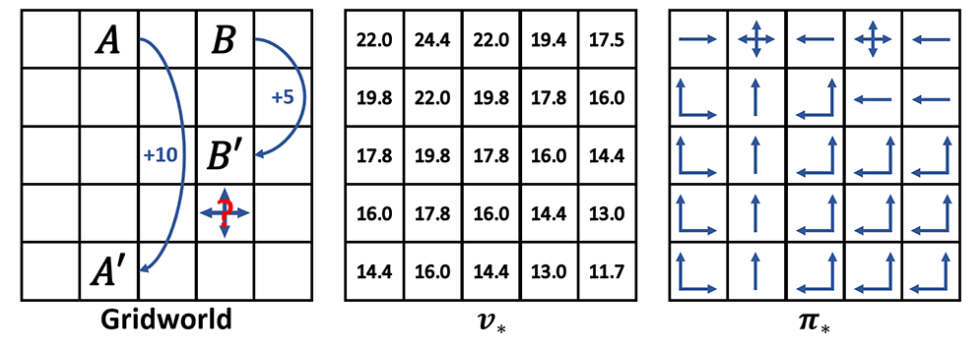
# MDP and Dynamic Programming Methods

## Monte Carlo Control

### Deriving Policies from Value Functions

* In this problem, our agent finds itself in a gridworld. Each grid square is a unique state, and the agent's goal is to maximise the discounted return (assume γ=0.9). In each state, four actions are available: north, south, east, and west.
* These actions are reliable, and cause our agent to move in its intended direction with probability 1.0. Actions which would cause the agent to move into a wall leave its state unchanged, but result in a reward of -1.
* Actions which would cause the agent to move into a wall leave its state unchanged, but result in a reward of −1. Actions which move the agent out of the state marked 'A' cause the agent to move to the state marked 'A′', and result in a reward of +10.
* Actions which move the agent out of the state marked 'B' cause the agent to move to the state marked 'B′', and result in a reward of +5+5.
* Rewards for all other actions are 0.
* The environment is shown in the left panel of the figure below.



Given the optimal state-value function for this problem, shown in the centre panel of the above figure, how might we derive the optimal policy in each state? Let us consider the state below state B′, marked with a question mark in the left figure.

What is the value of the states we can reach?

* If we choose to go north, we will receive an immediate reward of 0 and reach a state with a value of 16.0.
* If we choose to go west, we will receive an immediate reward of 0 and reach a state with a value of 16.0.
* If we choose to go east, we will receive an immediate reward of 0 and reach a state with a value of 13.0.
* If we choose to go south, we will receive an immediate reward of 0 and reach a state with a value of 13.0.

The states that we can reach by going north or west from this state have the highest value, according to the optimal state-value function. This is what we want, because travelling to these states will give us the best expected discounted reward. So, in this state, the optimal policy would be to move either north or west.

What we have done here is act greedily with respect to the optimal state-value function. We've done a one-step look-ahead to all of the states that we can reach from our current state, and chosen action(s) which lead to the highest immediate reward and the next-state(s) with the highest value(s). Applying this greedy one-step look-ahead rule to every state, we can derive the optimal policy for this problem, shown in the right panel of the above figure (optimal actions are represented by directional arrows)

In fact, given the optimal state-value function for any MDP, and the ability to look ahead one step to see what next-states and rewards our actions will lead to, we can derive the optimal policy. If we have access to the optimal action-value function, we do not even need to do a one-step look-ahead, because we would know the value of each of the actions available in the current state, and could just pick the highest-valued action directly.

This is why value functions are so important in reinforcement learning: if we know the optimal value function(s) for a given MDP, we can use them to derive an optimal policy. Because of this, many reinforcement learning algorithms aim to learn value functions, and then use those value functions to derive their policies. We call such methods value-based reinforcement learning methods, and they will make up the vast majority of the methods we will look at during this unit.

### Summary (+what’s on the notebook)

* that **state-value functions** tell us the return we can expect to earn is we start in a given state and then follow a given policy thereafter.
* that **action-value functions** tell us the return we can expect to earn is we start in a given state, take a given action, and then follow a given policy thereafter.
* how we can use value functions to **compare policies** to each other.
* how, for any MDP, there is a set of **optimal policies** which are better than all others.
* how all of the optimal policies for a given MDP share common **optimal state-value functions** and**optimal action-value functions**.
* how we can use optimal value functions to **derive optimal policies**.

## Bellman’s Equations

The Bellman equations lie at the heart of the Dynamic Programming methods that we'll be looking at later this week, and tell us how the value of one state can be defined in terms of the values of its possible successors.

As we have previously discussed, it is possible to derive a recursive definition for the return: the return at time t� can be decomposed into the sum of the immediate reward and the return at time t+1�+1. The **Bellman equation** makes use of this recursive definition, and allows us to derive a relationship between the value of a state and the value of its successor states.

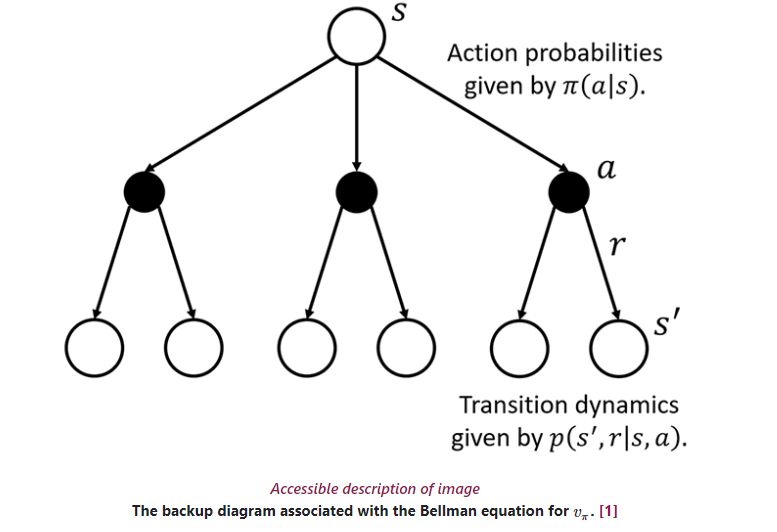
Let us derive the **Bellman equation** for vπ�� from first principles, starting with the definition of vπ�� that we have seen previously:

We can decompose the return at time t�, Gt��, into the immediate reward and the discounted return at time-step t+1�+1:

Let's now expand the expectation. Recall that the expected value of a random variable is the weighted sum of all possible values of that random variable, where the weights are the probabilities of each outcome occurring. In our case, the probability of a given outcome occurring depend on the actions our agent chooses (its policy π(a|s) and the environment's dynamics (the transition function p(s′,r′|s,a).

What we have ended up with is the Bellman equation for vπ. It might look quite complex at first, but the intuition behind it is actually quite simple: *It's the weighted sum of the value of all possible outcomes (decomposed into the immediate reward and the value of the next-state) weighted by the probability of each outcome occurring.*

The backup diagram is shown below, and illustrates that the value of a state s is based on the weighted sum of the immediate rewards r and the value of the next-states s′ that can be reached from it, via actions a.

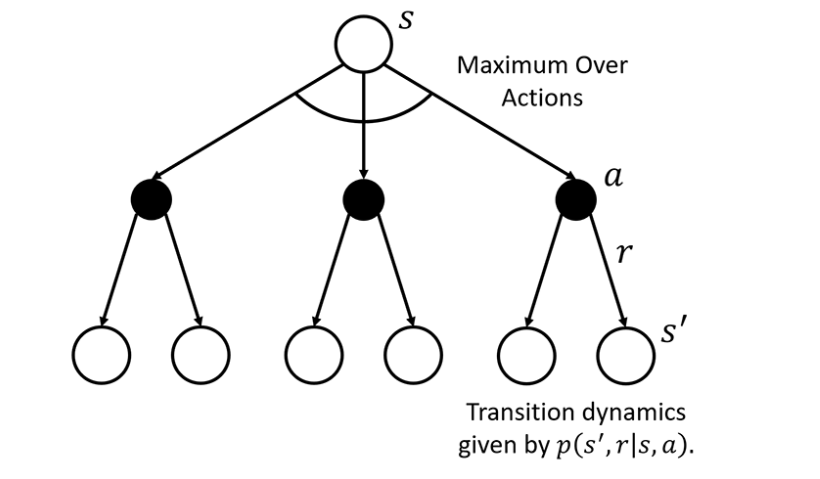
[](#_Bellman’s_Equations)

### Solving the Bellman Optimality Equation

As with any other value function, the optimal value function v∗ must also satisfy this constraint. However, because v∗ is the value function for the optimal policy π∗, we can write the Bellman equation for v∗ in a slightly different way. We call this the **Bellman optimality equation** for v∗:

Superficially, this looks very similar to the Bellman equation (3) that we derived previously. However, because we are now working with the optimal policy, we only consider the action which results in the highest expected return, because this is the action that the optimal policy would choose. Intuitively, the Bellman optimality equation expresses the fact that the value of a state under an optimal policy must equal the expected return of the best action available in that state.

The backup diagram associated with the Bellman optimality equation for v∗ is shown below.



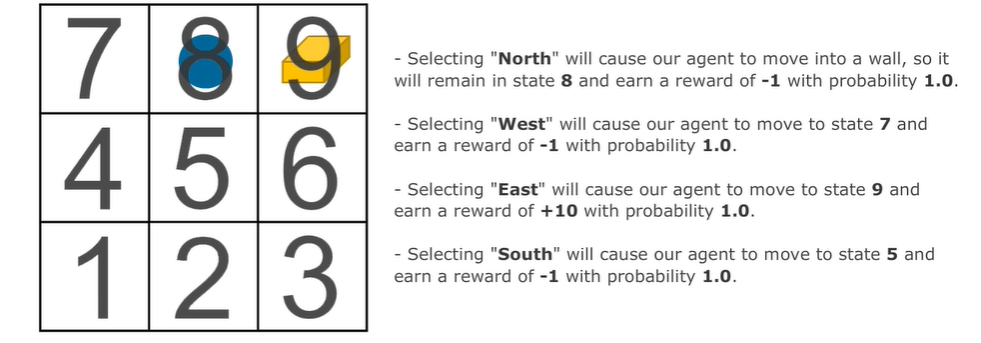
Unfortunately, because the Bellman optimality equation includes a maxmax operator, we're no longer working with a system of linear equations, which makes finding a solution more complicated; in most cases, we will be limited to approximating a solution using iterative methods instead of finding nice, closed-form solutions.

Dynamic Programming methods, which we'll be looking at soon, give us one way of solving the Bellman equations. Most reinforcement learning methods can be understood as finding approximate solutions to the Bellman optimality equation.

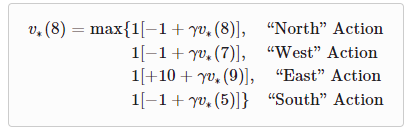
### Example: Grid Gold world

We have nine states (numbered 1 to 9), one corresponding to each grid square, and the four actions 'North', 'South', 'East', and 'West' are available in each state. Our agent starts in state 2, and state 9 is a terminal state. Actions that would move the agent into a wall result in no movement, but time advances as normal.

To keep the maths simpler for this example, we will assume that all actions are reliable: actions succeed as intended with probability 1.0. If our agent transitions to state 9, it earns a reward of +10. Otherwise, it earns a reward of −1.

For this example, we're going to be looking at state 88. Let's first look at the possible next-states and rewards that can be reached from this state.

With this information, we can write out the Bellman optimality equation:



## Dynamic Programming Methods

### Iterative Policy Evaluation

When first introducing a class of solution methods, we will start off by discussing how that class of solution methods treats the problem of computing the value function vπ�� for a given policy vπ��. We call this the **policy evaluation** or **prediction** problem. We're not trying to find an optimal policy yet, we're just trying to find the value function for a given policy.

Recall equation 3, f or an environment with n states, we have a set of n of these equations which we can solve for vπ. DP is a class of iterative methods for solving the Bellman equations, assuming that we have perfect knowledge of the environment's dynamics p(s′,r|s,a).

We can very easily define an iterative update rule based on the Bellman equation for vπ:

where vk is our approximation of vπ after k iterations.

Starting with some initial estimate v0, such as assuming that the value of all states is zero under policy π, we can apply the above update rule again and again in every state s∈S. After each iteration, we'll have a new estimate based on our previous estimate. Eventually, our estimate will converge (i.e., the difference between vk and vk+1 will fall below some threshold Δ) and we'll be left with vk+1≈vπ.

The algorithm that we have just described is called **Iterative Policy Evaluation**, and its pseudocode is presented below.

* Input π, the policy to be evaluated
* Algorithm parameter: a small threshold θ>0 determining accuracy of estimation
* Initialize V(s) arbitrarily, for s∈S, and V( terminal ) to 0
* Loop:
  + Δ←0
  + Loop for each s∈S:
  + v←V(s)
  + V(s)←∑aπ(a∣s)∑s′,rp(s′,r∣s,a)[r+γV(s′)]
  + Δ←max(Δ,|v−V(s)|)
* until Δ<θ(1)

### Policy Improvement

If we have perfect knowledge of the environment's dynamics, there is a relatively simple way of doing this for a given state s. We can calculate the value of taking each of the actions available in s, and change our policy to take the one with the highest value. More formally, we look at every action a∈A(s), and estimate its value:

If it is better to choose action a in s and then follow π thereafter than it is to simply follow π the whole time - that is, that qπ(s,a)>vπ(s) - then it makes sense to choose action a instead of whatever action π is choosing in s. Therefore, the policy π′ which chooses a in s, but which is otherwise identical to π, should be an improvement on π (formally, π′≥π).

Essentially, we perform policy improvement by greedily choosing the actions in each state which lead to the highest immediate rewards and highest-valued next-states. You may have already noticed, but this is very similar to the one-step look-ahead process that we have used previously to derive optimal policies from optimal value functions.

### Policy Iteration

* Initialization:
  + V(s)∈R (real num) and π(s)∈A(s) arbitrarily for all s∈S;V( terminal )≐0
* Policy Evaluation:
  + Loop:
    - Δ←0
    - Loop for each s∈S:
      * v←V(s)
      * V(s)←∑s′,rp(s′,r∣s,π(s))[r+γV(s′)]
      * Δ←max(Δ,|v−V(s)|)
  + until Δ<θ (a small positive number determining the accuracy of estimation)
* Policy Improvement:
  + policy-stable ← true
  + For each s∈S:
    - old-action ←π(s)
    - π(s)←argmaxa∑s′,rp(s′,r∣s,a)[r+γV(s′)]
    - If old-action ≠π(s), then policy-stable ← false
  + If policy-stable, then stop and return V≈v∗ and π≈π∗; else go to 2

### Value Iteration

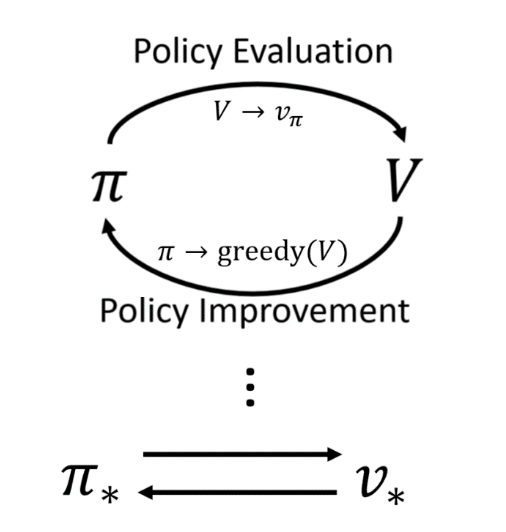
The **policy iteration** algorithm is a fantastic algorithm which, given some arbitrary initial policy π0, allows us to find iteratively better policies until we are left with the optimal policy π∗. However, it is far from perfect. We will now look at **Value Iteration**, another DP algorithm that uses many of the same ideas that led us to policy iteration, but which is much simpler.

* Algorithm parameter: a small threshold θ>0 determining accuracy of estimation  Initialize V(s), for all s∈S+,arbitrarily except that V( terminal )=0
* Loop:
  + Δ←0
  + Loop for each s∈S:
  + v←V(s)
  + V(s)←maxa∑s′,rp(s′,r∣s,a)[r+γV(s′)]
  + Δ←max(Δ,|v−V(s)|)
* until Δ<θ
* Output a deterministic policy, π≈π∗, such that:
  + π(s)=argmaxa∑s′,rp(s′,r∣s,a)[r+γV(s′)]

### Generalised Policy Iteration:

We're going to finish this lesson by introducing an idea that has been present in Policy Iteration and Value Iteration, but that we haven't explicitly discussed in its own right. This is the concept of **Generalised Policy Iteration (GPI)**, a term that is used to refer to any interaction of a policy evaluation process and a policy improvement process regardless of the granularity of either step.

We start off with an arbitrary policy and derive an estimate for its value function. Then, we use this value function estimate to derive a better policy. If we repeat this process enough times, we are guaranteed to converge to the optimal policy and value function.



The goal of GPI is to find an optimal policy π∗, a problem we will come to refer to as the **control problem.**

Clearly, policy iteration is an instance of GPI. It is a DP method that runs iterative policy evaluation until convergence in order to find an estimate of its current policy's value function, and then derives an improved policy by acting greedily with respect to that value function estimate. Likewise, value iteration is also an instance of GPI; the two processes are still present, but value iteration only performs one step of policy evaluation before choosing an action greedily.

As we will come to see throughout this course, almost all RL algorithms can be seen as instances of GPI. This is why, as we move onto other classes of solution methods, we will often start by looking at how to solve the policy evaluation/prediction problem: it is an essential first step in this GPI process.

### Summary

We have discussed:

* how we can use **iterative policy evaluation** to solve the**policy evaluation** problem of computing the state-value function vπ�� for a given policy π.
* how we can use **policy improvement** to derive a policy π′ which is an improvement over another policy π given its state-value function vπ.
* how we can combine these policy evaluation and policy improvement steps to create a single DP algorithm, **policy iteration**, for finding the optimal policy π∗.
* how **value iteration** simplifies policy iteration by combining policy evaluation and policy improvement into a single step.
* that any interaction between a policy evaluation process and a policy improvement process is known as **generalised policy iteration (GPI)**.
* that the goal of generalised policy iteration is to find an optimal policy, a problem known as the **control problem**.
* how most reinforcement learning algorithms can be described as instances of generalised policy iteration.

## Summary

We started by formally defining our agent's interaction with its environment as a Markov Decision Process (MDP). To define a sequential decision problem as an MDP, we need to define six things: a set of states, a set of actions, a transition function, a reward function, an initial state distribution, and a discount factor.

We then introduced value functions, useful functions which tell our agent how much future reward it can expect to earn after arriving in a given state or after taking a given action in a given state, while following a given policy. We discussed how these value functions can be used to compare policies and, crucially, how they can be used to derive them.

Based on these value functions, we introduced the Bellman equations. These equations define important recursive relationships between the values of states/state-action pairs and their successors. Solving the Bellman equation for vπ gives us the state-value function for a given policy π, which we can use to perform policy improvement. Solving the Bellman optimality equation for v∗ gives us the optimal state-value function, from which we can derive an optimal policy. These Bellman equations form the basis of the updated rules seen in many reinforcement learning methods.

Finally, we looked at our first concrete class of solution methods: **dynamic programming methods**. These methods use iterative update rules derived from the Bellman equations to solve sequential decision problems. We discussed how dynamic programming methods rely on having access to a perfect model of our agent's environment, and use bootstrapping (i.e., they base their value estimates on previous value estimates). We examined the iterative policy evaluation algorithm, which solves the policy evaluation/prediction problem, and the policy iteration and policy iteration algorithms, which solve the control problem.

One key limitation of Dynamic Programming methods is that they assume access to a perfect model of our agent's environment. Unfortunately, this assumption is often unrealistic. Next week, we will introduce a second family of solution methods that allow our agent to learn on-line using only an agent's experience, without assuming any prior knowledge about its environment whatsoever.

# Monte Carlo (MC) Methods

**Monte Carlo (MC)** methods are a class of solution methods that allow us to solve sequential decision problems without having any prior knowledge about our agent's environment. Instead of relying on a known world model, as DP methods do, MC methods rely on experience generated by our agent in its environment.

## Monte Carlo Predictions

### Monte Carlo Policy Evaluation

Let's imagine that have an agent following some arbitrary policy π in an environment whose transition dynamics we do not know (i.e., we cannot apply DP methods). We want to find the value function vπ for our agent's policy in this environment.

Recall that the value function vπ tells us the expected return that the agent would obtain if it started in a state s and selected actions according to π thereafter. To estimate the value of a state s under a policy π, an MC method would first generate many episodes of experience, with our agent selecting actions according to π. We then look through all of the episodes, average all of the returns observed after our agent visits state s, and use this average as an estimate of that state's value.

This makes sense, because we are literally averaging returns generated after our agent visited state s and followed π thereafter. This idea of collecting and averaging **sample returns**lies at the heart of all MC methods.

**Based on this, a very high-level MC algorithm for performing policy evaluation would work to:**

* simulate many episodes of interaction with an environment.
* average the sample returns earned after visiting each state

How would this look in a little more detail? Following the high-level algorithm discussed above, we'd start our agent in state some initial state and let it select actions according to π� until it reaches a terminal state. This would give us a single sample trajectory:

For each state that our agent visited along this trajectory, we'd compute the discounted sum of rewards observed after our agent's visit to that state. This discounted sum of rewards is known as a sample return, and it would form our estimate of the value of that state under policy π.

We can repeat this process many times. Each time, our agent would generate another episode of experience, from which we could compute new sample returns for each state. Our new estimates for the values of each state would be the average of all of the sample returns we'd computed for that state so far.

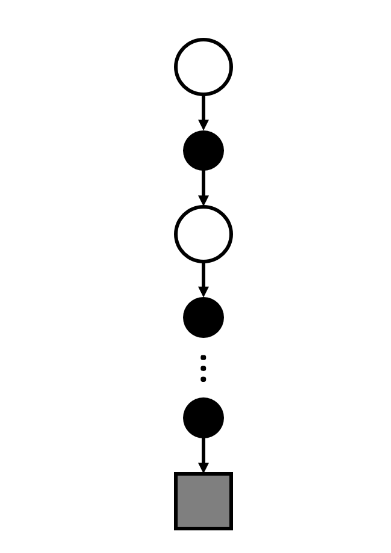
Unless both our agent's policy and environment are deterministic, it is likely that each episode of experience that our agent generates will be different. These differences might be caused by many things:

* Our agent might start each episode in a different initial state.
* Stochasticity in the environment might cause taking the same action in the same state to lead to different outcomes in different episodes.
* Stochasticity in our agent's policy itself might cause it to choose different actions in the same state in different episodes.

Differences between episodes will likely mean differences between the sample returns computed for each state. However, after enough repeated sampling, the average of all of the sample returns that our agent generates should end up being a good approximation of the true value function.

An expanded example, with some sample results, can be found in the course textbook as part of this week's required reading. It discusses using MC methods to perform policy evaluation in the card game of blackjack.

### Comparing Monte Carlo and Dynamic Programming Methods



Comparing that with the [figure](#_Bellman’s_Equations) for DP, we can clearly notice that there is a clear difference between MC and the DP backup diagram. The first obvious difference is that, instead of taking into account all the possible actions (and resulting rewards and next-states) available in a given state, MC methods sample only a single action in a given state. The second clear difference is that, instead of looking forward only one time-step and then bootstrapping off of the values of a state's immediate successors, MC methods continue sampling actions until a terminal state is reached, and base their value estimates on the rewards earned along the way. We can say that MC methods **sample** but, unlike DP methods, that they **don't bootstrap**.

MC methods generate full episodes of experience following a policy π, compute sample returns using the rewards observed in those trajectories, and base their updates solely on those sample returns