Monte-Carlo Planning Look Ahead Trees

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 Look-Ahead Trees
  - Sparse Sampling
  - Sparse Sampling via Recursive Bandits
  - UCT and variants

Sparse Sampling

- Rollout and policy switching do not guarantee optimality nor near optimality
  - Guarantee relative performance to base policies

- Can we develop Monte-Carlo methods that give us near optimal policies?
  - With computation that does NOT depend on number of states!
  - This was an open problem until late 90's.

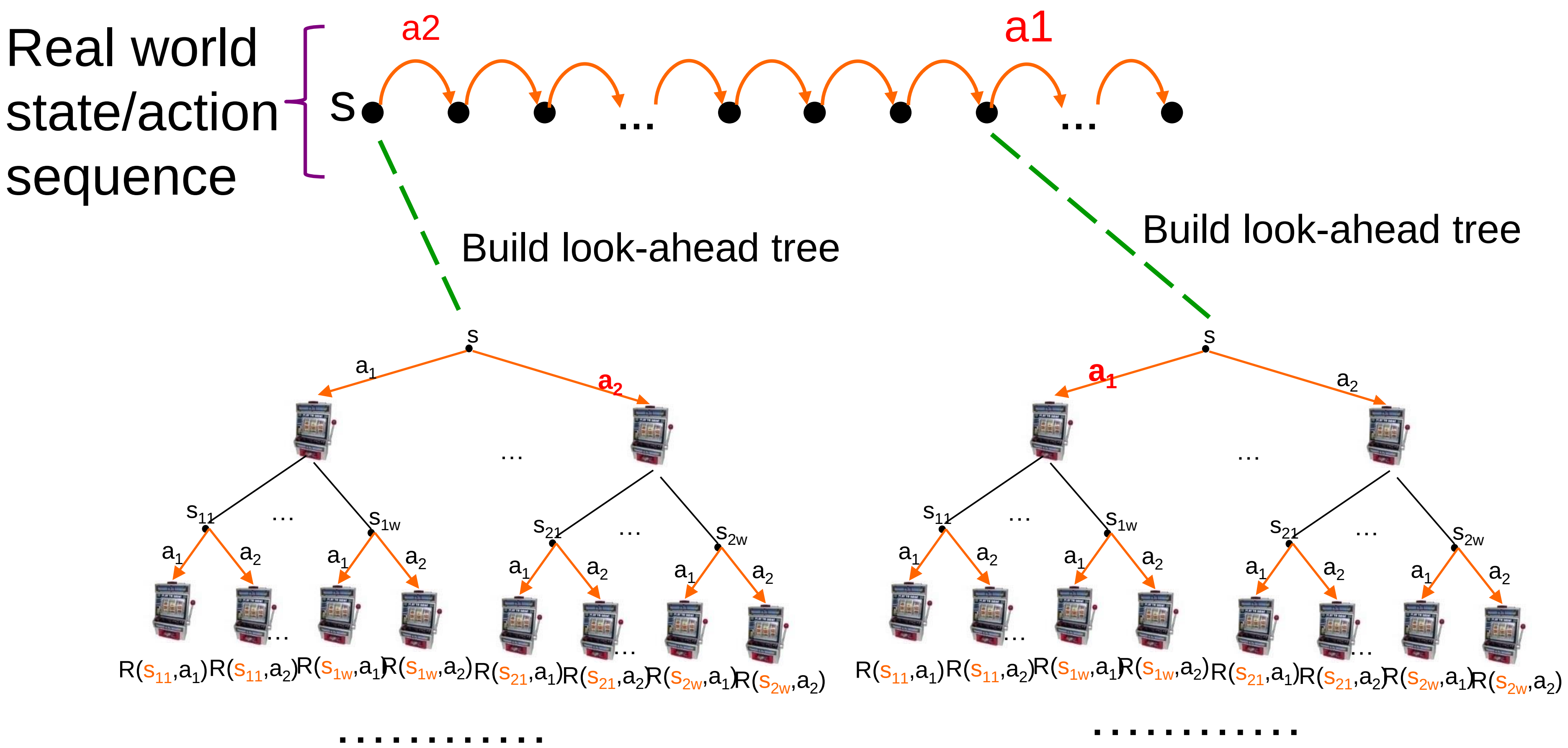
- In deterministic games and search problems it is common to build a look-ahead tree at a state to select best action
  - Can we generalize this to general stochastic MDPs?

Online Planning with Look-Ahead Trees

- At each state we encounter in the environment we build a **look-ahead tree of depth *h*** and use it to estimate optimal Q-values of each action
  - Select action with highest Q-value estimate

- *s* = current state of environment
- Repeat
  - *T* = **BuildLookAheadTree**(*s*) ;; sparse sampling or UCT
    ;; tree provides Q-value estimates for root action
  - a = **BestRootAction**(*T*) ;; action with best Q-value
  - Execute action *a* in environment
  - *s* is the resulting state

Planning with Look-Ahead Trees



Sparse Sampling

- Again focus on finite-horizons
  - Arbitrarily good approximation for large enough horizon *h*
- *h-horizon optimal Q-function (denoted Q*)*
  - Value of taking a in s and following π^* for h-1 steps
  - $Q^*(s,a,h) = E[R(s,a) + \beta V^*(T(s,a),h-1)]$
- Key identity (Bellman's equations):
  - $V^*(s,h) = \max_a Q^*(s,a,h)$
  - $\pi^*(x) = \text{argmax}_a\, Q^*(x,a,h)$
- Sparse sampling estimates Q-values by building sparse expectimax tree

Sparse Sampling

- Will present two views of algorithm
  - The first is perhaps easier to digest and doesn't appeal to bandit algorithms
  - The second is more generalizable and can leverage advances in bandit algorithms

1. Approximation to the full expectimax tree

2. Recursive bandit algorithm

Expectimax Tree

- Key definitions:
  - $V^*(s,h) = \max_a Q^*(s,a,h)$
  - $Q^*(s,a,h) = E[R(s,a) + \beta V^*(T(s,a),h-1)]$
- Expand definitions recursively to compute V*(s,h)

$V^*(s,h) = \max_{a1} Q(s,a1,h)$

$= \max_{a1} E[R(s,a1) + \beta\, V^*(T(s,a1),h-1)]$

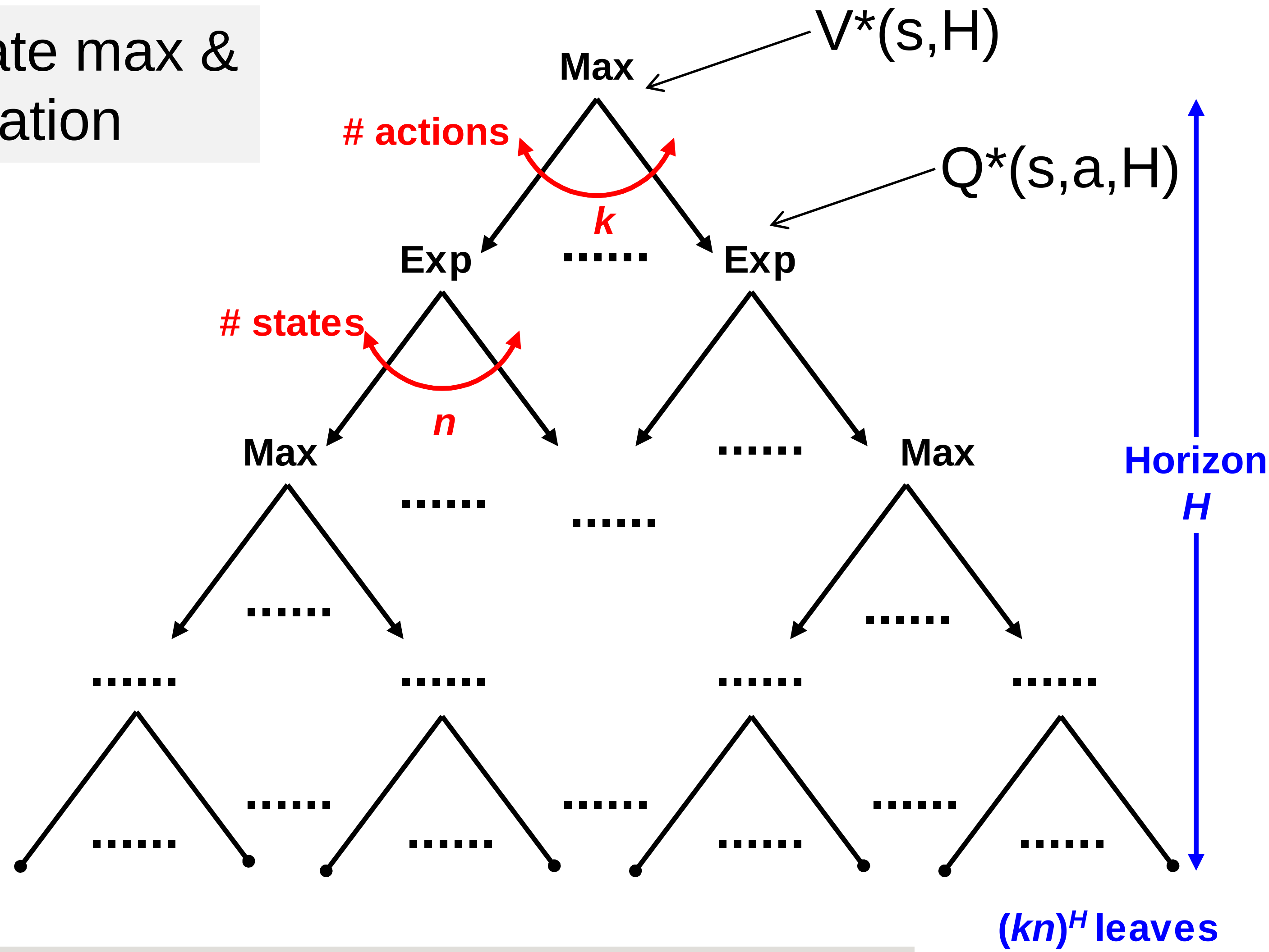
$= \max_{a1} E[R(s,a1) + \beta \max_{a2} E[R(T(s,a1),a2)+Q^*(T(s,a1),a2,h-1)]$

$= \quad \ldots\ldots$

- Can view this expansion as an expectimax tree
  - Each expectation is a weighted sum over states

Exact Expectimax Tree for $V^*(s,H)$

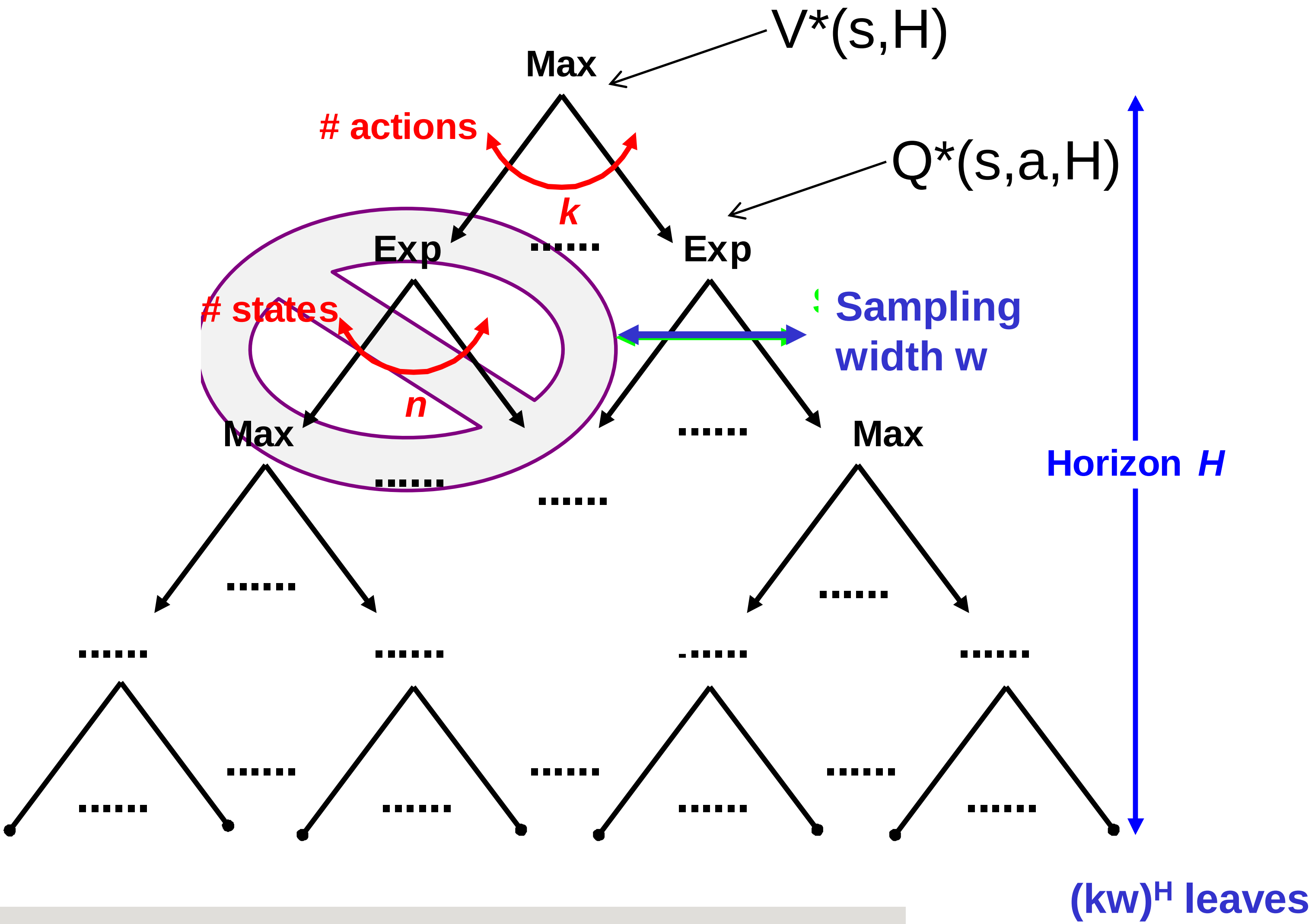
Alternate max & expectation



Compute root V* and Q* via recursive procedure

Depends on size of the state-space. Bad!

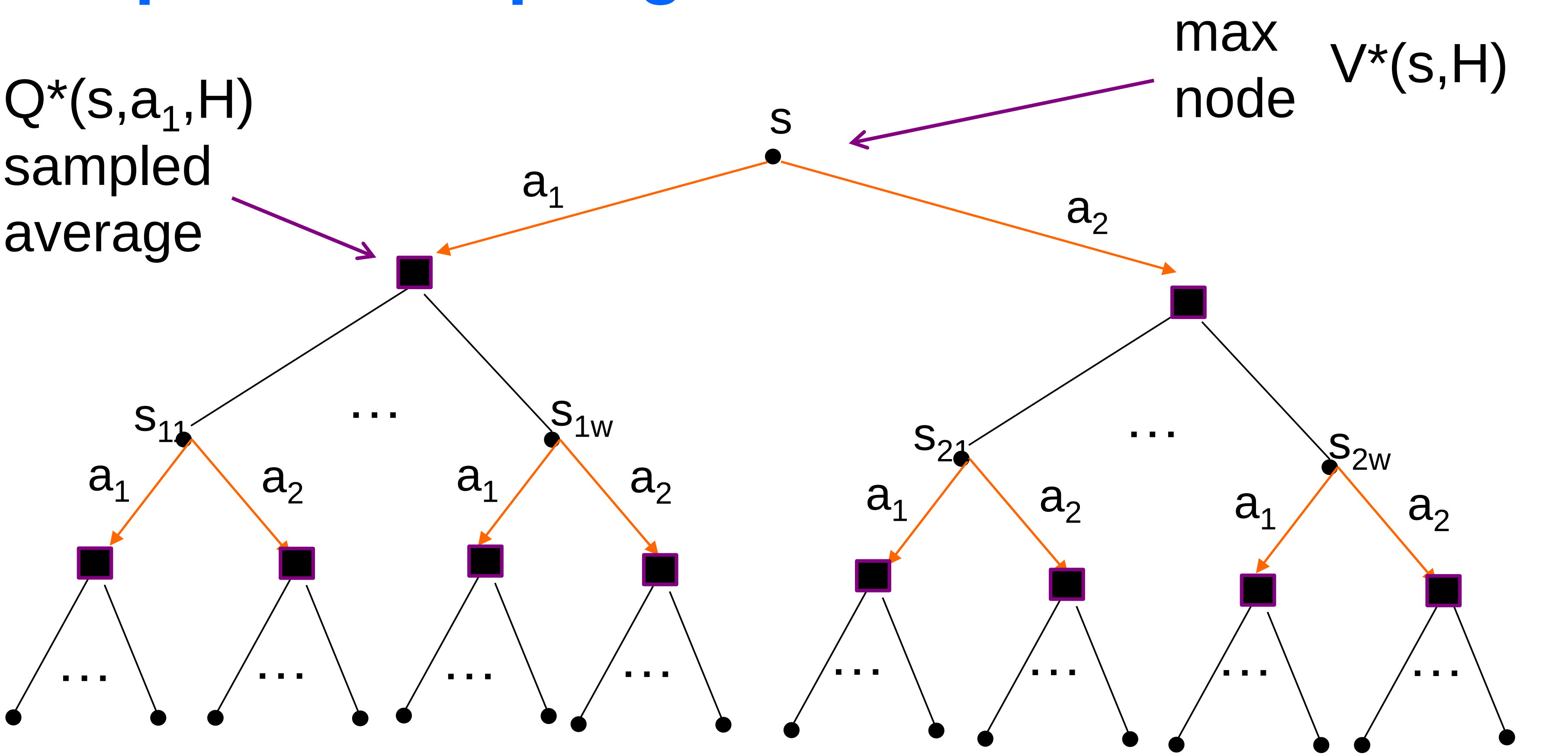
Sparse Sampling Tree



Replace expectation with average over *w* samples

w will typically be much smaller than *n.*

Sparse Sampling Tree



We could create an entire tree at each decision step and return action with highest Q* value at root.

High memory cost!

Sparse Sampling [Kearns et. al. 2002]

The Sparse Sampling algorithm computes root value via depth first expansion

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

SparseSampleTree(*s*,*h*,*w*)

If h=0 Return [0, null]

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-1),a*] = **SparseSample**(s_i,h-1,w)

        Q*(s,a,h) = Q*(s,a,h) + r_i + β V*(s_i,h-1)

    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* = argmax_a Q*(s,a,h)

Return [V*(s,h), a*]

Sparse Sampling (Cont'd)

- For a given desired accuracy, how large should sampling width and depth be?
  - Answered: Kearns, Mansour, and Ng (1999)
- **Good news:** gives values for w and *H* to achieve PAC guarantee on optimality
  - Values are 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*
  - Exponential in *H* is the best we can do in general
  - **In practice:** use small *H* & heuristic value at leaves

Sparse Sampling w/ Leaf Heuristic

Let $\hat{V}(s)$ be a heuristic value function estimator
Generally this is a very fast function, since it is evaluated at all leaves

SparseSampleTree(*s*,*h*,*w*)

~~If h=0 Return [0, null]~~ If h=0 Return [$\hat{\boldsymbol{V}}(s)$, null]

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-1),a*] = **SparseSample**(s_i,h-1,w)

Q*(s,a,h) = Q*(s,a,h) + r_i + β V*(s_i,h-1)

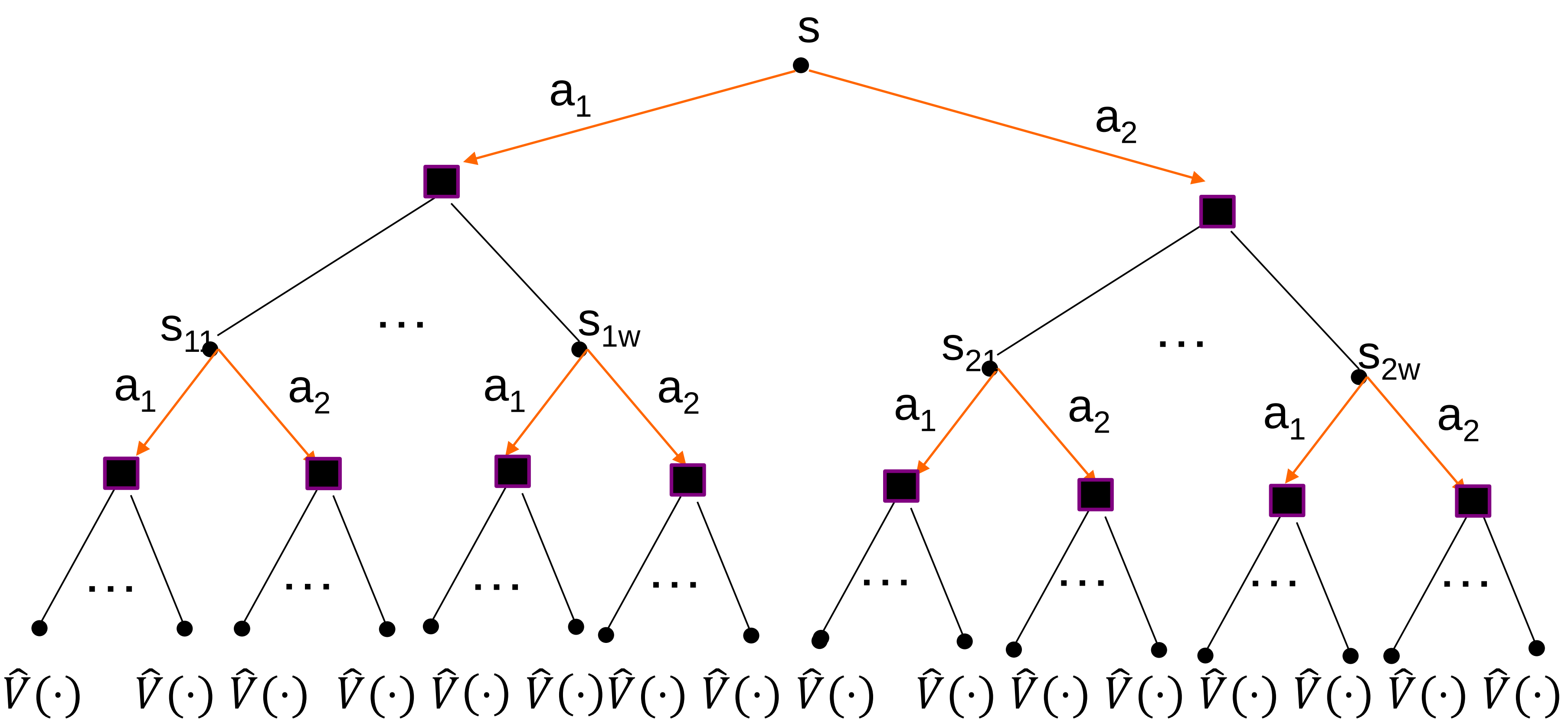
Q*(s,a,h) = Q*(s,a,h) / w ;; estimate of Q*(s,a,h)

V*(s,h) = $\max_a$ Q*(s,a,h) ;; estimate of V*(s,h)

a* = argmax_a Q*(s,a,h)

Return [V*(s,h), a*]

Shallow Horizon w/ Leaf Heuristic



Often a shallow sparse sampling search with a simple $\hat{V}$ at leaves can be very effective.

Anytime Behavior (or lack of it)

- **Bad News:** sparse sampling has poor "anytime behavior", which is often important in practice
- **Anytime Behavior:** good anytime behavior roughly means that an algorithm should be able to use small amounts of additional time to get small improvements
- Why doesn't sparse sampling have good anytime behavior?
  - Increasing information about a root action at depth h requires computing a sparse sub-tree of depth h.
  - Takes a lot of time for information to propagate to root

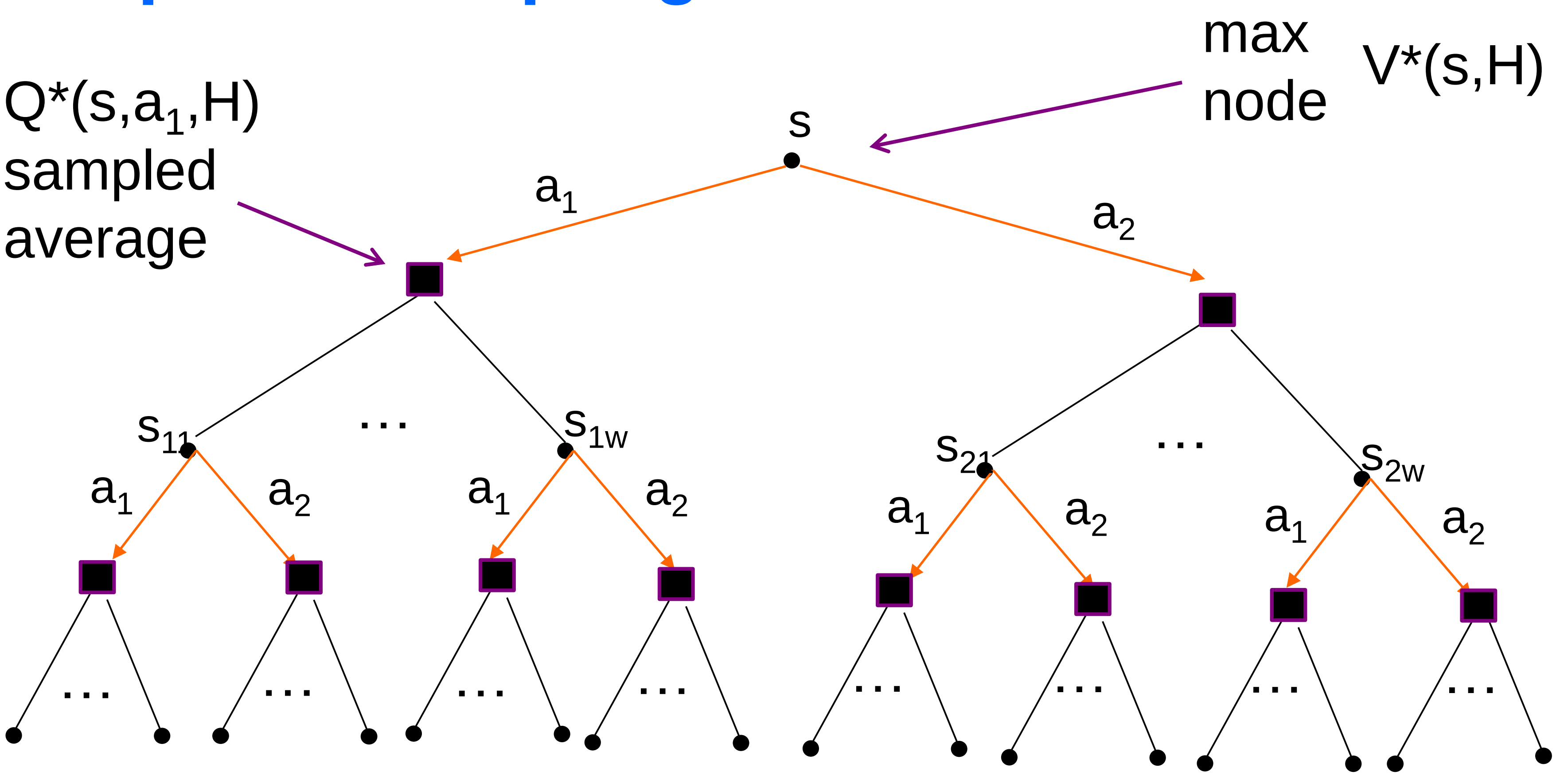
Sparse Sampling

- Will present two views of algorithm
  - The first is perhaps easier to digest
  - The second is more generalizable and can leverage advances in bandit algorithms

1. Approximation to the full expectimax tree

2. Recursive bandit algorithm
   - Consider horizon H=2 case first
   - Show for general H

Sparse Sampling Tree

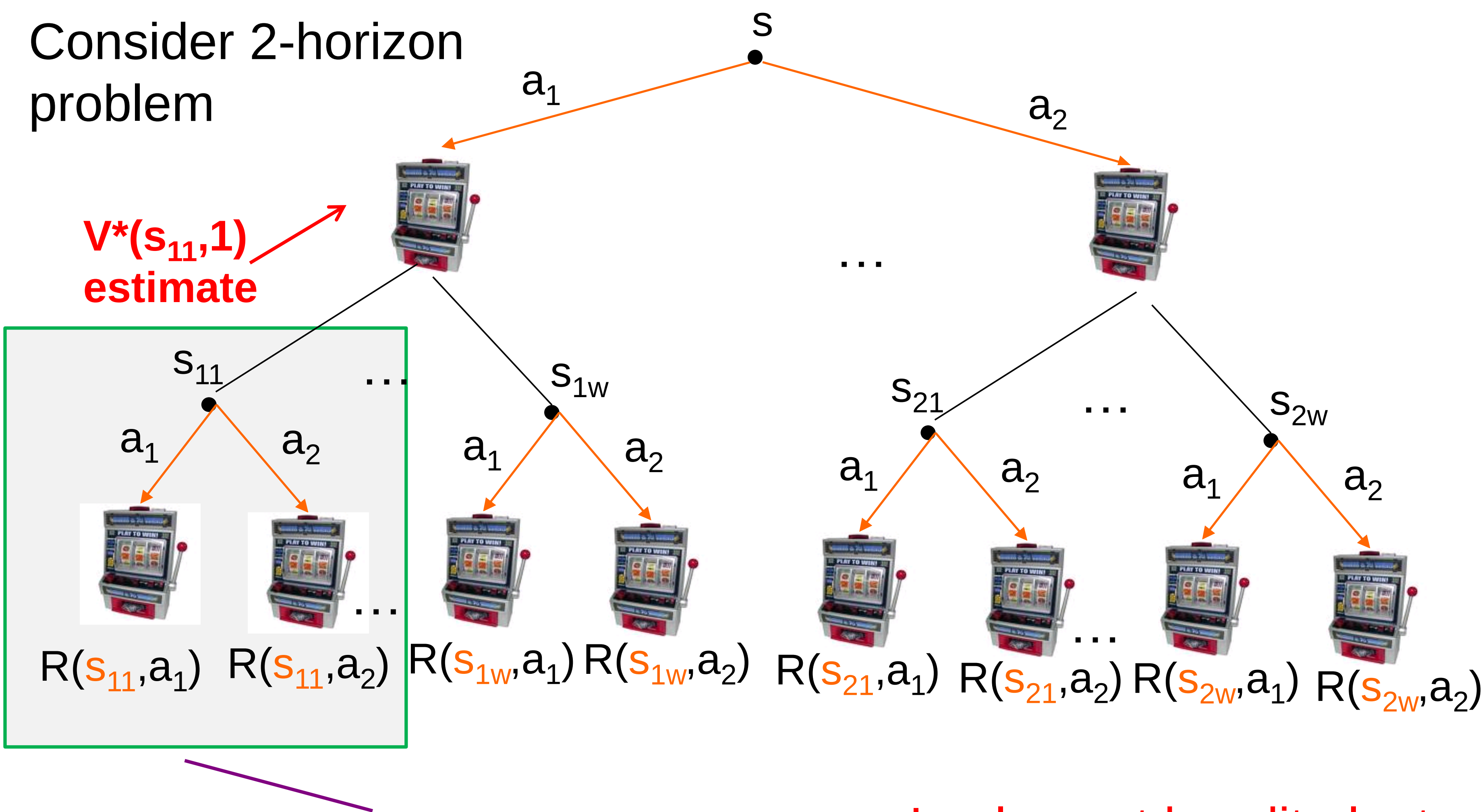


Each max node in tree is just a bandit problem.

I.e. must choose action with highest Q*(s,a,h)---approximate via bandit.

Bandit View of Sparse Sampling (H=2)

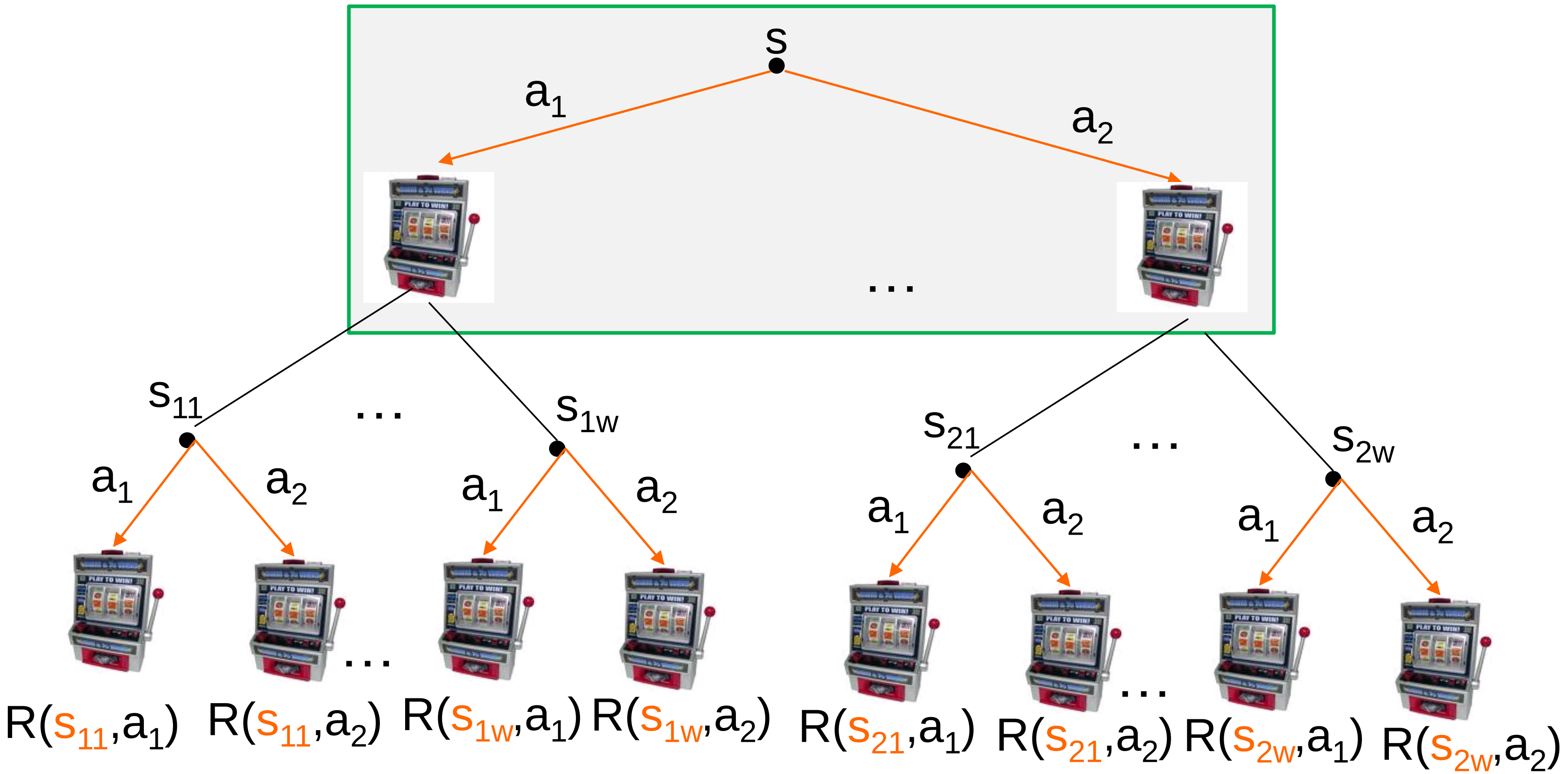
Consider 2-horizon problem



h=1: Traditional bandit problem
(stochastic arm reward $R(s_{11},a_i)$*)*

Implement bandit alg. to return estimated expected reward of best arm

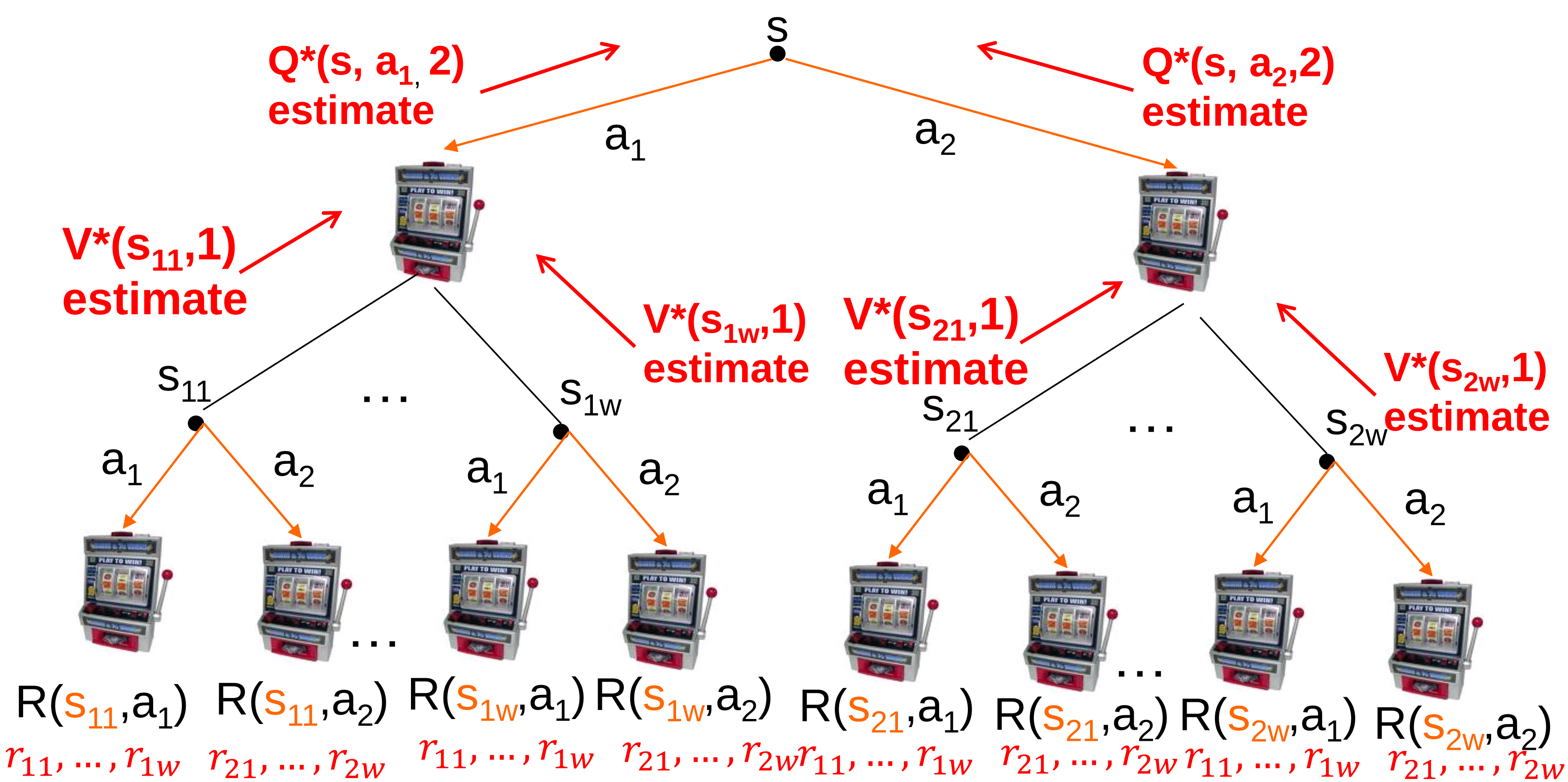
Bandit View of Sparse Sampling (H=2)



h=2: higher level bandit problem (finds arm with best Q* value for h=2)

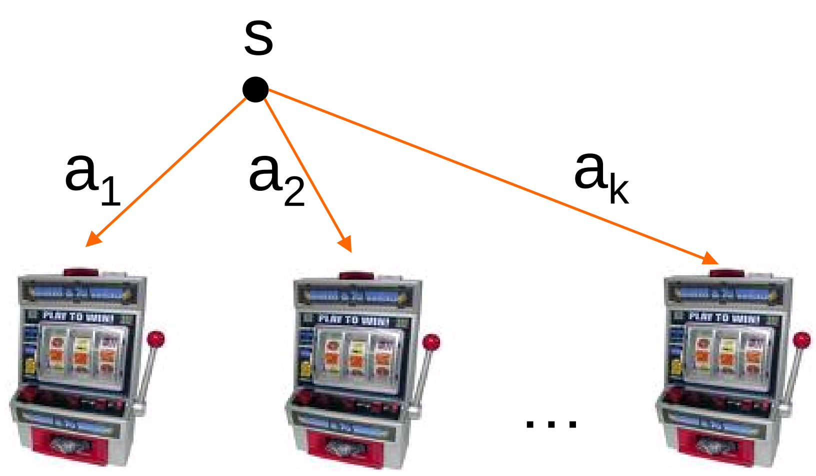
Pulling an arm returns a Q-value estimate by: 1) sample next state **s'**, 2) run h=1 bandit at **s',** return immediate reward + estimated value of s'

Bandit View of Sparse Sampling (h=2)



Consider UniformBandit using w pulls per arm

Bandit View: General Horizon H

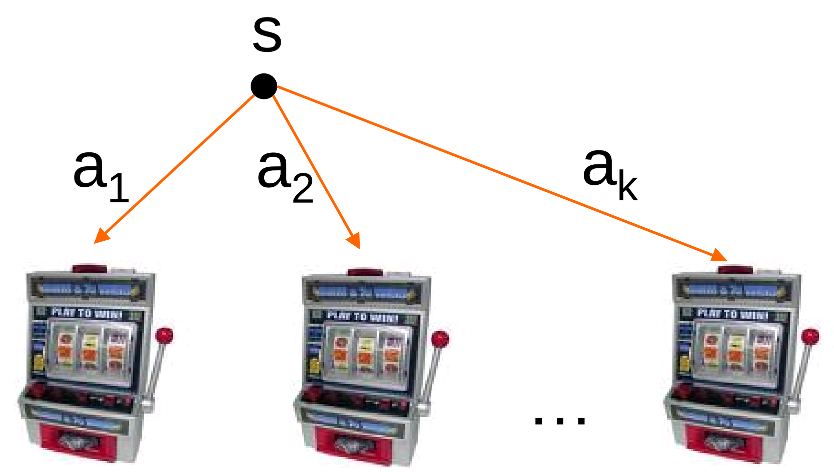


SimQ*(s,a_1,h)   SimQ*(s,a_2,h)   SimQ*(s,a_k,h)

- SimQ*(s,a,h) : we want this to return a random sample of the immediate reward and then h-1 value of resulting state when executing action a in s

- If this is (approx) satisfied then bandit algorithm will select near optimal arm.

Bandit View: General Horizon H



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

Definition:

BanditValue$(A_1, A_2, \ldots, A_k)$ returns estimated expected value of best arm (e.g. via UniformBandit)

SimQ*(s,a,h)

r = R(s,a)

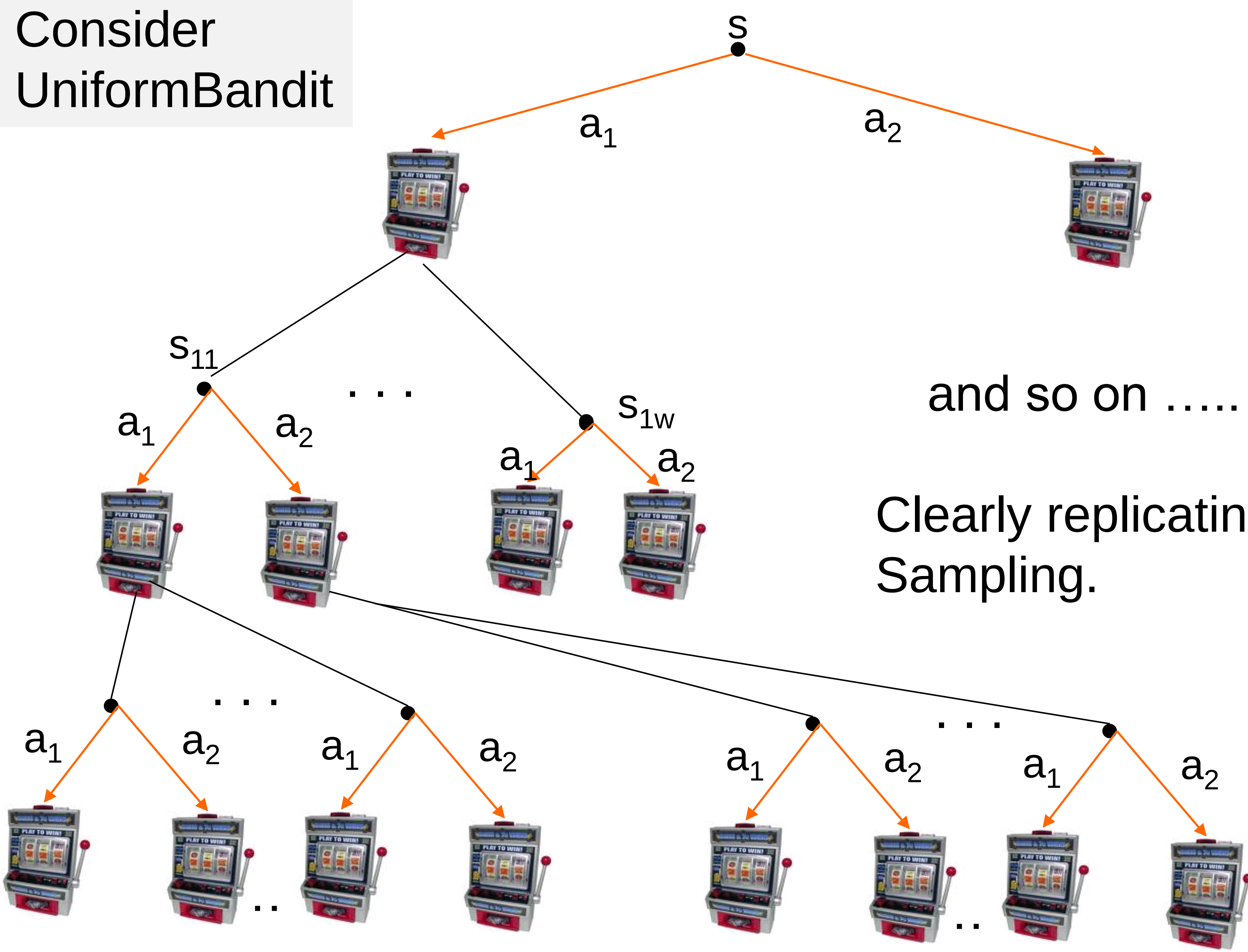
If h=1 then Return r

s' = T(s,a)

Return $r + \beta\ \mathbf{BanditValue}\overbrace{(\mathbf{SimQ}^*(\boldsymbol{s'}, \boldsymbol{a_1}, \boldsymbol{h}-\mathbf{1}), \ldots, \mathbf{SimQ}^*(\boldsymbol{s'}, \boldsymbol{a_k}, \boldsymbol{h}-\mathbf{1}))}^{\text{k-arm bandit problem at state s'}}$

Recursive UniformBandit: General H

Consider UniformBandit



and so on …..

Clearly replicating Sparse Sampling.

Recursive Bandit: General Horizon H

SelectRootAction(s,H)

Return $\mathrm{BanditAction}(\mathrm{SimQ}^*(s, a_1, H), \ldots, \mathrm{SimQ}^*(s, a_k, H))$

SimQ*(s,a,h)

r = R(s,a)

If h=1 then Return r

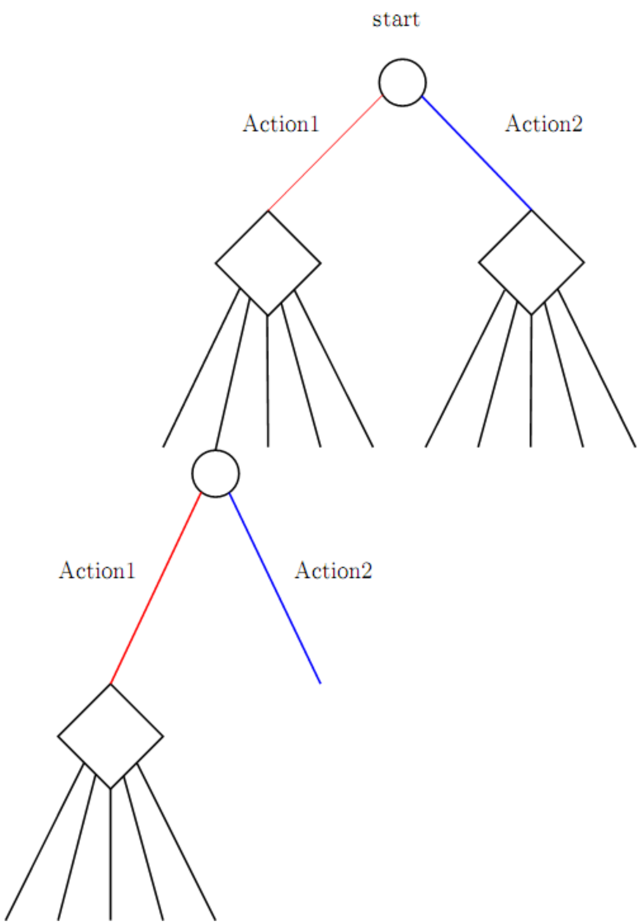
s' = T(s,a)

Return $r + \beta\ \mathrm{BanditValue}(\mathrm{SimQ}^*(s', a_1, h-1), \ldots, \mathrm{SimQ}^*(s', a_k, h-1))$

- When bandit is UniformBandit same as Sparse Sampling
- Can plug in more advanced bandit algorithms for possible improvement!

Uniform vs. Non-Uniform 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?
- Use non-uniform bandits



Non-Uniform Recursive Bandits

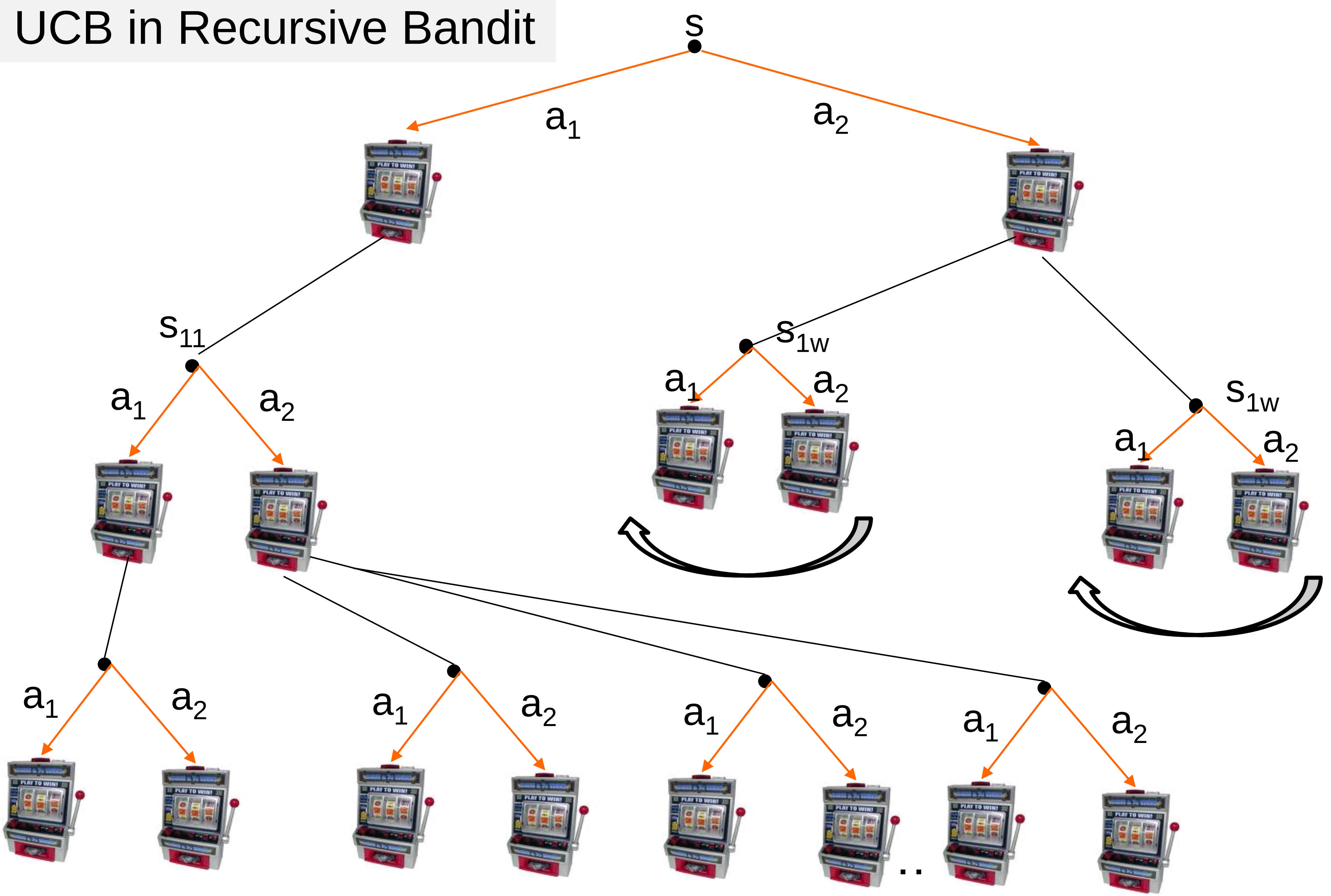
UCB-Based Sparse Sampling

- Use UCB as bandit algorithm
- There is an analysis of this algorithm's bias (it goes to zero)

H.S. Chang, M. Fu, J. Hu, and S.I. Marcus. An adaptive sampling algorithm for solving Markov decision processes. Operations Research, 53(1):126--139, 2005.

Recursive UCB: General H

UCB in Recursive Bandit



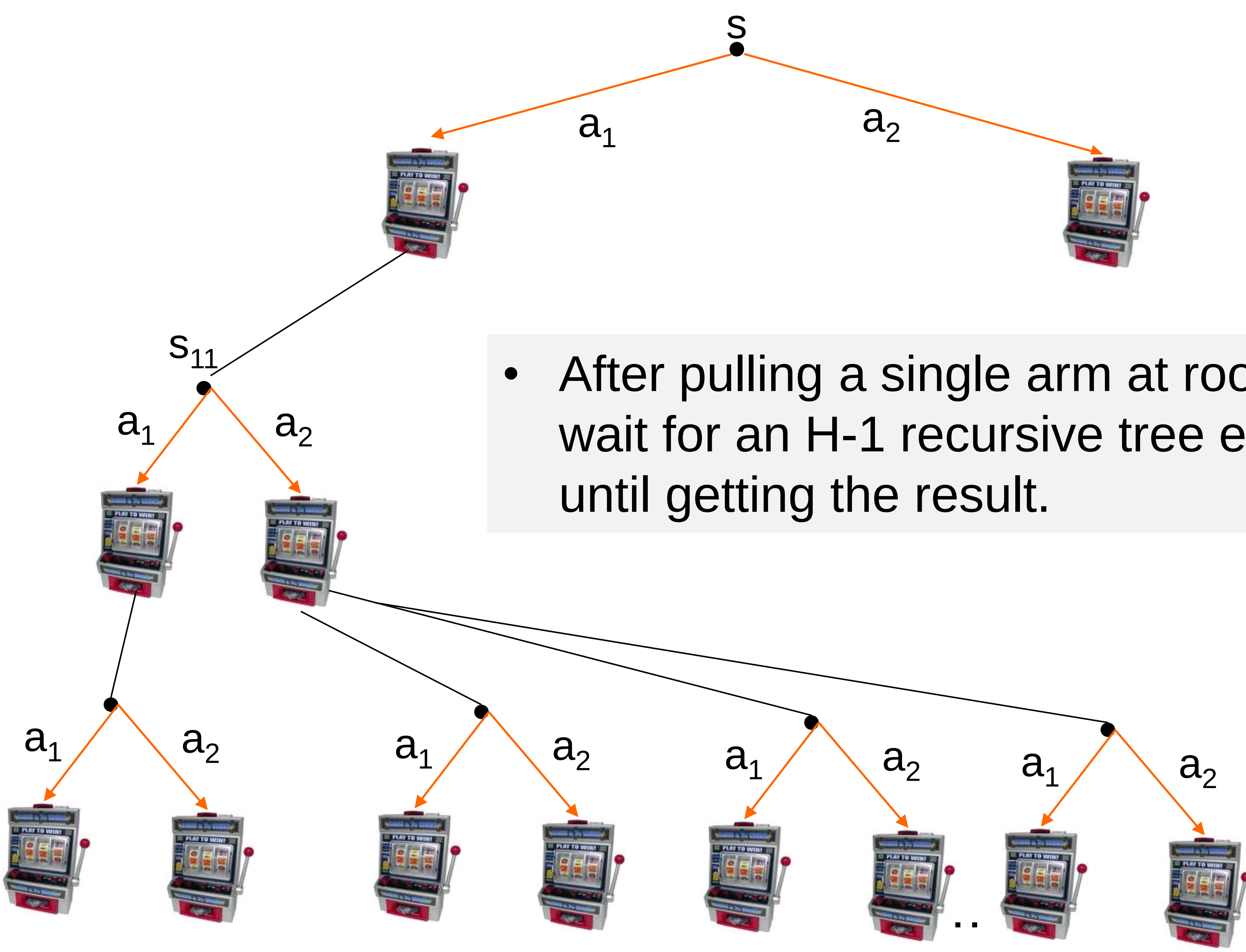
Non-UniformRecursive Bandits

- UCB-Based Sparse Sampling
  - Is UCB the right choice?
  - We don’t really care about cumulative regret.
  - My Guess: part of the reason UCB was tried was for purposes of leveraging its analysis
- $\epsilon - \mathrm{Greedy}$ Sparse Sampling
  - Use $\epsilon - \mathrm{Greedy}$ as the bandit algorithm
  - I haven’t seen this in the literature
  - Might be better in practice since it is more geared to simple regret
  - This would raise issues in the analysis (beyond the scope of this class).

Non-Uniform Recursive Bandits

- **Good News:** we might expect to improve over pure Sparse Sampling by changing the bandit algorithm
- **Bad News:** this recursive bandit approach has poor “anytime behavior”, which is often important in practice
- **Anytime Behavior:** good anytime behavior roughly means that an algorithm should be able to use small amounts of additional time to get small improvements
  - What about these recursive bandits?

Recursive UCB: General H



- After pulling a single arm at root we wait for an H-1 recursive tree expansion until getting the result.

Non-Uniform Recursive Bandits

- Information at the root only increases after each of the expensive root arm pulls
  - Much time passes between these pulls
- Thus, small amounts of additional time does not result in any additional information at root!
  - Thus, poor anytime behavior
  - Running for 10sec could essentially the same as running for 10min (for large enough H)
- Can we improve the anytime behavior?

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 Look-Ahead Trees
  - Sparse Sampling
  - Sparse Sampling via Recursive Bandits
  - Monte Carlo Tree Search: UCT and variants

UCT Algorithm

Bandit Based Monte-Carlo Planning. (2006). Levente Kocsis & Csaba Szepesvari. European Conference, on Machine Learning,

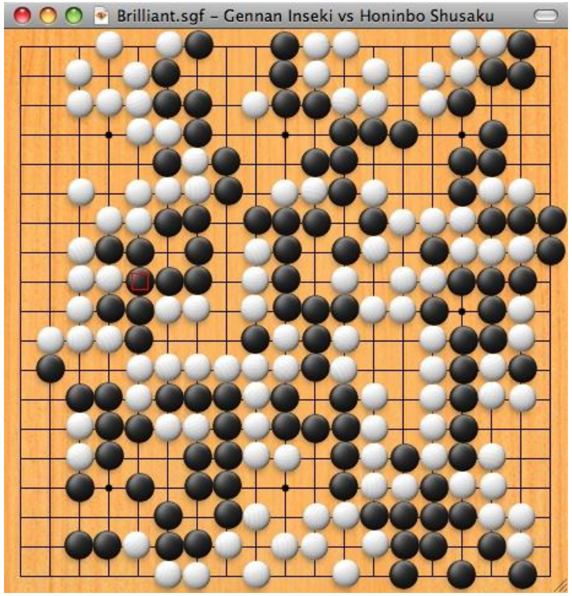
- UCT is an instance of Monte-Carlo Tree Search
  - Applies bandit principles in this framework
  - Similar theoretical properties to sparse sampling
  - Much better anytime behavior than sparse sampling

- Famous for yielding a major advance in computer Go

- A growing number of success stories
  - Practical successes still not understood so well

Monte Carlo Tree Search

Idea #1: board evaluation function via random rollouts



Evaluation Function:

- play many random games
- evaluation is fraction of games won by current player
- surprisingly effective

Even better if use rollouts that select better than random moves

Monte Carlo Tree Search

Idea #2: selective tree expansion

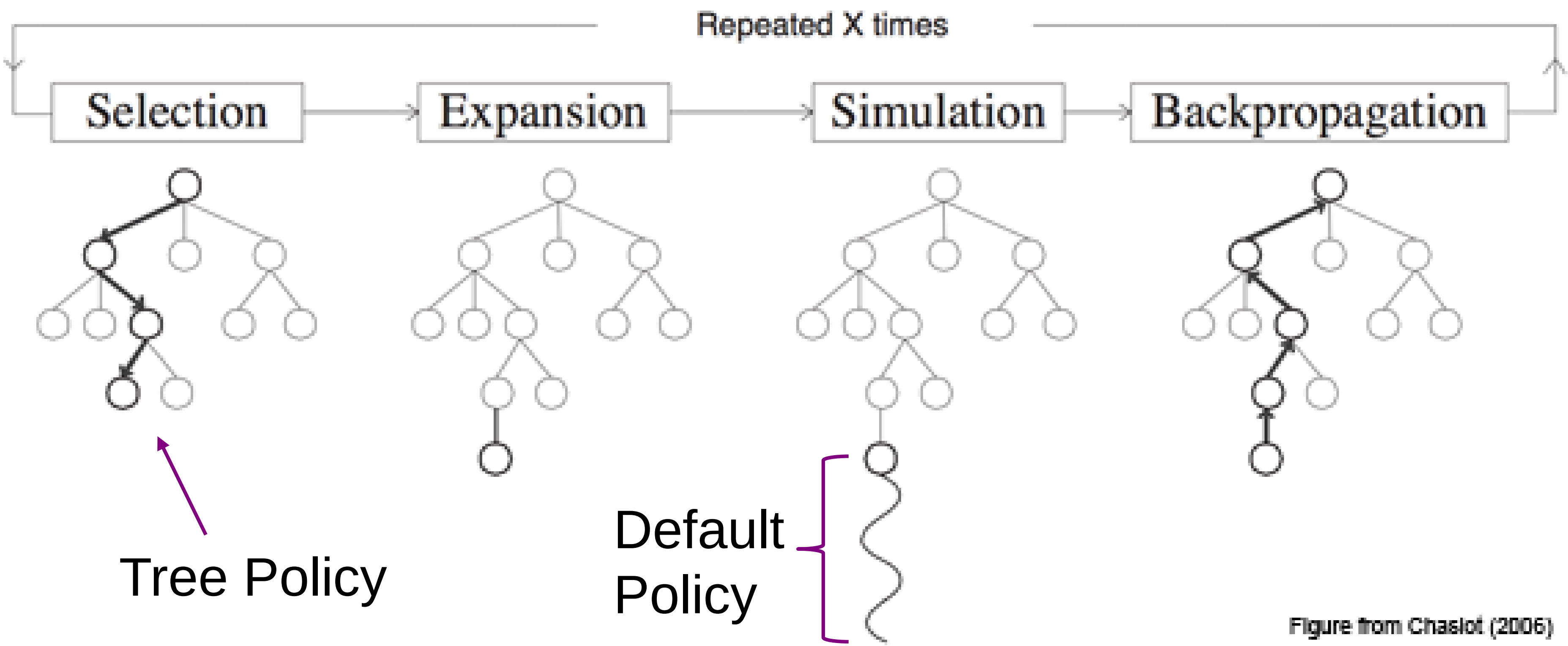
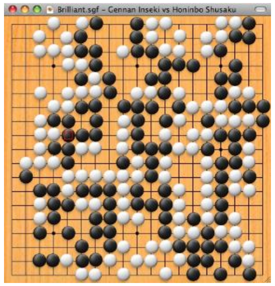
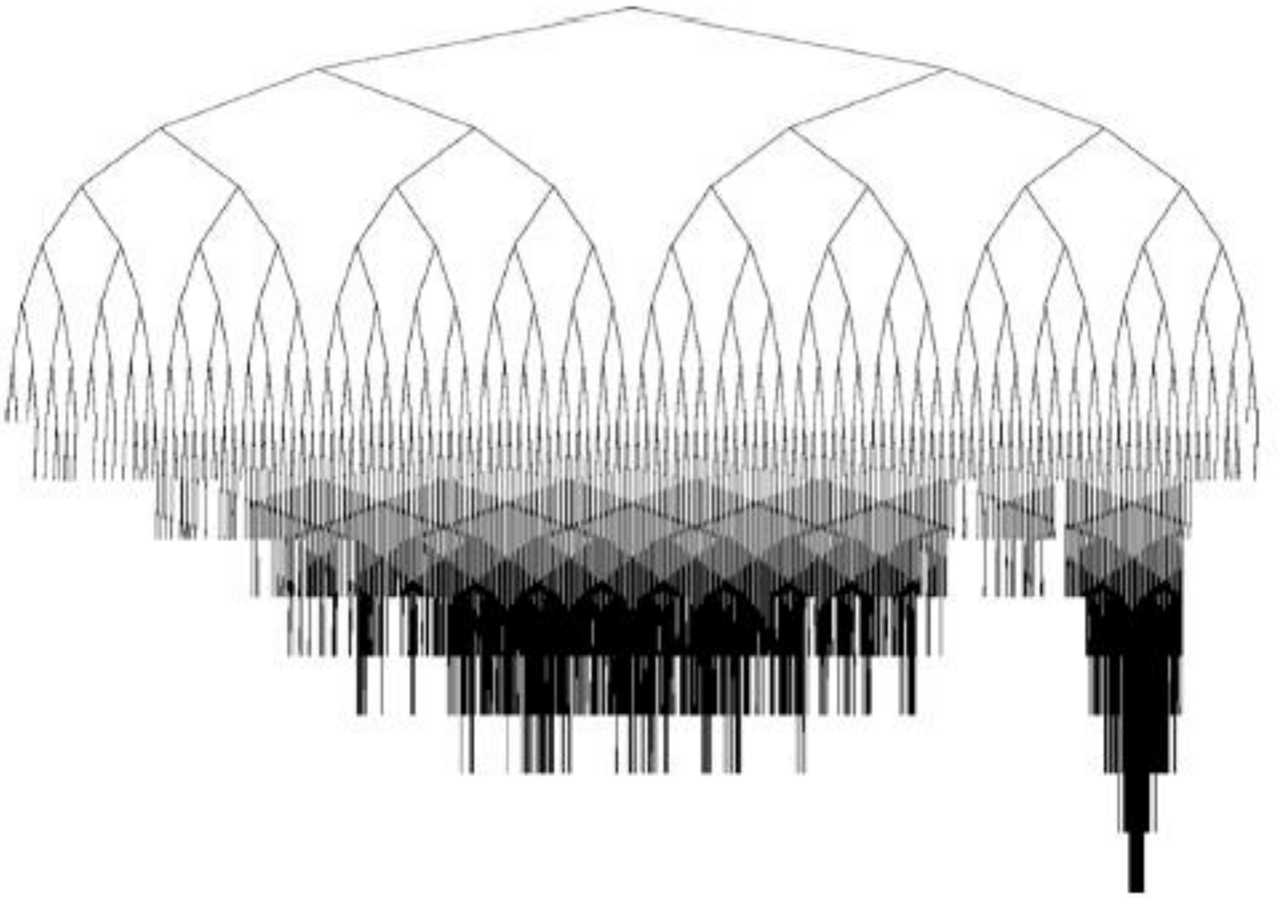


Figure from Chaslot (2006)

Monte Carlo Tree Search



Idea #2: selective tree expansion



Non-uniform tree growth

Monte-Carlo Tree Search: Informal

- Builds a sparse look-ahead tree rooted at current state by repeated Monte-Carlo simulation of a "**rollout policy**"
  - Rollout policy is the combination of tree policy and default policy on previous slide (produces trajectory from root to horizon)
- During construction each tree node *s* stores:
  - state-visitation count $n(s)$, action counts $n(s,a)$, action values $Q(s,a)$

What is the rollout policy?

- Repeat until time is up
  1. Execute rollout policy starting from root until horizon (generates a state-action-reward trajectory)
  2. Add first node not in current tree to the tree (expansion phase)
  3. Update statistics of each tree node *s* on trajectory
     - Increment $n(s)$ and $n(s,a)$ for selected action *a*
     - Update $Q(s,a)$ by total reward observed after the node

Rollout Policies

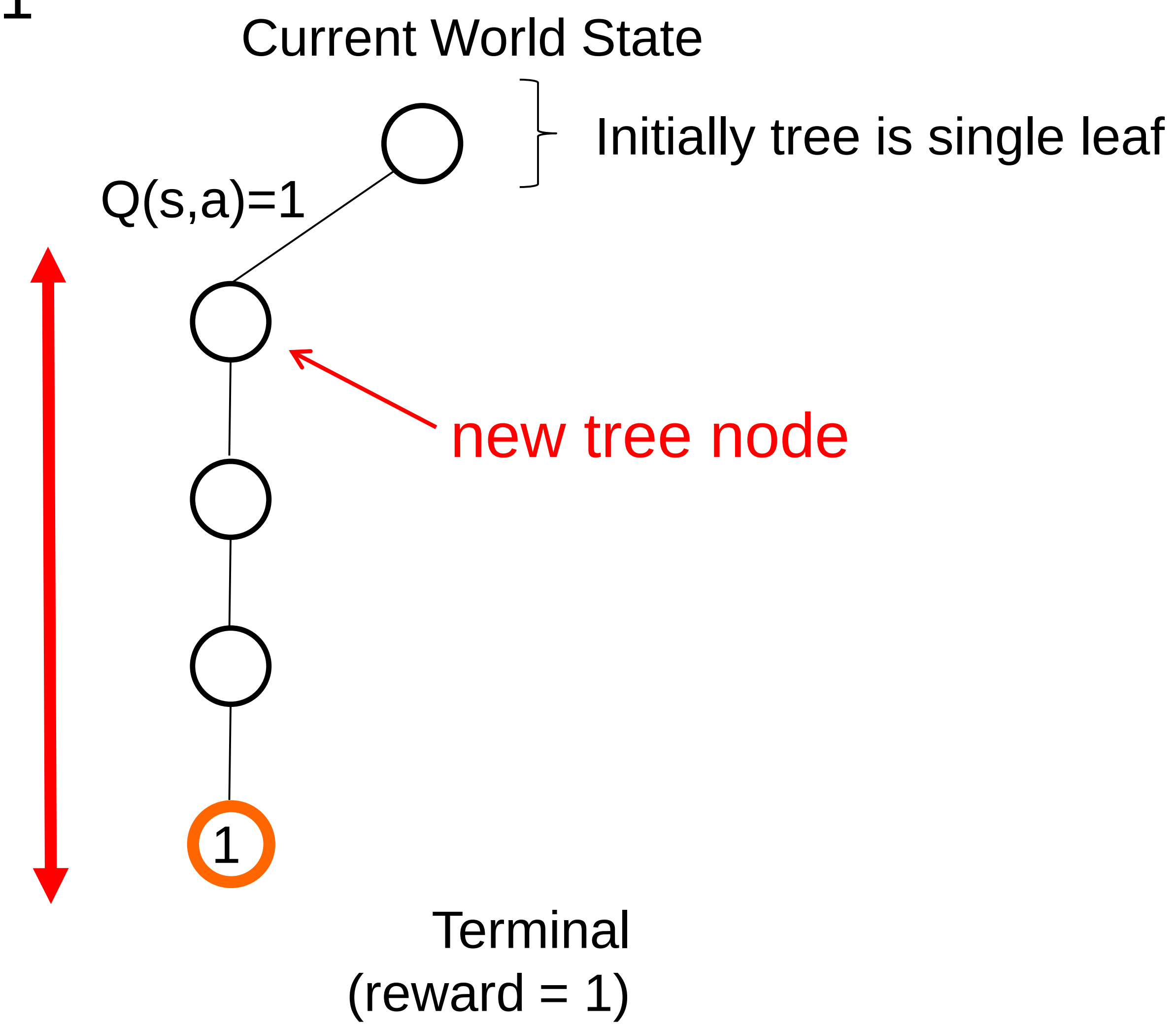
- Monte-Carlo Tree Search algorithms mainly differ on their choice of rollout policy

- Rollout policies have two distinct phases
  - **Tree policy:** selects actions at nodes already in tree (each action must be selected at least once)
  - **Default policy:** selects actions after leaving tree

- **Key Idea:** the tree policy can use statistics collected from previous trajectories to intelligently expand tree in most promising direction
  - Rather than uniformly explore actions at each node

At a leaf node tree policy selects a random action then executes default

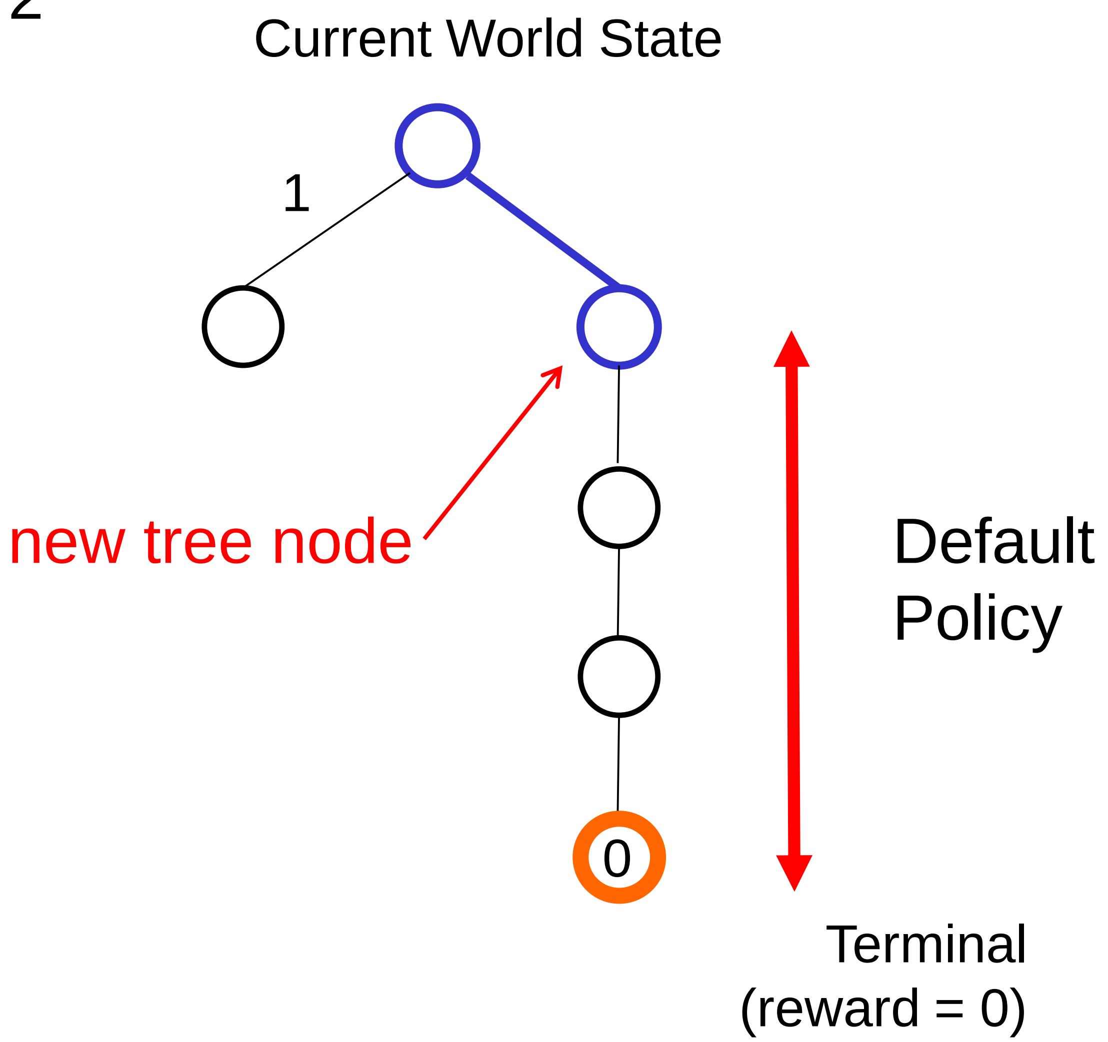
Iteration 1



Assume all non-zero reward occurs at terminal nodes.

Must select each action at a node at least once

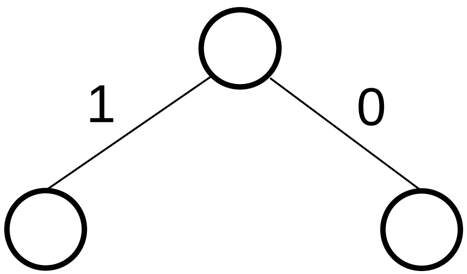
Iteration 2



Must select each action at a node at least once

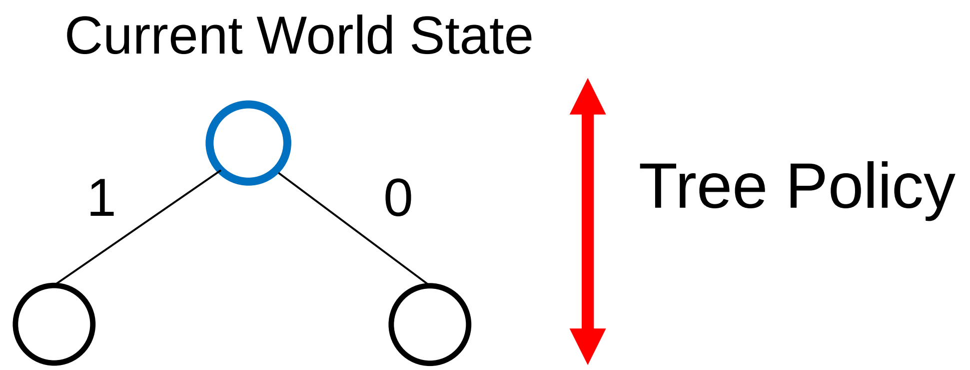
Iteration 3

Current World State



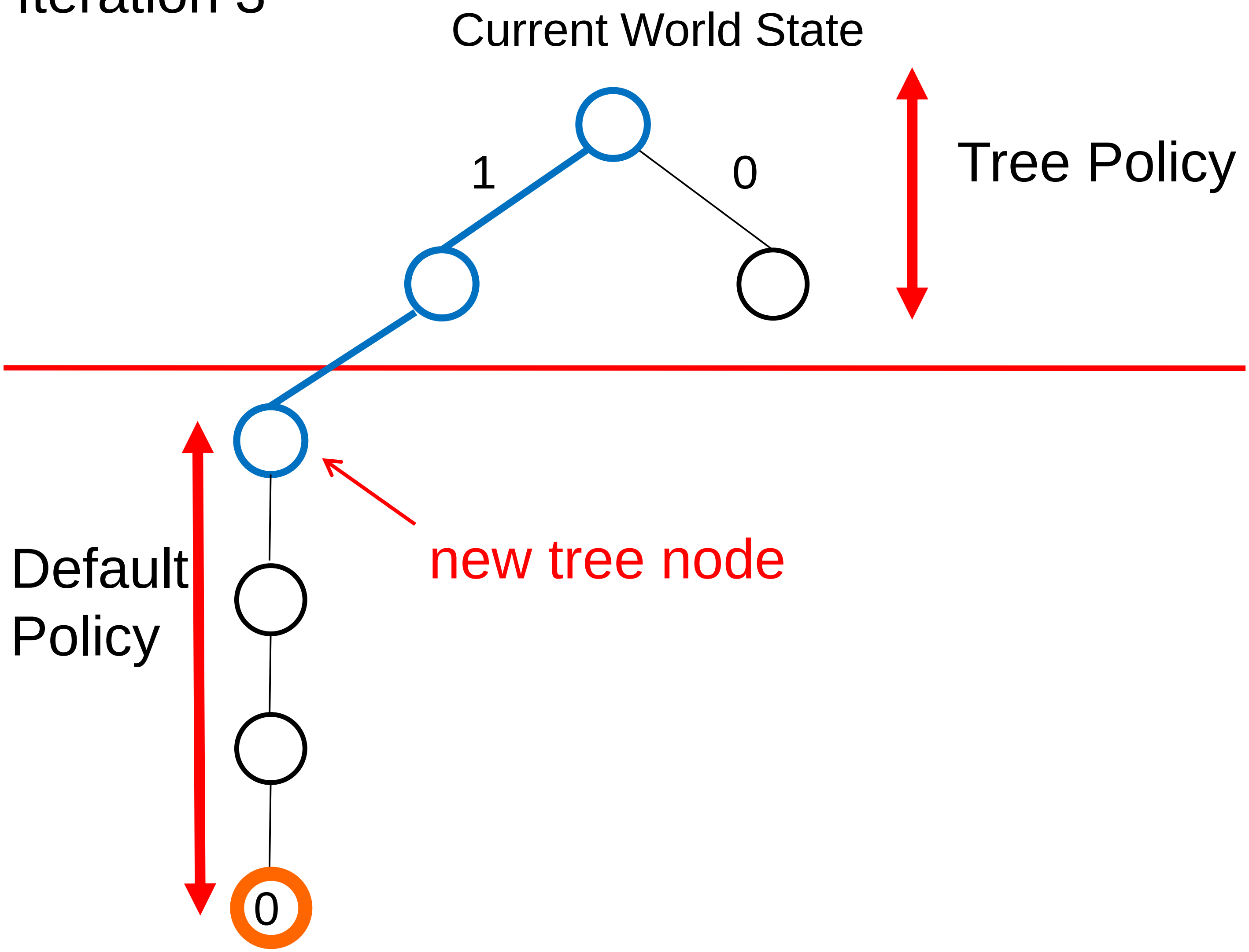
When all node actions tried once, select action according to tree policy

Iteration 3



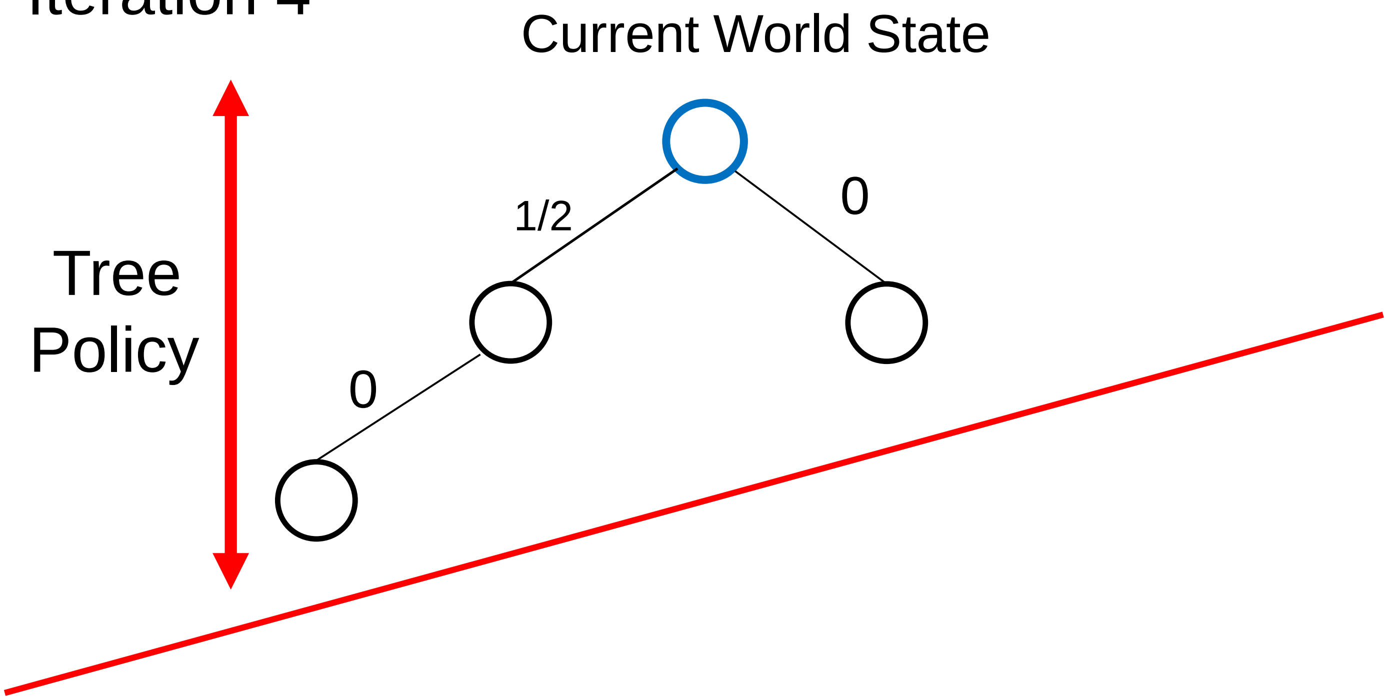
When all node actions tried once, select action according to tree policy

Iteration 3



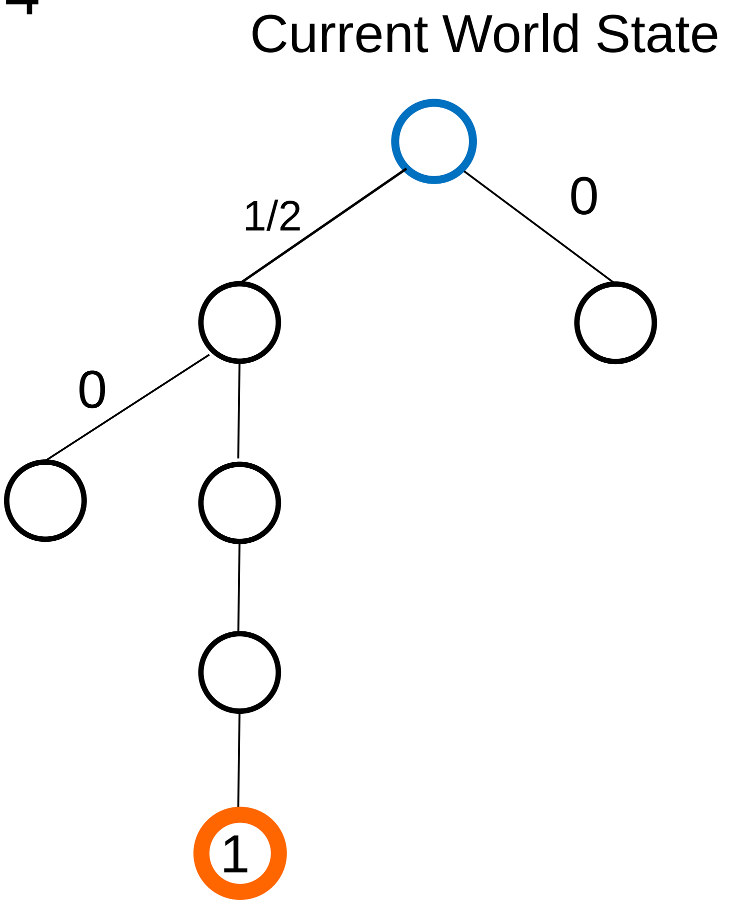
When all node actions tried once, select action according to tree policy

Iteration 4

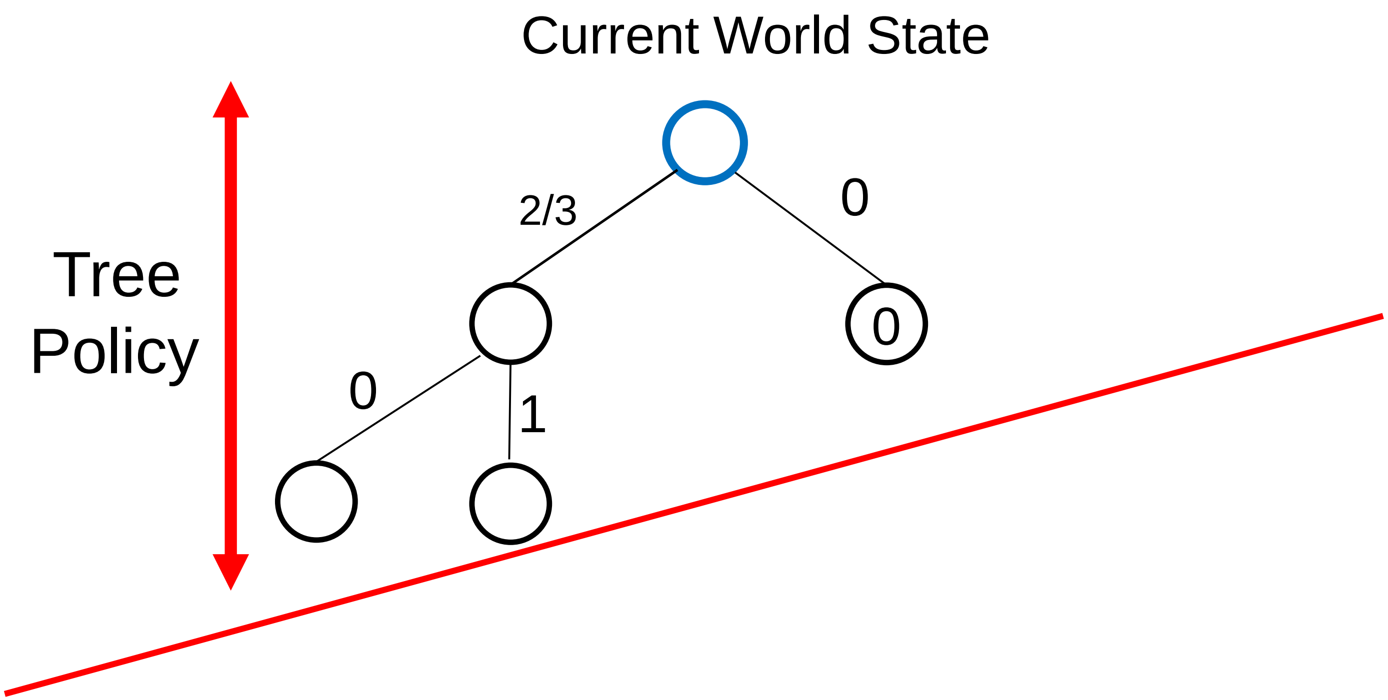


When all node actions tried once, select action according to tree policy

Iteration 4



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



What is an appropriate tree policy?
Default policy?

UCT Algorithm [Kocsis & Szepesvari, 2006]

- Basic UCT uses a random default policy
  - In practice often use hand-coded or learned policy

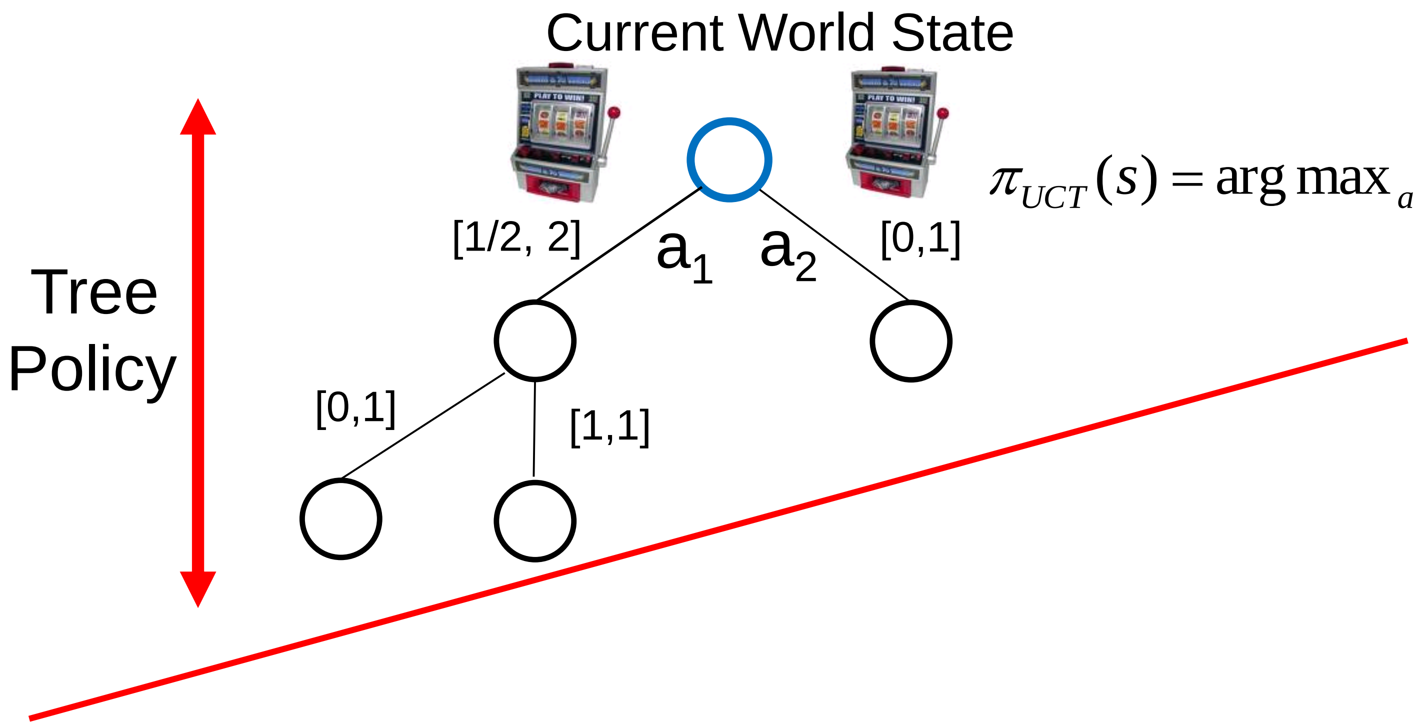
- Tree policy is based on UCB:
  - Q(s,a) : average reward received in current trajectories after taking action a in state s
  - n(s,a) : number of times action a taken in s
  - n(s) : number of times state s encountered

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

Theoretical constant that is empirically selected in practice

(theoretical results based on c equal to horizon H)

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

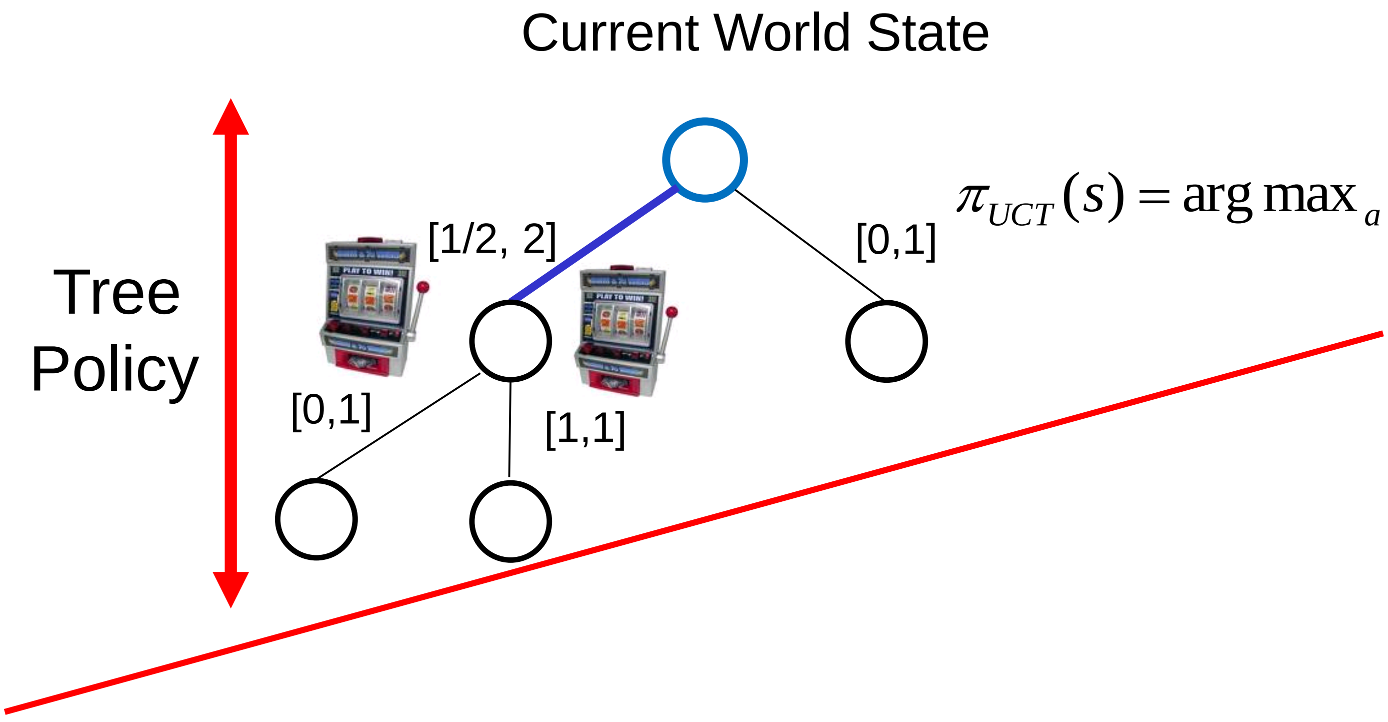


$$\pi_{UCT}(s) = \arg\max_a Q(s,a) + c\sqrt{\frac{\ln n(s)}{n(s,a)}}$$

Edges/actions store [Q(s,a), n(s,a)]

Nodes/states stores n(s)
= sum of n(s,a) over all actions
(not shown in animation)

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

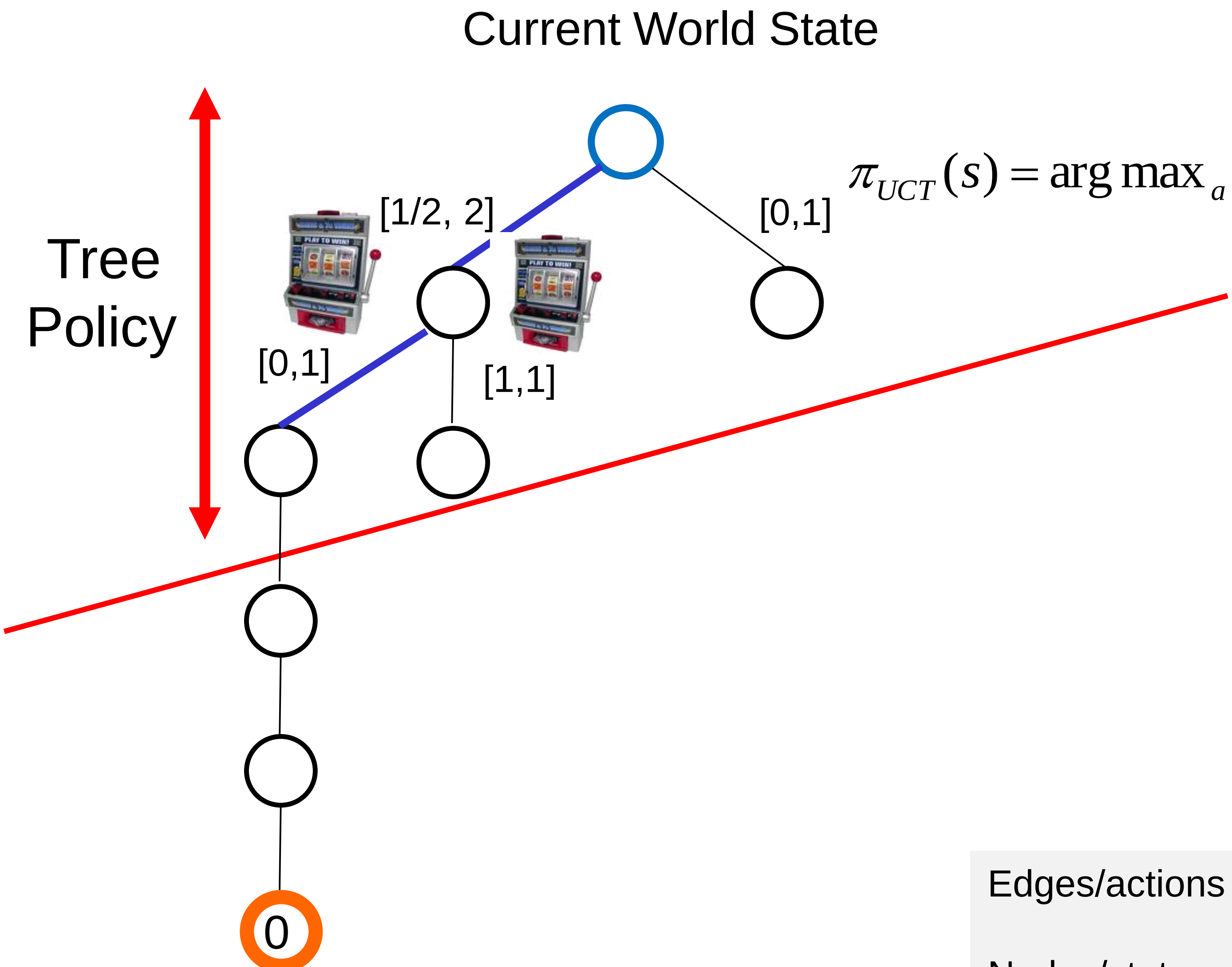


$$\pi_{UCT}(s) = \arg\max_a Q(s,a) + c\sqrt{\frac{\ln n(s)}{n(s,a)}}$$

Edges/actions store [Q(s,a), n(s,a)]

Nodes/states stores n(s)
= sum of n(s,a) over all actions
(not shown in animation)

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

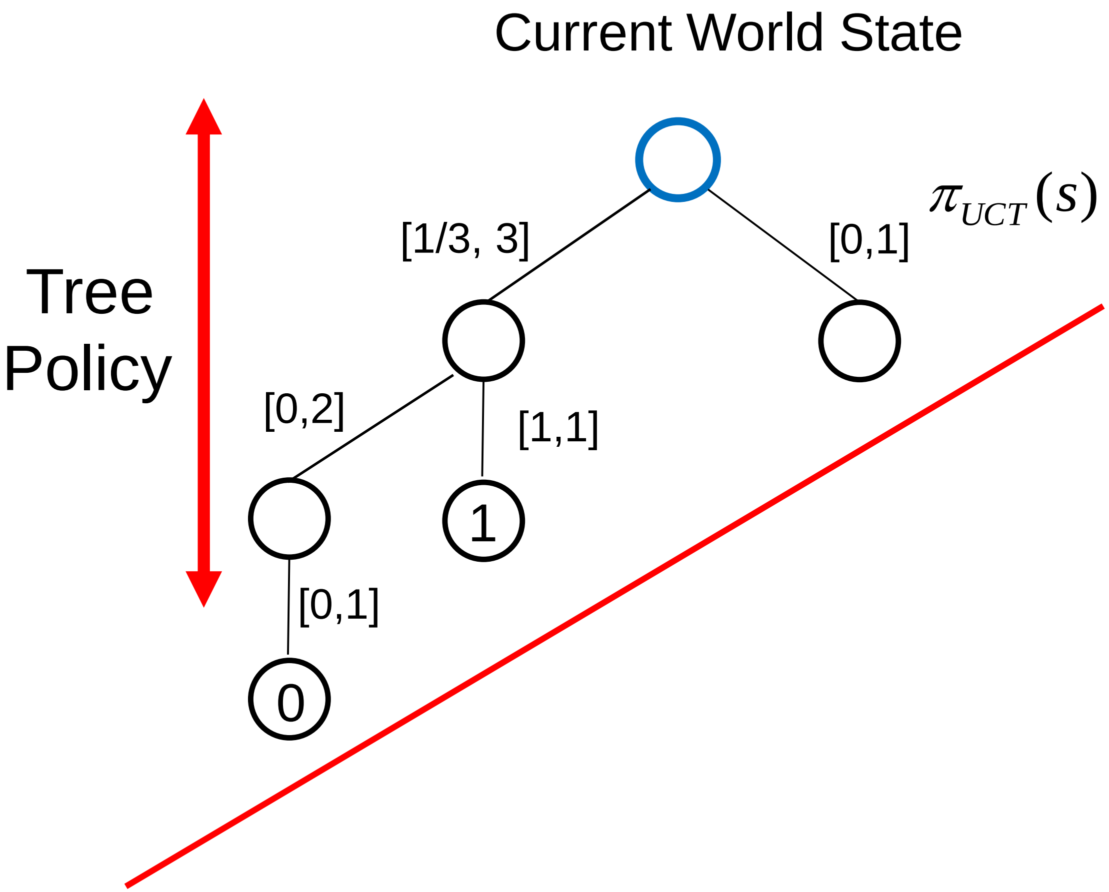


$$\pi_{UCT}(s) = \arg\max_a Q(s,a) + c\sqrt{\frac{\ln n(s)}{n(s,a)}}$$

Edges/actions store [Q(s,a), n(s,a)]

Nodes/states stores n(s)
= sum of n(s,a) over all actions
(not shown in animation)

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

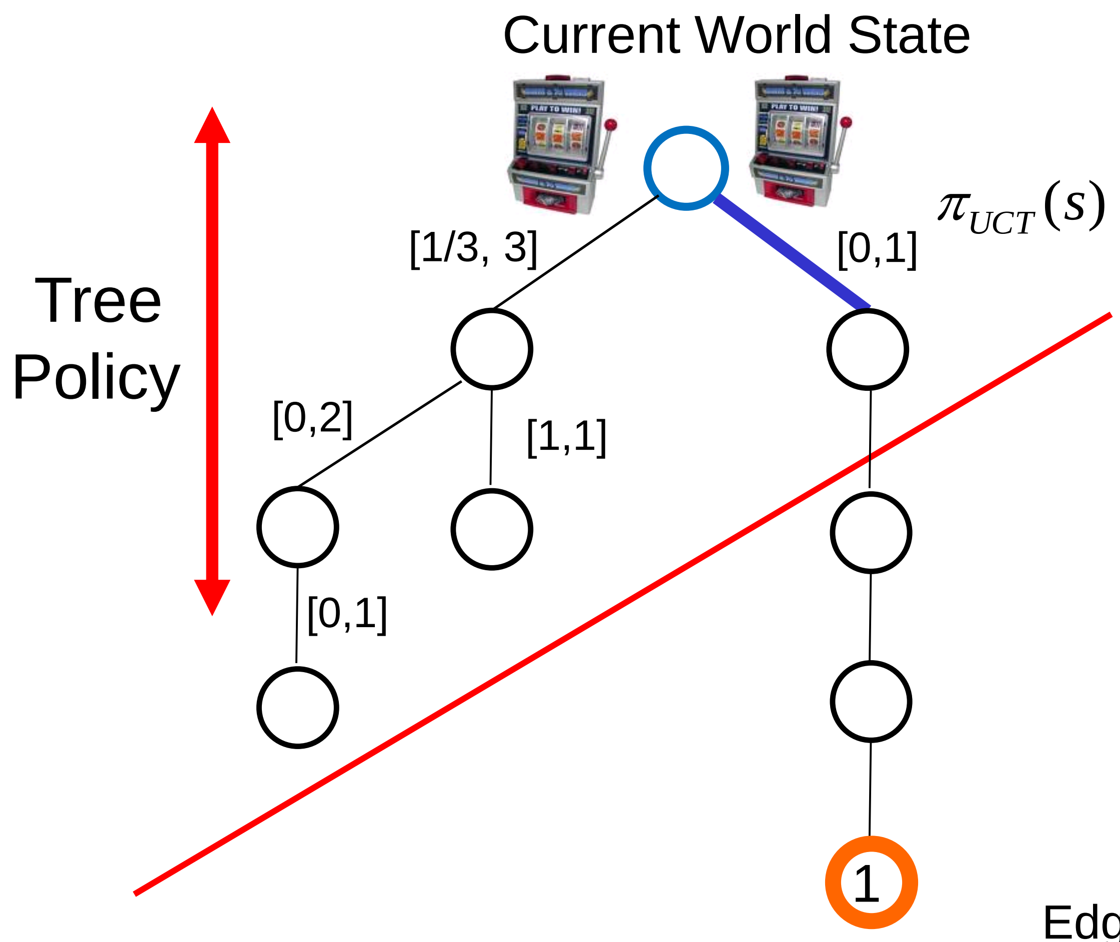


$$\pi_{UCT}(s) = \arg\max_a Q(s,a) + c\sqrt{\frac{\ln n(s)}{n(s,a)}}$$

Edges/actions store [Q(s,a), n(s,a)]

Nodes/states stores n(s)
= sum of n(s,a) over all actions
(not shown in animation)

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

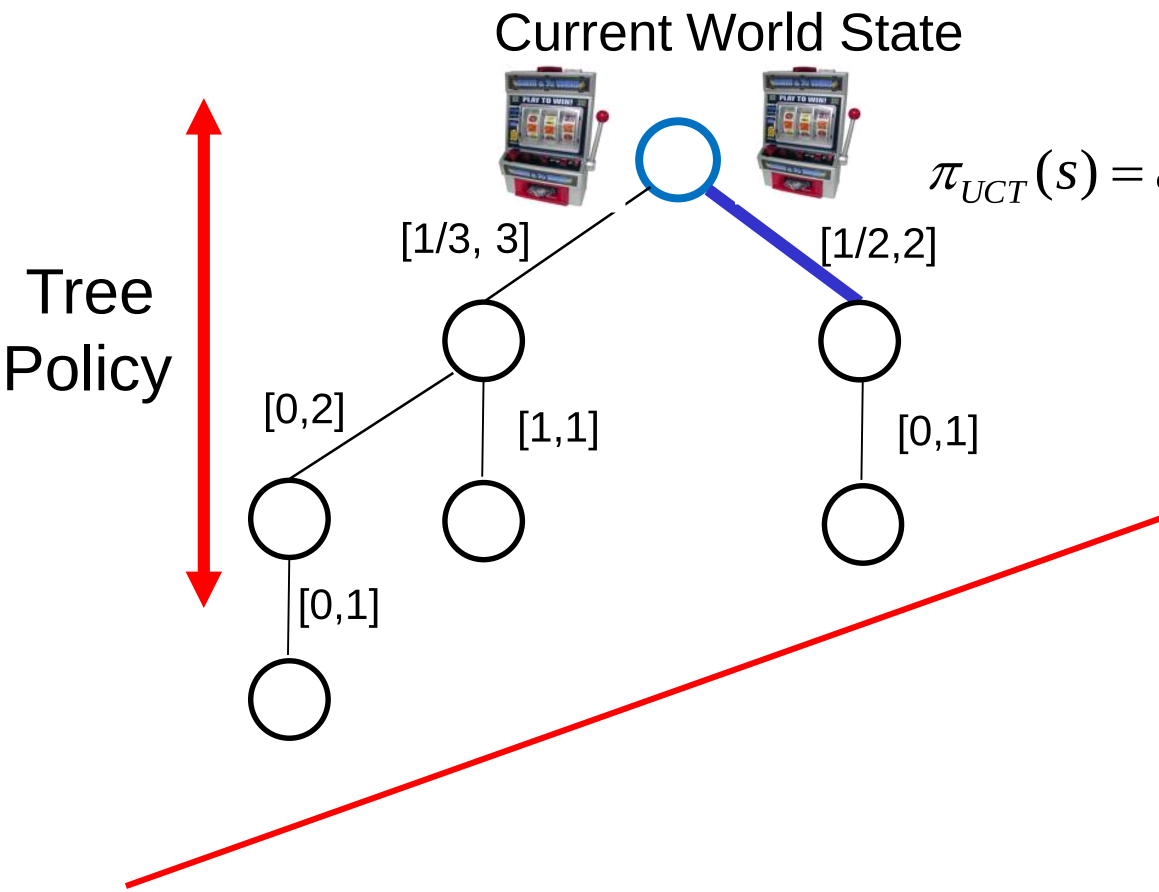


$$\pi_{UCT}(s) = \arg\max_a Q(s,a) + c\sqrt{\frac{\ln n(s)}{n(s,a)}}$$

Edges/actions store [Q(s,a), n(s,a)]

Nodes/states stores n(s)
= sum of n(s,a) over all actions
(not shown in animation)

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



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

Edges/actions store [Q(s,a), n(s,a)]

Nodes/states stores n(s)
= sum of n(s,a) over all actions
(not shown in animation)

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)

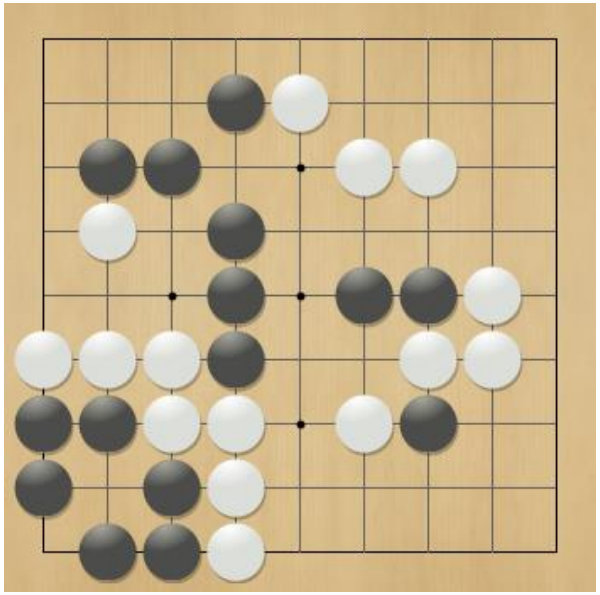
- The more simulations the more accurate

Garry Kasparov vs. Deep Blue (1997)

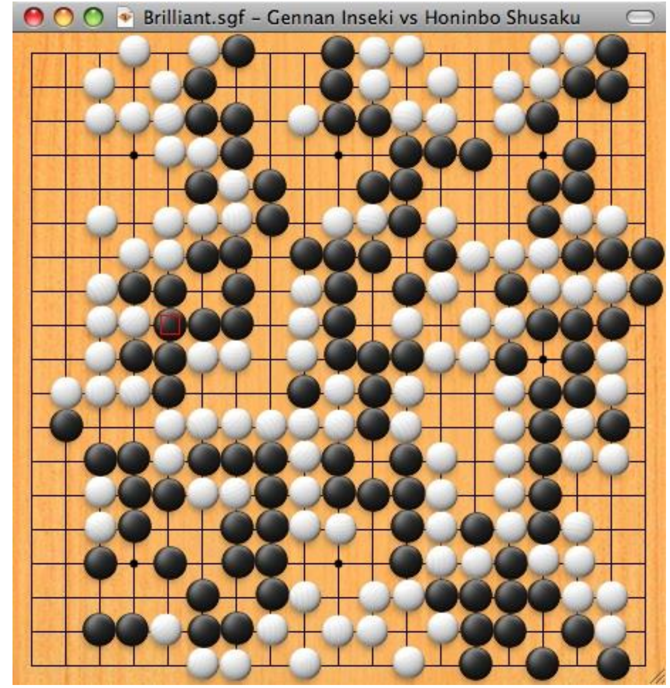
Deep Mind’s AlphaGo vs. Lee Sedol (2016)

Watson vs. Ken Jennings (2011)

Computer Go



9x9 (smallest board)



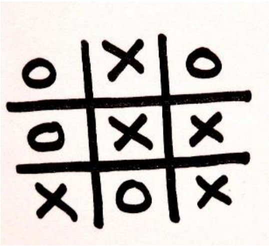
19x19 (largest board)

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

A Brief History of Computer Go

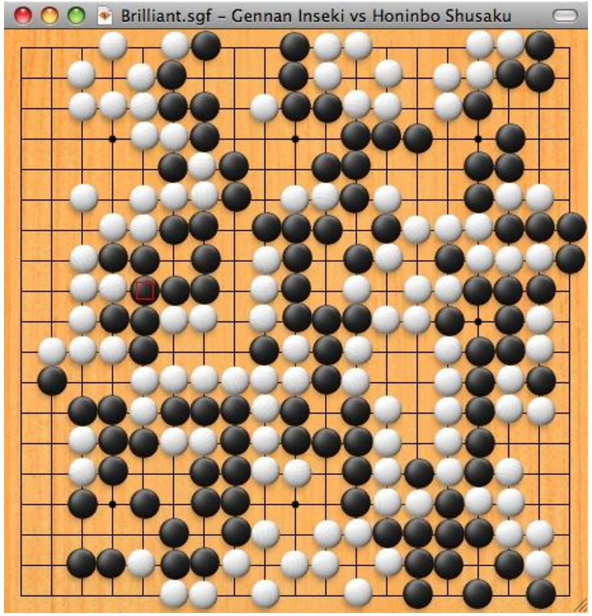
- *1997:* Super human Chess w/ Alpha-Beta + Fast Computer
- *2005*: Computer Go is impossible!

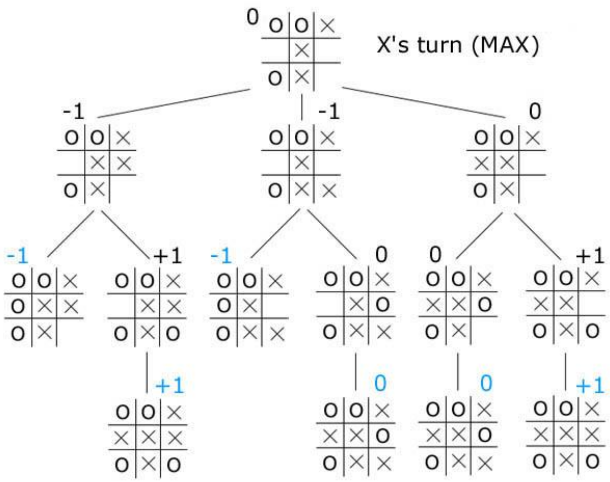
Why?



VS

VS





MiniMax Tree

- Branching Factor
  - Chess ≈ 35
  - Go ≈ 250
- Required search depth
  - Chess ≈ 14
  - Go ≈ much larger
- Leaf Evaluation Function
  - Chess – good hand-coded function
  - Go – no good hand-coded function

A Brief History of Computer Go

- *1997:* Super human Chess w/ Alpha-Beta + Fast Computer
- *2005*: Computer Go is impossible!
- *2006*: Monte-Carlo Tree Search applied to 9x9 Go (bit of learning)
- *2007*: Human master level achieved at 9x9 Go (bit more learning)
- *2008*: Human grandmaster level achieved at 9x9 Go (even more)

Computer GO Server rating over this period:
1800 ELO → 2600 ELO

- *2012*: Zen program beats former international champion Takemiya Masaki with only 4 stone handicap in 19x19
- ***2015: DeepMind's AlphaGo Defeats European Champion 5-0 (lots of learning)***
- ***2016: AlphaGo Defeats Go Legend Lee Sedol 4-1 (lots more learning)***

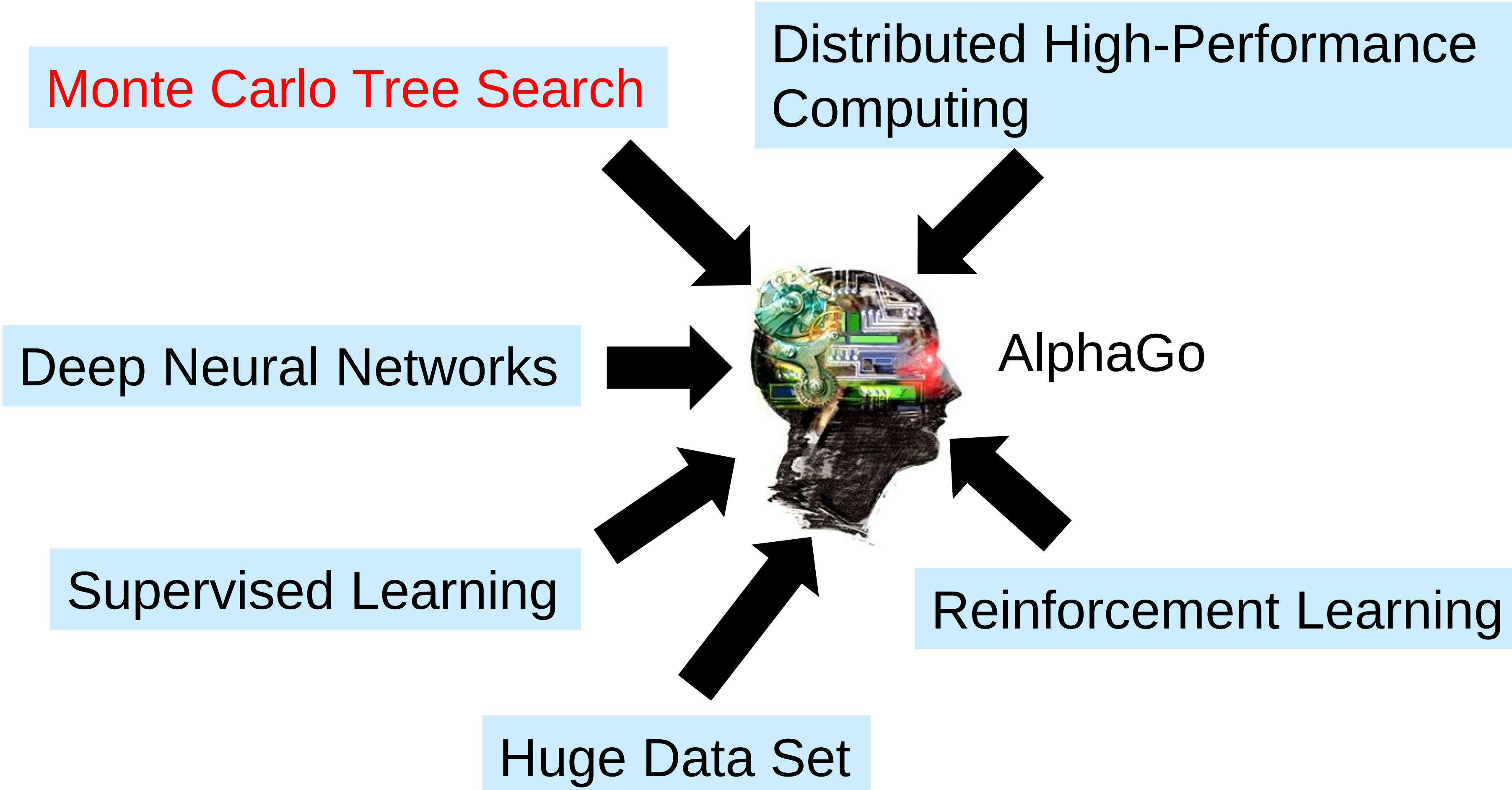
AlphaGo

- Deep Learning + Monte Carlo Tree Search + HPC
- Learn from 30 million expert moves and self play
- Highly parallel search implementation
- 48 CPUs, 8 GPUs (scaling to 1,202 CPUs, 176 GPUs)

Image: AP/ Lee Jin-man

March 2016 :
AlphaGo beats Lee Sedol 4-1

Arsenal of AlphaGo

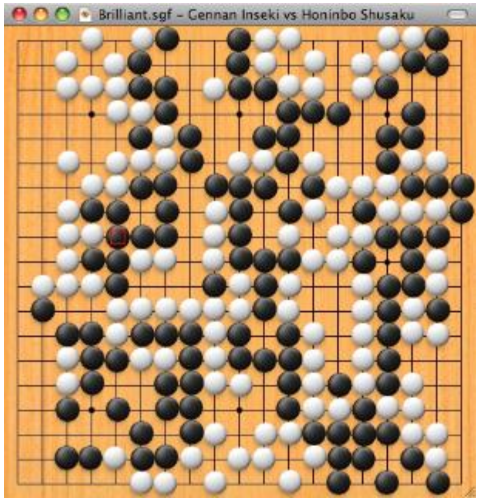


Mastering the game of Go with deep neural networks and tree search
Nature, 529, January 2016.

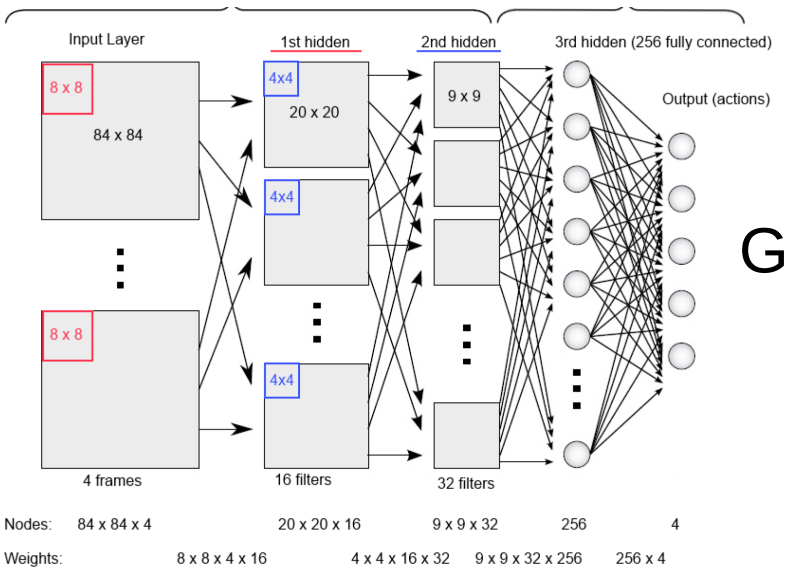
Deep Neural Networks

State-of-the-Art Performance: very fast GPU implementations allow training giant networks (millions of parameters) on massive data sets

Could a Deep NN learn to predict expert Go moves by looking at board position? Yes!



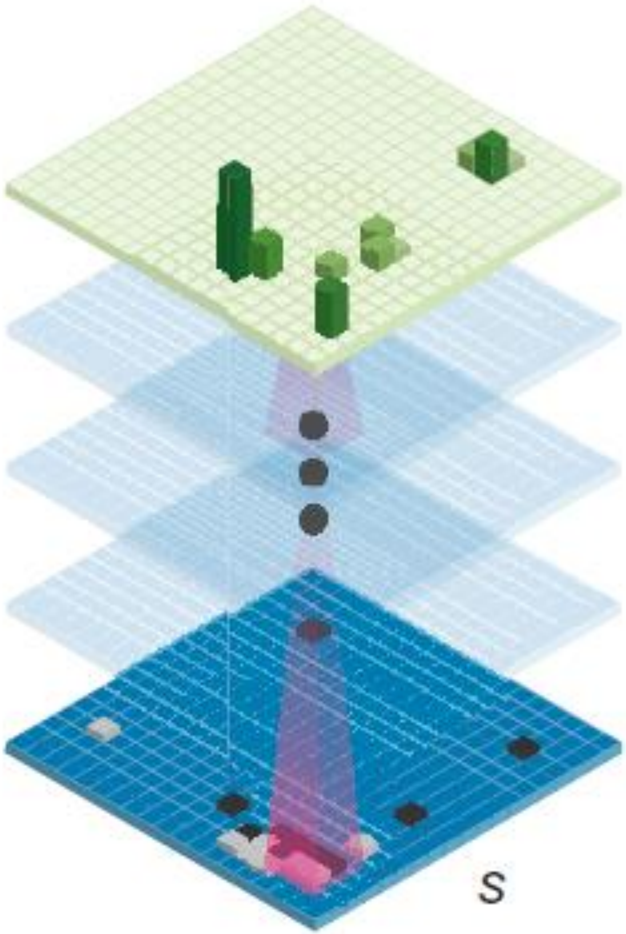
Deep Neural Network



Go Move

Supervised Learning for Go

Output: probability of each move



Deep NN Internal Layers

Input: Board Position

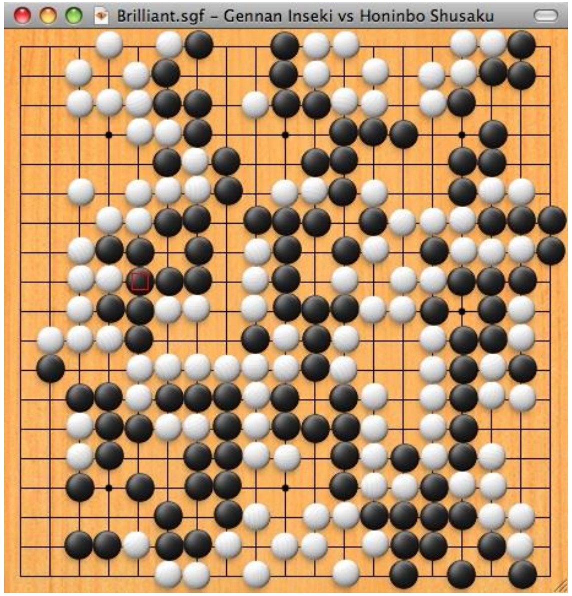
Trained for 3 weeks on 30 million expert moves

- 57% prediction accuracy!

Playing strength further improved via reinforcement learning

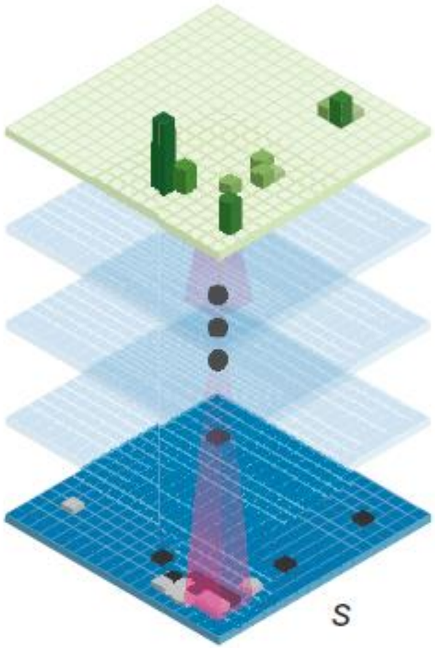
Monte Carlo Tree Search

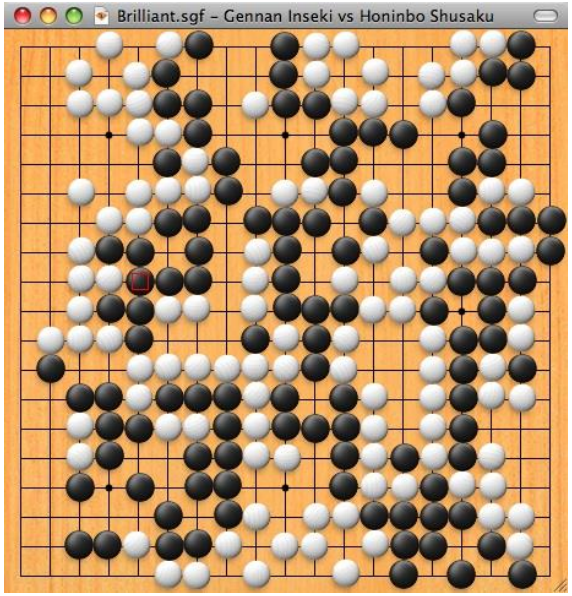
Idea: use deep NN for rollout evaluation

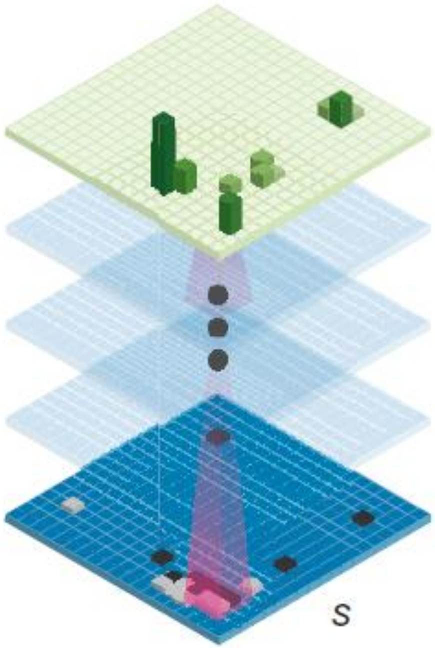


Monte Carlo Tree Search

Idea: use deep NN for rollout evaluation

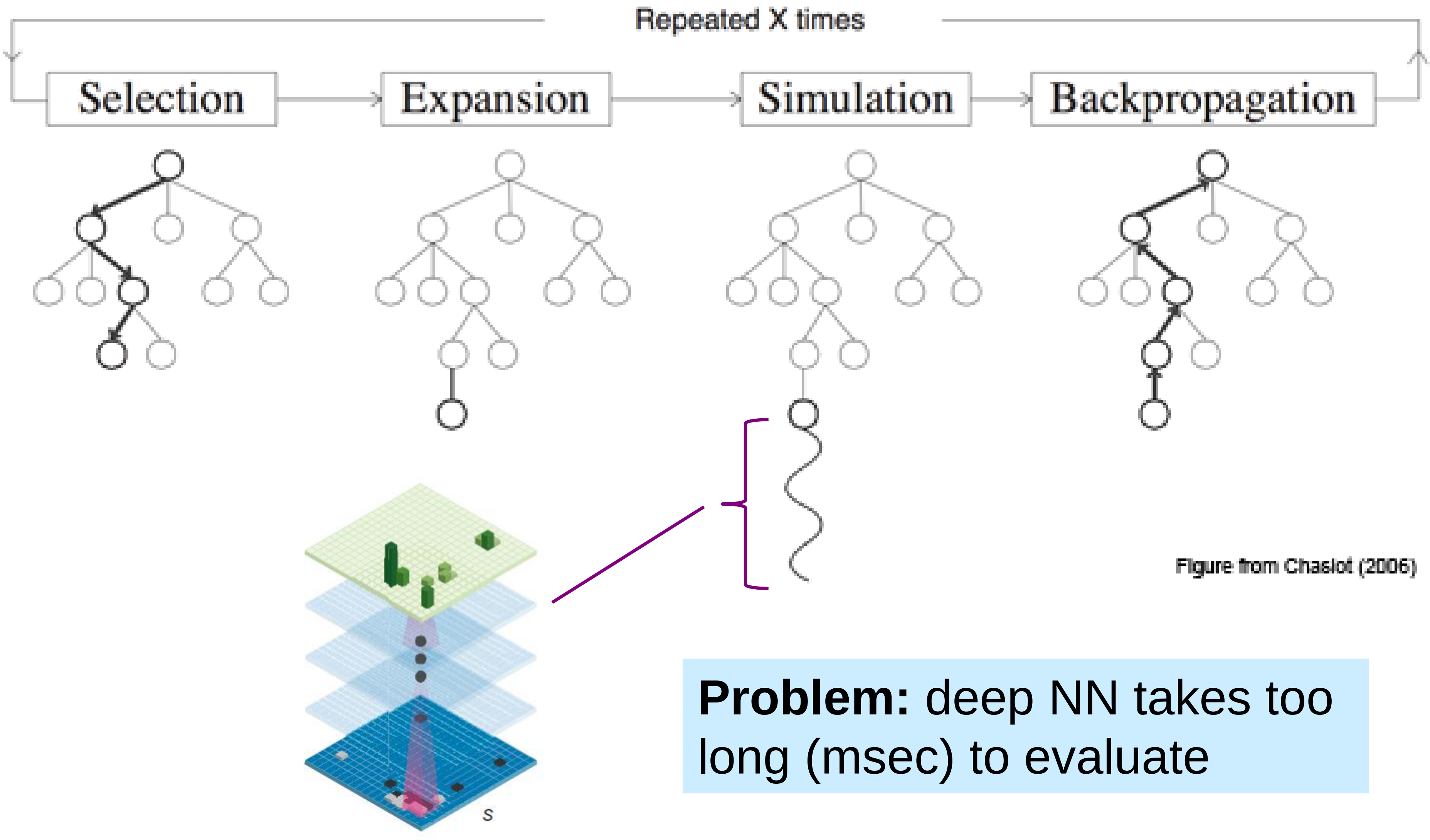






Monte Carlo Tree Search

Idea: use Deep NN for rollouts in Monte Carlo Tree Search



Problem: deep NN takes too long (msec) to evaluate

Monte Carlo Tree Search

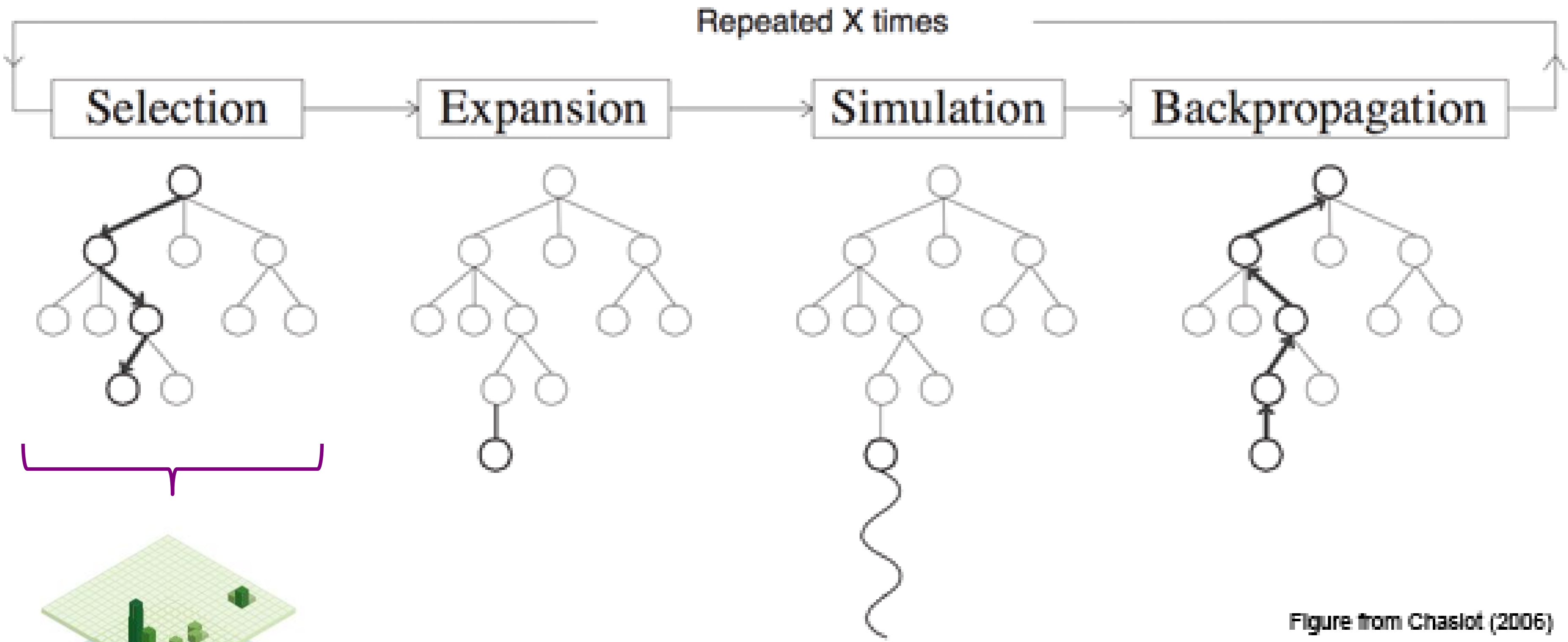
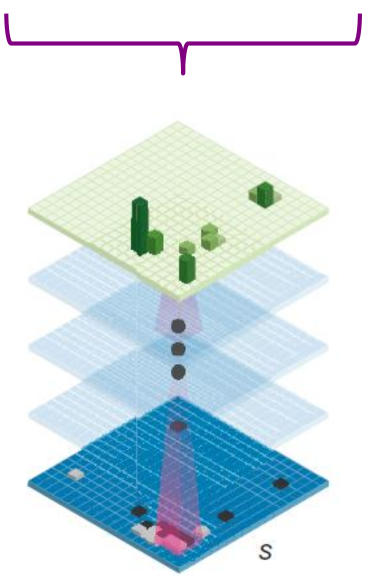


Figure from Chaslot (2006)



Solution: use deep NN to define tree policy in
- Evaluate once per tree node
- Use probabilities to bias search toward actions that look good to deep NN

Monte Carlo Tree Search

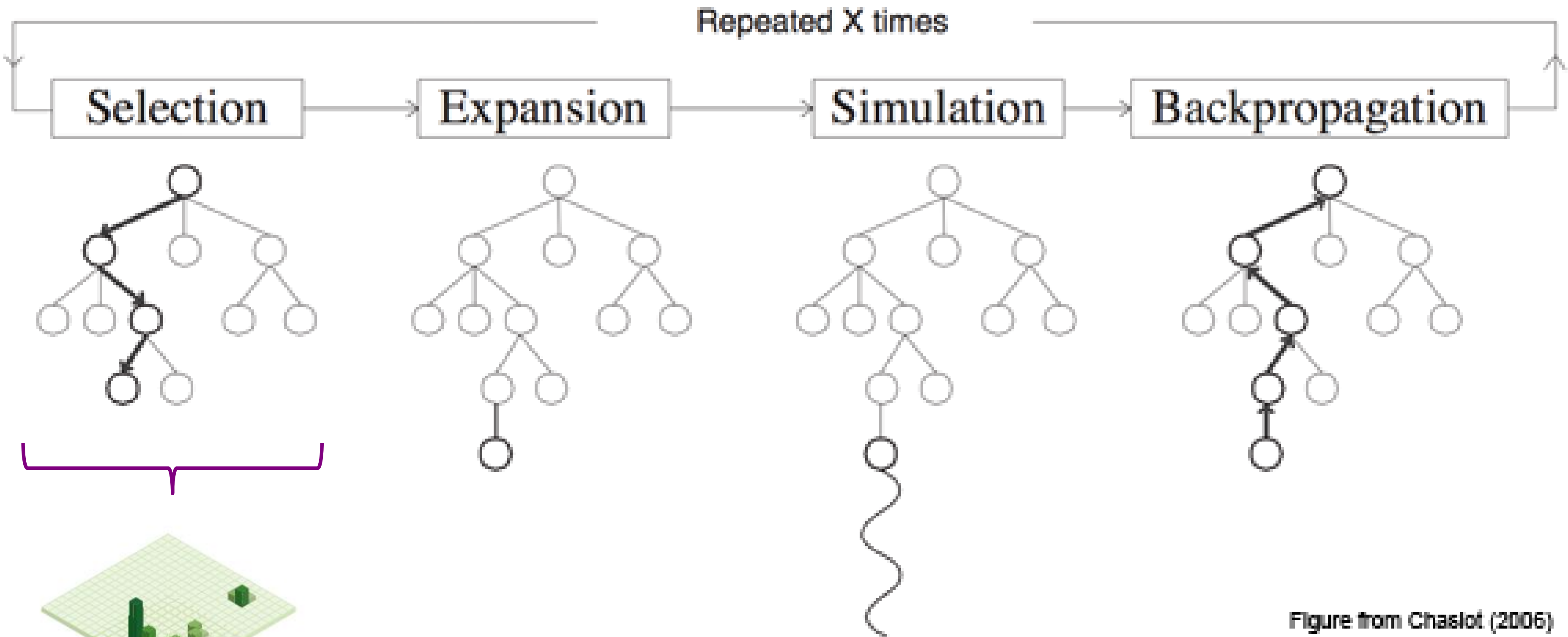
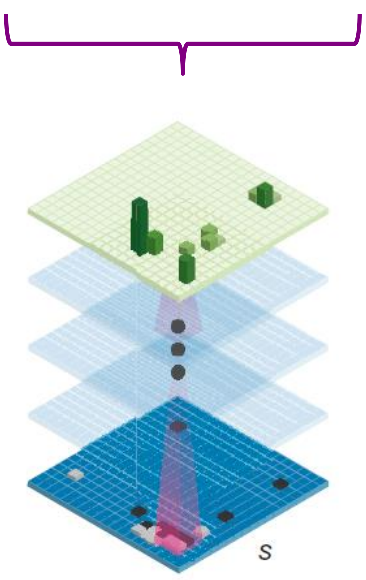


Figure from Chaslot (2006)

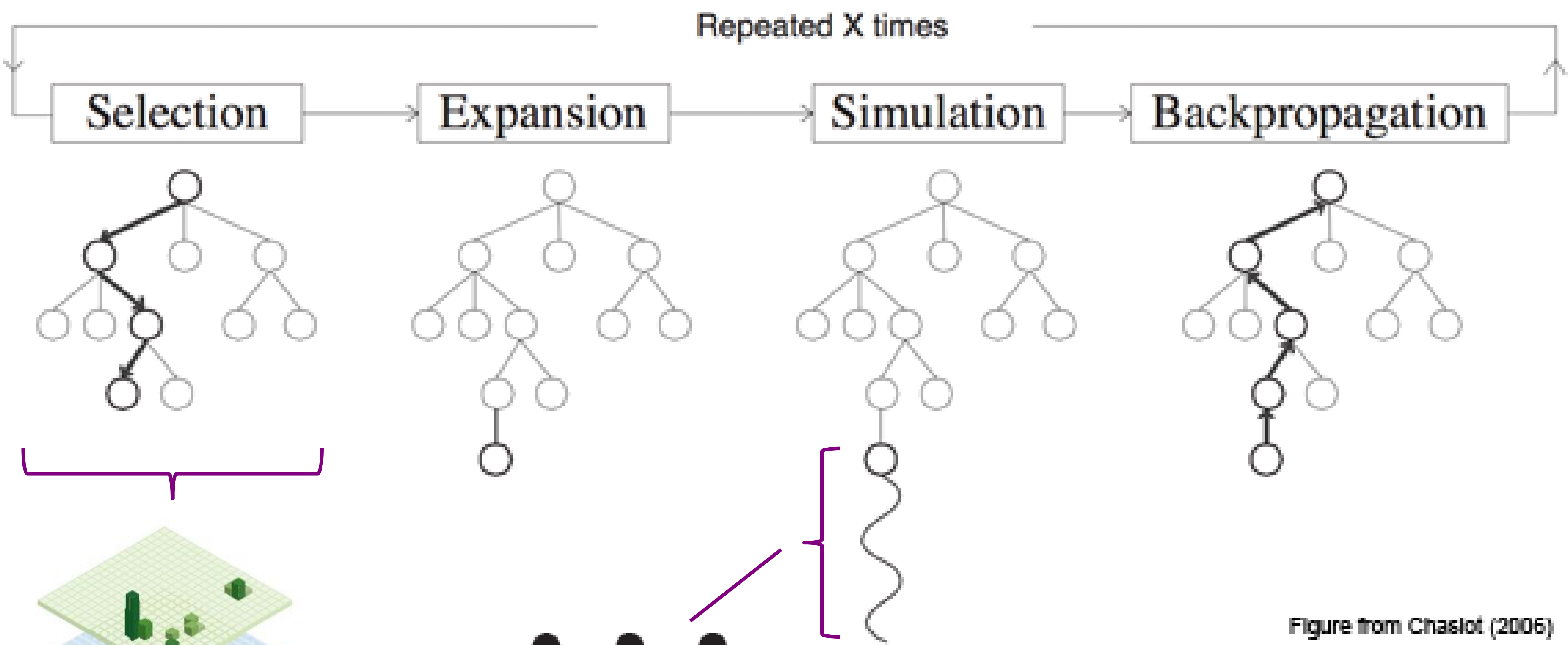


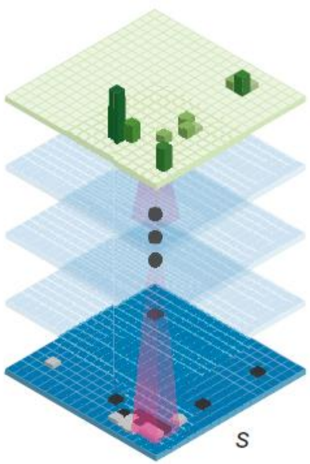
AlphaGo Tree Policy:

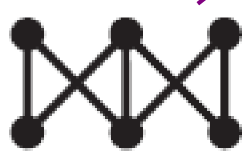
$$\arg\max_a Q(s,a) + c\,\frac{P(s,a)}{1+n(s,a)}$$

$P(s,a)$ probability of action from NN

Monte Carlo Tree Search



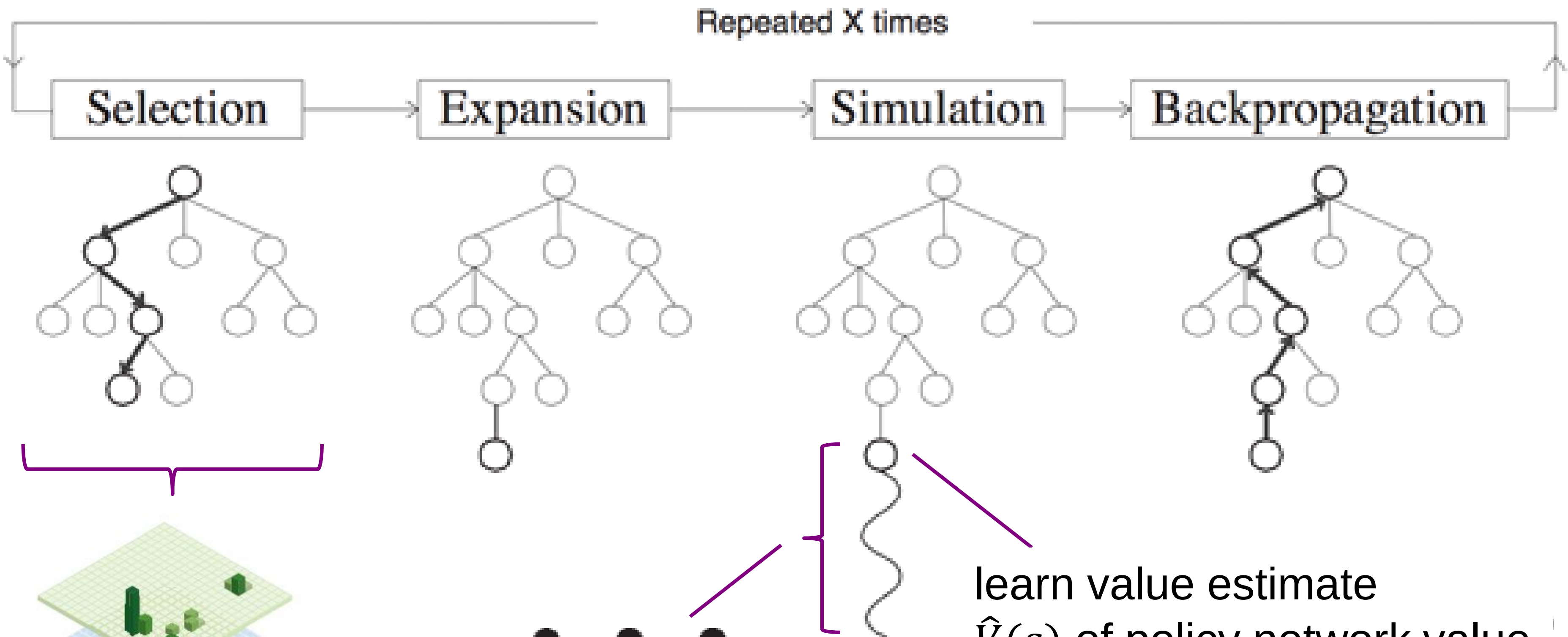


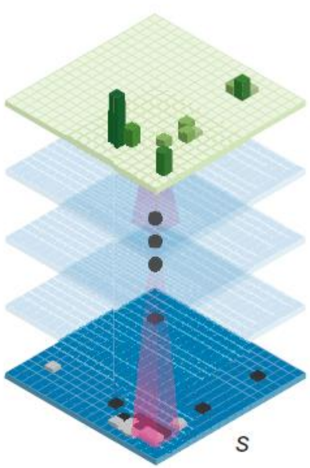


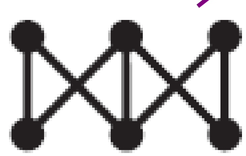
Solution Part 1: train smaller network for rollout

- Less accurate but much faster

Monte Carlo Tree Search







learn value estimate
$\hat{V}(s)$ of policy network value

Solution Part 2: learn state value estimator $\hat{\boldsymbol{V}}(s)$

- leaf evaluation combines rollout and $\hat{\boldsymbol{V}}(s)$

Other Successes

- Klondike Solitaire (wins 40% of games)
- General Game Playing Competition
- Probabilistic Planning Competition
- Real-Time Strategy Games
- Combinatorial Optimization
- Active Learning
- Computer Vision

- List is growing

- Usually extend UCT is some ways

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 and switching are great way to exploit existing policies and make them better
- If a good heuristic exists, then shallow sparse sampling can give good results
- UCT is often quite effective especially when combined with domain knowledge