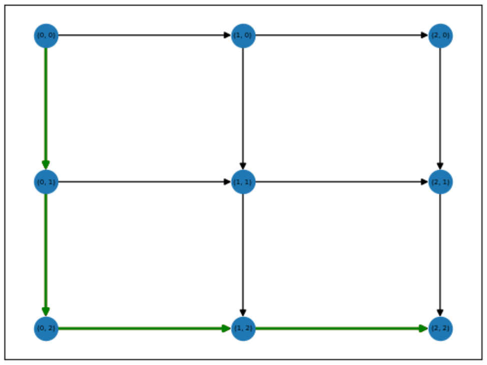
COMS W4701: Artificial Intelligence, Spring 2024

Homework 3b

Instructions: Compile all written solutions for this assignment in a single, typed PDF file. Coding solutions may be directly implemented in the provided Python file(s). **Do not modify any filenames or code outside of the indicated sections.** Submit all files on Gradescope in the appropriate assignment bins, and make sure to **tag all pages for written problems**. Please be mindful of the deadline and late policy, as well as our policies on citations and academic honesty.

Graph Bandits

Consider a graph traversal problem in which an agent can repeatedly traverse paths from a start to goal node. The edges have stochastic rewards drawn from log-normal distributions, but the means of these distributions are unknown. This can be posed as a multi-armed bandit problem, where each path is a separate bandit arm[1]. We can employ bandit algorithms to explore and learn about the weight distributions, with the ultimate goal of maximizing the agent's total rewards.



To simplify things, we will look at graphs with a grid structure, an example of which is shown above. The start node will always be on the upper left, and the goal node on the lower right. Every node, except the goal, may have either one or two successors, one below and/or one to the right. Thus, we only need to specify a *width* value to define a graph; the example above has width 3.

Source code for creating and displaying these graphs are in the `graph.py` and `utils.py` files. You should not need to reference these when writing the bandit algorithms. However, you do need to install the `networkx` library. You can verify that you have the right dependencies by simply running `python main.py` prior to starting, and you can use the `-w` option to specify the graph width (default is 2). You will see several copies of the graph pop up, along with a set of plots at the end, but they will be empty at this point.

[1]Credit to Professor Daniel Russo at the Columbia Business School for the problem formulation.

Part 1: Action Selection Strategies (6 points)

First, you will write the bandit action selection strategies in `bandit.py`. There are three such methods, each utilizing `self.value` as its respective parameter. Each should return the index of `self.Qvalues` corresponding to the chosen action.

- `choose_arm_egreedy()` implements ε-greedy action selection.
- `choose_arm_edecay()` implements ε-decaying action selection. Given the timestep t, number of bandit arms K, and strategy parameter c, it should compute a decayed value of ε as

$$\varepsilon = \min\left(1, \frac{cK}{t+1}\right).$$

  The larger the value of c, the longer that ε remains equal to 1 before starting to decrease. This allows the agent to perform pure exploration for a number of time steps.
- `choose_arm_ucb()` implements UCB action selection. The number of times that each arm has been chosen so far is stored in `self.arm_counts`.

Part 2: Bandit Simulation (12 points)

Now that we have our action selection strategies, we can write the `simulate()` method, which will run a bandit simulation for `self.N` steps. Each iteration first consists of action selection according to `self.strategy`: 0 for ε-greedy, 1 for ε-decaying, and 2 for UCB. Then pull the chosen arm using `pull_arm()`, and update the respective arm count and Q-value. The latter can be computed as an unweighted moving average of all rewards seen so far.

In addition, you should also define and update a NumPy array of length `self.N` storing the *regret* in each timestep. To do so, you should first note the best arm using `self.graph.shortest_path_ind()`. Then the regret is defined in each iteration as follows:

- If the best arm was chosen, then the regret is 0.
- Otherwise, call `pull_arm()` on the best arm. The regret is the difference between this reward and the actual reward received. (Note that we are computing actual, not expected, regret.)

Both the Bandit's `Qvalues` and `regret` arrays should be returned upon completion.

Part 3: Experiments (12 points)

This program has a number of arguments that allow you to modify the graph structure and weight generation, but for this assignment you will just experiment with the width. The `-s` option allows you to specify bandit strategy using integers 0, 1, or 2. Optionally, you can also set `-n`, number of steps per episode (default 10000), and `-m`, number of episodes total (default 100).

When a run is completed, you should see two sets of plots. The first set shows the underlying graph, with the upper left graph highlighting the optimal path and the "learned" paths using a bandit algorithm with three different parameter values. These should all mostly match, especially on smaller graphs. The second figure shows the cumulative regret over time (averaged over all episodes), and the proportion of paths taken that are the optimal one, also over time.

1. First test the ε-greedy strategy on a small graph (e.g., width 2) as well as a larger graph (e.g., width 6). You should see that the learned paths exactly or mostly match the optimal one (if not, try running it again). How does the rate of growth of the regret depend on ε? You may notice that the optimal path frequency is sometimes greater than $1-\varepsilon$, the exploit probability. Why is that?

2. Now try the ε-decreasing strategy, again starting with smaller graphs and then moving up to graphs of around width 6. Compare and contrast the results with those of (constant) ε-greedy. What range of values do we see regret go up to, and what do their rates of growth look like? Why is optimal path frequency now able to converge to 1 in many cases?

3. Finally, test out the UCB strategy. The results may bear some similarity to ε-decreasing, with some obvious distinct characteristics. For smaller graphs, you may sometimes see optimal path frequency shooting up to 1 very quickly. On the regret side, you may see the curves sometimes increasing by "steps" rather than doing so smoothly. Provide an explanation for each of these observations.

Please also attach a screenshot of the regret and optimal path path frequencies for each method on a graph of width 4.

Coding Submissions

You can submit just the completed `bandit.py` file under the HW3b Coding bin on Gradescope.