uc3m | Universidad Carlos III de Madrid

Degree in Applied Mathematics and Computing

Artificial Intelligence

2022-2023

AI Final Project

MDP for Temperature Control

Ilan Halioua Molinero | 100472908
100472908@alumnos.uc3m.es

Alejandro Fernández - Paniagua | 100473072
100473072@alumnos.uc3m.es

Group 121

TABLE OF CONTENTS

1. Executive summary

1.1. The problem

This program achieves efficient control of a thermostat that **optimizes energy consumption** while making sure that it **satisfies user's comfort needs** at half-hourly intervals. At a given instance, the temperature may not be the user's desired one, demanding the thermostat to make accurate decisions on how to adjust the heating based on the current state and desired outcome.
With this in mind, it can be seen that what this problem really boils down to, is to obtain an **optimal choice of action** for any state (temperature at a given moment) so that the thermostat manages to reach the desired room temperature in the shortest period of time.

1.2. The solution

This report provides a detailed walkthrough of the development of a general Markov Decision Process program, which allows users to specify their specific model by providing the program a text file with the attributes that compose their Mdo. Then, the program computes and returns the respective optimal policy, acquired through the resolution of the so-called recursive Bellman's Equations thanks to the implementation of the Value Iteration algorithm.

1.3. The value and importance of our solution

In the context of the given problem, this project assumes that the user's desired temperature is 22 degrees Celsius and that the possible actions are turning the heating on or off. However, our implementation goes beyond the scope of these assumptions allowing the user to freely customize the thermostat settings and in particular, to get to his/her desired temperature and costs of the actions. Some modifications to improve the basic conditions imposed in this problem could be: diminishing the intervals in which the thermostat adjusts the temperature, incorporating more concrete actions to control the thermostat such as setting the temperature to x degrees (i.e: t15, t30) or increasing/decreasing the thermostat temperature by y degrees (i.e:up1,down1,up5,…),etc.

In fact our solution solves a more general problem . The client who acquires our program could not only use it for automating temperature control, but for obtaining the optimal policy and lowest expected costs of any kind of problem modeled by a MDP i.e. in the framework made up of a set of states, actions, transitions and immediate costs.

2. Objectives

The main goal of the program consists of returning the optimal policy of an arbitrary MDP specified by the user. However, it is important to say that efficiency and generality are also in the list of goals of the program, since the solution is devised to be obtained by using the least computational power possible and attempting to serve as a general tool that can be used for any type of decision making problem, regardless of the criteria the user may have used to define the problem states, actions, costs, etc. In other words, the end goal of the program is providing users the ability to obtain the optimal policy of their MDP without imposing any requirements on the approach used to define the MDP, as long as these are just a special set of states, actions, costs and transitions.

3. Formal description of the MDP model

The main class defined in the program is the one that abstracts an Mdp object. However, this class is a broader elaboration of other important classes which are mentioned below:

3.1. States

A state is an object characterized by two attributes: a value and a goal status. The value can be anything that the user may consider. In many cases, it will be a numerical value (integer, float ,...) or a string. The goal status consists of a boolean value which reflects if the state is a goal state or not , for that particular problem.

3.2. Actions

An action consists of an object which is uniquely identified by its name. Similar to the case of the states, an action name will typically be a number or a string.

3.3. Transitions

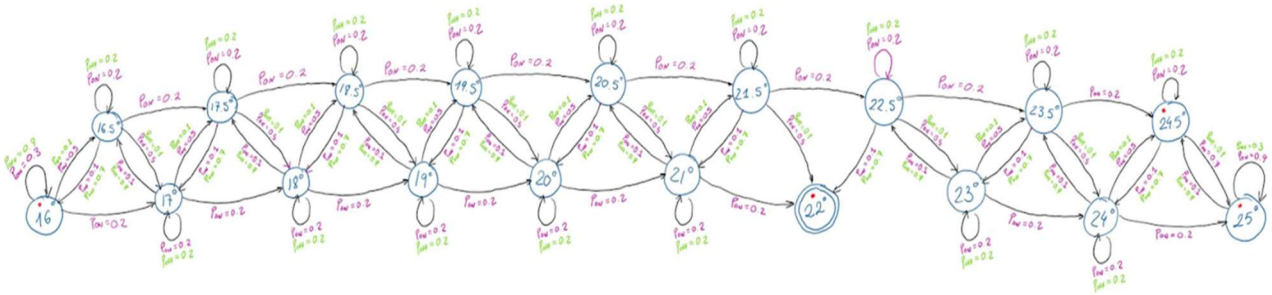
A transition is simply distinguished by three attributes (two of them from the same type): state_from . state_to and through. In practice, since the models are stochastic, they will have a probability binded to them, since there is some degree of noise when saying that the agent will transition from one state s to another state s' after taking action a.

3.4. Rewards

The concept of a reward or cost is used to establish some form of mutual understanding between the agent and the one who models the problem. These are interconnected with action objects, since they are used to guide the agent at a first glance on the action it should take (if any) in order to reach the goal state from the one where it is at a concrete time. Moreover, this decision will be made based on the action that will overall (accepting the randomness of the environment, which for these types of problems is fully observable) lead to the minimum expected cost to get to the goal state. This last thing is also known as the value of a state.

3.5. Graphical representation

Some of the first things we did when tackling the problem was to sketch a graphical representation of the NFA for the particular thermostat example. Below, one can appreciate this sketch:



4. Detailed cost model analysis

One of the key decisions made to model the problem was determining an accurate set of immediate costs for the actions: heating on / heating off. This was the main subproblem to solve in the design phase, as for the value iteration algorithm, costs play a fundamental role on the computation of the value functions for each state, leading to noticeable differences in the optimal costs and optimal policy our thermostat bases its decisions from.

This is why we had to spend some time investigating what could be the most reasonable set of costs that would make the value iteration and optimal policy algorithms achieve the following 2 principal objectives in the functioning of our thermostat, at half-hourly intervals:

1. **Optimize energy consumption**
2. **Satisfy user's comfort needs**

From this base, to start with, we decided to fully focus on item 1, and then based on the results obtained, we could adjust the costs to satisfy the second item concurrently:

1. Costs in terms of energy consumption

   Our initial query was to know what the usual consequences of turning the heat on are, in terms of energy consumption. When the heating is turned on, our sensor data shows that the room temperature increases by half a degree in 50% of cases after half an hour. In 20% of the cases, the temperature rises by one degree, while in another 20% of cases, it remains the same. Surprisingly, in 10% of cases, the temperature actually decreases by half a degree.

   However, the energy required to heat a room by a certain temperature depends on various factors, such as the size of the room, the insulation, the outdoor temperature, and the overall efficiency of the heating system. Therefore, the relationship between the change in room temperature and energy consumption lacks credibility. Anyways, from the research phase, we could get the following information: In most cases, turning on the heating system will consume some amount of energy, which will depend on the heating source and its efficiency. The amount of energy consumed can be measured in units of power (e.g., watts, kilowatts) or energy (e.g., joules, kilowatt-hours).

Moreover, the temperature variation in the room will depend on various factors, such as the initial temperature, the heating power, the insulation, and the ventilation. It is common for the temperature to rise gradually and not instantaneously.

So, to narrow the query, we decided to assume that the heating system is located at a standard room of dimensions in meters: 4.26 X 6.10. Which could be an estimate for the total energy consumption (measure in MWh) that could lead to the statistical conclusions made by the problem's statement, after checking the data collected from our sensor?

To estimate the energy consumption in MWh, we would need to know the type and efficiency of the heating system, as well as the outdoor temperature and other factors that can affect heat loss and energy consumption. Without this information, it is difficult to provide a valuable approximation.

However, we can make some additional assumptions to provide a clearer estimate. Let's assume that the heating system is a standard electric heater with an efficiency of 95%. According to engineering calculations, the energy required to heat a room of this size by 1 degree Celsius is approximately 2.6 kWh. Therefore, to raise the room temperature by 0.5 degrees Celsius, the energy consumption would be around 1.3 kWh.

Assuming the heating system is turned on for half an hour, the energy consumption would be 1.3 kWh/2 = 0.65 kWh or 0.00065 MWh.

Once we had this information we thought of thinking about the cost in terms of monetary cost. For example, using the above information, the average wholesale electricity price in Spain was nearly 70 euros per megawatt-hour in January 2023 (Spain Electricity Price - 2023 Data - 1998-2022 Historical – 2024: https://tradingeconomics.com/spain/electricity-price).

$$70 \rightarrow 1\text{MWh}$$

$$x \rightarrow 0.00065\text{MWh}$$

$$x = 0.0455 \text{ euros} = C(ON)$$

But this interpretation narrowed down the spectrum of clients that could accurately use our thermostat action costs, as the energy cost depends directly on the country. Therefore, we decided to keep for the moment: C(ON) as 0.00065 MWh and C(OFF) = 0, as no energy is being consumed when the heating is turned off. For the calculations in the program this value will be multiplied x 10^3 (in kWh) to work with more addressable costs C(ON) = 0.65, C(OFF) = 0.

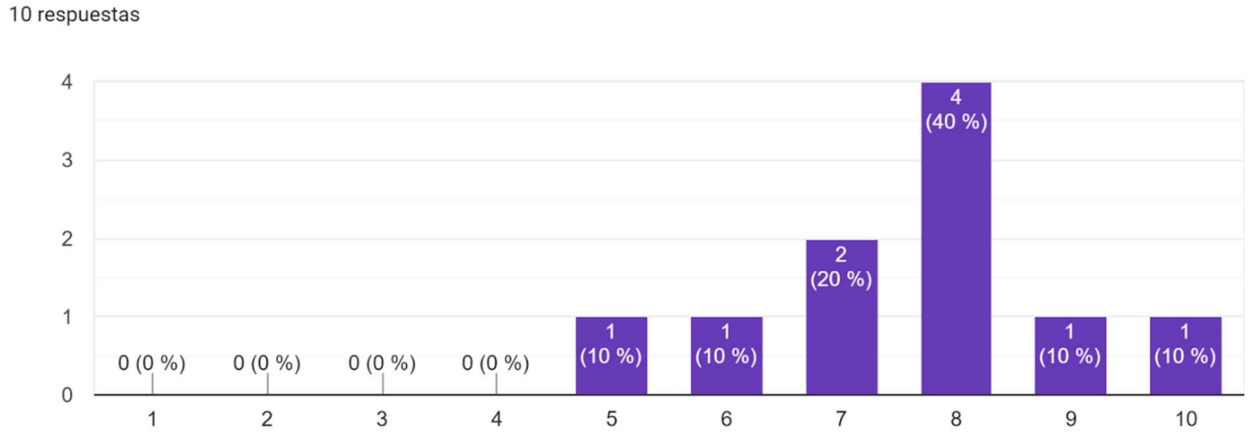
2. Costs in terms of user's comfort

Given the previous results (C(ON) = 0.65, C(OFF) = 0), a first step in determining the final immediate costs that would best fit this problem, could be to establish the percentage of privilege we would like to give to item 1 vs item 2. We decided to give a 50-50% of importance to each (i.e. same importance), but as said before, the user could base his costs on our research and decide which item (Energy Consumption or User's comfort) to give more importance to.

When we talk about user's comfort, what we really mean is to get the user's room, have the desired temperature, at the largest amount of time (even if that means having the thermostat on, for a long period of time which could result in a high level of energy consumption). The costs of the actions we have defined to represent the user's comfort are the following: C(ON) = 0.1, C(OFF) = 0.76 which are based on the standardized results of a survey made to the project managers, software developers and testers of this project

and to their relatives which range in a spectrum of age range between 14 to 60 years old and of different economic levels:

From 1 to 10 what is your level of agreement with the following statement: 'I am willing to consume some amount of energy to achieve my desired temperature, even if it means having the thermostat on for a long period of time.'



(https://forms.gle/LAgVzUD4kckaVd4M7)

Based on the survey responses, it seems that most users have a high level of agreement with the statement "I am willing to consume some amount of energy to achieve my desired temperature, even if it means having the thermostat on for a long period of time." The average level of agreement among the responses is 7.6 out of 10.

This suggests that most users place a high value on achieving their desired temperature and are willing to consume more energy to do so. Therefore, we have decided to assign a lower cost to C(ON) and a higher cost to C(OFF) to reflect the users' preferences (This will make the value iteration and optimal policy algorithms to increase the preferences towards putting the heating on to get the user's desired temperature).

Now with this information we have decided to assign a lower cost in terms of user comfort (e.g., 0.1) to reflect the users' willingness to consume more energy to achieve their desired temperature. Similarly, we have assigned a higher cost to C(OFF) in terms of user comfort (e.g., 0.76) to reflect the users' preference for maintaining their desired temperature even if it means consuming more energy.

Then, it follows that the final cost of C(ON) could be calculated as $\frac{0.1 + 0.65}{2} = 0.375$ and of C(OFF) = $\frac{0.76 + 0}{2} = 0.38$. For the calculations in the program this value will be multiplied x 10 to work with more addressable costs: C(ON) = 37.5, C(OFF) = 38.

5. Optimal policy for a certain set of costs

Given the set of costs previously defined (C(ON) = 37.5, C(OFF) = 38) and the simplicity assumption made in the problem for temperature control (states: 16º-25º in intervals of 0.5 degrees Celsius and with goal temperature: 22ºC), our program determines that the optimal policy for the following MDP is:

T (ºC)	16	16.5	17	17.5	18	18.5	19	19.5	20	20.5
Action	on	on	on	off	off	off	off	on	on	on

T (ºC)	21	21.5	22	22.5	23	23.5	24	24.5	25
Action	on	on	Not defined	off	off	off	off	off	off

Note that at 22 degrees Celsius (the goal state), there is not an Optimal Policy defined.
This is because, once in that state, no action would be an optimal one to take by definition.

If we stop a minute and analyze the actions our program has determined to be optimal for each state, we can see that according to the immediate costs established, and the transitions and their probabilities considered initially, the results follow what one would expect.
Actions ‘on’ are prioritized when temperature is below 22º to get with a higher probability to the goal state, but also rewarding the action ‘off’ as it is taking into account the high cost the thermostat could generate in terms of energy consumption. Whereas, in the range above 22º, a clear direction towards reaching the goal state is diminishing the temperature, which is more probable when taking the action off which at the same time, achieves to reduce the energy consumption which is something that was rewarded with the established costs.

Before starting working on this project, without even having thought about the immediate costs to use in the problem, we had a basic idea of what could the optimal policy for each state be.
Our instincts said that from 16º up to 21.5º, putting the heat on was going to be the optimal action, whereas for 22.5º up to the 25º limit, the optimal one was turning off the thermostat, but this assumption was given by personal bias in which without even realizing, our minds were highly rewarding the user’s comfort (i.e. reaching the goal temperature) and completely forgetting about the cost taking the action ‘on’ (i.e turning on the thermostat) would generate. This can be shown, as if we establish for example the cost of taking the action ‘on’ to be 10 while of the action ‘off’ to be 500, our optimal policy matches completely with the one predicted (User could try it). And this shows how modeling an accurate set of costs in a MDP, plays a key role in later obtaining the most accurate and precise optimal policy based on our needs.

6. Project phases

6.1. Design phase

Throughout the development of the program, we decided it would be convenient to abstract the main components of an MDP as independent classes that will later be stored in dictionaries and sets to facilitate the design of the algorithm that would give the optimal policy of the MDP object. This stage of the practice was pretty straight forward, since the definitions for all of these objects were clear from the theoretical lectures we had attended during the course. For the specific case of the declaration of the MDP class, we felt that it could be practical to define the MDP state`s set to be a Python set of state objects, since order is not important but belonging is the fundamental property for this case. The same applied to the definition of the MDP's actions attribute. Perhaps, the only special treatments were those given to define the MDP's Immediate costs and Transitions attributes which were defined as dictionaries where objects (actions and transition ones) served as keys of numerical values referring to costs and probabilities, respectively. This would later on prove to facilitate and smooth out the technical implementation of the Value Iteration and Optimal Policy's Algorithms

Also, we had to figure out a simple way for the user to give the details of his/her specific MDP through a text file. For ensuring the correctness of the input, we imposed specific format rules for the input text file that contains the information needed about any user's model. The text file must contain exactly 4 lines whose content order is the following: first line is dedicated to the exposition of the mdp's states, second line to the actions, third line to the immediate costs of each action and fourth line for the transitions and their probabilities. One can look at the following sample text file to make sure the format mentioned above has been understood:

```
states: happy*, sad , neutral, mad, sick, tired
actions: exercise, rest, eat
costs: c(exercise) = 2.0, c(rest) = 1.0 , c(eat) = 1.0
transitions: T(happy,happy,exercise) = 0.5, T(happy,sad,exercise) = 0.1, T(happy,mad,exercise) = 0.1, T(happy,neutral,exercise) = 0.2,
```

Now that the design of the input file was set, we just had to find a way to parse it, which was a matter of writing the right Python code that would do that job for us.

Last but not least, we decided that instead of returning the optimal policy for the user's input mdp via the python console, it could also come in handy to have all of this information output inside a text file named "OptimalPolicy.txt".

Having all of these details and steps defined, now it was the turn of the implementation phase.

6.2. Implementation phase

6.2.1 *Classes*

As it was said in the design phase section of this report, the first thing to do was defining the classes of the "independent" set of objects that would later on combine themselves with different data structures to give a body to the main Mdp class. Below, one can appreciate the part of our python source file that reveals the implementation for all of these mentioned above:

```
class State:
    def __init__(self, val,g = None):
        self.state_val = val
        self.goal = g

class Action:
    def __init__(self, name):
        self.action_name = name

class Transition:
    def __init__(self, p: State, q: State, a: Action):
        self.state_from = p
        self.state_to = q
        self.through = a

class MDP:
    def __init__(self, S = None, T = None, A = None, IC = None):
        self.States = S
        self.Transitions = T
        self.Actions = A
        self.ImmediateCosts = IC
```

6.2.2 Value Iteration

As for the implementation of the Value Iteration algorithm, it is advisable to first mention the formula that describes the so-called value of a state in an Mdp. When referring to the value of a state s, we refer to the lowest expected cost needed to get to any goal state of the Mdp in question, from s. Now, we talk about expectations since the models we are working with will be most of the time stochastic. Then, this details are enough to describe the value of a state s mathematically through the expression shown below:

$$V(s) = Min(ImmediateCost(a) + \sum_{s' \in States(Mdp)} Probability(s, s', a) \times V(s')), \forall a \in Actions(s)$$

Now, notice that the equation above is recursive and the system involved to solve V for every state in the Mdp is nonlinear (because min is not linear) having the same number of equations as unknowns (the cardinality of the set of states of the Mdp ie: |States(Mdp)|)

To solve it, we must use the value iteration method, which is explained below:

$$If\ s_n := Min(c(a) + \sum_{s' \in States(Mdp)} P(s, s', a) \times s'_{n-1})\ if\ n \geq 1\ (0\ otherwise)$$
$$\forall s \in States(Mdp)$$

$$We\ will\ claim\ that\ if\ the\ sequence\ converges,\ \lim_{n \to \infty} s_n = V(s)$$

We will claim that if the sequence converges, nsn= V(s)

Now, assuming that the convergence is something certain, which to our fortune, turns out to be 100 percent of the times, we have implemented the Value Iteration algorithm in the following way, as a method of the Mdp class:

```
def ValueIteration(self):
    done = False
    V = {key: 0.0 for key in self.States}
    while not done:
        OldV = V.copy()
        for s in V:
            if not s.goal:
                min = float('inf')
                for a in self.Actions:
                    v = self.ImmediateCosts[a]
                    for s_prime in self.States:
                        t = Transition(s, s_prime, a)
                        if t in self.Transitions:
                            v += self.Transitions[t] * OldV[s_prime]
                    if v < min:
                        min = v
                V[s] = min
        if OldV == V:
            done = True
    return V
```

6.2.3 Optimal Policy

During the design process we spotted a really nice fact, which is that the computation of the optimal policy relies indirectly on that of the Value Iteration. Therefore, we could see that the way to implement the optimal policy would be done analogously to the one for the Value Iteration.

To see that this fact, it is necessary to mention the definition of the Optimal Policy of a state s, denoted by $\boldsymbol{\pi}$…

$$\pi(s) = \text{argmin a}\,(c(a) + \sum_{s' \in S(Mdp)} P(s,s',a) \times V(s')) \; \forall a \in A(s)$$

Then, we went on to implement the OptimalPolicy method inside the Mdp class in the following manner, based on the latter information:

```
def OptimalPolicy(self):
    V = self.ValueIteration()
    OP = {s: None for s in self.States}
    for s in self.States:
        if not s.goal:
            min = float('inf')
            opt_action = None
            for a in self.Actions:
                c = self.ImmediateCosts[a]
                for s_prime in self.States:
                    t = Transition(s, s_prime, a)
                    if t in self.Transitions:
                        c += self.Transitions[t] * V[s_prime]
                if c < min:
                    opt_action = a
                    min = c
            OP[s] = opt_action
    return OP
```

Notes:

1- The function c is the same as the ImmediateCost function

2 - The comments of the code can be seen in the source file of the project itself (these snippets are part of the mdp.py file. They have been removed from the pictures here to not make the presentation look messy or overloaded)

6.2.4 The parser

Our program asks the user to introduce the file which contains the mdp definition of the problem he would like to get an optimal policy from. When the file is recognized as an input, the function setUp_mdpSolver(), passes as argument the input file and continues to call the parser function in the parser.py file. Once the parser returns the desired information, the set up function proceeds to initialize the MDP object and returns it. Once the main program receives the fully created mdp object, this one is used to call the optimal policy method defined on it.

As it can be seen, the key function in the correct functioning of our program is the parser. Without a well formed one, none of the other important functions would even work properly. So, how does our parser work?

As briefly mentioned before, our parser expects a file with at least 4 lines which start with 'states:'/ 'actions:' / 'costs:' and 'transitions:'. Thanks to this format and to the regex python module (re), the parser function separates in 4 blocks of code the parse of each mdp attribute. This is done with a if[states parsing]-elif[actions parsing]-elif[costs parsing]-elif[transitions and their probabilities parsing] with conditions of the following type: line.lower().startswith('str:'). It should be noted that these first matches are case insensitive to prevent errors if the user does not write the strings with the expected case. The list thing to mention about this structure is that it allows the user to alter the order of the lines. So for example, our parse would work well with a file like this:

```
states: 16, 16.5, 17, 17.5, 18, 18.5, 19, 19.5, 20, 20.5, 21, 21.5, 22*, 22.5, 23, 23.5, 24, 24.5, 25
actions: on, off
costs: c(on) = 37.5, c(off) = 38
transitions: T(16,16,on) = 0.3, T(16,16,off) = 0.9, T(16,16.5,on) = 0.5, T(16,16.5,off) = 0.1, T(16.5,16,on) = 0.1,
```

but also like this:

```
transitions: T(16,16,on) = 0.3, T(16,16,off) = 0.9, T(16,16.5,on) = 0.5, T(16,16.5,off) = 0.1, T(16.5,16,on) = 0.1,
costs: c(on) = 37.5, c(off) = 38
states: 16, 16.5, 17, 17.5, 18, 18.5, 19, 19.5, 20, 20.5, 21, 21.5, 22*, 22.5, 23, 23.5, 24, 24.5, 25
actions: on, off
```

And would not be affected by extra lines that don't start with the 4 expected strings.

Now let's focus on the parsing of each of these lines:

- **States:**

  The variable 'matches' stores all the introduced states with the format: r'([\w.]+(?:)?*?)'

  If the reader of this report is not familiar with regular expressions in python, this regular expression would match one or more-word characters or periods (user may introduce states with 'decimal' format as in our thermostat problem), optionally followed by an asterisk (which will serve as a flag for the goal state).

- **Actions:**

The variable ‘action_names’ stores all the introduced states with the format: r'\b[\w.]+\b'

This regular expression would match whole words consisting of word characters or periods.

- **Costs:**

The variable ‘costs’ stores all the introduced states with the format:

r'c\(\s*([\w.]+)\)\s*=\s*(\d+(?:\.\d+)?)'

This regular expression would match a cost expression in the form of 'c(name) = value', where 'name' represents the cost name (composed of word characters or periods) and 'value' represents the cost value (a numeric value).

- **Transitions:**

The variable ‘matches’ stores all the introduced states with the format:

r'T\(\s*([\w.]+(?:)?*?),\s*([\w.]+(?:)?*?),\s*([\w.]+)\)\s*=\s(\d+(?:\.\d+)?)' '

This regular expression would match a transitions and their respective probabilities when found in the form of 'T(name1, name2, name3) = value', where 'name1', 'name2', and 'name3' represent the parameters (composed of word characters or periods) and 'value' represents the numeric value (probability) assigned to the transition.

6.3. Testing phase

Far from a linear or sequential ordered procedure, the design and successful completion of the program was acquired through the progress and alternation of both implementation and testing phases , since both serve as a feedback mechanism to the other: there were sometimes when the implementation seemed to be right but the test results were not the ones expected, allowing us to go back to our craft and fix the programming errors. On the other hand, sometimes we realize that new types of tests should be considered and created in order to fully evaluate our program implementations. One of the most insightful examples we could give about the major importance of the testing stage of the project was the one where the program wouldmt return the optimal policy nor the Values of the states due to mutual error regarding the search of special types of transitions in the user's Mdp . This error came as a surprise to us at first, but after careful revision, the issue was clear. To illustrate and motivate the problem, we present below the place in the code where the error was located and the reason behind it:

Before the error	After the error

`for s_prime in self.States:`
`  t = Transition(s, s_prime, a)`
`  v += self.Transitions[t] * OldV[s_prime]`
 	
#Error description:
A user will probably save time and not type down those transitions with zero probability of occurring, meaning that after filling the Mdp object and calling the Value Iteration method, the program will iterate through all possible transitions for s_prime and always search for a transition, ignoring whether or not the current transition has zero probability of occurring. Therefore, this will result in an error by the time it searches on the transitions dictionary of the Mdp object for a transition with zero probability.
	`for s_prime in self.States:`
	`  t = Transition(s, s_prime, a)`
	`  if t in self.Transitions:`
	`  v += self.Transitions[t] * OldV[s_prime]`
	
	#Modifications to fix it:
	By realizing the source of the error, by thinking like a user. The error can be handled by simply adding an if statement that checks if the transition to look for has a nonzero probability of occurring. In addition, one could also say that regardless of the error , when a transition has zero probability of happening the addition would become useless, so it could have been ignored from the beginning, maybe being negligibly more "efficient".

Another error we encountered on the implementation of our Mdp object was also related to the Value Iteration algorithm. The source of the error could have never been spotted if it was not for the test we did with some familiar Mdp problems we had already solved in class. It consisted on missing the update of the minimum cost computed so far during the iterative process, which incorrectly would stay as positive infinity until the last iteration in which the last action would become the one with least cost. Check the before and after code section below:

Before the error	After the error

`for s in self.States:`
`  if not s.goal:`
`  min = float('inf')`
`  opt_action = None`
`  for a in self.Actions:`
`  c = self.ImmediateCosts[a]`
`  for s_prime in self.States:`
`  t = Transition(s, s_prime, a)`
`  if t in self.Transitions:`
`  c += self.Transitions[t] *V[s_prime]`
`  if c < min:`
`  opt_action = a`
`  OP[s] = opt_action`
 	
#Error description:
The minimum value found at a given moment is never updated, so min value remains the initializer container given since the first declaration of min, which is +infinity. Therefore, the optimal action will be for every s the one that was last in the set of actions for that particular execution
	`for s in self.States:`
	`  if not s.goal:`
	`  min = float('inf')`
	`  opt_action = None`
	`  for a in self.Actions:`
	`  c = self.ImmediateCosts[a]`
	`  for s_prime in self.States:`
	`  t = Transition(s, s_prime, a)`
	
	`  if t in self.Transitions:`
	`  c += self.Transitions[t] *V[s_prime]`
	`  if c < min:`
	`  opt_action = a`
	`  min = c`
	`  OP[s] = opt_action`
	
	#Modifications to fix it:
	Update the value of min with the cost

Finally, we added a rather unnecessary feature to our program which we came up with after experiencing frustration from our own selves. One could assume during the development of this practice that the user will have correctly given the information about its Mdp, including to not forget about adding the * char after the goal states that are instantiated in the text file : either when listing the set of states or when they appear in the list of transitions. The second one is more error prone and if the parser did not check out that the given info contains goal states among the data, a naive implementation of the project would go on to calculate the "Optimal Policy" of a supposed "Mdp" and return things on the output text file and screen that would be nonsense. In fact, each execution will randomly derive different "optimal policies". Now, with the enhancement on our parser, this is not the case, since the parses will check on goal states and if none, it will return an empty Mdp (all member equal to None), so that later on the main program will see that since the Mdp is empty, it doesn't make sense to call the Optimal Policy method with it, notifying the user. This fact is easily illustrated in the main.py source file:

Parser details involved in the solution for this exception	Main details involved in the solution of this exception

`if goal_S != 0 and goal_T != 0:`
`  return states, actions, costs, transitions`
`else:`
`  return None, None, None, None`
	`if user_mdp.States and user_mdp.Actions and user_mdp.Transitions and user_mdp.ImmediateCosts:`
	`  .`
	`  . (code to call and print the Optimal Policy)`
	`  .`
	
	`else:`
	`  print("\nIntroduce your mdp definition with the expected file format, please!")`

7. Budget

7.1. A financial estimate of how much the study and implementation would have cost if someone else had contracted it out.

Project managers	2
Software developers	2
Testers	2

Table. Team that developed this project

Role	Workdays	Salary per workday	Total cost
Project manager	3	600€	1800€
Software developers	7	350€	2450€
Testers	3	250€	750€

Table. Costs study of our program

Total budget: 5000€

8. Conclusions

8.1. Technical comments related to the development of the project.

The development of the practice was asked to be done through the Python programming language, which is in fact almost new to our team, since we have started to informally learn it during the beginning of this semester course. However, we have a sufficiently solid foundation on lower level languages such as C or C++, which has prevented us from getting stuck at the times when we had to use python data structures such as sets or dictionaries, since we were already familiar with their cousins in these other programming languages. We would also like to add that we found really interesting to see the uses of regular expressions in the python language to read data from files, since it made us feel like the subject of Automata Theory and Formal Languages , covering topics on Computation Theory had made a positive contribution to our set of problem solving and development tools.

8.2. Personal comments: difficulties, challenges, benefits, etc.

Although on the programming and modeling side of the project, we have learned a modest amount, the most remarkable area of benefit obtained from this project development has been the transition from the theoretical concepts of Mdp's and decision-making agents taught in class to their most practical version within the industry. Thanks to this practical approach, we have been able to fully grasp the ideas behind this AI field, making us think even further about how many problems that sounded too out of reach for us before, seem now reasonably solvable through these techniques. As an example on this apprenticeship-style questions we may make to ourselves as students on the subject , we have been thinking about a more sophisticated model in which the agent can recalculate the noise and certainty of the transitions for the different actions available, by gathering live data about the environment every x units of time and right after, makes a decision based on the newly computed optimal policy at that instant and particular state it is on. This sounds like a realistic explanation of how a cleaning robot may function, by retrieving data from the environment through the use of sensors that help detect walls and objects in their way (useful for partially observable environments) or position tracking when the map is fully laid down (fully observable environments).