# Planning, Learning and Decision Making

Group 27

78375 - João Pirralha

84758 - Rafael Ribeiro

## Homework 2. Markov decision problems

Consider an agent moving in the grid-world environment of Fig. 1. The agent must reach the goal cell, marked with “G”.

At each step, the agent may move in any of the four directions—up, down, left and right. Movement across a grey cell division succeeds with a 0.8 probability and fails with a 0.2 probability. Movements across colored cell divisions (blue or red) succeed with a 0.8 probability only if the agent has the corresponding colored key. Otherwise, they fail with probability 1. When the movement fails, the agent remains in the same cell.

To get a colored key, the agent simply needs to stand in the corresponding cell. In other words, as soon as the agent stands on the cell of a colored key, you may consider that it holds that key thereafter.

---

### Exercise 1.

---

**Important Note**

We assigned each grid position of the environment, a letter, from A to G, such that, the newly drawn board is as presented below:

```
| A | B | E |
    | C | F | G |
    | D |
```

B0 means in cell B with no keys; B1 in cell B with the red key; and B2 in cell B with both keys.

### 1.a.

### Identify the state space, X , and the action space, A, for the MDP. Assume that the agent never has the blue key without the red key and never reaches the goal without both keys.

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

#States
X = ('B0', 'C0', 'E0', 'F0', 'B1', 'C1', 'D1', 'E1', 'F1', 'A2', 'B2', '
C2', 'D2', 'E2', 'F2', 'G2')
print("States: {}".format(X))

#Actions
A = ('U', 'D', 'L', 'R')
print("Actions: {}".format(A))
```

```
States: ('B0', 'C0', 'E0', 'F0', 'B1', 'C1', 'D1', 'E1', 'F1', 'A2', 'B2'
, 'C2', 'D2', 'E2', 'F2', 'G2')
Actions: ('U', 'D', 'L', 'R')
```

---

**1.b.**

**Write down the transition probability matrix for the action “right” and a (possible) cost function for the MDP.**

**Make sure that the cost function is as simple as possible and verifies c(x, a) ∈ [0, 1] for all states x ∈ X and actions a ∈ A. Note, in particular, that the cost should depend only on the agent standing in the goal cell.**

In [2]:
```
#Outcomes
#Action R(ight)
PR = np.zeros((len(X), (len(X))))
PR[0, 0] = 0.2
PR[0, 2] = 0.8
PR[1, 1] = 0.2
PR[1, 3] = 0.8
PR[2, 2] = 1
PR[3, 3] = 1
PR[4, 4] = 0.2
PR[4, 7] = 0.8
PR[5, 5] = 0.2
PR[5, 8] = 0.8
PR[6, 6] = 1
PR[7, 7] = 1
PR[8, 8] = 1
PR[9, 9] = 0.2
PR[9, 10] = 0.8
PR[10, 10] = 0.2
PR[10, 13] = 0.8
PR[11, 11] = 0.2
PR[11, 14] = 0.8
PR[12, 12] = 1
PR[13, 13] = 1
PR[14, 14] = 0.2
PR[14, 15] = 0.8
PR[15, 15] = 1
print("Transition Probability Matrix for Right action: \n{}\n".format(PR
))

#Cost
C = np.zeros((len(X), len(A)))
C[:15] = 1
C[14, 3] = 0
C[15, 0:2] = 0
C[15, 3] = 0
C[15, 2] = 1
C = np.round(C / np.max(C), 3) #Making sure all the costs are between 0
and 1, in case a weight is changed acidentaly
print("Cost Function: \n{}\n".format(C))
```

```
Transition Probability Matrix for Right action:
[[0.2 0.  0.8 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.2 0.  0.8 0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.2 0.  0.  0.8 0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.2 0.  0.  0.8 0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.2 0.8 0.  0.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.2 0.  0.  0.8 0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.2 0.  0.  0.8 0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1.  0.  0. ]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.2 0.8]
 [0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  0.  1. ]]

Cost Function:
[[1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 1.]
 [1. 1. 1. 0.]
 [0. 0. 1. 0.]]
```

---

**1.c.**

**Compute the cost-to-go function associated with the policy in which the agent always goes right, using a discount γ = 0.9. You can use any software of your liking for the harder computations, but should indicate all other computations.**

Using the cost function defined in 1.b.:

- $J^{\rightarrow}_{G2} = 0 \ (goal \ state)$
- $J^{\rightarrow}_{F2} = 0.8 \times 0 + 0.2 \times (0 + \gamma \times (0.8 \times 0) + \gamma \times (0.2 \times (0 + \gamma \times (0.8 \times 0) + \gamma \times (0.2 \times (0 + \ldots))))) = 0$
  - $J^{\rightarrow}_{C2} = 0.8 \times (1 + \gamma \times J^{\rightarrow}_{F2}) + 0.2 \times (1 + \gamma \times (0.8 \times (1 + \gamma \times J^{\rightarrow}_{F2})) + \gamma \times (0.2 \times (1 + \ldots))$
    $= 0.8 \times (1 + \gamma \times J^{\rightarrow}_{F2}) + 0.2 + (0.2 \times \gamma) \times (0.8 \times (1 + \gamma \times J^{\rightarrow}_{F2})) + (0.2 \times \gamma) \times 0.2 + (0.2 \times \gamma)^2 \times (0.8 \times (1 + \gamma \times J^{\rightarrow}_{F2})) + \ldots$

$$
\begin{aligned}
&= (0.8 \times (1 + \gamma \times J^{\rightarrow}_{F2}) + 0.2 \times 1) \times (1 + (0.2 \times \gamma) + (0.2 \times \gamma)^2 + \ldots) \\
&= (0.8 \times (1 + \gamma \times J^{\rightarrow}_{F2}) + 0.2 \times 1) \times \sum_{n=0}^{\infty} (0.2 \times \gamma)^n \\
&= (0.8 \times (1 + \gamma \times J^{\rightarrow}_{F2}) + 0.2 \times 1) \times \frac{1}{1 - 0.2 \times \gamma} \\
&= (0.8 \times (1 + 0.9 \times 0) + 0.2 \times 1) \times \frac{1}{1 - 0.2 \times 0.9} \\
&= (0.8 + 0.2) \times \frac{1}{1 - 0.2 \times 0.9} \\
&\approx 1.2195
\end{aligned}
$$

The remaining states will never reach the goal state with the policy in which the agent always goes right, so the agent will be penalized forever:

- $J^{\rightarrow}_{remaining} = 1 + 1 \times \gamma + 1 \times \gamma^2 + \ldots = \sum_{n=0}^{\infty} \gamma^n = \frac{1}{1-\gamma} = 10$

Thus:

$$
J^{\rightarrow} = \begin{bmatrix}
10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 10 \\ 1.2195 \\ 10 \\ 10 \\ 0 \\ 0
\end{bmatrix}
$$

Next we experimentally validade our answer for $J^{\rightarrow}_{F2}$ and $J^{\rightarrow}_{C2}$, which are the most difficult:

In [3]:
```
# Simulation for validation
ITERATIONS = 10**5
GAMMA = 0.9
ACTION = 3 # Always move right
simStates = np.arange(16)

def simulate(initialState):
    cost = 0
    for i in range(ITERATIONS):
        state = initialState
        j = 0
        while state != 15:
            cost += C[state, ACTION] * GAMMA ** j
            state = np.random.choice(simStates, p=PR[state])
            j += 1
    return cost / ITERATIONS

print("(Simulation) J→F2:\t {}".format(round(simulate(14), 4)))
print("(Simulation) J→C2:\t {}".format(round(simulate(11), 4)))
```

```
(Simulation) J→F2:	 0.0
(Simulation) J→C2:	 1.2173
```

Alternatively, we can also compute J systematically using the second equation in slide 39 of lec8.pdf:

In [4]:
```
J = np.linalg.inv(np.eye(len(X)) - 0.9 * PR) @ C[:,3]
print(J.reshape((len(X), 1)))
```

```
[[10.       ]
 [10.       ]
 [10.       ]
 [10.       ]
 [10.       ]
 [10.       ]
 [10.       ]
 [10.       ]
 [10.       ]
 [10.       ]
 [ 1.2195122]
 [10.       ]
 [10.       ]
 [ 0.       ]
 [ 0.       ]]
```