# EE496 : COMPUTATIONAL INTELLIGENCE
# RL02: REINFORCEMENT LEARNING : MDP II

UGUR HALICI
METU: Department of Electrical and Electronics Engineering (EEE)
METU-Hacettepe U: Neuroscience and Neurotechnology (NSNT)

## Iterative Solution

Define $U_1(s)$ to be the utility if the agent is at state s and lives for 1 time step

$$U_1(s) = R(s)$$

Calculate this for all states $s$

Define $U_2(s)$ to be the utility if the agent is at state s and lives for 2 time steps

$$U_2(s) = R(s) + \gamma \max_a \sum_{s'} T(s,a,s') U_1(s')$$

This has already been calculated above

## The Bellman Update

More generally, we have:

$$U_{i+1}(s) = R(s) + \gamma \max_a \sum_{s'} T(s,a,s')U_i(s')$$

- This is the maximum possible expected sum of discounted rewards (ie. the utility) if the agent is at state s and lives for i+1 time steps
- This equation is called the Bellman Update

# The Bellman Update

As the number of iterations goes to infinity, $U_{i+1}(s)$ converges to an equilibrium value U*(s).

- The final utility values U*(s) are solutions to the Bellman equations. Even better, they are the unique solutions and the corresponding policy is optimal
- This algorithm is called **Value-Iteration**
- The optimal policy is given by:

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')U^*(s')$$

## The Value-Iteration Algorithm

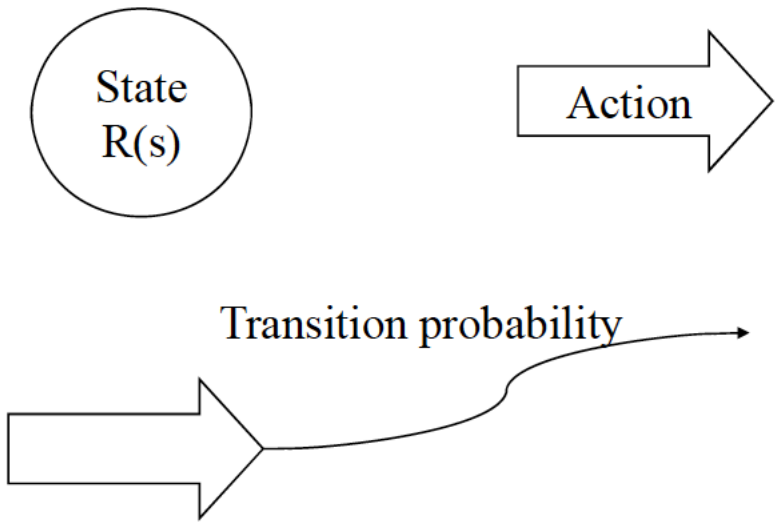
$$U_1(s) = R(s)$$
$$U_{i+1}(s) = R(s) + \gamma \max_a \sum_{s'} T(s,a,s')U_i(s')$$

- Apply Bellmann update until until utility function converges (to U*(s))
- The optimal policiy is given by:

$$\pi^*(s) = \arg\max_a \sum_{s'} T(s,a,s')U^*(s')$$

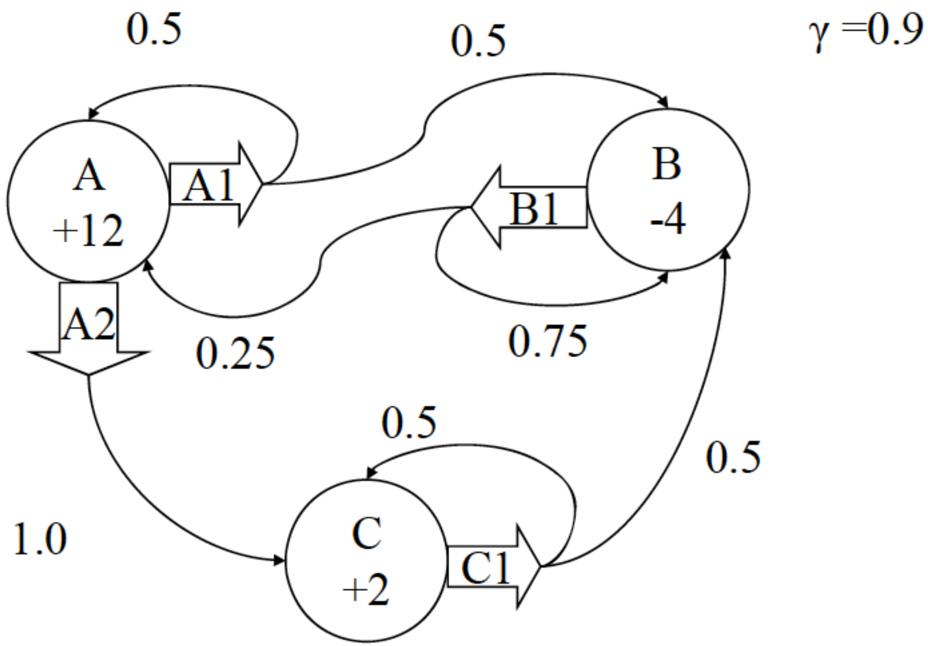
## Example

- We will use the following convention when drawing MDPs graphically:



# Example

- 



# Example

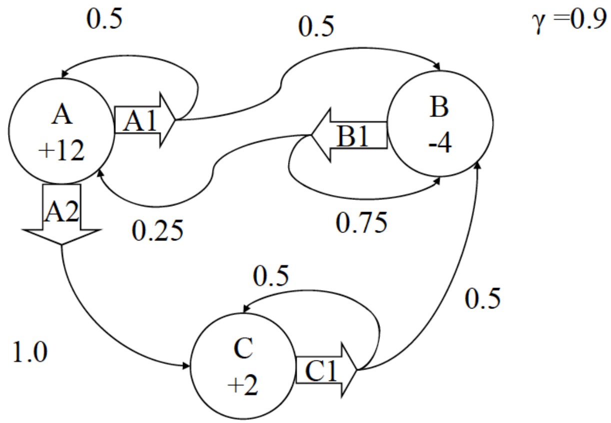
i=1

$U_1$ (A) = R(A)=12

$U_1$ (B) = R(B)=- 4

$U_1$(C) =R(C) 2

| $U_1$(A) | $U_1$(B) | $U_1$(C) |
|---|---|---|
| 12 | -4 | 2 |




i=2

$U_2$(A) = 12 + (0.9) * max{(0.5)(12)+(0.5)(-4), (1.0)(2)}

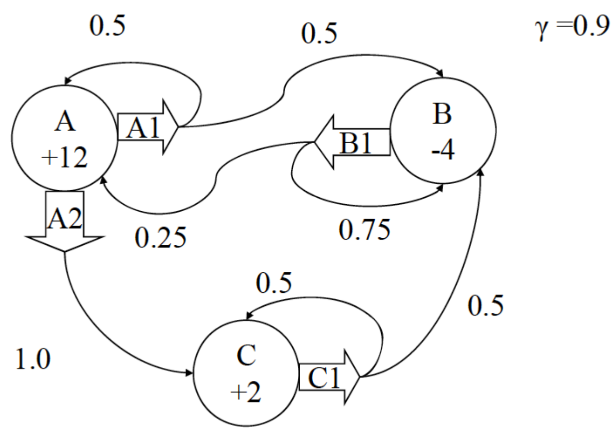
= 12 + (0.9)*max{4.0, 2.0} = 12 + 3.6 = 15.6

$U_2$(B) = -4 + (0.9) * {(0.25)(12)+(0.75)(-4)} = -4 + (0.9)*0 = -4

$U_2$(C) = 2 + (0.9) * {(0.5)(2)+(0.5)(-4)} = 2 + (0.9)*(-1) = 2-0.9 = 1.1

# Example

| $U_2(A)$ | $U_2(B)$ | $U_2(C)$ |
|---|---|---|
| 15.6 | -4 | 1.1 |




i=3

$U_3(A)$ = 12 + (0.9) * max{(0.5)(15.6)+(0.5)(-4) (1 0)(1 1)}

= 12 + (0 9) *max{5 8 1 1} = 12 +(0.9)(5.8) = 17.22

$U_3(B)$ = -4 + (0.9) * {(0.25)(15.6)+(0.75)(-4)}

= -4 + (0.9)*(3.9-3) = -4 + (0.9)(0.9) = -3.19

$U^3(C)$ = 2 + (0.9) * {(0.5)(1.1)+(0.5)(-4)}

= 2 +(0.9)*{0.55-2.0} = 2 + (0.9)(-1.45) = 0.695

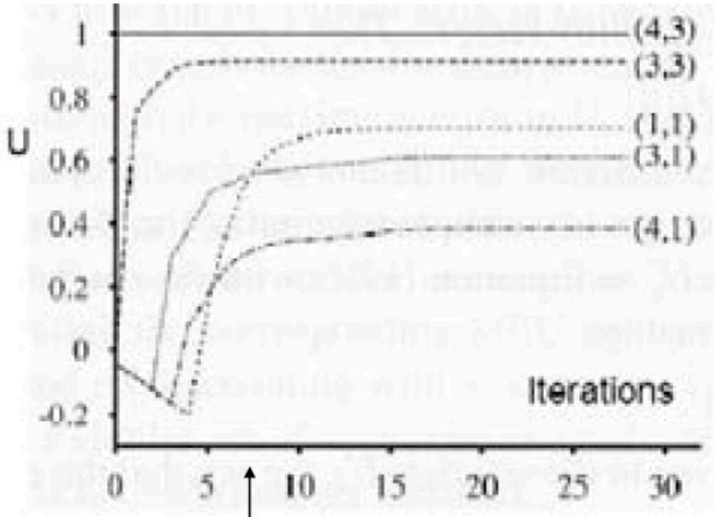
## The Bellman Update

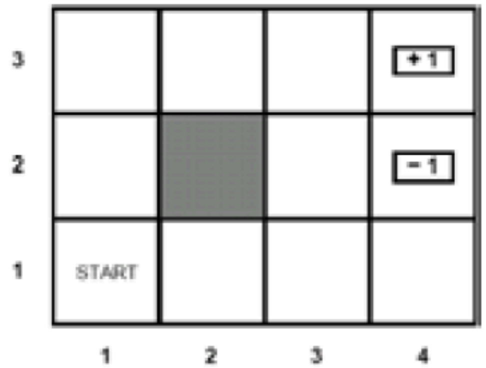
What exactly is going on?

- Think of each Bellman update as an update of each local state
- If we do enough local updates we end up updates, propagating information throughout the state space

# Value Iteration on the Maze

•




Notice that rewards are negative until a path to (4,3) is found, resulting in an increase in U

# Value-Iteration Termination

When do you stop?

In an iteration over all the states, keep track of the maximum change in utility of any state (call this δ )

When δ  is less than some pre- defined threshold, stop

This will give us an approximation to the true utilities, we can act greedily based on theapproximated state utilities

## Comments

- Value iteration is designed around the idea of the utilities of the states
- The computational difficulty comes from the max operation in the bellman equation
- Instead of computing the general utility of a state (assuming acting optimally), a much easier quantity to compute is the utility of a state assuming a policy

# Utility policy at state s

$U_\pi(s)$: the utility of policy $\pi$ at state s

$U^*(s)$ can be considered as $U_{\pi^*}(s)$ where $\pi^*$ is an optimal policy

Given a fixed policy, can compute its utility at state ***s*** as follows:

$$U_\pi(s) = R(s) + \gamma \sum_{s'} T(s, \pi(s), s') \cdot U_\pi(s')$$

Note the difference from:

$$U(s) = R(s) + \gamma \max_a \sum_{s'} T(s, a, s') U(s')$$

## Evaluating a Policy

- Once we compute the utilities, we can easily improve the current policy by onestep look- ahead:

$$\pi'(s) = \arg\max_a \sum_{s'} T(s,a,s')U_\pi(s')$$

- This suggests a different approach for finding optimal policy

# Policy Iteration

- Start with a randomly chosen initial policy $\pi_0$
- Iterate until no change in utilities:

1. Policy evaluation: given a policy $\pi_i$, calculate the utility $U_i(s)$ of every state s using policy $\pi_i$

2. Policy improvement: calculate the new policy $\pi_{i+1}$ using one-step look-ahead based on $U_i(s)$ ie.

$$\pi_{i+1}(s) = \arg\max_{a} \sum_{s'} T(s, a, s') U_i(s')$$

# Policy Evaluation

- Policy improvement is straightforward
- Policy evaluation requires a simpler version of the Bellman equation
- Compute U(s) for every state s using $\pi_i$

$$U_i(s) = R(s) + \gamma \sum_{s'} T(s, \pi_i(s), s')U_i(s')$$

- Notice that there is no max operator, so the above equations are linear! O(n3) where n is the number of states

## Modified Policy Evaluation

- $O(n^3)$ is still too expensive for large state spaces
- Instead of calculating exact utilities, we could calculate approximate utilities
- The simplified Bellman update is:

$$U_{i+1}(s) \leftarrow R(s) + \gamma \sum_{s'} T(s, \pi_i(s), s') U_i(s')$$

- Repeat the above k times to get the next utility estimate

This is called modified policy iteration

## Comparison

- Which would you prefer, policy or value iteration?

Depends…

- If you have lots of actions in each state: policy iteration
- If you have a pretty good policy to start with: policy iteration
- If you have few actions in each state: value iteration

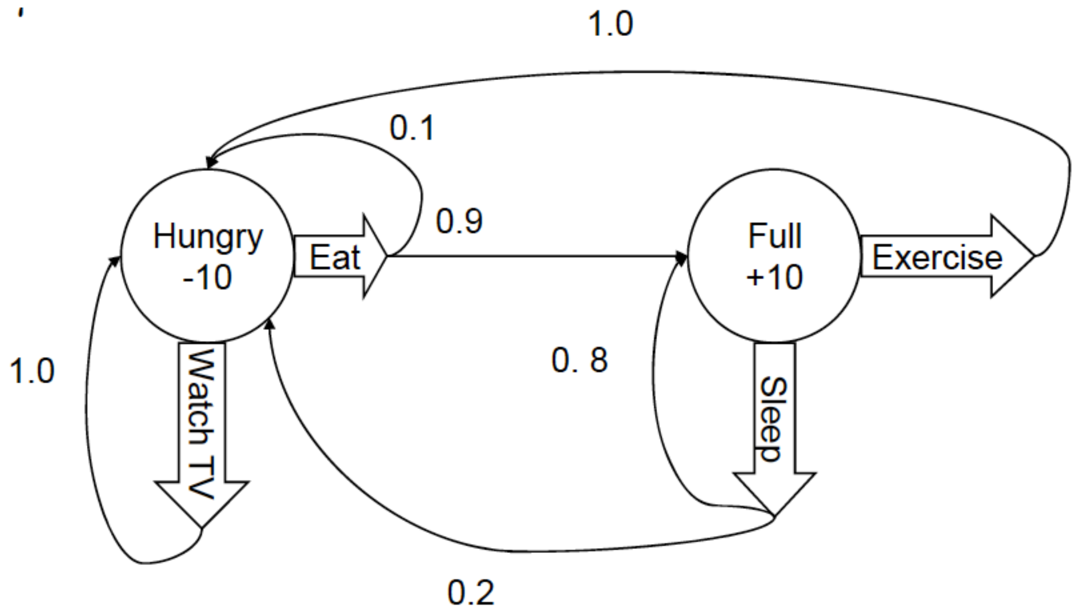
# Policy iteration comments

- Each step of policy iteration is guaranteed to strictly improve the policy at some state when improvement is possible
- Converge to optimal policy
- Gives exact value of optimal policy

# Policy Iteration Example

Do one iteration of policy iteration on the MDP below. Assume an initial policy of $\pi_1$(Hungry) = Eat and $\pi_1$(Full) = Sleep.

Let $\gamma = 0.9$



# Review: Policy Iteration

- Start with a randomly chosen initial policy π0
- Iterate until no change in utilities:

**1. Policy evaluation**: given a policy πi, calculate the utility Ui(s) of every state s using policy πi by solving the system of equations:

$$U_{\pi_i}(s) = R(s) + \gamma \sum_{s'} T(s, \pi_i(s), s') U_i(s')$$

**2. Policy improvement:** calculate the new policy $\pi_{i+1}$ using one-step look-ahead based on Ui(s):

$$\pi_{i+1}(s) = \arg\max_a \sum_{s'} T(s, a, s') U_i(s')$$

# Policy Iteration Example

Policy Evaluation Phase

**Use initial policy for Hungry: $\pi_1$(Hungry) = Eat**

$U_1$(Hungry) = -10 + (0.9)[(0.1)$U_1$(Hungry)+(0.9)$U_1$(Full)]

⇒$U_1$(Hungry) = -10 + ( 0.09)$U_1$(Hungry)+(0.81)$U_1$(Full)

⇒(0.91)$U_1$(Hungry)-(0.81)$U_1$(Full) = -10

**Use initial policy for Full: $\pi_1$(Full) = Sleep.**

$U_1$(Full) = 10 + (0.9)[(0.8)$U_1$(Full) + (0.2)$U_1$(Hungry)]

⇒U1(Full) = 10 + (0.72)$U_1$(Full) + (0.18)$U_1$(Hungry)]

⇒(0.28)$U_1$(Full) - (0.18)$U_1$(Hungry) = 10

# Policy Iteration Example

$(0.91)U_1(\text{Hungry})-(0.81)U_1(\text{Full}) = -10$ ....(Eq. 1)

$(0.28)U_1(\text{Full}) - (0.18)U_1(\text{Hungry}) = 10$ ... (Eq. 2)

Solve for $U_1(\text{Hungry})$ and $U_1(\text{Full})$

From Equation 1:

$(0.91)U_1(\text{Hungry}) = -10+(0.81)U_1(\text{Full})$

$\Rightarrow U_1(\text{Hungry}) = (-10/0.91)+(0.81/0.91)U_1(\text{Full})$

$\Rightarrow U_1(\text{Hungry})=-10.9+(0.89)U_1(\text{Full})$

# Policy Iteration Example

$(0.91)U_1(\text{Hungry})-(0.81)U_1(\text{Full}) = -10$ ....(Eq. 1)

$(0.28)U_1(\text{Full}) - (0.18)U_1(\text{Hungry}) = 10$ ... (Eq. 2)

Solve for $U_1(\text{Hungry})$ and $U_1(\text{Full})$

Substitute $U_1(\text{Hungry})=-10.9+(0.89)U_1(\text{Full})$ into Eq. 2

$(0.28)U_1(\text{Full}) - (0.18)\mathbf{[-10.9+(0.89)U_1(Full)]}=10$

$\Rightarrow(0.28)U_1(\text{Full}) + 1.96-(0.16)U_1(\text{Full})=10$

$\Rightarrow(0.12)U_1(\text{Full})=8.04$

$\Rightarrow U_1(\text{Full})=67$

$\Rightarrow U_1(\text{Hungry})=-10.9+(0.89)(67)=-10.9+59.63=48.7$

# Policy Iteration Example

$$
\begin{aligned}
&\pi_2(\text{Hungry}) \\
&= \operatorname*{argmax}_{\{\text{Eat},\text{WatchTV}\}} \begin{Bmatrix} \text{T(Hungry, Eat, Full)}\text{U}_1\text{(Full)} + & \\ \quad \text{T(Hungry, Eat, Hungry)}\text{U}_1\text{(Hungry)} & \text{[Eat]} \\ \text{T(Hungry, WatchTV, Hungry)}\text{U}_1\text{(Hungry)} & \text{[WatchTV]} \end{Bmatrix} \\
&= \operatorname*{argmax}_{\{\text{Eat},\text{WatchTV}\}} \begin{Bmatrix} (0.9)\text{U1(Full)} + (0.1)\text{U1(Hungry)} & \text{[Eat]} \\ (1.0)\text{U1(Hungry)} & \text{[WatchTV]} \end{Bmatrix} \\
&= \operatorname*{argmax}_{\{\text{Eat},\text{WatchTV}\}} \begin{Bmatrix} (0.9)(67) + (0.1)(48.7) & \text{[Eat]} \\ (1.0)(48.7) & \text{[WatchTV]} \end{Bmatrix} \\
&= \operatorname*{argmax}_{\{\text{Eat},\text{WatchTV}\}} \begin{Bmatrix} 65.2 & \text{[Eat]} \\ 48.7 & \text{[Watch]} \end{Bmatrix} \\
&= \text{Eat}
\end{aligned}
$$

# Policy Iteration Example

$$\pi_2(\text{Full})$$

$$= \underset{\{\text{Exercise,Sleep}\}}{\text{argmax}} \begin{Bmatrix} \text{T(Full, Exercise, Hungry)U}_1\text{(Hungry)} & \text{[Exercise]} \\ \text{T(Full, Sleep, Full)U}_1\text{(Full)} + & \\ \quad \text{T(Full, Sleep, Hungry)U}_1\text{(Hungry)} & \text{[Sleep]} \end{Bmatrix}$$

$$= \underset{\{\text{Exercise,Sleep}\}}{\text{argmax}} \begin{Bmatrix} (1.0)\text{U}_1\text{(Hungry)} & \text{[Exercise]} \\ (0.8)\text{U}_1\text{(Full)} + (0.2)\text{U}_1\text{(Hungry)} & \text{[Sleep]} \end{Bmatrix}$$

$$= \underset{\{\text{Exercise,Sleep}\}}{\text{argmax}} \begin{Bmatrix} (1.0)(48.7) & \text{[Exercise]} \\ (0.8)(67) + (0.2)(48.7) & \text{[Sleep]} \end{Bmatrix}$$

$$= \underset{\{\text{Exercise,Sleep}\}}{\text{argmax}} \begin{Bmatrix} 48.7 & \text{[Exercise]} \\ 63.34 & \text{[Sleep]} \end{Bmatrix}$$

$$= \text{Sleep}$$

Therefore:

- π2 (Hungry) =Eat
- π2 (Full) = Sleep

# So far ....

Given an MDP model we know how to find optimal policies

– Value Iteration or Policy Iteration

• But what if we don't have any form of the model of the world (e.g., T, and R)

– Like when we were babies . . .

– All we can do is wander around the world observing what happens, getting rewarded and punished

- This is what reinforcement learning about