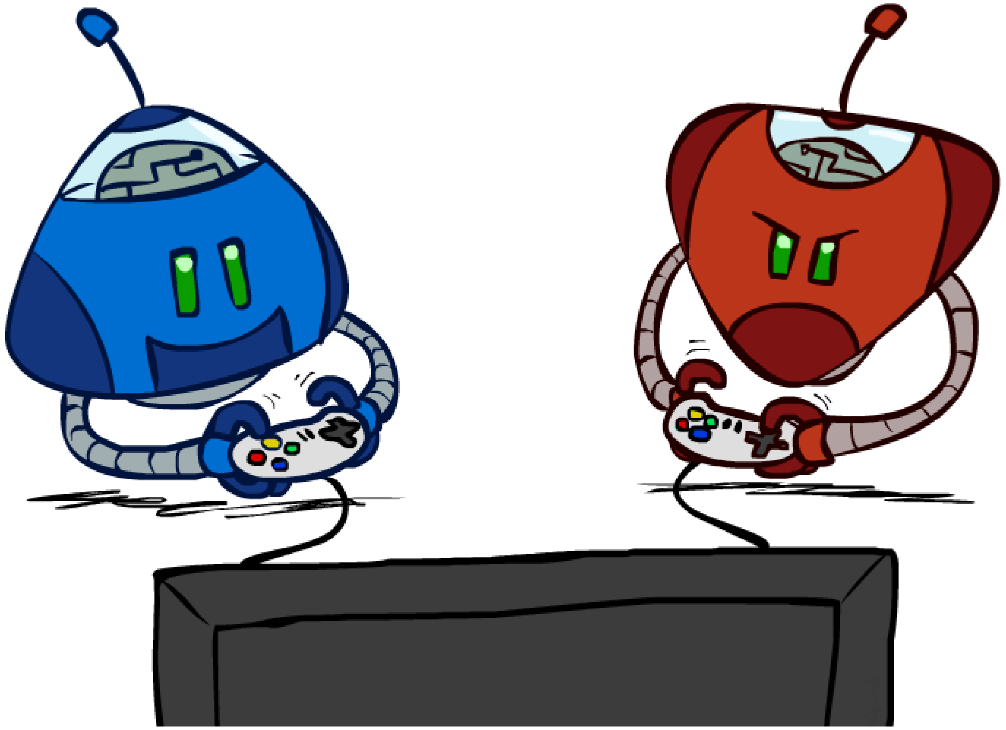
AI: Representation and Problem Solving

Adversarial Search



Instructor: Pat Virtue

Slide credits: CMU AI, 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

α/β pruning

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

Heuristic

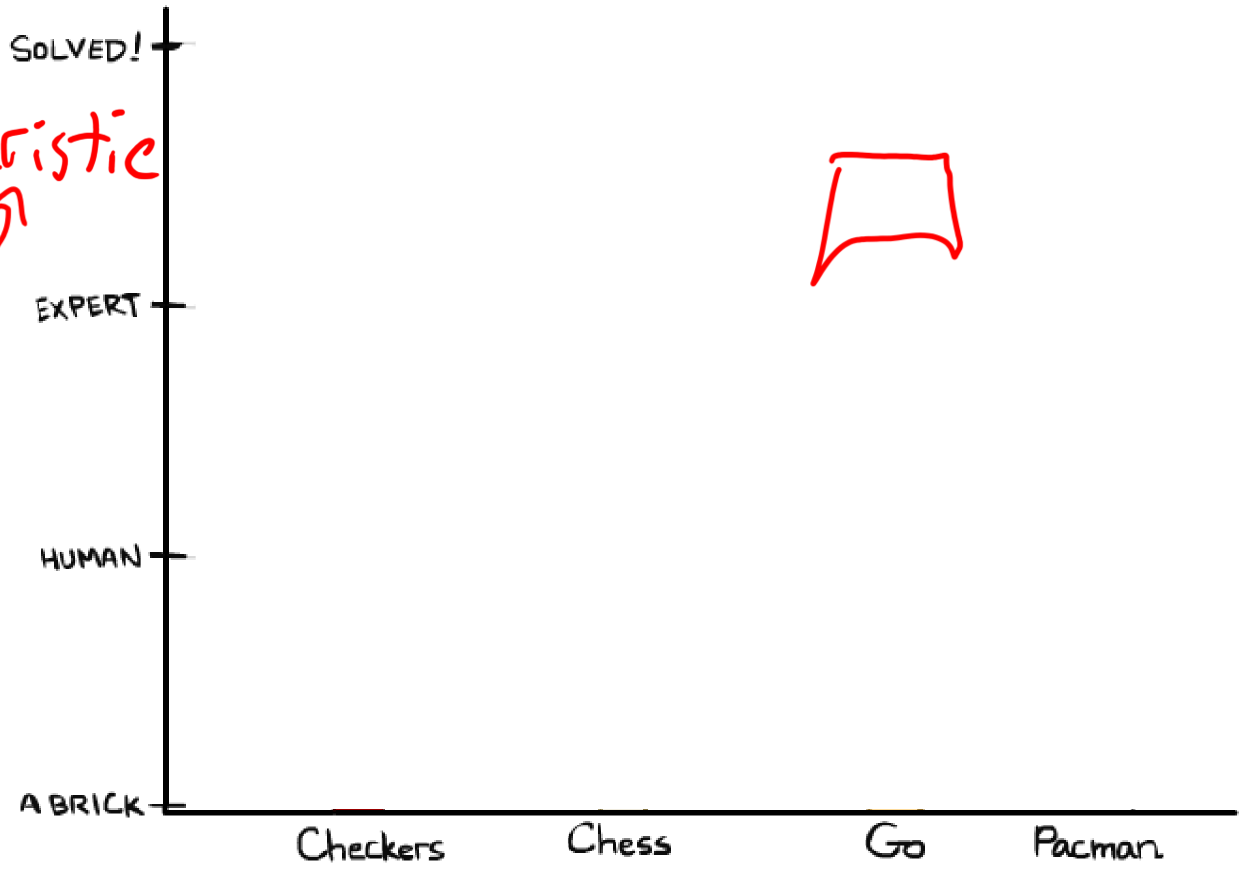
Chess:

- 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).

300

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.



Types of Games

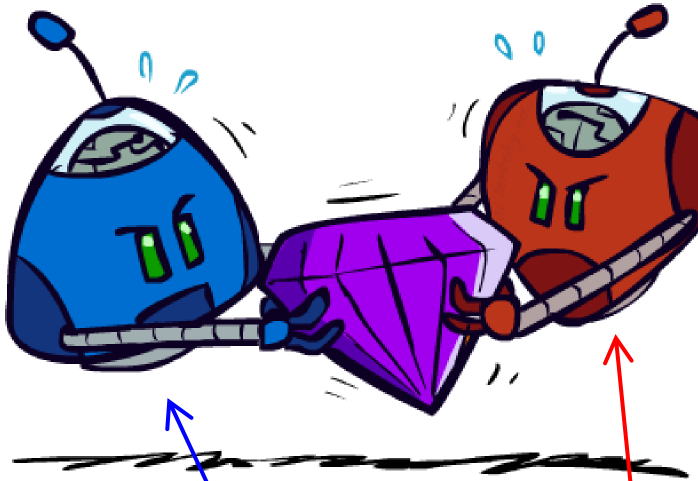
Many different kinds of games!

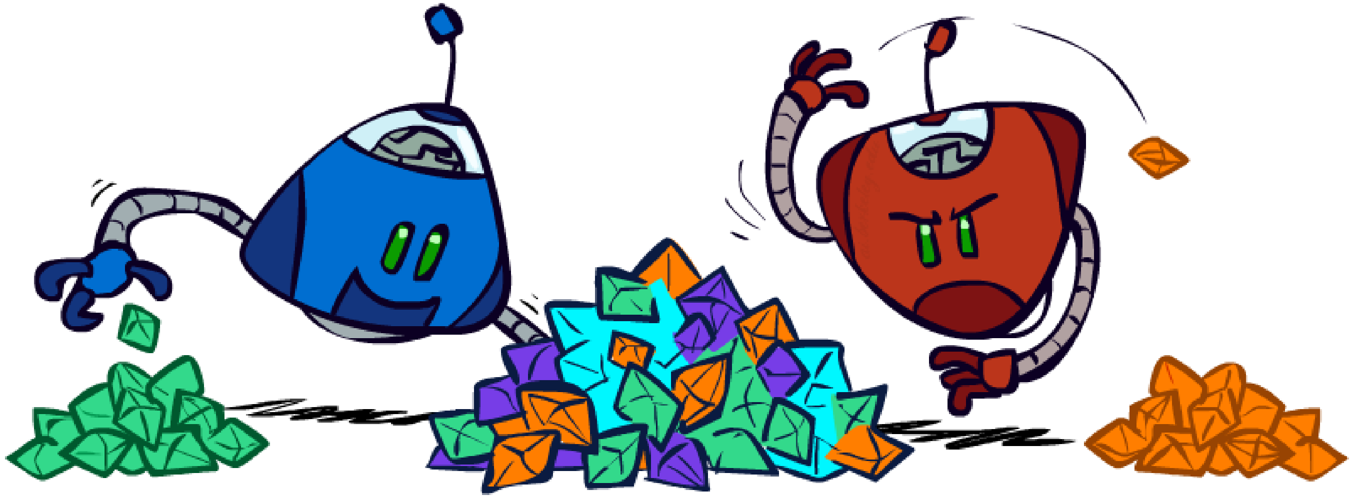
Axes:

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

Want algorithms for calculating a ***contingent plan*** (a.k.a. strategy or policy) which recommends a move for every possible eventuality

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

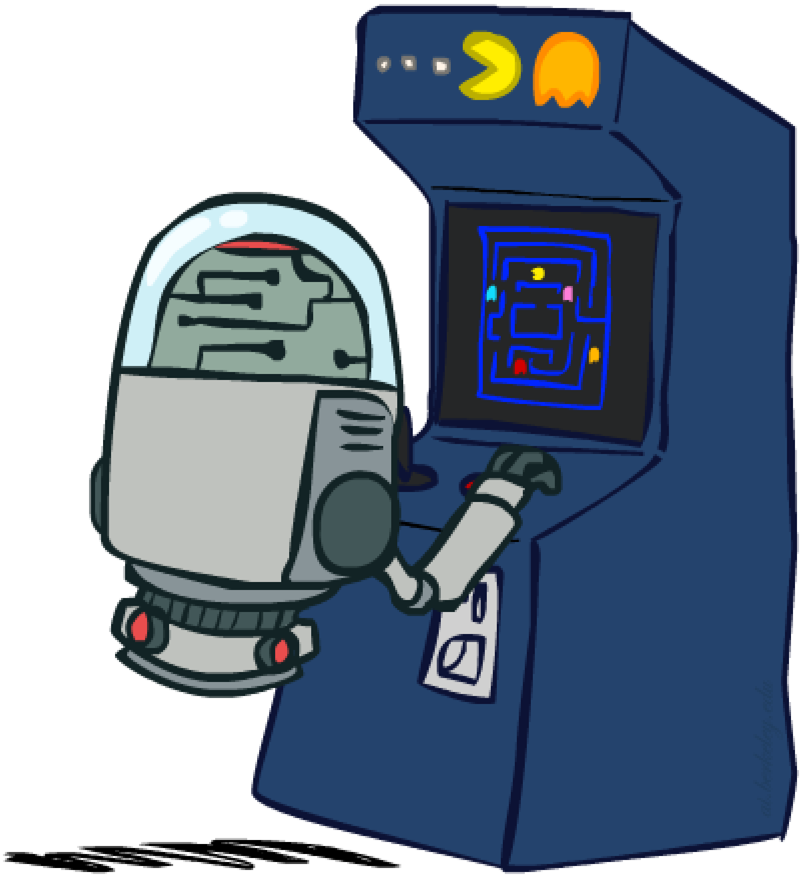
Standard games are:

deterministic, observable, two-player, turn-taking, and zero-sum

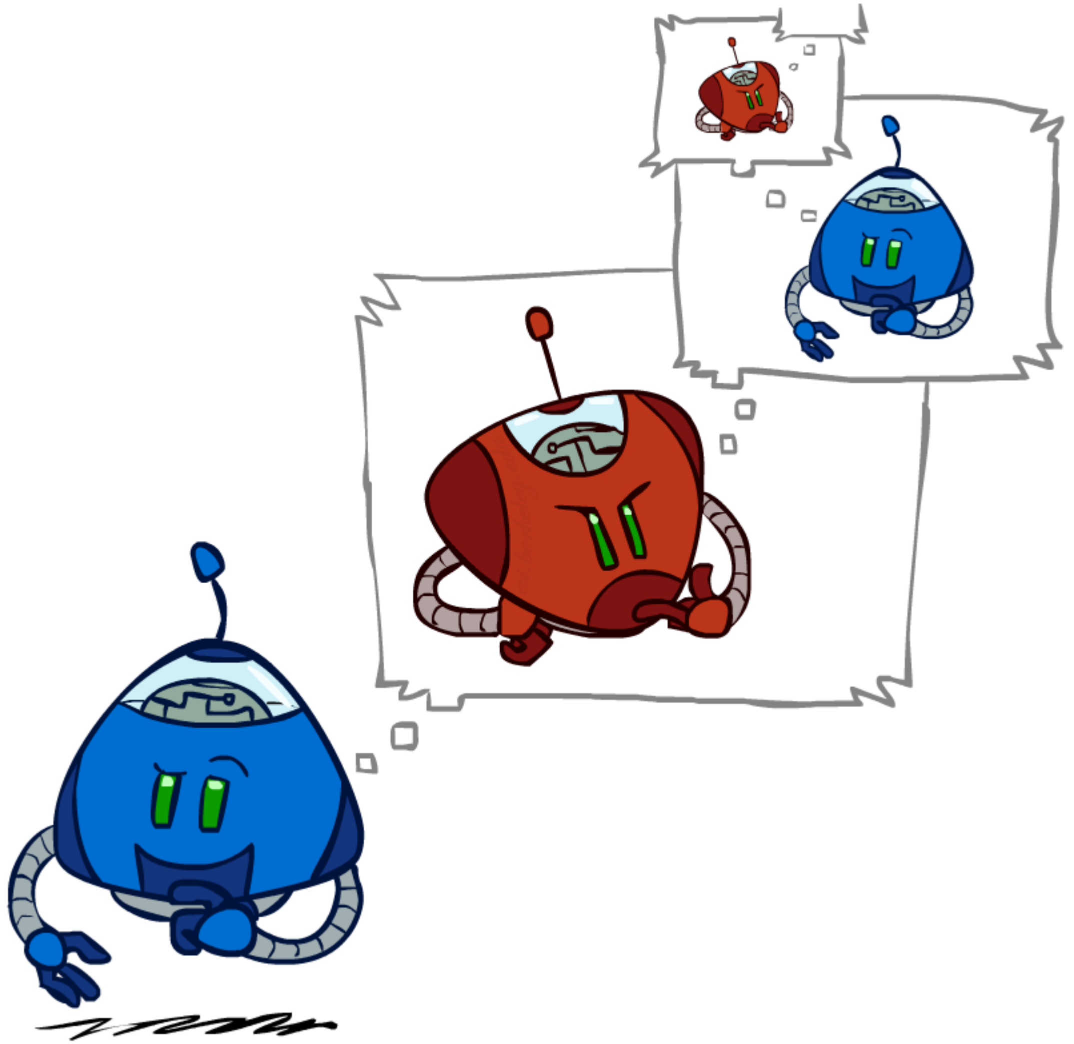
Game formulation:

- 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)

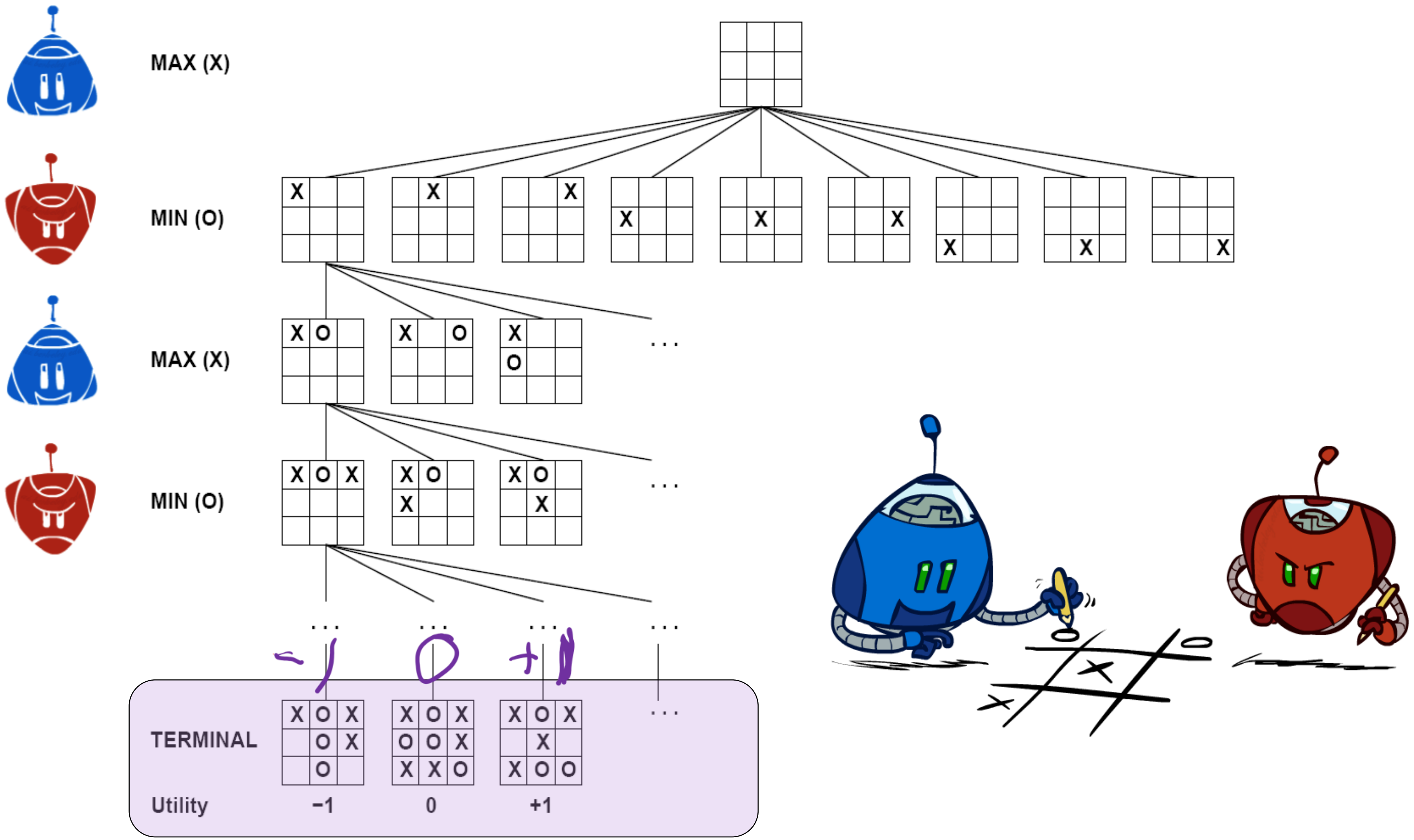


Adversarial Search



Minimax

- States
- Actions
- Values

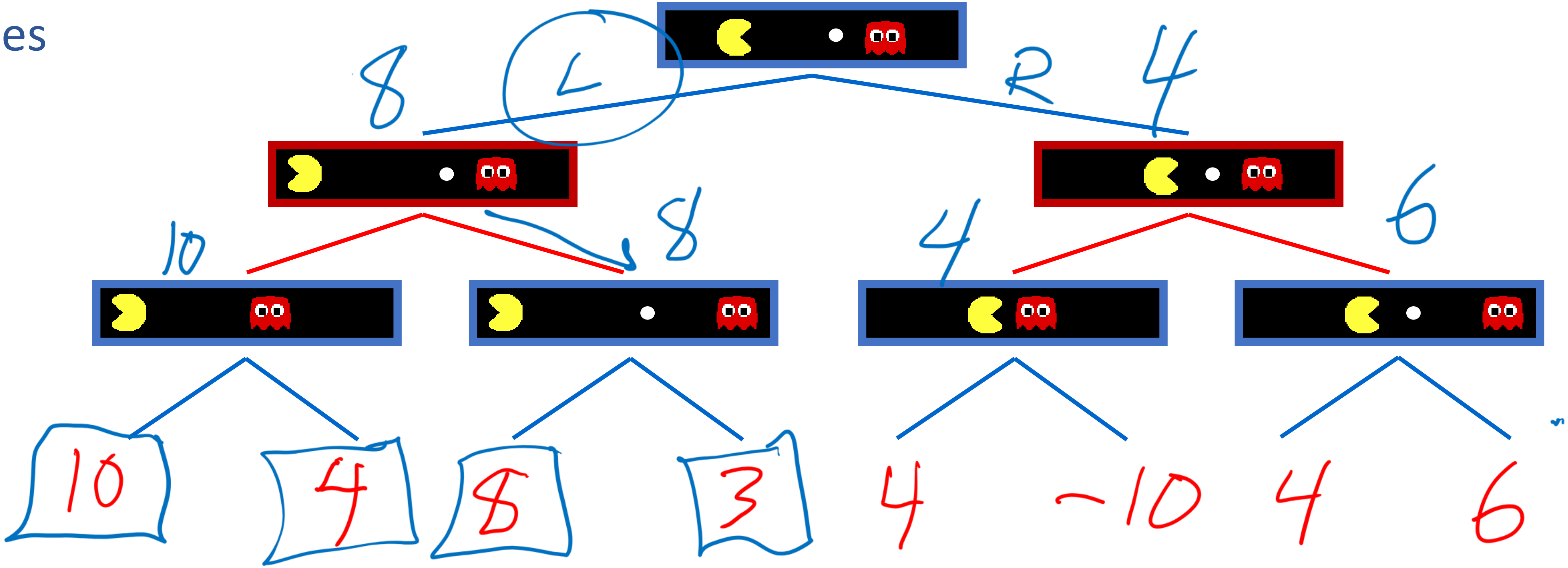


Minimax

States

Actions

Values



Poll 1

What is the minimax value at the root?

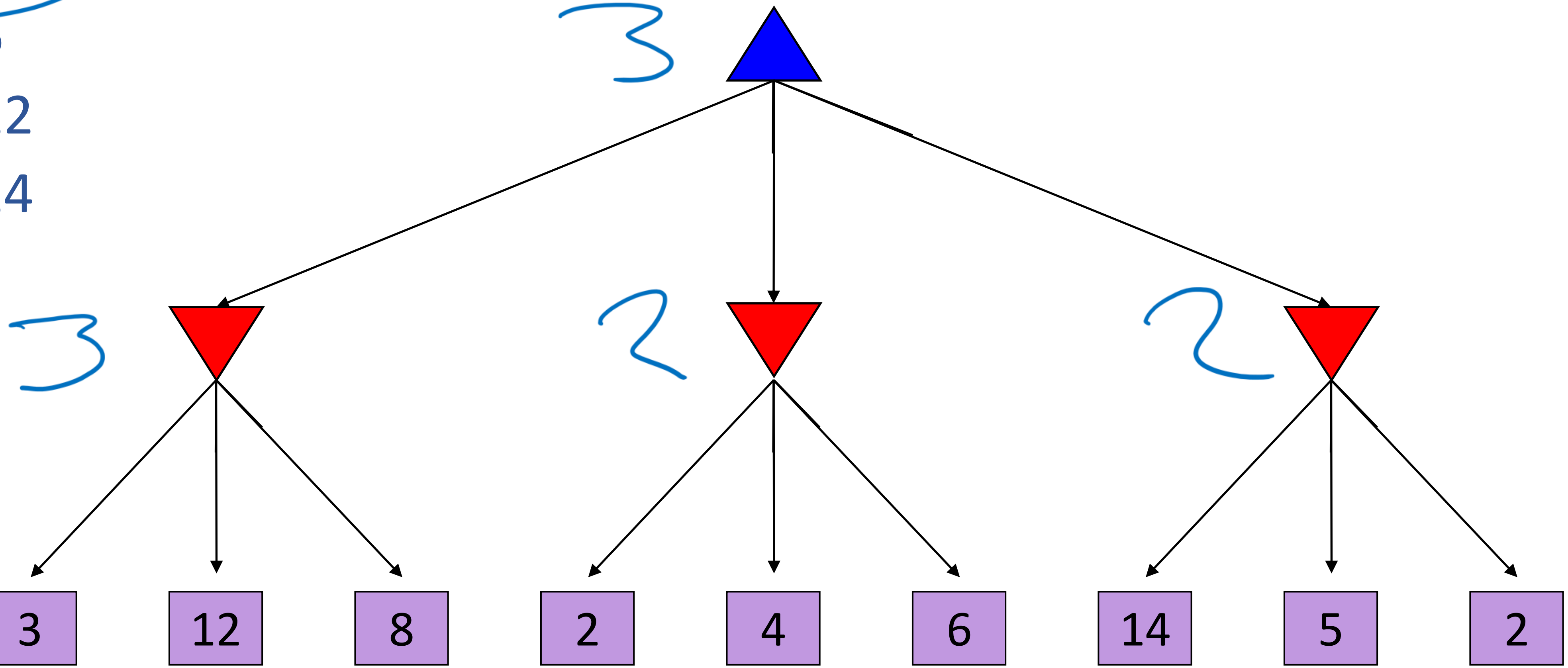
A) 2

B) 3

C) 6

D) 12

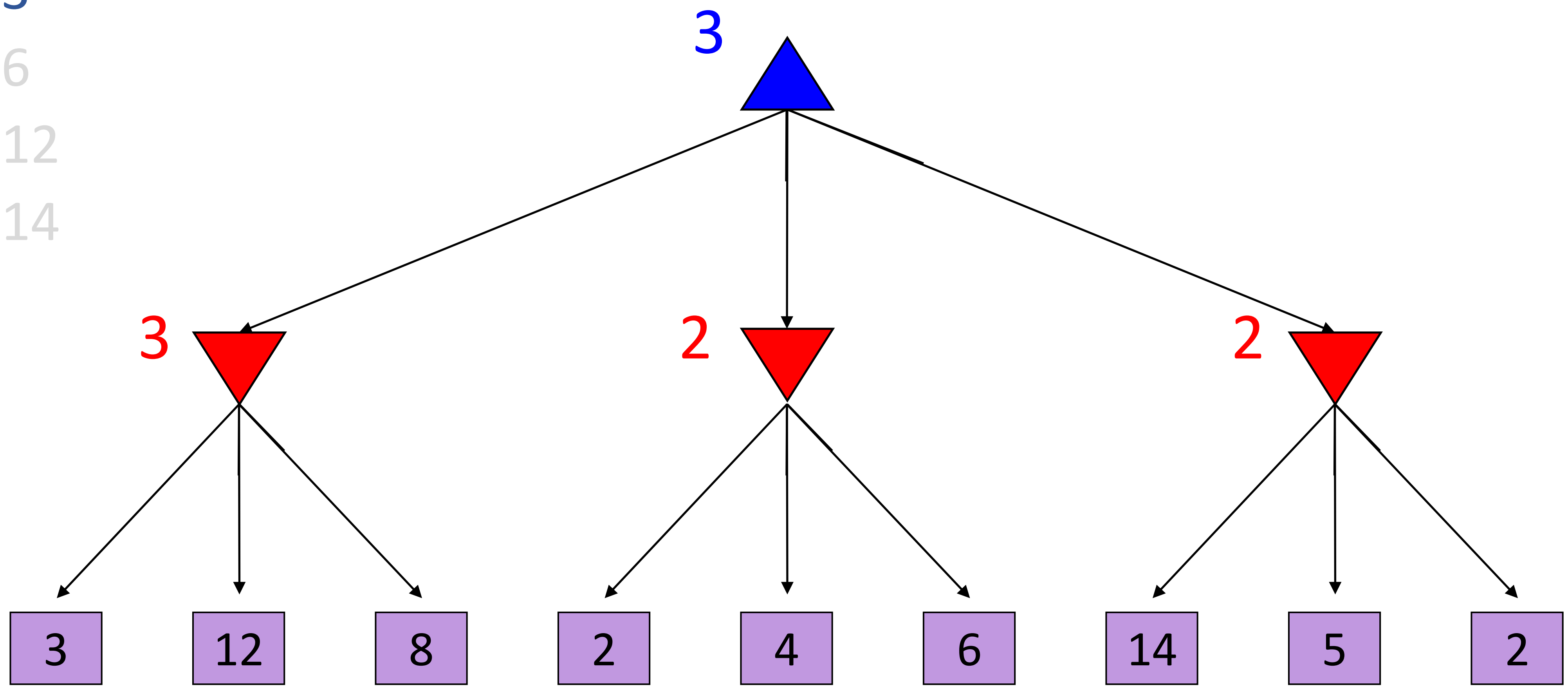
E) 14



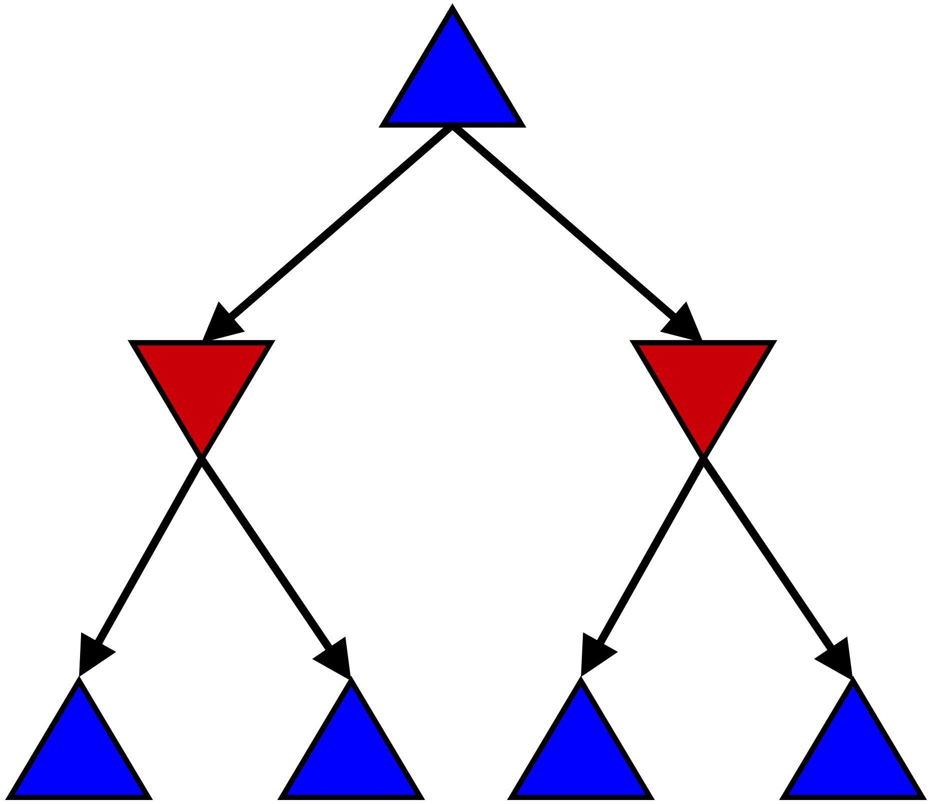
Poll 1

What is the minimax value at the root?

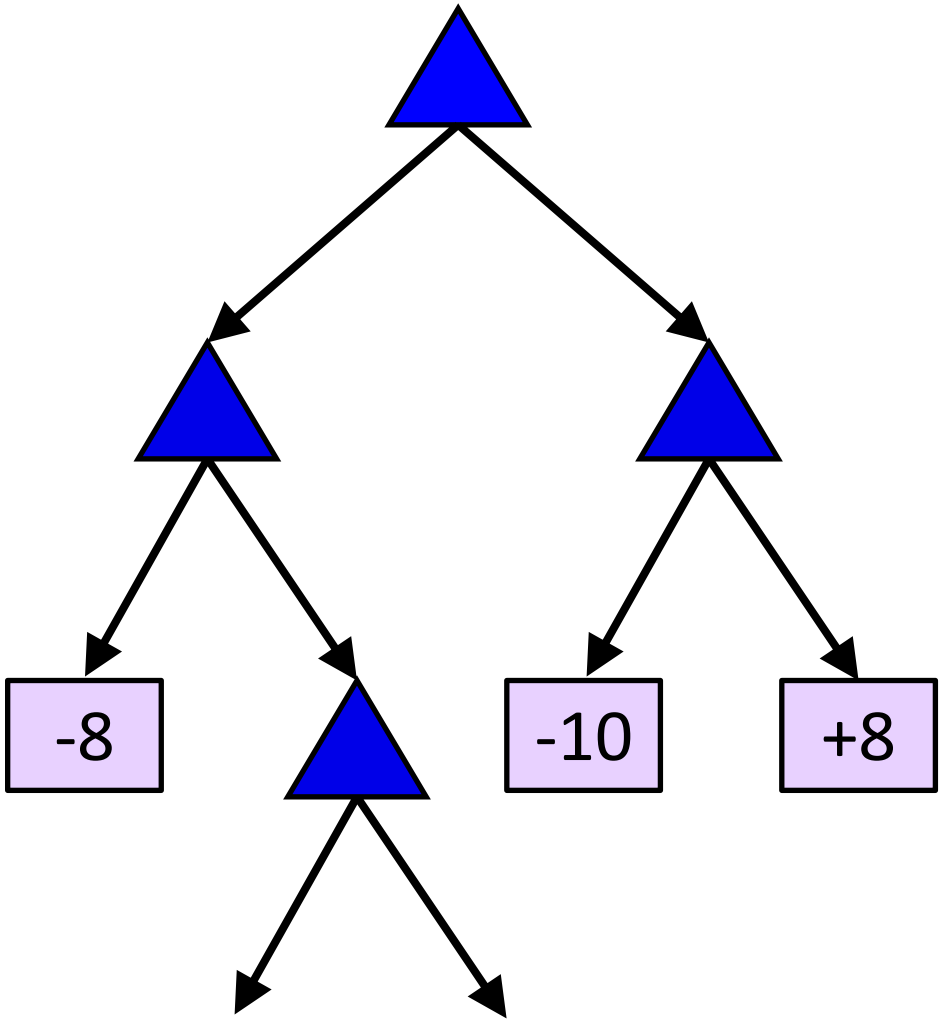
A) 2
B) 3
C) 6
D) 12
E) 14



Minimax Code

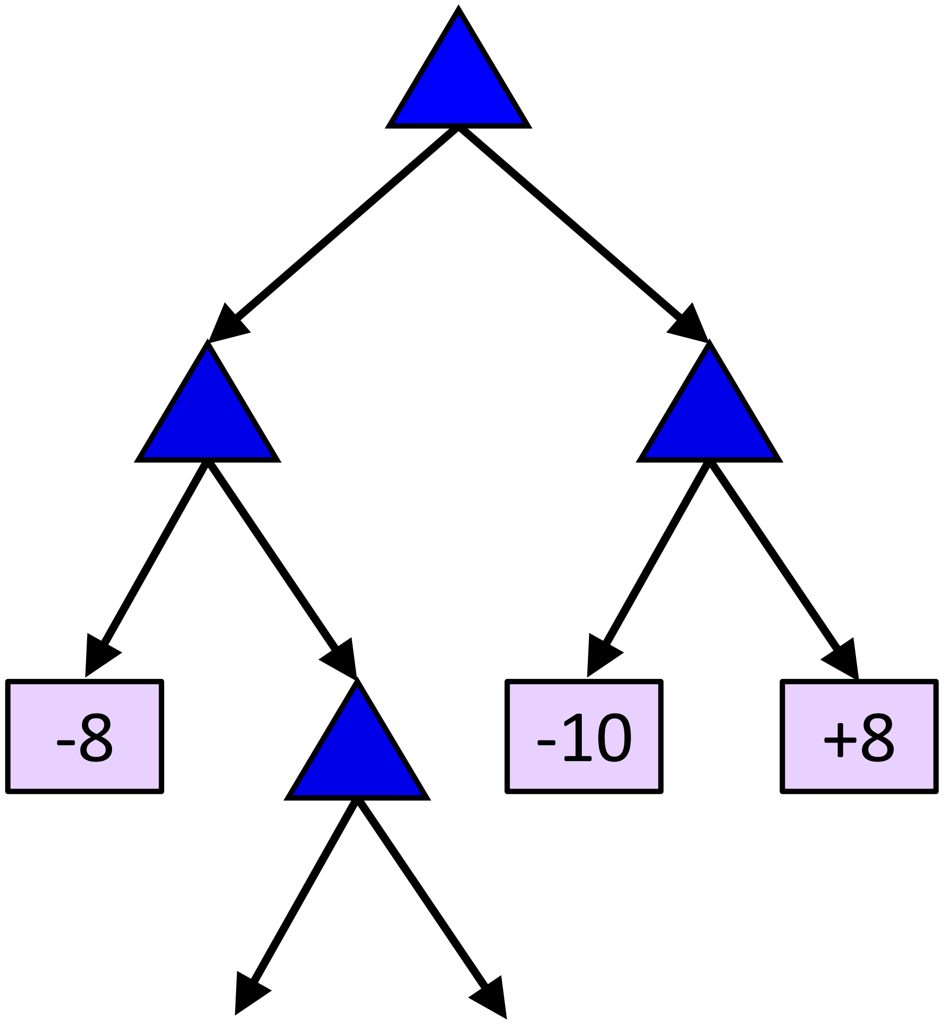


Max Code



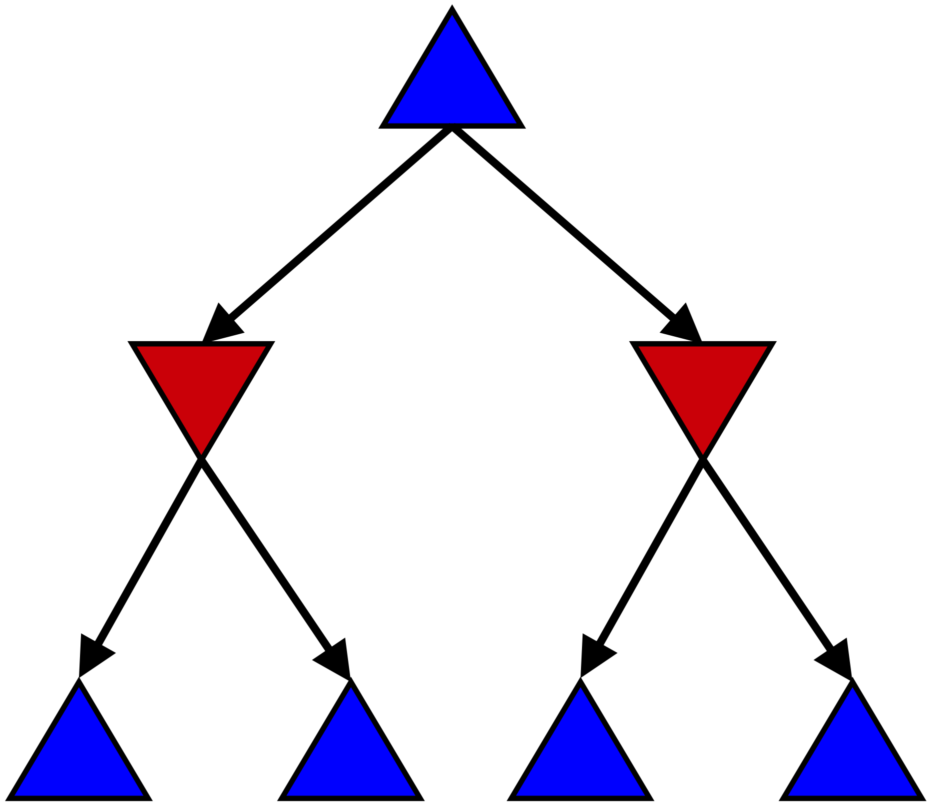
Max Code

```
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 = max_value(next_state)

        if next_value > best_value:
            best_value = next_value

    return best_value
```



Minimax Code

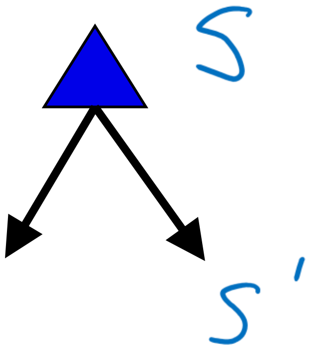
```
def max_value(state):

    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):
```



Minimax Notation

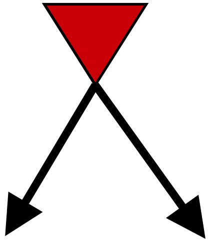
$a \in \{ action(s) \}$

```
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):
```

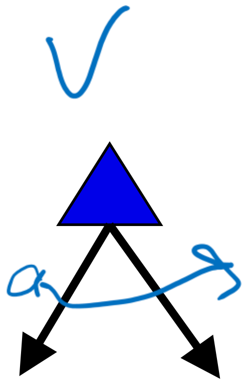


$$V(s) = \max_a \boxed{V(s')},$$

where $s' = result(s,a)$



Minimax Notation

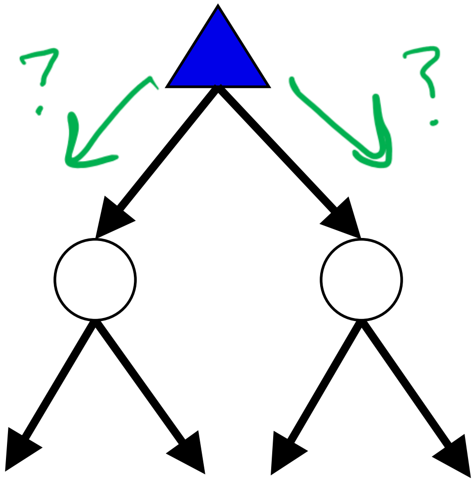


$$V(s) = \max_{a} V(s'),$$

where $s' = result(s,a)$

best_action

action in for loop



$$\hat{a} = \operatorname*{argmax}_{a} V(s'),$$

where $s' = result(s,a)$

Generic Game Tree Pseudocode

```
function  minimax_decision( state )
    return  argmax a in state.actions  value( state.result(a) )

function  value( state )
    if  state.is_leaf
        return state.value

    if  state.player  is  MAX
        return  max a in state.actions  value( state.result(a) )

    if  state.player  is  MIN
        return  min a in state.actions  value( state.result(a) )
```

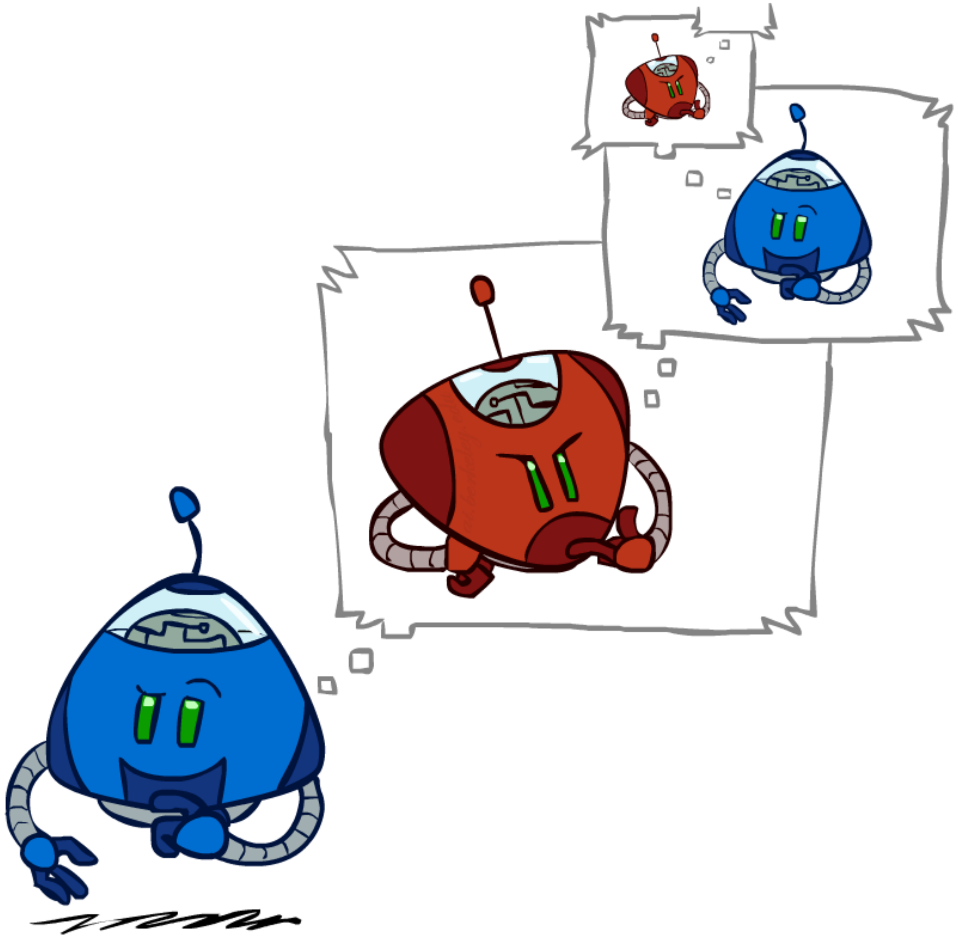
Minimax Efficiency

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?
- Bounded rationality – Herbert Simon



Resource Limits

Resource Limits

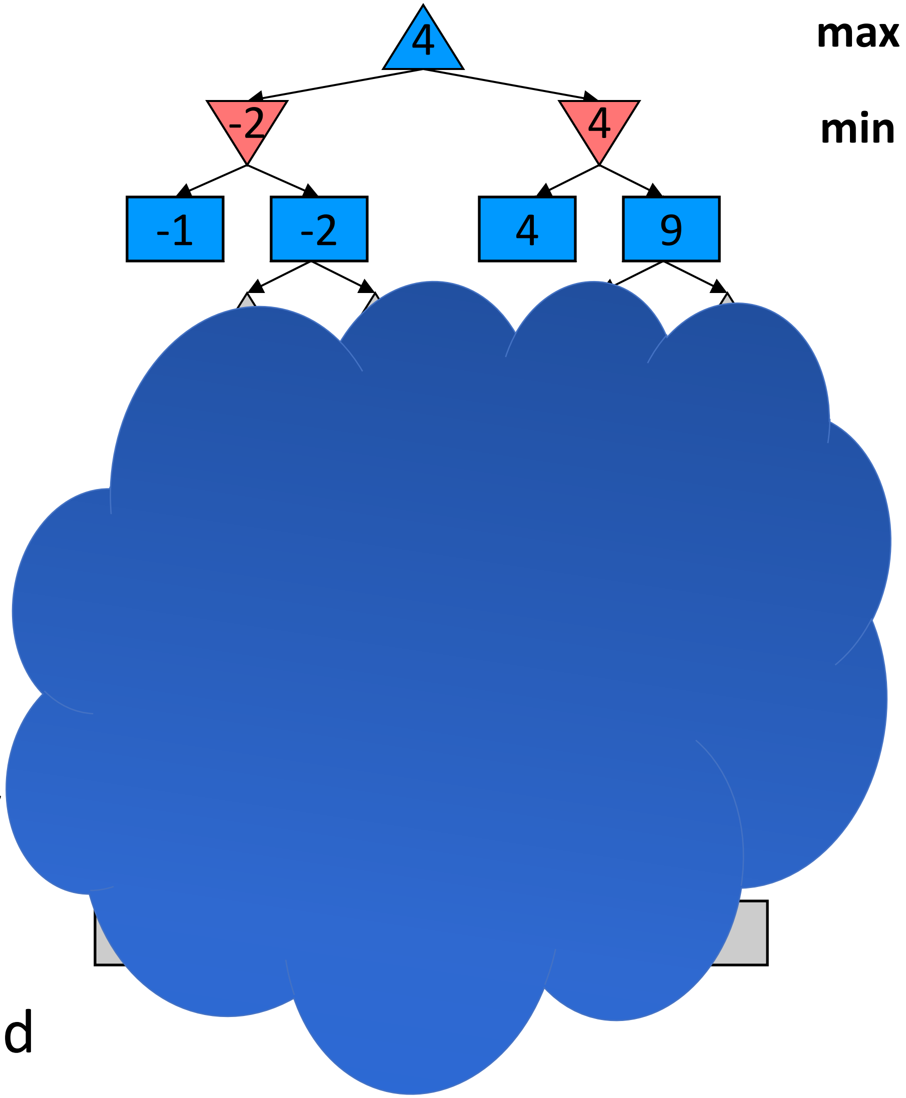
Problem: In realistic games, cannot search to leaves!

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

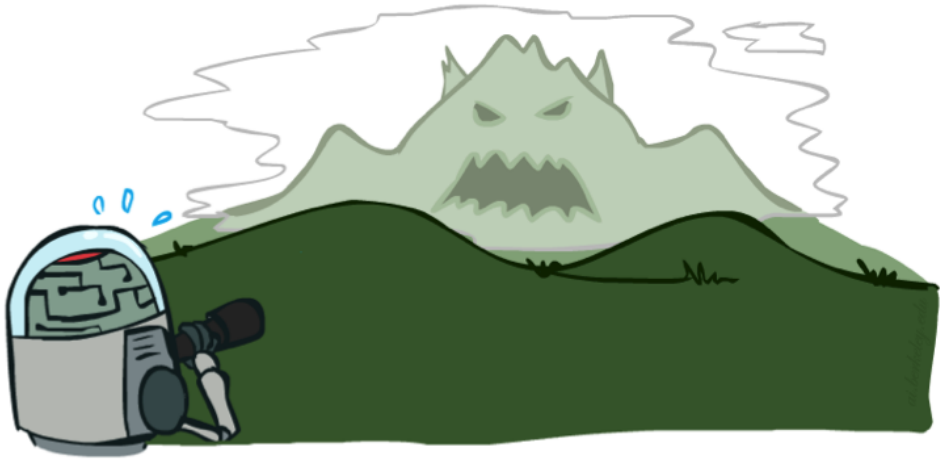


Example:

- 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

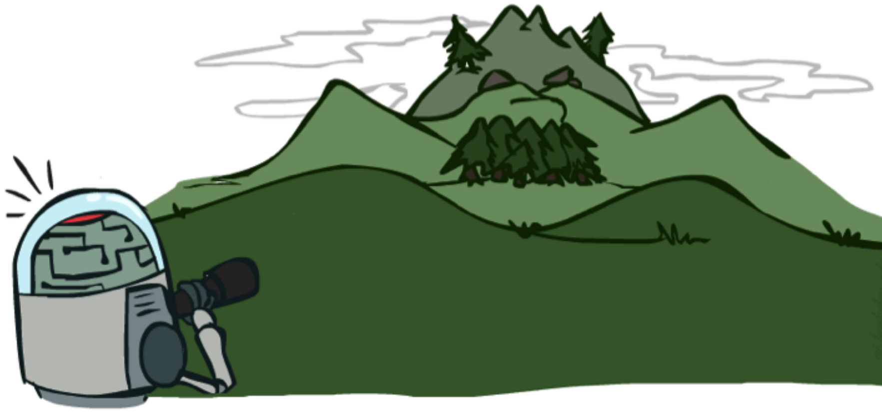
Depth Matters

Evaluation functions are always imperfect



Deeper search => better play (usually)

- Or, deeper search gives same quality of play with a less accurate evaluation function



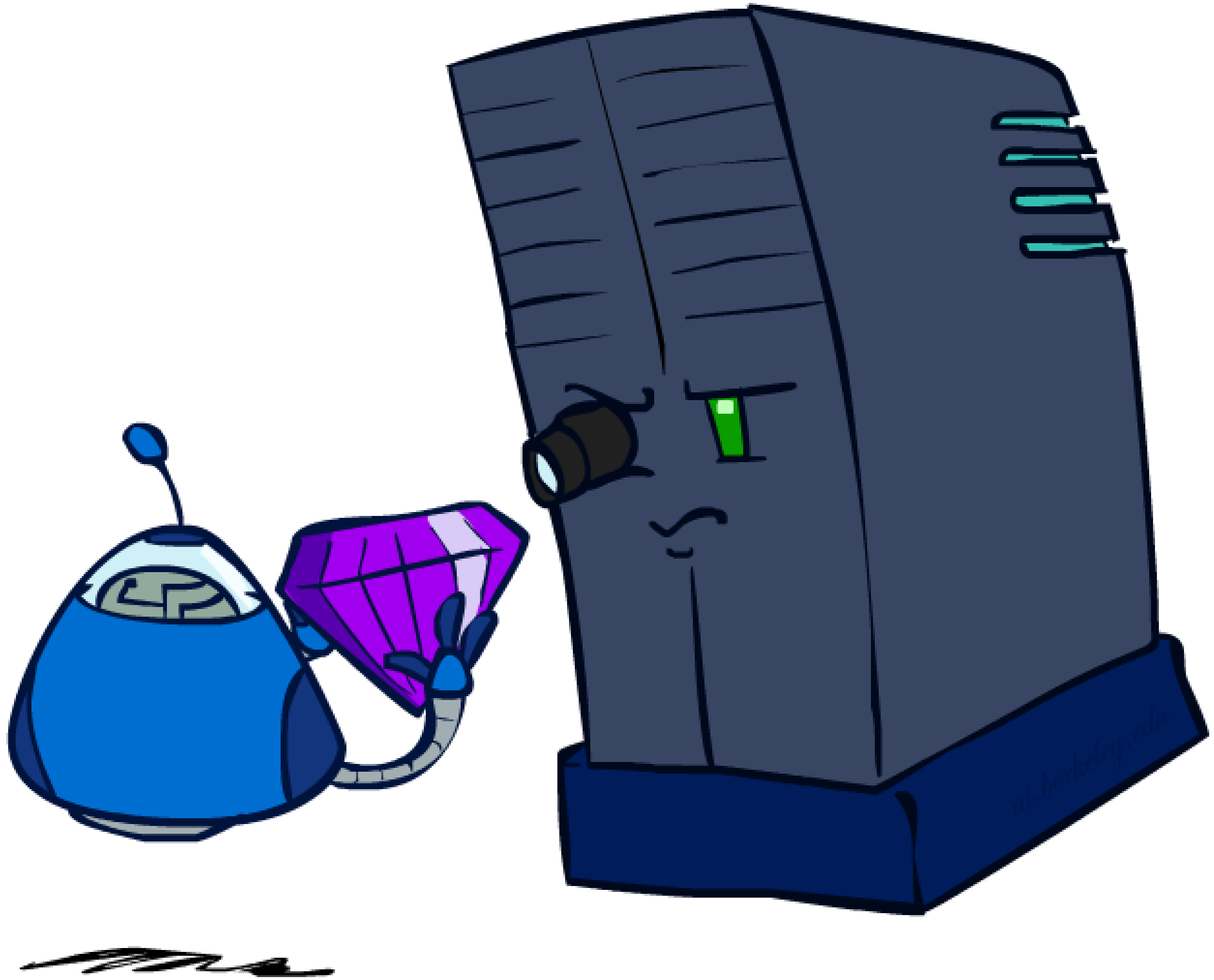
An important example of the tradeoff between complexity of features and complexity of computation

[Demo: depth limited (L6D4, L6D5)]

Demo Limited Depth (2)

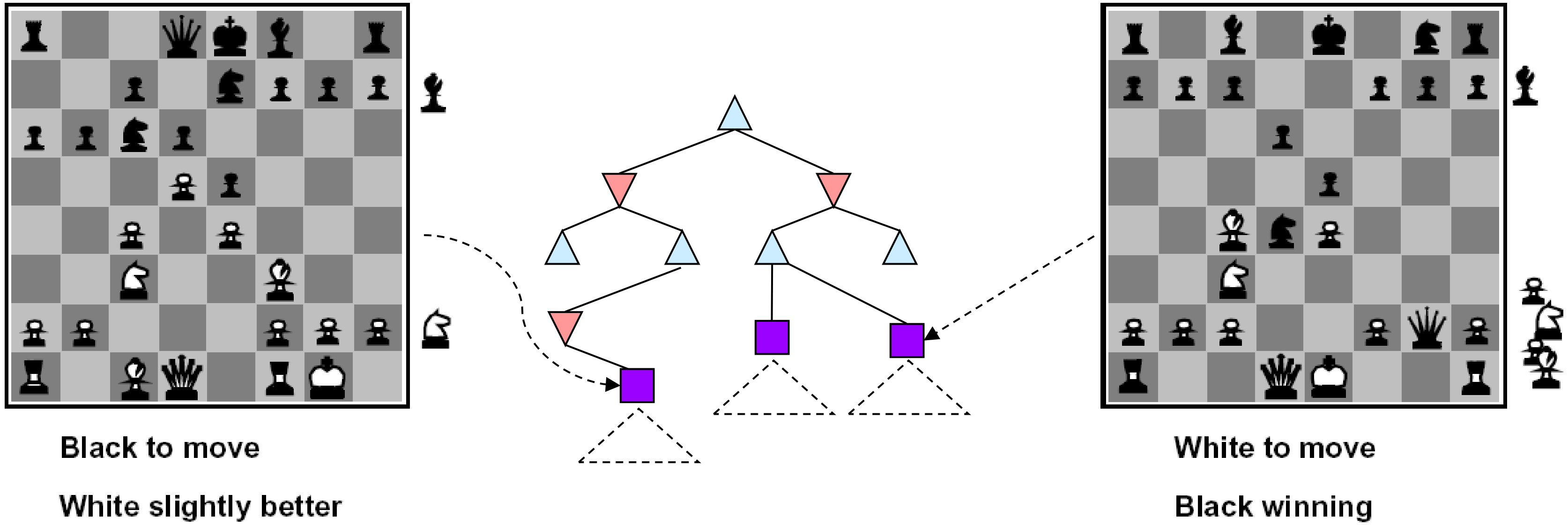
Demo Limited Depth (10)

Evaluation Functions



Evaluation Functions

Evaluation functions score non-terminals in depth-limited search

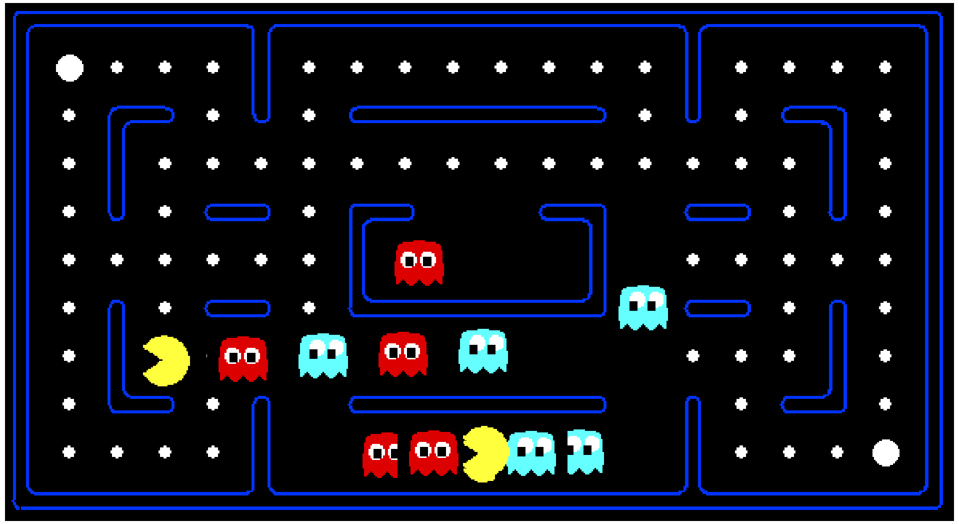


Ideal function: returns the actual minimax value of the position

In practice: typically weighted linear sum of features:

- EVAL(s) = $w_1 f_1(s) + w_2 f_2(s) + \dots + w_n f_n(s)$
- E.g., $w_1 = 9$, $f_1(s)$ = (num white queens – num black queens), etc.

Evaluation for Pacman

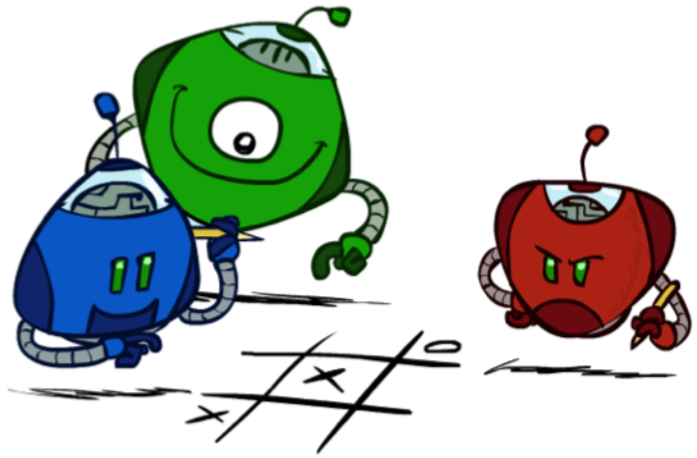


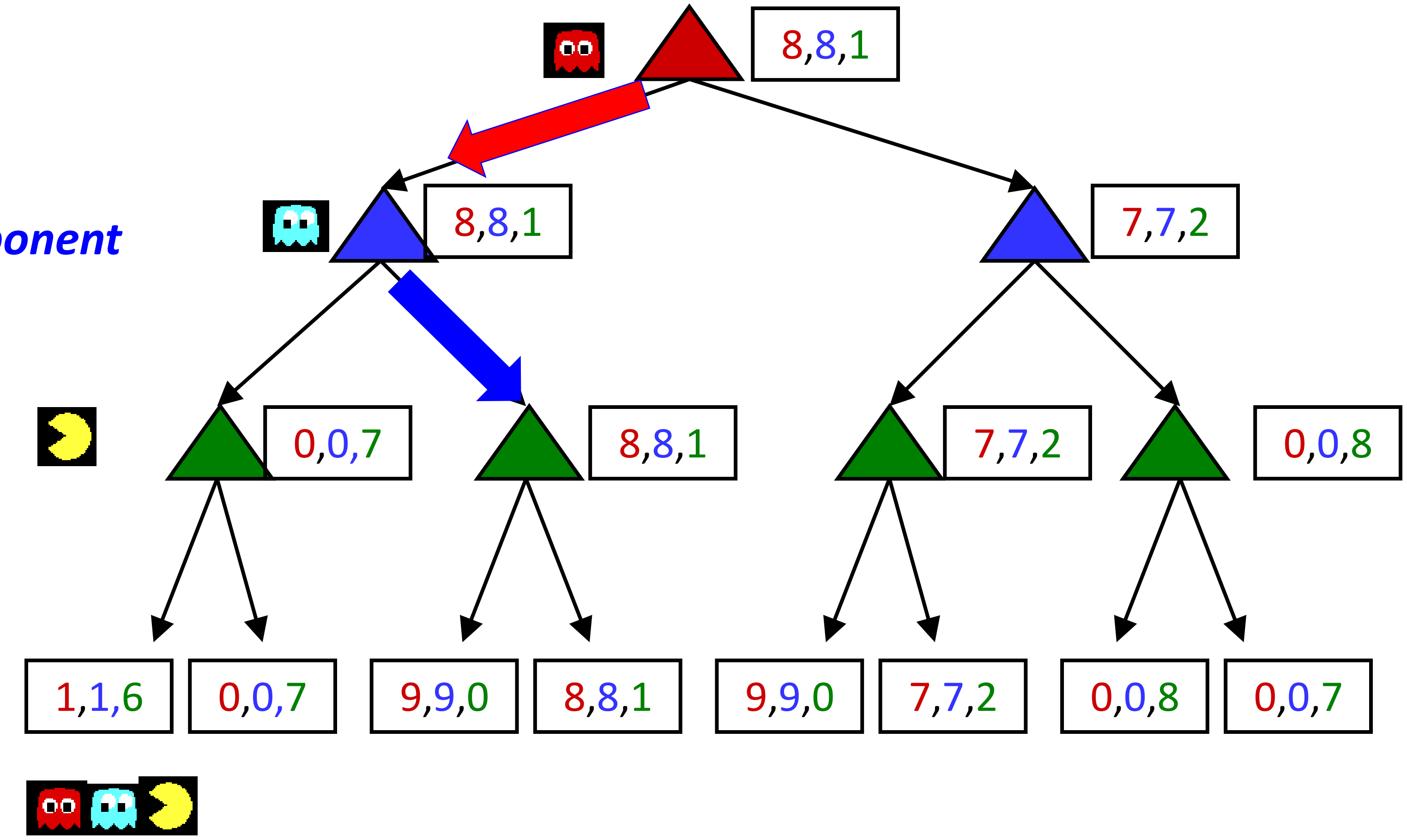
Generalized minimax

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

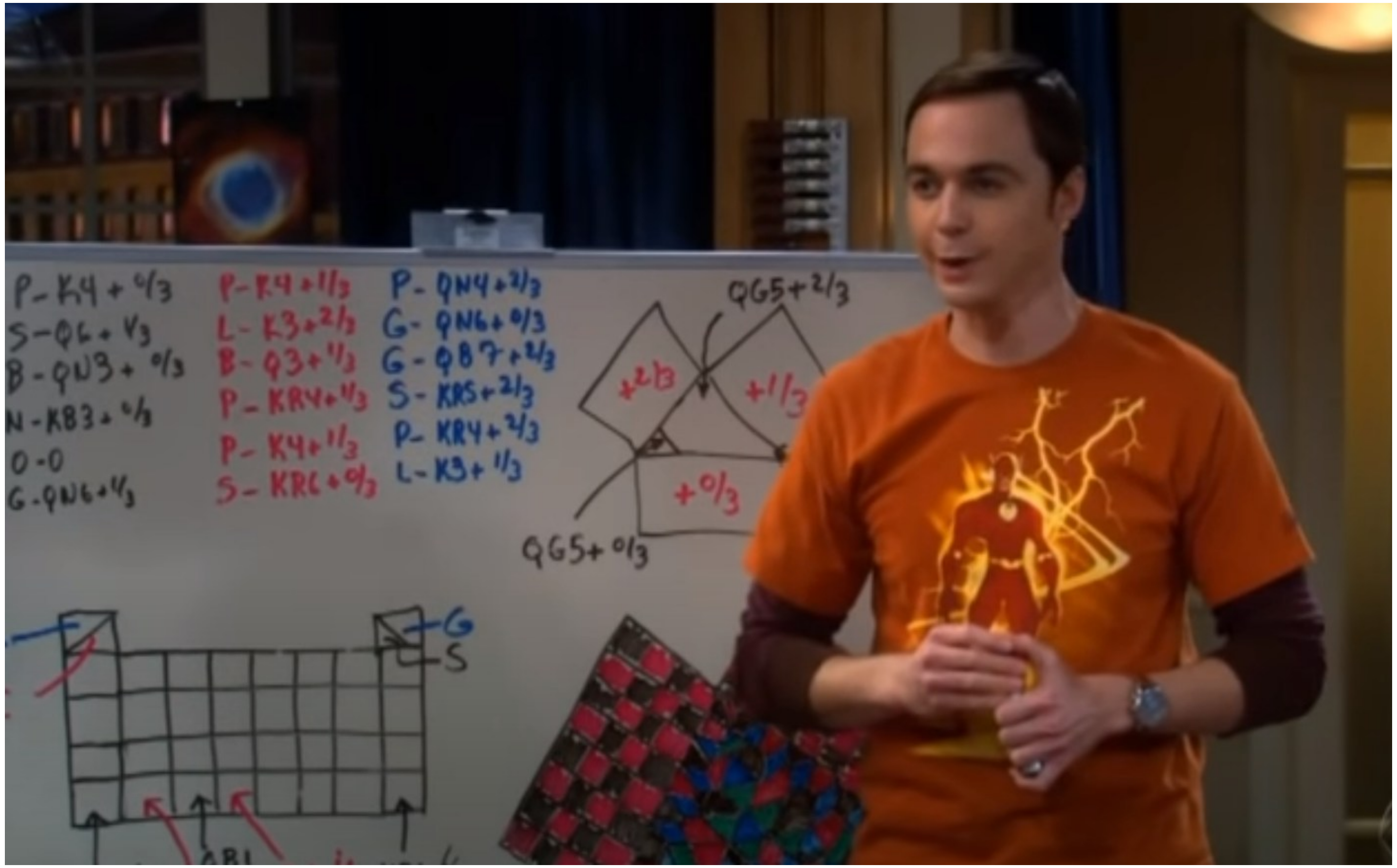
Generalization of minimax:

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





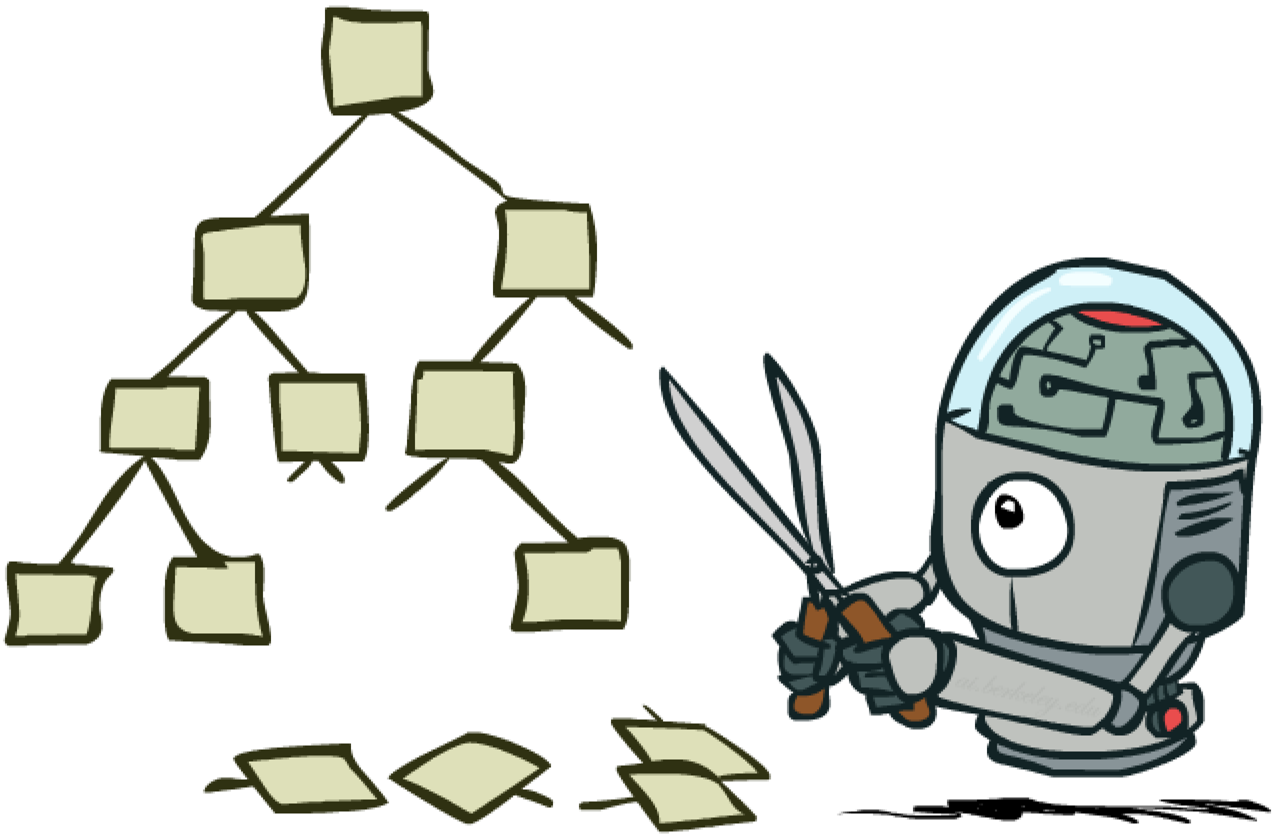
Generalized minimax



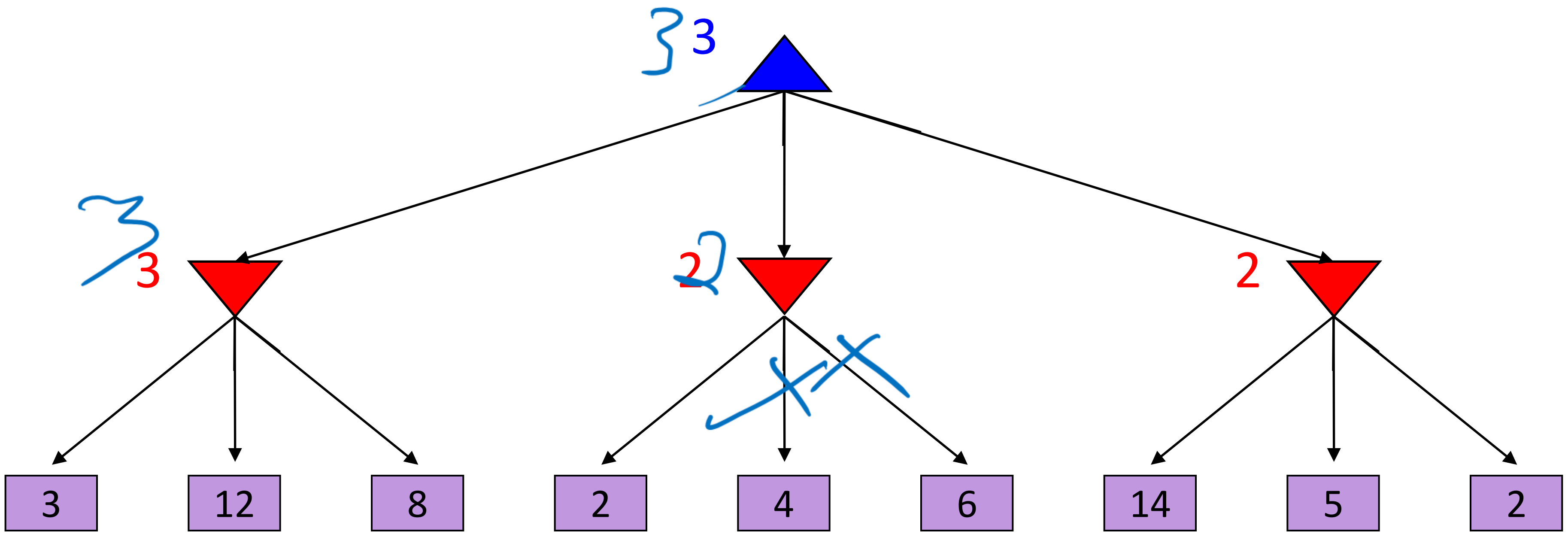
[Three Person Chess](https://www.youtube.com/watch?v=HHVPutfveVs)

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

Game Tree Pruning

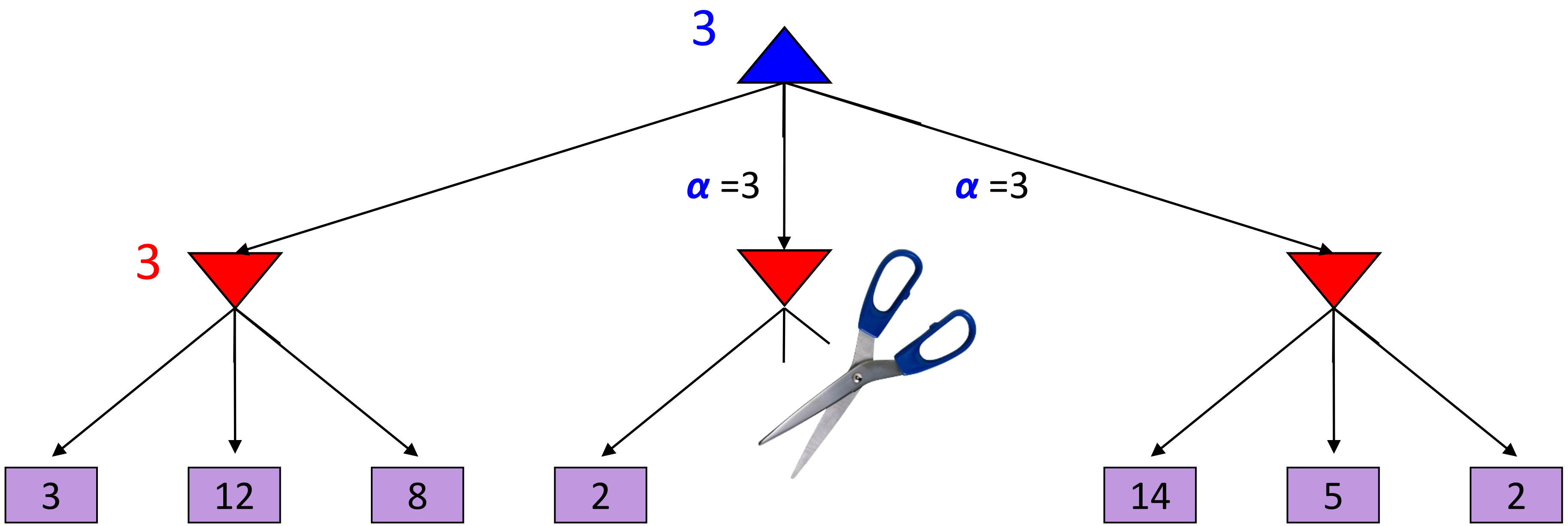


Minimax Example



Alpha-Beta Example

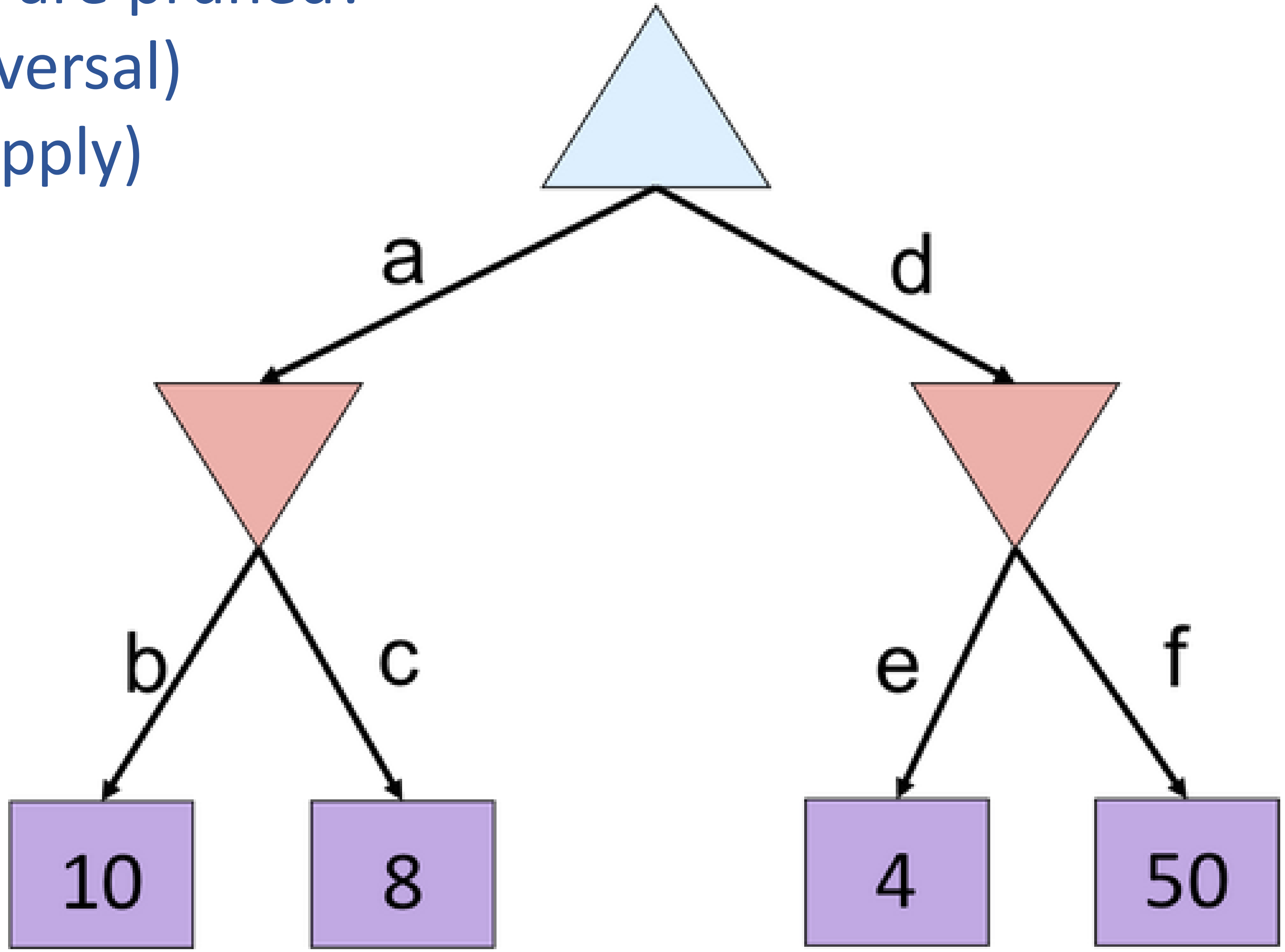
α = best option so far from any MAX node on this path



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

Poll 2

Which branches are pruned?
(Left to right traversal)
(Select all that apply)



Poll 3

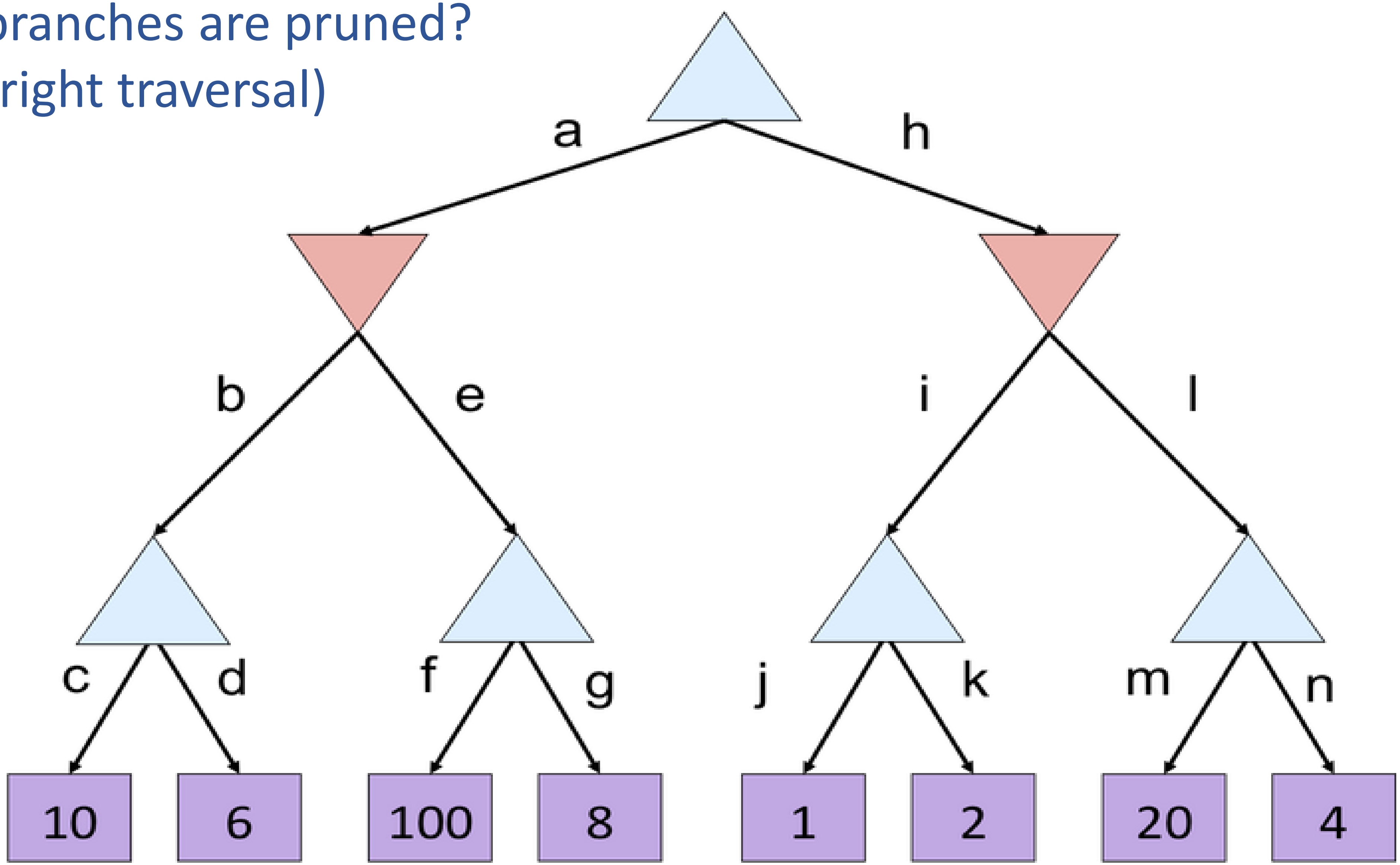
Which branches are pruned?
(Left to right traversal)

A) e, l

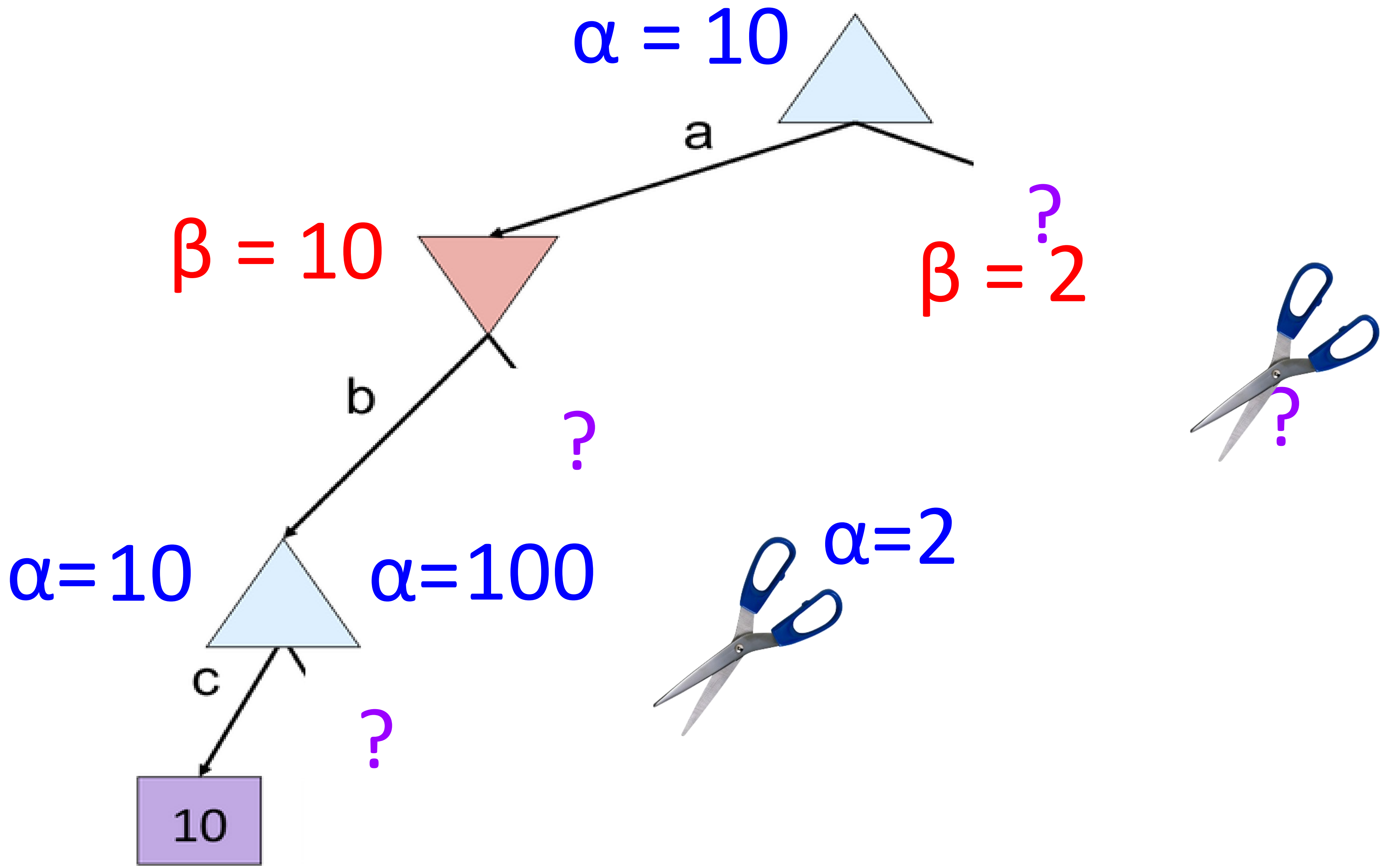
B) g, l

C) g, k, l

D) g, n



Poll 3



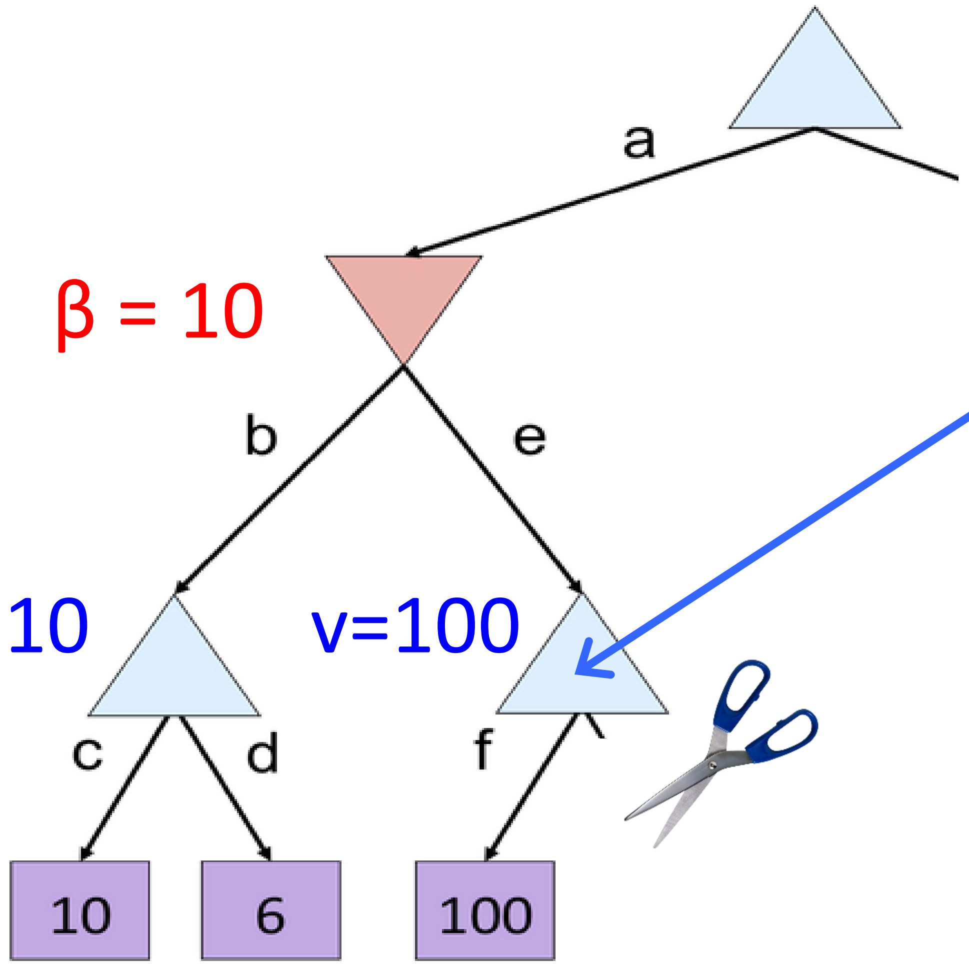
Alpha-Beta Implementation

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
```

Alpha-Beta Poll 3

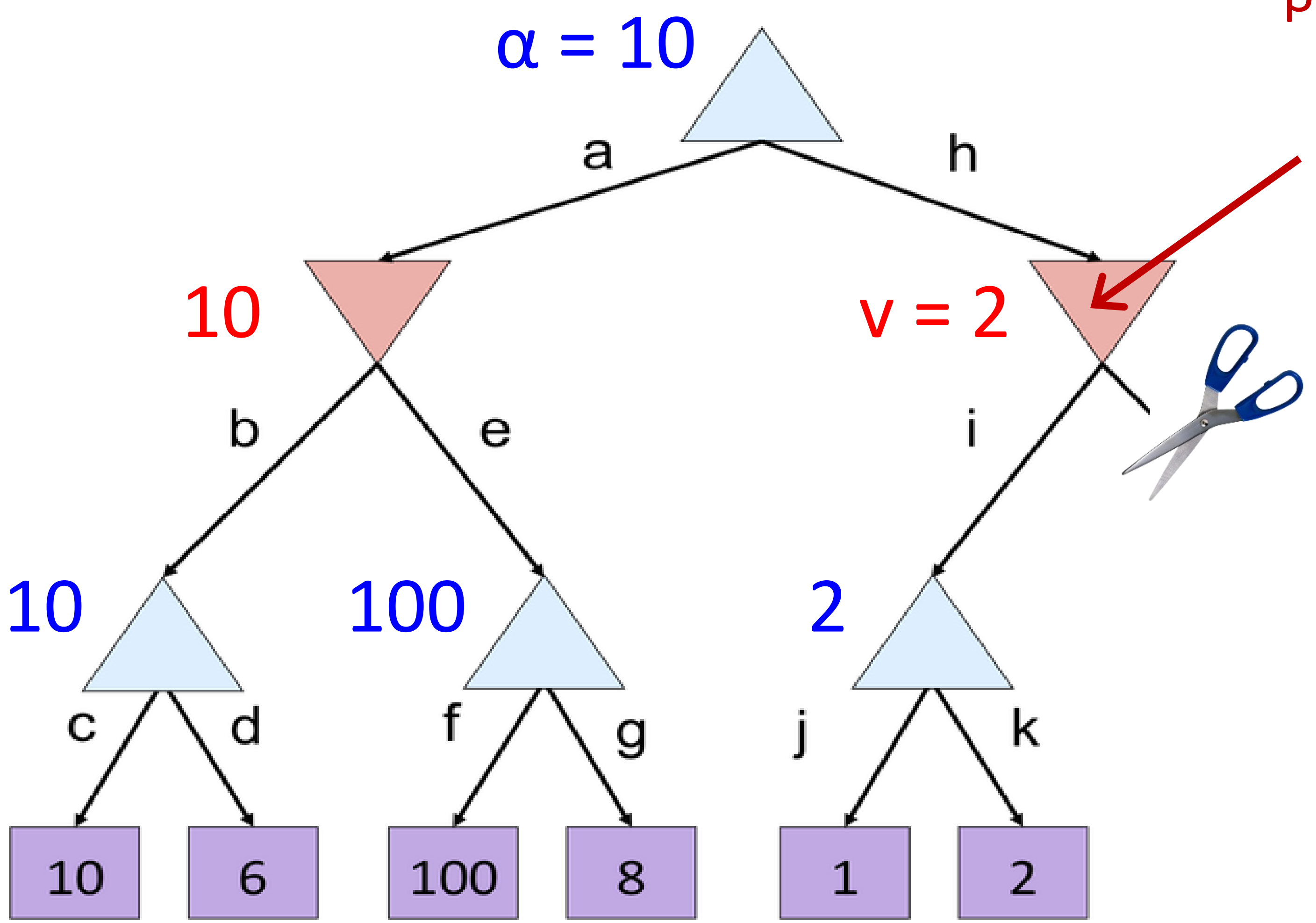


α: MAX's best option on path to root

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

Alpha-Beta Poll 3



α: MAX's best option on path to root

β: MIN's best option on path to root

```
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 Pruning Properties

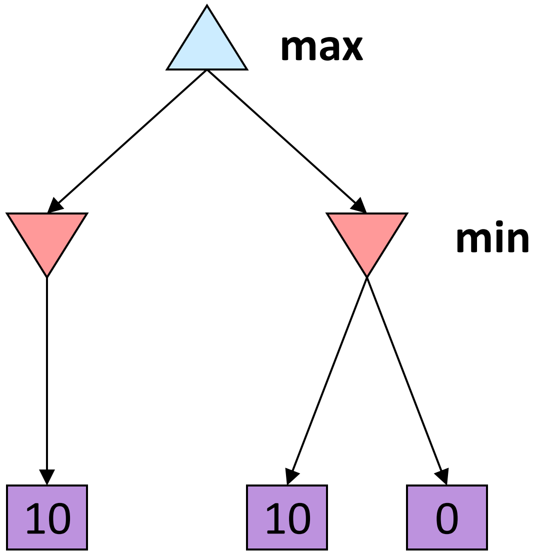
Theorem: This pruning has ***no effect*** on minimax value computed for the root!

Good child ordering improves effectiveness of pruning

- Iterative deepening helps with this

With "perfect ordering":

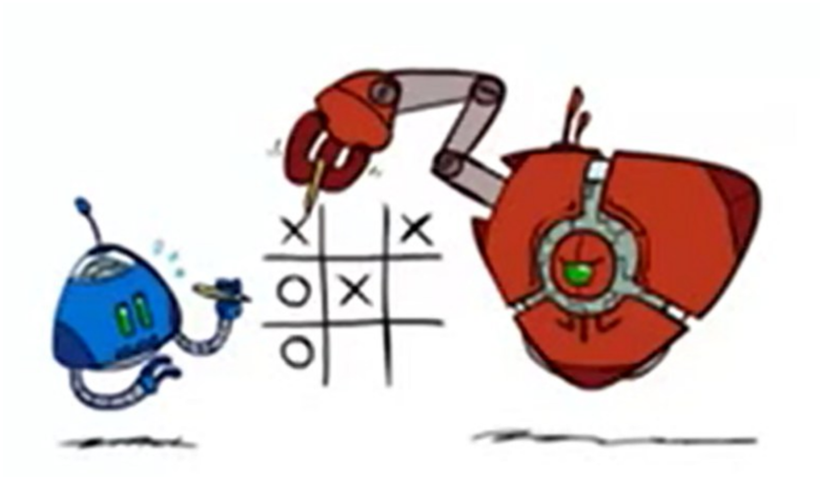
- Time complexity drops to $O(b^{m/2})$
- Doubles solvable depth!
- 1M nodes/move => depth=8, respectable

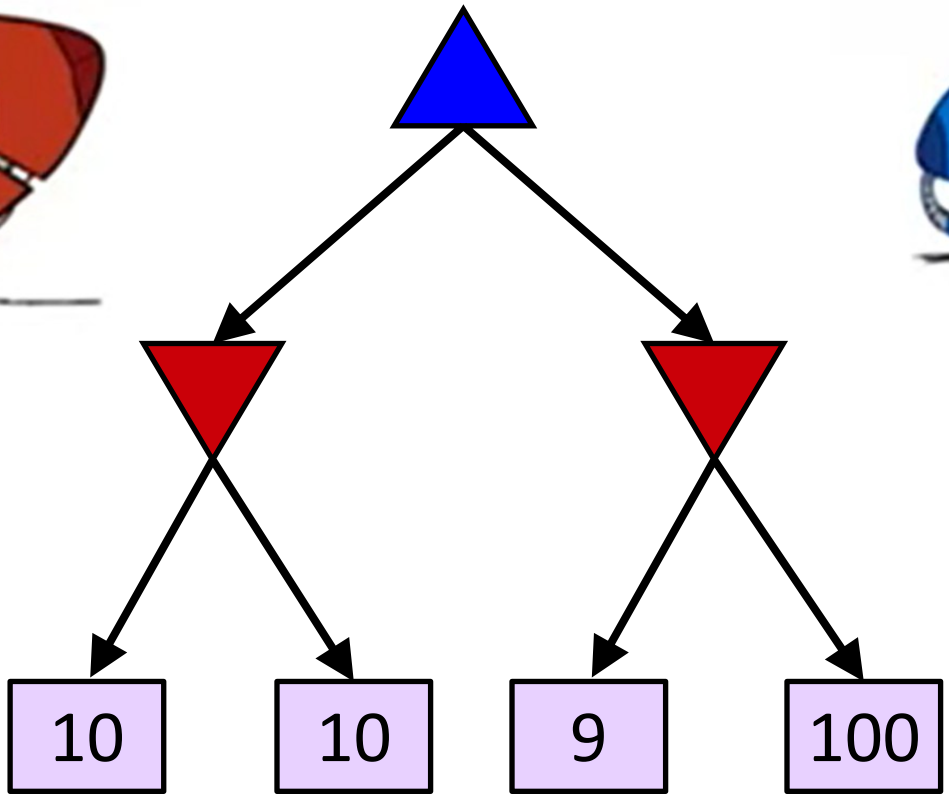


This is a simple example of metareasoning (computing about what to compute)

Modeling Assumptions

Know your opponent





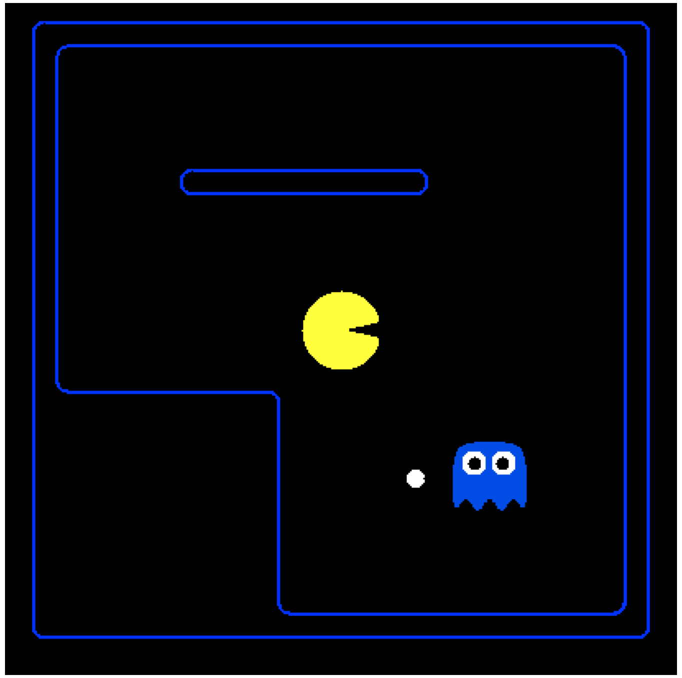
Poll 4

How well would a minimax Pacman perform against a ghost that moves randomly?

A. Better than against a minimax ghost

B. Worse than against a minimax ghost

C. Same as against a minimax ghost



Fine print

- Pacman: uses depth 4 minimax as before
- Ghost: moves randomly

[Demo: L7D6)]

Modeling Assumptions

Minimax autonomous vehicle?

Image: https://corporate.ford.com/innovation/autonomous-2021.html

Minimax Driver?

https://youtu.be/5PRrwlkPdNI?t=52

Clip: How I Met Your Mother, CBS

Modeling Assumptions

Dangerous Pessimism

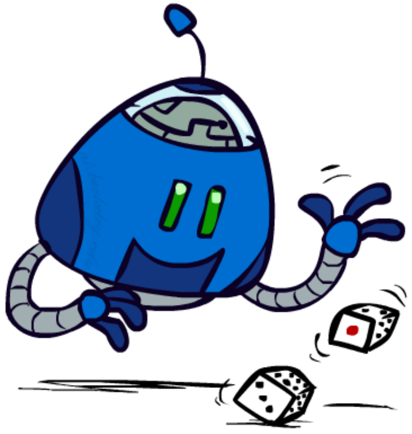
Assuming the worst case when it's not likely

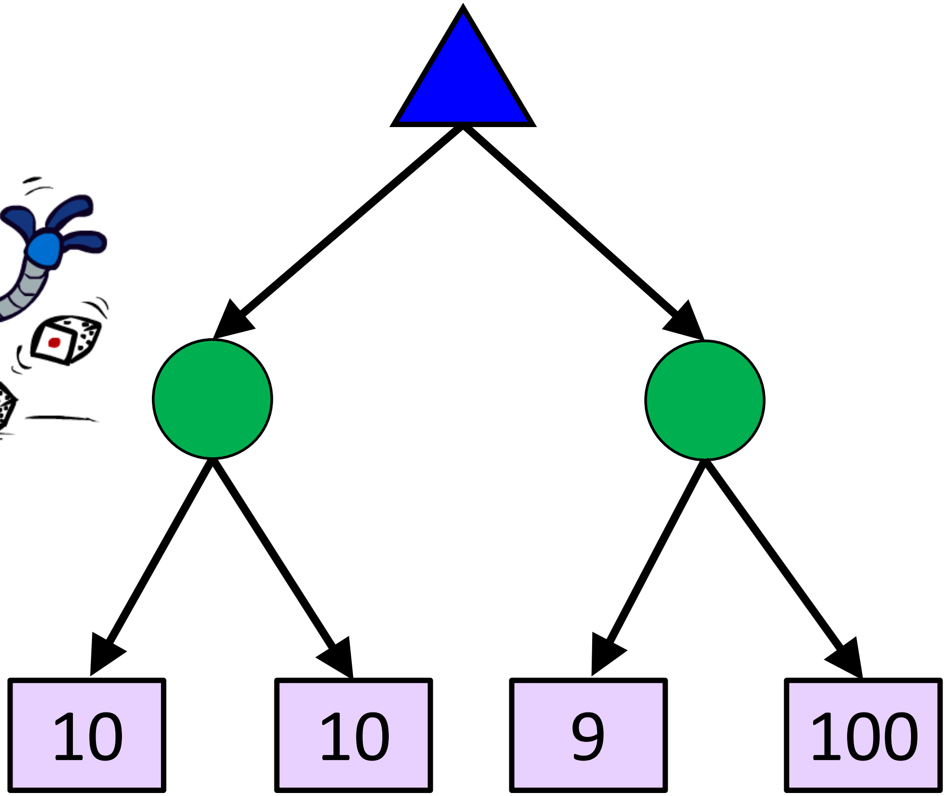
Dangerous Optimism

Assuming chance when the world is adversarial

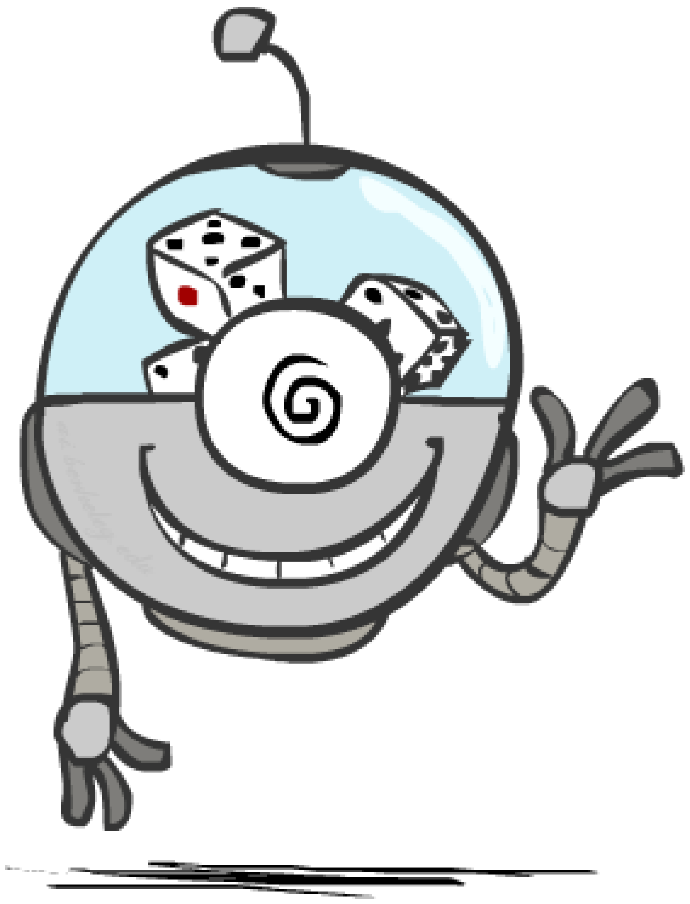
Modeling Assumptions

Chance nodes: Expectimax





Probabilities



Probabilities

A random variable represents an event whose outcome is unknown

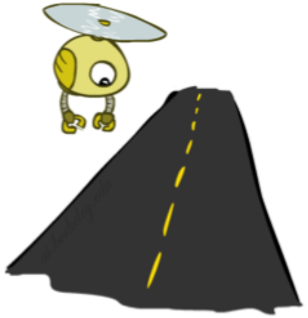
A probability distribution is an assignment of weights to outcomes

Example: Traffic on freeway

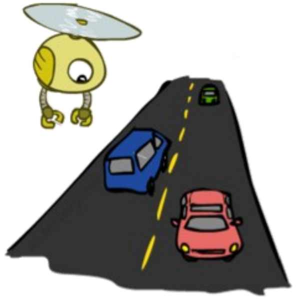
- Random variable: T = whether there's traffic
- Outcomes: T in {none, light, heavy}
- Distribution:

  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

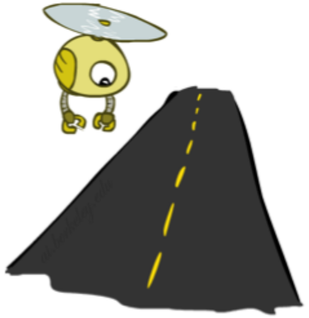
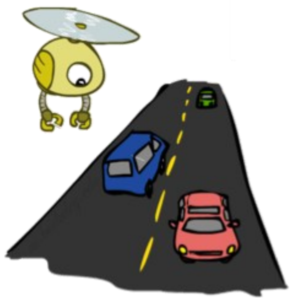
Expected Value

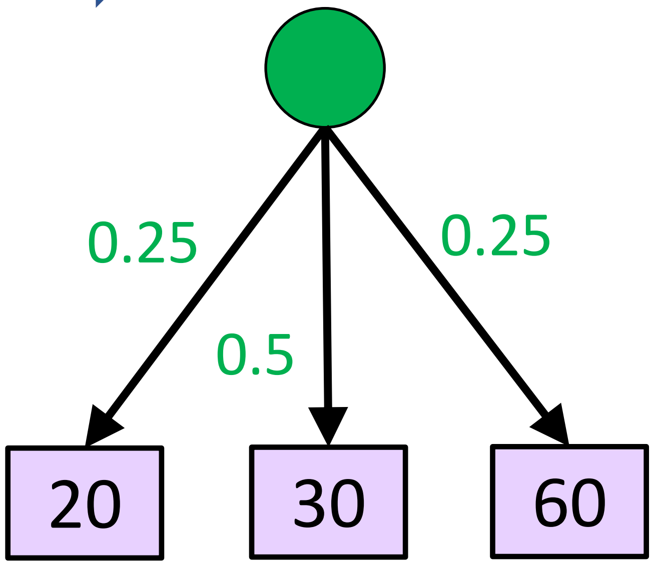
Expected value of a function of a random variable:

Average the values of each outcome,
weighted by the probability of that outcome

Example: How long to get to the airport?

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

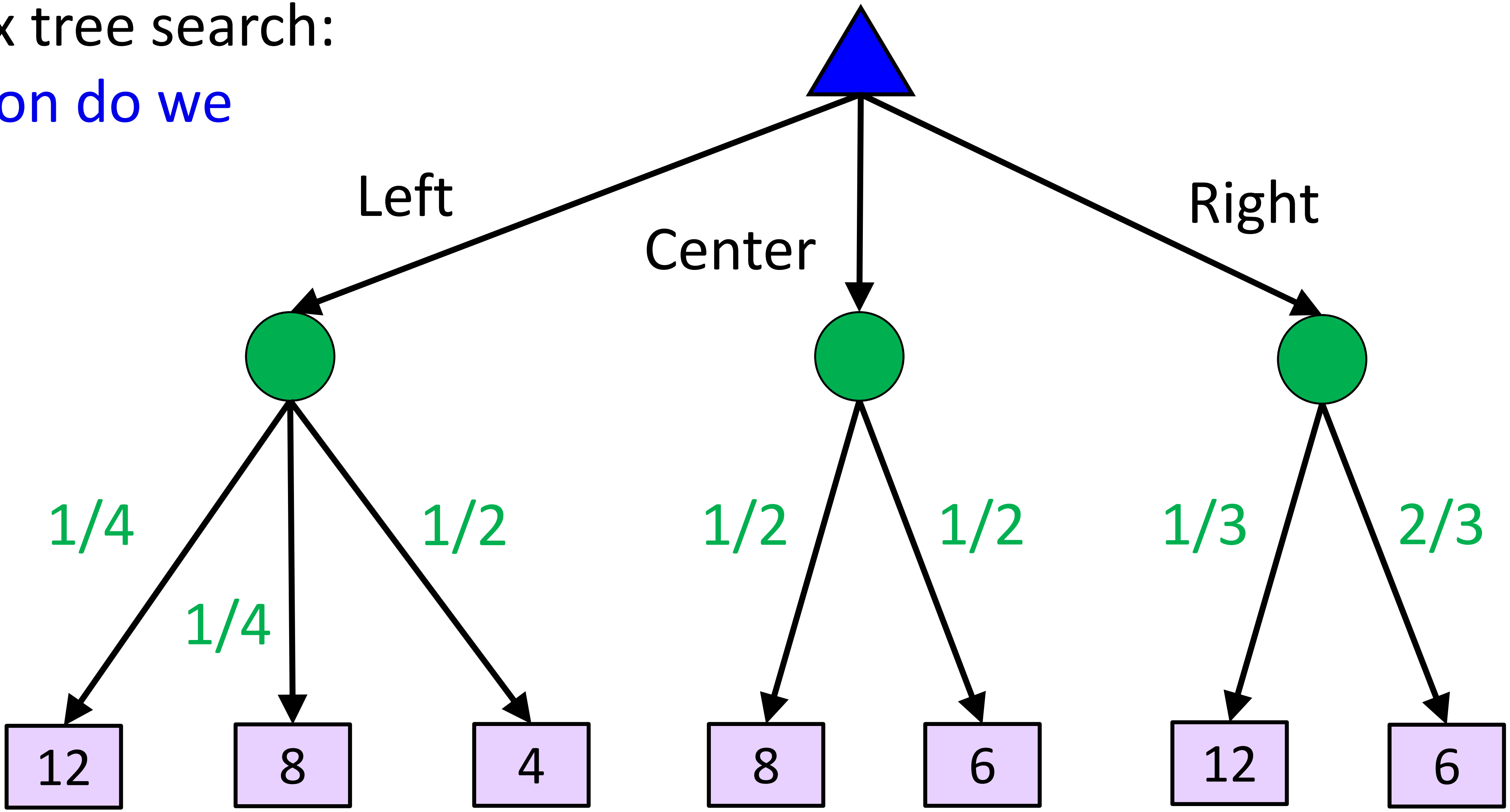


Poll 5

Expectimax tree search:

Which action do we choose?

A: Left
B: Center
C: Right
D: Eight



Poll 5

Expectimax tree search:

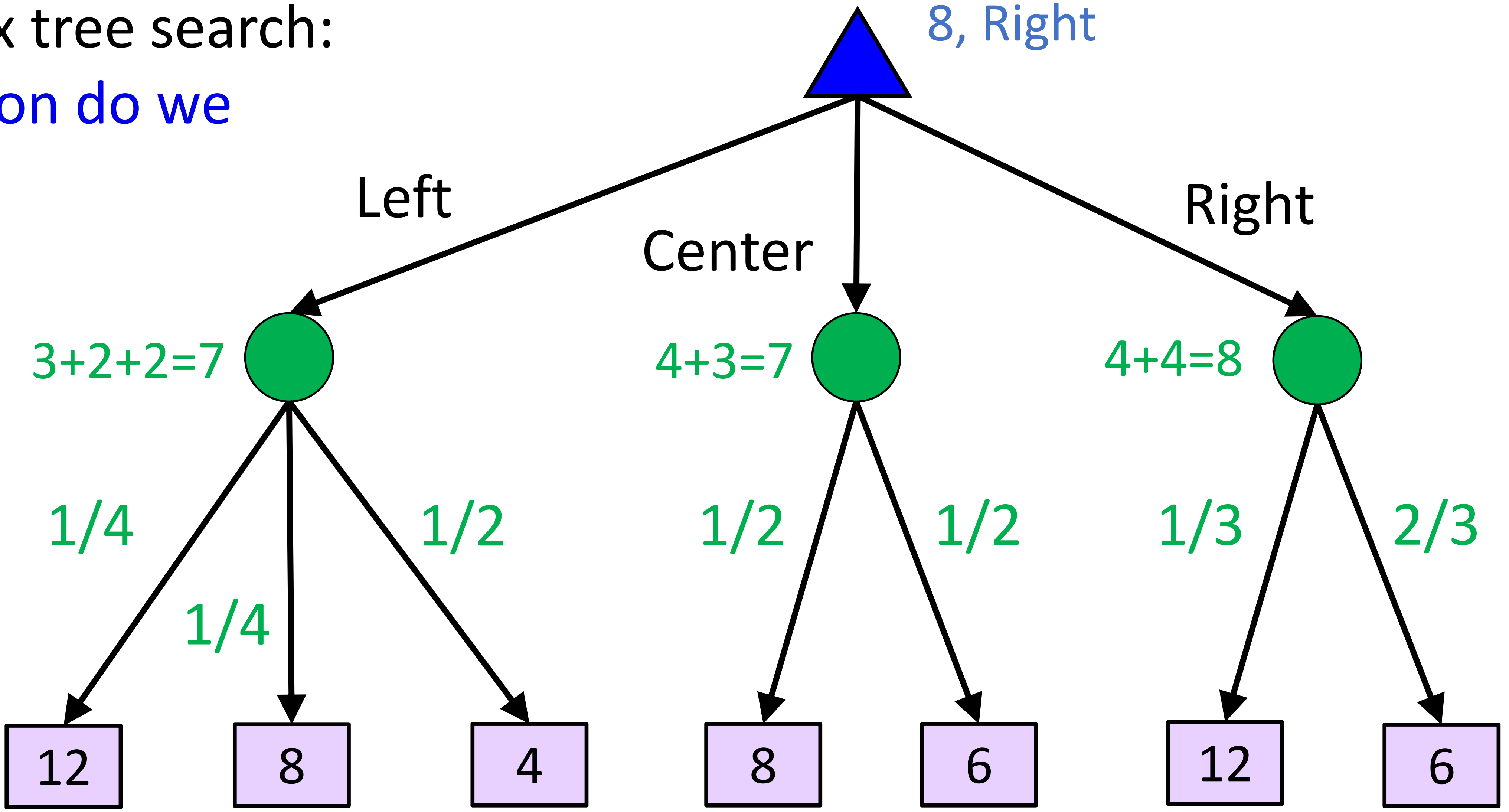
Which action do we choose?

A: Left

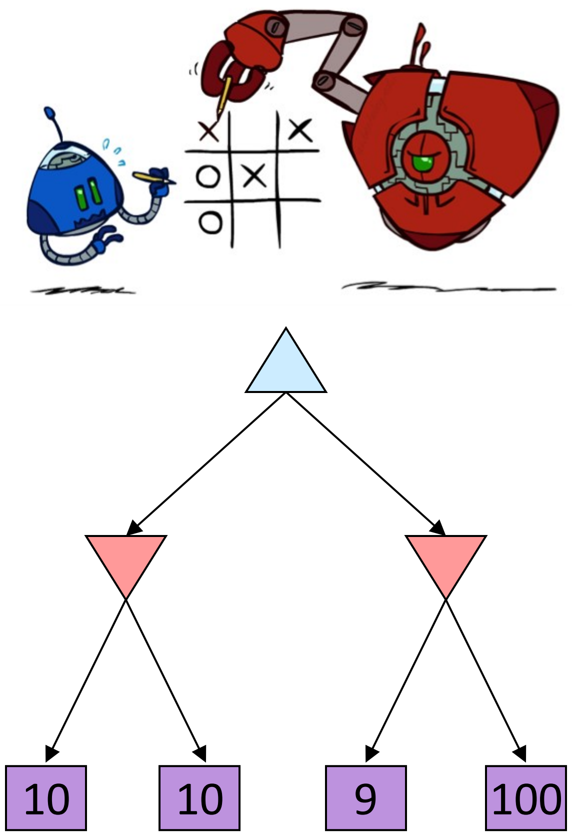
B: Center

C: Right

D: Eight

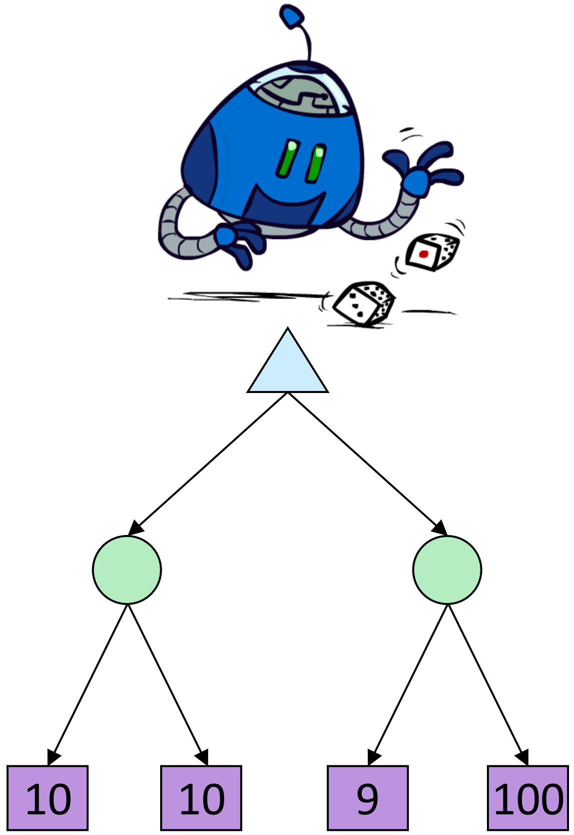


Chance outcomes in trees



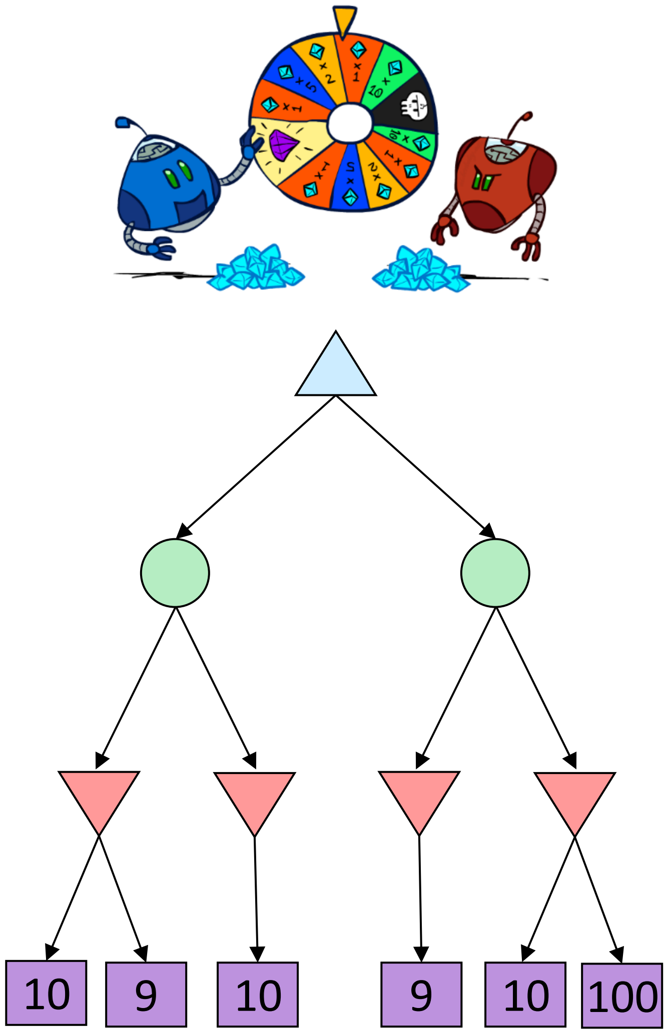
Tictactoe, chess

Minimax



Tetris, investing

Expectimax



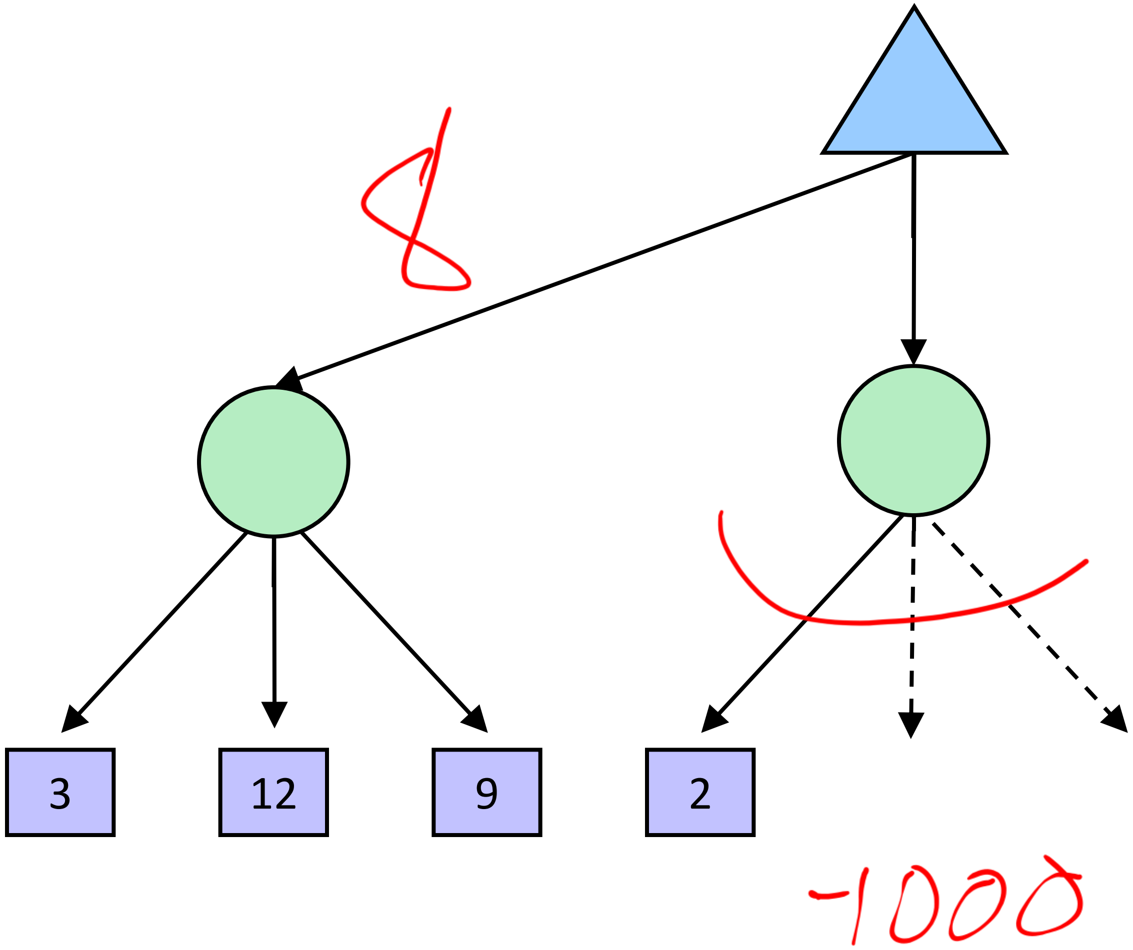
Backgammon, Monopoly

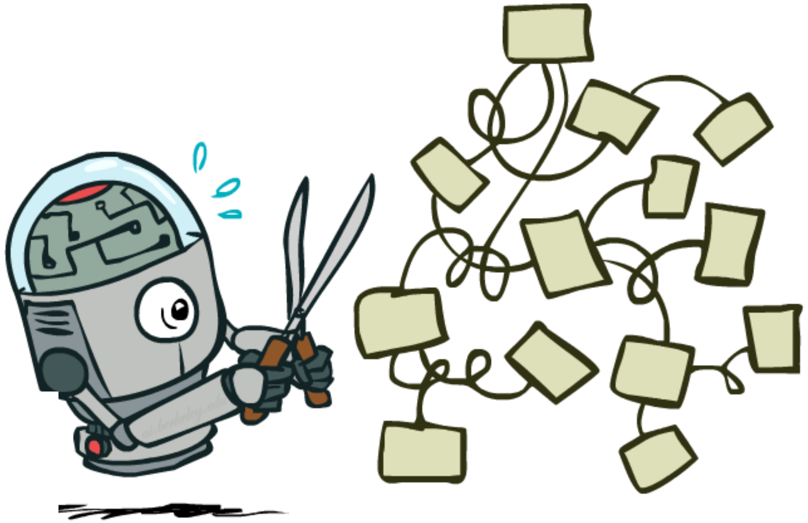
Expectiminimax

Expectimax Code

```
function  value( state )
    if  state.is_leaf
        return  state.value

    if  state.player  is  MAX
        return  max a in state.actions  value( state.result(a) )

    if  state.player  is  MIN
        return  min a in state.actions  value( state.result(a) )

    if  state.player  is  CHANCE
        return  sum s in state.next_states  P( s ) * value( s )
```

Expectimax Pruning?



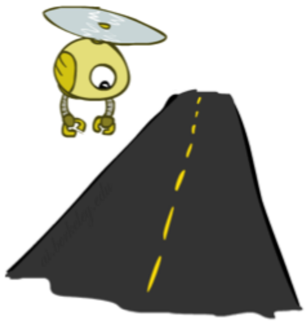
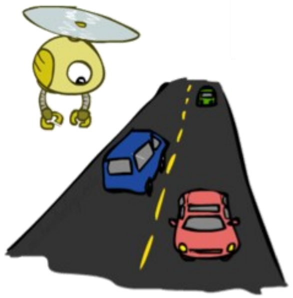


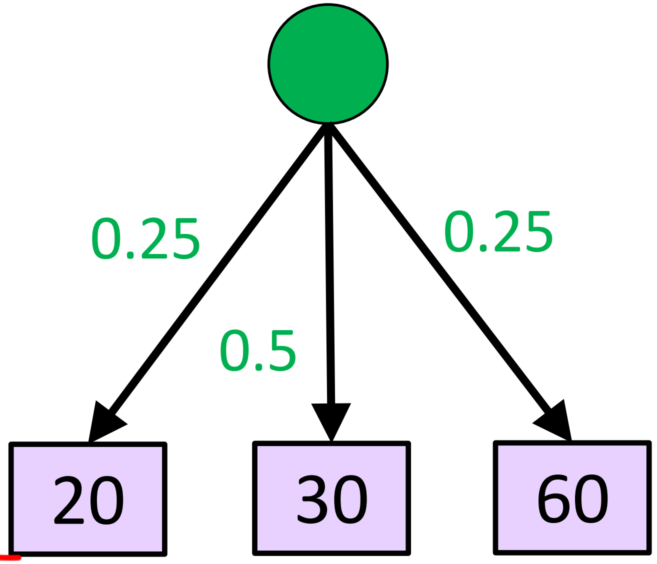
Expectimax Notation

Time: 20 min × + 30 min × + 60 min ×

Probability: 0.25 0.50 0.25

$$20 \times 0.25 + 30 \times 0.50 + 60 \times 0.25$$

Max node notation

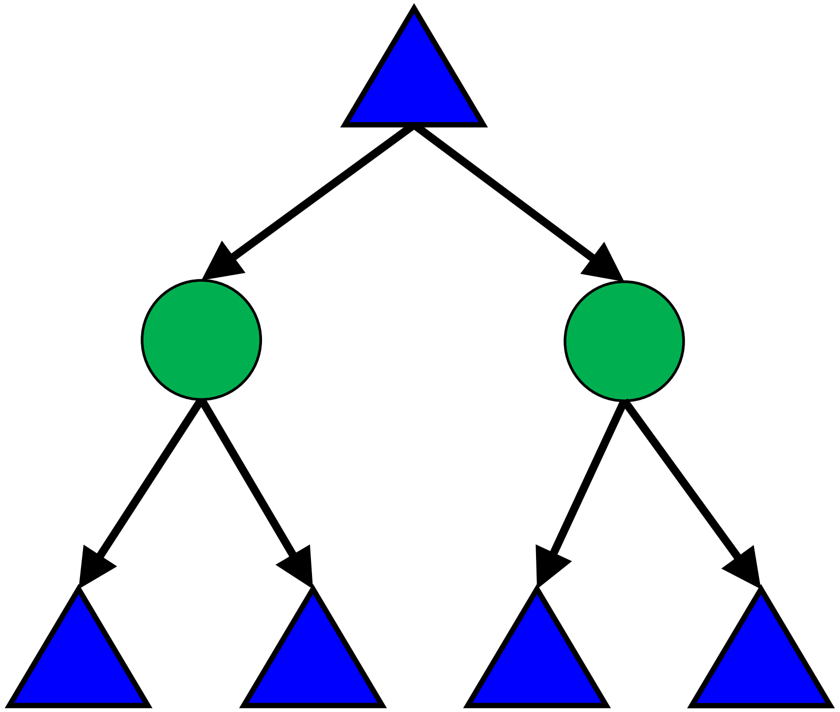
$$V(s) = \max_{a} V(s'),$$

where $s' = result(s, a)$

Chance node notation

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

Preview: MDP/Reinforcement Learning Notation



$$V(s) = \max_a \sum_{s'} \left[P(s')\, V(s') \right]$$

Preview: MDP/Reinforcement Learning Notation

Standard expectimax: $V(s) = \max_a \sum_{s'} P(s'|s,a)V(s')$

Bellman equations: $V(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$

Value iteration: $V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')], \quad \forall\, s$

Q-iteration: $Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')], \quad \forall\, s,a$

Policy extraction: $\pi_V(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')], \quad \forall\, s$

Policy evaluation: $V_{k+1}^{\pi}(s) = \sum_{s'} P(s'|s,\pi(s))[R(s,\pi(s),s') + \gamma V_k^{\pi}(s')], \quad \forall\, s$

Policy improvement: $\pi_{new}(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_{old}}(s')], \quad \forall\, s$

Preview: MDP/Reinforcement Learning Notation

Standard expectimax: $V(s) = \max_a \sum_{s'} P(s'|s,a)V(s')$

Bellman equations: $V(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')]$

Value iteration: $V_{k+1}(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V_k(s')], \quad \forall\, s$

Q-iteration: $Q_{k+1}(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} Q_k(s',a')], \quad \forall\, s,a$

Policy extraction: $\pi_V(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V(s')], \quad \forall\, s$

Policy evaluation: $V^{\pi}_{k+1}(s) = \sum_{s'} P(s'|s,\pi(s))[R(s,\pi(s),s') + \gamma V^{\pi}_k(s')], \quad \forall\, s$

Policy improvement: $\pi_{new}(s) = \operatorname{argmax}_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma V^{\pi_{old}}(s')], \quad \forall\, s$