Universidad Carlos III

Artificial Intelligence 2022-23

Course 2022-23

MARKOV DECISION PROCESS FOR TEMPERATURE CONTROL

88

József Iván Gafo (100456709)

Marcos González Vallejo (100472206)

Index

Executive summary

The project consists of performing a number of iterations of the Bellman equations to obtain the optimal policy. First we are given a statement, where we can find the information about a Markov decision process such as states, actions and probabilities. For the costs, we had to make an analysis and choose the ones that made more sense. The objective is to obtain the optimal policy for a desired temperature that, in the statement, is 22º, but can be a general objective. Moreover, the temperature can vary between 16º and 25º and a different probability of executing each action is given for each different action. From a given state, the temperature can vary, if in action on: +0,5, -0,5,+1 or stay and in off: +0,5, -0,5 or stay. Additionally, the actions can also be generic, from 2, as the statement says to n.

Objectives

The objective for this project is to create a program in python that is able to calculate the optimal policy for any given cost for the Temperature Control and to do an analysis of the different costs. Also, the program must be able to calculate the policy for any number of actions.

Formal description of the MDP model

The Markov decision process that we were given had the following information:

-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:

In the statement, the following actions are specified, however, our program must be as general as possible so more actions may be included.
{on , off}

-Probabilities:

on:

Heat on	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
16	0,3	0,5	0,2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16,5	0,1	0,2	0,5	0,2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0,1	0,2	0,5	0,2	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17,5	0	0	0,1	0,2	0,5	0,2	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0,1	0,2	0,5	0,2	0	0	0	0	0	0	0	0	0	0	0	0
18,5	0	0	0	0	0,1	0,2	0,5	0,2	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0,1	0,2	0,5	0,2	0	0	0	0	0	0	0	0	0	0
19,5	0	0	0	0	0	0	0,1	0,2	0,5	0,2	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0,1	0,2	0,5	0,2	0	0	0	0	0	0	0	0
20,5	0	0	0	0	0	0	0	0	0,1	0,2	0,5	0,2	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0,1	0,2	0,5	0,2	0	0	0	0	0	0
21,5	0	0	0	0	0	0	0	0	0	0	0,1	0,2	0,5	0,2	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0,1	0,2	0,5	0,2	0	0	0	0
22,5	0	0	0	0	0	0	0	0	0	0	0	0	0,1	0,2	0,5	0,2	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0,1	0,2	0,5	0,2	0	0
23,5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,1	0,2	0,5	0,2	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,1	0,2	0,5	0,2
24,5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,1	0,2	0,7
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,1	0,9

off:

Heat off	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
16	0,9	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
16,5	0,7	0,2	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17	0	0,7	0,2	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
17,5	0	0	0,7	0,2	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
18	0	0	0	0,7	0,2	0,1	0	0	0	0	0	0	0	0	0	0	0	0	0
18,5	0	0	0	0	0,7	0,2	0,1	0	0	0	0	0	0	0	0	0	0	0	0
19	0	0	0	0	0	0,7	0,2	0,1	0	0	0	0	0	0	0	0	0	0	0
19,5	0	0	0	0	0	0	0,7	0,2	0,1	0	0	0	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0	0,7	0,2	0,1	0	0	0	0	0	0	0	0	0
20,5	0	0	0	0	0	0	0	0	0,7	0,2	0,1	0	0	0	0	0	0	0	0
21	0	0	0	0	0	0	0	0	0	0,7	0,2	0,1	0	0	0	0	0	0	0
21,5	0	0	0	0	0	0	0	0	0	0	0,7	0,2	0,1	0	0	0	0	0	0
22	0	0	0	0	0	0	0	0	0	0	0	0,7	0,2	0,1	0	0	0	0	0
22,5	0	0	0	0	0	0	0	0	0	0	0	0	0,7	0,2	0,1	0	0	0	0
23	0	0	0	0	0	0	0	0	0	0	0	0	0	0,7	0,2	0,1	0	0	0
23,5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,7	0,2	0,1	0	0
24	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,7	0,2	0,1	0
24,5	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,7	0,2	0,1
25	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0,7	0,3

-Costs:

Refers to the cost associated with turning the heater on and off, and their values are determined by the user.

Detailed cost model analysis

For the same costs of on and off (let's say 1), optimal policy from 16 to 21.5, both included are on, and the rest are off. The more we increase the cost of on while maintaining the cost of off constant (let's say at 1), more and more values from 16 until 21.5, will transform into off from on (starting at 16), as the cost of on will be too high to choose it. Even if for action off, the cost of increasing the temperature is 0.1, is more worth than using action on because of its high cost.

Also, if we increase the cost of off while maintaining the cost of on constant, more and more values from 25 to 22 will become on (starting at 25) because the cost of off is so high that the minimum is actually the expected cost using on. That is to say, it is easier to lower the temperature by using action on, than using action off because of the huge cost of action off. Obviously, when the cost of one of them is 0 and the other is different from 0, the one with 0 cost will be the optimal policy for each state.
Here is a table with some examples of the output of the optimal policy.

costs on:	costs off:	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
1	1	on	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
2	1	on	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
1	2	on	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
5	1	on	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
10	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
15	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
20	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
25	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
30	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
35	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
40	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
45	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
50	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
55	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
60	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
70	1	off	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
80	1	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
90	1	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off

100	1	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
120	1	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
140	1	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
160	1	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
200	1	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
250	1	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
300	1	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
400	1	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
550	1	off	off	off	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
900	1	off	off	off	off	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
100	0,5	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
100	0,25	off	off	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
100	0,1	off	off	off	off	off	on	on	on	on	on	on	on	off	off	off	off	off	off	off
100	0	off	off	off	off	off	off	off	off	off	off	off	off	off	off	off	off	off	off	off
0,1	0,1	on	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off
100	100	on	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off

1000	1	off	off	off	off	off	on	on	on	on	on	on	on	off	off	off	off	off	off	off
1	1000	on	on	on	on	on	on	on	on	on	on	on	on	off	off	off	off	on	on	on
1000	1.5	off	off	off	on	on	on	on	on	on	on	on	on	off	off	off	off	off	off	off

As it can be seen, in on: 1 and off: 1, optimal policy is the default one, on from 16 to 21.5 and off from 22 to 25. However, if we increase on and leave off at 1, at on: 10 and off: 1 there is one off in 16 as it is cheaper to select action off to increase temperature from 16 because of the cost of on. Then, at on: 80 and off: 1 there is one more off, now in 16.5 because of the same reason. Then, much later on on: 550 and off: 1, there are 3 off, in 16, 16.5 and 17, because on is more and more costly each time, so to accomplish the minimum cost, the result of using a lower probability to increase temperature is cheaper than of using the action on. Then, at on: 900 and off: 1, we now have 4 offs because of the same reason as before and finally, although this is much longer, at on: 1000 and off: 1, we have 5 offs. In the inverse way, a similar thing occurs, for instance, at on: 1 and off: 1000, there are three on in 25, 24.5 and 24. The more we increase off while keeping on constant, the more 'ons' at the left of the screen will appear.

Optimal policy

The optimal policy of each state varies depending on the costs. In the previous section, a table with the optimal policy of each state for some costs is given. However, if we have to select a certain set of costs, as the statement asks, we would choose:
cost off: 1, cost on:2. It makes sense as turning heat on is more expensive than keeping it off. In this case, the optimal policy is as follows: from 16 to 21.5 º both boundaries included, the optimal policy is on as the objective is 22º. However, from 22 until 25, the optimal policy is off as we desire to lower it.

```
The value for Cost ON is:1
The value for Cost OFF is:2
Cost On= 1.0 and Cost off= 2.0
The optimal policy for each node is:  ['node 16.0: on', 'node 16.5: on', 'node 17.0: on', 'node 17.5: on', 'node 18.0: on', 'node 18.5: on', 'node 19.0: on', 'node 19.5: on', 'n
ode 20.0: on', 'node 20.5: on', 'node 21.0: on', 'node 21.5: on', 'node 22.0: off', 'node 22.5: off', 'node 23.0: off', 'node 23.5: off', 'node 24.0: off', 'node 24.5: off', 'no
de 25.0: off']
```

Project phases

Design:

For this project we decided to code a program that can calculate the optimal policy for n actions and k nodes, so in case the clients want to add an action can do it without modifying all the code. Our code is divided into 4 files (main, optimal_policy_exception, ia_prob.xlxs and optimal_policy and)

1) Main.py:

In this file we are in charge of defying a list called **cells**, that is in charge of storing the cells of probabilities of the actions. If the client wants to add another action, all it only needs to do is to append on the list cells a string containing the cells of the excel file where we can find the probabilities for that new action.

2) Optimal_policy_exception.py:

This file is in charge of creating a class that we will be using to raise our personalized errors on the class OptimalPolicy.

3) ia_prob.xlsx:

This excel file is in charge of containing all the probabilities for the actions on the sheet ia_prob. If we want to add an action we simply had to add the values of probabilities on this sheet and then on main.py append it on the cells list. If we want to add a new node because we want that our temperature goes from 16 to 25’5 we simply have to add a new column for every action.

4) optimal_policy:

This module is in charge of containing the class OptimalPolicy that can calculate the optimal policy for k nodes and n actions. This module can be divided into 6 parts (Imports,

global_variables, (inside the class OptimalPolicy)__init__, calculate_optimal_policy, main functions and __init__ functions.

a) Imports:

In this part we import libraries that we will be using in the class **OptimalPolicy**.

- The libraries that we import are xlwings (a library to open excel files),
- Decimal (a library to use float numbers with high precision) we used this library to do operations with floats because we wanted to avoid errors like 0.2*0.1=0.020000000000000004.
- OptimalPolicyException is a custom class that we created to raise our personalized errors

b) Global Variables:

This module has 2 global variables that is the **EXCEL_PATH** containing the name of the excel file, but if we moved the excel path to another folder we would need to put inside the full path to arrive at the desired excel. The second global variable is **EXCEL_SHEETS** that contains a string that tells us in which excel_sheet do we find the probabilities of actions.

c) OptimalPolicy, __init__:

In this part we execute the init of OptimalPolicy. The class OptimalPolicy receives 2 parameters that are the **excel_cells_list** and **desired_temperature**. The first parameter contains a list with all the ranges of cells for every action and the last parameter is to indicate the temperature node that the user desires. In our problem is the node 12 that is 22 ºC (22=16+**12***0.5).

After, we define program parameters that the programmer can change. The first one is self._print, this parameter if activated (True) when executed the program will show all the operations performed by the program, useful when debugging to observe if our program does the calculations well. The second is self._max_limit_it , this parameter is a limit of iterations that we put to stop the program in case it iterates to infinite or a very big number. Lastly, we have self._ precision, this parameter is to say how many digits-1 do we take into account when iterating. e.g “1.23456”[:4]=”1.23”. The reason we decided to include this parameter is because there was a point when iterating there was a very small difference between the previous value and the actual value and we knew that it wouldn’t change the values much, so we decided to take the first 99 digits and ignore the rest.

After we initialized the list containing the cost of every action and the list of probabilities of every action. Next we use a method to populate the cost_list and then populate the probabilities list where it contains on every position, another list with rows of probabilities for every node for the action i . Lastly, we define self._length (containing the total number of nodes) and if self._print is activated. We print the probabilities table for every action so we can observe that our program successfully extracted the probabilities from the excel file.

d) calculate_optimal_policy:

This method is the main function of our class that is in charge of calculating the optimal policy for our problem. We first use a main function to calculate the values for every node, then we perform the optimal policy function to calculate the optimal policy for every node. And finally we print the result on the console and we return the list of optimal policies for every node.

e) OptimalPolicy, main functions:

In this part we can find the main functions that are used to calculate the optimal policy. The main functions are self._ calculate values, self._belman_eq, self._ stochastic domain and self._optimal_policy.

self._calculate_values:

This part is in charge of calculating the values of each node using the bellman operation.

We first populate a list for **prev_value** and **value** with “-1” and “0” respectively. That will be used to put the values of the previous and actual iteration for every node. We used a string because we can say to the program how many digits or characters we want to read. We have a variable called iteration that is primarily used for the prints of our program and restrictions of the max number of iteration.

Then we start calculating the values. We do a loop using the while statement where it will iterate until the **value** list is equal to the **prev_value** list and also that **iteration** is smaller or equal than the **max limit iteration**. Inside the while if self._print activated we print the number of iteration and the actual and previous value for every node (the reason we included this print is for debugging because it allows us the reason as to why it has decided to enter the while again.
After, we use a for loop to save the actual values into the before values list, because now we will be calculating the actual values on the new iteration. We use a loop because we want a deep copy where we copy the values and not the address of the elements of the list values.

Then, we create a for loop for **self._length** nodes, where we check before that the node **tn** doesn't correspond to the **DESIRED_TEMPT** because this value will always be constant at **value** 0. If that is not the case we use the **bellman equation function** to calculate the **new actual value** for the node **tn** and assign it to the **value** list, choosing **self._precision** characters. After executing this function we increase the **iteration** by 1 and then we iterate again in the while loop.

self._belman_eq:

This method is to calculate the bellman eq of node tn on the iteration n. The parameters of this method are:
tn is the node that we want to calculate the Bellman equation,
prev_values are the previous values of each node and
the number of iterations that is only used, if we activate self._print.
What this function does is to calculate the stochastic domain for every action and then calculate the minimum between every action on node tn. If self._print is activated, we print the min operations where we can observe what the min operations are doing, and if it is doing it well. Finally we return a string that returns the minimum value. We decided to convert it into a string because when using a string we can decide how many digits we take for example "1.23456"[:4]="1.23" that later will be used on the main code.

self._stochastic_domain:

This method is in charge of calculating the stochastic domain for a specific node. The parameters of this function are the probability table of a single action , tn that is the node that we want to calculate the stochastic domain, the list of previous values for all the nodes and the cost of the single action. In this function we use the class "Decimal" because we want precision when operating float numbers to avoid imprecisions (avoid errors like 0.1*0.2). This function is basically in charge of summing the cost of action with the sumatory of the probability of Tn+1 knowing that the previous node was Tn multiplied by the previous node. And we return the result of this operation on a float number.

self._optimal_ploicy:

This function does the same thing as the bellman equation but instead of returning the minimum between the actions it returns an integer where it gives the best action for that node.

We first define **best_action_value** and **best_action** (that are variables to store the best action value and number of actions). Then, we do a loop where we calculate the stochastic

domain for action and then use the conditional if we found a better action (with a lower value). After the loop we check that we found the best action else we raise an error. Finally, if print is activated we print the best action for node tn and then return the best action.

f) OptimalPolicy, __init__ functions:

In this part we define methods that are used on the __init__ of the class OptimalPolicy. The methods are self._extract_excel_data,self. _print_table and self._create_cost_list.

self._extract_excel_data:

This method is in charge of opening and extracting the probabilities of every action that has cells and save it on self._prob_table and close the excel file. If an error occurs during this process we raise our personalized error.

self._print_table:

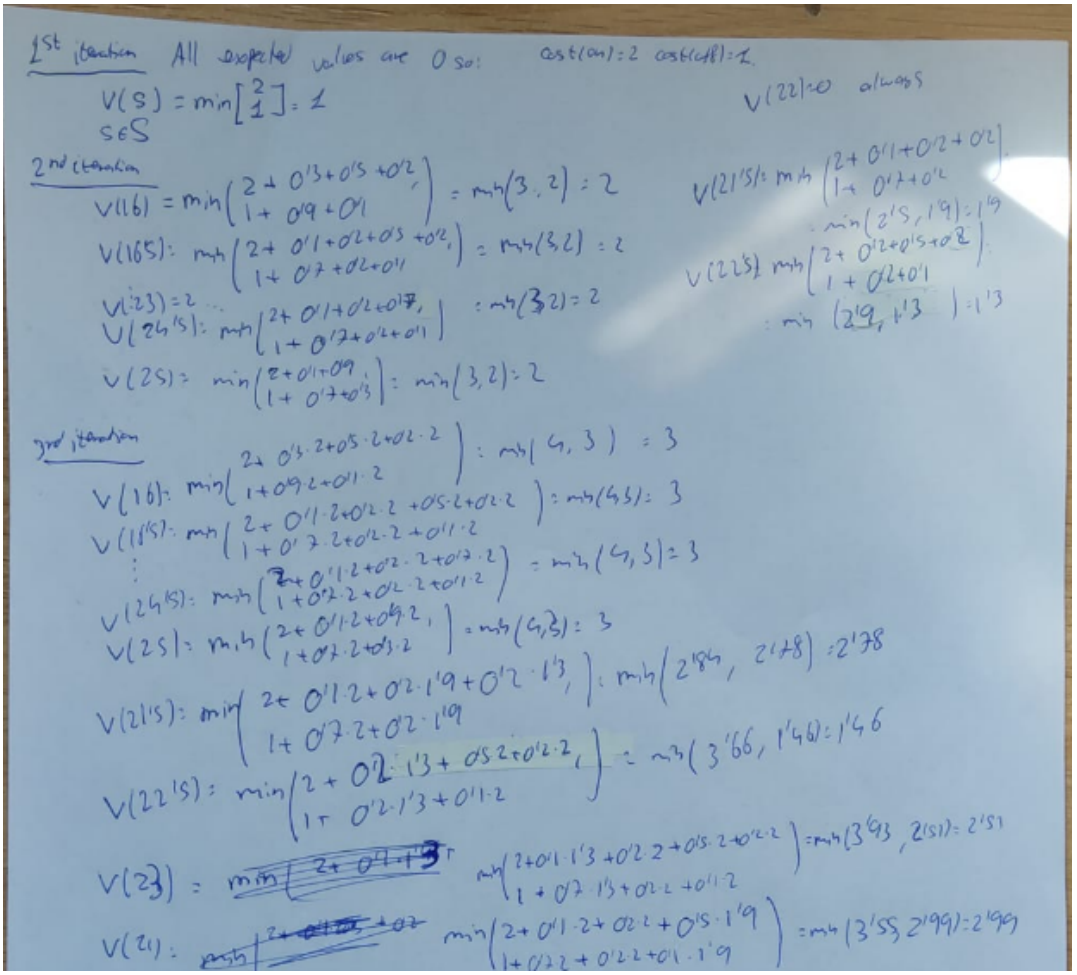
This method is only activated if self._print is on and what it does is to print the probability table for every action. This method is useful for debugging because we can check that our program has successfully imported the probabilities from every action from the excel file.

self._create_cost_list:

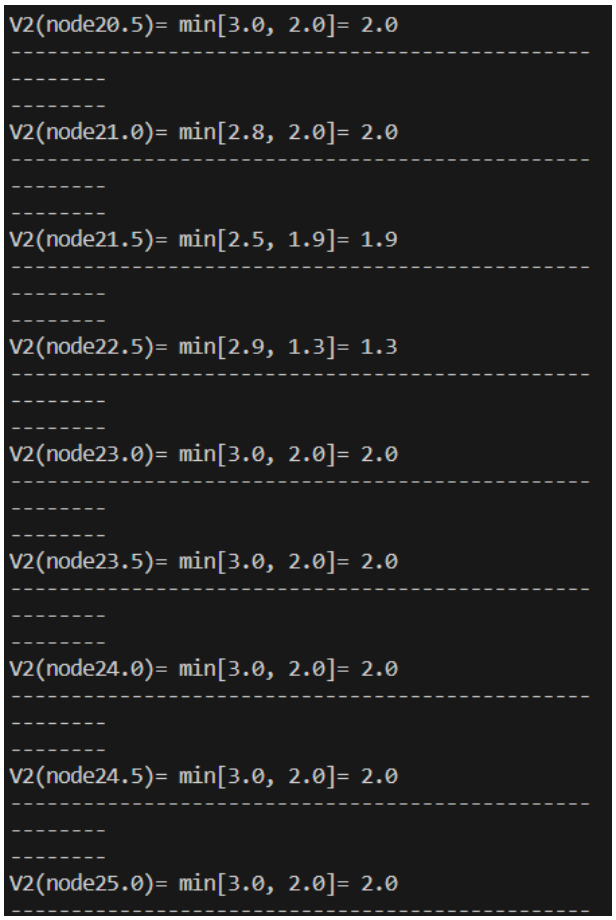
This method is used to create the costs of every action. What it does is a loop for every action and ask the user the cost for it, and we append on self._cost_list. If an error occurs when using the input we raise our personalized error. We decided to use the input function because we can ask the user directly the desired cost of every action and we wouldn't have to change the cost manually on the code. This method was very useful when performing the cost analysis.

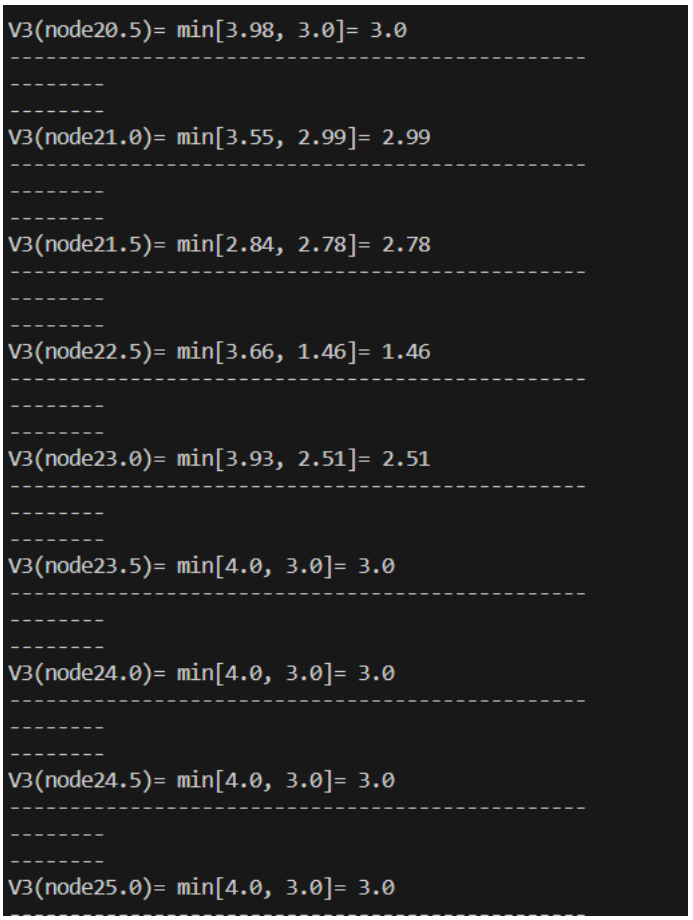
Testing:

For testing, we calculated by hand, 3 iterations of the states that actually change, here is the photo:



As it can be seen in the photo, for the first iteration, all expected values are 1. For the second iteration, all expected values are 2 but the ones closest to the objective, 21.5 and 22.5 which are 1.9 and 1.3 respectively. Finally, in iteration 3, the only values that change are 21, 21.5, 22.5 and 23, whose values are: (2.99, 2.78, 1.46 and 2.51) respectively. Here are 2 screenshots of the output of our program. Obviously, iteration 1 is only 1 for all values but 22.





Budget

Financially, a project like this one, won't be cheap as for a good performance, a good environment must be prepared for the people that will write the code. Assuming 2 people will perform the code, a salary of 2000 € will be given to each of them. Moreover, the offices will also be a cost as well. Approximately, they will cost about 400€ a month. Moreover, some expenditures such as water, electricity etc, will be also added, approximately 200 € per month. Finally, a computer for each programmer will cost about 1500 each. In total, an amount of 7600 € will be spent on a project like this one.

Conclusions

(technical comments related to the development of the project and personal comments: difficulties, challenges, benefits, etc.).

To conclude, we think that the project was very helpful to make us understand to a higher level the concept of Markov Decision Process. Thanks to this project, we discovered several characteristics of the MDP that we did not realize just by doing problems by hand, such as the importance of having an objective state so that the cost in that state is 0, and the iterations stop at one point. Before realizing that, the output of our program was infinite, it never stopped because it did not converge because we did not have a 0 cost state. Once, we assigned 0 to the cost of 22 for all operations, the program finally converged. Moreover, the need of doing the program for an arbitrary number of iterations was a challenge because our idea was just to do the program for actions on and off, but in a class we were told that it had to be done for any number of actions. Therefore, we had to change our code and we decided to follow an object oriented approach.