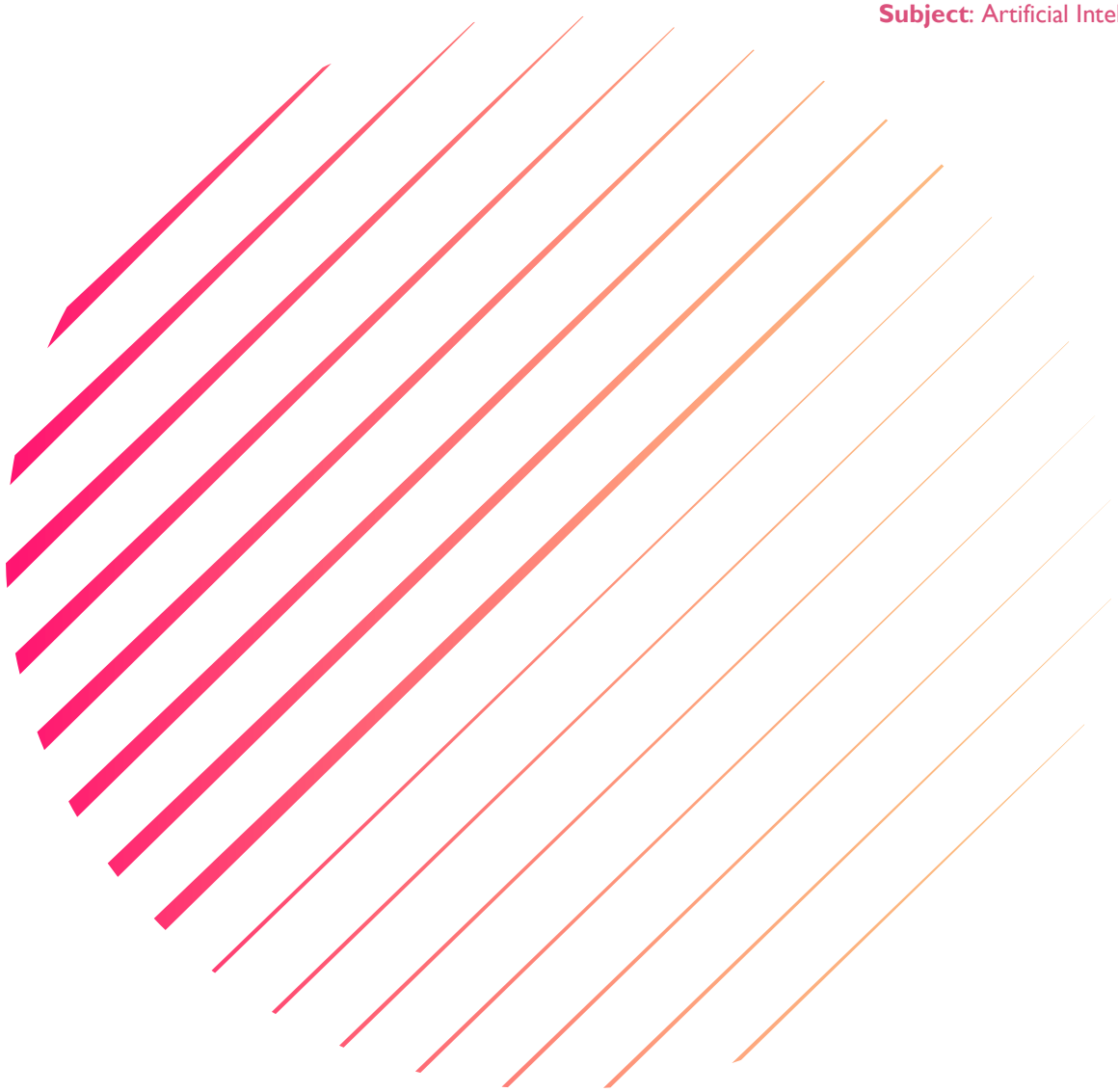
FINAL PROJECT REPORT

Markov Decision Process for Temperature Control

Degree: Applied Mathematics and Computing
Subject: Artificial Intelligence



Name: Sheila Gallardo Redondo
NIA:100472913
Group: 121

Name: Alexandra Perruchot-Triboulet Rodríguez
NIA:100453024
Group: 121

Executive Summary

Objectives 3

Formal Description of the MDP Model 3

Definition of our MDP 3

MDP is represented by a Graph 4

Bellman's Equations 4

Optimal Policy 5

Cost Model and Optimal Policy Analysis 5

Project Phases 7

Design 7

Implementation 8

Conclusions 10

Objectives

The main objective of this project is to build a thermostat that optimises energy consumption without forgetting the comfort of the user. We thought of doing it in a general way so that it works for any MDP we want to build, i.e., for any input states, actions, costs, transitions, and goals.

Formal Description of the MDP Model

Definition of our MDP

Our MDP consists of: < *S*, *A*, *T*, *C* >

- **Set of States** (*S*) : we have consider our states to be the possible temperatures of the room. These temperatures are within the range of 16°C and 25°C with a difference of 0.5°C between each state. This gives us a total of 19 states, that we have called "gradosxx", where xx corresponds to the actual number of degrees of the room.

$$S = \{\text{'grados16'}, \text{'grados16.5'}, \text{'grados17'}, \text{'grados17.5'}, \ldots, \text{'grados25'}\}$$

- **Set of Actions** (*A*): these are the possible actions of the thermostat. The thermostat must decide each half an hour whether to switch the heating *on* or *off*.

$$A = \{\text{'on'}, \text{'off'}\}$$

- **Set of transitions** (*T*): a transition (*t*) is defined by an origin state, destination state, the action applied, and the probability associated to the action.

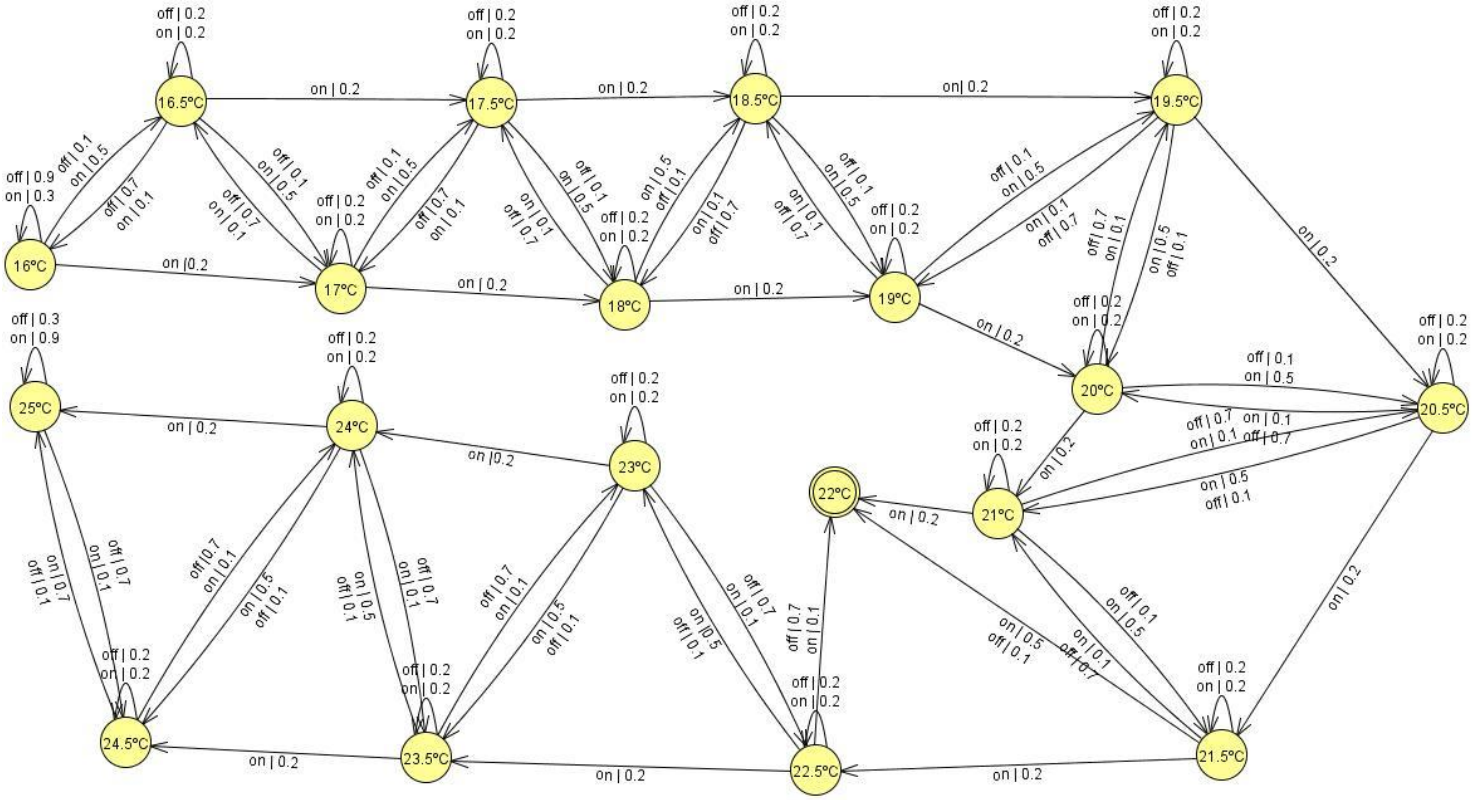
$$t = \langle origin\ state \rangle + \langle action \rangle + \langle probability \rangle \rightarrow \langle destination\ state \rangle$$

$$T = \{t_1, t_2, \ldots, t_n\}$$

- **Costs** (*C*): each action has a specific cost associated to it. We have created a dictionary called 'costs' which associates actions (dictionary keys) with their respectively cost (dictionary values).

$$C = \{\text{'on'}: < cost\ for\ action\ on >, \text{'off'}: < cost\ for\ action\ off >\}$$

MDP is represented by a Graph



Bellman's Equations

We have implemented general bellman's equations for each one of our states, so that we could compute their expected values iteratively. In our model, the Bellman's Equations look like this:

$$V_{i+1}(s) = \min\{cost(on) + \sum_{s' \in S} P(s'|s, on) * V_i(s'), cost(off) + \sum_{s' \in S} P(s'|s, off) * V_i(s')\}$$

where:

- s' is the arrival state.
- $P(s'|s, < action >)$ is the probability of arriving to state s' from state s applying the corresponding action.

We set up the expected value of our goal ('grados22') to zero, since, by definition, the expected value of a state is the cost to arrive to the goal from that state: V('grados22') = 0.
We set the expected value of all states at the first iteration (iteration 0) to zero: $V_0(s) = 0, \forall s \in S$.

To compute the expected values, we apply the Value Iteration algorithm. This algorithm ends when each one of the states' expected values reaches a fix point (in our case when we obtain a precision of 0.0001 for each one).

Optimal Policy

Once we have the expected value for each one of our states, we can compute the optimal policy. In this case, the optimal policy equation is like that:

$$\pi^*(s) = \arg min\ \{cost(on) + \sum_{s' \in S} P(s'|s, on) * V(s'), cost(off) + \sum_{s' \in S} P(s'|s, off) * V(s')\}$$

This will give us the optimal action to be applied from each state to arrive to our goal. This will be the action which gives us a lower cost.

Cost Model and Optimal Policy Analysis

Firstly, we have calculated the mean cost to maintain the thermostat switched on in Spain per half an hour, this is 0.1675 €. With that, we have considered different cases and calculated the cost of each one of the actions applied to the thermostat.

1. Here we have considered the 'base' case, where we consider the price of having the thermostat switched on (0.1675€) and the fact that the person needs to wait half an hour (0.5) to change the state of the thermostat ('on' or 'off').

   Costs: { 'on': 0.5 + 0.1675, 'off': 0.5 + 0}

This gives us the following results:

States	Optimal Policy ($\pi^*(s)$)	Cost (to 3.d.p)
'grados16'	'on'	10.239
'grados16.5'	'on'	9.519
'grados17'	'on'	8.701
'grados17.5'	'on'	7.869
'grados18'	'on'	7.035
'grados18.5'	'on'	6.200
'grados19'	'on'	5.365
'grados19.5'	'on'	4.532
'grados20'	'on'	3.695
'grados20.5'	'on'	2.870
'grados21'	'on'	2.001
'grados21.5'	'on'	1.293
'grados22'	None	0.0
'grados22.5'	'off'	0.833
'grados23'	'off'	1.667
'grados23.5'	'off'	2.500
'grados24'	'off'	3.330
'grados24.5'	'off'	4.147
'grados25'	'off'	4.861

In the table above, we can observe how as we approach the goal, the cost, regardless of the action applied, is reduced. Besides this, we can see that when we apply action 'on' and action 'off' from states that are at the same distance from the goal, action 'on' has always the higher cost.

2. In this second case, we have the same situation as before, but we give much less importance to the time the person is waiting (0.0005).

   Costs: { 'on': 0.0005 + 0.1675, 'off': 0.0005 + 0}

This gives us the following results:

States	Optimal Policy ($\pi^*(s)$)	Cost (to 4.d.p)
'grados16'	'off'	2.1533
'grados16.5'	'off'	2.1493
'grados17'	'on'	2.1171
'grados17.5'	'on'	1.9325
'grados18'	'on'	1.7250
'grados18.5'	'on'	1.5153
'grados19'	'on'	1.3054
'grados19.5'	'on'	1.0960
'grados20'	'on'	0.8852
'grados20.5'	'on'	0.6764
'grados21'	'on'	0.4620
'grados21.5'	'on'	0.2680
'grados22'	None	0.0
'grados22.5'	'off'	0.0008
'grados23'	'off'	0.0017
'grados23.5'	'off'	0.0025
'grados24'	'off'	0.0033
'grados24.5'	'off'	0.0041
'grados25'	'off'	0.0049

Comparing the 2 tables above, we can observe how the optimal policy for each state changes with respect to the cost of the action. Since we are given less importance to the waiting time, this leads to a change in the optimal policy of the two first states of our MDP ('grados16' and 'grades16.5'), which now have as optimal policy 'off' action.

3. In this case, we look at the benefit of the electricity company. For that reason, we give a higher cost to the situation where the thermostat is turned off. We add the price of light to the cost of the 'off' action because it's the money the company is losing.

   Costs: {'on': 0.0005 + 0, 'off': 0.0005 + 0.1675}

This gives us the following results:

States	Optimal Policy ($\pi^*(s)$)	Cost (to 3.d.p)
'grados16'	'on'	0.068
'grados16.5'	'on'	0.067
'grados17'	'on'	0.067
'grados17.5'	'on'	0.066
'grados18'	'on'	0.065
'grados18.5'	'on'	0.065
'grados19'	'on'	0.064
'grados19.5'	'on'	0.064
'grados20'	'on'	0.063
'grados20.5'	'on'	0.064
'grados21'	'on'	0.057
'grados21.5'	'on'	0.078
'grados22'	None	0.0
'grados22.5'	'off'	0.280
'grados23'	'off'	0.559
'grados23.5'	'off'	0.833
'grados24'	'on'	1.069
'grados24.5'	'on'	1.101
'grados25'	'on'	1.105

Comparing this table with the one from the first case, we can see again how the optimal policy for each state changes with respect to the cost of the action. Since we are given more importance to the benefit of the company, this leads to a change in the optimal policy of the last three states of our MDP ('grados24', 'grados24.5' and 'grades25'), which now have as optimal policy 'on' action.

Project Phases

Design

To implement the MDP with python, we have decided to create one text file and two python scripts.
In the text file "**Transitions.txt**" we have written the transitions of our model to then read it and extract the information.
In the first python file "**Functions_MDP.py**" we have implemented two classes called 'Transition' and 'MDP' (which includes the function 'solve_bellman' that obtains the expected values and the optimal policy of its states given the goal and a dictionary of costs) and three functions outside this two classes called 'read_file' (in charge of reading and extracting the information stored in any input file), 'create_MDP' (in charge of creating an object of type 'MDP' to create a Markov Decision Process Model, taking as inputs a set of states, a set actions and a list of objects of type 'Transition' which will be the corresponding transitions of our MDP) and 'solve_MDP' (which is the function that calls all functions to solve the MDP, given as input the text file with the transitions, the dictionary of costs and the goal).
In the second python file "**main.py**", we import the function 'solve_MDP' from the first script to solve the MDP of our specific case. We define the costs (for actions 'on' and 'off' respectively), the goal

('grados22') and we pass as input the text file we have previously created defining our transitions. Finally, we print the results.

Implementation

- Transitions.txt

  Here, we define the transitions as follow:

  <origin state>--<action>--<probability>--<arrival state>

- Functions_MDP.py

  In the **class 'Transition'**, we simply define it to store the origin state, the action applied from it, the probability associated to it and the arrival state.
  Inside the **'MDP' class**, we define the class with a dictionary as the attribute, where we are going to associate each state with another dictionary where its keys are the actions, and the values are empty lists to store a list with the destination state and the probability.
  We have implemented two functions inside 'MDP', **'add_transition'**, that receives as input an object of type 'Transition' and stores the destination state and probability in a list to then store it in the corresponding list of the dictionary (where the origin state and action are the keys).
  The second function is **'solve_bellman'**, which receives as inputs a dictionary associating the actions (keys) to their corresponding cost (values) and the goal state. In this function, we have first created a dictionary 'bellman' to store the results of bellman's equations and be able to access them, the states will be the keys and the expected values the corresponding values.
  Initially, we set those values to zero (since we are at iteration 0).
  Moreover, we create two additional dictionaries, 'bellman_aux' where we are going to store the newly computed expected values and 'optimal_policy' to store the action corresponding to the optimal policy.
  To represent the value iteration algorithm, we have implemented a while loop without condition, to compute the expected values, and we stop when we arrive to the precision we want (0.0001).
  We define an auxiliary variable 'cont' (which we will use later) and set its value to False.
  Then, we implement a for loop to go over every state in the set of states and compute its expected value. First, we check if the state is the goal state ('grados22') and if so, we set its expected value to zero (bellman_aux['grados22'] = 0) and the optimal policy to None (optimal_policy['grados22'] = None). If it's not the case, we compute its expected value using the bellman's equation. To do that, we create another dictionary 'eq', where the keys will be the actions and its values the part of the bellman's equation that corresponds to that action, we do this so that later we can compare the values and take the minimum one.
  We implement another for loop to go through all actions and compute its part of the bellman's equation (checking that there exists any transition with this state and action). We do that by adding the cost of that action with the probability times the previous expected value of the arrival state. To do this, we have implemented another for loop so that it goes through all transitions stored in the corresponding position of the dictionary defined in the class (self._states) and when we have calculated all the values, we take the key (the action) with minimum value (min_key), store it in 'optimal_policy' and the corresponding value (eq[min_key]) in 'bellman_aux'.
  Furthermore, we have implemented another for loop (inside the while) to check if the expected value of each state has reached the precision established before. If one of the expected values doesn't fulfil this precision, we set the variable cont to True and we save the values stored in the 'bellman_aux' dictionary into the 'bellman' dictionary, to use them in the next iteration. Finally, we

check the value of the cont variable, if this one is True, we continue iterating, but, if this one is False, we exit the while loop and return the 'bellman' dictionary containing the expected values of each state and the 'optimal_policy' dictionary containing the optimal policy of each state.

Outside the 'MDP' class, we have implemented three more functions. The first one is the **'read_file'** function, which reads the file passed as a parameter. It opens the file and splits its content by lines, we store them in a vector and implement a for loop to go line by line and split them again every time we find '--'. We do this so that we can have each part of all transitions separately and be able to store all states in the **'states'** set (defined previously) and all actions in the **'actions'** set (defined previously). We then define a 'Transition' type object having as attributes each one of the parts mentioned before and store it in **'list_transitions'** (defined previously). Once we have read all the file, we return 'states', 'actions' and 'list_transitions'.
The second function is **'create_MDP'**. Inside this one, we create a 'MDP' type object (**'mdp'**) with the input parameters of the function (a set of states and a set of actions) as attributes. Then, we implement a for loop to add each transition (with the 'add_transition' function defined previously) from a list of transitions (which was passed as input to the function) to the 'mdp'. Finally, we return the 'mdp' object.
The last one is **'solve_MDP'** which is the function that calls every function mentioned before. Firstly, it calls 'read_file' with the file passed as an input parameter ('**file_name**'), its returned values are passed as inputs to the 'create_MDP' function. Once the 'mdp' has been created, we call the 'solve_bellman' function which takes as inputs the '**costs**' and '**goal**' parameters passed to 'solve_MDP'. Finally, we return the output values of 'solve_bellman'.

- main.py

  We import the 'solve_MDP' function from the previous script ('Functions_MDP.py'). Inside the main, we define the dictionary of costs ('costs') and the goal (goal = 'grados22') of our MDP. Now, we call 'solve_MDP' passing as input parameters: "Transitions.txt", 'costs' and the goal. Finally, we print the results.

Testing

To check that our general MDP solver works correctly for any general case, we have taken as a MDP one that we did in class as an exercise. Since we know the results we should obtain when computing the expected values and optimal policy, we can check if our 'solver_MDP' function works correctly.
We have written the transitions of this MDP in a new text file ('test.txt') and define the new costs and goal:

- costs = {'p': 1, 'q': 1}
- goal = 'C'

In this example, the set of states, the set of actions and the list of transitions are:

- states = {'A', 'B', 'C'}
- actions = {'p', 'q'}
- list of transitions: [A--p--0.8--B, A--p--0.2--A, A--q--0.1--C, A--q--0.9--A, B--q--0.9--C,B--q--0.1--A]

Below, there's a table to compare the expected results (the ones given in the solution of the exercise) and the actual results (the ones given by our 'solve_MDP' function):

	Expected Result	Actual Result
V(A)	2.4	2.499
V(B)	1.2	1.249
V(C)	0	0
$\pi^*(A)$	'p'	'p'
$\pi^*(B)$	'q'	'q'
$\pi^*(C)$	Not defined (None)	None

As we can observe, the expected and actual results are the same, which proves that our solve_MDP function works for any MDP.

Conclusions

We think this assignment helped us to understand deeply about Markov Decision Processes.
At first, we struggled in how to represent this model in python. Our first approach was to build matrices representing the Conditional Probability Tables, one per action. As this had a very high complexity ($O(n^4)$) and it was very difficult to access the data needed, we implemented the functions and classes mentioned in previous sections instead. With this new implementation, we simulate the representation of a graph to represent our MDP. This approach has a lower complexity (higher efficiency), it makes it easier to extract the data we want, and, to understand the code, it is simpler.
Moreover, this project has helped us in the computational thinking point of view, we now know that to carry out a task like this one, we should first come up with a plan and then implement it using appropriate data structures and algorithms.
Furthermore, it has also been useful in order to improve our python programming skills. Before this lab, we didn't know what a dictionary was, or how it could help us to store and recover variables.
Finally, as we have implemented a general method to solve MDPs, this will benefit us whenever we need to solve one, as we can check if our results (computed by hand) are correct, or, even better, we can insert our model, and obtain the results immediately.