Lecture 5: Search with Other Agents

Shuai Li

John Hopcroft Center, Shanghai Jiao Tong University

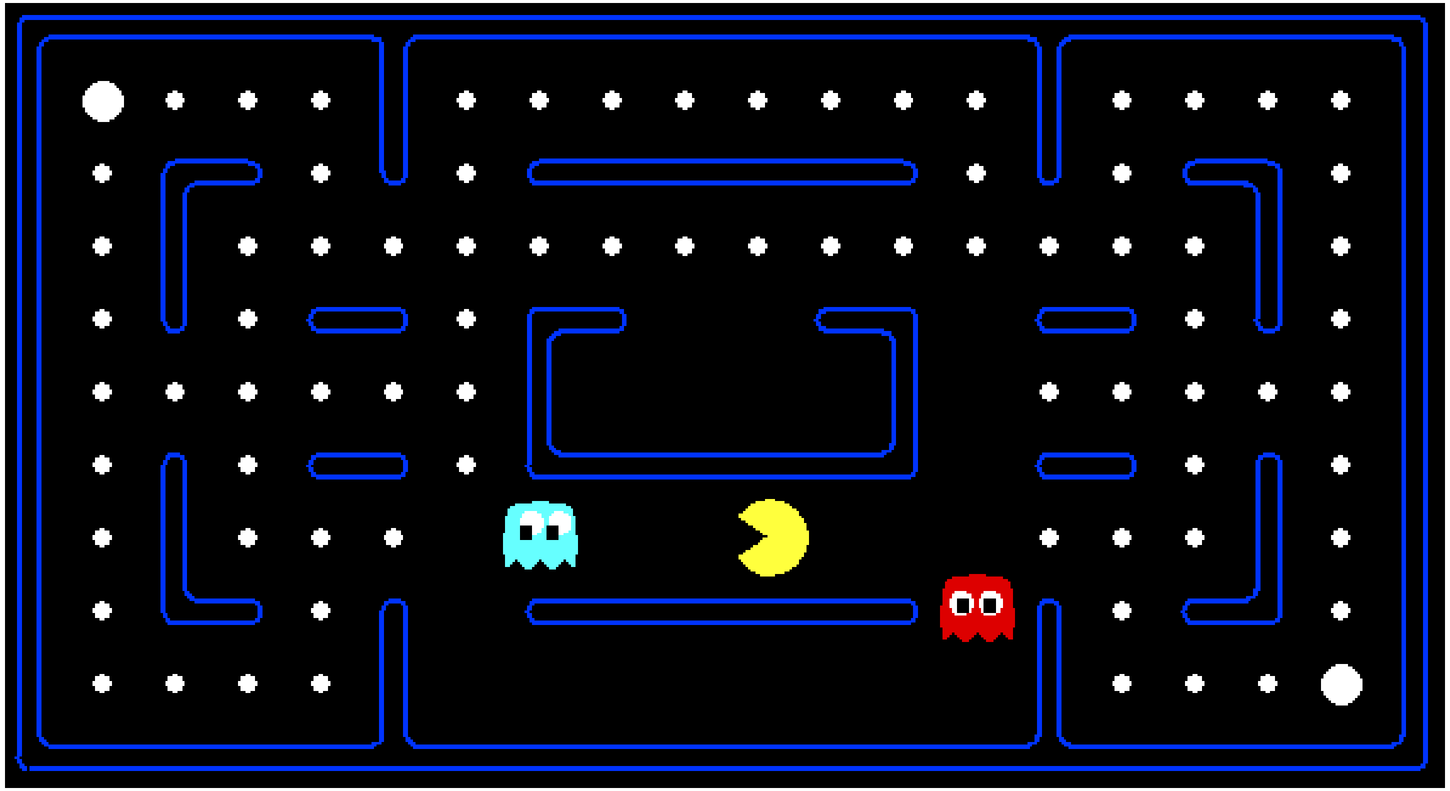
https://shuaili8.github.io

https://shuaili8.github.io/Teaching/CS410/index.html

Part of slide credits: CMU AI & http://ai.berkeley.edu

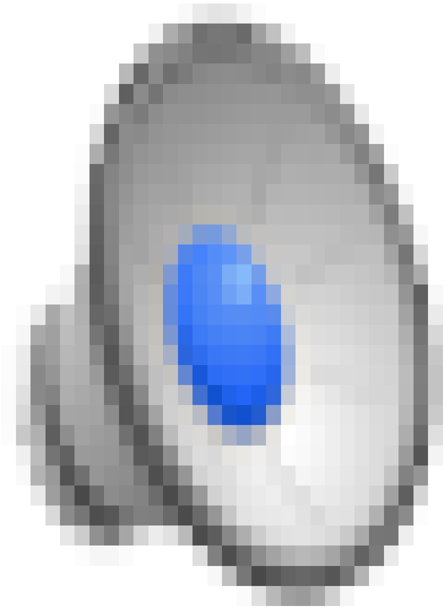
Game Type

Behavior from Computation



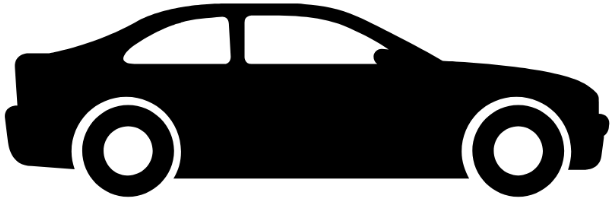
[Demo: mystery pacman (L6D1)]

Video of Demo Mystery Pacman



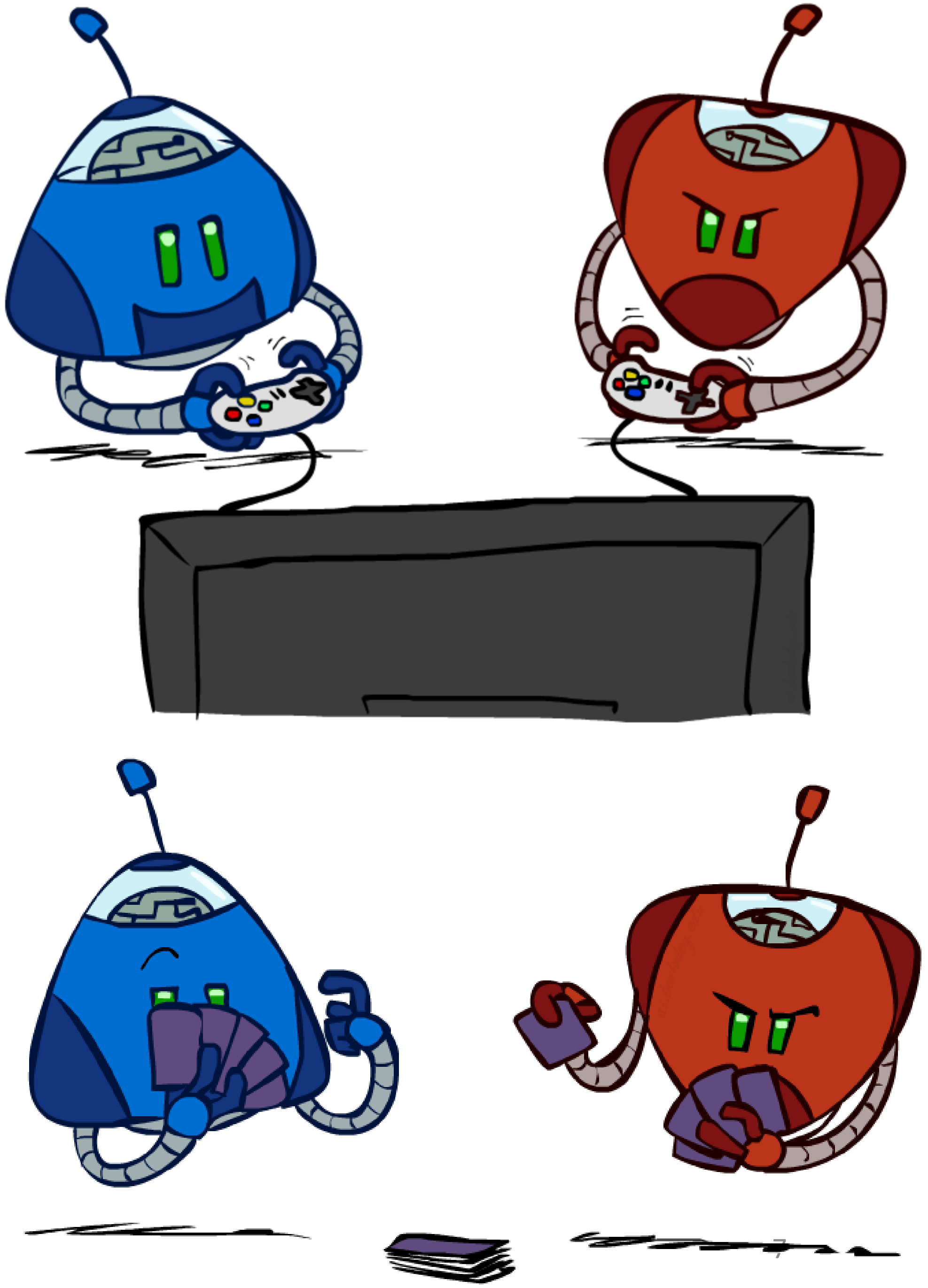
Agents Getting Along with Other Agents

Agents Getting Along with Humans



Types of Games

- Many different kinds of games!
- Axes:
  - Deterministic or stochastic?
  - One, two, or more players?
  - Zero sum?
  - Perfect information (can you see the state)?
- Want algorithms for calculating a ***contingent plan*** (a.k.a. strategy or policy) which recommends a move for every possible eventuality



History of Game AI

1956 checkers

1992 backgammon

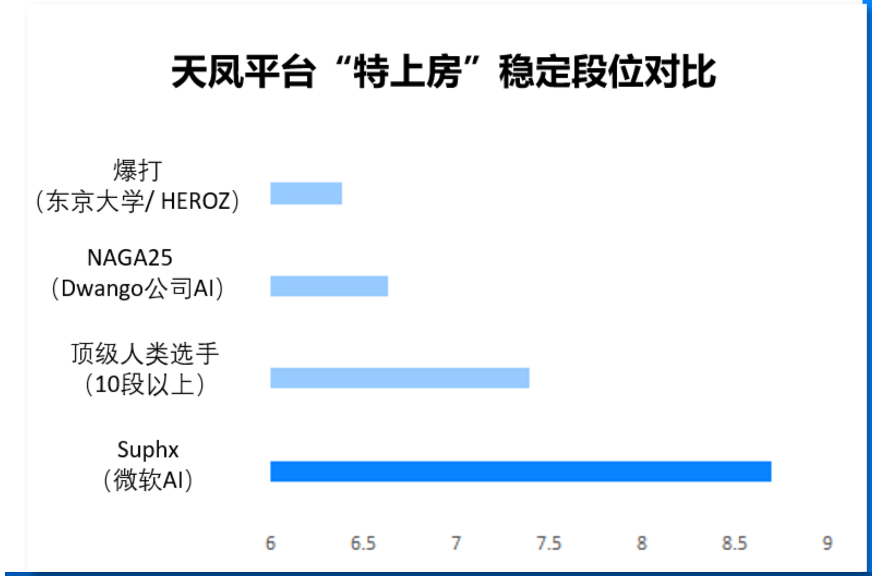
1994 checkers

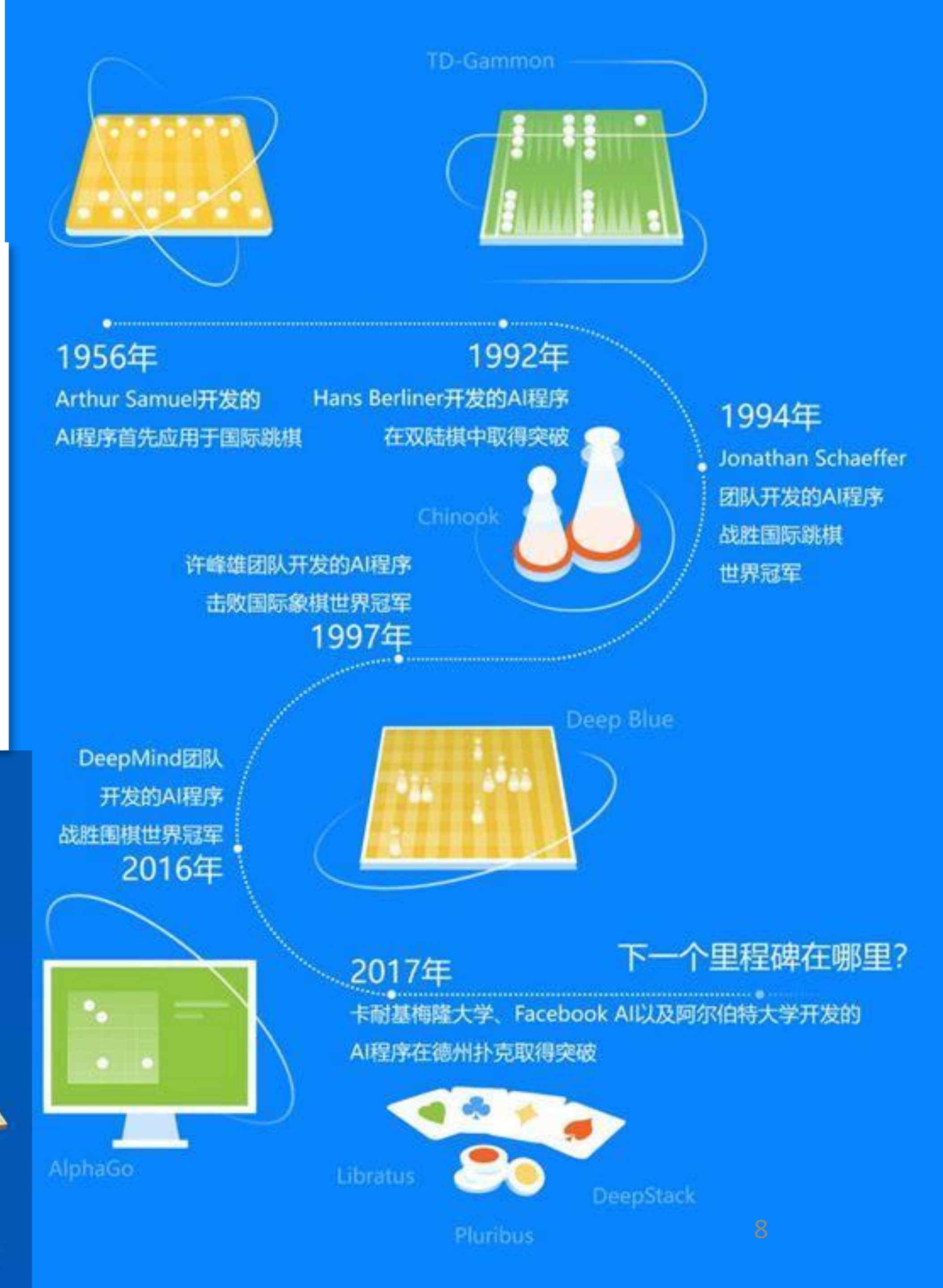
1997 chess

2016 Go

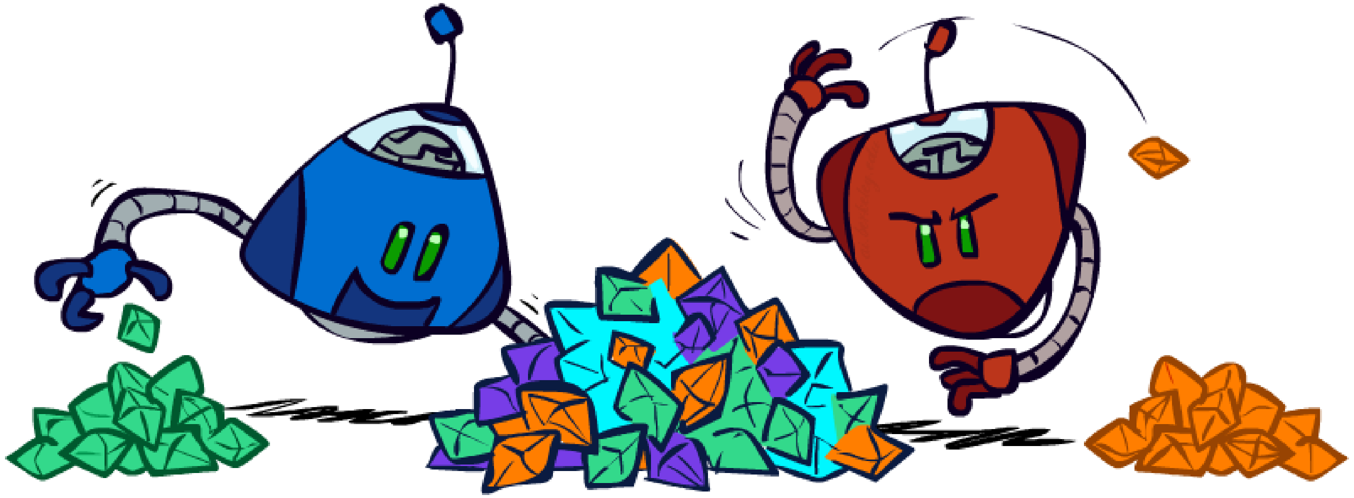
2017 Texas hold’em

2019 Majiang

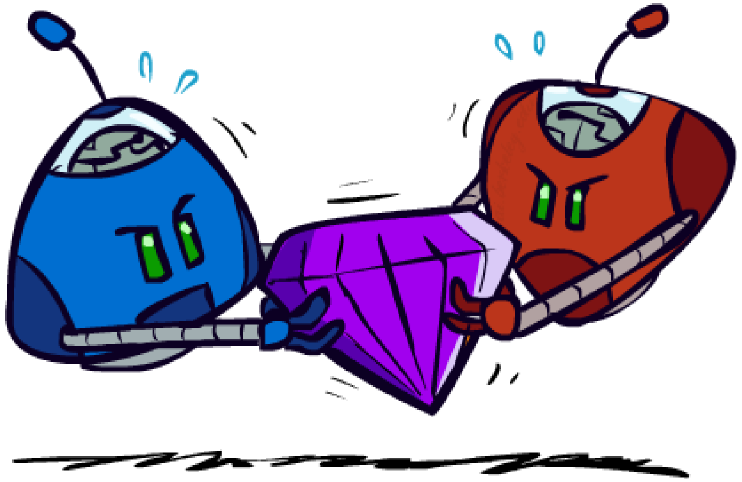




Types of Games 2

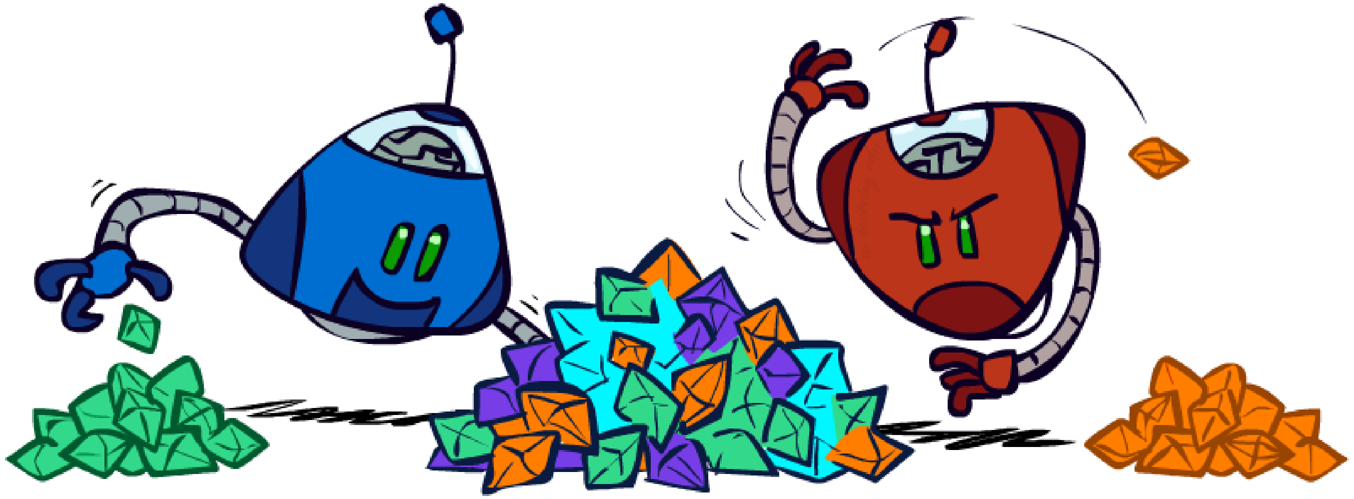


- General Games
  - Agents have independent utilities (values on outcomes)
  - Cooperation, indifference, competition, shifting alliances, and more are all possible
    - We don't make AI to act in isolation, it should a) work around people and b) help people
    - That means that every AI agent needs to solve a game



- Zero-Sum Games
  - Agents have opposite utilities (values on outcomes)
  - Lets us think of a single value that one maximizes and the other minimizes
  - Adversarial, pure competition

Types of Games 3



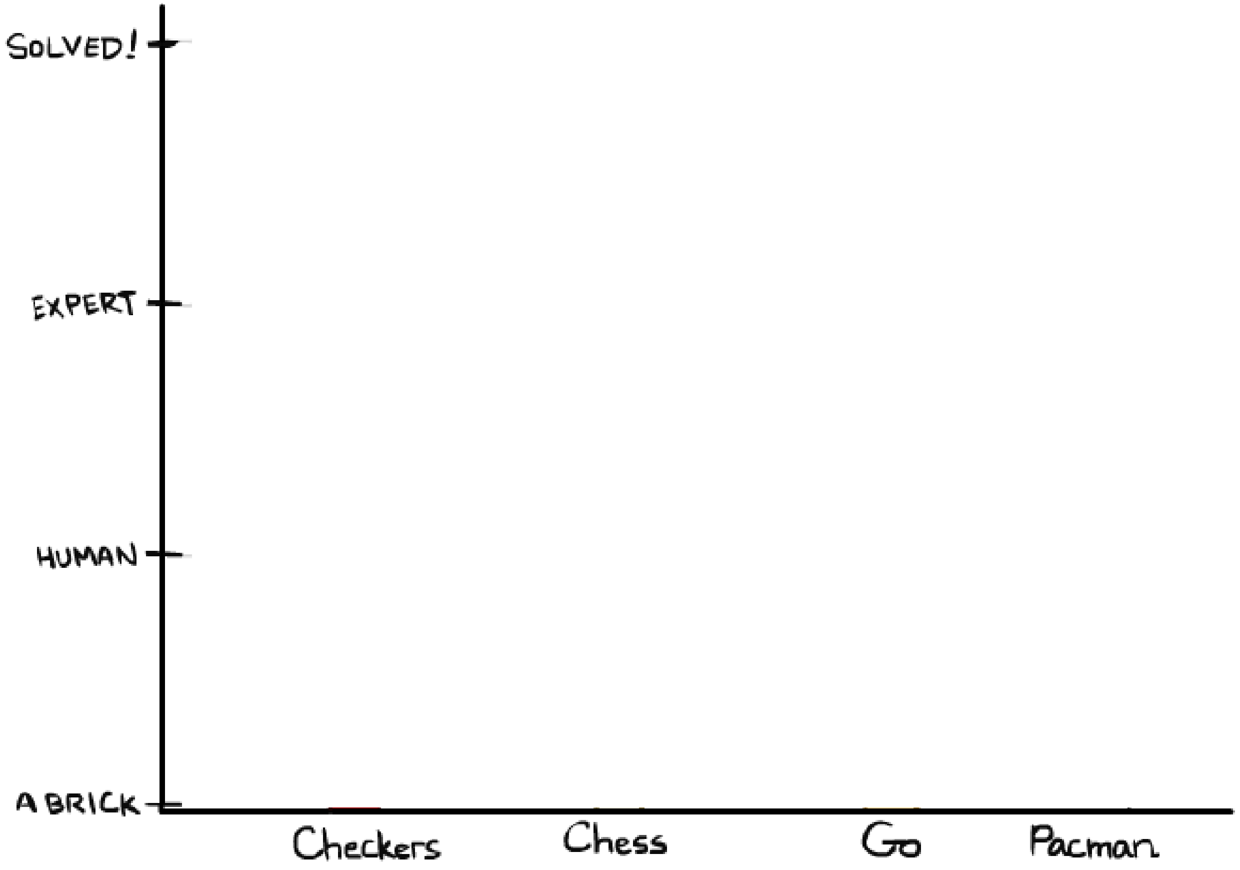
- Common payoff games
  - Discussion: Use a technique you've learned so far to solve one!

Zero-Sum Games

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!

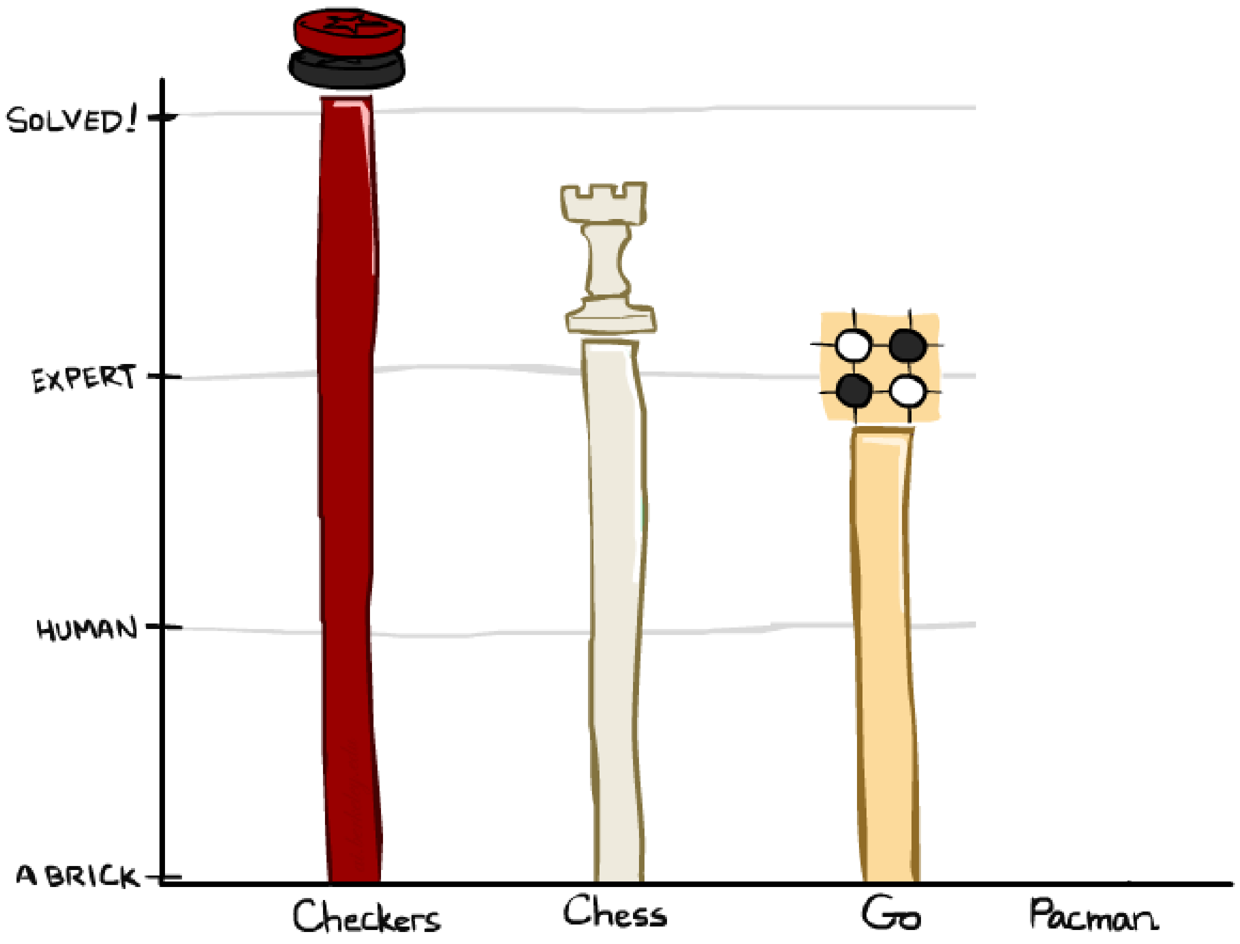
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go: 2016: Alpha GO defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function.**

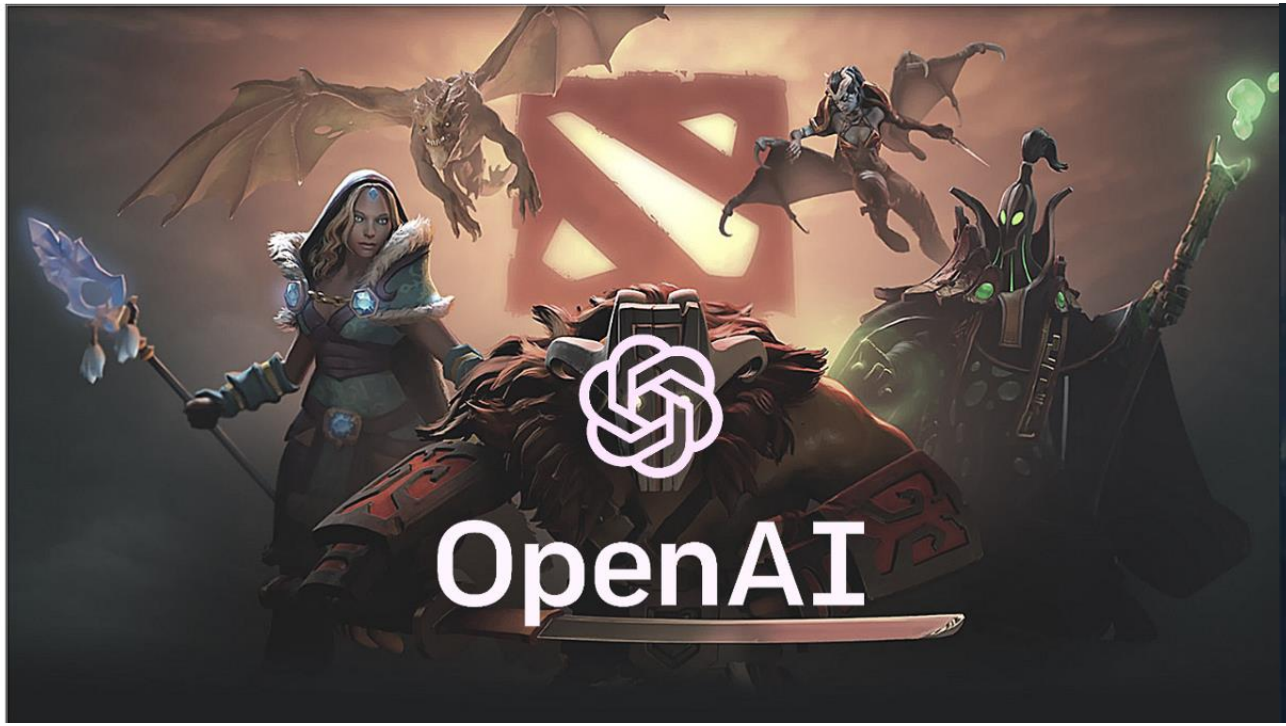


Zero-Sum Games 2

- **Checkers:** 1950: First computer player. 1994: First computer champion: Chinook ended 40-year-reign of human champion Marion Tinsley using complete 8-piece endgame. 2007: Checkers solved!
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.
- **Go :2016: Alpha GO defeats human champion. Uses Monte Carlo Tree Search, learned evaluation function.**
- **Pacman**

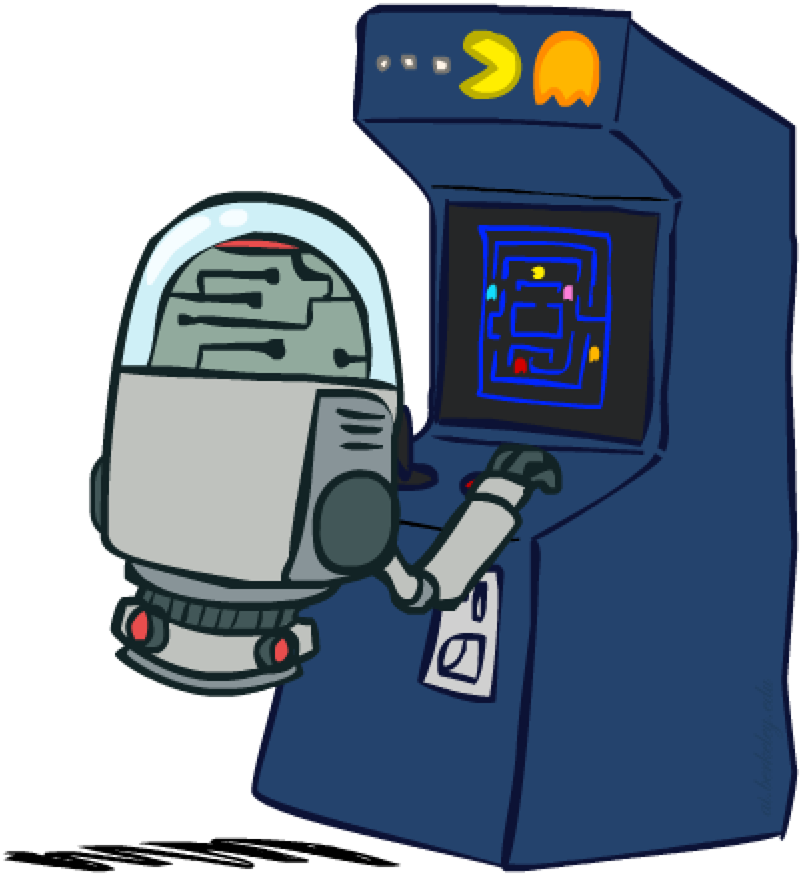


Game playing – state of the art

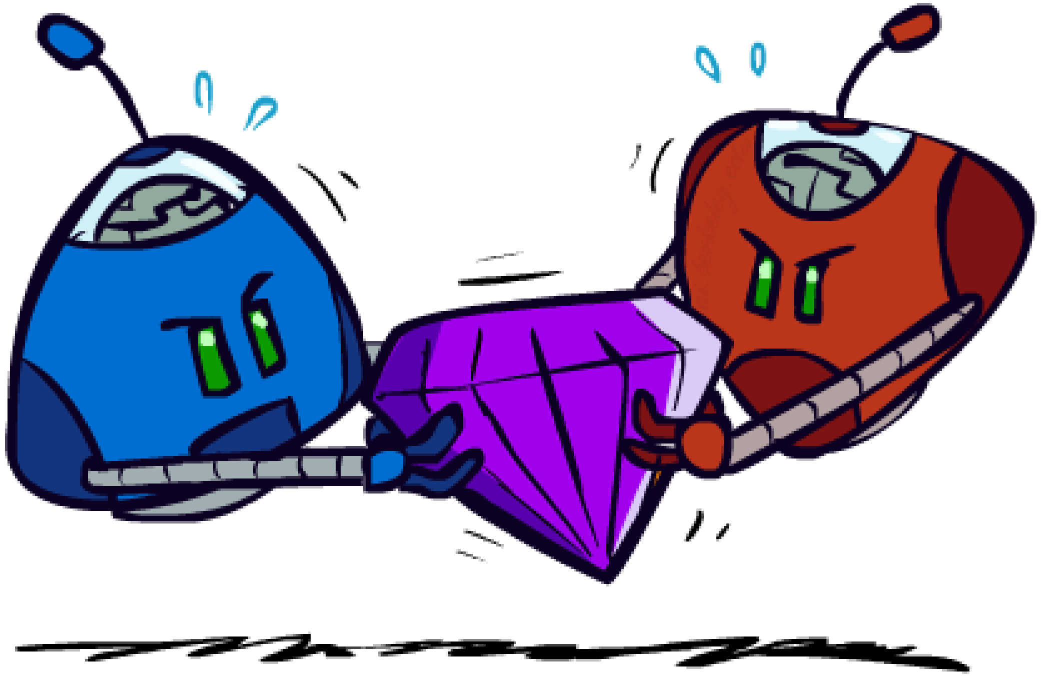


"Standard" Games

- Standard games are deterministic, observable, two-player, turn-taking, zero-sum
- Game formulation:
  - States: S (start at s_0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: SxA $\rightarrow$ S
  - Terminal Test: S $\rightarrow$ {t,f}
  - Terminal Utilities: SxP $\rightarrow$ R

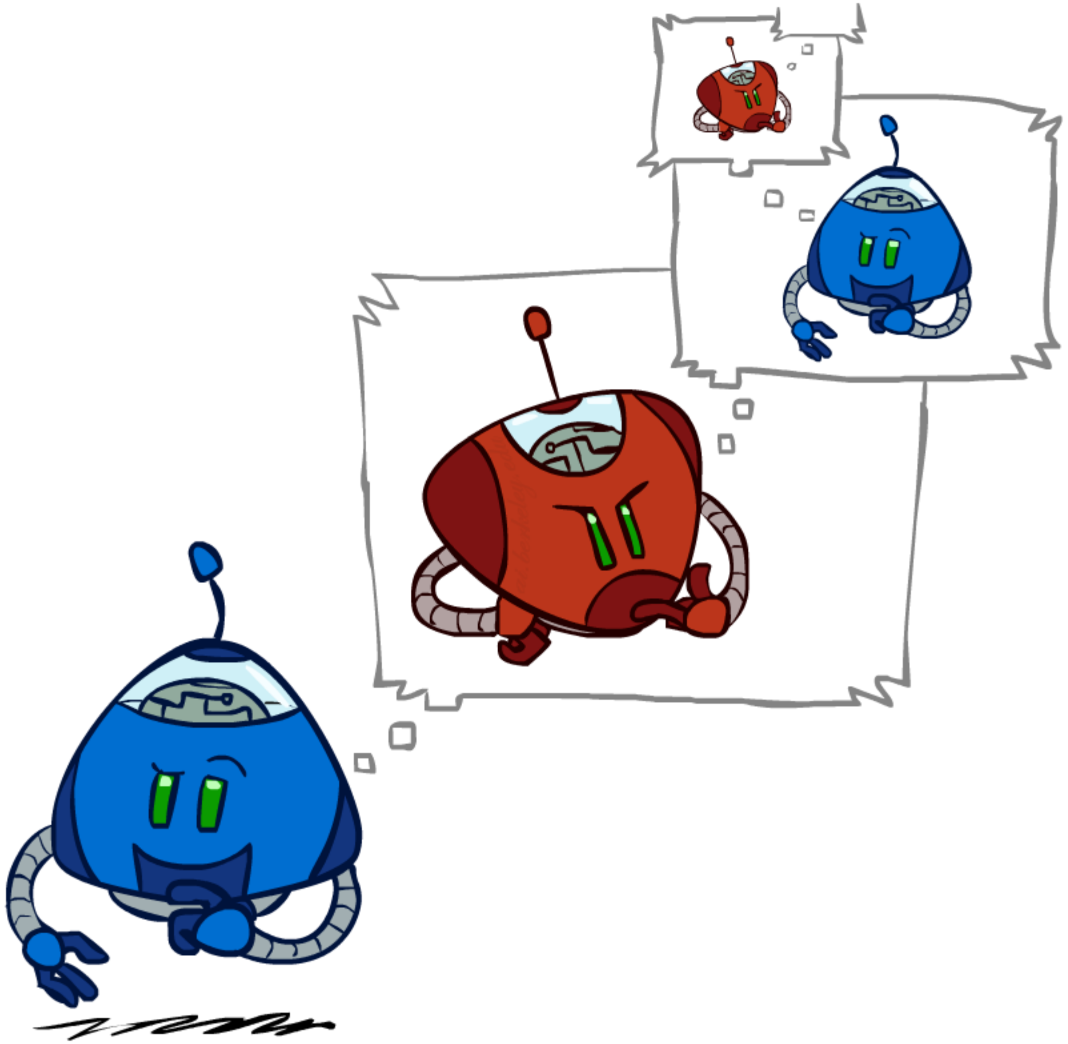


- Solution for a player is a policy: S $\rightarrow$ A

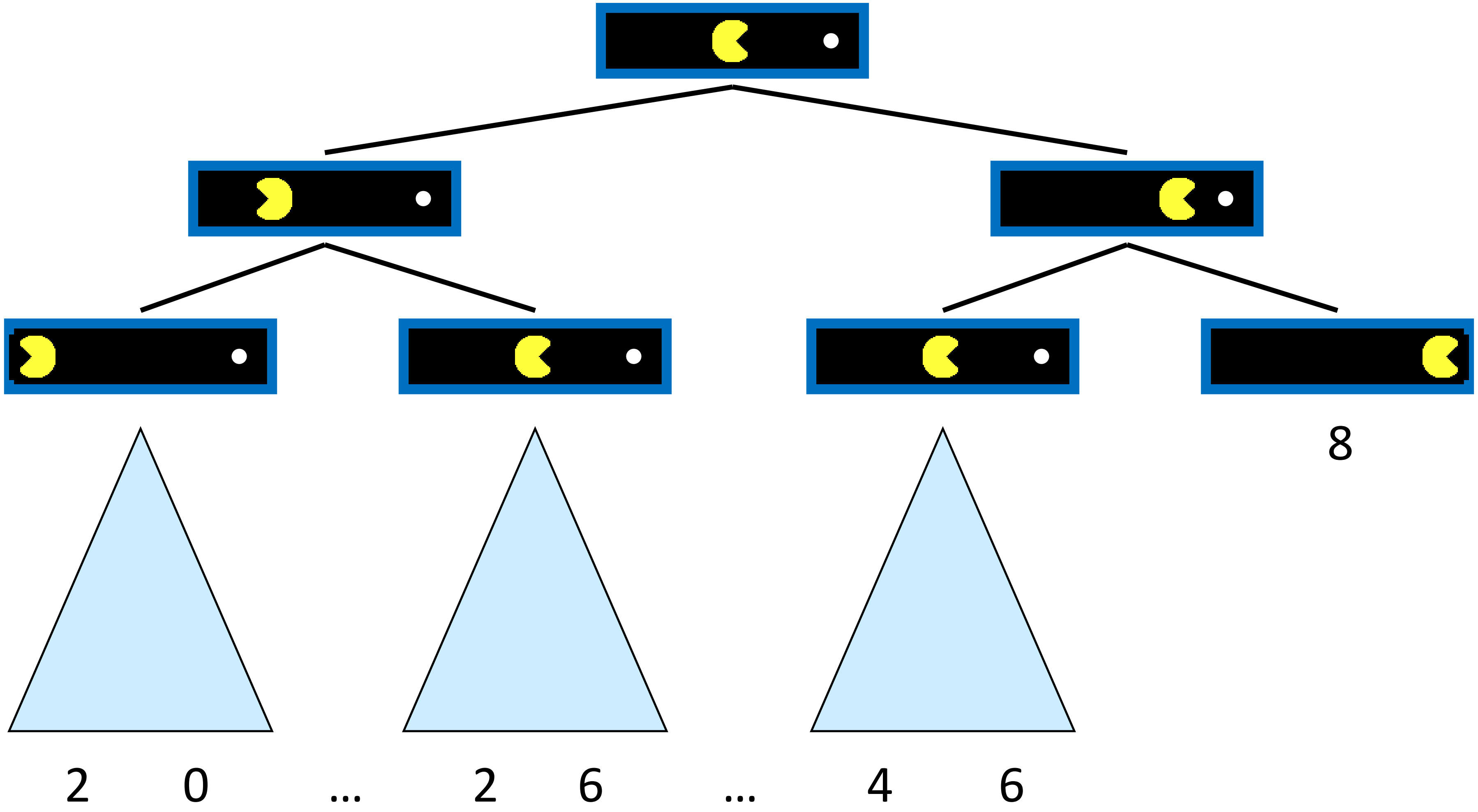


Adversarial Search

Cost -> Utility!



Single-Agent Trees

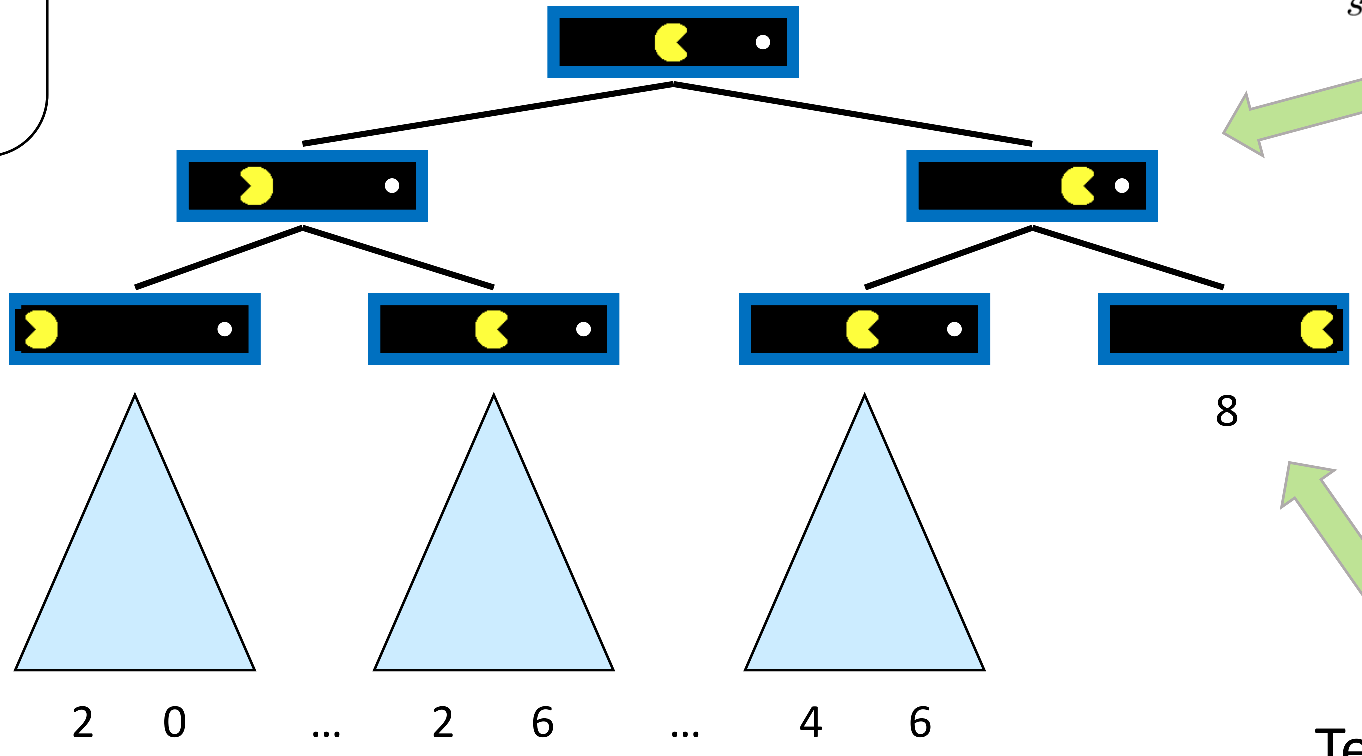


Single-Agent Trees: Value of a State

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

Non-Terminal States:

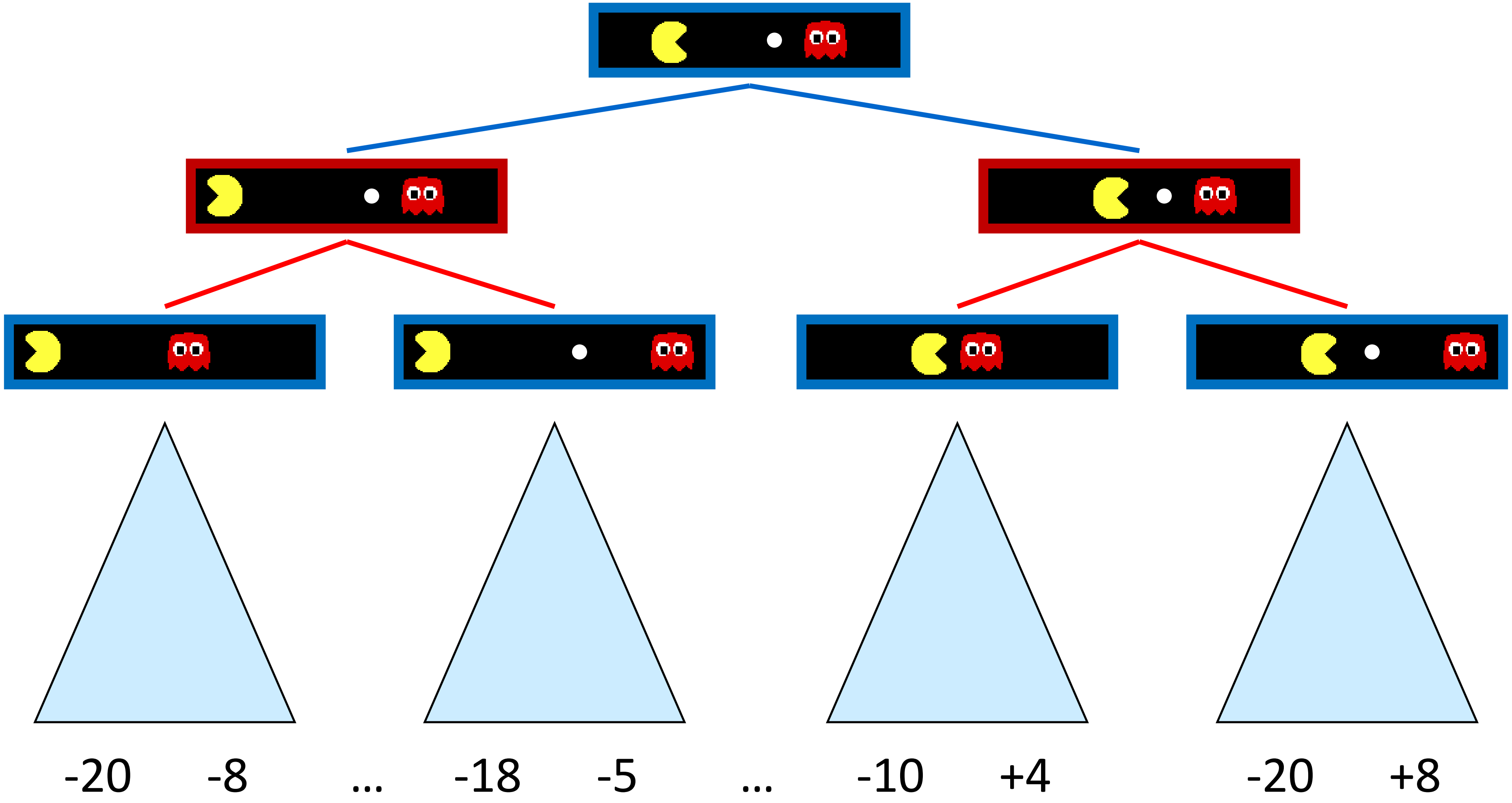
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$



Terminal States:

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

Adversarial Game Trees



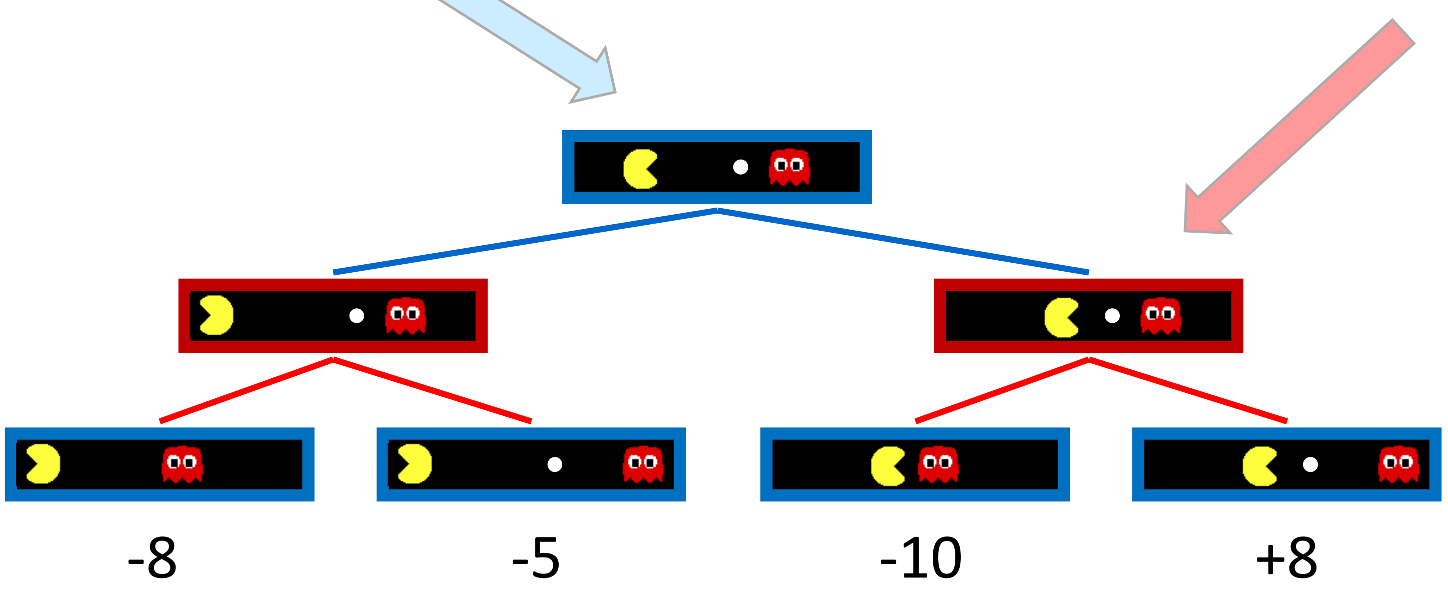
Adversarial Game Trees: Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

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

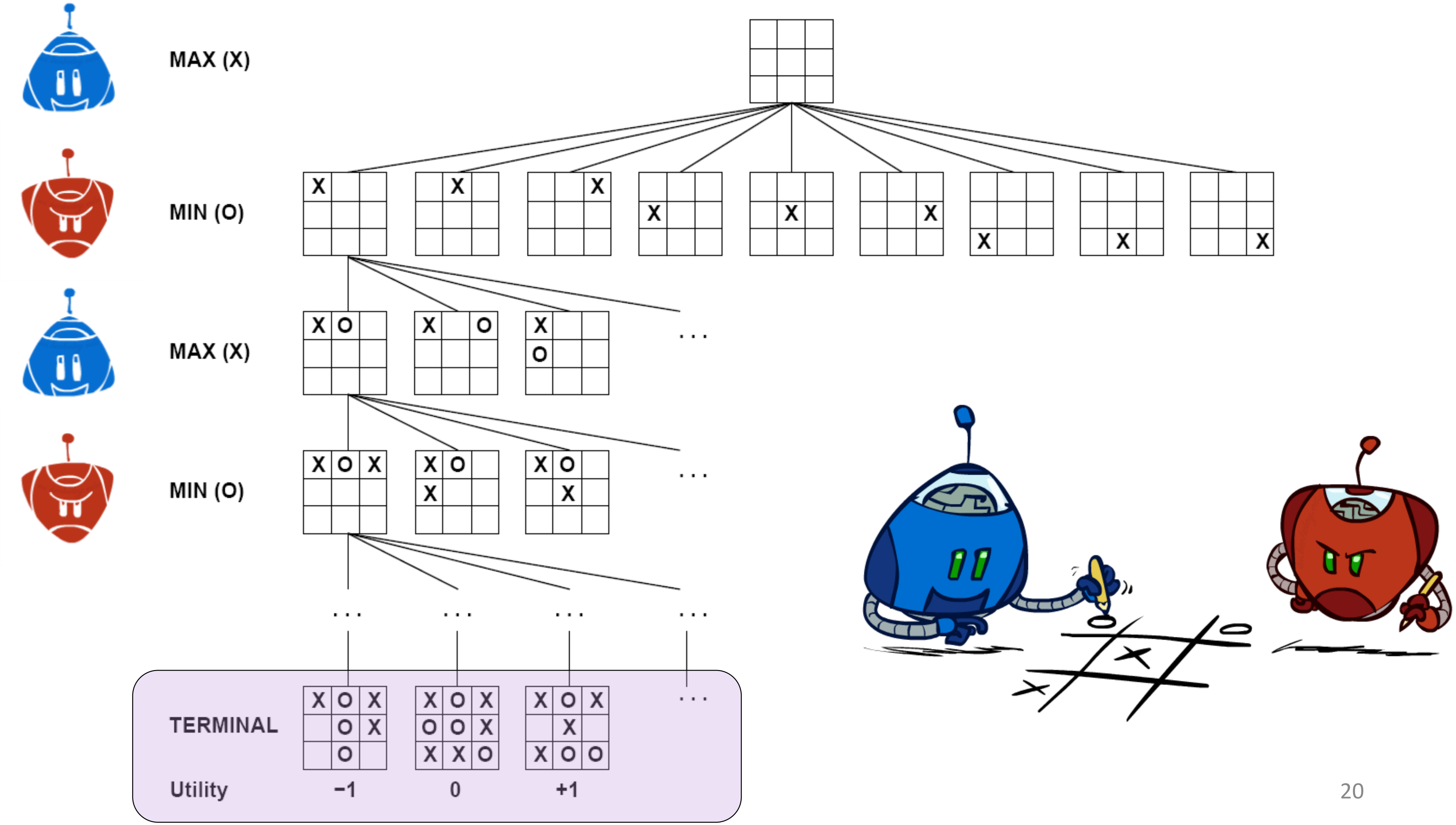


Terminal States:

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

Example: Tic-Tac-Toe Game Tree

- States
- Actions
- Values

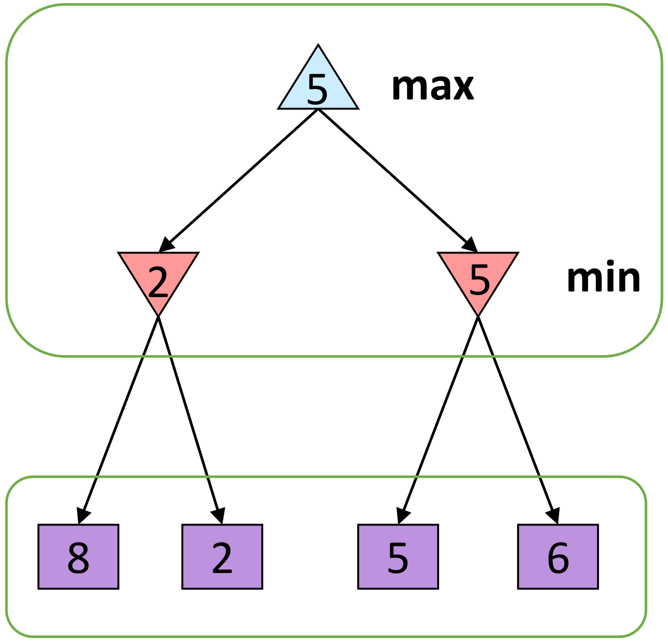


Minimax Search

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

- 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 values: computed recursively



Terminal values: part of the game

Minimax Implementation

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

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

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

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

Minimax Implementation (Dispatch)

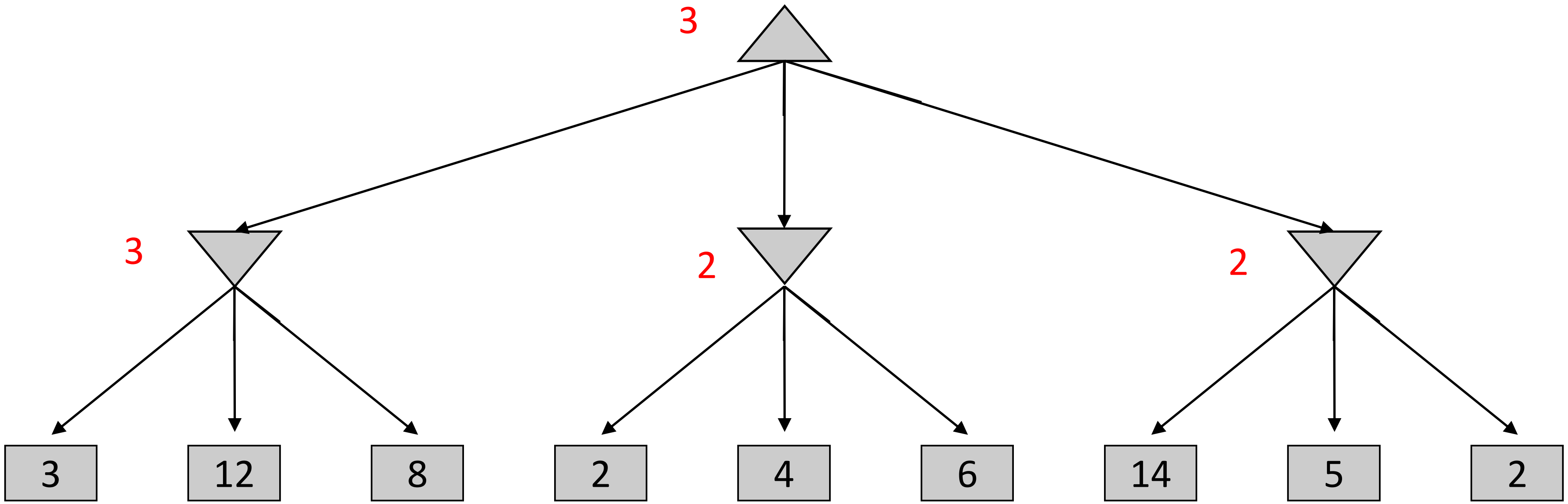
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is MIN: return min-value(state)
```

```
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
```

Example

- Actions?

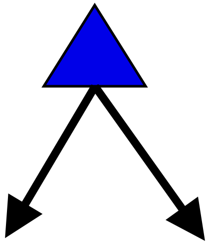


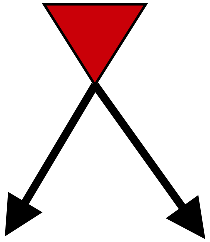
Pseudocode for Single Agent

```
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
```

Pseudocode for Minimax Search

```
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):
```





$$V(s) = \max_a V(s'),$$
where $s' = result(s, a)$

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

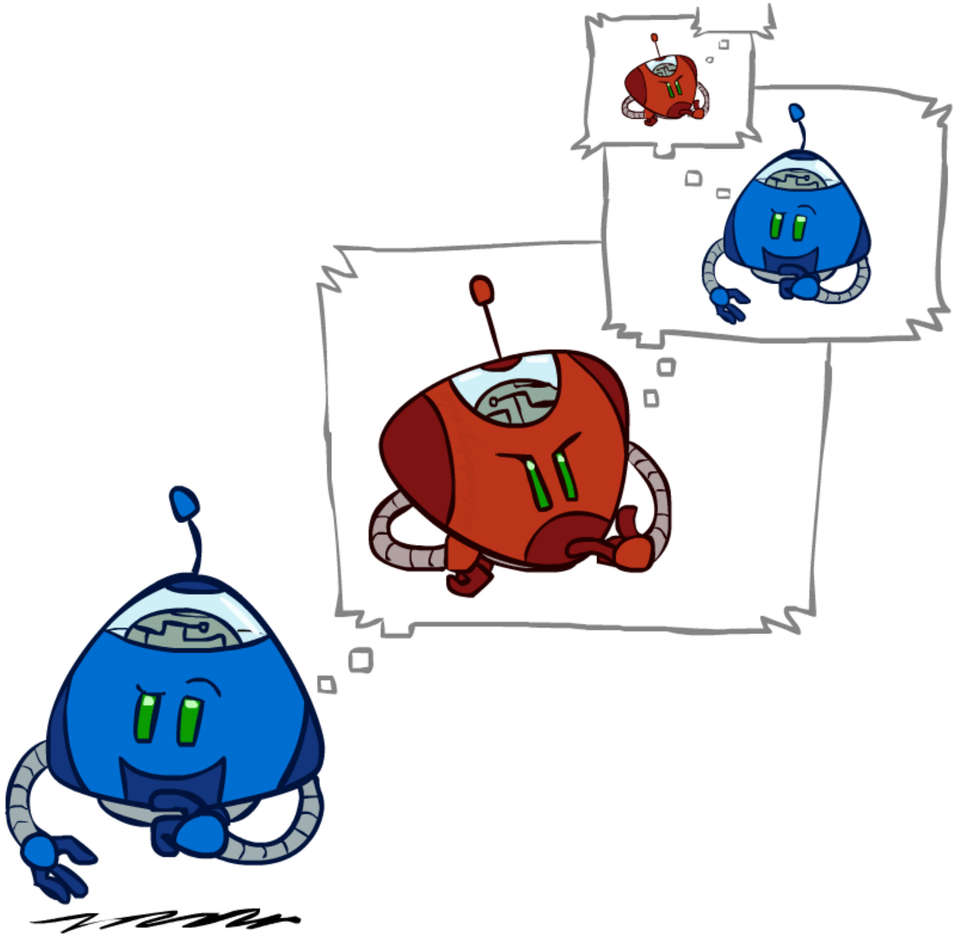
Pseudocode for Generic Game Tree

```
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

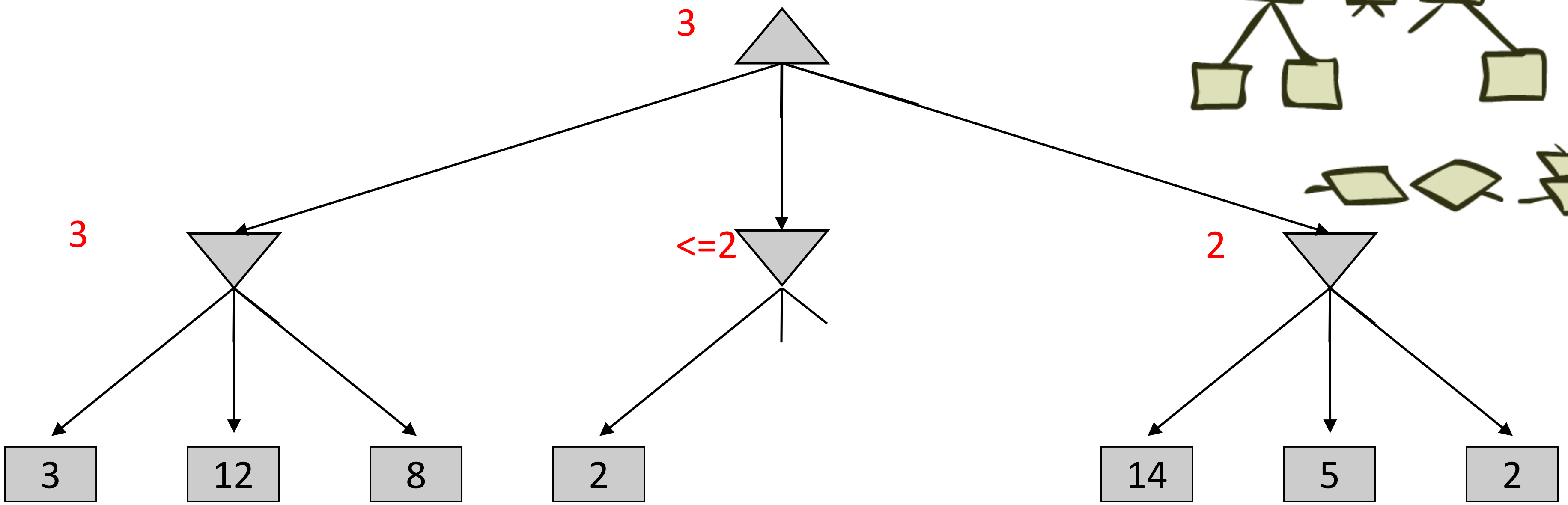
- How efficient is minimax?
  - Just like (exhaustive) DFS
  - Time: $O(b^m)$
  - Space: $O(bm)$

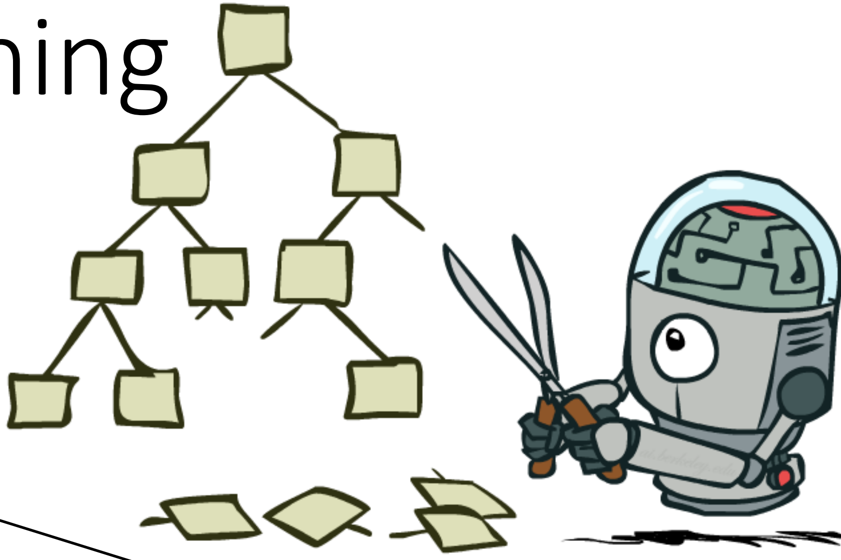
- Example: For chess, $b \approx 35$, $m \approx 100$
  - Exact solution is completely infeasible
  - But, do we need to explore the whole tree?
  - Humans can't do this either, so how do we play chess?
  - Bounded rationality – Herbert Simon



Resource Limits

Resource Limits: Game Tree Pruning

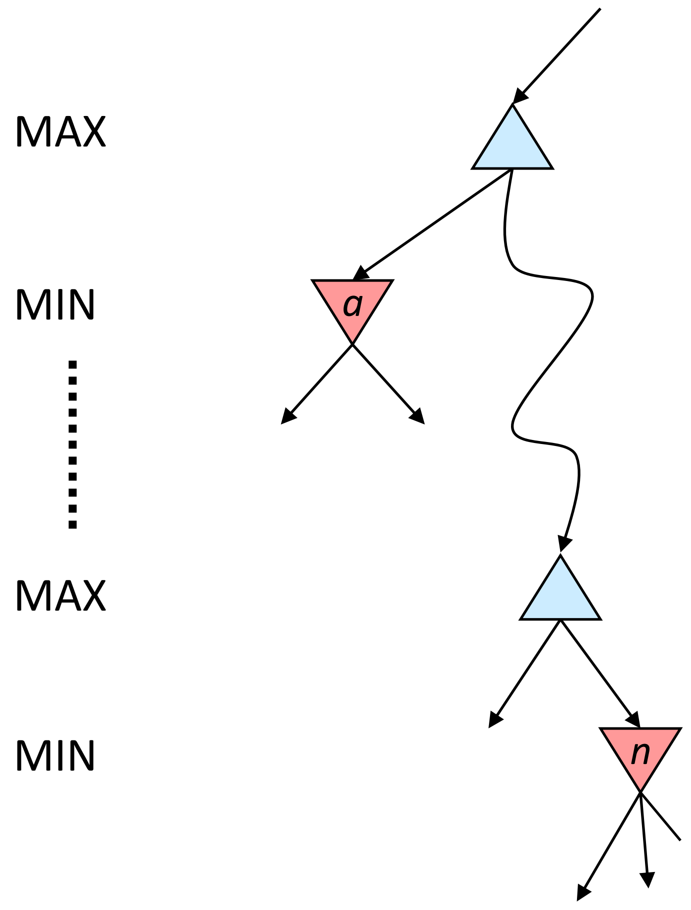




The order of generation matters: more pruning is possible if good moves come first

Game Tree Pruning: Alpha-Beta Pruning

- General configuration (MIN version)
  - We're computing the MIN-VALUE at some node *n*
  - We're looping over *n*'s children
  - *n*'s estimate of the childrens' min is dropping
  - Who cares about *n*'s value? MAX
  - Let *a* be the best value that MAX can get at any choice point along the current path from the root
  - If *n* becomes worse than *a*, MAX will avoid it, so we can stop considering *n*'s other children (it's already bad enough that it won't be played)



- MAX version is symmetric

Alpha-Beta Implementation

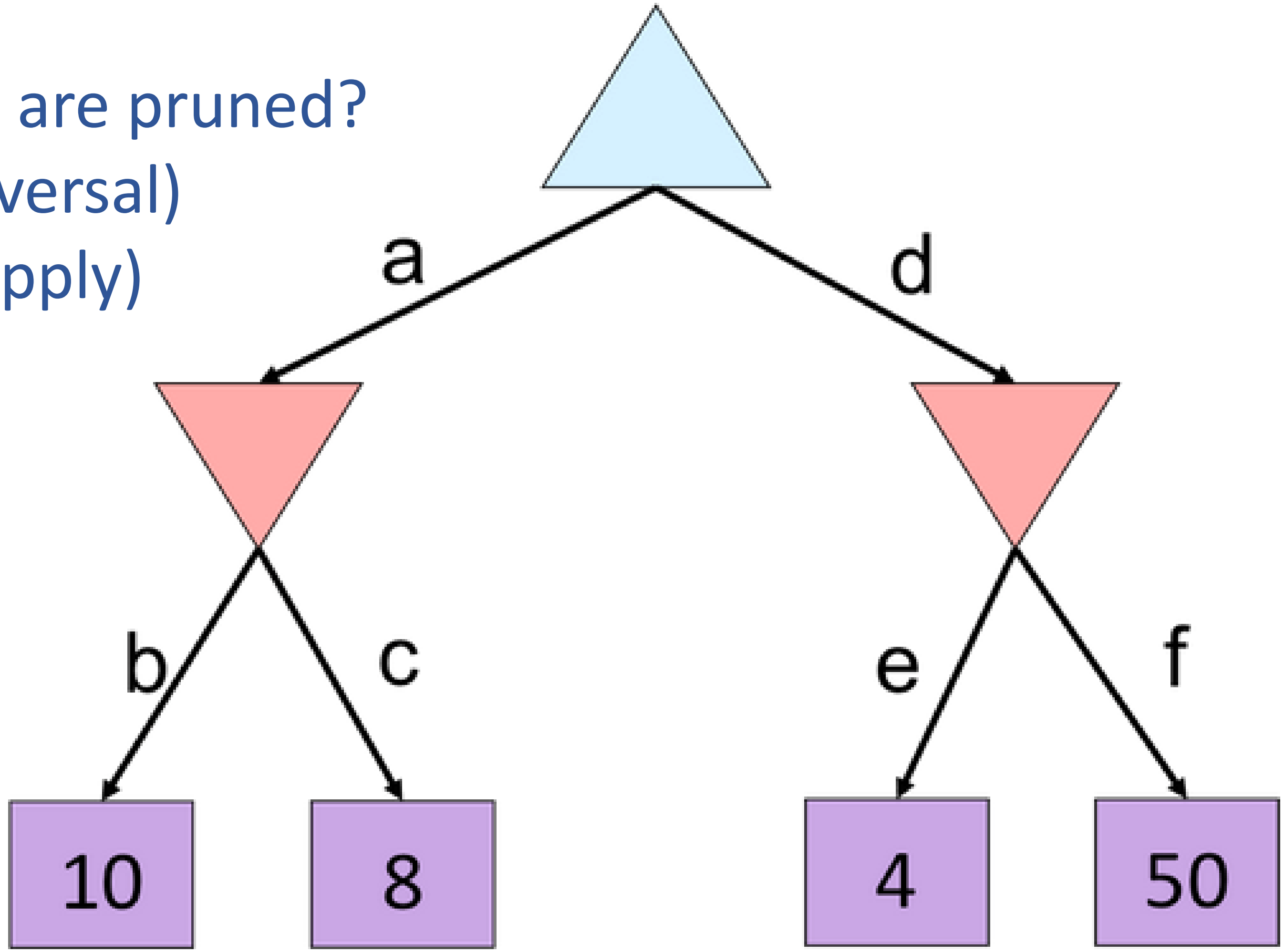
α: MAX's best option on path to root
β: MIN's best option on path to root

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

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

Quiz

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



Quiz 2

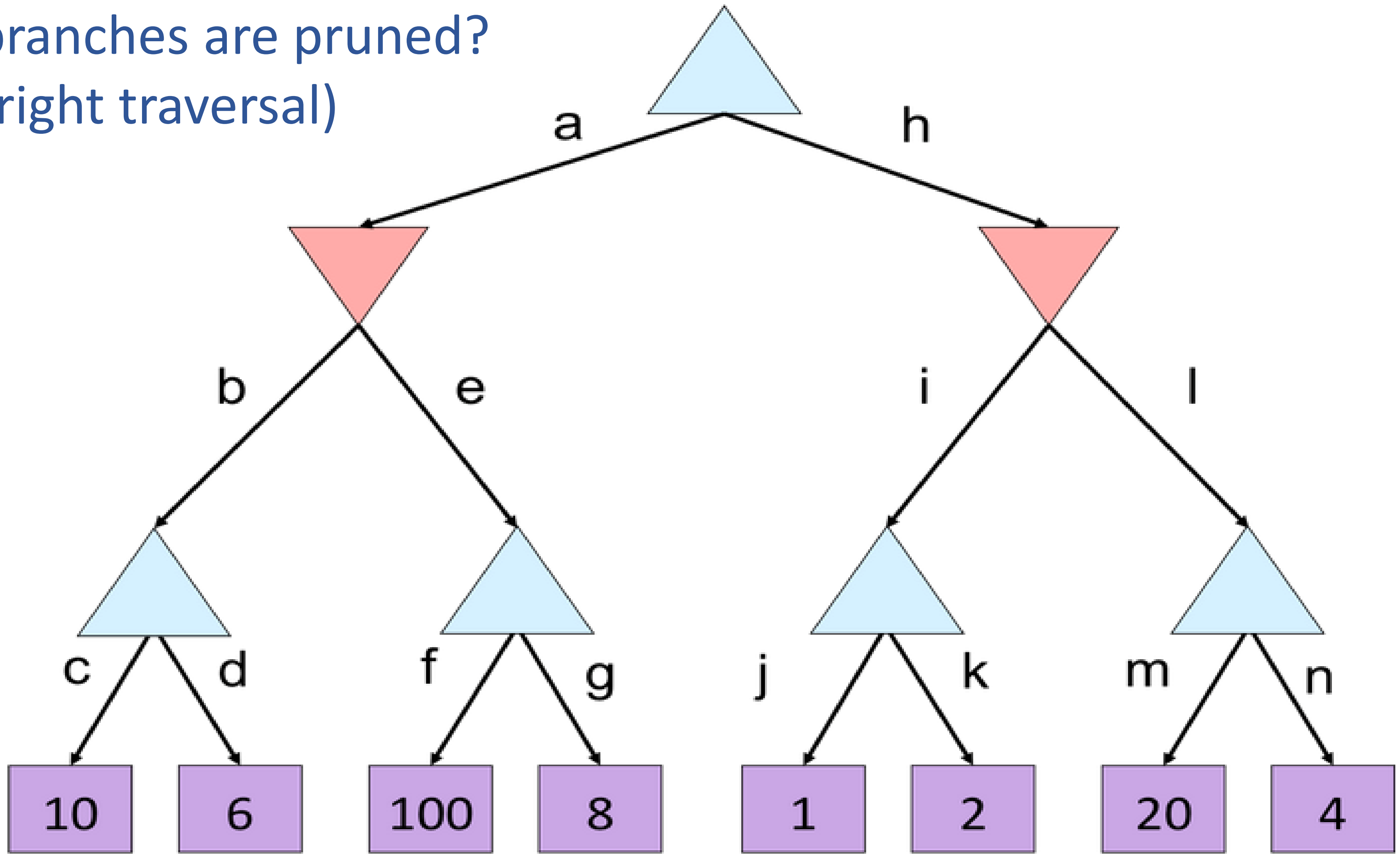
Which branches are pruned?
(Left to right traversal)

A) e, l

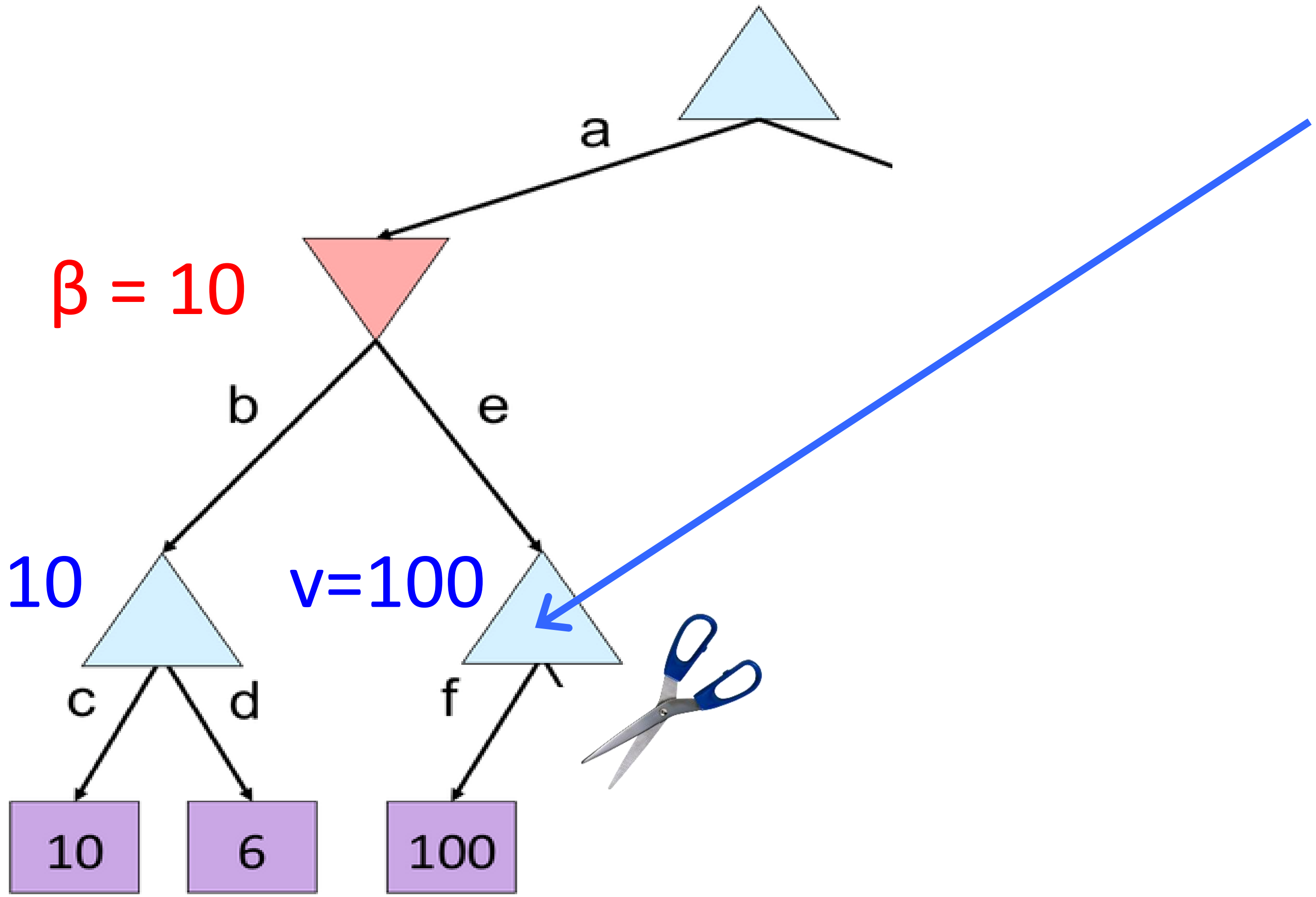
B) g, l

C) g, k, l

D) g, n



Quiz 2 - 2

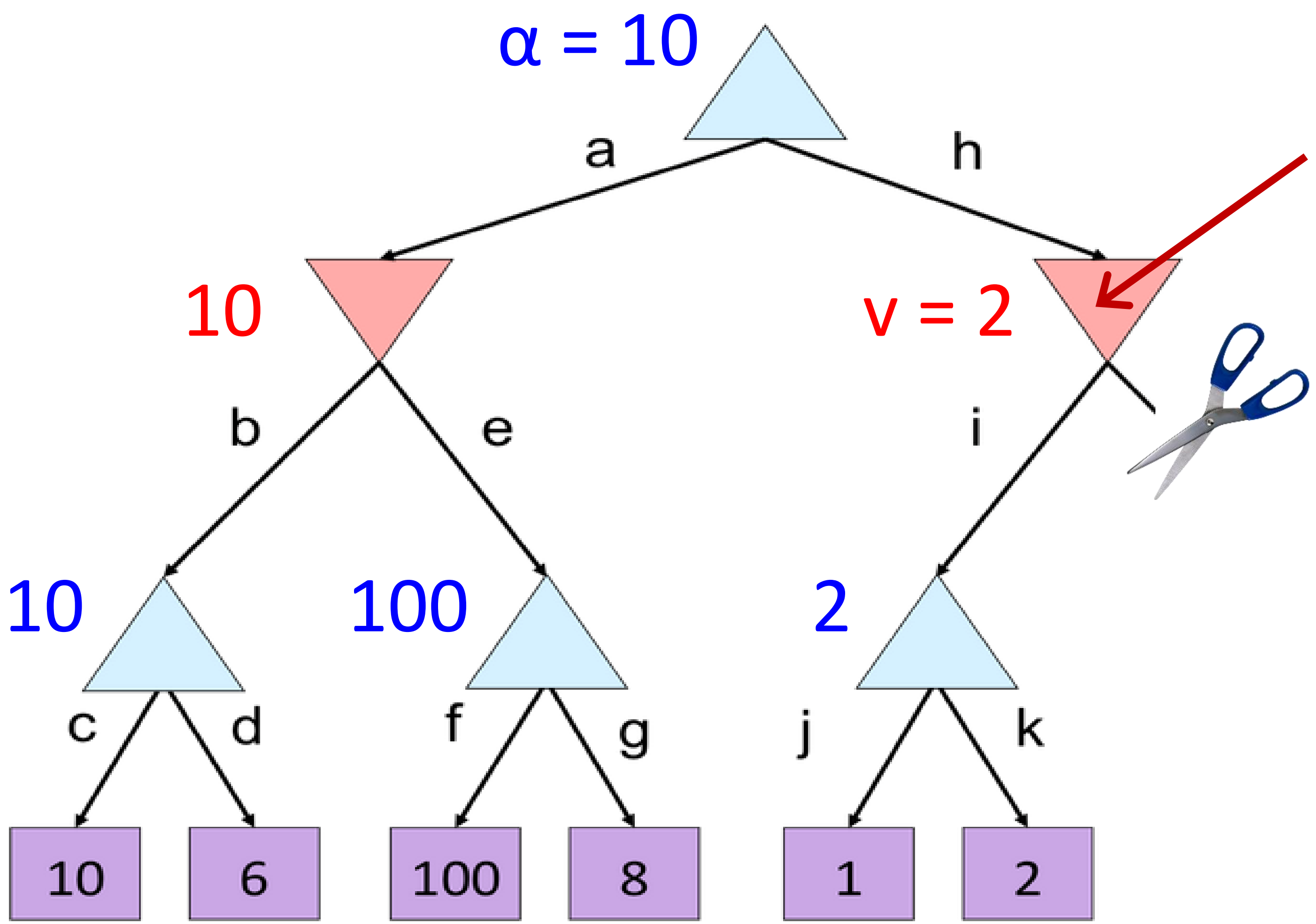


α: MAX's best option on path to root

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

Quiz 2 - 3

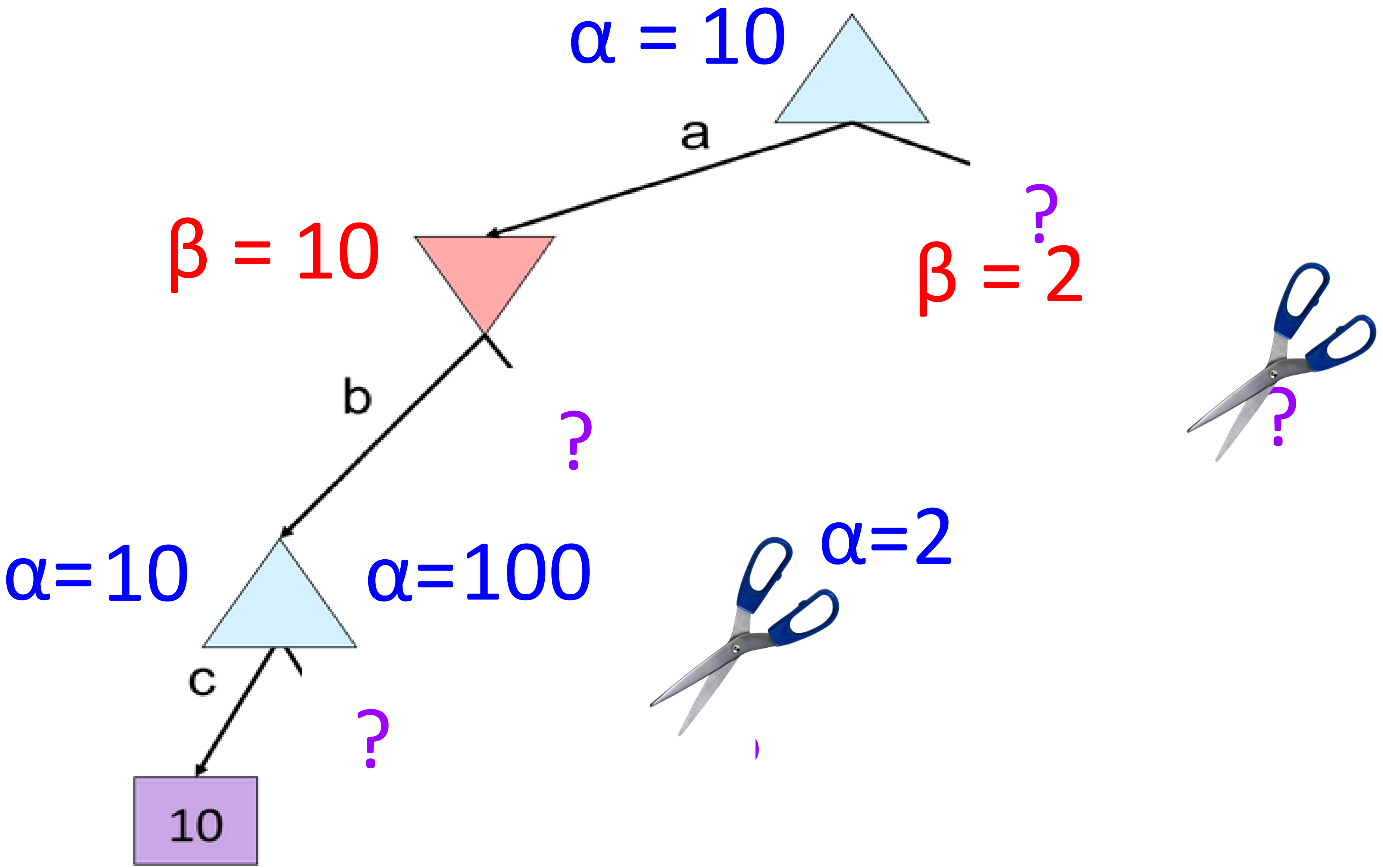


α: MAX's best option on path to root

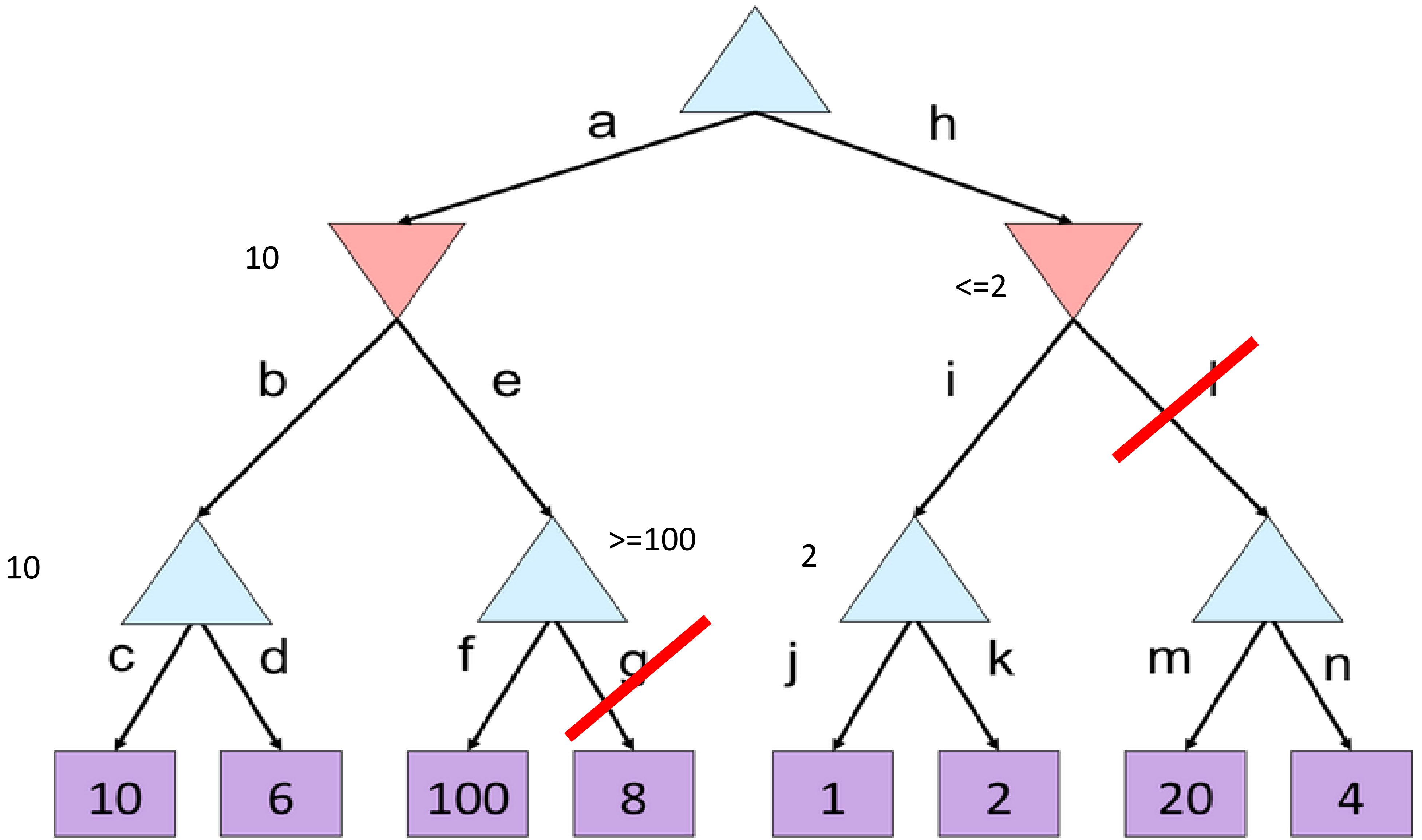
β: MIN's best option on path to root

```
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
```

Quiz 2 - 4



Quiz 2 - 5



Alpha-Beta Pruning Properties

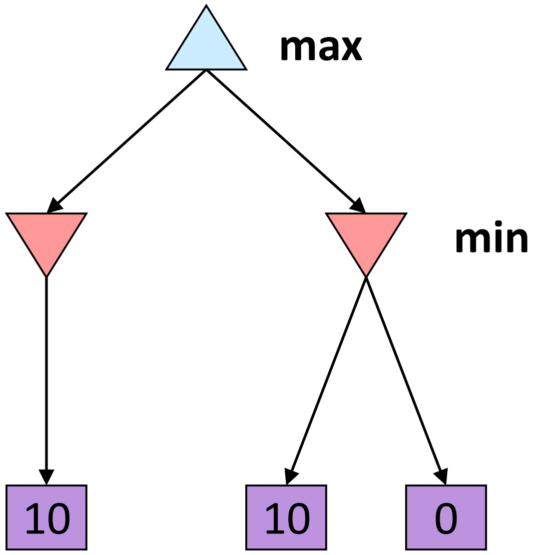
- This pruning has no effect on minimax value computed for the root!

- Values of intermediate nodes might be wrong
  - Important: children of the root may have the wrong value
  - So the most naïve version won't let you do action selection

- Good child ordering improves effectiveness of pruning

- With "perfect ordering":
  - Time complexity drops to $O(b^{m/2})$
  - Doubles solvable depth!
  - Chess: 1M nodes/move => depth=8, respectable
  - Full search of complicated games, is still hopeless…

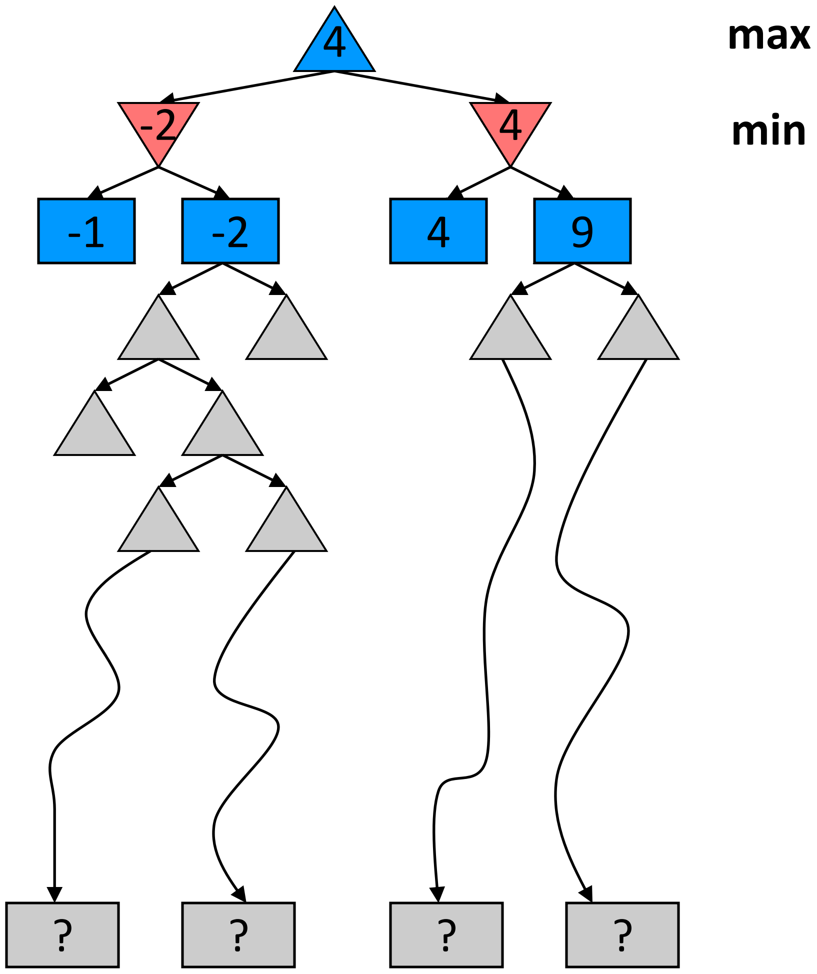
- This is a simple example of metareasoning (computing about what to compute)



Resource Limits II
Bounded lookahead

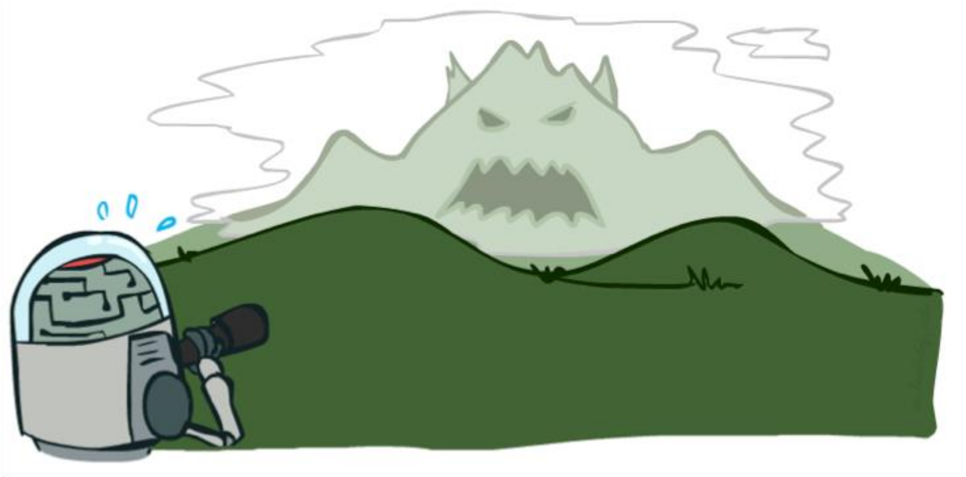
Depth-limited search

- Problem: In realistic games, cannot search to leaves!
- Solution: Depth-limited search
  - Instead, search only to a limited depth in the tree
  - Replace terminal utilities with an evaluation function for non-terminal positions
- Example:
  - Suppose we have 100 seconds, can explore 10K nodes / sec
  - So can check 1M nodes per move
  - For chess, $b \approx 35$ so reaches about depth 4 – not so good
  - α-β reaches about depth 8 – decent chess program
- Guarantee of optimal play is gone
- More plies makes a BIG difference
- Use iterative deepening for an anytime algorithm



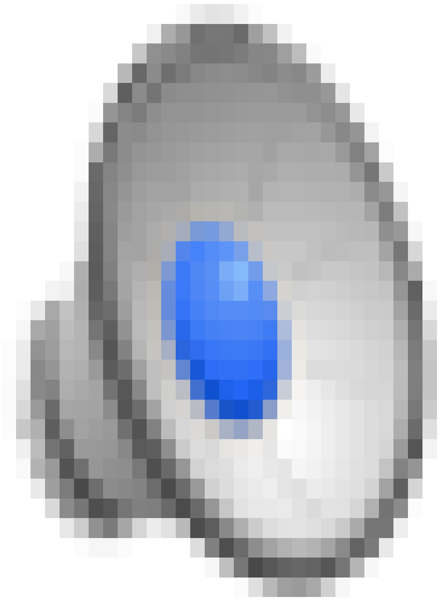
Depth Matters

- 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

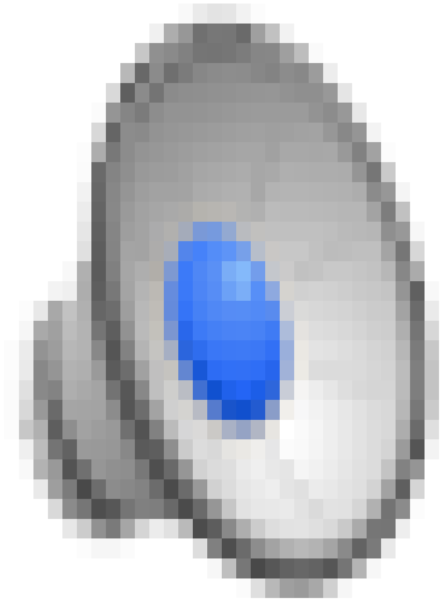


[Demo: depth limited (L6D4, L6D5)]

Video of Demo Limited Depth (2)

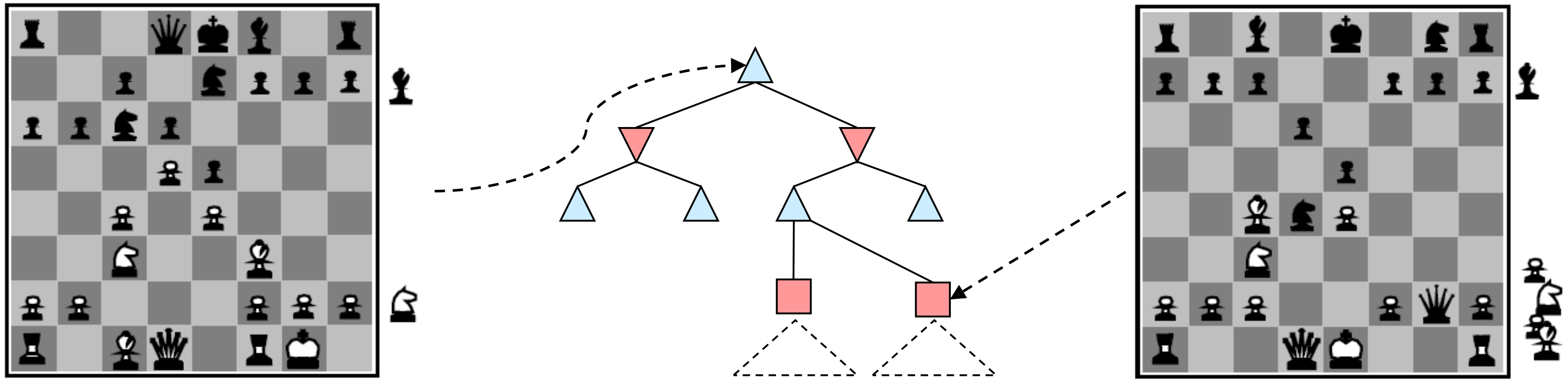


Video of Demo Limited Depth (10)



Evaluation Functions

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



Black to move

White slightly better

White to move

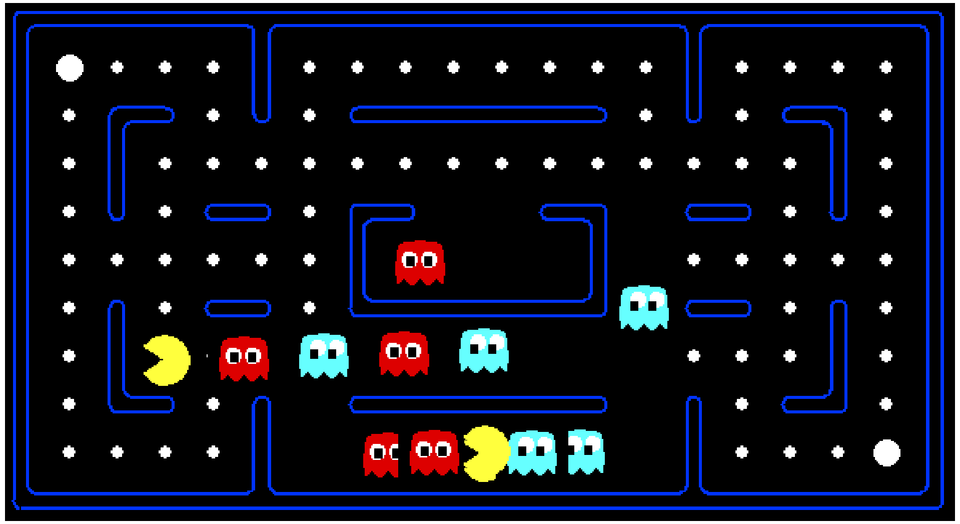
Black winning

- Ideal function: returns the actual minimax value of the position
- In practice: typically weighted linear sum of features:

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

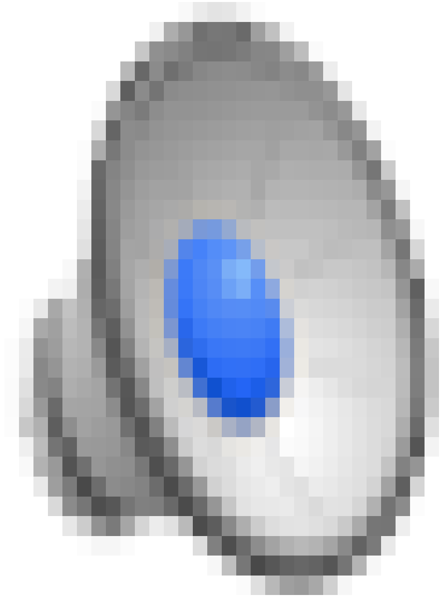
  - e.g. $w_1 = 9$, $f_1(s)$ = (num white queens – num black queens), etc.

Evaluation for Pacman

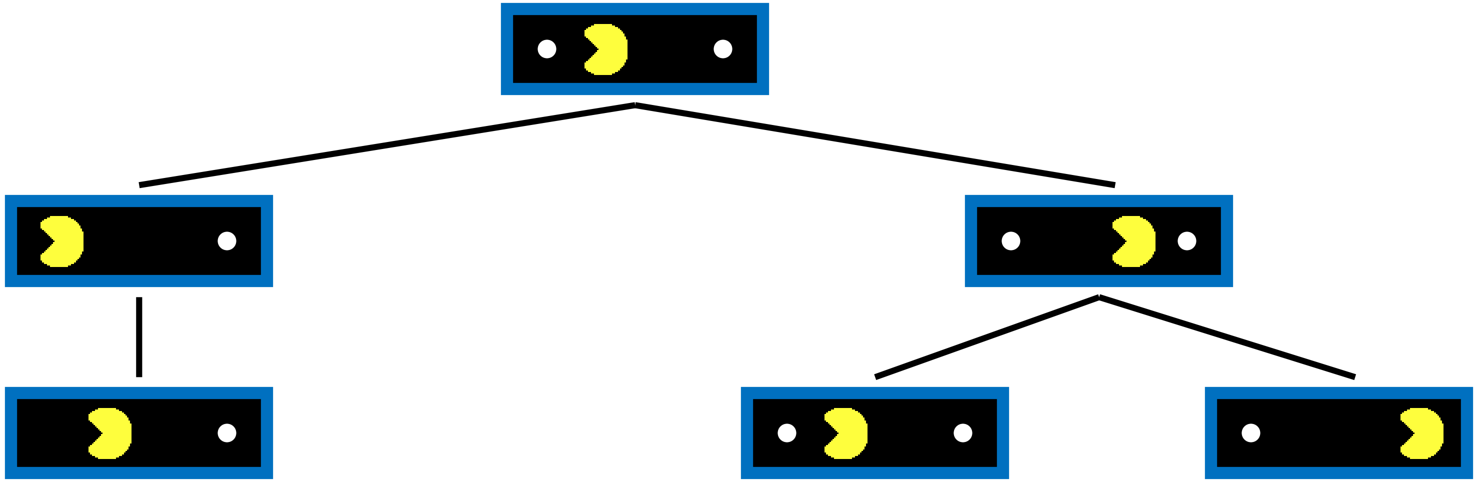


[Demo: thrashing d=2, thrashing d=2 (fixed evaluation function), smart ghosts coordinate (L6D6,7,8,10)]

Video of Demo Thrashing (d=2)

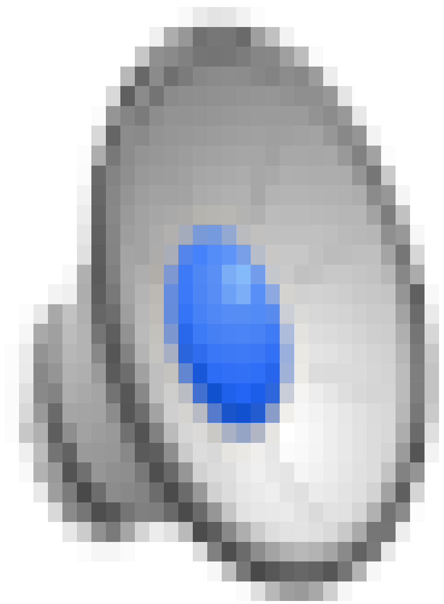


Why Pacman Starves

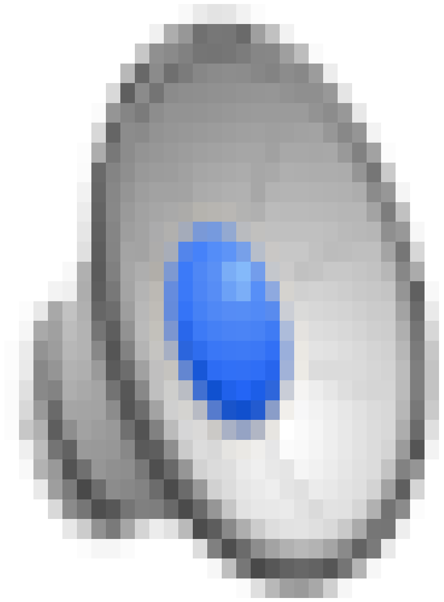


- A danger of replanning agents!
  - He knows his score will go up by eating the dot now (west, east)
  - He knows his score will go up just as much by eating the dot later (east, west)
  - There are no point-scoring opportunities after eating the dot (within the horizon, two here)
  - Therefore, waiting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

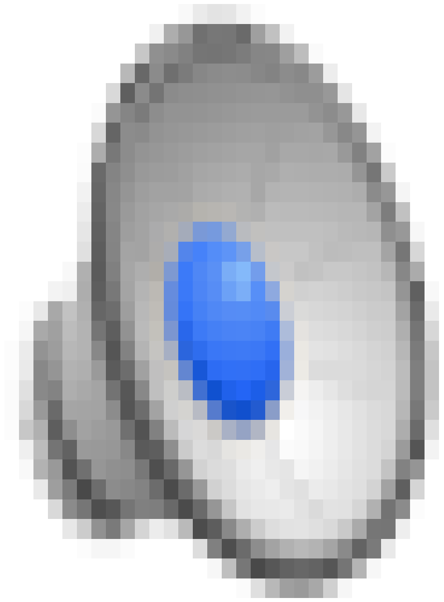
Video of Demo Thrashing -- Fixed (d=2)



Video of Demo Smart Ghosts (Coordination)



Video of Demo Smart Ghosts (Coordination) – Zoomed In

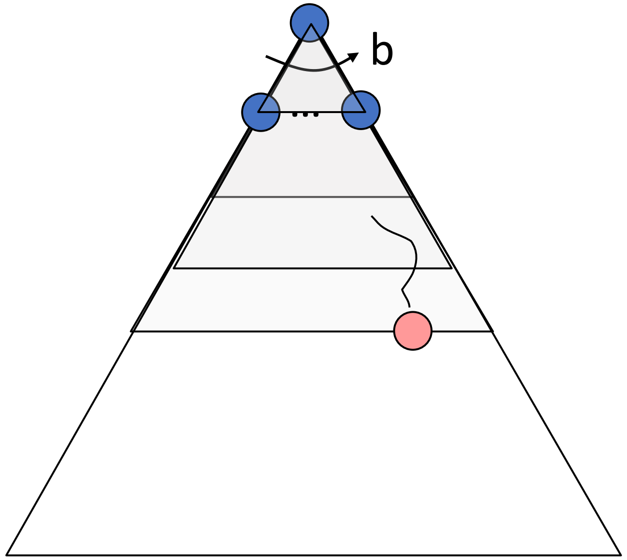


Iterative Deepening

Iterative deepening uses DFS as a subroutine:

1. Do a DFS which only searches for paths of length 1 or less. (DFS gives up on any path of length 2)
2. If “1” failed, do a DFS which only searches paths of length 2 or less.
3. If “2” failed, do a DFS which only searches paths of length 3 or less.

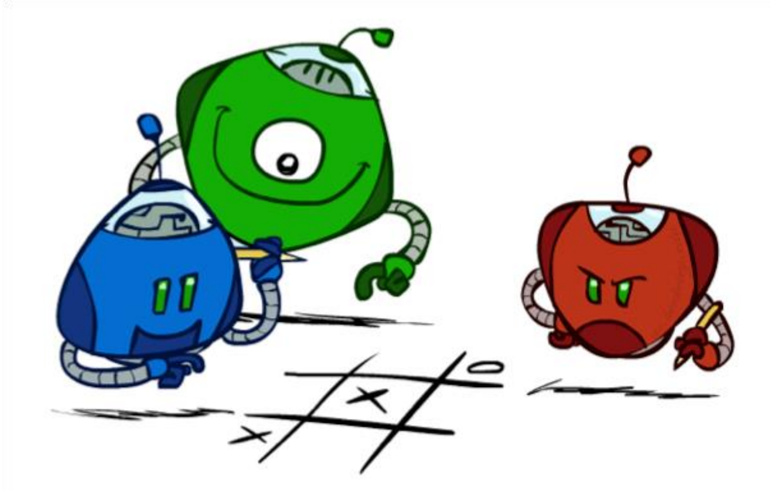
....and so on.



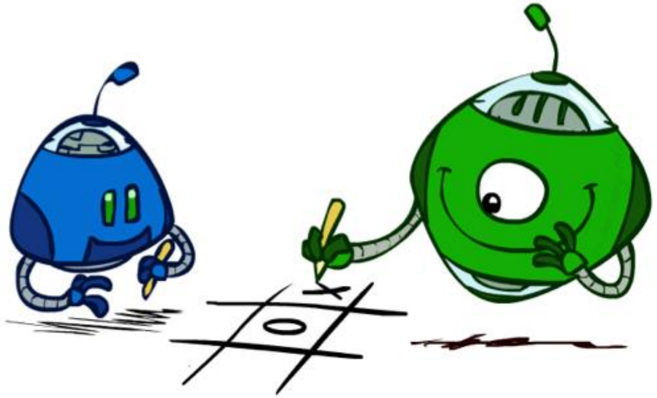
Why do we want to do this for multiplayer games?

Note: wrongness of eval functions matters less and less the deeper the search goes!

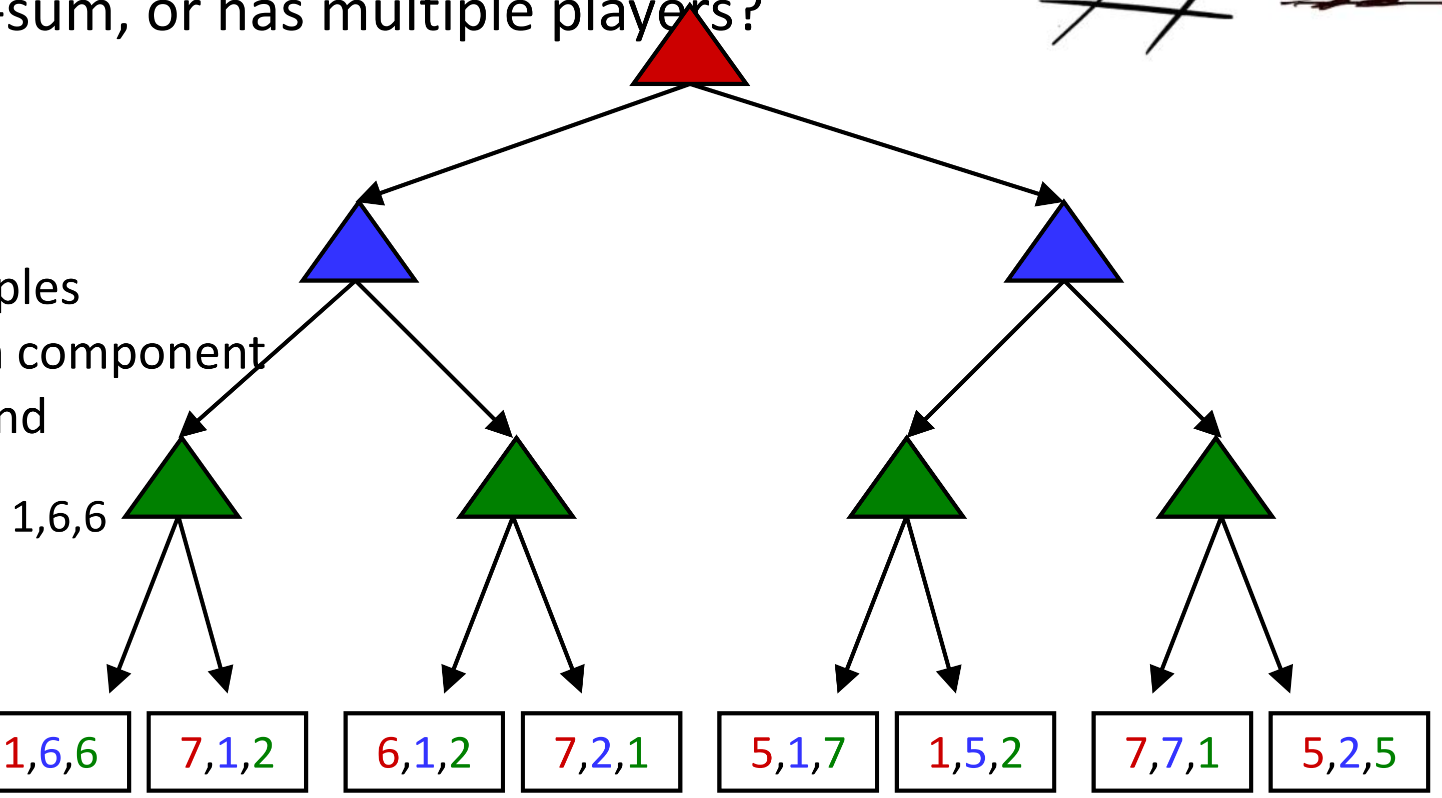
Generalized minimax

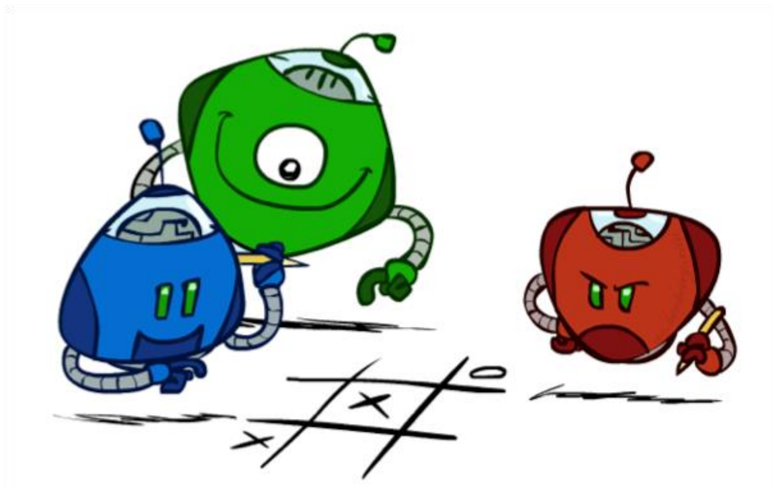


Multi-Agent Utilities



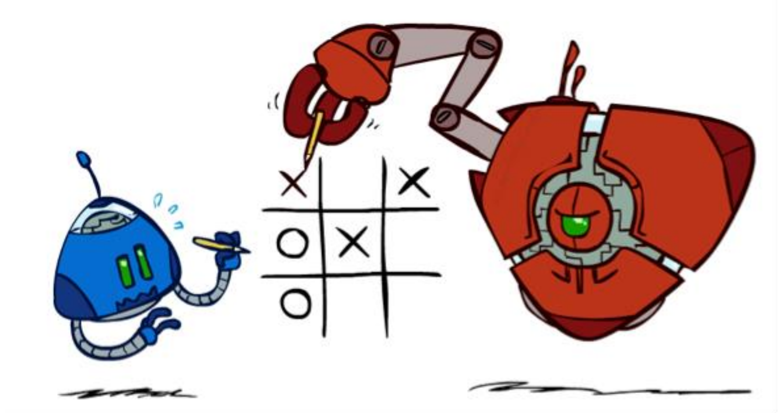
- What if the game is not zero-sum, or has multiple players?
- Generalization of minimax:
  - Terminals have utility tuples
  - Node values are also utility tuples
  - Each player maximizes its own component
  - Can give rise to cooperation and competition dynamically…

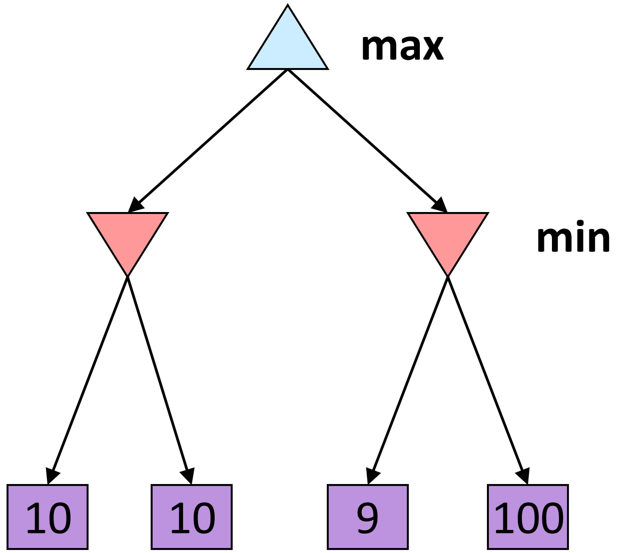


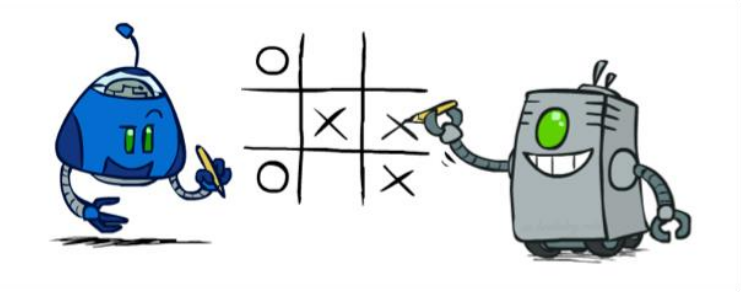


Modeling Assumptions

What if your opponent isn't playing optimally?



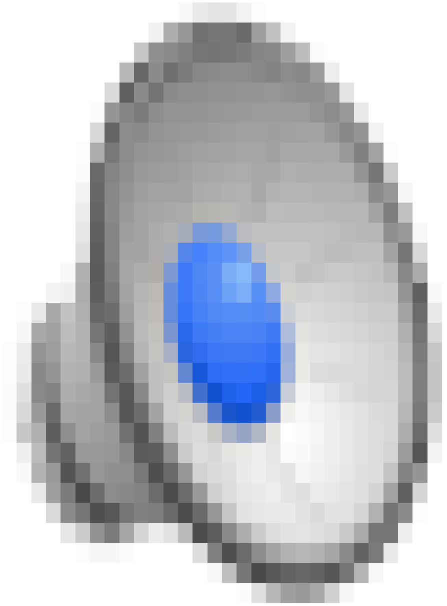




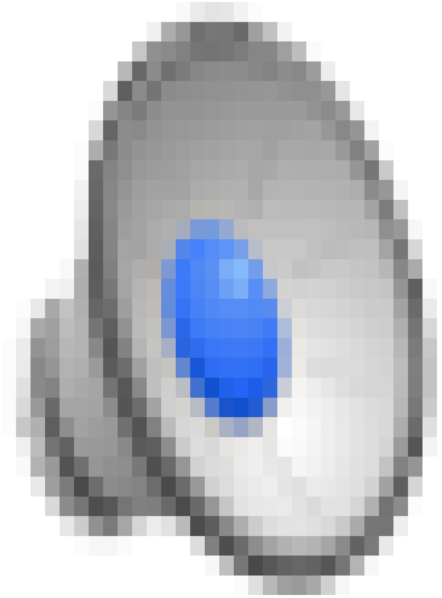
Optimal against a perfect player. Otherwise?

[Demo: min vs exp (L6D2, L6D3)]

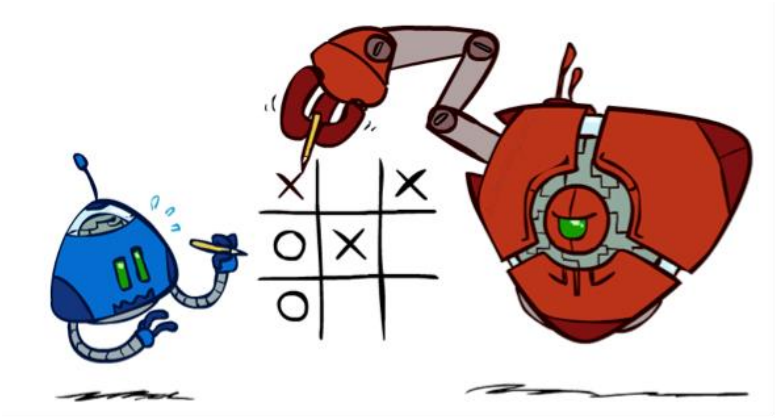
Video of Demo Min vs. Exp (Min)

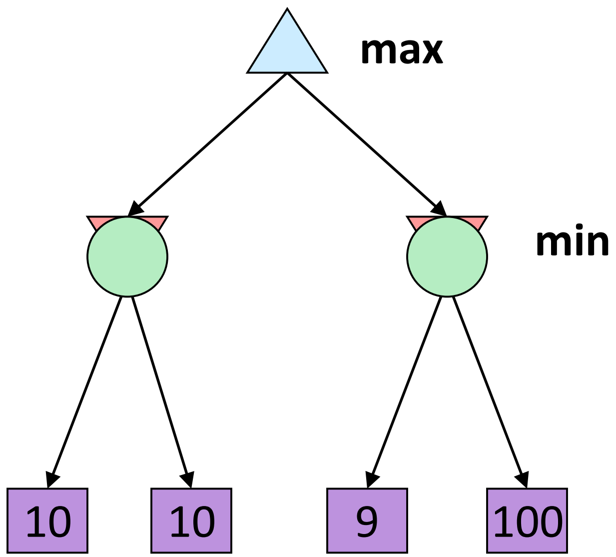


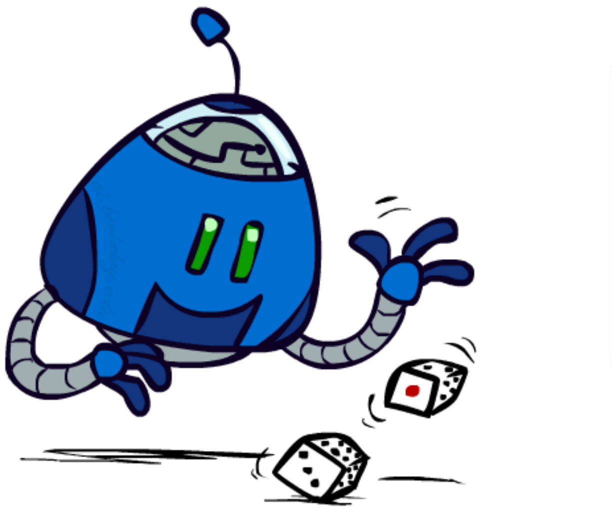
Video of Demo Min vs. Exp (Exp)



Worst-Case vs. Average Case



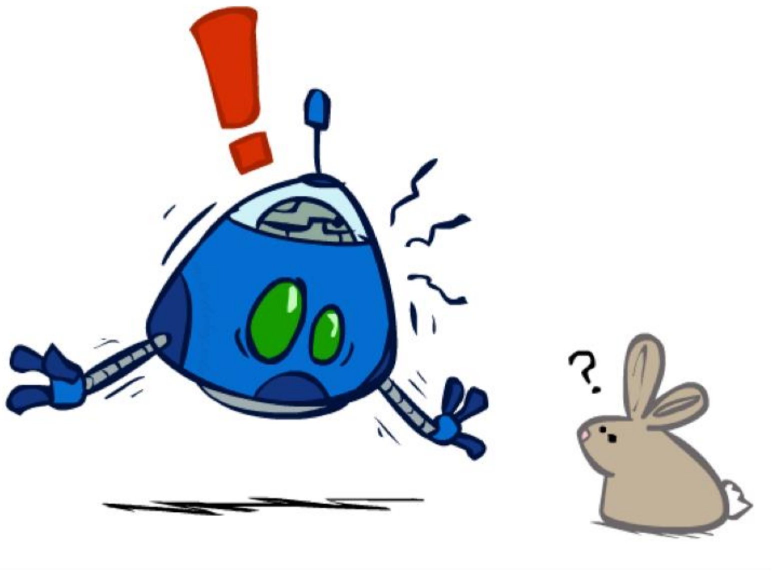




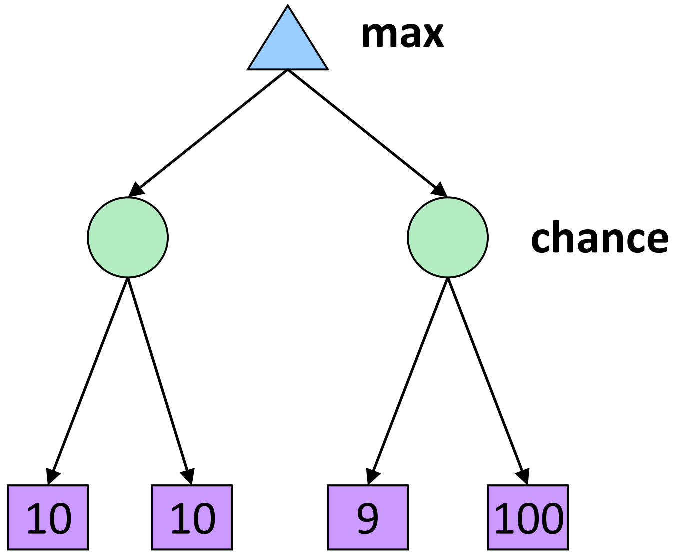
Idea: Uncertain outcomes controlled by chance, not an adversary!

Why not minimax?

- Worst case reasoning is too conservative
- Need average case reasoning



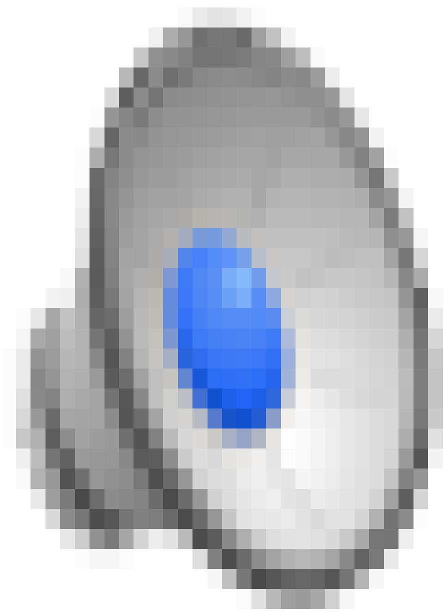
Expectimax Search



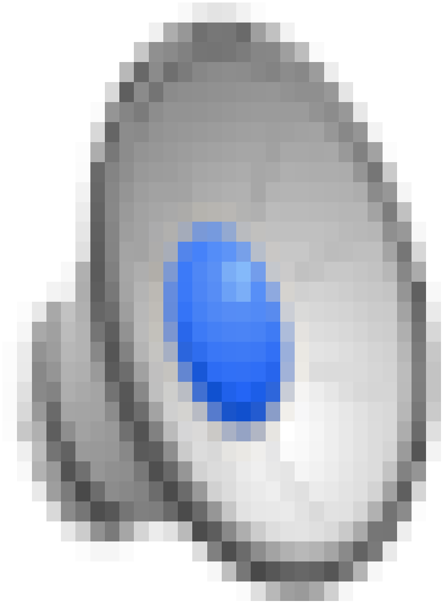
- Why wouldn't we know what the result of an action will be?
  - Explicit randomness: rolling dice
  - Unpredictable opponents: the ghosts respond randomly
  - Unpredictable humans: humans are not perfect
  - Actions can fail: when moving a robot, wheels might slip
- Values should now reflect average-case (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
- Later, we'll learn how to formalize the underlying uncertain-result problems as Markov Decision Processes

[Demo: min vs exp (L7D1,2)]

Video of Demo Minimax vs Expectimax (Min)



Video of Demo Minimax vs Expectimax (Exp)



Expectimax Pseudocode

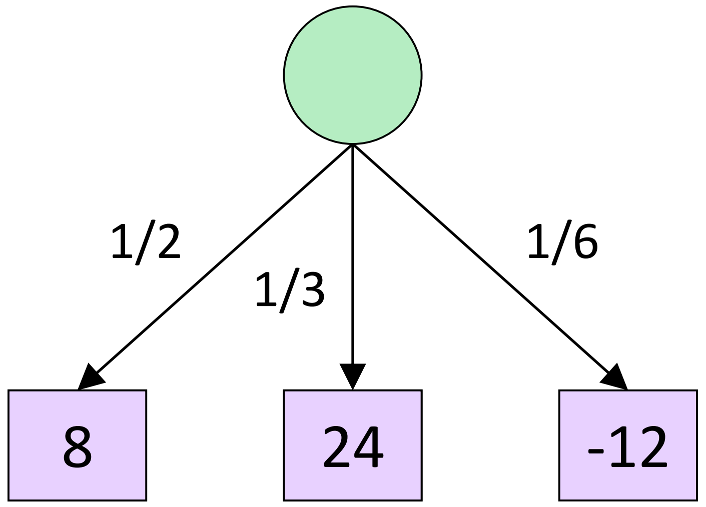
```
def value(state):
    if the state is a terminal state: return the state's utility
    if the next agent is MAX: return max-value(state)
    if the next agent is EXP: return exp-value(state)
```

```
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
```

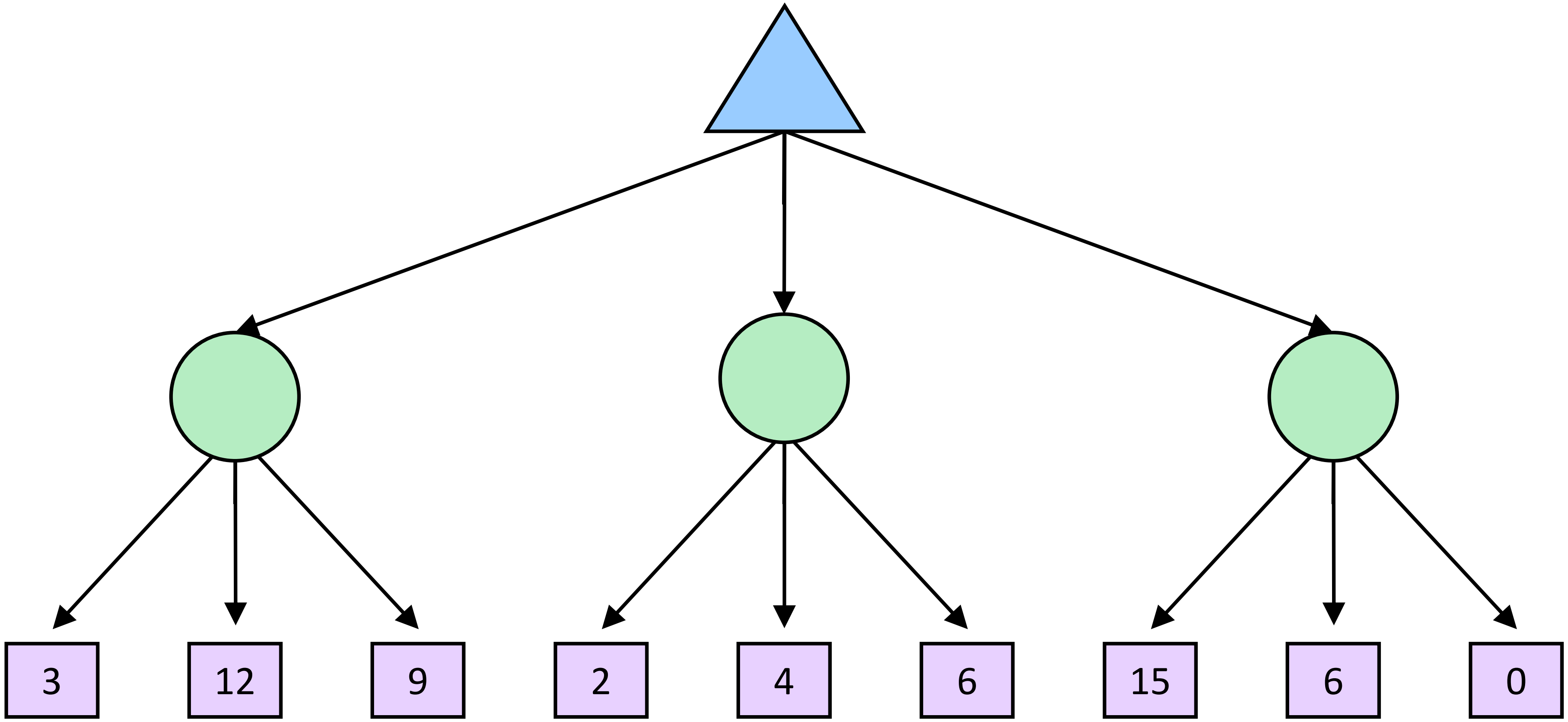
Expectimax Pseudocode 2

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

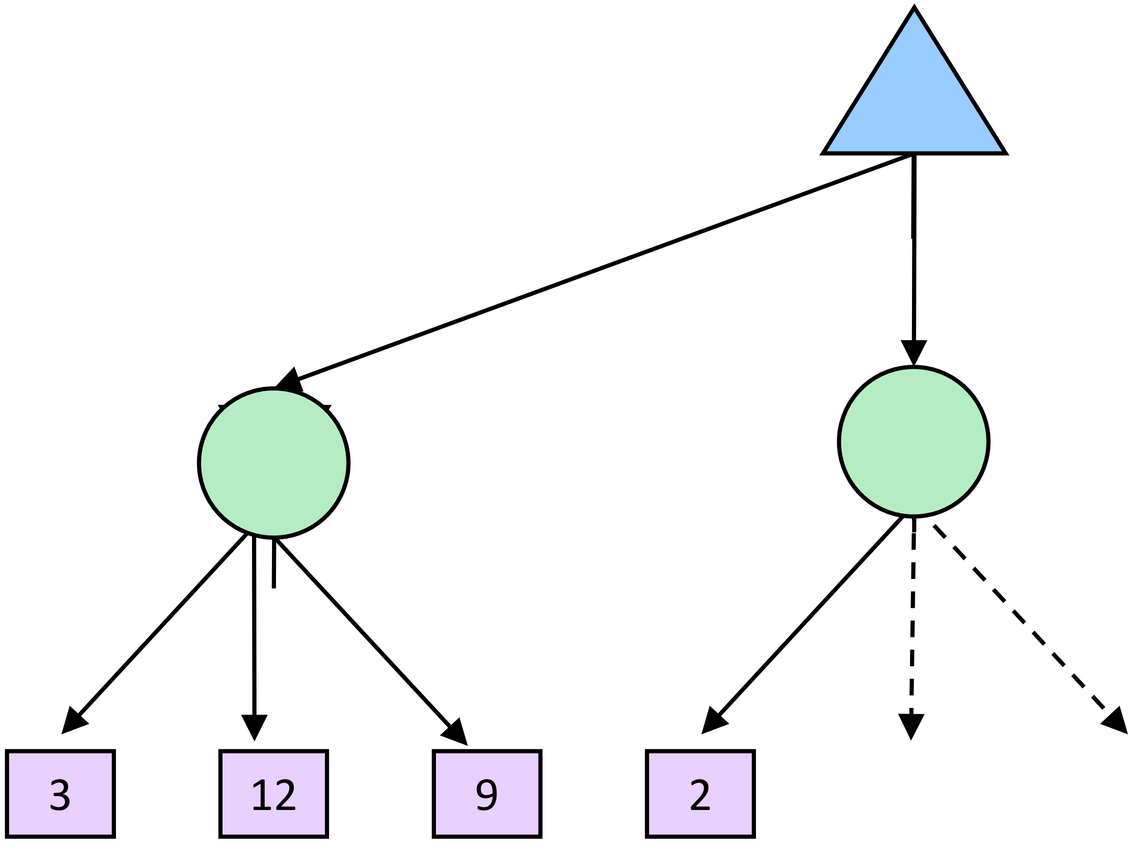


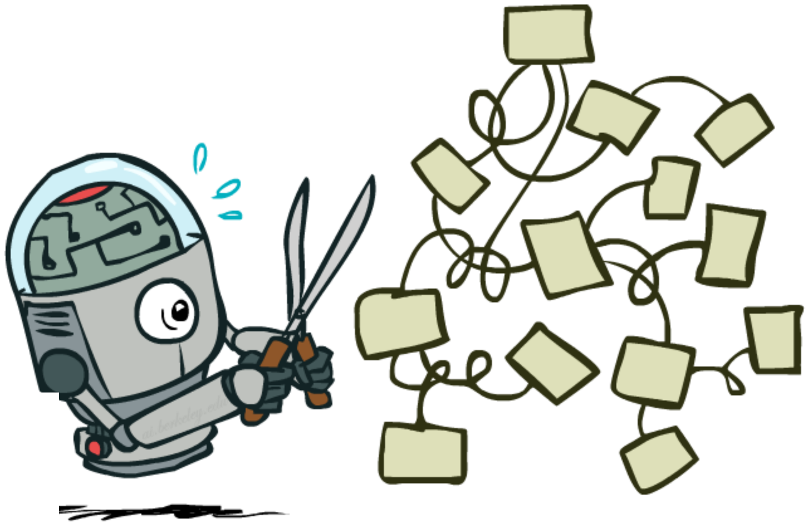
$$v = (1/2)\ (8) + (1/3)\ (24) + (1/6)\ (-12) = 10$$

Example

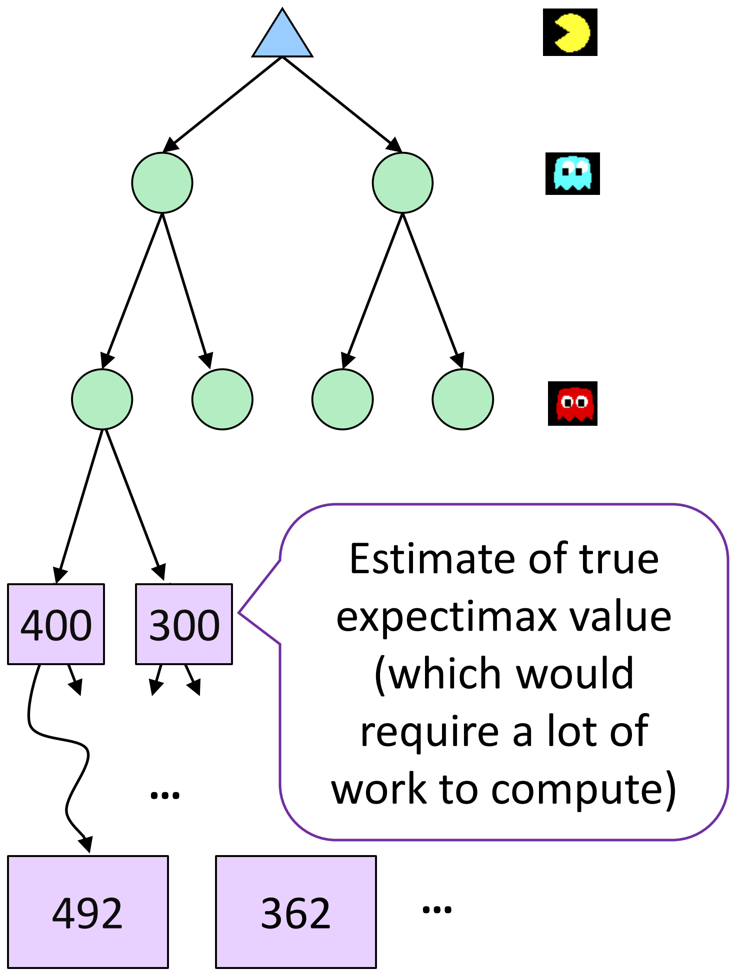


Expectimax Pruning?





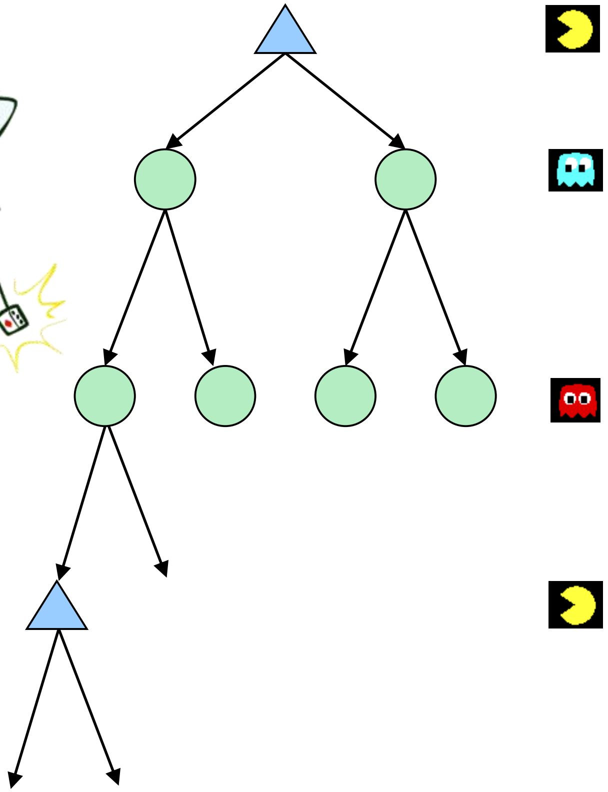
Expectimax: Depth-Limited



What Probabilities to Use?

- In expectimax search, we have a probabilistic m… how the opponent (or environment) will behave in an…
    - 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!

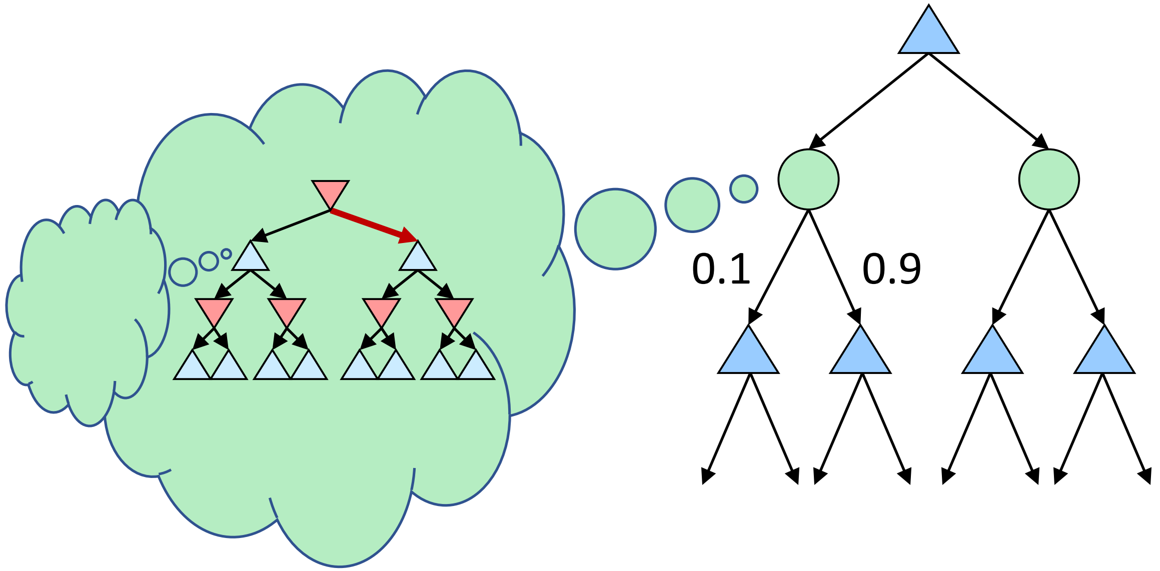
- For now, assume each chance node magically comes along with probabilities that specify the distribution over its outcomes



Having a probabilistic belief about another agent's action does not mean that the agent is flipping any coins!

Quiz: Informed Probabilities

- Let's say you know that your opponent is actually running a depth 2 minimax, using the result 80% of the time, and moving randomly otherwise
- Question: What tree search should you use?



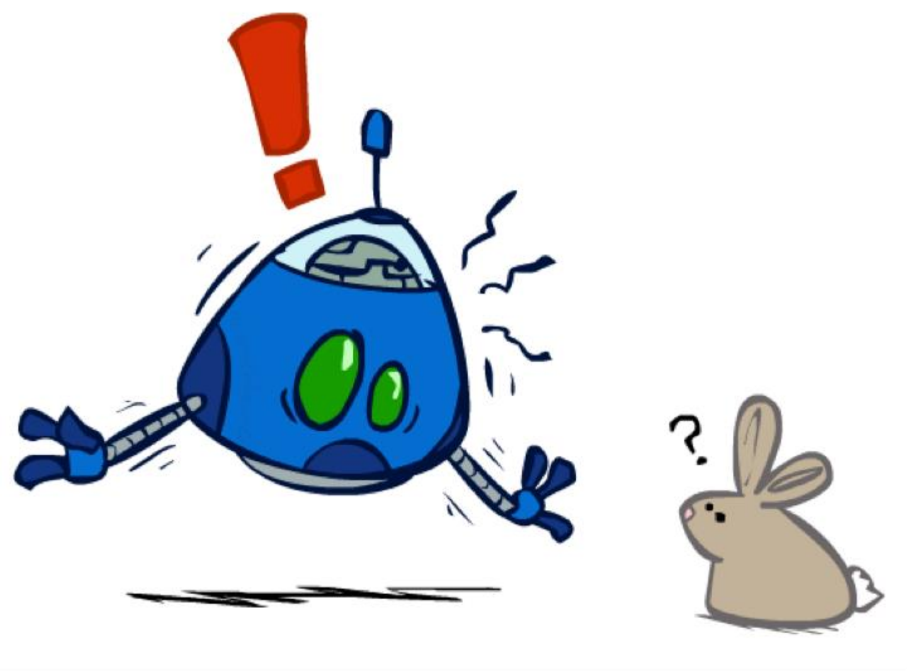
- Answer: Expectimax!
  - To figure out EACH chance node's probabilities, you have to run a simulation of your opponent
  - This kind of thing gets very slow very quickly
  - Even worse if you have to simulate your opponent simulating you…
  - … except for minimax and maximax, which have the nice property that it all collapses into one game tree

This is basically how you would model a human, except for their utility: their utility might be the same as yours (i.e. you try to help them, but they are depth 2 and noisy), or they might have a slightly different utility (like another person navigating in the office)

Dangerous Pessimism/Optimism

Dangerous Pessimism

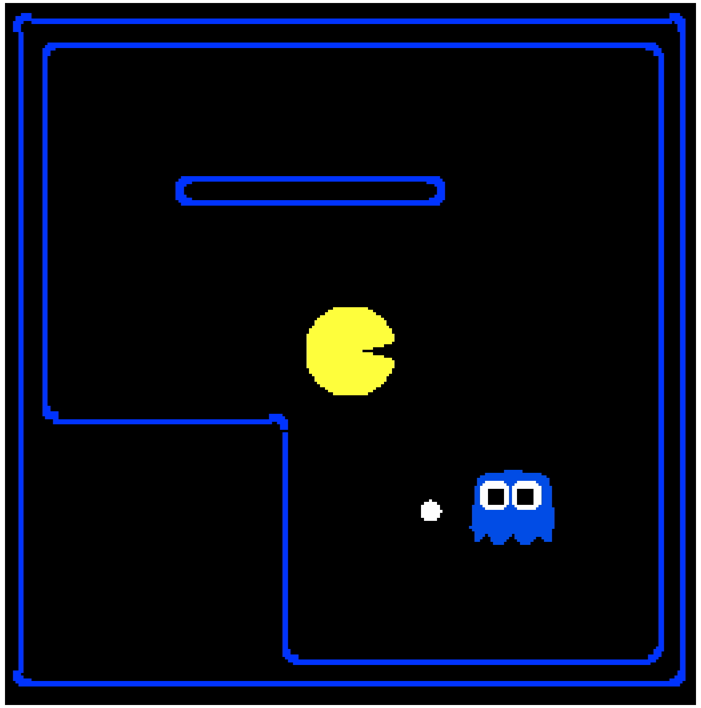
Assuming the worst case when it's not likely



Dangerous Optimism

Assuming chance when the world is adversarial

Assumptions vs. Reality



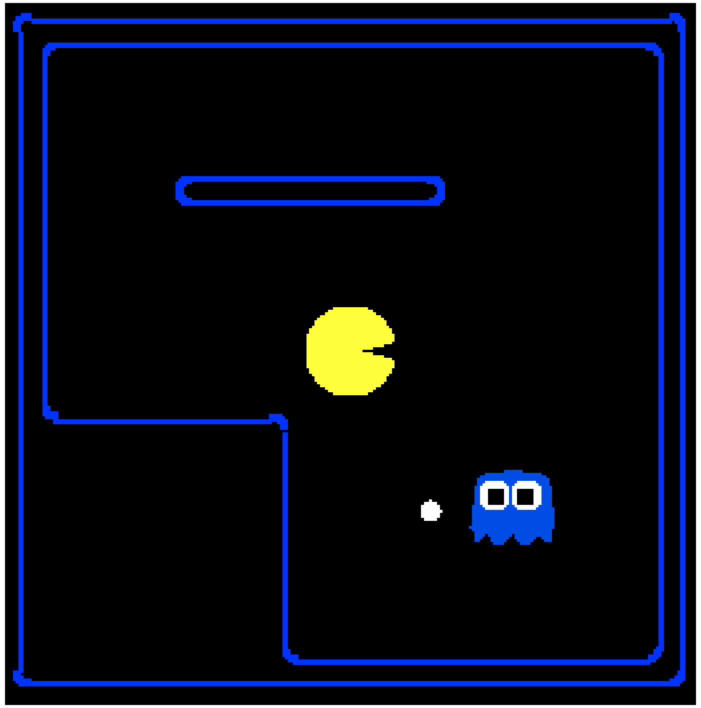
	Adversarial Ghost	Random Ghost
Minimax Pacman		
Expectimax Pacman		

Results from playing 5 games

Pacman used depth 4 search with an eval function that avoids trouble
Ghost used depth 2 search with an eval function that seeks Pacman

[Demos: world assumptions (L7D3,4,5,6)]

Assumptions vs. Reality 2



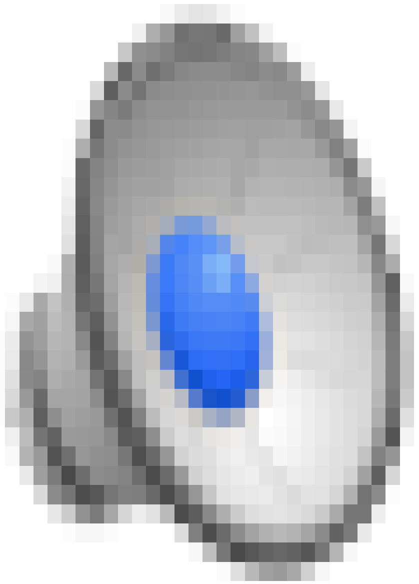
	Adversarial Ghost	Random Ghost
Minimax Pacman	Won 5/5 Avg. Score: 483	Won 5/5 Avg. Score: 493
Expectimax Pacman	Won 1/5 Avg. Score: -303	Won 5/5 Avg. Score: 503

Results from playing 5 games

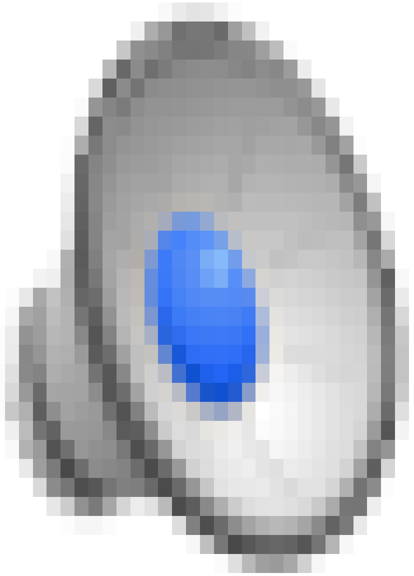
Pacman used depth 4 search with an eval function that avoids trouble
Ghost used depth 2 search with an eval function that seeks Pacman

[Demos: world assumptions (L7D3,4,5,6)]

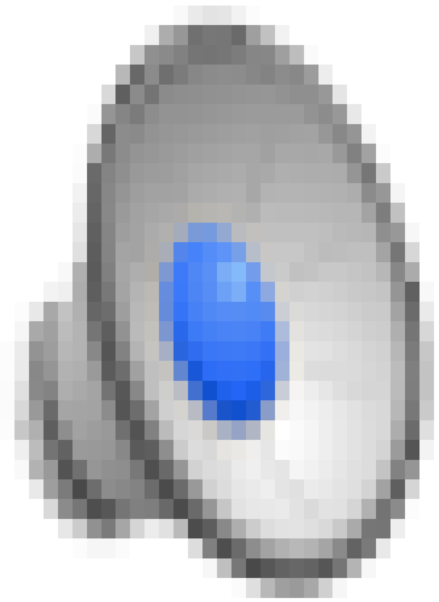
Video of Demo World Assumptions Random Ghost – Expectimax Pacman



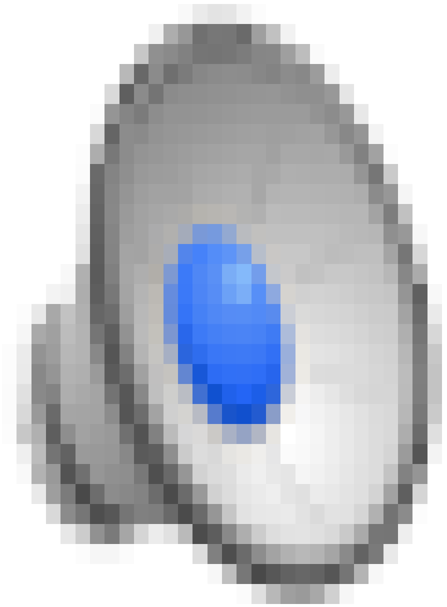
Video of Demo World Assumptions
Adversarial Ghost – Minimax Pacman



Video of Demo World Assumptions Adversarial Ghost – Expectimax Pacman

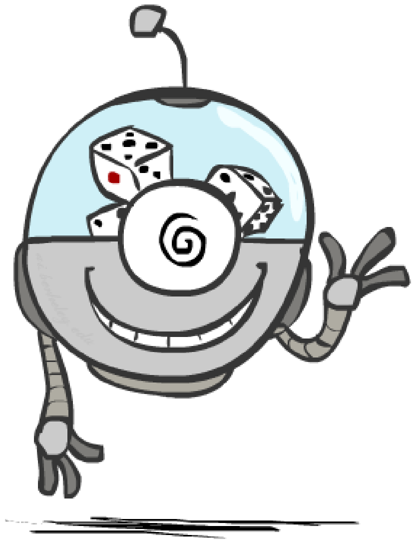


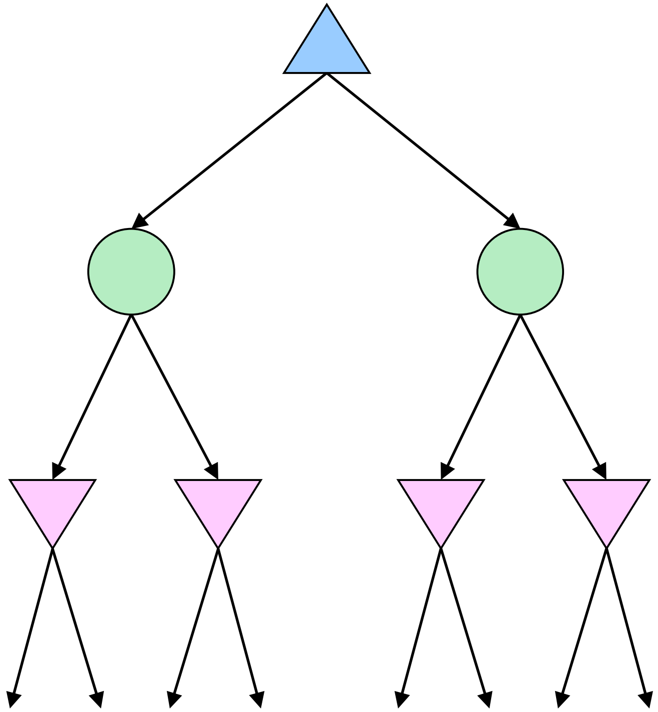
Video of Demo World Assumptions Random Ghost – Minimax Pacman



Mixed Layer Types

- E.g. Backgammon
- Expectiminimax
  - Environment is an extra "random agent" player that moves after each min/max agent
  - Each node computes the appropriate combination of its children





Example: Backgammon

- Dice rolls increase *b*: 21 possible rolls with 2 dice
  - Backgammon $\approx$ 20 legal moves
  - Depth 2 = $20 \times (21 \times 20)^3 = 1.2 \times 10^9$
- As depth increases, probability of reaching a given search node shrinks
  - So usefulness of search is diminished
  - So limiting depth is less damaging
  - But pruning is trickier…
- Historic AI: TDGammon uses depth-2 search + very good evaluation function + reinforcement learning: world-champion level play
- 1st AI world champion in any game!

Summary

Shuai Li

https://shuaili8.github.io

- Types of games
- Adversarial Game Trees: Minimax search
- Resource Limits I: Alpha-Beta Pruning
- Resource Limits II: Depth-limited search
    - Evaluation functions
    - Iterative deepening

Questions?