AI Planning for Autonomy

8-9. Markov Decision Processes (MDPs)

What about actions with uncertain outcomes?

Tim Miller

THE UNIVERSITY OF MELBOURNE

Agenda

Agenda

Relevant Reading

- *Any* introduction to probability theory — see the related reading on the LMS if you are unfamiliar.
- Chapter 6 of *A Concise Introduction to Models and Methods of Automated Planning* by Geffner and Bonet. Available from the LMS.
- Chapter 17 of *Artificial Intelligence — A Modern Approach* by Russell and Norvig. Available in the university library and online in PDF format.

Removing Assumptions

Classical Planning

So far, we have looked at classical planning. Classical planning tools can produce solutions quickly in large search spaces; but assume:

- Deterministic events
- Environments change only as the result of an action
- Perfect knowledge (omniscience)
- Single actor (omnipotence)

Throughout the remainder of this subject, we are going to look at how to relax a few of these assumptions, starting with the first: deterministic events.

Markov Decision Processes

Markov Decision Processes (MDPs) remove the assumption of deterministic events and instead assume that each action could have multiple outcomes, with each outcome associated with a probability.

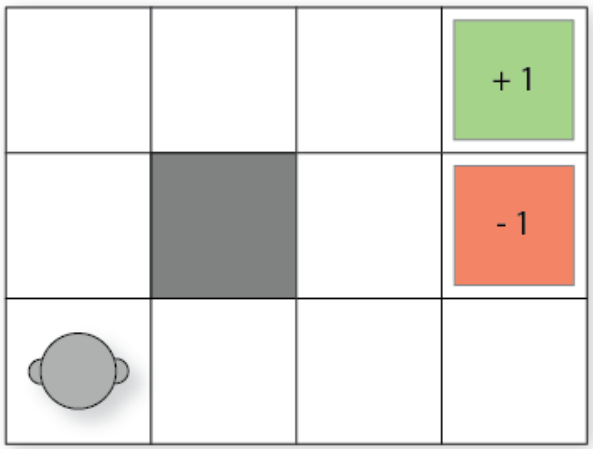
For example:

- Flipping a coin has two outcomes: heads ($\frac{1}{2}$) and tails ($\frac{1}{2}$)
- Rolling two dices together has twelve outcomes: 2 ($\frac{1}{36}$), 3 ($\frac{1}{18}$), 4 ($\frac{3}{36}$), . . . , 12 ($\frac{1}{36}$)
- When trying to pick up an object with a robot arm, there could be two outcomes: successful ($\frac{4}{5}$) and unsuccessful ($\frac{1}{5}$)

MDPs have been successfully applied to planning in many domains: robot navigation, planning which areas of a mine to dig for minerals, treatment for patients, maintenance scheduling on vehicles, and many others.

Canonical example: Grid World

Our agent is in the bottom left cell of a grid. The grey square is a wall. The two labelled cells give a *reward*: 1 for reaching the top-right cell, but a negative reward of -1 for the cell immediately below.



But! Things can go wrong — sometimes the effects of the actions are not what we want:

- If the agent tries to move north, 80% of the time, this works as planned (provided the wall is not in the way)
- 10% of the time, trying to move north takes the agent west (provided the wall is not in the way);
- 10% of the time, trying to move north takes the agent east (provided the wall is not in the way)
- If the wall is in the way of the cell that would have been taken, the agent stays put.

Agenda

Discounted Reward Markov Decision Processes

MDPs are **fully observable, probabilistic** state models. The most common formulation of MDPs is a *Discounted-Reward* Markov Decision Process:

- a state space S
- initial state $s_0 \in S$
- actions $A(s) \subseteq A$ applicable in each state $s \in S$
- **transition probabilities** $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$
- **rewards** $r(s, a, s')$ positive or negative of transitioning from state s to state s' using action a
- a **discount factor** $0 \leq \gamma \leq 1$

What is different from classical planning? Four things:

- The transition function is no longer deterministic. Each action has a probability of $P_a(s'|s)$ of ending in state s' if a is executed in the state s.
- There are no goals. Each action receives a reward when applied. The value of the reward is dependent on the state in which it is applied.
- There are no action costs. These are modelled as negative rewards.
- We have a *discount factor*.

Discounted rewards

The discount factor determines how much a future reward should be discounted compared to a current reward.

For example, would you prefer \$100 today or \$100 in a year's time? We (humans) often *discount* the future and place a higher value on nearer-term rewards.

Assume our agent receives rewards $r_1, r_2, r_3, r_4, \ldots$ in that order. If γ is the discount factor, then the discounted reward is:

$$\begin{aligned} V &= r_1 + \gamma r_2 + \gamma^2 r_3 + \gamma^3 r_4 + \ldots \\ &= r_1 + \gamma(r_2 + \gamma(r_3 + \gamma(r_4 + \ldots))) \end{aligned}$$

If V_t is the value received at time-step t, then $V_t = r_t + \gamma V_{t+1}$

MDP: Example action

Probabilistic PDDL is one way to represent an MDP. It extends PDDL with a few additional constructs. Of most relevance is that outcomes can be associated with probabilities. The following describes the "Bomb and Toilet" problem, in which one of two packages contains a bomb. The bomb can be diffused by dunking it into a toilet, but there is a 0.05 probability of the bomb clogging the toilet.

```
(define (domain bomb-and-toilet)
  (:requirements :conditional-effects :probabilistic-effects)
  (:predicates (bomb-in-package ?pkg) (toilet-clogged)
               (bomb-defused))
  (:action dunk-package
    :parameters (?pkg)
    :effect (and (when (bomb-in-package ?pkg)
                 (bomb-defused))
                 (probabilistic 0.05 (toilet-clogged))))
```

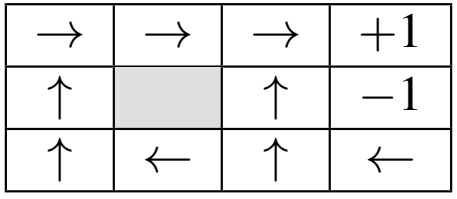
Solutions for MDP problems: policies

The planning problem for discounted-reward MDPs is different to that of classical planning because the actions are non-deterministic. Instead of a sequence of actions, an MDP produces a *policy*.

A policy, π is a function that tells an agent which is the best action to choose in each state. A policy can be *deterministic* or *stochastic*.

A deterministic policy $\pi : S \rightarrow A$ is a *mapping* from states to actions. It specifies which action to choose in every possible state. Thus, if we are in state s, our agent should choose the action defined by $\pi(s)$.

A graphical representation of the policy for Grid World is:



So, in the initial state (bottom left cell), following this policy the agent should go up.

Solutions for MDP problems: policies (continued)

Of course, agents do not work with graphical policies. The output from an MDP planner would look more like this:

```
at(0,0) => move_right
at(0,1) => move_right
at(0,2) => move_right
at(0,3) => stay
at(1,0) => move_up
at(1,2) => move_up
at(1,3) => move_up
at(2,0) => move_up
at(2,1) => move_left
at(2,2) => move_up
at(2,3) => move_left
```

An agent can then parse this in and use it by determining what state it is in, looking up the action for that state, and executing the action. Then repeat.

Stochastic policies

A stochastic policy $\pi : S \times A \rightarrow \mathbb{R}$ specifies the *probability distribution* from which an agent should select an action. Intuitively, $\pi(s, a)$ specifies the probability that action a should be executed in state s.

To execute a stochastic policy, we could just take the action with the maximum $\pi(s, a)$. However, in many domains, it is better to select an action based on the probability distribution; that is, choose the action probablistically such that actions with higher probability are chosen proportionally to their relative probabilities.

In this subject, we will focus only on deterministic policies, but stochastic policies have their place.

Expected Value of Policy for Discounted Reward MDPs

For discounted-reward MDPs, optimal solutions maximise the *expected discounted accumulated reward* from the initial state s_0. But what is the expected discounted accumulated reward?

- In Discounted Reward MDPs, the **expected discounted reward from** s is

$$V^{\pi}(s) = E_{\pi}[\sum_{i} \gamma^{i} r(a_i, s_i) \mid s_0 = s, a_i = \pi(s_i)]$$

Thus, $V^{\pi}(s)$ defines the expected value of following the policy π from state s.

So for our Grid World example, assuming only the -1 and +1 states have rewards/costs, the expected value is:

$\gamma^5 \times 1 \times (0.8^5)$	(optimal movement)
$\gamma^7 \times 1 \times (0.8^7)$	(first move only fails)
$\ldots$	(etc.)

Agenda

Bellman equations (Discounted-Reward MDPs)

The *Bellman equations*, identified by Richard Bellman, describe the condition that must hold for a policy to be optimal. It generalises to problems other than MDPs, but we consider only MDPs here.

For discounted-reward MDPs the Bellman equation is defined recursively as:

$$V(s) = \max_{a \in A(s)} \sum_{s' \in S} P_a(s'|s) \left[r(s, a, s') + \gamma \, V(s') \right]$$

Thus, V is optimal **if** for all states s, $V(s)$ describes the total discounted reward for taking the action with the highest reward over an indefinite/infinite horizon.

The reward of an action is: the sum of the immediate reward for all states possibly resulting from that action plus the discounted future reward of those states; times the probability of that action occurring.

Bellman equations – An Alternate Formulation

Sometimes, Bellman equations are described slightly differently, using what is known as Q-functions.

If $V(s)$ is the expected value of being in state s and acting optimally according to our policy, then we can also describe the *Q-value* of being in a state s, choosing action a and then acting optimally according to our policy as:

For discounted-reward MDPs the Bellman equation is defined recursively as:

$$Q(s,a) = \sum_{s' \in S} P_a(s'|s) \left[r(s,a,s') + \gamma \, V(s') \right]$$

This is just the expression inside the $\max$ expression in the Bellman equation. Using this, sometimes you may see the Bellman equation then defined as:

$$V(s) = \max_{a \in A(s)} Q(s,a)$$

The two definitions are equivalent, but you may seem them defined in both ways. However, when we move onto reinforcement learning later, we will use Q functions more explicitly.

Using the Bellman Equations: Value Iteration

Value Iteration finds the optimal value function V^* solving the Bellman equations iteratively, using the following algorithm:

- Set V_0 to arbitrary value function; e.g., $V_0(s) = 0$ for all s.
- Set V_{i+1} to result of Bellman's **right hand side** using V_i in place of V:

$$V_{i+1}(s) := \max_{a \in A(s)} \sum_{s' \in S} P_a(s'|s) \left[r(s, a, s') + \gamma \, V_i(s') \right]$$

This converges exponentially fast to the optimal policy as iterations continue.

Theorem

$V_i \mapsto V^*$ as $i \mapsto \infty$. That is, given an infinite amount of iterations, it will be optimal.

The complexity of each iteration is $O(|S|^2|A|)$. On each iteration, we iterate in an outer loop over all states in S, and in each outer loop iteration, we need to iterate over all states ($\sum_{s' \in S}$), meaning $|S|^2$ iterations. But also within each outer loop iteration, we need to calculate the value for every action to find the maximum.

Value Iteration in Practice

Value Iteration converges to the optimal value function V^* asymptotically, but in practice, the algorithm is stopped when the **residual** $R = \max_s |V_{i+1}(s) - V_i(s)|$ reaches some pre-determined threshold ϵ – that is, when the largest change in the values between iterations is "small enough".

The resulting greedy policy π_V has it's **loss** bounded by $2\gamma R/1-\gamma$.

It is clear to see that the value iteration can be easily parallelised by updating the value of many states at once: the values of states at step $t+1$ are dependent only on the value of other states at step t.

A policy can now be easily defined: in a state s, given V, choose the action with the highest expected cost (or reward).

Value iteration example: Grid World.

Assuming no action costs and $\gamma = 0.9$.

After 1 iteration

0.00	0.00	0.00	+1
0.00		0.00	−1
0.00	0.00	0.00	0.00

After 2 iterations

0.00	0.00	0.72	+1
0.00		0.00	−1
0.00	0.00	0.00	0.00

After 3 iterations

0.00	0.52	0.78	+1
0.00		0.43	−1
0.00	0.00	0.00	0.00

After 4 iterations

0.37	0.66	0.83	+1
0.00		0.51	−1
0.00	0.00	0.31	0.00

After 5 iterations

0.51	0.72	0.84	+1
0.27		0.55	−1
0.00	0.22	0.37	0.13

After 100 iterations

0.64	0.74	0.85	+1
0.57		0.57	−1
0.49	0.43	0.48	0.28

After 1000 iterations

0.64	0.74	0.85	+1
0.57		0.57	−1
0.49	0.43	0.48	0.28

Policy after 1000 iterations

→	→	→	+1
↑		↑	−1
↑	←	↑	←

Value iteration demo

http://www.cs.ubc.ca/~poole/demos/mdp/vi.html

Deciding How to Act

Given a policy that is (close to) optimal, how should we then select the action to play in a given state? It is reasonably straightforward: select the action that maximises our expected utility. So, given a value function V, we can select the action with the highest expected reward using:

$$\underset{a \in A(s)}{\operatorname{argmax}} \sum_{s' \in S} P_a(s'|s) \left[r(s, a, s') + \gamma \, V(s') \right]$$

This is known as *policy extraction*, because it extracts a policy for a value function (or Q-function). This can be calculated 'on the fly' at runtime.

Alternatively, given a Q-function instead of a value function, we can use:

$$\underset{a \in A(s)}{\operatorname{argmax}} \, Q(s, a)$$

This is simple to decide than using the value functions because we do not need to sum over the set of possible output states.

Policy Iteration

The other common way that MDPs are solved is using *policy iteration* – an approach that is similar to value iteration. While value iteration iterates over value functions, policy iteration iterates over policies themselves, creating a strictly improved policy in each iteration (except if the iterated policy is already optimal).

Policy iteration first starts with some (non-optimal) policy, such as a random policy, and then calculates the value of each state of the MDP given that policy — this step is called the *policy evaluation*. It then updates the policy itself for every state by calculating the expected cost/reward of each action applicable from that state.

The basic idea here is that policy evaluation is easier to computer than value iteration because the set of actions to consider is fixed by the policy that we have so far.

Policy evaluation

The *expected cost* of policy π from s to goal, $V^\pi(s)$, is weighted avg of cost of the possible state sequences defined by that policy times their probability given π.

However, the expected costs $V^\pi(s)$ can also be characterised as a solution to the equation

$$V^\pi(s) = \sum_{s' \in S} P_a(s'|s) \; [r(s, a, s') + \gamma \; V^\pi(s')]$$

where $a = \pi(s)$, and $V^\pi(s) = 0$ for goal states

This set of linear equations can be solved analytically using MATLAB, by a VI-like procedure, or whatever – we do not care for this subject.

The *optimal expected cost* $V^*(s)$ is $\max_\pi V^\pi(s)$ and the *optimal policy* is the $\arg\max$

Policy Iteration

Let $Q^\pi(a, s)$ be the expected cost from s when doing a first and then following the policy π:

$$Q^\pi(a, s) = \sum_{s' \in S} P_a(s'|s) \left[r(s, a, s') + \gamma V^\pi(s') \right]$$

When $Q^\pi(a, s) < Q^\pi(\pi(s), s)$, π *strictly improved* by changing $\pi(s)$ to a

Policy Iteration computes π^* by a sequence of policy evaluations and improvements:

1. Starting with arbitrary policy π
2. Compute $V^\pi(s)$ for all s (policy evaluation)
3. Improve π by setting $\pi(s) := \text{argmax}_{a \in A(s)} Q^\pi(a, s)$ (improvement)
4. If π changed in 3, go back to 2, else *finish*

This algorithm finishes with an optimal π^* after a finite number of iterations, because the number of policies is finite, bounded by $O(|A|^{|S|})$, unlike value iteration, which can theoretically require infinite iterations.

However, each iteration costs $O(|S|^2|A| + |S|^3)$. Empirical evidence suggests that the most efficient is dependent on the particular MDP model being solved.

The Curse of Dimensionality

Thus, solving MDPs using these algorithms is polynomial in the size of the state space, but exponential in the number of variables if we use a PDDL-like language to represent our problem. That is, if there are N number of variables, each a Boolean, there are 2^N number of states. Value iteration and policy iteration require us to keep a vector of size $|2^N|$.

Question: Can we do better?

Answer: Yes! Using function approximation, which we will see in a couple of weeks

Agenda

Partially Observable MDPs

MDPs assume that the agent always knows exactly what state it is in — the problem is fully-observable. However, this is not valid for many tasks; e.g. an unmanned aerial vehicle searching in a earthquake zone for survivors will by definition not know the location of survivors; a card-player agent will not know the cards its opponent holds; etc.

Partially-observable MDPs (POMDPs) relax the assumption of full-observability. A POMDP is defined as:

- states $s \in S$
- set of goal states $G \subseteq S$
- actions $A(s) \subseteq A$
- transition probabilities $P_a(s'|s)$ for $s \in S$ and $a \in A(s)$
- initial **belief state** b_0
- reward function $r(s, a, s')$
- a **sensor model** given by probabilities $P_a(o|s)$, $o \in Obs$

Solving POMDPs (an intuitive overview)

Solving POMDPs is very similar to solving MDPs. In fact, the same algorithms apply. The only difference is that we case the POMDP problem as a standard MDP problem with a new state space: each state is a **probability distribution** over the set S. Thus, each state of the POMDP is a **belief state**, which defined the probability of being in each state S.

Like MDPs, solutions are policies that map belief states into actions.

Optimal policies minimise the expected cost to go from b_0 to G.

We will not cover this in detail in these notes. However, POMDPs are clearly a generalisation of MDPs, and they have had a much larger impact on planning for autonomy than standard MDPs.

Summary: MDPs

We covered Markov Decision Processes (MDPs). They differ from classical planning in that actions can have more than one possible outcome. Each outcome has an associated probability.

There are two solutions for exhaustively calculating the optimal policy: value iteration and policy iteration. These are both based on dynamic programming – specifically, they use the Bellman equations to iteratively improve on a non-optimal solution.

Heuristic search can also be used, but does not produce solutions that are as general – the work only for states that are reachable from the initial state of the search.

Partially-observable MDPs generalise MDPs by admitting descriptions in which the environment is not fully observable. Techniques for solving these are the same as MDPs, but just over a larger search space.

What's next? How to *learn* the probabilities over the action outcomes using *reinforcement learning*.