Monte-Carlo Planning

Alan Fern
Large Worlds

• We have considered basic model-based planning algorithms

• **Model-based planning**: assumes MDP model is available
  ^ Methods we learned so far are at least poly-time in the number of states and actions
  ^ Difficult to apply to large state and action spaces (though this is a rich research area)

• We will consider various methods for overcoming this issue
Approaches for Large Worlds

**Planning with compact MDP representations**

1. Define a language for compactly describing an MDP
   - MDP is exponentially larger than description
   - E.g. via Dynamic Bayesian Networks
2. Design a planning algorithm that directly works with that language

- Scalability is still an issue
- Can be difficult to encode the problem you care about in a given language
- Study in last part of course
Approaches for Large Worlds

- **Reinforcement learning w/ function approx.**
  1. Have a learning agent directly interact with environment
  2. Learn a compact description of policy or value function

- Often works quite well for large problems
- Doesn’t fully exploit a simulator of the environment when available
- We will study reinforcement learning later in the course
Approaches for Large Worlds: Monte-Carlo Planning

- Often a simulator of a planning domain is available or can be learned from data.
Large Worlds: Monte-Carlo Approach

• Often a **simulator** of a planning domain is available or can be learned from data

• **Monte-Carlo Planning**: compute a good policy for an MDP by interacting with an MDP simulator
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 planners are applicable.
MDP: Simulation-Based Representation

- A simulation-based representation gives: S, A, R, T, I:
  - finite state set S (|S|=n and is generally very large)
  - finite action set A (|A|=m and will assume is of reasonable size)
  - Stochastic, real-valued, bounded reward function \( R(s,a) = r \)
    - Stochastically returns a reward \( r \) given input \( s \) and \( a \)
  - Stochastic transition function \( T(s,a) = s' \) (i.e. a simulator)
    - Stochastically returns a state \( s' \) given input \( s \) and \( a \)
    - Probability of returning \( s' \) is dictated by \( Pr(s' | s,a) \) of MDP
  - Stochastic initial state function I.
    - Stochastically returns a state according to an initial state distribution

These stochastic functions can be implemented in any language!
Outline

• Uniform Monte-Carlo
  ▶ Single State Case (Uniform Bandit)
  ▶ Policy rollout
  ▶ Approximate Policy Iteration
  ▶ Sparse Sampling

• Adaptive Monte-Carlo
  ▶ Single State Case (UCB Bandit)
  ▶ UCT Monte-Carlo Tree Search
Single State Monte-Carlo Planning

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

\[
\begin{align*}
\text{s} & \quad a_1 \quad a_2 \quad a_k \\
R(s, a_1) & \quad R(s, a_2) \quad \cdots \quad R(s, a_k)
\end{align*}
\]

Multi-Armed Bandit Problem
PAC Bandit Objective: Informal

- Probably Approximately Correct (PAC)
  - Select an arm that probably (w/ high probability) has approximately the best expected reward
  - Use as few simulator calls (or pulls) as possible to guarantee this

\[ R(s,a_1) \quad R(s,a_2) \quad \ldots \quad R(s,a_k) \]

Multi-Armed Bandit Problem
PAC Bandit Algorithms

• Let $k$ be the number of arms, $R_{\text{max}}$ be an upper bound on reward, and $R^* = \max_i E[R(s, a_i)]$ (i.e. $R^*$ is the best arm in expectation)

Definition (Efficient PAC Bandit Algorithm): An algorithm ALG is an efficient PAC bandit algorithm iff for any multi-armed bandit problem, for any $0<\delta<1$ and any $0<\varepsilon<1$, ALG pulls a number of arms that is polynomial in $1/\varepsilon$, $1/\delta$, $k$, and $R_{\text{max}}$ and returns an arm index $j$ such that with probability at least $1-\delta$

$$R^* - E[R(s, a_j)] \leq \varepsilon$$

• Such an algorithm is efficient in terms of # of arm pulls, and is probably (with probability $1-\delta$) approximately correct (picks an arm with expected reward within $\varepsilon$ of optimal).
UniformBandit Algorithm
NaiveBandit from [Even-Dar et. al., 2002]

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

Can we make this an efficient PAC bandit algorithm?
Aside: Additive Chernoff Bound

- Let $R$ be a random variable with maximum absolute value $Z$. An let $r_i, i=1,\ldots,w$ 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 $E[R]$

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

Equivalently:

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

  \[
  \left| E[R] - \frac{1}{w} \sum_{i=1}^{w} r_i \right| \leq Z \sqrt{\frac{1}{w} \ln \frac{1}{\delta}}
  \]
Aside: Coin Flip Example

• Suppose we have a coin with probability of heads equal to \( p \).

• Let \( X \) be a random variable where \( X=1 \) if the coin flip gives heads and zero otherwise. (so \( Z \) from bound is 1)

\[
E[X] = 1*p + 0*(1-p) = p
\]

• After flipping a coin \( w \) times we can estimate the heads prob. by average of \( x_i \).

• The Chernoff bound tells us that this estimate converges exponentially fast to the true mean (coin bias) \( p \).

\[
Pr \left( \left| p - \frac{1}{w} \sum_{i=1}^{w} x_i \right| \geq \varepsilon \right) \leq \exp \left( - \varepsilon^2 w \right)
\]
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?
Uniform Bandit PAC Bound

• For a single bandit arm the Chernoff bound says:

\[
\left| E[R(s, a_i)] - \frac{1}{w} \sum_{j=1}^{w} r_{ij} \right| \leq R_{\text{max}} \sqrt{\frac{1}{w} \ln \frac{1}{\delta'}}
\]

• Bounding the error by \( \varepsilon \) gives:

\[
R_{\text{max}} \sqrt{\frac{1}{w} \ln \frac{1}{\delta'}} \leq \varepsilon \quad \text{or equivalently} \quad w \geq \left( \frac{R_{\text{max}}}{\varepsilon} \right)^2 \ln \frac{1}{\delta'}
\]

• Thus, using this many samples for a single arm will guarantee an \( \varepsilon \)-accurate estimate with probability at least \( 1 - \delta' \)
Uniform Bandit PAC Bound

- So we see that with \( w \geq \left( \frac{R_{\text{max}}}{\varepsilon} \right)^2 \ln \frac{1}{\delta'} \) samples per arm, there is no more than a \( \delta' \) probability that an individual arm’s estimate will not be \( \varepsilon \)-accurate.
  - But we want to bound the probability of any arm being inaccurate.

The union bound says that for \( k \) events, the probability that at least one event occurs is bounded by the sum of individual probabilities

\[
\Pr(A_1 \text{ or } A_2 \text{ or } \ldots \text{ or } A_k) \leq \sum_{i=1}^{k} \Pr(A_k)
\]

- Using the above \# samples per arm and the union bound (with events being “arm \( i \) is not \( \varepsilon \)-accurate”) there is no more than \( k\delta' \) probability of any arm not being \( \varepsilon \)-accurate.

- Setting \( \delta' = \frac{\delta}{k} \) all arms are \( \varepsilon \)-accurate with prob. at least \( 1 - \delta \).
UniformBandit PAC Bound

Putting everything together we get:

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

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

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:

  - So we have an **efficient** PAC algorithm

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

• Uniform Monte-Carlo
  ▲ Single State Case (Uniform Bandit)
  ▲ Policy rollout
  ▲ Approximate Policy Iteration
  ▲ Sparse Sampling

• Adaptive Monte-Carlo
  ▲ Single State Case (UCB Bandit)
  ▲ UCT Monte-Carlo Tree Search
Policy Improvement via Monte-Carlo

- Now consider a very large 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?
Recall: Policy Improvement Theorem

\[ Q_\pi(s, a) = R(s) + \beta \sum_{s', T(s, a, s')} V_\pi(s') \]

- The Q-value function of a policy gives expected discounted future reward of starting in state \( s \), taking action \( a \), and then following policy \( \pi \) thereafter.

- **Define:** \( \pi'(s) = \arg \max_a Q_\pi(s, a) \)

- **Theorem [Howard, 1960]:** For any non-optimal policy \( \pi \) the policy \( \pi' \) a strict improvement over \( \pi \).

- Computing \( \pi' \) amounts to finding the action that maximizes the Q-function of \( \pi \).
  - Can we use the bandit idea to solve this?
Policy Improvement via Bandits

\begin{itemize}
  \item **Idea:** define a stochastic function \( \text{SimQ}(s,a,\pi) \) that we can implement and whose expected value is \( Q_{\pi}(s,a) \)
  \item Then use Bandit algorithm to PAC select an improved action
\end{itemize}

How to implement SimQ?
Q-value Estimation

• SimQ might be implemented by simulating the execution of action $a$ in state $s$ and then following $π$ thereafter
  - But for infinite horizon problems this would never finish
  - So we will approximate via finite horizon

• The $h$-horizon Q-function $Q_π(s,a,h)$ is defined as: expected total discounted reward of starting in state $s$, taking action $a$, and then following policy $π$ for $h$-1 steps

• The approximation error decreases exponentially fast in $h$ (you will show in homework)

$$|Q_π(s,a) - Q_π(s,a,h)| \leq β^h V_{max}$$

$$V_{max} = \frac{R_{max}}{1 - β}$$
Policy Improvement via Bandits

Refined Idea: define a stochastic function $\text{SimQ}(s, a, \pi, h)$ that we can implement, whose expected value is $Q_\pi(s, a, h)$

Use Bandit algorithm to PAC select an improved action

How to implement SimQ?
Policy Improvement via Bandits

\[
\text{SimQ}(s, a, \pi, h) = \\
\begin{align*}
    r &= R(s, a) \quad \text{simulate } a \text{ in } s \\
    s &= T(s, a) \\
    \text{for } i = 1 \text{ to } h-1 \\
    &\quad r = r + \beta^i R(s, \pi(s)) \quad \text{simulate } h-1 \text{ steps} \\
    &\quad s = T(s, \pi(s)) \quad \text{of policy} \\
\end{align*}
\]

Return \( r \)

- Simply simulate taking \( a \) in \( s \) and following policy for \( h-1 \) steps, returning discounted sum of rewards
- Expected value of SimQ\((s, a, \pi, h)\) is \( Q_\pi(s, a, h) \) which can be made arbitrarily close to \( Q_\pi(s, a) \) by increasing \( h \)
Policy Improvement via Bandits

\[
\text{SimQ}(s,a,\pi,h) = \\
\begin{align*}
  r &= R(s,a) \quad \text{simulate } a \text{ in } s \\
  s &= T(s,a) \\
  \text{for } i = 1 \text{ to } h-1 \\
  &\quad r = r + \beta^i R(s, \pi(s)) \quad \text{simulate } h-1 \text{ steps of policy} \\
  &\quad s = T(s, \pi(s)) \\
  \text{Return } r 
\end{align*}
\]

Trajectory under \( \pi \)

- Sum of rewards = SimQ(s,a_1,\pi,h)
- Sum of rewards = SimQ(s,a_2,\pi,h)
- \( \vdots \)
- Sum of rewards = SimQ(s,a_k,\pi,h)
Policy Rollout Algorithm

Rollout[π,h,w](s)
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

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

Samples of $\text{SimQ}(s,a_i,\pi,h)$
$q_{11} \quad q_{12} \ldots q_{1w} \quad q_{21} \quad q_{22} \ldots q_{2w} \quad q_{k1} \quad q_{k2} \ldots q_{kw}$
Policy Rollout: # of Simulator Calls

- For each action \( w \) calls to SimQ, each using \( h \) sim calls

- Total of \( kwh \) calls to the simulator
Policy Rollout Quality

- Let \( a^* \) be the action that maximizes the true Q-function \( Q_\pi(s,a) \).
- Let \( a' \) be the action returned by Rollout[\( \pi,h,w \)](s).
- Putting the PAC bandit result together with the finite horizon approximation we can derive the following:

\[
\text{If } w \geq \left( \frac{R_{\text{max}}}{\epsilon} \right)^2 \ln \frac{k}{\delta} \text{ then with probability at least } 1 - \delta
\]

\[
|Q_\pi(s,a^*) - Q_\pi(s,a')| \leq \epsilon + \beta^h V_{\text{max}}
\]

But does this guarantee that the value of Rollout[\( \pi,h,w \)](s) will be close to the value of \( \pi' \)?
Policy Rollout: Quality

• How good is Rollout[π, h, w] compared to π’?

• **Bad News.** In general for a fixed h and w there is always an MDP such that the quality of the rollout policy is arbitrarily worse than π’.

• The example MDP is somewhat involved, but shows that even small error in Q-value estimates can lead to large performance gaps compared to π’
  ▲ But this result is quite pathological
Policy Rollout: Quality

• How good is Rollout[π,h,w] compared to π’?

• **Good News.** If we make an assumption about the MDP, then it is possible to select h and w so that the rollout quality is close to π’.
  ▲ This is a bit involved.
  ▲ Assume a lower bound on the difference between the best Q-value and the second best Q-value

• **More Good News.** It is possible to select h and w so that Rollout[π,h,w] is (approximately) no worse than π for any MDP
  ▲ So at least rollout won’t hurt compared to the base policy
  ▲ At the same time it has the potential to significantly help
Multi-Stage Rollout

- A single call to Rollout[$\pi$, $h$, $w$](s) approximates one iteration of policy iteration starting at policy $\pi$
  - But only computes the action for state s rather than all states (as done by full policy iteration)

- We can use more computation time to approximate multiple iterations of policy iteration via nesting calls to Rollout

- Gives a way to use more time in order to improve performance
Multi-Stage Rollout

Each step requires khw simulator calls for Rollout policy

Trajectories of SimQ(s,a_i,Rollout[π,h,w],h)

• Two stage: compute rollout policy of “rollout policy of π”
• 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
  ▪ Policy for card game of Hearts
  ▪ Policy for game of Backgammon
  ▪ Solitaire playing policy

• Policy rollout is a general and easy way to improve upon such policies given a simulator

• Often observe substantial improvement, e.g.
  ▪ Compiler instruction scheduling
  ▪ Backgammon
  ▪ Network routing
  ▪ Combinatorial optimization
  ▪ Game of GO
  ▪ Solitaire
Example: Rollout for 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>

- Multiple levels of rollout can payoff but is expensive

Can we somehow get the benefit of multiple levels without the complexity?
Outline

• **Uniform Monte-Carlo**
  - Single State Case (Uniform Bandit)
  - Policy rollout
  - Approximate Policy Iteration
  - Sparse Sampling

• **Adaptive Monte-Carlo**
  - Single State Case (UCB Bandit)
  - UCT Monte-Carlo Tree Search
Approximate Policy Iteration: Main Idea

- Nested rollout is expensive because the “base policies” (i.e. nested rollouts themselves) are expensive

- Suppose that we could approximate a level-one rollout policy with a very fast function (e.g. O(1) time)

- Then we could approximate a level-two rollout policy while paying only the cost of level-one rollout

- Repeatedly applying this idea leads to approximate policy iteration
Return to Policy Iteration

Approximate policy iteration:

- Only computes values and improved action at some states.
- Uses those to infer a fast, compact policy over all states.
Approximate Policy Iteration

1. Generate trajectories of rollout policy (starting state of each trajectory is drawn from initial state distribution $I$)
2. “Learn a fast approximation” of rollout policy
3. Loop to step 1 using the learned policy as the base policy

What do we mean by generate trajectories?

Technically rollout only approximates $\pi'$. 

Sample $\pi'$ trajectories using **rollout**

Learn fast approximation of $\pi'$

Current Policy

$\pi'$ trajectories

$\pi$

$\pi'$
Generating Rollout Trajectories

Get trajectories of current rollout policy from an initial state

Random draw from $i$

Run policy rollout

$s \rightarrow a_1 \rightarrow a_2 \rightarrow \ldots \rightarrow a_k \rightarrow \ldots$

Run policy rollout
Generating Rollout Trajectories

Get trajectories of current rollout policy from an initial state

Multiple trajectories differ since initial state and transitions are stochastic
Generating Rollout Trajectories

Get trajectories of current rollout policy from an initial state

Results in a set of state-action pairs giving the action selected by “improved policy” in states that it visits.

\[ \{(s_1, a_1), (s_2, a_2), \ldots, (s_n, a_n)\} \]
Approximate Policy Iteration

1. Generate trajectories of rollout policy (starting state of each trajectory is drawn from initial state distribution $I$)
2. “Learn a fast approximation” of rollout policy
3. Loop to step 1 using the learned policy as the base policy

What do we mean by “learn an approximation”? 

to approximates $\pi'$. 

Sample $\pi'$ trajectories using **rollout** 

Learn fast approximation of $\pi'$ 

Current Policy 

$\pi'$ trajectories
Aside: Classifier Learning

- A **classifier** is a function that labels inputs with class labels.
- “Learning” classifiers from training data is a well studied problem (decision trees, support vector machines, neural networks, etc).

![Diagram of classifier learning process]

### Training Data

\[ \{(x_1, c_1), (x_2, c_2), \ldots, (x_n, c_n)\} \]

### Learning Algorithm

#### Example problem:

- \( x_i \) - image of a face
- \( c_i \in \{\text{male, female}\} \)

**Classifier**

\[ H: X \rightarrow C \]
Aside: Control Policies are Classifiers

A control policy maps states and goals to actions.

\[ \pi : \text{states} \rightarrow \text{actions} \]

Training Data

\[ \{(s_1, a_1), (s_2, a_2), \ldots, (s_n, a_n)\} \]

Learning Algorithm

Classifier/Policy

\[ \pi : \text{states} \rightarrow \text{actions} \]
Approximate Policy Iteration

- Sample \( \pi' \) trajectories using **rollout**
- Learn a classifier based on \( T \) to approximate \( \pi' \)
  
\[ T = \{(s_1,a_1), (s_2,a_2), \ldots, (s_n,a_n)\} \]

1. Generate trajectories of rollout policy
   Results in training set of state-action pairs along trajectories

2. Learn a classifier based on \( T \) to approximate rollout policy

3. Loop to step 1 using the learned policy as the base policy
Approximate Policy Iteration

Sample $\pi'$ trajectories using **rollout**

$\pi'$ training data

\{(s_1, a_1), (s_2, a_2), ..., (s_n, a_n)\}

Learn **classifier** to approximate $\pi'$

Current Policy

- The hope is that the learned classifier will capture the general structure of improved policy from examples
- Want classifier to quickly select correct actions in states outside of training data (classifier should generalize)
- Approach allows us to leverage large amounts of work in machine learning
API for Inverted Pendulum

Consider the problem of balancing a pole by applying either a positive or negative force to the cart.

The state space is described by the velocity of the cart and angle of the pendulum.

There is noise in the force that is applied, so problem is stochastic.
Experimental Results

A data set from an API iteration. + is positive action, x is negative (ignore the circles in the figure)
Experimental Results

Support vector machine used as classifier.
(take CS534 for details)
Maps any state to + or −

Learned classifier/policy after 2 iterations: (near optimal)
blue = positive, red = negative
API for Stacking Blocks

Consider the problem of forming a goal configuration of blocks/crates/etc. from a starting configuration using basic movements such as pickup, putdown, etc.

Also handle situations where actions fail and blocks fall.
The resulting policy is fast near optimal. These problems are very hard for more traditional planners.
Summary of API

• Approximate policy iteration is a practical way to select policies in large state spaces

• Relies on ability to learn good, compact approximations of improved policies (must be efficient to execute)

• Relies on the effectiveness of rollout for the problem

• There are only a few positive theoretical results
  ▲ convergence in the limit under strict assumptions
  ▲ PAC results for single iteration

• But often works well in practice
Another Useful Technique: Policy Switching

- Suppose you have a set of base policies \( \{\pi_1, \pi_2, \ldots, \pi_M\} \)

- Also suppose that the best policy to use can depend on the specific state of the system and we don’t know how to select.

- Policy switching is a simple way to select which policy to use at a given step via a simulator
Another Useful Technique: Policy Switching

- The stochastic function $\text{Sim}(s, \pi, h)$ simply estimates the $H$-horizon value of $\pi$ from state $s$.
- Implement by running multiple trials of $\pi$ starting from state $s$ and averaging the discounted rewards across trials.
- Use Bandit algorithm to PAC select best policy and then select action chosen by that policy.
Policy Switching

PolicySwitch[\{\pi_1, \pi_2, \ldots, \pi_M\}, h, w](s)

1. For each \pi_i run Sim(s, \pi_i, h) w times
2. Let \(i^*\) be index of policy with best average result
3. Return action \(\pi_{i^*}(s)\)

Sim(s, \pi_i, h) trajectories

Each simulates taking following \(\pi_i\) for \(h\) steps.
Policy Switching: # of Simulator Calls

- For each policy use $w$ calls to Sim, each using $h$ simulator calls
- Total of $Mhw$ calls to the simulator
- Does not depend on number of actions!
Policy Switching: Quality

- How good is PolicySwitch[\{\pi_1, \pi_2, \ldots, \pi_M\},h,w]?
- **Good News.** The value of this policy from any state is no worse than the value of the best policy in our set from that state.

- That is, if we let \( \pi_{PS} \) denote the ideal switching policy (always pick the best policy index), then from any state \( s \) and any horizon:
  \[
  V^{\pi_{ps}}(s,h) \geq \max_i V^{\pi_i}(s,h)
  \]
  For non-ideal case, were we pick nearly best policy (in a PAC sense) we add an error term to the bound.

- **Proof:** Use induction on \( h \). Base case of \( h=0 \) is trivial. Now for the inductive case ....
Policy Switching: Quality

\[ V^{\pi_{ps}}(s, h) = E\left[ R(s, a_{ps}) + \beta V^{\pi_{ps}}(T(s, a_{ps}), h-1) \right], \]

\[ a_{ps} = \pi_{ps}(s) = \pi_{i^*}(s), \]

\[ i^* \text{ is index of selected policy} \]
Policy Switching: Quality

\[ V^{\pi_{ps}} (s, h) = \mathbb{E} \left[ R(s, a_{ps}) + \beta V^{\pi_{ps}} (T(s, a_{ps}), h - 1) \right], \]

\[ a_{ps} = \pi_{ps} (s) = \pi_i^* (s), \]

\[ i^* \text{ is index of selected policy} \]

\[ \geq \mathbb{E} \left[ R(s, a_{ps}) + \beta \max_i V^{\pi_i} (T(s, a_{ps}), h - 1) \right], \]

by inductive hypothesis.
Policy Switching: Quality

\[ V^{\pi_{ps}}(s, h) = E \left[ R(s, a_{ps}) + \beta V^{\pi_{ps}}(T(s, a_{ps}), h-1) \right], \]

\[ a_{ps} = \pi_{ps}(s) = \pi_{i*}(s), \]

\[ i* \text{ is index of selected policy} \]

\[ \geq E \left[ R(s, a_{ps}) + \beta \max_i V^{\pi_i}(T(s, a_{ps}), h-1) \right], \]

by inductive hypothesis

\[ \geq E \left[ \max_i \left( R(s, \pi_{i*}(s)) + \beta V^{\pi_i}(T(s, \pi_{i*}(s)), h-1) \right) \right] \]

\[ \geq E \left[ \max_i \left( R(s, \pi_i(s)) + \beta V^{\pi_i}(T(s, \pi_i(s)), h-1) \right) \right], \]

\[ i* \text{ was selected to be the max over } i \]
Policy Switching: Quality

\[ V^{\pi_{ps}}(s, h) = E\left[R(s, a_{ps}) + \beta V^{\pi_{ps}}(T(s, a_{ps}), h-1)\right], \]

\[ a_{ps} = \pi_{ps}(s) = \pi_{i*}(s), \]

\( i^* \) is index of selected policy

\[ \geq E\left[R(s, a_{ps}) + \beta \max_i V^{\pi_i}(T(s, a_{ps}), h-1)\right], \]

by inductive hypothesis

\[ \geq E\left[\max_i \left(R(s, \pi_{i*}(s)) + \beta V^{\pi_i}(T(s, \pi_{i*}(s)), h-1)\right)\right] \]

\[ \geq E\left[\max_i \left(R(s, \pi_i(s)) + \beta V^{\pi_i}(T(s, \pi_i(s)), h-1)\right)\right], \]

\( i^* \) was selected to be the max over \( i \)

\[ \geq \max_i E\left[R(s, \pi_i(s)) + \beta V^{\pi_i}(T(s, \pi_i(s)), h-1)\right], \]

sum of a max is no greater than max of sum
Policy Switching: Quality

\[ V^{\pi_{ps}}(s, h) = E\left[R(s, a_{ps}) + \beta V^{\pi_{ps}}(T(s, a_{ps}), h-1)\right], \]

\[ a_{ps} = \pi_{ps}(s) = \pi_{i*}(s), \]

\[ i* \text{ is index of selected policy} \]

\[ \geq E\left[R(s, a_{ps}) + \beta \max_i V^{\pi_i}(T(s, a_{ps}), h-1)\right], \]

by inductive hypothesis

\[ \geq E\left[\max_i \left(R(s, \pi_{i*}(s)) + \beta V^{\pi_i}(T(s, \pi_{i*}(s)), h-1)\right)\right] \]

\[ \geq E\left[\max_i \left(R(s, \pi_i(s)) + \beta V^{\pi_i}(T(s, \pi_i(s)), h-1)\right)\right], \]

\[ i* \text{ was selected to be the max over } i \]

\[ \geq \max_i E\left[R(s, \pi_i(s)) + \beta V^{\pi_i}(T(s, \pi_i(s)), h-1)\right], \]

sum of a max is no greater than max of sum

\[ \geq \max_i V^{\pi_i}(s, h) \]
Policy Switching Summary

• Easy way to produce an improved policy from a set of existing policies.
  ▲ Will not do any worse than the best policy in your set.

• Complexity does not depend on number of actions.
  ▲ So can be practical even when action space is huge, unlike policy rollout.

• Can combine with rollout for further improvement
  ▲ Just apply rollout to the switching policy.