Monte Carlo Tree Search

(Slides by Alan Fern, Subbarao Kambhampati, Lisa Torrey, Dan Weld)
Learning/Planning/Acting

Planning
Monte-Carlo Planning
Reinforcement Learning
Motivation

• Domain experts devise the simulator but don’t understand AI languages
• Probability distributions not easily expressible in AI languages
• Successor functions too large to be represented declaratively
• Domain models hidden from control person
Monte-Carlo Planning

- Often a simulator of a planning domain is available or can be learned from data
  - Even when domain can’t be expressed via MDP language

Klondike Solitaire

Fire & Emergency Response
Example Domains with Simulators

- Traffic simulators
- Robotics simulators
- Military campaign simulators
- Computer network simulators
- Emergency planning simulators
  - large-scale disaster and municipal
- Sports domains (Madden Football)
- Board games / Video games
  - Go / RTS

In many cases Monte-Carlo techniques yield state-of-the-art performance. Even in domains where model-based planner is applicable.
Slot Machines as MDP?
Outline

• Uniform Sampling
  – PAC Bound for Single State MDPs
  – Policy Rollouts for full MDPs

• Adaptive Sampling
  – UCB for Single State MDPs
  – UCT for full MDPs
Single State Monte-Carlo Planning

- Suppose MDP has a single state and k actions
  - Figure out which action has best expected reward
  - Can sample rewards of actions using calls to simulator
  - Sampling a is like pulling slot machine arm with random payoff function $R(s,a)$

Multi-Armed Bandit Problem
PAC Bandit Objective

Probably Approximately Correct (PAC)

- Select an arm that probably (w/ high probability, 1-δ) has approximately (i.e., within ε) the best expected reward
- Use as few simulator calls (or pulls) as possible

Multi-Armed Bandit Problem
UniformBandit Algorithm
NaiveBandit from [Even-Dar et. al., 2002]

1. Pull each arm $w$ times (uniform pulling).
2. Return arm with best average reward.

How large must $w$ be to provide a PAC guarantee?
Aside: Additive Chernoff Bound

• Let $R$ be a random variable with maximum absolute value $Z$. An let $r_i$ (for $i=1, \ldots, w$) be i.i.d. samples of $R$

• The Chernoff bound gives a bound on the probability that the average of the $r_i$ are far from $E[R]$

Chernoff Bound

$$\Pr\left( \left| E[R] - \frac{1}{w} \sum_{i=1}^{w} r_i \right| \geq \varepsilon \right) \leq \exp\left( - \left( \frac{\varepsilon}{Z} \right)^2 w \right)$$

Equivalently:

With probability at least $1 - \delta$ we have that,

$$\left| E[R] - \frac{1}{w} \sum_{i=1}^{w} r_i \right| \leq Z \sqrt{\frac{1}{w} \ln \frac{1}{\delta}}$$
UniformBandit PAC Bound

With a bit of algebra and Chernoff bound we get:

If $w \geq \left( \frac{R_{\max}}{\epsilon} \right)^2 \ln \frac{k}{\delta}$ for all arms simultaneously

$$\left| E[R(s, a_i)] - \frac{1}{w} \sum_{j=1}^{w} r_{ij} \right| \leq \epsilon$$

with probability at least $1 - \delta$

• That is, estimates of all actions are $\epsilon$-accurate with probability at least $1 - \delta$
• Thus selecting estimate with highest value is approximately optimal with high probability, or PAC
# Simulator Calls for UniformBandit

- **Total simulator calls for PAC:**
  \[ k \cdot w = O\left(\frac{k}{\epsilon^2 \ln \frac{k}{\delta}}\right) \]

- Can get rid of \( \ln(k) \) term with more complex algorithm [Even-Dar et. al., 2002].
Outline

• Uniform Sampling
  – PAC Bound for Single State MDPs
  – Policy Rollouts for full MDPs

• Adaptive Sampling
  – UCB for Single State MDPs
  – UCT for full MDPs
Now consider a multi-state MDP.
Suppose we have a simulator and a non-optimal policy
  – E.g. policy could be a standard heuristic or based on intuition
Can we somehow compute an improved policy?
Policy Rollout Algorithm

1. For each $a_i$, run $\text{SimQ}(s,a_i,\pi,h)$ $w$ times
2. Return action with best average of $\text{SimQ}$ results

SimQ($s,a_i,\pi,h$) trajectories
Each simulates taking action $a_i$ then following $\pi$ for $h-1$ steps.

Samples of SimQ($s,a_i,\pi,h$)

$q_{11} \ q_{12} \ldots \ q_{1w}$
$q_{21} \ q_{22} \ldots \ q_{2w}$
$q_{k1} \ q_{k2} \ldots \ q_{kw}$
Policy Rollout: # of Simulator Calls

- For each action, \( w \) calls to SimQ, each using \( h \) sim calls
- Total of \( khw \) calls to the simulator

SimQ\((s,a_i,\pi,h)\) trajectories
Each simulates taking action \( a_i \) then following \( \pi \) for \( h-1 \) steps.
Multi-Stage Rollout

Each step requires $khw$ simulator calls

Trajectories of $\text{SimQ}(s,a_i,\text{Rollout}(\pi),h)$

- Two stage: compute \textit{rollout policy of rollout policy} of $\pi$
- Requires $(khw)^2$ calls to the simulator for 2 stages
- In general exponential in the number of stages
Rollout Summary

• We often are able to write simple, mediocre policies
  ▲ Network routing policy
  ▲ Compiler instruction scheduling
  ▲ Policy for card game of Hearts
  ▲ Policy for game of Backgammon
  ▲ Solitaire playing policy
  ▲ Game of GO
  ▲ Combinatorial optimization

• Policy rollout is a general and easy way to improve upon such policies
• Often observe substantial improvement!
### Example: Rollout for Thoughtful Solitaire

[Yan et al. NIPS’04]

<table>
<thead>
<tr>
<th>Player</th>
<th>Success Rate</th>
<th>Time/Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human Expert</td>
<td>36.6%</td>
<td>20 min</td>
</tr>
<tr>
<td>(naïve) Base Policy</td>
<td>13.05%</td>
<td>0.021 sec</td>
</tr>
<tr>
<td>1 rollout</td>
<td>31.20%</td>
<td>0.67 sec</td>
</tr>
<tr>
<td>2 rollout</td>
<td>47.6%</td>
<td>7.13 sec</td>
</tr>
<tr>
<td>3 rollout</td>
<td>56.83%</td>
<td>1.5 min</td>
</tr>
<tr>
<td>4 rollout</td>
<td>60.51%</td>
<td>18 min</td>
</tr>
<tr>
<td>5 rollout</td>
<td>70.20%</td>
<td>1 hour 45 min</td>
</tr>
</tbody>
</table>
Example: Rollout for Thoughtful Solitaire
[Yan et al. NIPS’04]

<table>
<thead>
<tr>
<th>Player</th>
<th>Success Rate</th>
<th>Time/Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human Expert</td>
<td>36.6%</td>
<td>20 min</td>
</tr>
<tr>
<td>(naïve) Base Policy</td>
<td>13.05%</td>
<td>0.021 sec</td>
</tr>
<tr>
<td>1 rollout</td>
<td>31.20%</td>
<td>0.67 sec</td>
</tr>
<tr>
<td>2 rollout</td>
<td>47.6%</td>
<td>7.13 sec</td>
</tr>
<tr>
<td>3 rollout</td>
<td>56.83%</td>
<td>1.5 min</td>
</tr>
<tr>
<td>4 rollout</td>
<td>60.51%</td>
<td>18 min</td>
</tr>
<tr>
<td>5 rollout</td>
<td>70.20%</td>
<td>1 hour 45 min</td>
</tr>
</tbody>
</table>
## Example: Rollout for Thoughtful Solitaire

[Yan et al. NIPS’04]

<table>
<thead>
<tr>
<th>Player</th>
<th>Success Rate</th>
<th>Time/Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human Expert</td>
<td>36.6%</td>
<td>20 min</td>
</tr>
<tr>
<td>(naïve) Base Policy</td>
<td>13.05%</td>
<td>0.021 sec</td>
</tr>
<tr>
<td>1 rollout</td>
<td>31.20%</td>
<td>0.67 sec</td>
</tr>
<tr>
<td>2 rollout</td>
<td>47.6%</td>
<td>7.13 sec</td>
</tr>
</tbody>
</table>

The table above illustrates the success rate and time taken per game for different players in Thoughtful Solitaire. The information is derived from Yan et al.’s work presented at NIPS’04.
Example: Rollout for Thoughtful Solitaire

[Yan et al. NIPS’04]

<table>
<thead>
<tr>
<th>Player</th>
<th>Success Rate</th>
<th>Time/Game</th>
</tr>
</thead>
<tbody>
<tr>
<td>Human Expert</td>
<td>36.6%</td>
<td>20 min</td>
</tr>
<tr>
<td>(naïve) Base Policy</td>
<td>13.05%</td>
<td>0.021 sec</td>
</tr>
<tr>
<td>1 rollout</td>
<td>31.20%</td>
<td>0.67 sec</td>
</tr>
<tr>
<td>2 rollout</td>
<td>47.6%</td>
<td>7.13 sec</td>
</tr>
<tr>
<td>3 rollout</td>
<td>56.83%</td>
<td>1.5 min</td>
</tr>
<tr>
<td>4 rollout</td>
<td>60.51%</td>
<td>18 min</td>
</tr>
<tr>
<td>5 rollout</td>
<td>70.20%</td>
<td>1 hour 45 min</td>
</tr>
</tbody>
</table>

Deeper rollout can pay off, but is expensive
Outline

• Uniform Sampling
  – PAC Bound for Single State MDPs
  – Policy Rollouts for full MDPs

• Adaptive Sampling
  – UCB for Single State MDPs
  – UCT for full MDPs
Non-Adaptive Monte-Carlo

- What is an issue with Uniform sampling?
  - time wasted equally on all actions!
  - no early learning about suboptimal actions

- Policy rollouts
  - 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??
Regret Minimization Bandit Objective

- **Problem:** find arm-pulling strategy such that the expected total reward at time n is close to the best possible (i.e. pulling the best arm always)

- 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 \(a\) based on current experience
- **n(a)**: number of pulls of arm \(a\)
- Action choice by UCB after \(n\) pulls:

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

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

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

Assumes payoffs in \([0,1]\)

Doesn’t waste much time on sub-optimal arms unlike uniform!
**UCB Algorithm** [Auer, Cesa-Bianchi, & Fischer, 2002]

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

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

\[ \frac{8}{\Delta^2_a} \ln n \]

where \( \Delta_a \) is regret of arm \( a \)

- Hence, the expected regret after \( n \) arm pulls compared to optimal behavior is bounded by \( O(\log n) \)
- No algorithm can achieve a better loss rate
Outline

• Uniform Sampling
  – PAC Bound for Single State MDPs
  – Policy Rollouts for full MDPs

• Adaptive Sampling
  – UCB for Single State MDPs
  – UCT for full MDPs
UCB Based Policy Rollout

• Allocate samples non-uniformly
  – based on UCB action selection
  – More sample efficient than uniform policy rollout
    – Still suboptimal.
• Instance of Monte-Carlo Tree Search
  – Applies principle of UCB
  – Some nice theoretical properties
  – Better than policy rollouts – asymptotically optimal
  – Major advance in computer Go

• Monte-Carlo Tree Search
  – Repeated Monte Carlo simulation of a rollout policy
  – Each rollout adds one or more nodes to search tree

• Rollout policy depends on nodes already in tree
At a leaf node perform a random rollout

Current World State

Initially tree is single leaf
At a leaf node perform a random rollout

Current World State

Initially tree is single leaf

Rollout Policy

Terminal (reward = 1)
At a leaf node perform a random rollout

Current World State

Initially tree is single leaf

Rollout Policy

Terminal (reward = 1)
Must select each action at a node at least once

Current World State

1

Rollout Policy

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

Current World State

1/2

1

0

1

0

1

0

Value set to average (or max)
When all node actions tried once, select action according to tree policy
When all node actions tried once, select action according to tree policy.
When all node actions tried once, select action according to tree policy

What is an appropriate tree policy? Rollout policy?
• Basic UCT uses random rollout 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

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

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

9x9 (smallest board)  19x19 (largest board)

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

Human champions refuse to compete against computers, because software is too bad.

<table>
<thead>
<tr>
<th></th>
<th>Chess</th>
<th>Go</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size of board</td>
<td>8 x 8</td>
<td>19 x 19</td>
</tr>
<tr>
<td>Average no. of moves per game</td>
<td>100</td>
<td>300</td>
</tr>
<tr>
<td>Avg branching factor per turn</td>
<td>35</td>
<td>235</td>
</tr>
<tr>
<td>Additional complexity</td>
<td>Players can pass</td>
<td>Players can pass</td>
</tr>
</tbody>
</table>
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.*)

*ELO rating 1800 → 2600*
UCT $\rightarrow$ World Class 9x9 Go Player

- UCT + Value Function Approximation + RAVE

- Value function approximation
  - Provides an initial estimate of $V(s)$ via a heuristic

- RAVE
  - generalizes the tree policy to new states
Generalization in Learning

- In realistic situations, we cannot possibly learn about every single state!
  - Too many states to visit them all in training
  - Too many states to even hold the q-tables in memory

- Instead, we want to generalize:
  - Learn about some small number of training states from experience
  - Generalize that experience to new, similar states
  - This is a fundamental idea in machine learning, and we’ll see it over and over again
Value Function Approximation

Assume
\[ V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s) \]

GO: \[ Q(s, a) = \text{sigmoid}(w_1 f_1(s, a) + w_2 f_2(s, a) + \ldots + w_n f_n(s, a)) \]

How to learn the weights?

sample: estimated value of \( V(s) \) after taking action
\[ V(s) \leftarrow V(s) + \alpha (\text{sample} - V(s)) \]
\[ V(s) \leftarrow V(s) + \alpha (r + \gamma V(s') - V(s)) \]
TD learning with Value Approx

\[ V(s) \leftarrow V(s) + \alpha (r + \gamma V(s') - V(s)) \]

\[ w_i \leftarrow w_i + \alpha (r + \gamma V(s') - V(s)).f_i(s) \]

(Widrow-Hoff rule or Delta rule)

GO: \[ w_i \leftarrow w_i + \alpha (Q(s',a') - Q(s,a)).f_i(s,a) \]
Rapid Action Value Estimation (RAVE)

- Goal: information sharing within tree policy
- Typically

\[
Q(s,a) = \text{AVG}[q^1(s,a), q^2(s,a), ...]
\]

RAVE value

\[
Q(s,a) = \text{AVG}[q^1(s,a), q^2(s,a), ..., q^1(s'a), q^2(s',a) ...]
\]

(for all \(s'\) in the subtree of \(s\))

RAVE: quick information transfer but error prone
Master level 9x9 GO

• **UCT + RAVE**
  – Rely on RAVE initially and gradually shift to real value
  – Using linear combination with decaying RAVE weight

• **UCT + RAVE + FN APPROX**
  – Initialize RAVE value as the function approx. value
  – Initialize n(s,a) based on the confidence of fn approx.

• Observation: UCT depends heavily on quality of function approximation.

• 3-dan (master) level performance in 9x9 GO.
  – First program to beat a human in 9x9 GO
  – Best software in 19x19 GO ~2008.
Other Successes

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

- Probabilistic Planning (MDPs)

- Usually extend UCT in some ways
Improvements/Issues

• Use domain knowledge to improve the base policies
  – E.g.: don’t choose obvious stupid actions
  – better policy does not imply better UCT performance

• Learn a heuristic function to evaluate positions
  – Use heuristic to initialize leaves

• *Interesting question: UCT versus minimax*
Summary

• Multi-armed Bandits
  – Principles of both RL and Monte-Carlo

• Monte-Carlo Planning
  – Exploration/Exploitation tradeoff
  – Uniform/Adaptive Sampling

• Value Function Approximation

• RAVE heuristic in Go.