CE811: Game Artificial Intelligence

Lab 3: Control

27th October 2016

The objective of this lab is to implement some of the algorithms seen in the previous lecture about control in RL. For this, we will us the small gridworld example as a benchmark to test these algorithms. There is some initial code that you can download from the website to work on.

The Small GridWorld problem

In this lab, we will use the same benchmark employed in the last one: the small gridworld problem.

The small gridworld problem is a simple benchmark that is represented by a grid of $N \times N$ dimensions. All positions in the grid that are not terminal states provide a reward of -1, while the terminal states come with a reward of 0. There are again two versions of the gridworld implemented in the software provided for this lab, one that has the terminal state at the center of the grid and another with two terminal states, placed at the upper left and lower right corners.

In this benchmark, each state is defined as a position in the grid (row,col), and the possible actions are moving one cell Right, Left, Up and Down. If any of these actions would take an agent outside the grid, the agent does not move.

If an agent were to be dropped in any position of the grid, his best (optimal) strategy would be to move (one cell at a time) towards the closest terminal state, using the shortest path to it.

The code provided

The assets provided for this lab are a bunch of Java files. Take your time to familiarize yourself with the structure of the code. Here there is a brief description:

- **(New to Lab 3)** Package *algorithms.control*: This packages contains:
  - A base class for the different control algorithms (*Control*).
  - The skeletons of the classes where you will program the different control algorithms.
  - A class *Agent* that simulates an agent using a control algorithm to reach one of the goals of the *GridWorld* problem.
- Package *algorithms.planning*: Algorithms for planning: Monte Carlo, Value Iteration and Temporal Difference Learning.
- Package *policy*: Defines an abstract class for policies, the random policy and the skeletons for the Greedy and the ϵ-Greedy policies.
- Package *gridworld*: describes the gridworld and the two variants of the problem.
- Package *utils*: Contains some useful stuff for this codebase.

1 Preparing the control algorithm

One of the main aspects that we need to work on in order to program the control algorithms is preparing the policies. First of all, in the class Policy, there are a few new require methods. Here's a summary:

- A new couple of abstract methods have been added in order to give the probabilty of picking an action and sampling an action based on the action-values ($q(s, a)$) instead of the state-values ($v(s)$). These functions will need to be implemented in the inherited classes.
- Two methods for sampling an action based on the probabilities, one for state-value and the other for action-state values.
- A method (*moveGreedily*) that, instead of sampling, returns the action that maximizes the state-action pair $q(s, a)$ from a given state.

Implement the following functions (like in the past lab, look for the `TODO` tags in the code):

- Both `sample` functions in *Policy.java.* Hint: the code for these functions should be no longer than about 10 lines... and you can actually get around it using the same code! Make good use of the functions `prob` and `choiceWeighted`.
- The contents of the function `moveGreedily` in *Policy.java.*

Unfortunately we cannot test this yet. One of the things we need is to implement the functions `prob` and `sampleAction` in the subclasses (*RandomPolicy*, *GreedyPolicy* and *EGreedyPolicy*). The code provides this implementation for the state-values $v(s)$ in all subclasses, and the action-values $q(s, a)$ for the random policy.

Therefore, **implement** now `prob` and `sampleAction` in the classes *GreedyPolicy* and *EGreedyPolicy.*

2 Monte Carlo Control

The choice of which control algorithm is used by the agent is in the function `setControl` in `Agent.java`. You will need to modify this method to provide the agent with one or another control algorithm.

For Monte Carlo Control, you need to go to *MonteCarloControl.java* and implement the function `execute`. In the `execute()` method, you should also update the values `minVal` and `maxVal` with the minimum and maximum values of $v(s)$ for all $s \in S$ in the gridworld.

This method must perform one iteration of the algorithm, which consists of:

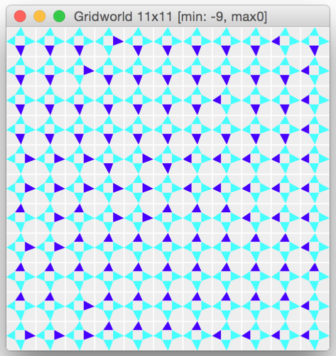
- Iterate through all states in the grid.
- For each state, if it is terminal, all the values $q(s, a)$ should be updated to the reward of a terminal state.
- It the state is not terminal, the state-action value $q(s, a)$ should be updated, for each action, according to the MC update rule (remember: G_t is the Return):

$$N(s_t, a_t) \leftarrow N(s_t, a_t) + 1$$
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \tfrac{1}{N(s_t,a_t)}(G_t - Q(s_t, a_t))$$

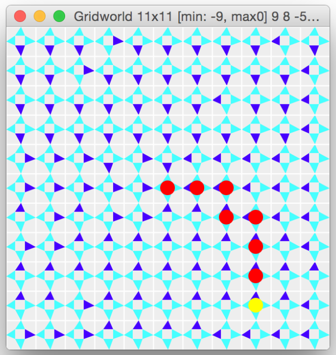
Once this has been implemented, you can test if it works by running *Test.java.* For this lab, the frame shows, on each state, an arrow that indicates the directions from that position in the grid. The colours of these arrows are light or dark blue, being the latter the actions that have the highest state-action value $q(s, a)$ from that state or position in the grid. A possible outcome of the Monte Carlo Control algorithm for a 11×11 gridworld with the terminal state in the center is shown below.

Another interesting way of testing the system is by allowing the agent to move around the level in order to arrive at the terminal state. This agent would select, for each state that is visited, the action that maximizes the state-action $q(s, a)$ value for that state. Implement the method `move` in *Agent.java*, so it returns the action that the agent would take from its current state `(cRow, cCol)` (The line that adds the state to the array `positions` is there just for graphical purposes).

You can test your algorithm by running *Test.java*, indicating the number of iterations that you want to train (by calling the method `ag.train()`).



The main method in *Test.java* also contains a piece of code that, once the algorithm has finished training, allows the user to click on a cell of the grid. This represents the initial state of the agent (and it's painted as a yellow dot). If the algorithm and the move functions were implemented correctly, you should see how a sequence of red dots indicates the trajectory to the closest terminal state. This trajectory (if the number of iterations is high enough) should be optimal. Note that this means that any change in position should be done through a dark blue arrow, and the number of moves is the minimum achievable. An example of these trajectories is shown also below, for the gridworld (center) problem.



3 Q-Learning

Once everything works with Monte Carlo Control, implement the *Q-Learning* method. Remember that you can modify the control algorithm the agent uses inside the method `setControl` in *Agent.java*. For *Q-Learning*, implement the method `qlearning()` in *QLearning.java*. Pay attention to the instruction indicated in the comments of this method. Try again different trajectories in the gridworld frame to test the algorithms.

Questions

See if you can answer the following questions (ask if you can't!):

- Which one of the implemented algorithms works best? How does this change when modifying the dimensions of the gridworld?
- Which one of the algorithms seen here converge faster to a stable solution? When do you know that this has happened?
- How would you modify the training to stop when the algorithm has converged?
- If you have some time, try to implement SARSA in `Sarsa.java`.