# DATA130008 Introduction to Artificial Intelligence

复旦大学大数据学院
School of Data Science, Fudan University

魏忠钰

# Adversarial Search

Data Intelligence and Social Computing Lab (DISC)

October 26th, 2021

# Outline

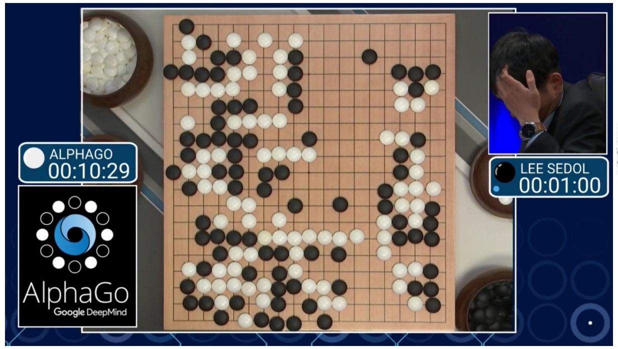
- Type of game

# Types of Games

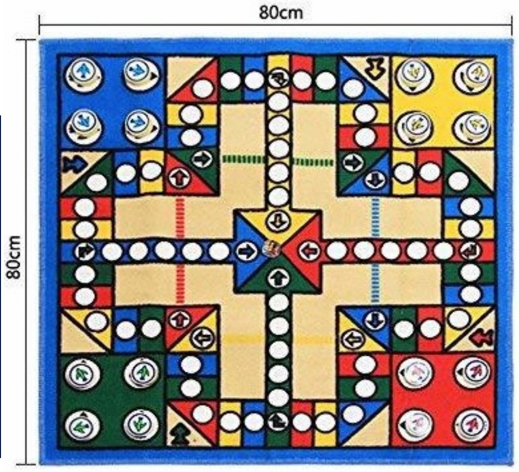
- **Deterministic** or stochastic ?
  - **GO, Chess** VS Aeroplane Chess, Monopoly
- One, **two**, or more players ?
- **Zero sum** or not ?
  - **GO** VS Contra
- **Perfect information** ?
  - **GO** VS DOTA




**Chess**




**GO**




**Aeroplane Chess**


**Monopoly**

# Zero-Sum Games

- Zero-Sum Games
  - Agents have opposite utilities
  - Think of a single value that one maximizes and the other minimizes
  - **Adversarial**, pure competition

- General Games
  - Agents have independent utilities
  - Cooperation, competition, indifference and more are all possible

# Example: Non-zero sum game

## *The Prisoners' Dilemma*

| | | **Prisoner A Choices** | |
|---|---|---|---|
| | | *Stay Silent* | *Confess and Betray* |
| **Prisoner B Choices** | *Stay Silent* | Each serves one month in jail<br>**2 months in total** | Prisoner A goes free<br>**12 months in total**<br>Prisoner B serves full year in jail |
| | *Confess and Betray* | Prisoner A serves full year in jail<br>**12 months in total**<br>Prisoner B goes free | Each serves three months in jail<br>**6 months in total** |

# Deterministic Games with Multiple players

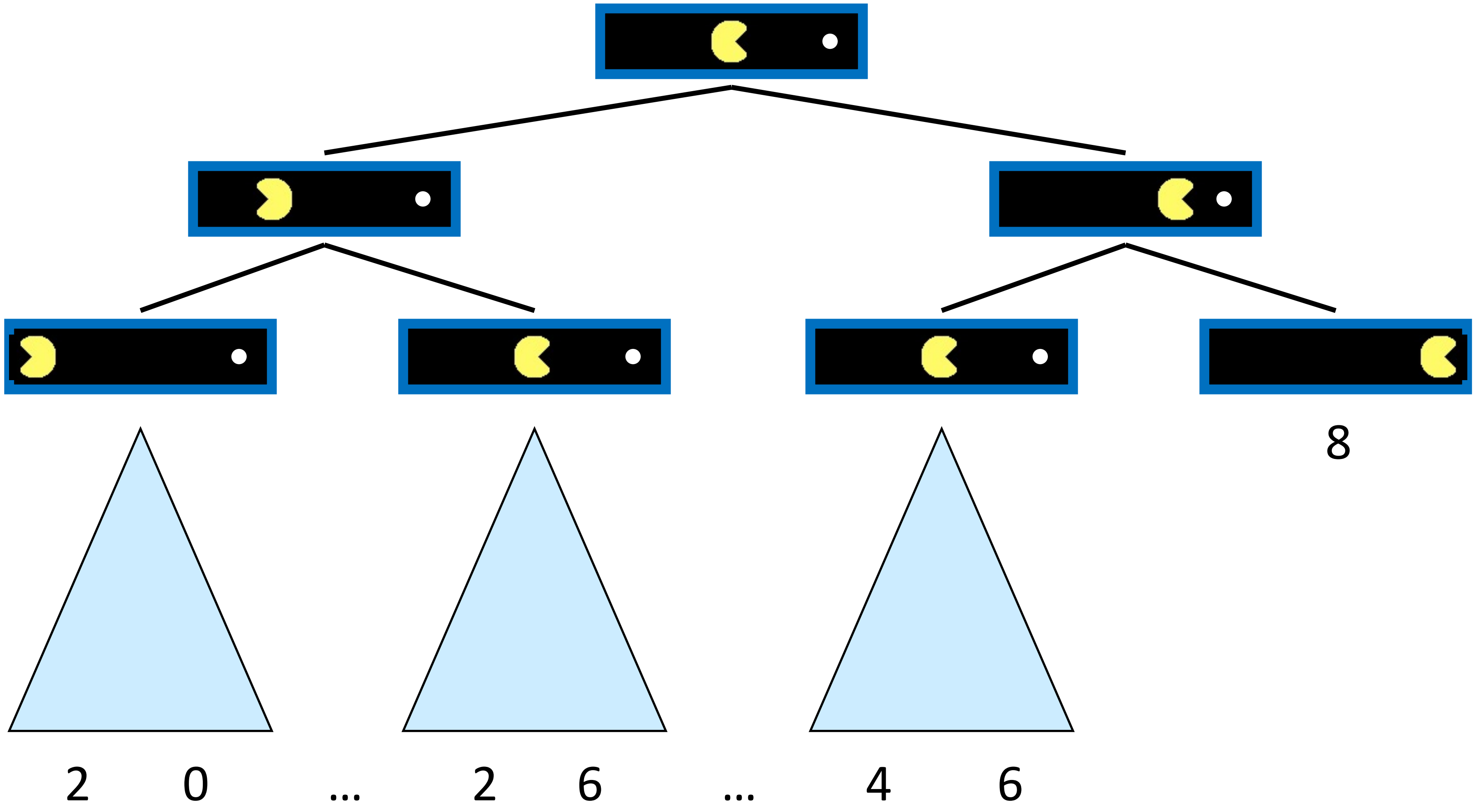
- S: states (start at $s_0$)
- **Player (s):** the player has the move in this state
- Actions (s): A set of legal moves in a state
- Results (s, a): A transition model, return the results of a move
- Terminal Test (s): {true, false} if s is the terminal state
- **Terminal Utilities (s, p):** A utility function gives the final numeric value of a game

- Zero Sum

# Outline

- Type of Game
- Adversarial Search

# Single-Agent Trees

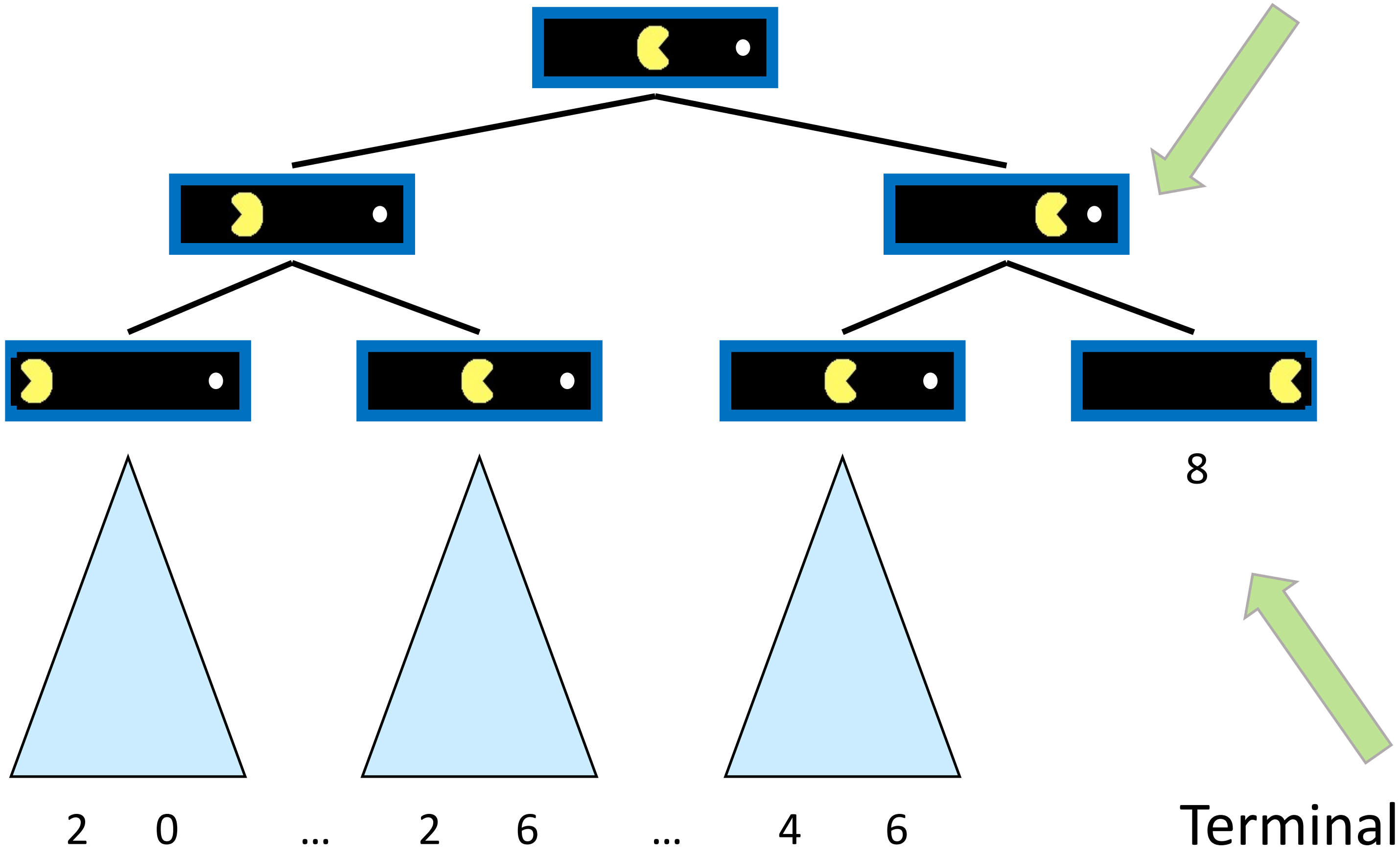


# Value of a State

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

Non-Terminal States:

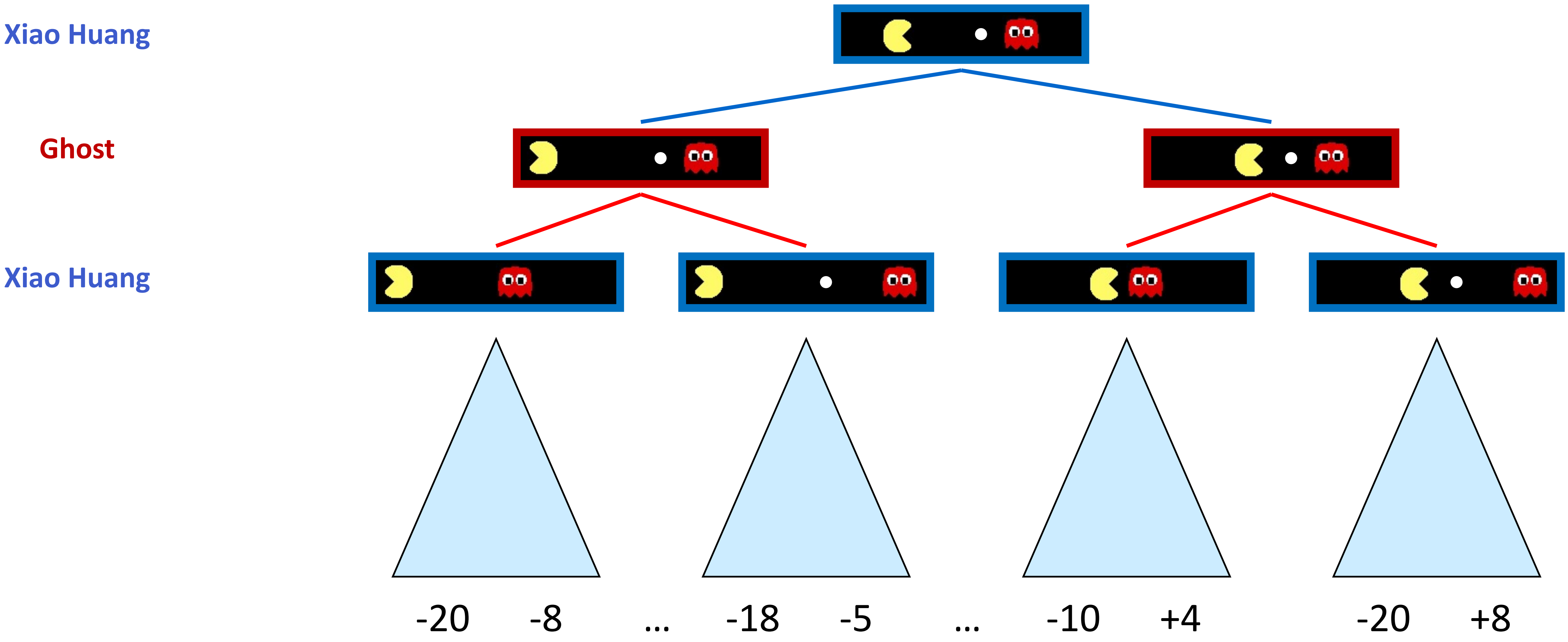
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$




Terminal States:

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

# Adversarial Game Trees



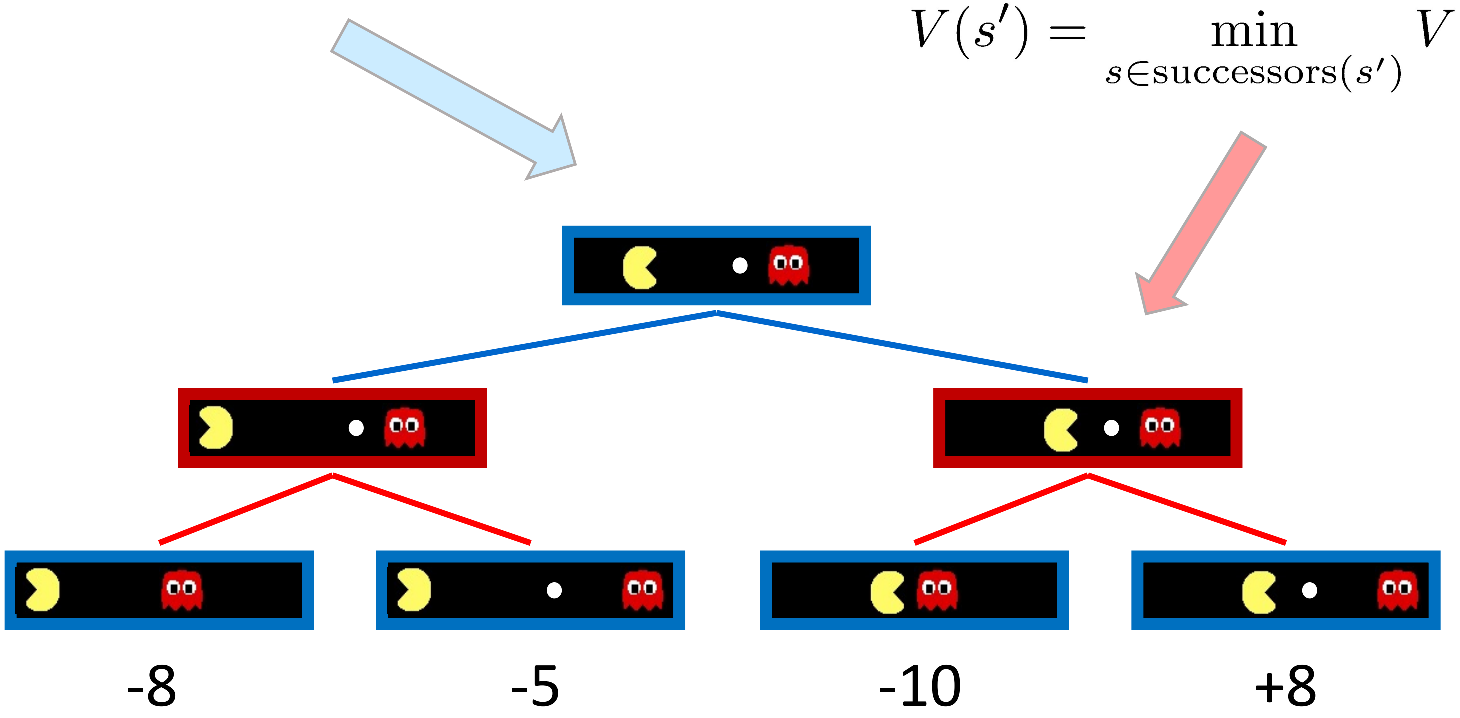
# Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

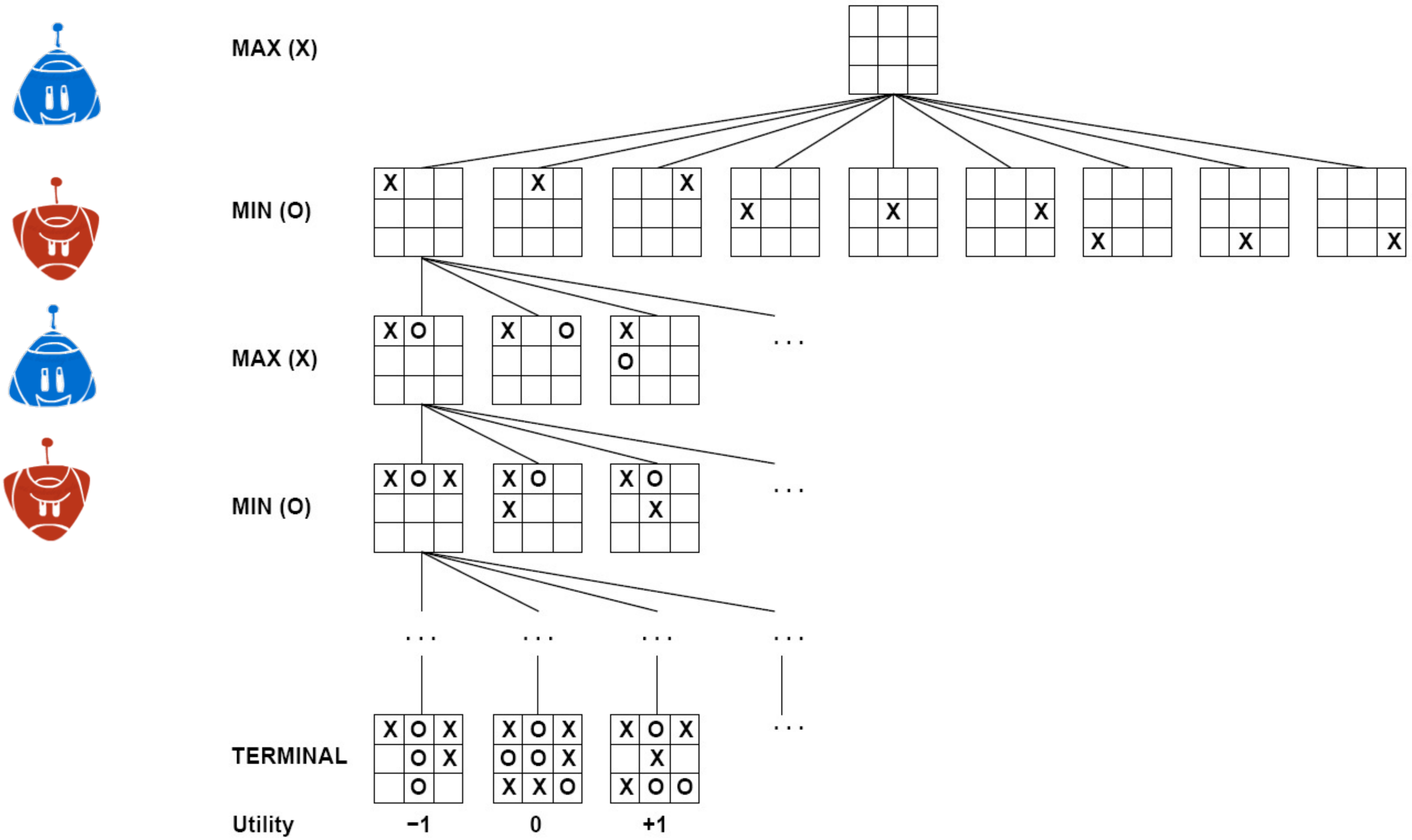
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$




Terminal States:

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

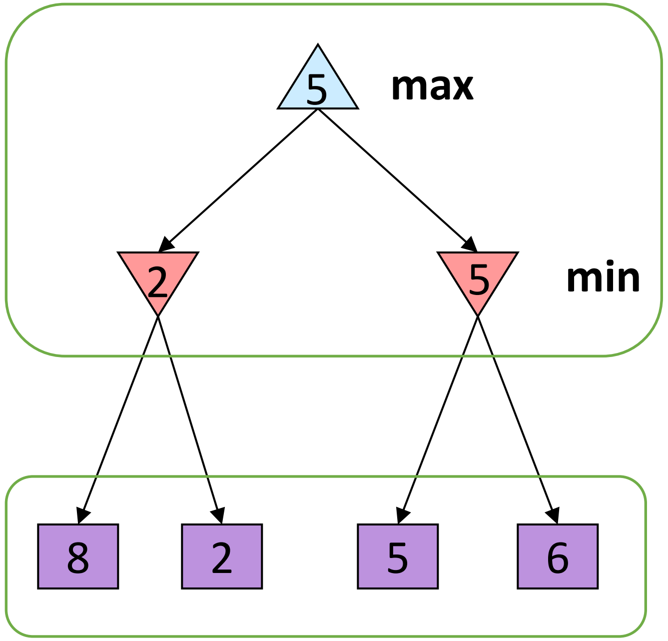
# Tic-Tac-Toe Game Tree



# Adversarial games

- Deterministic, zero-sum games:
  - Tic-tac-toe, chess
  - One player maximizes result
  - The other minimizes result

**Minimax values: computed recursively**




**Terminal values: part of the game**

- Minimax search:
  - A state-space search tree
  - Players alternate turns
  - Compute each node's minimax value: the best achievable utility against a rational (optimal) adversary

# Minimax Implementation

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

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

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

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

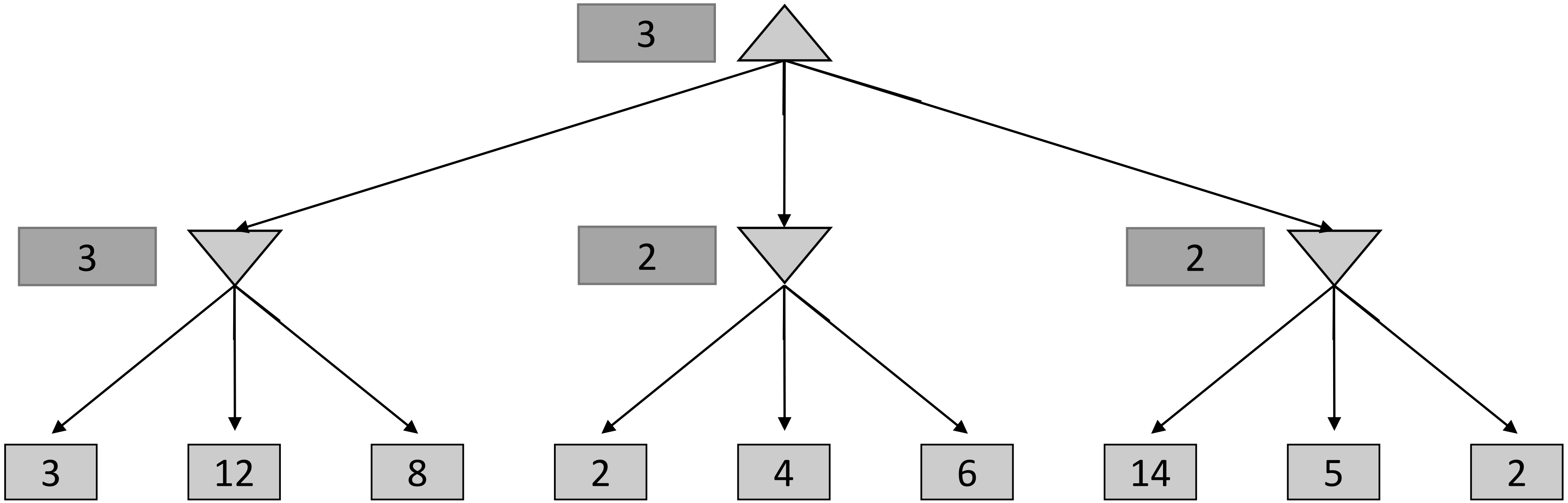
# Minimax Implementation (Dispatch)

```
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 Example




1. Value inference : use minimax search to identify the value for each node
2. Action generation : generate the path based on the value

# Outline

- Type of Game
- Adversarial Search
- Evaluation Function

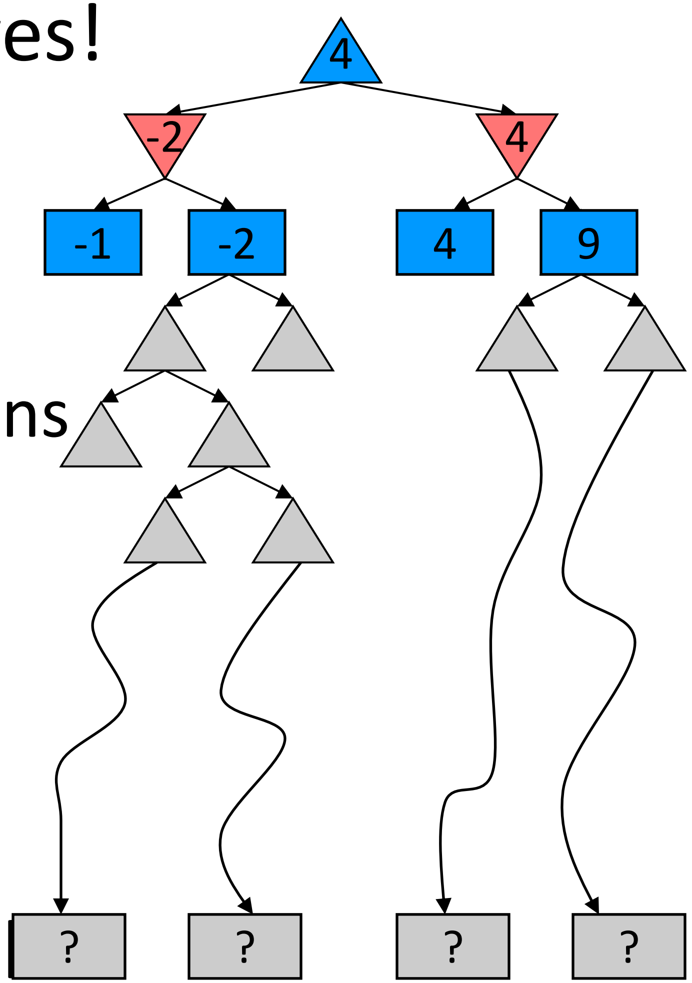
# Minimax Efficiency

- 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
  - $8^{140}$years for a computer that can process 1,000,000 nodes per second
  - But, do we need to explore the whole tree?

# Depth-limited Search with Evaluation Function

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Search only to a limited depth in the tree
  - Need an **evaluation function** for non-terminal positions
- Guarantee of optimal play is gone
- More steps forward makes a BIG difference
- How to determine an appropriate depth for search
  - Use iterative deepening



# Evaluation Functions

- Evaluation functions score non-terminals in depth-limited search
- Linear combination of various factors

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

Eval(s) = material + mobility + king-safety + center-control

material = $10^{100}$(K – K′) + 9(Q – Q′) + 5(R – R′)+ 3(B – B′ + N – N′) + 1(P – P′)

mobility = 0.1(num-legal-moves – num-legal-moves′)

- Table-based evaluation function
  - Compute values for states in advance and store them in a table
  - Too many states to compute. Can be done for the starting and the ending.

- Machine-learning based evaluation function

# Issues about Evaluation Functions

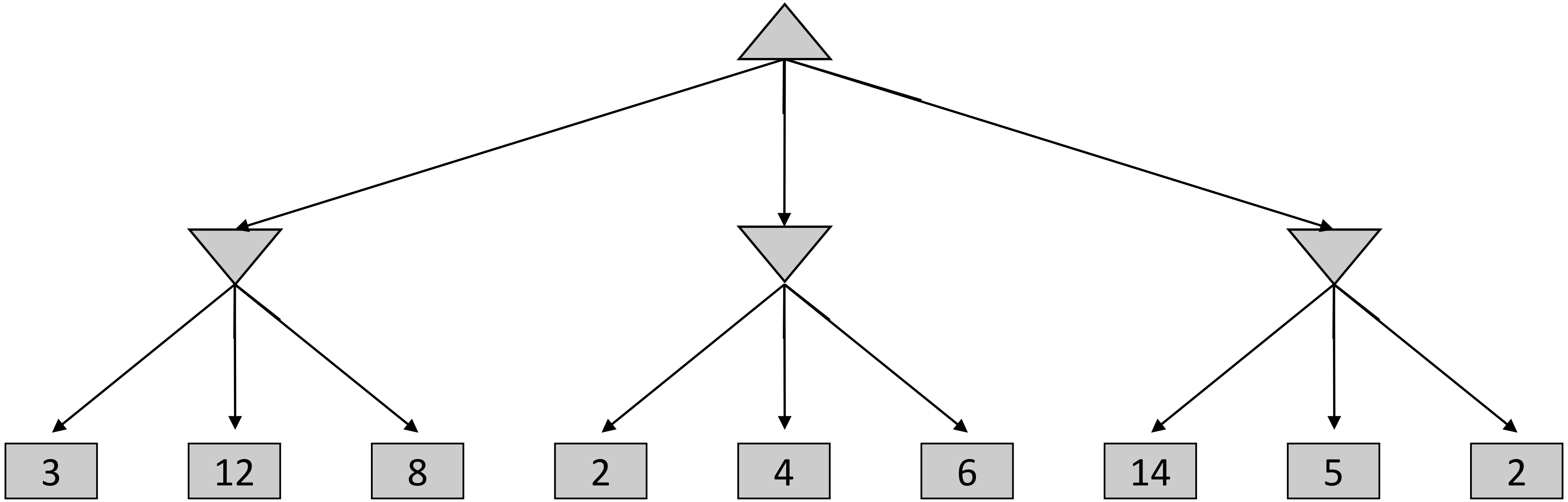
- ✓ Principles for designing evaluation function
  - ✓ Utility for a win state should be higher than a tie.  (correct)
  - ✓ The computation of evaluation function should be efficient. (quick)
  - ✓ It should be related to the chance of wining the game. (consistent)

- **It takes time to compute the evaluation function.**
  - An important example of the tradeoff between **complexity of features** and **complexity of computation**
  - Ideal function: returns the actual minimax value of the position

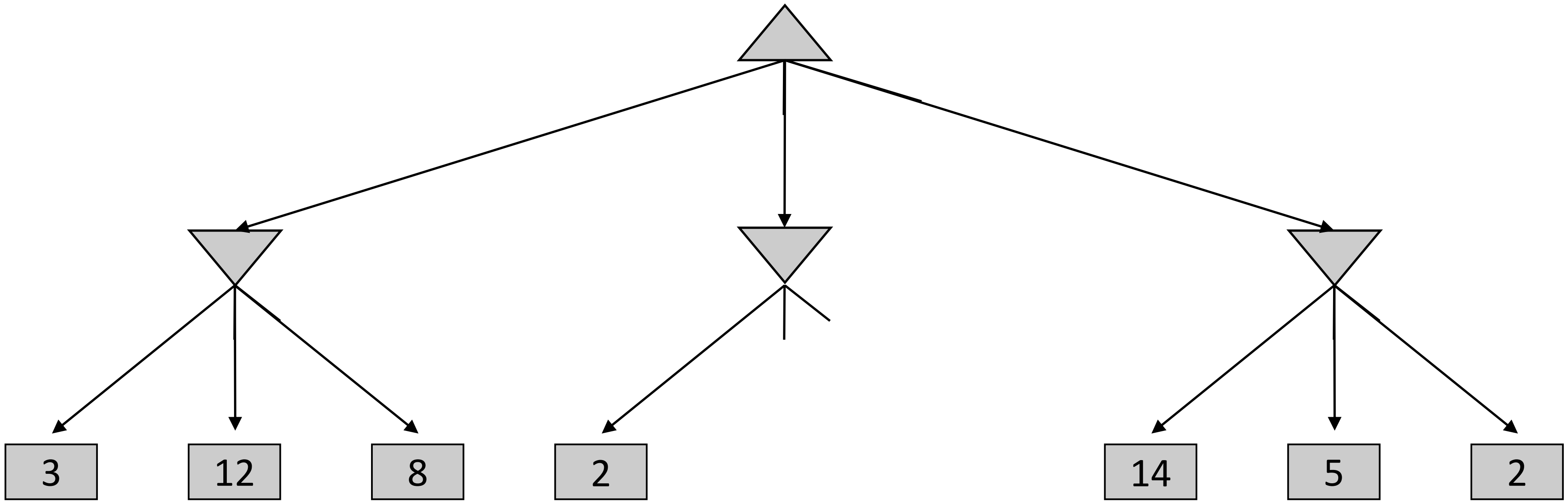
# Outline

- Type of Game
- Adversarial Search
- Evaluation Function
- Game Tree Pruning

# Minimax Example



# Minimax Pruning

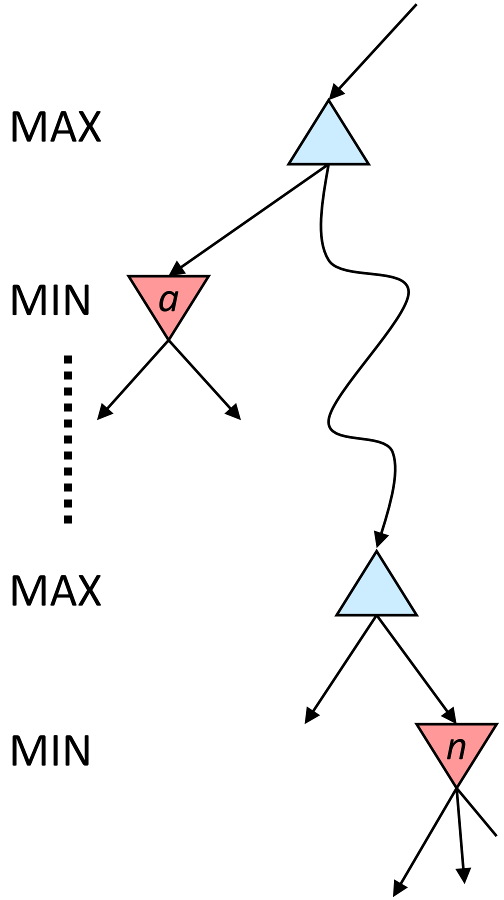


# Alpha-Beta Pruning for MIN node

- Suppose we have a MAX node as root and it has **the best value *a***

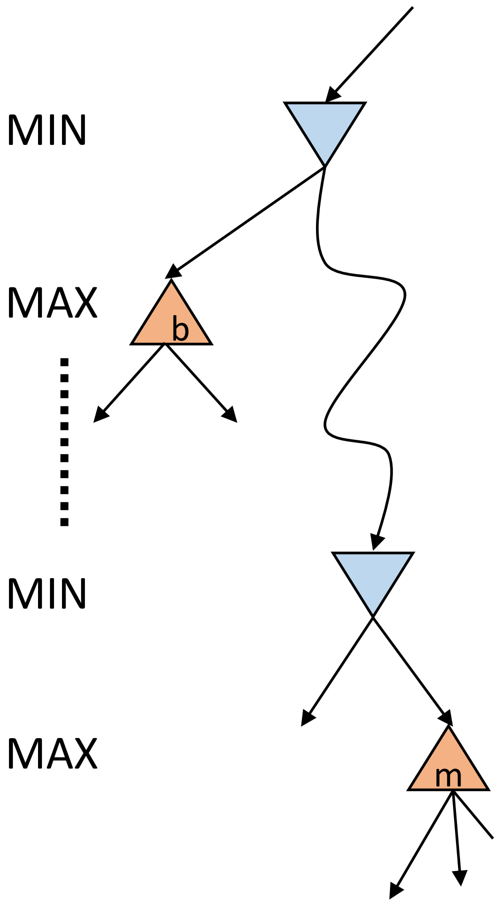
- We're computing the MIN-VALUE at some node *n*
- We're looping over *n*'s children
- ***n*'s estimate of the children's min is decreasing**

- **If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*'s other children**



# Alpha-Beta Pruning for MAX node

- Suppose we have a MIN node as root and it has **the best value *b***
- We're computing the MAX-VALUE at some node *m*
- We're looping over *m*'s children
- ***m*'s estimate of the children's' max is increasing**
- **If *m* becomes bigger than *b*, MIN will avoid it, so we can stop considering *m*'s other children**



# Alpha-Beta Pruning

- Set a range of value for each node with two parameters
  - [Alpha, Beta]
  - Obtain the initial value from parent node

- Update Process
  - Alpha: update in a max node, the best value it can achieve
  - Beta: update in a min node, the least value it can achieve

- Pruning Process
  - MAX node: if the current value is higher than beta, prune the subtree
  - MIN node: if the current value is lower than alpha, prune the subtree

# 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:
update of v           v = max(v, value(successor, α, β))
update of alpha       α = max(α, v)
pruning               if v ≥ β return v
        return v
```

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

- **Update alpha in MAX node, and prune subtree of a MAX node using beta**
- **Update beta in MIN node, and prune subtree of a MIN node using alpha**

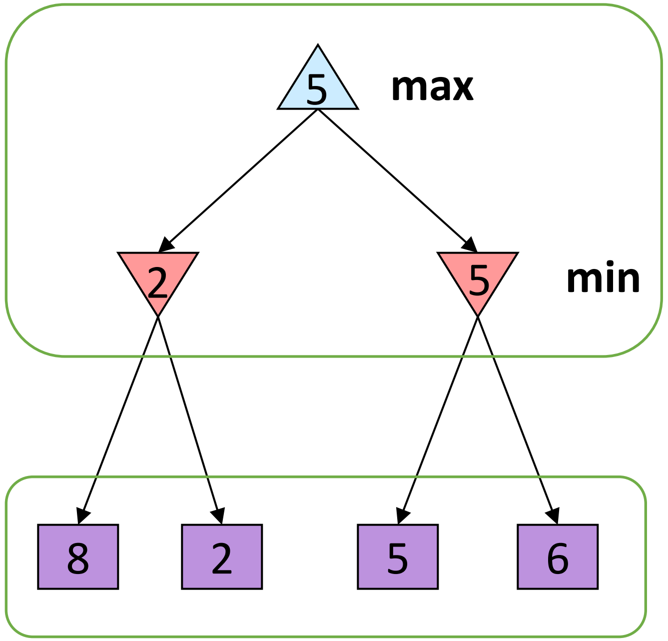
# Adversarial Search (Minimax)

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

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

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

**Figure 5.7** The alpha–beta search algorithm. Notice that these routines are the same as the MINIMAX functions in Figure 5.3, except for the two lines in each of MIN-VALUE and MAX-VALUE that maintain $\alpha$ and $\beta$ (and the bookkeeping to pass these parameters along).

- Alpha-Beta-Search (state): the entry of the search; State is the root of the search
- MAX-VALUE(state, $\alpha$, $\beta$): compute the value for a MAX node; state is the current state; $\alpha$ is the MAX's best option on path to root; $\beta$ is the MIN's best option on path to root.
- MIN-VALUE(state, $\alpha$, $\beta$): compute the value for a MIN node
- ACTIONS (state): get all the possible actions for state
- RESULT (state, action): get the successor state for current state

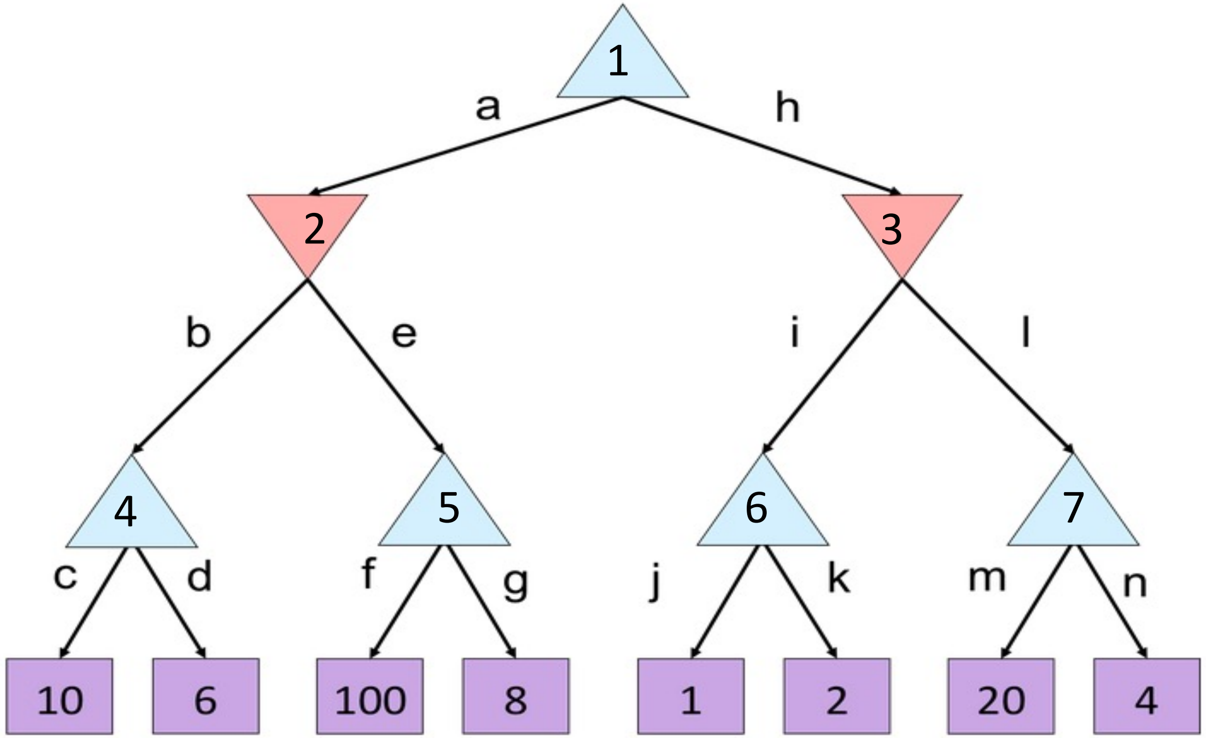
**Minimax values: computed recursively**




**Terminal values: part of the game**

# In class exercise

| node | v | Alpha | beta |
|---|---|---|---|
| 1 | | | |
| 2 | | | |
| 3 | | | |
| 4 | | | |
| 5 | | | |
| 6 | | | |
| 7 | | | |




1. **Which edges are pruned?**
2. **Write down the value, alpha and beta for each node after processing.**

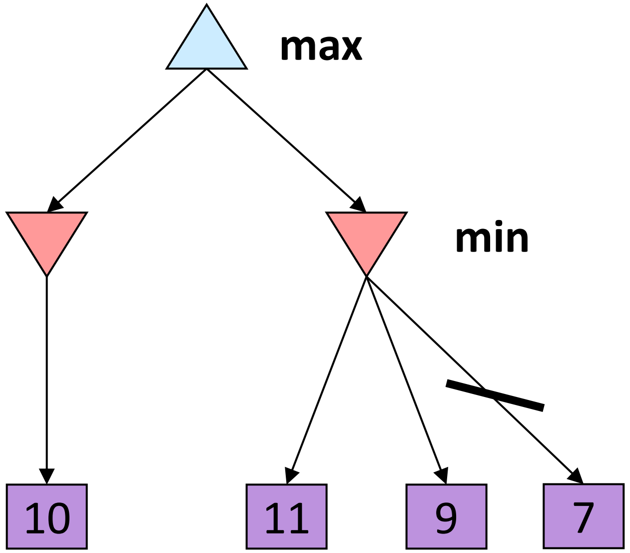
```
def max-value(state, α, β):
    If terminal (state),
        return value
    initialize v = -∞
    for each successor of state:
        v = max(v, value(successor, α, β))
        α = max(α, v)
        if v ≥ β return v
    return v
```

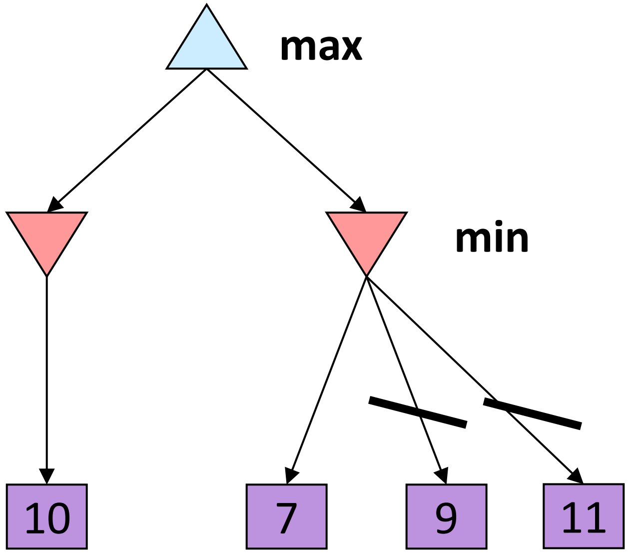
```
def min-value(state , α, β):
    If terminal (state),
        return value
    initialize v = +∞
    for each successor of state:
        v = min(v, value(successor, α, β))
        β = min(β, v)
        if v ≤ α return v
    return v
```

# Alpha-Beta Pruning Properties

- This pruning has no effect on minimax value computed for the root!
- Minimax values of intermediate nodes might be wrong

- Ordering Matters
  - Good child ordering improves effectiveness of pruning
  - MAX node needs decreasing order of children nodes
  - MIN node needs increasing order of children nodes

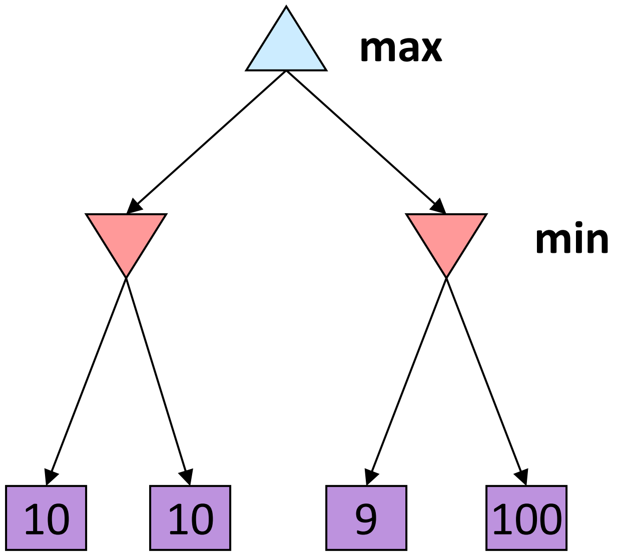


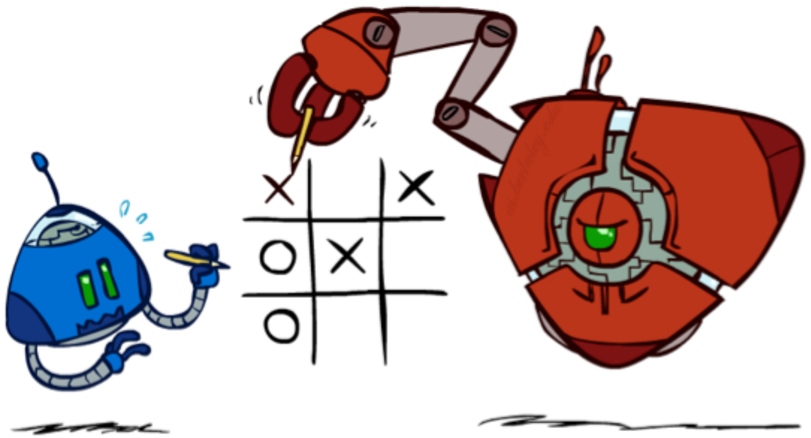



# Outline

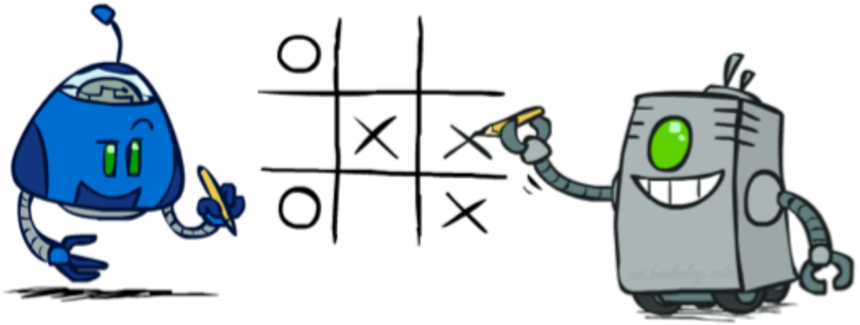
- Type of Game
- Adversarial Search
- Evaluation Function
- Game Tree Pruning
- Uncertain Outcomes

# Minimax Properties

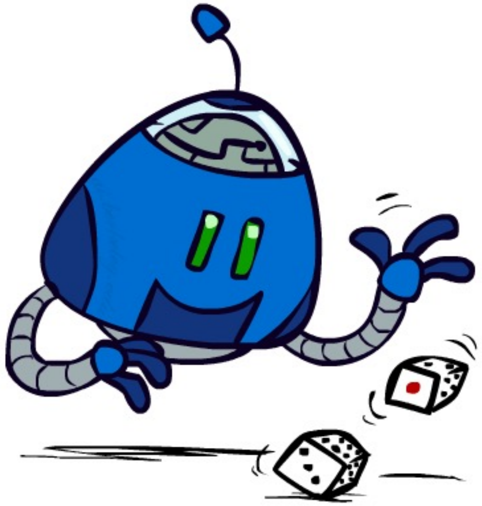


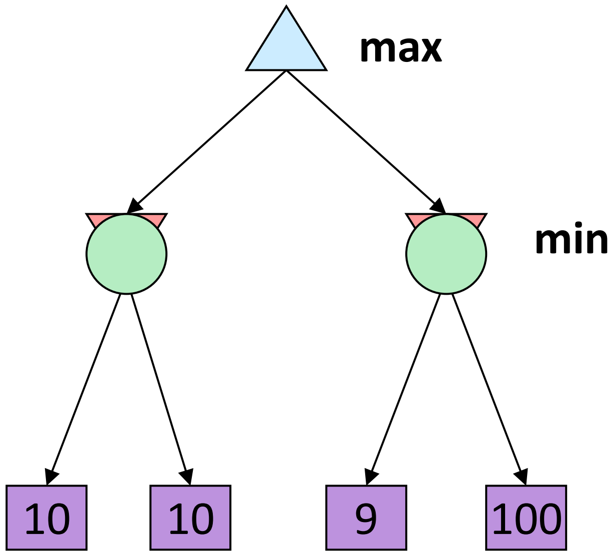



**Optimal against a perfect player.** Otherwise?



# Worst-Case vs. Average Case






Idea: Uncertain outcomes controlled by chance.

# Reminder: Probabilities

- A random variable represents an event whose outcome is unknown
- A probability distribution is an assignment of weights to outcomes

- Some laws of probability:
  - Probabilities are always non-negative
  - Probabilities over all possible outcomes sum to one

- 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

# 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?
  - 20 mins if there **no traffic**.
  - 30 mins if there is **light traffic**.
  - 60 mins if there is **heavy traffic**.
  - P(T=none) = 0.25, P(T=light) = 0.50, P(T=heavy) = 0.25

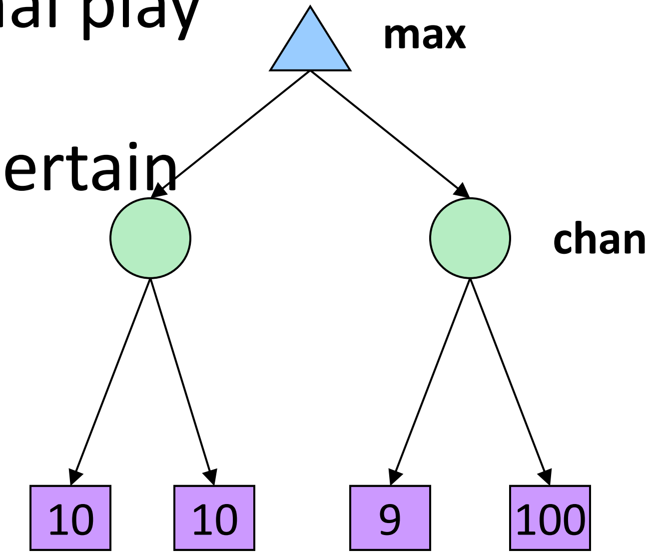
20 min x 0.25 + 30 min x 0.50 + 60 min x 0.25 ⇒ 35 min

# Expectimax 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
  - I.e. take weighted average (expectation) of children



# Expectimax Pseudocode

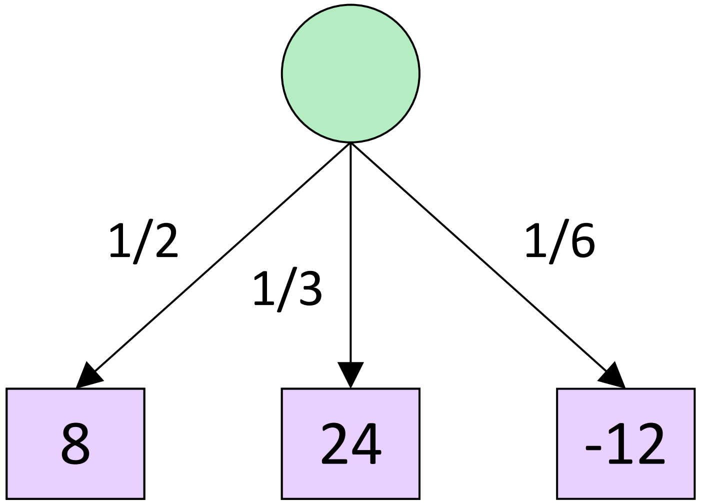
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
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
```

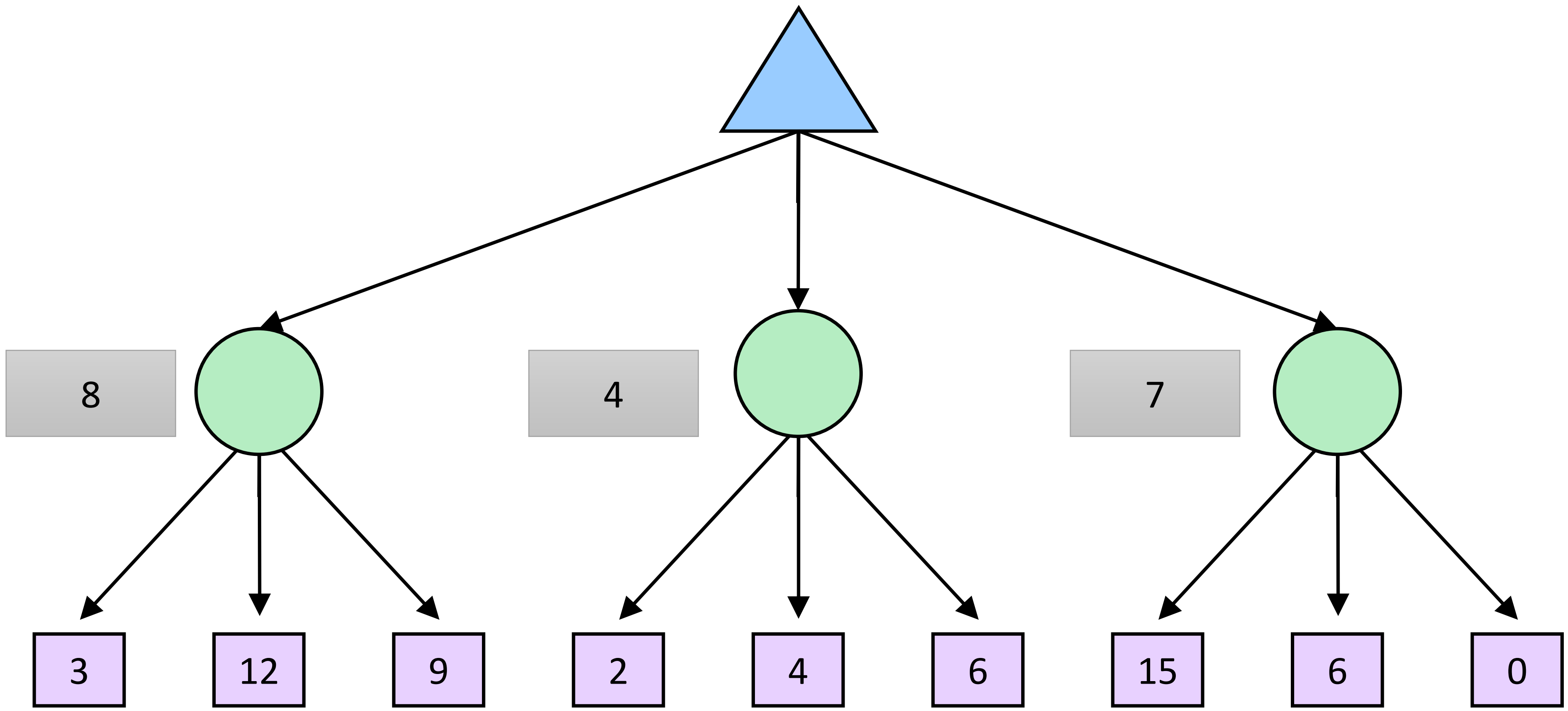
# Expectimax Pseudocode

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

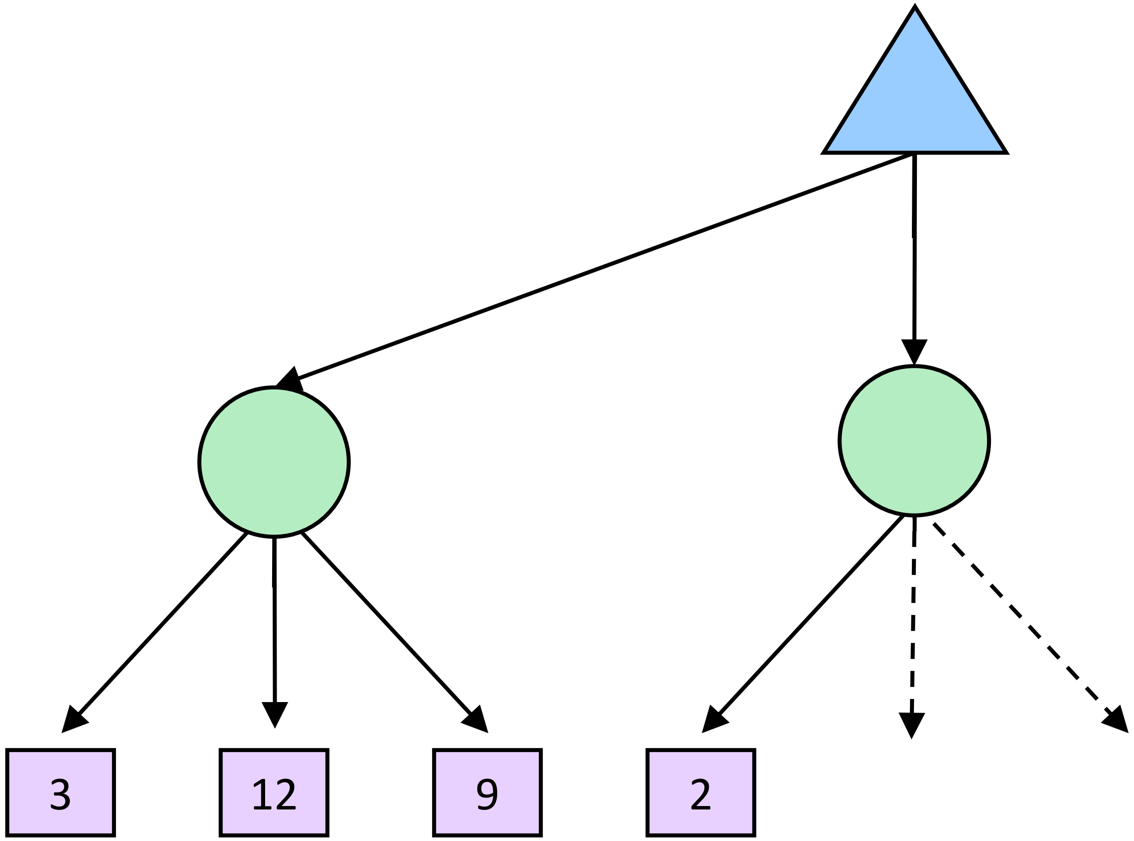



$$v = (1/2)\,(8) + (1/3)\,(24) + (1/6)\,(-12) = 10$$

# Expectimax Example

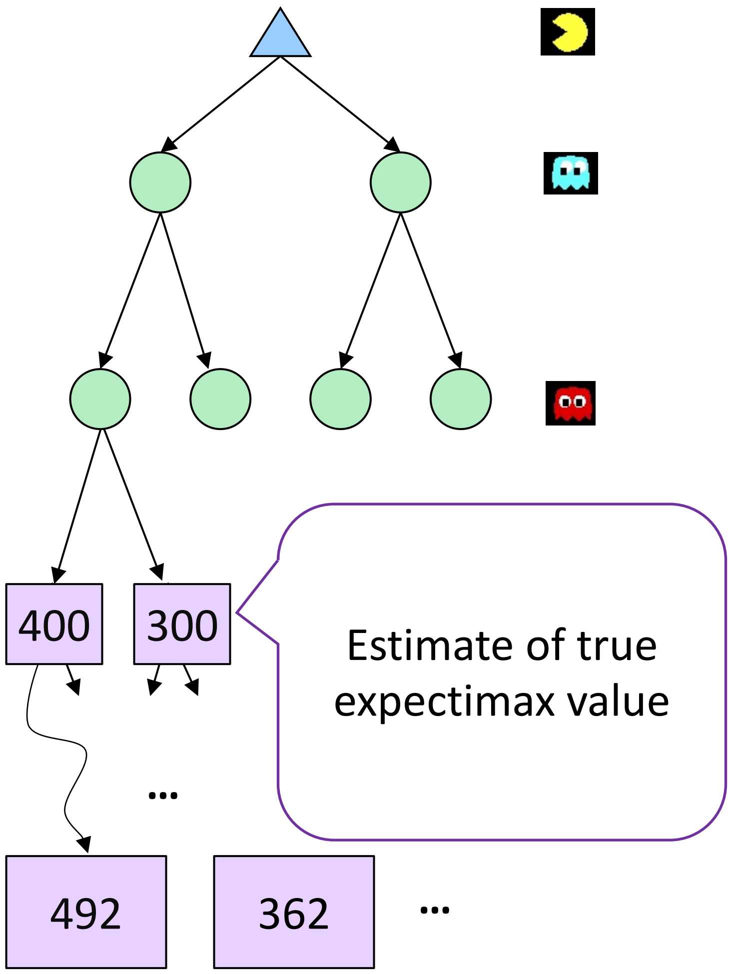


# Expectimax Pruning?




No Prunning!
All Children nodes are involved.

# Depth-Limited Expectimax



# What Probabilities to Use?

- We have a chance node for any outcome out of our control: opponent or environment

- A probabilistic model to describe how the opponent (or environment) will behave in any state

  - Model could be a simple uniform distribution (roll a die)

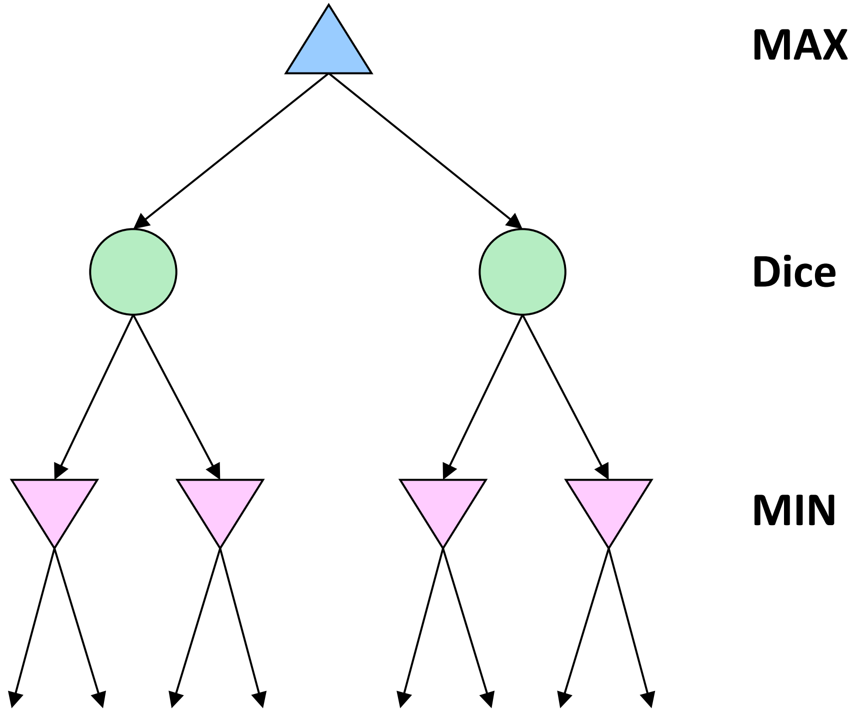
  - Model could be sophisticated and require a great deal of computation
    - Model the ghost based on its historical behaviors

# Outline

- Type of Game
- Adversarial Search
- Evaluation Function
- Game Tree Pruning
- Uncertain Outcomes
- Other Game Types

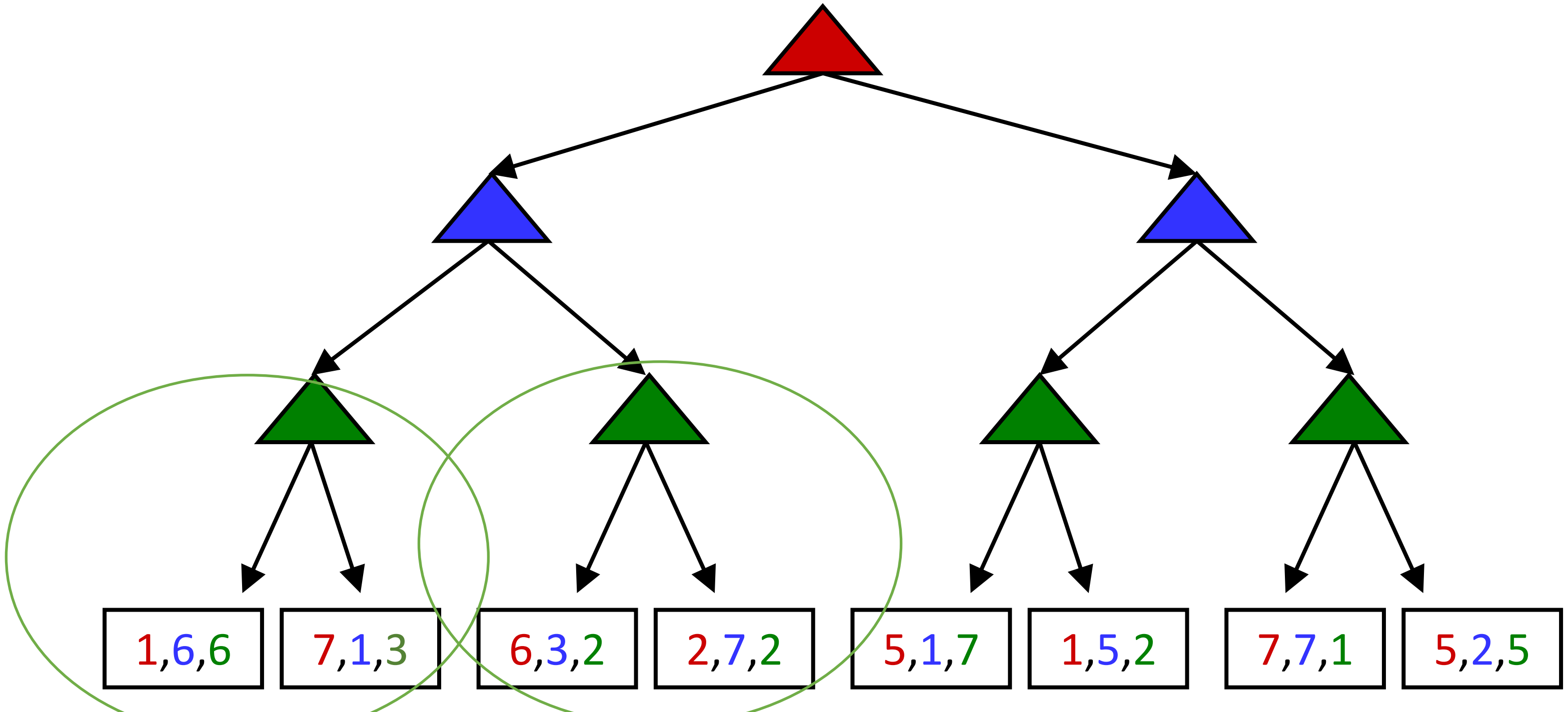
# Mixed Layer Types

- E.g. Monopoly with two participants
- Expectiminimax
  - Environment is an extra “random agent” player that moves after each min/max agent
  - Each node computes the appropriate combination of its children



# Multi-Agent Utilities

- 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.



# Outline

- Type of Game
- Adversarial Search
- Game Tree Pruning
- Uncertain Outcomes
- Other Game Types
- Utility

# Maximum Expected Utility

- Principle of maximum expected utility:
  - A rational agent should choose the action that maximizes its expected utility, given its knowledge

$$action = argmax\ ExpectedUtility(a|e)$$

- Questions:
  - Where do utilities come from?
  - How do we ensure the agent is rational with such guidance?

# Utilities

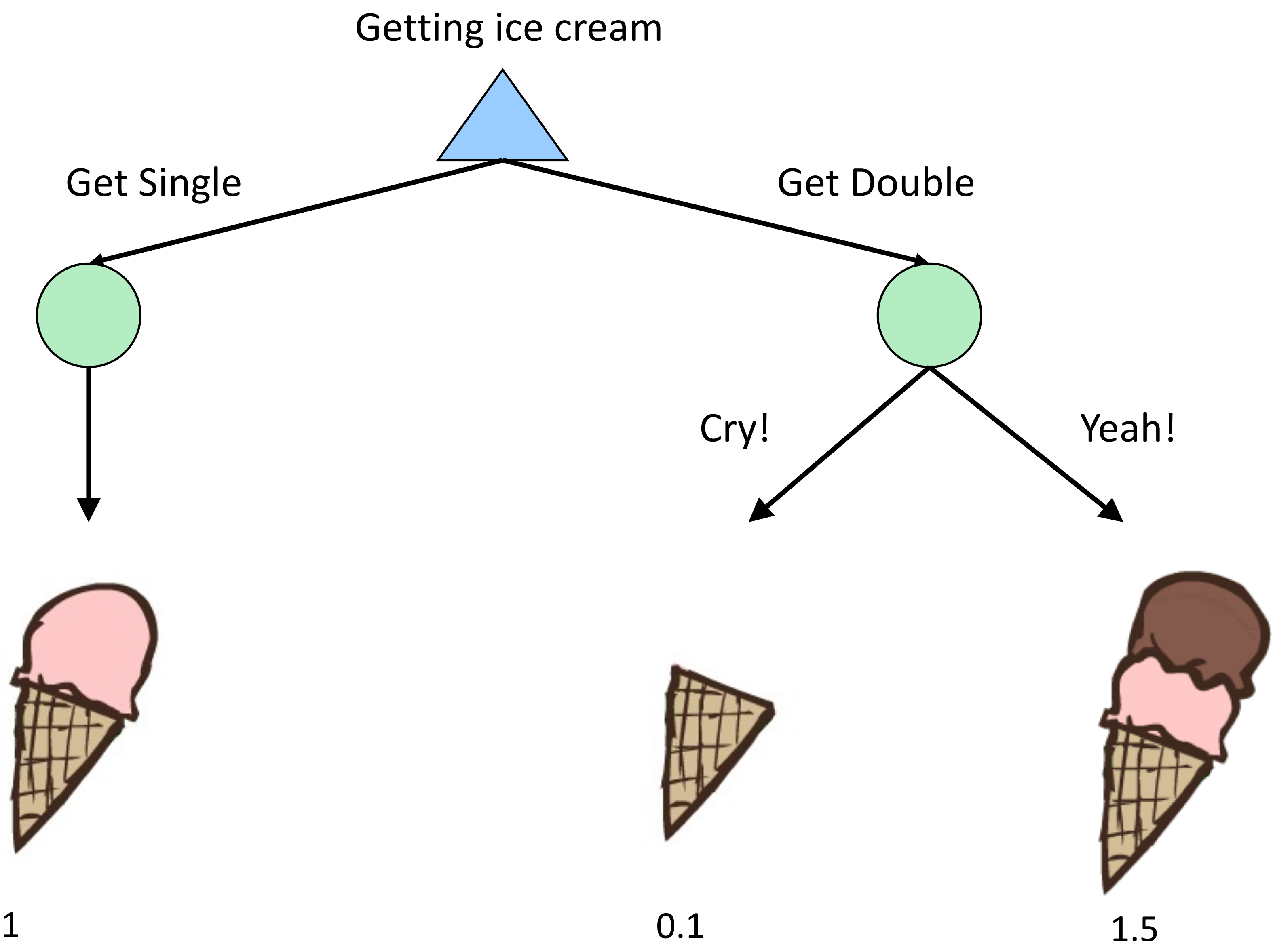
- Utilities are functions **from outcomes (states of the world) to real numbers** that describe an agent's **preferences**

- Where do utilities come from?
  - In a game, may be simple (+1/-1)
  - In general, utilities function summarizes the agent's preferences

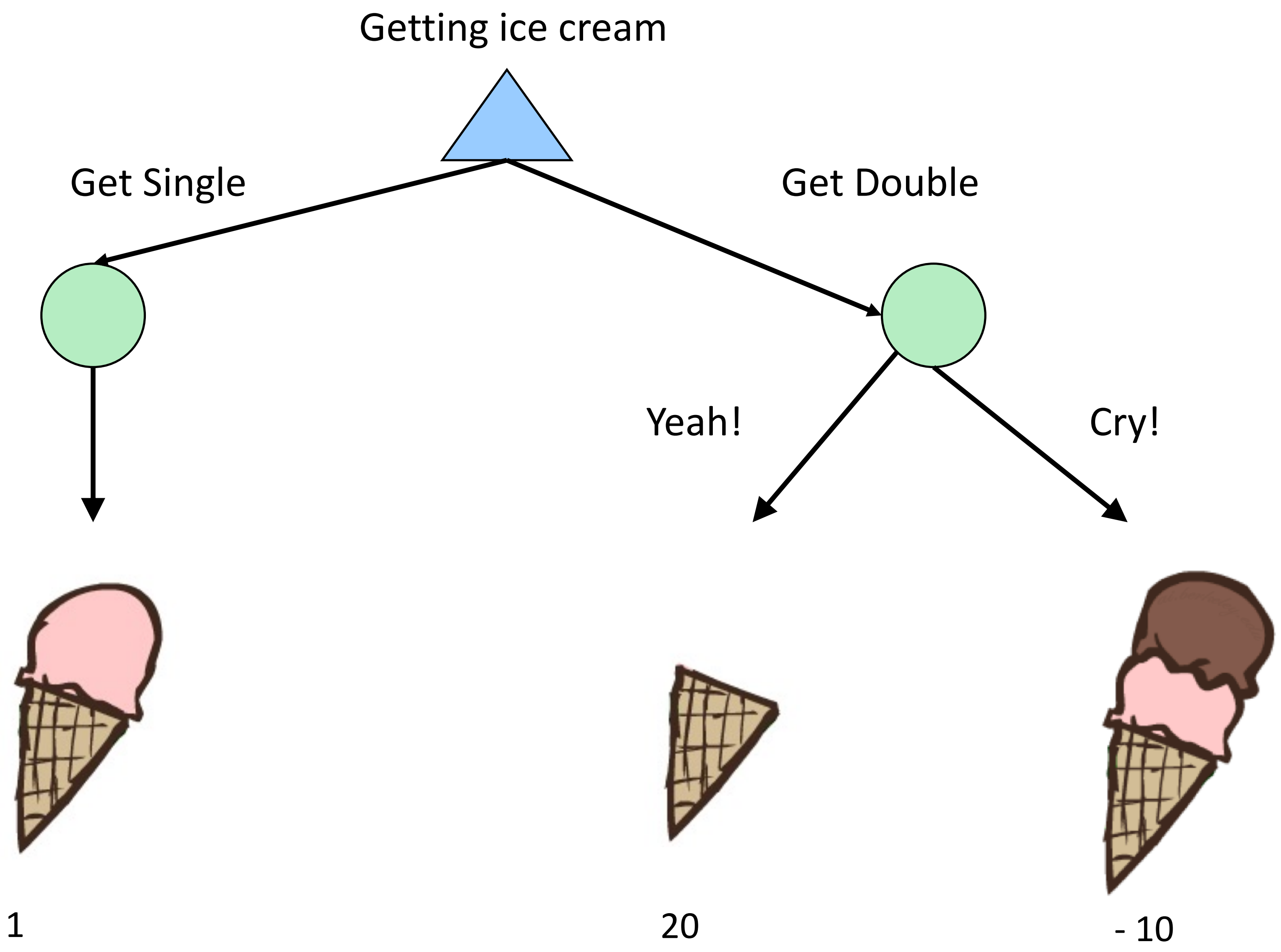
- In practice, we design a utility function and let behaviors emerge

- Why bother utility? Why not define behavior directly?
  - Note: an agent can be entirely rational (consistent with MEU) without representing or manipulating utilities and probabilities.
  - E.g., a lookup table for perfect tic-tac-toe, a reflex vacuum cleaner

# Utilities: Uncertain Outcomes



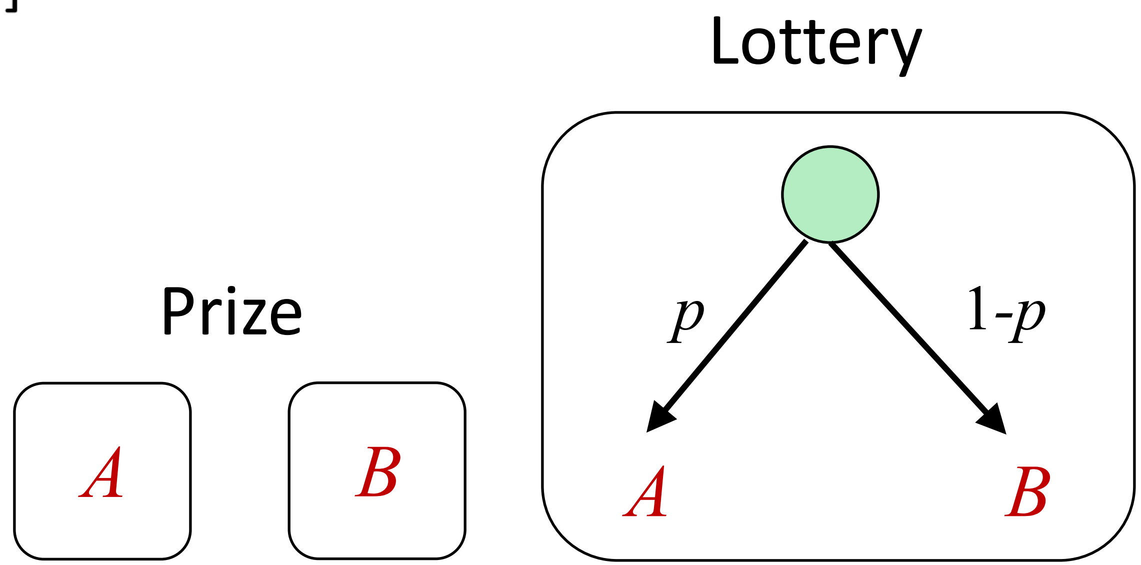
# Utilities: Uncertain Outcomes



# Preferences

- An agent must have preferences among:
  - Prize: $A$, $B$, etc.
  - Lotteries: situations with uncertain prizes
    - Each uncertain decision can be interpreted as a lottery

$$L = [p, A;\ (1-p), B]$$



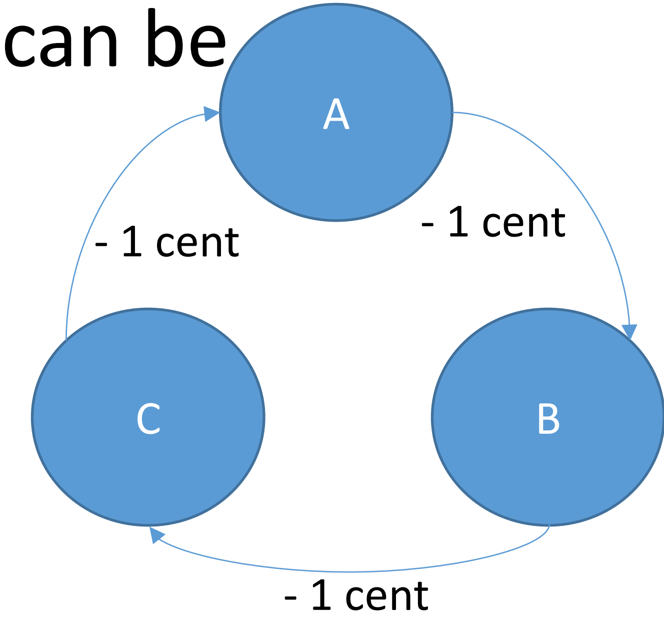

- Notation:
  - Preference: $A \succ B$
  - Indifference: $A \sim B$

# Rational Preferences

- We want some constraints on preferences before we call them rational, such as:

Axiom of Transitivity: $(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$

- For example: an agent with intransitive preferences can be induced to give away all its money
  - Say, C>B>A and A>C
  - If C > B, then an agent with B would pay 1 cent to get C
  - If B > A, then an agent with A would pay 1 cent to get B
  - If A > C, then an agent with C would pay 1 cent to get A



# Rational Preferences

## The Axioms of Rationality

Orderability

$$(A \succ B) \vee (B \succ A) \vee (A \sim B)$$

Transitivity

$$(A \succ B) \wedge (B \succ C) \Rightarrow (A \succ C)$$

Continuity

$$A \succ B \succ C \Rightarrow \exists p \; [p, A; \; 1-p, C] \sim B$$

Substitutability

$$A \sim B \Rightarrow [p, A; \; 1-p, C] \sim [p, B; 1-p, C]$$

Monotonicity

$$A \succ B \Rightarrow$$
$$(p \geq q \Leftrightarrow [p, A; \; 1-p, B] \succeq [q, A; \; 1-q, B])$$

Theorem: Rational preferences imply behavior describable as maximization of expected utility → Rationality!

# MEU Principle

- Theorem [Ramsey, 1931; von Neumann & Morgenstern, 1944]
  - Given any preferences satisfying these constraints, there exists a real-valued function U such that:

$$U(A) \geq U(B) \;\Leftrightarrow\; A \succeq B$$

$$U([p_1, S_1;\; \ldots\; ;\; p_n, S_n]) = \sum_i p_i U(S_i)$$

  - I.e. values assigned by U preserve preferences of both prizes and lotteries
- Maximum expected utility (MEU) principle:
  - Choose the action that maximizes expected utility

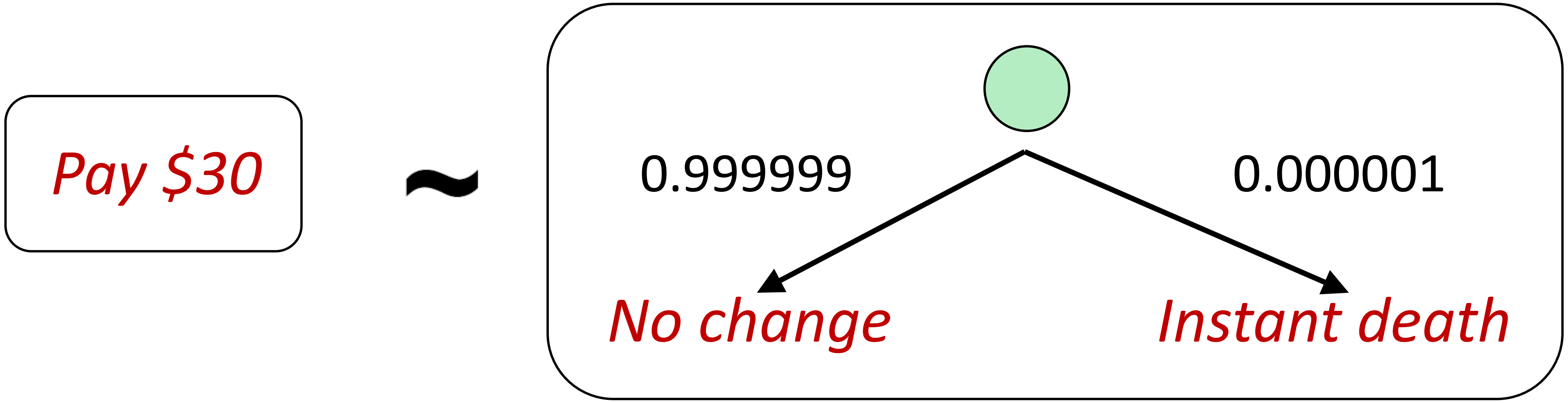
- **Rational Preferences → Rational Utility → Rational Agent**

# Outline

- Type of Game
- Adversarial Search
- Game Tree Pruning
- Uncertain Outcomes
- Other Game Types
- Utility
- Human Utility

# Human Utilities

- Normalized utilities: $u_+ = 1.0$, $u_- = 0.0$
- Utilities map states to real numbers.
- Standard approach to assessment of human utilities:
  - Compare a prize A to a standard lottery $L_p$ between
    - "best possible prize" $u_+$
    - "worst possible outcome" $u_-$
  - Adjust lottery probability p until indifference: $A \sim L_p$



# Utility of your life

- Micromorts: one-millionth chance of death, useful for paying to reduce product risks, etc.
  - 1000 dollar for safety airbag (reduce the Micromorts from 1000 miles to 6000 miles)

## Micromort examples

- QALYs (quality adju ... ırs, useful for medical decisio

| Death from | Micromorts per exposure |
|---|---|
| Scuba diving | 5 per dive |
| Skydiving | 7 per jump |
| Base-jumping | 430 per jump |
| Climbing Mt. Everest | 38,000 per ascent |

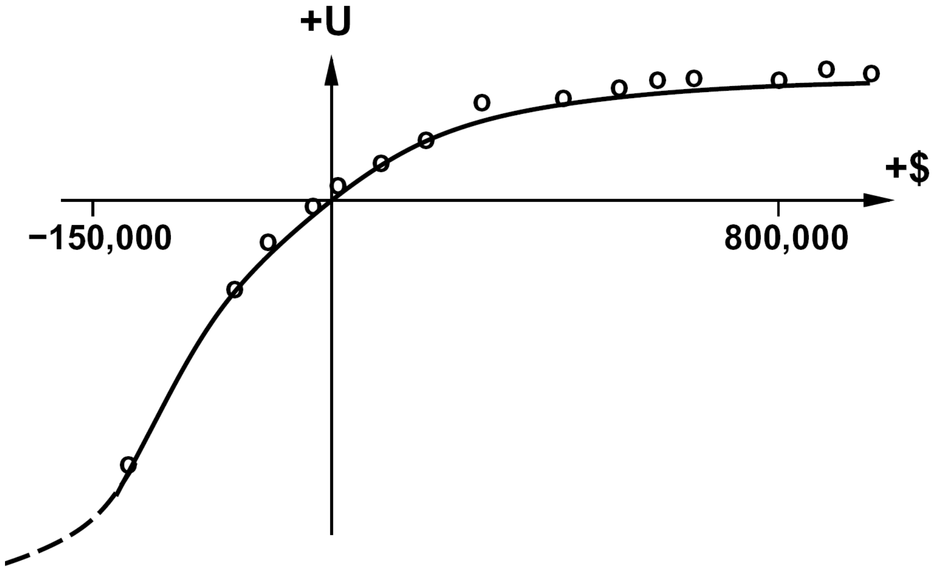
QALYs (quality adjusted life year):
quality-adjusted life years, useful for medical decisions involving

| 1 Micromort | |
|---|---|
| Train travel | 6000 miles |
| Jet | 1000 miles |
| Car | 230 miles |
| Walking | 17 miles |
| Bicycle | 10 miles |
| Motorbike | 6 miles |

# Money as Utility

- We can use having money (or being in debt) as the the utility.
- Given a lottery L = [p, $X; (1-p), $Y]
  - The expected monetary value EMV(L) is p*X + (1-p)*Y
  - U(L) = p*U($X) + (1-p)*U($Y)

  - Typically, U(L) < U( EMV(L) )
  - In this sense, people are risk-averse
  - When deep in debt, people are risk-seeking



# Example: Insurance

- Consider the lottery [0.5, $1000;  0.5, $0]
  - What is its expected monetary value?  ($500)
  - What is its certainty equivalent?
    - $400 for most people

- Difference of $100 is the insurance
  - There's an insurance industry because people will pay to reduce their risk
  - If everyone were risk-neutral, no insurance needed!

- It's win-win: you'd rather have the $400 and the insurance company would rather have the lottery (they have many lotteries)

# Example: Human Rationality?

- Famous example of Allais (1953)
  - A: [0.8, $4k;    0.2, $0]
  - B: [1.0, $3k;    0.0, $0]

  - C: [0.2, $4k;    0.8, $0]
  - D: [0.25, $3k;    0.75, $0]