CSE599i: Online and Adaptive Machine Learning **Winter 2018**

Lecture 19: Monte Carlo Tree Search

Lecturer: Kevin Jamieson *Scribes: Christopher Mackie, Hunter Schafer, Nathaniel Yazdani*

1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a policy-optimization algorithm for finite-horizon, finite-size Markov Decision Processes, based on random episode sampling structured by a decision tree [1]. In this section, we describe Monte Carlo Tree Search in generality, laying the groundwork for the bandit-based extension discussed in the following section. As a preview, that extension essentially provides an effective yet still generic instantiation, with proven probabilistic guarantees, for an especially challenging implementation-specific component of MCTS.

1.1 Summary of Markov Decision Processes

A *Markov Decision Process* (MDP) is a probabilistic model for reward-incentivized, memoryless, sequential decision-making. An MDP models a scenario in which an *agent* (the decision maker) iteratively observes the current *state*, selects an *action*, observes a consequential probabilistic state *transition*, and receives a *reward* according to the outcome. Importantly, the agent decides each action based on the current state alone and not the full history of past states, providing a Markov independence property. Mathematically, an MDP consists of the following components:

- a state space (set), $\mathcal{X}$;
- an action space (set), $\mathcal{A}$;
- a transition probability function, $\mathcal{P} : \mathcal{X} \times \mathcal{A} \times \mathcal{X} \to [0, 1]$; and
- a reward function, $\mathcal{R} : \mathcal{X} \times \mathcal{A} \to [0, 1]$.

MCTS places some restrictions on the kind of MDPs that we may consider. We require that an MDP under consideration be finite; this means that the state- and action-spaces must both have finite size, though they may be fairly large nonetheless. We also require that the MDP be *finite-horizon*; this property means that any transition sequence $x_0, a_{0,1}, x_1, a_{1,2}, x_2, \ldots$ of nonzero probability (*i.e.*, $\prod_{i \geq 0} \mathcal{P}(x_i, a_{i,i+1}, x_{i+1}) > 0$) must have finite length. A finite-length transition sequence of which every extension has zero probability is called an *episode* of the MDP. Given the Markov independence inherent in any MDP (*i.e.*, transition probability independent of past history), the finite-horizon restriction implies that some states of the MDP are *terminal*, such that any transition from one has zero probability. By analogy, states that are not terminal are called *nonterminal*. Lastly, MCTS assumes an undiscounted MDP; that is, MCTS assumes that the MDP has unit discount factor. Technically, one could easily adapt MCTS to work with non-unit discount factors, but requiring a finite-horizon MDP typically makes discounting irrelevant.

From the components of an MDP, we define the *optimal state value function* $V^* : \mathcal{X} \to \mathbb{R}$ and *optimal state-action value function* $Q^* : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$:

$$V^*(x) = \max_{a \in \mathcal{A}} Q^*(x, a)$$
$$Q^*(x, a) = \mathcal{R}(x, a) + \sum_{x' \in \mathcal{X}} \mathcal{P}(x, a, x')V^*(x')$$

The function V gives the maximum expected reward for any decision policy when starting from state x, and the Q function similarly gives the maximum expected reward for any decision policy when starting from state x with action a. A rigorous derivation of these functions and proof of their stated properties are beyond the scope of these lecture notes. The classic problem for an MDP is to come up with a *decision policy* that is optimal with respect to expected reward. A decision policy — or simply "policy" — is a function from states to actions, $\pi : \mathcal{X} \to \mathcal{A}$, and for such a policy to be optimal with respect to expected reward, it must hold that $\pi(x) = \arg\max_{a \in \mathcal{A}} Q(x, a)$ for any state $x \in \mathcal{X}$. The intuition for this problem, sometimes called *planning*, is that an agent following an optimal decision policy can maximize its acquired reward, in expectation.

Several algorithms exist to calculate or approximate such an optimal policy (*e.g.*, Q-learning), but most such algorithms become either computationally intractable or hopelessly imprecise when the state- and/or action-spaces are very large. As an algorithm for policy optimization, MCTS has shown great success for such larger-scale problems. Two qualities of MCTS critical to this success are that it is *incremental* and that it is *iterative*. MCTS is incremental in the sense that it approximates each decision in the optimal policy individually (*i.e.*, step by step), and it is iterative in the sense that it can use whatever available computation budget to improve its approximation of the optimal decision as much as possible. In combination, these two qualities enable MCTS to scale to MDPs of seemingly intractable size, by focusing its computational resources on the next local decision of immediate concern.

1.2 Applications to Games

Practitioners most frequently apply MCTS is *game theory*, where the goal is to learn the optimal policy for a Markov game. A Markov game can be thought of an extension of game theory to MDP-like environments where a player may take an action from a state, but the reward and state transitions are uncertain as they depend on the adversary's strategy [2]. Generally the focus is on two-player zero-sum Markov games since it is a well researched area of game theory where there is no concern of cooperation between players and the rewards are symmetric for players except for positive/negative values.

Markov games are much like MDPs where there is a set of states S and a set of actions A_i for each player i; in the two-player setting call the action set of the player A and the opponent O. Define the reward of the player's action a in state s where the opponent plays action o in return as $R(s, a, o)$ and where the system will transition to a new state s' after the moves where $s' \sim T(s, a, o, \cdot)$ for the transition kernel T. Since we are considering zero-sum games, the absolute value of the reward is the same for each player but is positive for the current player and negative for the opponent; this simplifies analysis so we can deal with just one reward function and change the sign depending on the player. For most common games like Go and Chess the transition and reward functions are deterministic given the actions of the player and the opponent, but we consider them non-deterministic values sine the player and opponent may use randomized strategies.

Finding an optimal policy in this scenario seems impossible since it depends critically on which adversary is used. The way this is resolved is by evaluating a policy with respect to the worst opponent for that policy. The goal now is to find a policy that will maximize the reward knowing that this worst case opponent will then minimize the reward after the action is played (the fact that this is a zero-sum game makes it so the opponent will maximizes your negative reward); this idea is used widely in practice in what is known as the *minimax* principle. This optimal policy is a bit pessimistic since you won't always be playing against a worst-case opponent for that policy, but it does allow to construct a policy for a game that can be used against any adversary. One technical note is it's common to allow a policy to be non-deterministic as it might not be optimal to play deterministically in games like rock-paper-scissors.

Using this minimax strategy changes the optimal value function for a state $s \in S$

$$V^*(s) = \max_{\pi \in \Delta^{|A|-1}} \min_{o \in O} \sum_{a \in A} Q(s, a, o)\pi_a$$

where $\Delta^{|A|-1}$ is the probability simplex over the A actions. The quality of action a in state s against action o is

$$Q^*(s, a, o) = R(s, a, o) + \gamma \sum_{s'} T(s, a, o, s')V^*(s')$$

One algorithm for finding this minimax optimal policy called minimax-Q is exactly like Q-learning except using a min-max instead of just a max like in the definition of the state value above [2].

1.2.1 Minimax Trees

Another popular algorithm to estimate the optimal action from a state against this worst case opponent is the Minimax algorithm. The idea behind this algorithm is that the opponent will play optimally given your action and select an action that maximizes their reward; therefore you should select an action that maximizes your reward given that the opponent will be minimizing your reward after your action has been chosen. To compute which action is optimal, the algorithm forms a game tree from the current state s is the root node where each action is a branch to nodes of result states s' (where s' is the state resulting from taking action a from state s). The important part of this tree representation of choices is that each level of the tree alternates which players it is representing; on the first level it is the first player, the second layer is their opponent (the second player), the third is the first player, etc. The Minimax tree allows us to explore all possible scenarios up to a certain depth of the tree to simulate what the opponent will do under this worst-case assumption.

For the same reasons listed above, Q-learning and Minimax like strategies may be infeasible for games like Go where the state and action spaces are incredibly large; real-world games can have hundreds of actions and over a quindecillion (*i.e.* 1 trillion trillion trillion trillion) possible states. Instead what is used is a version of MCTS that explores the Minimax game tree described above in a depth-first wise fashion to estimate the optimal action.

1.3 The Algorithm

The MCTS algorithm centers around iterative manipulation of a *decision tree*, happening in four phases: *selection*, *expansion*, *simulation*, and *backpropagation*. Figure 1 diagrams how the phases of an iteration interact with the decision tree. To instantiate MCTS for an MDP, the user must, at least, supply a representation of states and of actions; a handful of operations on states and actions (*e.g.*, enumerate actions and determine whether state is terminal); and a generative model of the MDP that stochastically generates a new state given a nonterminal state and an action. At a high level, MCTS repeatedly samples episodes from this generative model in order to approximate the expected reward down the various paths recorded in the decision tree. In the following paragraphs, we explain these core aspects of MCTS in detail.

The decision tree encodes prefixes of episodes – maximal finite transition sequences of nonzero probability – observed in the current run of MCTS. The nodes of the decision tree represent states of the underlying MDP, with the root's state given as input to the algorithm. A branch from a parent node to a child node is labeled by the action taken in sampled transitions from the parent's state to the child's state. Figure 2 illustrates the structure of a decision tree in MCTS. Notice how multiple branches may share the same action label, because an MDP transitions to a new state stochastically – not deterministically – given a state and action.

Until exhaustion of a user-specified time budget, MCTS iteratively samples full episodes (originating at the root's state) from the generative model, expanding the decision tree incrementally for each observed episode. To sample an episode, MCTS needs a decision policy with which to feed actions to the generative model. The selection phase and simulation phase of an MCTS iteration determine this decision policy

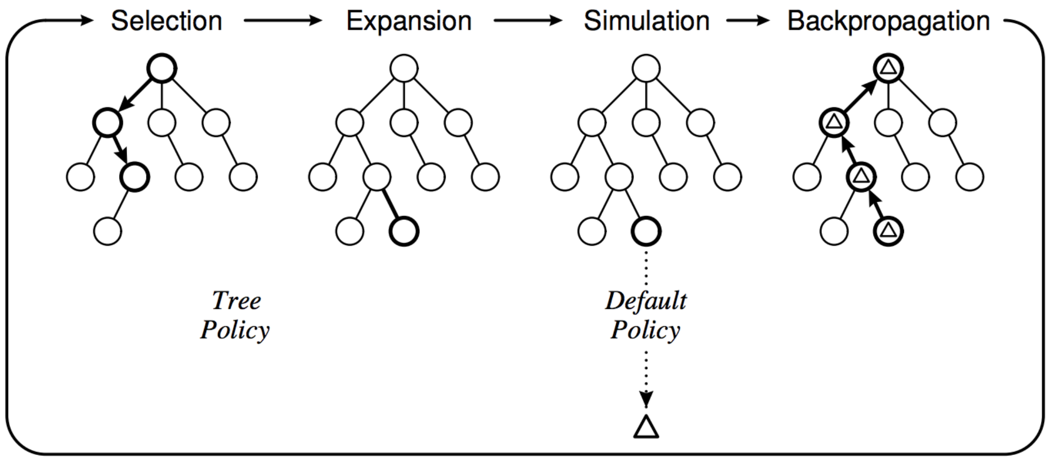

Figure 1: Diagram of the phases of an MCTS iteration (credit to [1]).

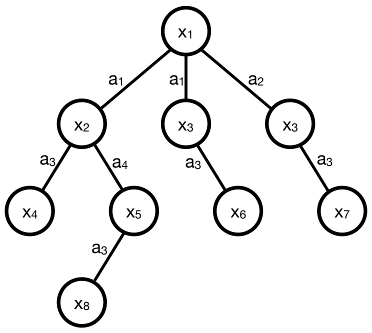

Figure 2: Illustration of a decision tree in MCTS.

dynamically. Selection induces a decision policy, known as the *tree policy*, to navigate through the existing decision tree, attempting to strike a balance between exploration of unknown decision paths and exploitation of known, promising decision paths. Once selection reaches the end of the existing decision tree – by observing a transition triple unrecorded in the decision tree – the tree is expanded with a new leaf node, and simulation begins. Simulation aims to choose actions quickly and stochastically, in order to finish sampling the episode as quickly as possible. When the episode is complete (having reached a terminal state), MCTS backpropagates the reward accumulated in the episode up the decision path taken by selection.

Since the goal of MCTS is to optimize policy decisions from the root state, MCTS accumulates statistics from sampled episodes in each node. In particular, MCTS tracks a quantity N, the number of observed transitions into a node's state from its parent's state using the branch's action, and a quantity R, the total reward observed in any sampled episode after passing through the node. As these quantities better approximate probabilistic expectation, $R_{x,a}/N_{x,a}$ better approximates the Q-value for a state x and action a, where $R_{x,a}$ is the total reward observed for all children of the node for x connected by a branch for a (respectively for $N_{x,a}$).

Algorithm 1 presents the pseudocode for the MCTS algorithm. As hinted at in the introduction, generic MCTS is highly configurable. At the very least, instantiating the algorithm requires providing the MDP's probabilistic transition relation $\mathcal{P} : \mathcal{X} \times \mathcal{A} \times \mathcal{X} \to [0,1]$ and reward function $\mathcal{R} : \mathcal{X} \times \mathcal{A} \times \mathcal{X} \to [0,1]$, but implementors must also define exploratory action selection (SelectAction), which characterizes the tree policy. While not strictly necessary, implementors routinely tweak simulative action selection (RandomAction), which similarly characterizes the default policy, and the determination for the approximately best action (BestAction) as well. As a practical matter, implementations typically use a technique like transposition tables to mitigate state duplication within the decision tree, effectively turning the decision *tree* into a decision *dag* (directed acyclic graph).

Algorithm 1 Monte Carlo Tree Search

```
procedure MonteCarloTreeSearch($x_0$)
    $\tau_0 \leftarrow$ MakeTree($x_0$)    ▷ Initialize decision tree with root state $x_0$
    repeat    ▷ Main, iterative loop of the algorithm
        $\tau_i, r \leftarrow$ TreePolicy($\tau_0$)    ▷ Exploration, returning new leaf and current reward
        $x_i \leftarrow \tau_i$.state
        $r \leftarrow$ DefaultPolicy($x_i$)    ▷ Simulation, completing episode and returning total reward
        Backpropagate($\tau_i$, $r$)    ▷ Update node statistics along exploration path
    until Timeout()
    return BestAction($\tau_0$)    ▷ Pick the approximately optimal root-level action
end procedure

procedure TreePolicy($\tau$)    ▷ Decision policy for exploration
    $r \leftarrow 0$
    $x \leftarrow \tau$.state
    while Nonterminal($x$) do
        $a \leftarrow$ SelectAction($x$)    ▷ Heuristically select an action from $\mathcal{A}$
        $x' \leftarrow$ Transition$_{\mathcal{P}}$($x$, $a$)    ▷ Sample transition from generative model
        $r \leftarrow r + \mathcal{R}(x, a, x')$
        if $\tau$.children[$a$][$x'$] = null then    ▷ Is this the first observation of $x \xrightarrow{a} x'$?
            $\tau$.children[$a$][$x'$] $\leftarrow$MakeTree($x'$)    ▷ Initialize leaf node for state $x'$
            return $\tau$.children[$a$][$x'$], $r$    ▷ Move on to simulation phase
        end if
        $\tau \leftarrow \tau$.children[$a$][$x'$]
        $x \leftarrow x'$
    end while
    return $\tau, r$
end procedure

procedure DefaultPolicy($x, r$)    ▷ Decision policy for simulation
    while Nonterminal($x$) do
        $a \leftarrow$ RandomAction($x$)    ▷ Randomly select an action from $\mathcal{A}$
        $x' \leftarrow$ Transition$_{\mathcal{P}}$($x$, $a$)    ▷ Sample transition from generative model of $\mathcal{P}$
        $r \leftarrow r + \mathcal{R}(x, a, x')$
    end while
    return $r$
end procedure

procedure Backpropagate($\tau, r$)    ▷ Update statistics along path to this tree node
    repeat
        $\tau$.reward $\leftarrow \tau$.reward $+ r$
        $\tau$.count $\leftarrow \tau$.count $+ 1$
        $\tau \leftarrow \tau$.parent
    until $\tau$ = null
    return
end procedure

procedure BestAction($\tau$)    ▷ Pick the approximately best action at this tree node
    return $\arg\max_{a \in \mathcal{A}} \frac{\sum_{\tau' \in \tau.\text{children}[a][\cdot]} \tau'.\text{reward}}{\sum_{\tau' \in \tau.\text{children}[a][\cdot]} \tau'.\text{count}}$
end procedure
```

2 MCTS with Multi-armed Bandits

Generally speaking, MCTS uses a simple timeout as its stopping condition. Therefore, a desirable property for MCTS is that the probability of selecting a sub-optimal decision quickly approaches zero as the algorithm is given more time. Moreover, another related desirable property is that the probability of a sub-optimal decision will indeed converge to zero given unbounded time. Unfortunately, general MCTS does not guarantee any of the properties above. As we will see, there is an algorithm, UCT (UCB applied to trees) [3], which achieves the desired properties by taking a statistical approach to balancing the exploration-exploitation trade-off.

2.1 Multi-armed Bandits

To understand UCT, one must first understand *multi-armed bandits*, a statistical problem integral to the UCT algorithm. In this subsection, we briefly review the problem and a common algorithm, UCB1 [4], used by UCT.

The premise of multi-armed bandits is this: At each time step $t \geq 1$, the player must choose one of K arms — each associated with an independent random reward sequence $X_{i,t}$ for $i \in \{1..K\}$ — from which to receive a reward, based on the player's past observation of rewards from each arm. For simplicity, these rewards are generally assumed to lie in $[0, 1]$, and we assume a fixed mean μ_i for each arm, with a unique arm having a maximum mean μ^*. Some policy $\pi(t)$ picks an arm $I_t \in \{1..K\}$ to pull at time t. The regret of such a policy is the loss incurred for not having picked the best arm always, which is shown in Equation 1, where $T_i(t) = \sum_{s=1}^{t} \mathbf{1}(I_s = i)$ is the number of times arm i has been pulled up to time t.

$$R_n = \max_i \left\{ \mathbb{E}\left[\sum_{t=1}^{n} X_{i,t} \right] \right\} - \mathbb{E}\left[\sum_{j=1}^{K} \sum_{t=1}^{T_j(n)} X_{j,t} \right] \tag{1}$$

It has been shown that the best possible policy $\pi(t)$ has a regret bounded by $\Omega(\ln n)$, and any policy that is within a constant factor of this is considered to balance the exploration-exploitation trade-off in the tree policy well. One such algorithm is UCB1. [3]

The high level concept of UCB1 is simple: pick the arm with the highest confidence bound on its mean reward. To facilitate this, UCB1 tracks the empirical average of each arm's rewards $\bar{X}_{i,T_i(t-1)}$ up to time t. As a result, when incorporating UCB into MCTS, it becomes necessary to track empirical average reward in MCTS. The policy is outlined in Equation 2, where $c_{t,s}$ is a bias sequence over time t and sample size s, picked such that both Equation 3 and Equation 4 are satisfied.

$$\text{UCB1}(t) = \operatorname*{arg\,max}_{i \in \{1..K\}} \left\{ \bar{X}_{i,T_i(t-1)} + c_{t-1,T_i(t-1)} \right\} \tag{2}$$

$$\mathbb{P}(\bar{X}_{i,s} \geq \mu_i + c_{t,s}) \leq t^{-4} \tag{3}$$

$$\mathbb{P}(\bar{X}_{i,s} \leq \mu_i - c_{t,s}) \leq t^{-4} \tag{4}$$

For standard multi-armed bandits, the bias sequence $c_{t,s} = \sqrt{\frac{2 \ln t}{s}}$ is appropriate. Note that the bias sequence is proportional to time and inversely proportional to sample size. It is designed so that confidence bounds expand slowly over time, but contract quickly as we gather more data.

2.2 Drifting Multi-armed Bandits

For more detail on any of the theorems discussed in this section, please refer to [5]. In order to understand UCT, we must first relax the restraints on K-armed bandits to allow each arm's mean μ_i to change over time

t. While we can no longer rely on the assumption that the mean of each arm is fixed from $t = 1$ onward, we can assume that the expected value of the empirical averages converge. We let $\bar{X}_{i,n} = \frac{1}{n}\sum_{t=1}^{n} X_{i,t}$ be the empirical average of arm i at time n, and $\mu_{i,n} = \mathbb{E}[\bar{X}_{i,n}]$ be its expected value. Therefore, we now have a sequence of expected means for each arm i, namely $\mu_{i,n}$. We assume that these expected means eventually converge to one final mean $\mu_i = \lim_{n\to\infty} \mu_{i,n}$. We further define a sequence of offsets for each arm as $\delta_{i,n} = \mu_{i,n} - \mu_i$. We also make the following assumptions about the reward sequence:

Assumption 1. *Fix $1 \le i \le K$. Let $\{\mathcal{F}_{i,t}\}_t$ be a filtration such that $\{X_{i,t}\}_t$ is $\{\mathcal{F}_{i,t}\}$-adapted and $X_{i,t}$ is conditionally independent of $\mathcal{F}_{i,t+1}, \mathcal{F}_{i,t+2}, \ldots$ given $\mathcal{F}_{i,t-1}$[1]. Then $0 \le X_{i,t} \le 1$ and the limit of $\mu_{i,n} = \mathbb{E}[\bar{X}_{in}]$ exists. Further, we assume that there exist a constant $C_p > 0$ and an integer N_p such that for $n \ge N_p$, for any $\delta > 0$, $\Delta_n(\delta) = C_p\sqrt{n\ln(1/\delta)}$, the following bounds hold:*

$$\mathbb{P}(n\bar{X}_{i,n} \ge n\mathbb{E}[\bar{X}_{i,n}] + \Delta_n(\delta)) \le \delta$$
$$\mathbb{P}(n\bar{X}_{i,n} \le n\mathbb{E}[\bar{X}_{i,n}] - \Delta_n(\delta)) \le \delta$$

This assumption allows us to define a bias sequence $c_{t,s}$ for time t and sample size s which satisfies Equation 3 and Equation 4. This sequence is as follows:

$$c_{t,s} = 2C_p\sqrt{\frac{\ln t}{s}} \tag{5}$$

We define $\Delta_i = \mu^* - \mu_i$ to be the loss of arm i. Recall that since the expected mean of each arm converges, the mean offset $\delta_{i,t}$ converges to zero. Therefore, there exists a time $N_0(\epsilon)$ at which the uncertainty of the true mean rewards are guaranteed to be within a factor ϵ of their distance from the optimal mean, and the uncertainty of the optimal mean is guaranteed to be within the same factor ϵ of its distance to to closest suboptimal mean. Therefore, even though we still have some uncertainty as to what the true means really are, we have enough information to know which is probably the best, as $\mu^*_{N_0(\epsilon)}$ is closer to μ^* than it is to any $\mu_{i,N_0(\epsilon)}$. More formally, $N_0(\epsilon) : \mathbb{R} \to \mathbb{N}$ is a function which returns the minimum t for which $2|\delta_{i,t}| \le \epsilon\Delta_i$ for all arms i, and $2|\delta_{j^*,t}| \le \epsilon \min_i \Delta_i$. Under Assumption 1, and using the preceding definitions, we can upper bound the expected number of times that a suboptimal arm will be played by UCB1 when the means are allowed to drift. The bounds follow:

Theorem 1. *Consider UCB1 applied to a non-stationary problem where the pay-off sequence satisfies Assumption 1 and where the bias sequence, $c_{t,s}$, used by UCB1 is given by Equation 5. Fix $\epsilon > 0$. Let $T_i(n)$ denote the number of plays of arm i. Therefore if i is the index of a suboptimal arm then*

$$\mathbb{E}[T_i(n)] \le \frac{16C_p^2 \ln n}{(1-\epsilon)^2\Delta_i^2} + N_0(\epsilon) + N_p + 1 + \frac{\pi^2}{3}$$

.

To provide an intuition for the above bound, we quickly review some of the values invoked in it. First of all, recall that N_p is a constant chosen based on the definition of the drifting means bias sequence $c_{t,s}$ for time t and sample size s. It is a lower bound on t such that another constant C_p can be used in Equations 3 and 4. Also we reference the value $N_0(\epsilon)$. This value is a lower bound for t at which μ^*_t is closer to μ^* than it is to any $\mu_{i,t}$. The value also incorporates a notion of how small the distance is using ϵ.

Because we allow the means to drift, we cannot directly use regret as a benchmark for the effectiveness of UCB, but we can instantiate a similar measure, and prove a bound on it using Theorem 1. This measure involves a few odd quantities for which we will attempt to give some intuition. Let $\bar{X}_n = \sum_{i=1}^{K} \frac{T_i(n)}{n}\bar{X}_{i,T_i(n)}$, where $\frac{T_i(n)}{n}$ is the empirical probability of pulling arm i, and $\bar{X}_{i,T_i(n)}$ is the empirical mean of arm i. Therefore $\bar{X}_n$ is like an empirical expected reward for UCB at time $t = n$. We cannot use this value directly to compare to the best possible reward μ^*, because $T_i(n)$ is itself a random variable. Therefore the measure $|\mathbb{E}[\bar{X}_n] - \mu^*|$ is similar to regret, as it is proportional to the loss incurred for not having always played the best arm. A bound on the measure follows:

[1]If this terminology is unfamiliar, it is sufficient to say that the reward process can only rely on past information, it cannot see into the future.

Theorem 2. *Under the assumptions of Theorem 1,*

$$|\mathbb{E}[\bar{X}_n] - \mu^*| \leq |\delta_n^*| + O\left(\frac{K(C_p^2 \ln n + N_0)}{n}\right)$$

where $N_0 = N_0(1/2)$*.*

To provide further intuition for the above, recall that δ_n^* is the distance between the optimal arm's mean at time $t = n$, and the true optimal mean. Furthermore, the theorem is proven using the value $N_0(\epsilon)$ where $\epsilon = 1/2$.

Furthermore, using Theorem 1, we can derive a lower bound on the number of times some arm i will be played. This result follows:

Theorem 3. *Under the assumptions of Theorem 3, there exists some positive constant* ρ *such that for all arms* i *and* n, $T_i(n) \geq \lceil \rho \log(n) \rceil$*.*

We need the above result to prove that the optimal reward converges quickly to its true mean in the drifting multi armed bandit problem. We omit the proof of the result, but it is natural that it would require both the lower bound on $T_i(n)$ and the upper bound on $\mathbb{E}[T_i(n)]$. This quick convergence is one of the two most important properties for incorporating UCB into MCTS. If the drifting means converge slowly then we cannot hope to quickly find an optimal arm. The bound follows:

Theorem 4. *Fix an arbitrary* $\delta > 0$ *and let* $\Delta_n = 9\sqrt{2n \ln 2/\delta}$*. Let* n_0 *be such that*

$$\sqrt{n_0} \geq O(K(C_p^2 \ln n_0 + N_0(1/2)))$$

Then for any $n \geq n_0$*, under the assumptions of Theorem 1 the following bounds hold true:*

$$\mathbb{P}(n\bar{X}_n \geq n\mathbb{E}[\bar{X}_n] + \Delta_n) \leq \delta$$
$$\mathbb{P}(n\bar{X}_n \leq n\mathbb{E}[\bar{X}_n] - \Delta_n) \leq \delta$$

Another extremely important result for using UCB in MCTS is that when the means can drift, UCB still finds the best arm when given infinite time. This result follows:

Theorem 5. *Under the assumptions of Theorem 1 it holds that*

$$\lim_{t \to \infty} P(I_t \neq i^*) = 0$$

2.3 UCT

UCT is simply a marriage of MCTS and UCB1. The main idea is to treat each internal node in MCTS as a K-armed bandit, where the arms are the actions available at that state. A separate instance of UCB1 is run on each internal state, modified to accommodate the drifting means present in this problem. The necessity of the drifting means generalization can be seen by realizing that UCT is essentially a tree of separate UCB1 instances. The means of the action rewards in each of these multi-armed bandit problems depend on the instances lower in the tree. As we explore the tree, we gain a clearer understanding of the true means, reducing noisiness introduced by a partial exploration deeper in the tree. Simply put, UCT is MCTS where we use UCB1 to select an action in the tree policy. See Algorithm 2 for pseudo code of the action selection.

Recall Theorems 4 and 5 from the previous section. These state that when means can drift in the multi armed bandit problem, UCB still finds the optimal solution quickly with high probability, and given enough time it always finds the optimal arm. When incorporating UCB into MCTS, we also get these properties in UCT, as stated in Theorem 6.

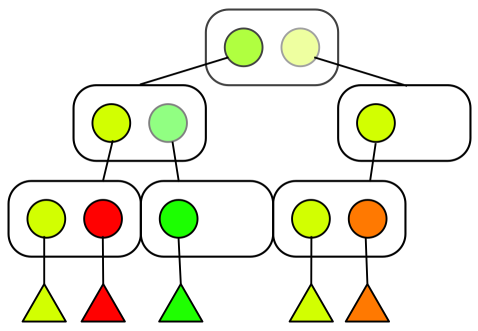

Figure 3: Depiction of a UCT search tree

Algorithm 2 UCT Action Selection

```
procedure SELECTACTION(position, depth)
    nActions ← $|\mathcal{A}|$
    nsum ← 0
    for $i \in \{1..\text{nActions}\}$ do
        nsum ← nsum + position.actions[i].n    ▷ Add the number of times the $i$th action has been chosen
    end for
    maxv ← $-\infty$
    for $i \in \{1..\text{nActions}\}$ do
        if position.actions[i].n = 0 then
            $v \leftarrow +\infty$
        else
            $v \leftarrow \text{position.actions[i].value} + \sqrt{\frac{2 \cdot \ln(\text{nsum})}{\text{position.actions[i].n}}}$
        end if
        if $v >$ maxv then
            maxv ← $v$
            $j \leftarrow i$
        end if
    end for
    return position.actions[j]
end procedure
```

Theorem 6. *Consider algorithm UCT running on a game tree of depth D, branching factor K with stochastic payoffs at the leaves. Assume that the payoffs lie in the interval $[0, 1]$. Then the bias of the estimated expected payoff, $\bar{X}_n$, is $O((KD\log(n) + K^D)/n)$. Further, the failure probability at the root converges to zero as the number of samples grows to infinity.*

A detailed proof of the preceding theorem is beyond the scope of these lecture notes (see [5] for such a proof), however we will provide a sketch of the proof. We must induct on D to prove the theorem. In the base case of $D = 1$, UCT is reduced to a single instantiation of UCB. Therefore Theorems 4 and 5 lead directly to our desired result. In the inductive case, we assume the result holds for any tree of depth $D-1$, and consider a tree of depth D. All of the children of the root node are trees of depth $D-1$ which, by induction, satisfy the theorem. Therefore, we consider only the single UCB instance at the root. Using the theorems from the previous section (particularly Theorems 2, 4, and 5), we can then show that the theorem holds at the root as well.

3 MCTS for Go

The game of Go has proved to be one of the biggest challenges for AI due to its complexity and the effect that moves have on long term success of the game [6]. The application to more sophisticated MCTS algorithms like UCT to games like Go have greatly improved the rankings of Go systems to the top of the amateur level [6].

Recently in 2015, Google's AlphaGo program was the first program to defeat a 9-dan (i.e. top-ranked) professional in a game without handicaps. That version of AlphaGo, named AlphaGo Fan, was fairly complicated and used multiple neural networks trained on expert play to guide decision making. Afterwards, AlphaGo Lee was created that used a similar design as that of AlphaGo Fan, but depended more on data generated from self-play to learn the value functions of states. AlphaGo Lee was the most famous iteration of AlphaGo since it was the one that beat the European champion Lee Sedol in 2016. In 2017, Google released an AI on an online Go server that was able to beat top human players 60-0 named AlphaGo Master. Google published their final design AlphaGo Zero which shared many similarities to AlphaGo Master but removed some complications relating to evaluating states. This final version, AlphaGo Zero, learned the strategies of the game of Go exclusively from games of self-play and using a single neural network to do so.

This section will have no theoretical results but serve rather as an exposition on how to specialize MCTS to be successful in a specific application.

3.1 AlphaGo Zero

AlphaGo Zero is a specific instantiation of the general MCTS algorithm that uses a neural network learned using reinforcement learning to help guide the decision making process. One of the most important things about this system is that it is trained only by playing against itself and improving from there; it learned the strategy of how to play Go from no prior background. To understand how AlphaGo Zero works, we first specify how its decision process fits in the general MCTS framework and then we will describe how it is trained. This entire section uses [7] as a reference so further citations will be omitted.

3.1.1 AlphaGo Zero MCTS

MCTS for AlphaGo works very similarly to how it works in UCT and the terminology is referenced heavily from there. While the computational budget hasn't been exceeded, episodes are played out to iteratively builds up a game tree from a start state s in order to estimate which action is optimal from from s. During each episode, a game is played out by using a Tree Policy to decide a path down the known tree, a leaf node is expanded and it's value estimated using a trained neural network, and that information is percolated back up the tree.

Because the game alternates between two players on each turn, each level of the tree models a different player just like a minimax tree. The algorithm stays the same at each level of the tree but just makes decision to make the current player win. This makes sense for games like Go because it's a perfect information game and you can use the same estimation technique to approximate how the opponent will play optimally with an approximation that improves with computation time.

Phase One: Play-out iterations to build tree

The Tree Policy that determines the path down the tree on each episode that the algorithm is run is:

$$a_t = \arg\max_{a \in \mathcal{A}} Q(s_t, a) + U(s_t, a)$$

This is very similar to UCT because $Q(s_t, a)$ represents an empirical average of the value of taking action a from state s_t. $U(s_t, a)$ is a an additional term to encourage exploration when the value estimates lack high-confidence and favors actions that have a higher prior-probability. Each episode i ends the tree policy at a leaf node s_L^i. After n episodes in total, we define the terms above as

$$N(s,a) = \sum_{i=1}^{n} \mathbb{1}(s,a,i)$$
$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} \mathbb{1}(s,a,i) V(s_L^i)$$
$$U(s,a) = cP(s,a) \frac{\sqrt{\sum_{b \in \mathcal{A}} N(s,b)}}{1 + N(s,a)}$$

where $\mathbb{1}(s,a,i)$ is an indicator that action a was chosen from state s on iteration i, $V(s_L^i)$ is an estimate of the value of a state (in the case of games it is an indication of how likely it is that the current player will win), and $P(s,a)$ is a prior probability of selecting action a from state s. These Q and V functions are exactly estimates of the Q-values and V-values of the game.

The interesting thing to ask is how does AlphaGo Zero determine the values of $V(s)$ and $P(s,a)$. Generally, MCTS uses a Default Policy to complete a sampled episode in order to estimate the value of a state and usually only uses a uniform prior over the actions. AlphaGo Zero instead uses a single neural network f_θ, that when given a state s, outputs $(\boldsymbol{p}, v) = f_\theta(s)$ where $\boldsymbol{p} \in \Delta^{||\mathcal{A}||-1}$ is a probability distribution over the possible actions and $v \in \mathbb{R}$ is a value estimate for that state. The algorithm uses $P(s,a) = \boldsymbol{p}(a)$ and $V(s) = v$. Unlike the standard definition of MCTS, Alpha Go Zero does not use a default policy to perform a rollout in order to achieve an estimate of the value of a state. Alpha Go Zero instead uses this learned network as the sole estimate of the value of the state based off of the games of self-play it has seen in training; previous versions of AlphaGo used a strategy that mixed the result of using a simple default policy as well as a neural network evaluation but they removed the mixing of evaluations for AlphaGo Zero. Details on how the weights θ of the neural network are learned are described section 3.1.2

The other components of AlphaGo Zero's MCTS work similarly to their counterparts in UCT. Expansion occurs at leaf nodes that have been visited more than some threshold number of times. When a leaf node s' is expanded, it is evaluated by the neural network to get $(\boldsymbol{p}', v') = f_\theta(s')$ and it's value v' and probabilities for it's children branches $\boldsymbol{p}$' are recorded for future use. Like Minimax, the layers of the AlphaGo tree at each height alternate between which player the model is predicting for (the turn of the player switches each time one moves). To make the network optimize predictive accuracy for the value and probabilities of the current player, the backpropagation step alternates the reward of the leaf at each step up the tree to be positive/negative depending on the perspective of the player at that level. This process can be seen in Figure4 below.

Phase Two: Determine action to play

After the timeout for building up the tree has expired, AlphaGo Zero will play an action a from the root state s_t with probability proportional to the number of times that action was chosen during Phase One. To do this, AlphaGo Zero creates a probability distribution $\boldsymbol{\pi}_t$ over the actions from the state s_t such that $\boldsymbol{\pi}_t(a) \propto N(s_t, a)^{1/\tau}$ for some hyperparameter τ; when $\tau = 1$ the distribution exactly matches the ratios of the visit counts, while when $\tau \to 0$ the probability mass focuses on the action that was chosen most often. Using this distribution to selects actions improves the performance of AlphaGo Zero because $\boldsymbol{\pi}_t$ is a refinement of the prediction $\boldsymbol{p}_t$ for the start state s_t; as MCTS is allowed to run, it starts selecting actions with high value estimates more frequently rather than relying on the prior probability bonus exploration term.

These two phases can be seen pictorially in Figure 5 a.

3.1.2 Training f_θ

Previous versions of AlphaGo required the use of labeled data from a database of games from expert human players. AlphaGo Zero differs from these versions in the fact that it does not use games labeled by humans

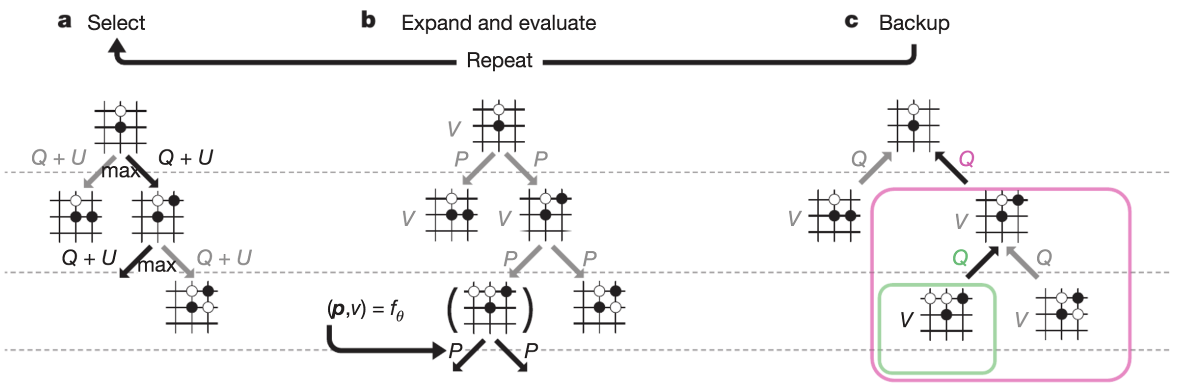

Figure 4: MCTS for AlphaGo Zero [7]. **a.** Uses the tree policy to select a path down the known tree until a leaf node. **b.** A leaf node is expanded by evaluating it using f_θ. **c.** The value is backpropagated up the tree alternating the value according to which player moved at that state (pictured 2 players Green and Pink)

but rather plays games against itself to learn Go strategies. The idea behind the training process is that we can use MCTS as a policy-optimization algorithm to improve the predictions of the neural network.

Consider the case when at a state s_t the network outputs that we should play according to $\boldsymbol{p}_t$. After running MCTS (which involves computing the Q and U terms for each action) for many iterations, we find the empirical distribution of the actions chosen from s_t based on the visit counts is actually $\boldsymbol{\pi}_t$. If $\boldsymbol{\pi}_t$ puts high probability on an action that $\boldsymbol{p}_t$ did not then we should update the model to closer match $\boldsymbol{\pi}_t$ because it favors actions that have higher empirical value estimates as the algorithm is allowed to run longer. Remember that the goal of this process is to make it so that the network can output probability distributions over the actions from the state that can lead to a win. The idea behind this is that since AlphaGo Zero's tree policy eventually starts picking actions that have high value after the exploration term has diminished, the network should be updated to predict distributions that more closely match the empirical one seen by MCTS.

The network f_θ starts with completely random weights and plays games following its outputs. Each game uses the MCTS algorithm described above at each state of the game s_t which is allowed to run for some short amount of time ($\approx 0.4s$) . The action probabilities from the start state $\boldsymbol{\pi}_t$ are recorded for that state s_t and then just like the MCTS algorithm, AlphaGo will pick an action $a_t \sim \boldsymbol{\pi}_t$ and transition to the next state. This process is continued until the end of the game at time T, where the result of the game $z \in \{-1, 1\}$ is computed to indicate if the game was a win/loss. All of the states from the game are used to build up a large training set by recording from the game the sequence $\{(s_t, \boldsymbol{\pi}_t, z_t)\}_{t=1}^{T}$, where $z_t = \pm z$ depending on which is the current player for each time t during that game.

For a specific setting of the hyperparameters, the neural network parameters θ are continuously optimized using stochastic gradient descent using the set of moves from the last 500,000 games of self-play. The loss function that is minimized when considering a training example $(s_i, \boldsymbol{\pi}_i, z_i)$ is

$$\ell_i(\theta) = (z_i - v)^2 - \boldsymbol{\pi}_i^T \log \boldsymbol{p} + c\,||\theta||^2$$

where $(\boldsymbol{p}, v) = f_\theta(s_i)$. Minimizing this loss function minimizes the error predicting the value and makes the probability distributions predicted more similar to the ones found using MCTS by measuring the cross-entropy. This training process can be seen in Fig5.

There are technicalities in this procedure like running multiple processes asynchronously. The following three things are done at the same time concurrently using the processes described above in this section:

1. The currently best performing network is used to generate games of self-play. The best network is determined in Process 3.

2. The current set of weights θ_t are continuously updated using the stochastic gradient descent process described in the paragraph above using the last 500,000 games of self-play data to sample states from.

3. The current set of weights θ_t are used to play against the current best network θ_*. If the current set of weights θ_t defeats the best set of weights θ_* more than 55% of the time during 400 evaluation games, then θ_t becomes the new best parameter values; these are the parameters that will be used for future games of self-play in Process 1.

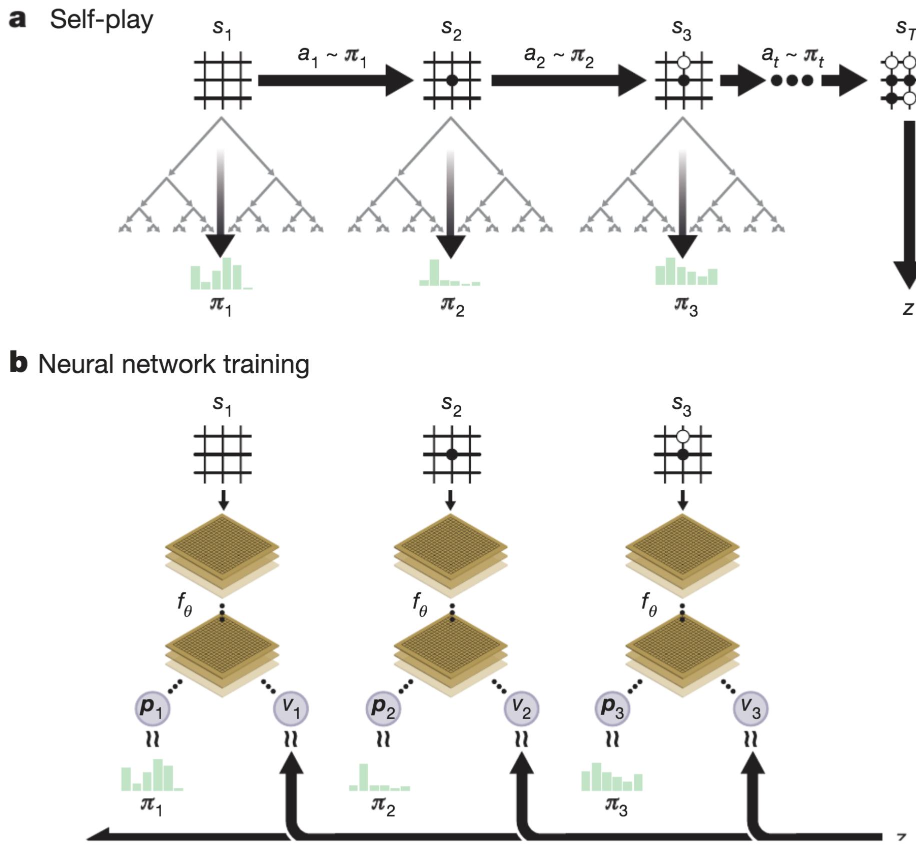

Figure 5: A single game of self-play and updating the network to better match the results [7]

3.1.3 Results

After 36 hours of training, AlphaGo Zero was able to beat AlphaGo Lee which was the best performing system of the previous design that used multiple neural networks and fast rollouts to estimate state values. After only 3 days of training, AlphaGo Zero was able to beat AlphaGo Lee 100 games to 0 in games where AlphaGo Zero was run on 1 machine with 4 Tensor Processing Units (TPUs) while AlphaGo Lee was run in a distributed manner over many machines using 48 TPUs in total. This is an incredible empirical result given that AlphaGo Lee not only was able to beat the world champion in a formal match, but also AlphaGo took weeks to train using millions of example games from expert play while AlphaGo Zero took only a few days and no human interaction at all.

To show that MCTS really is helping refine the policy used by AlphaGo, the researches compared the AlphaGo system against a system that just uses the same neural network to determine the distribution over actions for a state. At a state s_t, this comparison system evaluated $(\boldsymbol{p}, v) = f_\theta(s_t)$ and played an action $a_t \sim \boldsymbol{p}$ without using MCTS to refine the distribution using play-outs. The AlphaGo Zero system got an Elo rating of over 5000 while the system with just the neural network got just above 3000; an Elo gap of 200 corresponds approximately to a 75% probability of winning [8]. This suggests while the design of their network is beneficial and is learning strategies for Go, there is a huge benefit to coupling this with MCTS to look ahead and refine the policy.

References

[1] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, and S. Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, March 2012.

[2] Michael Littman. Markov games as a framework for multi-agent reinforcement learning, 1994.

[3] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *Proceedings of the 17th European Conference on Machine Learning*, ECML'06, pages 282–293, Berlin, Heidelberg, 2006. Springer-Verlag.

[4] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Machine Learning*, 47(2):235–256, May 2002.

[5] Levente Kocsis, Csaba Szepesvri, and Jan Willemson. Improved monte-carlo search.

[6] S. Gelly, L. Kocsis, M. Schoenauer, M. Sebag, D. Silver, Cs. Szepesvári, and O. Teytaud. The grand challenge of computer Go: Monte Carlo tree search and extensions. *Communications of the ACM*, 55(3):106–113, 2012.

[7] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 10 2017.

[8] Remi Coulom. Whole-history rating: A bayesian rating system for players of time-varying strength. *Conference on Computers and Games*, 04 2008.