Carnegie Mellon
School of Computer Science

Deep Reinforcement Learning and Control

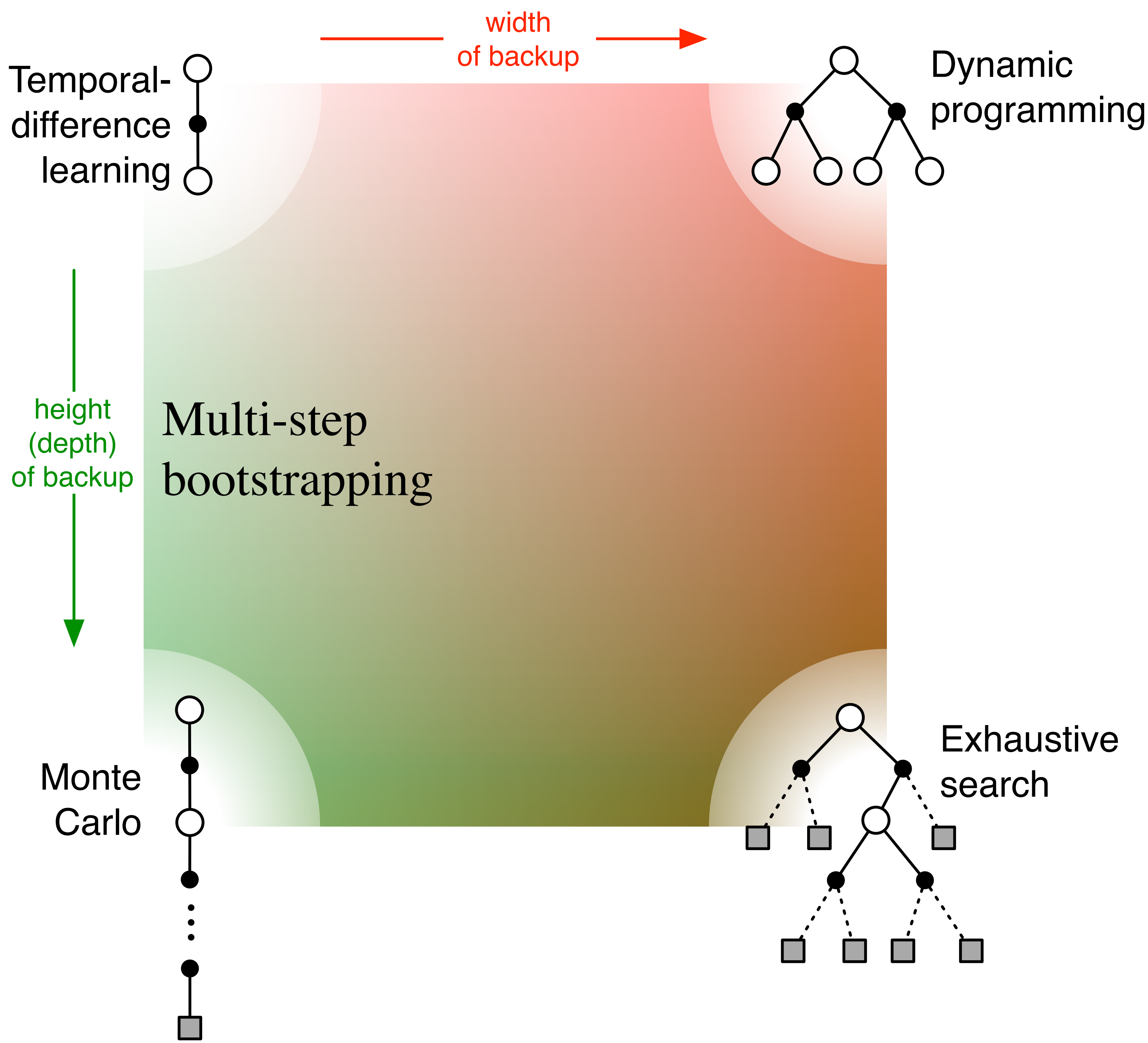
Multi-step Bootstrapping

CMU 10-403

Katerina Fragkiadaki

Slides from Rich Sutton

Unified View



n-step TD Returns/Targets

- **Monte Carlo:** $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$

n-step TD Returns/Targets

- Monte Carlo: $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$

- TD: $G_t^{(1)} \doteq R_{t+1} + \gamma V_t(S_{t+1})$
  - Use V_t to estimate remaining return

n-step TD Returns/Targets

- Monte Carlo: $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$
- TD: $G_t^{(1)} \doteq R_{t+1} + \gamma V_t(S_{t+1})$
  - Use V_t to estimate remaining return
- *n*-step TD:
  - 2 step return: $G_t^{(2)} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_t(S_{t+2})$

n-step TD Returns/Targets

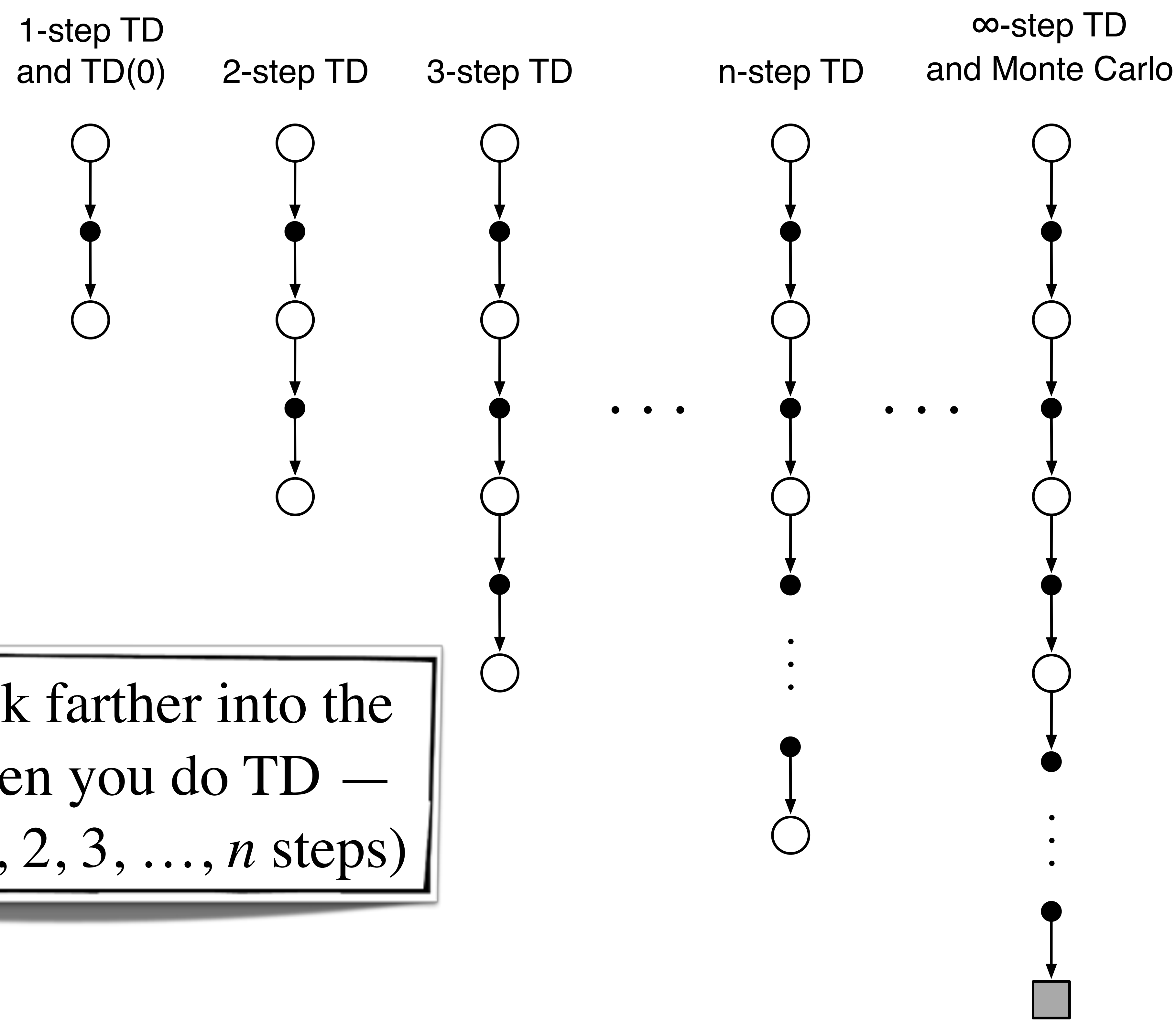
- Monte Carlo: $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$

- TD: $G_t^{(1)} \doteq R_{t+1} + \gamma V_t(S_{t+1})$
  - Use V_t to estimate remaining return

- *n*-step TD:
  - 2 step return: $G_t^{(2)} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_t(S_{t+2})$
  - *n*-step return: $G_t^{(n)} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_t(S_{t+n})$

    with $G_t^{(n)} \doteq G_t$ if $t + n \geq T$

n-step TD Prediction



Idea: Look farther into the future when you do TD — backup (1, 2, 3, …, n steps)

n-step TD

- Recall the *n*-step return:

$$G_t^{(n)} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}), \quad n \geq 1, 0 \leq t < T-n$$

- Of course, this is not available until time *t+n*
- The natural algorithm is thus to wait until then:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha \left[G_t^{(n)} - V_{t+n-1}(S_t) \right], \qquad 0 \leq t < T$$

- This is called *n*-step TD

n-step TD for estimating $V \approx v_\pi$

Initialize $V(s)$ arbitrarily, $s \in \mathcal{S}$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (for S_t and R_t) can take their index mod n

Repeat (for each episode):
    Initialize and store $S_0 \neq$ terminal
    $T \leftarrow \infty$
    For $t = 0, 1, 2, \ldots$:
    | If $t < T$, then:
    | Take an action according to $\pi(\cdot|S_t)$
    | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
    | If S_{t+1} is terminal, then $T \leftarrow t + 1$
    | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
    | If $\tau \geq 0$:
    | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
    | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_\tau^{(n)})$
    | $V(S_\tau) \leftarrow V(S_\tau) + \alpha \left[G - V(S_\tau)\right]$
    Until $\tau = T - 1$

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_{11} \rightarrow S_{12} \ldots S_T$$

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_{11} \rightarrow S_{12} \ldots S_T$$

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_{11} \rightarrow S_{12} \ldots S_T$$

$$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_{11} \rightarrow S_{12} \ldots S_T$$

n-step TD for estimating $V \approx v_\pi$

Initialize $V(s)$ arbitrarily, $s \in \mathcal{S}$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (for S_t and R_t) can take their index mod n

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_\tau^{(n)})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha\,[G - V(S_\tau)]$
  Until $\tau = T - 1$

No value update

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_{11} \rightarrow S_{12} \ldots S_T$

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_{11} \rightarrow S_{12} \ldots S_T$

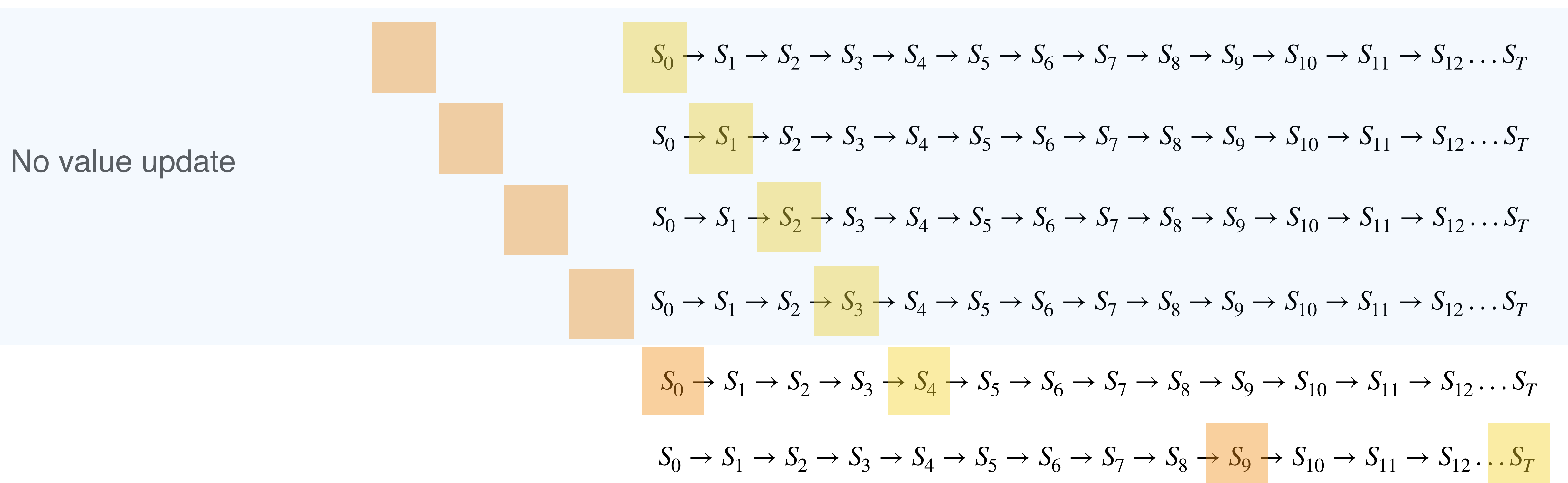
$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_{11} \rightarrow S_{12} \ldots S_T$

$S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_4 \rightarrow S_5 \rightarrow S_6 \rightarrow S_7 \rightarrow S_8 \rightarrow S_9 \rightarrow S_{10} \rightarrow S_{11} \rightarrow S_{12} \ldots S_T$

n-step TD for estimating $V \approx v_\pi$

Initialize $V(s)$ arbitrarily, $s \in \mathcal{S}$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (for S_t and R_t) can take their index mod n

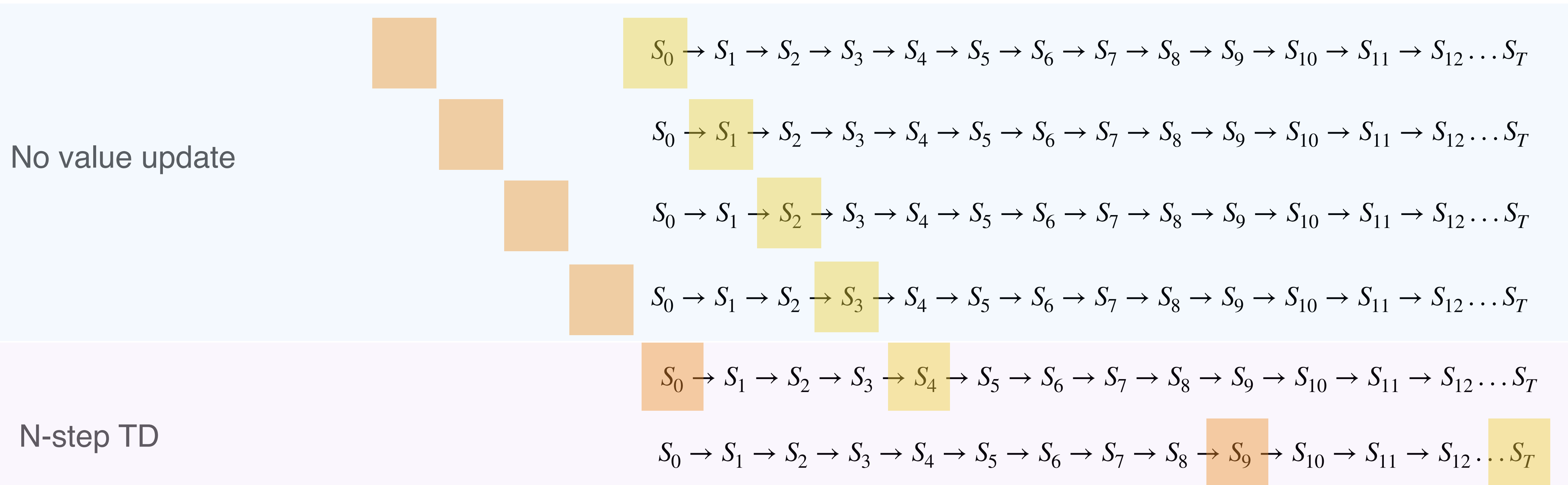
Repeat (for each episode):
    Initialize and store $S_0 \neq$ terminal
    $T \leftarrow \infty$
    For $t = 0, 1, 2, \ldots$:
    | If $t < T$, then:
    | Take an action according to $\pi(\cdot|S_t)$
    | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
    | If S_{t+1} is terminal, then $T \leftarrow t + 1$
    | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
    | If $\tau \geq 0$:
    | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
    | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_\tau^{(n)})$
    | $V(S_\tau) \leftarrow V(S_\tau) + \alpha\,[G - V(S_\tau)]$
    Until $\tau = T - 1$



n-step TD for estimating $V \approx v_\pi$

Initialize $V(s)$ arbitrarily, $s \in \mathcal{S}$
Parameters: step size $\alpha \in (0,1]$, a positive integer n
All store and access operations (for S_t and R_t) can take their index mod n

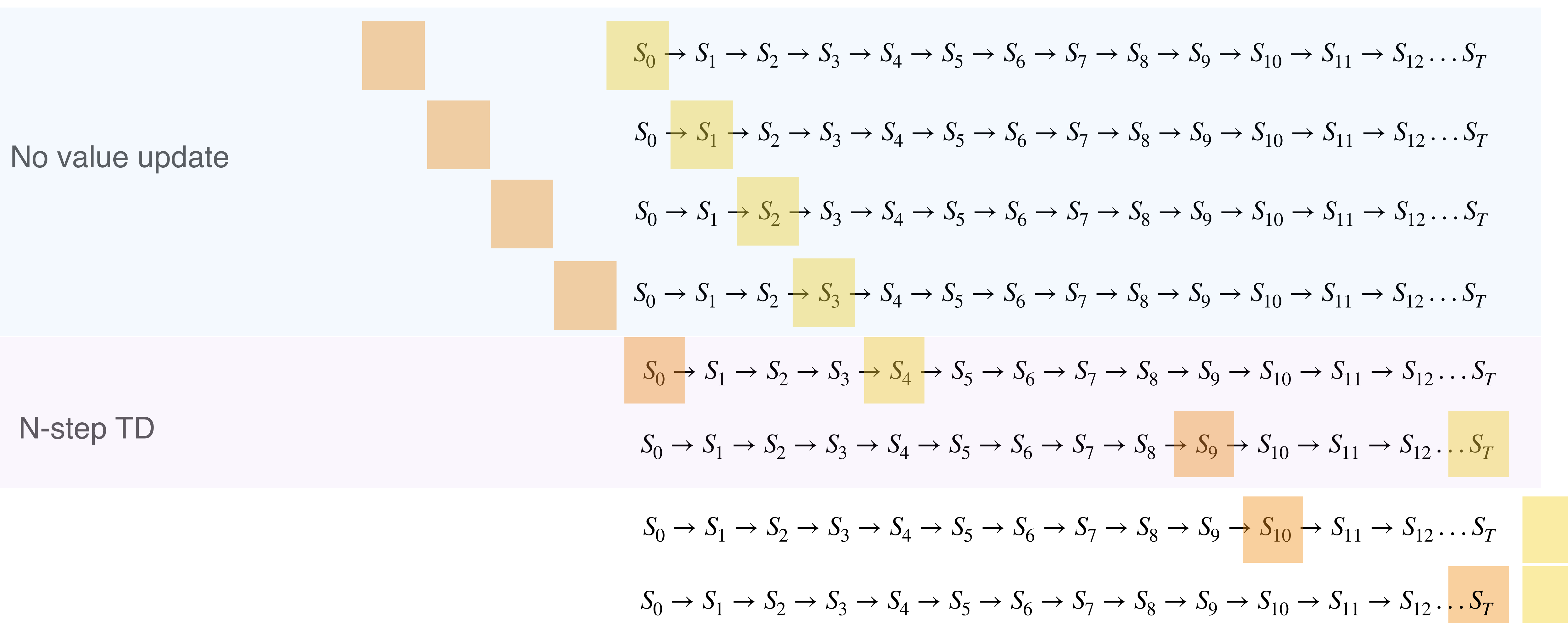
Repeat (for each episode):
    Initialize and store $S_0 \neq$ terminal
    $T \leftarrow \infty$
    For $t = 0, 1, 2, \ldots$:
    | If $t < T$, then:
    | Take an action according to $\pi(\cdot|S_t)$
    | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
    | If S_{t+1} is terminal, then $T \leftarrow t+1$
    | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
    | If $\tau \geq 0$:
    | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
    | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_\tau^{(n)})$
    | $V(S_\tau) \leftarrow V(S_\tau) + \alpha\,[G - V(S_\tau)]$
    Until $\tau = T - 1$



n-step TD for estimating $V \approx v_\pi$

Initialize $V(s)$ arbitrarily, $s \in \mathcal{S}$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (for S_t and R_t) can take their index mod n

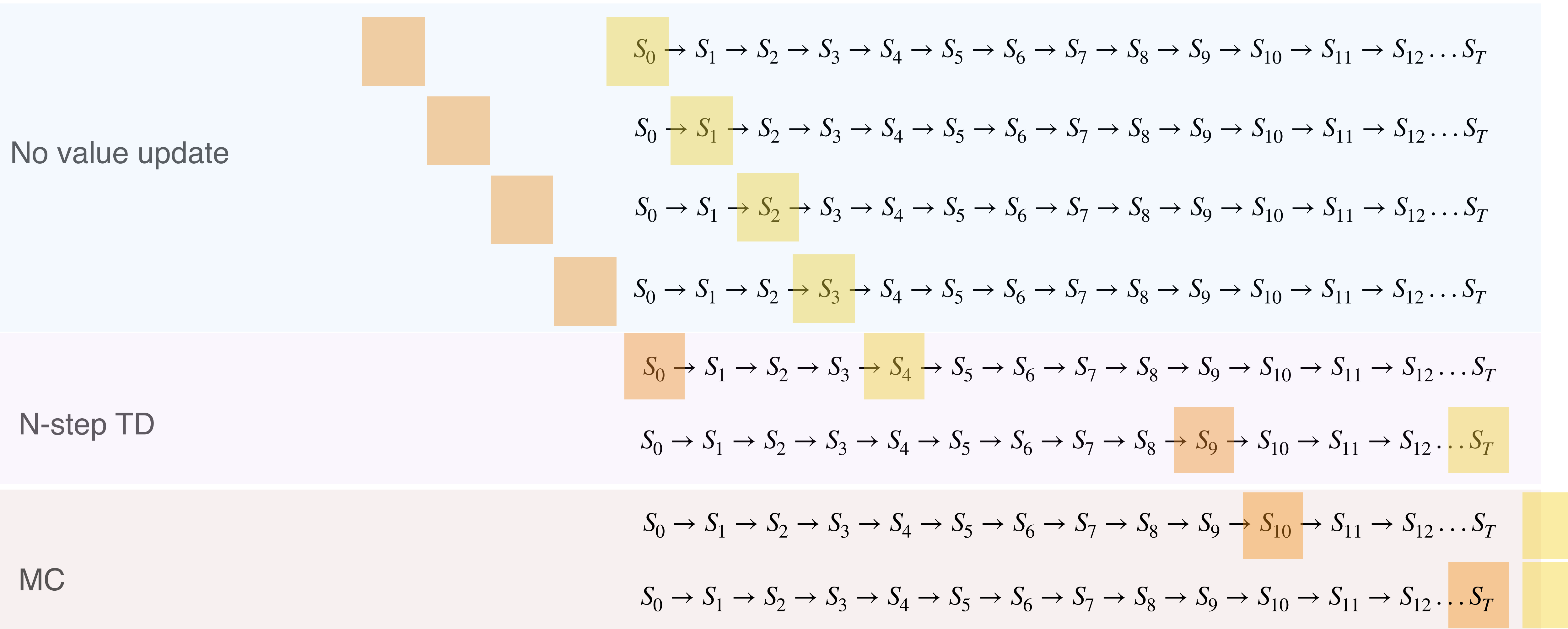
Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \dots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_\tau^{(n)})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha [G - V(S_\tau)]$
  Until $\tau = T - 1$



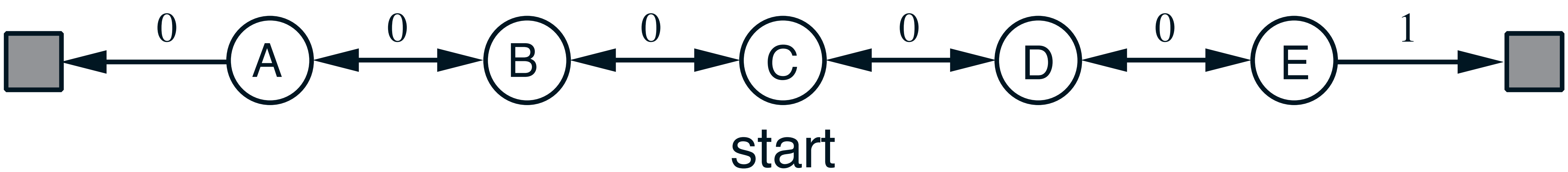
n-step TD for estimating $V \approx v_\pi$

Initialize $V(s)$ arbitrarily, $s \in \mathcal{S}$
Parameters: step size $\alpha \in (0, 1]$, a positive integer n
All store and access operations (for S_t and R_t) can take their index mod n

Repeat (for each episode):
  Initialize and store $S_0 \neq$ terminal
  $T \leftarrow \infty$
  For $t = 0, 1, 2, \ldots$:
  | If $t < T$, then:
  | Take an action according to $\pi(\cdot|S_t)$
  | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}
  | If S_{t+1} is terminal, then $T \leftarrow t + 1$
  | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)
  | If $\tau \geq 0$:
  | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n,T)} \gamma^{i-\tau-1} R_i$
  | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ $(G_\tau^{(n)})$
  | $V(S_\tau) \leftarrow V(S_\tau) + \alpha \left[G - V(S_\tau)\right]$
  Until $\tau = T - 1$

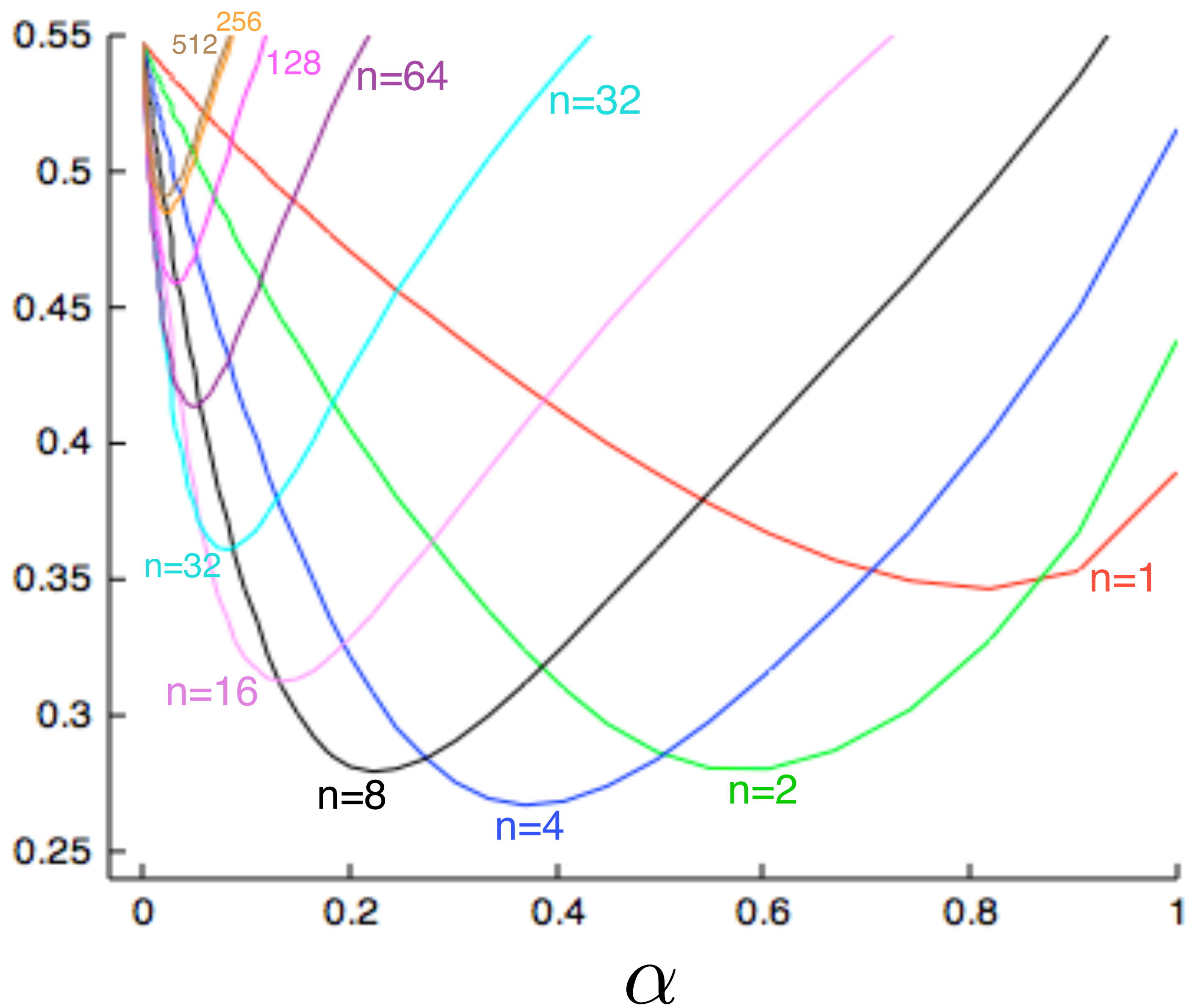


Random Walk Examples



A Larger Example – 19-state Random Walk

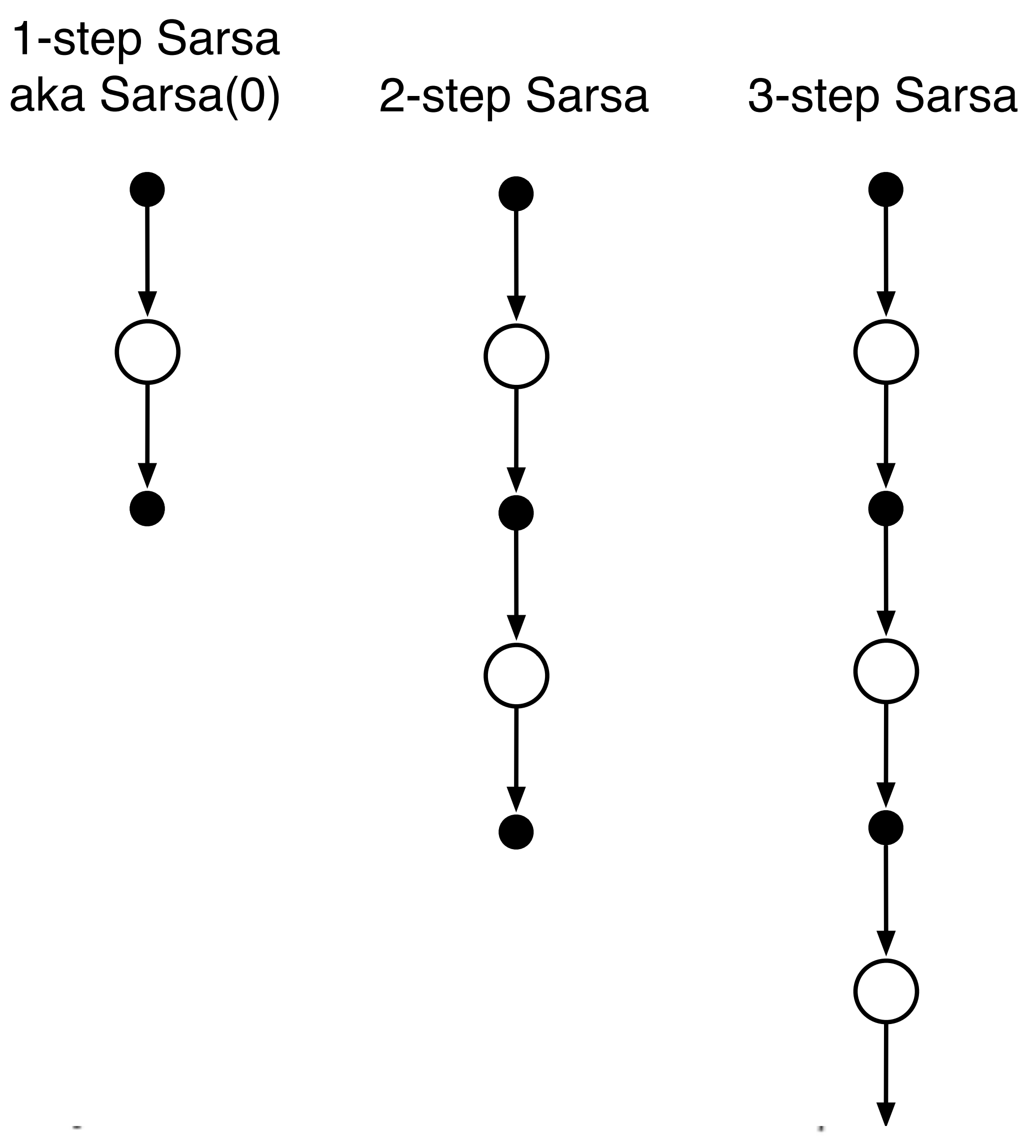
Average RMS error over 19 states and first 10 episodes

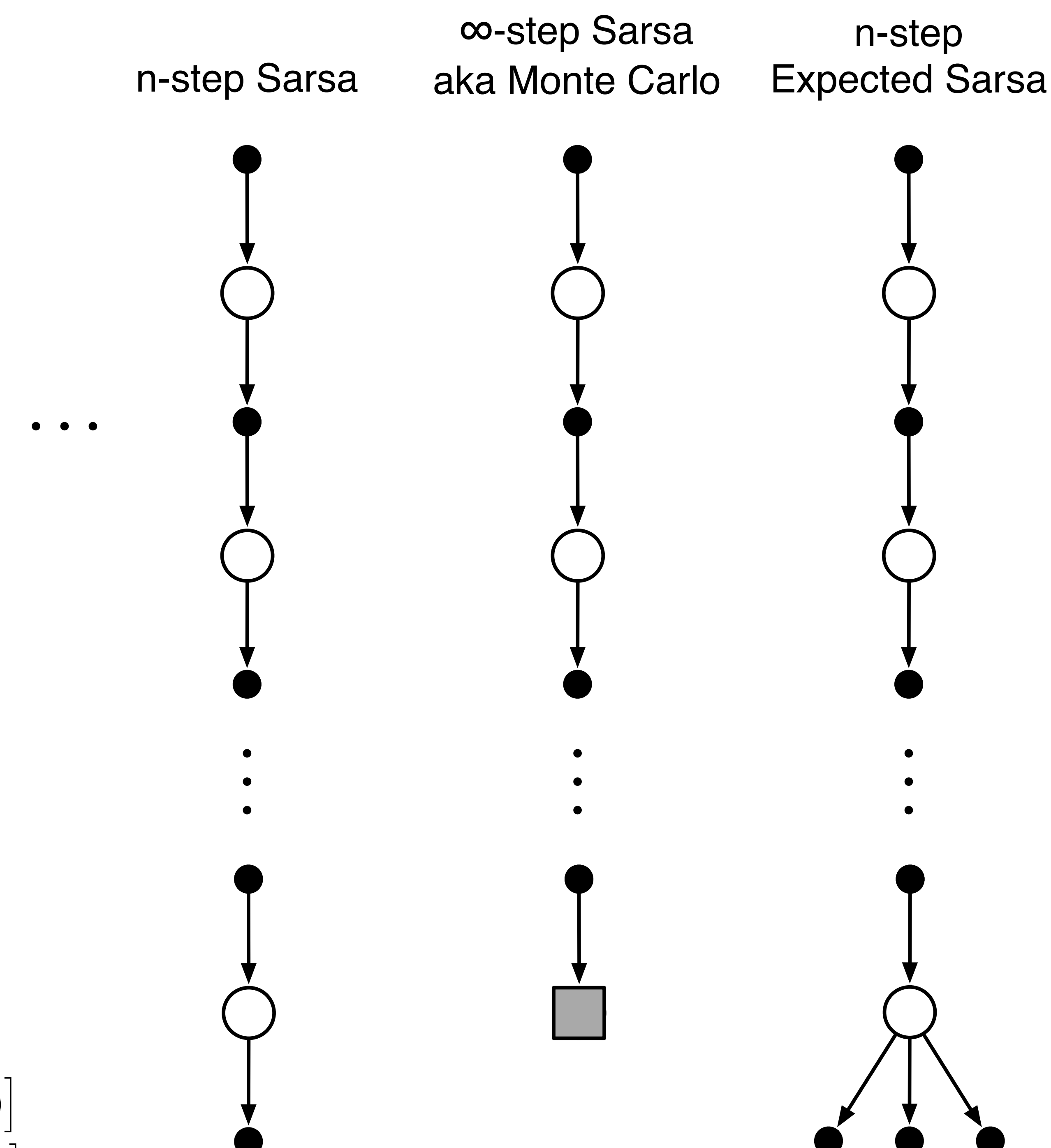


n-step TD results

- An intermediate α is best
- An intermediate n is best

It’s much the same for action values





$$Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1},A_{t+1}) - Q(S_t,A_t)\right]$$

$$\begin{aligned} Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma\mathbb{E}[Q(S_{t+1}, A_{t+1}) \mid S_{t+1}] - Q(S_t, A_t)\Big] \\ &\leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma\sum \pi(a|S_{t+1})Q(S_{t+1}, a) - Q(S_t, A_t)\Big] \end{aligned}$$

On-policy *n*-step Action-value Methods

- Action-value form of *n*-step return

$$G_t^{(n)} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \underline{Q_{t+n-1}(S_{t+n}, A_{t+n})}$$

- *n*-step Sarsa:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \left[G_t^{(n)} - Q_{t+n-1}(S_t, A_t) \right]$$

- *n*-step Expected Sarsa is the same update with a slightly different *n*-step return:

$$G_t^{(n)} \doteq R_{t+1} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n \underline{\sum_a \pi(a|S_{t+n}) Q_{t+n-1}(S_{t+n}, a)}$$

Off-policy *n*-step Methods by Importance Sampling

- Recall the *importance-sampling ratio*:

$$\rho_t^{t+n} \doteq \prod_{k=t}^{\min(t+n-1,T-1)} \frac{\pi(A_k|S_k)}{\mu(A_k|S_k)}$$

- We get off-policy methods by weighting updates by this ratio

- Off-policy *n*-step TD:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha \rho_t^{t+n} \left[G_t^{(n)} - V_{t+n-1}(S_t) \right]$$

- Off-policy *n*-step Sarsa:

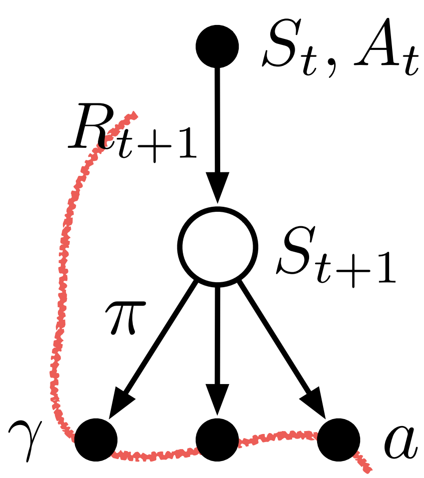
$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \rho_{t+1}^{t+n} \left[G_t^{(n)} - Q_{t+n-1}(S_t, A_t) \right]$$

- Off-policy *n*-step Expected Sarsa:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha\, \rho_{t+1}^{t+n-1} \left[G_t^{(n)} - Q_{t+n-1}(S_t, A_t) \right]$$

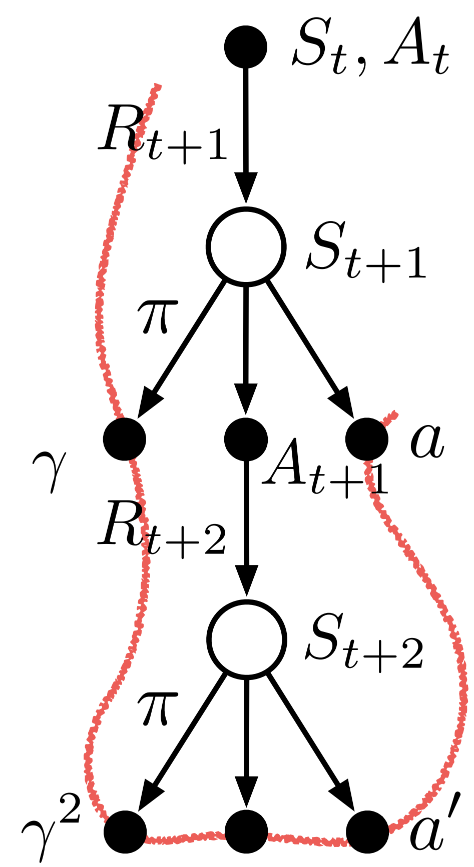
Off-policy Learning w/o Importance Sampling: The *n*-step Tree Backup Algorithm

Expected Sarsa
and 1-step Tree Backup

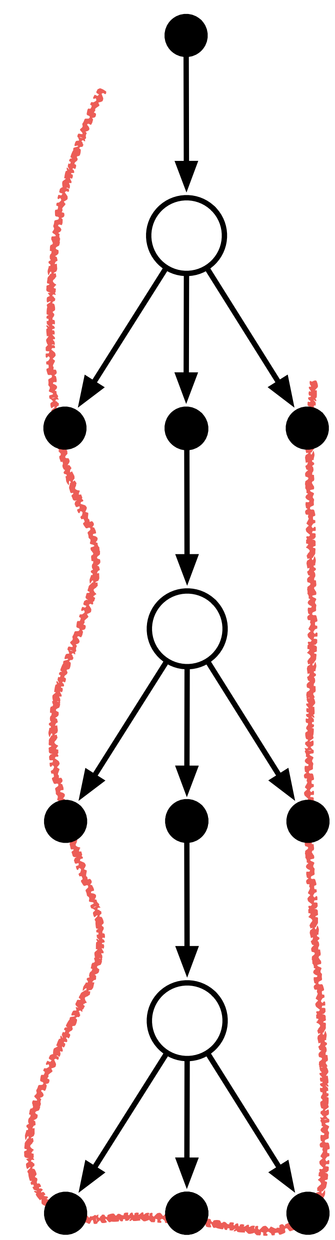


$$R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) Q(S_{t+1}, a)$$

2-step Tree Backup



3-step TB



Target

$$R_{t+1} + \gamma \sum_{a \neq A_{t+1}} \pi(a|S_{t+1}) Q(S_{t+1}, a)$$
$$+\gamma\, \pi(A_{t+1}|S_{t+1}) \left(R_{t+2} + \gamma \sum_{a'} \pi(a'|S_{t+2}) Q(S_{t+2}, a') \right)$$

Conclusions Regarding *n*-step Methods

- Generalize Temporal-Difference and Monte Carlo learning methods, sliding from one to the other as n increases
  - $n = 1$ is TD as in Chapter 6
  - $n = \infty$ is MC as in Chapter 5
  - an intermediate n is often much better than either extreme
  - applicable to both continuing and episodic problems
- There is some cost in computation
  - need to remember the last n states
  - learning is delayed by n steps
  - per-step computation is small and uniform, like TD

Carnegie Mellon
School of Computer Science

Deep Reinforcement Learning and Control

Monte Carlo Tree Search

CMU 10-403

Katerina Fragkiadaki

Part of slides inspired by Sebag, Gaudel

Definitions

Learning: the acquisition of knowledge or skills through experience, study, or by being taught.

Planning: any computational process that uses a model to create or improve a policy

$$\text{Model} \xrightarrow{\text{Planning}} \text{Policy}$$

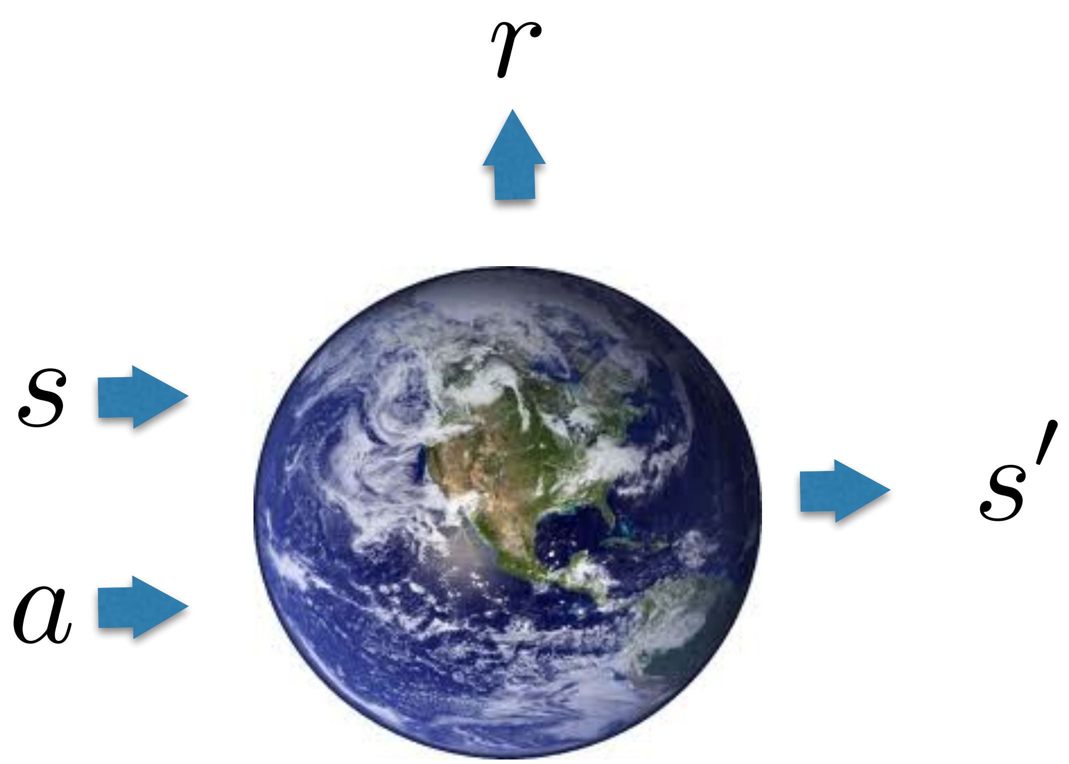
This lecture

Computing value functions combining learning and planning using Monte Carlo Tree Search

Computing value functions combining learning and planning in other ways will be revisited in later lectures

Model

Anything the agent can use to predict how the environment will respond to its actions, concretely, the state transition T(s'|s,a) and reward R(s,a).

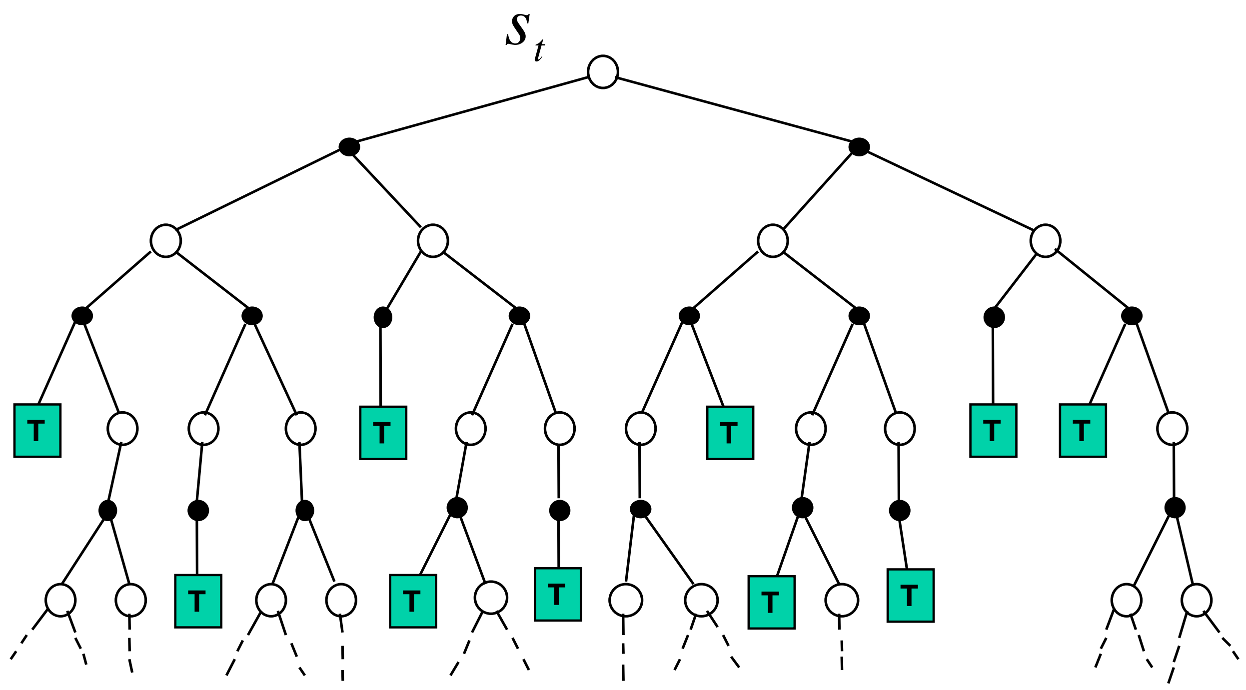


Ground-truth

this includes transitions of the state of the environment and the state of the agent

Online Planning with Search

1. Build a search tree **with the current state of the agent** at the root
2. Compute value functions using simulated episodes (reward usually only on final state, e.g., win or loose)
3. Select the next move to execute
4. Execute it
5. GOTO 1



Why online planning?

Why don’t we *just* learn a value function directly for every state offline, so that we do not waste time online?

- Because the environment has many many states (consider Go 10^170, Chess 10^48, real world ….)
- Very hard to compute a good value function for each one of them, most you will never visit
- Thus, condition on the current state you are in, try to estimate the value function of the relevant part of the state space online
- Focus your resources on sub-MDP starting from now, often dramatically easier than solving the whole MDP

Curse of dimensionality

- The sub-MDP rooted at the current state the agent is in may still be very large (too many states are reachable), despite much smaller than the original one.
- Too many actions possible: large tree branching factor
- Too many steps: large tree depth

I cannot exhaustively search the full tree

Curse of dimensionality

Consider hex on an NxN board.

branching factor $\leq N^2$

$2N \leq$ depth $\leq N^2$

board size	max branching factor	min depth	tree size	depth of 10^{10} nodes
6x6	36	12	$>10^{17}$	7
8x8	64	16	$>10^{28}$	6
11x11	121	22	$>10^{44}$	5
19x19	361	38	$>10^{96}$	4

Goal of HEX: to make a connected line that links two antipodal points of the grid

How to handle curse of dimensionality?

Intelligent instead of exhaustive search

1. The depth of the search may be reduced by position evaluation: truncating the search tree at state s and replacing the subtree below s by an approximate value function v(s)=v*(s) that predicts the outcome from state s.
2. The breadth of the search may be reduced by sampling actions from a policy p(a|s), that is, a probability distribution over plausible moves a in position s, instead of trying every action.

Position evaluation

We can estimate values for states in two ways:

- Engineering them using human experts (DeepBlue)
- Learning them from self-play (TD-gammon)

Problems with human engineering:

- tiring
- non transferrable to other domains.

YET: that's how Kasparov was first beaten.

http://stanford.edu/~cpiech/cs221/apps/deepBlue.html

Monte-Carlo position evaluation

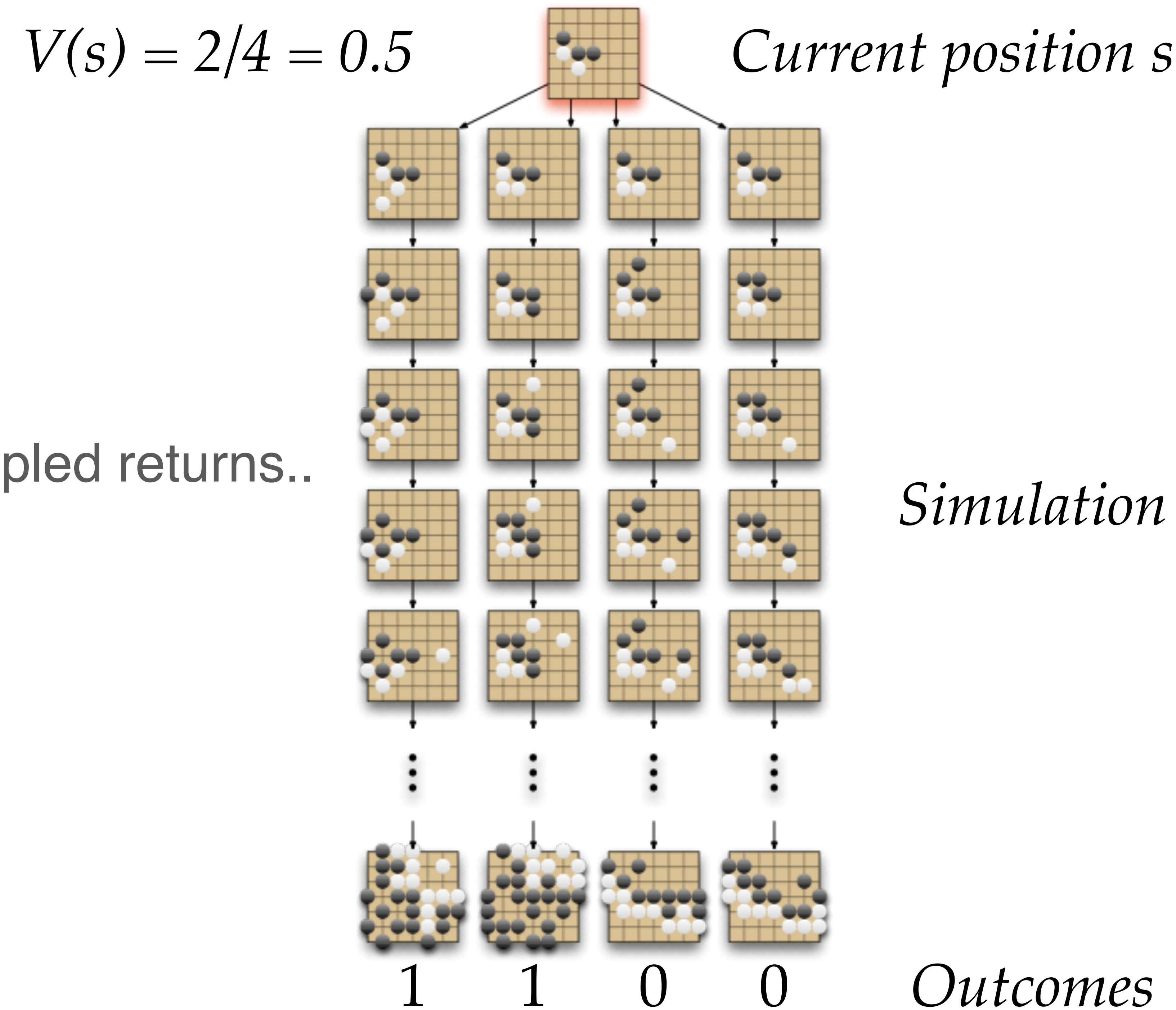
```
function MC_BoardEval(state):
    wins = 0
    losses = 0
    for i=1:NUM_SAMPLES
        next_state = state
        while non_terminal(next_state):
            next_state = random_legal_move(next_state)
        if next_state.winner == state.turn: wins++
        else: losses++ #needs slight modification if draws possible
    return (wins - losses) / (wins + losses)
```

What policy shall we use to draw our simulations?

The cheapest one is random..

Monte-Carlo position evaluation in Go

Averaging sampled returns..



Simplest Monte-Carlo Search

- For action selection, I need to be estimating not state but rather state-action values.
- But! Since we assume dynamics given, we can simply use one step look-ahead!

Simplest Monte-Carlo Search

Given a deterministic transition function $\mathbf{T}$, a root state s and a simulation policy π (potentially random)

For each action $a \in \mathscr{A}$

$$Q(s,a) = \text{MC-boardEval}(s'), \quad s' = \mathrm{T}(s,a)$$

Select root action: $a = \text{argmax}_{a \in \mathscr{A}} Q(s,a)$

Simplest Monte-Carlo Search

Given a deterministic transition function T, a root state s and a simulation policy π (potentially random)

Simulate K episodes from current (real) state:

$$\{s, a, R_1^k, S_1^k, A_1^k, R_2^k, S_2^k, A_2^k, \ldots, S_T^k\}_{k=1}^K \sim \mathrm{T}, \pi$$

Evaluate action value function of the root by mean return:

$$Q(s,a) = \frac{1}{K}\sum_{k=1}^{K} G_k \xrightarrow{\mathrm{P}} q_\pi(s,a)$$

Select root action: $a = \mathrm{argmax}_{a \in \mathcal{A}} Q(s,a)$

Can we do better?

- Could we be **improving our simulation policy** the more simulations we obtain?
- Yes we can! We can have two policies:
  1. Internal to the tree: keep track of action values Q not only for the root but also for nodes internal to a tree we are expanding, and (maybe) use \epsilon-greedy(Q) to improve the simulation policy over time
  2. External to the tree: we do not have Q estimates and thus we use a random policy

In MCTS, the simulation policy improves

- Any better ideas for the simulation policy?

Upper Confidence Bound (UCB)

$$A_t \doteq \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right] \qquad A_t \sim \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right]$$

ith action score:

$$\underbrace{v_i}_{\text{value estimate}} + \underbrace{C}_{\text{tunable parameter}} \times \sqrt{\frac{\ln(\overbrace{N}^{\text{parent node visits}})}{\underbrace{n_i}_{\text{number of visits}}}}$$

- score is decreasing in the number of visits (explore)
- score is increasing in a node's value (exploit)
- always tries every option once

Finite-time Analysis of the Multiarmed Bandit Problem, Auer, Cesa-Bianchi, Fischer, 2002

Monte-Carlo Tree Search

1. **Selection**
   - Used for nodes we have seen before
   - Pick according to UCB
2. **Expansion**
   - Used when we reach the frontier
   - Add one node per playout
3. **Simulation**
   - Used beyond the search frontier
   - Don't bother with UCB, just play randomly
4. **Backpropagation**
   - After reaching a terminal node
   - Update value and visits for states expanded in selection and expansion

Bandit based Monte-Carlo Planning, Kocsis and Szepesvari, 2006

Monte-Carlo Tree Search

Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```

For every state within the search tree we bookkeep # of visits and # of wins

Monte-Carlo Tree Search

Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```

Monte-Carlo Tree Search

MCTS helper functions

Sample actions based on UCB score

```
function UCB_sample(state):
    weights = []
    for child of state:
        w = child.value + C * sqrt(ln(state.visits) / child.visits)
        weights.append(w)
    distribution = [w / sum(weights) for w in weights]
    return child sampled according to distribution
```

(unrolling)

```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

Monte-Carlo Tree Search

MCTS helper functions

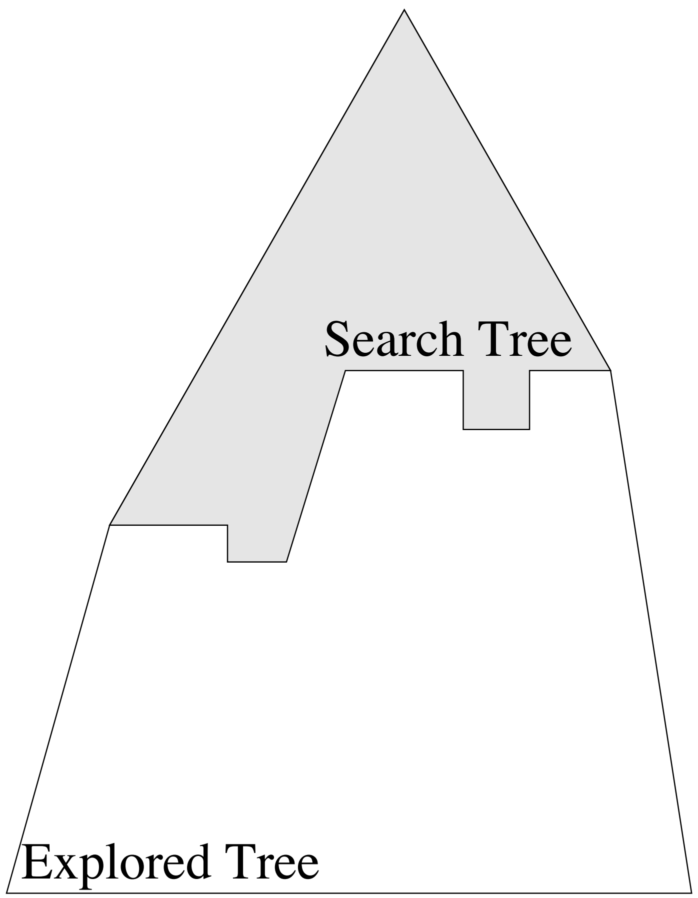
```
function expand(state):
    state.visits = 1
    state.value = 0


function update value(state, winner):

    if winner == state.turn:
        state.value += 1
    else:
        state.value -= 1
```

Basic MCTS pseudocode

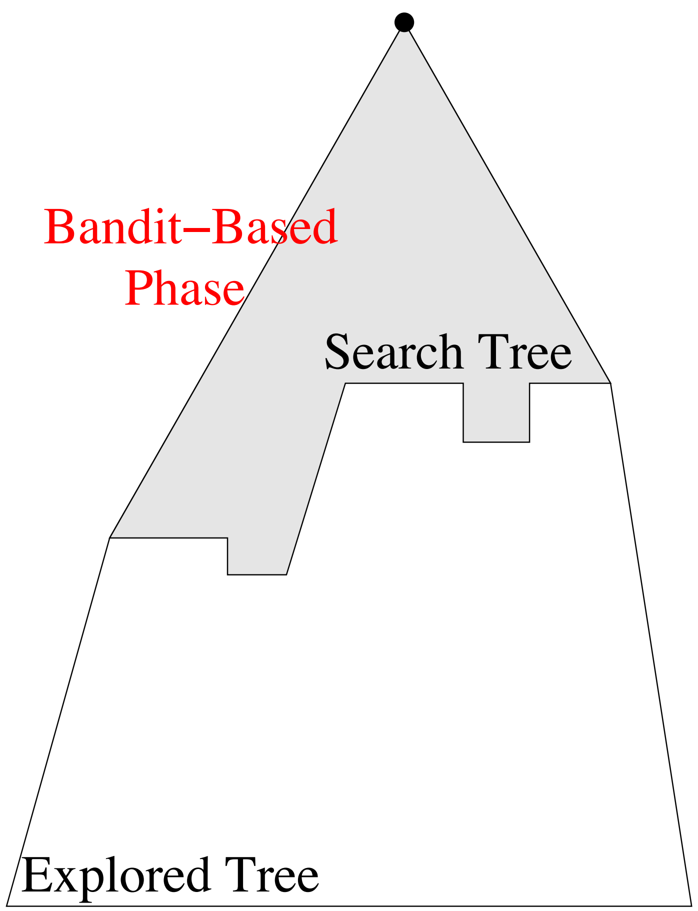
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
       next_state = UCB_sample(state)
       winner = MCTS_sample(next_state)
    else:
       if some children of state expanded:
           next_state = expand(random unexpanded child)
       else:
           next_state = state
       winner = random_playout(next_state)
    update_value(state, winner)
```



Seacrh tree contains states whose all children have been tried at least once

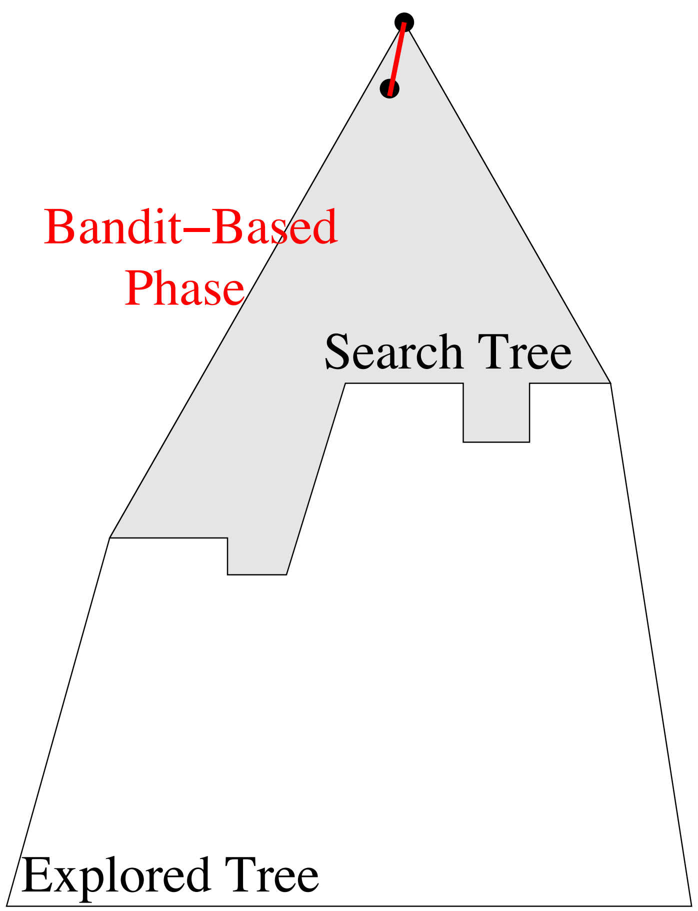
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



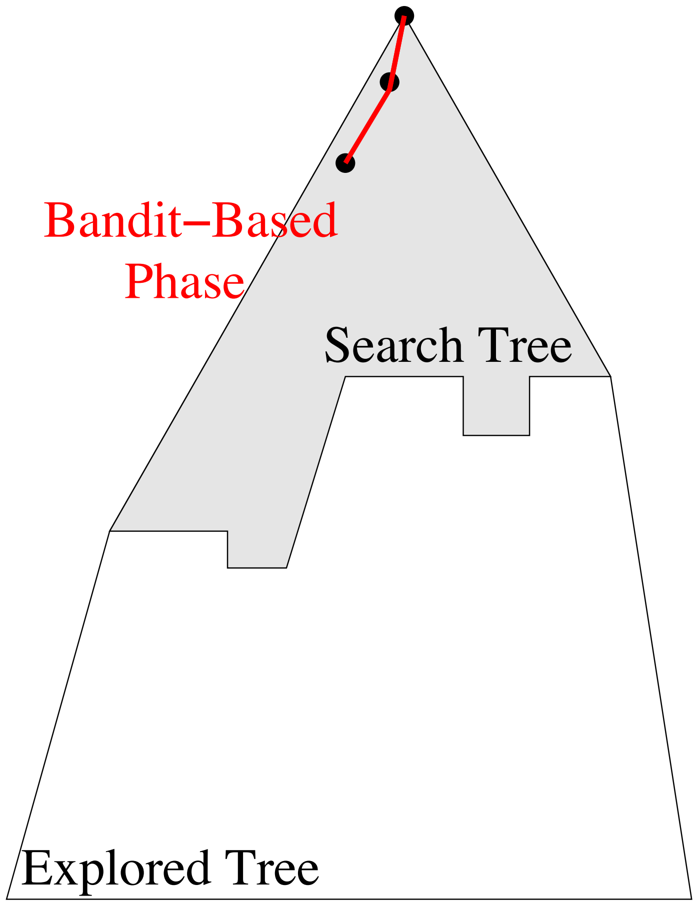
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



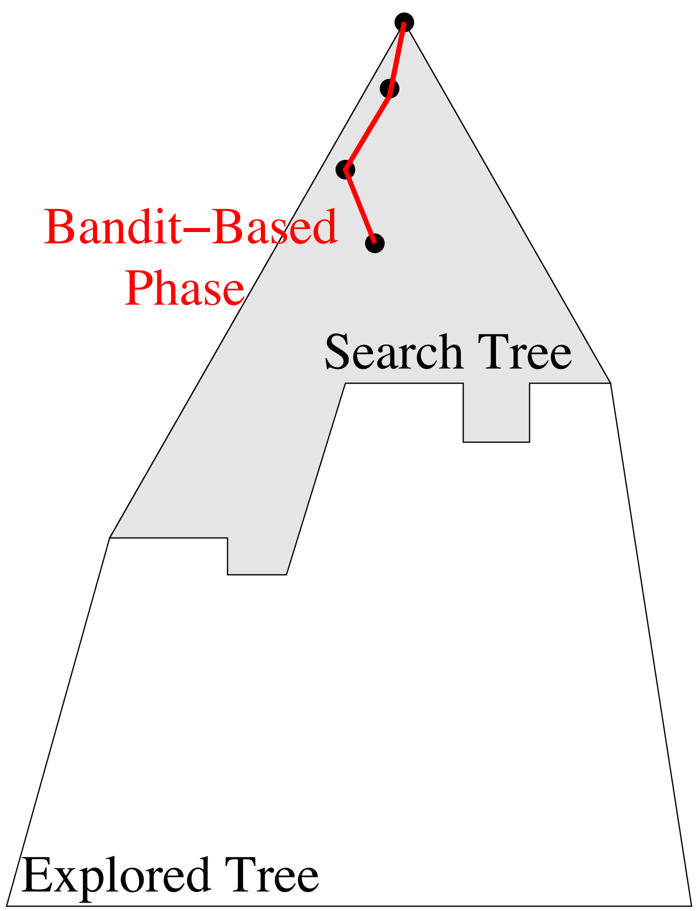
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



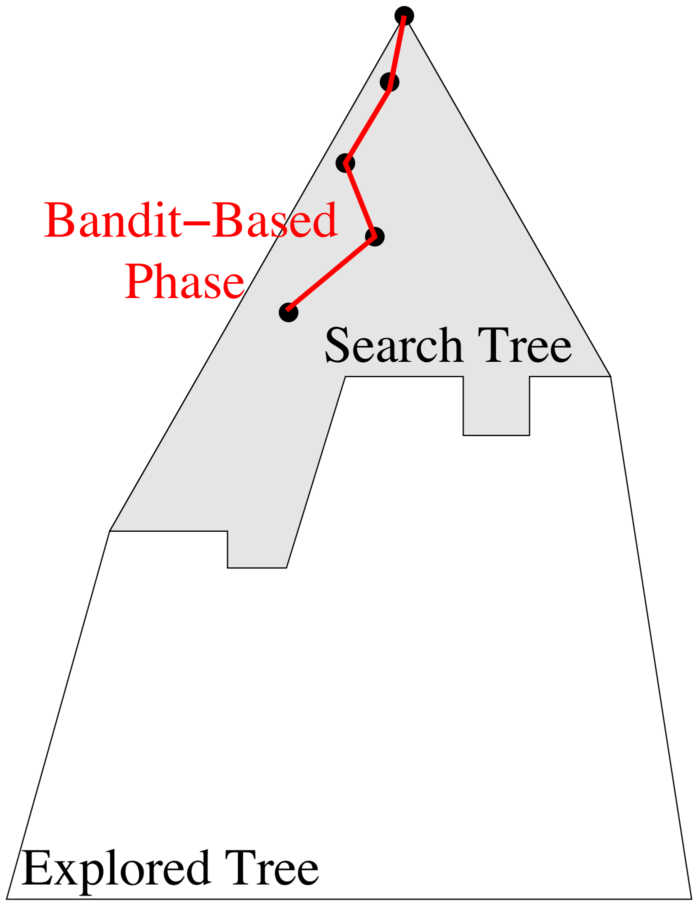
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



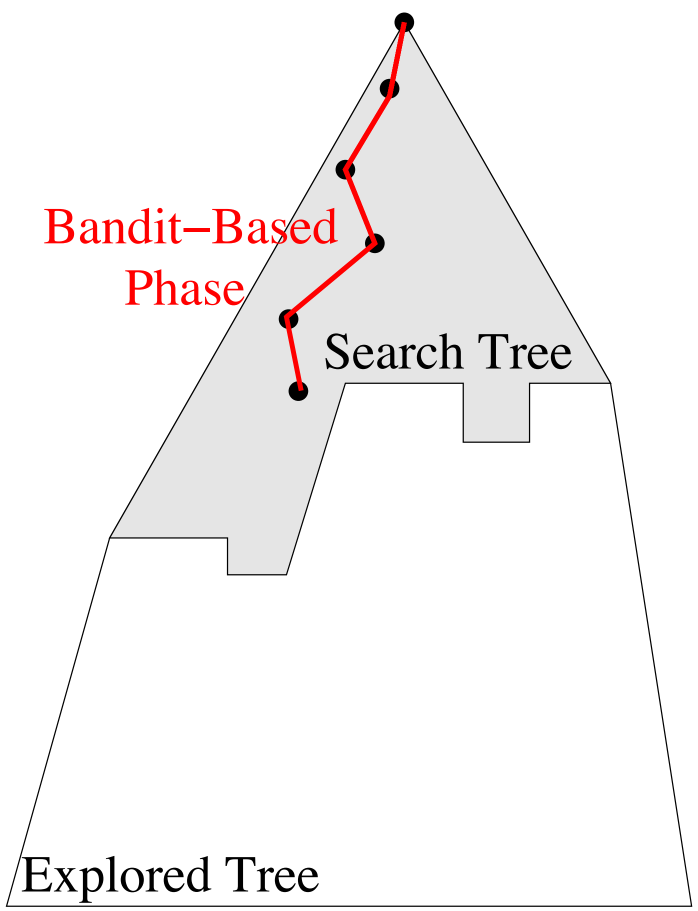
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



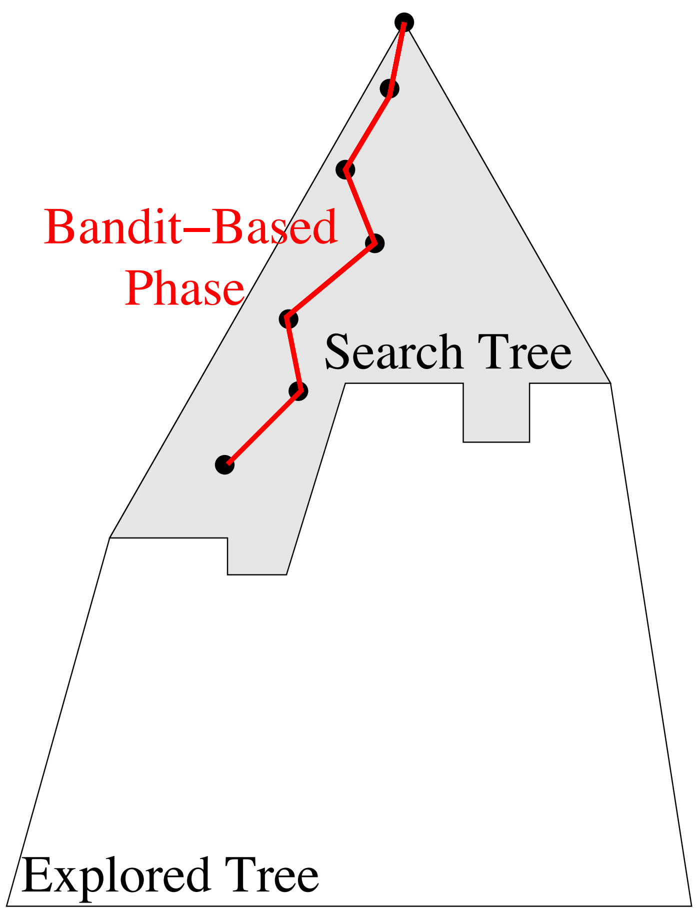
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



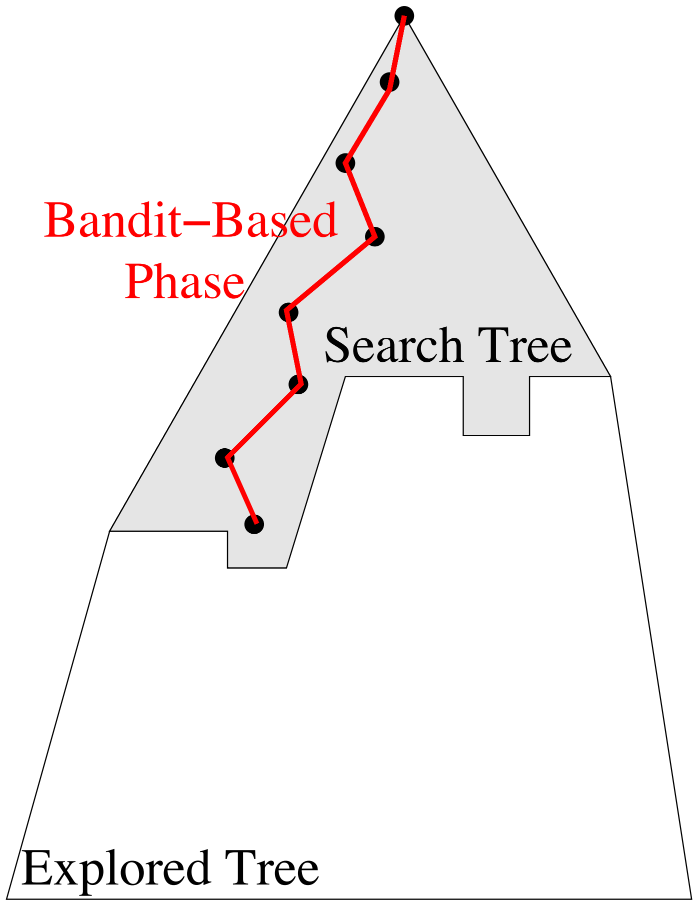
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



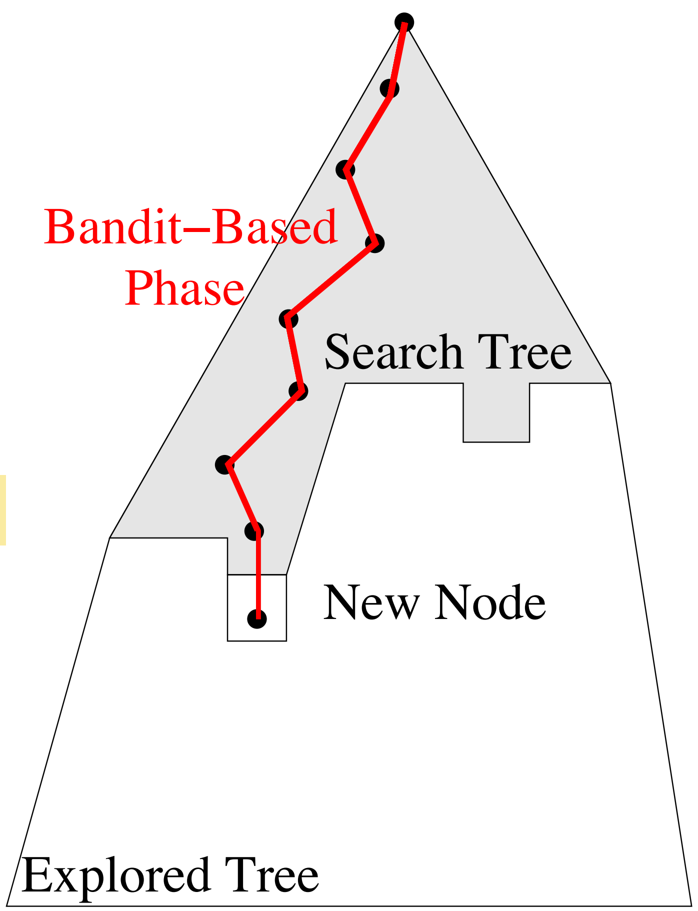
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



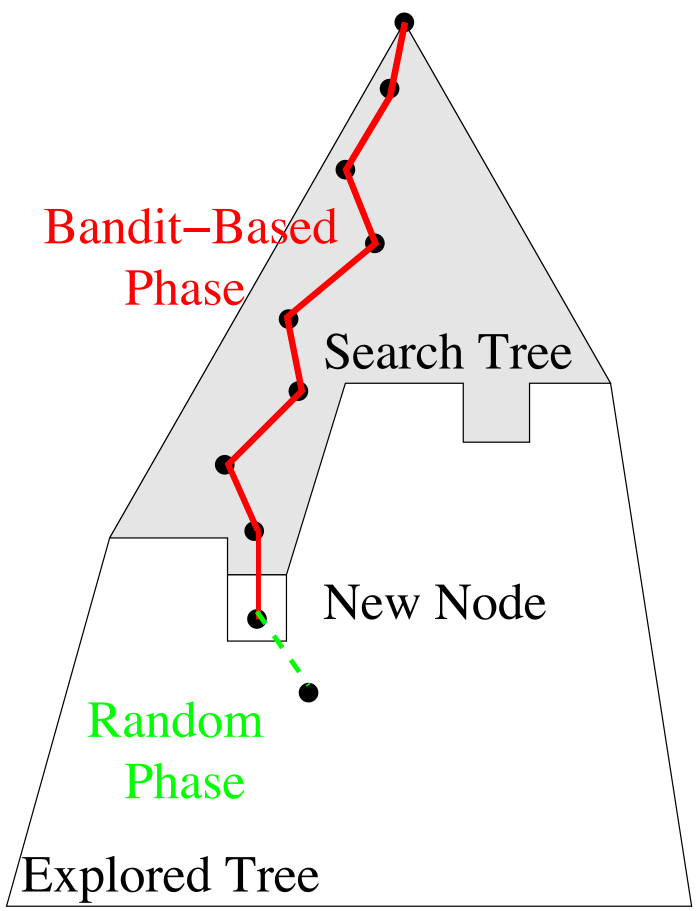
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



Basic MCTS pseudocode

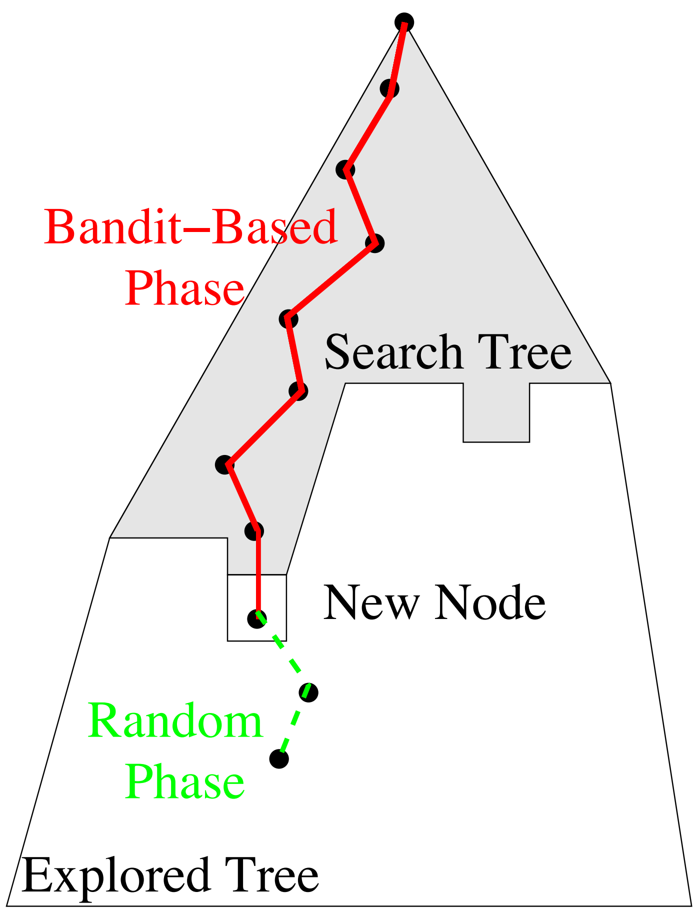
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```




```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

Basic MCTS pseudocode

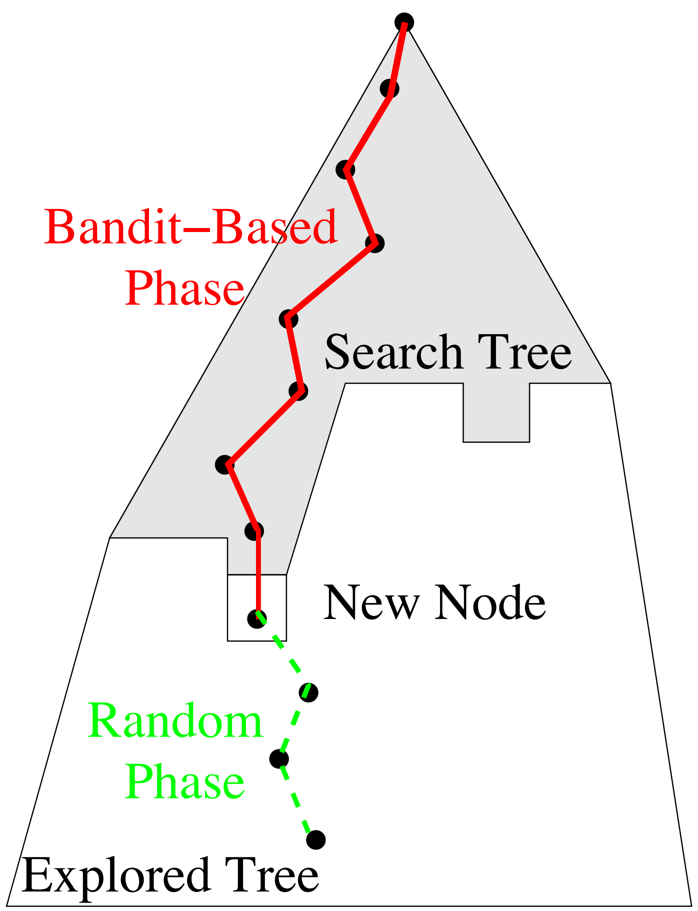
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

Basic MCTS pseudocode

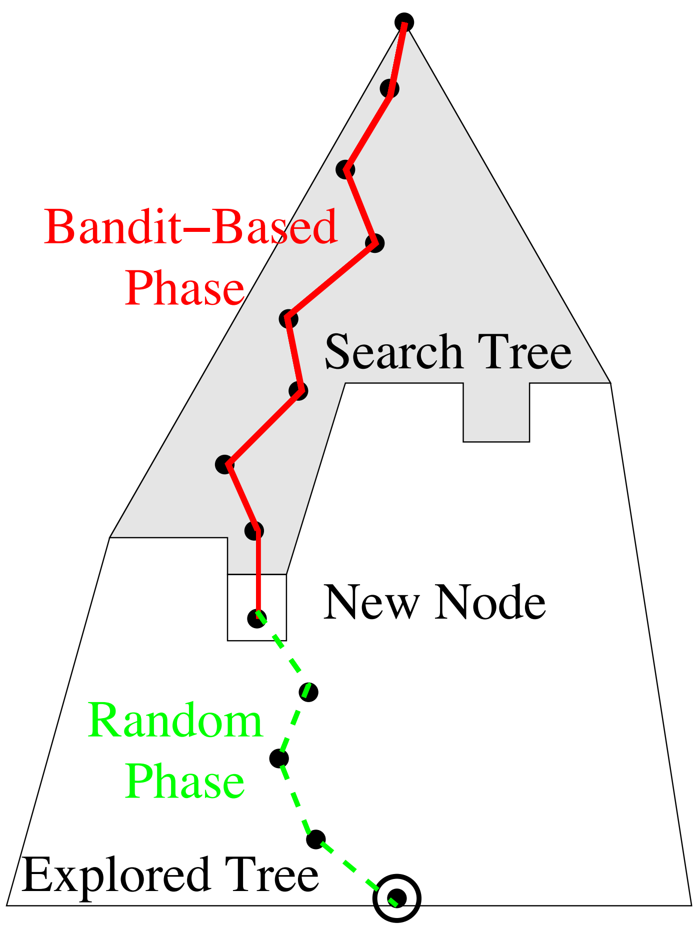
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```




```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

Basic MCTS pseudocode

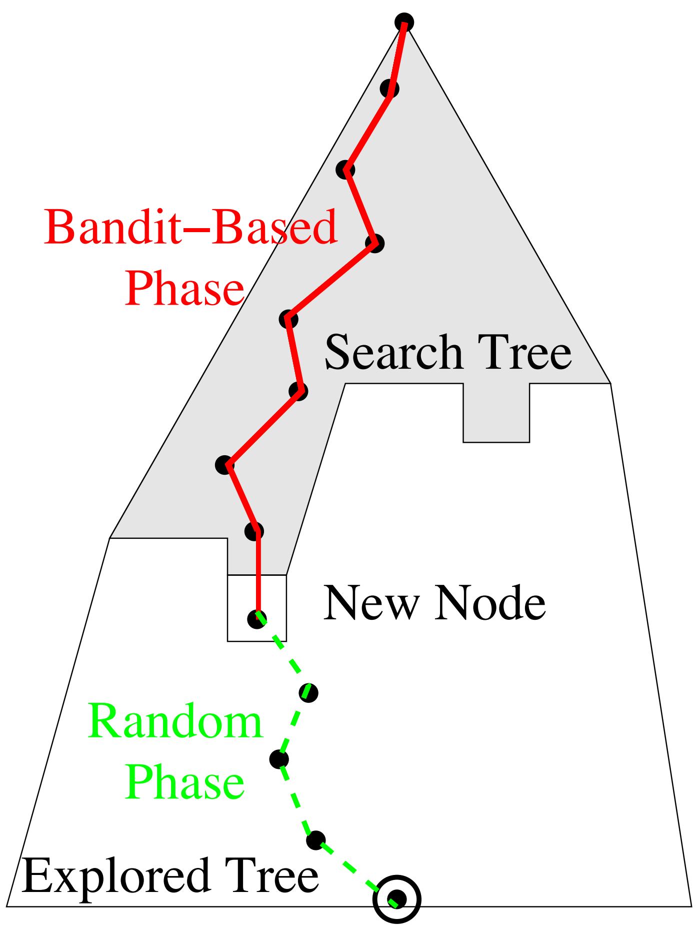
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```




```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



Monte-Carlo Tree Search