Monte-Carlo Planning
Look Ahead Trees

Alan Fern
Monte-Carlo Planning Outline

• Single State Case (multi-armed bandits)
  ▶ A basic tool for other algorithms

• Monte-Carlo Policy Improvement
  ▶ Policy rollout
  ▶ Policy Switching

• Monte-Carlo Look-Ahead Trees
  ▶ Sparse Sampling
  ▶ Sparse Sampling via Recursive Bandits
  ▶ UCT and variants
Sparse Sampling

• Rollout and policy switching do not guarantee optimality or near optimality
  ▶ Guarantee relative performance to base policies

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

• In deterministic games and search problems it is common to build a look-ahead tree at a state to select best action
  ▶ Can we generalize this to general stochastic MDPs?
Online Planning with Look-Ahead Trees

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

• $s = \text{current state of environment}$

• Repeat
  ▶ $T = \text{BuildLookAheadTree}(s)$ ;; sparse sampling or UCT
    ;; tree provides Q-value estimates for root action
  ▶ $a = \text{BestRootAction}(T)$ ;; action with best Q-value
  ▶ Execute action $a$ in environment
  ▶ $s$ is the resulting state
Planning with Look-Ahead Trees

Real world state/action sequence

Build look-ahead tree

Build look-ahead tree

\[ R(s_{11}, a_1) R(s_{11}, a_2) R(s_{1w}, a_1) R(s_{1w}, a_2) R(s_{21}, a_1) R(s_{21}, a_2) R(s_{2w}, a_1) R(s_{2w}, a_2) \]

\[ R(s_{11}, a_1) R(s_{11}, a_2) R(s_{1w}, a_1) R(s_{1w}, a_2) R(s_{21}, a_1) R(s_{21}, a_2) R(s_{2w}, a_1) R(s_{2w}, a_2) \]
Sparse Sampling

- Again focus on finite-horizons
  - Arbitrarily good approximation for large enough horizon $h$

- $h$-horizon optimal $Q$-function (denoted $Q^*$)
  - Value of taking $a$ in $s$ and following $\pi^*$ for $h-1$ steps
  - $Q^*(s,a,h) = E[R(s,a) + \beta V^*(T(s,a),h-1)]$

- Key identity (Bellman’s equations):
  - $V^*(s,h) = \max_a Q^*(s,a,h)$
  - $\pi^*(x) = \arg\max_a Q^*(x,a,h)$

- Sparse sampling estimates $Q$-values by building sparse expectimax tree
Sparse Sampling

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

1. Approximation to the full expectimax tree

2. Recursive bandit algorithm
Expectimax Tree

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

• Expand definitions recursively to compute \( V^*(s,h) \)
  \[ V^*(s,h) = \max_{a_1} Q(s,a_1,h) \]
  \[ = \max_{a_1} E[R(s,a_1) + \beta V^*(T(s,a_1),h-1)] \]
  \[ = \max_{a_1} E[R(s,a_1) + \beta \max_{a_2} E[R(T(s,a_1),a_2)+Q^*(T(s,a_1),a_2,h-1)]] \]
  \[ = \quad \ldots \]

• Can view this expansion as an expectimax tree
  \[ \text{Each expectation is a weighted sum over states} \]
Compute root $V^*$ and $Q^*$ via recursive procedure

Depends on size of the state-space. Bad!
Sparse Sampling Tree

- $V^*(s,H)$
- $Q^*(s,a,H)$
- Sampling depth $H_s$
- Sampling width $C$
- Horizon $H$
- # states
- # actions
- (kC)$H_s$ leaves
- $(kn)^H$ leaves
- Replace expectation with average over $w$ samples
- $w$ will typically be much smaller than $n.$
We could create an entire tree at each decision step and return action with highest $Q^*$ value at root.

High memory cost!
Sparse Sampling [Kearns et. al. 2002]

The Sparse Sampling algorithm computes root value via depth first expansion

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

\begin{verbatim}
SparseSampleTree(s,h,w)
If h=0 Return [0, null]

For each action $a$ in $s$
    $Q^*(s,a,h) = 0$
    For $i = 1$ to $w$
        Simulate taking $a$ in $s$ resulting in $s_i$ and reward $r_i$
        $[V^*(s_i,h),a^*] = SparseSample(s_i,h-1,w)$
        $Q^*(s,a,h) = Q^*(s,a,h) + r_i + \beta V^*(s_i,h)$
        $Q^*(s,a,h) = Q^*(s,a,h) / w$ ;; estimate of $Q^*(s,a,h)$

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

Return $[V^*(s,h), a^*]$
\end{verbatim}
Sparse Sampling (Cont’d)

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

- **Good news:** gives values for \( w \) and \( H \) to achieve PAC guarantee on optimality
  - Values are independent of state-space size!
  - First near-optimal general MDP planning algorithm whose runtime didn’t depend on size of state-space

- **Bad news:** the theoretical values are typically still intractably large---also exponential in \( H \)
  - Exponential in \( H \) is the best we can do in general
  - **In practice:** use small \( H \) & heuristic value at leaves
Sparse Sampling w/ Leaf Heuristic

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

### SparseSampleTree\( (s,h,w) \)

- If \( h=0 \) Return \([0, \text{null}]\)
- If \( h=0 \) Return \([ \hat{V}(s), \text{null}]\)

For each action \( a \) in \( s \)

\[
Q^*(s,a,h) = 0
\]

For \( i = 1 \) to \( w \)

Simulate taking \( a \) in \( s \) resulting in \( s_i \) and reward \( r_i \)

\[
[V^*(s_i,h), a^*] = \text{SparseSample}(s_i, h-1, w)
\]

\[
Q^*(s,a,h) = Q^*(s,a,h) + r_i + \beta V^*(s_i,h)
\]

\[
Q^*(s,a,h) = Q^*(s,a,h) / w \quad ;; \text{estimate of } Q^*(s,a,h)
\]

\[
V^*(s,h) = \max_a Q^*(s,a,h) \quad ;; \text{estimate of } V^*(s,h)
\]

\[
a^* = \arg\max_a Q^*(s,a,h)
\]

Return \([V^*(s,h), a^*]\)
Often a shallow sparse sampling search with a simple \( \hat{V} \) at leaves can be very effective.
Sparse Sampling

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

1. Approximation to the full expectimax tree

2. Recursive bandit algorithm
  ▶ Consider horizon H=2 case first
  ▶ Show for general H
Each max node in tree is just a bandit problem.
I.e. must choose action with highest $Q^*(s,a,h)$---approximate via bandit.
Bandit View of Sparse Sampling (H=2)

Consider 2-horizon problem

\[ V^*(s_{11}, 1) \text{ estimate} \]

\[ R(s_{11}, a_1), R(s_{11}, a_2) \]

\[ R(s_{1w}, a_1), R(s_{1w}, a_2) \]

\[ R(s_{21}, a_1), R(s_{21}, a_2) \]

\[ R(s_{2w}, a_1), R(s_{2w}, a_2) \]

\[ h=1: \text{ Traditional bandit problem (stochastic arm reward } R(s_{11}, a_i)) \]

Implement bandit alg. to return estimated expected reward of best arm
**Bandit View of Sparse Sampling (H=2)**

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

**Pulling an arm returns a Q-value estimate by**: 1) sample next state $s'$, 2) run $h=1$ bandit at $s'$, return immediate reward + estimated value of $s'$
Consider UniformBandit using \( w \) pulls per arm
Bandit View: General Horizon H

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

- If this is (approx) satisfied then bandit algorithm will select near optimal arm.
Bandit View: General Horizon H

Definition:

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

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

SimQ*(s, a, h)
$r = R(s,a)$

If h=1 then Return $r$

$s' = T(s,a)$

Return $r + \beta$ BanditValue(SimQ*(s', $a_1$, h - 1), ..., SimQ*(s', $a_k$, h - 1))
Consider UniformBandit

Recursive UniformBandit: General H

and so on ..... Clearly replicating Sparse Sampling.
Recursive Bandit: General Horizon H

SelectRootAction(s,H)
   Return BanditAction(SimQ∗(s, a₁, H), ..., SimQ∗(s, aₖ, H))

SimQ∗(s,a,h)
   r = R(s,a)
   If h=1 then Return r
   s’ = T(s,a)
   Return r + β BanditValue(SimQ∗(s’, a₁, h – 1), ..., SimQ∗(s’, aₖ, h – 1))

• When bandit is UniformBandit same as Sparse Sampling
• Can plug in more advanced bandit algorithms for possible improvement!
Uniform vs. Non-Uniform Bandits

- Sparse sampling wastes time on bad parts of tree
  - Devotes equal resources to each state encountered in the tree
  - Would like to focus on most promising parts of tree
- But how to control exploration of new parts of tree vs. exploiting promising parts?
- Use non-uniform bandits
Non-Uniform Recursive Bandits

UCB-Based Sparse Sampling

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

Recursive UCB: General H

UCB in Recursive Bandit
Non-UniformRecursive Bandits

• UCB-Based Sparse Sampling
  ▲ Is UCB the right choice?
  ▲ We don’t really care about cumulative regret.
  ▲ My Guess: part of the reason UCB was tried was for purposes of leveraging its analysis

• \( \epsilon \) — Greedy Sparse Sampling
  ▲ Use \( \epsilon \) — Greedy as the bandit algorithm
  ▲ I haven’t seen this in the literature
  ▲ Might be better in practice since it is more geared to simple regret
  ▲ This would raise issues in the analysis (beyond the scope of this class).
Non-Uniform Recursive Bandits

- **Good News:** we might expect to improve over pure Sparse Sampling by changing the bandit algorithm

- **Bad News:** this recursive bandit approach has poor “anytime behavior”, which is often important in practice

- **Anytime Behavior:** good anytime behavior roughly means that an algorithm should be able to use small amounts of additional time to get small improvements

  ▲ What about these recursive bandits?
Recursive UCB: General H

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

• Information at the root only increases after each of the expensive root arm pulls
  ▶ Much time passes between these pulls

• Thus, small amounts of additional time do not result in any additional information at root!
  ▶ Thus, poor anytime behavior
  ▶ Running for 10sec could essentially the same as running for 10min (for large enough H)

• Can we improve the anytime behavior?
Monte-Carlo Planning Outline

- Single State Case (multi-armed bandits)
  - A basic tool for other algorithms

- Monte-Carlo Policy Improvement
  - Policy rollout
  - Policy Switching

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

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

- Famous for yielding a major advance in computer Go

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

---

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

- Builds a sparse look-ahead tree rooted at current state by repeated Monte-Carlo simulation of a “rollout policy”
- During construction each tree node $s$ stores:
  - state-visitiation count $n(s)$
  - action counts $n(s,a)$
  - action values $Q(s,a)$

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

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

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

• **Key Idea:** the tree policy can use statistics collected from previous trajectories to intelligently expand tree in most promising direction
  ▶ Rather than uniformly explore actions at each node
At a leaf node tree policy selects a random action then executes default

Iteration 1

Current World State

Initially tree is single leaf

Default Policy

new tree node

Terminal
(reward = 1)

Assume all non-zero reward occurs at terminal nodes.
Must select each action at a node at least once

Iteration 2

Current World State

new tree node

Default Policy

Terminal
(reward = 0)
Must select each action at a node at least once

Iteration 3

Current World State

1/2

1

0
When all node actions tried once, select action according to tree policy

Iteration 3

Current World State

1/2

1 0

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

Current World State

Tree Policy

Iteration 3

Default Policy

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

Iteration 4

Current World State

Tree Policy

1/2

1/3

0

0
When all node actions tried once, select action according to tree policy

Iteration 4

Current World State

1/3

1/2

0

0

1
When all node actions tried once, select action according to tree policy

Current World State

Tree Policy

What is an appropriate tree policy? Default policy?
UCT Algorithm [Kocsis & Szepesvari, 2006]

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

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

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

Theoretical constant that is empirically selected in practice (theoretical results based on $c$ equal to horizon $H$)
When all state actions tried once, select action according to tree policy

$$\pi_{UCT}(s) = \arg \max_a Q(s, a) + c \sqrt{\frac{\ln n(s)}{n(s, a)}}$$
When all node actions tried once, select action according to tree policy

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

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

\[ \pi_{UCT}(s) = \arg \max_a Q(s, a) + c \sqrt{\ln n(s) / n(s, a)} \]
When all node actions tried once, select action according to tree policy

\[ \pi_{UCT}(s) = \arg \max_a Q(s, a) + c \sqrt{\frac{\ln n(s)}{n(s, a)}} \]
When all node actions tried once, select action according to tree policy

\[ \pi_{UCT}(s) = \arg \max_a Q(s, a) + c \sqrt{\frac{\ln n(s)}{n(s, a)}} \]
When all node actions tried once, select action according to tree policy

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

• To select an action at a state s
  ▲ Build a tree using N iterations of monte-carlo tree search
    ■ Default policy is uniform random
    ■ Tree policy is based on UCB rule
  ▲ Select action that maximizes Q(s,a)
    (note that this final action selection does not take the exploration term into account, just the Q-value estimate)

• The more simulations the more accurate
Computer Go

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

9x9 (smallest board)

19x19 (largest board)
A Brief History of Computer Go

- **2005**: Computer Go is impossible!
- **2006**: UCT invented and applied to 9x9 Go \( (\text{Kocsis, Szepesvari; Gelly et al.}) \)
- **2007**: Human master level achieved at 9x9 Go \( (\text{Gelly, Silver; Coulom}) \)
- **2008**: Human grandmaster level achieved at 9x9 Go \( (\text{Teytaud et al.}) \)

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

- Klondike Solitaire (wins 40% of games)
- General Game Playing Competition
- Real-Time Strategy Games
- Combinatorial Optimization

- List is growing
- Usually extend UCT in some ways
Some Improvements

- Use domain knowledge to handcraft a more intelligent default policy than random
  - E.g. don’t choose obviously stupid actions
  - In Go a fast hand-coded default policy is used

- Learn a heuristic function to evaluate positions
  - Use the heuristic function to initialize leaf nodes (otherwise initialized to zero)
Other bandits?

• UCT was partly motivated by the question of how to use a UCB-like rule for tree search
  ▲ It is questionable whether UCB is the best choice for a tree policy

• Root Node:
  ▲ We only care about selecting the best action.
  ▲ Suggests trying $\epsilon$-Greedy at root

• Non-Root Nodes:
  ▲ The cumulative reward at these nodes is used as the value estimate by parent nodes
  ▲ Suggests we would like a small cumulative regret
  ▲ Suggests UCB might be more appropriate
Recent work has considered such a UCT variant

- Use 0.5-Greedy as tree policy at root and UCB at non-root nodes
- They also consider some other alternatives at the root that are more tuned to simple regret
- The results in that paper show that this simple change can improve performance significantly
  - The generality of this result remains to be seen
UCB at root

0.5-Greedy at root

Results in the Sailing Domain

Summary

• When you have a tough planning problem and a simulator
  ▲ Try Monte-Carlo planning

• Basic principles derive from the multi-arm bandit

• Policy rollout and switching are great way to exploit existing policies and make them better

• If a good heuristic exists, then shallow sparse sampling can give good results

• UCT is often quite effective especially when combined with domain knowledge