Monte-Carlo Planning II

Alan Fern
Outline

• Preliminaries: Markov Decision Processes
• What is Monte-Carlo Planning?
• Uniform Monte-Carlo
  ▶ Single State Case (UniformBandit)
  ▶ Policy rollout
  ▶ Approximate Policy Iteration
  ▶ Sparse Sampling
• Adaptive Monte-Carlo
  ▶ Single State Case (UCB Bandit)
  ▶ UCT Monte-Carlo Tree Search
Sparse Sampling

• Rollout does not guarantee optimality or near optimality
  ▲ Neither does approximate policy iteration in general

• 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?

• **Sparse Sampling** is one such algorithm
  ▲ Strong theoretical guarantees of near optimality
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

• $s =$ current state

• Repeat
  ▲ $T =$ BuildLookAheadTree$(s)$ ;; sparse sampling or UCT ;; tree provides Q-value estimates for root action
  ▲ $a =$ BestRootAction$(T)$ ;; action with best Q-value
  ▲ Execute action $a$ in environment
  ▲ $s$ is the resulting state
Sparse Sampling

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

• $h$-horizon optimal $Q$-function
  ▲ Value of taking $a$ in $s$ and following for $\pi^*$ 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

• Again focus on finite-horizons

\[ V^*(s,h) = \max_{a_1} Q(s,a_1,h) \]

\[ = \max_{a_1} E[R(s,a_1) + \beta V^*(T(s,a_1),h-1)] \]

\[ = \max_{a_1} E[R(s,a_1) + \beta \max_{a_2} E[R(T(s,a_1),a_2) + Q^*(T(s,a_1),a_2,h-1)]] \]

\[ \uparrow Q^*(s,a,h) = E[R(s,a) + \beta V^*(T(s,a),h-1)] \]

• Key identity (Bellman’s equations):

\[ \uparrow V^*(s,h) = \max_{a} E[R(s,a) + \beta \max_{a} Q^*(T(s,a),h-1)] \]
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
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 $V^*(s,h)$
  
  $V^*(s,h) = \max_{a_1} Q(s,a_1,h)$
  
  $= \max_{a_1} E[R(s,a_1) + \beta V^*(T(s,a_1),h-1)]$
  
  $= \max_{a_1} E[R(s,a_1) + \beta \max_{a_2} E[R(T(s,a_1),a_2)+Q^*(T(s,a_1),a_2,h-1)]]$
  
  $= \ldots$

• Can view this expansion as an expectimax tree
  - Each expectation is really a weighted sum over states
Exact Expectimax Tree for $V^*(s,H)$

Alternate max & expectation

Compute root $V^*$ and $Q^*$ via recursive procedure
Depends on size of the state-space. Bad!
Sparse Sampling Tree

Replace expectation with average over $w$ samples

$w$ will typically be much smaller than $n$. 

$V^*(s, H)$

$Q^*(s, a, H)$

Horizon $H$

$(kw)^H$ leaves

Sampling width $w$
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 $a^*$

**SparseSampleTree**($s,h,w$)

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 $r_i$

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

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

$Q^*(s,a,h) = Q^*(s,a,h) / w$ ;; 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 $[V^*(s,h), a^*]$
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 policy arbitrarily close to optimal
  ▲ Values are independent of state-space size!
  ▲ First near-optimal general MDP planning algorithm whose runtime didn’t depend on size of state-space

• **Bad news:** the theoretical values are typically still intractably large---also exponential in $H$
  ▲ Exponential in $H$ is the best we can do in general
  ▲ **In practice:** use small $H$ and use heuristic at leaves
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
Bandit View of Expectimax Tree

Alternate max and expectation

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 Assuming we Know $V^*$

$\text{Sim}Q^*(s,a_i,h) = R(s,a_i) + \beta V^*(T(s,a_i),h-1)$

$\text{Sim}Q^*(s,a_1,h)$  $\text{Sim}Q^*(s,a_2,h)$  $\text{Sim}Q^*(s,a_k,h)$

$\text{Sim}Q^*(s,a,h)$

$s' = T(s,a)$  
$r = R(s,a)$  
Return $r + \beta V^*(s',h-1)$

- Expected value of $\text{Sim}Q^*(s,a,h)$ is $Q^*(s,a,h)$
  - Use UniformBandit to select approximately optimal action
But we don’t know $V^*$

- To compute $\operatorname{SimQ}^*(s,a,h)$ need $V^*(s',h-1)$ for any $s'$

- Use recursive identity (Bellman’s equation):
  \[ V^*(s,h-1) = \max_a Q^*(s,a,h-1) \]

- **Idea:** Can recursively estimate $V^*(s,h-1)$ by running h-1 horizon bandit based on $\operatorname{SimQ}^*$
  \[ \text{Bandit returns estimated value of best action rather than just returning best action} \]

- **Base Case:** $V^*(s,0) = 0$, for all $s$
Recursive UniformBandit

\[ \text{SimQ}(s, a_i, h) \]

Recursively generate samples of
\[ R(s, a_i) + \beta V^*(T(s, a_i), h-1) \]

UniformBandit will generate \( w \) sample next states from each action at each state
Recursive UniformBandit

SimQ(s,a_i,h)
Recursively generate samples of
R(s, a_i) + β V*(T(s, a_i),h-1)

Returns V*(s_{11},h-1) estimate

Returns V*(s_{12},h-1) estimate
When bandit is UniformBandit same as sparse sampling

Each state generates $kw$ new states ($w$ states for each of $k$ bandits)

Total # of states in tree $(kw)^h$

Can plug in more advanced bandit algorithms for possible improvement!
Uniform vs. Adaptive 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?
- Need adaptive bandit algorithm that explores more effectively
Outline

• Preliminaries: Markov Decision Processes
• What is Monte-Carlo Planning?
• Uniform Monte-Carlo
  ▶ Single State Case (UniformBandit)
  ▶ Policy rollout
  ▶ Sparse Sampling
• Adaptive Monte-Carlo
  ▶ Single State Case (UCB Bandit)
  ▶ UCT Monte-Carlo Tree Search
Regret Minimization Bandit Objective

- **Problem:** find arm-pulling strategy such that the expected total reward at time $n$ is close to the best possible (one pull per time step)
  - Optimal (in expectation) is to pull optimal arm $n$ times
  - UniformBandit is poor choice --- waste time on bad arms
  - Must balance *exploring* machines to find good payoffs and *exploiting* current knowledge
UCB Adaptive Bandit Algorithm
[Auer, Cesa-Bianchi, & Fischer, 2002]

- \( Q(a) \): average payoff for action \( a \) (in our single state \( s \)) based on current experience
- \( n(a) \): number of pulls of arm \( a \)
- Action choice by UCB after \( n \) pulls:

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

- **Theorem:** The expected regret (i.e. sub-optimality) after \( n \) arm pulls compared to optimal behavior is bounded by \( O(\log n) \)
- No algorithm can achieve a better loss rate

Assumes payoffs in \([0,1]\)
**UCB Algorithm** [Auer, Cesa-Bianchi, & Fischer, 2002]

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

**Value Term:**
- favors actions that looked good historically

**Exploration Term:**
- actions get an exploration bonus that grows with \( \ln(n) \)

Expected number of pulls of sub-optimal arm \( a \) is bounded by:

\[ \frac{8}{\Delta_a^2} \ln n \]

where \( \Delta_a \) is regret of arm \( a \) (i.e. the amount of sub-optimality)

Doesn’t waste much time on sub-optimal arms, unlike uniform!
UCB for Multi-State MDPs

1. UCB-Based Policy Rollout:
   - Use UCB to select actions instead of uniform
   - Likely to use samples more efficiently by concentrating on promising actions
   - Unclear if this could be shown to have PAC properties

2. UCB-Based Sparse Sampling
   - Use UCB to make sampling decisions at internal tree nodes
   - There is an analysis of this algorithm’s bias

UCB-based Sparse Sampling [Chang et. al. 2005]

- Use UCB instead of Uniform to direct sampling at each state
- Non-uniform allocation

\[ \text{SimQ}^*(s_{11}, a_1, h-1) \quad \text{SimQ}^*(s_{11}, a_k, h-1) \]

- But each \( q_{ij} \) sample requires waiting for an entire recursive \( h-1 \) level tree search
- Better but still very expensive!
Outline

• Preliminaries: Markov Decision Processes

• What is Monte-Carlo Planning?

• Uniform Monte-Carlo
  ▲ Single State Case (UniformBandit)
  ▲ Policy rollout
  ▲ Sparse Sampling

• Adaptive Monte-Carlo
  ▲ Single State Case (UCB Bandit)
  ▲ UCT Monte-Carlo Tree Search
**UCT Algorithm**  [Kocsis & Szepesvari, 2006]

- UCT is an instance of **Monte-Carlo Tree Search**
  - Applies principle of UCB
  - 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 fully understood
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 stores:
  ▲ state-visitation 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

What is the rollout policy?
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 policy.

**Current World State**

- Initially tree is single leaf

**Default Policy**

- Terminal node (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

Default Policy

Terminal
(reward = 0)

new tree node
Iteration 3

Current World State

Must select each action at a node at least once
When all node actions tried once, select action according to tree policy

Iteration 3

Current World State

Tree Policy

1/2

1

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

Iteration 3

Current World State

1/2

Tree Policy

Default Policy

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

Iteration 4

Current World State

Tree Policy
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

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 must be selected empirically in practice
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

\[ \pi_{UCT}(s) = \arg \max_a Q(s, a) + c \sqrt{\frac{\ln n(s)}{n(s, a)}} \]
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 *(Kocsis, Szepesvari; Gelly et al.)*
- **2007**: Human master level achieved at 9x9 Go *(Gelly, Silver; Coulom)*
- **2008**: Human grandmaster level achieved at 9x9 Go *(Teytaud et al.)*

Computer GO Server rating over this period:
1800 ELO $\rightarrow$ 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 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)
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 is a great way to exploit existing policies and make them better

- If a good heuristic exists, then shallow sparse sampling can give good gains

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