Artificial Intelligence

Lec 9: Searching with Non-Deterministic Actions, Environments with Multiple Agents

Pratik Mazumder

Content taken from various sources including
CS188 Intro to AI at UC Berkeley

Searching with Non-Deterministic Actions

- Till now, we have assumed that the **environment** is **fully observable and deterministic**.

- And the agent knows what the **exact effects of each action** are
  - can **“exactly determine”** which **state** results **from** any sequence of **actions** and always knows which state it is in.

- Therefore, percepts or observations from the environment do not provide any additional information after each action.

- In a **non-deterministic environment**, there can be **many** possible **outcomes** or **effects** of an action.

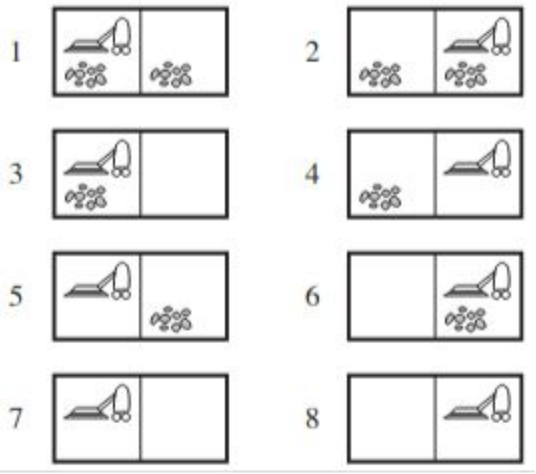
- When the environment is non-deterministic, percepts become useful.
  - percepts tell the agent which of the possible outcomes of its actions has actually occurred.

  - Therefore, the agent’s future actions will depend on the percepts

- **Solution** to a problem is **not a sequence** but a **contingency plan** (also known as a strategy) that specifies what to do depending on what percepts are received

Environment with Non-Deterministic Actions: Example

- The **erratic vacuum world** has two squares with dirt in them, and the vacuum cleaner is either in the left or right square.
- There are three actions—Left, Right, and Suck
- The state space has eight states and the goal is to clean up all the dirt (states 7 and 8).
- **Trivially solvable** with the **search algorithms** we know till now, **if** the **environment** is **observable, deterministic, and completely known**
  - The **solution** is an **action sequence**
- Now suppose that we introduce **nondeterminism** in the form of a powerful but erratic vacuum cleaner. E.g.
  - Suck action applied to a dirty square, cleans the square and sometimes cleans up dirt in an adjacent square, too.
  - Suck action applied to a clean square, the action sometimes deposits dirt on the carpet.

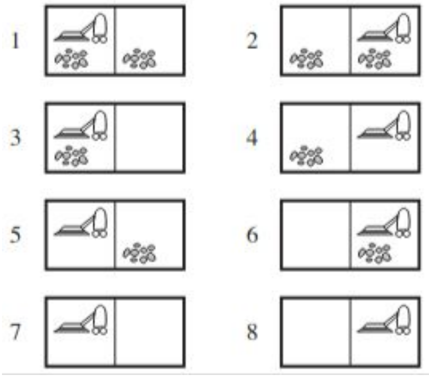


Can you predict the exact sequence of actions now?

No

Environment with Non-Deterministic Actions: Example

- Instead of **defining the transition model** by a RESULT function that returns a single state, we use a RESULTS function **that returns a set of possible outcome states**.
    - e.g., in the erratic vacuum world, the Suck action in state 1 leads to a state in the set {5, 7}, i.e., the dirt in the right-hand square may or may not be vacuumed up.



- We also need to **generalize** the notion of a **solution** to the problem.
    - e.g., if we start in state 1, there is **no single sequence of actions** that solves the problem.
    - Instead, we need a **contingency plan** such as the following:
      [Suck, if State = 5 then [Right, Suck] else [ ]]

- **Solutions** for nondeterministic problems can contain **nested if–then–else statements.**
    - this means that they are **trees rather than sequences**.
    - this allows the **selection of actions based on contingencies arising during execution**.
- Many problems in the real, physical world are contingency problems because exact prediction is impossible.

AND – OR Search Trees

- We need a search tree first to find solutions in this setting: AND-OR Search Trees
    - Contains OR nodes and AND nodes

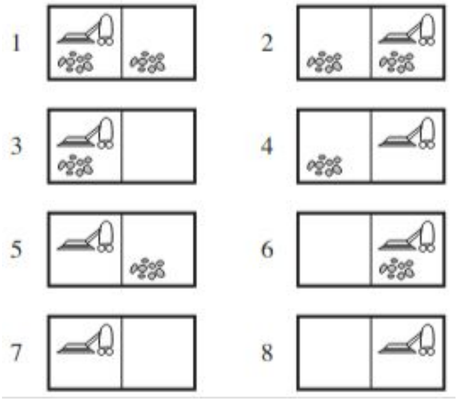
- In a deterministic environment, the only branching is introduced by the agent’s own choices in each state

- Nodes where the agent chooses a deterministic action to move to another state are called OR-Nodes

    - e.g. at an OR NODE node the agent chooses Left or Right or Suck

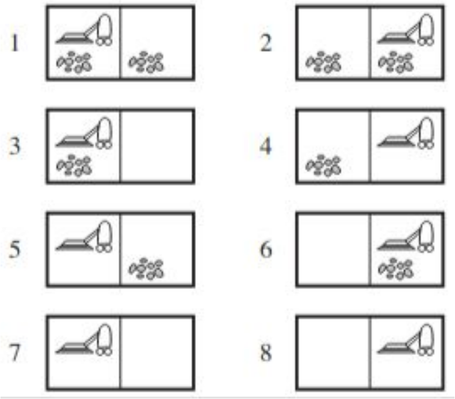
- In a nondeterministic environment, branching is also introduced by the **environment’s choice of outcome for each action**. These are the AND nodes.

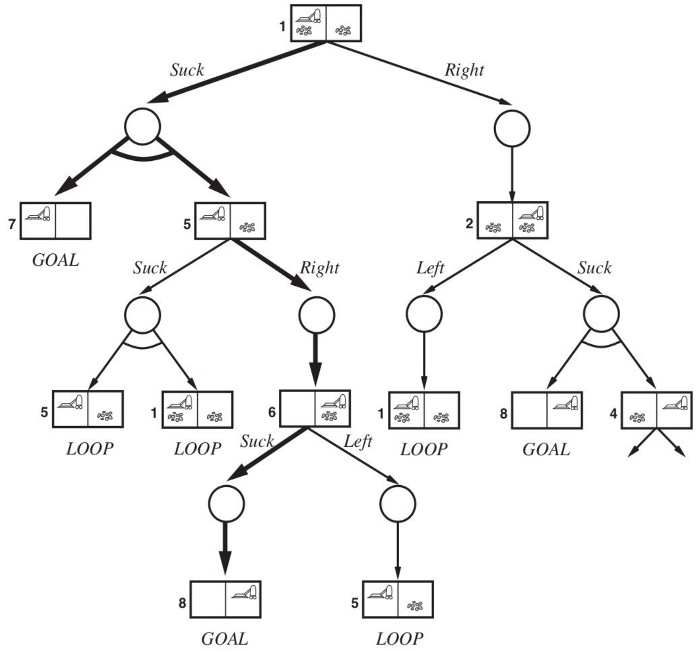
    - e.g. the Suck action in state 1 leads to a state in the set {5, 7}, so the agent would need to find a plan for state 5 and for state 7



AND – OR Search Trees

- A solution for an AND – OR search problem is a subtree that
  - has a goal node at every leaf,
  - specifies one action at each of its OR nodes,
  - includes every outcome branch at each of its AND nodes.
  - sometimes has a trimmed subtree in case of revisiting a node





Searching for solution

Recursive, depth-first algorithm for AND – OR graph search

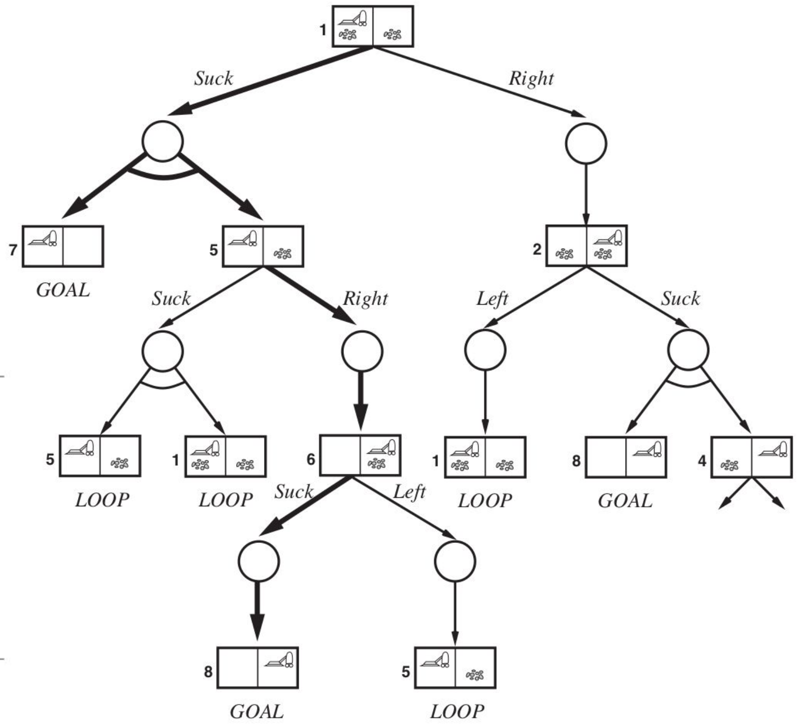
function AND-OR-GRAPH-SEARCH(*problem*) **returns** *a conditional plan, or failure*
  OR-SEARCH(*problem*.INITIAL-STATE, *problem*, [ ])

---

function OR-SEARCH(*state*, *problem*, *path*) **returns** *a conditional plan, or failure*
  if *problem*.GOAL-TEST(*state*) **then return** the empty plan
  if *state* is on *path* **then return** *failure*
  for each *action* **in** *problem*.ACTIONS(*state*) **do**
    plan ← AND-SEARCH(RESULTS(*state*, *action*), *problem*, [*state* | *path*])
    if *plan* ≠ *failure* **then return** [*action* | *plan*]
  return *failure*

---

function AND-SEARCH(*states*, *problem*, *path*) **returns** *a conditional plan, or failure*
  for each s_i **in** *states* **do**
    $plan_i$ ← OR-SEARCH(s_i, *problem*, *path*)
    if $plan_i$ = *failure* **then return** *failure*
  return [**if** s_1 **then** $plan_1$ **else if** s_2 **then** $plan_2$ **else** ... **if** s_{n-1} **then** $plan_{n-1}$ **else** $plan_n$]

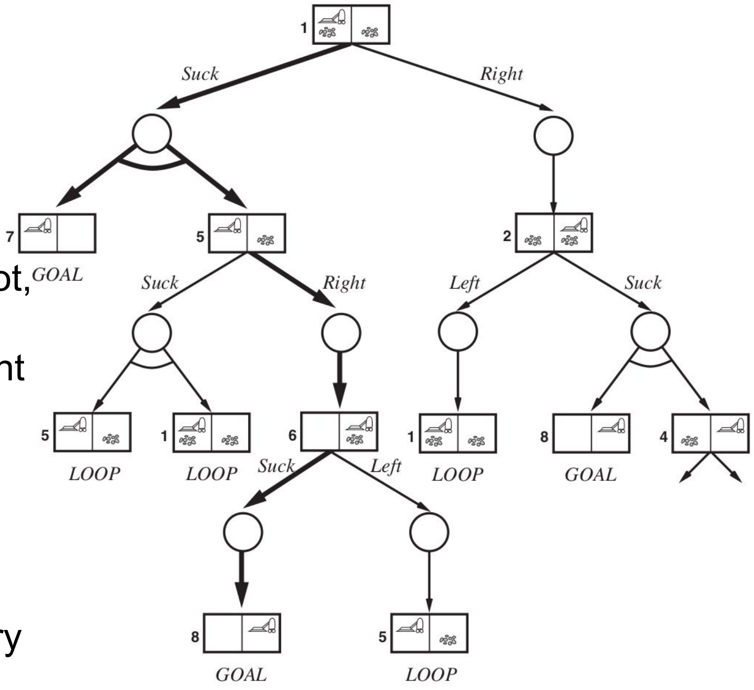


Searching for solution

Recursive, depth-first algorithm for AND – OR graph search

- If the current state is identical to a state on the path from the root, then it returns with failure.
  - This doesn’t mean that there is no solution from the current state.
  - it simply means that if there is a solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded.

- With this check, we ensure that the algorithm terminates in every finite state space, because every path must reach
  a goal, a dead end, or a repeated state

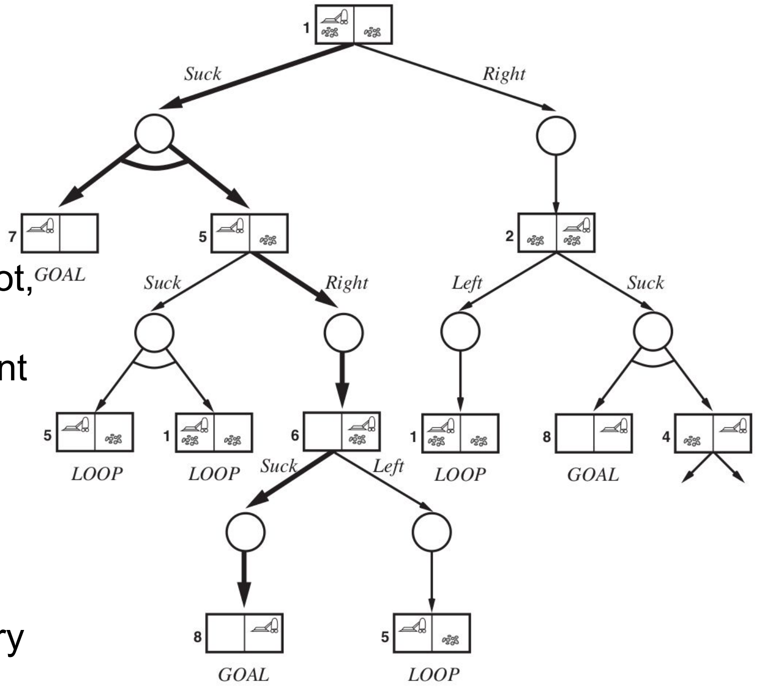


Searching for solution

Recursive, depth-first algorithm for AND – OR graph search

- If the current state is identical to a state on the path from the root, then it returns with failure.
  - This doesn't mean that there is no solution from the current state.
  - it simply means that if there is a solution, it must be reachable from the earlier incarnation of the current state, so the new incarnation can be discarded.

- With this check, we ensure that the algorithm terminates in every finite state space, because every path must reach a goal, a dead end, or a repeated state



Solution is not just a sequence of action. More like a tree.

Environments with Multiple Agents

- Until now, we have been looking into environments having a single agent interacting with the environment.

- Multiagent environments are environments in which **multiple** agents **interact with each other** to achieve their **individual goals**.
    - Compete or Cooperate or Both

- In chess, the opponent entity B is trying to maximize its performance measure, which, by the rules of chess, minimizes agent A’s performance measure.

    - Thus, it is a **competitive** multiagent environment.

- In the taxi-driving environment, avoiding collisions maximizes the performance measure of all agents

    - So it is a partially **cooperative** multiagent environment.

    - Also partially competitive because, for example, only one car can occupy a parking space.

Environments with Multiple Agents

- Multiple agents lead to more complex environments/ecosystems
  - Inspired by evolution
  - games, robotics, generative adversarial networks (GANs)
- We’ll focus on games, but multi-agent ideas come up in many areas of AI
- Why games?
  - They are usually good reasoning problems, formal (with fixed rules) and non-trivial
  - Direct comparison with humans and other computer systems easy, e.g. AlphaGo

Types of Games

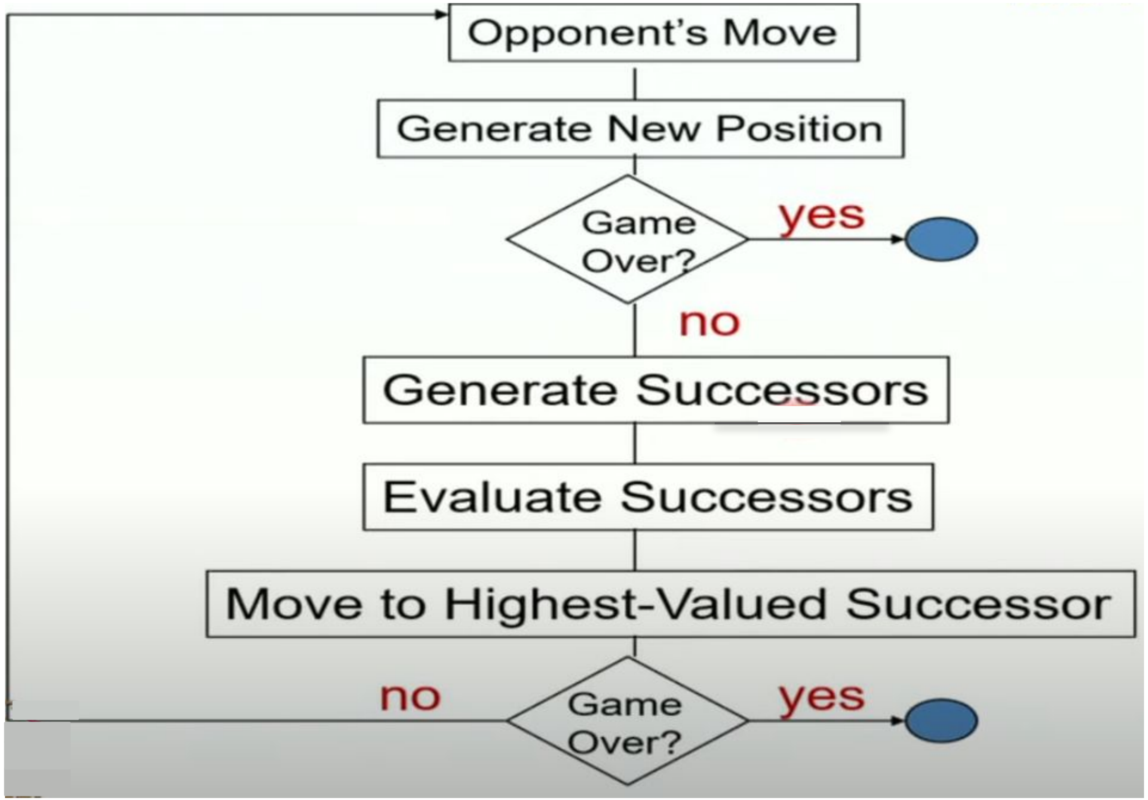
- Many different kinds of games!
  - Deterministic or stochastic?
    - Is there any randomness, like rolling a die?
    - Chess, Backgammon, Ludo
  - Zero sum?
    - Purely competitive or just a general multiplayer environment
  - Perfect information (can you see the state)?
    - Poker – Cards of opponent not visible
    - Chess – All pieces visible
- Want algorithms for calculating a strategy (policy) which recommends a move from each state

Games vs General Search Problems

Player 1 cannot plan a simple sequence of actions to reach the goal, considering only his/her possible moves.

- After player 1 makes a move, now player 2 **takes control** and makes the next move [Unpredictable opponent]

Two Player Game



Deterministic Games

- Many possible formalizations, one is:
  - States: S (start at s0)
  - Players: P={1...N} (usually take turns)
  - Actions: A (may depend on player / state)
  - Transition Function: SxA -> S
  - Terminal Test: S -> {true,false}
  - Terminal Utilities: SxP -> R
    - e.g. Win/Draw/Lose/Maximize Money
- Solution for a player is a policy: S -> A

Zero-Sum Games

- 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

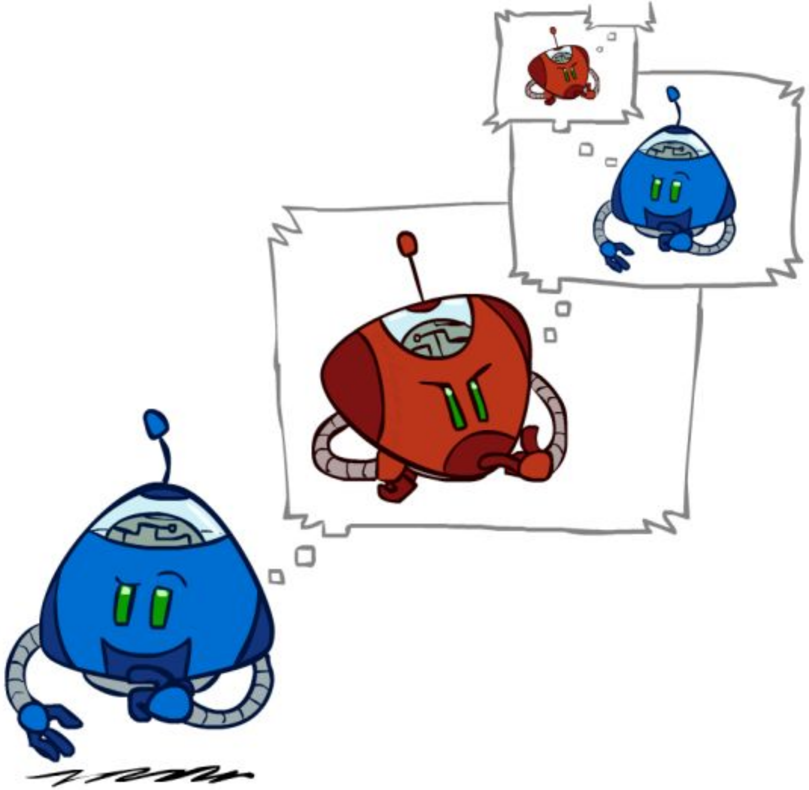
- General Games
  - Agents have independent utilities (values on outcomes).
  - Cooperation, indifference, competition, and more are all possible.

Solving Zero-Sum Games

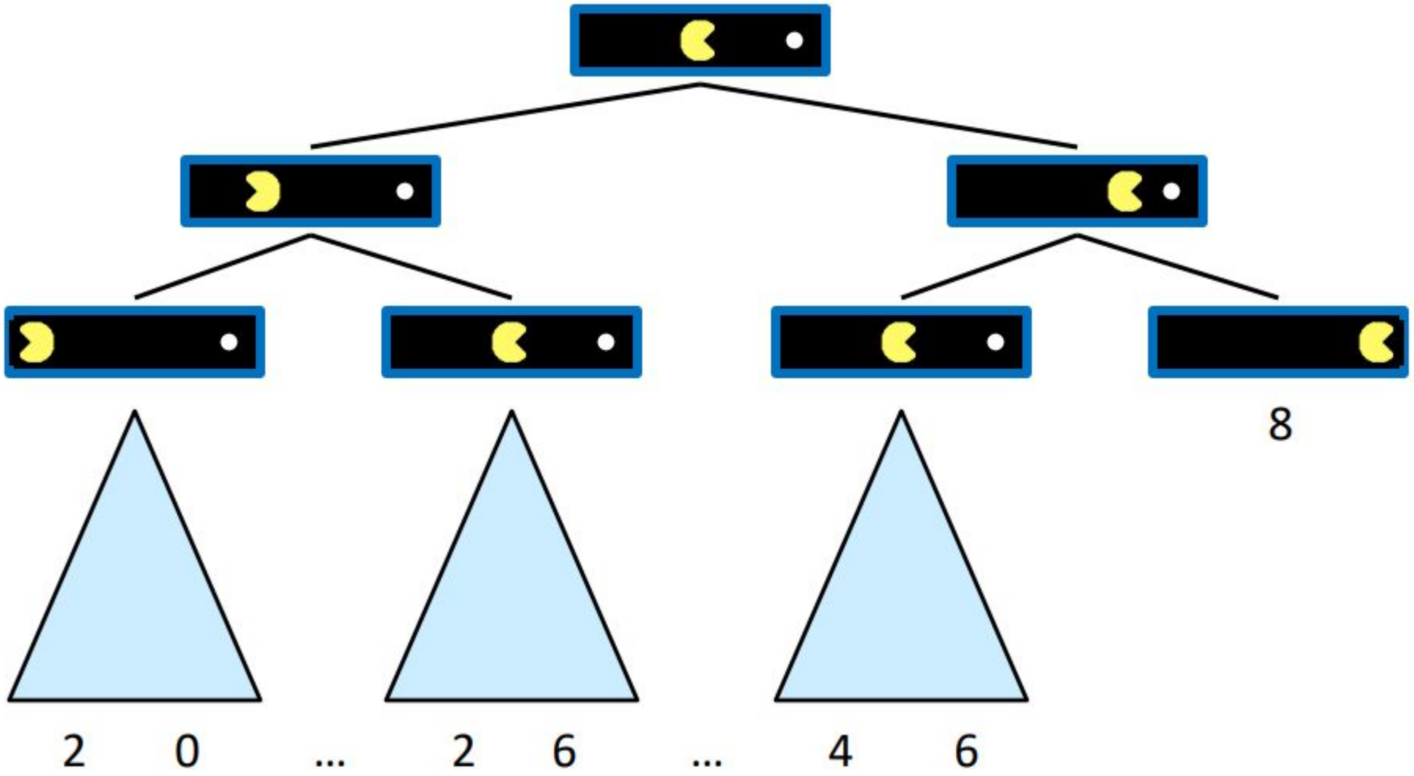
Similar to search

- but the player is not just looking at the future of its actions.
- but also thinking about what actions the other player would take as well.
- but if the other player was thinking about what actions it would take and so on

A recursive chain of “what if”

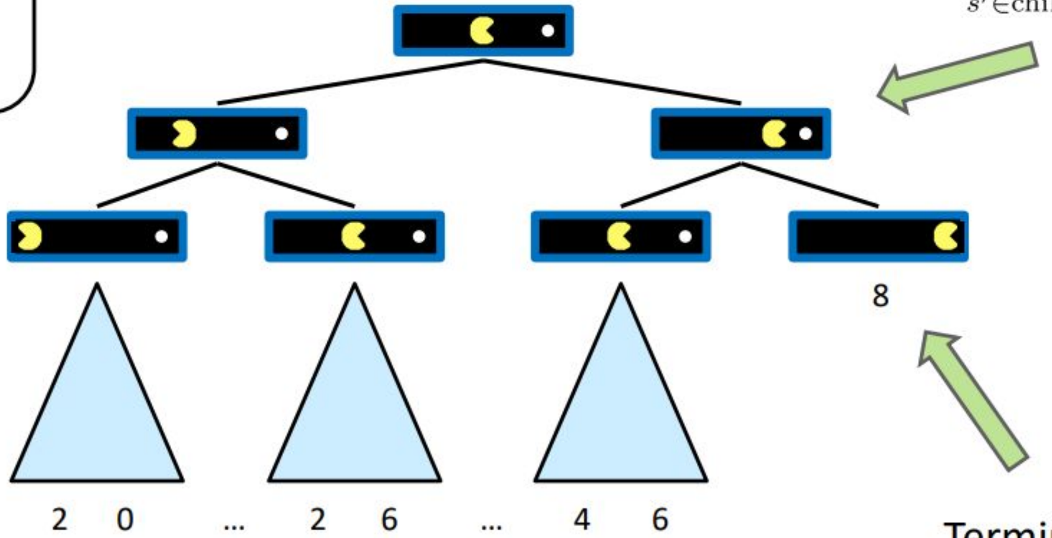


Single-Agent Search Trees



Value of a State

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



Non-Terminal States:

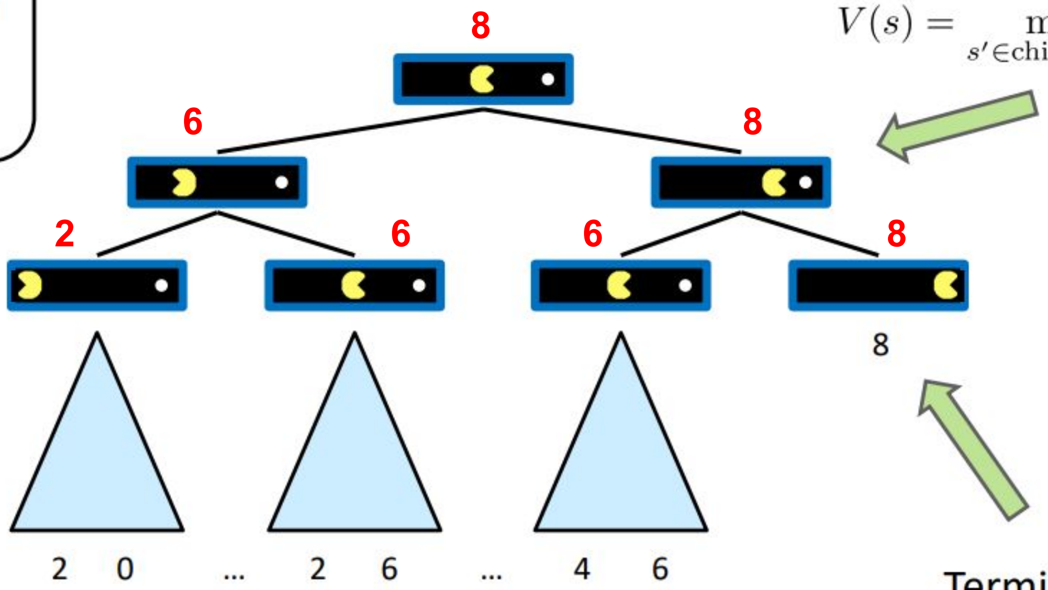
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$

Terminal States:

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

Value of a State

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



Non-Terminal States:

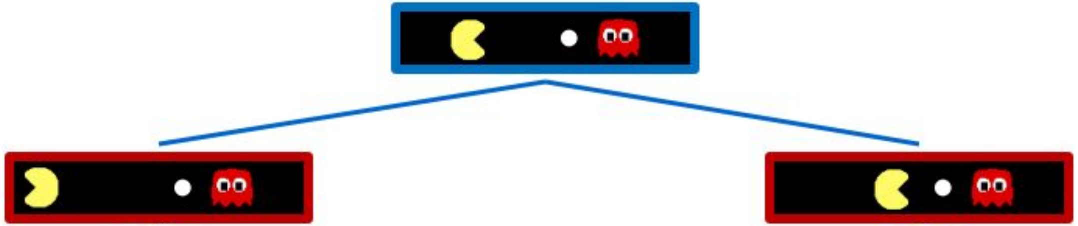
$$V(s) = \max_{s' \in \text{children}(s)} V(s')$$

Terminal States:

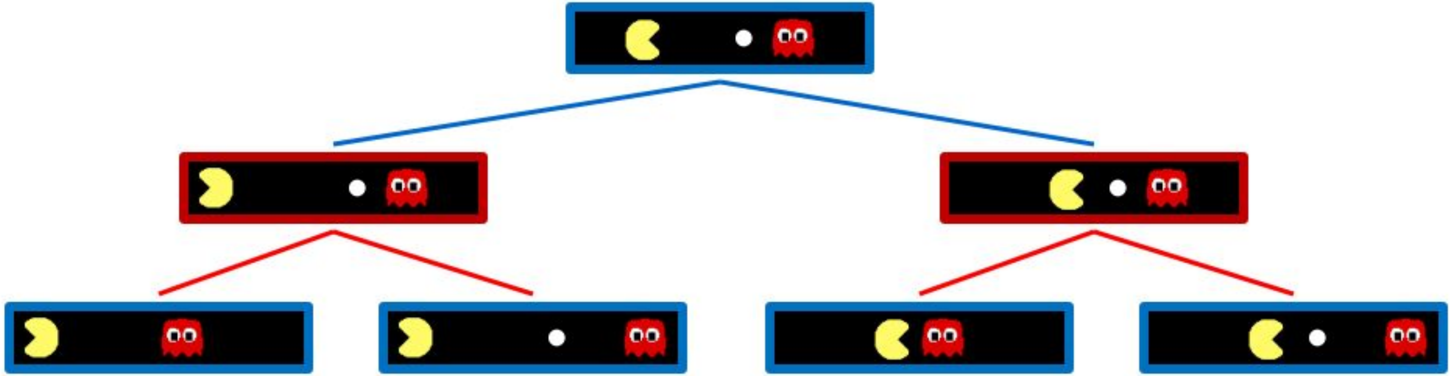
$$V(s) = \text{known}$$

Adversarial Game Trees

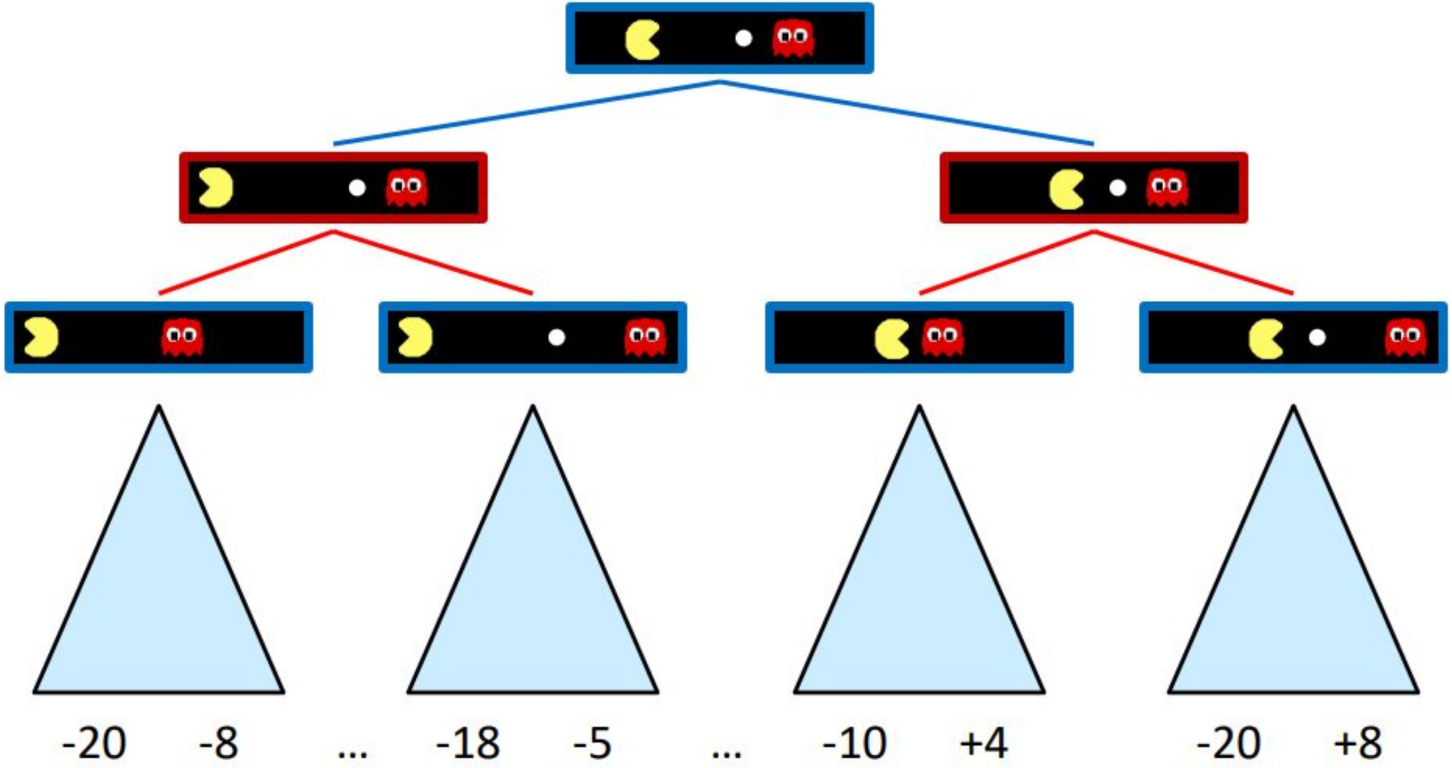
Adversarial Game Trees



Adversarial Game Trees



Adversarial Game Trees



Optimal Decision in Games

- In a **normal search** problem, the **optimal solution** would be a **sequence of actions** leading to a **goal** state.
    - a terminal state that is a win.

- In **adversarial search** with players MIN and MAX, MIN has something to say about the optimal solution for MAX, and vice versa.

- MAX therefore must find a contingent **strategy**, which specifies MAX’s move in the initial state.
    - then MAX’s moves in the states resulting from every possible response by MIN,
    - then MAX ’s moves in the states resulting from every possible response by MIN to those moves,
    - and so on.

- This is somewhat similar to the AND – OR search algorithm.

- Roughly speaking, an **optimal strategy** leads to **outcomes** that are at least **as good as** any other **strategy** when one is playing an **infallible opponent**.

Optimal Decision in Games

- Given a **game tree**, the **optimal** strategy can be determined from the **minimax value of each node**.

- The **minimax** value of a node is the **utility** (for player MAX) of being in the **corresponding state**, **assuming** that both **players play optimally** from there to the end of the game.

- The minimax value of a terminal state is just its utility (known).

- Given a choice,

  - **player MAX** prefers to **move** to a state of **maximum** value,

  - whereas player **MIN** prefers a state of **minimum** value

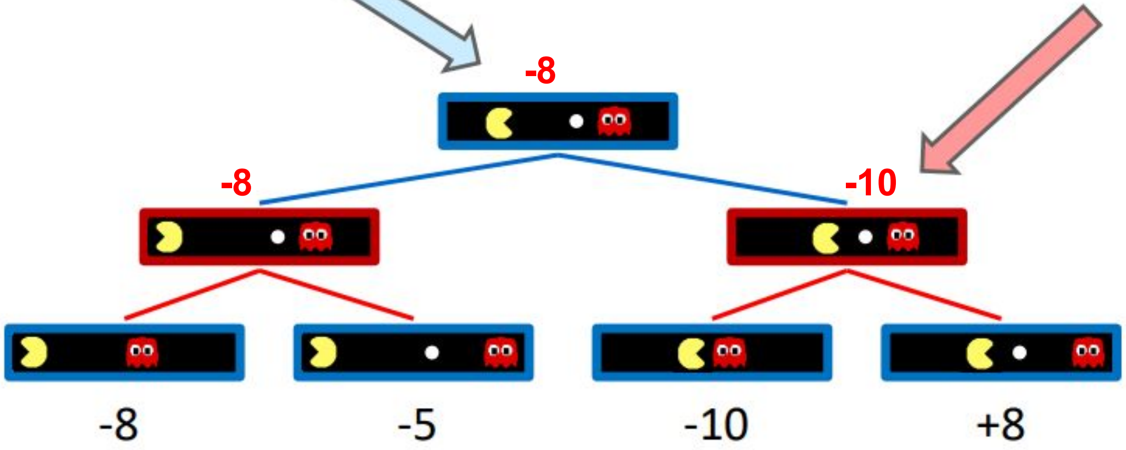
Optimal Decision in Games: Minimax Values

States Under Agent's Control:

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

States Under Opponent's Control:

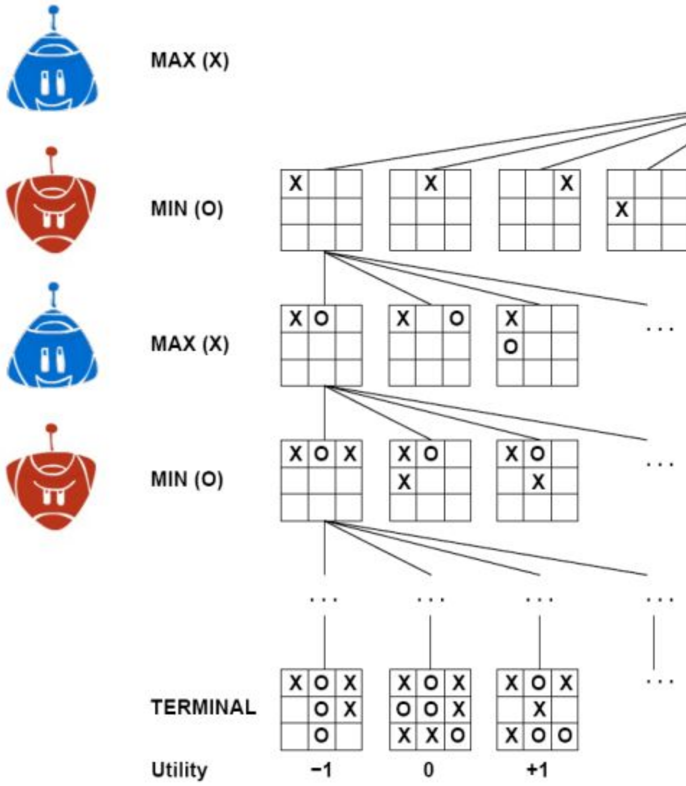
$$V(s') = \min_{s \in \text{successors}(s')} V(s)$$

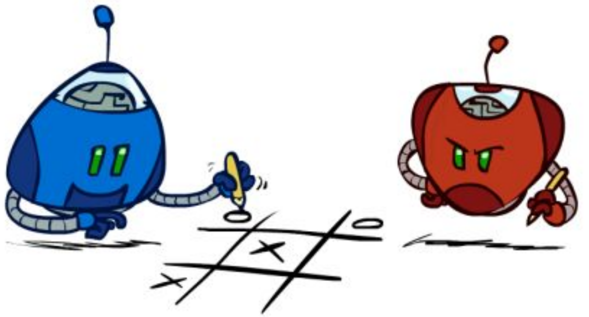


Terminal States:

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

Tic-Tac-Toe Game Tree





Adversarial Search (Minimax)

- 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**