

CMPSCI 383: Artificial Intelligence  
Lecture 17, November 8, 2011  
**Making Complex Decisions**

Philip Thomas (TA)

# Disclaimer

- I'm not covering everything in 17.1-17.3

# Sequential Decision Problems

- Chapter 16 was “one shot”
  - Where should the airport be placed?
  - Should I accept a certain bet?
- What about problems where an agent must make a sequence of decisions?
- We assume that a decision will influence the future decisions that must be made.
  - Robot control (helicopter / balancing)
  - Elevator scheduling
  - Anesthesia administration, DRAM schedulers, Backgammon...

# Black Sheep Wall

Chess	Poker
Checkers	Blackjack
Tag	Marco Polo

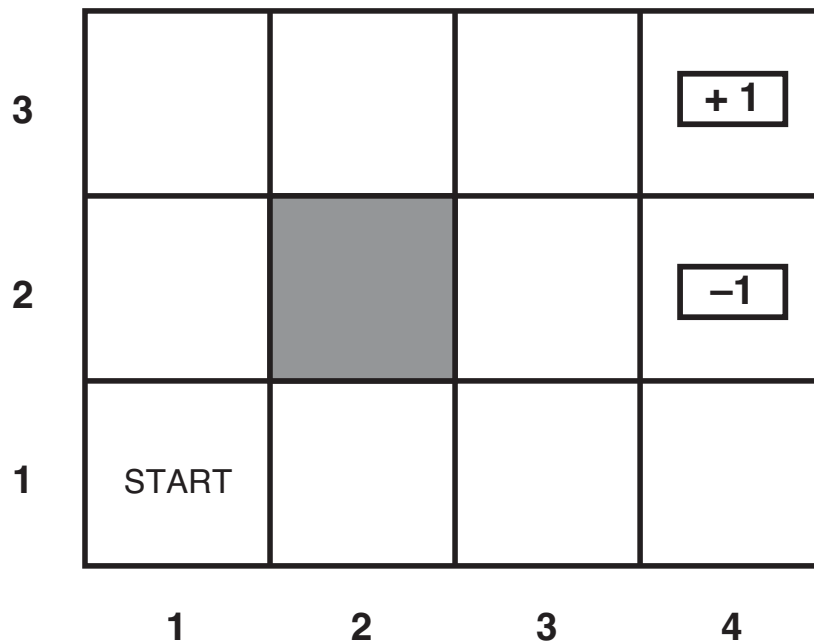
Fully Observable

Partially Observable

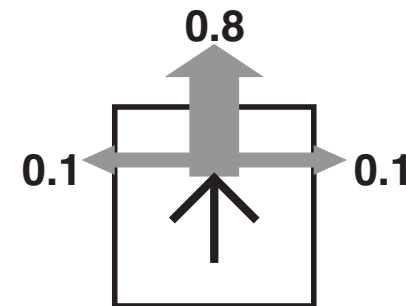
For now, we assume the problem is fully observable.

# A Simple Example

- “Gridworld” with 2 Goal states
- Actions: Up, Down, Left, Right
- Fully observable: Agent knows where it is



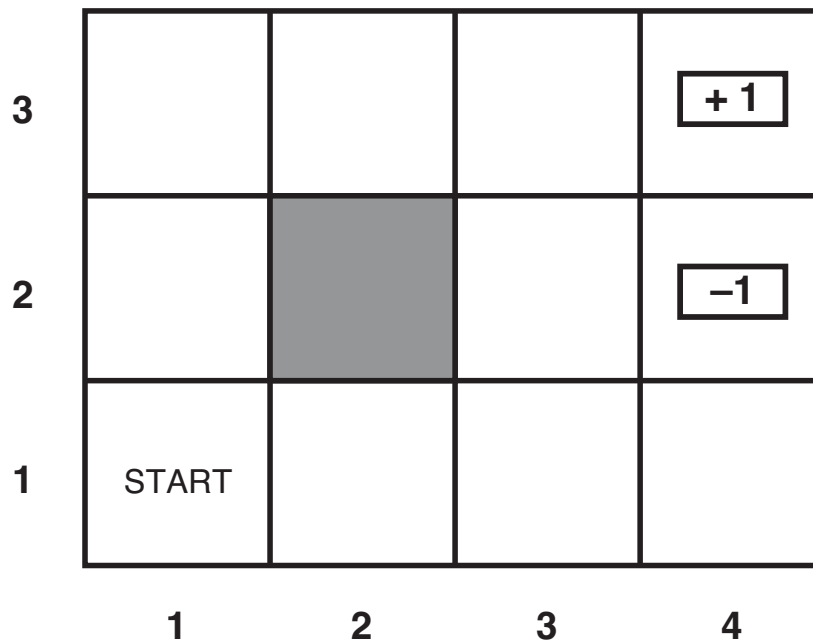
(a)



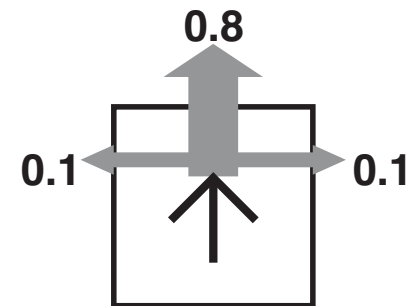
(b)

# Transition Model

$$P(s' | s, a)$$



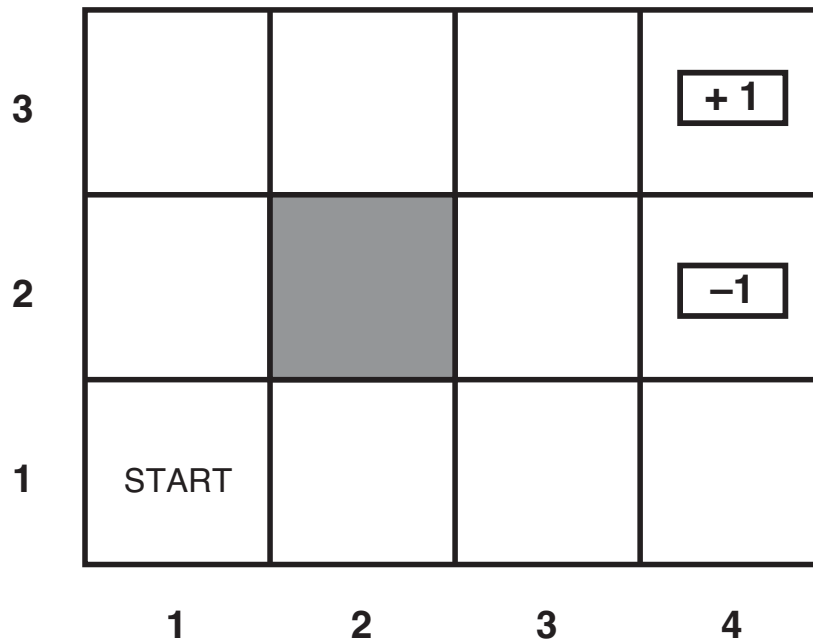
(a)



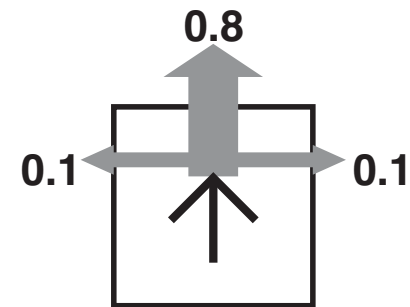
(b)

# Markov Assumption

$$P(s' | s, a)$$



(a)



(b)

# Markov Assumption

- ... is it reasonable?
- Real world problems where it applies?
- Real world problems where it doesn't apply?



# Agent's Utility Function

- Performance depends on the entire sequence of states and actions.
  - “Environment history”
- In each state, the agent receives a reward  $R(s)$ .
  - The reward is real-valued. It may be positive or negative.
- Utility of environment history = sum of reward received.

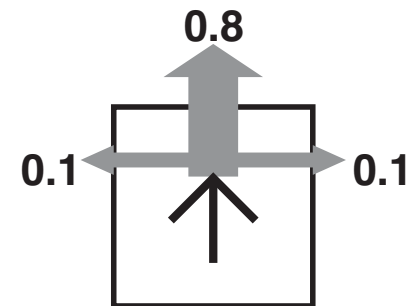
# Reward function

$$P(s' | s, a)$$

$$R(s)$$

3	-.04	-.04	-.04	<div>+1</div>
2	-.04		-.04	<div>-1</div>
1	START	-.04	-.04	-.04
	1	2	3	4

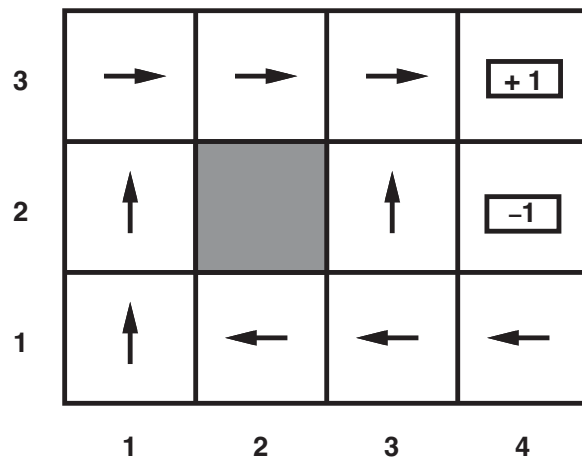
(a)



(b)

# Decision Rules

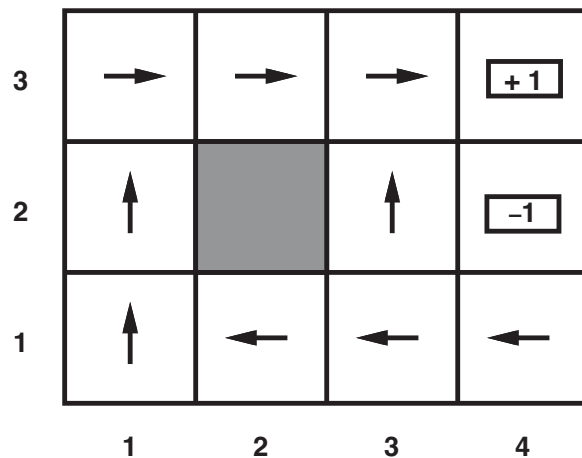
- Decision rules say what to do in each state.
- Often called policies,  $\pi$ .
- Action for state  $s$  is given by  $\pi(s)$ .



(a)

# Our Goal

- Find the policy that maximizes the expected sum of rewards.
- Called an *optimal policy*.



(a)

# Markov Decision Process (MDP)

- $M=(S,A,P,R)$
- $S$  = set of possible states
- $A$  = set of possible actions
- $P(s' | s,a)$  gives transition probabilities
- $R$  = reward function
- Goal, find an optimal policy,  $\pi^*$ .

# Break from the book

- Let's talk about why MDPs are awesome.

# MDPs in Literature

## Background

A  $d$ -dimensional continuous-state Markov decision process (MDP) is a tuple  $M = (S, A, P, R)$ , where  $S \subseteq \mathbb{R}^d$  is a set of possible state vectors,  $A$  is a set of actions,  $P$  is the transition model (with  $P(\mathbf{x}, a, \mathbf{x}')$  giving the probability of moving from state  $\mathbf{x}$  to state  $\mathbf{x}'$  given action  $a$ ), and  $R$  is the reward function (with  $R(\mathbf{x}, a, \mathbf{x}')$  giving the reward obtained from executing action  $a$  in state  $\mathbf{x}$  and transitioning to state  $\mathbf{x}'$ ). Our goal is to learn a policy,  $\pi$ , mapping state vectors to actions so as to maximize return (discounted sum of rewards). When  $P$  is known, this can be achieved by learning a *value function*,  $V$ , mapping state vectors to return, and se-

# MDPs in Literature

## 2. Background

Sequential decision problems are often formulated as MDPs, each a tuple  $M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ , where  $\mathcal{S}$  and  $\mathcal{A}$  are the sets of possible states and actions respectively,  $\mathcal{P}$  gives state transition probabilities:  $\mathcal{P}(s, a, s') = \Pr(s_{t+1}=s'|s_t=s, a_t=a)$ , where  $t$  is the current time step, and  $\mathcal{R}(s, a, s', r) = \Pr(r_t=r|s_t=s, a_t=a, s_{t+1}=s')$  is the reward distribution.  $\mathcal{R}$  represents the reward distribution rather than the expected reward to facilitate proofs in the appendix. If  $\mathcal{S}$ ,  $\mathcal{A}$ , or  $\mathcal{U}$  are uncountable, replace the corresponding probability distributions with probability density functions, summations with integrals, and mixima with suprema. An agent,  $A$ , with time-varying parameters  $\theta_t \in \Theta$  (typically function approximator weights, learning rates,




# MDPs in the Real World



Wikipedia

it

 [Log in / create account](#)

[Article](#)

[Discussion](#)

[Read](#)

[Edit](#)

[View history](#)



## TD-Gammon

From Wikipedia, the free encyclopedia

**TD-Gammon** was a [computer backgammon](#) program developed in [1992](#) by [Gerald Tesauro](#) at IBM's [Thomas J. Watson Research Center](#). Its name comes from the fact that it is an [artificial neural net](#) trained by a form of [temporal-difference learning](#), specifically [TD-lambda](#).

TD-Gammon achieved a level of play just slightly below that of the top human backgammon players of the time. It explored strategies that humans had not pursued and led to advances in the theory of correct backgammon play.



DIA  
opedia

it

edia

lia  
ortal  
es  
edia

l)

Log in / create account



Article **Discussion**

Read

Edit

View history

Search



## Reinforcement learning

From Wikipedia, the free encyclopedia

*For reinforcement learning in psychology, see [Reinforcement](#).*

Inspired by [behaviorist psychology](#), **reinforcement learning** is an area of [machine learning](#) in [computer science](#), concerned with how an *agent* ought to take *actions* in an *environment* so as to maximize some notion of cumulative *reward*. The problem, due to its generality, is studied in many other disciplines, such as [control theory](#), [operations research](#), [information theory](#), [simulation-based optimization](#), [statistics](#), and [Genetic Algorithms](#). In the operations research and control literature the field where reinforcement learning methods are studied is called *approximate dynamic programming*. The problem has been studied in the [theory of optimal control](#), though most studies there are concerned with existence of optimal solutions and their characterization, and not with the learning or approximation aspects. In [economics](#) and [game theory](#), reinforcement learning may be used to explain how equilibrium may arise under [bounded rationality](#).

In machine learning, the environment is typically formulated as a [Markov decision process](#) (MDP), and many reinforcement learning algorithms for this context are highly related to [dynamic programming](#) techniques. The main difference to these classical techniques is that reinforcement learning algorithms do not need the knowledge of the MDP and they target large MDPs where exact methods become infeasible.

Reinforcement learning differs from standard [supervised learning](#) in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected. Further, there is a focus on on-line performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge). The exploration vs. exploitation trade-off in reinforcement learning has been most thoroughly studied through the [multi-armed bandit](#) problem and in finite MDPs.

The basic reinforcement learning model consists of:

1. a set of environment states  $S$ ;
2. a set of actions  $A$ ;
3. rules of transitioning between states;
4. rules that determine the *scalar immediate reward* of a transition; and
5. rules that describe what the agent observes.

The rules are often stochastic. The observation typically involves the scalar immediate reward associated to the last transition. In many works, the agent is also assumed to observe the current environmental state, in which case we talk about *full observability*, whereas in the opposing case we talk about *partial observability*. Sometimes the set of actions available to the agent is restricted (e.g., you cannot spend more money than what you possess).

A reinforcement learning agent interacts with its environment in discrete time steps. At each time  $t$ , the agent receives an observation  $o_t$ , which typically includes the reward  $r_t$ . It then chooses an action  $a_t$  from the set of actions available, which is subsequently sent to the environment. The environment moves to a new state  $s_{t+1}$  and the reward  $r_{t+1}$  associated with the *transition*  $(s_t, a_t, s_{t+1})$  is determined. The goal of a reinforcement learning agent is to collect as much reward as possible. The [agent](#) can choose any action as a function of the history and it can even randomize its action selection.

When the agent's performance is compared to that of an agent which acts optimally from the beginning, the difference in performance gives rise to the notion of *regret*. Note that in order to act near optimally, the agent must reason about the long term consequences of its actions: In order to maximize my future income I better go to school now, although the immediate monetary reward associated with this might be negative.

Thus, reinforcement learning is particularly well suited to problems which include a long-term versus short-term reward trade-off. It has been applied successfully to various problems, including [robot control](#), elevator scheduling, [telecommunications](#), [backgammon](#) and [checkers](#) ([Sutton and Barto 1998](#), Chapter 11).

Two components make reinforcement learning powerful: The use of samples to optimize performance and the use of function approximation to deal with large environments. Thanks to these two key components, reinforcement learning can be used in large environments in any of the following situations:

- A model of the environment is known, but an analytic solution is not available;
- Only a simulation model of the environment is given (the subject of [simulation-based optimization](#));
- The only way to collect information about the environment is by interacting with it.

# MDPs in the Real World

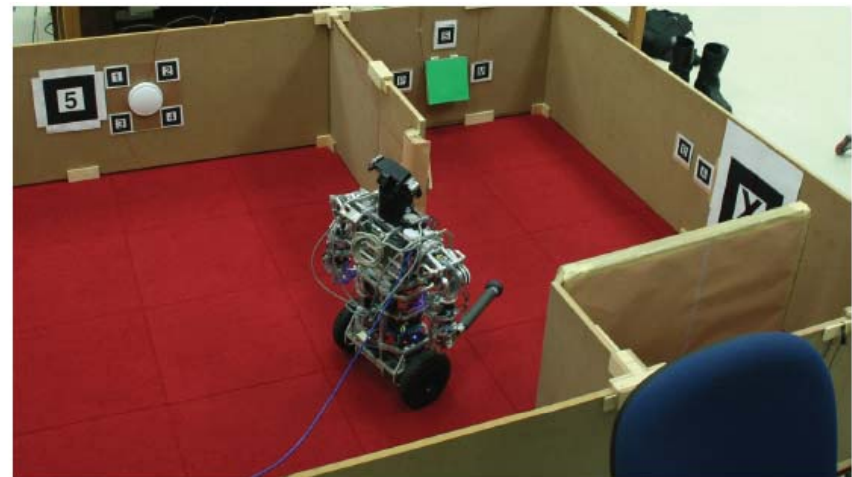
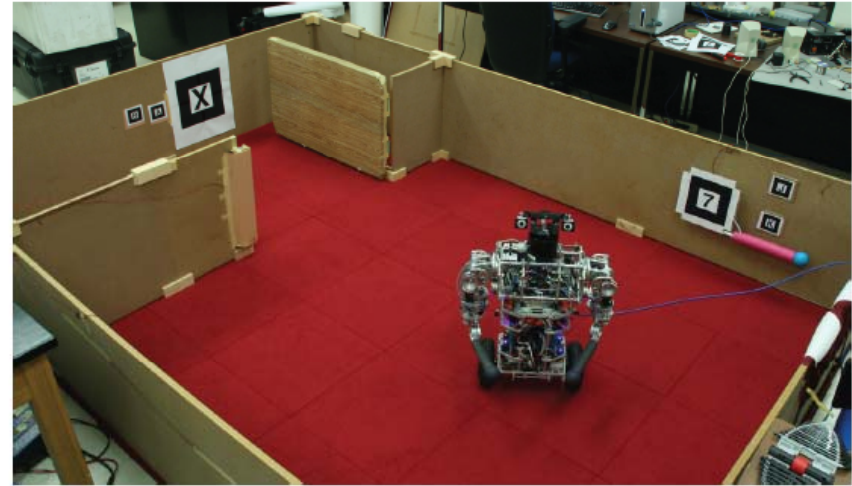
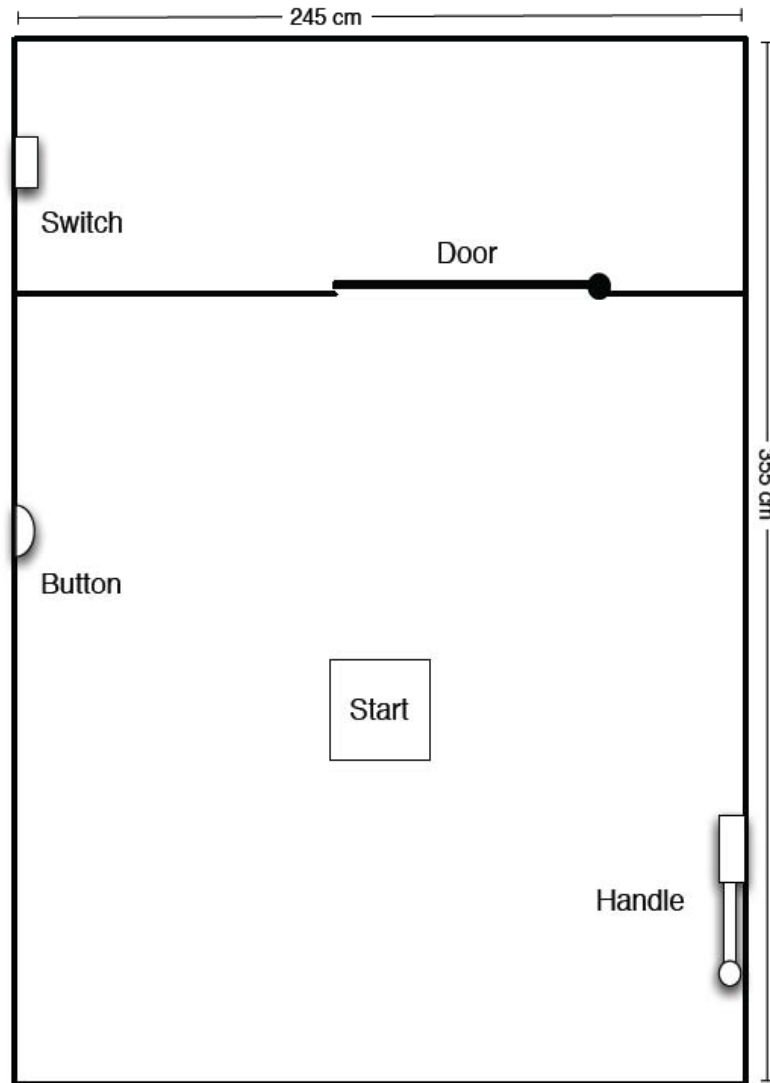


Fig. 2. The first task in the Red Room Domain.

# MDPs in the Real World

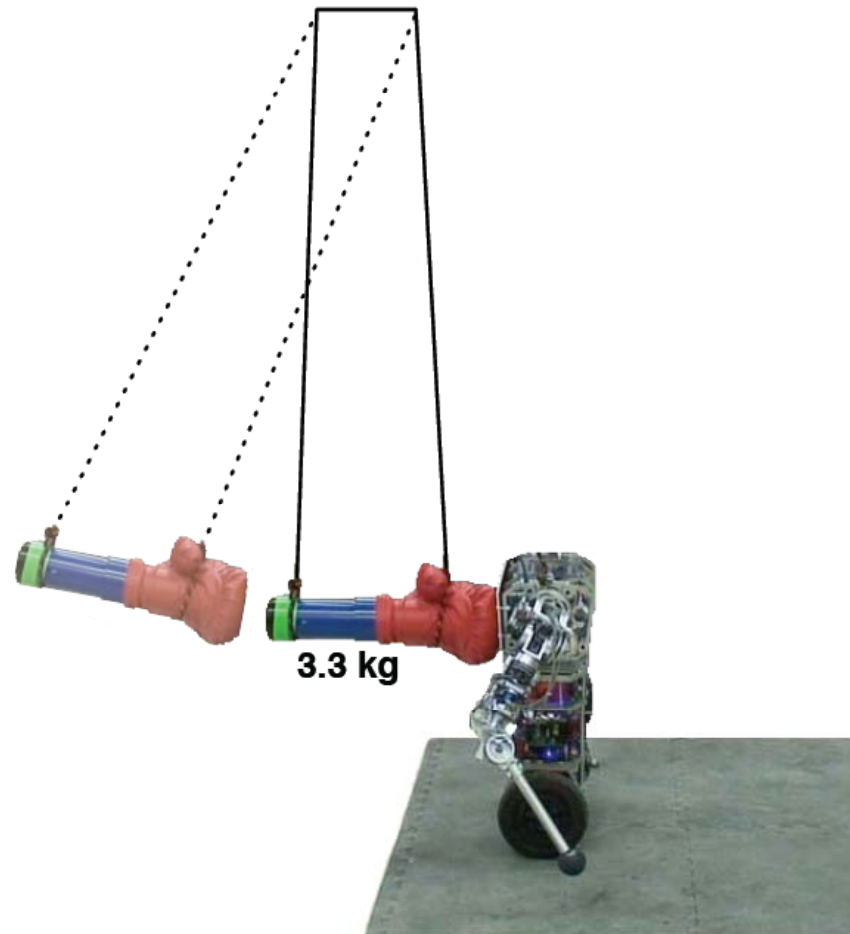
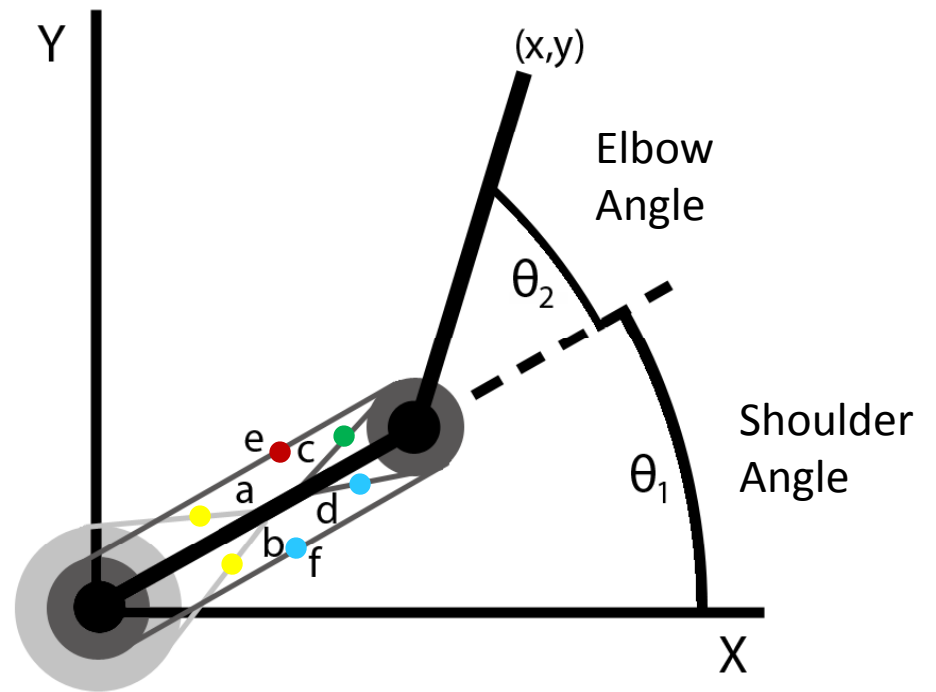
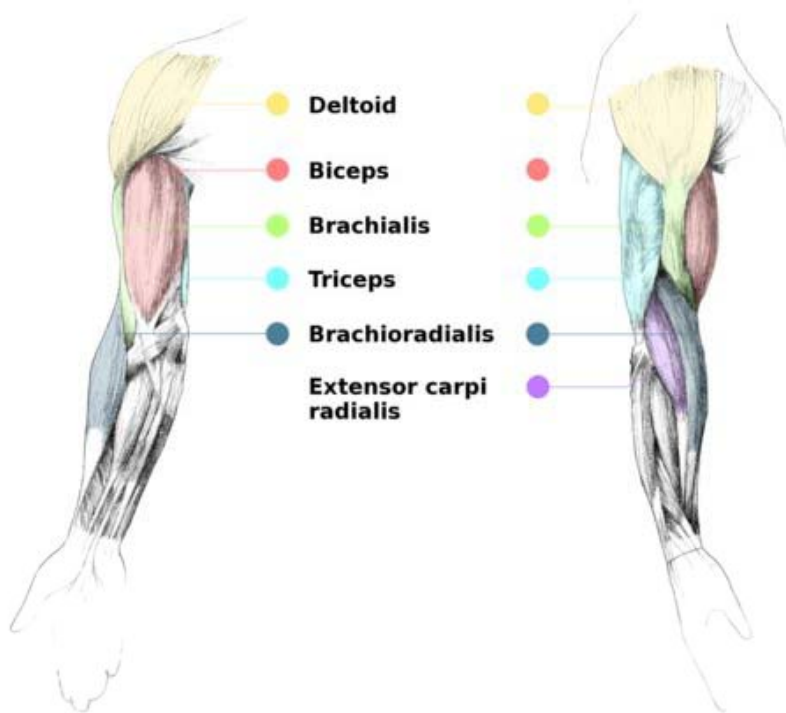


Fig. 1: The uBot-5 situated in the impact pendulum apparatus.

# MDPs in the Real World

- <http://heli.stanford.edu/>

# MDPs in the Real World



# MDPs in the Real World

Pictures removed for online version

Optotrak Certus System for detecting arm position

Planar, frictionless movement

3D arm model = 5 degrees of freedom, 102 muscles

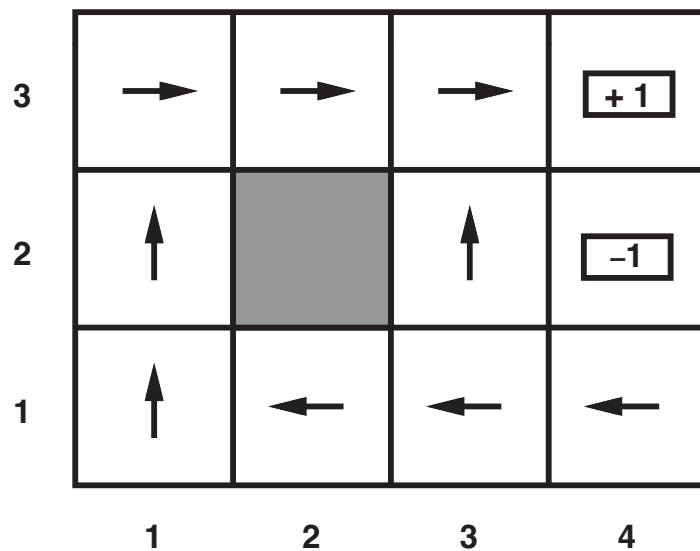
# MDPs in the Real World

- Elevator Scheduling
- DRAM Scheduling
- Propofol Administration
- Operations research
- Games (e.g. Tetris, Mario)
- ...

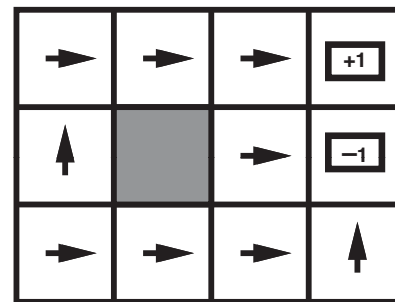


# MDPs-RL-Neuroscience

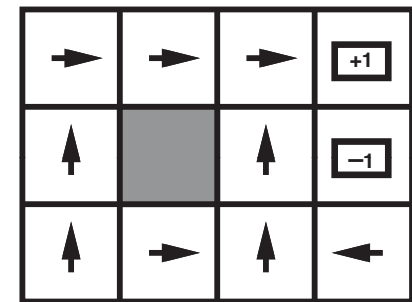
# Back to the book



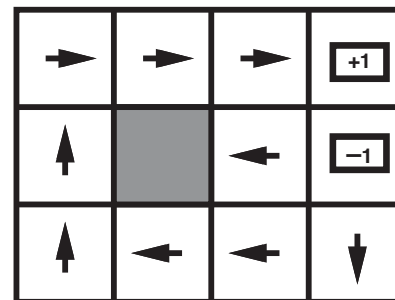
(a)



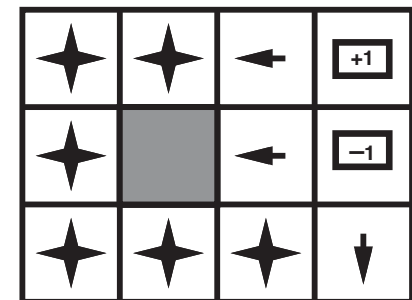
$$R(s) < -1.6284$$



$$-0.4278 < R(s) < -0.0850$$



$$-0.0221 < R(s) < 0$$



$$R(s) > 0$$

(b)

# Finite/Infinite Horizon

- Finite horizon: the “game” ends after  $N$  steps.
- Infinite horizon: the “game” never ends
- “With a finite horizon, the optimal action in a given state could change over time.”
  - The optimal policy is *nonstationary*.
- With infinite horizon, the optimal policy is *stationary*.

# Utilities over time

- Additive rewards:

$$U_h([s_0, s_1, s_2, \dots]) = R(s_0) + R(s_1) + R(s_2) + \dots$$

- Discounted rewards:

$$U_h([s_0, s_1, s_2, \dots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$$

- Discount factor:  $\gamma \in [0, 1]$

# Gamma



0.9^x, x from 0 to 30



[Examples](#) [Random](#)

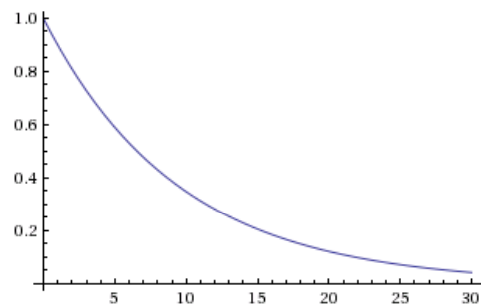
Input interpretation:

plot

0.9<sup>x</sup>

x = 0 to 30

Plot:



Computed by [Wolfram Mathematica](#)

Download as: [PDF](#) | [Live Mathematica](#)



0.95^x, x from 0 to 30



[Examples](#) [Random](#)

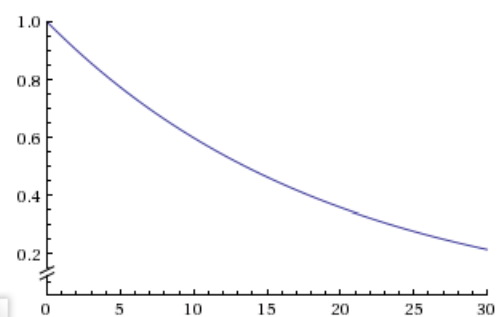
Input interpretation:

plot

0.95<sup>x</sup>

x = 0 to 30

Plot:



Computed by [Wolfram Mathematica](#)

Download as: [PDF](#) | [Live Mathematica](#)

# Discounted Rewards

- Would you rather have a marshmallow now, or two in 20 minutes?
- Infinite sums!

$$U_h([s_0, s_1, s_2, \dots]) = R(s_0) + R(s_1) + R(s_2) + \dots$$

$$U_h([s_0, s_1, s_2, \dots]) = R(s_0) + \gamma R(s_1) + \gamma^2 R(s_2) + \dots$$

# Proper Policy

- A *proper policy* is a policy that is guaranteed to reach a terminal state.
  - Which of the policies for the gridworld are proper?

# Average Reward

$$U_h([s_0, s_1, s_2, \dots]) = R(s_0) + R(s_1) + R(s_2) + \dots$$

- What if we took the average?

$$U_h([s_0, s_1, s_2, \dots]) = \frac{1}{n} (R(s_0) + R(s_1) + R(s_2) + \dots)$$

- Can be applied to infinite horizon problems
  - “beyond the scope of this book”



# Utility of States

- Given a policy, we can define the utility of a state:

$$U^{\pi}(s) = E \left[ \sum_{t=0}^{\infty} \gamma^t R(s_t) \right]$$

- Optimal policy (for state  $s$ )

$$\pi_s^* = \arg \max_{\pi} U^{\pi}(s)$$

# Utility and Rewards

- Shorthand  $U^{\pi^*}(s) = U(s)$
- Utility = long term total reward from s onwards
- Reward = short term reward from s

# Utility

3	0.812	0.868	0.918	<div>+ 1</div>
2	0.762		0.660	<div>- 1</div>
1	0.705	0.655	0.611	0.388
	1	2	3	4

# Optimal Policy

- Optimal policy doesn't depend on what state you start in:

$$\pi_s^* = \arg \max_{\pi} U^{\pi}(s)$$

$$\pi^*(s) = \arg \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')$$

# Searching for Optimal Policies

- Bellman Equation

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')$$

- If we write out the bellman equation for all  $n$  states, we get  $n$  equations, with  $n$  unknowns:  $U(s)$ .
- We can solve this system of equations to determine the Utility of every state.

# Value Iteration

- The equations are non-linear, so we can't use standard linear algebra methods.

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U(s')$$

- Value iteration: start with random initial values for each  $U(s)$ , iteratively update each value to fit the right-hand side of the equation:

$$U_{i+1} \leftarrow R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s' | s, a) U_i(s')$$

# Value Iteration

- The update is applied simultaneously to every state.
- If this update is applied infinitely often, we are guaranteed to find the true  $U(s)$  values.
  - There is one unique solution
- Given the true  $U(s)$  values, how can we select actions? (Maximum expected utility – MEU)

$$a_t = \arg \max_{a \in A(s)} \sum_{s'} P(s' | s_t, a) U_i(s')$$

# Question

- If the value function is approximate (has error), can you still get an optimal policy?

$$a_t = \arg \max_{a \in A(s)} \sum_{s'} P(s' | s_t, a) U_i(s')$$



# Policy Iteration

- If one action is clearly better than all others, then the exact magnitude of state utilities need not be precise.
- Policy iteration interleaves two steps:
  - Policy evaluation: Given a policy, compute the utility of each state for that policy
  - Policy improvement: Calculate a new MEU policy
- Terminate when the policy doesn't change the utilities.
- Guaranteed to converge to an optimal policy

# Policy Iteration

- In what ways is it better?
- Policy evaluation is for a fixed policy – no arg max. We don't need to solve a set of nonlinear equations.
- The Bellman equation simplifies to:

$$U(s) = R(s) + \gamma \sum_{s'} P(s' | s, \pi(s)) U(s')$$

- These equations are linear (the max has been removed)

# Policy Iteration

- The  $n$  linear equations can be solved in  $O(n^3)$  with standard linear algebra methods.
- So, the benefit of policy iteration is that the policy evaluation step is much easier (and policy improvement is trivial).
- If  $O(n^3)$  is still too much, we can perform a few of the simplified value-iteration steps:

$$U_{i+1} \leftarrow R(s) + \gamma \sum_{s'} P(s' | s, \pi(s)) U_i(s')$$

# Asynchronous Policy Iteration

- We said the utility of every state is updated simultaneously. This isn't necessary.
- You can pick a subset of the states and apply either policy improvement or value iteration to that subset.
- Given certain conditions, this is also guaranteed to converge to an optimal policy.