Reinforcement Learning

<ins>**In this lecture. . .**</ins>

- (Some of) The mathematical foundations of reinforcement learning

- Valuing states and actions

- Finding the optimal strategy in a game

Introduction

Reinforcement learning is one of the main types of machine learning. Using a system of rewards and punishments an algorithm learns how to act or behave. It might learn to play a game or move around an environment.

The key to reinforcement learning is that **the game or environment is not explicitly programmed**. There is no rule book to which we can refer.

This means that the algorithm has to learn the consequences of taking actions from taking those actions.

The algorithm learns by trial and error.

What Is It Used For?

Reinforcement learning is used for

- Learning how to interact with an environment especially when the environment or the result of interactions aren't known in advance

- Example: Learn how to optimally bid at an auction

- Example: Learn which adverts to present to individual consumers

- Example: Learn how to play video games

While unsupervised learning is about finding relationships in data, and supervised learning is about deciding what something is, reinforcement learning is about teaching the machine to *do* something.

And it really is inspired by behavioural psychology. Sit, down, fetch, roll over, are all commands that with the right sort of reinforcement any child will eventually understand. We want an algorithm to learn what actions to take so as to maximize some reward (and/or minimize punishment).

Because you want the machine to learn to attain some goal it is very common to see the method used for playing games. And our examples here will also be from, or related to, games.

Trial and Error

On one hand you want the machine to take advantage of/exploit everything it has learned in order to win the game, say, but on the other hand it won't learn anything unless it has done plenty of exploration beforehand.

Getting a balance between exploiting and exploring will be important to our learning algorithms.

The structure of this lecture is to start with simple motivating examples. Then I move on to some elegant mathematics.

Although you will be seeing some rigorous mathematics in the simple examples when it comes to reinforcement learning proper we won't necessarily have that much in the way of rigorous underpinnings, depending on the problem we are looking at.

So to me this is a slightly strange topic. We first make a cake putting all the ingredients into a cake tray. We successfully make a tasty cake. Ok, now let's try to make the cake with the same ingredients but without the tray.

Jargon

I'm going to explain some preliminary, basic, jargon while referring to a few common games.

- **Action:** What are the decisions you can make?

  Which one-armed bandit do you choose in a casino? Where do you put your cross in the game of Noughts and Crosses? How many, and which, cards do you exchange in a game of draw poker?

  Those are all example of actions.

  Blackjack! Take a card or not?

- **Reward/Punishment:** You take an action and you might get rewarded.

  You press a button on the wall in Doom and the BFG is revealed.

  You take your opponent's piece in checkers. It's white chocolate and you eat it.

  Those are examples of immediate rewards. But there might not be any reward until the end of the game.

  At the end of the chess game the winner gets the $1,000 prize.

  But there aren't just rewards. To win the prize you have to be the first to solve the jigsaw puzzle. Every second you take can be thought of as a punishment.

  Blackjack! $$$

- **State:** The state is a representation of how the game is now. Think of it as a snapshot of the gameboard, for example.

  E.g. the positions of the Os and Xs in a game of O&Xs. The state is an interesting concept. How much information is needed to represent a state?

  Sometimes the amount of information you must store is large. In Go there are typically 361 points, each of which could be empty, or be occupied by a white or black stone.

  The state might not be represented by discrete quantities. At what angle should you kick the ball when taking a penalty?

  And for reinforcement learning proper we won't even know what information represents the state.

  Blackjack! What card is the dealer showing and what do your cards add up to?

- **Markov Decision Process (MDP):** Markovian means that what happens going forward only depends on the current state. Some of the justification for Reinforcement Learning comes from the mathematics of MDPs.

  Markov refers to there being no memory if your state includes enough information. So that is part of the trick of learning how to play any game. Keep track of as many variables as needed, but no more.

  Blackjack!

- **Value Function:** A value function measures how good or bad a state or an action is. If we are in some state now and all future states are good then the value at our present state will be high. But that value depends on what actions we take now and in the future.

  Value comes from accumulation of rewards, it is how good our current position is given what we do now and in the future.

  Blackjack! You've got 19 and the dealer is showing a six. Yay!

- **Policy:** A policy is a set of rules governing what action to take in each state. That policy might be deterministic. Or it might be random.

  Ultimately in reinforcement learning we want the policy to maximize value at each state. But of course *a priori* we don't know what the best policy is. . . that's what we are trying to get the machine to learn.

  Blackjack! See the tables in books on Blackjack for the optimal policy.

What follows

- Introducing rewards and value — the multi-armed bandit
- Introducing states — a simple maze
- Introducing the action-value function and optimizing — the maze continued

Rewards and Value: The multi-armed bandit

You'll know of the one-armed bandit. It is a gambling machine found in casinos and bars. Originally these machines had a lever, the arm, that you pull, and this causes cylinders, on which there are pictures of various fruit, to spin. (Hence the name Fruit Machine.) If they stop on the right combination of lemons, cherries, etc. then you win a monetary prize.

In the multi-armed bandit problem we have several such bandits each with a different probability of winning a fixed amount, the same amount for each bandit.

The goal of the multi-armed bandit as a problem in reinforcement learning is to choose among the bandits, pull the lever, see if you win, try a different bandit, and by looking at your rewards learn which is the bandit with the best odds.

This problem is quite straightforward. There is first of all no state as such. But there are actions, which bandit to choose. We want to assign a value to each action. And then based on these values decide which action to take.

This is how it goes. . .

- There will be ten bandits. The probabilities of winning for each bandit are

Bandit 1	10%
Bandit 2	50%
Bandit 3	60%
Bandit 4	80%
Bandit 5	10%
Bandit 6	25%
Bandit 7	60%
Bandit 8	45%
Bandit 9	75%
Bandit 10	65%

But we won't be explicitly telling "The Machine" what these probabilities are!

- The value of each action, each bandit, the function Q, will simply be the average reward for that bandit. This average is only updated after each pull of a lever.

  And the average is calculated as the total actual reward so far, using randomly generated success/fail at each pull, divided by the total number of times that bandit has been chosen.

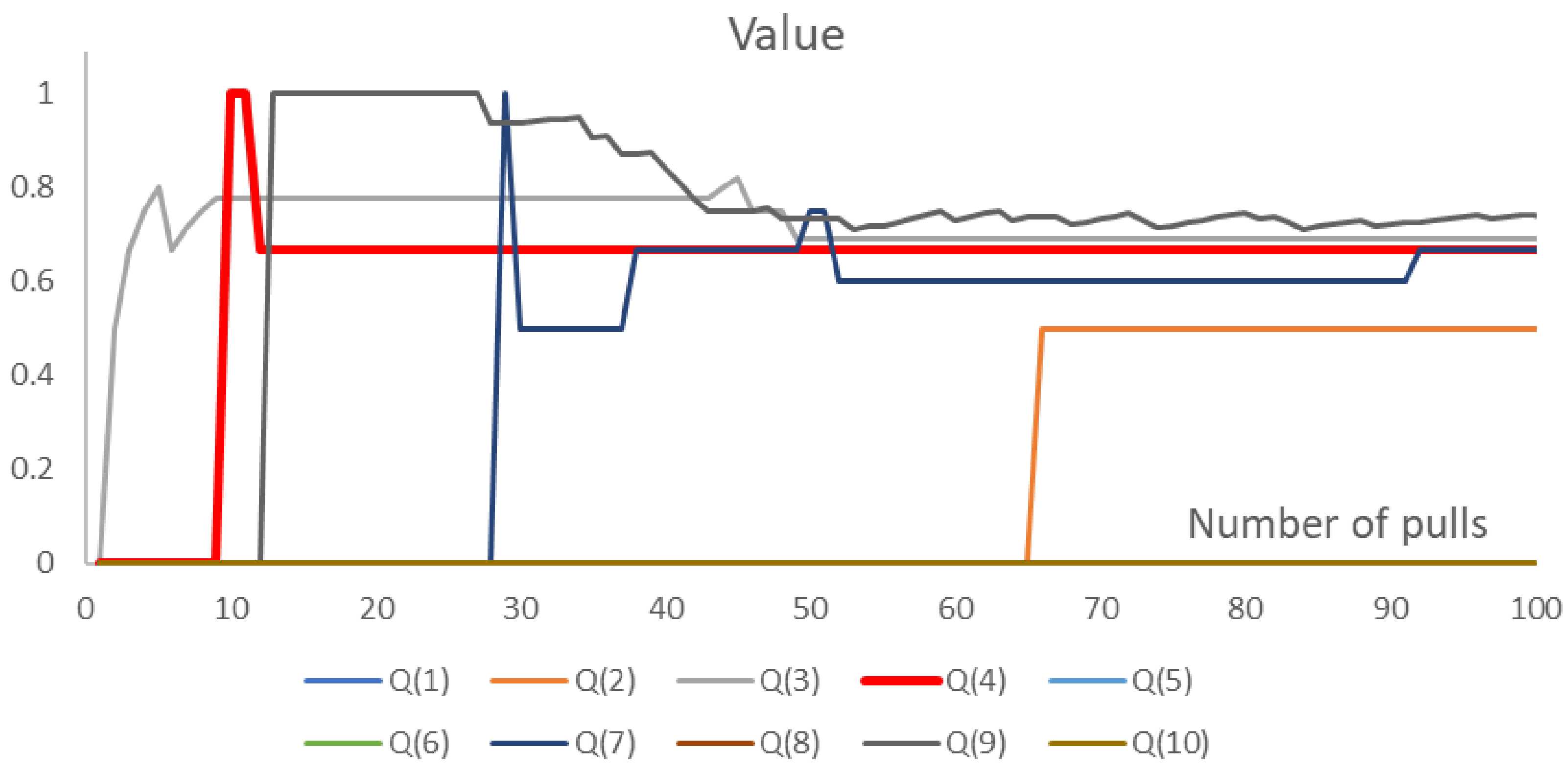
  To all intents and purposes the reinforcement algorithm does not *know* about the odds, only *experiences* them.

- After each pull we have to choose the next bandit to try. **This is done by most of the time choosing the action (the bandit) that has the highest value at that point in time. But every now and then, let's say at a random 10% of the time, we simply choose an action (bandit) at random with all bandits equally likely.**

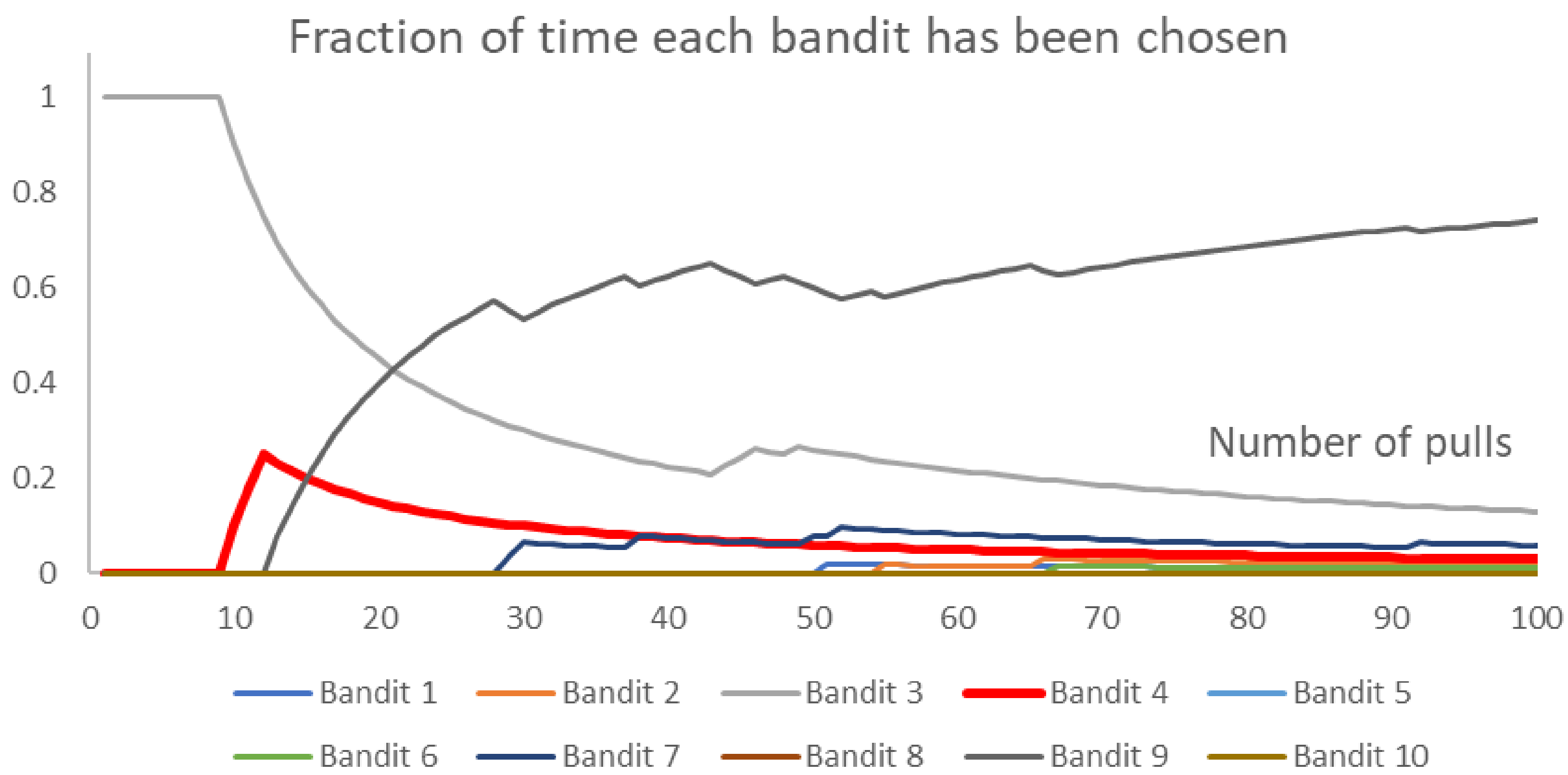
That last bullet point describes what is known as an ϵ-greedy policy.

You choose the best action so far, but you also do occasional exploration in case you haven't yet found the best policy. The random policy happens a fraction, ϵ, of the time.

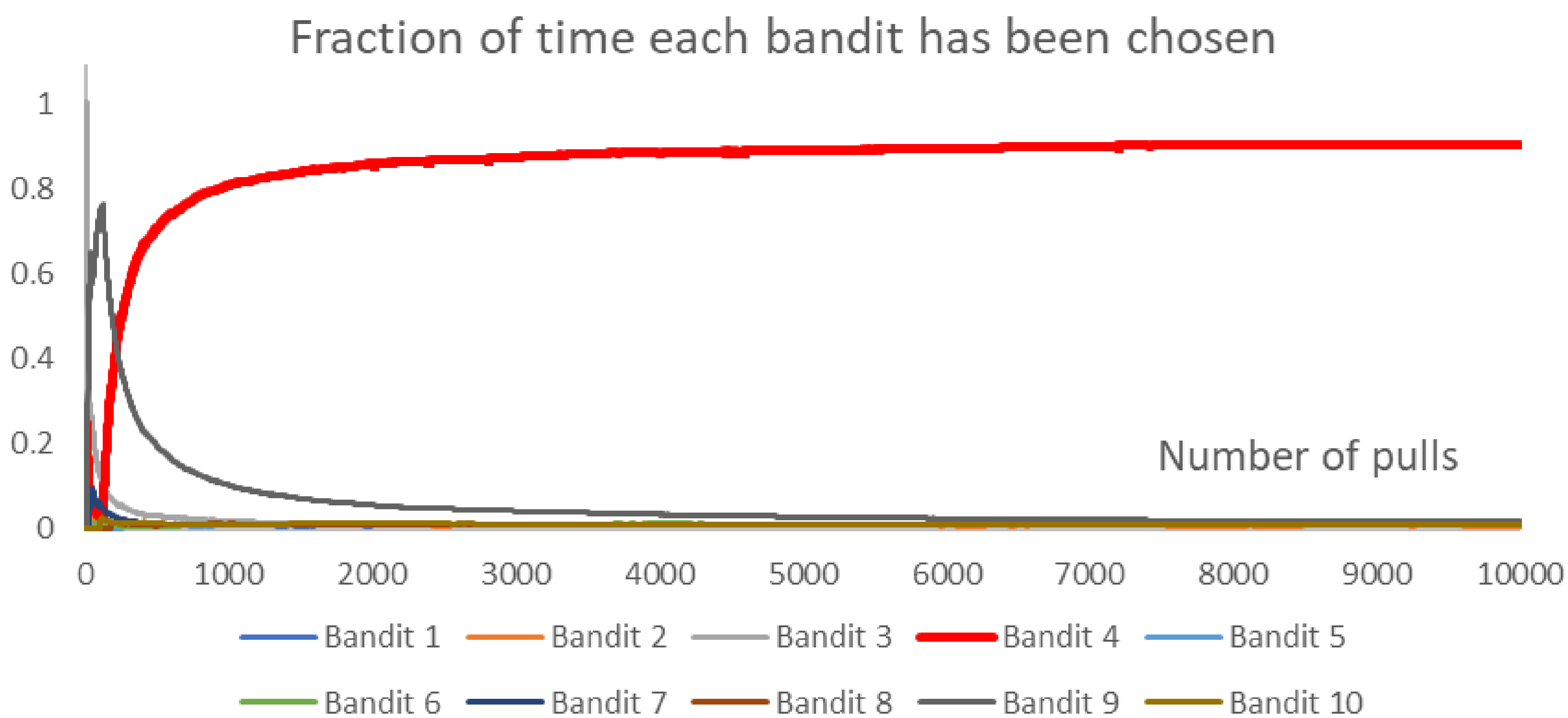
In the figure is shown the value function for each of the ten bandits as a function of the total number of pulls so far. I have used Q to denote this value, we'll see more of Q later.



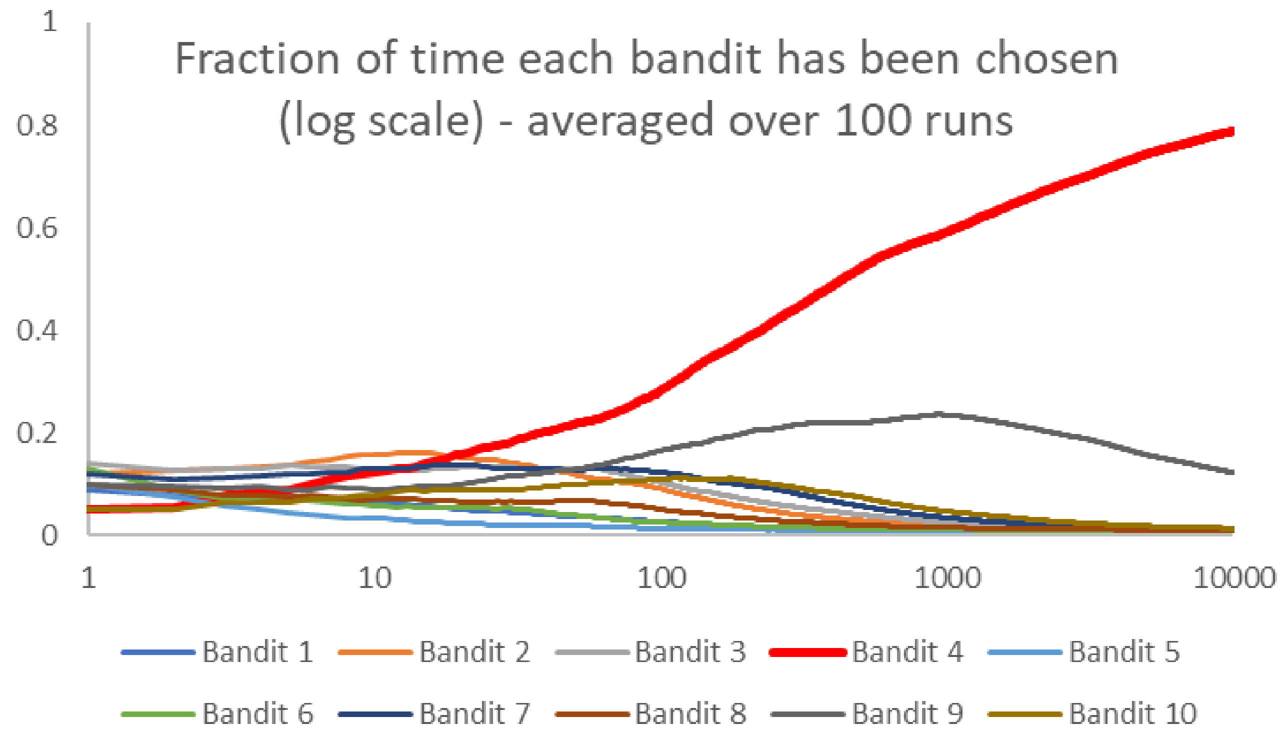
And below we see the fraction of time each bandit has been chosen.



Below is shown the fraction of pulls for each bandit up to 10,000 pulls. Clearly Bandit 4 has become the best choice. And if you look at the table you'll see that it does indeed have the highest probability of success.

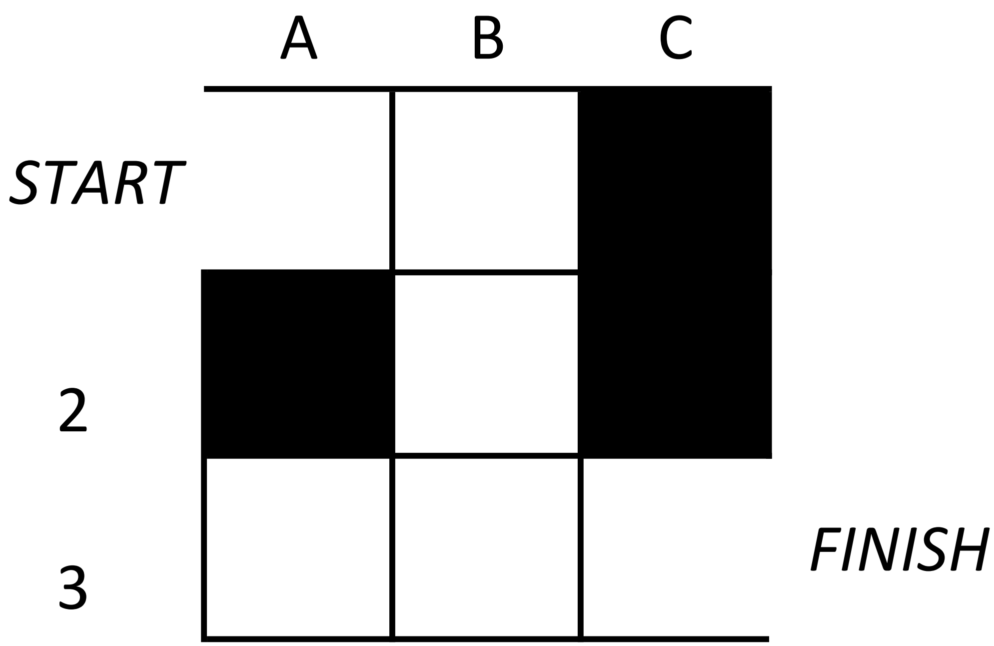


The correct bandit will eventually be chosen however the evolution of the Qs and the fraction of time each bandit is chosen will depend for a while on which bandits are chosen at random. So below I show the results of doing 100 runs of 10,000 pulls each.



States: A Maze

Our next example is a maze. This will be used to illustrate the idea of states.



Start in A3, the finish is cell C3.

It's a simple example because first of all the reward will be -1 for every action, i.e. every step.

Why?

Several steps in the following:

First, choose a **policy** and see how well it does.

This introduces several different types of maths, useful in reinforcement learning.

Second, introduce the **action-value function**.

Then find the **optimal** route out of the maze.

Simple policy to start with: Make a move from one cell to a neighbouring cell equally likely: If there is only one neighbour then the probability of moving to it is 100%, if two neighbours then 50% each, etc. At this point there is nothing special about Cells A1 and C3, other than we can only go in one direction from A1 and we stop when we get to C3, there is no attempt to go from Start to Finish. The transition probabilities tell us where to go with no goal in mind.

		To					
		A1	**A3**	**B1**	**B2**	**B3**	**C3**
From	**A1**	0	0	1	0	0	0
	A3	0	0	0	0	1	0
	B1	0.5	0	0	0.5	0	0
	B2	0	0	0.5	0	0.5	0
	B3	0	0.333	0	0.333	0	0.333
	C3	0	0	0	0	0	1

Solving the Markov-chain problem

We can write down a recursive relationship for the expected reward (negative of the number of steps it will take to get from any cell to the finish).

Denote the expected reward as a function of the state, i.e. the cell we are in, $v(s_t)$.

We have relationships like

$$v(\text{A1}) = -1 + v(\text{B1})$$

because whatever the number of expected steps from B1, the expected number from A1 is one more. Similarly

$$v(\text{A3}) = -1 + v(\text{B3}).$$

From Cell B1 we can go in two directions, both equally likely, and so

$$v(\mathrm{B1}) = -1 + \frac{1}{2}\left(v(\mathrm{A1}) + v(\mathrm{B2})\right).$$

And so on...

$$v(\mathrm{B2}) = -1 + \frac{1}{2}\left(v(\mathrm{B1}) + v(\mathrm{B3})\right),$$

$$v(\mathrm{B3}) = -1 + \frac{1}{3}\left(v(\mathrm{A3}) + v(\mathrm{B2}) + v(\mathrm{C3})\right)$$

(from B3 there are three actions you can take) and finally

$$v(\mathrm{C3}) = 0 + v(\mathrm{C3}).$$

You never leave Cell C3.

These can be written compactly using vector notation.

Let's write v as a vector $\mathbf{v}$ with each entry representing a state, A1, A3, . . . , C3. Similarly we'll write the reward as a vector $\mathbf{r}$.

The final answer for $\mathbf{v}$ will be the negative of the expected number of steps for each state.

So the first five entries in this reward vector, $\mathbf{r}$, will be minus one, and the final entry will be zero. This just means that in going from state to state we get a reward of -1, but since we can't leave cell C3 there is a zero for that entry. And we shall write the transition matrix as $\mathbf{P}$.

Writing the above in vector form to find $\mathbf{v}$ all we have to do is solve

$$\mathbf{v} = \mathbf{r} + \mathbf{P}\mathbf{v}. \tag{1}$$

This tells us the relationship between all the expected values, one for each state.

It is a version of the Bellman equation.

This can be solved by putting the two terms in $\mathbf{v}$ onto one side and inverting a matrix.

However since inverting matrices can be numerically time consuming it is often easier, and certainly will be in high dimensions, to iterate according to

$$\mathbf{v}_{k+1} = \mathbf{r} + \mathbf{P}\mathbf{v}_k.$$

This iterative method of updating a value function inspires several techniques in reinforcement learning.

We can see the iteration process in action below. Here we started with the $\mathbf{v_0}$ having all entries zero.

	A	B	C	
START	0.00	0.00		$k = 0$
2		0.00		
3	0.00	0.00	0.00	*FINISH*

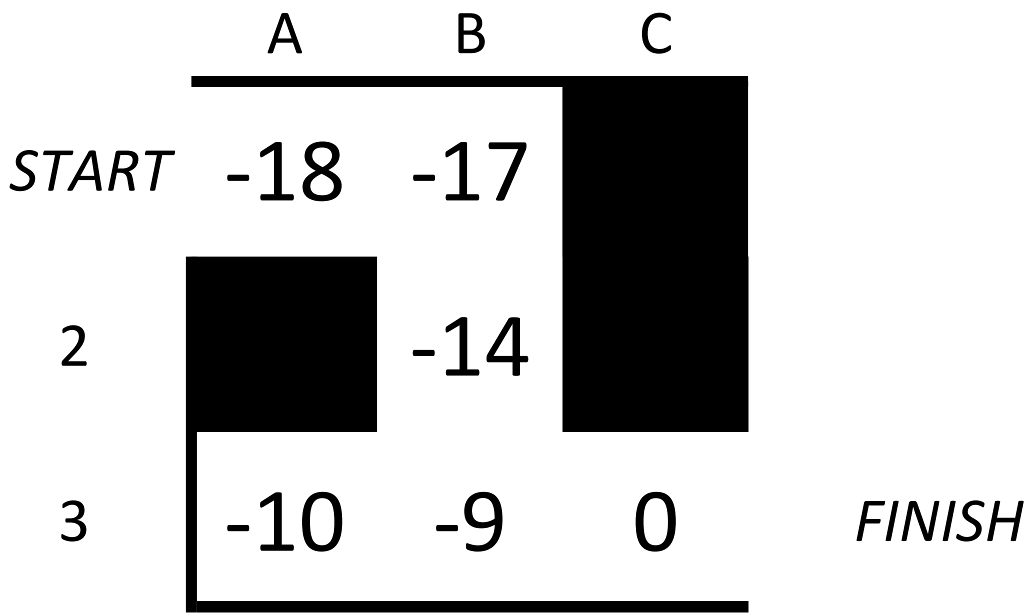
-1.00	-1.00		$k = 1$
	-1.00		
-1.00	-1.00	0.00	

-2.00	-2.00		$k = 2$
	-2.00		
-2.00	-1.67	0.00	

-3.00	-3.00		$k = 3$
	-2.83		
-2.67	-2.33	0.00	

-4.00	-3.92		$k = 4$
	-3.67		
-3.33	-2.83	0.00	

The final result, after many iterations, is



What to take away from this

The maze problem has introduced the ideas of

- Choosing a policy *a priori*

- The value of each state for that policy

- Iterating to find the solution

The maze continued: Optimizing

But this looks to me like it's taking an awfully long time to get from A1 to C3 if you move randomly, 18 steps on average.

Of course, there is a big difference between the above and the optimal solution.

Ultimately our goal in the maze is trying to get from start to finish asap. We want to deduce, rather trivially in this maze, that, for example, the optimal policy when in cell B3 is to move right. That will come soon, after I've introduced some more notation.

Value Notation

State-value function: That's our v above. Given a strategy — above we said we move randomly — each state has a value.

But this doesn't help us find the optimal *strategy*.

Introducing the **action-value function:**

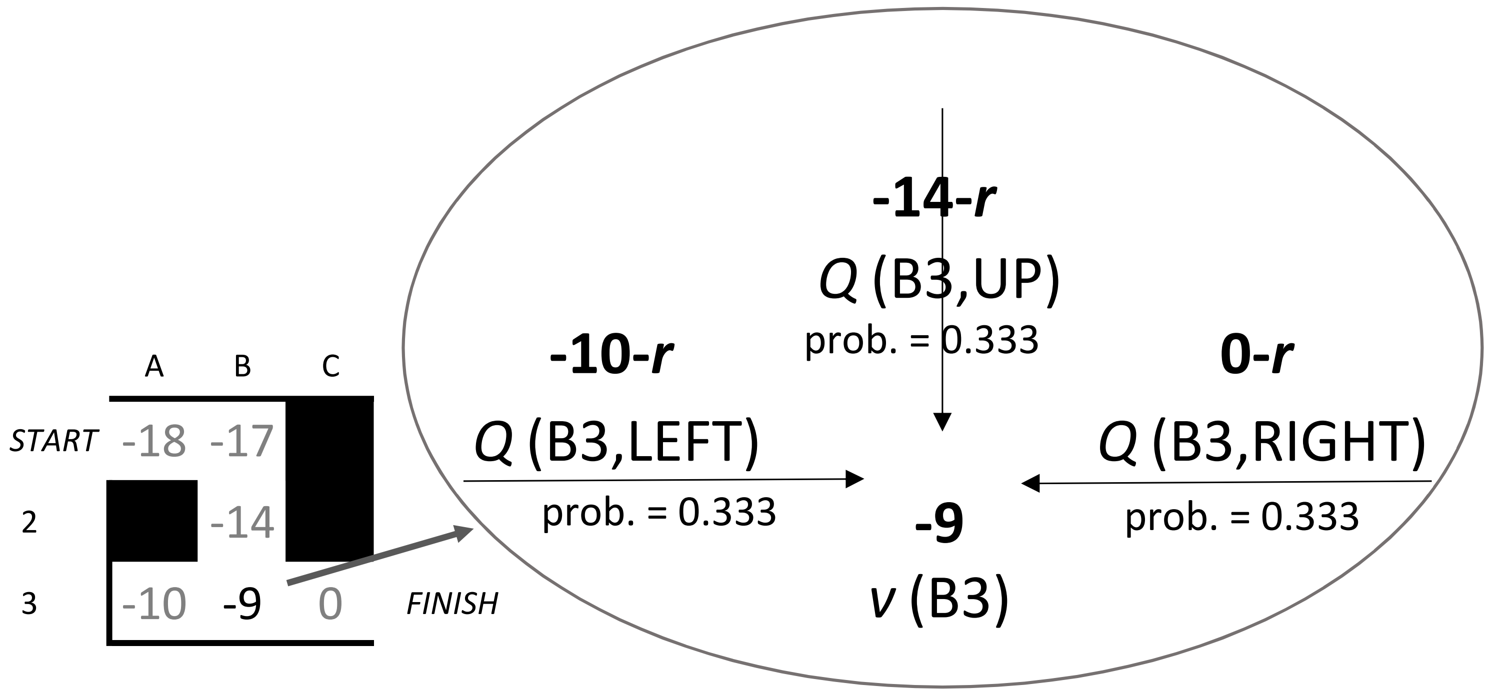
$$Q(\text{state}, \text{action}).$$

This is the quantity that will soon tell us what is the best action to take in each state, the one that maximizes Q.

That's why I used Q for the bandit example. There was no 'state' there but there were 10 possible 'actions.'

This relationship between the two types of value function is shown below for Cell B3 of our maze.

I've written the -1 reward as $-r$ just to help emphasise what the numbers mean, that the reward is added between states, after an action.



Since our policy was originally stochastic with Left, Up and Right actions all having probability one third we have

$$v(\text{B3}) = \frac{1}{3}\left(Q(\text{B3}), \text{LEFT}) + Q(\text{B3}), \text{UP}) + Q(\text{B3}), \text{Right})\right)$$

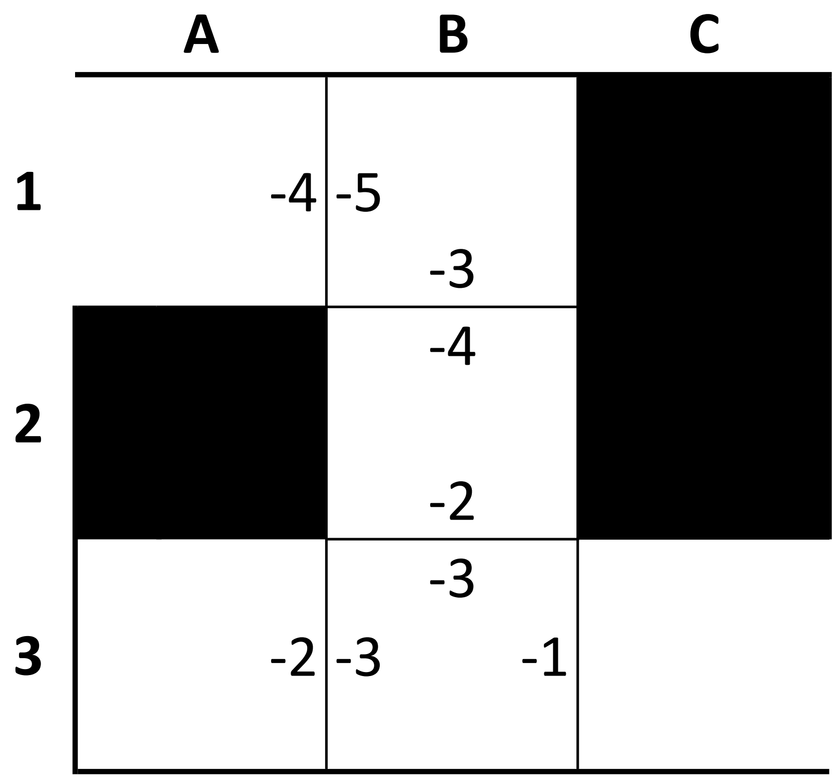
$$= \frac{1}{3}(-10 - r - 14 - r + 0 - r) = -9.$$

You can see the benefits of the action-value function Q from the figure. Out of the three possible actions in state B3 it's moving to the right that has the highest-value Q.

Of course, that's assuming that the state values in the adjacent cells are correct.

You should anticipate an **iterative solution for the Q function**, and thus the fastest route through the maze.

Here is the action-value function Q_* for our maze. That is, the action-value function for the optimal (that's the star *) policy.



Blackjack!

Results for hard hands after a few thousand games are shown below. The results are starting to home in on the classical, recognised optimal strategy for hitting given by the bordered boxes.

Hit or Stand		Dealer									
		2	3	4	5	6	7	8	9	10	A
	12	H	H	H	H	H	H	H	H	H	H
H	13	H	S	S	S	S	H	H	H	H	H
A	14	S	S	S	S	S	H	H	H	H	H
R	15	S	S	S	S	S	H	H	H	H	H
D	16	S	S	S	S	S	H	H	H	H	H
	17	S	S	S	S	S	S	S	S	S	S
	18	S	S	S	S	S	S	S	S	S	S
	19	S	S	S	S	S	S	S	S	S	S
	20	S	S	S	S	S	S	S	S	S	S
	21	S	S	S	S	S	S	S	S	S	S

Summary

Please take away the following important ideas

- There's a lot of theory underpinning reinforcement learning but. . .

- State-value function for valuing a given policy

- You need the action-value function to find the best policy