Value Iteration and Policy Iteration

Original by Michiel van der Ree and Lambert Schomaker*

March 19, 2015

1 Problem description

The algorithms that you will implement this week are designed to solve *sequential decision problems* in which an agent's utility depends on a sequence of decisions. They are formally described in sections 17.1 – 17.3 of the Russell & Norvig book. We repeat here the Bellman equation giving the utility of a state s:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s'). \tag{1}$$

Here, $U(s)$ is the utility of s, $R(s)$ the reward of s, γ a discount factor, $A(s)$ the set of actions that can be performed in s and $P(s'|s,a)$ the probability that the agent will end up in state s' if it performs action a in state s.

We are interested in finding an optimal policy for sequential decision problems. A policy specifies what the agent should do for any state it might reach. A policy is usually denoted by π, and $\pi(s)$ is the action recommended by the policy π for state s. In this assignment you will implement value iteration and policy iteration to find the optimal policy in two mazes: the simple 4×3 environment from the Russell & Norvig book (p. 646) and the larger maze depicted in figure 1. In the first problem, the agent moves in the intented direction with $p = 0.8$ and in each of the perpendicular directions with $p = 0.1$. In the second problem, the agent moves in the intended direction with $p = 0.7$, and in each other direction with $p = 0.1$. In both problems non-goal states are set to have a reward of -0.04.

Islands problem

Before beginning with programming answer the following question: Consider the archipelago depicted in figure 2. Here, each island is a state. In s_1 and s_2, our agent can choose to jump to any of the islands but has a 50% chance of staying in the same state if it does. Once the agent reaches s_3 it will stay there. s_1 does not have anything special to offer, s_2 is covered by non-lethal spikes and s_3 has a treasure chest in it. This is reflected by reward values $R(s_1) = 0$, $R(s_2) = -1$ and $R(s_3) = 1$. We are interested in the optimal policy for this problem and try to find it using policy iteration. At our current iteration, we try to jump to s_2 in s_1 and try to jump to s_3 in s_2.

Given the above information, write out the system of linear equations describing the utilities of states under the current policy and solve it. Use a discount factor $\gamma = 0.5$. Will the policy change? If so, repeat the procedure until convergence.

2 Programming in Python

You will be asked to complete two methods of the Python class `Map` defined in `mdp.py`. Python is an intuitive, high-level programming language which you should be able to pick up fairly quickly even if you haven't encountered it before. If you have any questions regarding the language you can consult the internet, the lab assistants or Python-savvy fellow students.

The `mdp` module provides the following classes and functions to help you on your way:

*Adapted from `http://uhaweb.hartford.edu/compsci/ccli/ql.htm`

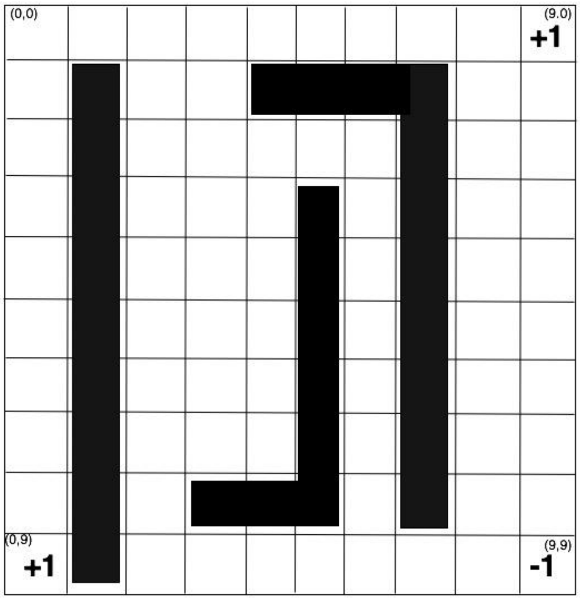

Figure 1: Maze 2.

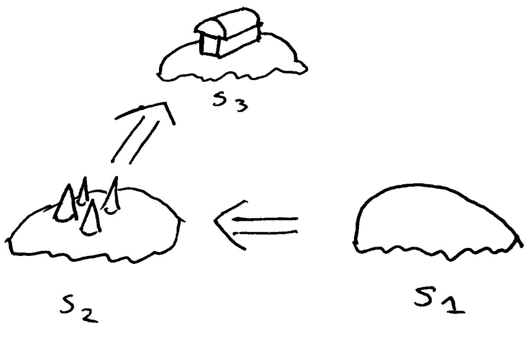

Figure 2: The islands problem.

- `States` have a reward, a set of transition probabilities, a set of legal actions, a policy and an indicator of whether it is a goal state. The class provides methods to compute the expected utility of an action and to find the best action.
- `Map` is the class you will have to complete. In addition to the `valueIteration` and `policyIteration` methods, it provides methods to pretty-print both the policy and the utility of the agent in each state to the command line. It also provides a method `calculateUtilitiesLinear` which is expanded upon below.
- Functions `MakeRNProblem` and `Make2DProblem` define the problems introduced in section 1.

Once finished, you can run your code by starting the Python interpreter (simply `python`) in the same folder as your code and evaluating the following:

```
>>> import mdp
>>> m1 = mdp.makeRNProblem()
>>> m1.valueIteration()
>>> m1.printValues()
:--------:--------:--------:--------:
|  0.301 |  0.472 |  0.682 |    1   |
:--------:--------:--------:--------:
|  0.184 |        |  0.344 |   -1   |
:--------:--------:--------:--------:
|  0.093 |  0.093 |  0.187 | -0.001 |
:--------:--------:--------:--------:

>>> m2 = mdp.makeRNProblem()
>>> m2.policyIteration()
>>> m2.printActions()
:--------:--------:--------:--------:
|   >>   |   >>   |   >>   |    1   |
:--------:--------:--------:--------:
|   /\   |        |   /\   |   -1   |
:--------:--------:--------:--------:
|   /\   |   >>   |   /\   |   <<   |
:--------:--------:--------:--------:
```

Alternatively, you can save the above commands in a script `foo.py` and run `python foo.py`.

A final note on the `calculateUtilitiesLinear` method: In policy iteration, the max operator in equation 1 vanishes since you simply evaluate the action prescribed by the current policy. Without that max operator, the utility equations for all states simplify to a system of linear equations which can be solved by the numerical computing package of your choice. In the `calculateUtilitiesLinear` method, the interfacing between the `Map` class and the NumPy library's least-squares routine is performed. Do check the definition of `calculateUtilitiesLinear` and the NumPy documentation of `linalg.lstsq` to get a grasp of what is happening in this method.

Useful Python functions in this exercise:
`random.choice`, `max`, `abs`

3 Value iteration

Complete class `Map`'s `valueIteration` method in the following way:

1. Set each non-goal state's utility to 0.
2. Until the largest change in utility is smaller than a predefined stop criterion, repeat the following:
   - For all non-goal states, calculate the new utility values according to equation 1.
   - Update each non-goal state's utility using the values just calculated.

Then, verify that your implementation works as expected.

4 Policy iteration

Complete `Map`'s `policyIteration` method in the following way:

1. Initialize the policy by chosing a random action to be performed in each non-goal state.
2. Until the policy does not change, repeat the following:
   - Calculate the utility estimates of all non-goal states under the current policy by calling `Map`'s `calculateUtilitiesLinear` method.
   - Update the policy according to these new utility estimates

Again, verify that your implementation works as expected.

5 Comparing your algorithms

Compare the two algorithms you just implemented on the two problems provided. Things you might want to consider here:

- Do both algorithms find the same solutions?
- What is the number of iterations required for each algorithm to find a solution?
- What is the time* required for each algorithm to find a solution?
- What is the influence of the discount factor on the way both algorithms perform?
- What is the influence of the stop criterion δ in value iteration?

Try to explain any differences you encounter.

Finally, answer the following question: In the problems provided, transition probabilities $P(s'|s,a)$ are explicitly defined. In real-world problems such a transition function generally is unknown. Explain in your own words how reinforcement learning algorithms such as Q-Learning find solutions for sequential decision problems without knowing the transition function explicitly.

*Check Python's `time` module