# CE811: Game Artificial Intelligence

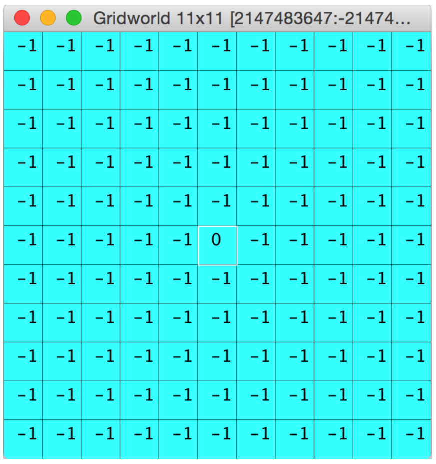
## Lab 2: Planning

20th October 2016

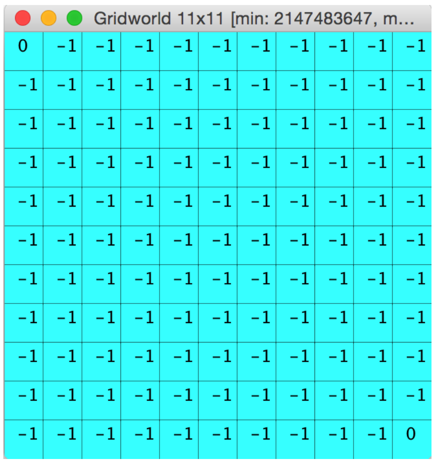
The objective of this lab is to implement some of the algorithms seen in the previous lecture. For this, we will us the small gridworld example as a benchmark to test these algorithms. There is some initial code that you can download from the website to work on.

## The Small GridWorld problem

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 actually two versions of the gridworld implemented in the software provided for this lab:



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | 0 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |



| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 |
| -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 0 |

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. The scenario on the left has the terminal state at the center of the grid, while the scenario on the right has two terminal states, placed at the upper left and lower right corners.

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:

- Package *algorithms.planning*: It contains a class, *PlanningAlgorithm*, that is the base class for all algorithms that you will develop in this lab.
- Package *policy*: Defines an abstract class for policies, and the skeleton of the random policy.

- Package *gridworld*: describes the gridworld and the two variants of the problem.
- Package *utils*: Contains some useful stuff for this codebase.

## 1 Preparing the planning algorithm

One of the basic things we need for the subsequent algorithms to program is a function that calculates the return form a state, following a policy. You can implement this function in `PlanningAlgorithm.getReturn(row, col)` (look for the `TODO` tag). Note: use the value of gamma $\gamma$ to limit the duration of the look ahead. This is, define a minimum value for $\gamma$ so the algorithms stops iterating when reached certain number of steps.

Another important aspect is to prepare the policies. In this assignment, we'll start working with a random policy. The class *Policy* declares two functions to retrieve the probability of taking a certain action from a position in the grid (`prob`) and a function sampleAction, that returns an action sampled from the possible actions from a state.

You need to implement now these two functions in the class *RandomPolicy*, look for the `TODO` indicators.

## 2 Value Iteration

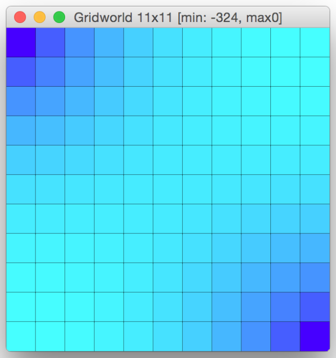
The algorithm Value Iteration has a dedicated class *ValueIteration* where you need to implement two aspects (again, look for the `TODO` indicators):

- Initialization of the $v(s)$ values for all states in the grid. You can do this in the constructor of *ValueIteration*.
- A helper function that returns the highest value $v(s')$ after applying one of the possible actions from the current state. This is used in the value iteration algorithm.
- A single iteration of the algorithm, in the function `execute()`. This method should generate the next set of state-value values $v(s)$, iterating through all states in the gridworld, and updating the value of v(s) according to the value iteration update:

$$v(s) \leftarrow \max_{a \in A} R_s^a + \gamma \sum_{s'} p(s' \mid s, a) v_k(s')$$

In the `execute()` method, you should update the values `minVal` and `maxVal` with the minimum and maximum values of $v(s)$ for all $s \in S$.

If you run the class *Test*, you will be able to check if the algorithm works or not. You will see a frame showing the states of the gridworld, with different colours in it. Darker blue means higher $v(s)$ values, while lighter blue represents lower values. A correct result will show darker blue near the terminal states, fading into light blue as the distance from these increase. Something like this:



In some cases, it is difficult to appreciate the difference in the tones. Because of this, you can also click on each square and the title bar will show the $v(s)$ value of that particular cell. You should see how these values increase as you select values closer to the terminal states. The program also prints the $v(s)$ values to console.

## 3 Dynamic Programming

Make a copy of the Value Iteration class and change the update rule so it uses a more generic Dynamic Programming rule, such as (using the random policy):

$$v(s) \leftarrow \sum_a \pi(a \mid s) \Big( R_s^a + \gamma \sum_{s'} p(s' \mid s, a) v_k(s') \Big)$$

What differences can you see between these two approaches?

## 4 Monte Carlo and Temporal Difference Learning

Once you are done with Dynamic Programming and Value Iteration, do the same for Monte Carlo (in MonteCarlo.java) and Temporal Difference Learning (in TD.java). You can test these two methods in the same way as the previous methods. Check the `TODO` indicators $6 - 9$ to identify where to insert the new code.

## Questions

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

- Try different sizes for the gridworld. How do they compare when the size changes?
- Can you see any difference between the algorithms? Which ones work better for each size?
- Also, have you realized the time it takes for each algorithm to reach an optimal solution? Actually, how can we know if the solution obtained is optimal or not?
- If you have some time to spare at the end of the lab, try to implement TD($\lambda$), as described in the slides.