Master in applied sciences
Computer science and engineering
Year 2018-2019

Introduction to Complex problem solving

Assignement 1

INFO8003-1
Damien Ernst

Olivier Moitroux
s152122

Liège

Février 2019

2 Implementation of the domain

The code is mainly built upon two classes that are `Grid` and `Simulator`. The main parameters can be easily tuned in the calling script.

Domain representation

The class `grid` is used to represent the state space and takes as parameters the dimensions of the grid and an initialisation object (a number, list of list or the data of another grid) to fill the individual cells. It should be noted that the **origin** has been put in the **bottom left** of the board so that, in the provided domain instance of the assignment, the red cell is considered to be at position (0,1). The value of y increases as we go up (0,1) while the value of x increases by going to the right (1,0).

-3	1	-5	0	19
6	3	8	9	10
5	-8	4	1	-8
6	-9	4	19	-5
-20	-17	-4	-3	9

Figure 1 – Illustration of the test domain instance. 6 is considered to be in (0, 1). Going up (0,1) would result in a reward of 5.

The action space is represented by a dictionary `Moves` with direction (left-right-up-down) being the keys and the values the corresponding to a pair $(\Delta x, \Delta y)$.

A function `_rewardFun` simply return the value of the grid for the cell located in the input position.

Unit testing

Some testing have been done in order to test the behaviour of the grid by just affecting variables, playing around the move function and so on. All of the tests are available in `test.py`.

Simulation test

Back to the main script, a call to the function `simulateCell()` of the class `Simulator` simulates the execution of a stationary policy on a given cell of the grid. Redundant computations can be avoided by supplying to this function the previous result obtained as well as the initial corresponding time.

J1	J2	J3	J4
6	10.95	16.83	13.92

Table 1 – Result of simulation of the policy "always up" on the cell [0, 0].

3 Expected return of a policy

This time, the `simulator` will simulate the policy from every possible beginning states. The main script displays a 4-time-steps simulation in a deterministic setting and another 4-time-steps simulation in a stochastic setting. For this report, I will instead display graphically the result of several simulations. Figure 2 represents the evolution (over N) of the cumulative expected reward for each cell in the grid. Each column is assigned a colour for better visualisation. Indeed, it is expected that the cell belonging to the same column will get a close cumulative reward (offset) as the agent would rapidly be "stuck" against a wall. The table below show the correspondance between the colours and the number of the column in the grid (from left to right in figure 1)

red	green	blue	cyan	yellow
1	2	3	4	5

Table 2 – Mapping between column numbers and colours

Evolution of the expected cumulative result with N

Deterministic setting

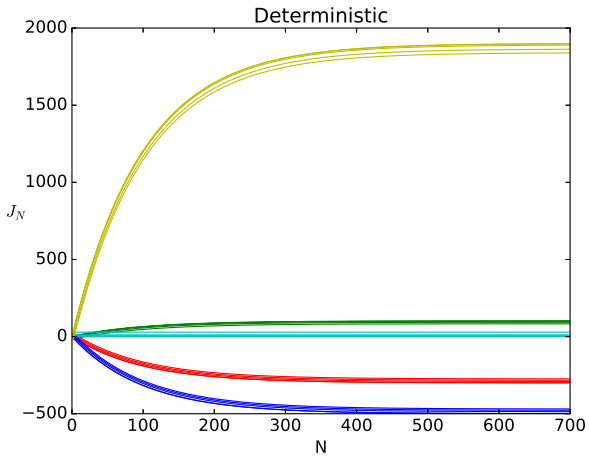
As can be observed from figure 2(a), the expected cumulative result seems to reach a plateau after approximately 500 time steps. This slow convergence is due to the value of gamma, 0.99, which is very close to 1. The right-most column reaches the highest value quite logically, as the reward for the corresponding upper position is the highest (19). On the other hand, an agent starting in the 3rd column, in blue, would face the upper wall and get stuck in a cell with a reward of -5 (the lowest).

Stochastic setting

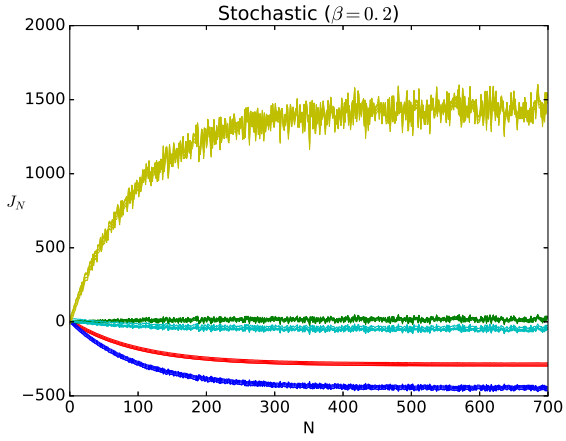
To generate figures 2(b), 2(c), each result for a given time step is averaged over 200 simulations. We can, again, highlight the same general trend as for the deterministic case. However, the lines are much less smooth. The range of reward values in the right-most column being quite different as the left-most one (cfr. return to [0, 0] in the dynamic), it seems reasonable to get less smooth yellow lines. In opposition, the red curves aren't really affected by the noise in the dynamic. The next position will be, indeed, guaranteed to be in the same column.

Evolution of the expected cumulative with beta at a fixed step

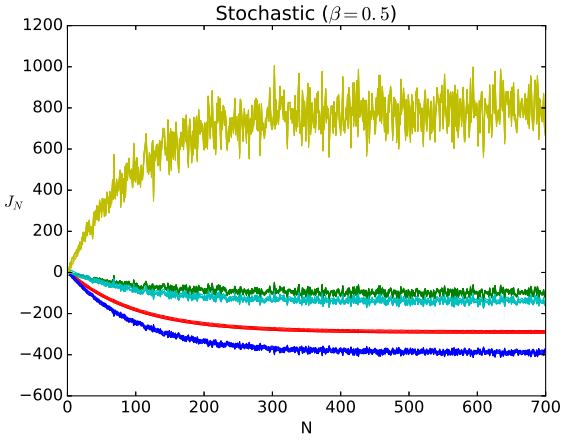
Figure 2(b) has been generated with $\beta = 0.2$. Increasing this value to $\beta = 0.5$ leads to a higher variance. Again, this makes sense as the reward values get more spread when considering a probability of $1/2$ to get back to [0, 0] (cfr. entropy). Pushing the value of β towards the bound of the interval [0, 1] will result in a decrease of the variance of the expected cumulative reward at a given time step. Figure 3 helps better visualise this effect.



(a) Evolution of J_N in a deterministic setting



(b) Evolution of J_N in a stochastic setting



(c) Evolution of J_N in a stochastic setting

Figure 2 – Evolution of The expected cumulative reward with N increasing

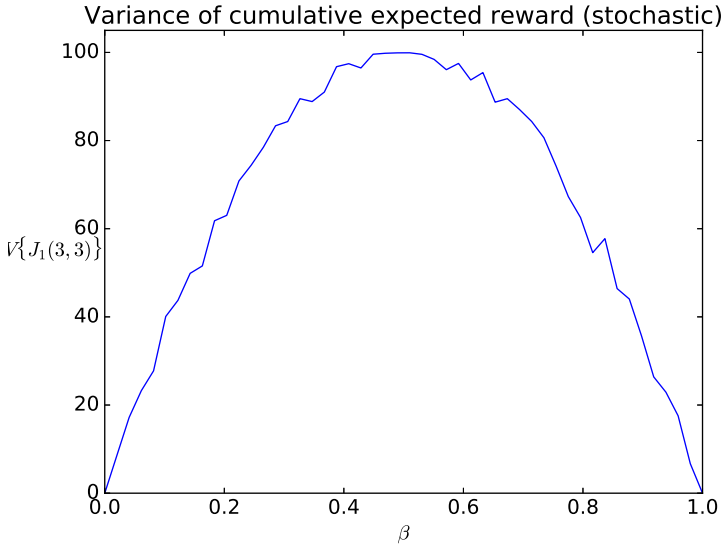


Figure 3 – Evolution of the variance of $J(3,3)$ when $N = 1$