6CCS3AIN & 7CCSMAIN, 2018

Coursework 2

(Version 1.0)

1 Introduction

This coursework exercise asks you to write code to create an MDP-solver to work in the Pacman environment that we used for the practical exercises.

Read all these instructions before starting.

This exercise will be assessed.

2 Getting started

You should download the file `pacman-cw2.zip` from KEATS. This contains a familiar set of files that implement Pacman, and version 5 of `api.py` which defines the observability of the environment that you will have to deal with, and the same non-deterministic motion model that the practicals used.

Version 5 of `api.py` (which you will already have met if you worked through Practical 7), further extends what Pacman can know about the world. In addition to knowing the location of all the objects in the world (walls, food, capsules, ghosts), Pacman can now see what state the ghosts are in, and so can decide whether they have to be avoided or not (we will ignore the score when marking the coursework so there is no need to chase ghosts and try to eat them).

3 What you need to do

3.1 Write code

This coursework requires you to write code to control Pacman and win games using an MDP-solver. There is a (rather familiar) skeleton piece of code to take as your starting point in the file `mdpAgents.py`. This code defines the class `MDPAgent`.

There are two main aims for your code:

(a) Be able to win at least four games in ten in `smallGrid`

(b) Be able to win at least two games in ten in `mediumClassic`

To win games, Pacman has to be able to eat all the food. For this coursework, "winning" just means getting the environment to report a win. Score is irrelevant.

3.2 Things to bear in mind

Some things that you may find helpful:

(a) We will evaluate whether your code can win games in `smallGrid` by running:

`python pacman.py -q -n 10 -p MDPAgent -l smallGrid`

`-l` is shorthand for `-layout`. `-p` is shorthand for `-pacman`. `-q` runs the game without the interface (making it faster).

(b) We will evaluate whether your code can win games in `mediumClassic` by running:

`python pacman.py -q -n 10 -p MDPAgent -l mediumClassic`

The `-n 10` runs ten games in a row.

(c) When using the `-n` option to run multiple games, the same agent (the same instance of `MDPAgent.py`) is run in all the games.

That means you might need to change the values of some of the state variables that control Pacman's behaviour in between games. You can do that using the `final()` function.

(d) There is no requirement to use any of the methods described in the practicals, though you can use these if you wish.

(e) If you wish to use the map code I provided in `MapAgent`, you may do this, but you need to include comments that explain what you used and where it came from (just as you would for any code that you make use of but don't write yourself).

(f) You can only use libraries that are part of a the standard Python 2.7 distribution. This ensures that (a) everyone has access to the same libraries (since only the standard distribution is available on the lab machines) and (b) we don't have trouble running your code due to some library incompatibilities.

3.3 Write a report

Write up a description of your program along with your evaluation in a separate report that you will submit along with your code.

As you work through your implementation of the MDP-solver, you will find that you are making lots of decisions about how precisely to translate your ideas into working code. The report should explain these at length. The perfect report will give enough detail that we don't feel we have to read your code in order to understand what you code does (we will read it anyway).

Given the requirement for your code to be based around an MDP-solver, you would be wise to include a description of how your code solves the MDP, and which bits of the code do this solving.

Remember, when writing your report, that there is credit for creative and beautiful solutions. Make sure you highlight these aspects of your work, especially things that make your work unique.

Having said that, reports that are needlessly long will not get any more marks. We value concise reports (we have to read a lot of them).

Your report should also analyse the performance of your code. Because there is a certain amount of randomness in the behaviour of the ghosts, a good analysis will run multiple games to assess Pacman's performance. For example, you might like to try running:

```
python pacman.py -n 50 -q -p MDPAgent -l mediumClassic
```

to get a statistically significant number of runs so that you can, for example, establish the average score with some accuracy. (Of course, to decide whether this was a statistically significant number of runs, you would have to do some statistical analysis — it might well need more runs.) All the conclusions that you present in your analysis should be justified by the data that you have collected.

3.4 Limitations

There are some limitations on what you can submit.

(a) Your code must be in Python 2.7.

Code written in a language other than Python will not be marked.

Code written in Python 3.X is unlikely to run with the clean copy of `pacman-cw2` that we will test it against. If is doesn't run, you will lose marks.

Code using libraries that are not in the standard Python 2.7 distribution will not run. If you choose to use such libraries and your code does not run as a result, you will lose marks.

(b) Your code must only interact with the Pacman environment by making calls through functions in Version 5 of `api.py`. Code that finds other ways to access information about the environment will lose marks.

The idea here is to have everyone solve the same task, and have that task explore issues with non-deterministic actions.

(c) You are not allowed to modify any of the files in `pacman-cw2.zip` except `mdpAgents.py`.

Similar to the previous point, the idea is that everyone solves the same problem — you can't change the problem by modifying the base code that runs the Pacman environment.

(d) You are not allowed to copy, without credit, code that you might get from other students or find lying around on the Internet. We will be checking.

This is the usual plagiarism statement. When you submit work to be marked, you should only seek to get credit for work you have done yourself. When the work you are submitting is code, you can use code that other people wrote, but you have to say clearly that the other person wrote it — you do that by putting in a comment that says who wrote it. That way we can adjust your mark to take account of the work that you didn't do.

(e) Your code must be based on solving the Pacman environment as an MDP. If you don't submit a program that contains a recognisable MDP solver, you will lose marks.

4 What you have to hand in

Your submission should consist of a single ZIP file. (KEATS will be configured to only accept a single file.) This ZIP fIle must include a single PDF document (the report), and a single Python file (your code).

The ZIP file must be named:

`cw2-<lastname>-<firstname>.zip`

so my ZIP file would be named `cw2-parsons-simon.zip`.

Your report must be named:

`cw2-<lastname>-<firstname>.pdf`

so my report file would be named `cw2-parsons-simon.pdf`. Reports that are not in PDF format will lose marks.

Remember that we are going to evaluate your code by running your code by using variations on

```
python pacman.py -p MDPAgent
```

(see Section 5 for the exact commands we will use) and we will do this in a vanilla copy of the `pacman-cw2` folder, so the base class for your MDP-solving agent must be called `MDPAgent`.

To streamline the marking of the coursework, you must put all your code in one file, and this file must be called `mdpAgents.py`,

Do not just include the whole `pacman-cw2` folder. You should only include the one file that includes the code you have written.

Submissions that do not follow these instructions will lose marks. That includes submissions which are RAR files. RAR is not ZIP.

5 How your work will be marked

There will be six components of the mark for your work:

(a) Functionality

We will test your code by running your `.py` file against a clean copy of `pacman-cw2`.

As discussed above, for full marks for functionality, your code is required to:

(a) Be able to win four games out of ten in `smallGrid`
This will be assessed running your code with the command:
`python pacman.py -n 10 -q -p MDPAgent -l smallGrid`
and checking to see if it wins at least four games.

(b) Be able to win two games out of ten in `mediumClassic`
This will be assessed by running your code with the command:
`python pacman.py -n 10 -q -p MDPAgent -l mediumClassic`
and checking to see if it wins at least two games.
Since we will check it this way, you may want to reset any internal state in your agent using `final()` (see Section 3.2).

Partial credit will be given for code that fails to meet these requirements.

Since we have a lot of coursework to mark, we will limit how long your code has to demonstrate that it can win. We will terminate the run of the of `smallGrid` games after 2 minutes, and will terminate the run of the `mediumClassic` games after 5 minutes. If your code has failed to win within these times, we will mark it as if it lost. Note that we will use the `-q` command, which runs Pacman without the interface, to speed things up.

Code not written in Python will not be marked.

(b) Style

There are no particular requirements on the way that your code is structured, but it should follow standard good practice in software development and will be marked accordingly.

Remember that your code is only allowed to interact with the Pacman environment through version 5 of `api.py`. Code that does not follow this rule will lose marks.

(c) Documentation

All good code is well documented, and your work will be partly assessed by the comments you provide in your code. If we cannot understand from the comments what your code does, then you will lose marks.

(d) Report

The contents of the report are described above. Your work will be assessed against the criteria laid out there. The report must be submitted as a PDF document.

(e) Results

In addition to looking at your experimental results to assess the functionality of your code, we will be looking to check that you did some evaluation, that you analysed the data as required, and that you have drawn sensible conclusions from the experiments. Proper statistical analysis will be credited.

(f) Creativity

In order to give credit to students who come up with particularly beautiful, sophisticated and/or creative solutions, there will be some marks available to recognise solutions that go beyond the average.

As with all instances of creativity, I can't specify what this would, other than to say that we (the markers) will know it when we see it. Impress us.

And if you want to be sure that we don't miss your creative solutions, make sure your report tells us about them.

A copy of the marksheet, which shows the distribution of marks across the different elements of the coursework, will be available from KEATS.

6 Version list

- Version 1.0, November 19th 2018