Monte-Carlo Planning: Policy Improvement

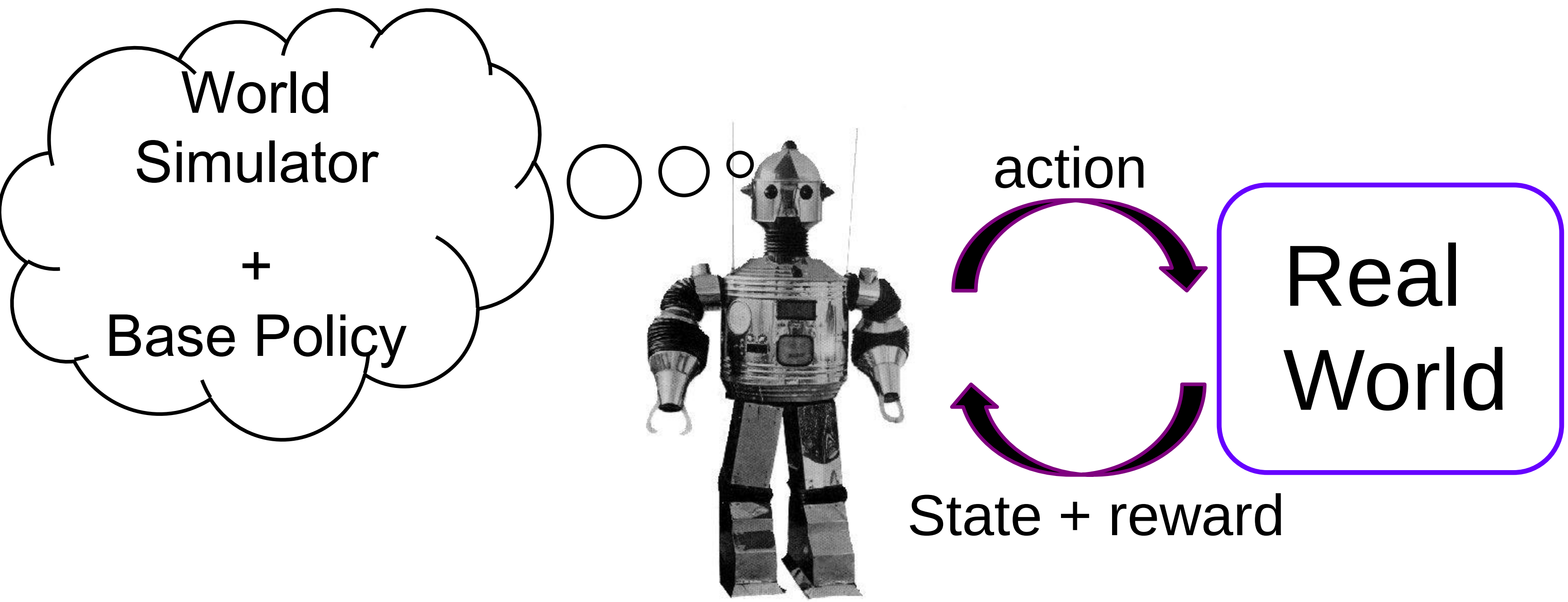
Alan Fern

Monte-Carlo Planning Outline

- Single State Case (multi-armed bandits)
  - A basic tool for other algorithms
- Monte-Carlo Policy Improvement
  - Policy rollout
  - Policy Switching
- Monte-Carlo Tree Search
  - Sparse Sampling
  - UCT and variants

Policy Improvement via Monte-Carlo

- Now consider a very large multi-state MDP.
- Suppose we have a simulator and a non-optimal policy
  - E.g. policy could be a standard heuristic or based on intuition
- Can we somehow compute an improved policy?



Recall: Policy Improvement Theorem

$$Q_\pi(s,a) = R(s,a) + \beta \sum_{s'} T(s,a,s') \cdot V_\pi(s')$$

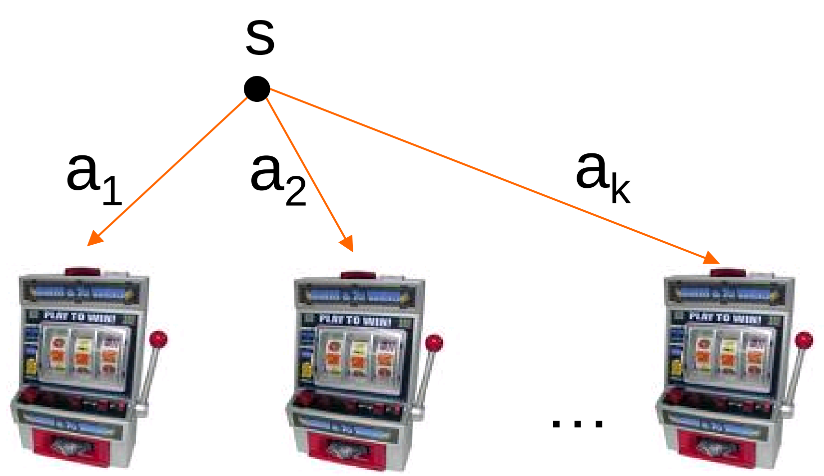
- The Q-value function of a policy gives expected discounted future reward of starting in state s, taking action a, and then following policy π thereafter

- **Define:** $\pi'(s) = \arg\max_a Q_\pi(s,a)$

- **Theorem [Howard, 1960]:** For any non-optimal policy π the policy π' a strict improvement over π.

- Computing π' amounts to finding the action that maximizes the Q-function of π
  - Can we use the bandit idea to solve this?

Policy Improvement via Bandits



SimQ(s,a_1,π) SimQ(s,a_2,π) SimQ(s,a_k,π)

- **Idea:** define a stochastic function **SimQ(s,a,π)** that we can implement and whose expected value is $Q_\pi(s,a)$
- Then use Bandit algorithm to select (approx) best action

How to implement SimQ?

Q-value Estimation

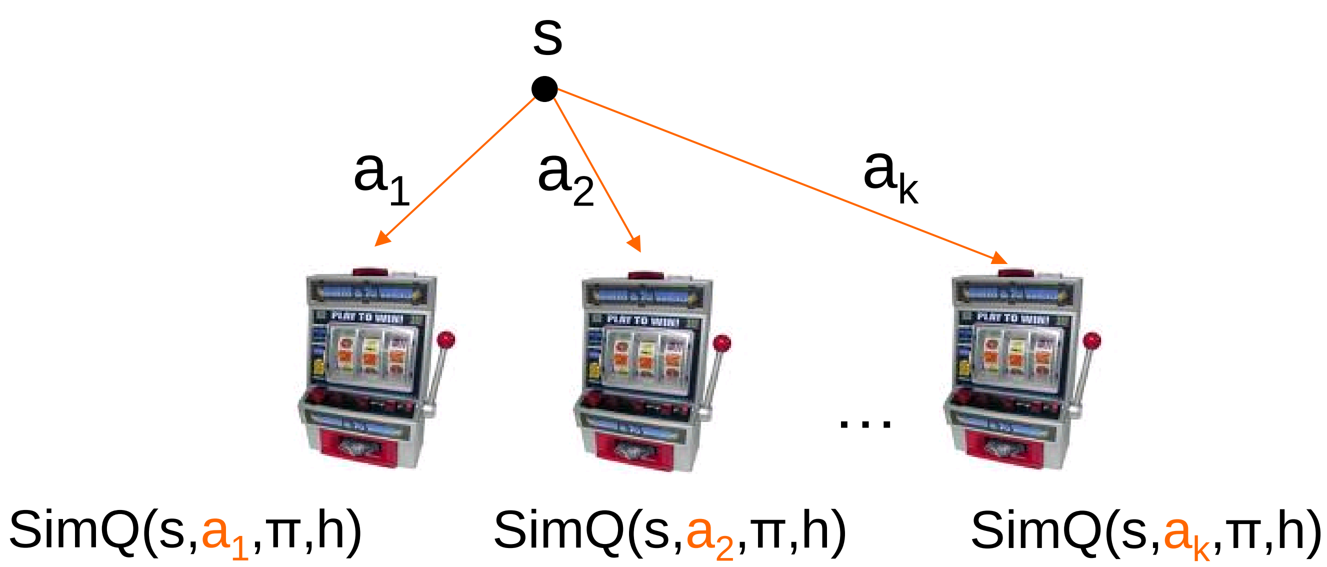
- SimQ might be implemented by simulating the execution of action *a* in state *s* and then following π thereafter
  - But for infinite horizon problems this would never finish
  - So we will approximate via finite horizon

- The *h*-horizon Q-function $Q_\pi(s,a,h)$ is defined as: expected total discounted reward of starting in state *s*, taking action *a*, and then following policy π for *h*-1 steps

- The approximation error decreases exponentially fast in *h*

$$\left|Q_\pi(s,a) - Q_\pi(s,a,h)\right| \le \beta^h V_{\max} \qquad V_{\max} = \frac{R_{\max}}{1-\beta}$$

Policy Improvement via Bandits



- **Refined Idea:** define a stochastic function **SimQ(s,a,π,h)** that we can implement, whose expected value is Q_π(s,a,h)

- Use Bandit algorithm to select (approx) best action

How to implement SimQ?

Policy Improvement via Bandits

SimQ(s,a,π,h)

r = R(s,a)
s = T(s,a)

simulate a in s

for i = 1 to h-1
    $r = r + \beta^i R(s, \pi(s))$
    $s = T(s, \pi(s))$

simulate h-1 steps of policy

Return r

- Simply simulate taking **a** in **s** and following policy for h-1 steps, returning discounted sum of rewards
- Expected value of SimQ(s,a,π,h) is $Q_\pi(s,a,h)$ which can be made arbitrarily close to $Q_\pi(s,a)$ by increasing h

Policy Improvement via Bandits

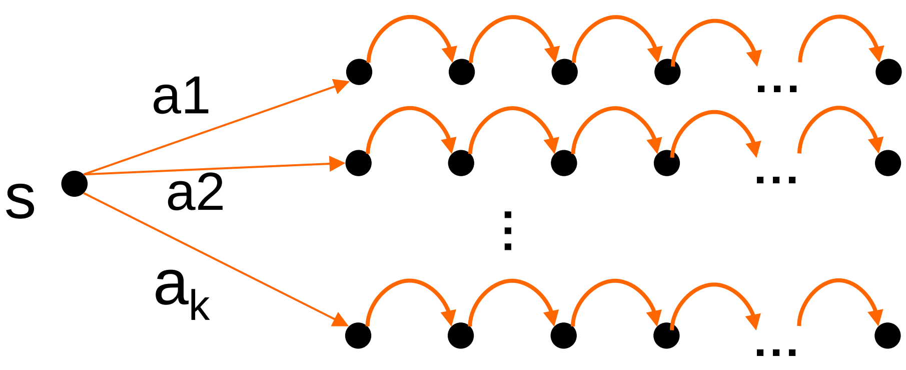
SimQ(s,a,π,h)

r = R(s,a)
s = T(s,a)
} simulate a in s

for i = 1 to h-1
    r = r + β^i R(s, π(s))
    s = T(s, π(s))
} simulate h-1 steps of policy

Return r

<u>Trajectory under π</u>

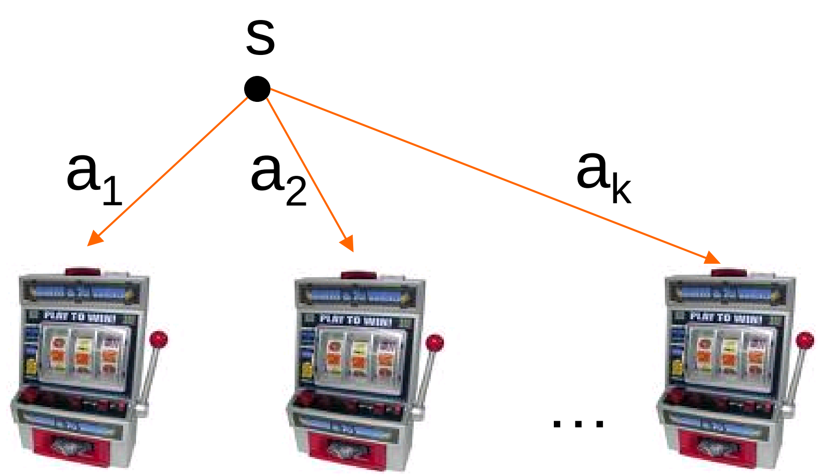


Sum discount rewards = SimQ(s,a_1,π,h)

Sum discount rewards = SimQ(s,a_2,π,h)

Sum discount rewards = SimQ(s,a_k,π,h)

Policy Improvement via Bandits



SimQ(s,a_1,π,h) SimQ(s,a_2,π,h) SimQ(s,a_k,π,h)

- **Refined Idea:** define a stochastic function **SimQ(s,a,π,h)** that we can implement, whose expected value is Q_π(s,a,h)

- Use Bandit algorithm to select (approx) best action

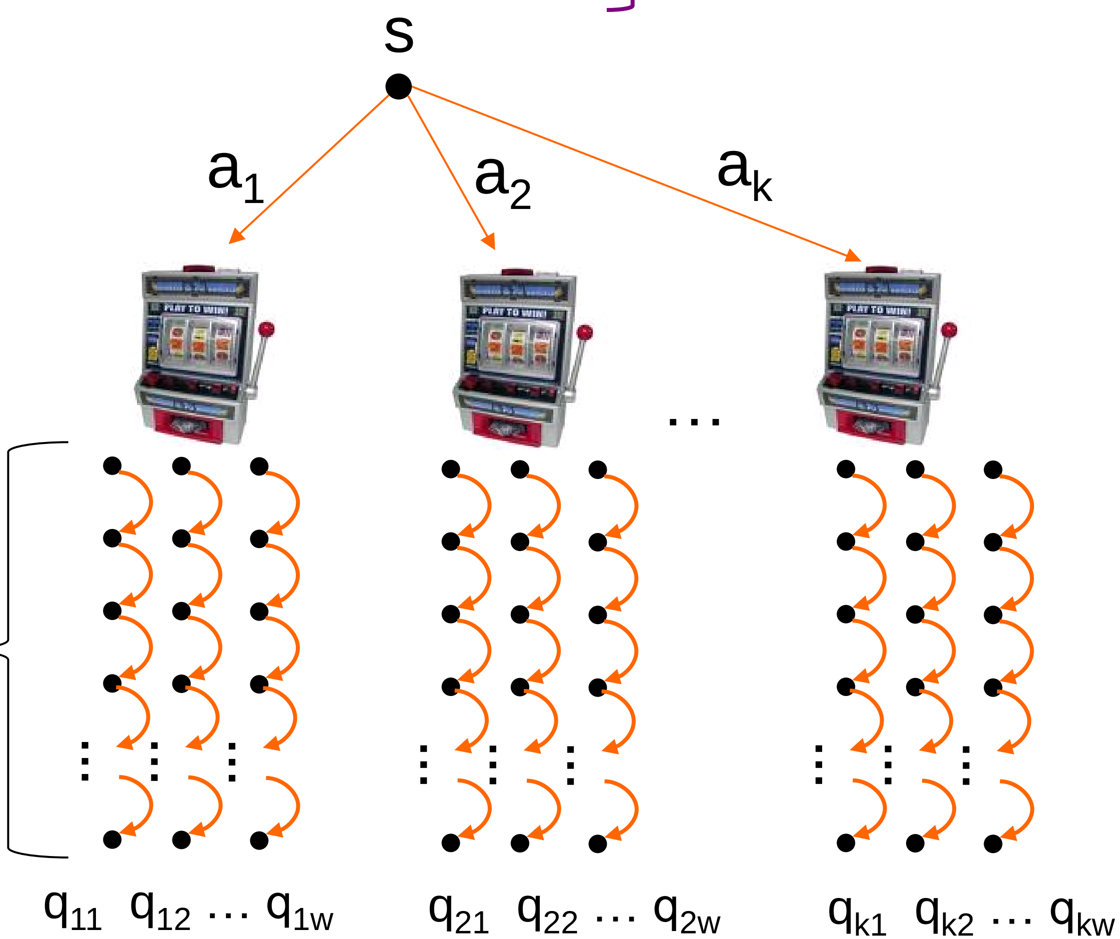
Which bandit objective/algorithm to use?

Traditional Approach: Policy Rollout

UniformRollout[π,h,w](s)

1. For each a_i run SimQ(s,a_i,π,h) **w** times
2. Return action with best average of SimQ results

UniformBandit for PAC objective

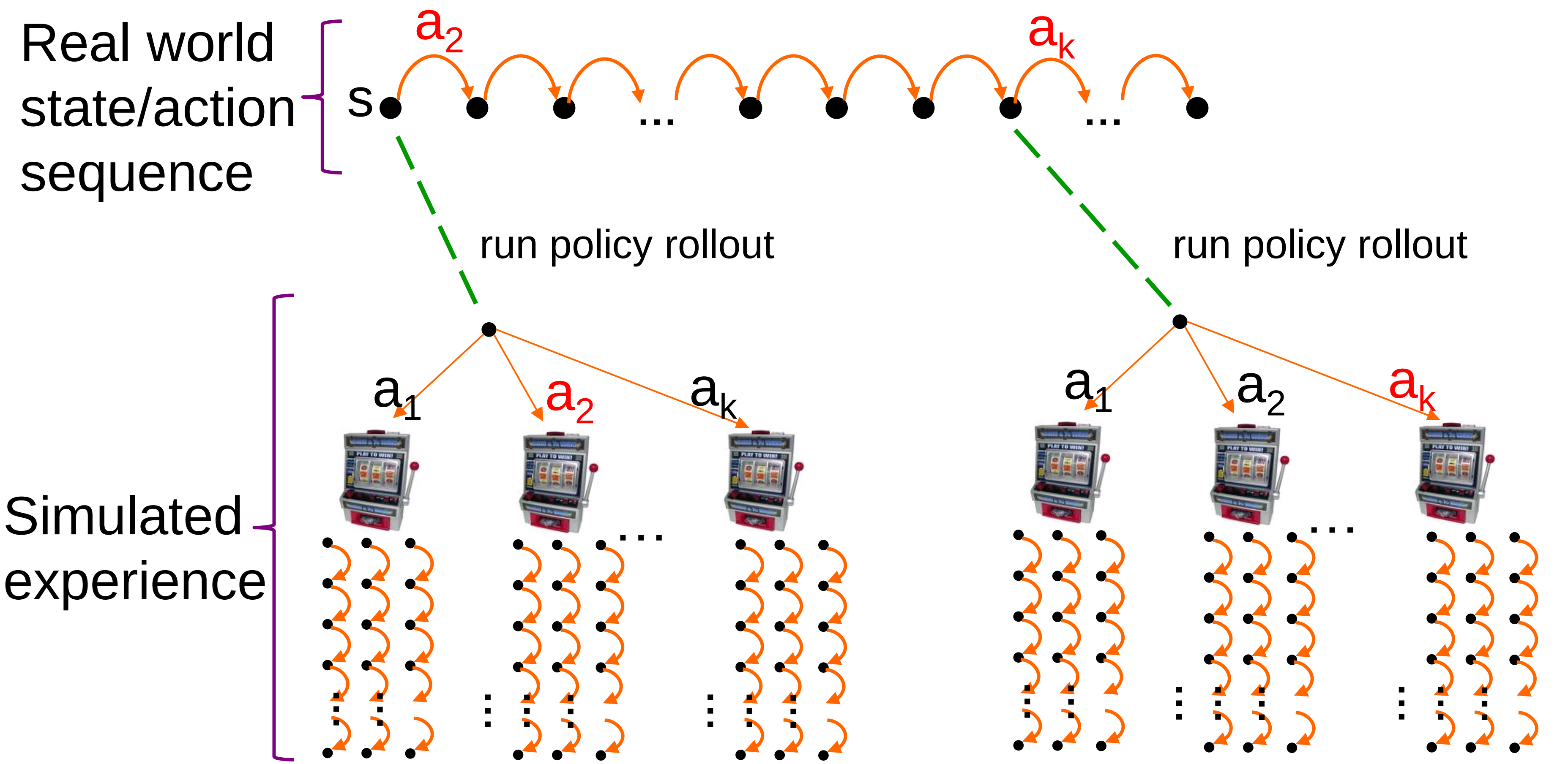


SimQ(s,a_i,π,h) trajectories

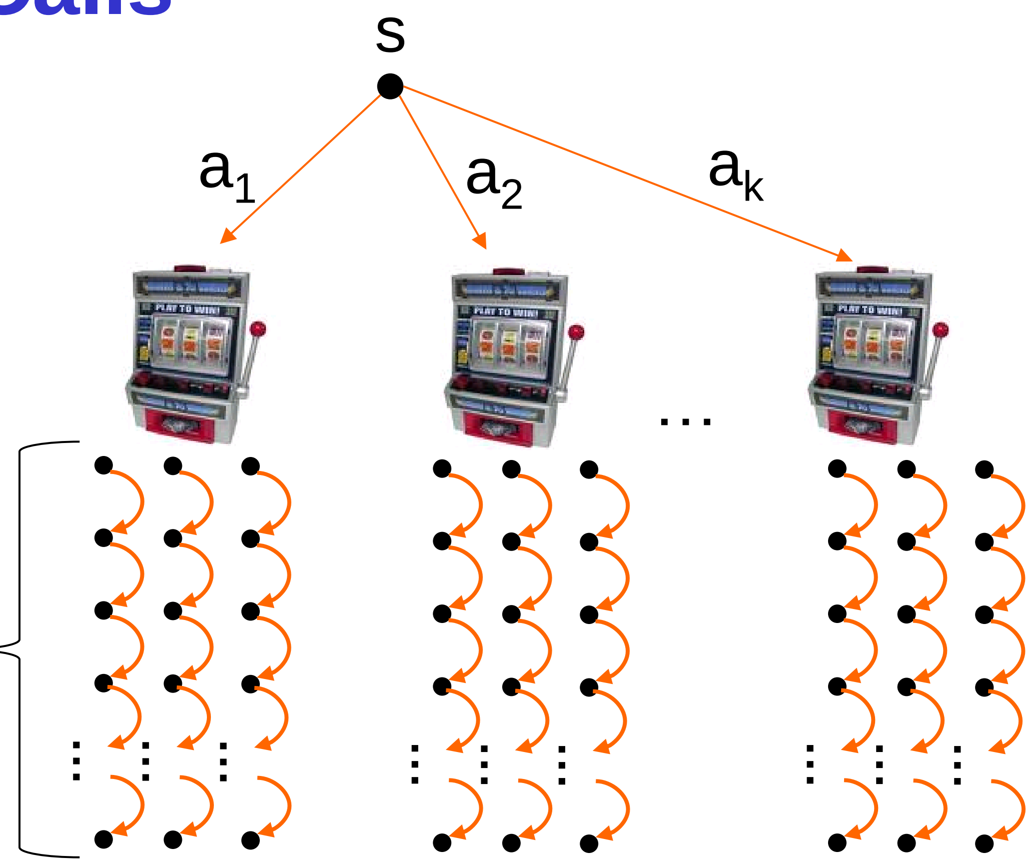
Each simulates taking action a_i then following π for h-1 steps.

Samples of SimQ(s,a_i,π,h)

Executing Rollout in Real World



Uniform Policy Rollout: # of Simulator Calls



SimQ(s,a_i,π,h) trajectories

Each simulates taking action a_i then following π for h-1 steps.

- For each action **w** calls to SimQ, each using h sim calls
- Total of khw calls to the simulator

Uniform Policy Rollout: PAC Guarantee

- Let a* be the action that maximizes the true Q-funciton $Q_\pi(s,a)$.
- Let a' be the action returned by UniformRollout[π,h,w](s).
- Putting the PAC bandit result together with the finite horizon approximation we can derive the following:

If $w \geq \left(\frac{R_{\max}}{\varepsilon}\right)^2 \ln\frac{k}{\delta}$ then with probability at least $1-\delta$

$$\left|Q_\pi(s,a^*) - Q_\pi(s,a')\right| \leq \varepsilon + \beta^h V_{\max}$$

But does this guarantee that the value of UniformRollout[π,h,w](s) will be close to the value of π' ?

Policy Rollout: Quality

- How good is UniformRollout[π,h,w] compared to π'?

- **Bad News.** In general for a fixed h and w there is always an MDP such that the quality of the rollout policy is arbitrarily worse than π'.

- The example MDP is somewhat involved, but shows that even small error in Q-value estimates can lead to large performance gaps compared to π'
  - But this result is quite pathological

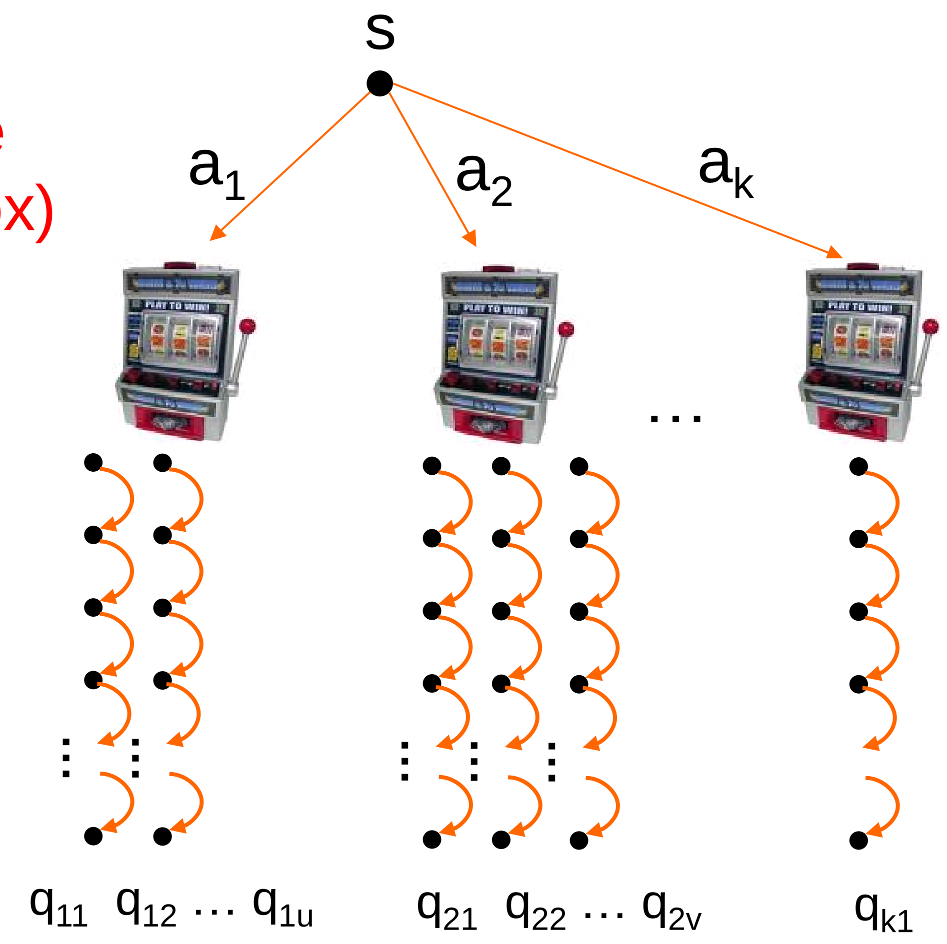
Policy Rollout: Quality

- How good is UniformRollout[π,h,w] compared to π'?
- **Good News.** If we make an assumption about the MDP, then it is possible to select h and w so that the rollout quality is close to π'.
  - This is a bit involved.
  - Assume a lower bound on the difference between the best Q-value and the second best Q-value
- **More Good News.** It is possible to select h and w so that Rollout[π,h,w] is (approximately) no worse than π for any MDP
  - So at least rollout won't hurt compared to the base policy
  - At the same time it has the potential to significantly help

Non-Uniform Policy Rollout

- Should we consider minimizing cumulative regret?

No! We really only care about finding an (approx) best arm.

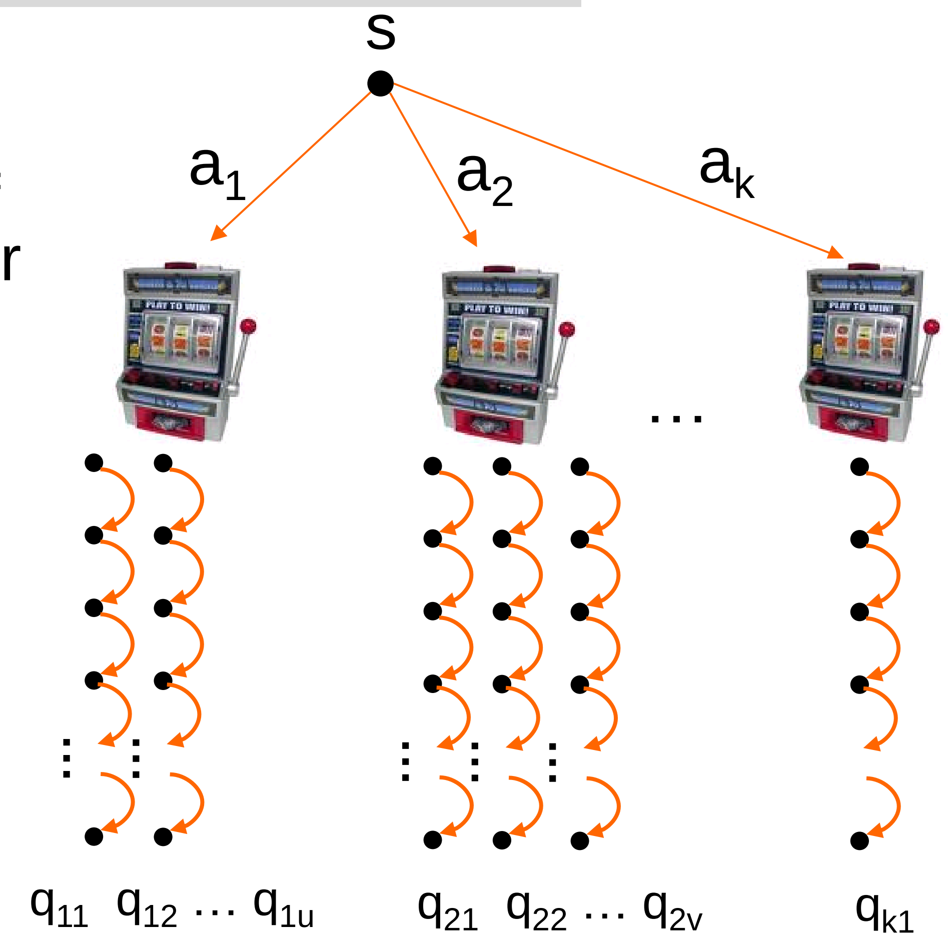


Non-Uniform Policy Rollout

PAC Setting: use **MedianElimination**

(parameterized by ϵ and δ instead of w)

- Often we are given a budget on number of samples (i.e. time per decision).
- MedianElimination not applicable.

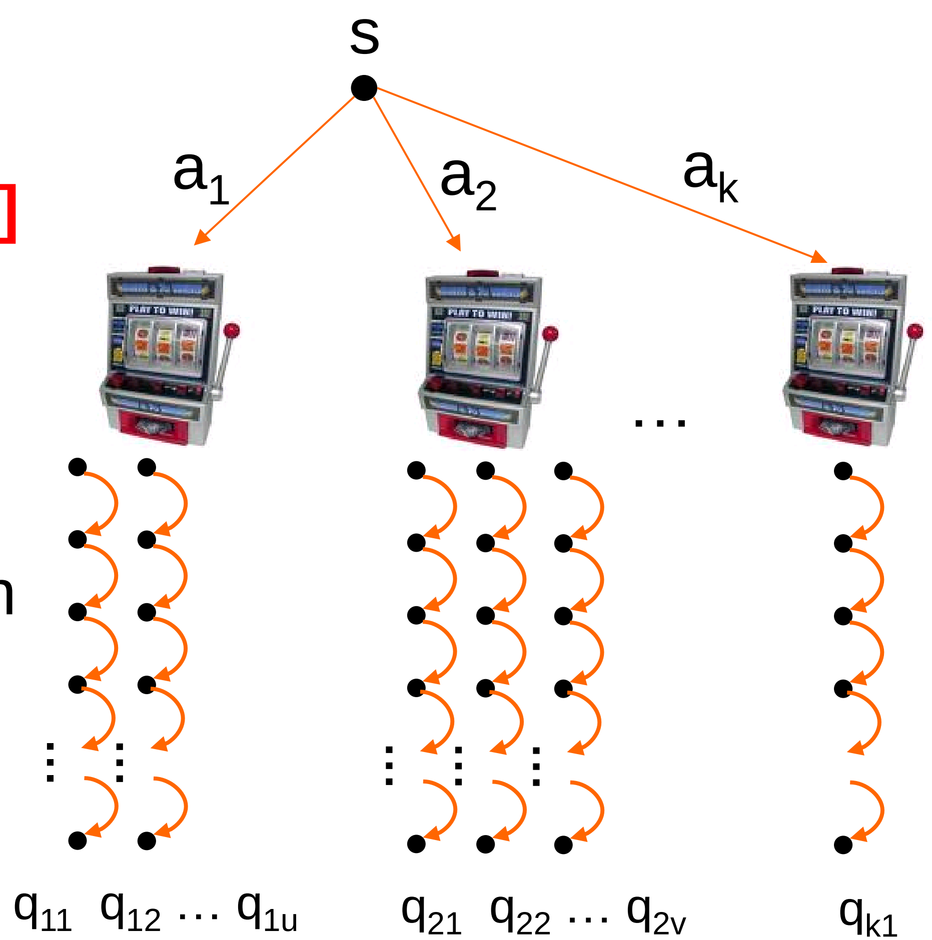


Non-Uniform Policy Rollout

Simple Regret: use **ϵ-Greedy**

(parameterized by budget n on # of pulls)

- Call this **ϵ-Rollout[π,h,n]**
- n is number of samples per step
- For $\epsilon = 0.5$ we might expect it to be better than UniformRollout for same # of total samples.



Multi-Stage Rollout

- In what follows we will use the notation **Rollout[π]** to refer to either UniformRollout[π,h,w] or ϵ-Rollout[π,h,n].

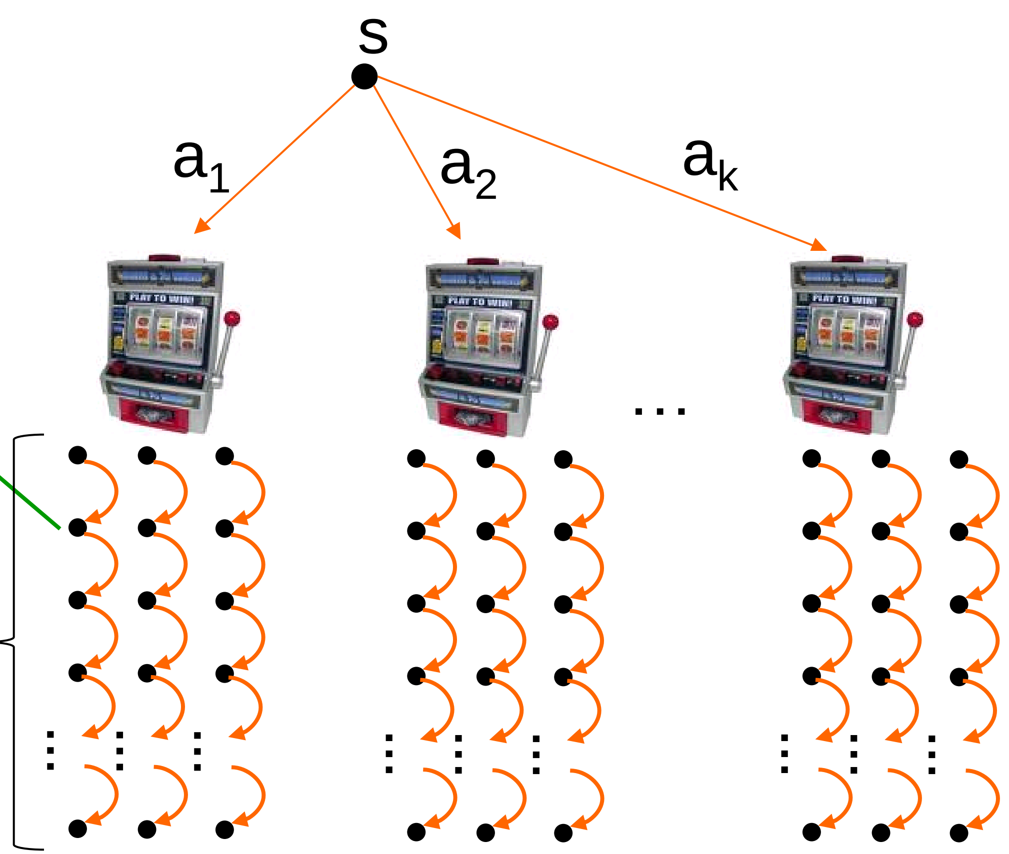
- A single call to Rollout[π](s) approximates one iteration of policy iteration inialized at policy π
  - But only computes the action for state s rather than all states (as done by full policy iteration)!

- We can use more computation time to approximate multiple iterations of policy iteration via nesting calls to Rollout

- Gives a way to use more time in order to improve performance

Multi-Stage Rollout

Each step requires khw simulator calls for Rollout policy



Trajectories of SimQ(s,a_i,Rollout[π],h)

- Two stage: compute rollout policy of "rollout policy of π"
- Requires $(khw)^2$ calls to the simulator for 2 stages
- In general exponential in the number of stages

Rollout Summary

- We often are able to write simple, mediocre policies
  - Network routing policy
  - Policy for card game of Hearts
  - Policy for game of Backgammon
  - Solitaire playing policy
- Policy rollout is a general and easy way to improve upon such policies given a simulator
- Often observe substantial improvement, e.g.
  - Compiler instruction scheduling
  - Backgammon
  - Network routing
  - Combinatorial optimization
  - Game of GO
  - Solitaire

Example: Rollout for Solitaire [Yan et al. NIPS'04]

Player	Success Rate	Time/Game
Human Expert	36.6%	20 min
(naïve) Base Policy	13.05%	0.021 sec
1 rollout	31.20%	0.67 sec
2 rollout	47.6%	7.13 sec
3 rollout	56.83%	1.5 min
4 rollout	60.51%	18 min
5 rollout	70.20%	1 hour 45 min

- Multiple levels of rollout can payoff but is expensive

Monte-Carlo Planning Outline

- Single State Case (multi-armed bandits)
  - A basic tool for other algorithms
- Monte-Carlo Policy Improvement
  - Policy rollout
  - Policy Switching
- Monte-Carlo Tree Search
  - Sparse Sampling
  - UCT and variants

Another Useful Technique: Policy Switching

- Sometimes policy rollout can be too expensive when the number of actions is large (time scales linearly with number of actions)

- Sometimes we have multiple base policies and it is hard to pick just one to use for rollout.

- Policy switching helps deal with both of these issues.

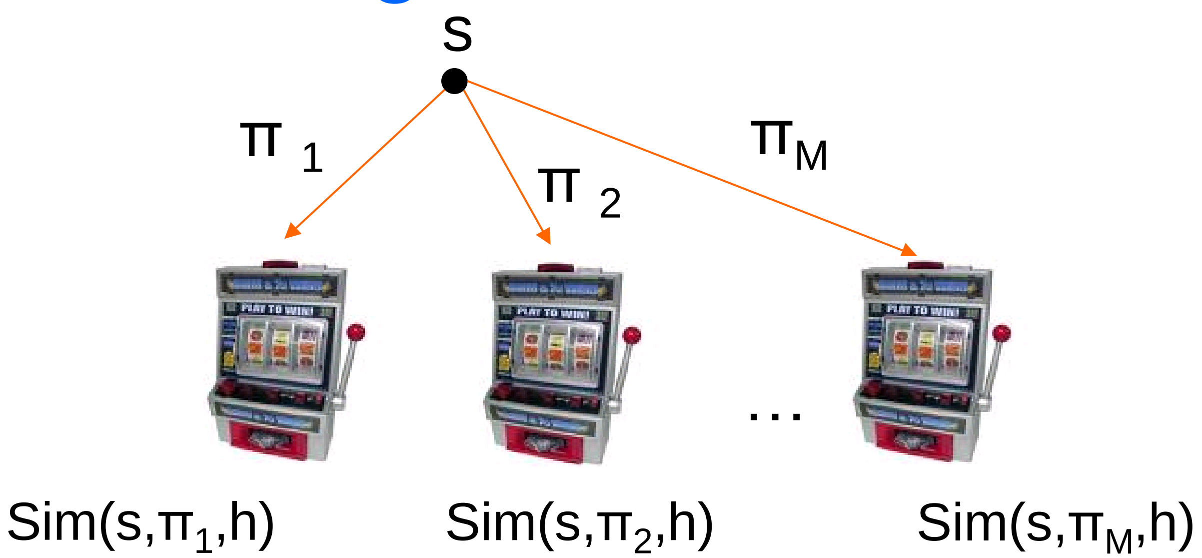
Another Useful Technique: Policy Switching

- Suppose you have a set of base policies $\{\pi_1, \pi_2, \ldots, \pi_M\}$

- Also suppose that the best policy to use can depend on the specific state of the system and we don't know how to select.

- Policy switching is a simple way to select which policy to use at a given step via a simulator

Another Useful Technique: Policy Switching

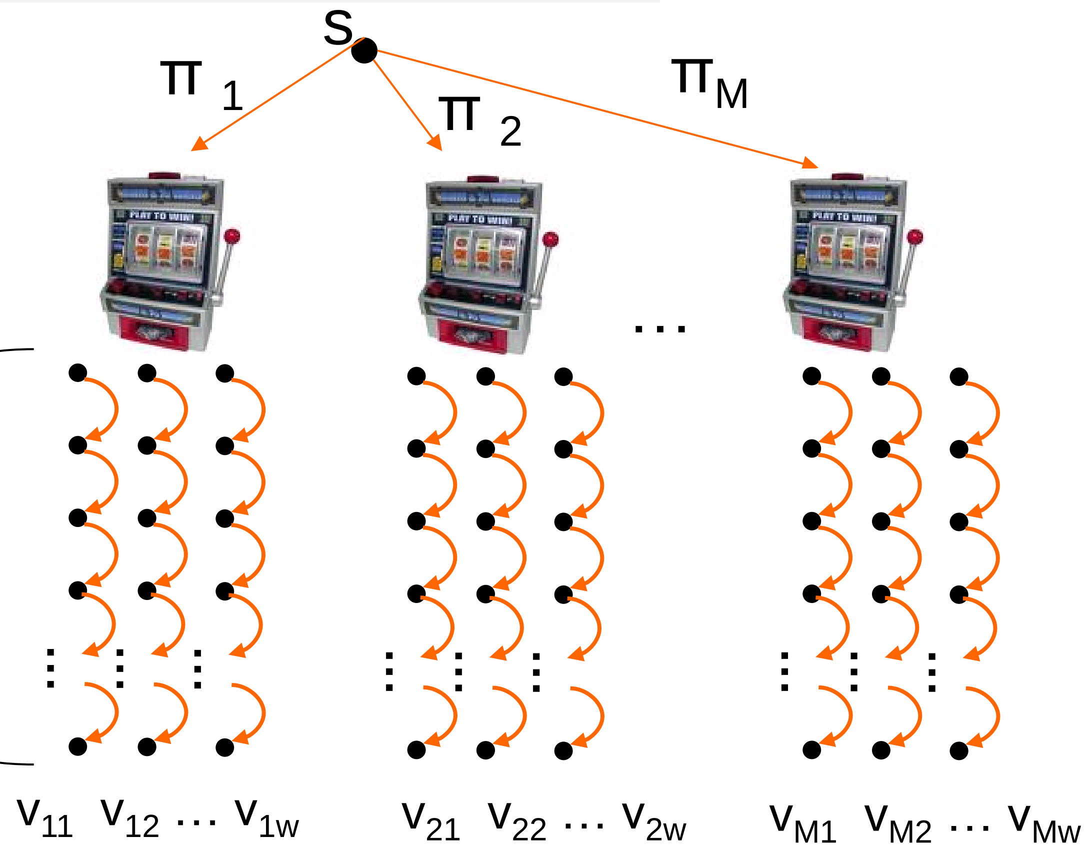


- The stochastic function **Sim(s,π,h)** simply samples the h-horizon value of π starting in state s
- Implement by simply simulating π starting in s for h steps and returning discounted total reward
- Use Bandit algorithm to select best policy and then select action chosen by that policy

Uniform Policy Switching

UniformPolicySwitch[{π_1, π_2,…, π_M},h,w](s)

1. For each π_i run Sim(s,π_i,h) **w** times
2. Let i* be index of policy with best average result
3. Return action π_{i^*}(s)

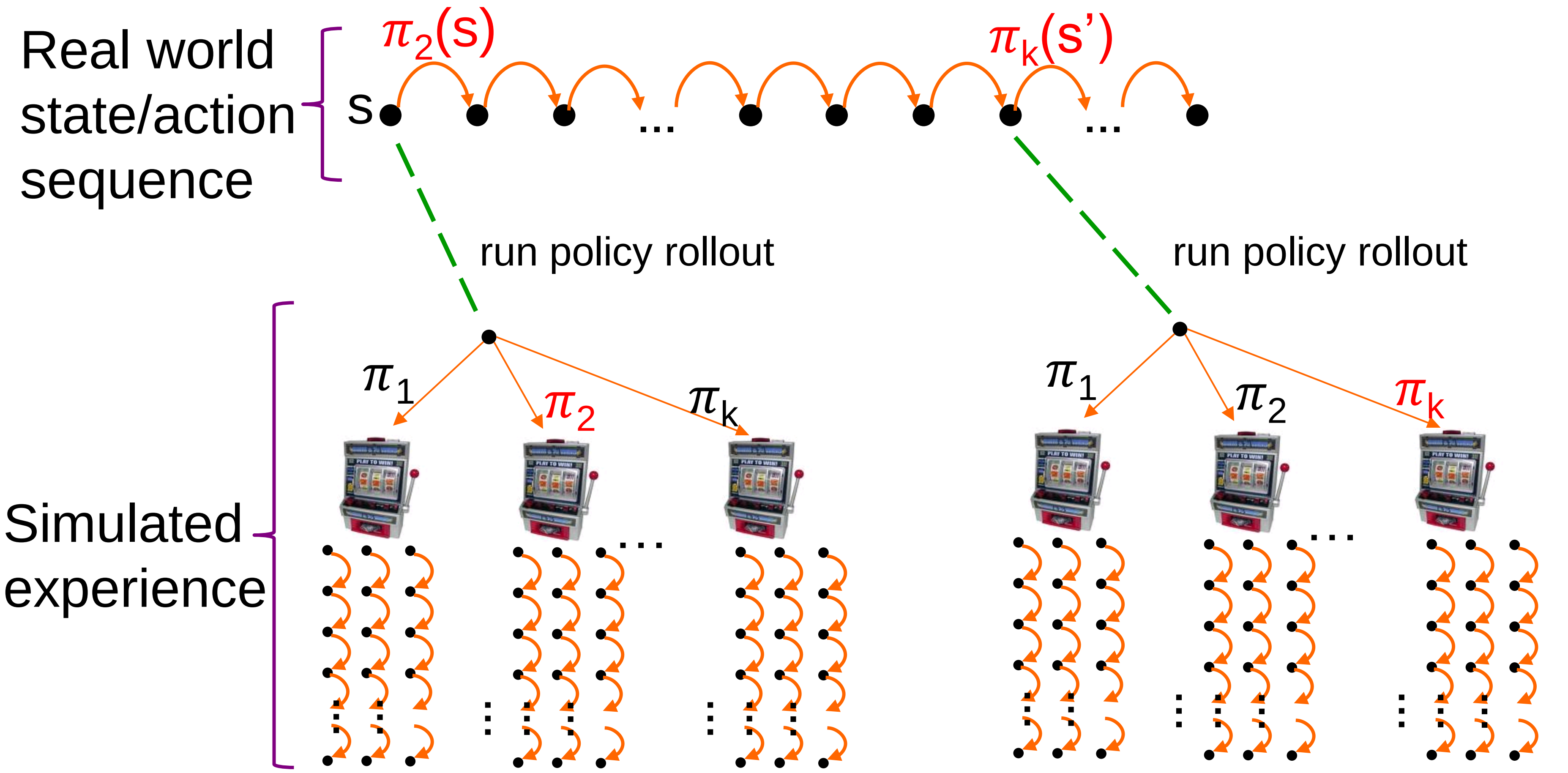


Sim(s,π_i,h) trajectories

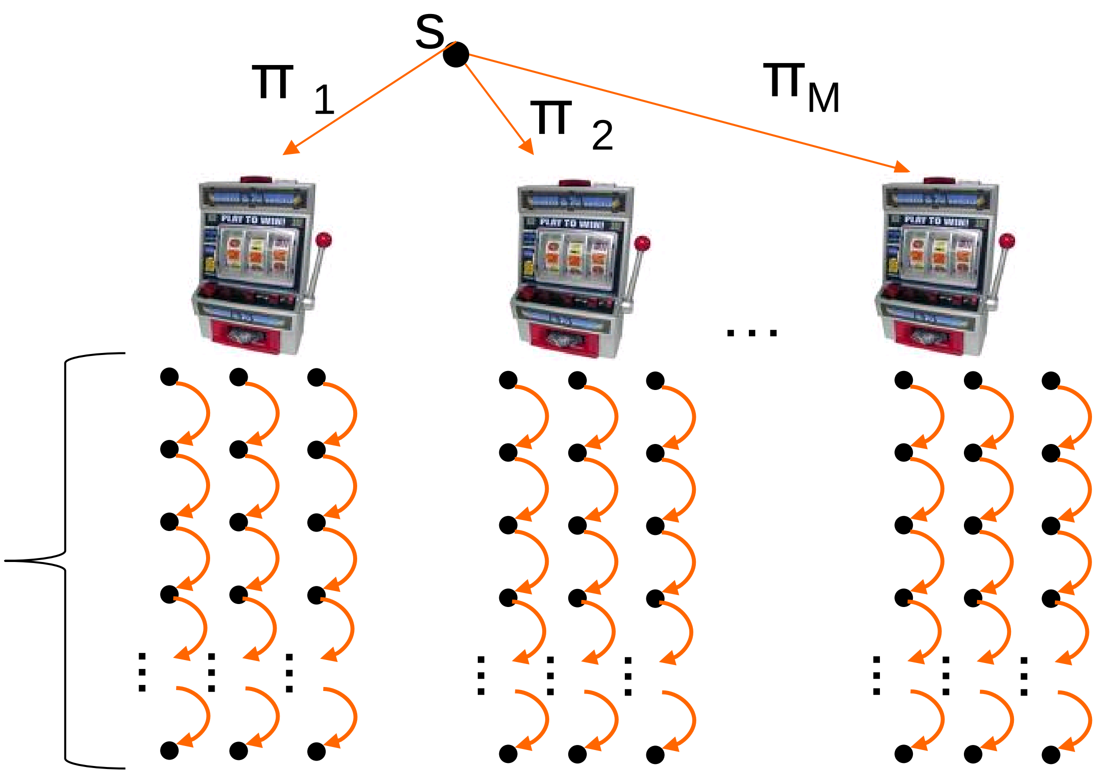
Each simulates following π_i for h steps.

Discounted cumulative rewards

Executing Policy Switching in Real World



Uniform Policy Switching: Simulator Calls



Sim(s,π_i,h) trajectories

Each simulates taking following π_i for h steps.

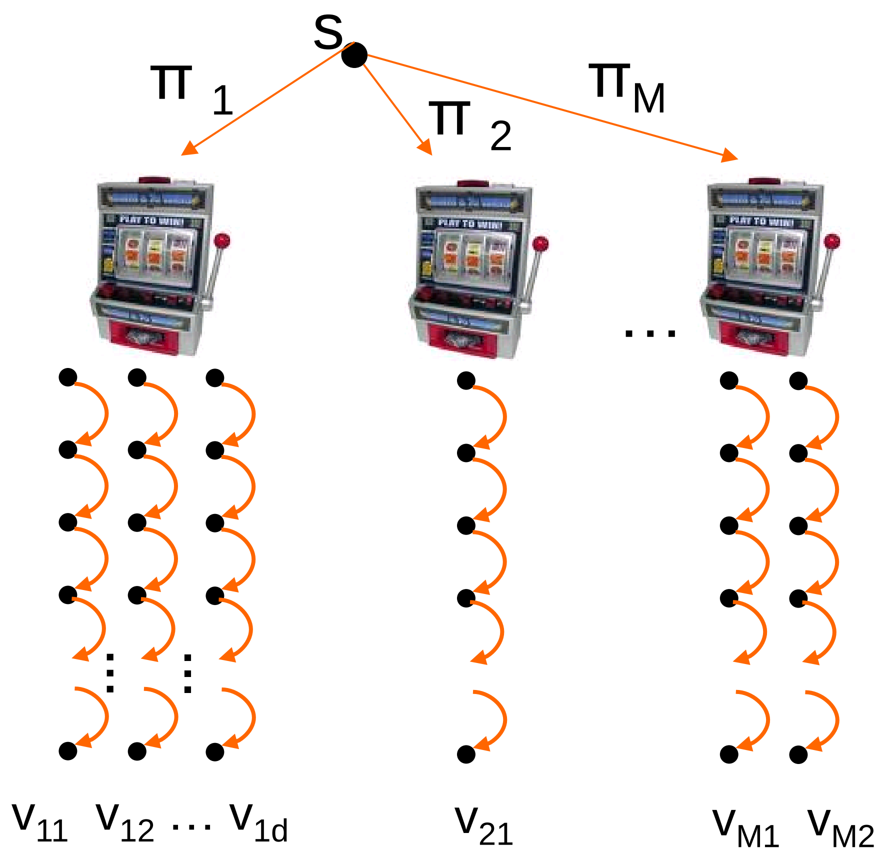
- For each policy use **w** calls to Sim, each using h simulator calls
- Total of Mhw calls to the simulator
- Does not depend on number of actions!

ϵ-Greedy Policy Switching

- Similar to rollout we can have a non-uniform version that takes a total number of trajectories n as an argument

ϵ-PolicySwitch[{$\pi_1,\ldots,\pi_M$},h,n]

Use ϵ-Greedy as the bandit algorithm for n pulls and return best arm/policy.



Policy Switching: Quality

- Let π_{ps} denote the ideal switching policy
  - Always pick the best policy index at any state

Theorem: For any state s, $\max_i V_{\pi_i}(s) \le V_{\pi_{ps}}(s)$.

- The value of the switching policy is at least as good as the best single policy in the set
  - It will often perform better than any single policy in set.
  - For non-ideal case, were bandit algorithm only picks approximately the best arm we can add an error term to the bound.

Proof

Theorem: For any state s, $\max_i V_{\pi_i}(s) \leq V_{\pi_{ps}}(s)$.

We'll use the following property.

Proposition: For any policy π and value function V, if $V \leq B_\pi[V]$, then $V \leq V_\pi$

Recall $B_\pi[V](s) = R(s) + \sum_{s'} T(s, \pi(s), s') \cdot V(s')$ is the restricted Bellman backup.

So all we need to do is prove that $\max_i V_{\pi_i} \leq B_{\pi_{ps}}\left[\max_i V_{\pi_i}\right]$ since this will imply that $\max_i V_{\pi_i} \leq V_{\pi_{ps}}$ as desired.

Proof *(to simply notation and without loss of generality, assume rewards only depend on state and are deterministic)*

Prove that $\max_i V_{\pi_i} \leq B_{\pi_{ps}}\left[\max_i V_{\pi_i}\right]$

Let i^* be the index of the best policy in state s.

$$B_{\pi_{ps}}\left[\max_i V_{\pi_i}\right](s) = R(s) + \sum_{s'} T\big(s, \pi_{ps}(s), s'\big) \cdot \max_i V_{\pi_i}(s')$$

$$\geq\ R(s) + \max_i \sum_{s'} T(s, \pi_{i^*}(s), s') \cdot V_{\pi_i}(s')$$

$$= \max_i \left[R(s) + \sum_{s'} T(s, \pi_{i^*}(s), s') \cdot V_{\pi_i}(s')\right]$$

$$\geq \max_i \left[R(s) + \sum_{s'} T(s, \pi_i(s), s') \cdot V_{\pi_i}(s')\right]$$

$$= \max_i V_{\pi_i}(s)$$

Policy Switching Summary

- Easy way to produce an improved policy from a set of existing policies.
  - Will not do any worse than the best policy in your set.
- Complexity does not depend on number of actions.
  - So can be practical even when action space is huge, unlike policy rollout.
- Can combine with rollout for further improvement
  - Just apply rollout to the switching policy.