#!/usr/bin/env python
coding: utf-8

Learning and Decision Making

Laboratory 2: Markov decision problems
#
In the end of the lab, you should export the notebook to a Python script (``File >> Download as >> Python (.py)``). Make sure that the resulting script includes all code written in the tasks marked as "**Activity n. N**", together with any replies to specific questions posed. Your file should be named `padi-labKK-groupXXX.py`, where `KK` corresponds to the lab number and the `XXX` corresponds to your group number. Similarly, your homework should consist of a single pdf file named `padi-hwKK-groupXXX.pdf`. You should create a zip file with the lab and homework files and submit it in Fenix **at most 30 minutes after your lab is over**.
#
Make sure to strictly respect the specifications in each activity, in terms of the intended inputs, outputs and naming conventions.
#
In particular, after completing the activities you should be able to replicate the examples provided (although this, in itself, is no guarantee that the activities are correctly completed).

1. The MDP Model
#
Consider once again the garbage collection problem described in the Homework and for which you partially wrote a Markov decision problem model. In this lab, you will consider a larger version and slightly modified version of the same problem, described by the diagram:
#
<img src="garbage-big.png">
#
Recall that the MDP should describe the decision-making process of the truck driver. In the above domain,
#
* At any time step, garbage is _at most_ in one of the cells marked with a garbage bin.
* When the garbage truck picks up the garbage from one of the bins, it becomes ``loaded``.
* While the truck is loaded, no garbage appears in any of the marked locations.
* The driver has six actions available: `Up`, `Down`, `Left`, `Right`, `Pick`, and `Drop`.
* Each movement action moves the truck to the adjacent stop in the corresponding direction, if there is one. Otherwise, it has no effect.
* The `Pick` action succeeds when the truck is in a location with garbage. In that case, the truck becomes "loaded".
* The `Drop` action succeeds when the loaded truck is at the recycling plant. After a successful drop, the truck becomes empty, and garbage may now appear in any of the marked cells with a total probability of 0.3.
#
In this lab you will use an MDP based on the aforementioned domain and investigate how to evaluate, solve and simulate a Markov decision problem.
#
Throughout the lab, unless if stated otherwise, use $\gamma=0.99$.
#
$$\diamond$$
#
In this first activity, you will implement an MDP model in Python. You will start by loading the MDP information from a `numpy` binary file, using the `numpy` function `load`. The file contains

the list of states, actions, the transition probability matrices and cost function. After you load the file, you can index the resulting object as a dictionary to access the different elements.

#
Activity 1.
#
Write a function named `load_mdp` that receives, as input, a string corresponding to the name of the file with the MDP information, and a real number γ between 0 and 1. The loaded file will contain 4 arrays:
#
* An array `X` that contains all the states in the MDP represented as strings. In the garbage collection environment above, for example, there is a total of 462 states, each describing the location of the truck in the environment, the location of the garbage (or `None` if no garbage exists in the environment), and whether the truck is `loaded` or `empty`. Each state is, therefore, a string of the form `"(p, g, t)"`, where:
* `p` is one of `0`, ..., `32`, indicating the location of the truck;
* `g` is either `None` or one of `1`, `9`, `10`, `11`, `18`, `19`, `20`, `21`, `23`, `27`, `28`, `29`, indicating that no garbage exists (`None`), or that there is garbage in one of the listed stops;
* `t` is either `empty` or `loaded`, indicating whether the truck is loaded or not.
* An array `A` that contains all the actions in the MDP, represented as strings. In the garbage collection environment above, for example, each action is represented as a string `"Up"`, `"Down"`, `"Left"`, `"Right"`, `"Pick"`, and `"Drop"`.
* An array `P` containing `len(A)` subarrays, each with dimension `len(X)` × `len(X)` and corresponding to the transition probability matrix for one action.
* An array `c` with dimension `len(X)` × `len(A)` containing the cost function for the MDP.
#
Your function should create the MDP as a tuple `(X, A, (Pa, a = 0, ..., len(A)), c, g)`, where `X` is a tuple containing the states in the MDP represented as strings (see above), `A` is a tuple containing the actions in the MDP represented as strings (see above), `P` is a tuple with `len(A)` elements, where `P[a]` is an `np.array` corresponding to the transition probability matrix for action `a`, `c` is an `np.array` corresponding to the cost function for the MDP, and `g` is a float, corresponding to the discount and provided as the argument γ of your function. Your function should return the MDP tuple.
#

In[1]:

```
import numpy as np
import numpy.random as rand

def load_mdp(file,y):
    file_loaded = np.load(file)
    X=()
    A=()
    P=()

    X = file_loaded['X']
    A = file_loaded['A']
    P = file_loaded['P']
    c = file_loaded['c']
    return (X,A,P,c,y)
```

```
M = load_mdp('garbage-big.npz', 0.99)

rand.seed(42)

# States
print('= State space (%i states) =' % len(M[0]))
print('\nStates:')
for i in range(min(10, len(M[0]))):
    print(M[0][i])

print('...')

# Random state
s = rand.randint(len(M[0]))
print('\nRandom state: s =', M[0][s])

# Last state
print('Last state: s =', M[0][-1])

# Actions
print('\n= Action space (%i actions) =\n' % len(M[1]))
for i in range(len(M[1])):
    print(M[1][i])

# Random action
a = rand.randint(len(M[1]))
print('\nRandom action: a =', M[1][a])

# Transition probabilities
print('\n= Transition probabilities =')

for i in range(len(M[1])):
    print('\nTransition probability matrix dimensions (action %s):' % M[1][i], M[2][i].shape)
    print('Dimensions add up for action "%s"?' % M[1][i], np.isclose(np.sum(M[2][i]), len(M[0])))

print('\nState-action pair (%s, %s) transitions to state(s)' % (M[0][s], M[1][a]))
print("s' in", np.array(M[0])[np.where(M[2][a][s, :] > 0)])

# Cost
print('\n= Costs =')
print('\nCost for the state-action pair (%s, %s):' % (M[0][s], M[1][a]))
print('c(s, a) =', M[3][s, a])


# Discount
print('\n= Discount =')
print('\ngamma =', M[4])


# We provide below an example of application of the function with the file `garbage-big.npz` that you can use as a first "sanity check" for your code. Note that, even fixing the seed, the results you obtain may slightly differ.
#
```

```
# ```python
# import numpy.random as rand
#
# M = load_mdp('garbage-big.npz', 0.99)
#
# rand.seed(42)
#
# # States
# print('= State space (%i states) =' % len(M[0]))
# print('\nStates:')
# for i in range(min(10, len(M[0]))):
#     print(M[0][i])
#
# print('...')
#
# # Random state
# s = rand.randint(len(M[0]))
# print('\nRandom state: s =', M[0][s])
#
# # Last state
# print('Last state: s =', M[0][-1])
#
# # Actions
# print('\n= Action space (%i actions) =\n' % len(M[1]))
# for i in range(len(M[1])):
#     print(M[1][i])
#
# # Random action
# a = rand.randint(len(M[1]))
# print('\nRandom action: a =', M[1][a])
#
# # Transition probabilities
# print('\n= Transition probabilities =')
#
# for i in range(len(M[1])):
#     print('\nTransition probability matrix dimensions (action %s):' % M[1][i], M[2][i].shape)
#     print('Dimensions add up for action "%s"?' % M[1][i], np.isclose(np.sum(M[2][i]), len(M[0])))
#
# print('\nState-action pair (%s, %s) transitions to state(s)' % (M[0][s], M[1][a]))
# print("s' in", np.array(M[0])[np.where(M[2][a][s, :] > 0)])
#
# # Cost
# print('\n= Costs =')
# print('\nCost for the state-action pair (%s, %s):' % (M[0][s], M[1][a]))
# print('c(s, a) =', M[3][s, a])
#
#
# # Discount
# print('\n= Discount =')
# print('\ngamma =', M[4])
# ```
#
# Output:
#
```

```
# = State space (462 states) =
#
# States:
# (0, None, empty)
# (0, 1, empty)
# (0, 9, empty)
# (0, 10, empty)
# (0, 11, empty)
# (0, 18, empty)
# (0, 19, empty)
# (0, 20, empty)
# (0, 21, empty)
# (0, 23, empty)
# ...
#
# Random state: s = (7, 28, empty)
# Last state: s = (32, None, loaded)
#
# = Action space (6 actions) =
#
# Up
# Down
# Left
# Right
# Pick
# Drop
#
# Random action: a = Right
#
# = Transition probabilities =
#
# Transition probability matrix dimensions (action Up): (462, 462)
# Dimensions add up for action "Up"? True
#
# Transition probability matrix dimensions (action Down): (462, 462)
# Dimensions add up for action "Down"? True
#
# Transition probability matrix dimensions (action Left): (462, 462)
# Dimensions add up for action "Left"? True
#
# Transition probability matrix dimensions (action Right): (462, 462)
# Dimensions add up for action "Right"? True
#
# Transition probability matrix dimensions (action Pick): (462, 462)
# Dimensions add up for action "Pick"? True
#
# Transition probability matrix dimensions (action Drop): (462, 462)
# Dimensions add up for action "Drop"? True
#
# State-action pair ((7, 28, empty), Right) transitions to state(s)
# s' in ['(8, 28, empty)']
#
# = Costs =
```

#
Cost for the state-action pair ((7, 28, empty), Right):
c(s, a) = 0.501
#
= Discount =
#
gamma = 0.99
```

Note: For debug purposes, we also provide a second file, `garbage-small.npz`, that contains a 6-state MDP that you can use to verify if your results make sense.

2. Prediction
#
You are now going to evaluate a given policy, computing the corresponding cost-to-go.

#
Activity 2.
#
Write a function `noisy_policy` that builds a noisy policy "around" a provided action. Your function should receive, as input, an MDP described as a tuple like that of **Activity 1**, an integer `a`, corresponding to the index of an action in the MDP, and a real number `eps`. The function should return, as output, a policy for the provided MDP that selects action with index `a` with a probability `1 - eps` and, with probability `eps`, selects another action uniformly at random. The policy should be a `numpy` array with as many rows as states and as many columns as actions, where the element in position `[s,a]` should contain the probability of action `a` in state `s` according to the desired policy.
#
Note: The examples provided correspond for the MDP in the previous garbage collection environment. However, your code should be tested with MDPs of different sizes, so **make sure not to hard-code any of the MDP elements into your code**.
#

In[2]:


```
def noisy_policy(MDP, a, eps):
    policy = np.zeros((len(MDP[0]), len(MDP[1])))
    for i in range(0,len(policy)):
        for j in range(0, len(policy[i])):
            if j == a:
                policy[i][j] = 1 - eps
            else:
                policy[i][j] = round(eps/(len(policy[i])-1), 2)
    return policy

# Noiseless policy for action "Left" (action index: 2)
pol_noiseless = noisy_policy(M, 2, 0.)

# Arbitrary state
s = 115 # State (8, 28, empty)
```

```
# Policy at selected state
print('Arbitrary state (from previous example):', M[0][s])
print('Noiseless policy at selected state (eps=0):', pol_noiseless[s, :])

# Noisy policy for action "Left" (action index: 2)
pol_noisy = noisy_policy(M, 2, 0.1)

# Policy at selected state
print('Noisy policy at selected state (eps=0.1):', np.round(pol_noisy[s, :], 2))

# Random policy for action "Left" (action index: 2)
pol_random = noisy_policy(M, 2, 0.75)

# Policy at selected state
print('Random policy at selected state (eps=0.75):', np.round(pol_random[s, :], 2))


# We provide below an example of application of the function with MDP from the example in **Activity 1**, that you can use as a first "sanity check" for your code. Note that, as emphasized above, your function should work with **any** MDP that is specified as a tuple with the structure of the one from **Activity 1**.
#
# ```python
# # Noiseless policy for action "Left" (action index: 2)
# pol_noiseless = noisy_policy(M, 2, 0.)
#
# # Arbitrary state
# s = 115 # State (8, 28, empty)
#
# # Policy at selected state
# print('Arbitrary state (from previous example):', M[0][s])
# print('Noiseless policy at selected state (eps=0):', pol_noiseless[s, :])
#
# # Noisy policy for action "Left" (action index: 2)
# pol_noisy = noisy_policy(M, 2, 0.1)
#
# # Policy at selected state
# print('Noisy policy at selected state (eps=0.1):', np.round(pol_noisy[s, :], 2))
#
# # Random policy for action "Left" (action index: 2)
# pol_random = noisy_policy(M, 2, 0.75)
#
# # Policy at selected state
# print('Random policy at selected state (eps=0.75):', np.round(pol_random[s, :], 2))
# ```
#
# Output:
#
# ```
# Arbitrary state (from previous example): (8, 28, empty)
# Noiseless policy at selected state (eps=0): [0. 0. 1. 0. 0. 0.]
# Noisy policy at selected state (eps=0.1): [0.02 0.02 0.9  0.02 0.02 0.02]
# Random policy at selected state (eps=0.75): [0.15 0.15 0.25 0.15 0.15 0.15]
# ```
```

#
Activity 3.
#
You will now write a function called `evaluate_pol` that evaluates a given policy. Your function should receive, as an input, an MDP described as a tuple like that of **Activity 1** and a policy described as an array like that of **Activity 2** and return a `numpy` array corresponding to the cost-to-go function associated with the given policy.
#
Note: The array returned by your function should have as many rows as the number of states in the received MDP, and exactly one column. Note also that, as before, your function should work with **any** MDP that is specified as a tuple with the same structure as the one from **Activity 1**. In your solution, you may find useful the function `np.linalg.inv`, which can be used to invert a matrix.
#

In[3]:

```
def evaluate_pol(MDP,pol):
    num=len(MDP[0])
    I = np.identity(num);
    Cpi = np.zeros((num, 1));
    Ppi = np.zeros((num,  num));
    prob = np.zeros((num,  num));
    for i in range(num):
        sum=0
        for j in range (0,len(MDP[1])):
            sum+= pol[i][j] * MDP[3][i][j]
        Cpi[i] = sum

    for i in range(num):
        for j in range(num):
            sum = 0;
            for a in range(len(MDP[1])):
                p_act = MDP[2][a];
                sum += pol[i][a] * p_act[i][j];
            prob[i][j] = sum;

    CostTG = np.linalg.inv(I-MDP[4]*prob).dot(Cpi)
    return CostTG

Jact2 = evaluate_pol(M, pol_noisy)

print('Dimensions of cost-to-go:', Jact2.shape)

print('\nExample values of the computed cost-to-go:')

s = 115 # State (8, 28, empty)
print('\nCost-to-go at state %s:' % M[0][s], np.round(Jact2[s], 3))

s = 429 # (0, None, loaded)
```

```
print('Cost-to-go at state %s:' % M[0][s], np.round(Jact2[s], 3))

s = 239 # State (18, 18, empty)
print('Cost-to-go at state %s:' % M[0][s], np.round(Jact2[s], 3))

# Example with random policy

rand.seed(42)

rand_pol = rand.randint(2, size=(len(M[0]), len(M[1]))) + 0.01 # We add 0.01 to avoid all-zero
rows
rand_pol = rand_pol / rand_pol.sum(axis = 1, keepdims = True)

Jrand = evaluate_pol(M, rand_pol)

print('\nExample values of the computed cost-to-go:')

s = 115 # State (8, 28, empty)
print('\nCost-to-go at state %s:' % M[0][s], np.round(Jrand[s], 3))

s = 429 # (0, None, loaded)
print('Cost-to-go at state %s:' % M[0][s], np.round(Jrand[s], 3))

s = 239 # State (18, 18, empty)
print('Cost-to-go at state %s:' % M[0][s], np.round(Jrand[s], 3))


# As an example, you can evaluate the random policy from **Activity 2** in the MDP from
**Activity 1**.
#
# ```python
# Jact2 = evaluate_pol(M, pol_noisy)
#
# print('Dimensions of cost-to-go:', Jact2.shape)
#
# print('\nExample values of the computed cost-to-go:')
#
# s = 115 # State (8, 28, empty)
# print('\nCost-to-go at state %s:' % M[0][s], np.round(Jact2[s], 3))
#
# s = 429 # (0, None, loaded)
# print('Cost-to-go at state %s:' % M[0][s], np.round(Jact2[s], 3))
#
# s = 239 # State (18, 18, empty)
# print('Cost-to-go at state %s:' % M[0][s], np.round(Jact2[s], 3))
#
# # Example with random policy
#
# rand.seed(42)
#
# rand_pol = rand.randint(2, size=(len(M[0]), len(M[1]))) + 0.01 # We add 0.01 to avoid all-zero
rows
# rand_pol = rand_pol / rand_pol.sum(axis = 1, keepdims = True)
#
```

```
# Jrand = evaluate_pol(M, rand_pol)
#
# print('\nExample values of the computed cost-to-go:')
#
# s = 115 # State (8, 28, empty)
# print('\nCost-to-go at state %s:' % M[0][s], np.round(Jrand[s], 3))
#
# s = 429 # (0, None, loaded)
# print('Cost-to-go at state %s:' % M[0][s], np.round(Jrand[s], 3))
#
# s = 239 # State (18, 18, empty)
# print('Cost-to-go at state %s:' % M[0][s], np.round(Jrand[s], 3))
# ```
#
# Output:
# ```
# Dimensions of cost-to-go: (462, 1)
#
# Example values of the computed cost-to-go:
#
# Cost-to-go at state (8, 28, empty): [94.678]
# Cost-to-go at state (0, None, loaded): [97.36]
# Cost-to-go at state (18, 18, empty): [95.865]
#
# Example values of the computed cost-to-go:
#
# Cost-to-go at state (8, 28, empty): [97.097]
# Cost-to-go at state (0, None, loaded): [91.185]
# Cost-to-go at state (18, 18, empty): [84.853]
# ```

# ### 3. Control
#
# In this section you are going to compare value and policy iteration, both in terms of time and number of iterations.

# ---
#
# #### Activity 4
#
# In this activity you will show that the policy in Activity 3 is _not_ optimal. For that purpose, you will use value iteration to compute the optimal cost-to-go, $J^*$, and show that $J^*\neq J^\pi$.
#
# Write a function called `value_iteration` that receives as input an MDP represented as a tuple like that of **Activity 1** and returns an `numpy` array corresponding to the optimal cost-to-go function associated with that MDP. Before returning, your function should print:
#
# * The time it took to run, in the format `Execution time: xxx seconds`, where `xxx` represents the number of seconds rounded up to $3$ decimal places.
# * The number of iterations, in the format `N. iterations: xxx`, where `xxx` represents the number of iterations.
#
```

Note 1: Stop the algorithm when the error between iterations is smaller than 10^{-8}. To compute the error between iterations, you should use the function `norm` from `numpy.linalg`.
#
Note 2: You may find useful the function ``time()`` from the module ``time``. You may also find useful the code provided in the theoretical lecture.
#
Note 3: The array returned by your function should have as many rows as the number of states in the received MDP, and exactly one column. As before, your function should work with **any** MDP that is specified as a tuple with the same structure as the one from **Activity 1**.
#
#

In[4]:

```
import time as t

def value_iteration(MDP):
    error = 1
    ite = 0;
    numStt = len(MDP[0])
    numAct = len(MDP[1])
    J = np.zeros((numStt, 1))
    start = t.time()

    while error > pow(10,-8):
        res = list();

        for a in range(0, numAct):
            c_a = MDP[3][:,a]
            c_a = c_a.reshape((numStt,1))
            Qa = c_a + MDP[4] * MDP[2][a].dot(J)
            res.append(Qa)
        Jtemp = np.min(res, axis=0)

        error = np.linalg.norm(Jtemp - J)
        ite +=1
        J = Jtemp

    print("Execution time:", np.round(t.time() - start, 3), "seconds");
    print("N. iterations:", ite);

    return J;

Jopt = value_iteration(M)

print('\nDimensions of cost-to-go:', Jopt.shape)

print('\nExample values of the optimal cost-to-go:')

s = 115 # State (8, 28, empty)
print('\nCost to go at state %s:' % M[0][s], Jopt[s])
```

```
s = 429 # (0, None, loaded)
print('Cost to go at state %s:' % M[0][s], Jopt[s])

s = 239 # State (18, 18, empty)
print('Cost to go at state %s:' % M[0][s], Jopt[s])

print('\nIs the policy from Activity 2 optimal?', np.all(np.isclose(Jopt, Jact2)))


# For example, using the MDP from **Activity 1** you could obtain the following interaction.
#
# ```python
# Jopt = value_iteration(M)
#
# print('\nDimensions of cost-to-go:', Jopt.shape)
#
# print('\nExample values of the optimal cost-to-go:')
#
# s = 115 # State (8, 28, empty)
# print('\nCost to go at state %s:' % M[0][s], Jopt[s])
#
# s = 429 # (0, None, loaded)
# print('Cost to go at state %s:' % M[0][s], Jopt[s])
#
# s = 239 # State (18, 18, empty)
# print('Cost to go at state %s:' % M[0][s], Jopt[s])
#
# print('\nIs the policy from Activity 2 optimal?', np.all(np.isclose(Jopt, Jact2)))
# ```
#
# Output:
# ```
# Execution time: 0.535 seconds
# N. iterations: 2045
#
# Dimensions of cost-to-go: (462, 1)
#
# Example values of the optimal cost-to-go:
#
# Cost to go at state (8, 28, empty): [39.73849429]
# Cost to go at state (0, None, loaded): [37.69488235]
# Cost to go at state (18, 18, empty): [38.18476409]
#
# Is the policy from Activity 2 optimal? False
# ```

# ---
#
# #### Activity 5
#
# You will now compute the optimal policy using policy iteration. Write a function called `policy_iteration` that receives as input an MDP represented as a tuple like that of **Activity 1** and returns an `numpy` array corresponding to the optimal policy associated with that MDP. Before returning, your function should print:
```

* The time it took to run, in the format `Execution time: xxx seconds`, where `xxx` represents the number of seconds rounded up to 3 decimal places.
* The number of iterations, in the format `N. iterations: xxx`, where `xxx` represents the number of iterations.
#
Note: If you find that numerical errors affect your computations (especially when comparing two values/arrays) you may use the `numpy` function `isclose` with adequately set absolute and relative tolerance parameters (e.g., 10^{-8}). You may also find useful the code provided in the theoretical lecture.
#

In[7]:

```
#Activity 5
import time as t
import math

def policy_iteration(MDP):
    X = M[0]
    A = M[1]
    P = M[2]
    c = M[3]
    y = M[4]
    k = 0
    start_time = t.time()
    pol = np.ones((len(X), len(A))) / len(A)
    quit = False
    while not quit:
        Q = np.zeros((len(X), len(A)))
        c_pi = np.sum(c*pol, axis=1, keepdims = True)
        P_pi = pol[:, 0, None] * P[0]

        for a in range(1, len(A)):
            P_pi += pol[:, a, None] * P[a]
        J = np.linalg.inv(np.eye(len(X)) - y * P_pi).dot(c_pi)

        for a in range(len(A)):
            Q[:, a, None] = c[:, a, None] + y * P[a].dot(J)
        Qmin = np.min(Q, axis=1, keepdims = True)
        p_new = np.isclose(Q, Qmin, atol=1e-8, rtol=1e-8).astype(int)
        p_new = p_new / p_new.sum(axis=1, keepdims = True)
        quit = (pol == p_new).all()

        pol = p_new
        k+=1

    print('Execution time: ', round(t.time() - start_time, 3))
    print('N. iterations: ', k)
    return pol

popt = policy_iteration(M)
```

```
print('\nDimension of the policy matrix:', popt.shape)

rand.seed(42)

print('\nExamples of actions according to the optimal policy:')

# Select random state, and action using the policy computed
s = 115 # State (8, 28, empty)
a = rand.choice(len(M[1]), p=popt[s, :])
print('Policy at state %s: %s' % (M[0][s], M[1][a]))

# Select random state, and action using the policy computed
s = 429 # (0, None, loaded)
a = rand.choice(len(M[1]), p=popt[s, :])
print('Policy at state %s: %s' % (M[0][s], M[1][a]))

# Select random state, and action using the policy computed
s = 239 # State (18, 18, empty)
a = rand.choice(len(M[1]), p=popt[s, :])
print('Policy at state %s: %s' % (M[0][s], M[1][a]))

# Verify optimality of the computed policy

print('\nOptimality of the computed policy:')

Jpi = evaluate_pol(M, popt)
print('- Is the new policy optimal?', np.all(np.isclose(Jopt, Jpi)))


# For example, using the MDP from **Activity 1** you could obtain the following interaction.
#
# ```python
# popt = policy_iteration(M)
#
# print('\nDimension of the policy matrix:', popt.shape)
#
# rand.seed(42)
#
# print('\nExamples of actions according to the optimal policy:')
#
# # Select random state, and action using the policy computed
# s = 115 # State (8, 28, empty)
# a = rand.choice(len(M[1]), p=popt[s, :])
# print('Policy at state %s: %s' % (M[0][s], M[1][a]))
#
# # Select random state, and action using the policy computed
# s = 429 # (0, None, loaded)
# a = rand.choice(len(M[1]), p=popt[s, :])
# print('Policy at state %s: %s' % (M[0][s], M[1][a]))
#
# # Select random state, and action using the policy computed
# s = 239 # State (18, 18, empty)
# a = rand.choice(len(M[1]), p=popt[s, :])
```

print('Policy at state %s: %s' % (M[0][s], M[1][a]))
#
Verify optimality of the computed policy
#
print('\nOptimality of the computed policy:')
#
Jpi = evaluate_pol(M, popt)
print('- Is the new policy optimal?', np.all(np.isclose(Jopt, Jpi)))
```
#
Output:
```
Execution time: 0.063 seconds
N. iterations: 11
#
Dimension of the policy matrix: (462, 6)
#
Examples of actions according to the optimal policy:
Policy at state (8, 28, empty): Down
Policy at state (0, None, loaded): Drop
Policy at state (18, 18, empty): Pick
#
Optimality of the computed policy:
- Is the new policy optimal? True
```

4. Simulation
#
Finally, in this section you will check whether the theoretical computations of the cost-to-go actually correspond to the cost incurred by an agent following a policy.

#
Activity 6
#
Write a function `simulate` that receives, as inputs
#
* An MDP represented as a tuple like that of **Activity 1**;
* A policy, represented as an `numpy` array like that of **Activity 2**;
* An integer, `x0`, corresponding to a state index
* A second integer, `length`
#
Your function should return, as an output, a float corresponding to the estimated cost-to-go associated with the provided policy at the provided state. To estimate such cost-to-go, your function should:
#
* Generate **`NRUNS`** trajectories of `length` steps each, starting in the provided state and following the provided policy.
* For each trajectory, compute the accumulated (discounted) cost.
* Compute the average cost over the 100 trajectories.
#
Note 1: You may find useful to import the numpy module `numpy.random`.
#

Note 2: Each simulation may take a bit of time, don't despair ☺️.
#

In[6]:


```
NRUNS = 100 # Do not delete this

def simulate(M, P_pi, x0, length):
    X = M[0];
    A = M[1];
    P = M[2];
    c = M[3];
    y = M[4];

    final_cost=0;

    for i in range(NRUNS):
        x_j = x0;
        accumulated_gamma=1;
        cost_i=0;
        for j in range(length):
            # choose length actions for each state i
            a_j = rand.choice(len(A), p= P_pi[x_j, :]);
            #cost associated with the chosen action at the given satte
            cost_i += accumulated_gamma * c[x_j,a_j];
            #next state
            x_j = rand.choice(len(X), p=P[a_j][x_j, :]);

            accumulated_gamma *= y;
        final_cost += cost_i;
    return final_cost/ NRUNS;

rand.seed(42)

# Select arbitrary state, and evaluate for the optimal policy
s = 115 # State (8, 28, empty)
print('Cost-to-go for state %s:' % M[0][s])
print('\tTheoretical:', np.round(Jopt[s], 4))
print('\tEmpirical:', np.round(simulate(M, popt, s, 1000), 4))

# Select arbitrary state, and evaluate for the optimal policy
s = 429 # (0, None, loaded)
print('Cost-to-go for state %s:' % M[0][s])
print('\tTheoretical:', np.round(Jopt[s], 4))
print('\tEmpirical:', np.round(simulate(M, popt, s, 1000), 4))

# Select arbitrary state, and evaluate for the optimal policy
s = 239 # State (18, 18, empty)
print('Cost-to-go for state %s:' % M[0][s])
print('\tTheoretical:', np.round(Jopt[s], 4))
print('\tEmpirical:', np.round(simulate(M, popt, s, 1000), 4))
```

Insert your code here.

For example, we can use this function to estimate the values of some random states and compare them with those from **Activity 4**.
#
```python
rand.seed(42)
#
Select arbitrary state, and evaluate for the optimal policy
s = 115 # State (8, 28, empty)
print('Cost-to-go for state %s:' % M[0][s])
print('\tTheoretical:', np.round(Jopt[s], 4))
print('\tEmpirical:', np.round(simulate(M, popt, s, 1000), 4))
#
Select arbitrary state, and evaluate for the optimal policy
s = 429 # (0, None, loaded)
print('Cost-to-go for state %s:' % M[0][s])
print('\tTheoretical:', np.round(Jopt[s], 4))
print('\tEmpirical:', np.round(simulate(M, popt, s, 1000), 4))
#
Select arbitrary state, and evaluate for the optimal policy
s = 239 # State (18, 18, empty)
print('Cost-to-go for state %s:' % M[0][s])
print('\tTheoretical:', np.round(Jopt[s], 4))
print('\tEmpirical:', np.round(simulate(M, popt, s, 1000), 4))
```
#
Output:
```
Cost-to-go for state (8, 28, empty):
Theoretical: [39.7385]
Empirical: 39.6626
Cost-to-go for state (0, None, loaded):
Theoretical: [37.6949]
Empirical: 37.5146
Cost-to-go for state (18, 18, empty):
Theoretical: [38.1848]
Empirical: 37.7935
```