Task 9 Lab: PlanetWars Bots

Summary

Use this lab to become familiar with the PlanetWars game code. We will create a basic “Rando” bot and “Best-Worst” bot. In a later spike you will extend on this work to create your own bots that exploit tactical analysis details.

Step 1: Download, Commit & Run

The sample code for this lab can be found in the samples folder in the directory for this task. To run the game, open a command window and change to the directory where the PlanetWars code is located. Provided Python and the PATH are set correctly, we run the main.py. In order to run the simulation successfully, we need to pass in some command line parameters. The full list of parameters is as follows:

- -h, --help show help message and exit
- -p [PLAYERS ...], --players [PLAYERS ...]
  - Players in this run. Should be the space separated names (no extension) of files found in /bots. Ignored if the map or replay has players hardcoded.
- -m MAP, --map MAP Filename (no extension) of map to play on.
- -r REPLAY, --replay REPLAY
  - Filename (no extension) of replay to run. Does nothing without either --gui or --logscript being provided
- --gui Runs with the graphical output
- --logscript LOGSCRIPT
  - Adds a log output script. Could be used to make game ticks/actions human readable or to print statistics about the game states.

So for an initial run between two bots, we need to type the following:

- `python main.py --gui -p Blanko OneSlowMove -m map001`

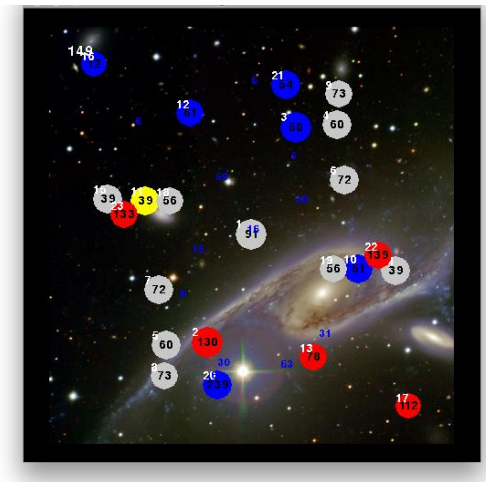

Fig 1: Game in progress showing full view

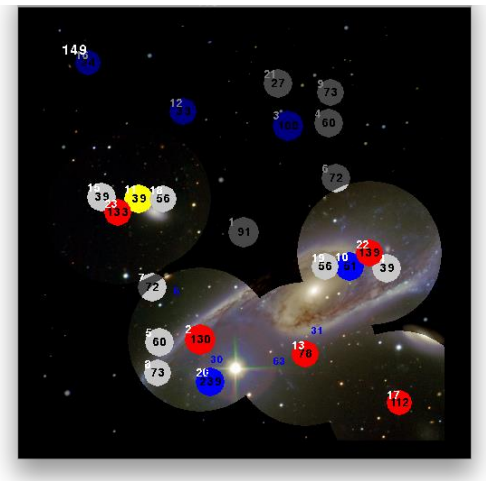

Fig 2: Red player’s façade view of the game

Figures 1 and 2 show a game in progress with different views. The game starts “paused” and can either be “stepped” by pressing the “N” key or un-paused by pressing the “P” key. Frame-rate can be changed by pressing the “-” or “+” keys. There are various viewing options: “[“ and “]” cycle through different player views, “A” gives the original all player view, and “L” cycles through different planet value types (id, num_ships, vision_age).

PlanetWars Architecture

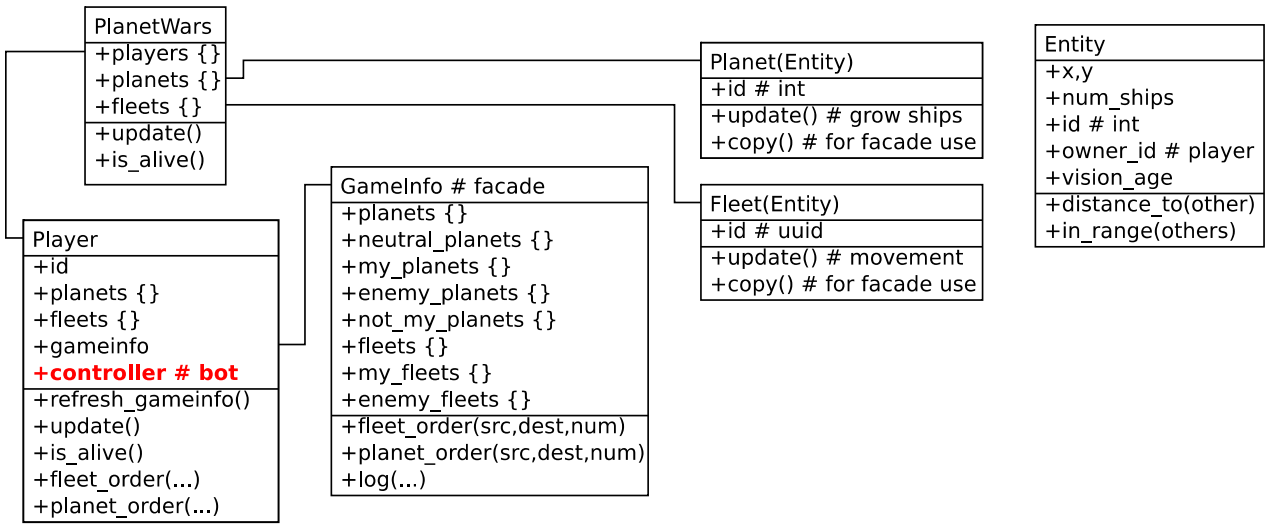

Fig 3. Simplified UML diagram of the PlanetWars architecture, showing major classes and composition. Note the bold red text indicating where bot code instance is used.

PlanetWars Files

```
main.py         # run this - the game loop
planet_wars.py  # PlanetWars class
entities.py     # Entity, Planet, Fleet classes
players.py      # GameInfo, Player
bots/           # Bot controllers
  Blanko.py
  OneMove.py
  Rando.py
  ...
maps/*.txt      # Planet map files
logs/*.log      # Battle logs
```

Controller Bot's

In Figure 3 you can see that each player has a "controller" bot instance. A bot is a simple Python class (it does not need to inherit from anything) with a single "update" method.

To create a bot you simply need to create a Python file in the bots folder, which contains a single class with the update method. The bot's class name much match **exactly** the Python file name. Take care - CaSe MatTeRs!

Each game "step" of a PlanetWars game instance happens in its update() method, which in turn calls each players update(), which then refresh their GameInfo façade object and pass it to their bot controller via the update method. The GameInfo instance also has two function which are used to issue "orders" (which the game may or may-not do depending on game rules).

```python
class MyNewBot(object):
    def update(self, gameinfo):
        # ... code to decide on src and dest ... then issue orders
        gameinfo.planet_order(src, dest, num_ships) # launch fleet from planet
        gameinfo.fleet_order(src, dest, num_ships) # launch fleet from fleet
```

The GameInfo instance is given to the bot controller as a "façade" object. It contains a copy of all the current information the player can have about the game space, limited by a "fog of war".

Step 2: Familiarisation

Run the simulation and become familiar with the operation and output. The default bots are Blanko and OneMove.

- Blanko is a "blank" bot (with lots of comments to help new bot creators) that has, as the name implies, a "blank" update method.
- OneMove bot has a very simple way of selecting a source planet, a target planet and then launching a fleet. When you feel that you understand this, move on to making your own bot.

Step 3: Bot 1 - Making A Random Bot (Rando.py)

We will be creating a new "random" based bot. Create a new Python file called Rando.py and save it in the bots directory. Paste the following skeleton code into the file to start with.

```
from random import choice

class Rando(object):

    def update(self, gameinfo):
        pass
```

Test it! Edit main.py so that one of the players now uses your "Rando" bot. (If you want to, use print to check that your bot is being called, or to print out some of the information you are being given with the GameInfo object. The random import of choice is for later.

In this new Rando bot, we will only launch one fleet at a time to a random planet. It's simple, but it works, and it's actually how many (new) human players approach similar games.

So, add the following code to check if we have any currently moving fleets.

```
        # only send one fleet at a time
        if gameinfo.my_fleets:
            return
```

Time to attack! The basic design (in words) is this:

1. Make sure we have a planet we can launch a fleet from. (Do you know why we check this?)
2. Make sure there is at least one planet to attack! (Again, make sure you know why this is.)
3. Select a random source planet from the planets we have (my_planets).
4. Select a random target planet from the planets that are not in our control (not_my_planets).
5. Launch a fleet if the source planet has more than 10 ships.
6. Only launch 75% of the ships available from the source planet.

Here's one way to do all that:

```
        # check if we should attack
        if gameinfo.my_planets and gameinfo.not_my_planets:
            # select random target and destination
            dest = choice(list(gameinfo.not_my_planets.values()))
            src = choice(list(gameinfo.my_planets.values()))
            # launch new fleet if there's enough ships
            if src.num_ships > 10:
                gameinfo.planet_order(src, dest, int(src.num_ships * 0.75) )
```

Step 4: Bot 2 - Finding the Best or Worst Target

Make a second new bot similar to Rando, but that targets either that planet with (a) the most (max) or (b) least number of ships. Name your bot whatever you want but remember that filename and class name must match.

You will want to sort through available planets (or fleets): this is the heart of any tactical analysis. You can use a for-loop and check each of the planet (or fleets) in a list and check its values for the strongest or weakest.

However, there are two nice features in Python that make this type of "search" a concise one line of code. They are the min/max functions, and lambda functions. (If you are not familiar with the ideas, it may not seem "simpler" at first, but with experience it is considered clearer. It's worth learning now!)

```
    src = max(gameinfo.my_planets.values(), key=lambda p: p.num_ships)
```

Don't get it? Ask your peers or staff - it's worth knowing! ☺

```
    # Find a target planet with the minimum number of ships.
    dest = min(gameinfo.not_my_planets.values(), key=lambda p: p.num_ships)

    # OR, (alternatively), use an inverse proportional maximum search...
    dest = max(gameinfo.not_my_planets.values(),
               key=lambda p: 1.0 / (1 + p.num_ships))
```

Git commit and push your code. Upload the code for your two bots to Canvas (rando.py and ???).

Extensions

Where to from here? Spikes! Perform tactical analysis (using what you know about how to get the "best" and "worst" planets) and then exploit it. *Battle it out! Add a defence mode … avoid long fleet travel … scout … look for weakness …*