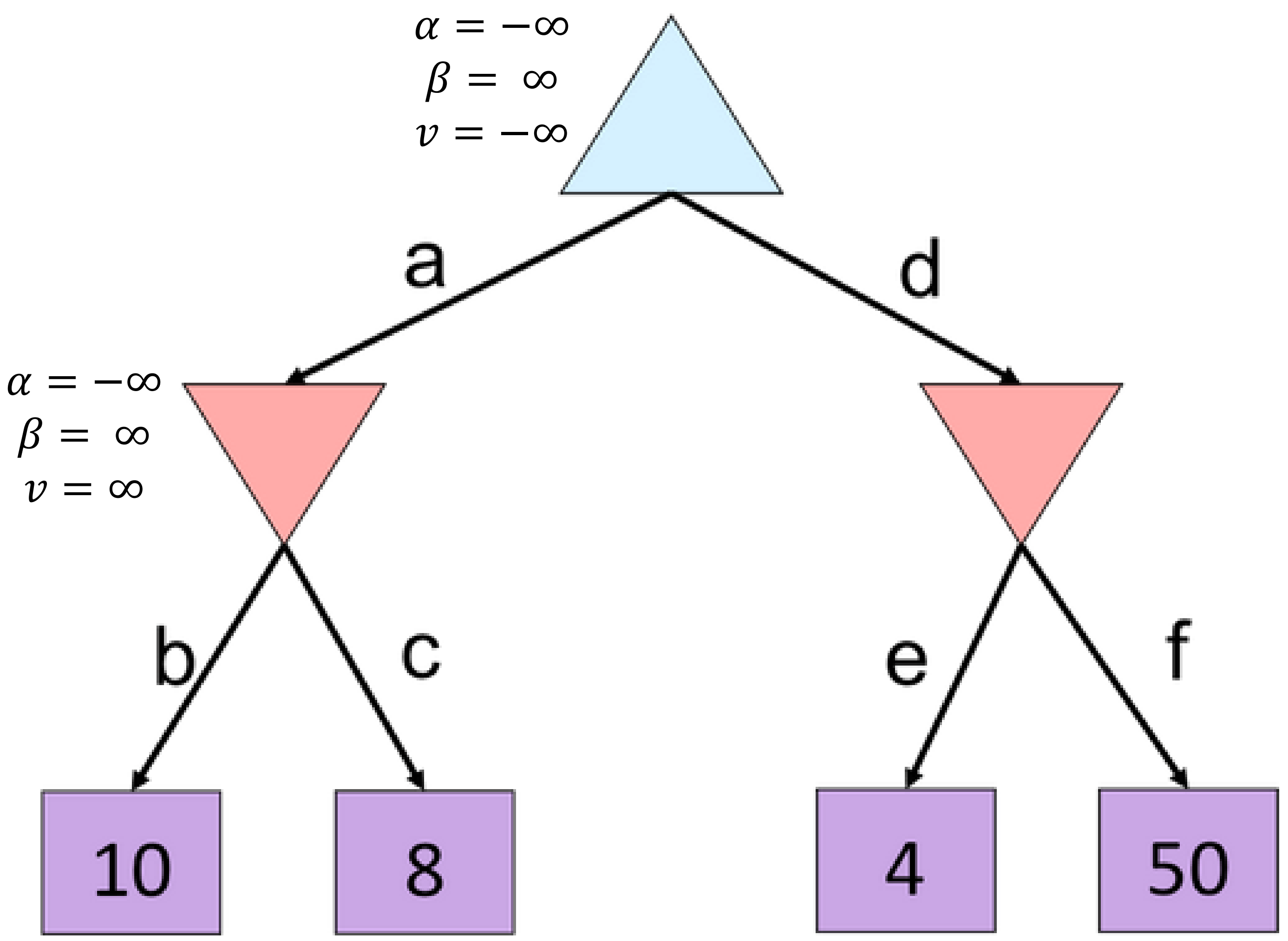
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



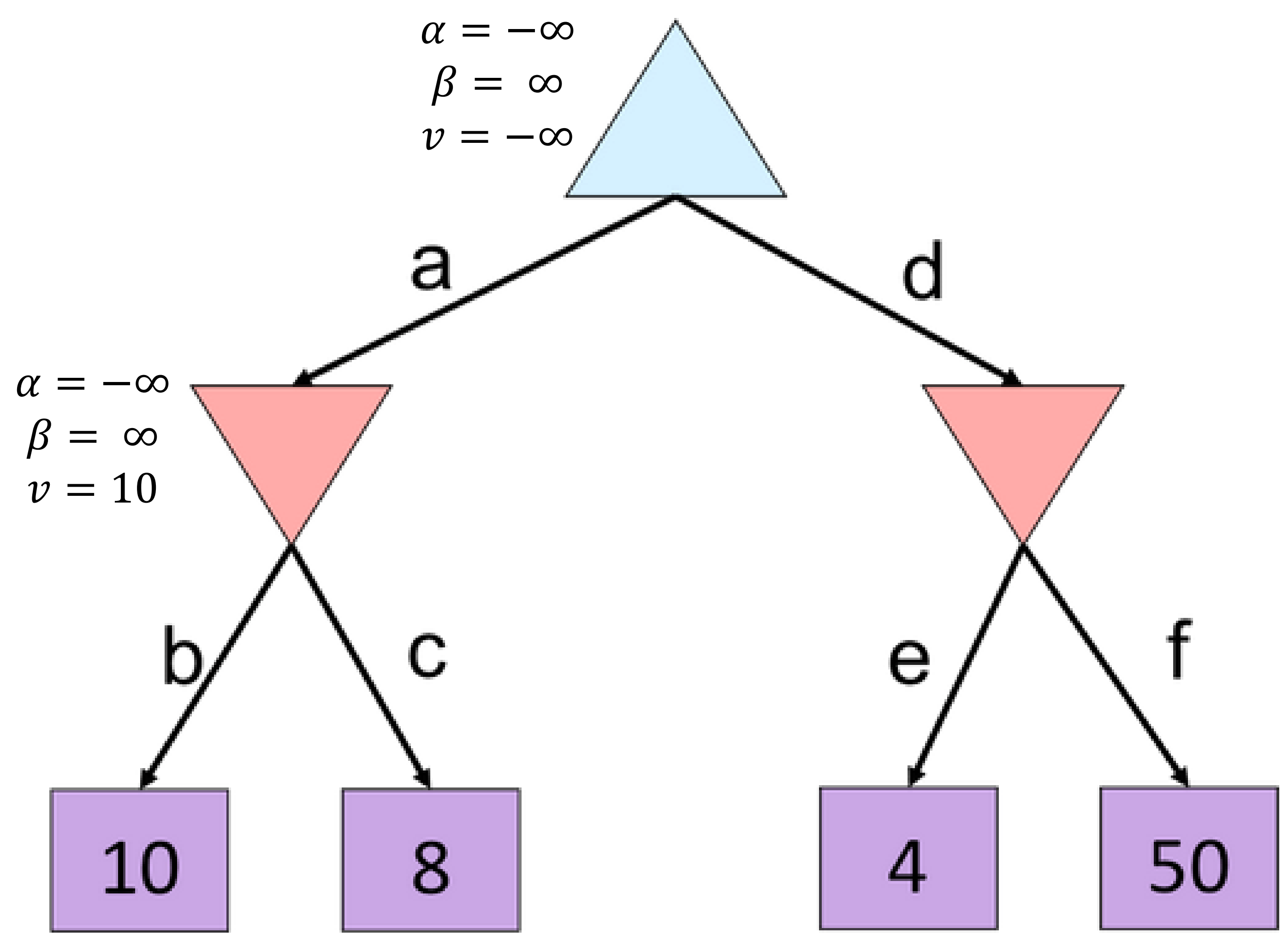
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



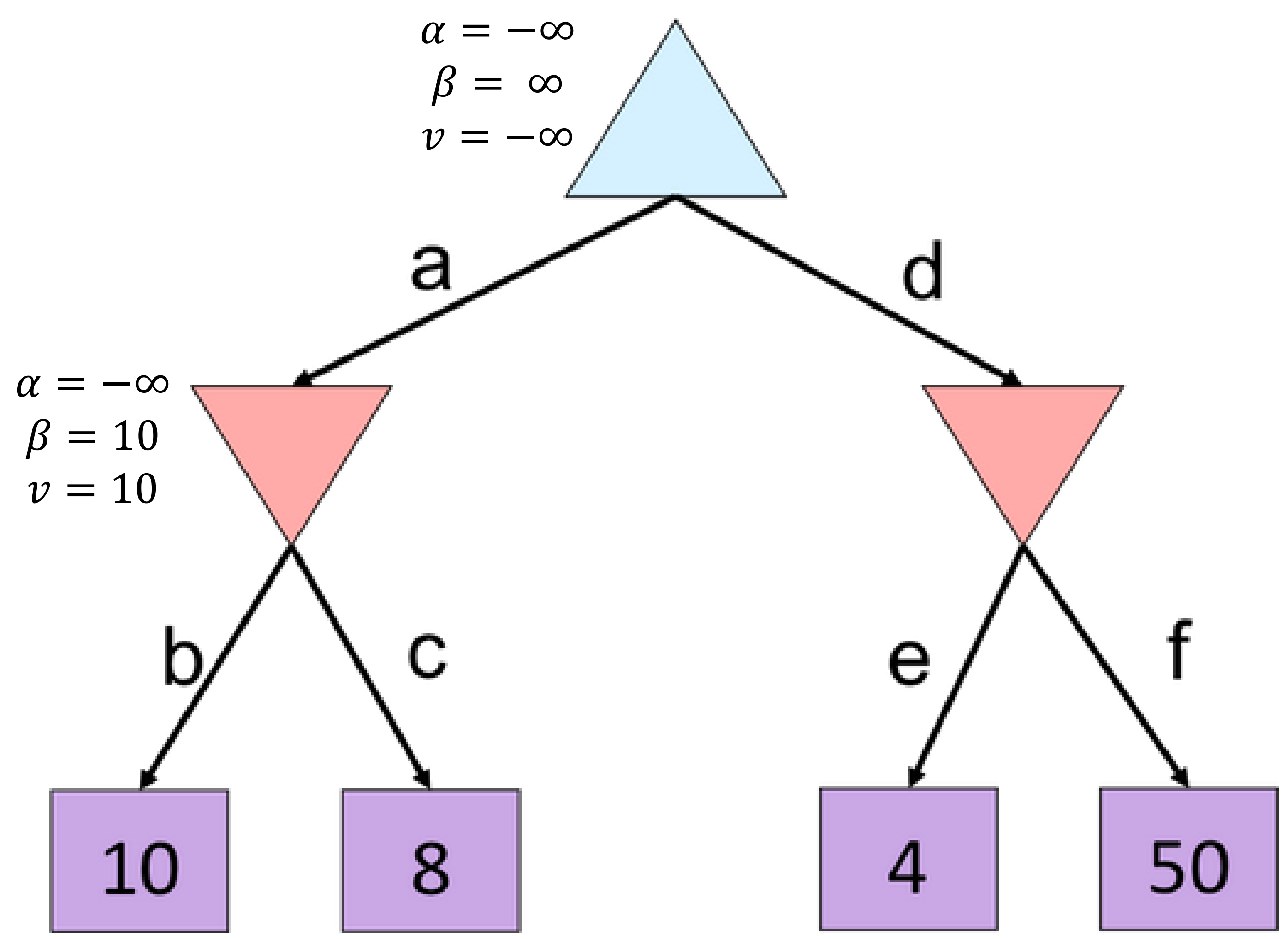
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



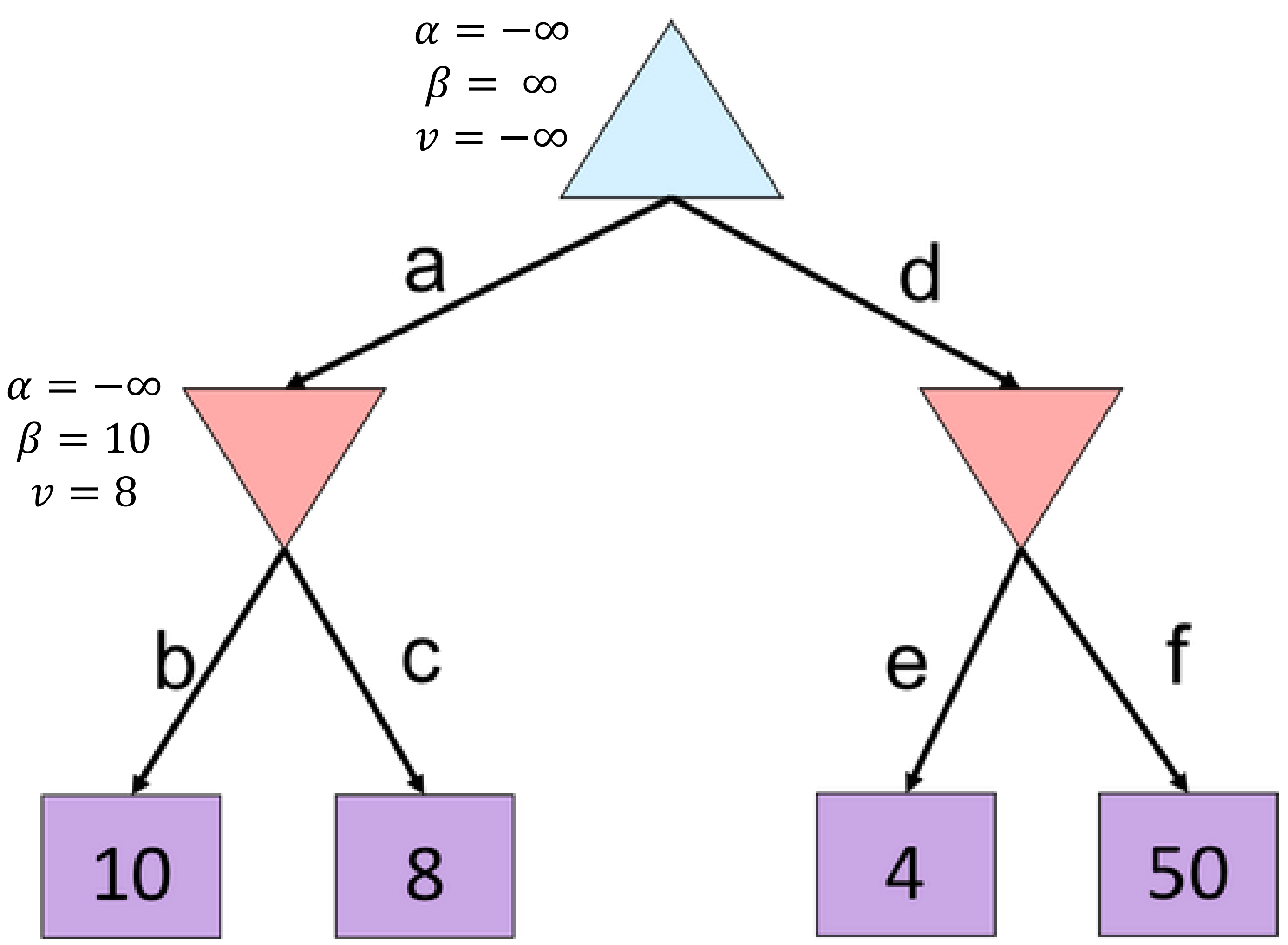
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



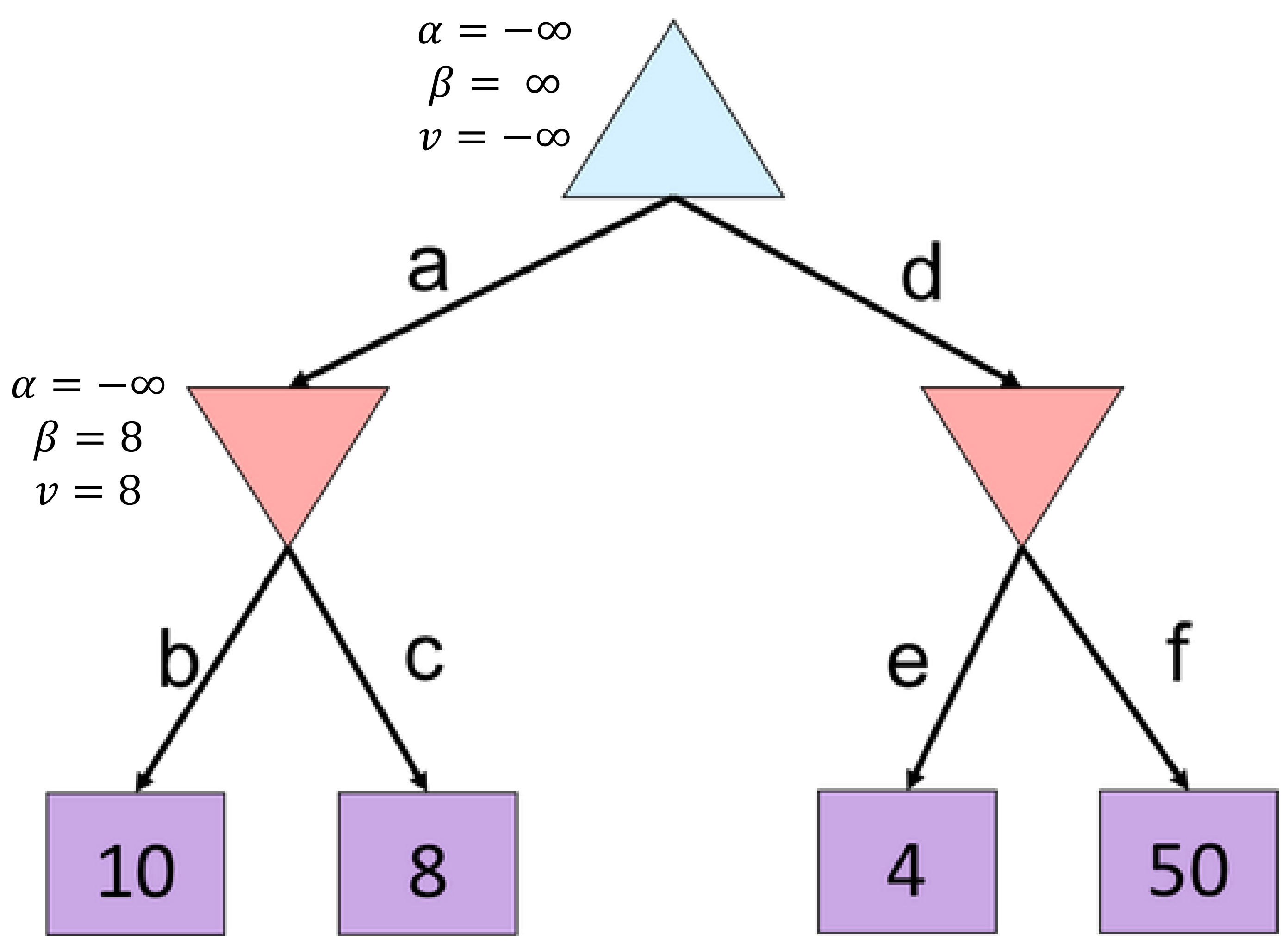
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



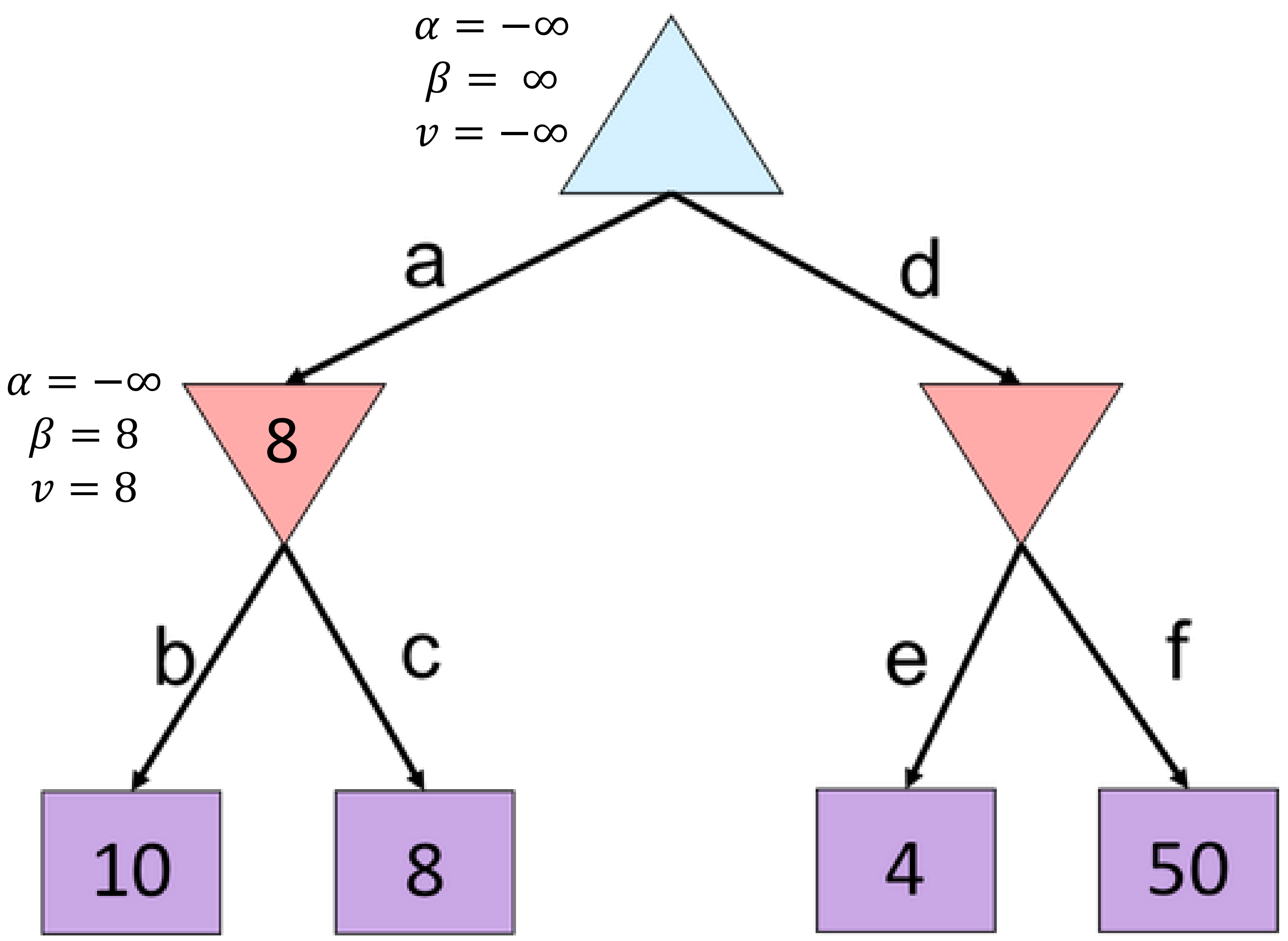
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



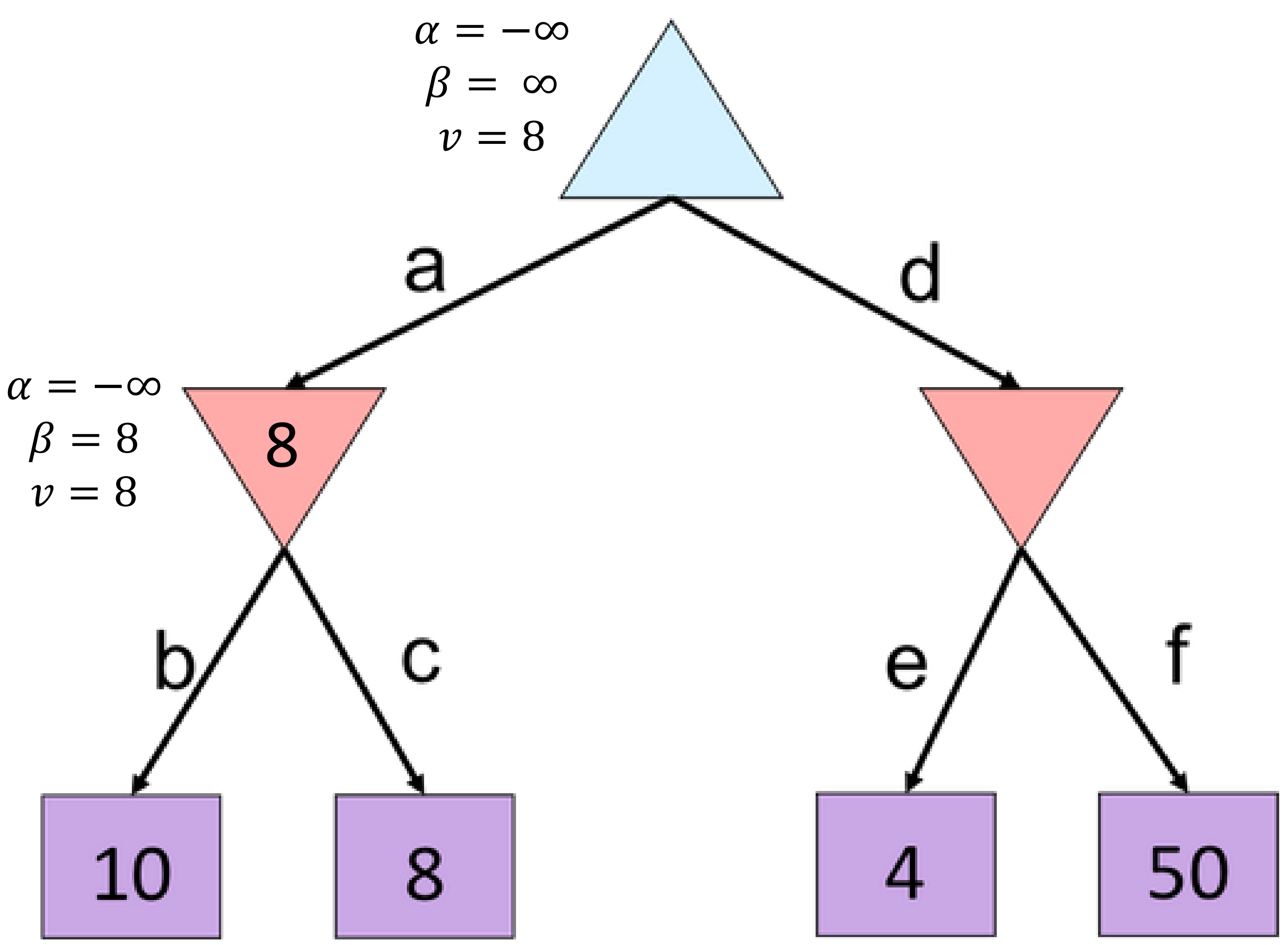
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



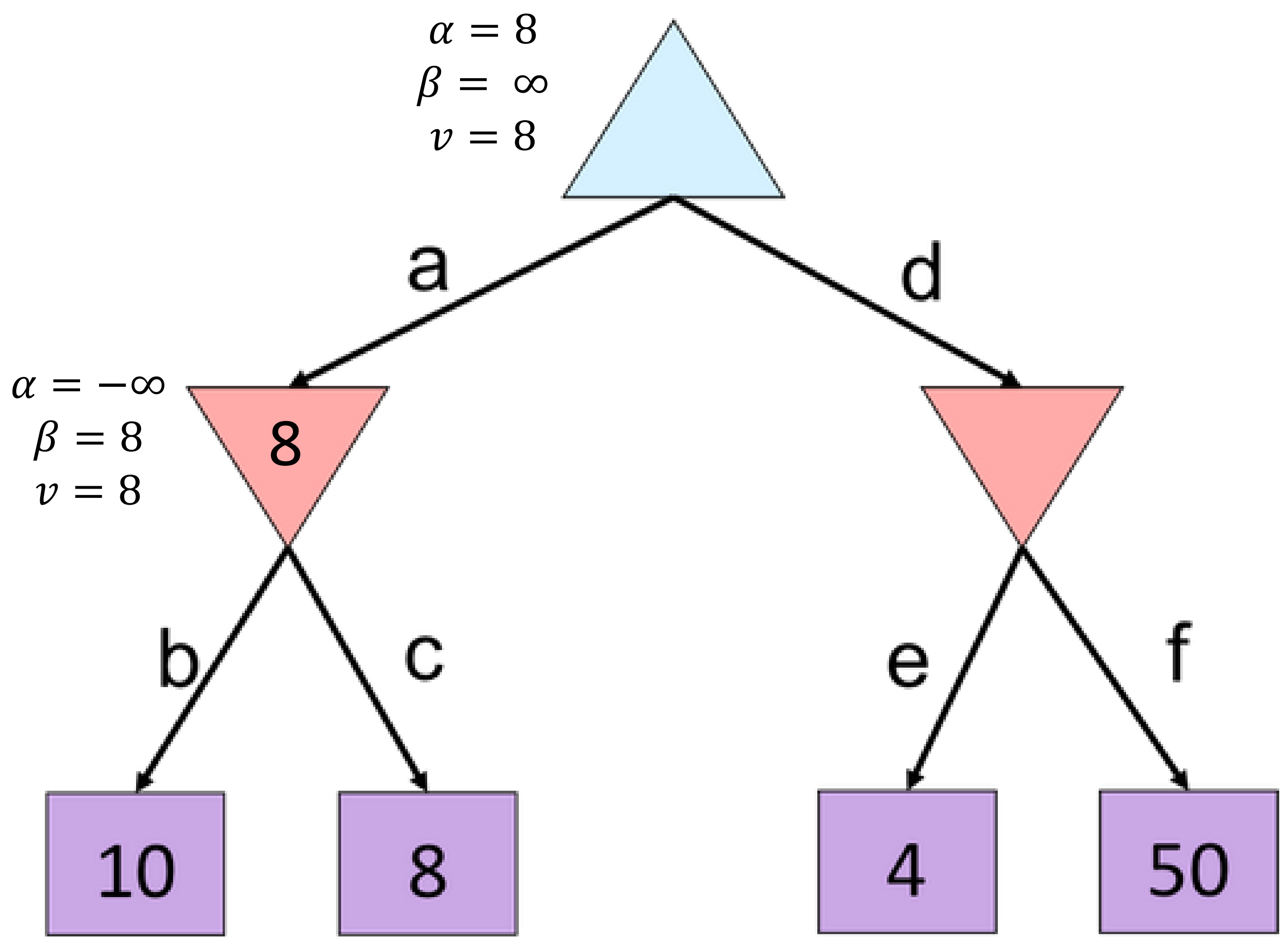
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



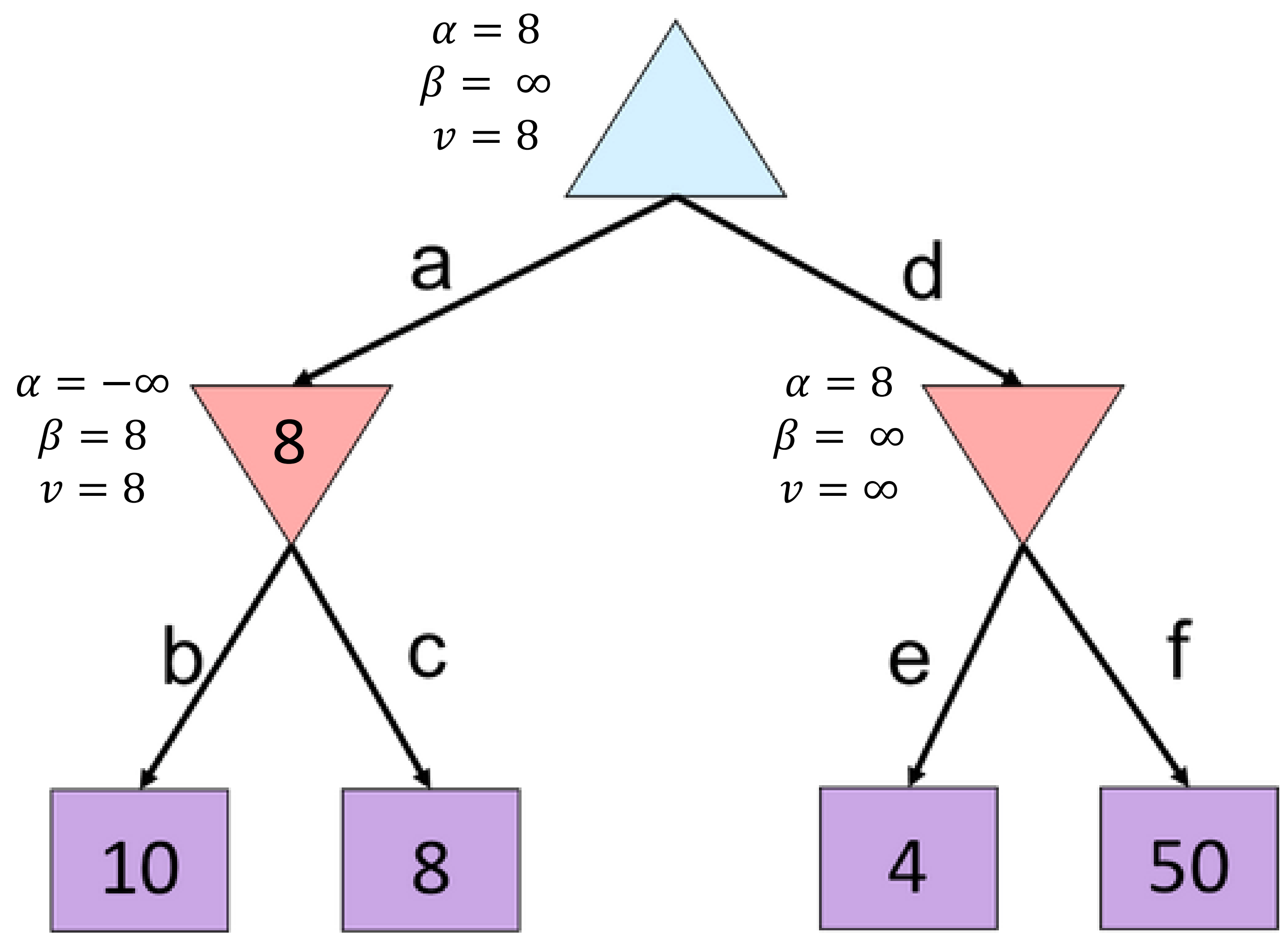
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



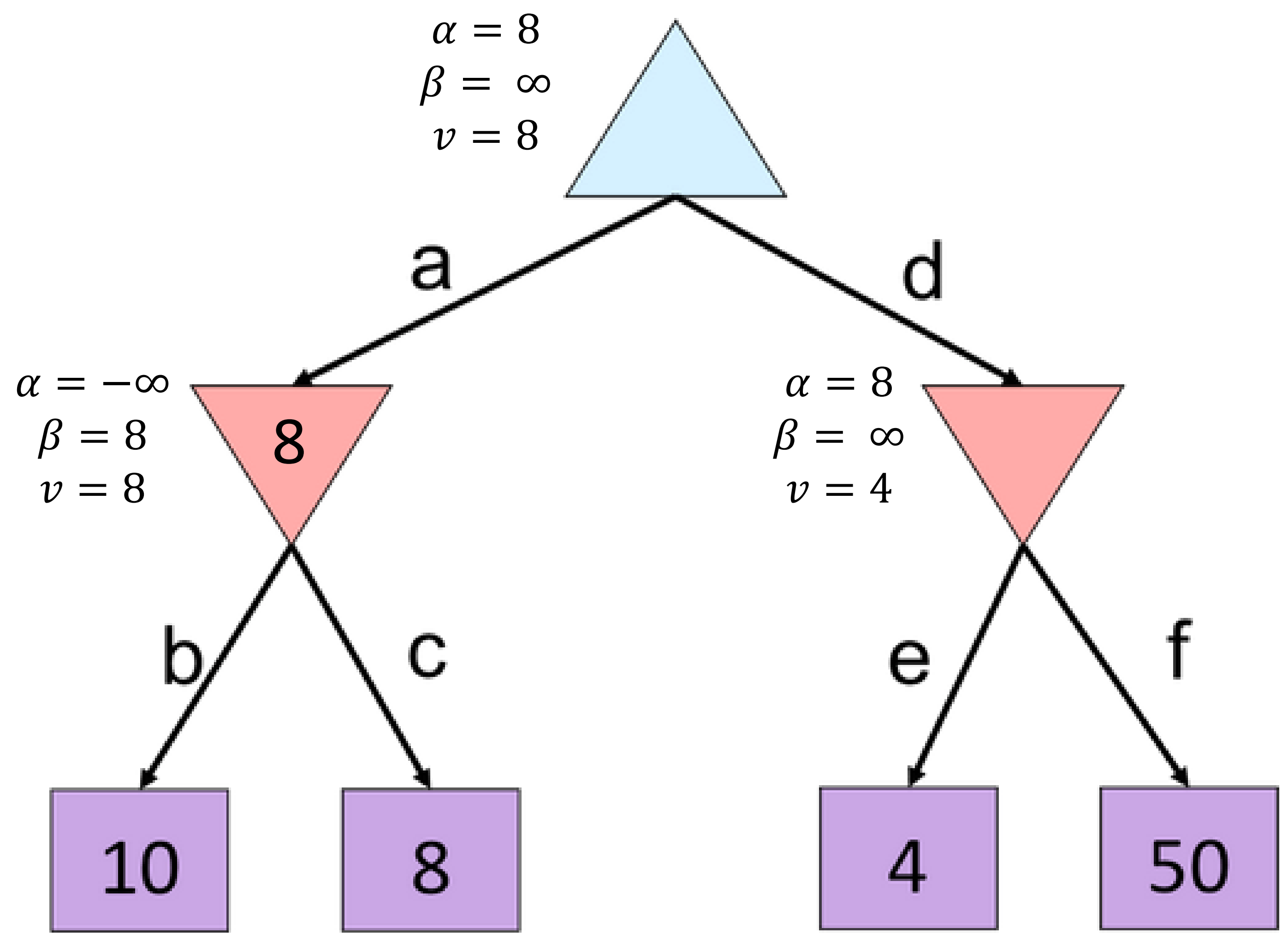
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        **α = max(α, v)**
    return v

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



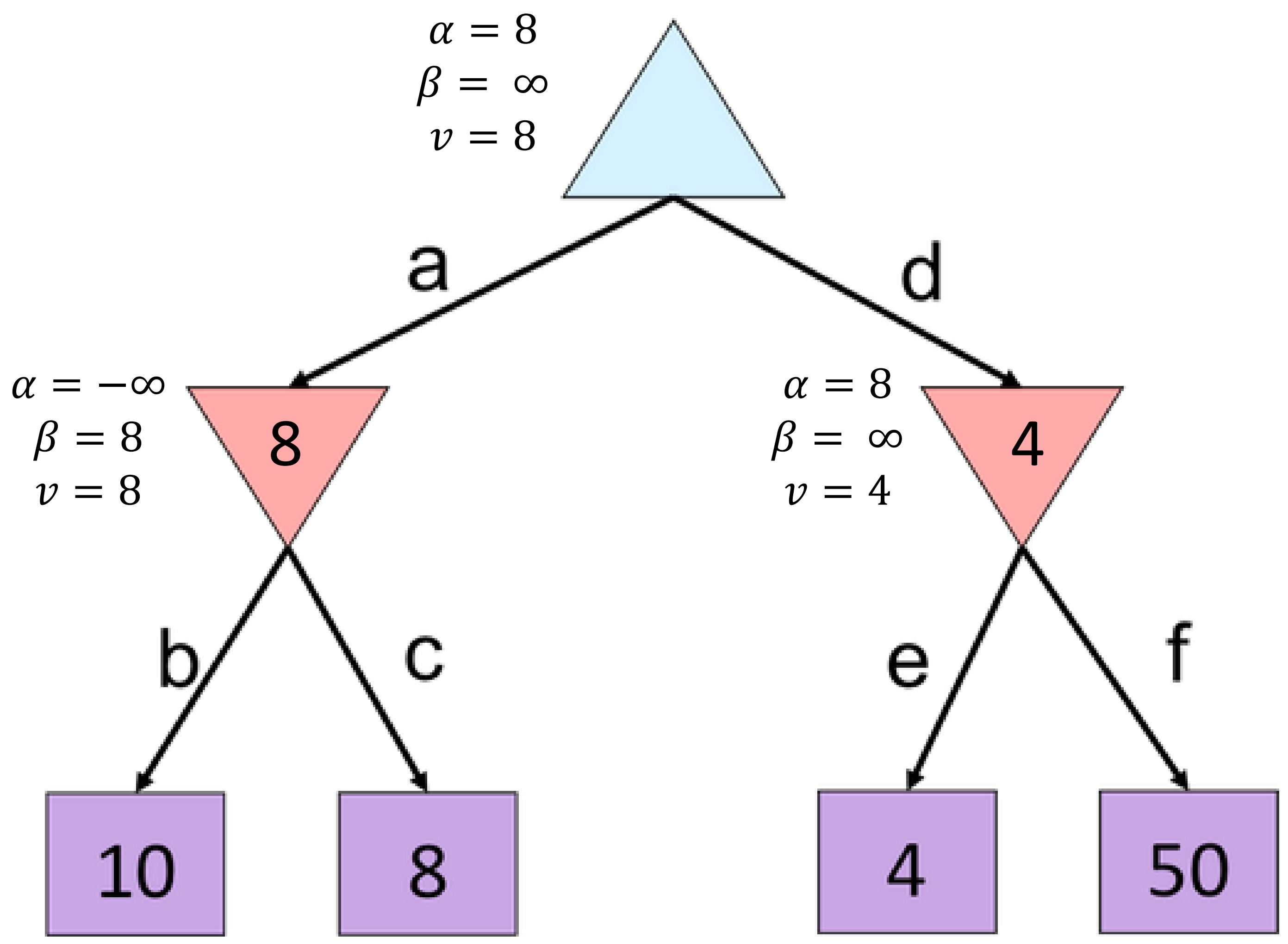
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



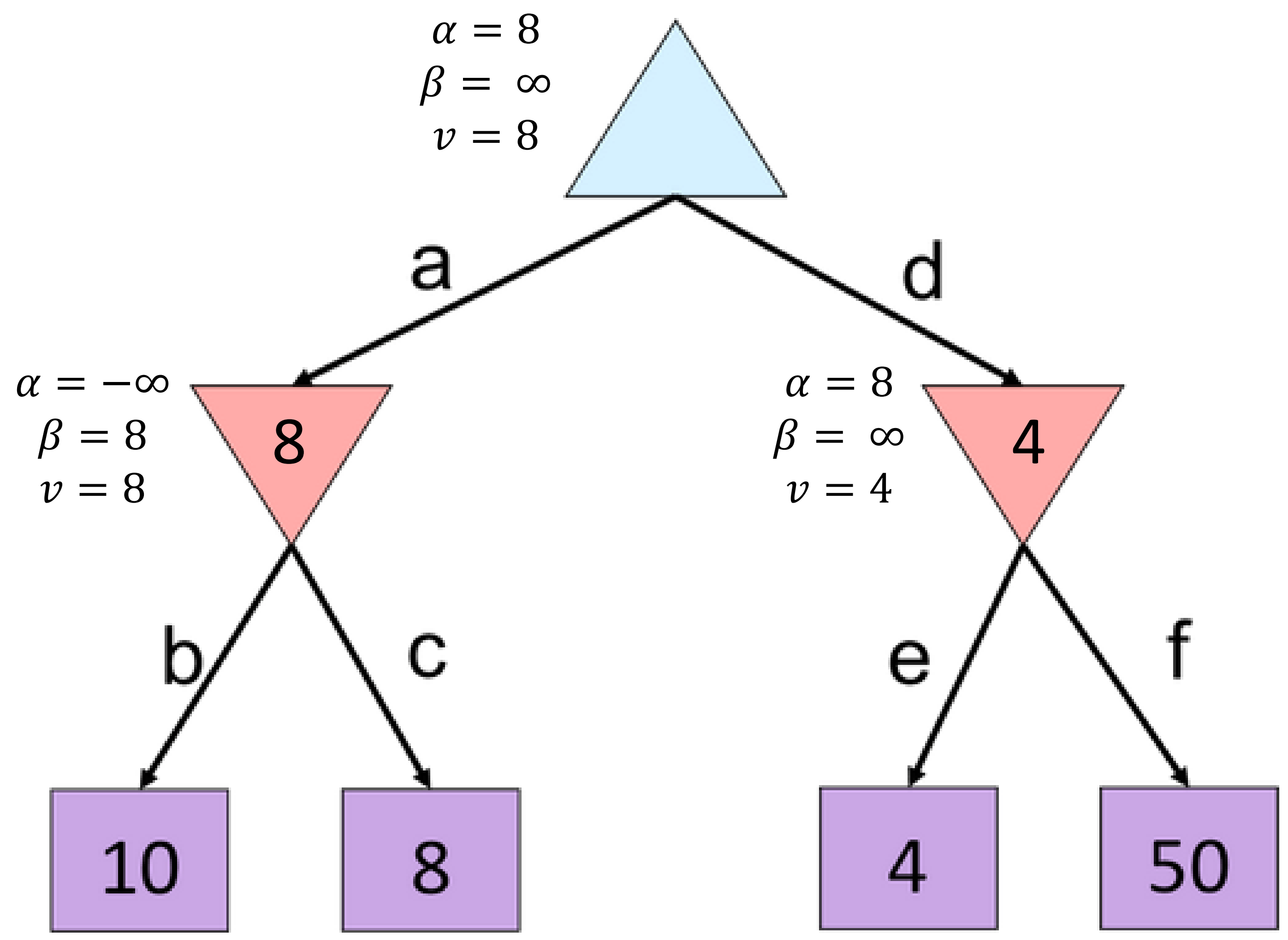
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



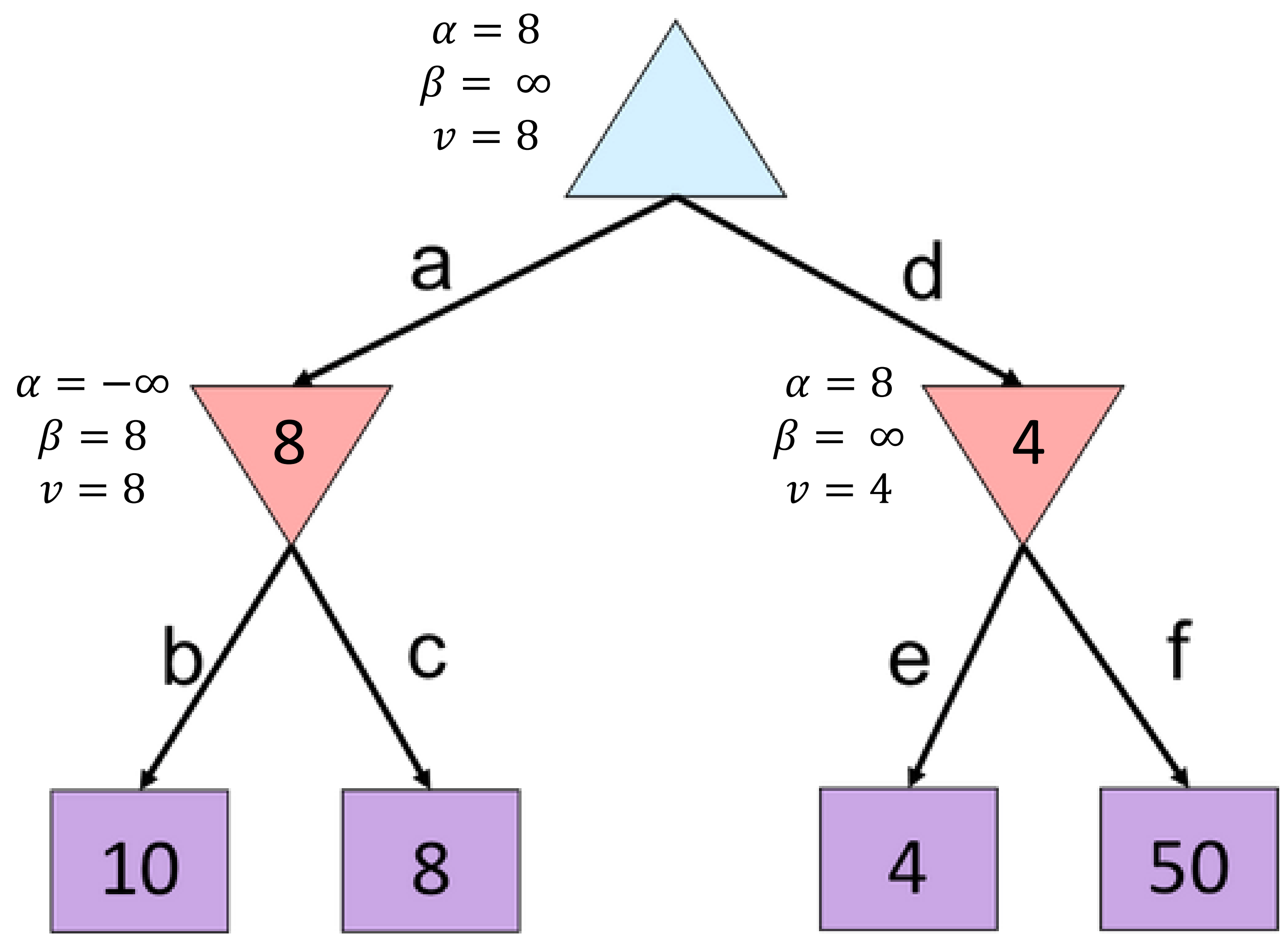
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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



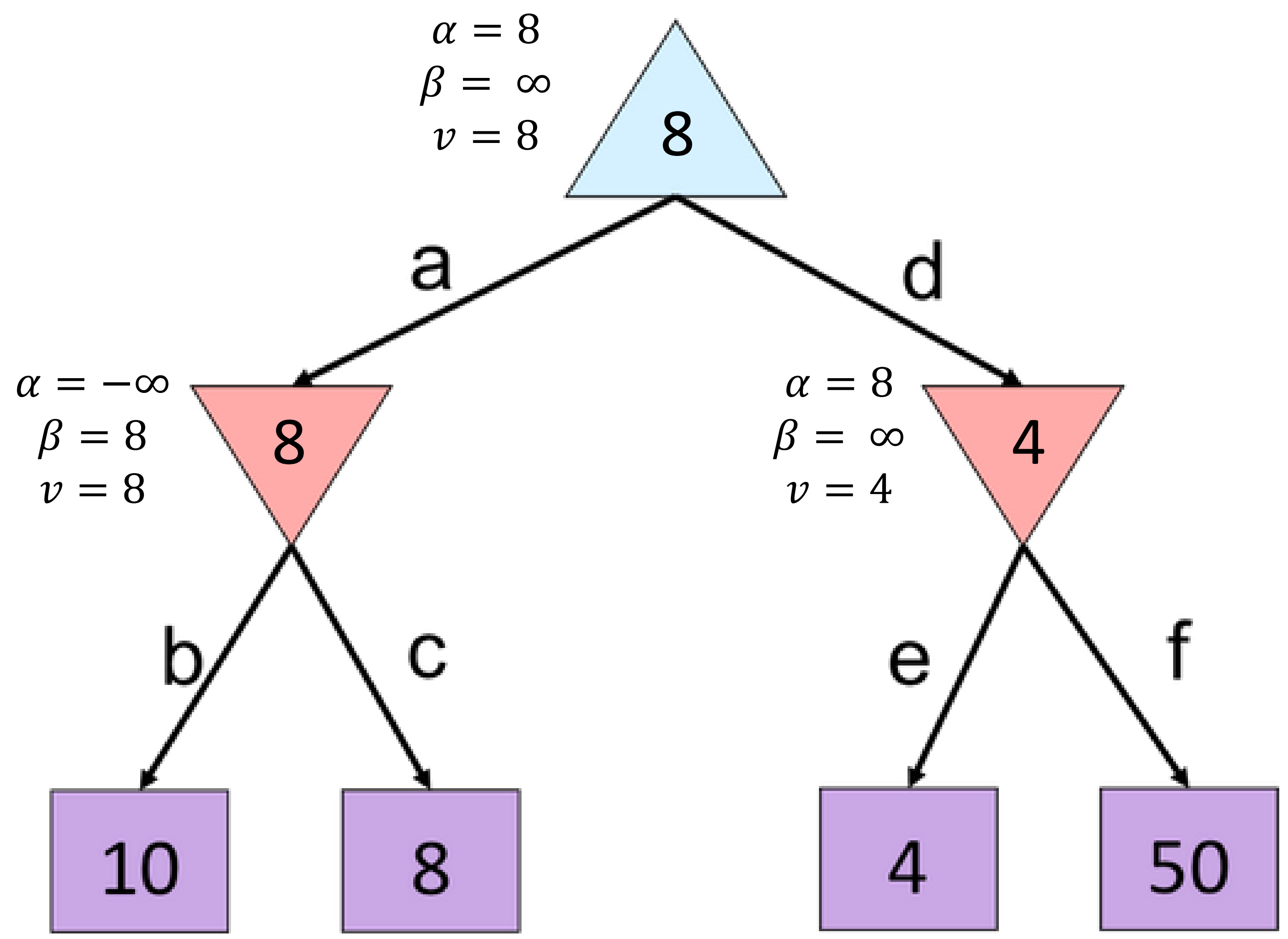
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        if v ≥ β
            return v
        α = max(α, v)
    return v

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
```

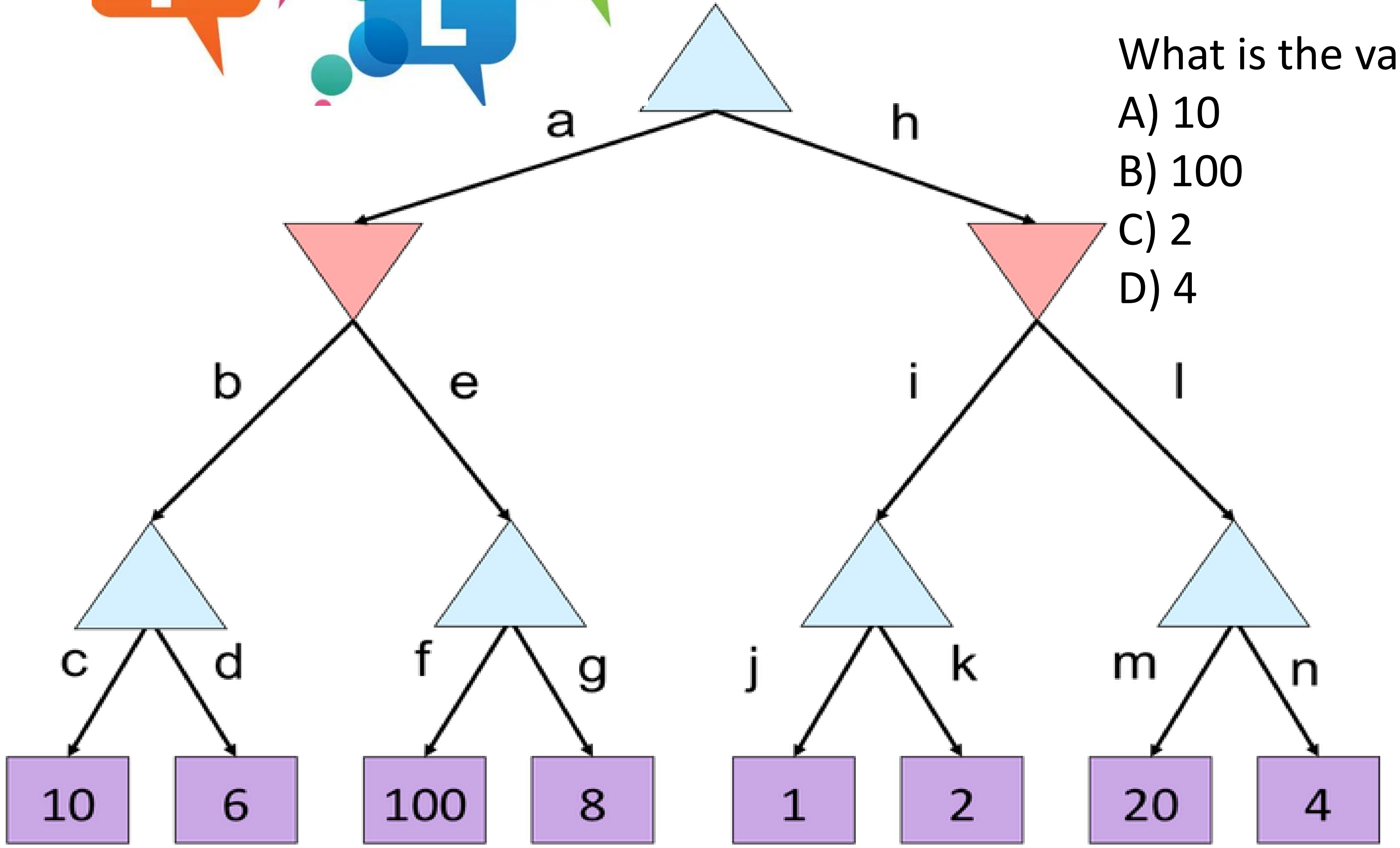
Alpha-Beta Small Example



```
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
```

POLL

4



What is the value of the top node?

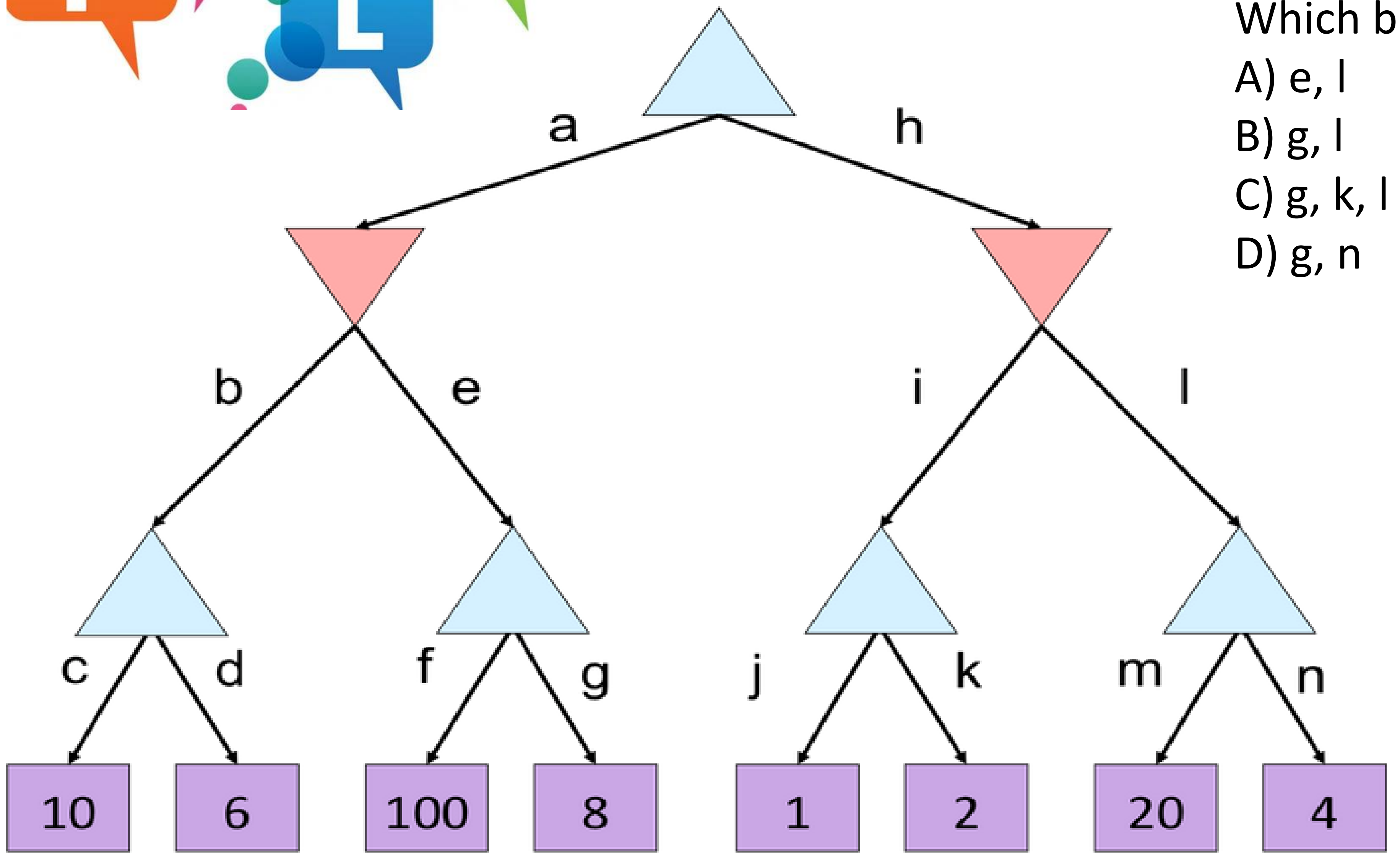
A) 10

B) 100

C) 2

D) 4

5



Which branches are pruned?

A) e, l

B) g, l

C) g, k, l

D) g, n

5-answer

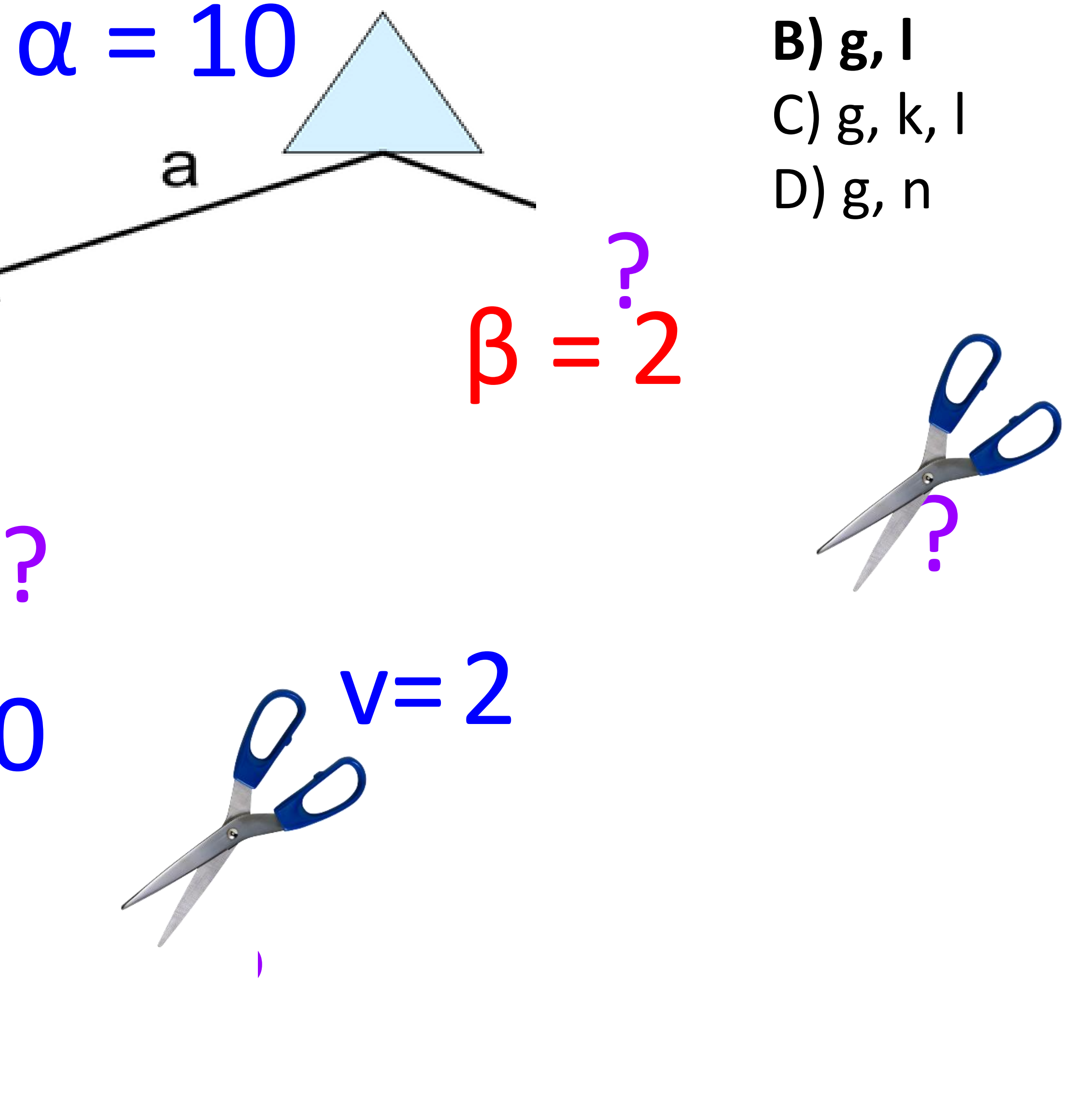
Which branches are pruned?

A) e, l

B) g, l

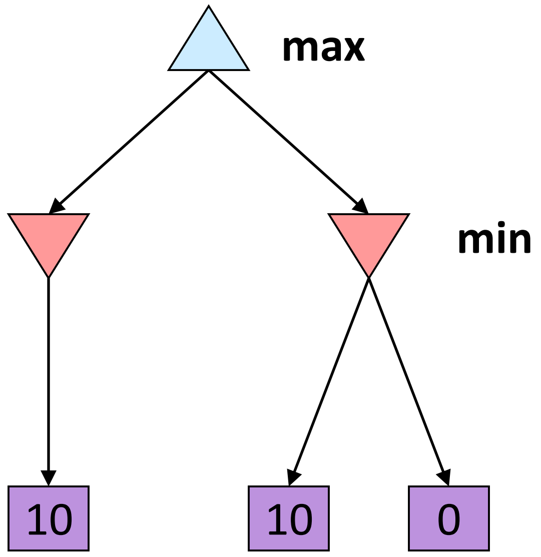
C) g, k, l

D) g, n



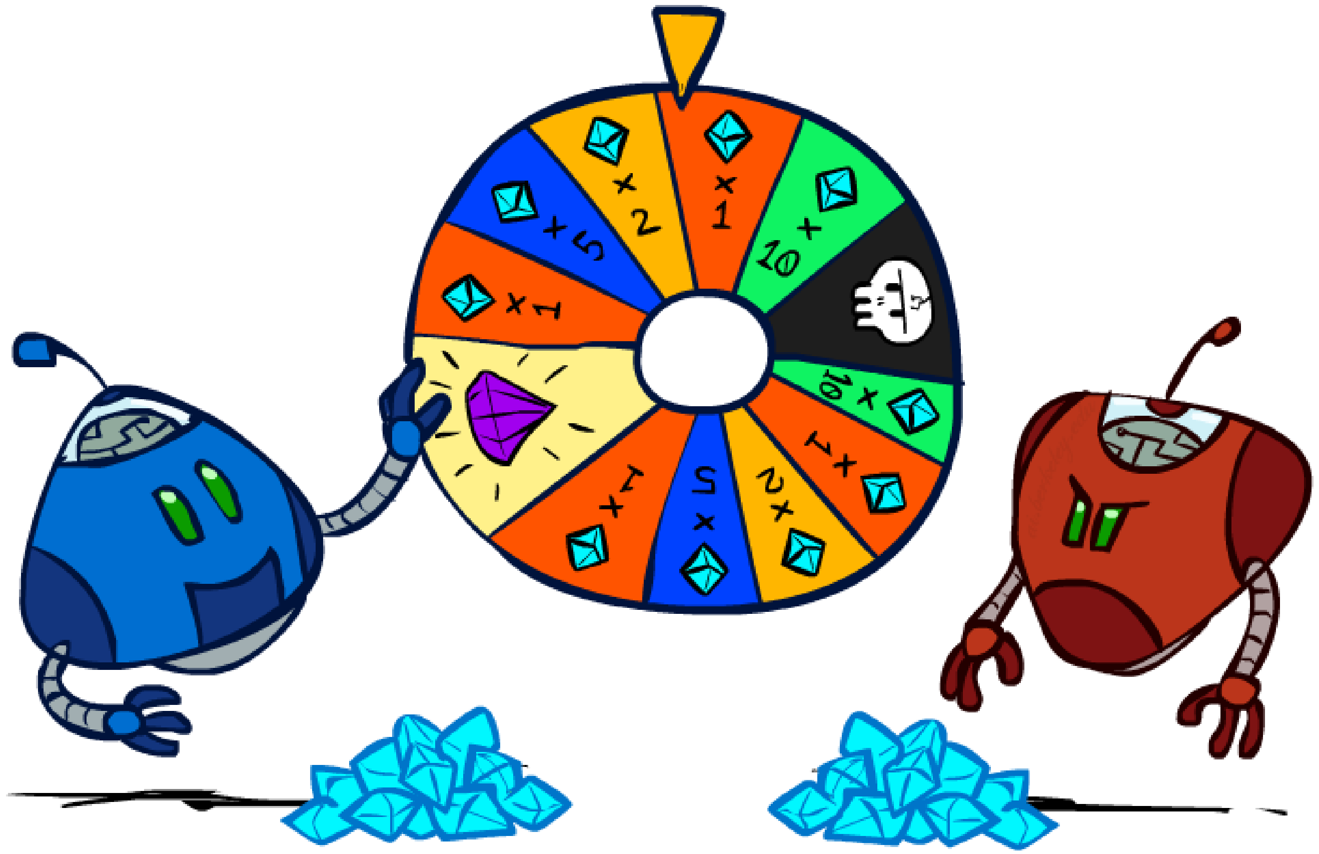
Alpha-Beta Pruning Properties

- 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



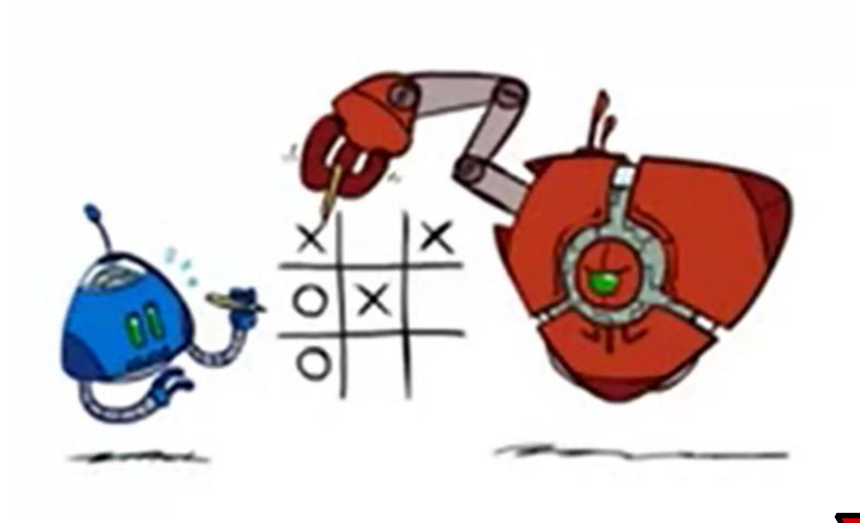
- This is a simple example of metareasoning (computing about what to compute)

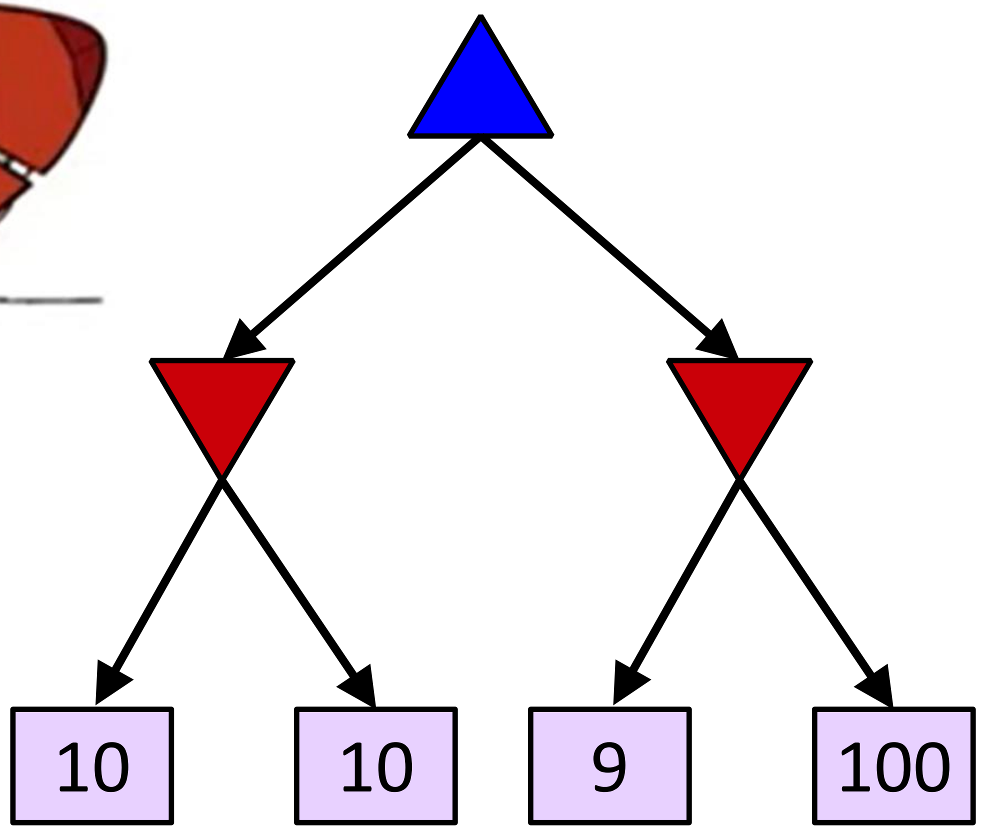
Games with uncertain outcomes



Modeling Assumptions

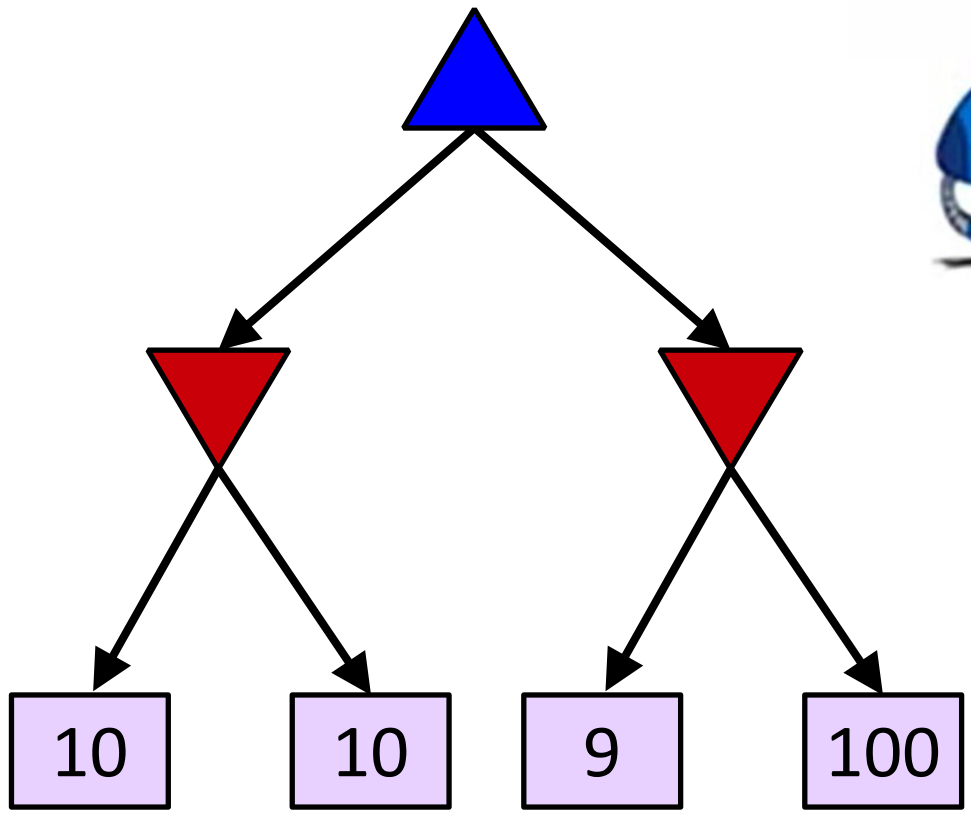
Know your opponent





Modeling Assumptions

Know your opponent



Modeling Assumptions

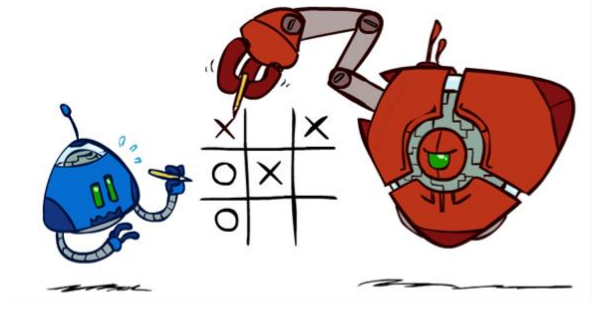
Dangerous Pessimism

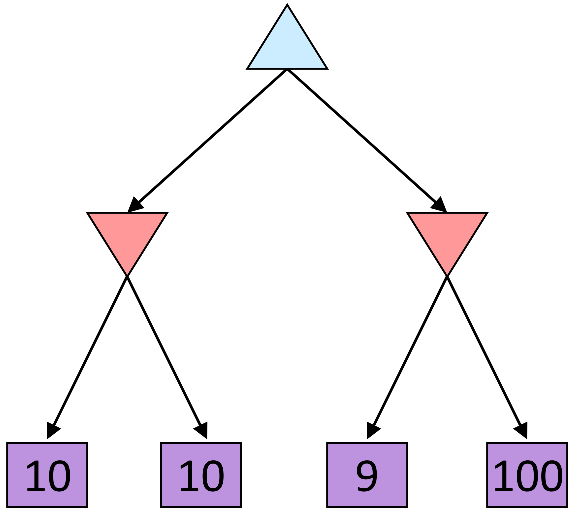
Assuming the worst case when it's not likely

Dangerous Optimism

Assuming chance when the world is adversarial

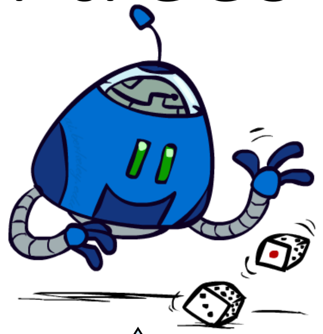
Chance outcomes in trees

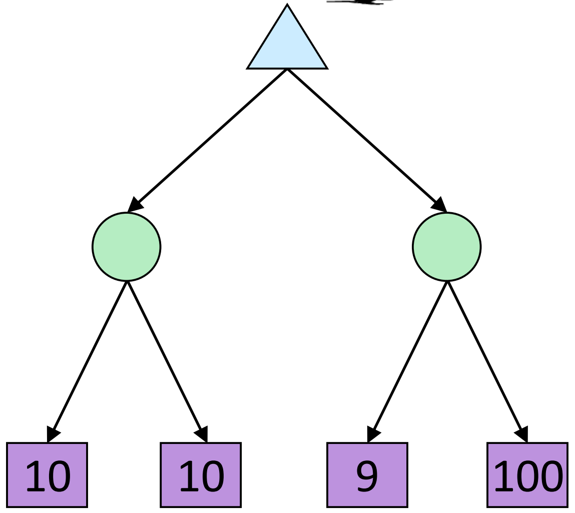




Tictactoe, chess

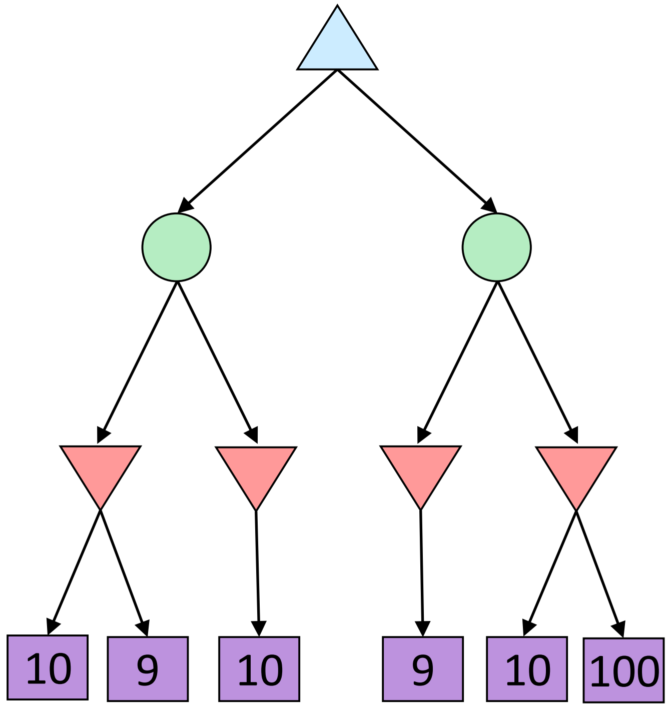
Minimax





Tetris, investing

Expectimax



Backgammon, Monopoly

Expectiminimax

Minimax

```
function decision(s) returns an action
    return the action a in Actions(s) with the highest
     value(Result(s,a))
```

↕

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

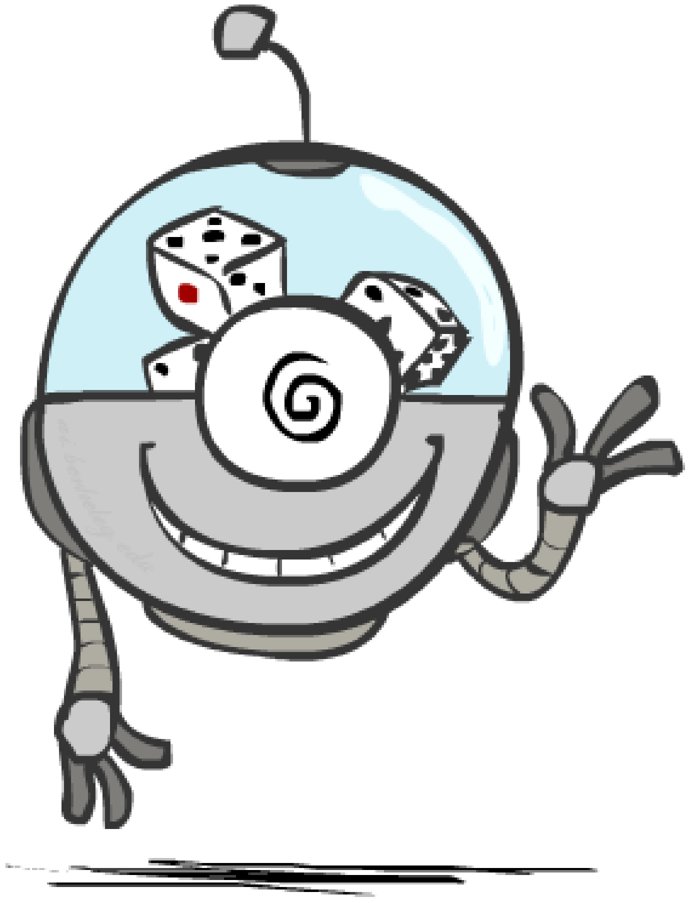
Expectiminimax

```
function decision(s) returns an action
    return the action a in Actions(s) with the highest
     value(Result(s,a))
```

↕

```
function value(s) returns a value
    if Terminal-Test(s) then return Utility(s)
    if Player(s) = MAX       then return $\max_{a \text{ in Actions}(s)}$ value(Result(s,a))
    if Player(s) = MIN       then return $\min_{a \text{ in Actions}(s)}$ value(Result(s,a))
    if Player(s) = CHANCE then return $\text{sum}_{a \text{ in Actions}(s)}$ Pr(a) * value(Result(s,a))
```

Probabilities



Probabilities

A random variable represents an event whose outcome is unknown

A probability distribution is an assignment of weights to outcomes

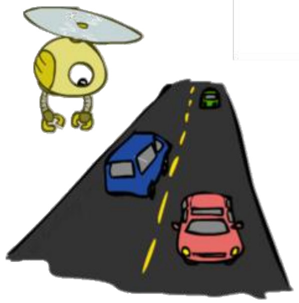
Example: Traffic on freeway

- Random variable: T = whether there's traffic
- Outcomes: T in {none, light, heavy}
- Distribution:

  P(T=none) = 0.25, P(T=light) = 0.50, P(T=heavy) = 0.25

Probabilities over all possible outcomes sum to one

0.25



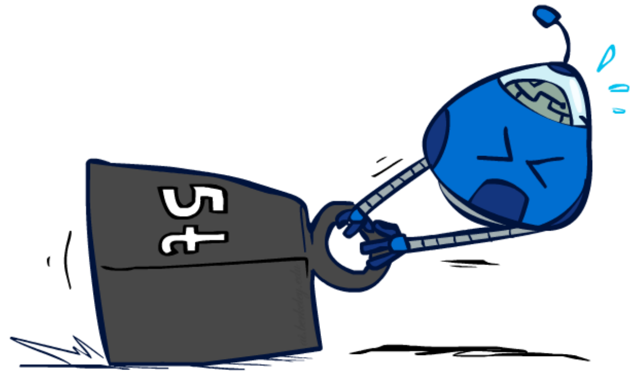
0.50

0.25

Expected Value

- The expected value of a random variable is the average, weighted by the probability distribution over outcomes

- Example: How long to get to the airport?



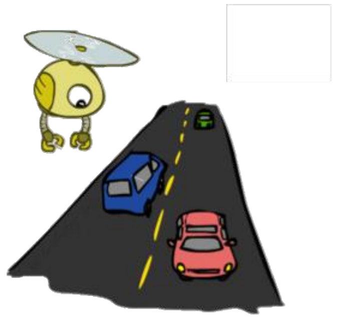
Time:	20 min		30 min		60 min		
	x	+	x	+	x	⇨	35 min
Probability:	0.25		0.50		0.25		

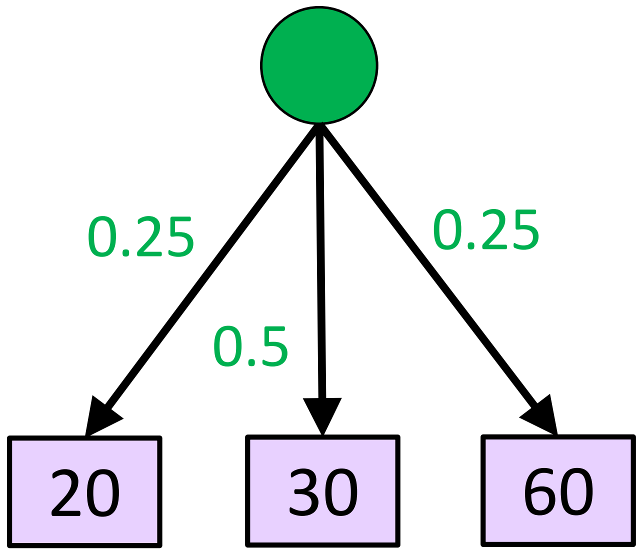
Expectations

Time: 20 min x 0.25 + 30 min x 0.50 + 60 min x 0.25

Probability:

Time:	20 min	30 min	60 min
	x	x	x
Probability:	0.25	0.50	0.25





Max node notation

$$V(s) = \max_{a} V(s'),$$

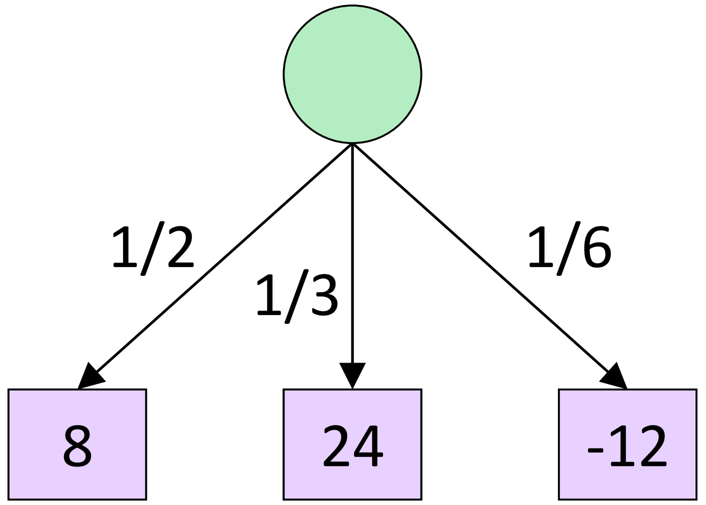
where $s' = re\quad lt(s,a)$

Chance node notation

$$V(s) = \underset{s'}{\square}\ P(s')\,V(s')$$

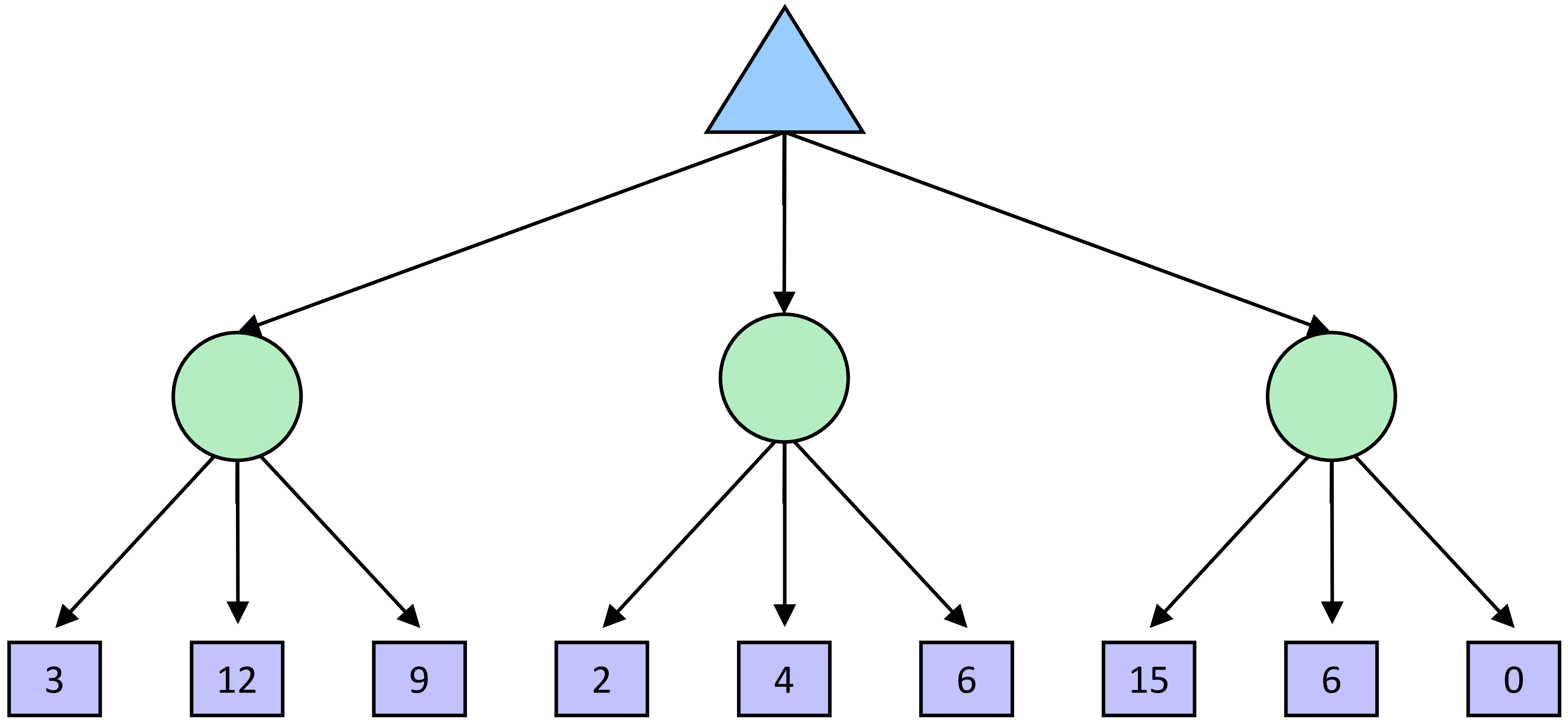
Expectimax Pseudocode

$\text{sum}_{a \text{ in Action}(s)} \Pr(a) * \text{value}(\text{Result}(s,a))$

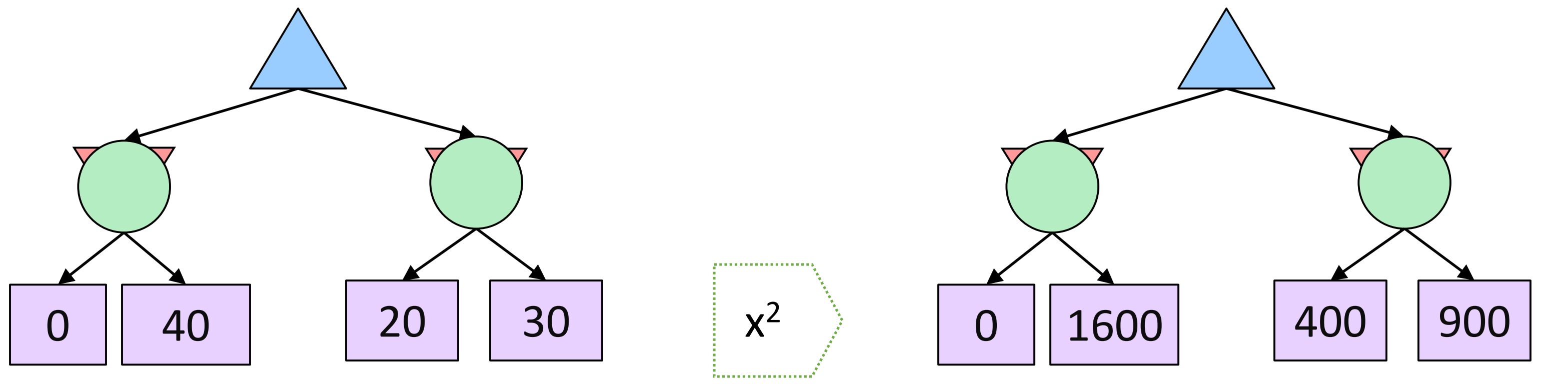


v = (1/2) (8) + (1/3) (24) + (1/6) (-12) = 10

Expectimax Example



What Values to Use?



x>y => f(x)>f(y)

f(x) = Ax+B where A>0

- For worst-case minimax reasoning, evaluation func n scale doesn't ma
  - We just want better states to have higher evalua ons (get the ord g right)
  - Minimax decisions are ***invariant with respect to monotonic tran formations on values***

- Expectiminimax decisions are ***invariant with respect to positiv affine transformations***
- Expectiminimax evaluation functions have to be aligned with actual win probabilities!

Summary

- Multi-agent problems can require more space or deeper trees to search
- Games require decisions when optimality is impossible
  - Bounded-depth search and approximate evaluation functions
- Games force efficient use of computation
  - Alpha-beta pruning
- Game playing has produced important research ideas
  - Reinforcement learning (checkers)
  - Iterative deepening (chess)
  - Rational metareasoning (Othello)
  - Monte Carlo tree search (Go)
  - Solution methods for partial-information games in economics (poker)
- Video games present much greater challenges – lots to do!
  - $b = 10^{500}$, $|S| = 10^{4000}$, $m = 10{,}000$