Simulation-Based Planning II

Alan Fern

* Based in part on slides by David Silver, Robert Givan, and Csaba Szepesvari

Simulation-Based Look-Ahead Trees

- The approaches we have discussed do not guarantee optimality or even near optimality

- Can we develop simulation-based methods that give us near optimal policies?

- 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?

- We will consider two methods for building such trees: **sparse sampling** and **UCT**
  - Both methods have strong theoretical guarantees of near optimality
  - UCT has produced impressive empirical results

Online Planning with Look-Ahead Trees

- At each state we encounter in the environment we build a look-ahead tree and use it to estimate Q-values of each action

```
s = current state
Repeat until terminal state
    T = BuildLookAheadTree(s) ;; sparse sampling or UCT
    a = BestRootAction(T)   ;; action with best Q-value
    Execute action a in environment
    s is the resulting state
```

Sparse Sampling

- Focus on finite-horizons
  - Arbitrarily good approximation for large enough horizon *h*

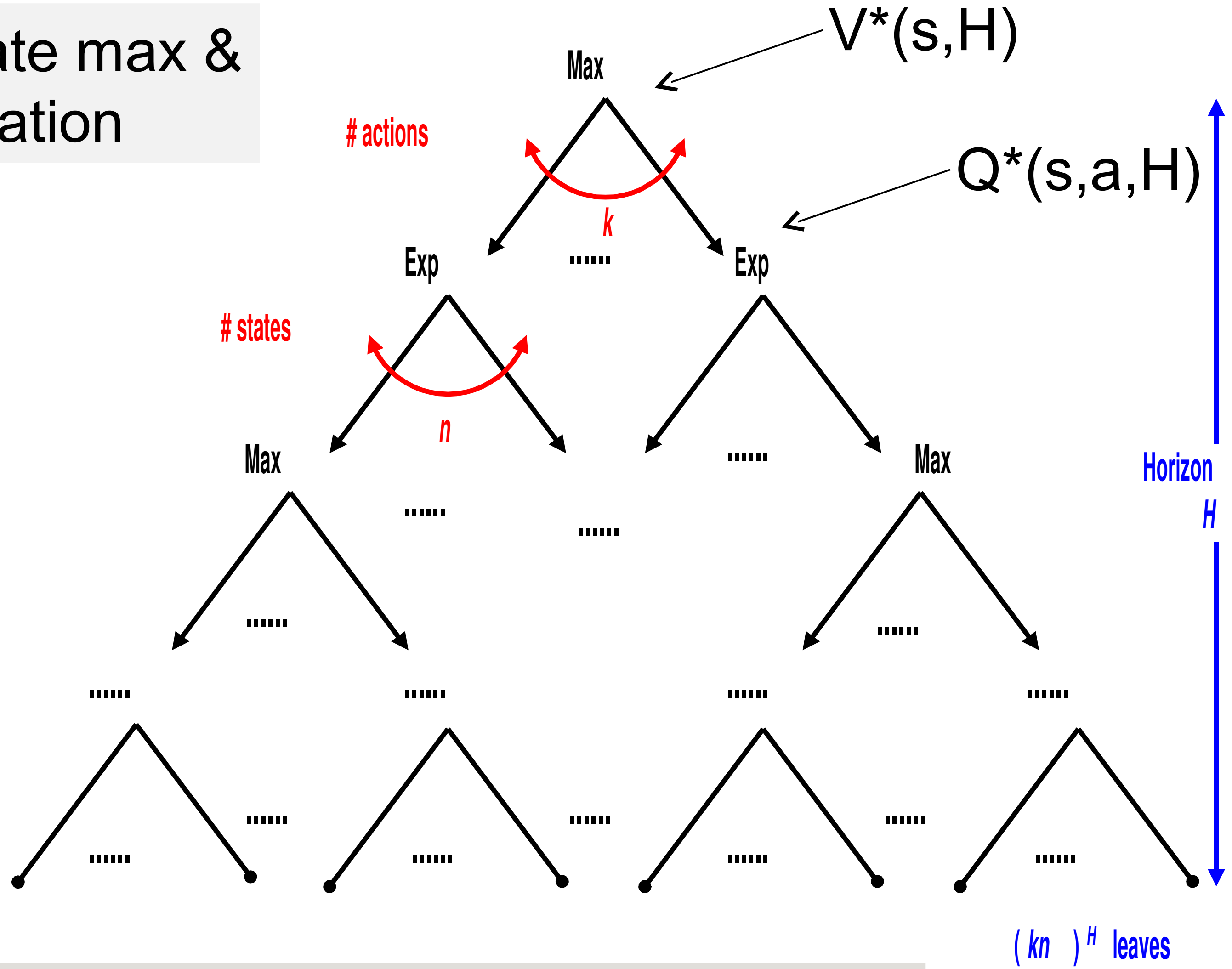
- Define $Q^*(s,a,h) = R(s,a) + E[V^*(s',h-1)]$
  - Optimal h-horizon value of action *a* at state *s.*
  - Q-value of action *a* at state *s*

- 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

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

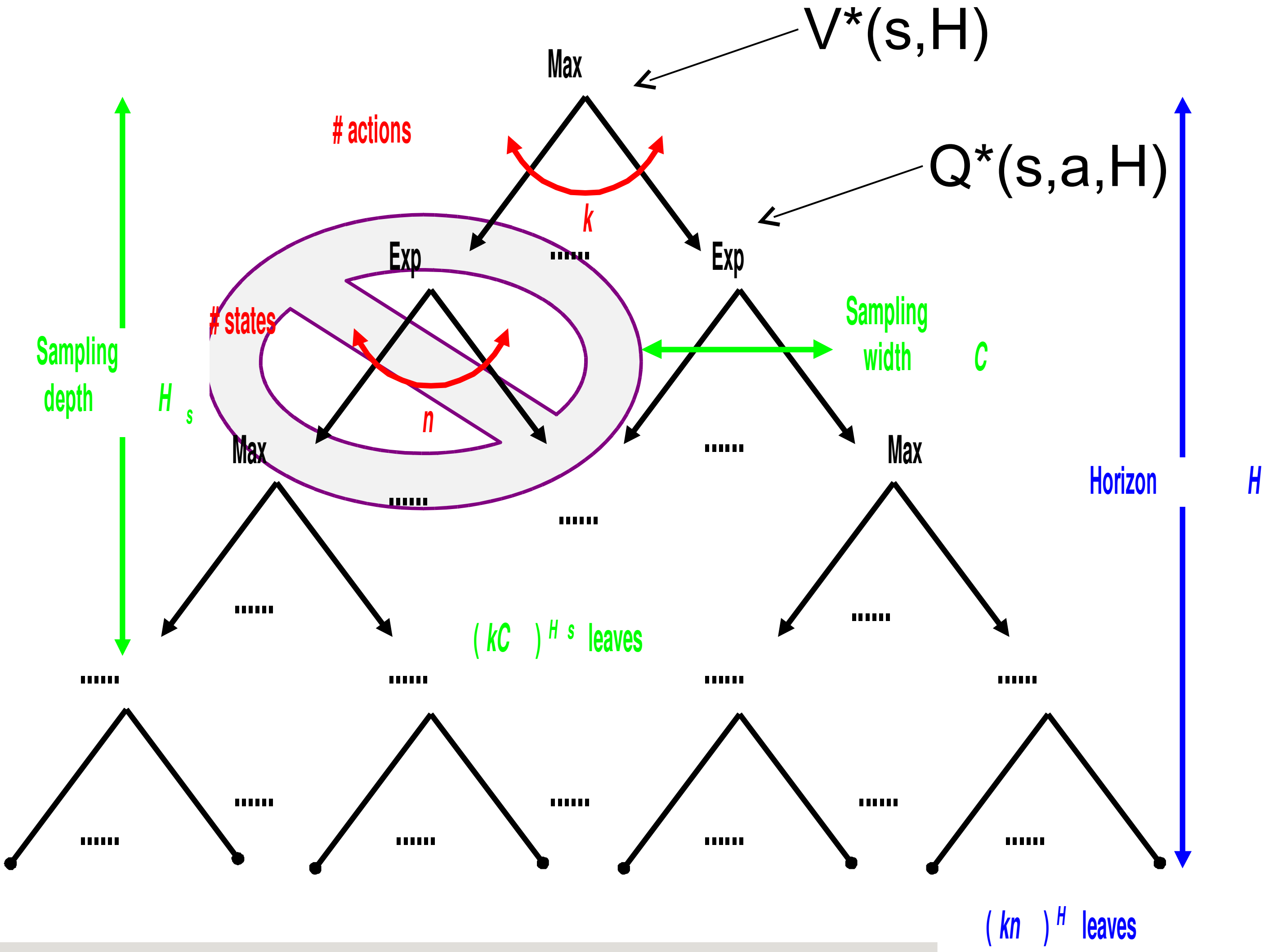
Alternate max & expectation



Compute root V* and Q* via recursive procedure

Depends on size of the state-space. Bad!

Sparse Sampling Tree



Replace expectation with average over *C* samples

C will typically be much smaller than *n.*

Sparse Sampling Pseudocode

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

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

Sparse Sampling (Cont’d)

- For a given desired accuracy, how large should sampling width and depth be?
  - Answered: Kearns, Mansour, and Ng (1999)

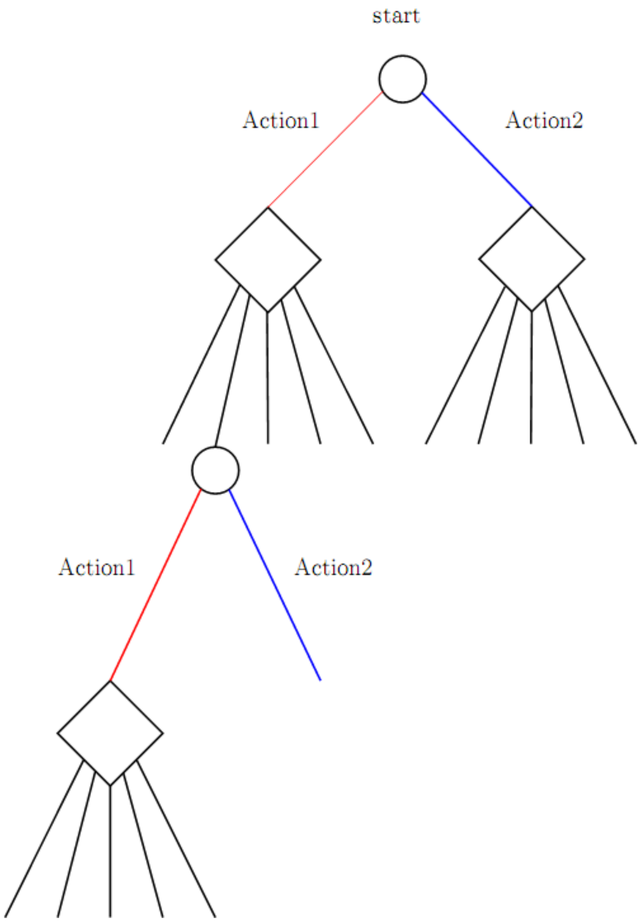
- **Good news:** give values for C and H_s to achieve policy arbitrarily close to optimal
  - Values are independent of state-space size!
  - First near-optimal general MDP planning algorithm whose runtime didn’t depend on size of state-space

- **Bad news:** the theoretical values are typically still intractably large---also exponential in H_s

- **In practice:** use small H_s and use heuristic at leaves (similar to minimax game-tree search)

Idea..

- Sparse sampling wastes time on bad parts of 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?
  - Breadth-first ➔ Depth-first!
  - Monte-Carlo Tree Search



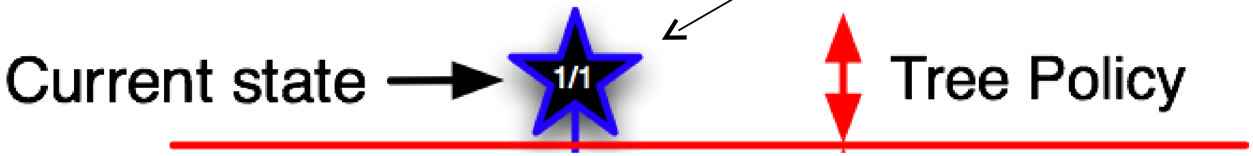
Monte-Carlo Tree Search

- Builds a search tree from current state by repeatedly simulating a special rollout policy from current state until reaching a terminal state
  - Each simulation adds one or more nodes to the current tree and updates value estimates of current nodes
- The rollout policy has two phases:
  - Tree policy (e.g. greedy) when in states already in tree
  - Default policy (e.g. uniform random) when arriving a states not yet in current tree

Initially tree is empty.

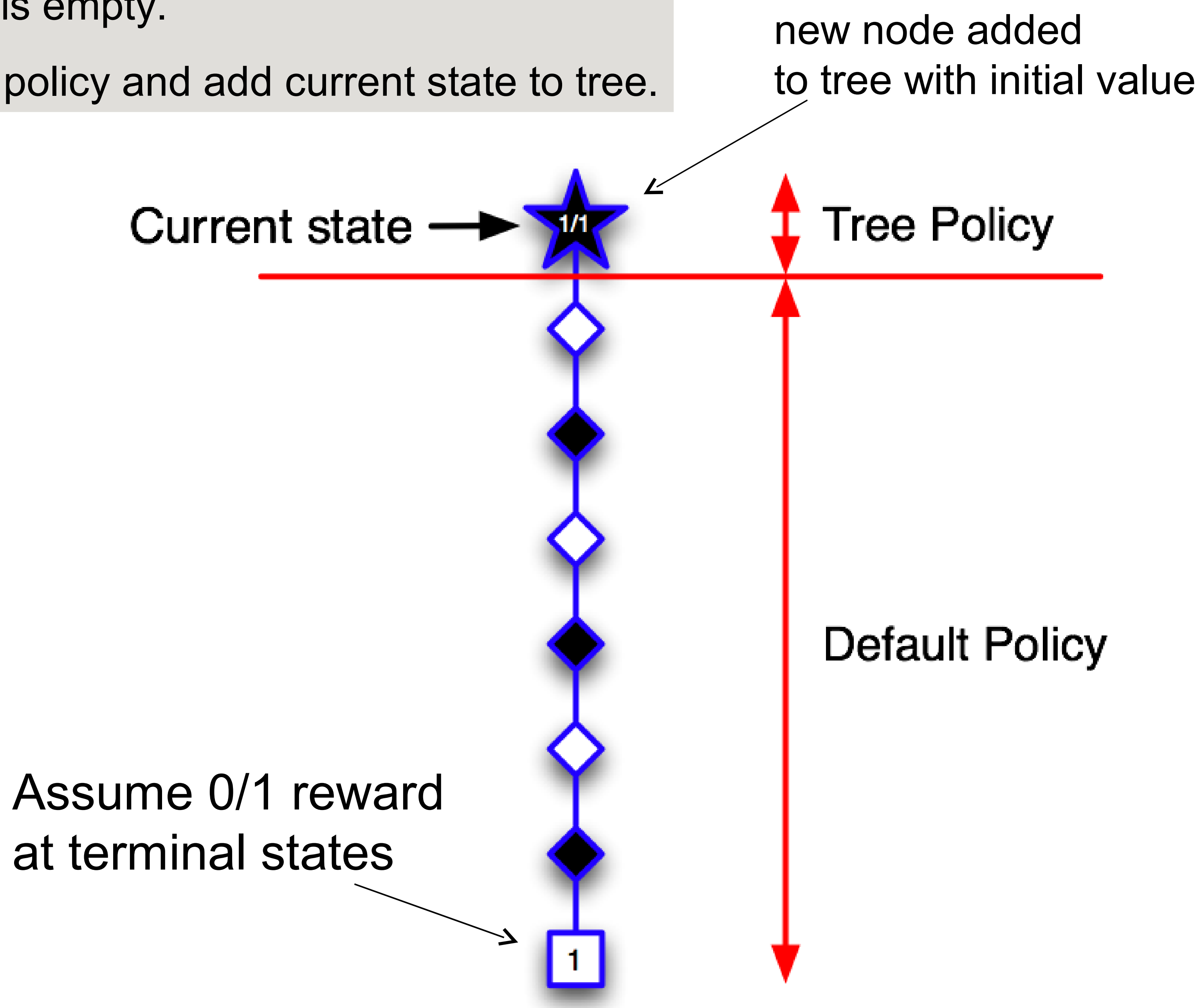
Run default policy and add current state to tree.

new node added
to tree with initial value



Initially tree is empty.

Run default policy and add current state to tree.



new node added to tree with initial value

Current state

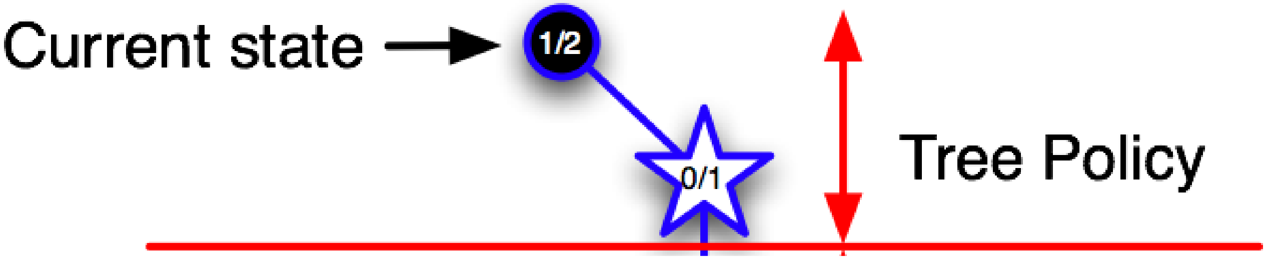
Tree Policy

Default Policy

Assume 0/1 reward at terminal states

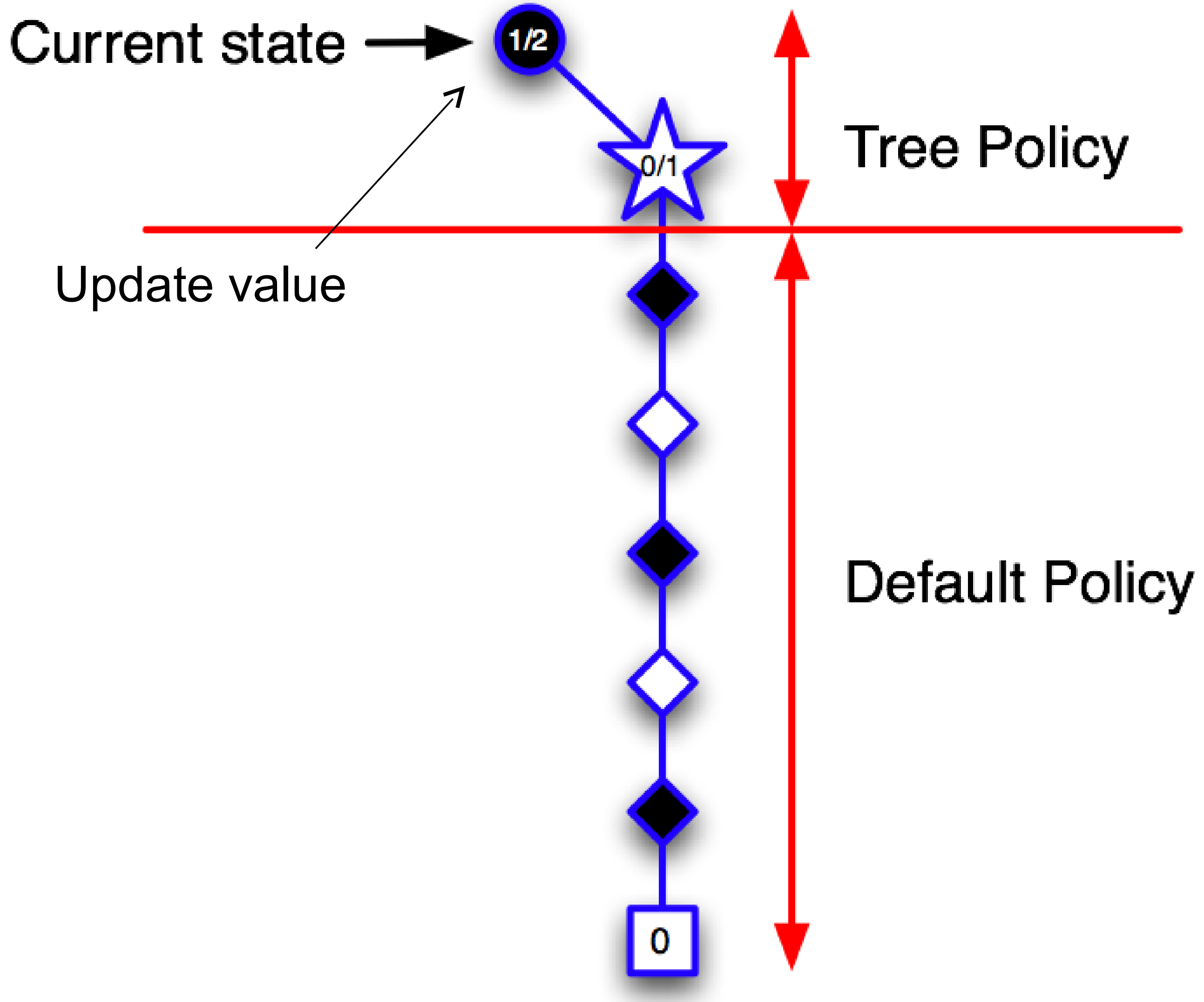
Use tree policy to select action from initial node

Results in new state (the star) after which following default policy.



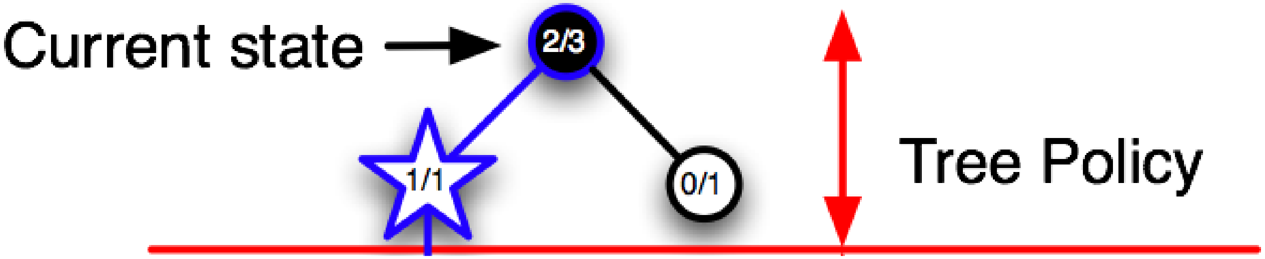
Use tree policy to select action from initial node

Results in new state (the star) after which following default policy.



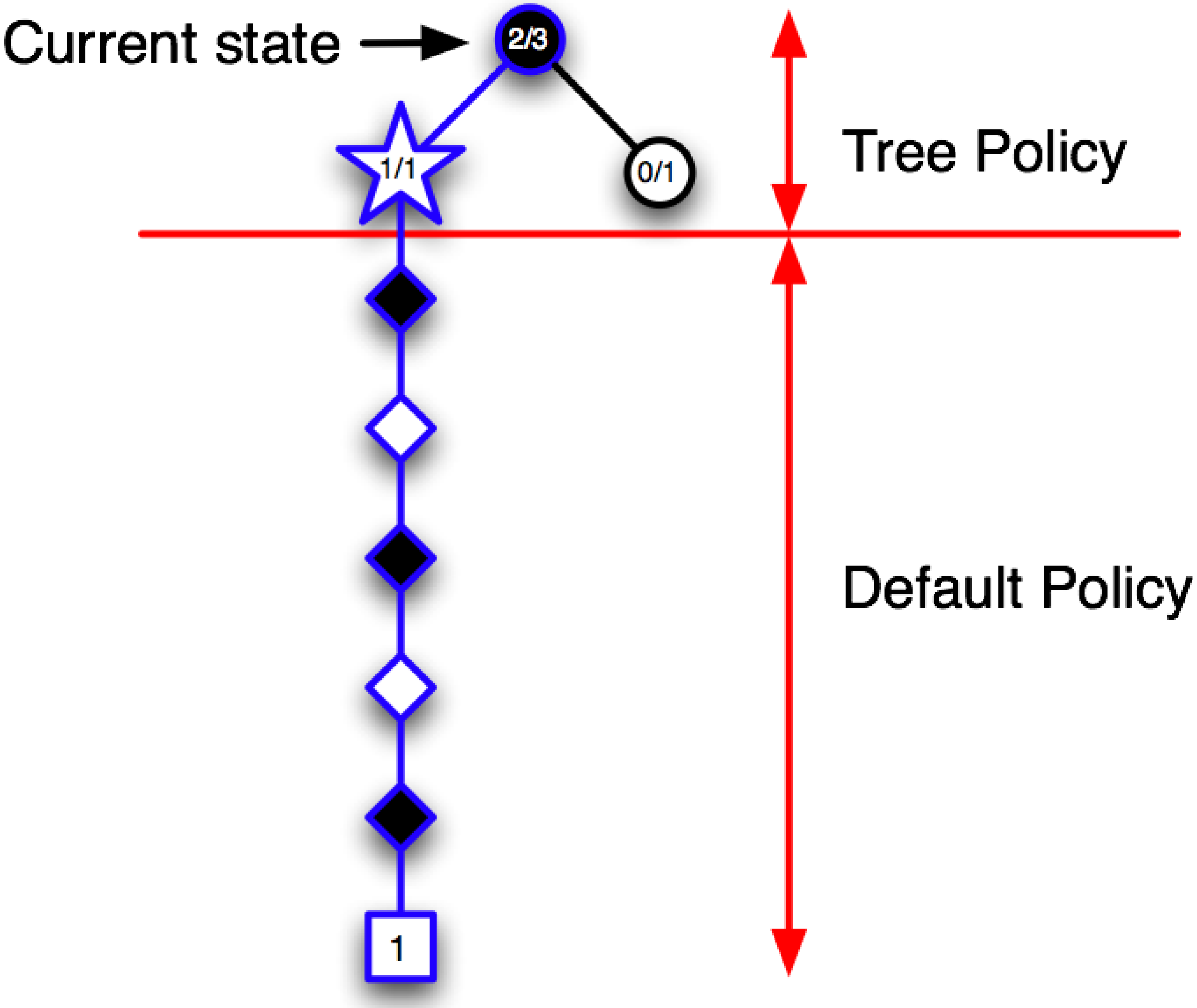
Tree selects different action from root.

Results in new state (the star) after which following default policy.

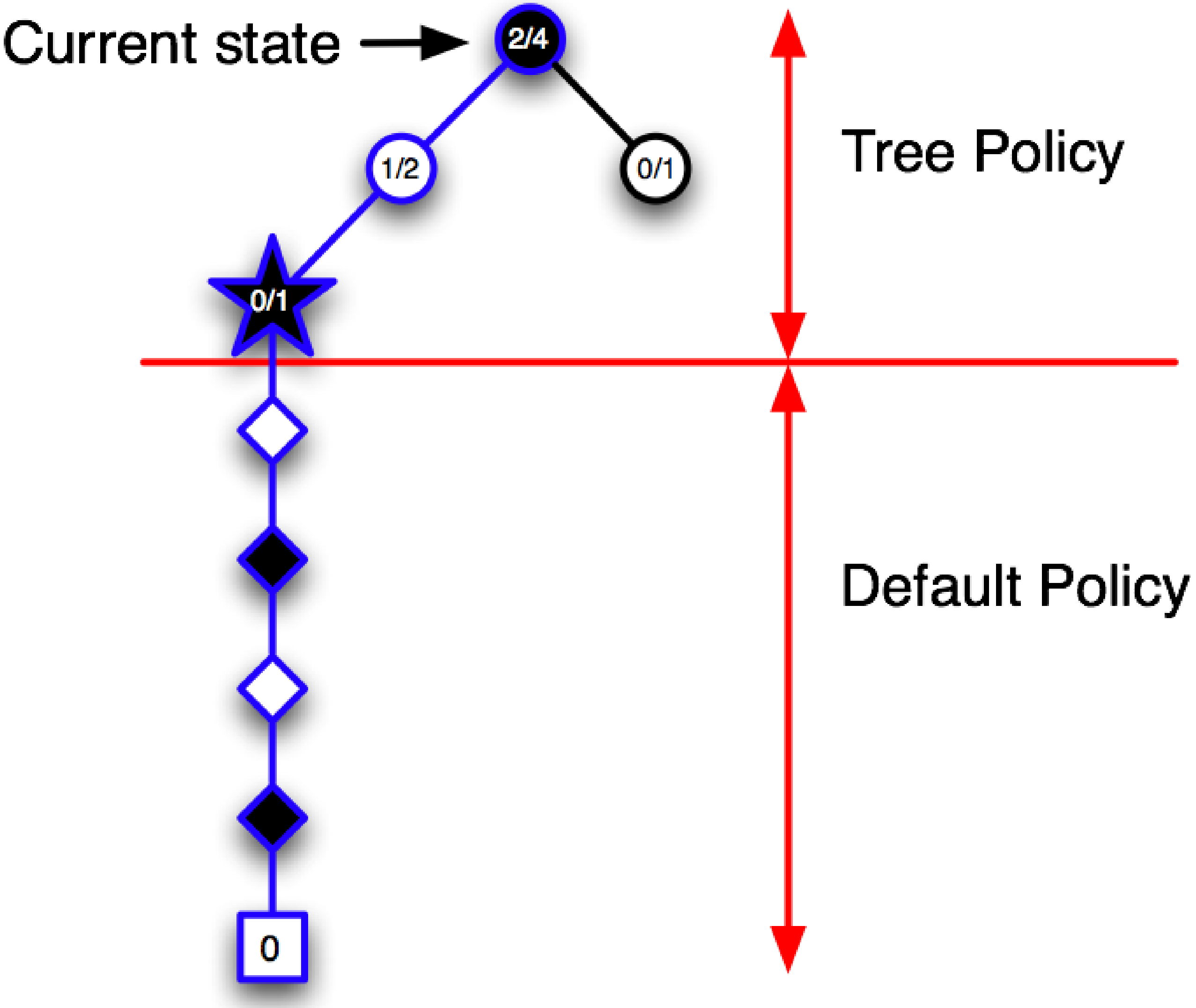


Tree selects different action from root.

Results in new state (the star) after which following default policy.

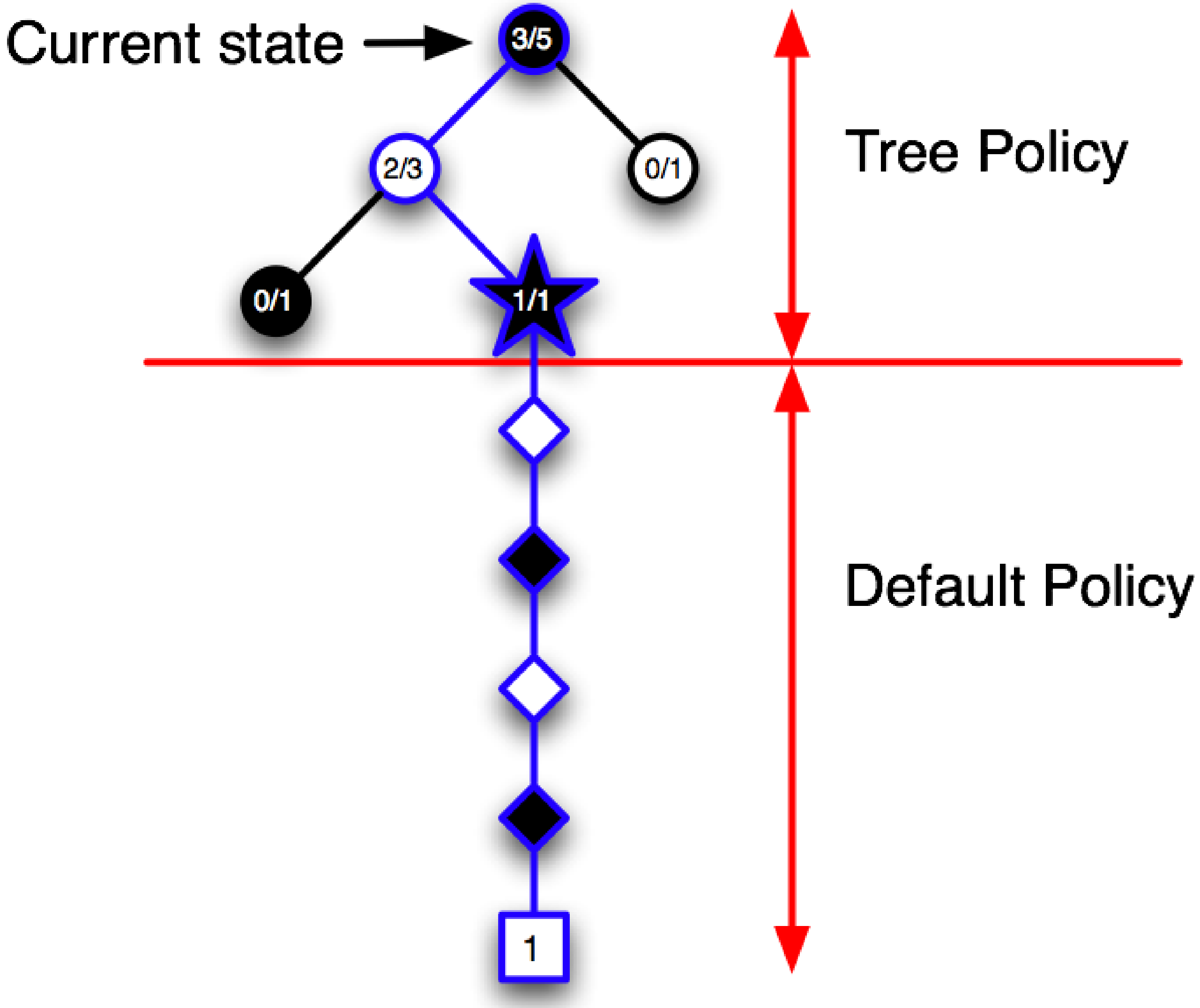


Fourth Simulation



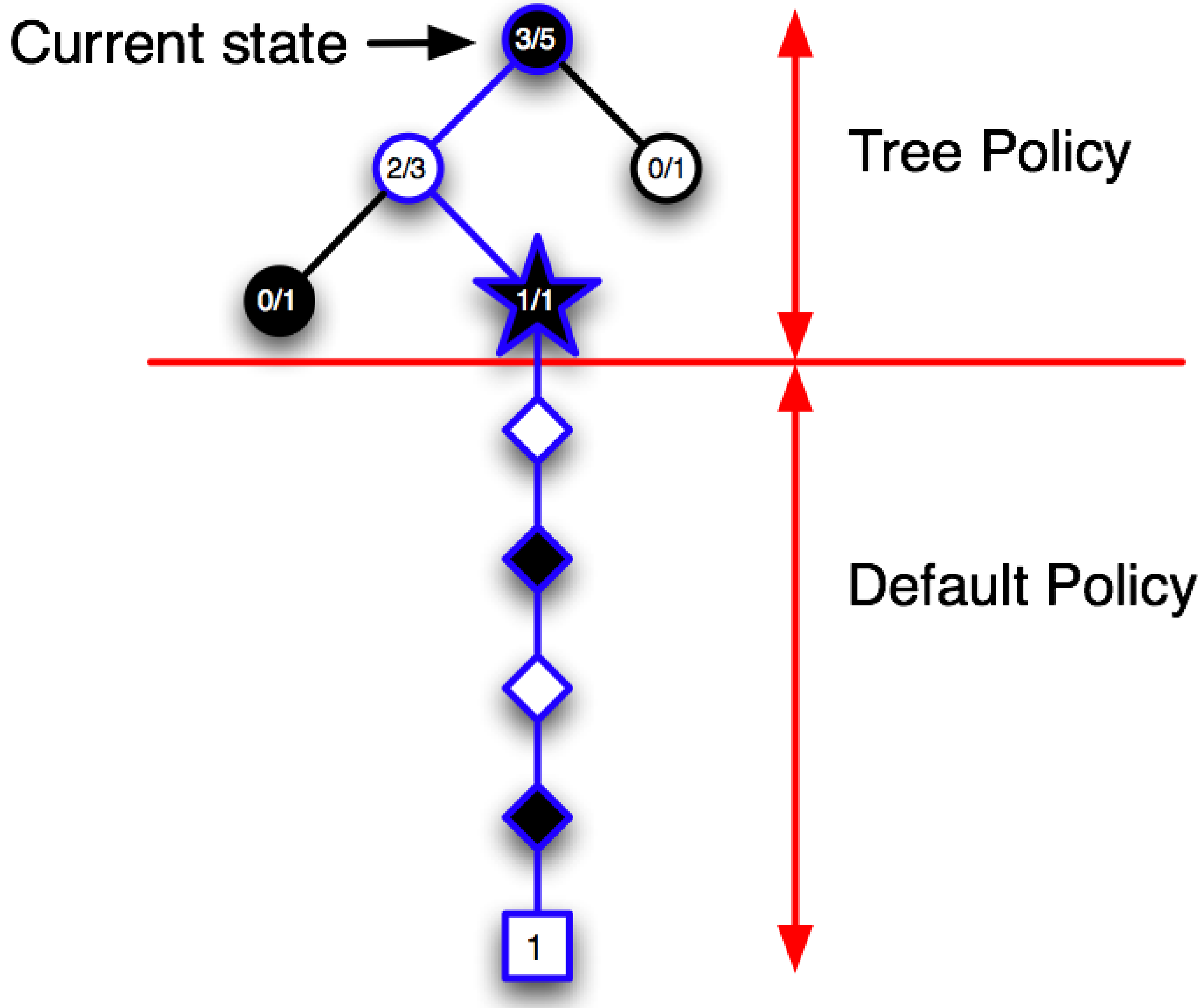
Continue building tree for N simulations

Select action from root that has highest value (here 2/3)



Continue building tree for N simulations

Select action from root that has highest value (here 2/3)



UCT Algorithm [Kocsis & Szepesvari, 2006]

- What tree and default policy should we use?
  - Balance exploration/exploitation.

- The UCT algorithm
  - **Default policy:** uniform random action selection
  - **Tree policy:** inspired by UCB multi-armed bandit algorithm

- Provides theoretical guarantees of near-optimality

Aside: Multi-Armed Bandit Problem

- There are a finite number of (slot) machines each with a single arm to pull
  - Each machine has an unknown expected payoff
- **Problem (roughly stated):** select arms to pull so as to do about as well as always selecting the best arm
  - Balance **exploring** machines to find good payoffs and **exploiting** current knowledge
- UCT is based on UCB
  an algorithm for multi-armed
  bandits

Aside: UCB Algorithm

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

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

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

- **Theorem**: The expected loss after *n* arm pulls compared to optimal behavior is bounded by O(log *n*)
- It has been shown that no algorithm can achieve a better loss rate

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

- How does this policy balance exploration and exploitation?

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

Value Term:
actions that have looked good historically will have a good Q-value and tend to be favored to actions w/ poor Q-value estimates

Exploration Term:
Actions that have not been tried many times relative to ln(n) will get a bonus from the exploration term and get selected even if the Q-value estimate is bad

The exploration term causes us to explore at just the right rate.

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

- UCT's tree policy treats each decision as a multi-armed bandit problem
- 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
- Tree policy of UCT (similar to UCB):

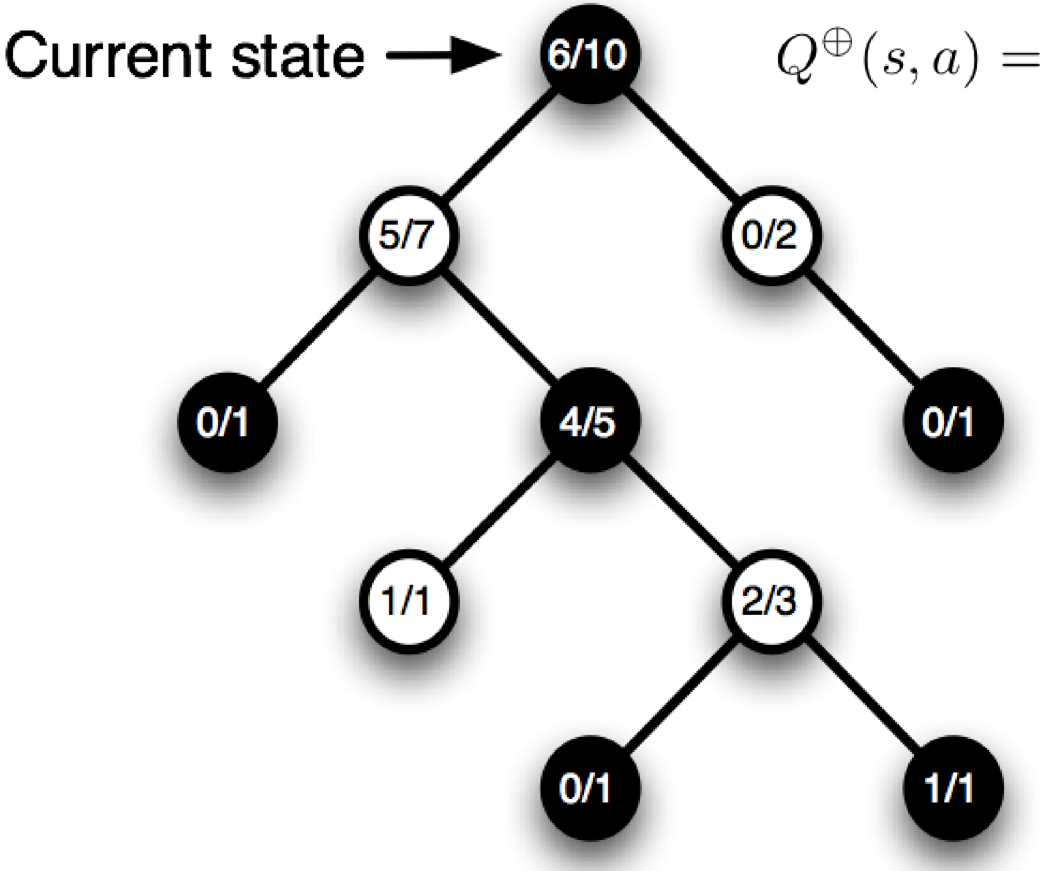
$$\pi^*(s) = \arg\max_a Q(s,a) + c\sqrt{\frac{\ln n(s)}{n(s,a)}}$$

Theoretical constant that must
be selected empirically in practice

UCT

Tree policy maximizes this value

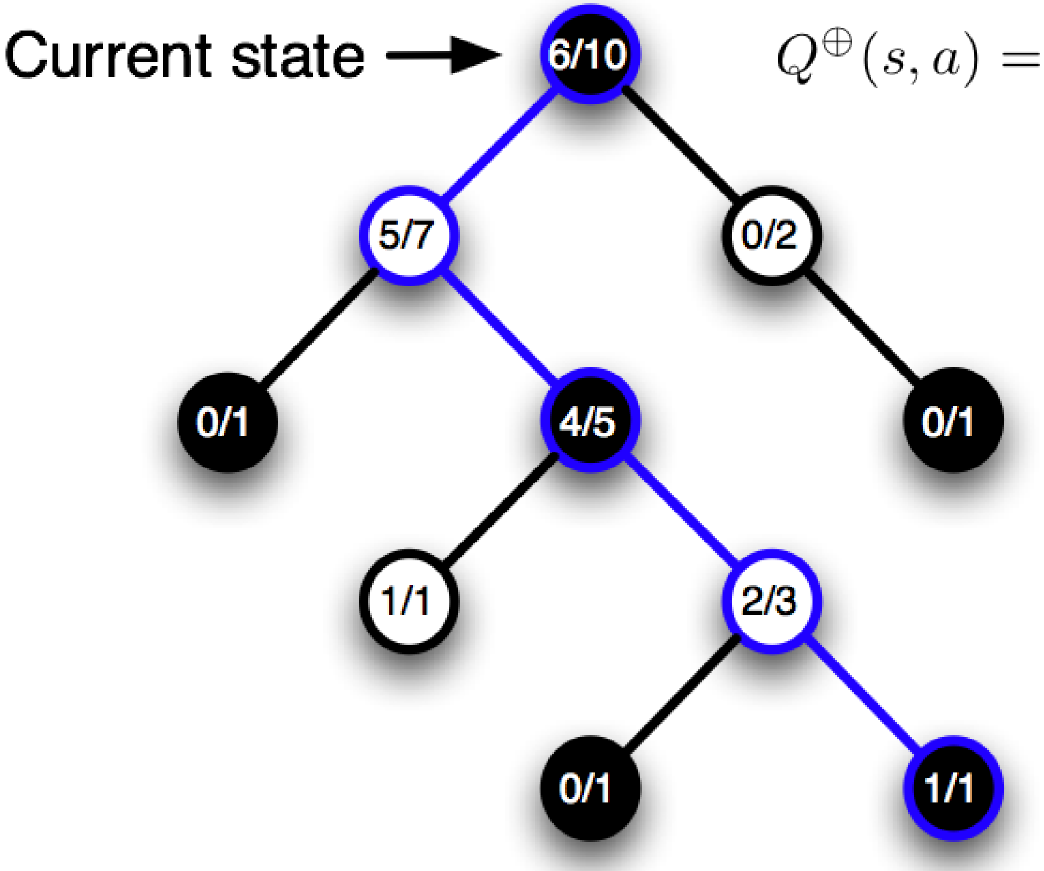
$$Q^{\oplus}(s,a) = Q(s,a) + c\sqrt{\frac{\log n(s)}{n(s,a)}}$$



Exploitation

Tree policy maximizes this value

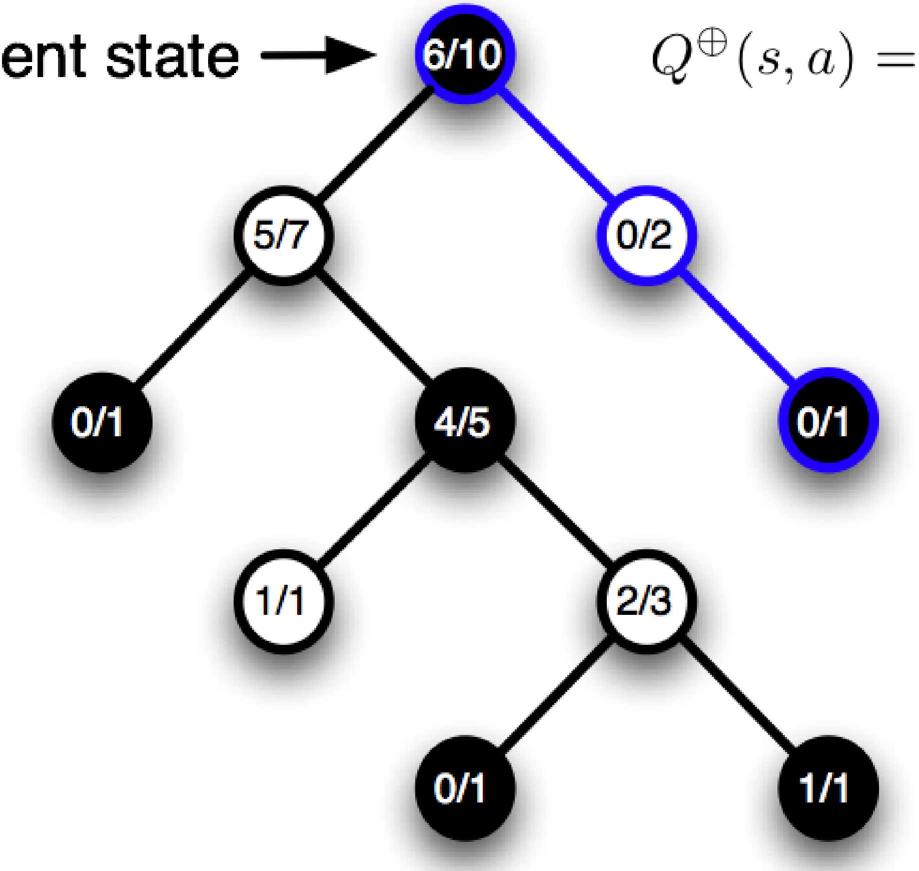
$$Q^{\oplus}(s,a) = \boxed{Q(s,a)} + c\sqrt{\frac{\log n(s)}{n(s,a)}}$$



Exploration

Tree policy maximizes this value

Current state →



$$Q^{\oplus}(s,a) = Q(s,a) + \boxed{c\sqrt{\frac{\log n(s)}{n(s,a)}}}$$

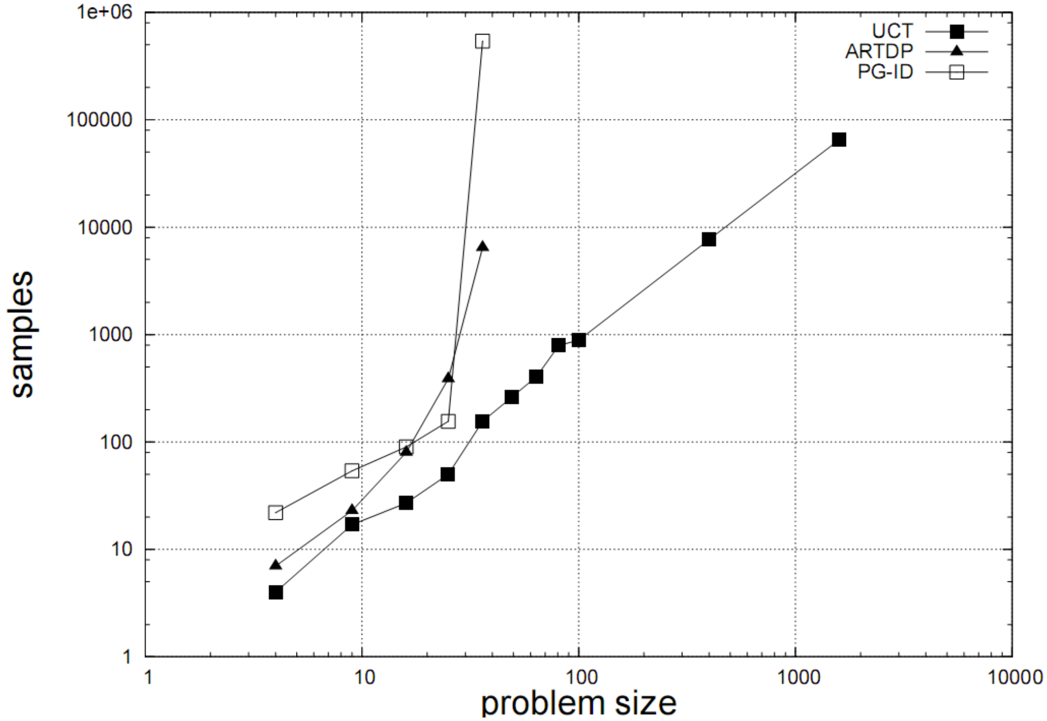
UCT Recap

- 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

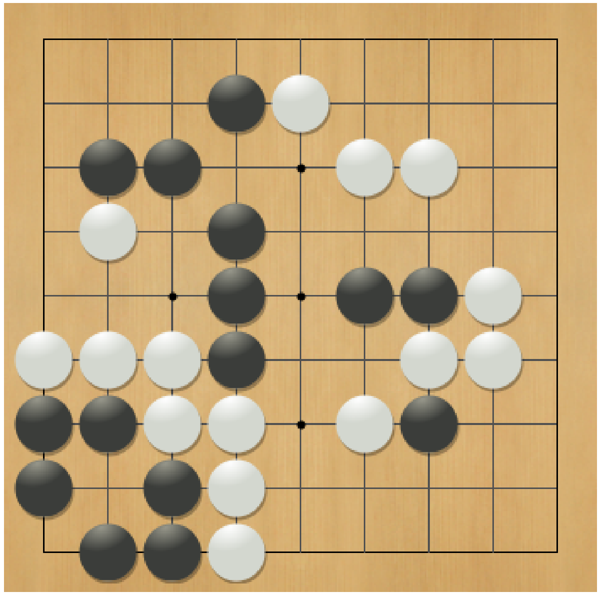
Results: Sailing

- ‘Sailing’: Stochastic shortest path

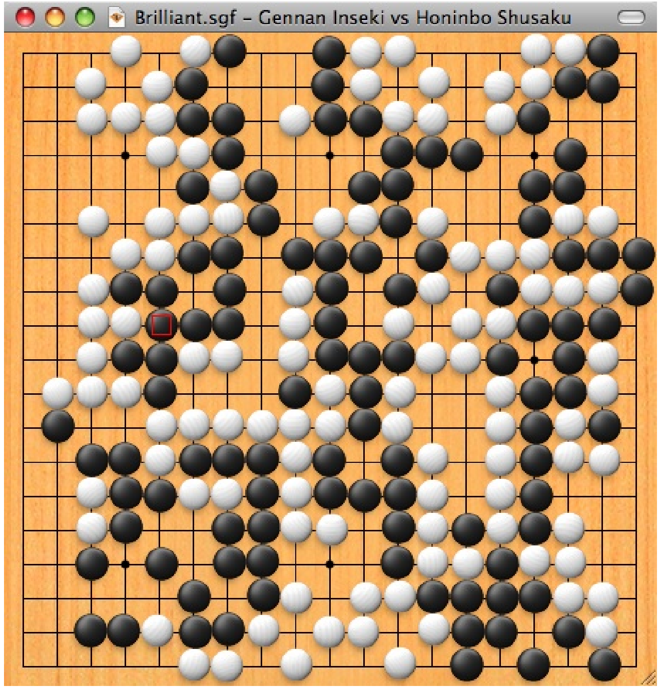


- Extension to two-player, full information games
- Major advances in go!

Computer Go



9x9 (smallest board)



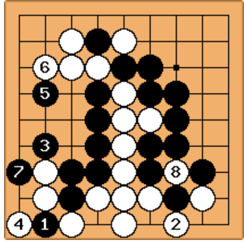
19x19 (largest board)

- “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.)*

Results: 9x9 Go



- Mogo (UCT-based)
  - A: Y. Wang, S. Gelly, R. Munos, O. Teytaud, and P-A. Coquelin, D. Silver
  - 100-230K simulations/move
  - Around since 2006 aug.
- CrazyStone (UCT-based)
  - A: Rémi Coulom
  - Switched to UCT in 2006
- Steenvreter (UCT-based)
  - A: Erik van der Werf
  - Introduced in 2007

- **Computer Olympiad (2007 December)**
  - **19x19**
    1. **MoGo**
    2. **CrazyStone**
    3. **GnuGo**
  - 9x9
    1. Steenvreter
    2. Mogo
    3. CrazyStone
- Guo Jan (5 dan), 9x9 board
  - Mogo black: 75% win
  - Mogo white: 33% win

CGOS: 1800 ELO → 2600 ELO

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

- Sparse Sampling and UCT are two approaches for building look-ahead trees for MDP planning
  - Allows us to use a simulator to achieve near optimal planning performance

- Sparse sampling
  - First near-optimal general MDP planner whose time complexity did not depend on state-space size
  - Not practical in practice (typically use small depths with heuristic at leaves)

- UCT
  - Attempts to intelligently expand the tree in the most promising directions
  - Major advance for computer Go (big surprise)