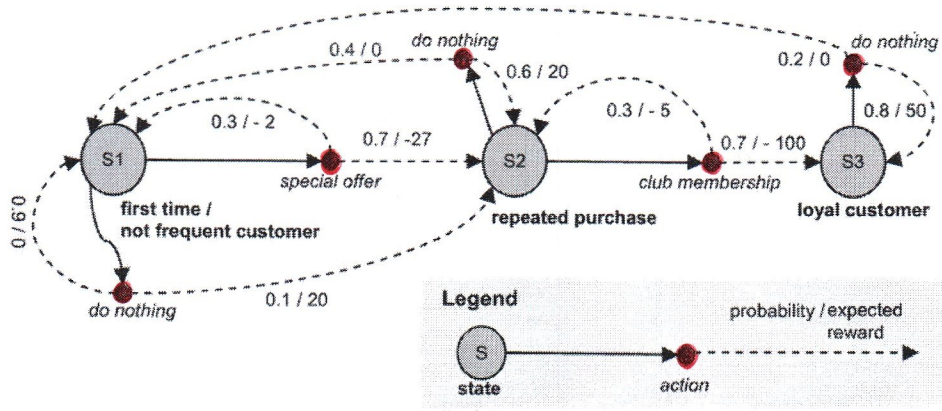
06 - Markov Decision Processes

In this exercise session, we will see how to model a sequential decision making problem with a Markov Decision Process (MDP). Then, we will focus on how to evaluate a given decision strategy (i.e., a policy), which is called the *prediction problem*, and how to find an optimal decision strategy, which is called the *control problem*.

MDP Model

A discounted MDP is defined by a tuple $\mathcal{M} := (\mathcal{S}, \mathcal{A}, P, R, \mu, \gamma)$, in which $\mathcal{S}$ is a set of states, $\mathcal{A}$ is a set of actions, P is the transition model, R is the reward function, μ is the initial state distribution, and γ is the discount factor.

Let us define all the components of $\mathcal{M}$ for the illustrative *Advertising Problem* depicted below.



In [1]:
```
import numpy as np

# number of states
nS = 3

# number of states and actions
nSA = 5

# we represent P with a matrix P_sas such that:
# P_sas [0, 0] : S1, do-nothing, S1
# P_sas [0, 1] : S1, do-nothing, S2
# ...
# P_sas [1, 0] : S2, do-nothing, S1
# ...
P_sas = np.array([[0.9, 0.1, 0.],
                  [0.3, 0.7, 0.],
                  [0.4, 0.6, 0.],
                  [0., 0.3, 0.7],
                  [0.2, 0, 0.8]])

# we represent R with a vector R_sa such that:
# R_sa [0] : expected R(S1, do-nothing)
# R_sa [1] : expected R(S1, special-offer)
# ...
R_sa = np.array([0.9*0. + 0.1*20,
                 0.3*(-2) + 0.7*(-27),
                 0.4*0. + 0.6*20,
                 0.3*(-5) + 0.7*(-100),
                 0.2*0. + 0.8*50])

# initial state distribution
# mu [0] : probility that S1 is the initial state
# ...
mu = np.array([1., 0., 0.])

# discount factor
gamma = 0.9

print('number of states nS:\n', nS, '\n')
print('number of states and action nSA:\n', nSA, '\n')
print('transition matrix P:\n', P_sas, '\n')
print('reward vector R:\n', R_sa, '\n')
print('initial state distribution mu:\n', mu, '\n')
print('discount factor gamma:\n', gamma)
```

formalization of the transition model

formalization of the expected rewards

every row is a state and an action and the columns represent the probability of transition to different states:

1st row corresponds to: "we are in S_1 and we do nothing"

0.9 | 0.1 | 0.0

prob. of being in S_1, do nothing and end up in S_1 again

prob. of being in S_1, do nothing and end up in S_2

prob. of being in S_1, do nothing and end up in S_3 (no chance)

if we are in S_1 and we do nothing, the expected reward is:

$$\sum_{i \in \text{all states}} (P_{1 \to i}) \cdot (r_{1 \to i})$$

```
number of states nS:
 3

number of states and action nSA:
 5

transition matrix P:
 [[0.9 0.1 0. ]
 [0.3 0.7 0. ]
 [0.4 0.6 0. ]
 [0.  0.3 0.7]
 [0.2 0.  0.8]]

reward vector R:
 [  2.  -19.5  12.  -71.5  40. ]
```

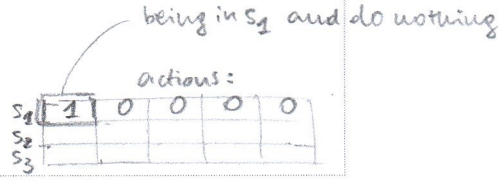
```
initial state distribution mu:
 [1. 0. 0.]

discount factor gamma:
 0.9
```

1. Prediction

Having defined the MDP, we now aim to compute the value of a given decision strategy (policy) interacting with the MDP, which is called the *prediction problem*.

First, let us define a policy that select the action *do-nothing* in every state of the MDP.

In [2]:
```
# we represent a policy with (nS, nS*nA) matrix, such that:
# pi [0, 0] : probability of taking action do-nothing in S1
# pi [0, 1] : probability of taking action special-offfer in S1
# ...
# pi [1, 2] : probability of taking action do-nothing in S2
# ...
# pi [2, 5] : probability of taking action do-nothing in S3
pi = np.array([[1., 0., 0., 0., 0.],
               [0., 0., 1., 0., 0.],
               [0., 0., 0., 0., 1.]])
```



State Value Function

We would like to compute the state value function $V^\pi(s)$ of π for every state $s \in \mathcal{S}$. From the Bellman expectation equation we have

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a|s) \Big(R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^\pi(s') \Big),$$

which in matrix form becomes

$$V^\pi = \pi \big(R + \gamma P V^\pi \big).$$

A first viable option is to compute the closed-form solution of the Bellman expectation equation as

$$V^\pi = \big(I - \gamma \pi P \big)^{-1} \pi R.$$

In [3]:
```
V = np.linalg.inv(np.eye(nS) - gamma * pi @ P_sas) @ (pi @ R_sa)

print('state value function:\n', V)
```

```
state value function:
 [ 36.36363636  54.54545455 166.23376623]
```

Note that, since πP is a stochastic matrix, we have some nice properties over its eigenvalues. Especially, we have guarantees that the matrix $(I - \gamma \pi P)$ is always invertible for $\gamma \in [0, 1)$.

In [4]:
```
eigenvalues, _ = np.linalg.eig(pi @ P_sas)
print('The eigenvalues of (pi * P_sas) are:\n', eigenvalues)
eigenvalues, _ = np.linalg.eig(gamma * pi @ P_sas)
print('The eigenvalues of (gamma * pi * P_sas) are:\n', eigenvalues)
eigenvalues, _ = np.linalg.eig(np.eye(nS) - gamma * pi @ P_sas)
print('The eigenvalues of (I - gamma * pi * P_sas) are:\n', eigenvalues)
```

```
The eigenvalues of (pi * P_sas) are:
 [0.8 0.5 1. ]
The eigenvalues of (gamma * pi * P_sas) are:
 [0.72 0.45 0.9 ]
The eigenvalues of (I - gamma * pi * P_sas) are:
 [0.28 0.55 0.1 ]
```

However, computing the inversion for the closed-form solution might be computatonally unfeasible for very large matrices (i.e., very large state sapces). In this case, a second option is to iteratively apply the Bellman expectation equation until we reach a fixed point.

In [5]:
```
V_old = np.zeros(nS)
tol = 0.0001
V = pi @ R_sa
while np.any(np.abs(V_old - V) > tol):
  V_old = V
  V = pi @ (R_sa + gamma * P_sas @ V)

print('state value function:\n', V)
```

```
state value function:
 [ 36.36277577  54.54459396 166.23290564]
```

Where we stop the loop whenever $\|V_k^\pi - V_{k-1}^\pi\|_\infty$ falls below a pre-specified tolerance.

State-Action Value Function

Similarily as for the state value function $V^\pi(s)$, we can compute the state-action value function $Q^\pi(s,a)$ of the policy π for every $a \in \mathcal{A}$ and $s \in \mathcal{S}$. Especially, we use the corresponding Bellman expectation equation

$$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) \sum_{a' \in \mathcal{A}} \pi(a'|s') Q^\pi(s',a'),$$

which in matrix form becomes

$$Q^\pi = R + \gamma P \pi Q^\pi,$$

and the closed form solution is

$$Q^\pi = \left(I - \gamma P \pi\right)^{-1} R.$$

In [6]:
```
Q = np.linalg.inv(np.eye(nSA) - gamma * P_sas @ pi) @ R_sa

print('state-action value function:\n', Q)
```

```
state-action value function:
 [ 36.36363636  24.68181818  54.54545455  47.95454545 166.23376623]
```

In [7]:
```
Q_old = np.zeros(nSA)
tol = 0.0001
Q = R_sa
while np.any(np.abs(Q_old - Q) > tol):
  Q_old = Q
  Q = R_sa + gamma * P_sas @ pi @ Q

print('state-action value function:\n', Q)
```

```
state-action value function:
 [ 36.36277577  24.68095759  54.54459396  47.95368487 166.23290564]
```

- ## Evaluating Different Policies

We can use the value function to compare a policy against another. Especially, we aim to compare a myopic policy, which always selects action *do-nothing* as before, and a far-sighted policy, which always selects a marketing campaign insetad (actions *special-offer* and *club-memebership* respectively).

In [8]:
```
# myopic policy
pi_myo = np.array([[1., 0., 0., 0., 0.],
                   [0., 0., 1., 0., 0.],
                   [0., 0., 0., 0., 1.]])

# far-sighted policy
pi_far = np.array([[0., 1., 0., 0., 0.],
                   [0., 0., 0., 1., 0.],
                   [0., 0., 0., 0., 1.]])
```

Let us compare the two policies for different γ, such as $\gamma \in \{0.5, 0.9, 0.99\}$.

In [9]:
```
gammas = [0.5, 0.9, 0.99]

for gamma in gammas:
  V_myo = np.linalg.inv(np.eye(nS) - gamma * pi_myo @ P_sas) @ (pi_myo @ R_sa)
  V_far = np.linalg.inv(np.eye(nS) - gamma * pi_far @ P_sas) @ (pi_far @ R_sa)
  print('gamma:', gamma)
  print('V_myo:', V_myo)
  print('V_far:', V_far, '\n')
```

```
gamma: 0.5
V_myo: [ 5.33333333 18.66666667 67.55555556]
V_far: [-47.62017804 -59.9347181   58.72997033]

gamma: 0.9
V_myo: [ 36.36363636  54.54545455 166.23376623]
V_far: [ -9.28892889  20.1890189  136.88568857]

gamma: 0.99
V_myo: [396.03960396 415.84158416 569.30693069]
V_far: [785.38314104 824.85475924 939.93202849]
```

huge difference even only with $\gamma = 0.9$ and $\gamma = 0.99$

Notably, the myopic strategy is better for lower values of γ, whereas the far-sighted one is better for larger γ. Indeed, the lower is γ and the more we discount future rewards, and it does make little sense to invest in a marketing campaign if we do not care about making the customers happy for future gains.

2. Control

We now have a way to evaluate a given policy in an MDP, or to solve the *prediction problem*. Can we also find an optimal policy such that $\pi^* \in \arg\max_{\pi \in \Pi} V^\pi(s), \forall s \in \mathcal{S}$, where Π is the set of all the policies?

This is what we call the *control problem*. In principle, one would like to solve the Bellman optimality equation:

$$V^*(s) = \max_{a \in A} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^*(s') \right\}.$$

Unfortunately, a closed-form solution is unfeasible, due to the non-linearity introduced by the $\max$ operator. Anyway, we will actually see three options to find an optimal policy:

How many policies do we have? Policies are ∞ in principle, because they're probability distributions over actions. However, we have a result: in reinforcement learning there ∃ an optimal policy which is DETERMINISTIC

1. Brute Force: evaluate all the policies and select the one maximizing the value function;
2. Policy Iteration: iterates between evaluating the currrent policy and improving the policy in the greedy direction;
3. Value Iteration: just compute the optimal value function directly and then recover the optimal policy.

2.1 Brute Force

Let us start with the brute force approach. First, we enumerate all the possibile deterministic policies.

In [10]:
```
policies = []
```

```
policies.append(np.array([[1., 0., 0., 0., 0.], [0., 0., 1., 0., 0.], [0., 0., 0., 0., 1.]]))
policies.append(np.array([[0., 1., 0., 0., 0.], [0., 0., 1., 0., 0.], [0., 0., 0., 0., 1.]]))
policies.append(np.array([[1., 0., 0., 0., 0.], [0., 0., 0., 1., 0.], [0., 0., 0., 0., 1.]]))
policies.append(np.array([[0., 1., 0., 0., 0.], [0., 0., 0., 1., 0.], [0., 0., 0., 0., 1.]]))
```

Then, we evaluate all of them to find the optimal one.

In [11]:
```
gamma = 0.9
V_max = np.zeros(nS)
i_max = -1
for i, pi in enumerate(policies):
  V = np.linalg.inv(np.eye(nS) - gamma * pi @ P_sas) @ (pi @ R_sa)
  print('Value of policy', i, 'is:', V)
  if np.all(V > V_max):
    V_max = V
    i_max = i

print('\nThe optimal policy is the', i_max ,'one:\n', policies[i_max])
```

```
Value of policy 0 is: [ 36.36363636  54.54545455 166.23376623]
Value of policy 1 is: [-12.93577982  15.96330275 134.5412844 ]
Value of policy 2 is: [ 30.56234719  42.29828851 162.50436605]
Value of policy 3 is: [ -9.28892889  20.1890189  136.88568857]

The optimal policy is the 0 one:
 [[1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1.]]
```

Clearly, the brute force approach is not feasible whenever the policy space (thus, the state and action spaces) is large. However, we have some smarter methods to address the *control problem*.

2.2 Policy Iteration

With policy iteration, we start from an initial policy π, and then we iterate between two steps:

1. Policy evaluation, in which we evaluate the current policy π, i.e., we compute

$$Q^{\pi} = \left(I - \gamma P \pi\right)^{-1} R.$$

2. Policy improvement, in which we build a new policy π' that selects the greedy action w.r.t. the evaluation of π, i.e.,

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V^{\pi}(s') \right\},$$

until we reach a fixed point. (Note that the equation of the policy improvement is different from the Bellman optimality equation as it is computed w.r.t. the current policy V^{π} instead of the optimal policy V^*, which is still unknown).

In [12]:
```
import numpy.matlib

# admissible actions
adm_actions = np.array([[1., 1., 0., 0., 0.], [0., 0., 1., 1., 0.], [0., 0., 0., 0., 1.]])

# we can start from any policy
pi = np.array([[0., 1., 0., 0., 0.], [0., 0., 0., 1., 0.], [0., 0., 0., 0., 1.]])

# initializations
Q = np.zeros(nSA)
Q_old = np.ones(nSA)
# main loop
while np.any(Q != Q_old):
  Q_old = Q
  # policy evaluation step
  Q = np.linalg.inv(np.eye(nSA) - gamma * P_sas @ pi) @ R_sa
  # policy improvement step
  greedy_rep = np.matlib.repmat(Q, nS, 1) * adm_actions
  greedy_rep[greedy_rep == 0] = - np.inf
  greedy_actions = [[i, np.argmax(greedy_rep[i, :])] for i in range(nS)]
  pi = np.array([1. if [x, y] in greedy_actions else 0.
                 for x, y in np.ndindex((nS, nSA))]).reshape(nS, nSA)

print('The optimal policy is:\n', pi)
```

```
The optimal policy is:
 [[1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1.]]
```

2.3 Value Iteration

Instead of iterating between policy evaluations and improvements, let us try to evaluate the optimal policy directly (i.e., to compute V^*), by repeatedly applying the Bellman optimality equation on the current value function V_k

$$V_{k+1}(s) \leftarrow \max_{a \in \mathcal{A}} \left\{ R(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s'|s,a) V_k(s') \right\}.$$

This procedure is guaranteed to V^* eventually (because the Bellman optimality equation induces a contraction).

Once we have V^*, we can easily recover the optimal policy, i.e., the greedy one w.r.t. V^*.

In [13]:
```
import numpy.matlib

# admissible actions
adm_actions = np.array([[1., 1., 0., 0., 0.], [0., 0., 1., 1., 0.], [0., 0., 0., 0., 1.]])
```

```
# initializations
V = np.zeros(nS)
V_old = np.ones(nS)
tol = 0.0001
# main loop
while np.any(np.abs(V_old - V) > tol):
  V_old = V
  greedy_rep = R_sa + gamma * P_sas @ V
  greedy_rep = np.matlib.repmat(greedy_rep, nS, 1) * adm_actions
  greedy_rep[greedy_rep == 0] = - np.inf
  V = np.amax(greedy_rep, axis=1)

print('The optimal value function is:\n', V)

# recovering the optimal policy
greedy_rep = R_sa + gamma * P_sas @ V
greedy_rep = np.matlib.repmat(greedy_rep, nS, 1) * adm_actions
greedy_rep[greedy_rep == 0] = - np.inf
greedy_actions = [[i, np.argmax(greedy_rep[i, :])] for i in range(nS)]
pi = np.array([1. if [x, y] in greedy_actions else 0.
               for x, y in np.ndindex((nS, nSA))]).reshape(nS, nSA)

print('\nThe optimal policy is:\n', pi)
```

```
The optimal value function is:
 [ 36.36277577  54.54459396 166.23290564]

The optimal policy is:
 [[1. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 1.]]
```

Homeworks

Here we propose some additional exercises in Python for you. They are not mandatory, but they can be helpful to better understand the contents of the lecture.

State and State-Action Value Functions

Above, we provided ways to compute V^π or Q^π alike. However, there is quite strong relation between the twos. Try to compute the state-action value function Q^π of a policy π directly from its value function V^π.

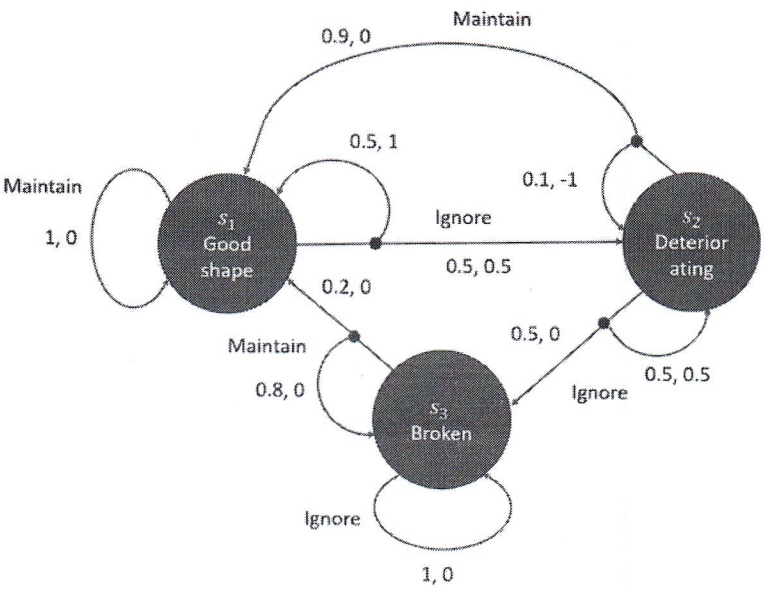
Stochastic Policies

Throughout this notebook we only considered deterministic policies. Let us try to answer the following:

- What can we say about the value of a stochastic policy even before computing it?
- Can you compute the value of a stochastic policy?
- What happens if we start the policy iteration algorithm from a stochastic policy?

MDP Modeling

Try to model in Python the MDP depicted below.



7 Markov Decision Processes

Exercise 7.1

Tell if the following statements about MDPs are true or false. Motivate your answers.

1. To solve an MDP we should take into account state-action pairs one by one;
2. An action you take on an MDP might influence the future rewards you gained;
3. A state considered by the agent acting on an MDP is always equal to the environment state;
4. Problems in which an agent knows the state of the environment and the MDP do not require the use of RL; Reinforcement learning
5. Policies applied to an MDP are influenced by other learning processes ongoing on the considered MDP.

Exercise 7.2

State if the following applications may be modeled by means of an MDP:

1. Robotic navigation in a grid world;
2. Stock Investment;
3. Robotic soccer;
4. Playing Carcassonne (board game).

Define the possible actions and states of each MDP you considered.

Exercise 7.3

Consider the following modeling of a classification problem as sequential decision

making problem:

$$o_i \leftarrow x_i$$
$$a_i \leftarrow \hat{y}_i$$
$$r_i \leftarrow 1 - |t_i - \hat{y}_i|$$

Does this correspondence makes sense? Comment adequately your answer.

Exercise 7.4

For each one of the following dichotomies in MDP modeling provide examples of problems with the listed characteristics:

1. Finite/infinite actions;
2. Deterministic/stochastic transitions;
3. Deterministic/stochastic rewards;
4. Finite/indefinite/infinite horizon;
5. Stationary/non–stationary environment.

Exercise 7.5

Are the following statements about the discount factor γ in a MDP correct?

- A myopic learner corresponds to have low γ values in the definition of the MDP;
- In an infinite horizon MDP we should avoid using $\gamma = 1$, while it is reasonable if the horizon is finite;
- γ is an hyper-parameter for the policy learning algorithm;
- Probability that an MDP will be played in the next round is γ.

Provide adequate motivations for your answers.

Exercise 7.6

The generic definition of policy is a stochastic function $\pi(h_i) = \mathbb{P}(a_i|h_i)$ which given a history $h_i = \{o_1, a_1, s_1, \ldots, o_i, a_i, s_i\}$ provides a distribution over the possible actions $\{a_i\}_i$.

Formulate the specific definition of a policy if the considered problem is:

1. Markovian, Stochastic, Non-stationary
2. History based, Stochastic, Stationary
3. Markovian, Deterministic, Stationary

Exercise 7.7

Comment the following statements about solving MDPs. Motivate your answers.

1. In a finite state MDP we just need to look for Markovian, stationary and deterministic optimal policies;
2. For finite time MDPs we should consider non-stationary optimal policies;
3. The results of coupling a specific policy and an MDP is a Markov process;
4. Given a policy we can compute P^{π} and R^{π} on an MDP;
5. The value function $V^{\pi^*}(s)$ contains all the information to execute the optimal policy π^* on a given MDP;
6. The action-value function $Q^{\pi^*}(s,a)$ contains all the information to execute the optimal policy π^* on a given MDP;
7. There is a unique optimal policy in an MDP;
8. There is a unique optimal value function in an MDP.

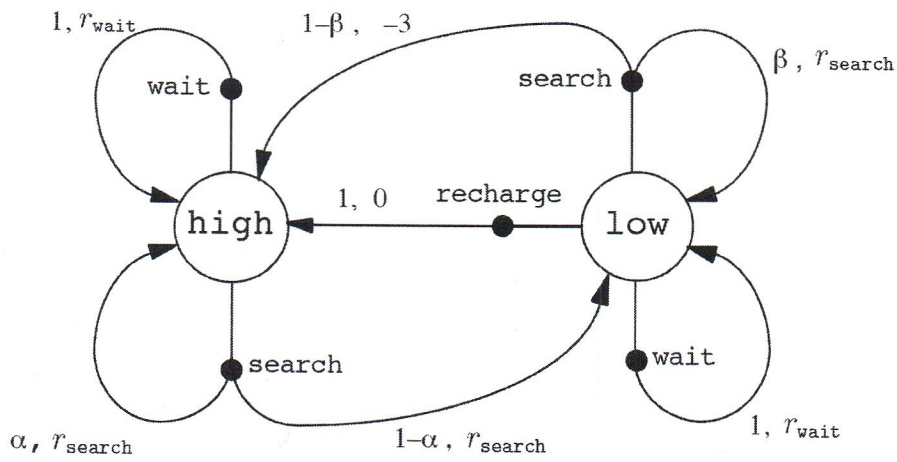


Figure 7.1: The MDP for the cleaning robot problem.
The states are for the battery of

Exercise 7.8

Consider the MDP in Figure 7.1 with $\alpha = 0.3$, $\beta = 0.5$, $\gamma = 1$, $r_{search} = 2$, $r_{wait} = 0$ and the following policy:

$$\pi(s|H) = 1$$
$$\pi(s|L) = 0.5$$
$$\pi(r|L) = 0.5$$

We cannot use the matrix inversion because we have a fixed horizon (we know that the process is going to stop in 2 steps)

Compute the Value function where the MDP stops after two steps. What happens if we consider a discount factor of $\gamma = 0.5$.

Compute the action-value function for each action value pair in the case the MDP stops after a single step.

Exercise 7.9

Provide the formulation of the Bellman expectation for V equations for the MDP in Figure 7.1, with $\alpha = 0.2$, $\beta = 0.1$, $r_{search} = 2$, $r_{wait} = 0$, $\gamma = 0.9$ and in the case we consider the policy:

$$\pi(H|s) = 1$$
$$\pi(L|r) = 1$$

Exercise 7.10

Tell if the following statements are TRUE or FALSE. Motivate your answers.

1. We are assured to converge to a solution when we apply repeatedly the Bellman expectation operator;
2. We are assured to converge to a solution when we apply repeatedly the Bellman optimality operator;
3. The Bellman solution to Bellman expectation operator is always a good choice to compute the value function for an MDP;
4. The solution provided by the iterative use of the Bellman expectation operator is always less expensive than computing the exact solution using the Bellman expectation equation;
5. The application of the Bellman optimality operator 10 times applied to a generic value function V_0 guarantees that $||V^* - T^{10}V_0||_\infty \leq \gamma^{10}||V^* - V_0||_\infty$

X Exercise 7.11

Which one would you chose between the use of the Bellman recursive equation vs. Bellman exact solution in the case we are considering the following problems:

1. Chess
2. Cleaning robot problem in Figure 7.1
3. Maze escape
4. Tic-tac-toe

Provide adequate motivations for your answers.

Exercise 7.12

Consider the MDP in Figure 7.2:

1. Provide the transition matrix for the policy $\pi(I|s_1) = 1, \pi(M|s_2) = 1, \pi(M|s_3) = 1$;
2. Provide the expected instantaneous reward for the previous policy;
3. Compute the value function for the previous policy in the case the MDP stops after two steps;
4. Compute the action-value function for each state-action pair in the case the MDP stops after a single step.

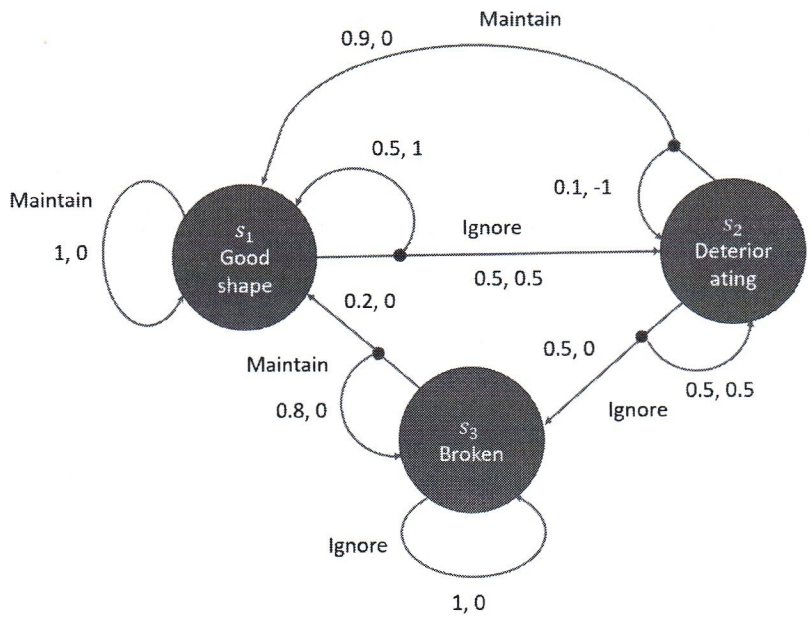


Figure 7.2: The MDP for machinery maintenance problem.

Solutions

Answer of exercise 7.1

1. FALSE State/action pairs determines the next state to be visited and the corresponding reward. Thus, to find a proper policy we should consider also the actions chosen in other states.

2. TRUE The action might determine the sequence of states you will visit in the future, thus the reward you will collect.

3. FALSE It depends on the problem we are tackling. Sometimes the state of the environment is completely known to an agent, sometimes it is partially/totally hidden from her. In some problems, even if the environment state is fully known, the agent might consider a different representation of the state to use, for instance if is too complex to be stored.

4. FALSE RL could be used for computational reasons, e.g., if the number of states is too large.

5. TRUE In the case a learner, other than the agent, is operating in the considered MDP we could have a non-stationary MDP.

Answer of exercise 7.2

An MDP is fully defined if you specify:

- $\mathcal{S}$ a set of states
- $\mathcal{A}$ a set of actions
- P a state transition probability matrix
- R a reward function, $R(s; a) = \mathbb{E}[r|s; a]$
- γ a discount factor,
- μ_i^0 a set of initial probabilities

Robotic navigation in a grid world:

- $\mathcal{S}$ each one of the locations of the grid
- $\mathcal{A}$ usually is go left, go right, go up and go down, no action in the goal state
- P a binary matrix where you have 1 if you reach s' starting from s with action a

and 0 otherwise

- R 0 if you are still away from the goal point and 1 if you reached the goal
- $\gamma = 1$
- μ_i^0 1 in the initial location, 0 otherwise

Stock Investment decision:

- $\mathcal{S}$ amount of money in the bank account, amount of stocks owned
- $\mathcal{A}$ amount of stocks to buy or sell at the next time instant
- P deterministic from a state to another
- R difference in the portfolio value between two time instant
- $\gamma = 0.9$
- μ_i^0 1 in the state corresponding to the amount of money one has in the bank account and the number of stocks she owns, 0 in all other states

Robotic soccer (let us consider the opponent team as a stochastic event):

- $\mathcal{S}$ position of each one of the robots of our team on the field, position of the ball
- $\mathcal{A}$ movement of each one of the robots of the team in the field
- P stochastic transition given from the not deterministic behaviour of the ball and the actions of the environment
- R 1 if a goal has been scored, 0 otherwise
- $\gamma = 1$ since the game ends after a finite amount of time
- μ_i^0 1 in the state corresponding to the initial disposition of the robots on the field and the initial possession of the ball.

Complete rules to play Carcassonne can be found at: `http://riograndegames.com/getFile.php?id=670`. Playing Carcassonne (given the fixed strategies of the other players):

- $\mathcal{S}$ all the possible disposition of the tiles on the table and of the meeples of each player on the tiles in a valid position and a score for each player
- $\mathcal{A}$ place a tile adjacent to a valid tile
- P deterministic state transition from a state to another

- R points scored in the turn or points scored in the final turn
- $\gamma = 1$ since the number of tiles is finite
- μ_i^0 1 on the initial tile on the board, 0 otherwise

Answer of exercise 7.3

The correspondence makes sense, since it is a specific case of an MDP having a single state. The use of traditional techniques to solve MDPs does not make sense, since they are unnecessarily complex for the problem they are trying to solve, i.e, there does not exist a temporal dependence over the predictions.

Answer of exercise 7.4

1.
   - Finite: robotic navigation (up, down, left and right)
   - Infinite actions: pole balancing with continuous space applied force
2.
   - Deterministic transitions: chess (given the opponent strategy)
   - Stochastic transitions: blackjack
3.
   - Deterministic rewards: robotic navigation (0 everywhere, 1 in the exit point)
   - Stochastic rewards: ad banner allocation (depending on clicks)
4.
   - Finite horizon: Carcassonne (finite number of tiles)
   - Indefinite horizon: chess
   - Infinite horizon: stock exchange
5.
   - Stationary environment: robotic navigation
   - Non-stationary environment: every game with another learner playing

Answer of exercise 7.5

- TRUE Low values of gamma means that we are not considering valuable the revenues gained in the far future. Conversely, in the case we are far–sighted learners, we should use a high value for γ;
- TRUE If we consider $\gamma = 1$ we are not discounting rewards gained in the future, which may lead to produce infinite cumulated rewards. On the contrary, it is reasonable not to discount rewards if we know the horizon is finite;

- FALSE γ is a parameter of the MDP we are considering and it is conceptually related to the problem we are facing and not to the learner. Different values for γ may correspond to different optimal policies for the MDP;
- TRUE By considering a specific $\gamma < 1$ we are somehow stating the fact that the process might end with a given probability at the next time step.

Answer of exercise 7.6

1. Since it is Markovian, the policy will depend only on the current state, i.e., $h_i = s_i$, but will still require information about the time instant we are in since the problem is non-stationary, thus:
$$\pi(s_i, i) = \mathbb{P}(a_i|s_i, i)$$
2. Since the policy is stationary it will not depend on the time instant we are at:
$$\pi(h_i) = \mathbb{P}(a_i|h_i)$$
3. The Markov property induce that we consider only $h_i = s_i$ and the fact that it is deterministic implies that the policy determines a single action at each state:
$$\pi(s_i) = a_i$$

Answer of exercise 7.7

1. TRUE We have insurance that at least one Markovian, deterministic and stationary optimal policy exists. This does not imply that it is only one or that all the optimal policies satisfies the aforementioned properties;
2. TRUE The fact that we know that the process will end implies that the optimal action might be influenced also by the number of rounds remaining;
3. TRUE Once you fix the policy there is no need of deciding anything else and the succession of the states becomes a Markovian process;
4. TRUE Once we fix the policy, we can compute the transition P^π and the reward R^π in each state. This is needed if we want to solve the Bellman expectation equation;
5. FALSE The knowledge of the optimal value in each state $V^{\pi^*}(s)$ does not imply that we know the optimal action to perform in each state to get it;

6. TRUE The state-value function $Q^{\pi^*}(s,a)$ specifies the value obtained at each state for each action. Thus the optimal policy is just $\pi^*(s) = \arg\max_a Q^{\pi^*}(s,a)$;
7. FALSE There might be multiple policies getting the same value in each state;
8. TRUE V^* is the unique fixed point for the Bellman optimality equation.

Answer of exercise 7.8

- Since $V^\pi(s) = \sum_{a\in\mathcal{A}} \pi(s|a)\left(R(s,a) + \gamma \sum_{s'\in\mathcal{S}} P(s'|a,s)V^\pi(s)\right)$ thus:

$$\begin{aligned}
V^\pi(H) &= 1[0.3(2 + V^\pi(H)) + 0.7(2 + V^\pi(L))] \\
&= 0.6 + 0.3(0.3 \cdot 2 + 0.7 \cdot 2) + 1.4 + 0.7[0.5 \cdot 0 + 0.5(0.5 \cdot (-3) + 0.5 \cdot 2)] \\
&= 0.6 + 0.6 + 1.4 - 0.175 = 2.425 \\
V^\pi(L) &= 0.5(0 + V^\pi(H)) + 0.5[0.5(-3 + V^\pi(H)) + 0.5(2 + V^\pi(L))] \\
&= 0 + 0.5(0.3 \cdot 2 + 0.7 \cdot 2) - 0.25 + 0.25(0.3 \cdot 2 + 0.7 \cdot 2) + \\
&+ 0.25[0.5 \cdot 0 + 0.5(0.5 \cdot (-3) + 0.5 \cdot 2)] \\
&= 0 + 1 - 0.25 + 0.5 - 0.0625 = 1.1875
\end{aligned}$$

In the case we have a discount factor $\gamma = 0.5$ we have:

$$\begin{aligned}
V^\pi(H) &= 1[0.3 \cdot 2 + 0.5 \cdot 0.3V^\pi(H)) + 0.7 \cdot 2 + 0.5 \cdot 0.7V^\pi(L)] \\
&= 0.6 + 0.15[0.3 \cdot 2 + 0.7 \cdot 2] + 1.4 + 0.35[0.5 \cdot 0 + 0.5(0.5 \cdot (-3) + 0.5 \cdot 2)]; \\
&= 0.6 + 0.3 + 1.4 - 0.0875 = 2.2125 \\
V^\pi(L) &= 0.5(0 + 0.5V^\pi(H)) + 0.5[0.5(-3 + 0.5V^\pi(H)) + 0.5(2 + 0.5V^\pi(L))] \\
&= 0 + 0.5 \cdot 0.5(0.3 \cdot 2 + 0.7 \cdot 2) - 0.75 + 0.125(0.3 \cdot 2 + 0.7 \cdot 2) + \\
&+ 0.5 + 0.25[0.5 \cdot 0 + 0.5(0.5 \cdot (-3) + 0.5 \cdot 2)] \\
&= 0.5 - 0.75 + 0.25 + 0.5 - 0.0625 = 0.4375
\end{aligned}$$

Clearly the discounted value for the states is lower than the not discounted one, in the case we consider an MDP with finite time horizon.

- The action-value function in our case correspond to the values of the expected instantaneous reward, thus:

$$\begin{aligned}
Q(L,w) &= 1 \cdot 0 = 0 \\
Q(L,s) &= 0.5 \cdot 2 + 0.5 \cdot (-3) = -0.5 \\
Q(L,r) &= 1 \cdot 0 = 0 \\
Q(H,w) &= 1 \cdot 0 = 0 \\
Q(H,s) &= 0.3 \cdot 2 + 0.7 \cdot 2 = 2
\end{aligned}$$

Answer of exercise 7.9

$$V(H) = 2 + 0.9[0.2V(H) + 0.8V(L)]$$
$$V(L) = 0 + 0.9V(H)$$

Or in matrix form:

$$V = \begin{bmatrix} 2 \\ 0 \end{bmatrix} + 0.9 \begin{bmatrix} 0.2 & 0.8 \\ 1 & 0 \end{bmatrix} V$$

Answer of exercise 7.10

1. TRUE Since it is possible to show that it is a contraction operator, it is assured to have a single optimal point;
2. TRUE As its expectation counterpart it is a contraction, thus converges to an optimal point;
3. FALSE In the case we have computational constraints (or equivalently a huge state space) we might resort to the recursive solution;
4. FALSE If we want to find the exact solution we might need more computational power than solving the linear system corresponding to the Bellman expectation equation;
5. TRUE Since the fact that the operator is a contraction will guarantee to shrink the distance from the optimal solution at each step of a factor γ.

Answer of exercise 7.11

1. RECURSIVE The state space is too large
2. EXACT The state space is small enough that the exact solution is feasible
3. NONE We do not have information about the state we are in (in a maze we don't know where we are, we cannot apply MPD otherwise it would be trivial)
4. EXACT Again we have a small enough state space

Answer of exercise 7.12

1.

$$P = \begin{bmatrix} 0.5 & 0.5 & 0 \\ 0.9 & 0.1 & 0 \\ 0.2 & 0 & 0.8 \end{bmatrix}$$

2.

$$R = \begin{bmatrix} 0.75 \\ -0.1 \\ 0 \end{bmatrix}$$

3.

$$\begin{aligned} V(s_1) &= R(s_1) + 0.5V(s_1) + 0.5V(s_2) \\ &= R(s_1) + 0.5R(s_1) + 0.5R(s_2) = 0.75 - 0.1 + 0.75 = 1.4 \\ V(s_2) &= R(s_2) + 0.9V(s_1) + 0.1V(s_2) \\ &= R(s_2) + 0.9R(s_1) + 0.1R(s_2) = -0.1 + 0.675 - 0.01 = 0.664 \\ V(s_3) &= R(s_3) + 0.8V(s_1) + 0.2V(s_3) \\ &= R(s_3) + 0.8R(s_1) + 0.2R(s_3) = 0 + 0.15 + 0 = 0.15 \end{aligned}$$

4.

$$\begin{aligned} &Q(M, s_1) = 0 \\ &Q(I, s_1) = 0.5 \cdot 0.5 + 1 \cdot 0.5 = 0.75 \\ &Q(M, s_2) = -1 \cdot 0.1 + 0 \cdot 0.9 = -0.1 \\ &Q(I, s_2) = 0 \cdot 0.5 + 0.5 \cdot 0.5 = 0.25 \\ &Q(M, s_3) = 0 \cdot 0.8 + 0 \cdot 0.2 = 0 \\ &Q(I, s_3) = 0 \end{aligned}$$