Monte-Carlo Planning: Basic Principles and Recent Progress

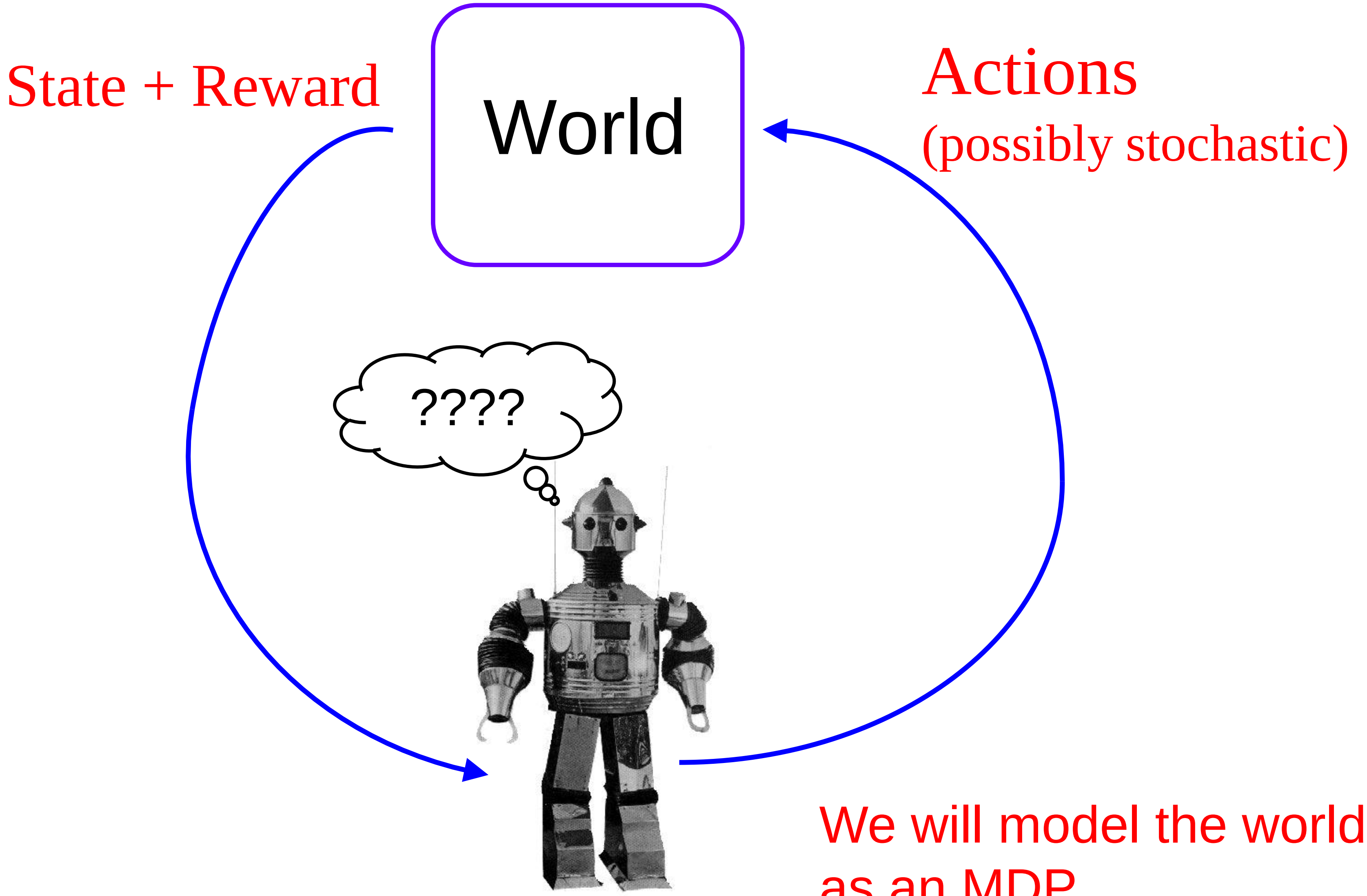
Alan Fern

School of EECS
Oregon State University

Outline

- Preliminaries: Markov Decision Processes
- What is Monte-Carlo Planning?
- Uniform Monte-Carlo
  - Single State Case (PAC Bandit)
  - Policy rollout
  - Sparse Sampling
- Adaptive Monte-Carlo
  - Single State Case (UCB Bandit)
  - UCT Monte-Carlo Tree Search

Stochastic/Probabilistic Planning: Markov Decision Process (MDP) Model

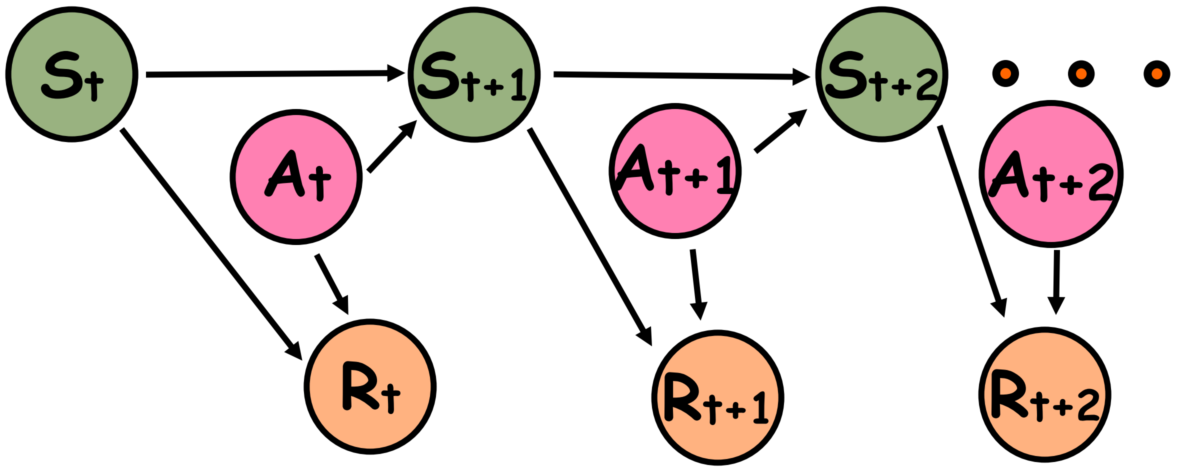


We will model the world as an MDP.

Markov Decision Processes

- An MDP has four components: S, A, P_R, P_T:
  - finite state set S
  - finite action set A
  - Transition distribution $P_T(s' \mid s, a)$
    - Probability of going to state **s'** after taking action **a** in state **s**
    - First-order Markov model
  - Bounded reward distribution $P_R(r \mid s, a)$
    - Probability of receiving immediate reward **r** after taking action **a** in state **s**
    - First-order Markov model

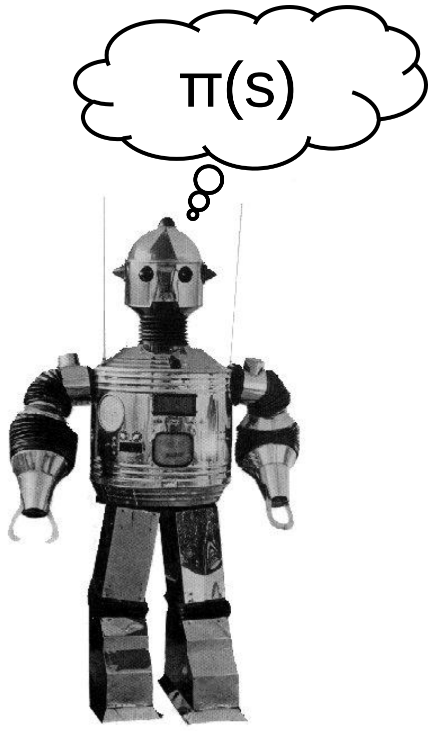
Graphical View of MDP



- **First-Order Markovian dynamics** (history independence)
  - Next state only depends on current state and current action
- **First-Order Markovian reward process**
  - Reward only depends on current state and action

Policies ("plans" for MDPs)

- Given an MDP we wish to compute a policy
  - Could be computed offline or online.
- A policy is a possibly stochastic mapping from states to actions
  - $\pi: S \rightarrow A$
  - $\pi(s)$ is action to do at state s
  - specifies a continuously reactive controller



How to measure goodness of a policy?

Value Function of a Policy

- We consider finite-horizon discounted reward, discount factor $0 \le \beta < 1$
- $V_\pi(s,h)$ denotes expected h-horizon discounted total reward of policy π at state s
  - Each run of π for h steps produces a random reward sequence: $R_1 \; R_2 \; R_3 \; \ldots \; R_h$
  - $V_\pi(s,h)$ is the expected discounted sum of this sequence

$$V_\pi(s,h) = E\left[\sum_{t=0}^{h} \beta^t R_t \mid \pi, s\right]$$

- Optimal policy π* is policy that achieves maximum value across all states

Relation to Infinite Horizon Setting

- Often value function $V_\pi(s)$ is defined over infinite horizons for a discount factor $0 \leq \beta < 1$

$$V_\pi(s) = E[\sum_{t=0}^{\infty} \beta^t R^t \mid \pi, s]$$

- It is easy to show that difference between $V_\pi(s,h)$ and $V_\pi(s)$ shrinks exponentially fast as h grows

$$\left|V_\pi(s) - V_\pi(s,h)\right| \leq \left(\frac{R_{\max}}{1-\beta}\right)\beta^h$$

- **h-horizon results apply to infinite horizon setting**

Computing a Policy

- Optimal policy maximizes value at each state
- Optimal policies guaranteed to exist [Howard, 1960]
- When state and action spaces are small and MDP is known we find optimal policy in poly-time via LP
  - Can also use value iteration or policy Iteration
- We are interested in the case of exponentially large state spaces.

Large Worlds: Model-Based Approach

1. Define a language for compactly describing MDP model, for example:
   - Dynamic Bayesian Networks
   - Probabilistic STRIPS/PDDL
2. Design a planning algorithm for that language

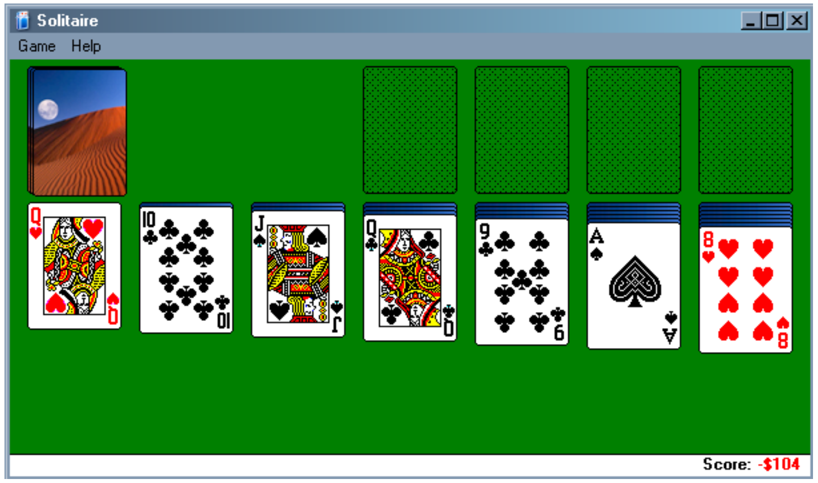
Problem: more often than not, the selected language is inadequate for a particular problem, e.g.

- Problem size blows up
- Fundamental representational shortcoming

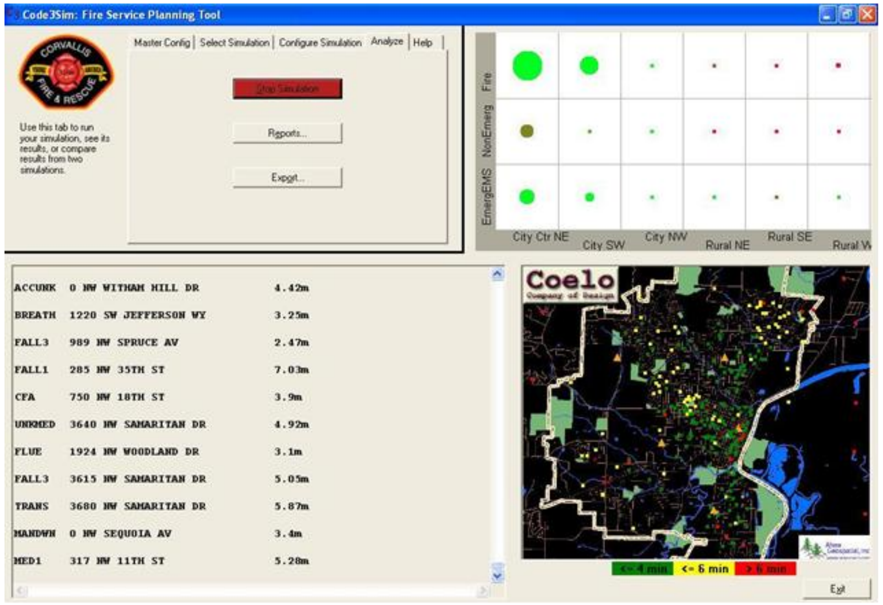
Large Worlds: Monte-Carlo Approach

- Often a simulator of a planning domain is available or can be learned from data
  - Even when domain can't be expressed via MDP language

Klondike Solitaire



Fire & Emergency Response

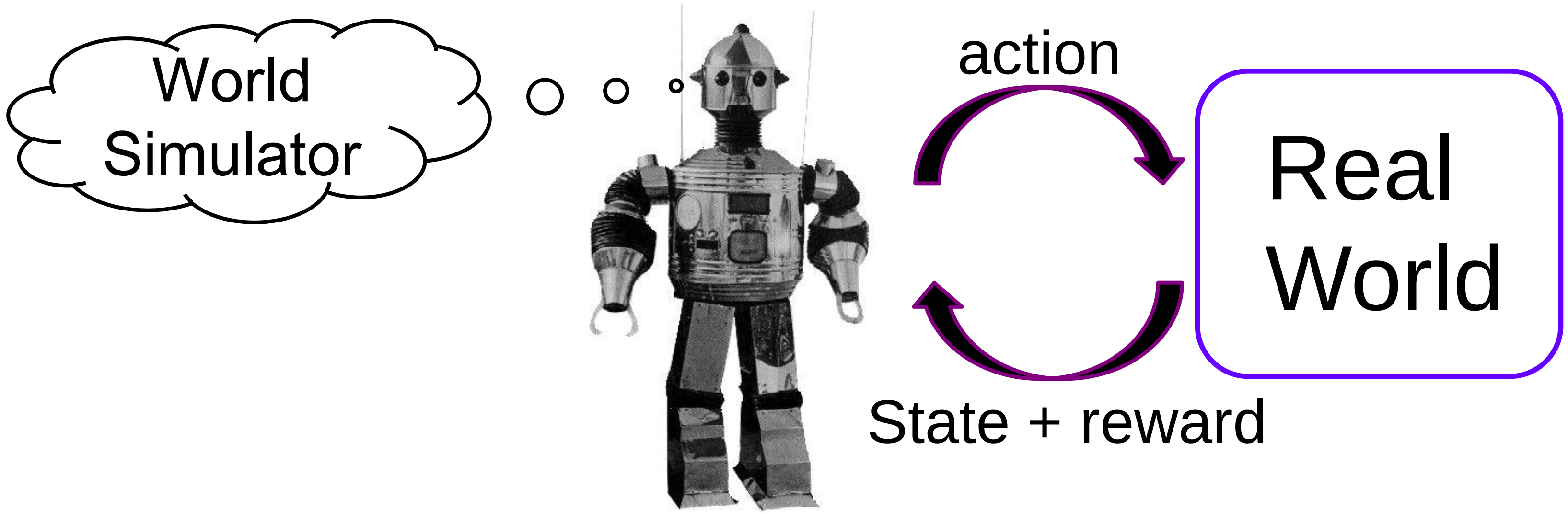


Use this tab to run your simulation, see its results, or compare results from two simulations.

```
ACCUNK   0 NW WITHAM HILL DR        4.42m
BREATH   1220 SW JEFFERSON WY       3.25m
FALL3    989 NW SPRUCE AV           2.47m
FALL1    285 NW 35TH ST             7.03m
CFA      750 NW 18TH ST             3.9m
UNKMED   3640 NW SAMARITAN DR       4.92m
FLUE     1924 NW WOODLAND DR        3.1m
FALL3    3615 NW SAMARITAN DR       5.05m
TRANS    3680 NW SAMARITAN DR       5.87m
MANDWN   0 NW SEQUOIA AV            3.4m
MED1     317 NW 11TH ST             5.28m
```

Large Worlds: Monte-Carlo Approach

- Often a simulator of a planning domain is available or can be learned from data
  - Even when domain can't be expressed via MDP language
- **Monte-Carlo Planning:** compute a good policy for an MDP by interacting with an MDP simulator



Example Domains with Simulators

- Traffic simulators
- Robotics simulators
- Military campaign simulators
- Computer network simulators
- Emergency planning simulators
  - large-scale disaster and municipal
- Sports domains (Madden Football)
- Board games / Video games
  - Go / RTS

In many cases Monte-Carlo techniques yield state-of-the-art performance. Even in domains where model-based planner is applicable.

MDP: Simulation-Based Representation

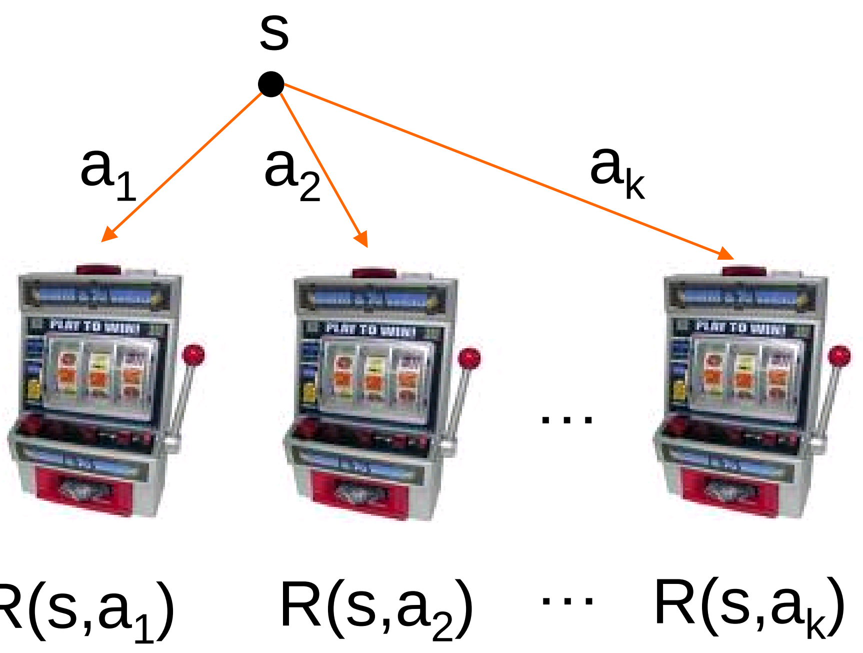
- A simulation-based representation gives: S, A, R, T:
  - finite state set S (generally very large)
  - finite action set A
  - Stochastic, real-valued, bounded reward function R(s,a) = r
    - Stochastically returns a reward **r** given input s and a
    - Can be implemented in arbitrary programming language
  - Stochastic transition function T(s,a) = s' (i.e. a simulator)
    - Stochastically returns a state **s'** given input s and a
    - Probability of returning **s'** is dictated by Pr(s' | s,a) of MDP
    - T can be implemented in an arbitrary programming language

Outline

- Preliminaries: Markov Decision Processes
- What is Monte-Carlo Planning?
- Uniform Monte-Carlo
  - Single State Case (Uniform Bandit)
  - Policy rollout
  - Sparse Sampling
- Adaptive Monte-Carlo
  - Single State Case (UCB Bandit)
  - UCT Monte-Carlo Tree Search

Single State Monte-Carlo Planning

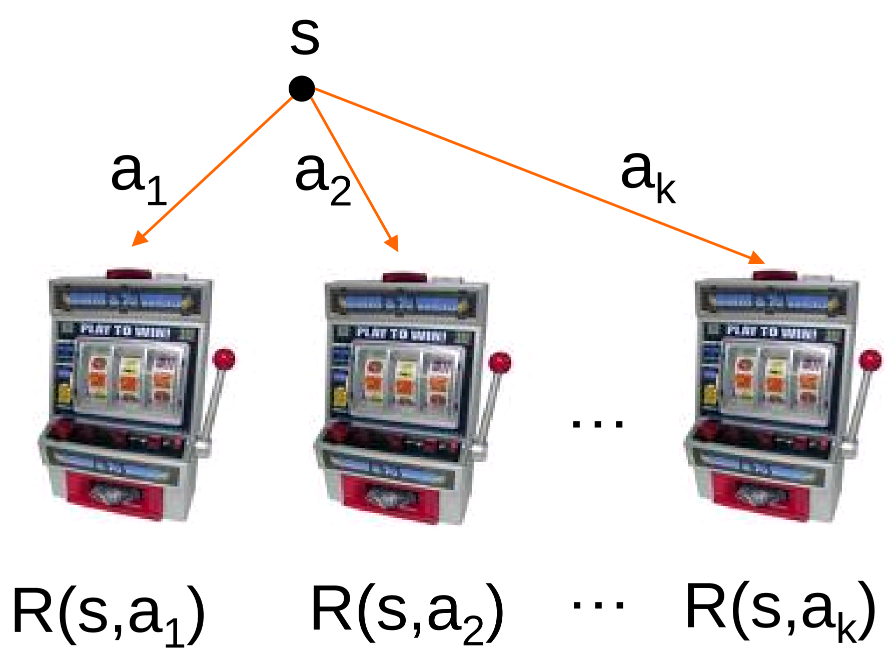
- Suppose MDP has a single state and k actions
  - Figure out which action has best expected reward
  - Can sample rewards of actions using calls to simulator
  - Sampling a is like pulling slot machine arm with random payoff function R(s,a)



Multi-Armed Bandit Problem

PAC Bandit Objective

- **Probably Approximately Correct (PAC)**
  - Select an arm that probably (w/ high probability) has approximately the best expected reward
  - Use as few simulator calls (or pulls) as possible

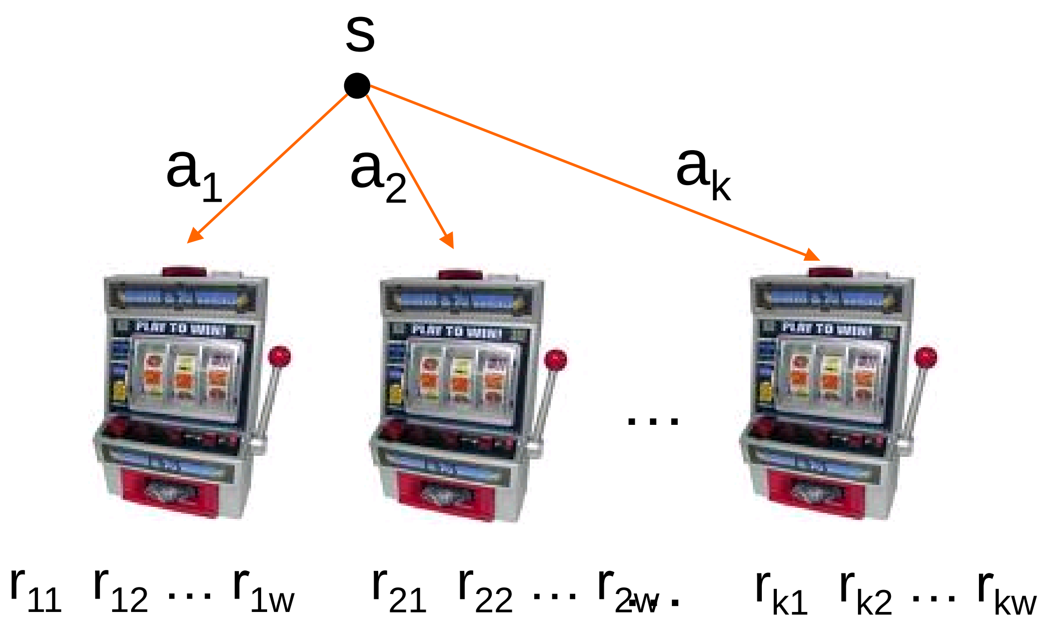


Multi-Armed Bandit Problem

UniformBandit Algorithm

NaiveBandit from [Even-Dar et. al., 2002]

1. Pull each arm **w** times (uniform pulling).
2. Return arm with best average reward.


s
a_1
a_2
a_k
…
r_{11} r_{12} … r_{1w}
r_{21} r_{22} … r_{2w} …
r_{k1} r_{k2} … r_{kw}

How large must w be to provide a PAC guarantee?

Aside: Additive Chernoff Bound

- Let R be a random variable with maximum absolute value Z. An let r_i i=1,…,w be i.i.d. samples of R
- The Chernoff bound gives a bound on the probability that the average of the r_i are far from E[R]

Chernoff Bound

$$\Pr\left(\left| E[R] - \tfrac{1}{w}\sum_{i=1}^{w} r_i \right| \geq \varepsilon \right) \leq \exp\left(-\left(\frac{\varepsilon}{Z} \right)^2 w \right)$$

Equivalently:

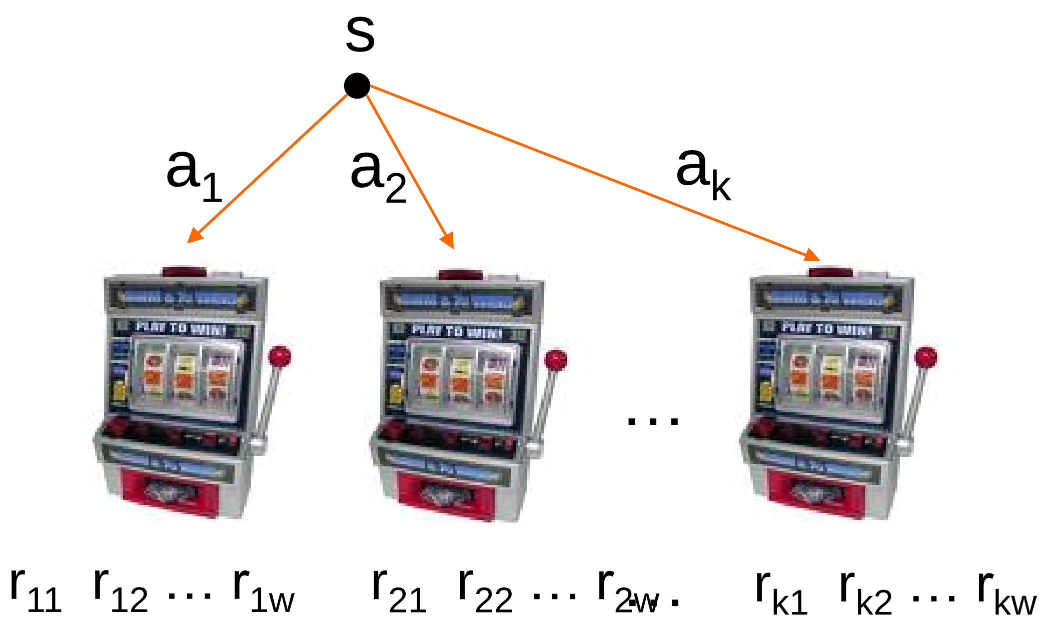
With probability at least $1-\delta$ we have that,

$$\left| E[R] - \tfrac{1}{w}\sum_{i=1}^{w} r_i \right| \leq Z\sqrt{\tfrac{1}{w}\ln\tfrac{1}{\delta}}$$

UniformBandit Algorithm

NaiveBandit from [Even-Dar et. al., 2002]

1. Pull each arm **w** times (uniform pulling).
2. Return arm with best average reward.



s
a_1
a_2
a_k
…
r_{11} r_{12} … r_{1w}
r_{21} r_{22} … r_{2w}
r_{k1} r_{k2} … r_{kw}

How large must w be to provide a PAC guarantee?

UniformBandit PAC Bound

With a bit of algebra and Chernoff bound we get:

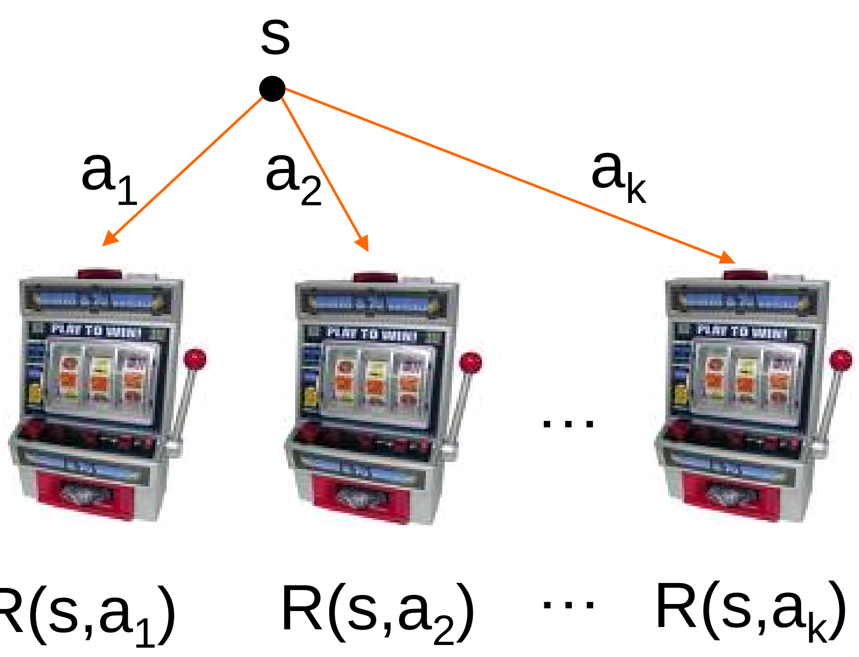
If $w \geq \left(\frac{R_{\max}}{\varepsilon}\right)^2 \ln\frac{k}{\delta}$ for all arms simultaneously

$$\left| E[R(s,a_i)] - \tfrac{1}{w}\sum_{j=1}^{w} r_{ij} \right| \leq \varepsilon$$

with probability at least $1-\delta$

- That is, estimates of all actions are ε– accurate with probability at least 1- δ
- Thus selecting estimate with highest value is approximately optimal with high probability, or PAC

# Simulator Calls for UniformBandit



- Total simulator calls for PAC: $k \cdot w = O\left(\frac{k}{\varepsilon^2} \ln \frac{k}{\delta} \right)$

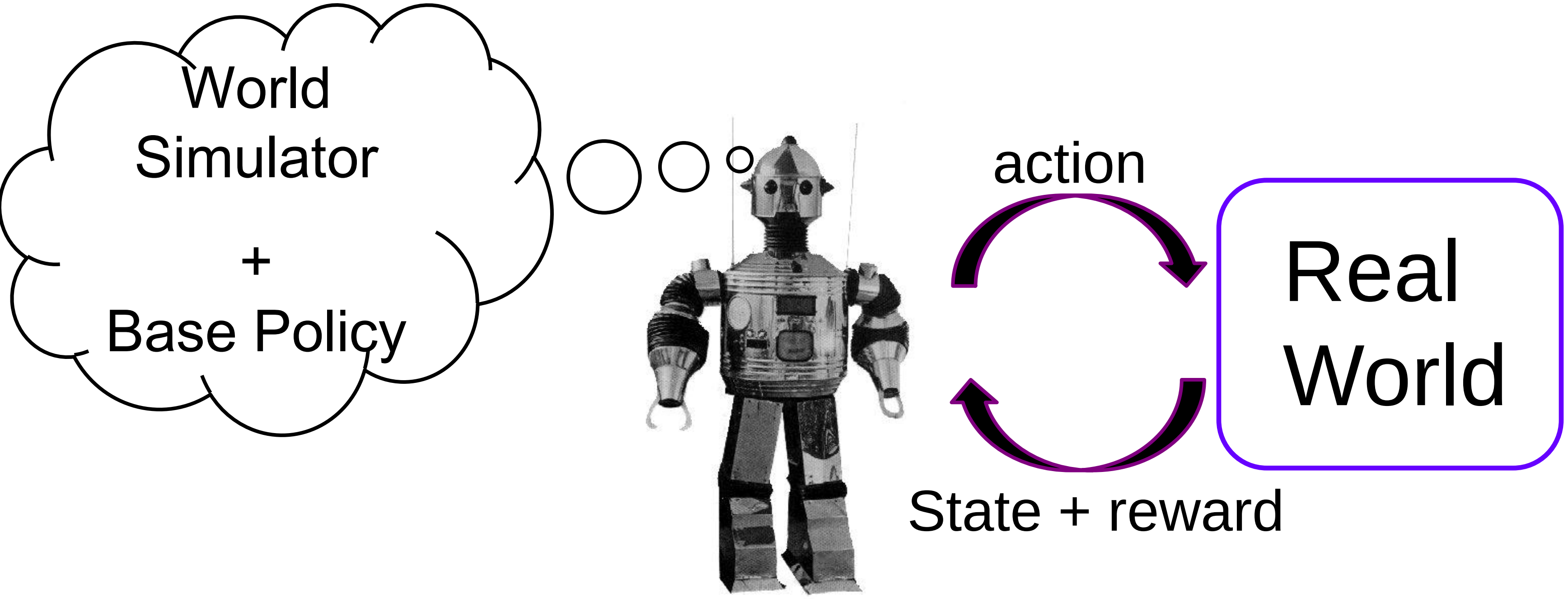
- Can get rid of ln(k) term with more complex algorithm [Even-Dar et. al., 2002].

Outline

- Preliminaries: Markov Decision Processes
- What is Monte-Carlo Planning?
- Non-Adaptive Monte-Carlo
  - Single State Case (PAC Bandit)
  - Policy rollout
  - Sparse Sampling
- Adaptive Monte-Carlo
  - Single State Case (UCB Bandit)
  - UCT Monte-Carlo Tree Search

Policy Improvement via Monte-Carlo

- Now consider a 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?



Policy Improvement Theorem

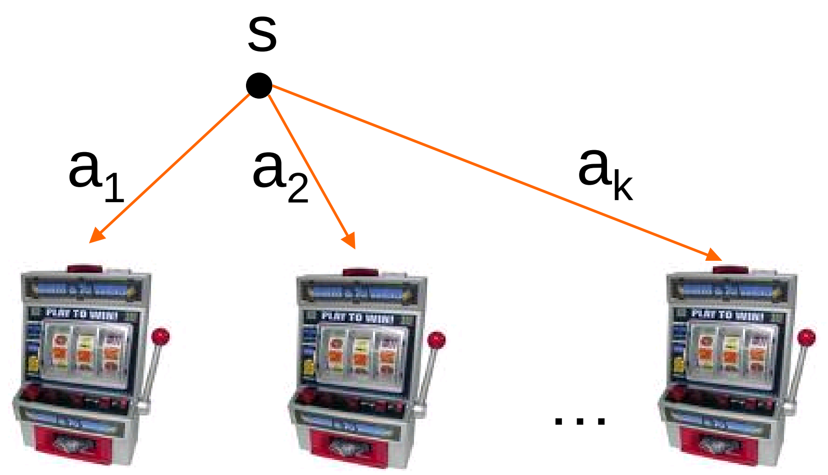
- The h-horizon Q-function $Q_\pi(s,a,h)$ is defined as: expected total discounted reward of starting in state s, taking action a, and then following policy π for h-1 steps

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

- **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
  - Can we use the bandit idea to solve this?

Policy Improvement via Bandits



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

- **Idea:** define a stochastic function **SimQ(s,a,π,h)** that we can implement and whose expected value is $Q_\pi(s,a,h)$

- Use Bandit algorithm to PAC select improved action

How to implement SimQ?

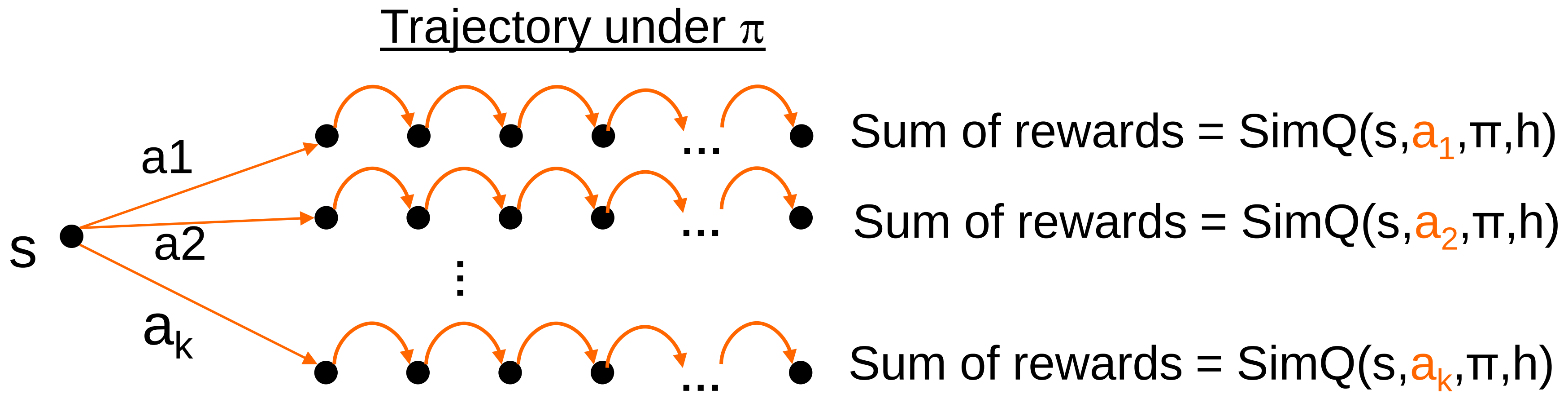
Policy Improvement via Bandits

```
SimQ(s,a,π,h)
    r = R(s,a)              } simulate a in s
    s = T(s,a)              }
    for i = 1 to h-1                          }
        r = r + $\beta^i$ R(s, π(s))          } simulate h-1 steps
        s = T(s, π(s))                        } 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 $Q_\pi(s,a,h)$

Policy Improvement via Bandits

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

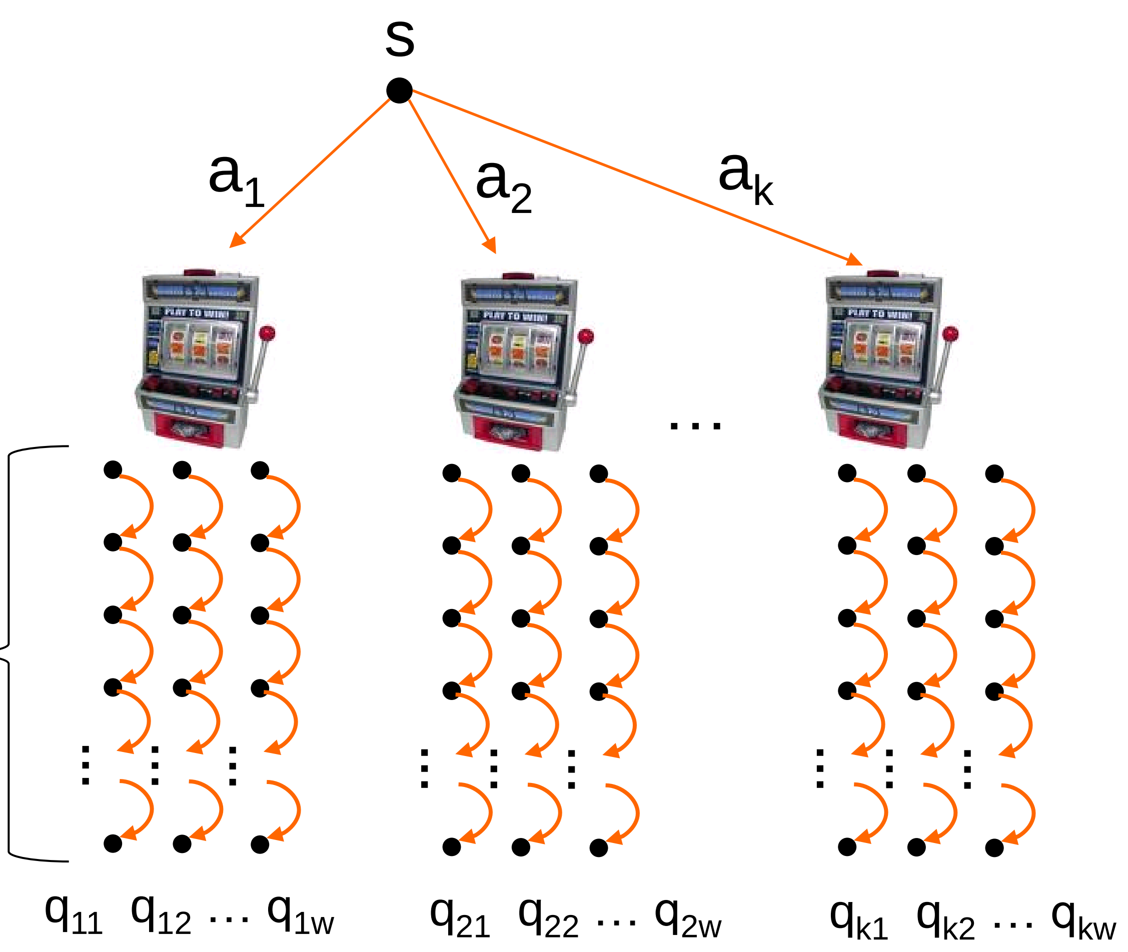

Trajectory under π
s
a1
a2
a_k
Sum of rewards = SimQ(s,a_1,π,h)
Sum of rewards = SimQ(s,a_2,π,h)
Sum of rewards = SimQ(s,a_k,π,h)

Policy Rollout Algorithm

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

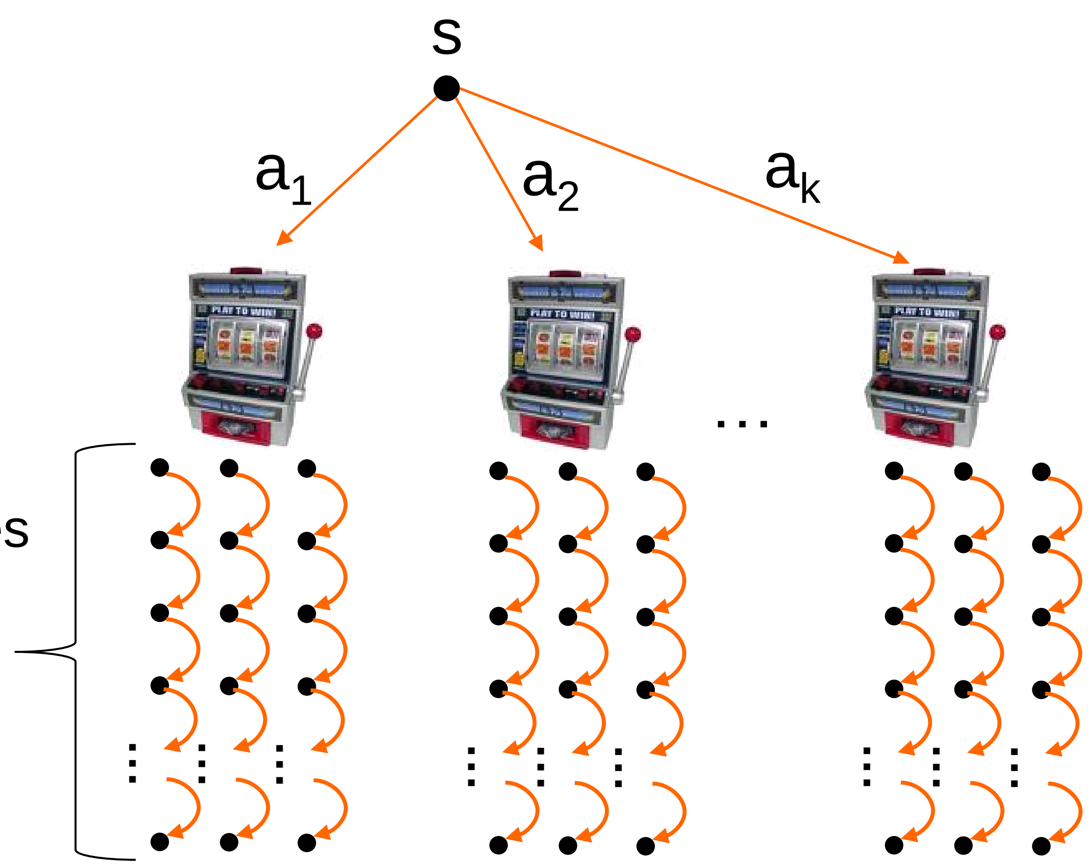
SimQ(s,a_i,π,h) trajectories

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


s
a_1
a_2
a_k
…
q_{11} q_{12} … q_{1w}
q_{21} q_{22} … q_{2w}
q_{k1} q_{k2} … q_{kw}

Samples of SimQ(s,a_i,π,h)

Policy Rollout: # of Simulator Calls

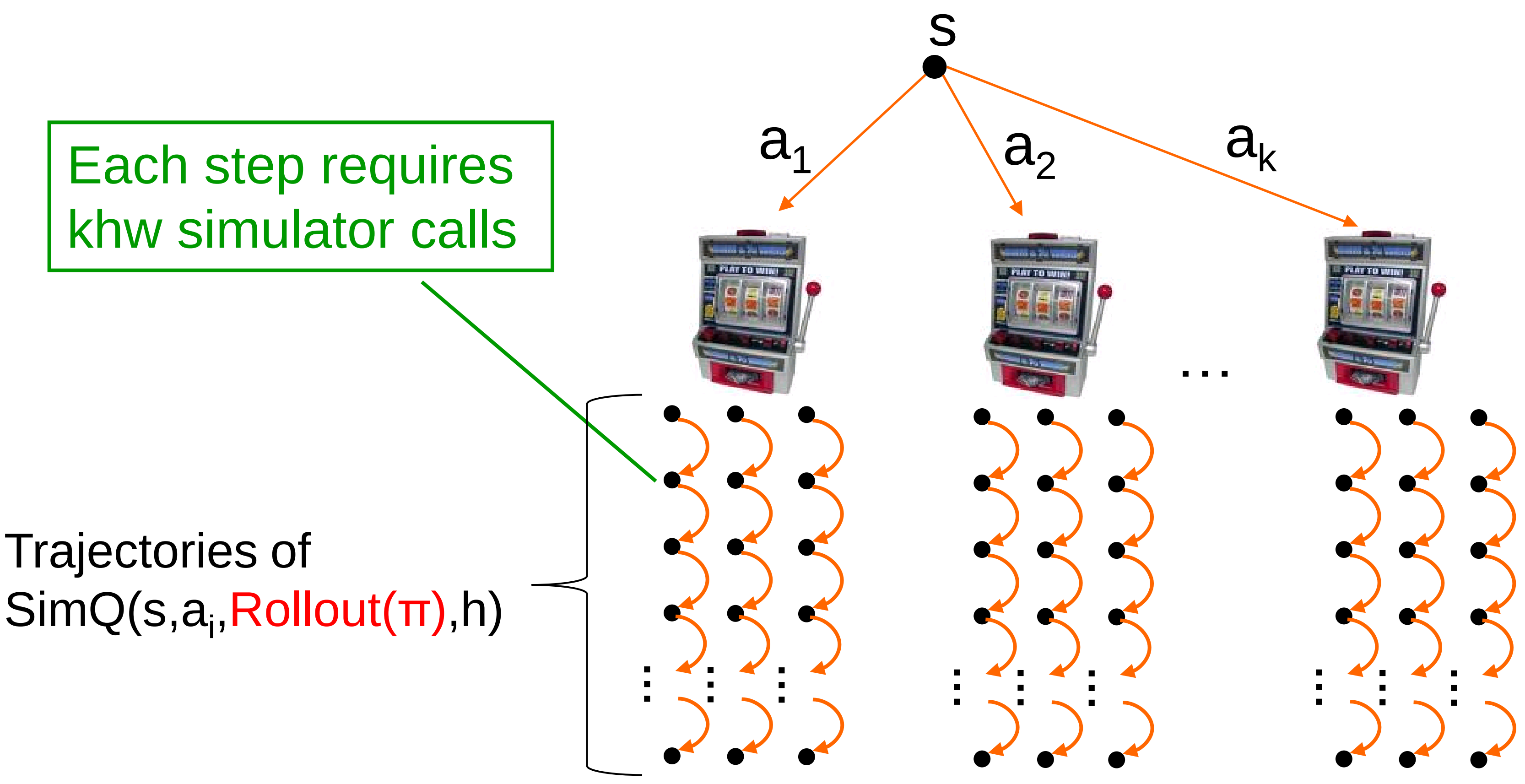


SimQ(s,a_i,π,h) trajectories

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

Multi-Stage Rollout



s
a_1
a_2
a_k
…
Each step requires khw simulator calls
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
- Often observe substantial improvement, e.g.
  - Compiler instruction scheduling
  - Backgammon
  - Network routing
  - Combinatorial optimization
  - Game of GO
  - Solitaire

Example: Rollout for Thoughtful 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

Outline

- Preliminaries: Markov Decision Processes
- What is Monte-Carlo Planning?
- Uniform Monte-Carlo
  - Single State Case (UniformBandit)
  - Policy rollout
  - Sparse Sampling
- Adaptive Monte-Carlo
  - Single State Case (UCB Bandit)
  - UCT Monte-Carlo Tree Search

Sparse Sampling

- Rollout does not guarantee optimality or near optimality

- Can we develop simulation-based methods that give us near optimal policies?
  - With computation that doesn’t depend on number of states!

- In deterministic games and problems it is common to build a look-ahead tree at a state to determine best action
  - Can we generalize this to general MDPs?

- **Sparse Sampling** is one such algorithm
  - Strong theoretical guarantees of near optimality

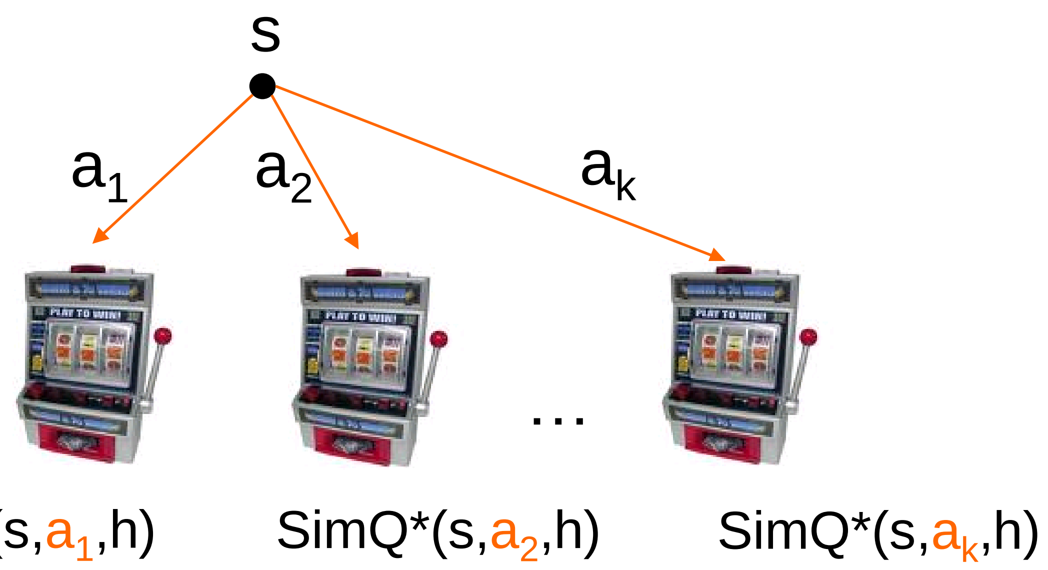
MDP Basics

- Let V*(s,h) be the optimal value function of MDP
- Define $Q^*(s,a,h) = E[R(s,a) + V^*(T(s,a),h-1)]$
  - Optimal h-horizon value of action *a* at state *s.*
  - R(s,a) and T(s,a) return random reward and next state
- **Optimal Policy:** $\pi^*(x) = \mathrm{argmax}_a\, Q^*(x,a,h)$
- What if we knew V*?
  - Can apply bandit algorithm to select action that approximately maximizes Q*(s,a,h)

Bandit Approach Assuming V*

SimQ*(s,a_i,h) =

$R(s, a_i) + V^*(T(s, a_i),h\text{-}1)$



```
SimQ*(s,a,h)
    s' = T(s,a)
    r = R(s,a)
    Return  r + V*(s',h-1)
```

- Expected value of SimQ*(s,a,h) is Q*(s,a,h)
  - Use UniformBandit to select approximately optimal action

But we don't know V*

- To compute SimQ*(s,a,h) need V*(s',h-1) for any s'

- Use recursive identity (Bellman's equation):
  - $V^*(s,h-1) = \max_a Q^*(s,a,h-1)$

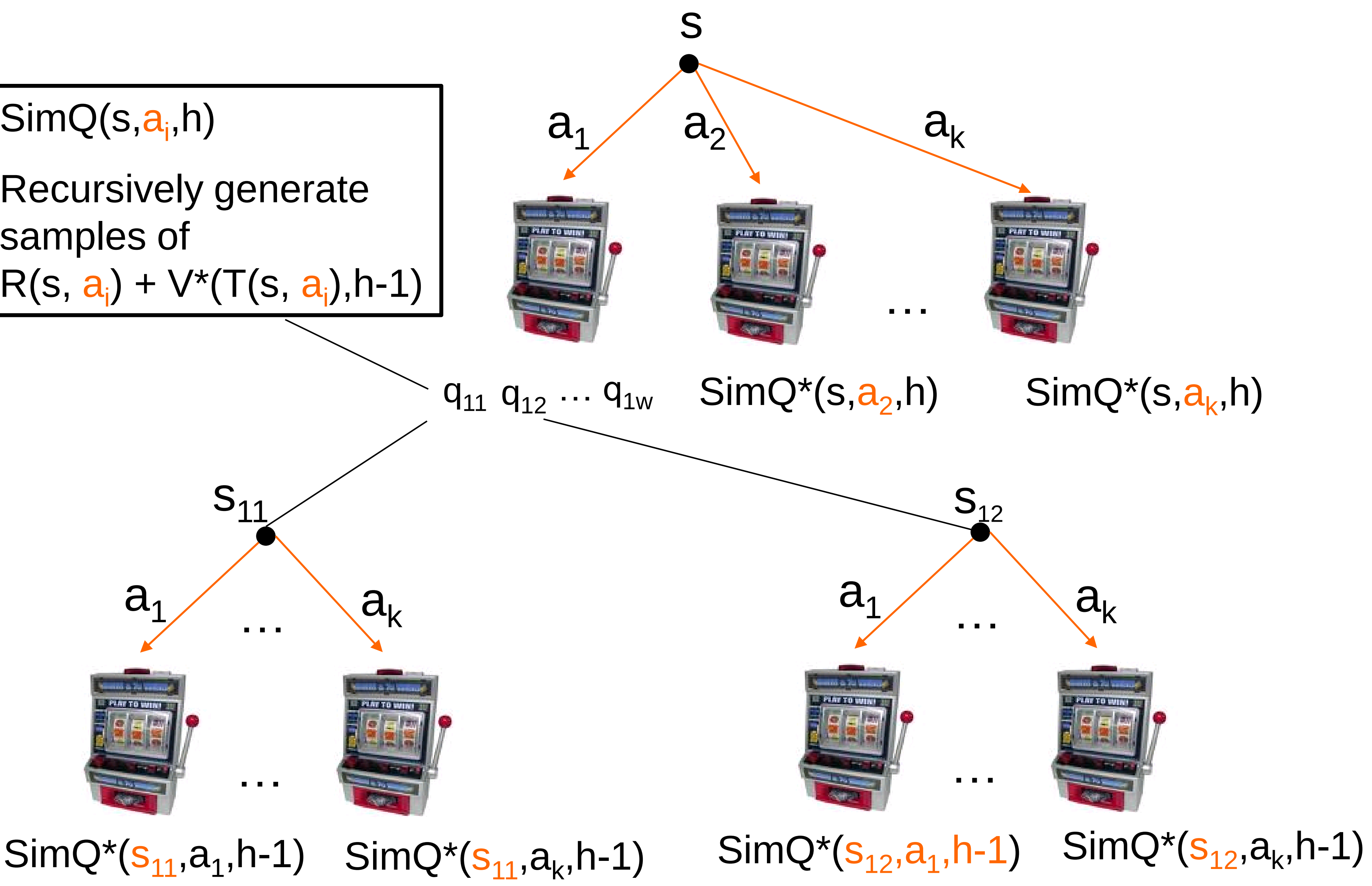
- **Idea:** Can recursively estimate V*(s,h-1) by running h-1 horizon bandit based on SimQ*

- **Base Case:** V*(s,0) = 0, for all s

Recursive UniformBandit

SimQ(s,a_i,h)

Recursively generate samples of
R(s, a_i) + V*(T(s, a_i),h-1)



s
a_1
a_2
a_k
…
q_{11} q_{12} … q_{1w}
SimQ*(s,a_2,h)
SimQ*(s,a_k,h)
s_{11}
s_{12}
a_1
a_k
…
SimQ*(s_{11},a_1,h-1)
SimQ*(s_{11},a_k,h-1)
SimQ*(s_{12},a_1,h-1)
SimQ*(s_{12},a_k,h-1)

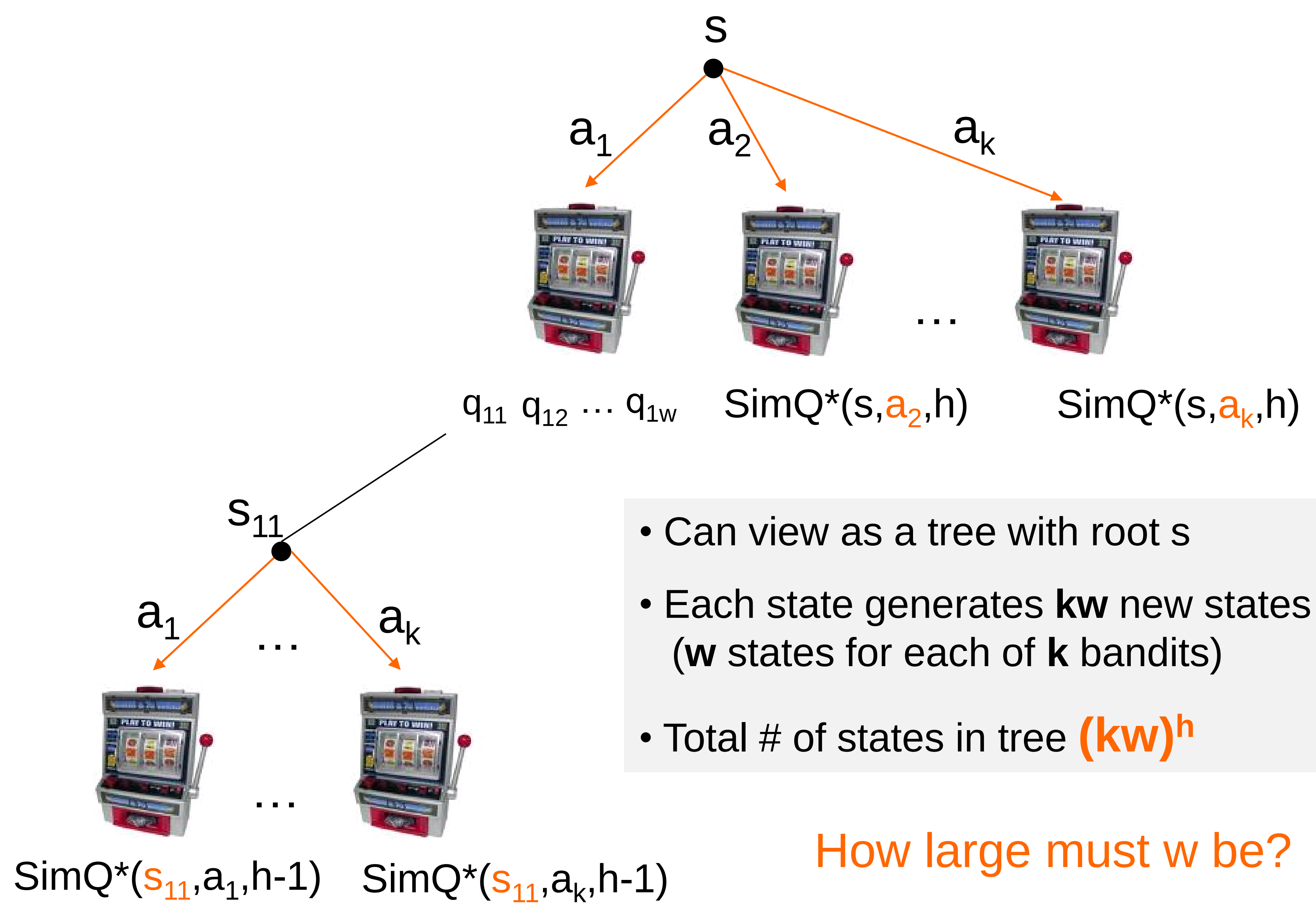
Sparse Sampling [Kearns et. al. 2002]

This recursive UniformBandit is called Sparse Sampling

Return value estimate V*(s,h) of state s and estimated optimal action a*

```
**SparseSampleTree**(*s*,*h*,*w*)
For each action $a$ in $s$
    $Q^*(s,a,h) = 0$
    For $i = 1$ to $w$
        Simulate taking $a$ in $s$ resulting in $s_i$ and reward $r_i$
        $[V^*(s_i,h),a^*]$ = **SparseSample**$(s_i,h-1,w)$
        $Q^*(s,a,h) = Q^*(s,a,h) + r_i + V^*(s_i,h)$
    $Q^*(s,a,h) = Q^*(s,a,h) / w$   ;; estimate of $Q^*(s,a,h)$
$V^*(s,h) = \max_a Q^*(s,a,h)$        ;; estimate of $V^*(s,h)$
$a^* = \text{argmax}_a Q^*(s,a,h)$
Return $[V^*(s,h), a^*]$
```

# of Simulator Calls


s
a_1
a_2
a_k
…
q_{11} q_{12} … q_{1w}
SimQ*(s,a_2,h)
SimQ*(s,a_k,h)
s_{11}
a_1
…
a_k
…
SimQ*(s_{11},a_1,h-1)
SimQ*(s_{11},a_k,h-1)

- Can view as a tree with root s
- Each state generates **kw** new states (**w** states for each of **k** bandits)
- Total # of states in tree **$(kw)^h$**

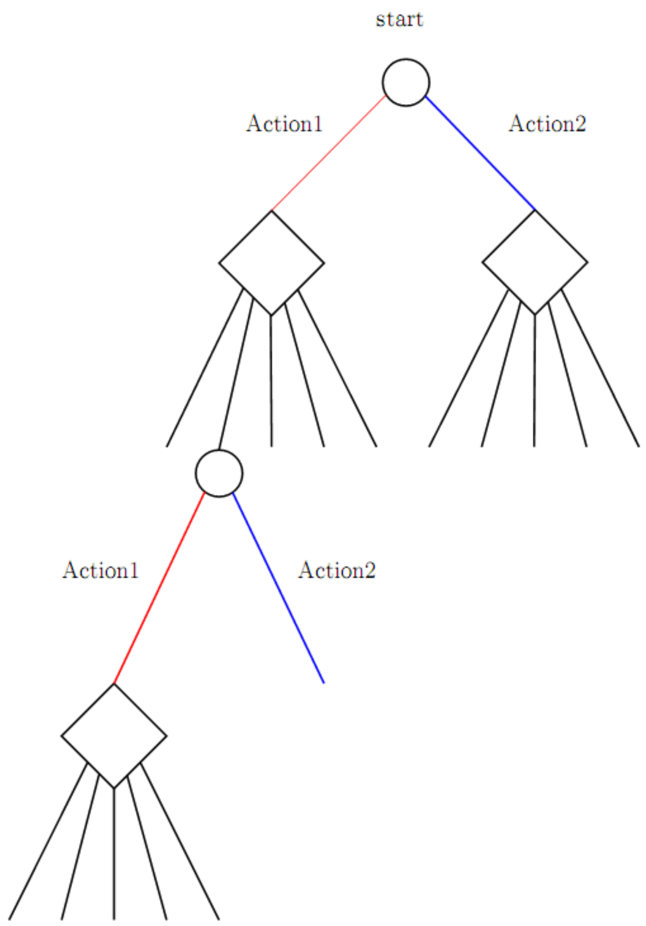
How large must w be?

Sparse Sampling

- For a given desired accuracy, how large should sampling width and depth be?
  - Answered: [Kearns et. al., 2002]
- **Good news:** can achieve near optimality for value of w independent of state-space size!
  - First near-optimal general MDP planning algorithm whose runtime didn't depend on size of state-space
- **Bad news:** the theoretical values are typically still intractably large---also exponential in **h**
- **In practice:** use small **h** and use heuristic at leaves (similar to minimax game-tree search)

Uniform vs. Adaptive Bandits

- Sparse sampling wastes time on bad parts of tree
  - Devotes equal resources to each state encountered in the tree
  - Would like to focus on most promising parts of tree
- But how to control exploration of new parts of tree vs. exploiting promising parts?
- Need adaptive bandit algorithm that explores more effectively

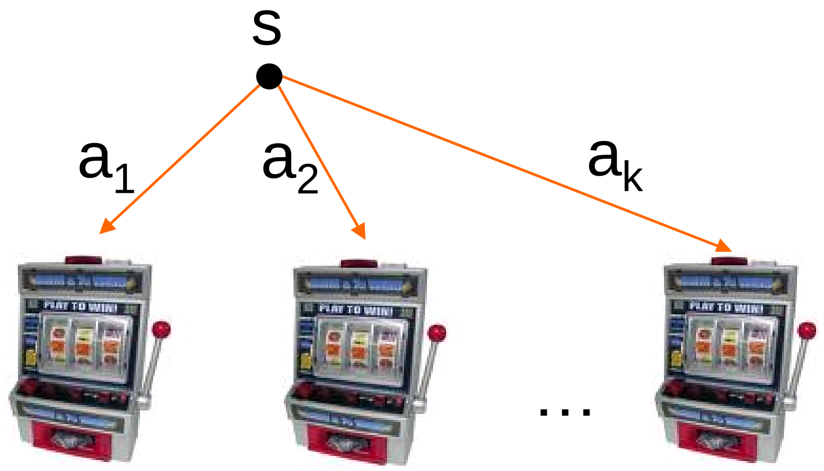


Outline

- Preliminaries: Markov Decision Processes
- What is Monte-Carlo Planning?
- Uniform Monte-Carlo
  - Single State Case (UniformBandit)
  - Policy rollout
  - Sparse Sampling
- Adaptive Monte-Carlo
  - Single State Case (UCB Bandit)
  - UCT Monte-Carlo Tree Search

Regret Minimization Bandit Objective

- **Problem:** find arm-pulling strategy such that the expected total reward at time n is close to the best possible (i.e. pulling the best arm always)
  - UniformBandit is poor choice --- waste time on bad arms
  - Must balance **exploring** machines to find good payoffs and **exploiting** current knowledge



UCB Adaptive Bandit Algorithm
[Auer, Cesa-Bianchi, & Fischer, 2002]

- Q(a) : average payoff for action a based on current experience
- n(a) : number of pulls of arm a
- Action choice by UCB after n pulls:

$$a^* = \arg\max_a Q(a) + \sqrt{\frac{2\ln n}{n(a)}}$$

Assumes payoffs in [0,1]

- **Theorem**: The expected regret after *n* arm pulls compared to optimal behavior is bounded by O(log *n*)
- No algorithm can achieve a better loss rate

UCB Algorithm [Auer, Cesa-Bianchi, & Fischer, 2002]

$$a^* = \arg\max_a Q(a) + \sqrt{\frac{2\ln n}{n(a)}}$$

Value Term:
favors actions that looked good historically

Exploration Term:
actions get an exploration bonus that grows with ln(n)

Expected number of pulls of sub-optimal arm **a** is bounded by:

$$\frac{8}{\Delta_a^2}\ln n$$

where Δ_a is regret of arm **a**

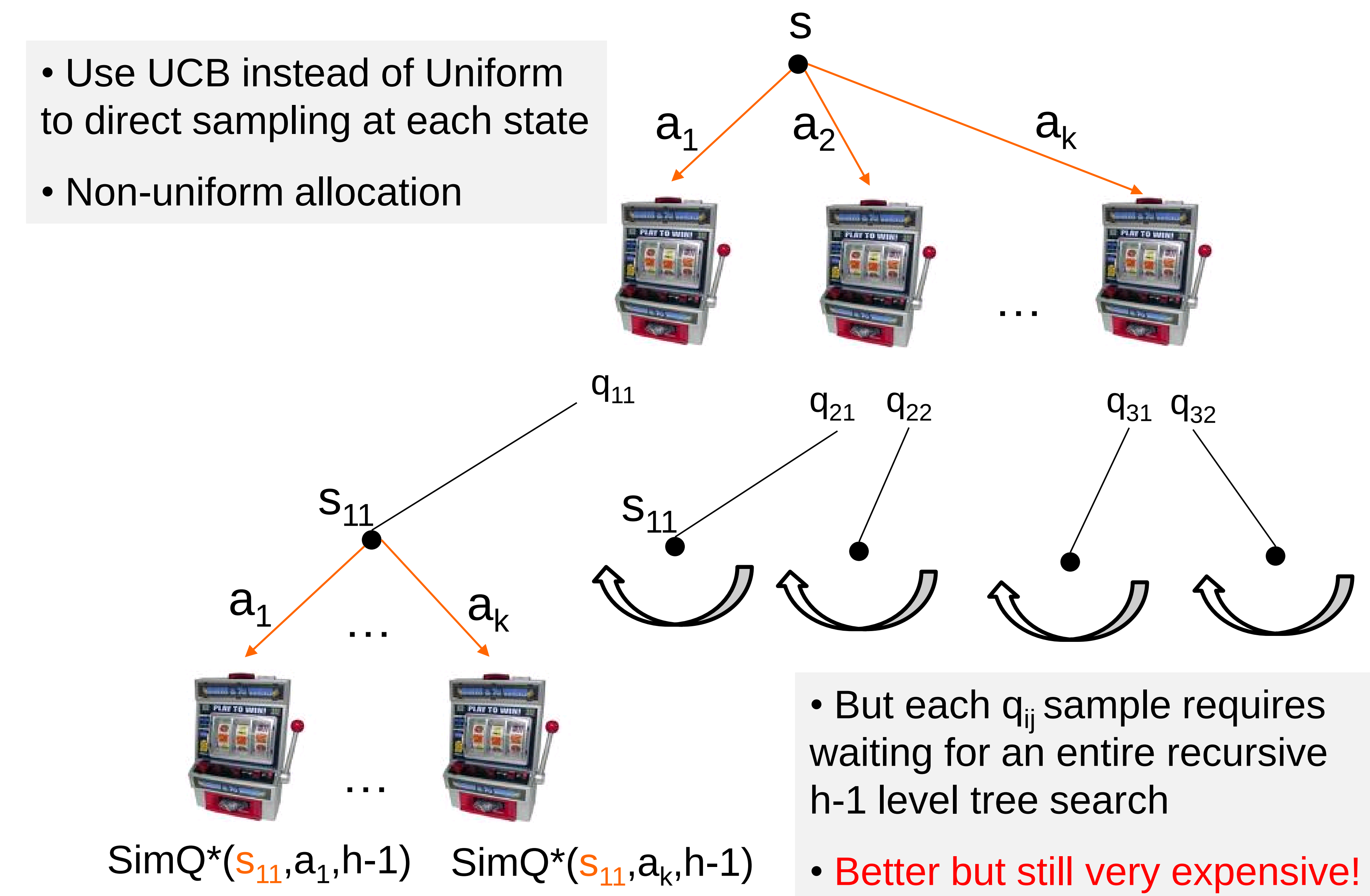
Doesn’t waste much time on sub-optimal arms unlike uniform!

UCB for Multi-State MDPs

- UCB-Based Policy Rollout:
  - Use UCB to select actions instead of uniform
- UCB-Based Sparse Sampling
  - Use UCB to make sampling decisions at internal tree nodes

UCB-based Sparse Sampling [Chang et. al. 2005]

- Use UCB instead of Uniform to direct sampling at each state
- Non-uniform allocation



- But each q_{ij} sample requires waiting for an entire recursive h-1 level tree search
- Better but still very expensive!

Outline

- Preliminaries: Markov Decision Processes
- What is Monte-Carlo Planning?
- Uniform Monte-Carlo
  - Single State Case (UniformBandit)
  - Policy rollout
  - Sparse Sampling
- Adaptive Monte-Carlo
  - Single State Case (UCB Bandit)
  - UCT Monte-Carlo Tree Search

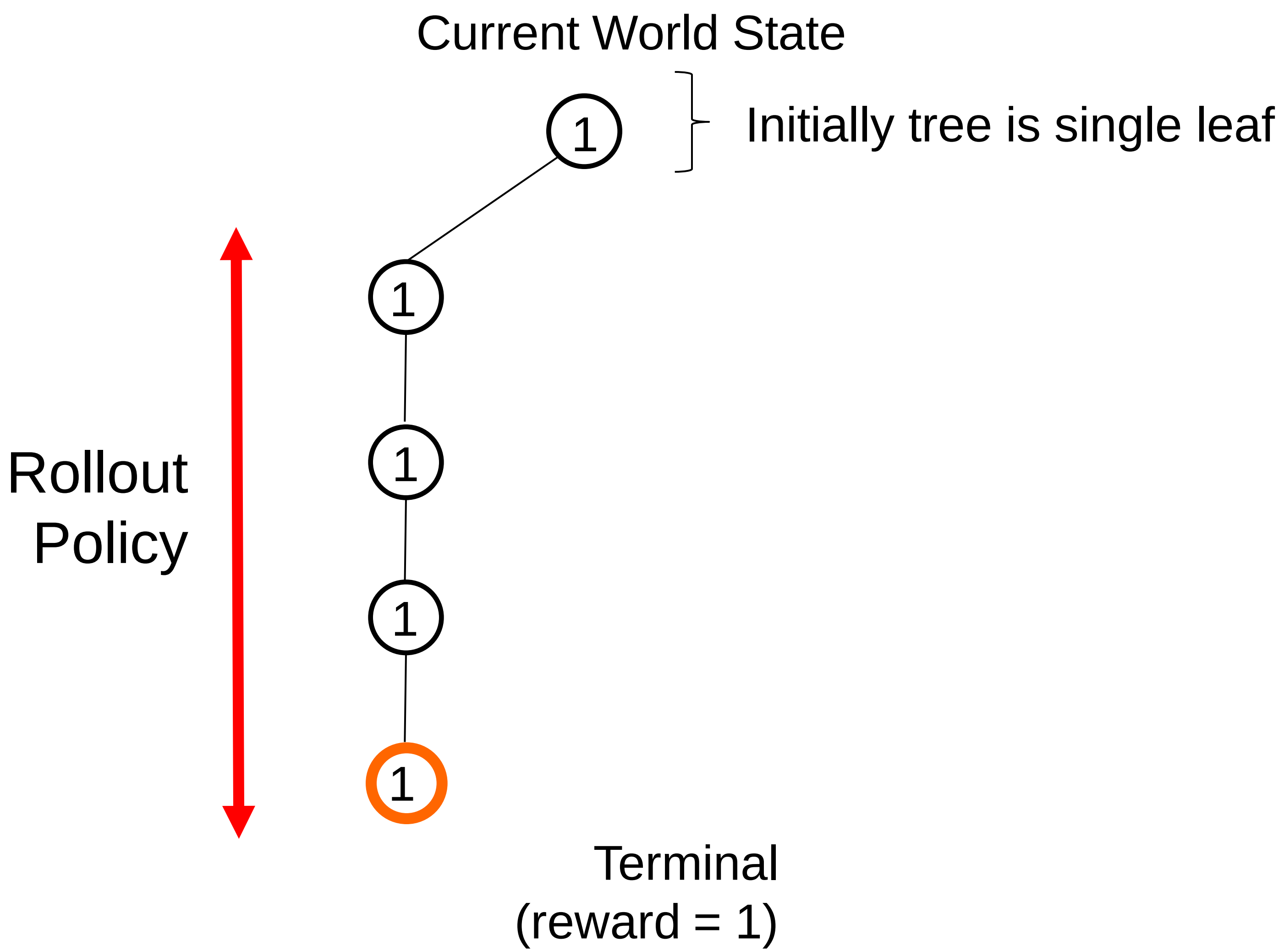
UCT Algorithm [Kocsis & Szepesvari, 2006]

- Instance of Monte-Carlo Tree Search
  - Applies principle of UCB
  - Some nice theoretical properties
  - Much better anytime behavior than sparse sampling
  - Major advance in computer Go

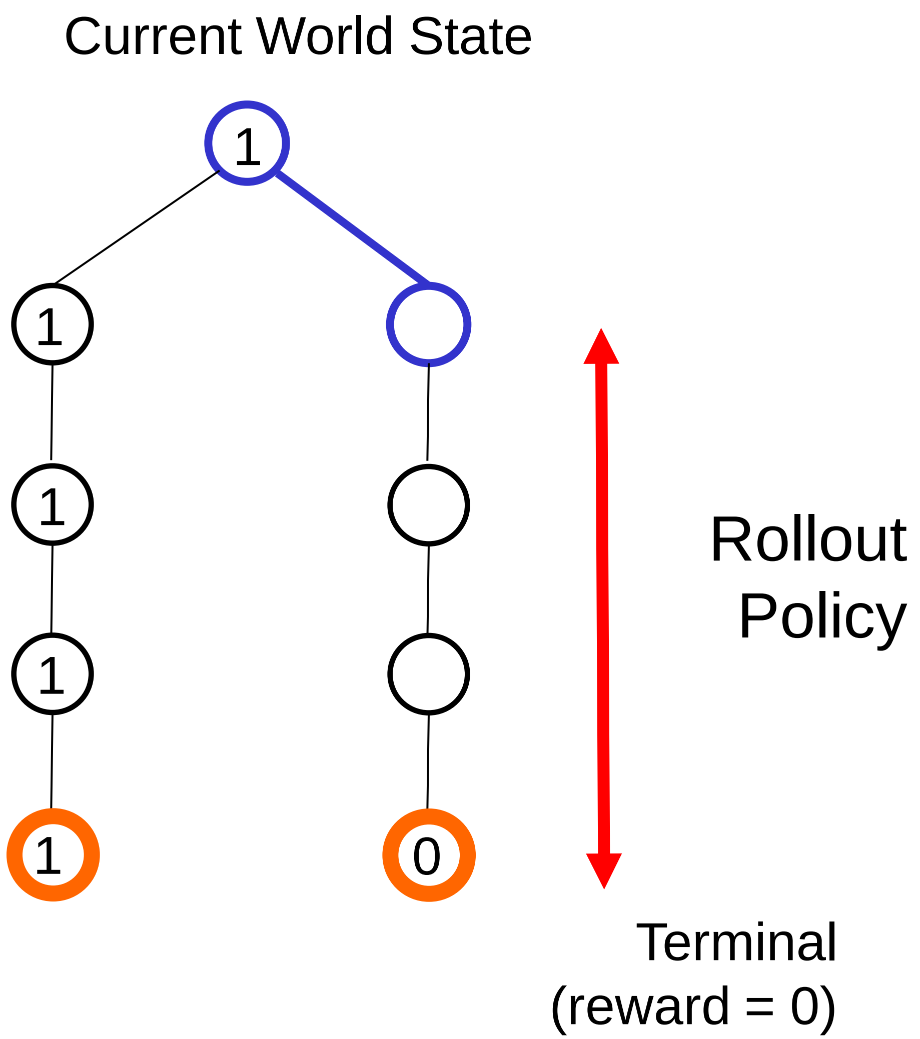
- Monte-Carlo Tree Search
  - Repeated Monte Carlo simulation of a rollout policy
  - Each rollout adds one or more nodes to search tree

- Rollout policy depends on nodes already in tree

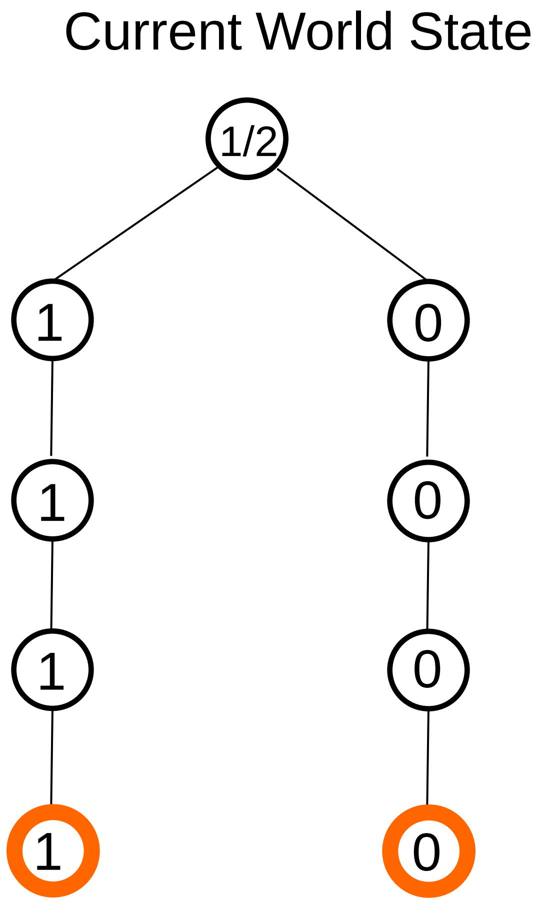
At a leaf node perform a random rollout



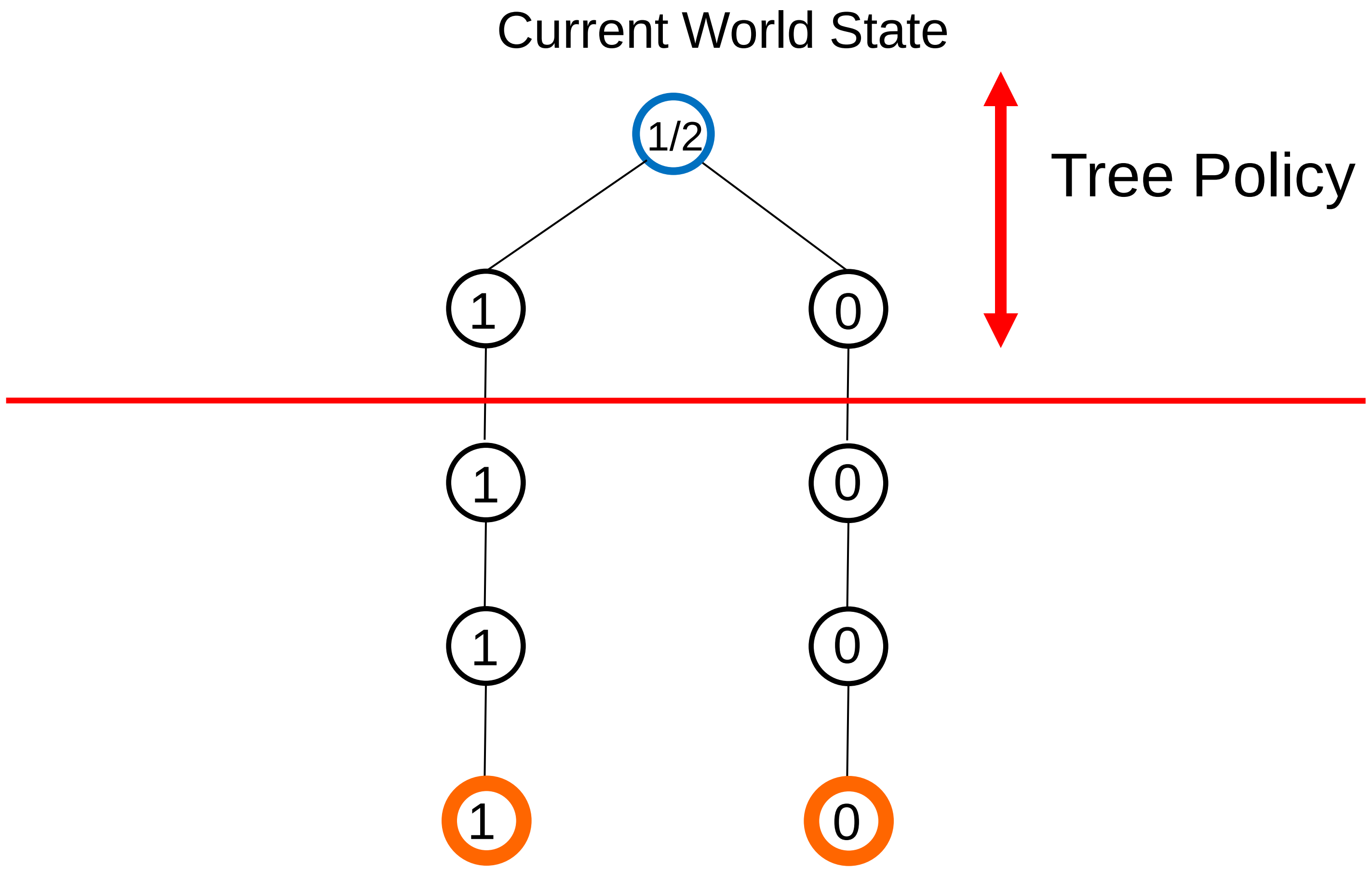
Must select each action at a node at least once



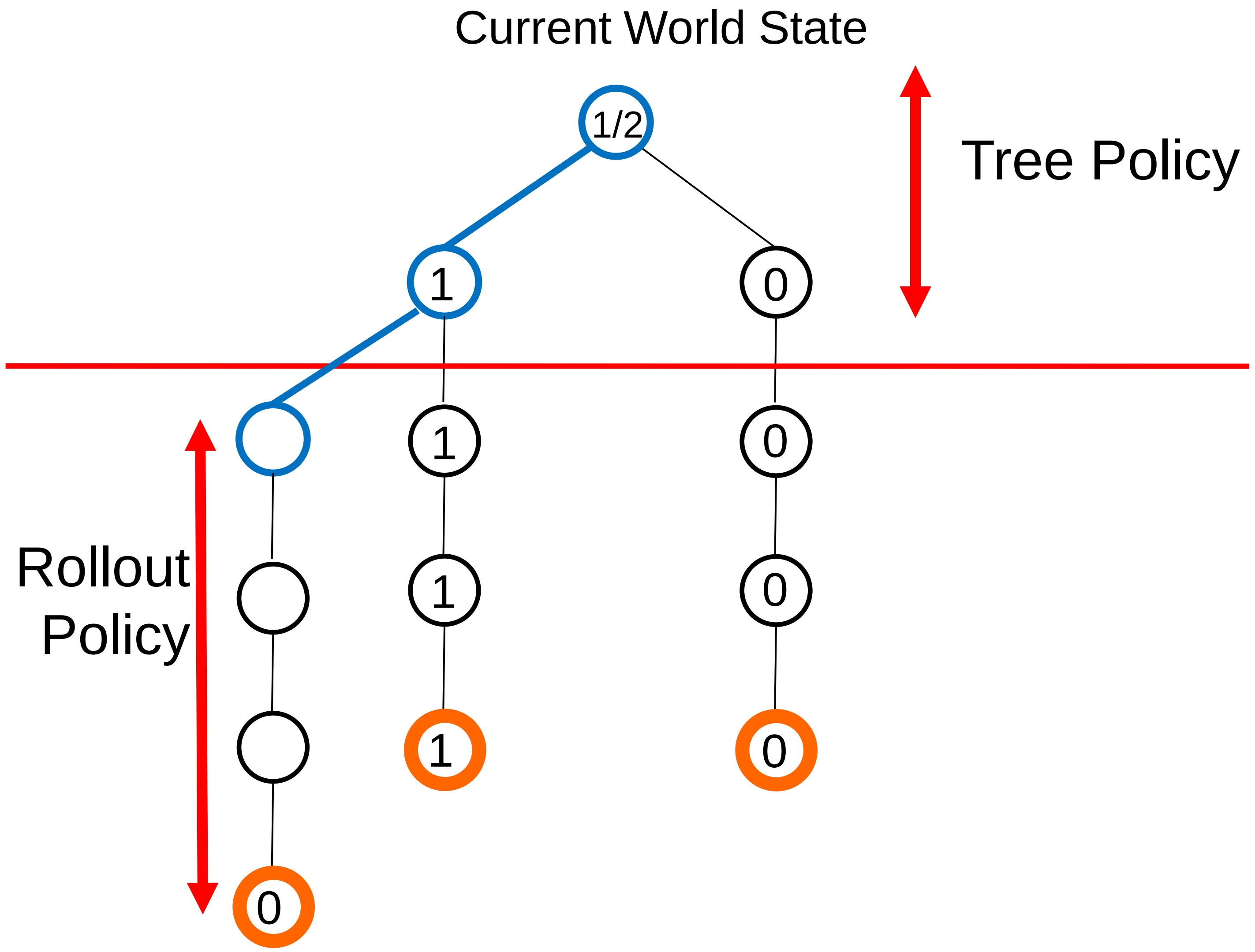
Must select each action at a node at least once



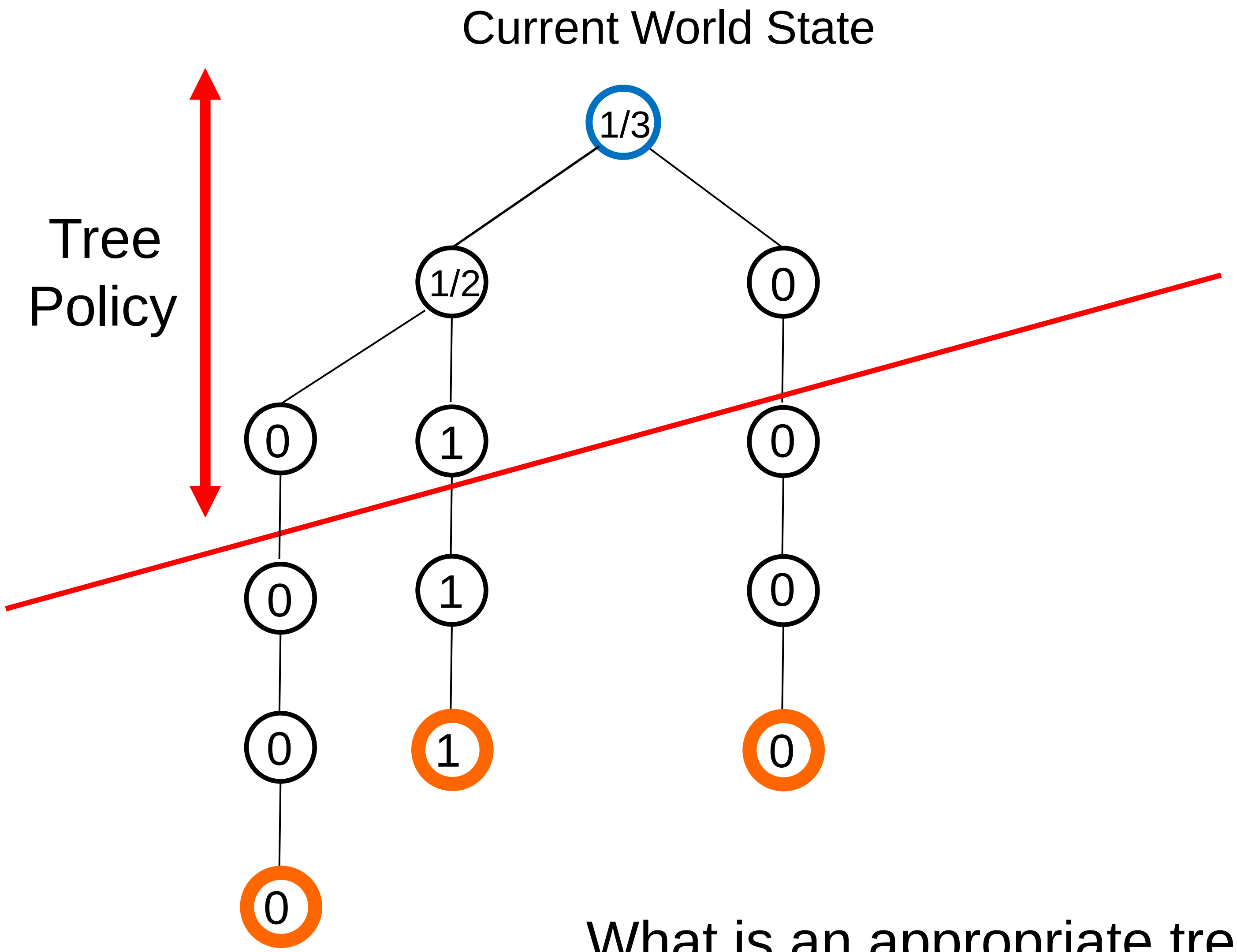
When all node actions tried once, select action according to tree policy



When all node actions tried once, select action according to tree policy



When all node actions tried once, select action according to tree policy



What is an appropriate tree policy? Rollout policy?

UCT Algorithm [Kocsis & Szepesvari, 2006]

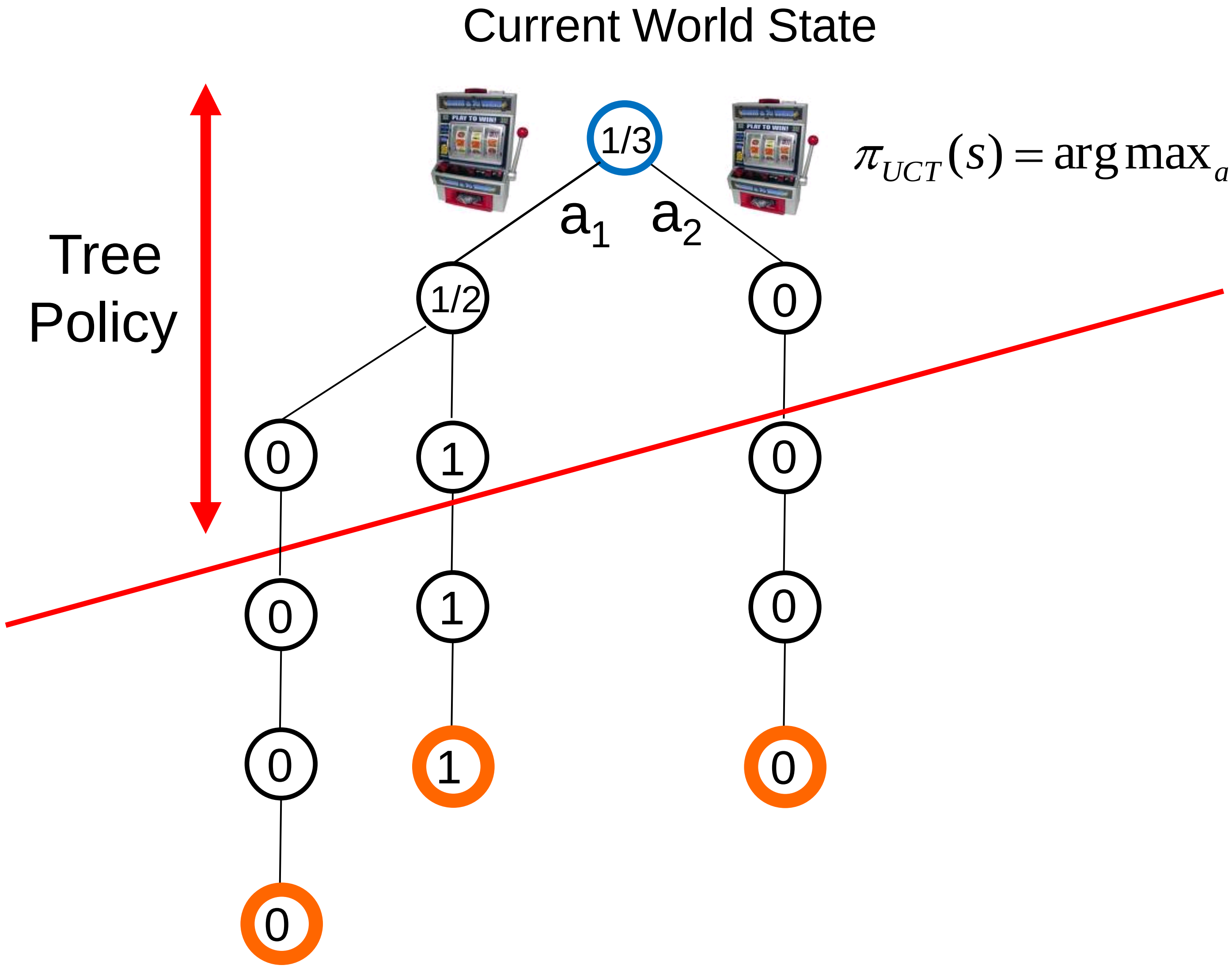
- Basic UCT uses random rollout policy

- Tree policy is based on UCB:
  - Q(s,a) : average reward received in current trajectories after taking action a in state s
  - n(s,a) : number of times action a taken in s
  - n(s) : number of times state s encountered

$$\pi_{UCT}(s) = \arg\max_a Q(s,a) + \underset{\nwarrow}{c}\sqrt{\frac{\ln n(s)}{n(s,a)}}$$

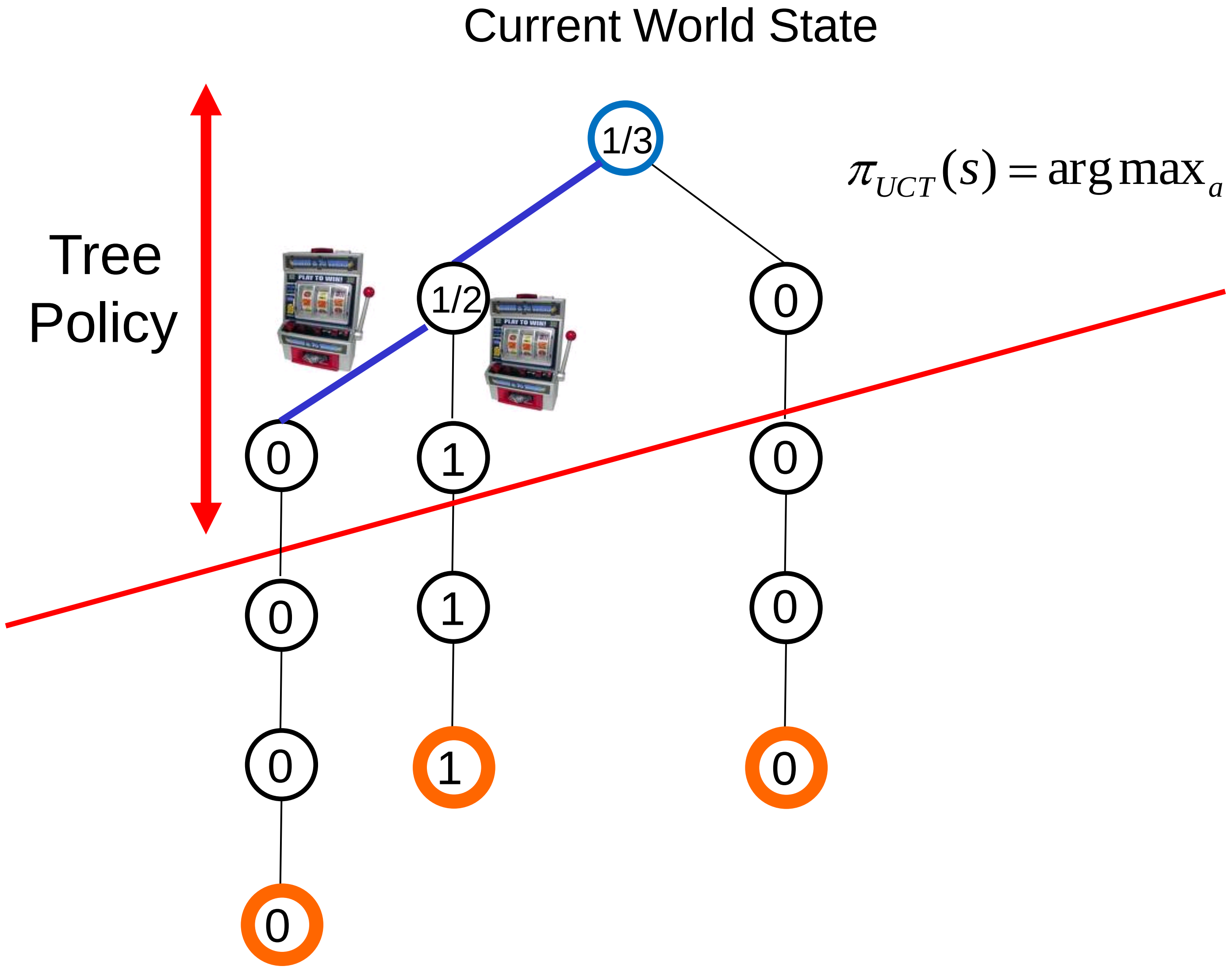
Theoretical constant that must
be selected empirically in practice

When all node actions tried once, select action according to tree policy



$$\pi_{UCT}(s) = \arg\max_a Q(s,a) + c\sqrt{\frac{\ln n(s)}{n(s,a)}}$$

When all node actions tried once, select action according to tree policy



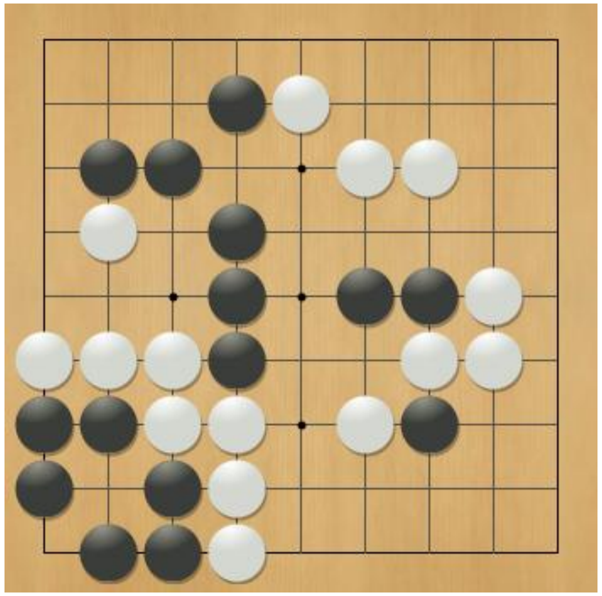
$$\pi_{UCT}(s) = \arg\max_a Q(s,a) + c\sqrt{\frac{\ln n(s)}{n(s,a)}}$$

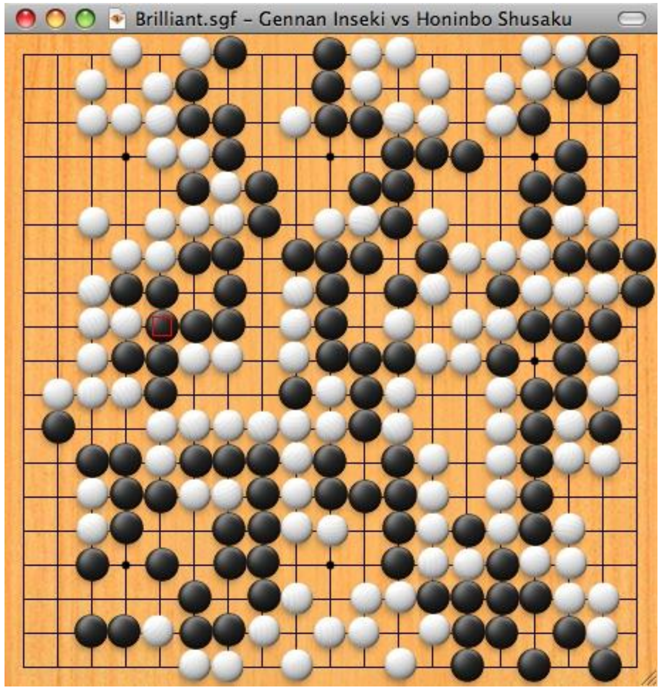
UCT Recap

- To select an action at a state s
  - Build a tree using N iterations of monte-carlo tree search
    - Default policy is uniform random
    - Tree policy is based on UCB rule
  - Select action that maximizes Q(s,a)
    (note that this final action selection does not take the exploration term into account, just the Q-value estimate)

- The more simulations the more accurate

Computer Go


9x9 (smallest board)



19x19 (largest board)

- “Task Par Excellence for AI” (Hans Berliner)
- “New Drosophila of AI” (John McCarthy)
- “Grand Challenge Task” (David Mechner)

A Brief History of Computer Go

- *2005*: Computer Go is impossible!
- *2006*: UCT invented and applied to 9x9 Go *(Kocsis, Szepesvari; Gelly et al.)*
- *2007*: Human master level achieved at 9x9 Go *(Gelly, Silver; Coulom)*
- *2008*: Human grandmaster level achieved at 9x9 Go *(Teytaud et al.)*

Computer GO Server: 1800 ELO → 2600 ELO

Other Successes

- Klondike Solitaire (wins 40% of games)
- General Game Playing Competition
- Real-Time Strategy Games
- Combinatorial Optimization

- List is growing

- Usually extend UCT is some ways

Some Improvements

- Use domain knowledge to handcraft a more intelligent default policy than random
  - E.g. don’t choose obviously stupid actions

- Learn a heuristic function to evaluate positions
  - Use the heuristic function to initialize leaf nodes (otherwise initialized to zero)

Summary

- When you have a tough planning problem and a simulator
  - Try Monte-Carlo planning
- Basic principles derive from the multi-arm bandit
- Policy Rollout is a great way to exploit existing policies and make them better
- If a good heuristic exists, then shallow sparse sampling can give good gains
- UCT is often quite effective especially when combined with domain knowledge