Computational Game Theory
Assignment 3

Solutions should be sent in PDF format to kvmoffae@vub.ac.be
Deadline : December 3 2015

1 N-Armed Bandit

Implement the N-Armed bandit problem using each of the following action selection methods:

- Random
- ϵ-Greedy with parameter ϵ
- Softmax with parameter τ

All Q_{a_i} are initialized to 0. The rewards are subject to noise according to a normal probability distribution with mean $Q^*_{a_i}$ and standard deviation σ_i, $i = 1, \ldots, N$. For each of the following exercises, run your simulation for 1000 time steps, average over 1000 runs and plot the following graphs:

- One combined plot of the average reward for each algorithm (1 graph)
- One plot per arm showing the $Q^*_{a_i}$ of that action along with the *actual* Q_{a_i} estimate over time (All action selection methods should be depicted on the same plot) (One graph per arm).
- For each algorithm, plot a histogram showing the number of times each action is selected (One graph per action selection strategy).

See Section 2.2 of "Reinforcement Learning, An Introduction" by Sutton and Barto for a more comprehensive explanation of the problem[1].

1.1 Exercise 1

Let $N = 4$, $Q^*_{a_i}$ and σ_i as in table 1.

Action	$Q^*_{a_i}$	σ_i
action #1	2.3	0.6
action #2	2.1	0.9
action #3	1.5	2
action #4	1.3	0.4

Table 1: $Q^*_{a_i}$ and σ_i for each action.

Run the following algorithms:

- Random
- ϵ-Greedy with parameter $\epsilon = 0$
- ϵ-Greedy with parameter $\epsilon = 0.1$
- ϵ-Greedy with parameter $\epsilon = 0.2$
- Softmax with parameter $\tau = 1$
- Softmax with parameter $\tau = 0.1$

Comment the results. Which algorithms arrive close to the best arm after 1000 iterations? Which one arrives there faster? Explain your findings?

[1] http://www.cs.ualberta.ca/%7Esutton/book/ebook/node16.html

1.2 Exercise 2

Re-run Exercise 1 doubling the standard deviations of each arm, and comment the results. Discuss the results. Is this problem harder or easier to learn? Do the performance of the algorithms change significantly? Which of the above performs best now?

1.3 Exercise 3

Re-run exercise 1 with two additional algorithms, and plot their results. The new algorithms have a time-varying parameter, which depends on the iteration number $t = 1, \ldots, 1000$ as follows:

- ϵ-Greedy with parameter $\epsilon(t) = 1/\sqrt{t}$
- Softmax with parameter $\tau = 4 * \frac{1000-t}{1000}$

Discuss the results. Are these algorithms better than the ones with a fixed parameter?

2 Windy Gridworld

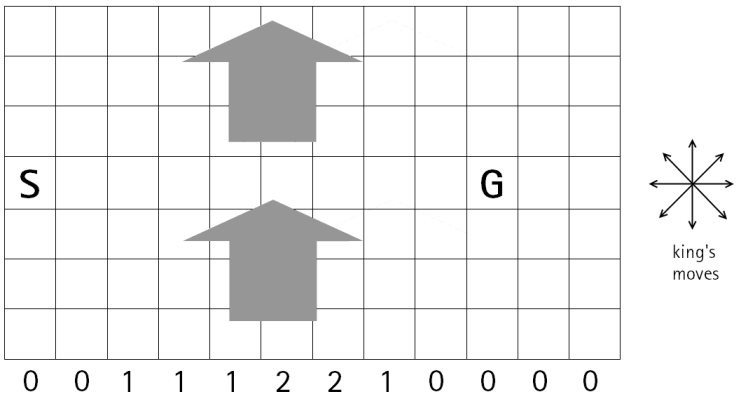

Figure 1: Gridworld in which movement is altered by a location-dependent, upward stochastic "wind".

Figure 1 shows a standard gridworld, with start (S) and goal (G) cells, but with one difference: there is a crosswind upward through the middle of the grid. The available actions in each cell are the king's moves — 8 actions in total for each cell. If any action would bring you outside the gridworld, you end up in the nearest cell (e.g. going northeast in the top left cell will bring you one cell to the right). In the middle region the resultant next cells are shifted upward by a stochastic "wind", the mean strength of which varies column by column. The mean strength of the wind is given below each column, in number of cells shifted upward. Due to stochasticity, the wind sometimes varies by 1 from the mean values given for each column. That is, a third of the time you are shifted upwards exactly according to the values indicated below the column, a third of the time you are shifted one cell above that, and another third of the time you are shifted one cell below that. For example, if you are two cells to the left of the goal (wind= 1) and you move *right*, then one-third of the time you end up one row north-east of that cell, one-third of the time you end up two rows north-east of that cell, and one-third of the time you end up at the same row east of that cell (left of the goal).

Implement the Q-learning algorithm[2] in the above problem with $\alpha = 0.1$, $\gamma = 0.9$ and initial $Q(s, a) = 0$ for all s, a. Each action generates a reward of $r_s = -1$, except for the actions that lead immediately to the goal cell ($r_g = 10$). Use the ϵ-Greedy action selection method with $\epsilon = 0.2$.

2.1 Plotting

Run the Q-learning algorithm for 5000 episodes and at the end of the learning plot the following on a single graph:

[2]The pseudo-code can be found at http://webdocs.cs.ualberta.ca/~sutton/book/ebook/node65.html

- the grid of the Windy gridworld
- arrow(s) in each cell indicating the best action(s)[3]
- one run from start to goal with the Greedy action selection (no exploration) on the learned policy showing the action selected in each cell (use here a different colour)
- for the same run above, plot a line displaying the path taken (i.e. all the visited cells) during that run

Figure 2 shows an example of a learned policy together with a sample run following that policy (applied to a slightly modified scenario).
In addition, plot the following two graphs:

- Total collected reward per episode versus the episode number
- Number of steps to reach the goal per episode versus the episode number

Discuss your results!

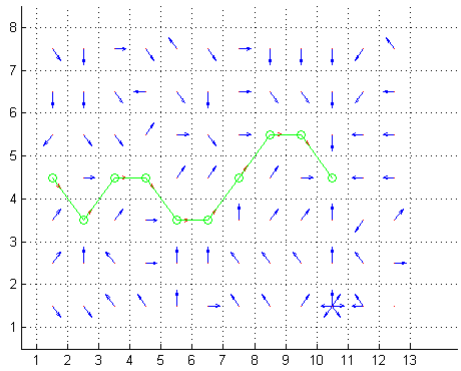

Figure 2: An example of a learned policy (arrows = best action) and a sample run following that policy (green path = visited cells). Note that in this example the scenario has been modified.

2.2 Discussion

Discuss the following:

1. How will the learning process change if we make $\epsilon = 0$, while keeping the other parameters the same?
2. How will the learning process change if we make $\gamma = 1$, while keeping the other parameters the same?

3 Windy Gridworld: Discussion

Elaborate on how you would consider the wind as a separate agent and use a sparse-interaction multi-agent learning algorithm in this context. How would you approach this? Which algorithm would you choose and why? Describe in detail!

NOTE: The choice of the programming language is free. Save yourself work by creating an abstraction for the number of actions and all parameters, so you can re-use your code. **Good luck !!!**

[3] In Matlab you can use the *quiver* function for this.