# EECS 492: Introduction to AI
# Homework 1 (100 pts)

## General Information

Due: 11:59 PM, September 19, 2016
Notes:

- For the written questions (Problem 1 and Problem 2 Part 5), put your answers in a pdf (with your name in the pdf) and submit to the corresponding assignment on Gradescope.
- For the coding questions (Problem 2 Parts 1-4), submit your code file firstname_lastname_agents.py to the corresponding assignment on canvas, and also **append the code** to the end of your pdf.
- Late homework will be penalized 10% per day (each day starts at 11:59 PM on the due day).
- Homework turned in after three days **will not be accepted**.

**Reminder**: All solutions must be your own. For any solutions that require programming, please comment and format your code to make it readable. You should code your solutions in **Python 2**, and your code should be able to run on CAEN.

## Problem 1: Characterizing an Agent's Environment [28 points]

In class we talked about specifying a task environment via the PEAS description. The PEAS description contains the following specifications: Performance Measure, Environment, Actuators, and Sensors.

In addition, environments can be described in terms of the following properties:

- Fully/Partially observable
- Deterministic/Stochastic
- Episodic/Sequential
- Static/Dynamic
- Discrete/Continuous
- Single-agent/Multi-agent

Describe the following agents using (A) the PEAS description and **also** (B) a description of the task environment. Justify your answers, briefly (no more than 1 or 2 sentences per term). Each question is worth 4 points (2 for PEAS and 2 for task environment).

1. Crossword Puzzle Solver
2. Roomba (https://en.wikipedia.org/wiki/Roomba)
3. Netflix Automatic Recommender System

4. Home Alarm
5. Boston Dynamics BigDog (https://www.youtube.com/watch?v=cNZPRsrwumQ)
6. Siri (https://www.apple.com/ios/siri/)
7. Paro Therapeutic Robot (http://www.parorobots.com/)

---

## Problem 2: Implementing an Automatic Farming Agent (72 pts)

Consider an automated farming robot, FA-bot. It operates in a grid world, similar to the world that the vacuum robot inhabits. The FA-bot can do four things:

1. It can move around the farm
2. It can sense properties of its environment
3. It can water crops
4. It can weed

The environment of the FA-bot is broken into grid squares. Each square may need to be weeded or it may need to be watered. The locations are assigned using grid positions (even if not all squares are available). The FA-bot must weed all squares that contain weeds, and water all squares that have insufficient water. See Figure 1 for an illustration.

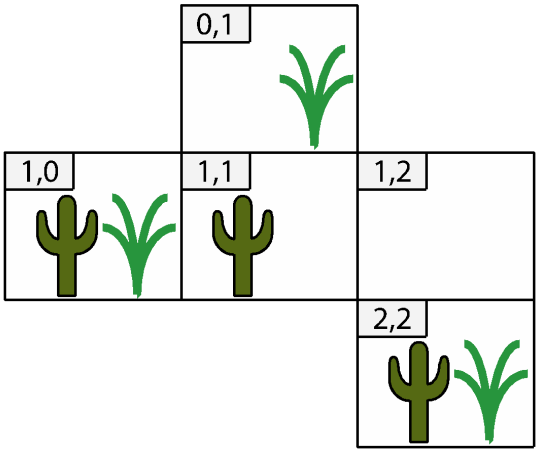


**Figure 1: Board Configuration – a square with a cactus indicates that the square needs to be watered, a square with a green plant indicates that it has to be weeded, and the square number is in the upper left corner (top left is position [0,0]).**

Environments are described by the text files env#.txt, where # is the environment number (e.g., env1.txt, env2.txt,...). The first line of the file specifies the number of rows and the number of columns of a grid. The second line contains the following configuration information, in order:

1. Starting agent row
2. Starting agent column

3. Direction agent is initially facing (north, south, east, west)
4. Initial amount of water
5. How much water is used per watering
6. Initial amount of power
7. Power used for watering
8. Power used for weeding
9. Power used for moving
10. Power used for sensing

The remaining rows describe the environment in which the FA-Bot finds itself. Each square contains either two Boolean values (e.g., TF) or two xx's (e.g., xx). Squares marked xx cannot be traversed by the agent. The other squares describe the need for watering (if the first Boolean value is true) and the need for weeding (if the second is true).

The following is an example of an environment file that corresponds to Figure 1:

```
3 3
0 1 north 100 1 100 1 1 1 1
xx FT xx
TT TF FF
xx xx TT
```

This means that the agent starts in square [0,1] facing north, has 100 units of water and power, and uses 1 unit of water with each watering action in addition to the 1 unit of power that any action consumes. If, for example, the agent sensed the characteristics of square [0, 1], it would learn that the square does not need to be watered (F), but does need to be weeded (T).

The goal configuration for this world is:

```
xx FF xx
FF FF FF
xx xx FF
```

The agent is successful only if there are no squares left to weed or water before the agent runs out of power, and no squares are left to water before the agent runs out of water. The agent should also try to conserve as much water and power as possible.

Note: you are provided with code for reading in these files.

---

## Part 1: Implementing a Simple Reflex Agent (5 pts)

### *Problem setup*

You have decided to use your AI knowledge to build an agent that will allow you to water and weed the farmland. You will first build a simple reflex agent for this task by completing the "choose_action" function under the "simple_reflex" class in agents.py. This agent cannot maintain any internal state variables, so its behavior is defined entirely by the current percept. The agent will receive a sequence of percepts, and take an action after each percept, until either success or failure is achieved. The actions and corresponding percepts are listed below:

| Actions | Description | Possible Resulting Percepts |
|---|---|---|
| - | Beginning the simulation | start |
| sense_water | Sense whether the current square needs watering | needs_watering, does_not_need_watering |
| sense_weed | Sense whether the current square needs weeding | needs_weeding, does_not_need_weeding |
| water | Water the current square | watering_succeeded |
| weed | Weed the current square | weeding_succeeded |
| move | Move in the direction the agent is facing. If there is a barrier, the agent remains where it is, but rotates clockwise. | move_succeeded, hit_barrier |

The agent will receive the "start" percept at the beginning of a simulation. The simulator will halt when it receives a success or failure signal and print the results.

You should write this agent to do **as well as possible**, given its circumstances, constraints, and goals. (So there is at least one right answer, which will produce optimal output for this agent; don't just return the same action every time and call it good).

---

## Part 2: Implementing a Reflex Agent with State (5 pts)

### Problem setup

You have decided that your FA-bot can do better by reasoning about its environment. To do this it needs state.

You will implement a reflex agent with state, under the state_reflex class in agents.py. It will plan its movements using breadth-first search (implemented for you in world.py).

### What can the agent do?

This new agent can move in any desired direction. Its other actions are the same as before. The actions are described in the table below:

| Actions | Description | Possible Resulting Percepts |
|---|---|---|
| - | Beginning the simulation | start |
| sense_water | Sense whether the current square needs watering | needs_watering, does_not_need_watering |
| sense_weed | Sense whether the current square needs weeding | needs_weeding, does_not_need_weeding |
| water | Water the current square | watering_succeeded |
| weed | Weed the current square | weeding_succeeded |
| move_north | The agent moves a space north, or does nothing if a barrier is hit | move_succeeded, hit_barrier |
| move_east | The agent moves a space east, or does nothing if a barrier is hit | move_succeeded, hit_barrier |
| move_south | The agent moves a space south, or does nothing if a barrier is hit | move_succeeded, hit_barrier |
| move_west | The agent moves a space west, or does nothing if a barrier is hit | move_succeeded, hit_barrier |

Unlike the agent of Part 1, this agent can see the layout of the environment in advance (though it does not know what needs to be watered and weeded until it senses a square). The agent is initialized by searching over the world with breadth-first search for a shortest path of movement through the environment. It stores this movement plan, and how many moves it has made, as state variables. This agent **may not use other state variables** besides the ones we have defined for you ("plan" and "position").

(Again, you should write this agent to do **as well as possible**).

---

## Part 3: Implementing a Randomized Simple Reflex Agent (5 pts)

### Problem setup

You want to see whether adding randomness to an agent can cause it to outperform the simple reflex agent, without using state or performing any search in advance.

### What can the agent do?

This randomized agent, implemented under class random_reflex in agents.py, has the same set of actions as in Part 2, but does not maintain any internal state, and chooses among move_north, move_east, move_south, and move_west uniformly at random when it decides to move. You may use the numpy.random.randint function for randomness.

---

## Part 4: Improving the Deterministic Simple Reflex Agent (8 pts)

### Problem Setup

You get to have some fun with this one. Suppose that your agent cannot know any information about the environment in advance. You want to improve the simple reflex agent from Part 1 without adding any randomness to your code. To do this, you allow the agent to maintain state variables, though it cannot do any search on the structure of the world or otherwise see the world in advance, as it could in Part 2.

Implement the class better_reflex_agent in agents.py, such that the resulting **deterministic** agent **strictly outperforms** the agent from Part 1 on **at least one** environment, and **does no worse than** that agent on any other environment, including the held-out ones. This may require a bit more thinking than the other parts, but you have more flexibility.

### What can the agent do?

This agent can initialize and maintain any number of state variables, store any amount of prior percepts/actions, use any **built-in** python data structures, and perform any number of additional computations necessary, as long as they are deterministic. **It may only use the actions available to the agent in Part 1**: sense_water, sense_weed, water, weed, move.

---

### Provided Code

In the file world.py we provide you with fully implemented classes GridSquare, and World, as well as several classes and functions for breadth-first search. The class World reads in an environment file during initialization and provides an interface for performing actions, getting percepts, and tracking the agent's water and power levels.

The file simulator.py contains the fully implemented functions simulate_agent() and run_simulations(). Simulate_agent() runs a particular agent on a particular world and prints the outcome. Run_simulations() is the function that we will use to test your code.

You should read the code files to make sure you understand how the provided code works.

### Code You Must Write

For parts 1, 2, 3, and 4, you must fill in the designated sections of the file agents.py by implementing the choose_action function for the four agent classes. The choose_action functions of the first three agent classes are very similar to each other, and should not take many lines of code.

### How Simulations Work

After the world is initialized, agents are created and initialized. Each simulation begins by feeding the agent the percept "start" via agent.choose_action(), and obtaining the agent's resulting action. The action is performed on the world and results in another percept, which is again fed into the agent. This process continues until the simulator receives either a "success" or "fail" percept, at which point it will halt and print the results. Your agent does not need to handle the "success" or "fail" signals. The world maintains the agent's position, direction, and power and water levels, so you don't have to.

### How we will run your code

We will test your code on some small held-out environments, mainly to make sure that it runs and that it does what it's supposed to. We will import your (renamed) version of agents.py into simulator.py and run

python simulator.py <firstname_lastname_agent> <env_name.txt> <agenttype>

from the command line, for each environment and agent type, where "firstname_lastname_agent" is the name of your agent file without the file extension, env_name.txt is an environment file, and "agenttype" is one of simple_reflex, state_reflex, and random_reflex. For example,

python simulator.py john_doe_agent env1.txt simple_reflex

You can test your code with these commands. Add 'v' to the end of the line for verbose output.

Your code's performance here is worth 6 points.

---

## Part 5: Written Questions for Problem 2

For Problem 2, include answers to the following questions in your pdf. Questions are worth 4 pts each, unless otherwise specified.

1. Write a PEAS description for FA-bot's environment in Part 1.
2. What is the definition of rational behavior in FA-bot's environment?
3. Can the simple reflex agent from Part 1 be perfectly rational for this environment? What about the agent from Part 2? Why or why not?
4. For **each** of parts 1, 2, 3, and 4, fill in a table like the following that contains the outcomes of each agent running on **each** provided environment. (For the randomized agent, there is, of course, no single correct answer. Provide averages over 5 runs, with percent of times succeeded in the "succeeded" column).

| Env. # | Succeeded? | # Moves | Water | Power | # Left to water | # Left to weed |
|---|---|---|---|---|---|---|
| | | | | | | |

5. Which agent performed the best overall? Why? Does your randomized simple reflex agent from Part 3 outperform your deterministic agent from Part 1 in any environments, and why? List the ones that it does, if any.
6. (7 pts) In which environment(s) did your agent from Part 4 outperform the agent from Part 1? Describe how you structured the better reflex agent, and why it performs better in some cases and no worse in others. (Or, if you did not succeed in making such an agent, describe what you were trying to do).

Resource constraints lead to trade-offs between different strategies that agents could implement. Consider the following simplified scenario: an agent must move along a single row of grid squares, which either do or don't contain weeds with about 50% probability (e.g., the grid looks like F T T F T …). The agent can move forward, sense for weeds, and weed (like our agents above, the weed action can be performed whether or not there is actually a weed). The agent wants to weed all squares while using as little power as possible.

7. Describe this agent's optimal strategy assuming it costs 1 unit of power to sense for weeds and 5 units of power to weed.
8. How does this agent's strategy change if it costs 5 units of power to sense for weeds and 1 unit of power to weed?

The internal logic of our agents above was greatly simplified by *environmental determinism*. We knew that if the agent attempted a "weed" action, for example, the action would succeed with certainty. But, what happens when certainty is no longer guaranteed? Assume that we are working with the state-based reflex FA-bot that you implemented from Part 2.

9. Now our FA-bot is on the fritz. We have been meaning to replace the agent, but, we haven't. As a result, the "weed" action only succeeds in removing weeds about 75% of the time! How would the performance of the state-based agent from Part 2 be affected by this indeterminism?
10. Describe how you would you modify that agent to handle this sort of indeterminism.

## Rules for Programming and Submission

- After you fill out agents.py, **rename it** to firstname_lastname_agents.py and submit it in canvas under assignment 1. (Of course, "firstname" and "lastname" should be replaced with your actual first and last name).
- Submit a neat pdf of answers to the questions (with your code pasted at the end) to Gradescope. Be sure to identify the correct pages with the correct problems. **Put your name in the pdf.**

- Include a comment at the top of your source file with your name
- Your program must run from a command line on CAEN computers.
- Use good style and layout. Comment your code well.

## Answer Format and Grading

- Answers to questions should be complete but concise - don't be more wordy than necessary. Most questions can be answered in a couple of sentences at the most.
- Partial or (possibly) full credit will be given to answers that are well-explained, even if they are different from our answers. (Or code that is well-written, even if it has errors).
- Points are broken down as follows:
    - 28 pts for Problem 1, or 4 pts per subproblem
    - 5 pts for the agent code of Problem 2 Part 1
    - 5 pts for the agent code of Problem 2 Part 2
    - 5 pts for the agent code of Problem 2 Part 3
    - 8 pts for the agent code of Problem 2 Part 4
    - 6 pts for all your agent code running and producing good output on our test environments
    - 43 pts for Problem 2, Part 5, 4 pts per subproblem, with 7pts for Q6.