Markov Decision Processes

Part 3: Computing State Utilities and Optimal Policies

CSE 4309 – Machine Learning

Vassilis Athitsos
Computer Science and Engineering Department
University of Texas at Arlington

Review: the Bellman Equation

2	+1	
1	START	-1
	1	2

$$\mathrm{U}(s) = R(s) + \gamma \max_{a \in A(s)} \left\{ \sum_{s'} [p(s' \mid s, a) U(s')] \right\}$$

- For each state s, we get a Bellman equation.
- If our environment has N states, we need to solve a system of N Bellman equations.
- In this system of equations, there is a total of N unknowns:
  - The N values $\mathrm{U}(s)$.
- There is an iterative algorithm for solving this system of equations, called the **<u>value iteration algorithm</u>**.

The Value Iteration Algorithm

- The value iteration algorithm computes the utility of each state for a Markov Decision Process.
- The algorithm takes the following inputs:
  - The set of states $\mathbb{S} = \{s_1, \dots, s_N\}$.
  - The set $\mathrm{A}(s)$ of actions available at each state s.
  - The transition model $p(s' \mid s, a)$.
  - The reward function $\mathrm{R}(s)$
  - The discount factor γ.
  - ε, which is the maximum error allowed in the utility of each state, in the result of the algorithm.

The Value Iteration Algorithm

function ValueIteration$(\mathbb{S}, \mathrm{A}, \mathrm{p}, \mathrm{R}, \gamma, \varepsilon)$

$\mathrm{N} =$ size of $\mathbb{S}$.

U' =new array of doubles, of size N.

Initialize all values of U' to 0.

repeat:

$\mathrm{U} =$ copy of array U'

$\delta = 0$

for each state s in $\mathbb{S}$:

$\mathrm{U}'[s] = R(s) + \gamma \max_{a \in A(s)} \left\{ \sum_{s'} \left[p(s' \mid s, a) \mathrm{U}[s'] \right] \right\}$

if $|\mathrm{U}'[s] - \mathrm{U}[s]| > \delta$ **then** $\delta = |\mathrm{U}'[s] - \mathrm{U}[s]|$

until $\delta < \varepsilon(1-\gamma)/\gamma$

return U

The Value Iteration Algorithm

```
function ValueIteration(𝕊, A, p, R, γ, ε)
    N = size of 𝕊.
    U′ =new array of doubles, of size N.
    Initialize all values of U′ to 0.
    repeat:
        U = copy of array U′
        δ = 0
        for each state s in 𝕊:
            U′[s] = R(s) + γ max_{a∈A(s)} {Σ_{s′} [p(s′| s, a)U[s′]]}
            if |U′[s] − U[s]| > δ then δ = |U′[s] − U[s]|
    until δ < ε(1 − γ)/γ
    return U
```

- We will skip the proof, but it can be proven that this algorithm converges to the correct solutions of the Bellman equations.
  - Details can be found in *S. Russell and P. Norvig, "Artificial Intelligence: A Modern Approach", third edition (2009), Prentice Hall.*

The Value Iteration Algorithm

```
function ValueIteration(𝕊, A, p, R, γ, ε)
    N = size of 𝕊.
    U′ =new array of doubles, of size N.
    Initialize all values of U′ to 0.
    repeat:
        U = copy of array U′
        δ = 0
        for each state s in 𝕊:
            U′[s] = R(s) + γ max_{a∈A(s)} {Σ_{s′}[p(s′| s, a)U[s′]]}
            if |U′[s] − U[s]| > δ then δ = |U′[s] − U[s]|
    until δ < ε(1 − γ)/γ
    return U
```

- The main operation of this algorithm is highlighted in red.
- We use the Bellman equation to update values $U(s)$ using the previous estimates for those values.
  - This update step is called a **Bellman update**.

The Value Iteration Algorithm

```
function ValueIteration(𝕊, A, p, R, γ, ε)
    N = size of 𝕊.
    U′ =new array of doubles, of size N.
    Initialize all values of U′ to 0.
    repeat:
        U = copy of array U′
        δ = 0
        for each state s in 𝕊:
            U′[s] = R(s) + γ max_{a∈A(s)} {Σ_{s′}[p(s′| s, a)U[s′]]}
            if |U′[s] − U[s]| > δ then δ = |U′[s] − U[s]|
    until δ < ε(1 − γ)/γ
    return U
```

- So, the value iteration algorithm can be summarized as follows:
  - Initialize utilities of states to zero values.
  - Repeat updating utilities of states using Bellman updates, until the estimated values converge.

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- We initialize all utility values to 0.

3				+1
2				-1
1	START			
	1	2	3	4

3	0	0	0	0
2	0		0	0
1	0	0	0	0
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after one round of updates:
  - The current estimate for each state s is $R(s)$.

3				+1
2				-1
1	START			
	1	2	3	4

3	-0.04	-0.04	-0.04	+1
2	-0.04		-0.04	-1
1	-0.04	-0.04	-0.04	-0.04
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after two rounds of updates:
  - Information about the +1 reward reached state $(3,3)$.

3				+1
2				-1
1	START			
	1	2	3	4

3	-0.08	-0.08	0.67	+1
2	-0.08		-0.08	-1
1	-0.08	-0.08	-0.08	-0.08
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after three rounds of updates:
  - Information about the +1 reward reached more states.

3				+1
2				-1
1	START			
	1	2	3	4

3	-0.11	0.43	0.73	+1
2	-0.11		0.35	-1
1	-0.11	-0.11	-0.11	-0.11
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after four rounds of updates:
  - Information about the +1 reward reached more states.

3				+1
2				-1
1	START			
	1	2	3	4

3	0.25	0.57	0.78	+1
2	-0.14		0.43	-1
1	-0.14	-0.14	0.19	-0.14
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after five rounds of updates:
  - Information about the +1 reward reached more states.

3				+1
2				-1
1	START			
	1	2	3	4

3	0.38	0.62	0.79	+1
2	0.12		0.47	-1
1	-0.16	0.07	0.24	-0.01
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after six rounds of updates:
  - Information about the +1 reward has reached all states.

3				+1
2				-1
1	START			
	1	2	3	4

3	0.45	0.64	0.79	+1
2	0.25		0.48	-1
1	0.04	0.15	0.30	0.05
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after seven rounds of updates:
  - Values keep getting updated.

3				+1
2				-1
1	START			
	1	2	3	4

3	0.48	0.65	0.79	+1
2	0.33		0.48	-1
1	0.16	0.21	0.32	0.09
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after eight rounds of updates:
  - Values continue changing.

3				+1
2				-1
1	START			
	1	2	3	4

3	0.50	0.65	0.80	+1
2	0.37		0.49	-1
1	0.23	0.23	0.34	0.11
	1	2	3	4

Utility Values

A Value Iteration Example

- Let's see how the value iteration algorithm works on our example.
- Assume:
  - $R(s) = -0.04$ if s is a non-terminal state.
  - $\gamma = 0.9$
- This is the result after 13 rounds of updates:
  - Values don't change much anymore after this round.

3				+1
2				-1
1	START			
	1	2	3	4

3	0.51	0.65	0.80	+1
2	0.40		0.49	-1
1	0.30	0.25	0.34	0.13
	1	2	3	4

Utility Values

Computing the Optimal Policy

- The value iteration algorithm computes $\mathrm{U}(s)$ for every state s.
- Once we have computed all values $\mathrm{U}(s)$, we can get the optimal policy π^* using this equation:

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \left\{ \sum_{s'} [p(s' \mid s, a) U(s')] \right\}$$

- Thus, $\pi^*(s)$ identifies the action that leads to the highest expected utility for the next state, as measured over all possible outcomes of that action.
- This approach is called **one-step look-ahead.**

Computing the Optimal Policy

- At the bottom we see the result of the value iteration algorithm for:
  - $R(s) = -0.02$ if s is a non-terminal state.
  - $\gamma = 1$
- How can we figure out the optimal policy based on that output?

3				+1
2				-1
1				
	1	2	3	4

Optimal Policy

3	0.90	0.93	0.95	+1
2	0.87		0.77	-1
1	0.85	0.82	0.79	0.57
	1	2	3	4

Utility Values

Computing the Optimal Policy

- Consider state $(2,3)$.
- What is the optimal action for that state?
- We must consider each action.
- If the action is "left", these are the possible next states:

Probability	Next State	Utility
0.8	(2,3)	0.77
0.1	(3,3)	0.95
0.1	(1,3)	0.79

- The weighted average is 0.79

3				+1
2				-1
1				
	1	2	3	4

Optimal Policy

3	0.90	0.93	0.95	+1
2	0.87		0.77	-1
1	0.85	0.82	0.79	0.57
	1	2	3	4

Utility Values

Computing the Optimal Policy

- Consider state $(2,3)$.
- What is the optimal action for that state?
- We must consider each action.
- If the action is "right", these are the possible next states:

Probability	Next State	Utility
0.8	(2,3)	-1
0.1	(3,3)	0.95
0.1	(1,3)	0.79

- The weighted average is -0.63

3				+1
2				-1
1				
	1	2	3	4

Optimal Policy

3	0.90	0.93	0.95	+1
2	0.87		0.77	-1
1	0.85	0.82	0.79	0.57
	1	2	3	4

Utility Values

Computing the Optimal Policy

- Consider state $(2,3)$.
- What is the optimal action for that state?
- We must consider each action.
- If the action is "up", these are the possible next states:

Probability	Next State	Utility
0.8	(3,3)	0.95
0.1	(2,4)	-1.00
0.1	(2,3)	0.77

- The weighted average is 0.74

3				+1
2				-1
1				
	1	2	3	4

Optimal Policy

3	0.90	0.93	0.95	+1
2	0.87		0.77	-1
1	0.85	0.82	0.79	0.57
	1	2	3	4

Utility Values

Computing the Optimal Policy

- Consider state $(2,3)$.
- What is the optimal action for that state?
- We must consider each action.
- If the action is "down", these are the possible next states:

Probability	Next State	Utility
0.8	(1,3)	0.79
0.1	(2,4)	-1.00
0.1	(2,3)	0.77

- The weighted average is 0.61

3				+1
2				-1
1				
	1	2	3	4

Optimal Policy

3	0.90	0.93	0.95	+1
2	0.87		0.77	-1
1	0.85	0.82	0.79	0.57
	1	2	3	4

Utility Values

Computing the Optimal Policy

- For state $(2,3)$, action "left" led to the highest expected utility for the next state.
- Thus, action "left" is the best action for state $(2,3)$.
- Note that choosing the best action is not always to try to move towards the best state.
  - At state $(2,3)$ the best action is towards the blocked square, to play it safe.
  - Going up is risky, it has a 10% chance to lead to the -1 state.

3				+1
2			←	-1
1				
	1	2	3	4

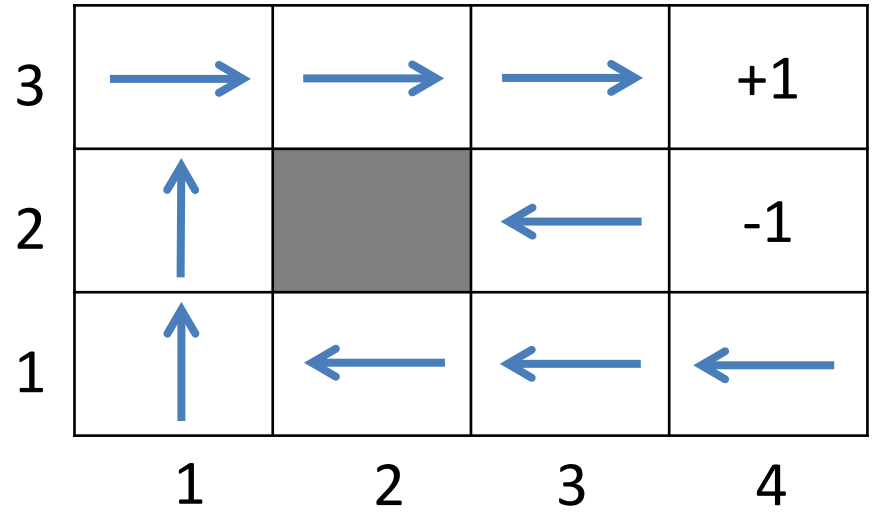
Optimal Policy

3	0.90	0.93	0.95	+1
2	0.87		0.77	-1
1	0.85	0.82	0.79	0.57
	1	2	3	4

Utility Values

Computing the Optimal Policy

- Here is the optimal policy for:
  - $R(s) = -0.02$ if s is a non-terminal state.
  - $\gamma = 1$
- Note that choosing the best policy is more complicated than simply pointing to the direction of highest reward.
  - At state $(2,3)$ the best action is towards the blocked square, to play it safe.
  - Going up is risky, it has a 10% chance to lead to the -1 state.



Optimal Policy

3	0.90	0.93	0.95	+1
2	0.87		0.77	-1
1	0.85	0.82	0.79	0.57
	1	2	3	4

Utility Values

The Policy Iteration Algorithm

- There is an alternative algorithm for computing optimal policies, that is more efficient.
- Remember that, if we know the utility of each state, we can compute the optimal policy π^* using:

$$\pi^*(s) = \underset{a \in A(s)}{\operatorname{argmax}} \left\{ \sum_{s'} [p(s' \mid s, a) U(s')] \right\}$$

- However, to get the right $\pi^*(s)$, we don't need to know the utilities very accurately.
  - We just need to know the utilities accurately enough, so that, for each state s, argmax chooses the right action.

The Policy Iteration Algorithm

- This alternative algorithm for computing optimal policies is called the **<u>policy iteration algorithm</u>**.
- It is an iterative algorithm.
- Initialization:
  - Initiate some policy π_0 with random choices for the best action at each state.
- Main loop:
  - **<u>Policy evaluation</u>**: given the current policy π_i, calculate utility values $U^{\pi_i}(\mathrm{s})$, corresponding to the utility of each state s **if the agent follows policy $\boldsymbol{\pi_i}$**.
  - **<u>Policy improvement</u>**: Given current utility values $U^{\pi_i}(\mathrm{s})$, use one-step look-ahead to compute new policy π_{i+1}.

The Policy Iteration Algorithm

- To be able to implement the policy iteration algorithm, we need to specify how to carry out each of the two steps of the main loop:
  - Policy evaluation.
  - Policy improvement.

The Policy Evaluation Step

- Task: calculate utility values $U^{\pi_i}(\mathrm{s})$, corresponding to the assumption that **the agent follows policy $\boldsymbol{\pi_i}$**.
- When the policy was not known, we used the **Bellman equation:**

$$\mathrm{U}(s) = R(s) + \gamma \max_{a \in A(s)} \left\{ \sum_{s'} [p(s' \mid s, a) U(s')] \right\}$$

- Now that the policy π_i is specified, we can instead use a simplified version of the **Bellman equation:**

$$U^{\pi_i}(s) = R(s) + \gamma \sum_{s'} [p(s' \mid s, \pi_i(s)) U^{\pi_i}(s')]$$

- Key difference: now $\pi_i(s)$ specifies the action for each state s, so we do not need to look for the max over all possible actions.

The Policy Evaluation Step

$$U^{\pi_i}(s) = R(s) + \gamma \sum_{s'} [p(s' \mid s, \pi_i(s)) U^{\pi_i}(s')]$$

- This is a linear equation.
  - The original Bellman equation, taking the max out of all possible actions, is **not** a linear equation.
- If we have N states, we get N linear equations of this form, with N unknowns.
- We can solve those N linear equations in $\mathrm{O}(N^3)$ time, using standard linear algebra methods.

The Policy Evaluation Step

- For large state spaces, $\mathrm{O}(N^3)$ is prohibitive.
- Alternative: do some rounds of iterations.

function PolicyEvaluation$(\mathbb{S}, \mathrm{p}, \mathrm{R}, \gamma, \pi_i, K, \mathrm{U})$
  $\mathrm{U}_0 =$ copy of U
  for $\boldsymbol{k = 1}$ **to** $\boldsymbol{K}$**:**
    for each state s in $\mathbb{S}$:
      $U_k(s) = R(s) + \gamma \sum_{s'} [p(s' \mid s, \pi_i(s)) U_{k-1}(s')]$
  return U_k

- Obviously, doing K iterations does not guarantee that the utilities are computed correctly.
- Parameter K allows us to trade speed for accuracy. Larger values lead to slower runtimes and higher accuracy.

The Policy Evaluation Step

- For large state spaces, $\mathrm{O}(N^3)$ is prohibitive.
- Alternative: do some rounds of iterations.

function PolicyEvaluation$(\mathbb{S}, \mathrm{p}, \mathrm{R}, \gamma, \pi_i, K, \mathrm{U})$
  $\mathrm{U}_0 =$ copy of U
  for $\boldsymbol{k = 1}$ **to** $\boldsymbol{K}$**:**
    for each state s in $\mathbb{S}$:
      $U_k(s) = R(s) + \gamma \sum_{s'} [p(s' \mid s, \pi_i(s)) U_{k-1}(s')]$
  return U_k

- The PolicyEvaluation function takes as argument a current estimate U.
  - See later how the PolicyEvaluation function is called from the PolicyIteration function.

The Policy Iteration Algorithm

function PolicyIteration$(\mathbb{S}, \mathrm{A}, \mathrm{p}, \mathrm{R}, \gamma, K)$

$N =$ size of $\mathbb{S}$.

$\mathrm{U} =$new array of size N, all values initialized to 0

$\pi =$ new array of actions, of size N

Initialize all values of π to random (but legal) actions

repeat:

$\mathrm{U} =$ **PolicyEvaluation**$(\mathbb{S}, \mathrm{p}, \mathrm{R}, \gamma, \pi, K, \mathrm{U})$

$\mathrm{unchanged} = \mathbf{true}$

for each state s in $\mathbb{S}$:

if $\max_{a \in A(s)} \left\{ \sum_{s'} \left[p(s' \mid s, a) \mathrm{U}[s'] \right] \right\} > \sum_{s'} \left[p(s' \mid s, \pi[s]) \mathrm{U}[s'] \right]$

$\pi[s] = \underset{a \in A(s)}{\mathrm{argmax}} \left\{ \sum_{s'} \left[p(s' \mid s, a) \mathrm{U}[s'] \right] \right\}$

$\mathrm{unchanged} = \mathbf{false}$

until $\mathrm{unchanged} == \mathbf{true}$

return π

The Policy Iteration Algorithm

function PolicyIteration$(\mathbb{S}, \mathrm{A}, \mathrm{p}, \mathrm{R}, \gamma, K)$

N = size of $\mathbb{S}$.

U =new array of size N, all values initialized to 0

π = new array of actions, of size N

Initialize all values of π to random (but legal) actions

repeat:

U = **PolicyEvaluation**$(\mathbb{S}, \mathrm{p}, \mathrm{R}, \gamma, \pi, K, \mathrm{U})$

unchanged = **true**

for each state s in $\mathbb{S}$:

if $\max_{a \in A(s)} \{\sum_{s'} [p(s' \mid s, a)\mathrm{U}[s']]\} > \sum_{s'} [p(s' \mid s, \pi[s])\mathrm{U}[s']]$

$\pi[s] = \operatorname{argmax}_{a \in A(s)} \{\sum_{s'} [p(s' \mid s, a)\mathrm{U}[s']]\}$

unchanged = **false**

until unchanged == **true**

return π

The main loop alternates between:

- Updating the utilities given the policy.
- Updating the policy given the utilities.

The main loop exits when the policy stops changing.

The Policy Iteration Algorithm

function PolicyIteration$(\mathbb{S}, \mathrm{A}, \mathrm{p}, \mathrm{R}, \gamma, K)$

N = size of $\mathbb{S}$.

U =new array of size N, all values initialized to 0

π = new array of actions, of size N

Initialize all values of π to random (but legal) actions

repeat:

$\mathrm{U} =$ **PolicyEvaluation**$(\mathbb{S}, \mathrm{p}, \mathrm{R}, \gamma, \pi, K, \mathrm{U})$

unchanged $=$ **true**

for each state s in $\mathbb{S}$:

if $\max_{a \in A(s)} \{\sum_{s'} [p(s' \mid s, a)\mathrm{U}[s']]\} > \sum_{s'} [p(s' \mid s, \pi[s])\mathrm{U}[s']]$

$\pi[s] = \operatorname{argmax}_{a \in A(s)} \{\sum_{s'} [p(s' \mid s, a)\mathrm{U}[s']]\}$

unchanged $=$ **false**

until unchanged == **true**

return π

The main loop alternates between:

- Updating the utilities given the policy.
- Updating the policy given the utilities.

The main loop exits when the policy stops changing.

The Policy Iteration Algorithm

function PolicyIteration$(\mathbb{S}, \mathrm{A}, \mathrm{p}, \mathrm{R}, \gamma, K)$

N = size of $\mathbb{S}$.

U =new array of size N, all values initialized to 0

π = new array of actions, of size N

Initialize all values of π to random (but legal) actions

repeat:

$\mathrm{U} =$ **PolicyEvaluation**$(\mathbb{S}, \mathrm{p}, \mathrm{R}, \gamma, \pi, K, \mathrm{U})$

$\mathrm{unchanged} = \mathbf{true}$

for each state s in $\mathbb{S}$:

if $\max_{a \in A(s)} \{\sum_{s'} [p(s' \mid s, a)\mathrm{U}[s']]\} > \sum_{s'} [p(s' \mid s, \pi[s])\mathrm{U}[s']]$

$\pi[s] = \underset{a \in A(s)}{\mathrm{argmax}} \{\sum_{s'} [p(s' \mid s, a)\mathrm{U}[s']]\}$

$\mathrm{unchanged} = \mathbf{false}$

until $\mathrm{unchanged} == \mathbf{true}$

return π

The main loop alternates between:

- Updating the utilities given the policy.
- Updating the policy given the utilities.

The main loop exits when the policy stops changing.

Markov Decision Processes: Recap

- In Markov Decision Processes:
  - Each state has a reward $R(s)$.
  - Each state sequence $(s_0,\ s_1, \dots, s_T)$ has a utility U_h which is computed by adding the discounted rewards of all states in the sequence.
  - An action can lead to multiple outcomes. The probability of each outcome given the state and the action is known.
  - A policy is a function mapping states to actions.
  - The utility of a state s_0 is the expected utility measured over all state sequences that can lead from s_0 to a terminal state, under the assumption that the agent follows the optimal policy.

Markov Decision Processes: Recap

- The value iteration algorithm computes the utility of each state using an iterative approach.
  - Once the utilities of all states have been computed, the optimal policy is defined by identifying, for each state, the action leading to the highest expected utility.
- The policy iteration algorithm is a more efficient alternative, at the cost of possibly losing some accuracy.
  - It computes the optimal policy directly, without computing exact values for the utilities.
  - Utility values are updated for a few rounds only, and not until convergence.