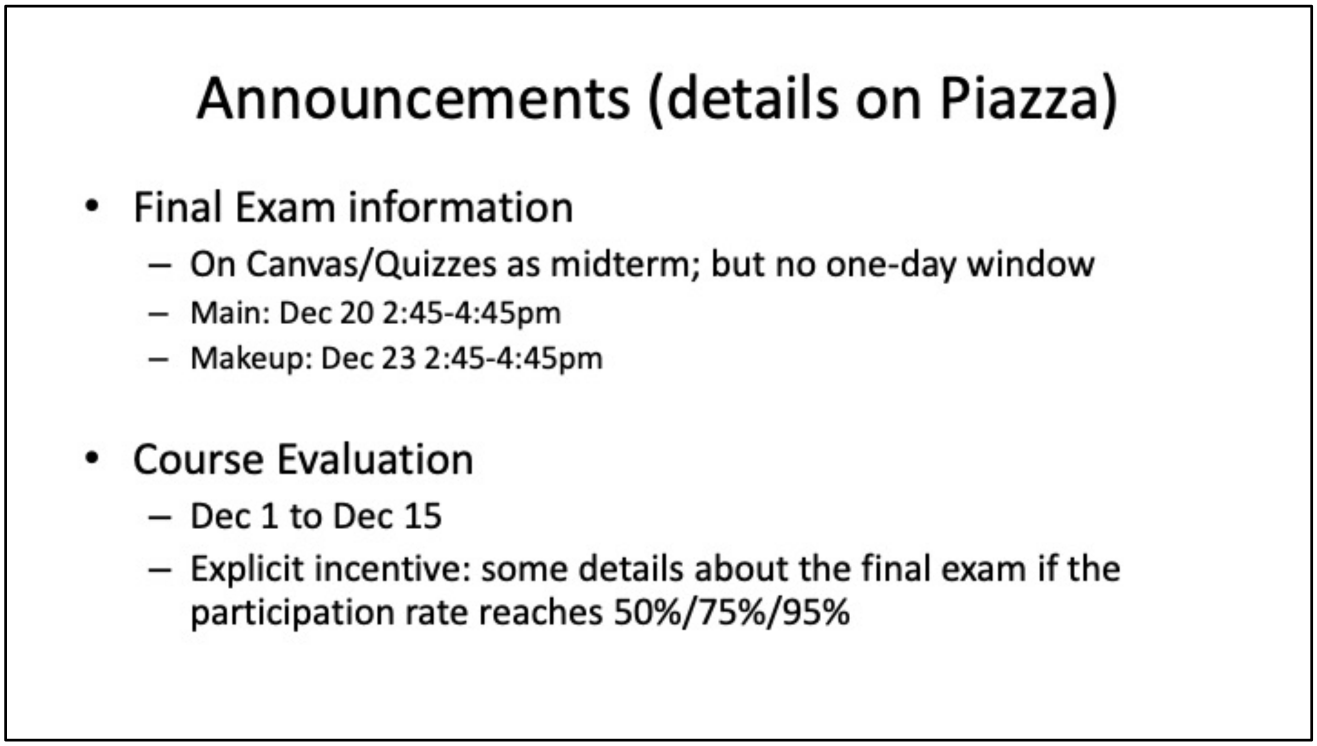
CS 540 Introduction to Artificial Intelligence
Review on Search, Games, and RL

Yingyu Liang
University of Wisconsin-Madison

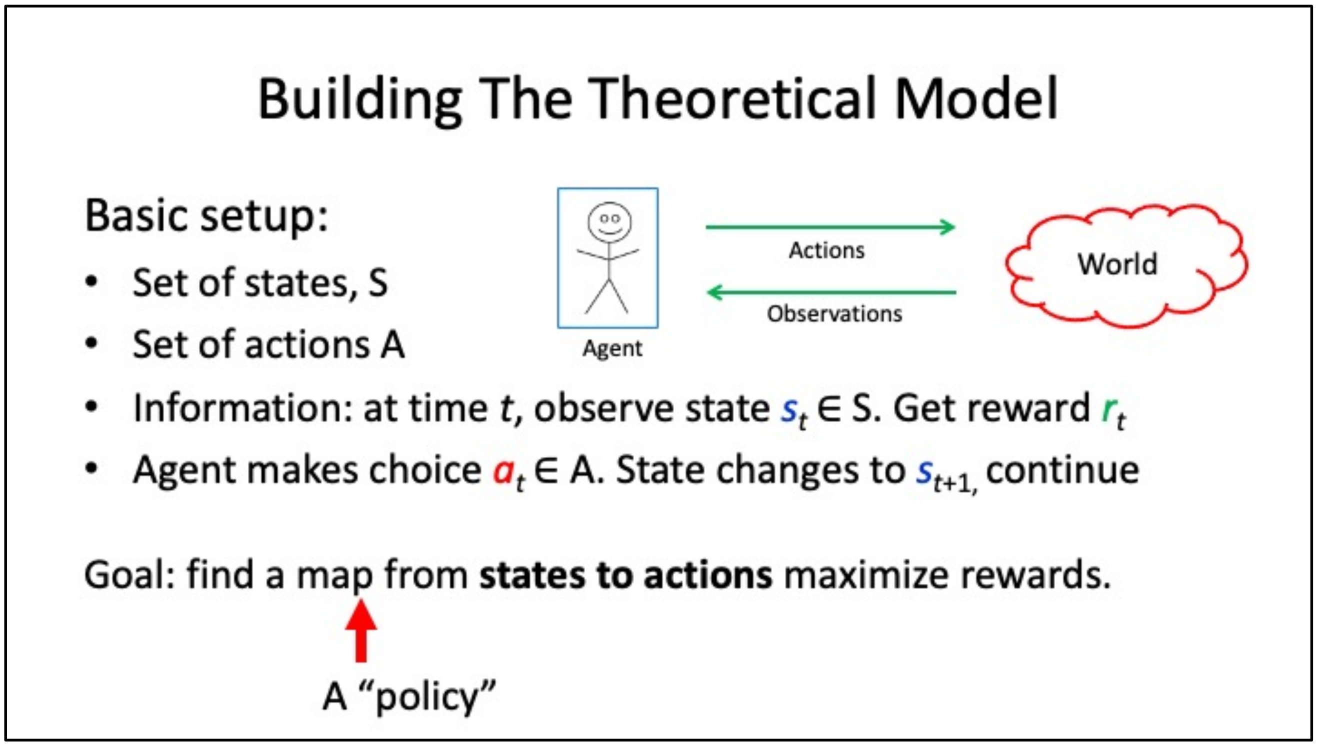
Dec 9, 2021

Based on slides by Fred Sala



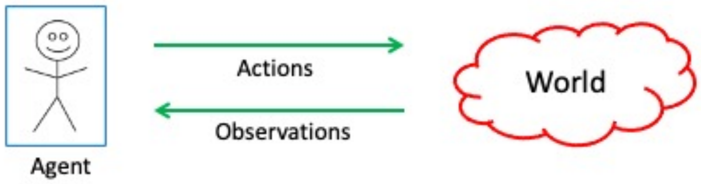
Announcements (details on Piazza)

- Final Exam information
  - On Canvas/Quizzes as midterm; but no one-day window
  - Main: Dec 20 2:45-4:45pm
  - Makeup: Dec 23 2:45-4:45pm
- Course Evaluation
  - Dec 1 to Dec 15
  - Explicit incentive: some details about the final exam if the participation rate reaches 50%/75%/95%



Building The Theoretical Model

Basic setup:



- Set of states, S
- Set of actions A
- Information: at time t, observe state $s_t \in$ S. Get reward r_t
- Agent makes choice $a_t \in$ A. State changes to $s_{t+1,}$ continue

Goal: find a map from **states to actions** maximize rewards.

A "policy"

We will go through these three steps: build the math model, introduce the notion of values that comes from our goal and enables decision making, and then design algorithms for computing the values.

We will begin with building the math model.

Recall that we have an agent interacting with the world. The interaction happens in rounds. In each round, the agent has some observations and takes some actions. The actions can change the state of the world, and the observations should consist of rewards for the agent.

To describe the state of the world, we introduce a state space S. To describe the action, we introduce an action space A.
What about observations? Here we consider the observations consist of the state of the world and the reward. More precisely, at time t, the agent can observe the state of the world at that time denoted as s_t, and get a reward at that time denoted as r_t.

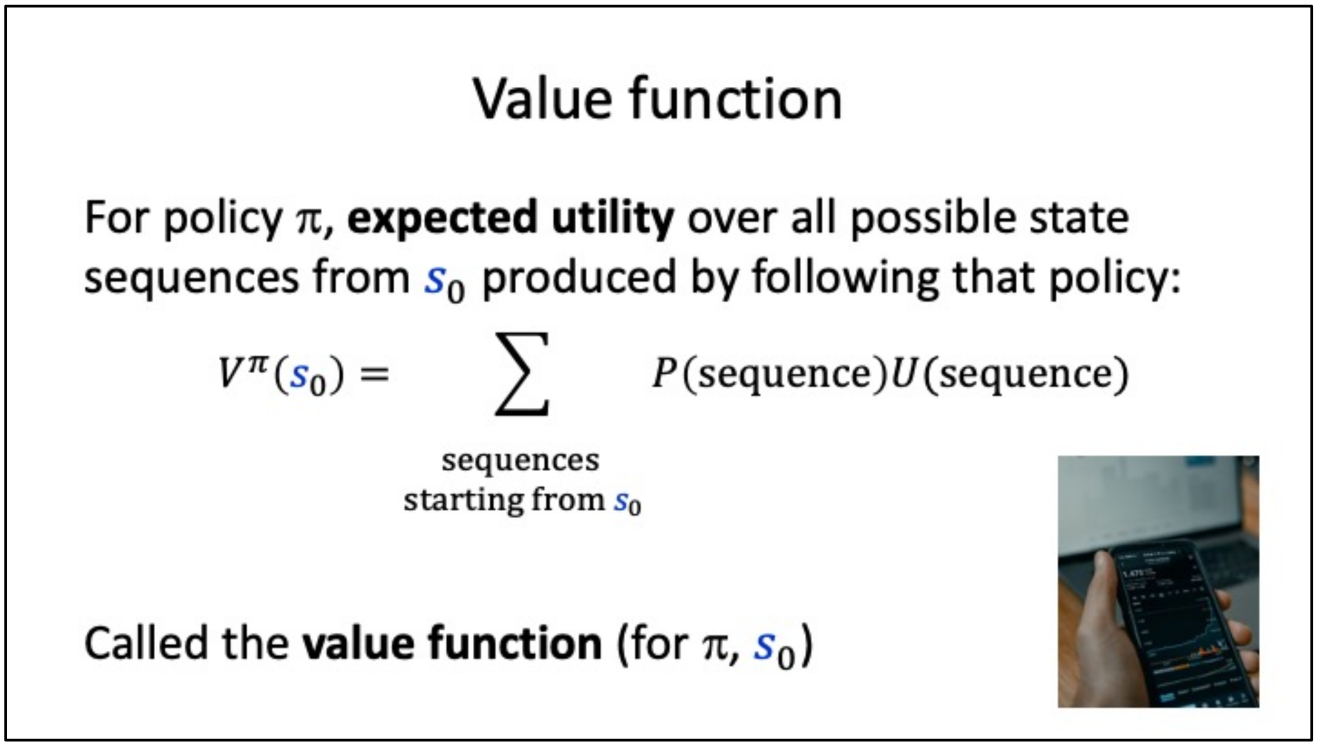
In summary, at each iteration t, the agent observes the state s_t and get a reward r_t, and then takes an action a_t. Then the state of the world changes to s_{t+1}, and we go to the next iteration.

The goal is then to take actions to maximize the rewards. More precisely, we would like to have a decision function that takes as input a state and output an action. This is called a policy, which is a map from states to actions.

3 things to be formalized:

1. State transition
2. Reward function
3. Policy maximizing the reward

For the first two we use Markov assumption leading to the MDP framework. For the last, we introduce the value function and define the optimal policy as the policy maximizing the value.



Value function

For policy π, **expected utility** over all possible state sequences from s_0 produced by following that policy:

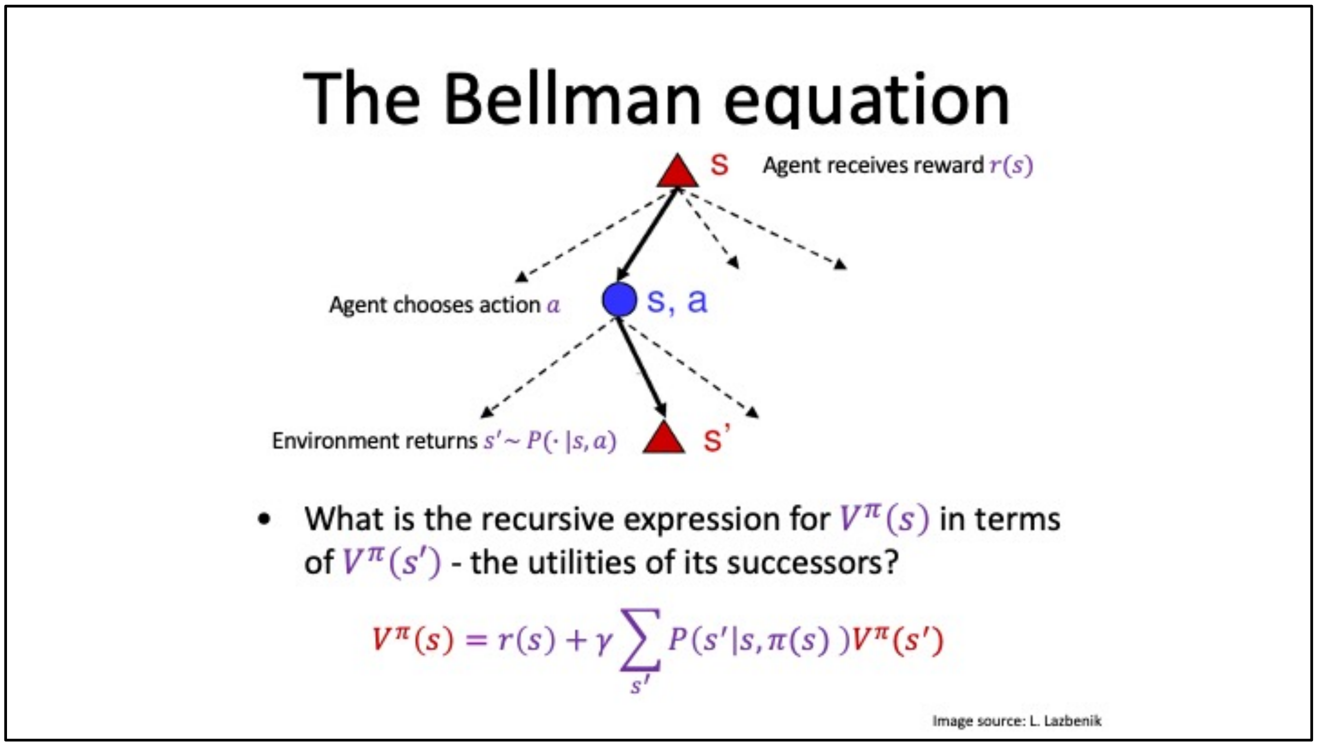
$$V^{\pi}(s_0) = \sum_{\substack{\text{sequences} \\ \text{starting from } s_0}} P(\text{sequence})U(\text{sequence})$$

Called the **value function** (for π, s_0)

The value function for a policy from an initial state is the expected reward/utility collected by following the policy starting from the initial state.

It's an expectation since we have randomness in the state transition. We also define the utility or reward collected for a sequence of states as the sum of the discounted rewards of the states in the sequence, so that we can get convergence.

To compute the value function, we introduce a key property: Bellman equation. It can be derived from the definition of the value function, by walking one step of the interaction.



The Bellman equation

s
Agent receives reward $r(s)$
Agent chooses action a
s, a
Environment returns $s' \sim P(\cdot\,|s,a)$
s’

- What is the recursive expression for $V^{\pi}(s)$ in terms of $V^{\pi}(s')$ - the utilities of its successors?

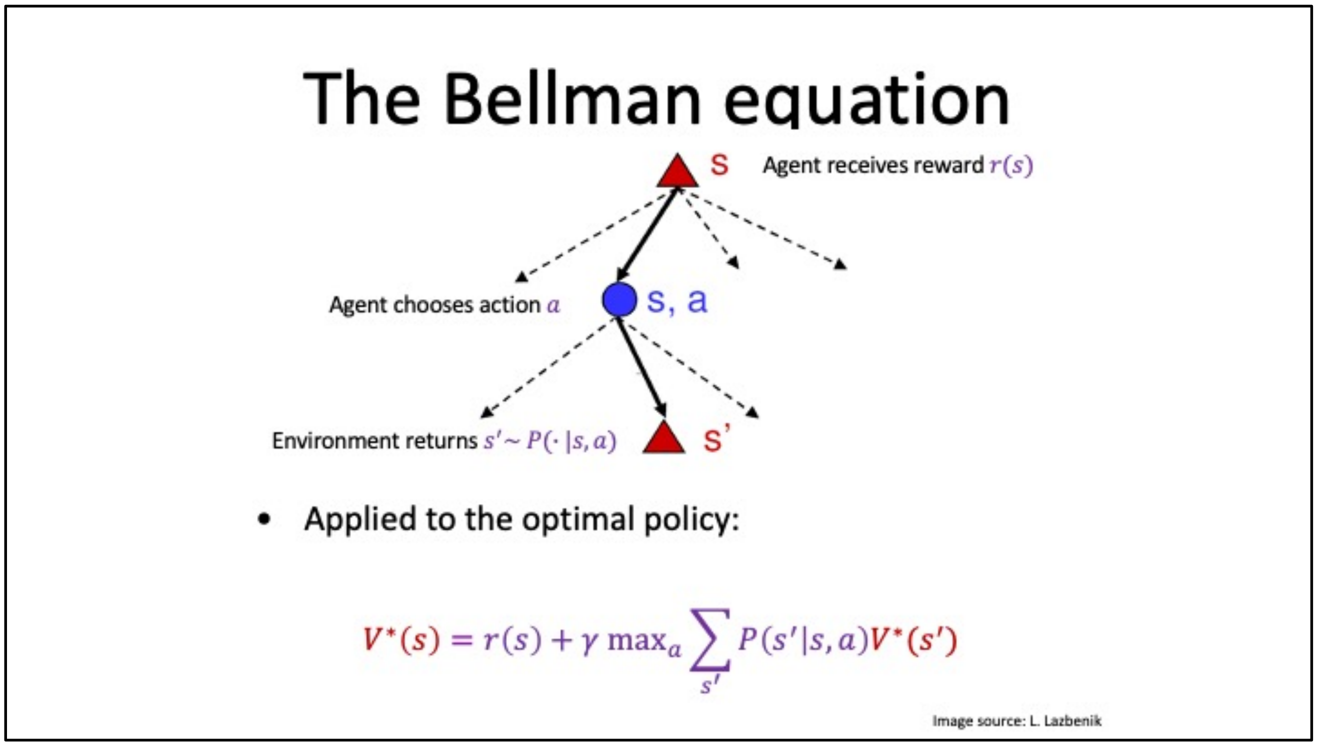
$$V^{\pi}(s) = r(s) + \gamma \sum_{s'} P(s'|s,\pi(s)\,)V^{\pi}(s')$$

Image source: L. Lazbenik

To explain better, we consider the tree description of the interaction.

Current state: s
Different actions lead to different children specified by (s, a) pairs.
The (s,a) pair then leads to a distribution over the next state s’ according to the state transition distribution.

Recall the definition of the value function of a policy \pi from a state s: it’s the expected utility accumulated starting from s and following the policy \pi. Break the accumulation of rewards into two parts: the first step and future steps.
1. In the first step we get the reward r(s) at the state s.
2. Suppose then the next state is s’. Then in the future steps, we will get utility accumulated starting from s’ and following the policy \pi. This is exactly the definition of the value of a policy \pi from a state s’! Considering the distribution of s’ and the discounted factor, the utility collected in future steps is the expectation of V^{\pi}(s’) over the distribution P(s’| s, \pi(s)).
In summary, we have
V^{\pi}(s) = r(s) + \gamma * E_{s’} V^{\pi}(s’) = r(s) + \gamma * \sum_{s’} P(s’|s, \pi(s)) V^{\pi}(s’)
This is the Bellman Equation for a general policy.



The Bellman equation

s
Agent receives reward $r(s)$
Agent chooses action a
s, a
Environment returns $s' \sim P(\cdot\,|s,a)$
s'

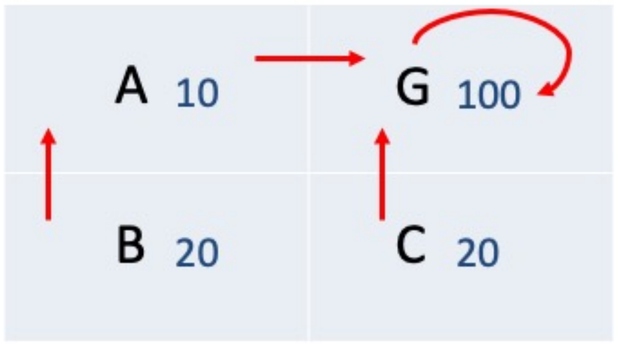
- Applied to the optimal policy:

$$V^*(s) = r(s) + \gamma \max_a \sum_{s'} P(s'|s,a)V^*(s')$$

Image source: L. Lazbenik

The Bellman's equation for the optimal policy is a special case.

Example



Deterministic transition. $\gamma = 0.8$, policy shown in red arrow.

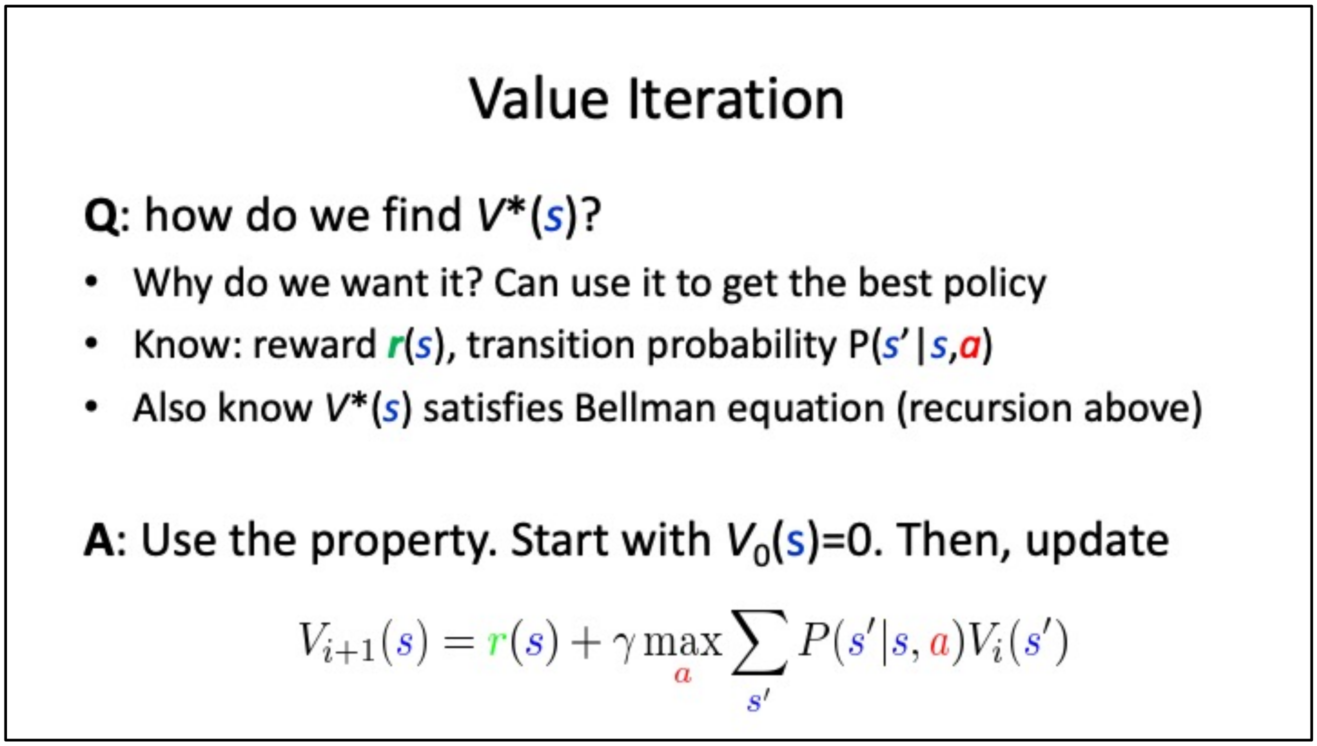
Apply Bellman equation on a given policy.

Consider G. After one step of the policy we get to the next state G deterministically.
Then by Bellman equation:
  Value of the policy on G = r(G) + \gamma * Value of the policy on G
Then
  Value of the policy on G = r(G) / (1-\gamma) = 100/(1-0.8) = 500

Consider A:
  value on A = r(A) + \gamma * value on G = 10 + 0.8 * 500 = 410

Consider B:
  value of B = r(B) + \gamma * value on A = 20 + 0.8 * 410 = 348

Consider C:
  value of C = r(C) + \gamma * value on G = 20 + 0.8 * 500 = 420



Value Iteration

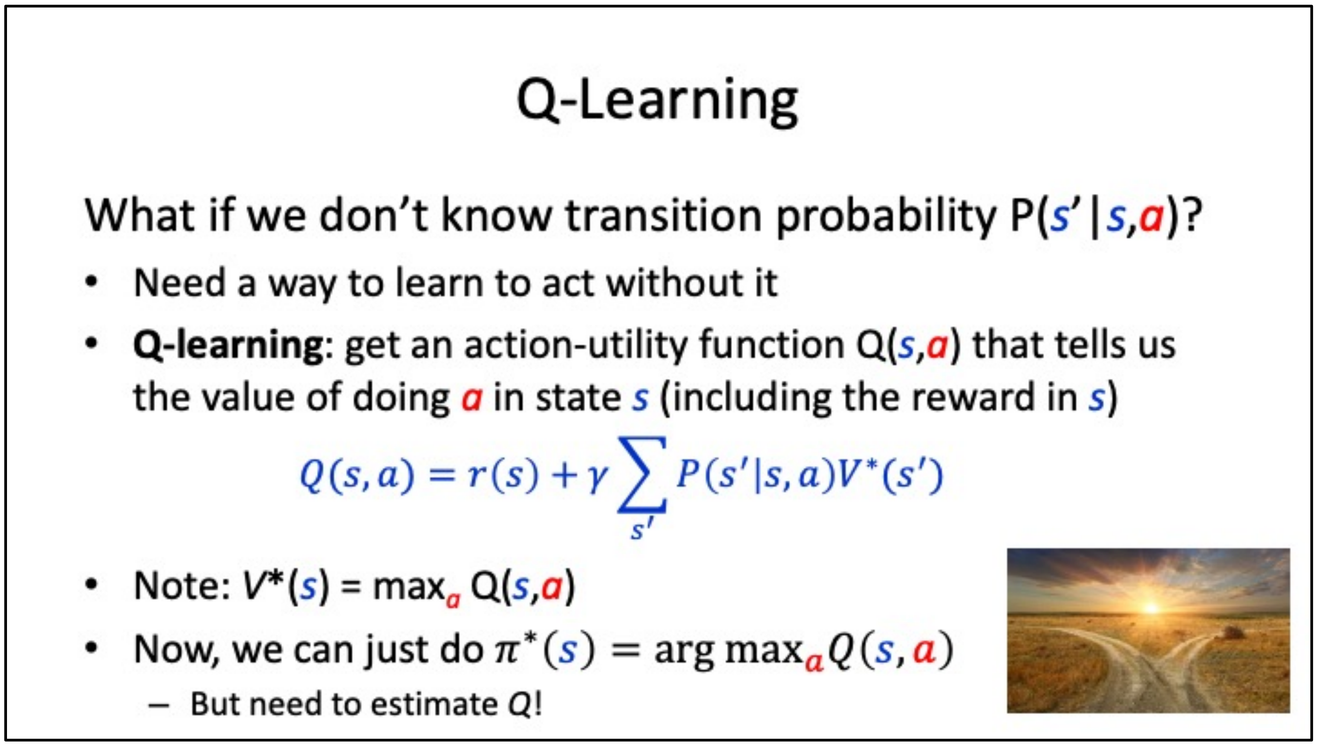
Q: how do we find $V^*(s)$?

- Why do we want it? Can use it to get the best policy
- Know: reward $r(s)$, transition probability $P(s'|s,a)$
- Also know $V^*(s)$ satisfies Bellman equation (recursion above)

A: Use the property. Start with $V_0(s)=0$. Then, update

$$V_{i+1}(s) = r(s) + \gamma \max_a \sum_{s'} P(s'|s,a) V_i(s')$$

Our original goal is to get V* without knowing the optimal policy, so cannot apply Bellman equation as we did in the example in the previous slide. But we can still use an iterative approach.



Q-Learning

What if we don't know transition probability $P(s'|s,a)$?

- Need a way to learn to act without it
- **Q-learning**: get an action-utility function $Q(s,a)$ that tells us the value of doing a in state s (including the reward in s)

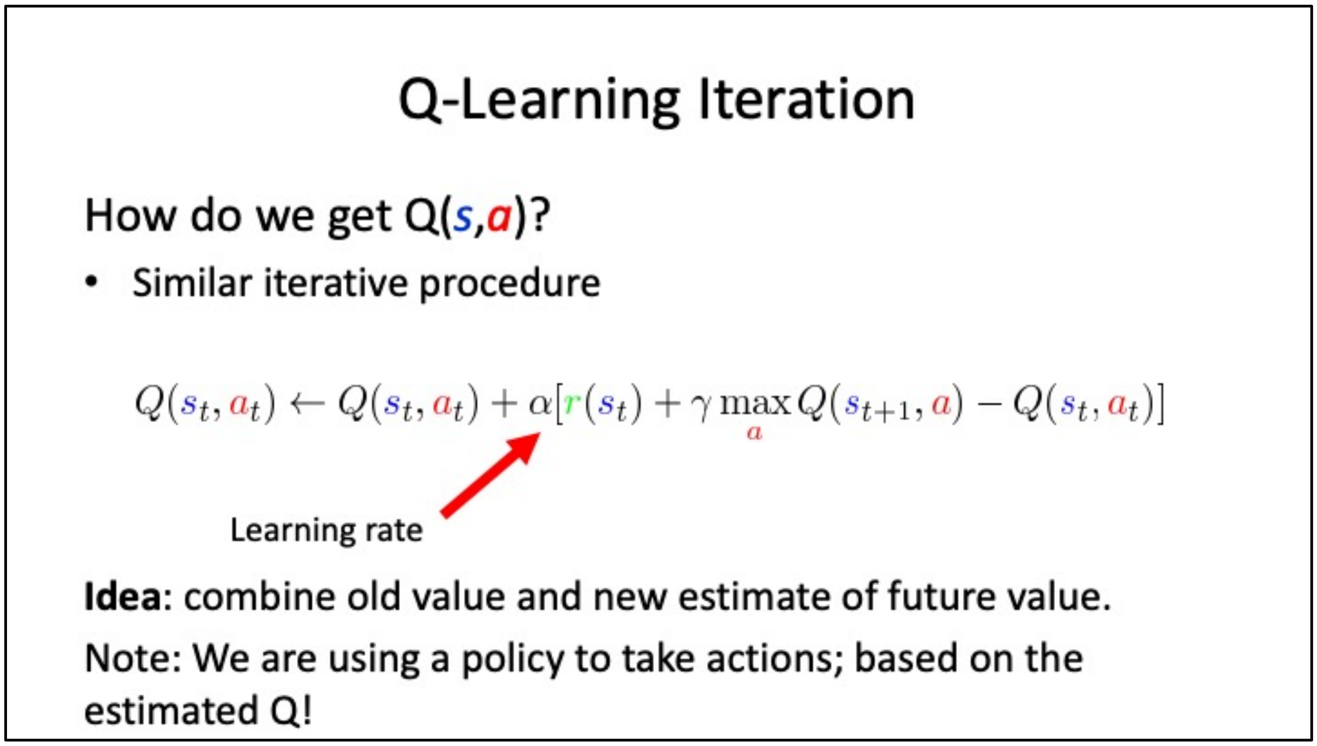
$$Q(s,a) = r(s) + \gamma \sum_{s'} P(s'|s,a) V^*(s')$$

- Note: $V^*(s) = \max_a Q(s,a)$
- Now, we can just do $\pi^*(s) = \arg\max_a Q(s,a)$
  - But need to estimate Q!

Note that definition of Q slightly different from the the expected utility of an action we talked about in the previous slide: Q includes the reward in s.

From Bellman equation we can introduce Q function. Then V* and \pi* has a simple form.

Definition of value function -> Bellman equation -> value iteration; and also Q function and Q-learning



Q-Learning Iteration

How do we get Q(*s*,*a*)?

- Similar iterative procedure

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r(s_t) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

Learning rate

Idea: combine old value and new estimate of future value.

Note: We are using a policy to take actions; based on the estimated Q!

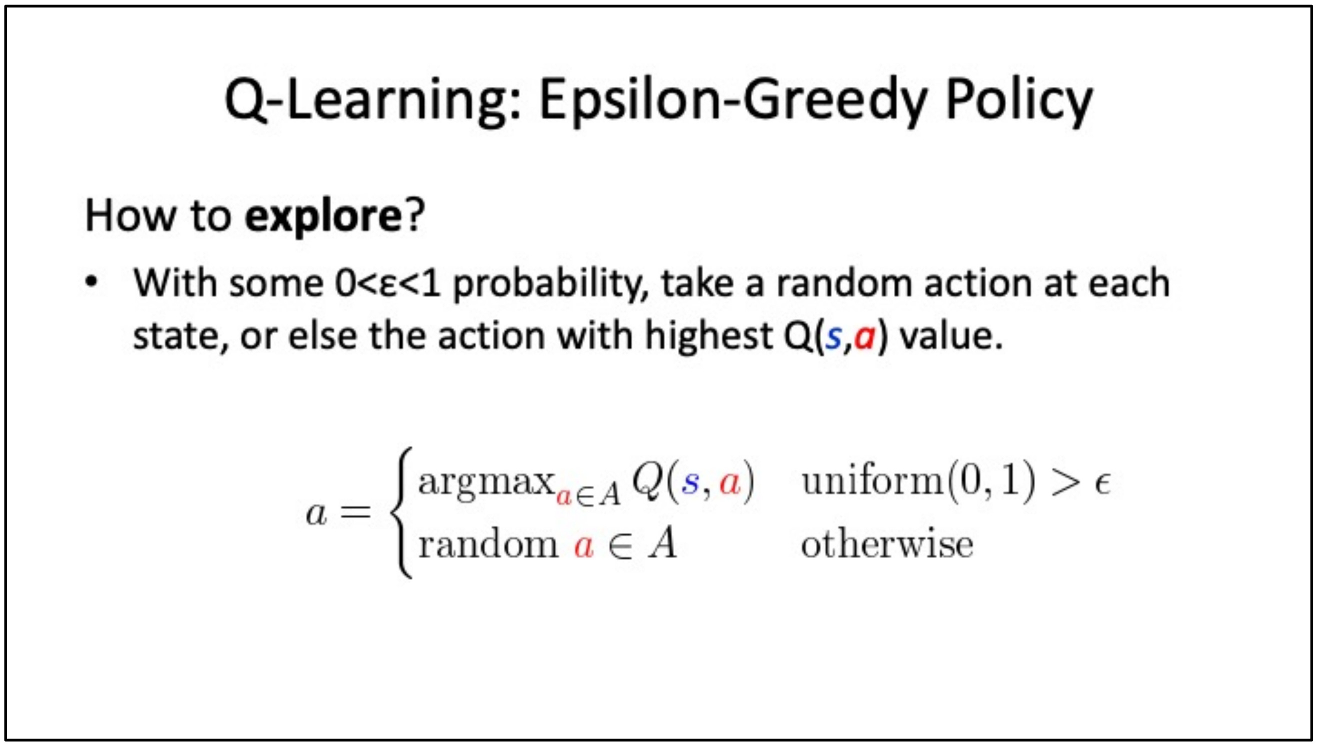
Suppose during training, we observe that: at time t, the agent is in the state s_t and gets reward r(s_t), and then takes an action a_t and goes to the next state s_{t+1}. Then we can use these observations to update the Q function.

The update rule RHS can be rewritten as:
  (1-\alpha) Q(s_t, a_t) + \alpha [r(s_t) + \gamma \max_a Q(s_{t+1}, a)]
It's a weighted sum of two terms: the old value Q(s_t, a_t) and a new estimate r(s_t) + \gamma \max_a Q(s_{t+1}, a).

Why is r(s_t) + \gamma \max_a Q(s_{t+1}, a)   a good estimate of Q(s_t, a_t)?
By definition:
  Q(s_t, a_t) = r(s_t) + \gamma E_{s'} V*(s').
In training we don't know the distribution of s'. We only have one sample s_{t+1}, so we use this sample to estimate the expectation E_{s'} V*(s'):
  new estimate of Q(s_t, a_t) = r(s_t) + \gamma V*(s_{t+1}).
But we also don't know V*(s_{t+1}). We can use the current estimation of Q function to estimate V*(s_{t+1}) = \max_{a} Q(s_{t+1},a). So we have
  new estimate of Q(s_t, a_t) = r(s_t) + \gamma = \max_{a} Q(s_{t+1},a).
This then leads to the new estimate in the update rule.

When alpha = 1, we use the new estimate to completely replace the old value. This is similar to value iteration. But we use a learning rate 0< alpha <1 to balance the old and new, so that the learning is more stable.

One thing still needs to be specified: How to choose the action a_t.



Q-Learning: Epsilon-Greedy Policy

How to **explore**?

- With some 0<ε<1 probability, take a random action at each state, or else the action with highest Q(*s*,*a*) value.

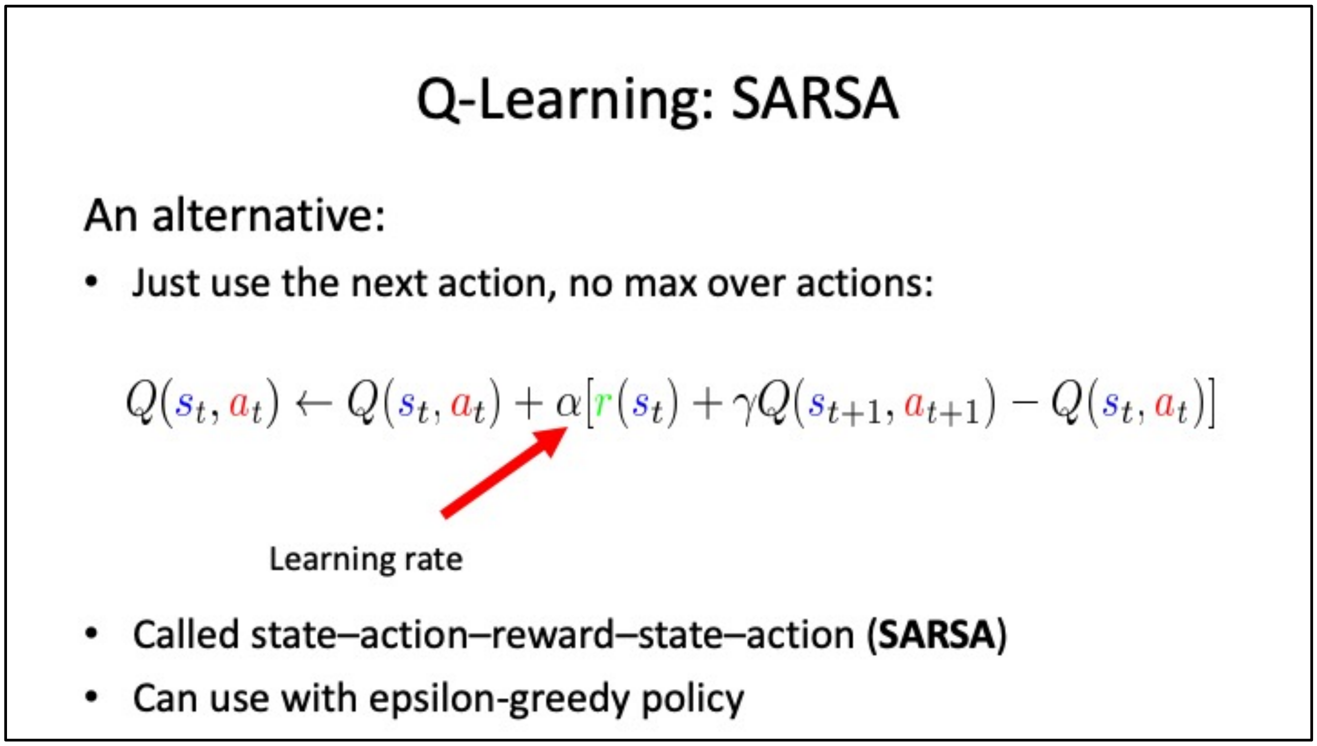
$$a = \begin{cases} \operatorname{argmax}_{a \in A} Q(s,a) & \text{uniform}(0,1) > \epsilon \\ \text{random } a \in A & \text{otherwise} \end{cases}$$

A simple but effective method to choose the action to tradeoff exploration and exploitation: with a small probability eps (eps is a parameter), the agent choose a random action; otherwise choose the best action according to the current estimate of Q.

In summary, in Q-learning

1. First use some method (like epsilon-greedy) to choose an action, get the observation s_t, r(s_t), a_t, s_{t+1}.
2. Use the observations in the update rule (like the one on the previous slide) to update the Q value for (s_t, a_t)
3. Repeat

We can have other action-choosing methods other than epsilon-greedy. We can also have other update rules.



Q-Learning: SARSA

An alternative:

- Just use the next action, no max over actions:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r(s_t) + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

Learning rate

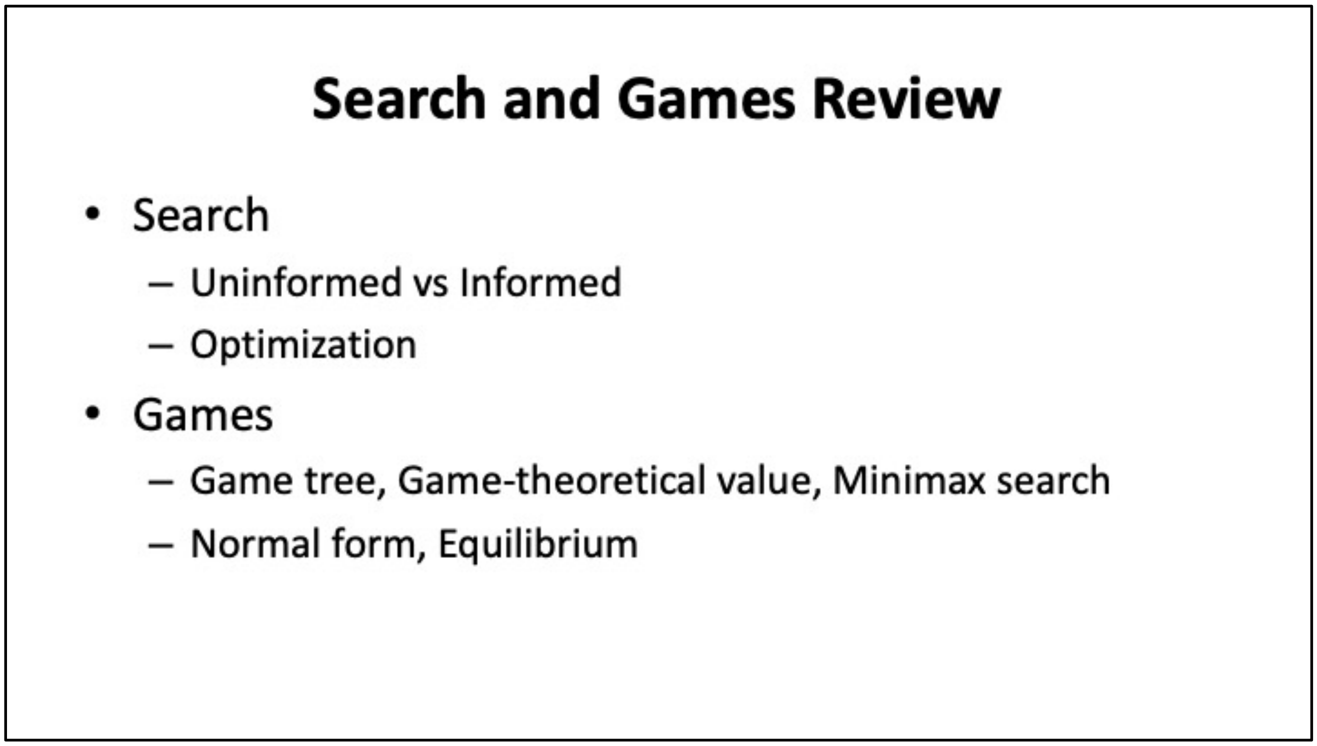
- Called state–action–reward–state–action (**SARSA**)
- Can use with epsilon-greedy policy

Another popular update rule is SARSA, which replace the max over actions with simply the next action. This is more efficient especially when there are many actions.

The name comes from the observations used for the update: state s_t, action a_t, reward r(s_t), state s_{t+1}, action a_{t+1}.

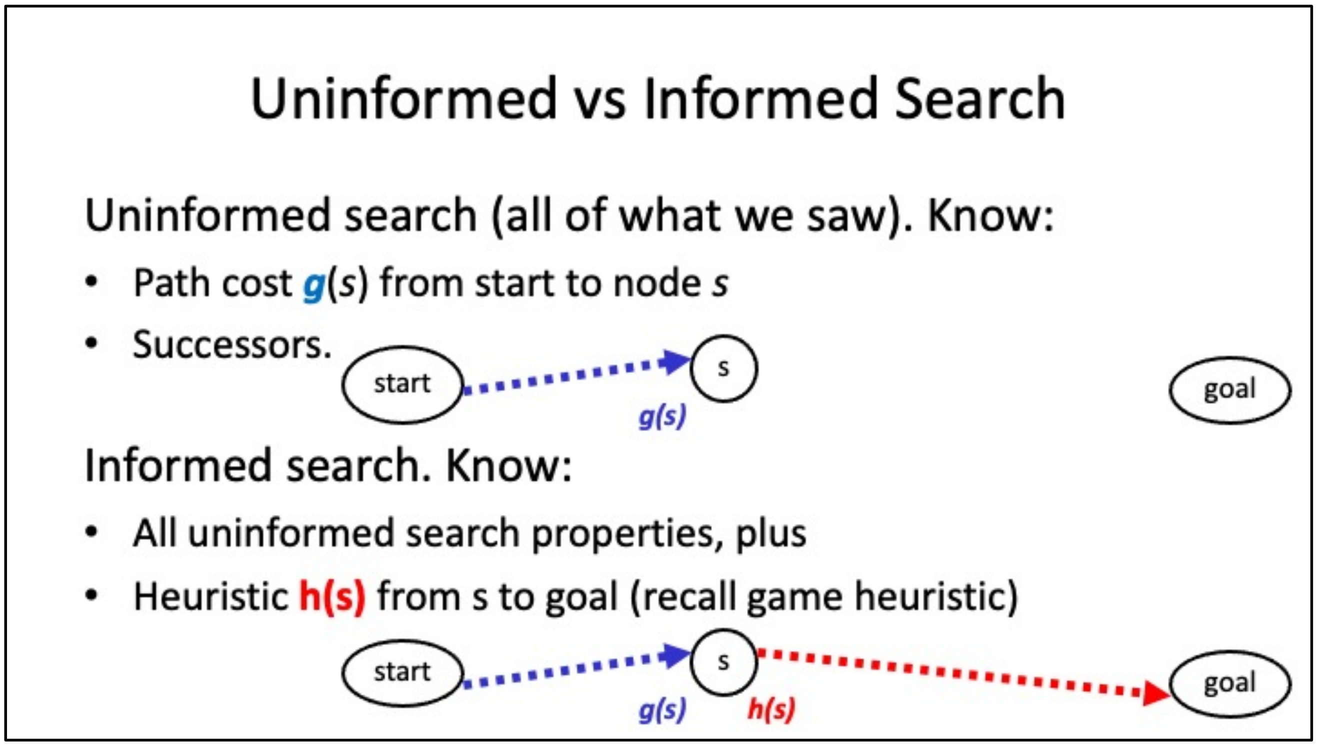
Summary of RL

- Reinforcement learning setup
- Mathematical formulation: MDP
- **Value functions & the Bellman equation**
- Value iteration
- Q-learning



Search and Games Review

- Search
  - Uninformed vs Informed
  - Optimization
- Games
  - Game tree, Game-theoretical value, Minimax search
  - Normal form, Equilibrium



Uninformed vs Informed Search

Uninformed search (all of what we saw). Know:

- Path cost ***g***(*s*) from start to node *s*
- Successors.

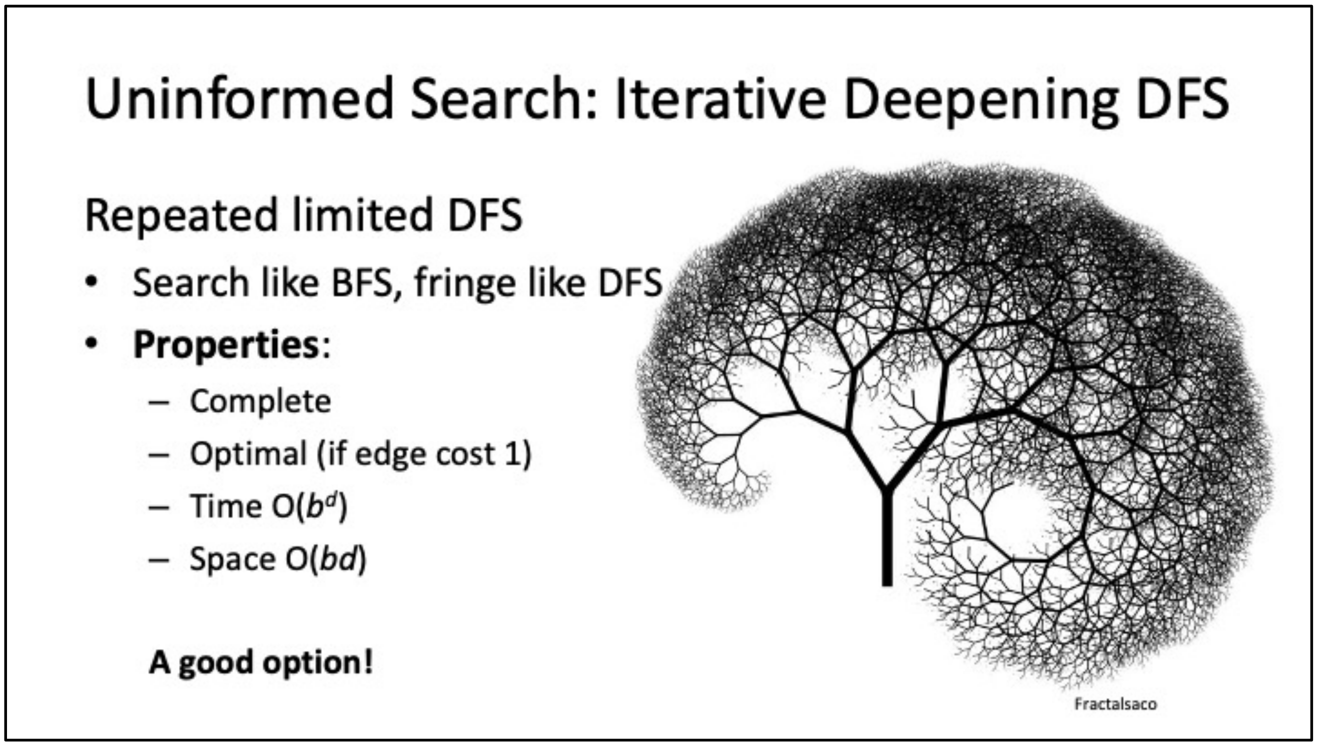
start
s
goal
g(s)

Informed search. Know:

- All uninformed search properties, plus
- Heuristic **h(s)** from s to goal (recall game heuristic)

start
s
goal
g(s)
h(s)

Key difference: informed search knows an additional function h(s), which can be regarded as an estimation of the cost from a state to the goal (or one of the goal states).

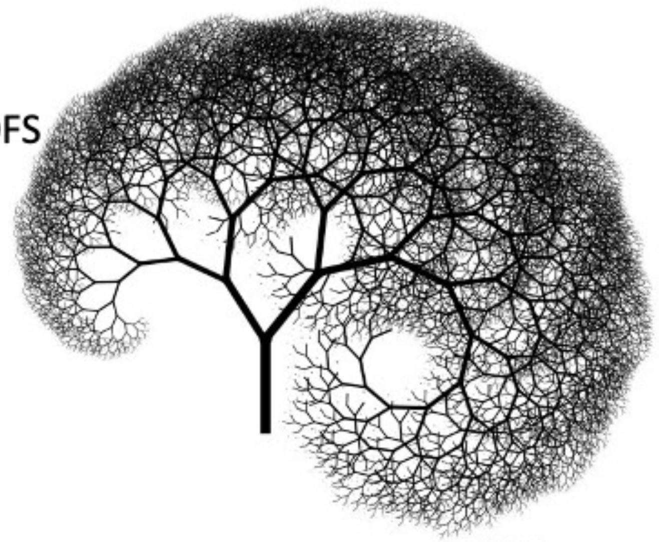


Uninformed Search: Iterative Deepening DFS

Repeated limited DFS

- Search like BFS, fringe like DFS
- **Properties**:
  - Complete
  - Optimal (if edge cost 1)
  - Time O(b^d)
  - Space O(bd)

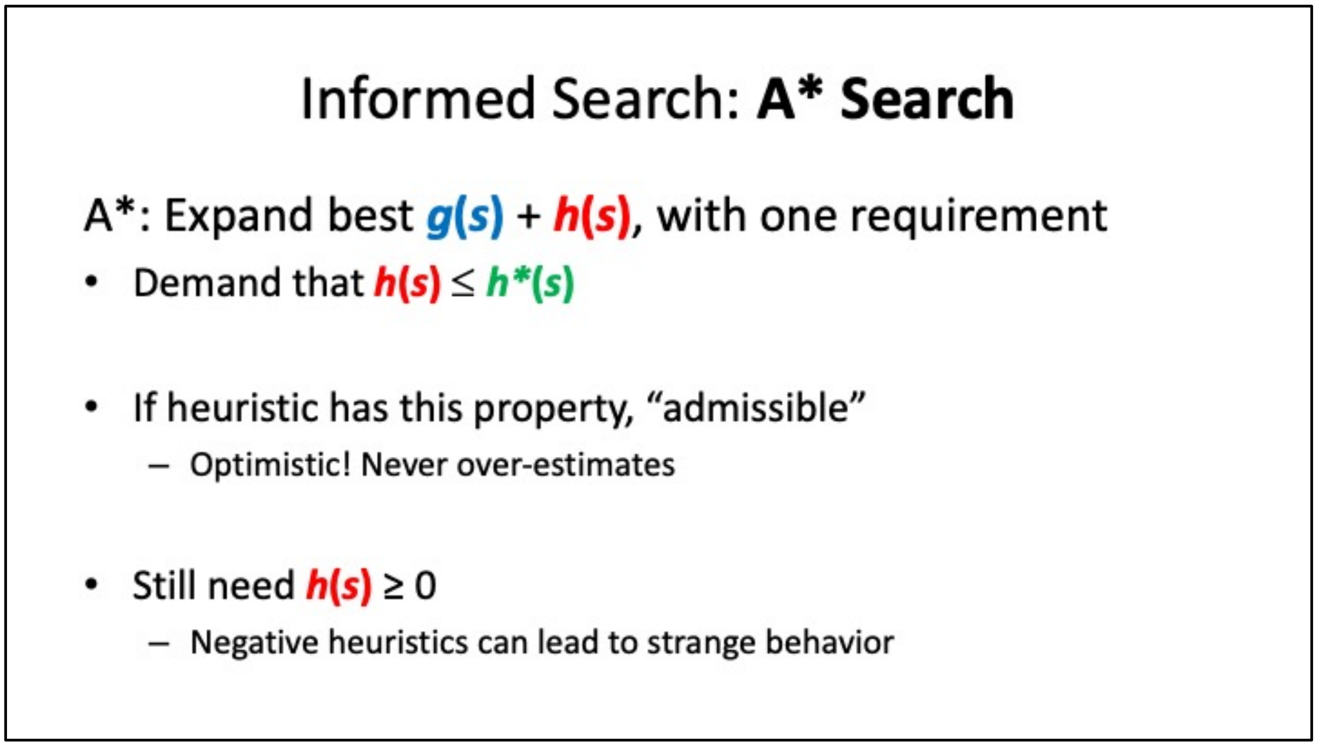
A good option!



All uninformed/informed search methods fall into the same general framework:

1. At the beginning put the initial state into the fringe
2. Then run in iterations:
   1) First pick a node from the fringe
   2) Check if goal; if so return
   3) Put the successors into the fringe

The preferred method is iterative deepening, which has good performance metrics.



Informed Search: **A* Search**

A*: Expand best $g(s)$ + $h(s)$, with one requirement

- Demand that $h(s) \leq h^*(s)$
- If heuristic has this property, "admissible"
  - Optimistic! Never over-estimates
- Still need $h(s) \geq 0$
  - Negative heuristics can lead to strange behavior

A* search = A search with an admissible heuristic.

Admissible: 0 <= h(s) <= true cost h*(s) for any state s.

Q1: You need to search a randomly generated state space graph with one goal, uniform edges costs. The goal is d=2 steps from the initial state, and the maximum depth is m=100. Considering worst case behavior, do you select BFS or DFS for your search?

1. BFS

2. DFS

Q1: You need to search a randomly generated state space graph with one goal, uniform edges costs. The goal is d=2 steps from the initial state, and the maximum depth is m=100. Considering worst case behavior, do you select BFS or DFS for your search?

[✓ 1. BFS]

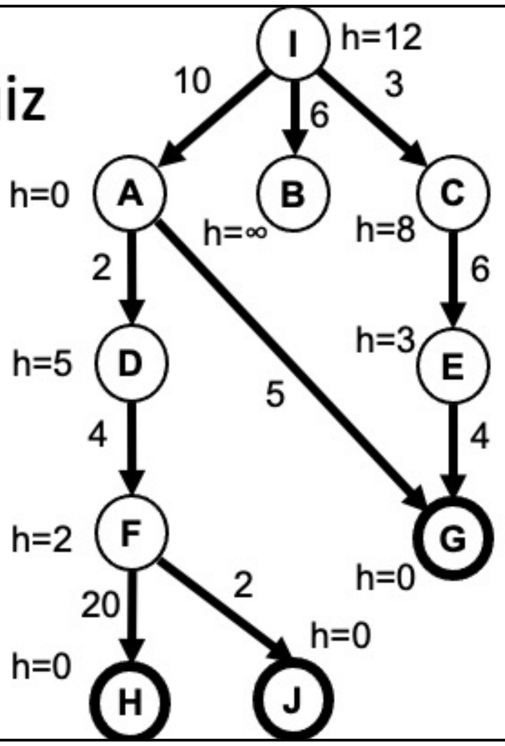
2. DFS

The goal is 2 steps from the initial state, which is good for BFS.

Break & Quiz

Q2: Consider the state space graph below. Goal states have bold borders. ***h*(*s*)** is show next to each node. What node will be expanded by A* after the initial state I?

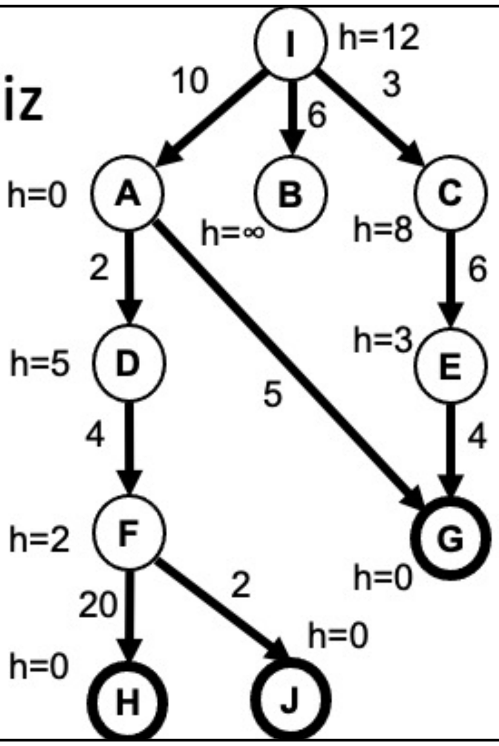
- A
- B
- C



Break & Quiz

Q2: Consider the state space graph below. Goal states have bold borders. ***h*(*s*)** is show next to each node. What node will be expanded by A* after the initial state I?

- A
- B
- C

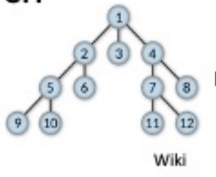


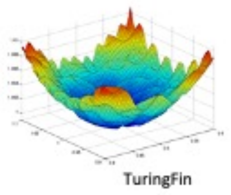
After expanding the initial state I, the fringe has A B C.
A's g + h value: 10+0
B's: 6 + \infinity
C's: 3 + 8
A* search will pick A.

Search vs. Optimization

Before: wanted a path from start state to goal state

- Uninformed search, informed search

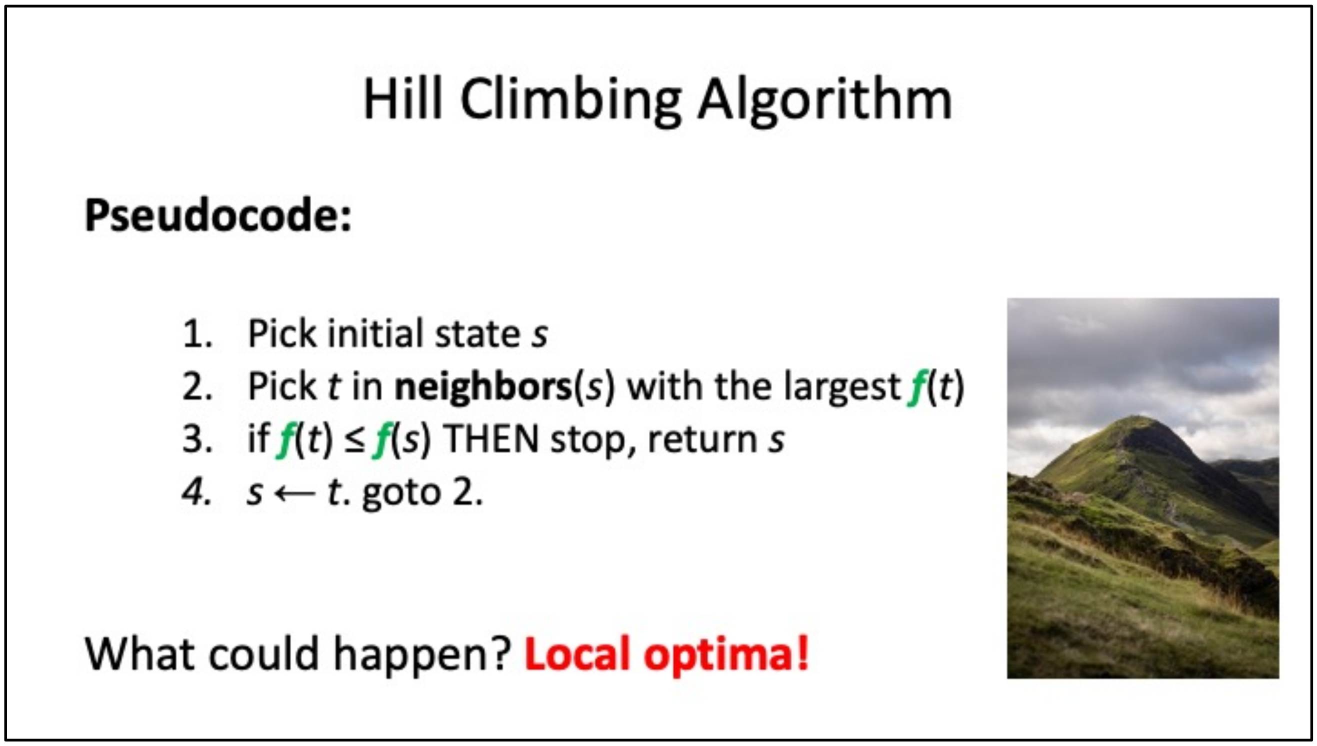




New setting: optimization

- States *s* have values ***f***(*s*)
- Want: *s* with optimal value ***f***(*s*) (i.e, **optimize** over states)
- Challenging setting: **too many states** for previous search approaches, but maybe not a continuous function for SGD.

Optimization-like search: want a state with the optimal value; don't care about solution path.



Hill Climbing Algorithm

Pseudocode:

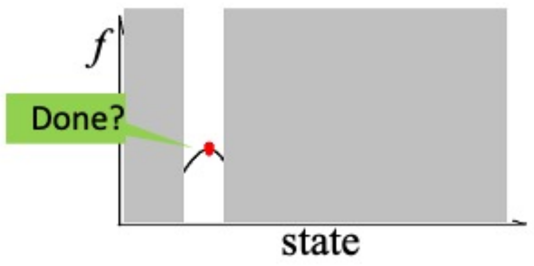
```
1. Pick initial state $s$
2. Pick $t$ in neighbors($s$) with the largest $f(t)$
3. if $f(t) \le f(s)$ THEN stop, return $s$
4. $s \leftarrow t$. goto 2.
```

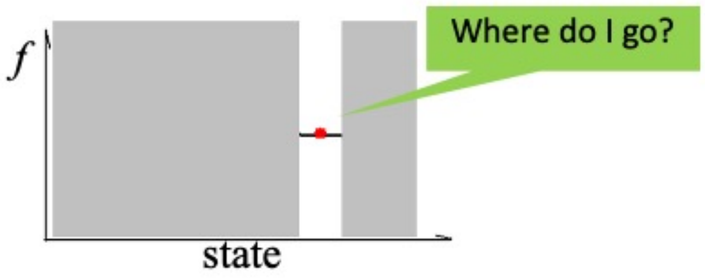
What could happen? **Local optima!**

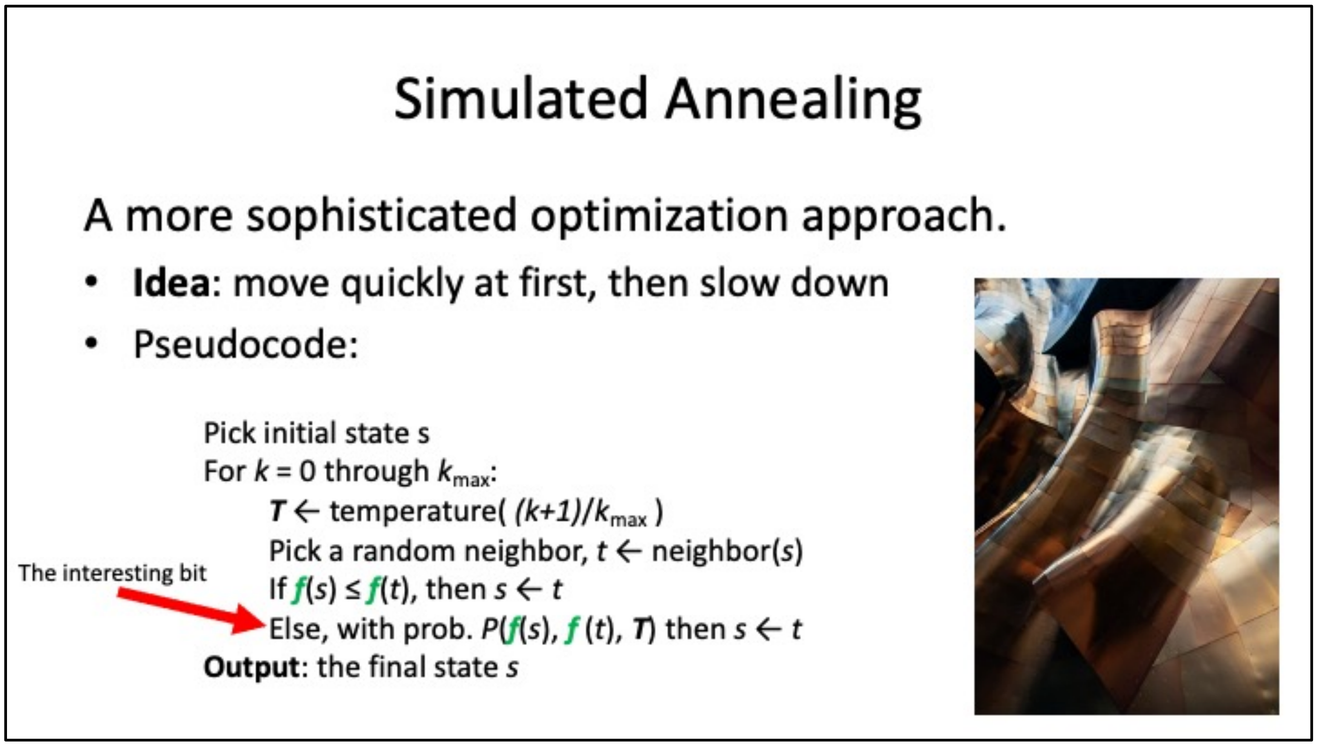
Key idea: keep greedily picking the best neighbor.

Hill Climbing: Local Optima

Note the **local optima**. How do we handle them?



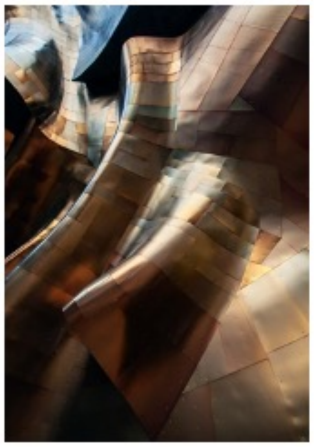




Simulated Annealing

A more sophisticated optimization approach.

- **Idea**: move quickly at first, then slow down
- Pseudocode:



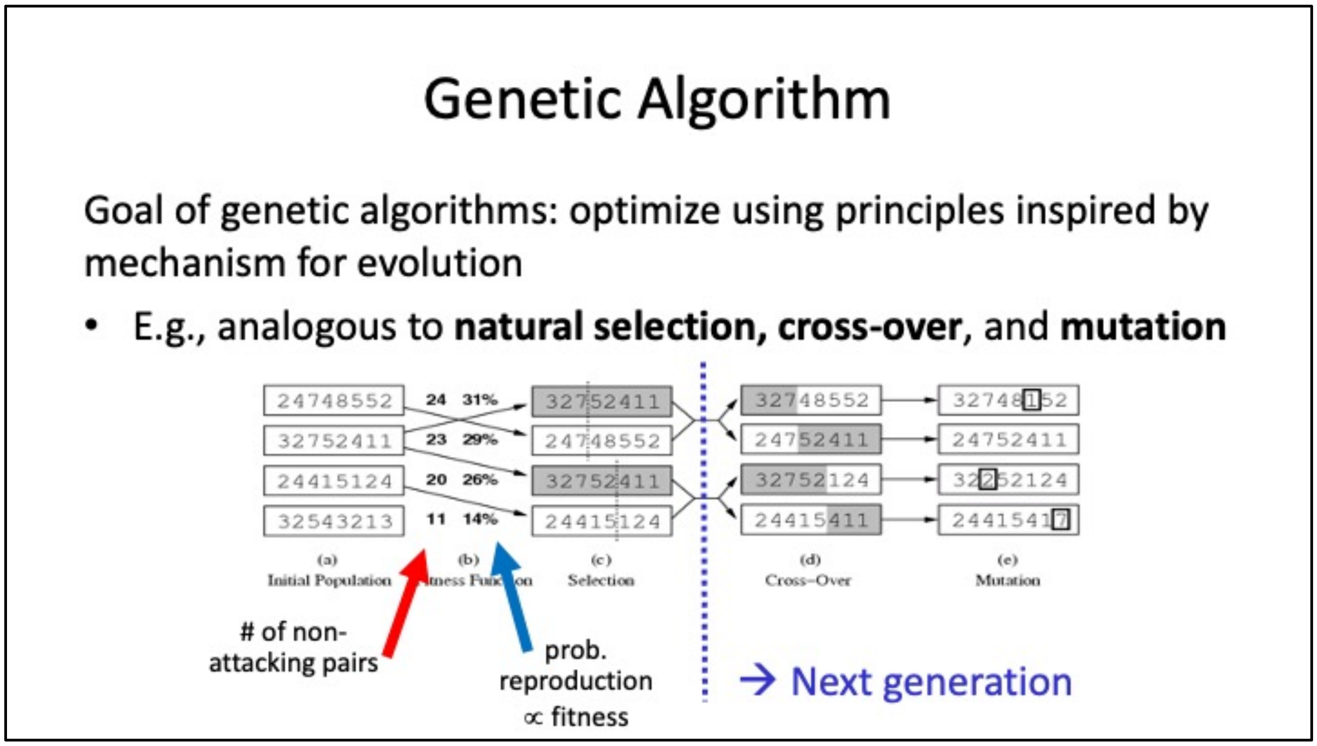
```
Pick initial state s
For $k$ = 0 through $k_{max}$:
    $T \leftarrow$ temperature( $(k+1)/k_{max}$ )
    Pick a random neighbor, $t \leftarrow$ neighbor($s$)
    If $f(s) \le f(t)$, then $s \leftarrow t$
    Else, with prob. $P(f(s), f(t), T)$ then $s \leftarrow t$
Output: the final state $s$
```

The interesting bit

Can be viewed as a variant of Hill-Climbing:

1. Randomly pick one neighbor instead of checking all neighbors
2. With some probability allow to go to a neighbor with worse value

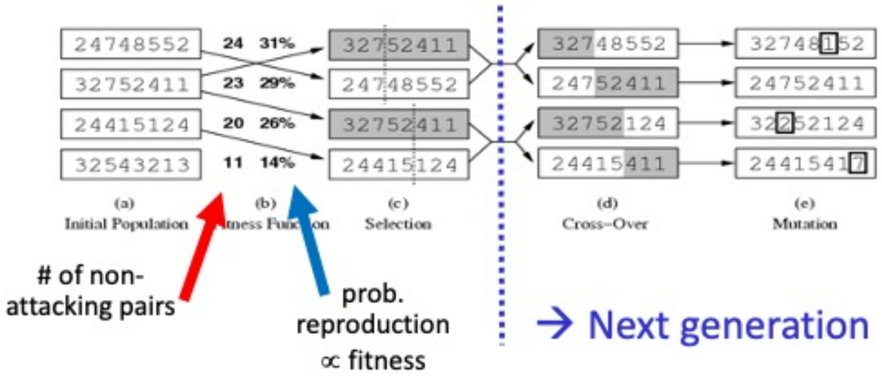
Key idea: in the early iterations, accept worse neighbors with larger probability for exploration; in later iterations, accept with smaller probability for exploitation; use a decreasing temperature parameter to implement this.
The probability should also be smaller for larger gaps between the values for the neighbor and the current state.



Genetic Algorithm

Goal of genetic algorithms: optimize using principles inspired by mechanism for evolution

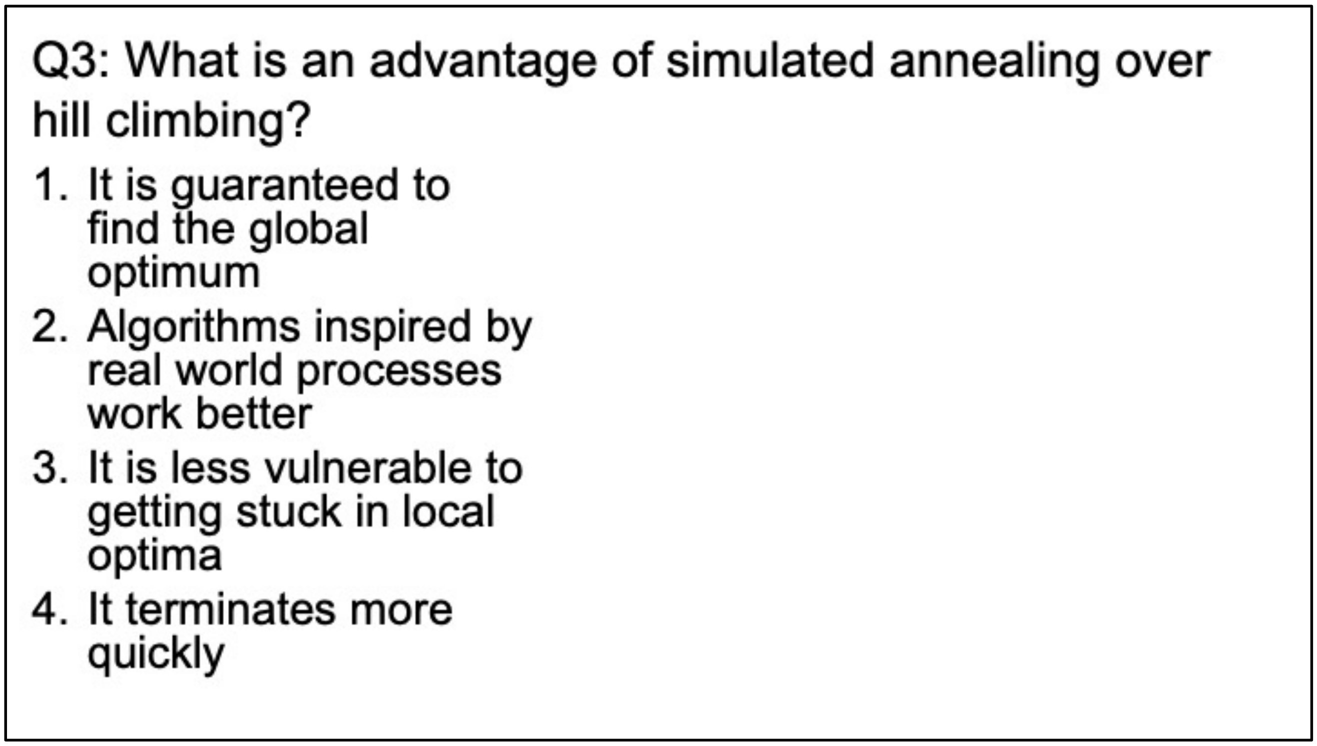
- E.g., analogous to **natural selection, cross-over**, and **mutation**



The genetic algorithm runs in generations, each generation has the following operations corresponding to natural selection, crossover and mutation.

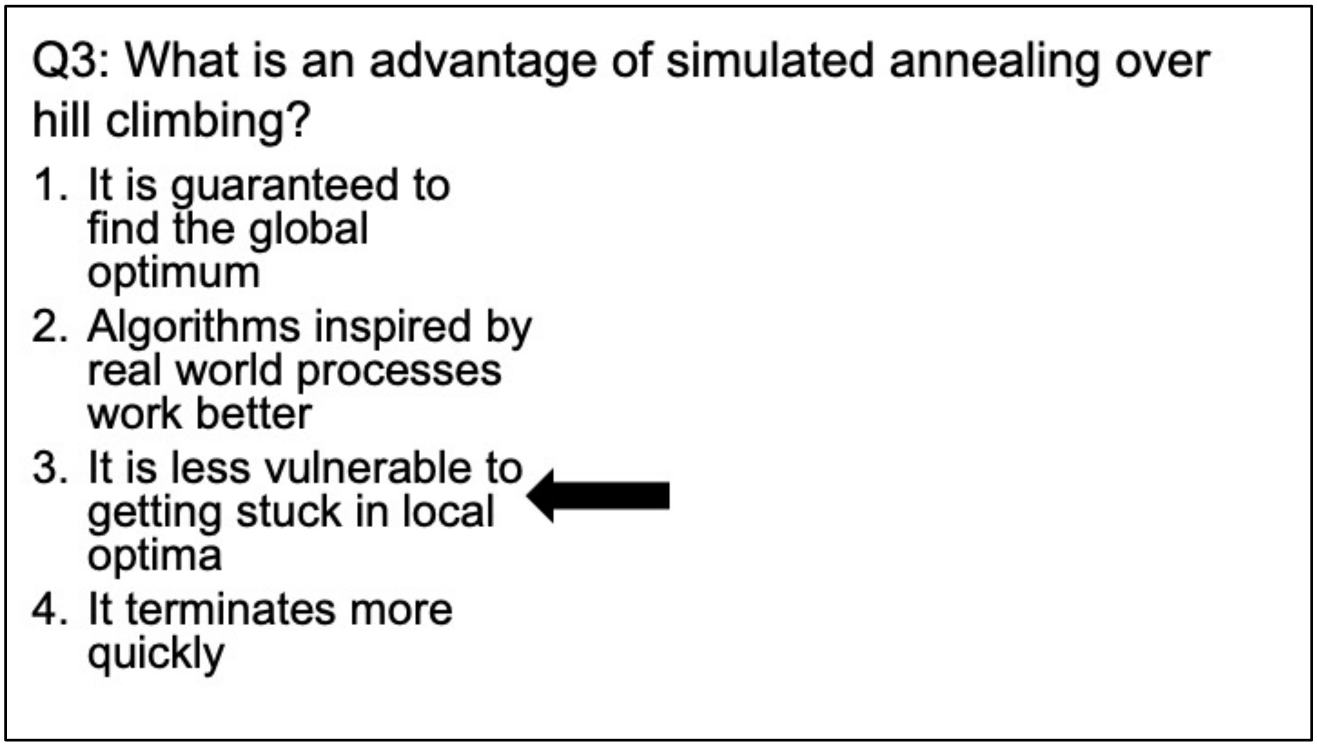
1. Compute the f value (the fitness), and then normalize them to get the probability of reproduction. Then do natural selection: by sampling from the reproduction probability distribution.
2. Then pair up the individuals; for each pair, pick a random location to cut each code into two segments; then do crossover by mixing up the segments.
3. Finally do mutation: (one standard variant) for each location in each individual, determine whether to do mutation with a small mutation probability; if yes, then replace the original symbol with a randomly pick symbol.

Repeat this until we are satisfied with the solution or run out of time budget.



Q3: What is an advantage of simulated annealing over hill climbing?

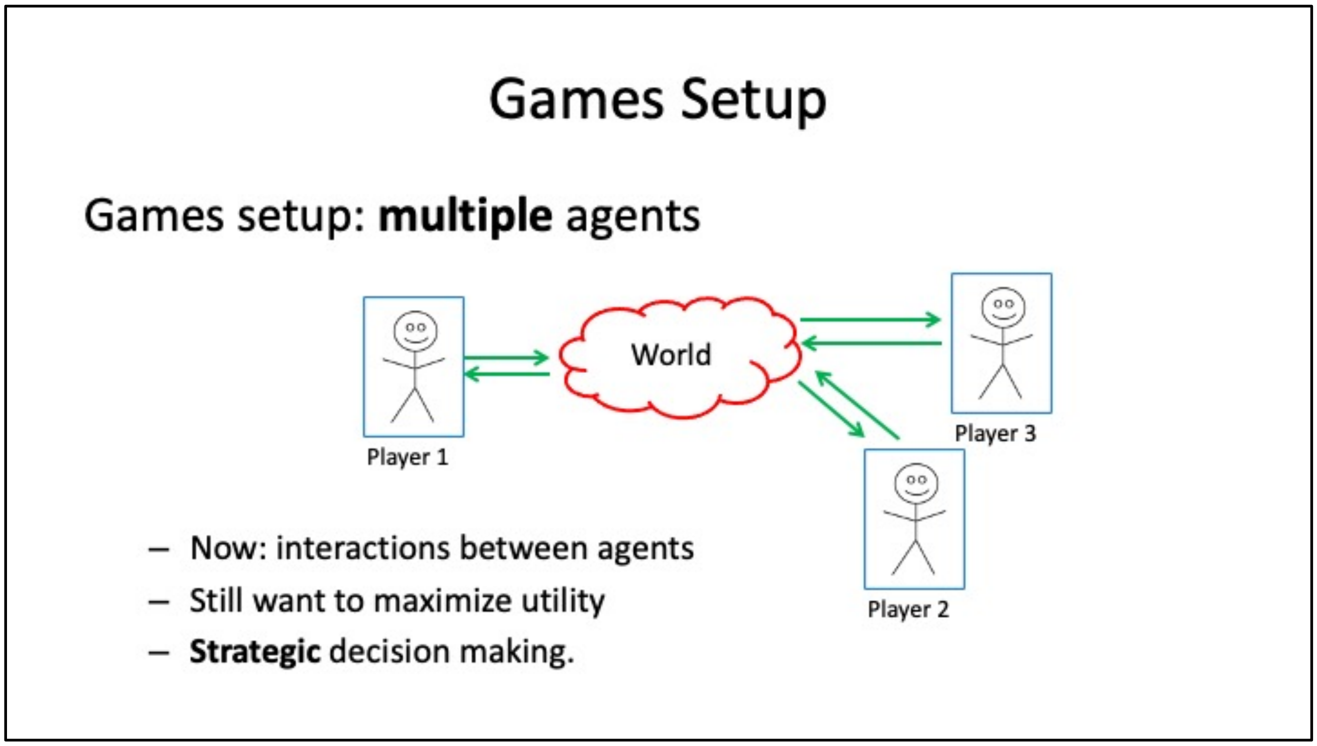
1. It is guaranteed to find the global optimum
2. Algorithms inspired by real world processes work better
3. It is less vulnerable to getting stuck in local optima
4. It terminates more quickly



Q3: What is an advantage of simulated annealing over hill climbing?

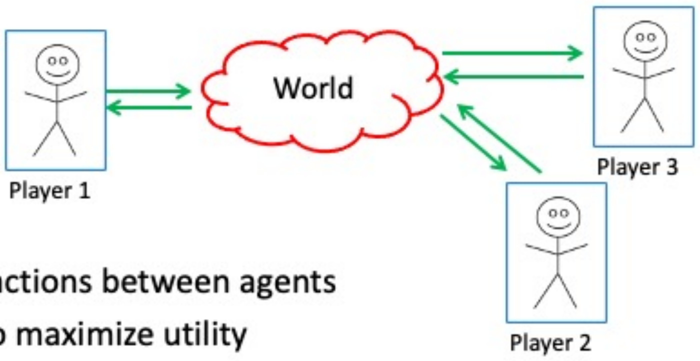
1. It is guaranteed to find the global optimum
2. Algorithms inspired by real world processes work better
3. [✓ It is less vulnerable to getting stuck in local optima]
4. It terminates more quickly

Simulated annealing may not find the global optimum and it may take longer time to terminate. But it allows some probability to accept a worse neighbor so can have better chance to escape local optima.



Games Setup

Games setup: **multiple** agents



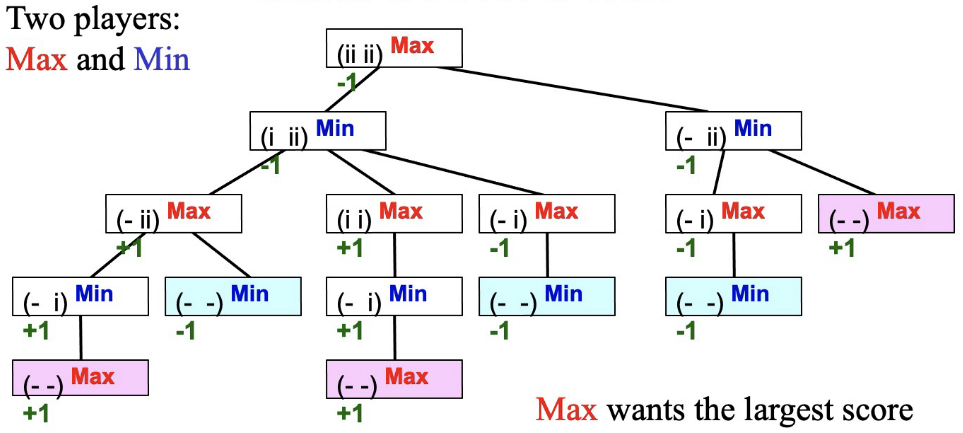
- Now: interactions between agents
- Still want to maximize utility
- **Strategic** decision making.

We can model the interactions between multiple agents (and potentially with the world). We assume that each player/agent has its own reward function and wants to maximize that. However, the reward function of any player depends on all the actions of all the players and the world. So the reward function captures the interaction.

What's the different between the world and a player? Players are rational or selfish or strategic. They want to maximize their own rewards. The world is not strategic: though its state can affect the rewards but it doesn't have or want to maximize its own reward.

Game tree for II-Nim

Two players:
Max and Min



Max wants the largest score
Min wants the smallest score

Use a game tree to formalize the sequential game.

Two other important implications:

1. We can compute the game-theoretical values easily from bottom up.
2. Once we have the values of the children of a current state, then we know which is the best

Minimax Algorithm

```
function Max-Value(s)
inputs:
    s: current state in game, Max about to play
output: best-score (for Max) available from s
    if ( s is a terminal state )
    then return ( terminal value of s )
    else
        α := – infinity
        for each s’ in Succ(s)
            α := max( α , Min-value(s’))
    return α
```

```
function Min-Value(s)
output: best-score (for Min) available from s
    if ( s is a terminal state )
    then return ( terminal value of s)
    else
        β := infinity
        for each s’ in Succs(s)
            β := min( β , Max-value(s’))
    return β
```

Time complexity?
- $O(b^m)$

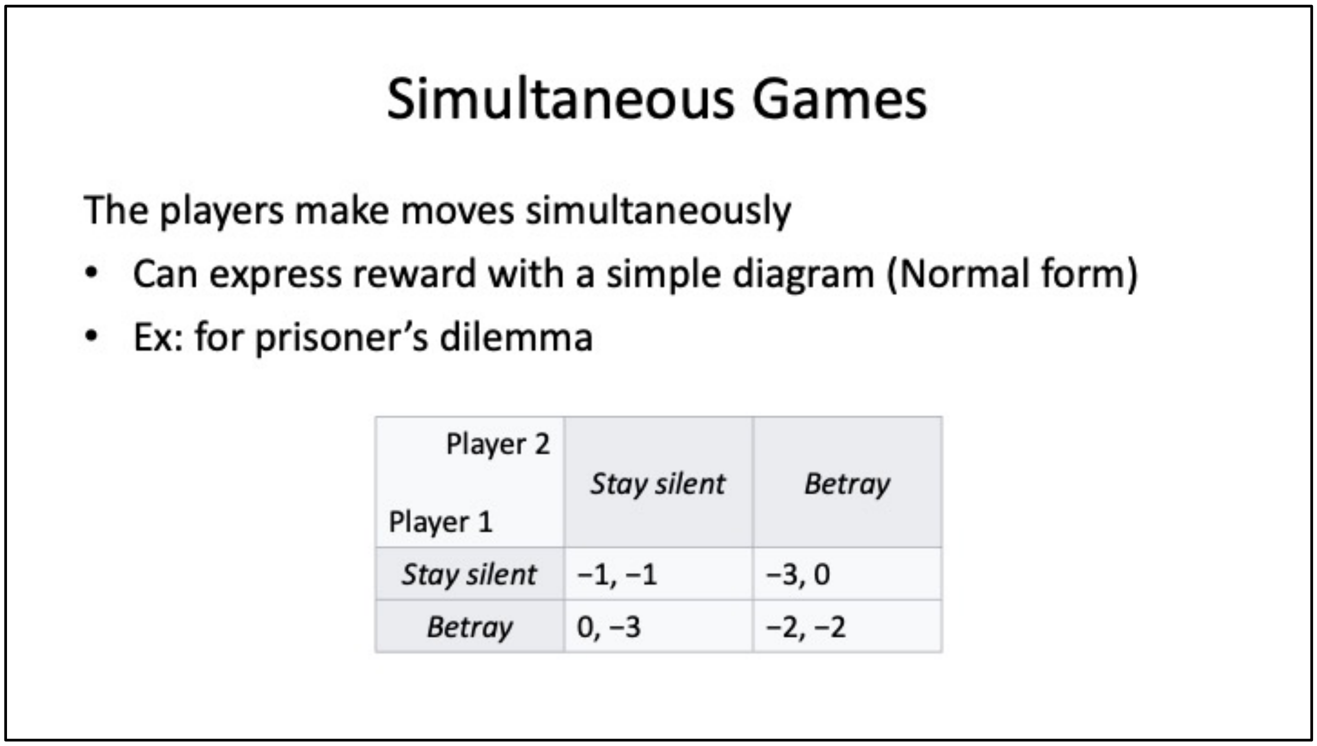
Space complexity?
- $O(bm)$

The minimax algorithm replaces the bottom-up computation with recursion.

The key idea of recursion: we assume that smaller problems are already solved, and we want to use the solutions for the smaller problems to solve the current problem.
Here, the smaller problems are the values of the children, and the current problem is the value of the current state.

On a state where Max is going to play:
1. it’s terminal then we can return the terminal score which is the value by definition
2. If not terminal, just take the maximum of the values of the children (here we pretend that we have already solve the smaller problems of computing the values of the children)

This is the Max-Value function. Similar for the Min-Value function that computes the value of a state where Min is going to play.



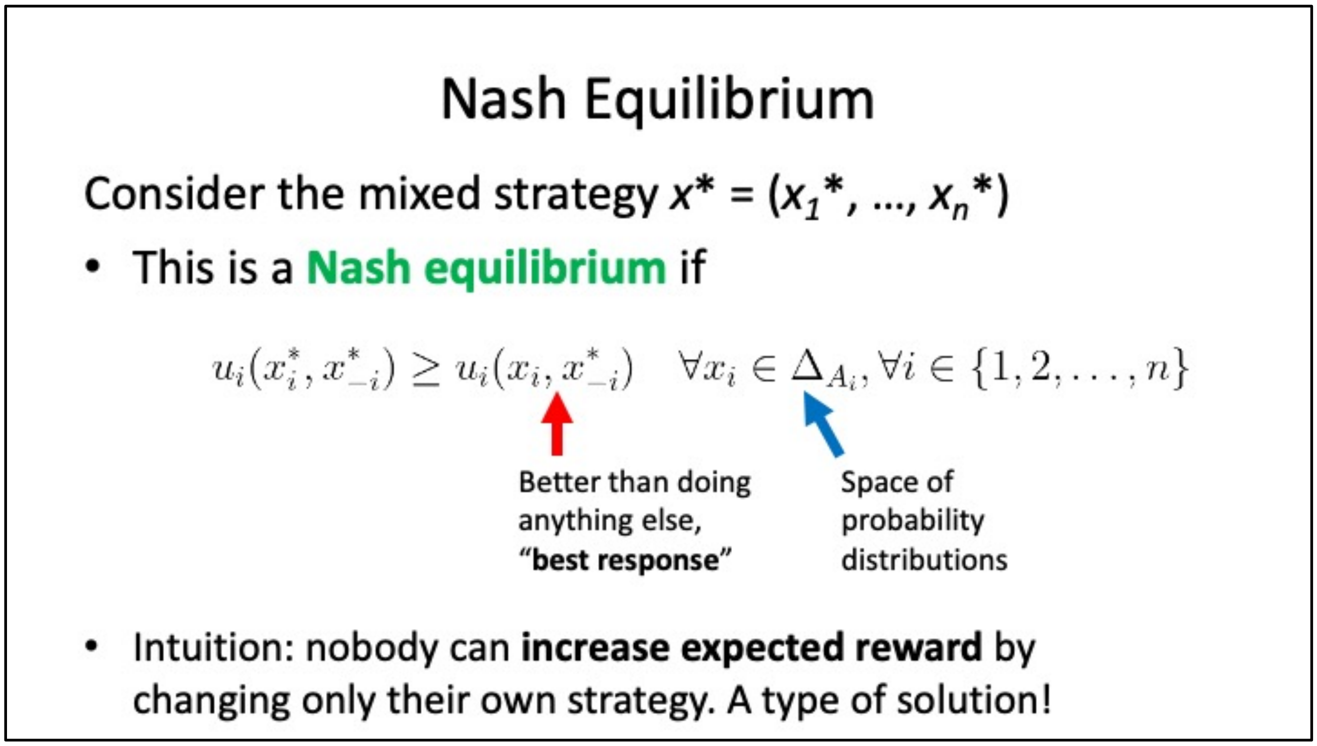
Simultaneous Games

The players make moves simultaneously

- Can express reward with a simple diagram (Normal form)
- Ex: for prisoner's dilemma

Player 2 Player 1	*Stay silent*	*Betray*
Stay silent	−1, −1	−3, 0
Betray	0, −3	−2, −2

Simultaneous games (one round) can be formalized using normal form (math definition of the reward/utility functions of the players). Can show in a diagram for illustration.



Nash Equilibrium

Consider the mixed strategy *x** = (*x*$_1$*, …, *x*$_n$*)

- This is a **Nash equilibrium** if

$$u_i(x_i^*, x_{-i}^*) \geq \underset{\uparrow}{u_i(x_i, x_{-i}^*)} \quad \forall x_i \in \Delta_{A_i}, \forall i \in \{1, 2, \ldots, n\}$$

Better than doing anything else, **"best response"**

Space of probability distributions

- Intuition: nobody can **increase expected reward** by changing only their own strategy. A type of solution!

Key notion: Nash equilibrium. This is the foundation for many important concepts in various areas, like price and market in economics.

In x*, each player has a mixed strategy, a probabilistic distribution over its action space.

Nash equilibrium: all the players do not have an incentive to deviate from its current mixed strategy.

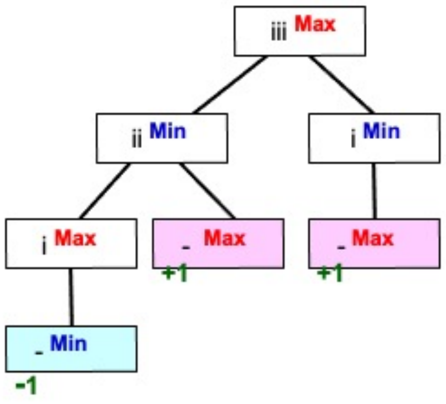
Focus on any player i: If the other players keep their strategies, then player i will suffer from using other mixed strategy.

Q4: Consider a variant of the Nim game. There is only 1 pile of 3 sticks. And the player takes 1 or 2 sticks from a pile. Which is true about the game tree?

1. Max always wins along all possible trajectories
2. The longest trajectory has 3 moves
3. There are 4 possible trajectories
4. None of the above

Q4: Consider a variant of the Nim game. There is only 1 pile of 3 sticks. And the player takes 1 or 2 sticks from a pile. Which is true about the game tree?

1. Max always wins along all possible trajectories
2. [✓ The longest trajectory has 3 moves]
3. There are 4 possible trajectories
4. None of the above



Acknowledgements: Based on slides from Yin Li, Jerry Zhu, Svetlana Lazebnik, Yingyu Liang, David Page, Mark Craven, Pieter Abbeel, Dan Klein