Coursework

November 4, 2021

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-cw.zip` from KEATS. This contains a familiar set of files that implement Pacman, and version 6 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 6 of `api.py`, 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.

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. For each move, you will need to have the model of Pacman's world, which consists of all the elements of a Markov Decision Process, namely:

- A finite set of states S;
- A finite set of actions A;
- A state-transition function $P(s'|s,a)$;
- A reward function R;
- A discount factor $\gamma \in [0,1]$;

Following this you can then compute the action to take, either via Value Iteration, Policy Iteration or Modified Policy Iteration. It is expected that you will correctly implement such a solver and optimize the choice of the parameters. 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) Win hard in `smallGrid`

(b) Win hard in `mediumClassic`

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

3.1.1 Getting Excellence points

There is a difference between winning a lot and winning *well*. This is why completing aim (a) and (b) from previous section allows you to collect up to 80 points in the Coursework. The remaining 20 points are obtained by having a high Excellence Score Difference in the `mediumClassic` layout, a metric that directly comes from having a high average winning score. This can be done through different strategies, for example through chasing eatable ghosts.

A couple of things to be noted. Let W be the set of games won, i.e., $|W| \in [0, 25]$. For any won game $i \in W$ define $s_w(i)$ to be the score obtained in game/run i.

- ΔS_e in the marksheet is the *Excellence Score Difference*. You can use the following formula to calculate it when you test your code and compare the result against the values in Table 3

  $$\Delta S_e = \sum_{i \in W} (s_w(i) - 1500) \tag{1}$$

  Losses count as 0 score and are not considered. If $\Delta S_e < 0$, we set it to 0 (you cannot have a negative excellence score difference).

- Because `smallGrid` does not have room for score improvement, we will only look at the `mediumClassic` layout
- You can still get excellence points if your code performs poorly in the number of wins; marking points are assigned independently in the two sections
- Note however that marking points are assigned such that it is not convenient for you to directly aim for a higher average winning score without securing previous sections's aims (a) and (b) first
- We will use the same runs in `mediumClassic` to derive the marks for Table 2 and Table 3.

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 25 -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 25 -p MDPAgent -l mediumClassic
```

The `-n 25` runs 25 games in a row.

(c) The time limit for evlauation is 25 minute for mediumClassic and 5 minutes for small grid. It will run on a high performance computer with 26 cores and 192 Gb of RAM. The time constraints are chosen after repeated practical experience and reflect a fair time bound.

(d) 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.

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

(f) 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).

(g) 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.

(h) You should comment your code and have a consistent style all over the file.

3.3 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-cw` 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 (in particular, NumPy is not allowed). 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 6 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-cw.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. Therefore, you are not allowed to modify the `api.py` file.

(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.

(f) The only MDP solvers we will allow are the ones presented in the lecture, i.e., Value iteration, Policy iteration and Modified policy iteration. In particular, Q-Learning is unacceptable.

(g) Your code must **only** use the results of the MDP solver to decide what to do. If you submit code which makes decisions about what to do that uses other information in addition to what the MDP-solver generates (like ad-hoc ghost avoiding code, for example), you will lose marks.

This is to ensure that your MDP-solver is the thing that can win enough games to pass the functionality test.

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 Python `.py` file (your code).

The ZIP file must be named:

`cw_<lastname>_<firstname>.zip`

so my ZIP file would be named `cw_mallmann-trenn_frederik.zip`.

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-cw` 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-cw` 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

See `cw-marksheet.pdf` for more information about the marking.

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-cw`.

As discussed above, the number of games you win determines the number of marks you get. Since we will check it this way, you may want to reset any internal state in your agent using `final()` (see Section 3.2). For the excellence marks, we will look at the winning scores for the `mediumClassic` layout.

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 25`smallGrid` games after 5 minutes, and will terminate the run of the 25 `mediumClassic` games after 25 minutes. If your code has failed to win enough games 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 6 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. At the same time, comments are not intended to be paragraph-long, but brief sentences. Good code should explain itself for the most part.

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 4 th 2021