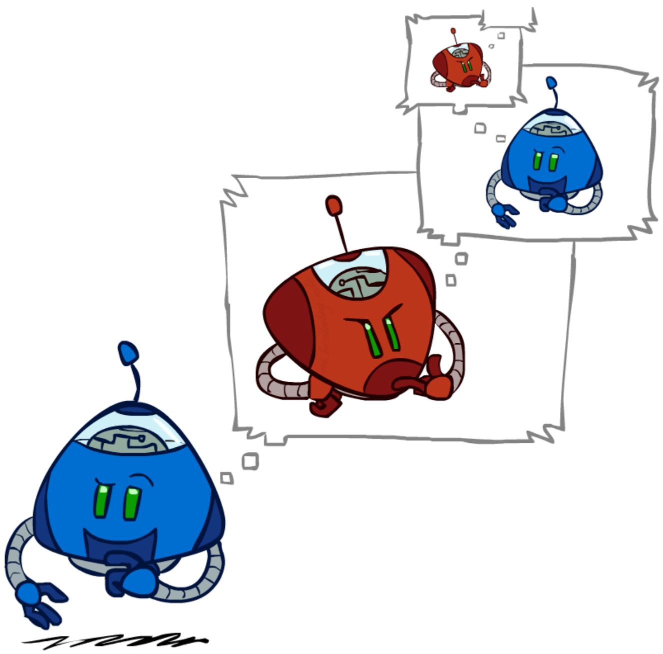
Adversarial Search

Lecture 6

How can we use search to plan ahead when other agents are planning against us?

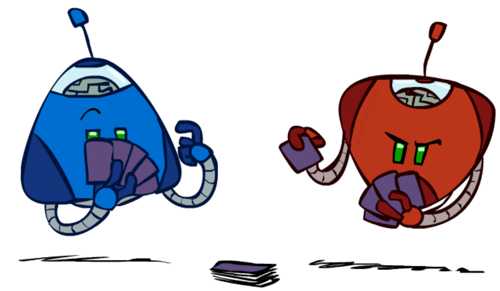
Agenda

- Games: context, history
- Searching via Minimax
- Scaling
  - $\alpha-\beta$ pruning
  - Depth-limiting
  - Evaluation functions
- Handling uncertainty with Expectiminimax



Characterizing Games

- There are many kinds of games, and several ways to classify them
  - Deterministic vs. stochastic
  - [Im]perfect information
  - One, two, multi-player
  - Utility (how agents value outcomes)
    - **Zero-sum**



- Algorithmic goal: calculate a **strategy** (or **policy**) that decides a move in each state

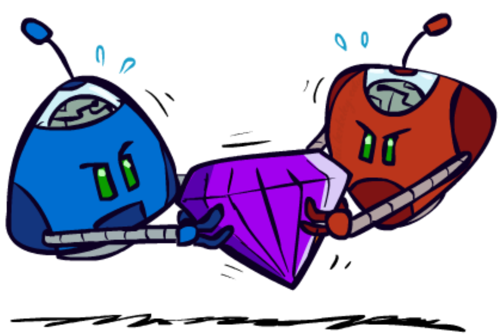
Utility

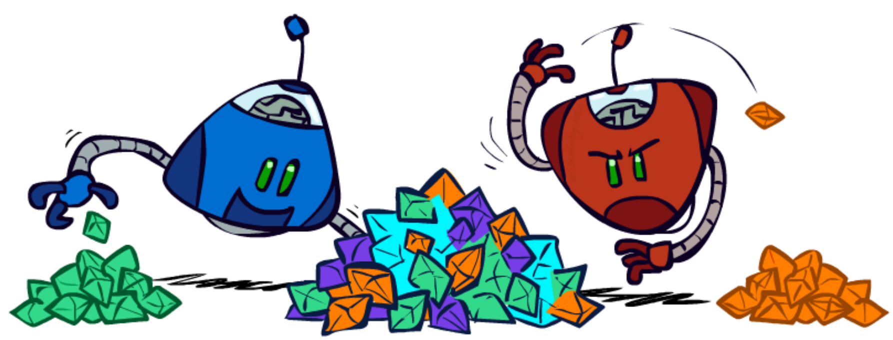
Zero/Constant-Sum

- Opposite utilities
- Adversarial, pure competition

General Games

- Independent utilities
- Cooperation, indifference, competition, and more are all possible





Examples: Perception vs. Chance

	Deterministic	Stochastic
Perfect	Chess, Checkers, Go, Othello	Backgammon, Monopoly
Imperfect	Battleship	Bridge, Poker, Scrabble

Checkers

- 1950: First computer player
- 1994: First computer champion (Chinook) ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame
- 1995: defended against Don Lafferty
- 2007: **solved**!

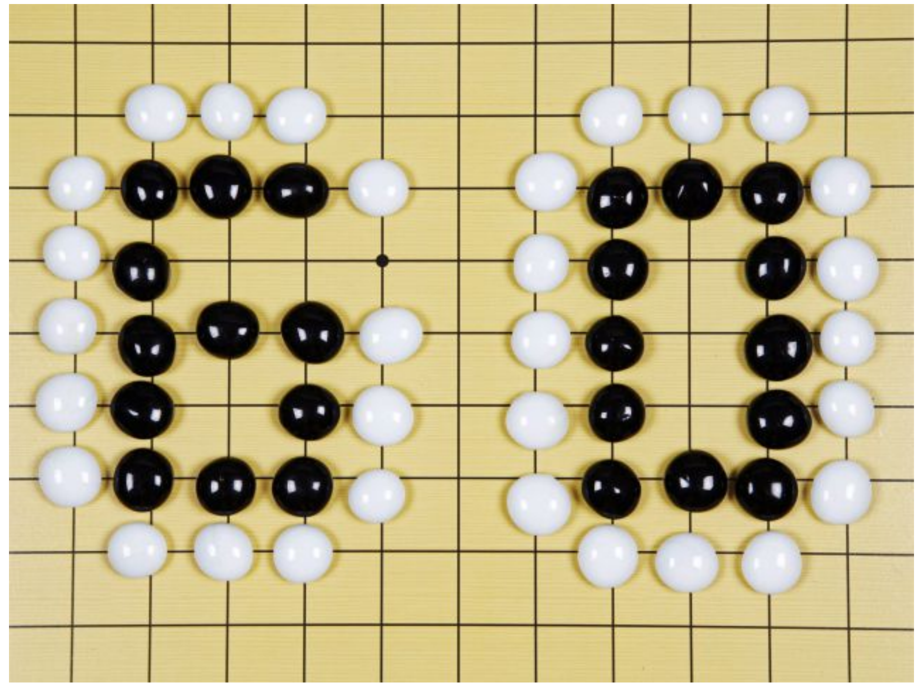
Chess

- 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

DeepBlue

Go

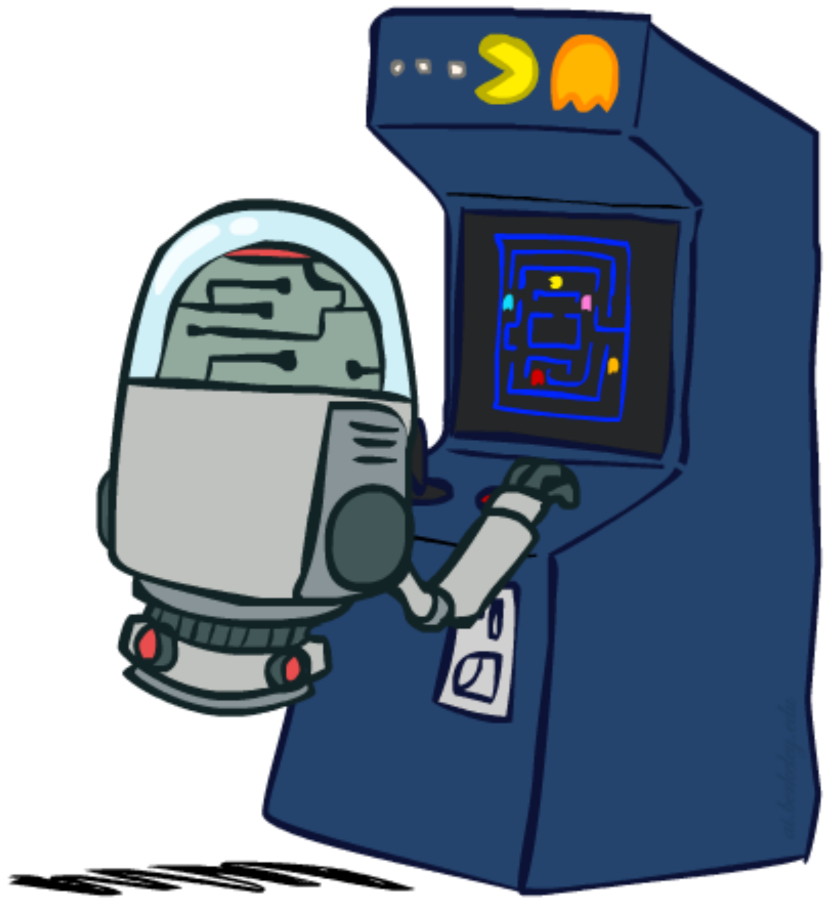
- Until recently, were not competitive at champion level
- 2016: beaten European champion
  - World champion game pending…
- ANNs for **policy** (what to do) and **evaluation** (how good is a board state)



AlphaGo

More Progress

- Othello: 1997, defeated world champion
- Bridge: 1998, competitive with human champions
- Scrabble: 2006, defeated world champion



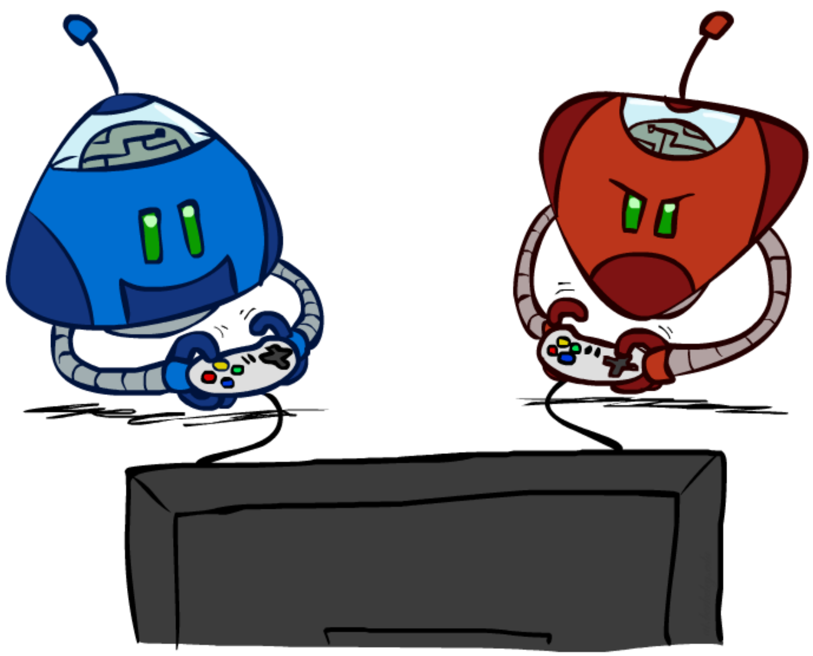
Game Formalism

- States: S (start at S_0)
- Players: P $\{1, \dots N\}$ (typically take turns)
- Actions: $Action(s)$, returns legal options
- Transition function: $S \times A \rightarrow S$
- Terminal test: $Terminal(s)$, returns T/F
- Utility: $S \times P \rightarrow \mathbb{R}$

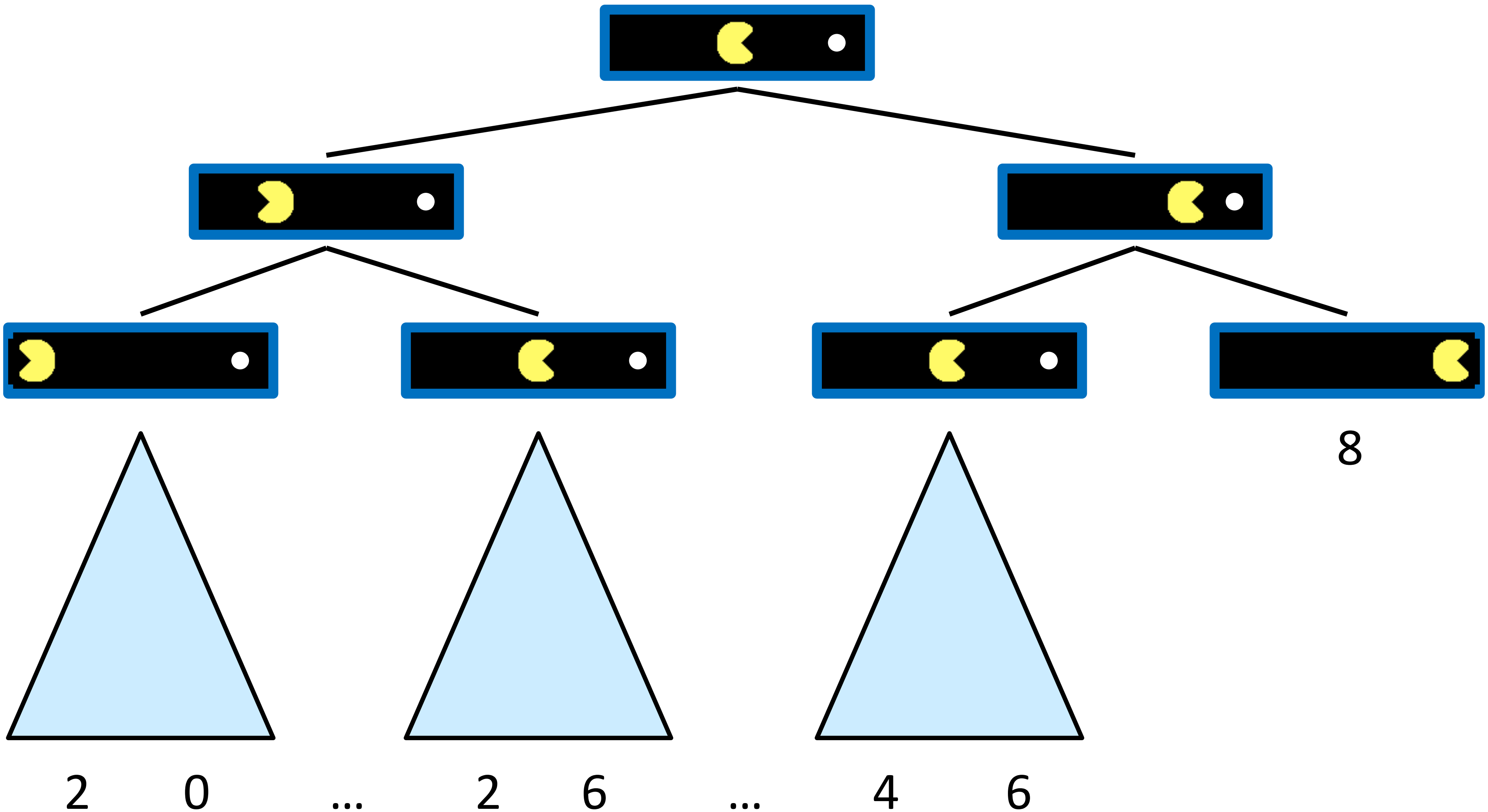
- Solution for a player is a **policy**: $S \rightarrow A$

Game Plan :)

- Start with deterministic, two-player adversarial games
- Issues to come
  - Multiple players
  - Resource limits
  - Stochasticity



Single-Agent Game Tree

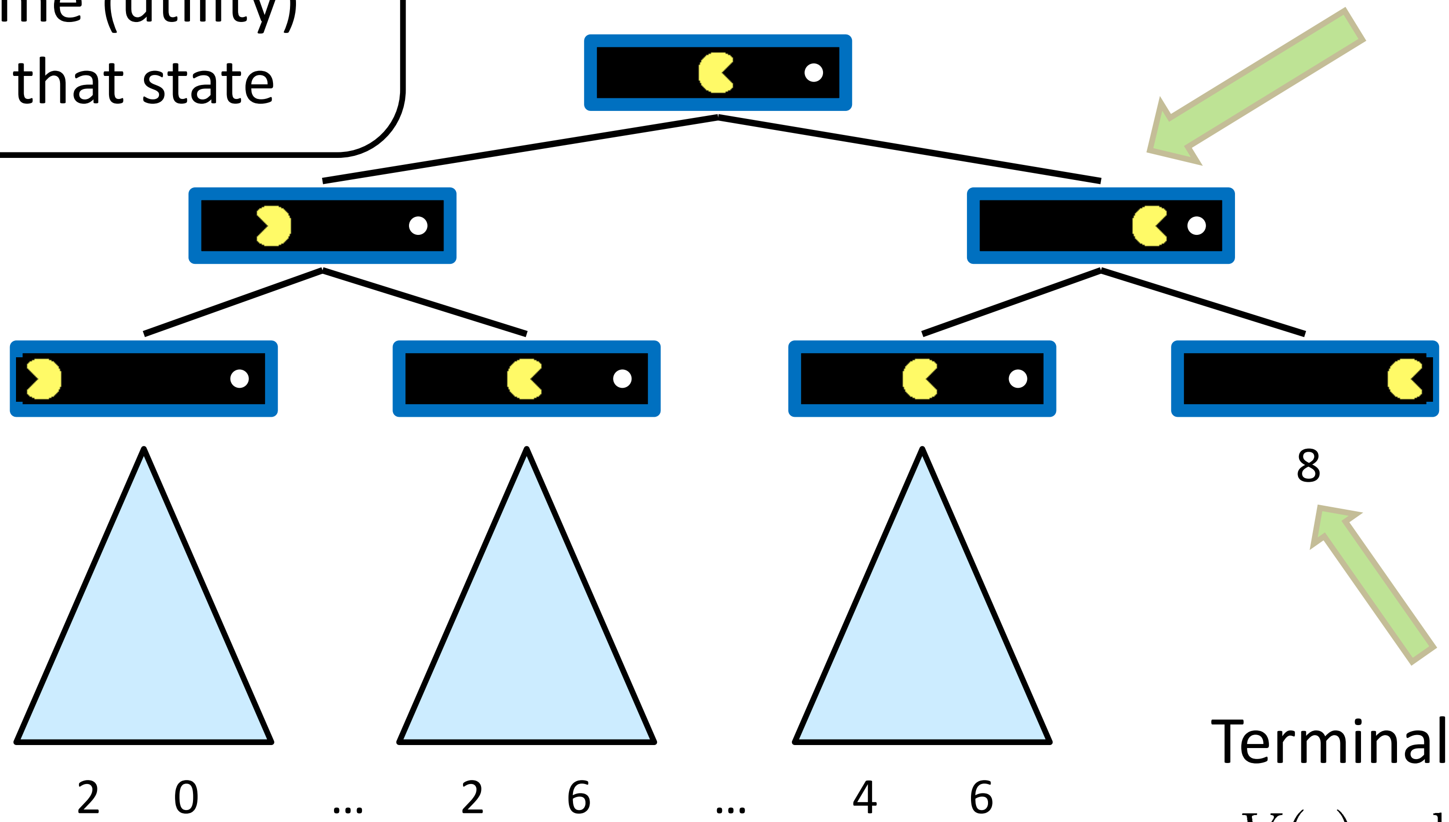


Value of a State

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

Non-Terminal States:

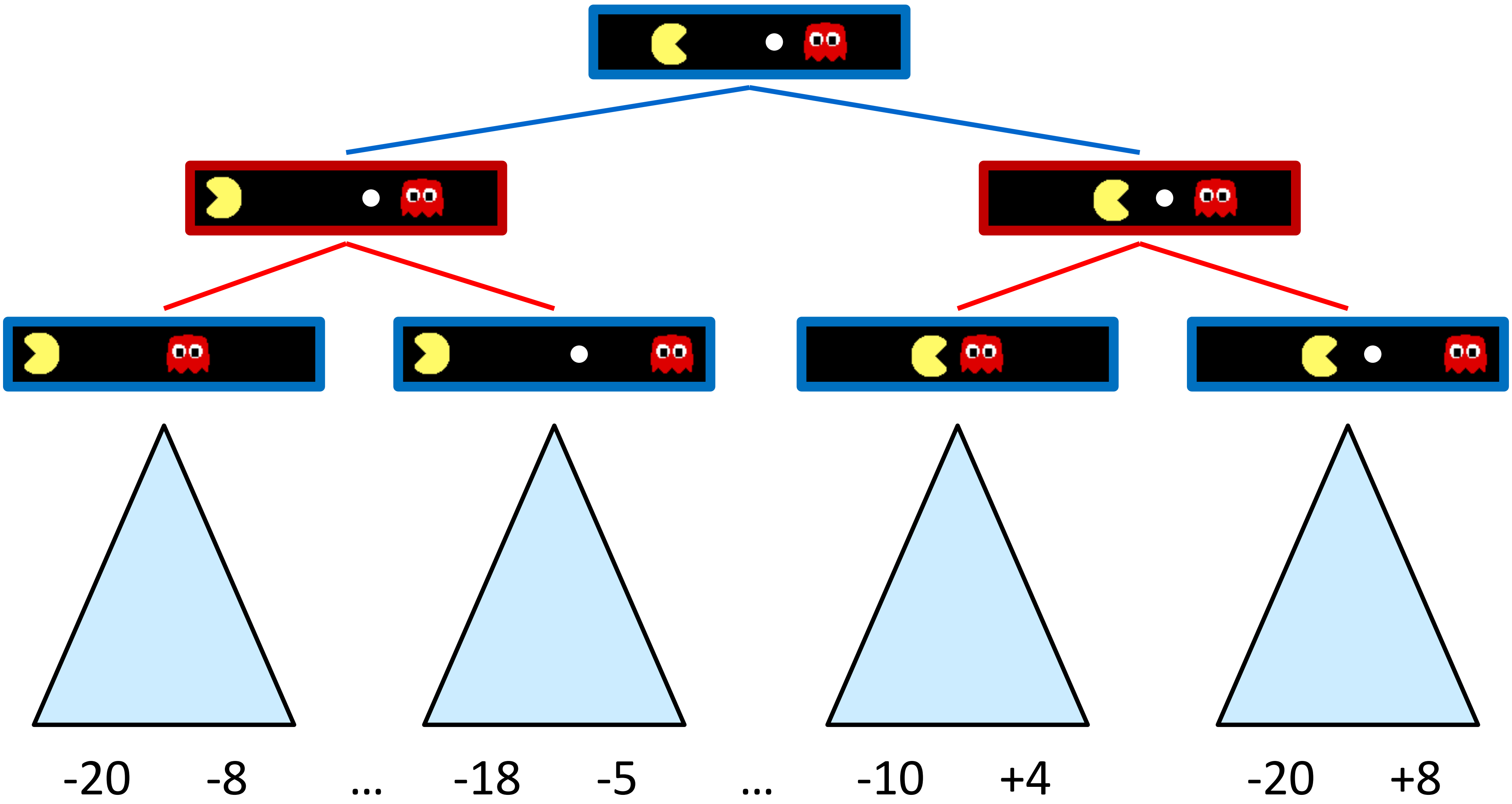
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

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

Adversarial Game Trees



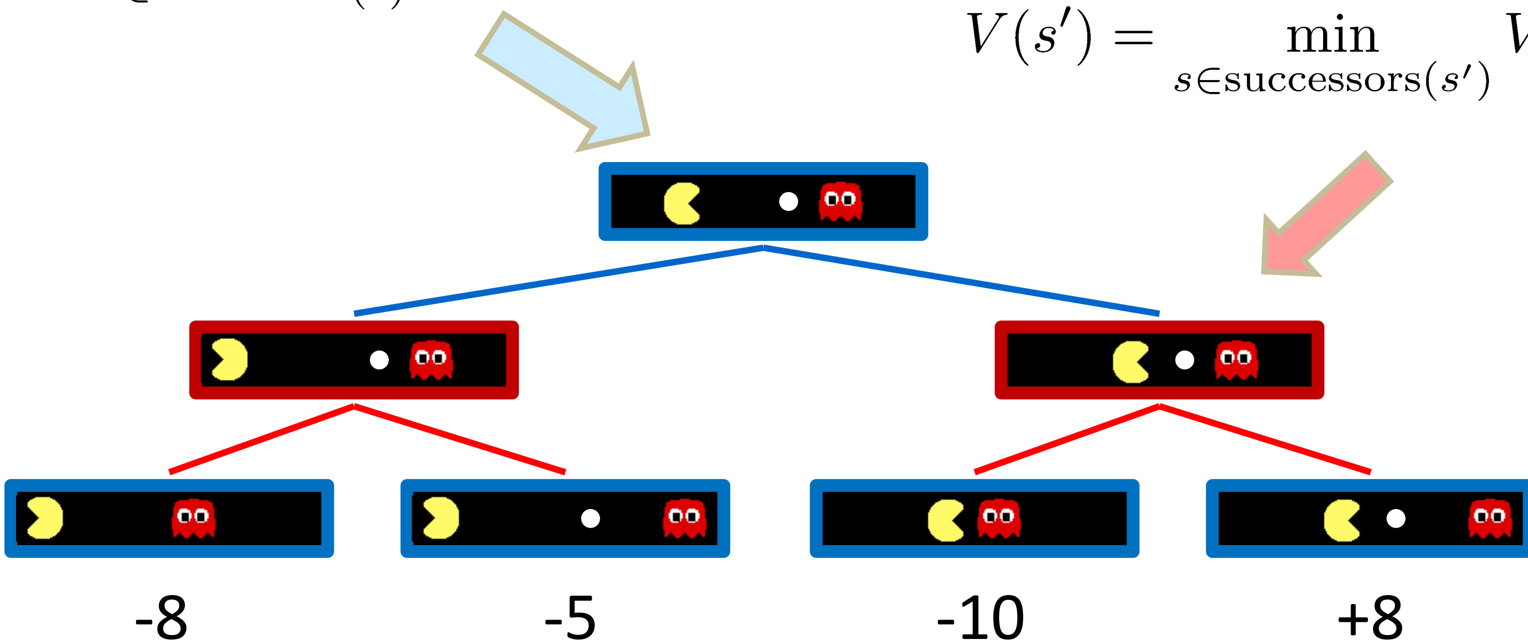
Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

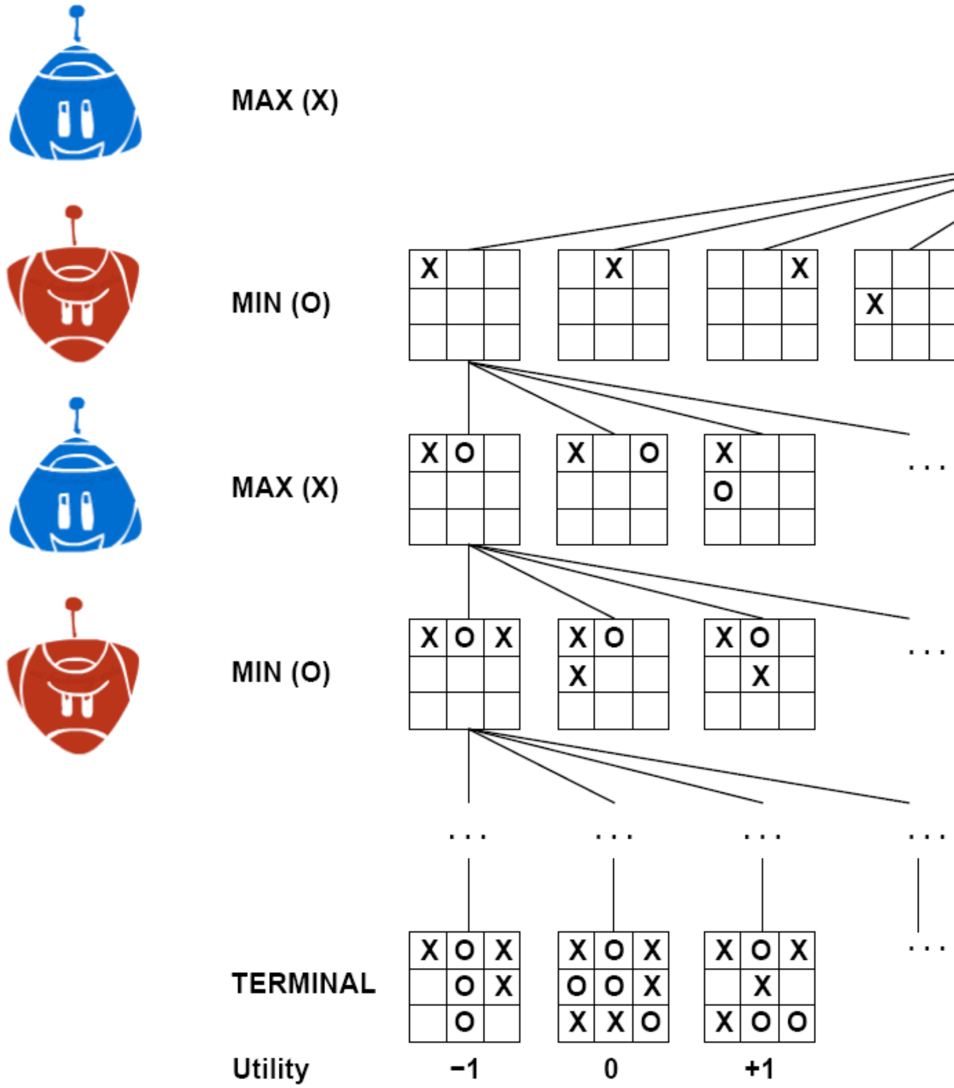
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

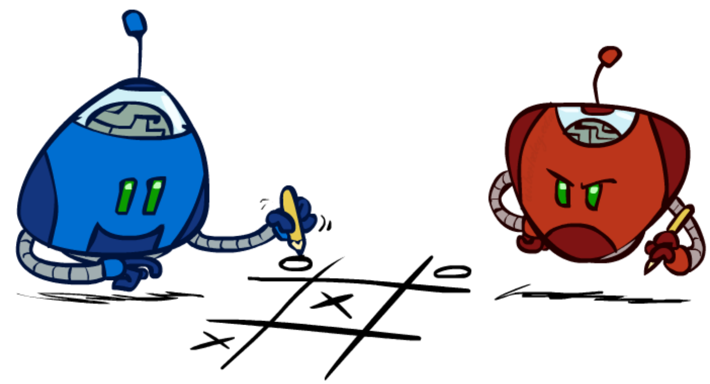


Terminal States:

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

Tic-Tac-Toe Game Tree

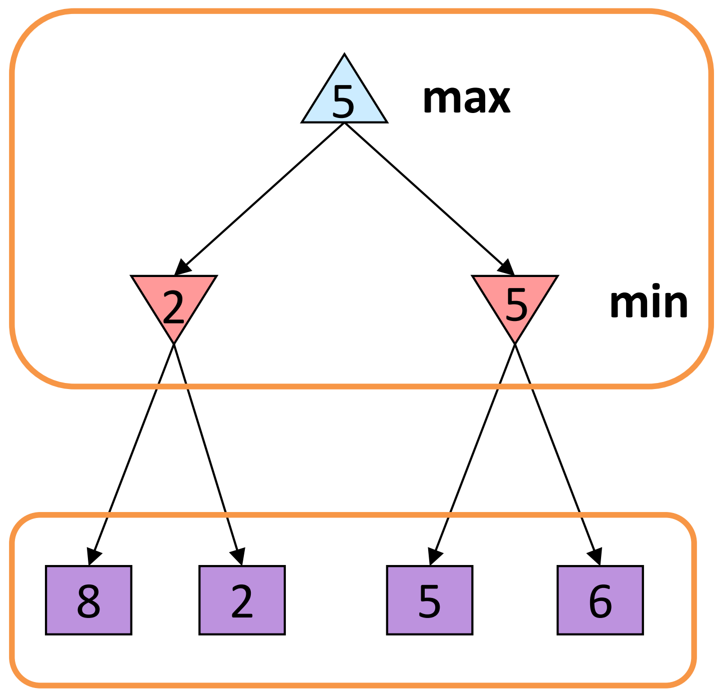




Adversarial Search via Minimax

- Deterministic, zero-sum
  - Tic-tac-toe, chess
  - One player maximizes
  - The other minimizes
- Minimax search
  - A 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

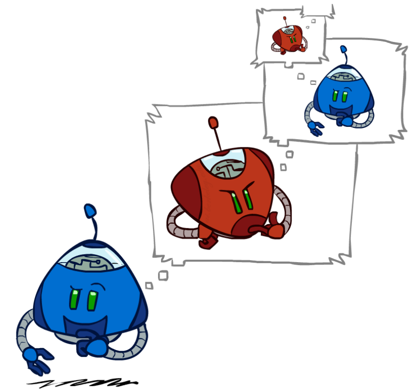
Minimax Implementation

```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

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

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

Minimax Evaluation

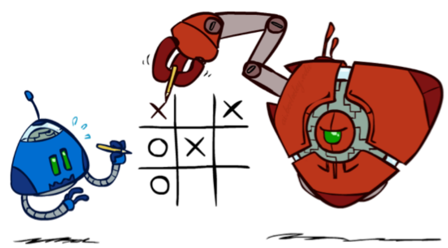


<u>Time</u>

- $\mathcal{O}(b^m)$
  - For chess: $b \approx 35,\, m \approx 100$

<u>Space</u>

- $\mathcal{O}(bm)$



<u>Complete</u>

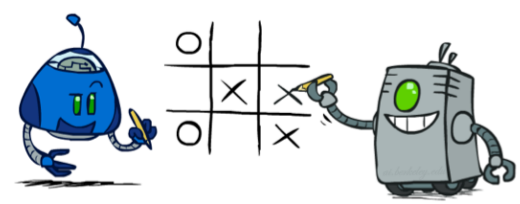
- Only if finite

Minimax-Min

<u>Optimal</u>

- Yes, against **optimal** opponent

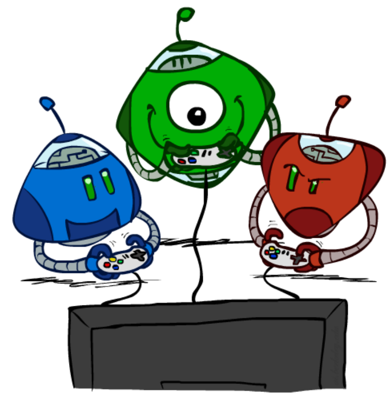
Minimax-Avg

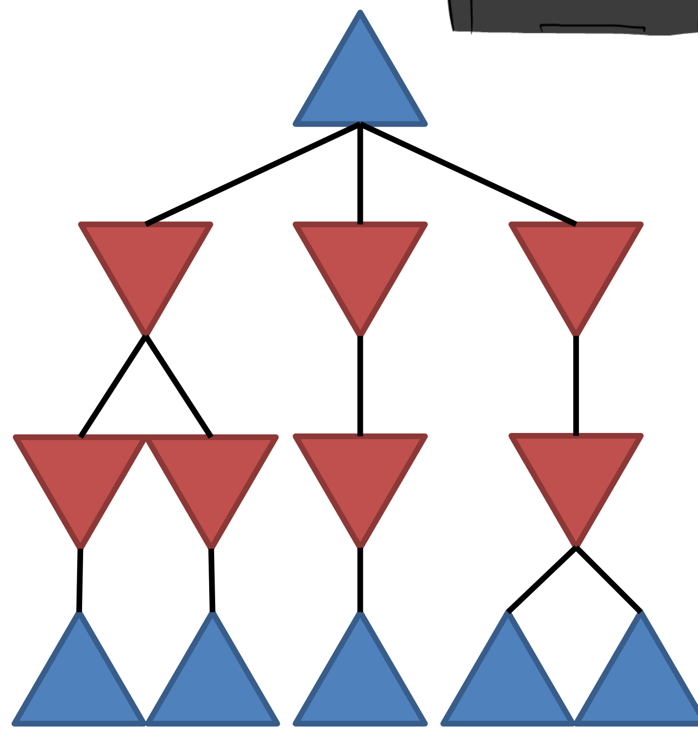


Multiple Players

Add a **ply** per player

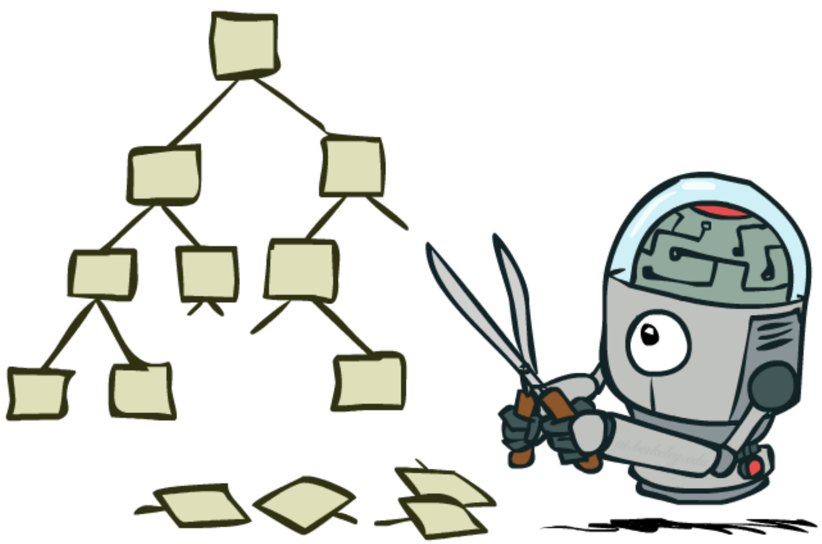
- Independent utility: use a vector of values, each player MAX own utility
- Zero-sum: each team sequentially MIN/MAX
  - In Pacman, have multiple MIN layers for each ghost per 1 Pacman move



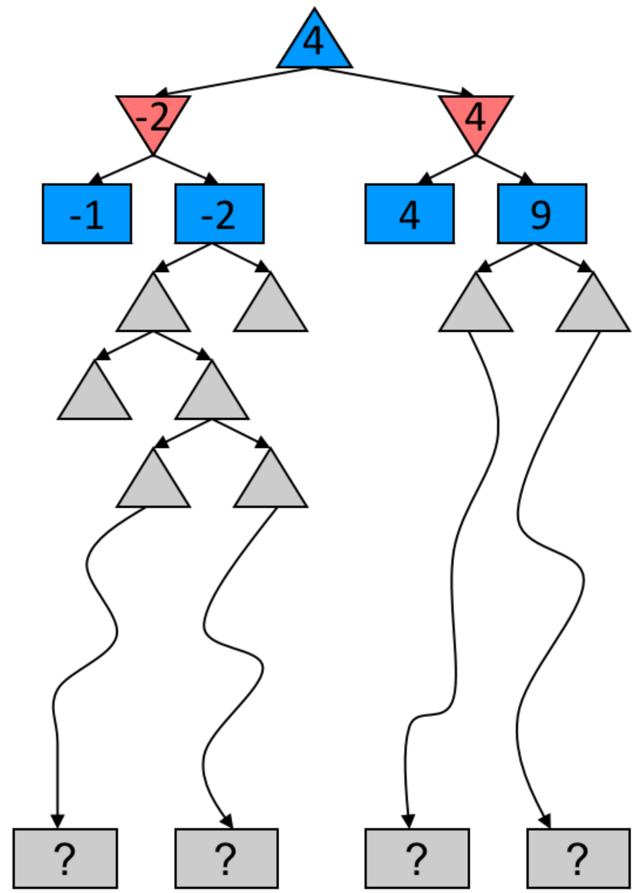


Scaling to Larger Games

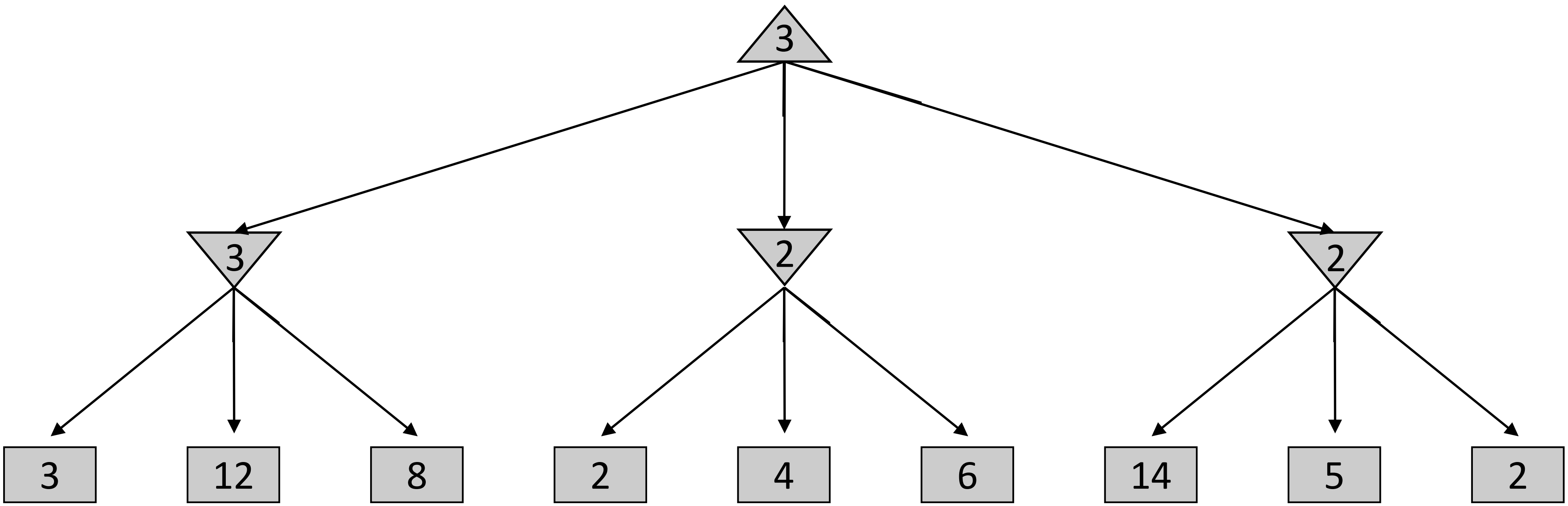
Tree Pruning



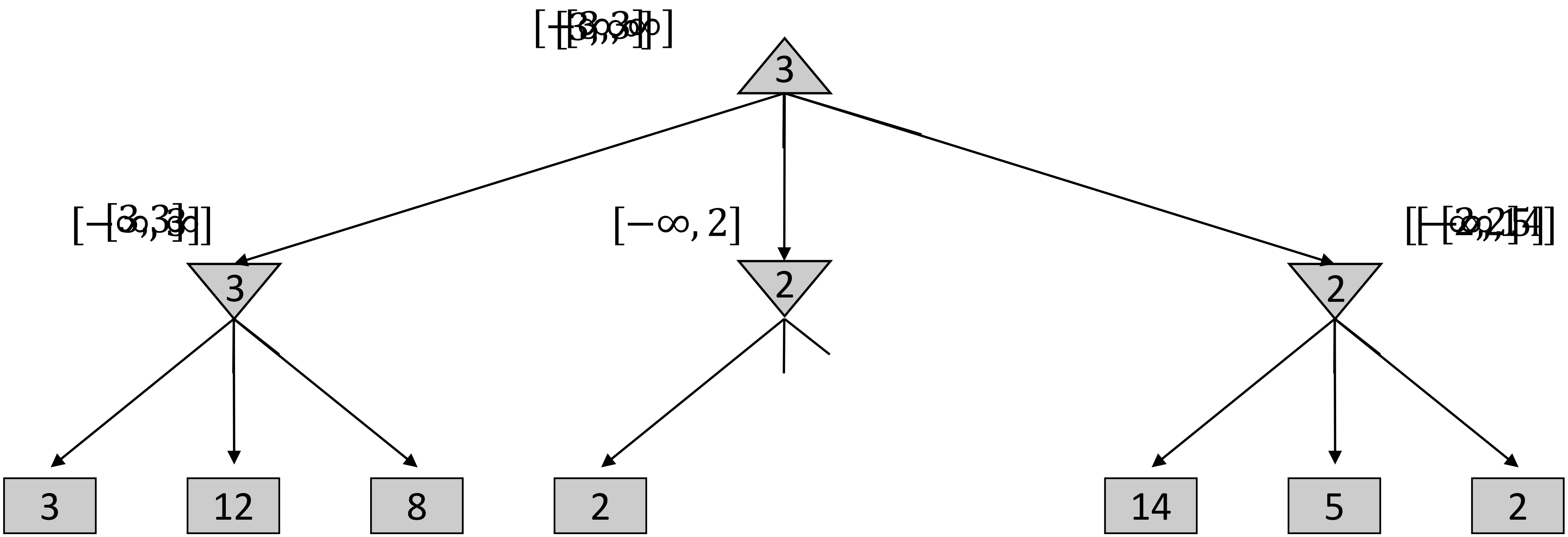
Depth-Limiting + Evaluation



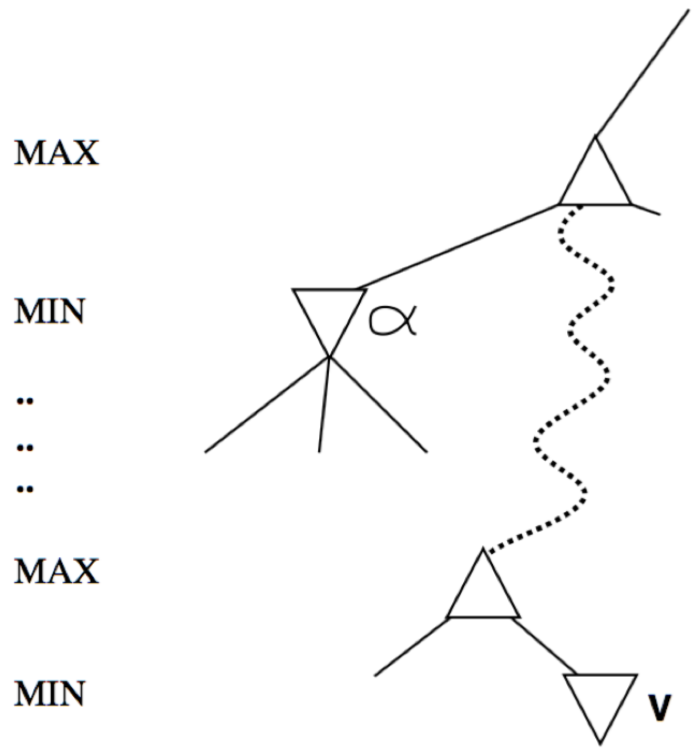
Minimax Example



Minimax Pruning



General Case



- α is the best value (to MAX) found so far off the current path
- If V is worse than α, MAX will avoid it – prune that branch
- Define β similarly for MIN

Alpha-Beta Pruning

```
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
```

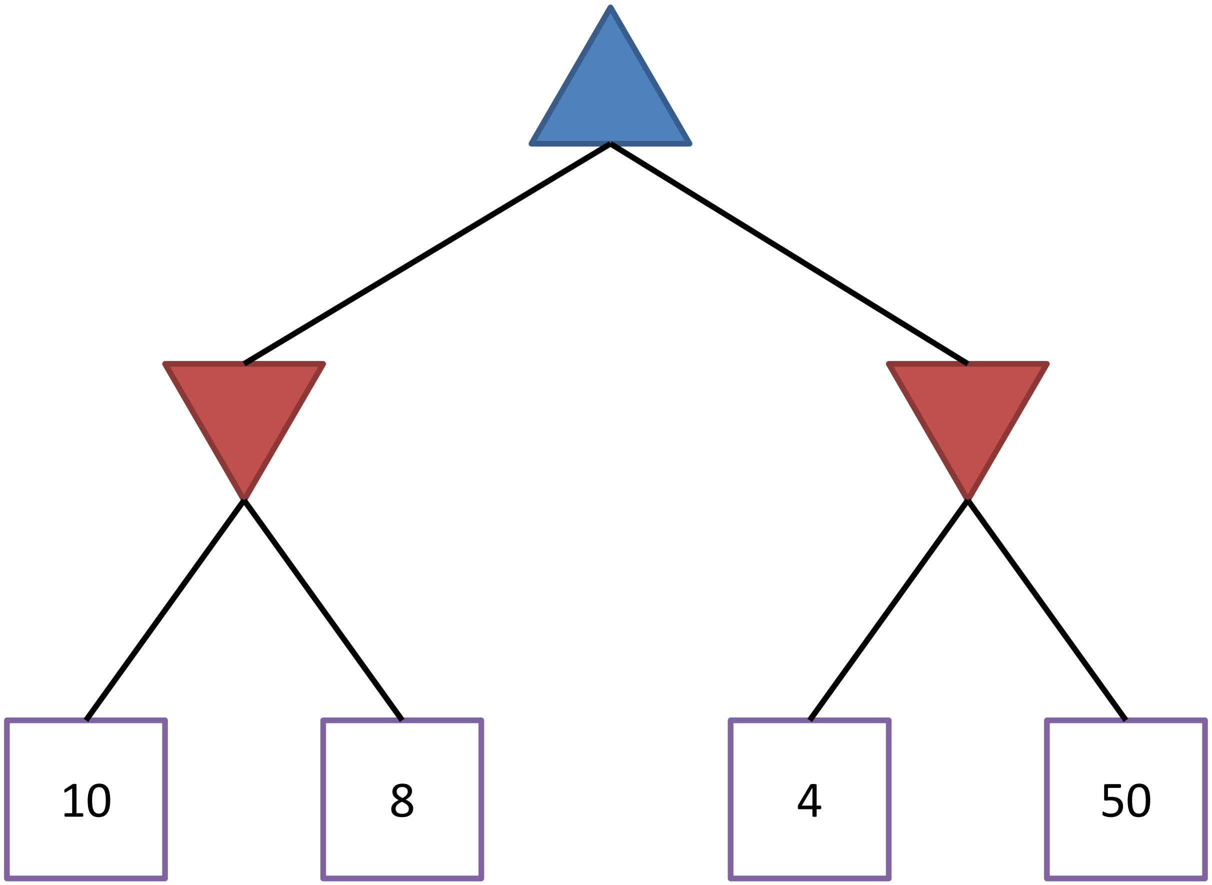
```
def max-value(state, α, β):
    initialize v = -∞
    for each successor of state:
        v = max(v,value(successor,α,β))
        if v ≥ β return v
        α = max(α, v)
    return v
```

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

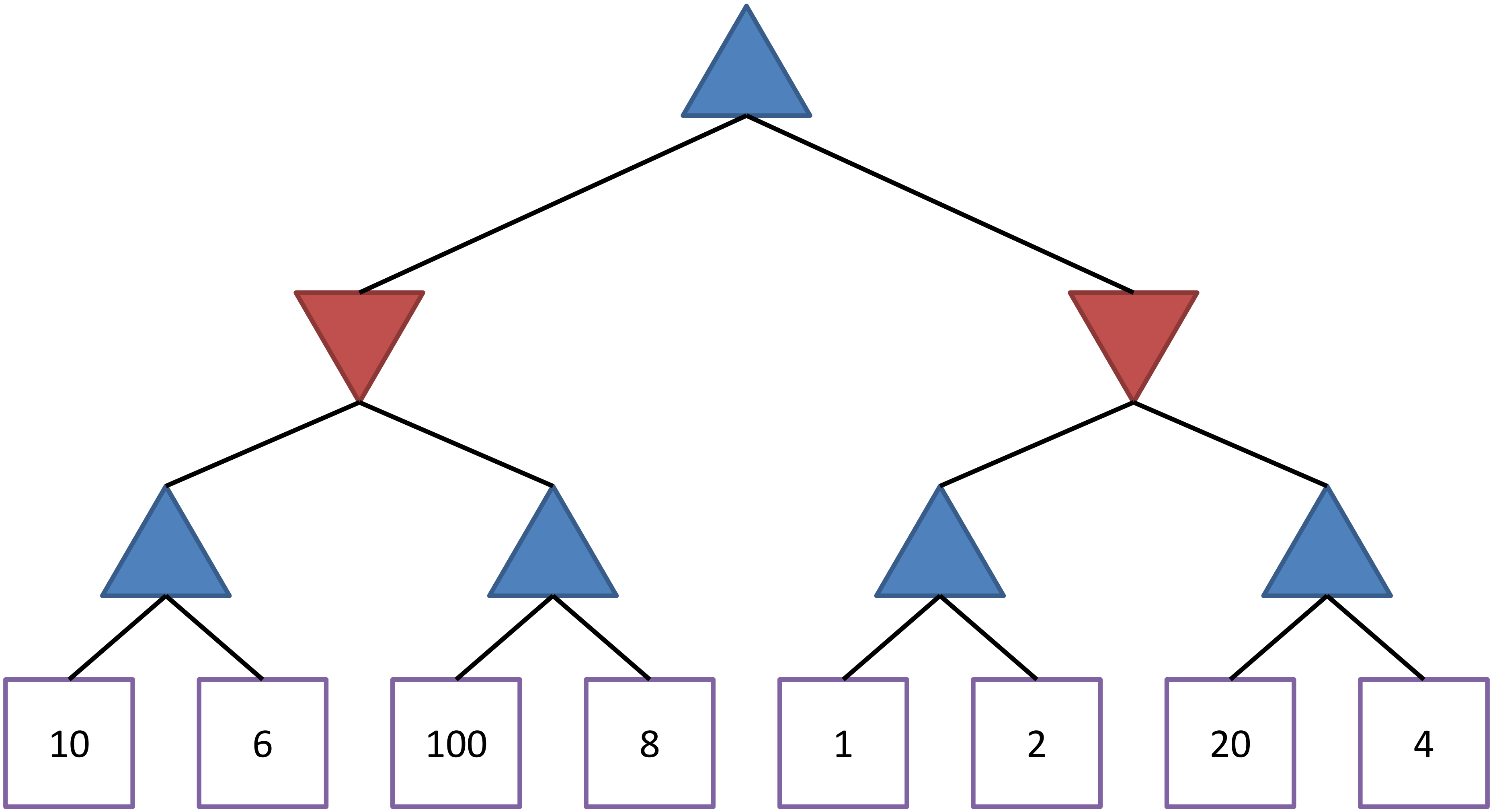
Alpha-Beta Properties

- Has no effect on minimax value computed for the root!
- Good child ordering improves effectiveness of pruning
- With "perfect ordering":
  - Time complexity drops to $\mathcal{O}(b^{m/2})$
  - Doubles solvable depth!
  - Full search of, e.g. chess, is still hopeless…
- This is a simple example of **metareasoning** (computing about what to compute)

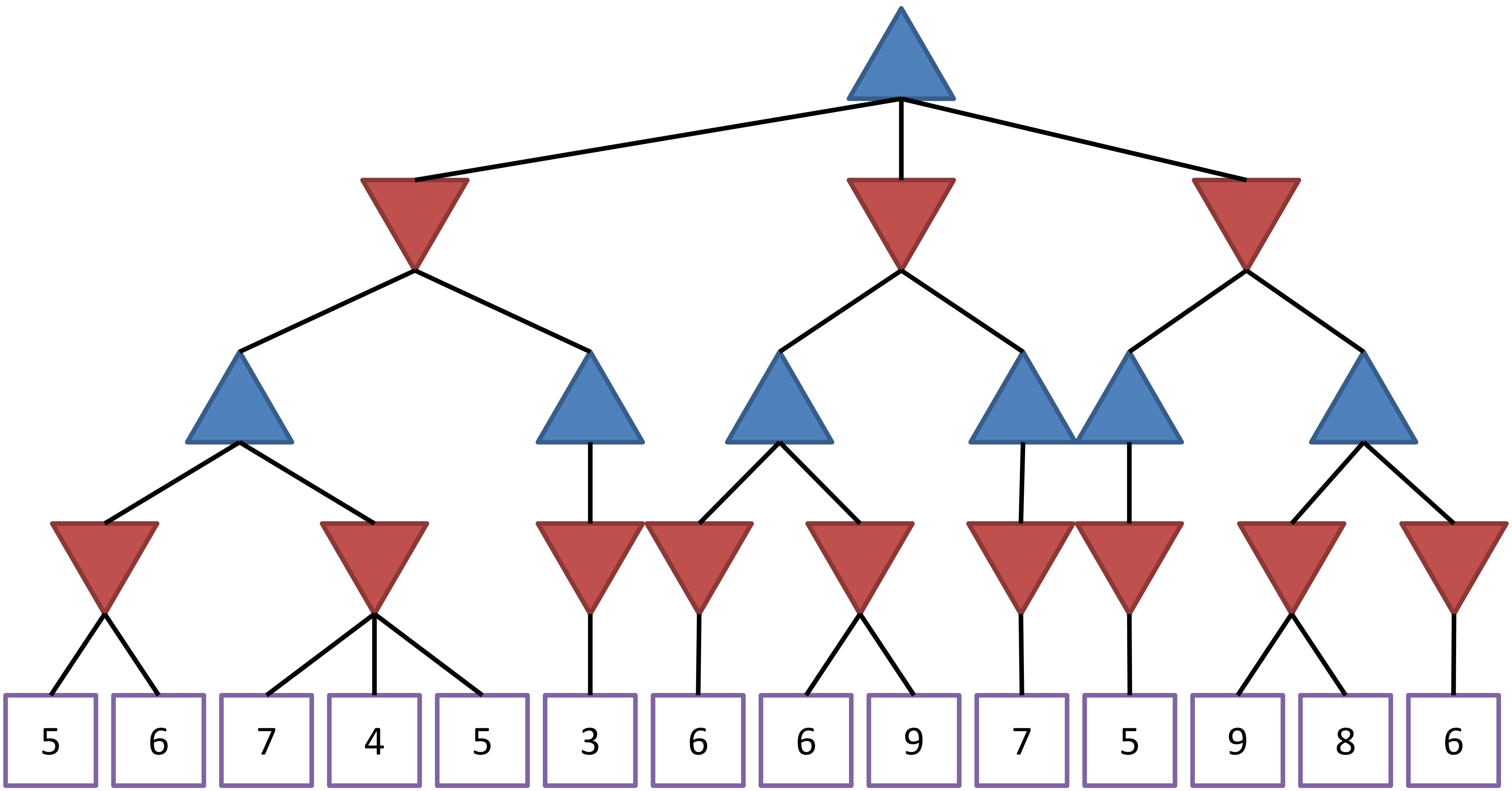
Checkup #1



Checkup #2



Checkup #3

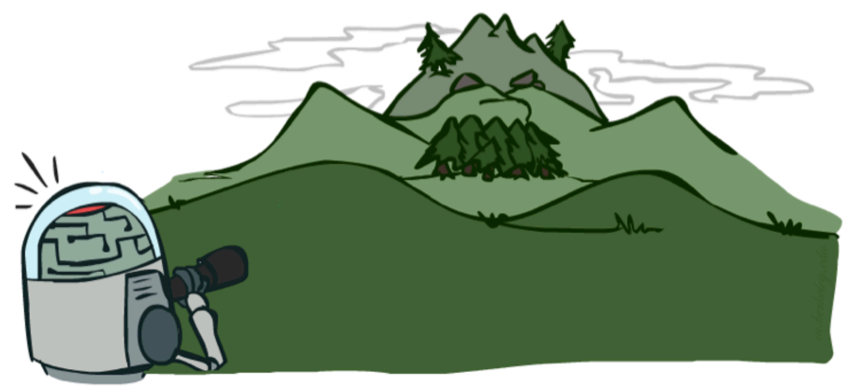


Resource Limits

- Problem: in realistic games, cannot search to leaves!
- Solution: depth-limited search
  1. Search only to a limited depth in the tree
  2. Replace terminal utilities with an **evaluation function** for non-terminal positions
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm

Search Depth Matters

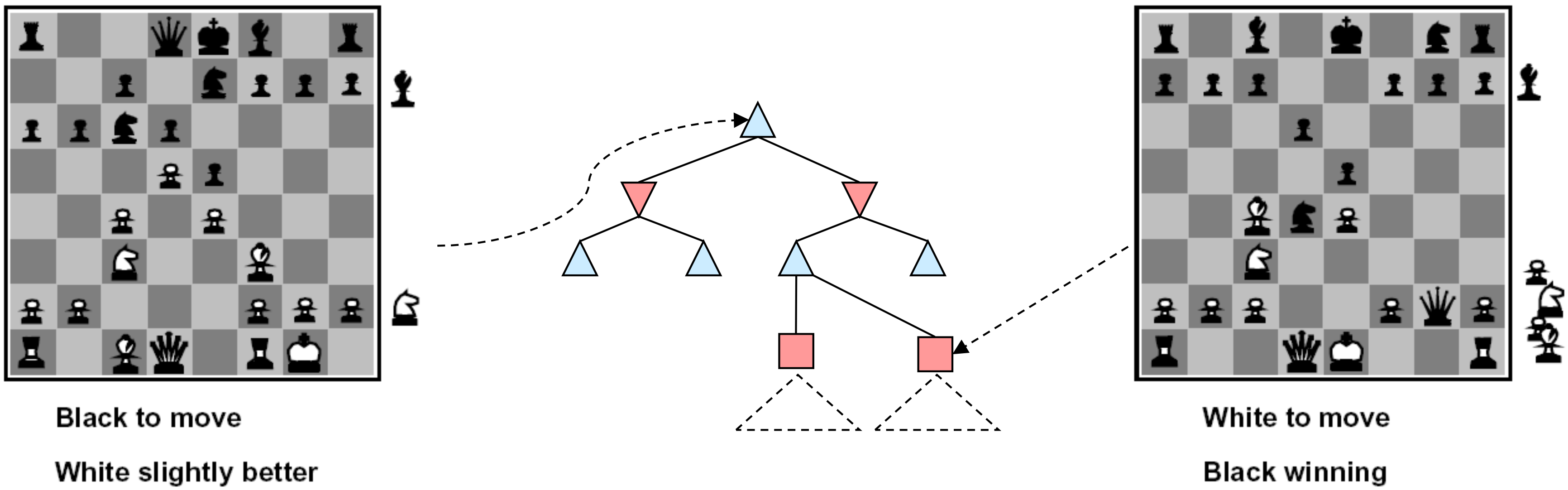
- Evaluation functions are always imperfect
- The deeper in the tree the evaluation function is buried, the less the quality of the evaluation function matters
- An important example of the tradeoff between complexity of features and complexity of computation



Depth2

Depth10

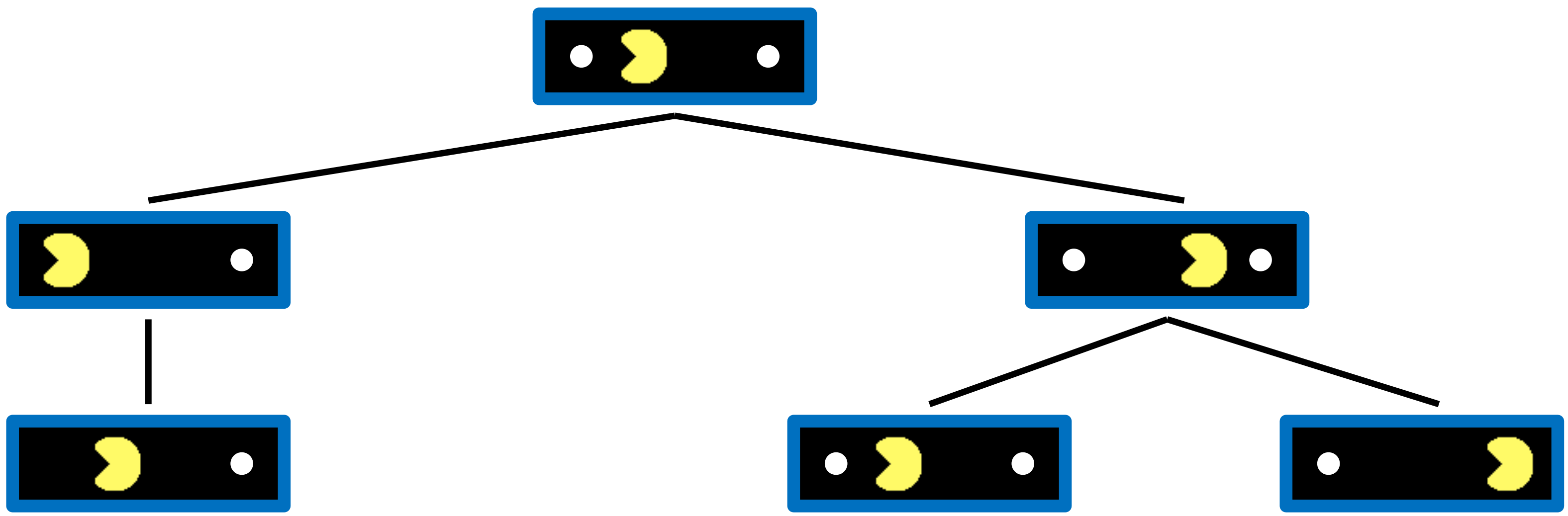
Evaluation Functions



- Evaluation functions score non-terminals in depth-limited search
- Ideal: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

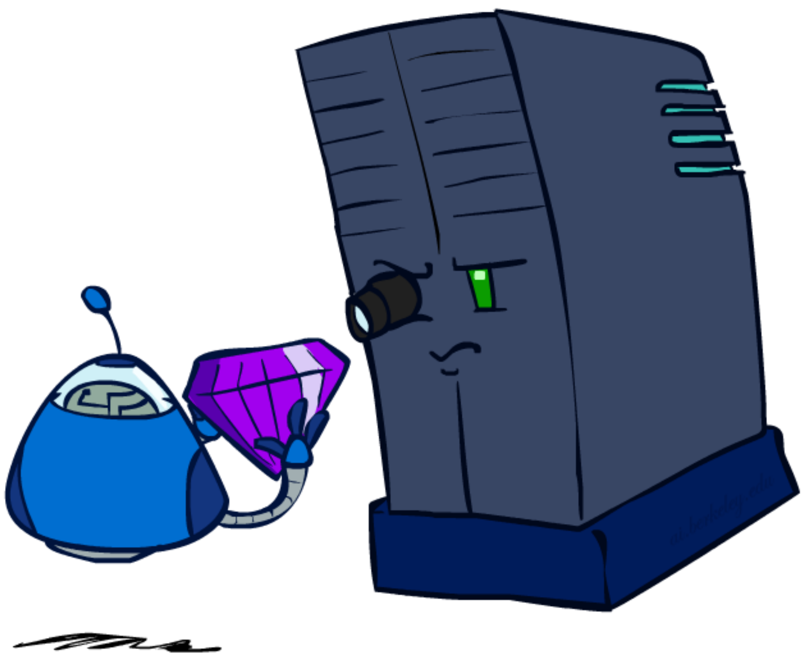
  e.g. $f_1(s) = (\text{num white queens} - \text{num black queens})$

Why Pacman Starves/Thrashes



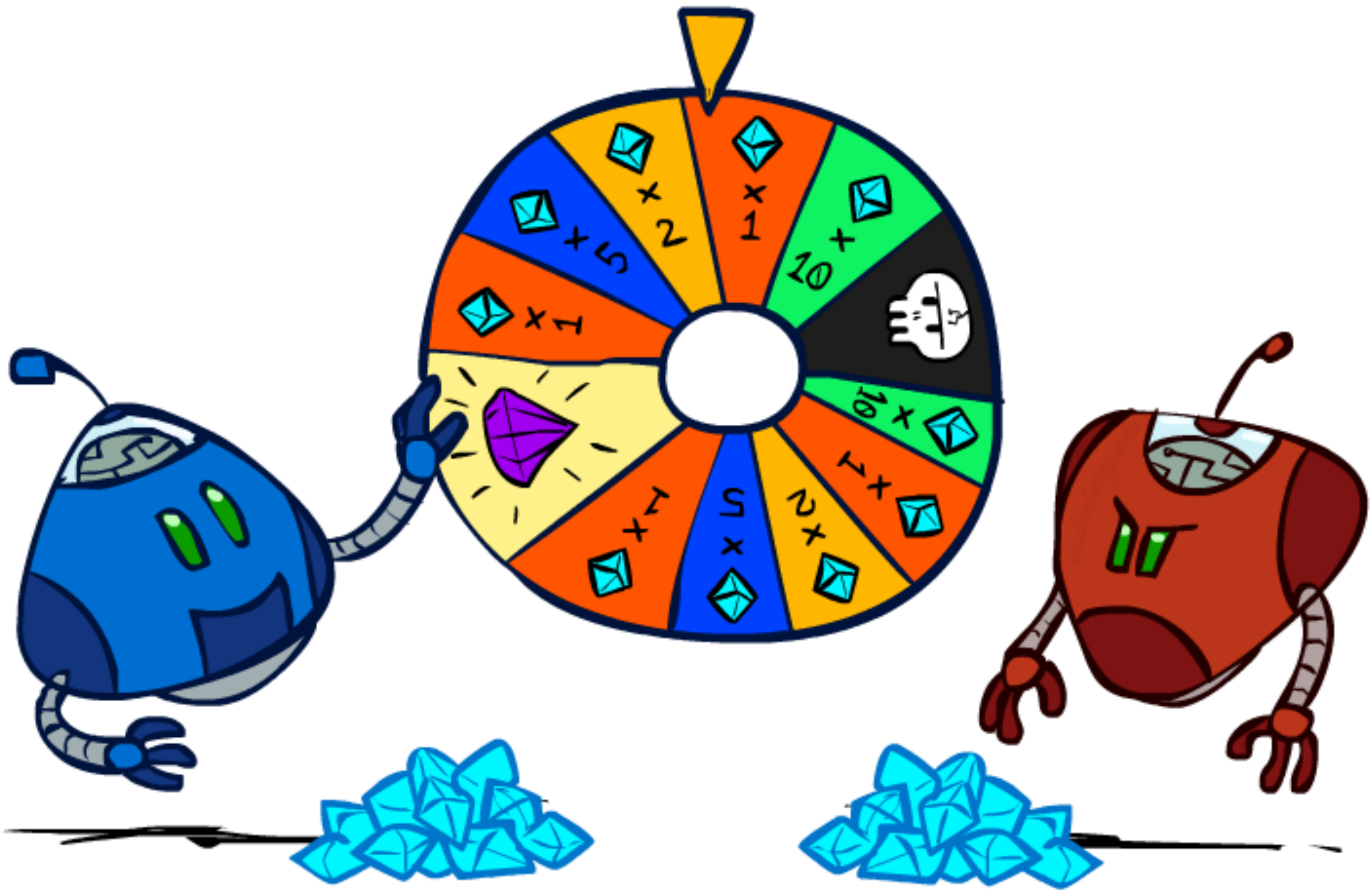
- A danger of replanning agents!
  - He knows his score will go up by eating a dot now
  - He knows his score will go up just as much by eating a dot later
  - There are no point-scoring opportunities after eating a dot (within the horizon, two here)
  - Therefore, waiting seems just as good as eating: he may go east, then back west in the next round of replanning!

Pacman/Ghost Evaluation

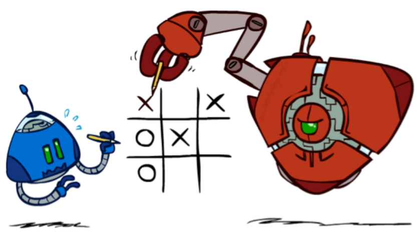


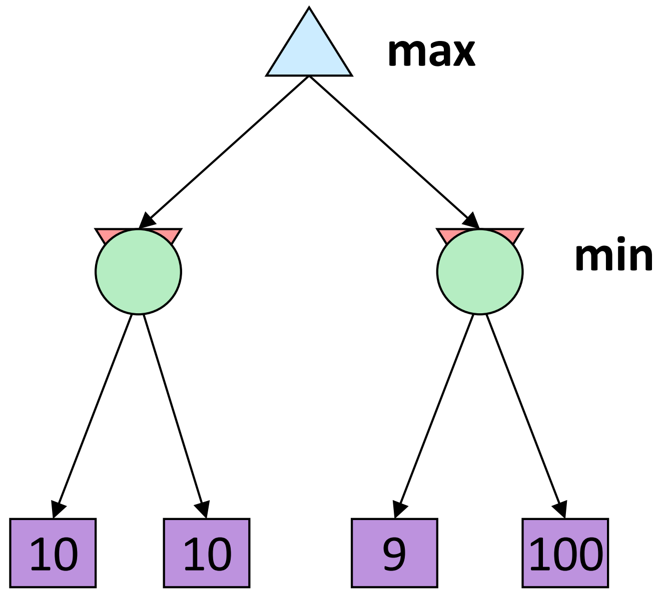
- Thrashing
- Thrashing-Fixed
- SmartGhosts-1
- SmartGhosts-2

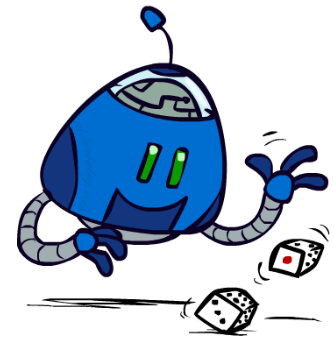
Nondeterministic Games



Worst Case vs. Average Case







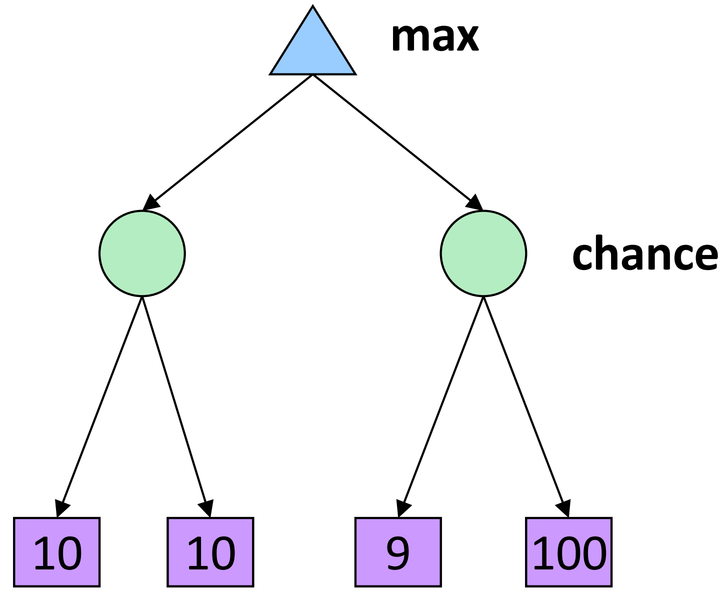
In nondeterministic games, chance is introduced by non-opponent stochasticity (e.g. dice, card-shuffling)

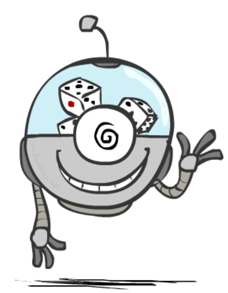
Expectiminimax Search

- Why wouldn’t we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Actions can fail: when moving a robot, wheels might slip

- Values should now reflect average-case (**expectimax**) outcomes, not worst-case (**minimax**) outcomes

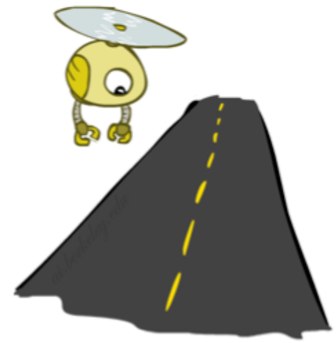
- Expectimax search: compute the average score under optimal play
  - Max nodes as in minimax search
  - **Chance nodes** are like min nodes but the outcome is uncertain
  - Calculate their **expected utilities**





Reminder: 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



0.25

0.50

0.25

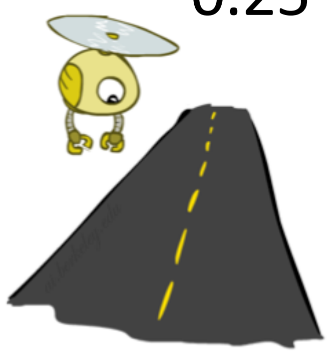
Reminder: Expectations

- The **expected value** of a function 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**
$P(T)$	0.25		0.50		0.25	

T



Expectiminimax Implementation

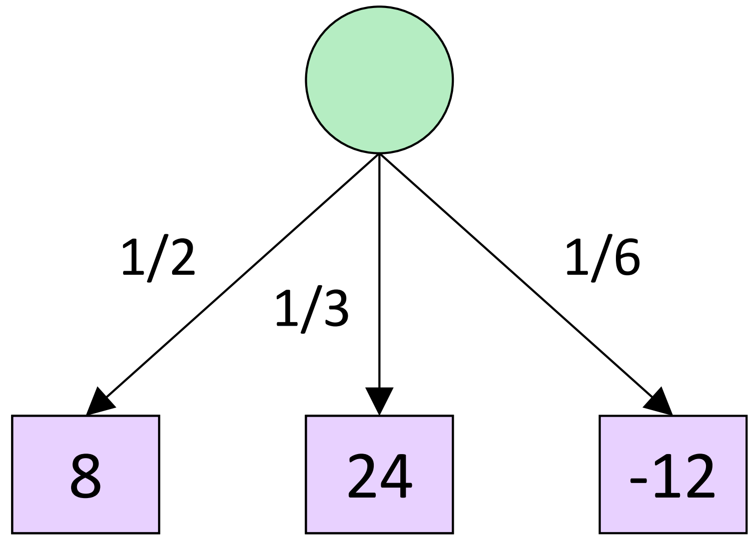
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

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

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

Expectiminimax Example

```
def exp-value(state):
    initialize v = 0
    for each successor of state:
        p = probability(successor)
        v += p * value(successor)
    return v
```

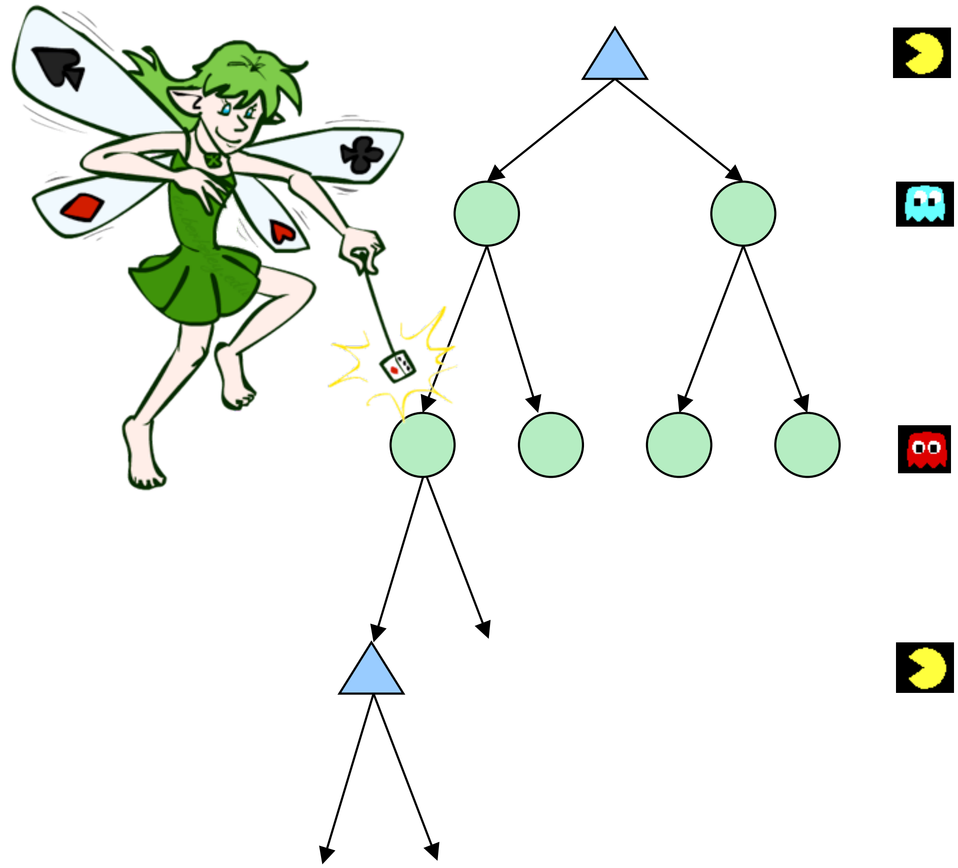


$$v = (1/2)\,(8) + (1/3)\,(24) + (1/6)\,(-12) = 10$$

Where Do Probabilities Come From?

- In expectiminimax search, we have a probabilistic model of how the opponent (or environment) will behave in any state
  - Model could be a simple uniform distribution (roll a die)
  - Model could be sophisticated and require a great deal of computation
  - We have a chance node for any outcome out of our control: opponent or environment
  - The model might say that adversarial actions are likely!

- For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes



Summary

- A game can be formulated as a search problem, with a solution **policy** $(S \rightarrow A)$
- For deterministic games, the **minimax** algorithm plays optimally (assuming the **game tree** is reasonable)
- To help with resource limitations, standard practice is to employ **alpha-beta pruning** and **depth-limited search** (with an **evaluation function**)
- To model uncertainty, the **expectiminimax** algorithm introduces **chance nodes** that employ a **probability distribution** over actions to model **expected utility**