CS 232: Artificial Intelligence

Fall 2024

Prof. Carolyn Anderson
Wellesley College

Reminders

- Next reading is Thinking Humans Chapter 8-9 Check PDF
- I have help hours today from 3:30-4:30 in W422
- Lyn has help hours Sunday 4-6
- I have help hours Monday from 4-5:15
- Shortened class + video on Tuesday

You Look Like A Thing And I Love You, Chapters 3-4

Chapter 3 describes using a neural network to generate sandwich recipes. We'll learn more about this technique later. Consider generating a layer cake recipe as a search problem, where the states are layers:
the start state is a layer of cake.

How would you define the following components of the search problem?

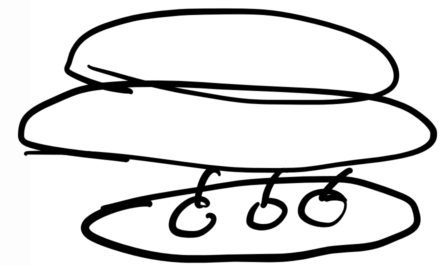
- Goal state
- Transition function
- Cost function

You Look Like A Thing And I Love You, Chapters 3-4

Start State: 1st layer of cake

Goal: based on desired layers
of cake
decorations
flavor compatibility
flavor versus decoration
structural integrity

Goal function



Transition function: flavors – which to pick next?
Size of layer
type of layer: frosting, filling, cake, decoration,

Cost: amount of ingredients
time to build
right flavor
$$$

Recap

Search

We've seen two kinds of search strategies so far:

- Uninformed search
  - Breadth-first search
  - Depth-first search
- Informed search
  - Uniform cost search — cost so far
  - Greedy best first search — heuristic-based guess about cost to reach the goal
  - A* search — cost to reach state + heuristic-based guess of cost to reach goal

$g(n)$

$h(n)$ how expensive was it to reach the state?

how expensive will it be to reach goal from here?

$f(n) = g(n) + h(n)$

Key: Admissibility

Inadmissible (pessimistic) heuristics break optimality by pushing good plans too far back on the frontier, which means they may never get expanded.

Admissible (optimistic) heuristics slow down bad plans but never outweigh true costs. That means that the true best plan will always be expanded.

A* search

Best-known form of best-first search.

Key Idea: avoid expanding paths that are already expensive, but expand most promising first.

Simple idea: *f(n)=g(n) + h(n)*

- ***g(n)*** the actual cost (so far) to *reach* the node
- ***h(n)*** estimated cost to *get from the node to the goal*
- ***f(n)*** estimated *total cost* of path through *n* to goal

Implementation: Frontier queue as priority queue by increasing *f(n)* *(as expected…)*

Adversarial Search

Search

So far, we have only considered one-player games.

What happens when we add another player?

Multiplayer Games

In competitive multiplayer games, we have to consider our opponent's possible actions, as well as our own.

We call this **adversarial search.**

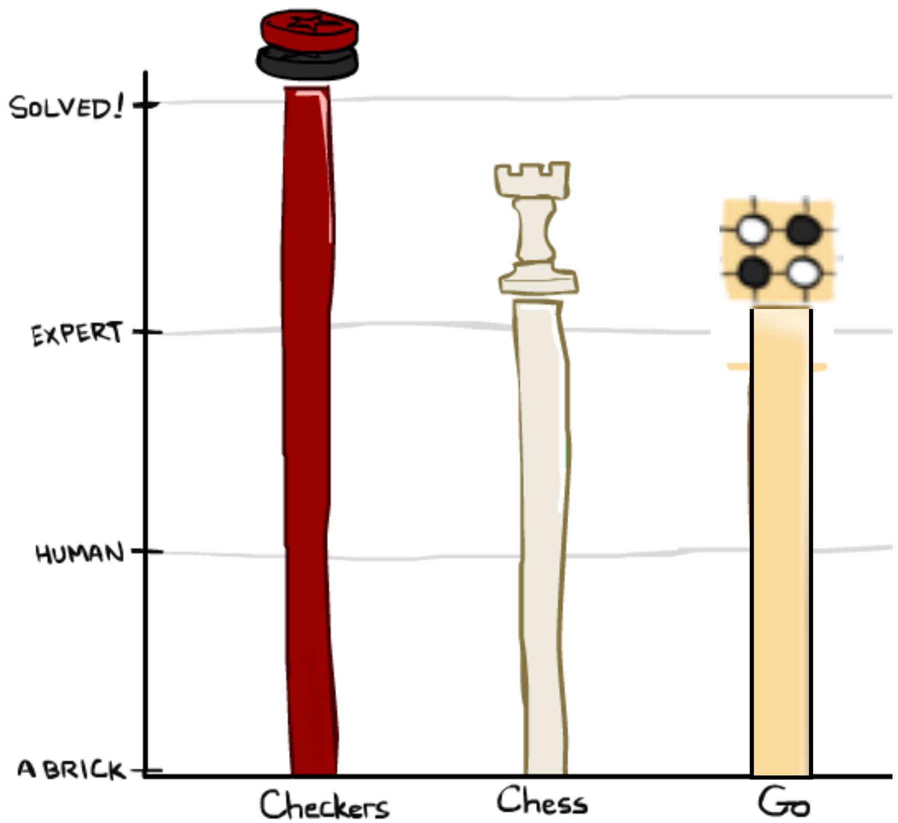
Game Playing State-of-the-Art

- **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!

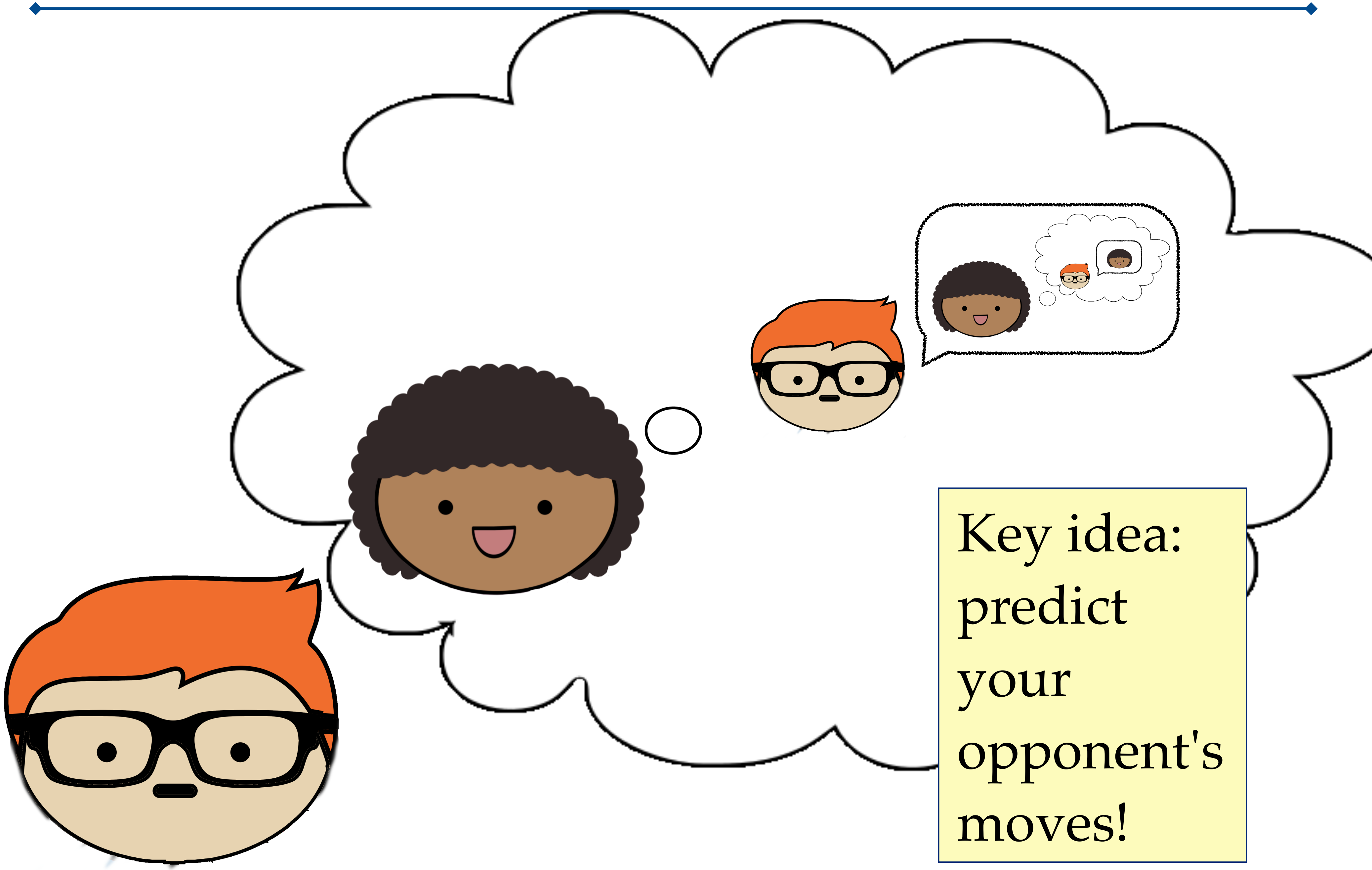
- **Chess:** 1997: Deep Blue defeats human champion Gary Kasparov in a six-game match. Deep Blue examined 200M positions per second, used very sophisticated evaluation and undisclosed methods for extending some lines of search up to 40 ply. Current programs are even better, if less historic.

- **Go: 2016: Alpha GO defeats human champion. Uses Monte Carlo Tree Search + neural network to learn evaluation function.**

- **Go + Chess + Shogi: 2017: Alpha Zero learns all 3 games using reinforcement learning to play against itself.**



Deterministic Games



Key idea: predict your opponent's moves!

Deterministic Games

States : S including a start state

Players : $P = \{1 \ldots N\}$

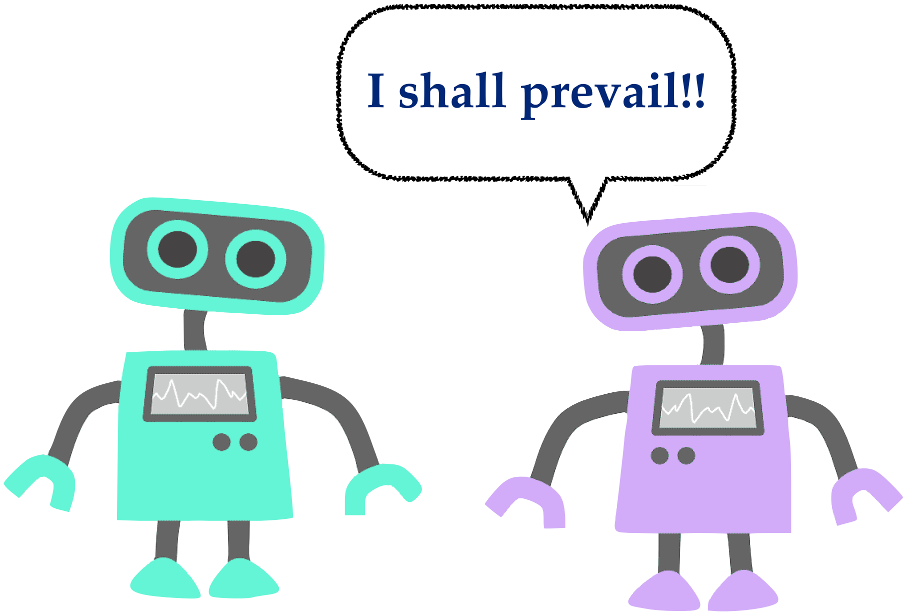
Actions : A (depend state & player)

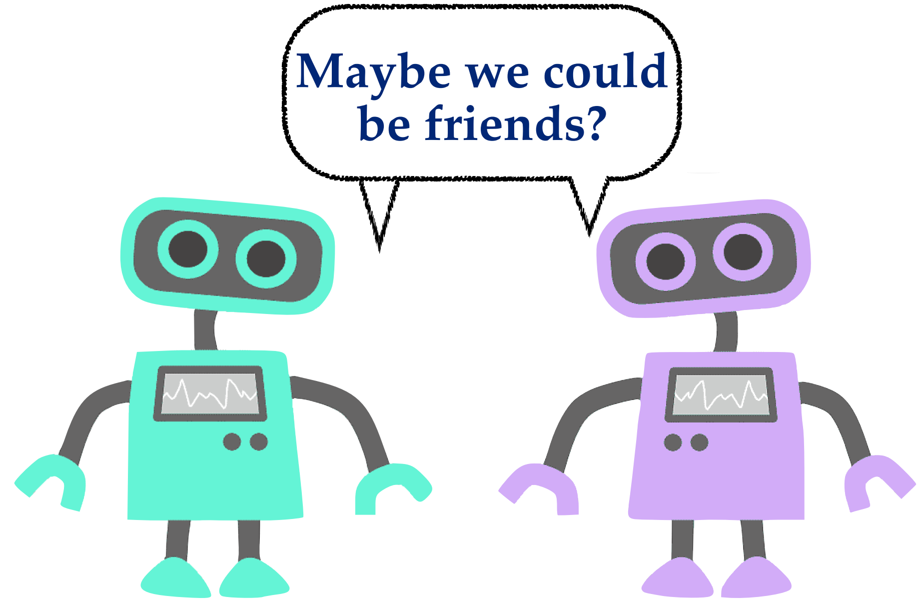
Transition Function : $T(s, a) \rightarrow s'$

Terminal test : Terminal(s)

Utility function : Utility(s)
(scoring end states) How good is that outcome?

Competing with Adversaries





Zero-Sum Games

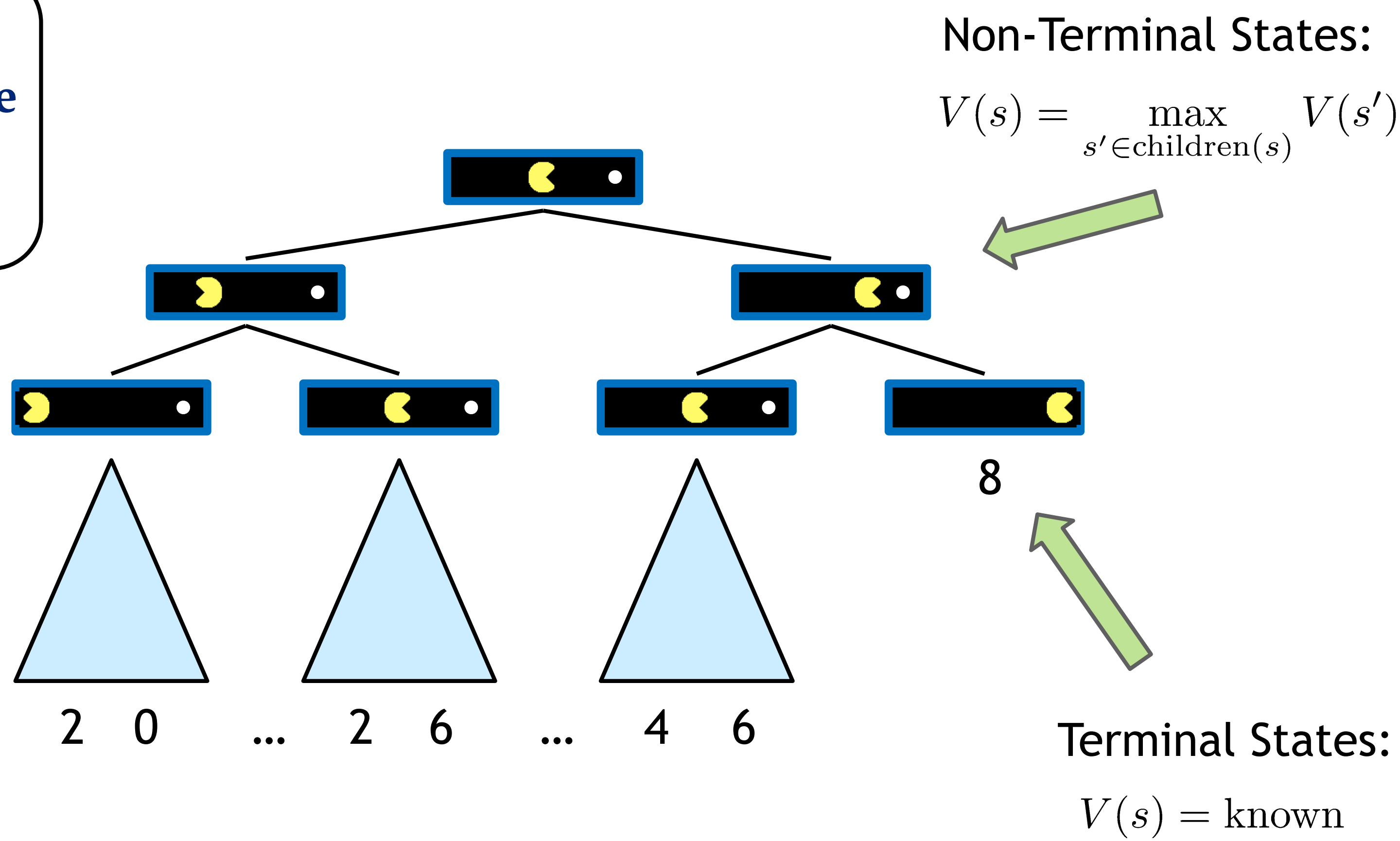
- Agents have opposite utilities (values on outcomes)
- A single score that one maximizes and the other minimizes
- Adversarial, pure competition

General Games

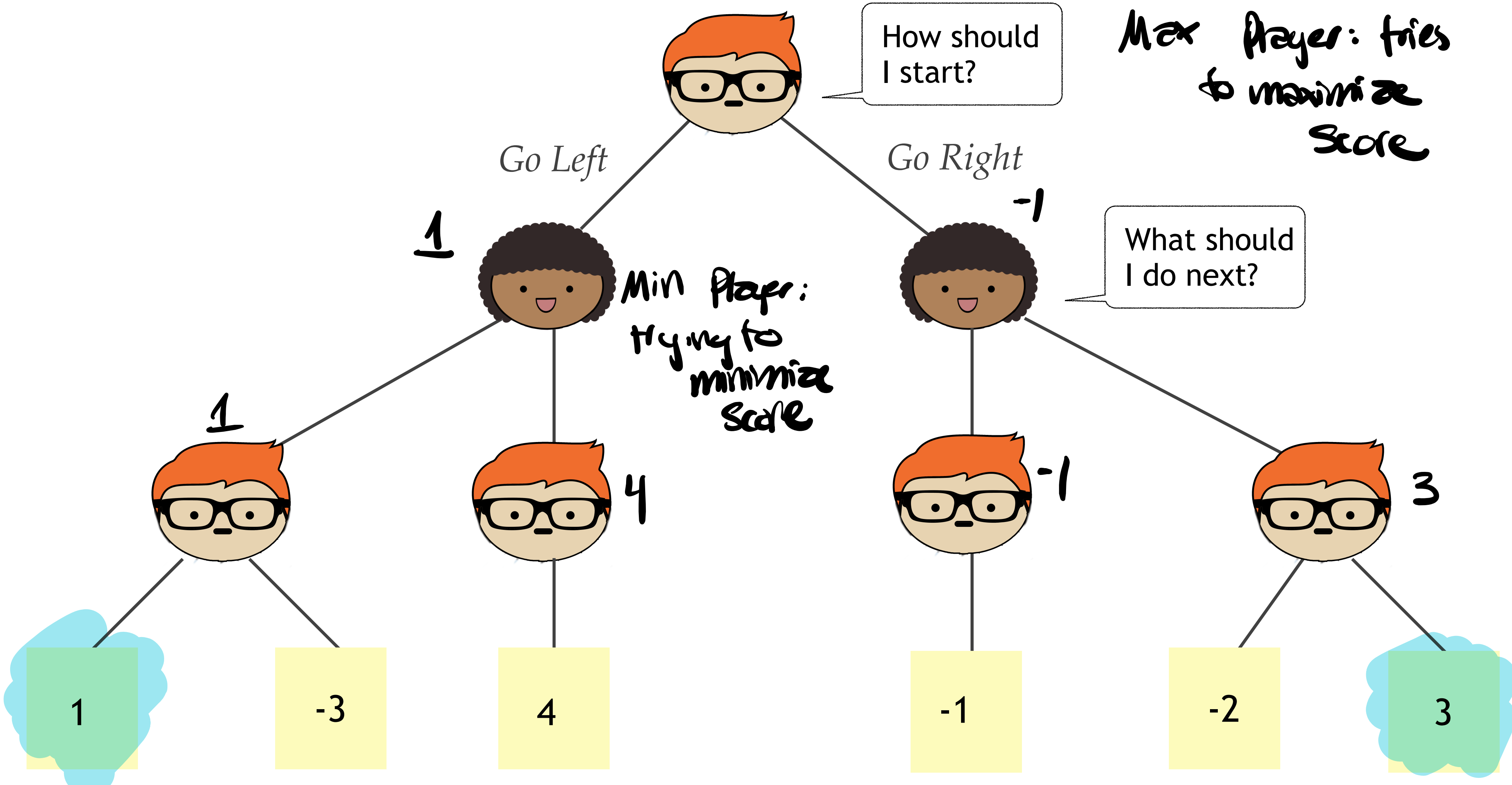
- Agents have independent utilities (values on outcomes)
- Cooperation, indifference, competition, and more are all possible

Value of a State

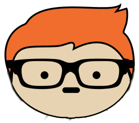
Value of a state:
The best achievable outcome (utility) from that state



Adversarial Game Trees

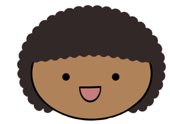


Minimax Values



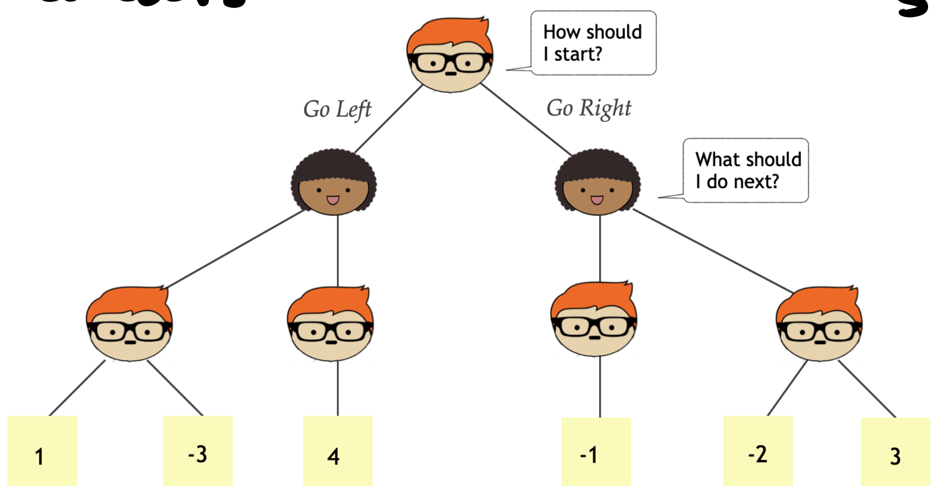
States Under Sam's Control:

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



States Under Thelma's Control:

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



Tic-Tac-Toe Game Tree

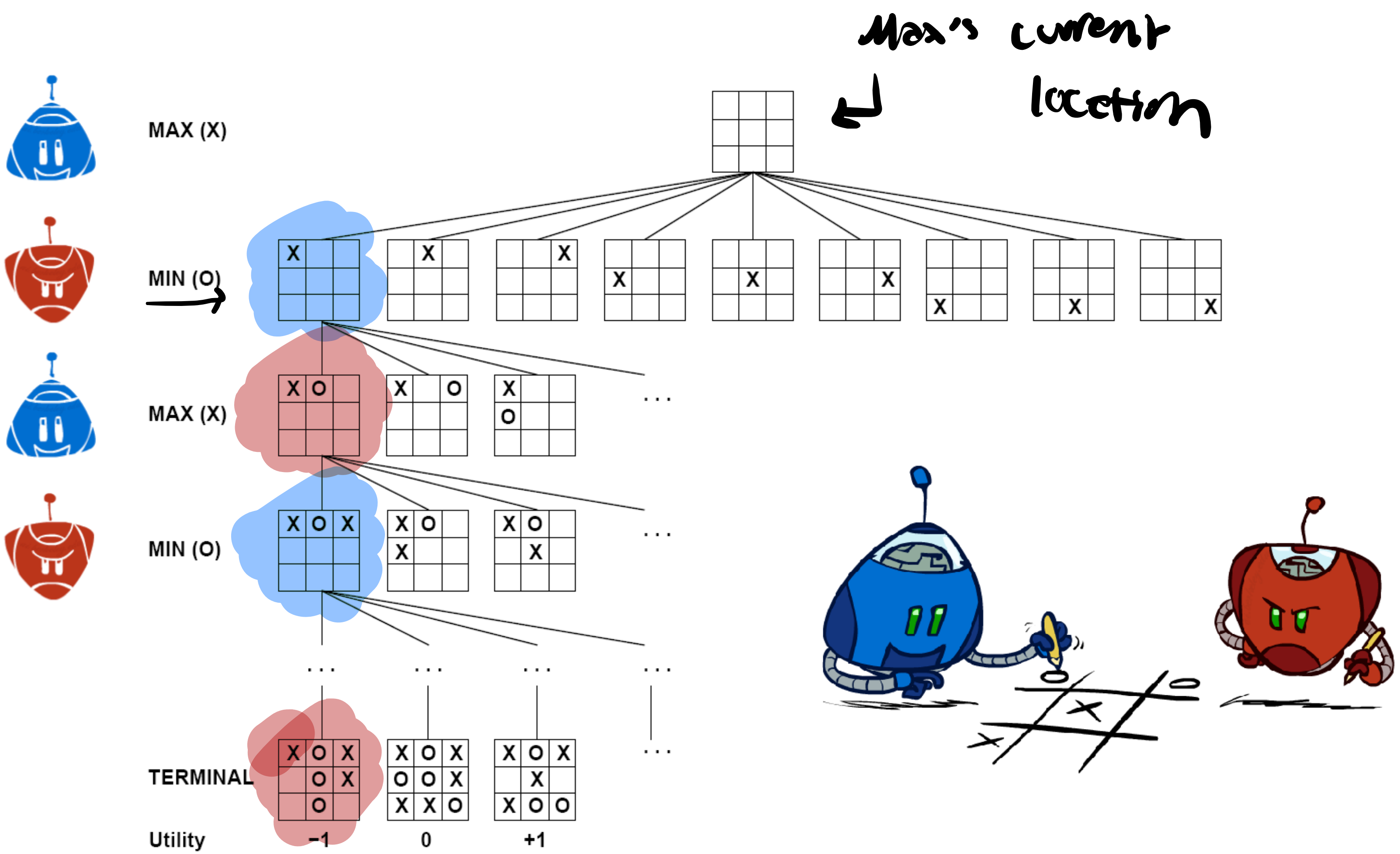


Image from CS188 Intro to AI at UC Berkeley

Minimax Search

In Minimax, we seek to optimize our score at the expense of our opponent.

We do this by reasoning recursively to predict their moves and compute the **expected utility** of various states we could reach.

Minimax Algorithm

```
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 Algorithm

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)

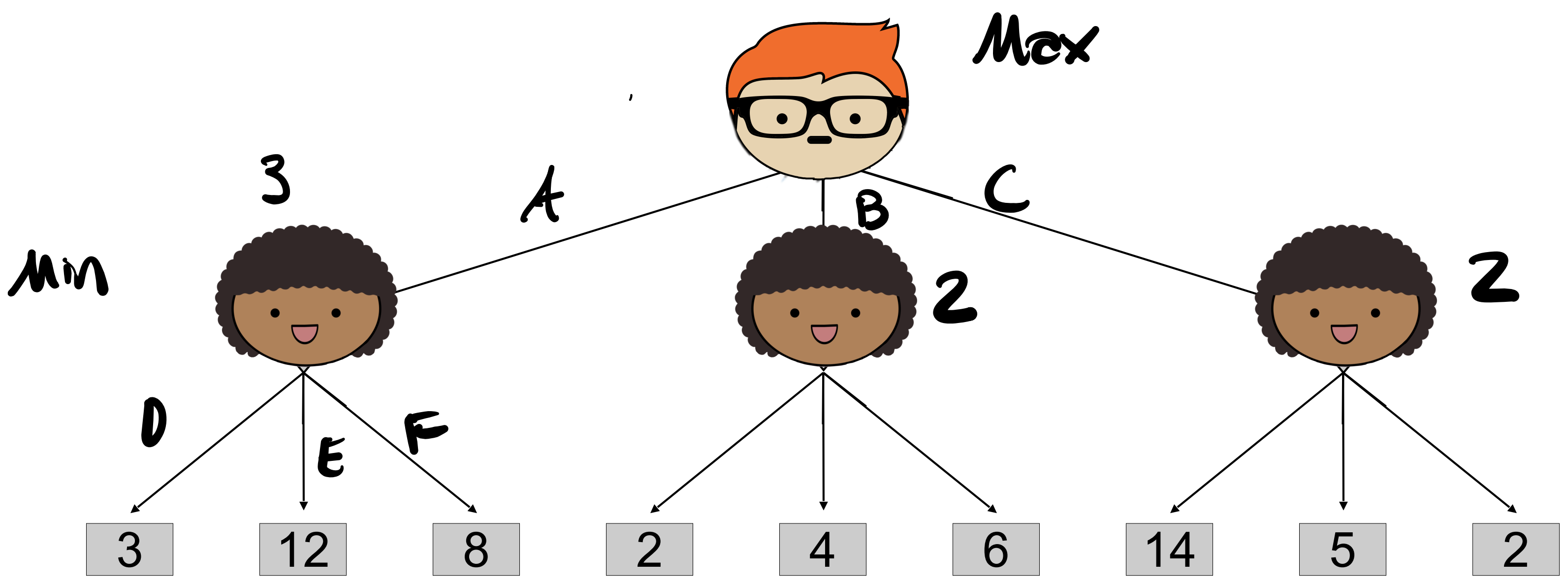
def max-value(state):
    initialize v = -∞
    for each successor of state:
        v = max(v, min-value(successor))
    return v

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

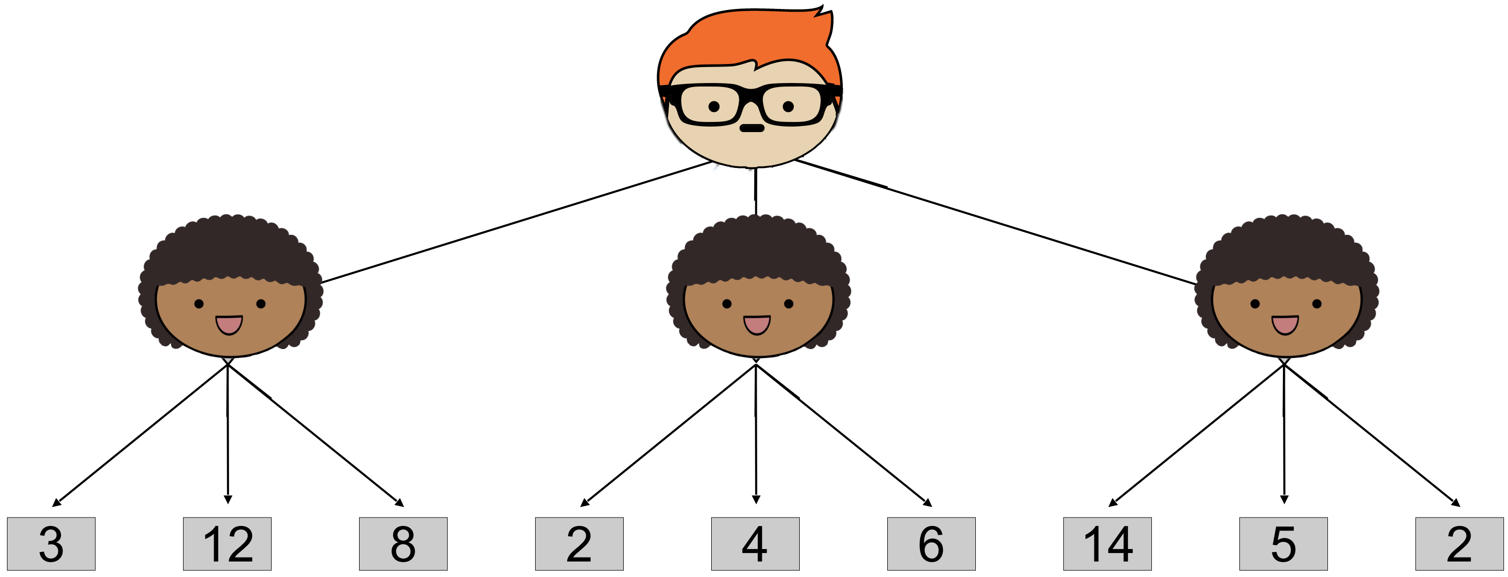
$$V(s) = \max_{s' \in \text{successors}(s)} V(s')$$

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

Minimax Example

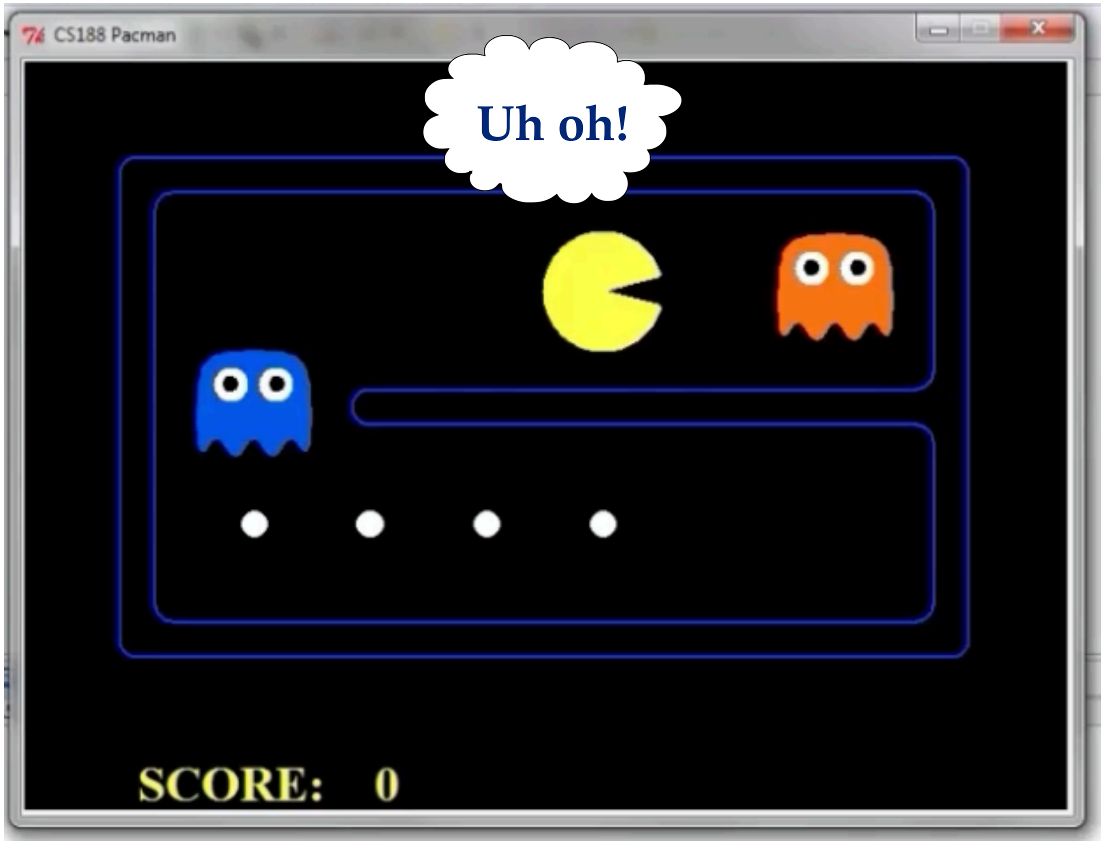


Minimax Example



Question: Is Minimax optimal?

Expectations v Reality: Pacman



Minimax Summary

- Rank final game states by their final scores (for tic-tac-toe or chess: win, draw, loss).
- Rank intermediate game states by whose turn it is and the available moves.
  - If it's X's turn, set the rank to that of the *maximum* move available. If a move will result in a win, X should take it.
  - If it's O's turn, set the rank to that of the *minimum* move available. If a move will result in a loss, X should avoid it.

Efficiency

Minimax Efficiency

How efficient is minimax?

- Just like (exhaustive) DFS
- Time: $O(b^m)$
- Space: $O(bm)$
- For chess, $b \approx 35$, $m \approx 100$

So, the exact solution is infeasible. But do we need to explore the whole tree?

Minimax Example

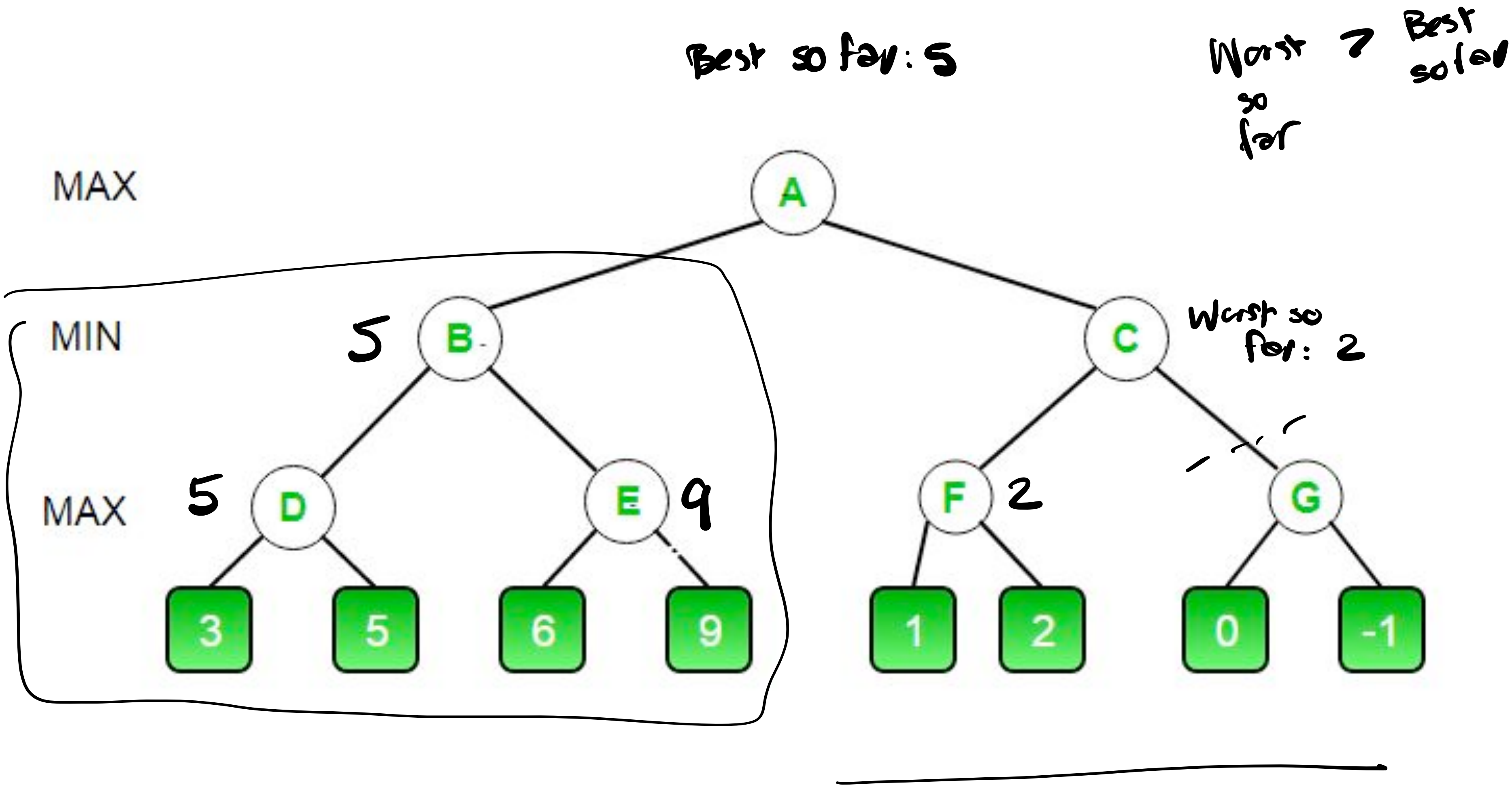


Image from www.geeksforgeeks.org/minimax-algorithm-in-game-theory-set-4-alpha-beta-pruning

Pruning

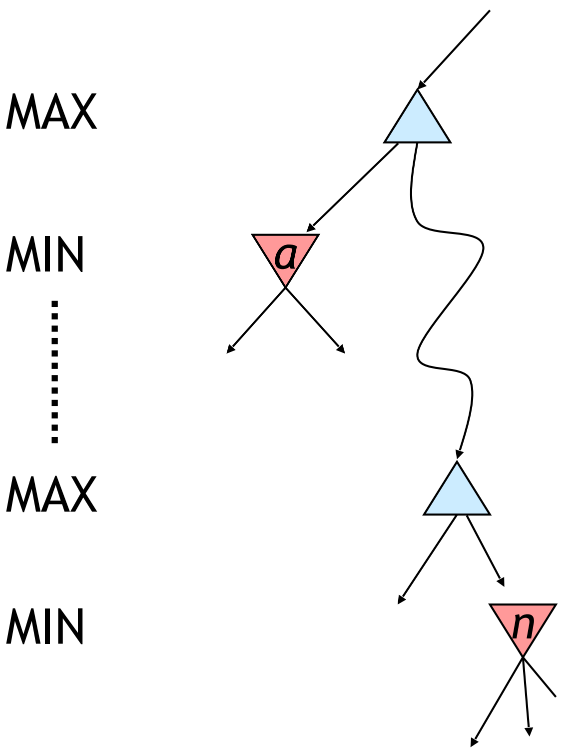
Key idea: give up on paths when you realize that they are worse than options you've already explore.

- Track the maximum score possible for the minimizing player (beta)
- Track the minimum score possible for the maximizing player (alpha)

Whenever the **maximum score for beta** becomes less than the **minimum score for alpha**, the maximizing player can stop searching down this path, because it will never be reached.

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
```