COL778: Principles of Autonomous Systems
Semester II, 2023-24

Monte Carlo Tree Search

Rohan Paul

Outline

- Last Class
  - Mult—armed Bandits
- This Class
  - Monte Carlo Tree Search
- Reference Material
  - Please follow the notes as the primary reference on this topic.

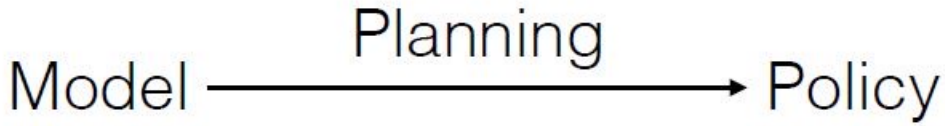
Acknowledgements

These slides are intended for teaching purposes only. Some material has been used/adapted from web sources and from slides by Nicholas Roy, Wolfram Burgard, Dieter Fox, Sebastian Thrun, Siddharth Srinivasa, Dan Klein, Pieter Abbeel, Max Likhachev, Alexander Amini (MIT Introduction to Deep Learning) and others. This lecture is primarily built on the slides from Katerina Fragkiadaki at CMU.

Definitions

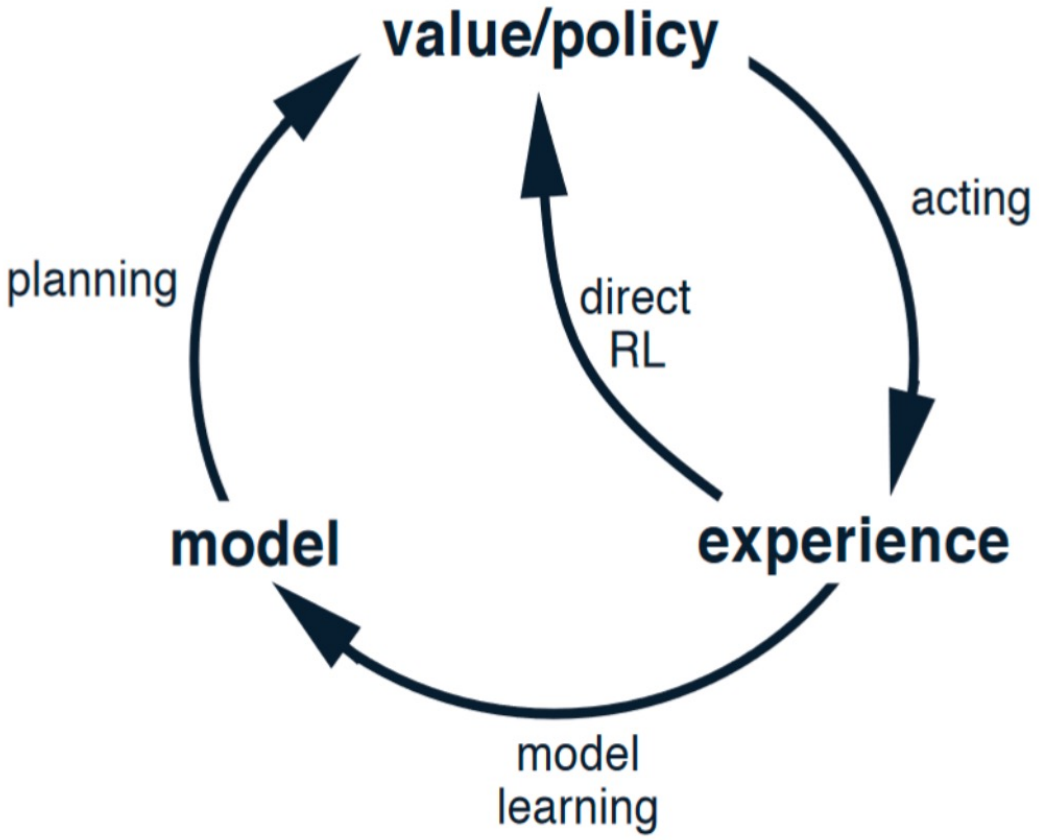
Learning: the acquisition of knowledge or skills through experience, study, or by being taught (value function, model, q-function etc.)

Planning: any computational process that uses a model to create or improve a policy. Policy prescribes the actions. Planning estimates the policy.



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

Learning, Planning and Acting



Online planning

Core idea: learn the value function from the current state. Do not learn the value function for all states.

- Because the environment has many many states (consider Go 10^170, Chess 10^48, real world may have a very large number of states).
- Very hard to compute a good value function for each one of them, most of them you will never visit at training time.
- 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.

Online Planning

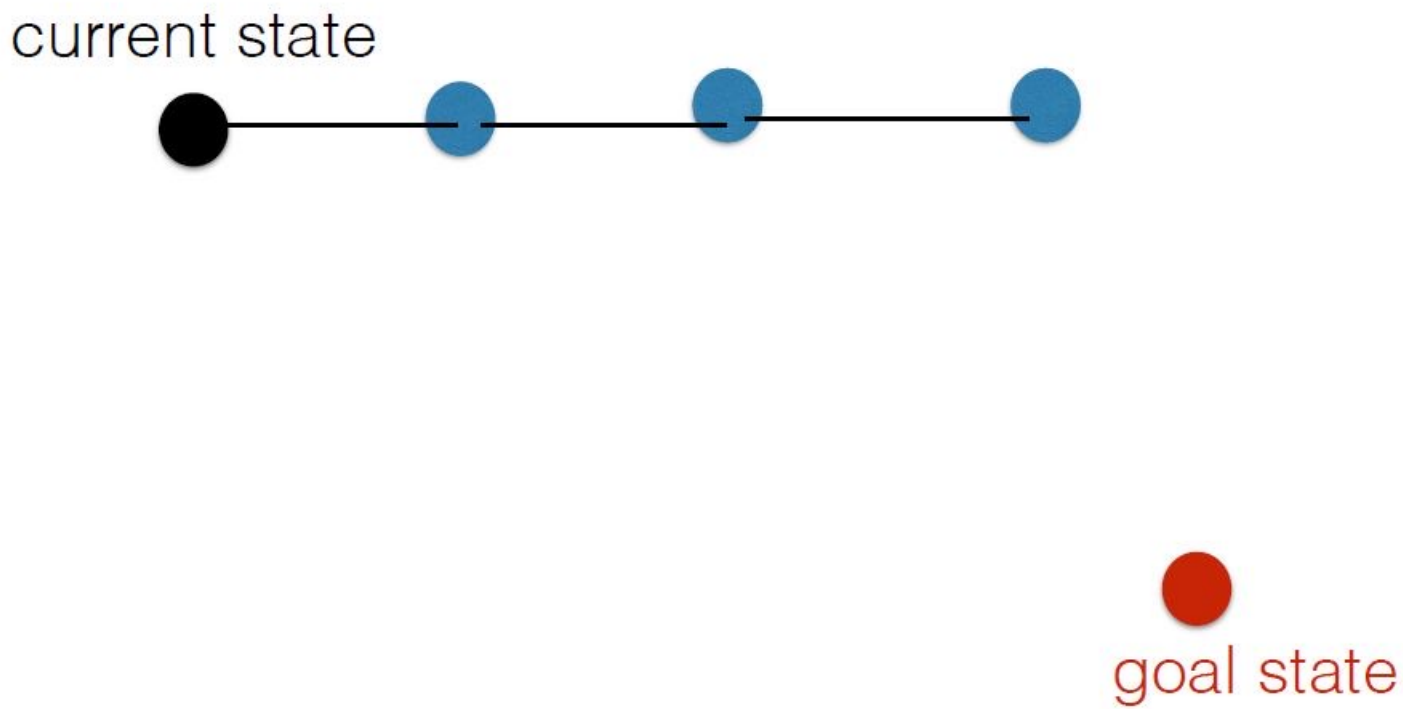
Unroll the model of the environment forward in time to select the right action sequences to achieve your goal.

Online Planning

Unroll the model of the environment forward in time to select the right action sequences to achieve your goal.

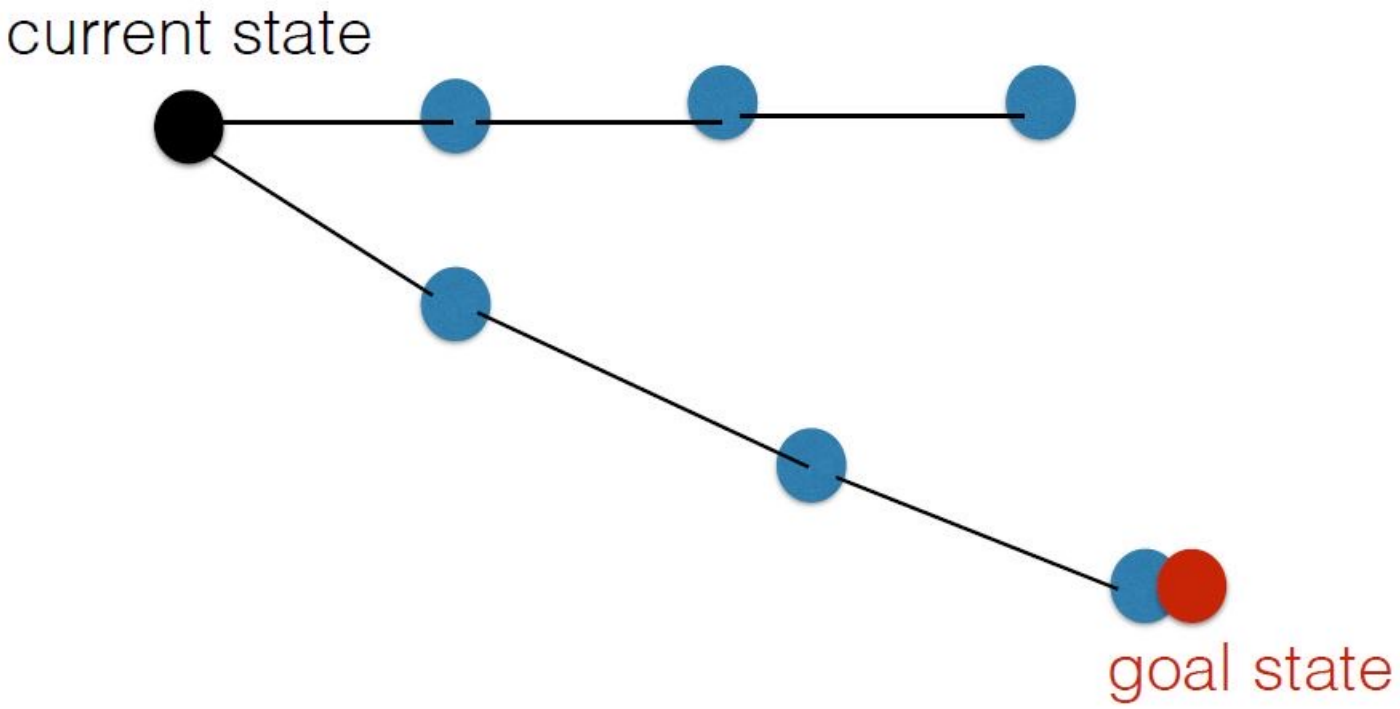
Online Planning

Unroll the model of the environment forward in time to select the right action sequences to achieve your goal.



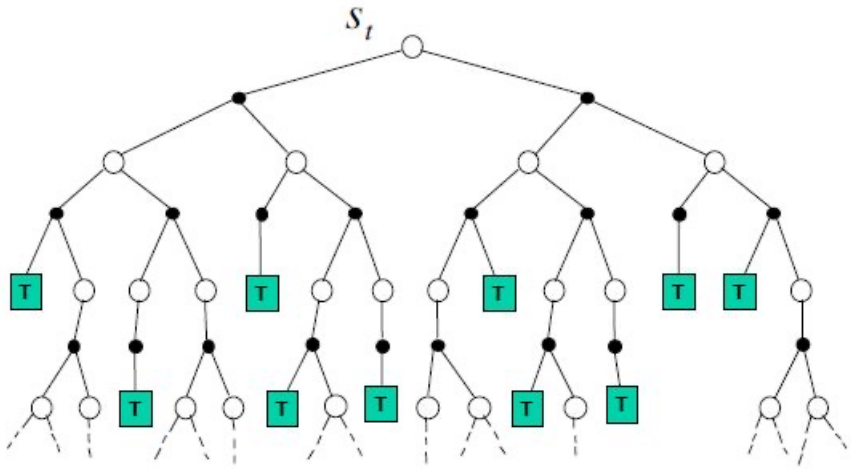
Online Planning

Unroll the model of the environment forward in time to select the right action sequences to achieve your goal.



Online Planning with Search

1. Build the “full” search tree **with the current state of the agent** at the root
2. Select the next move to execute using heuristics
3. Execute it
4. GOTO 1



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 the MDP being much smaller than the original one.
- Why?
  - Large tree branching factor (too many actions possible:)
  - Large tree depth (too many planning steps/depth of planning)

Intelligent Instead of Exhaustive Search

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.

Large depth
- Approximate the value function after a certain depth.

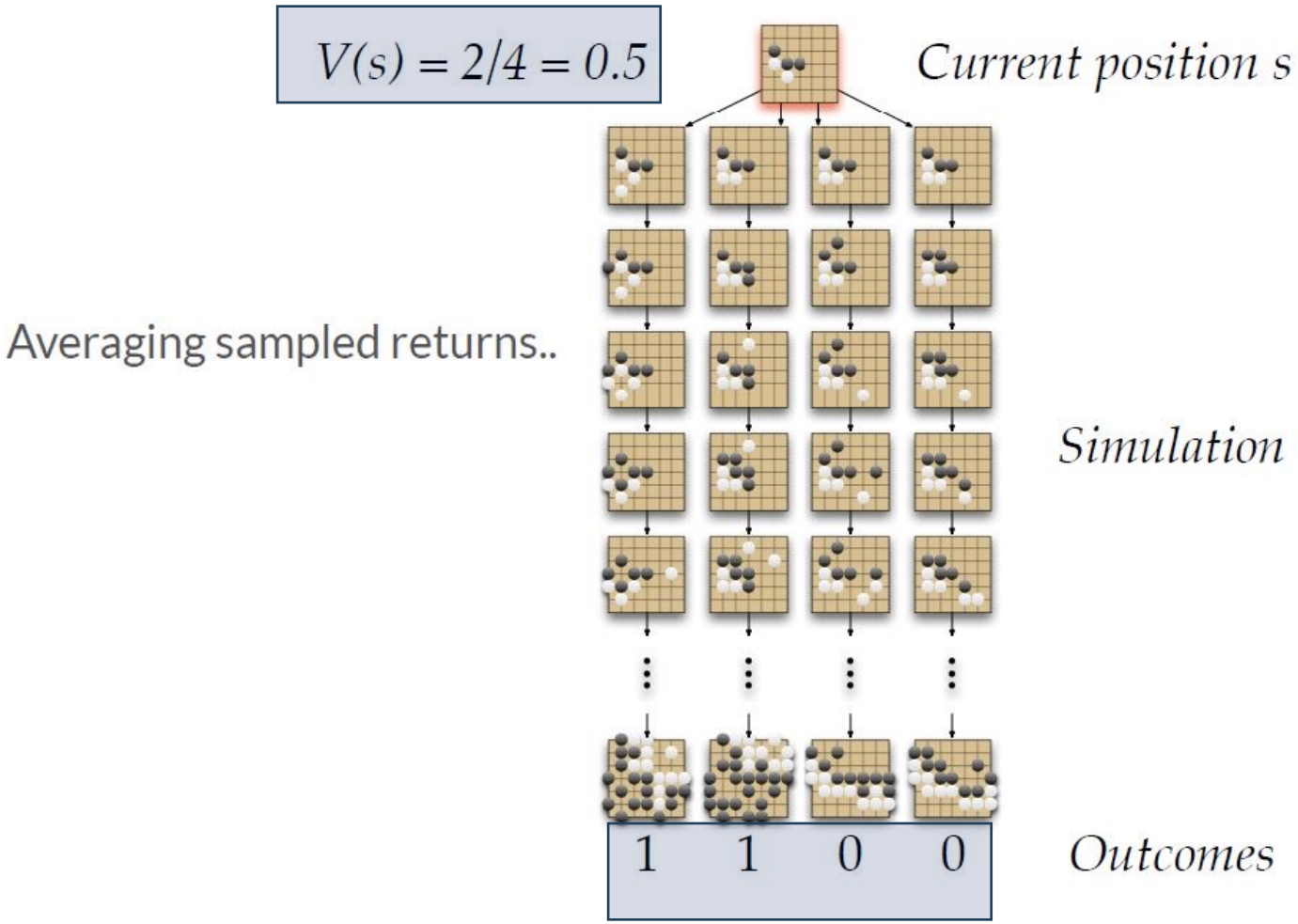
The breadth of the search may be reduced by sampling actions from a policy $p(a \mid s)$, that is, a probability distribution over plausible moves a in position s, instead of trying every action.

Branching factor
– sampling the actions

Estimating Expected Value via Simulations

Examine the outcomes and average the rewards obtained.

In essence, this is monte carlo estimation.



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 to use for sampling actions? Easiest is to select/sample actions at random

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:

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

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

Simplest Monte-Carlo Search

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

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

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

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

Can we do better?

- Can the simulation policy be improved with the more simulations?
- Solution: Have two policies
  - **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 use to improve the simulation policy over time
  - **External to the tree:** we do not have Q estimates and thus we use a random policy

- Options:
  - ϵ – greedy?
  - UCB?

Upper Confidence Bound (UCB)

$$A_t = \text{argmax}_a \left[Q_t(a) + c\sqrt{\frac{\log t}{N_t(a)}} \right]$$

- t : parent node visits
- $N_t(a)$: times the action has been tried out

- Probability of choosing an action:
  - decreases with the number of visits (explore)
  - increases with a node's value (exploit)
- Always tries every option once.

- A better exploration-exploitation than ϵ − greedy

Note: the regret grows sub-linearly for UCB.

Monte-Carlo Tree Search (MCTS)

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

Monte-Carlo Tree Search

```
function UCB_sample(node):
    weights = []
    for child of node:
        w = child.value
        w += C*sqrt(ln(node.visits) / child.visits)
        add w to weights
    distribution = normalize weights to sum to 1
    return child sampled according to distribution
```

Monte-Carlo Tree Search

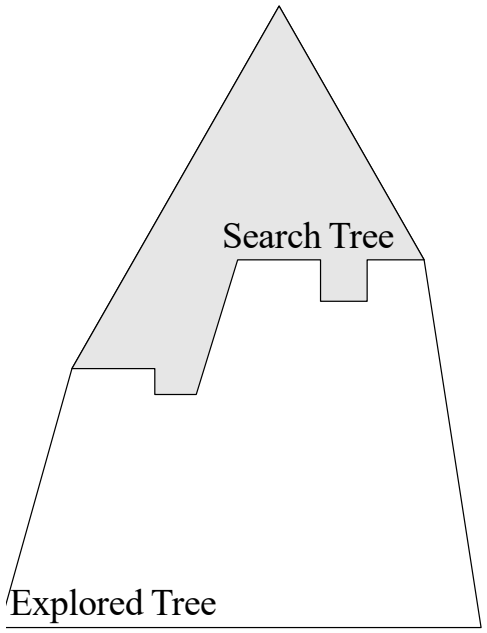
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```

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

Monte-Carlo Tree Search (helper functions)

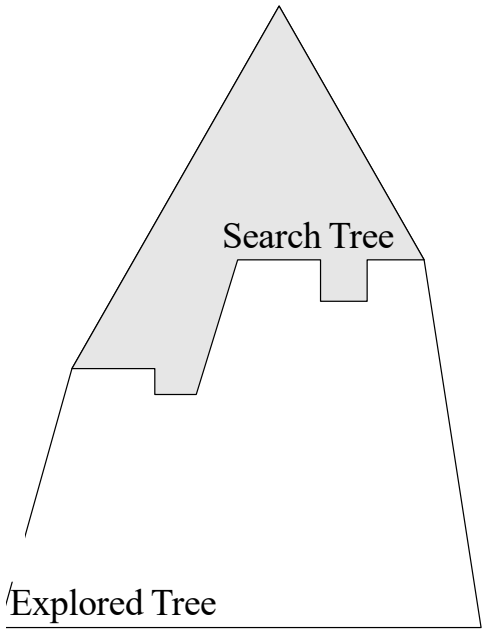
```
function random_playout(state):
    while state is not terminal:
        state = make a random move from state
    return outcome


function update_value(node, outcome):
    #combine the new outcome with the average value
    node.value *= node.visits
    node.visits++
    node.value += outcome
    node.value /= node.visits
```

```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```

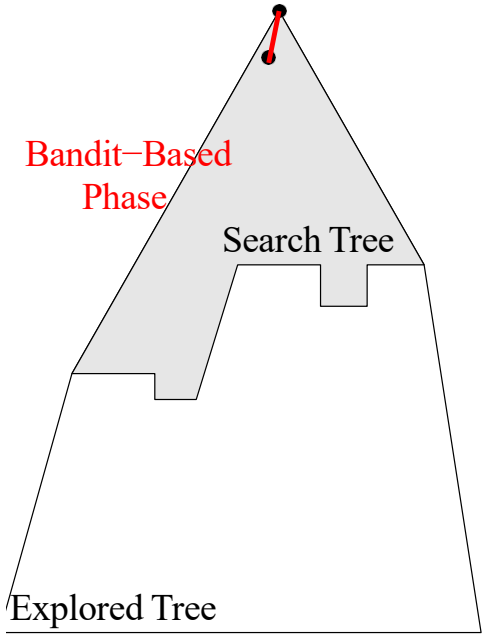


Search tree contains states whose children have been tried at least once

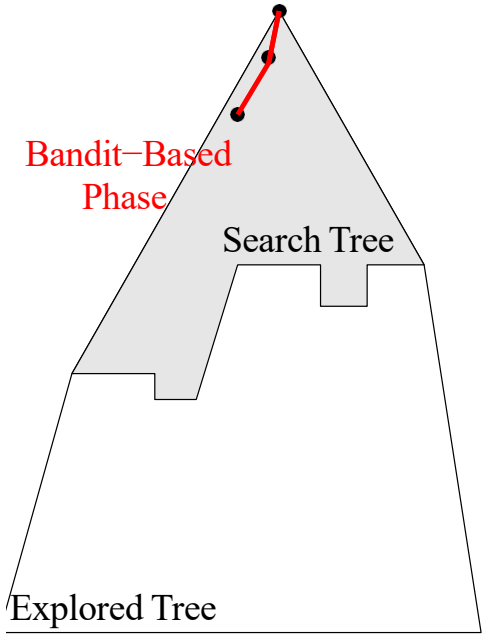
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



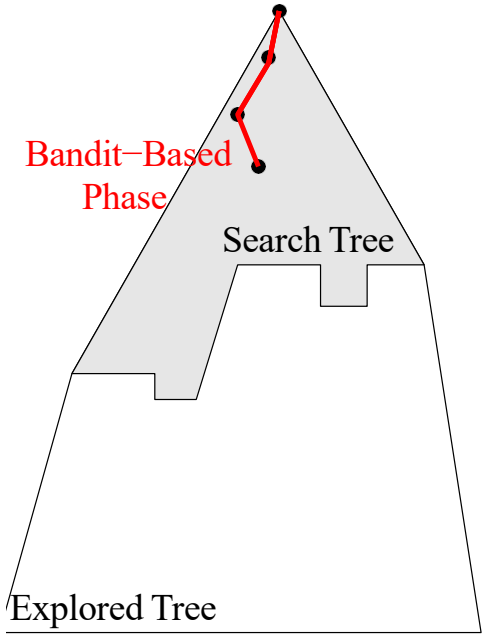
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



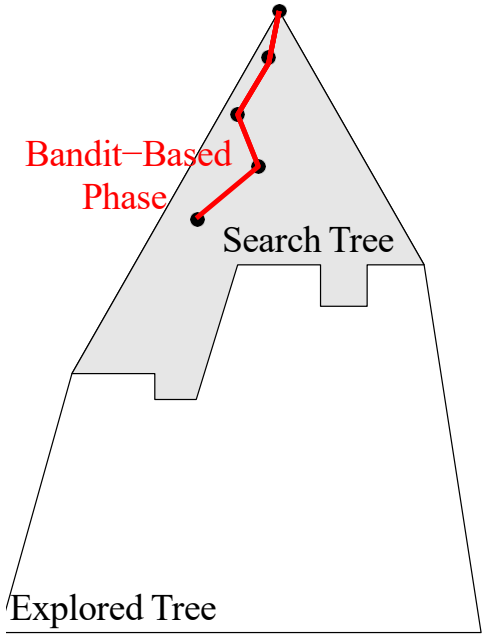
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



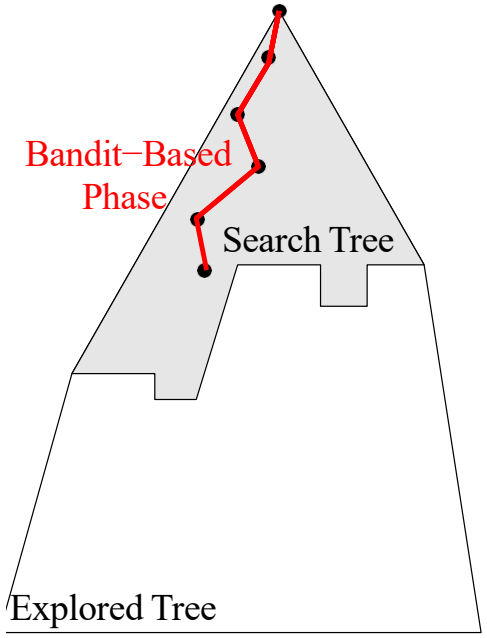
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



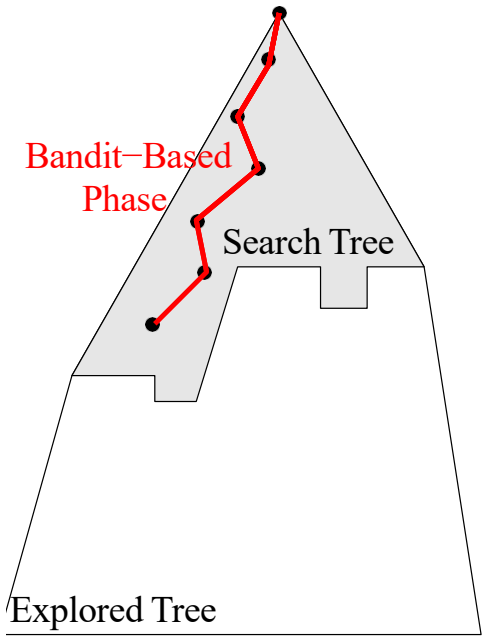
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



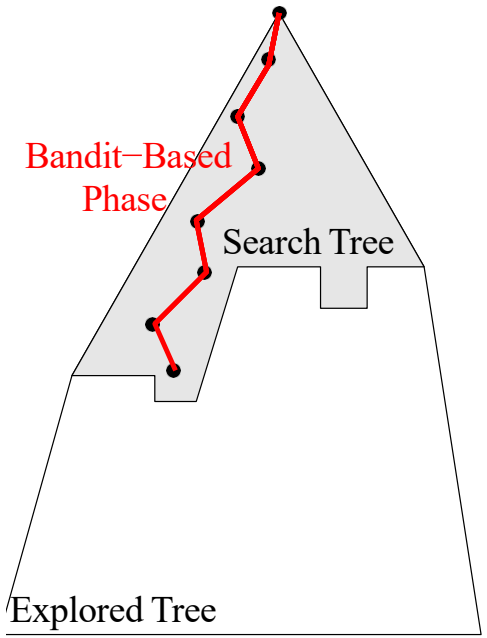
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



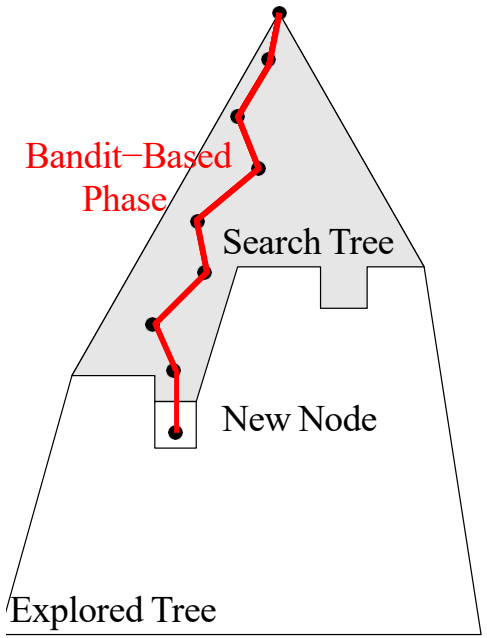
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



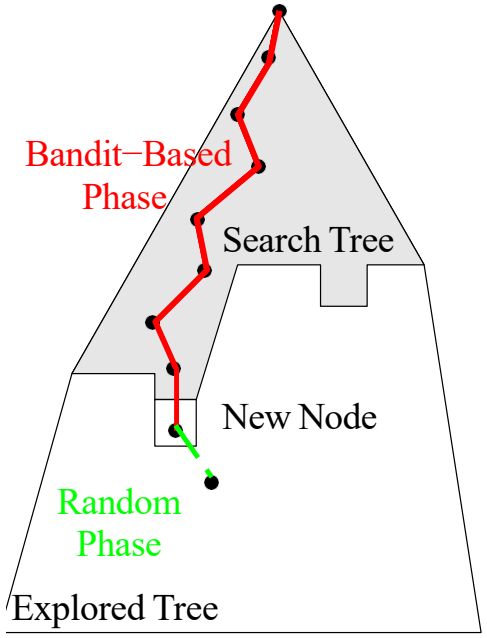
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



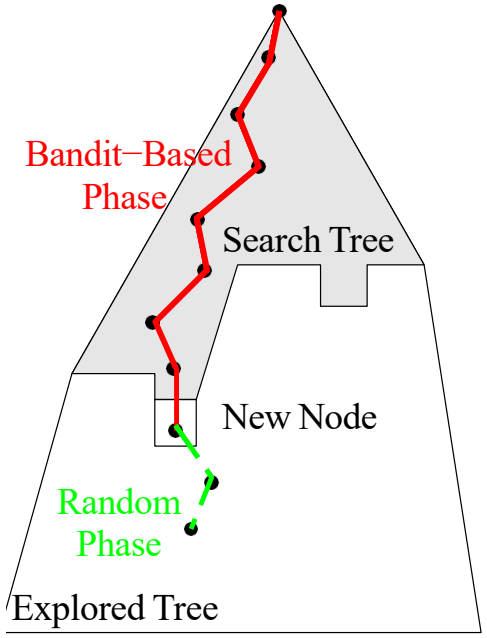
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



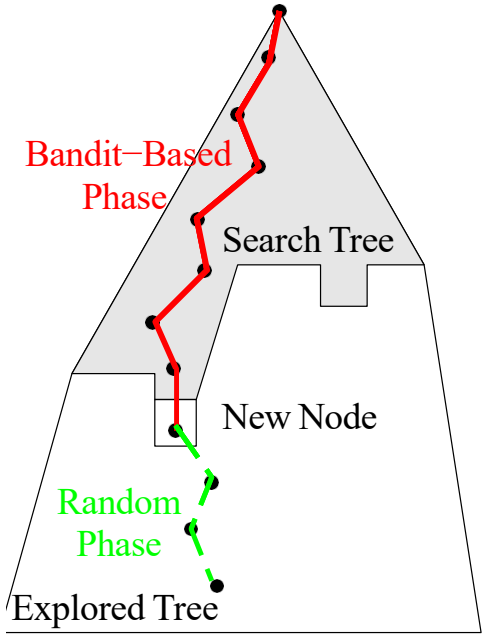
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



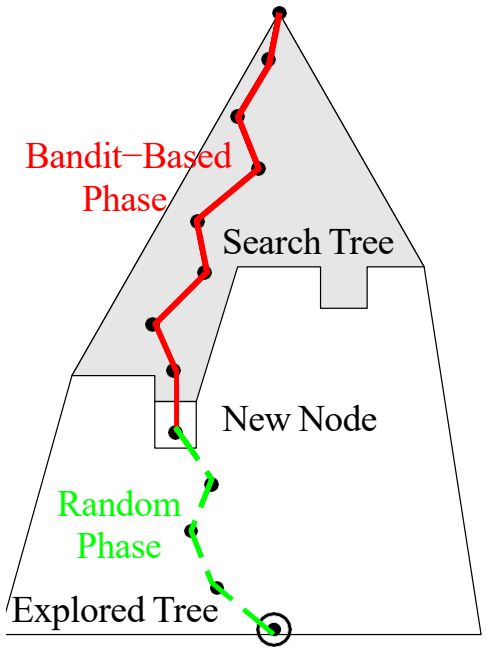
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



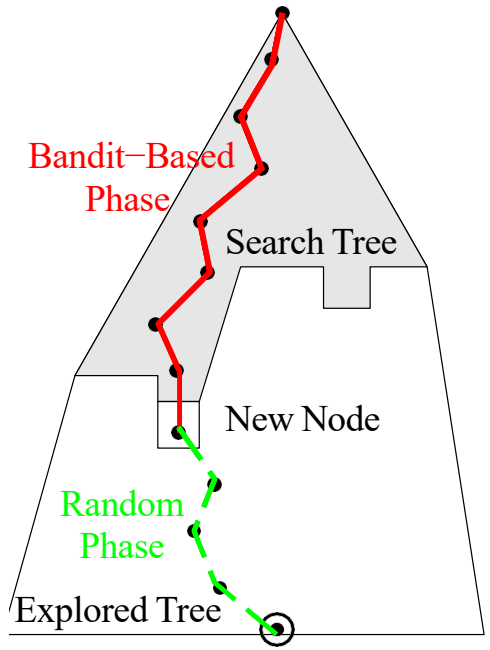
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



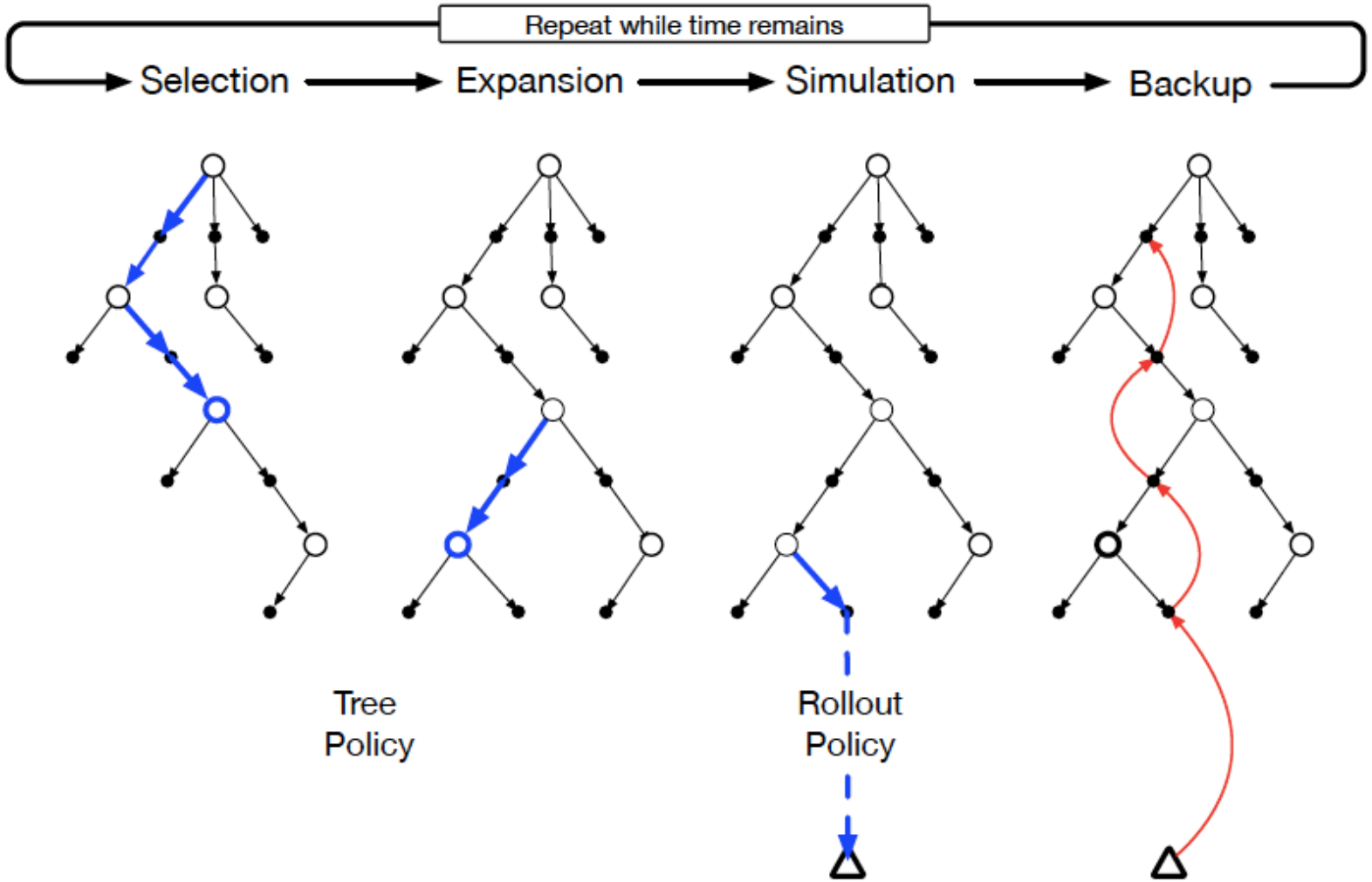
```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



```
function MCTS_sample(node)
    if all children expanded:#selection
        next = UCB_sample(node)
        outcome = MCTS_sample(next)
    else:#expansion
        next = random unexpanded child
        create node for next, add to tree
        #simulation
        outcome = random_playout(next.state)
    #backpropagation
    update_value(node, outcome)
```



Monte-Carlo Tree Search



Illustration

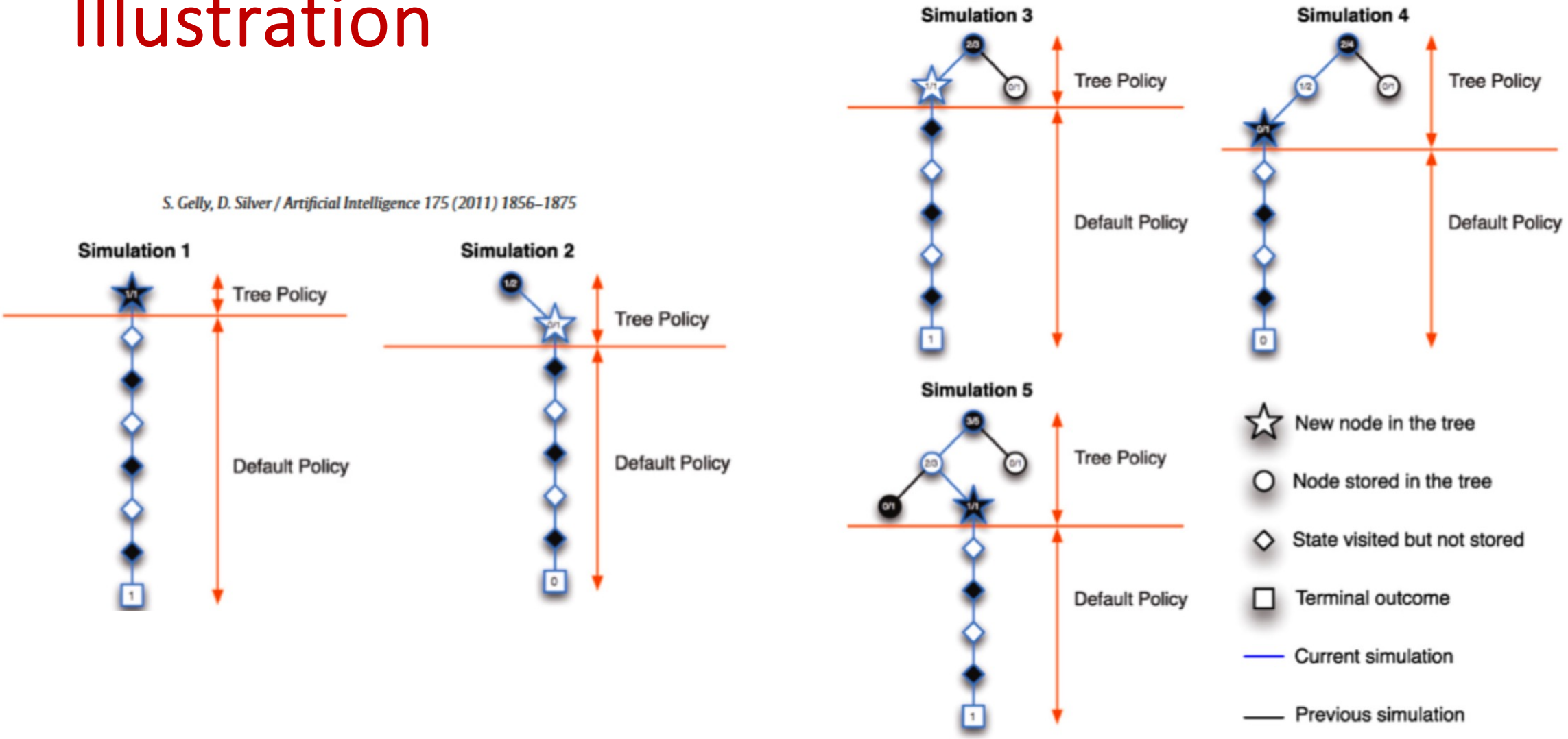


Figure 2: Demonstration of a simple MCTS. Each state has two possible actions (left/right) and each simulation has a reward of 1 or 0. At each iteration a new node (star) is added into the search tree. The value of each node in the search tree (circles and star) and the total number of visits is then updated

Pseudo-code

Algorithm 1 General MCTS algorithm

function MCTS(s_0)
    Create root node v_0 corresponding to s_o
    while within computational budget **do**
        $v_k \leftarrow TreePolicy(v_o)$
        $\Delta \leftarrow Simuation(v_k)$
        $Backprop(v_k, \Delta)$
    return $arg\max_a Q(s_o, a)$

Algorithm 2 Greedy Tree policy

function TREEPOLICY(v)
    $v_{next} \leftarrow v$
    if $|Children(v_{next})| \neq 0$ **then**
        $a \leftarrow arg\max_{a \in A} Q(v, a)$
        $v_{next} \leftarrow nextState(v, a)$
        $v_{next} \leftarrow TreePolicy(v_{next})$
    return v_{next}

Algorithm 3 Upper Confidence Tree policy

function TREEPOLICY(v)
    $v_{next} \leftarrow v$
    if $|Children(v_{next})| \neq 0$ **then**
        $a \leftarrow arg\max_{a \in A} Q(v, a) + \sqrt{\frac{2 \log N(v)}{N(v,a)}}$
        $v_{next} \leftarrow nextState(v, a)$
        $v_{next} \leftarrow TreePolicy(v_{next})$
    return v_{next}

Procedure – Backpropagation($s : S; a : A$)

Input: state-action pair (s, a)
Output: none

do
    $N(s, a) \leftarrow N(s, a) + 1$
    $G \leftarrow r + \gamma G$
    $Q(s, a) \leftarrow Q(s, a) + \frac{1}{N(s,a)}[G - Q(s, a)]$
    $s \leftarrow$ parent of s
    $a \leftarrow$ parent action of s
while $s \neq s_0$

References: https://web.stanford.edu/class/cs234/CS234Win2020/slides/lnotes14.pdf
https://gibberblot.github.io/rl-notes/single-agent/mcts.html