Master's thesis

Mircea Milencianu

December 3, 2020

Abstract

This is the abstract of the paper. Last to be written.

Contents

1 Introduction **2**

2 Literature Review **3**

3 Theoretical Approach **4**
- 3.1 Overview of Game Theory concepts 4
- 3.2 Itterated Prisonners Dillema 4
- 3.3 Axelrod first tournament 4
- 3.4 Monte Carlo simulation 4
- 3.5 Monte Carlo tournament 4

4 Implementation **5**
- 4.1 Short description 5
- 4.2 Python Programming language 5
- 4.3 Axelrod library description 6
  - 4.3.1 Player 9
  - 4.3.2 Action 11
  - 4.3.3 Strategy 11
  - 4.3.4 Match 12
  - 4.3.5 Tournament 12
- 4.4 Design and implementation 12
- 4.5 Outputs 12

5 Results **13**

6 Conclusions **14**

Chapter 1

Introduction

Chapter 2

Literature Review

Chapter 3

Theoretical Approach

3.1 Overview of Game Theory concepts

3.2 Itterated Prisonners Dillema

3.3 Axelrod first tournament

3.4 Monte Carlo simulation

3.5 Monte Carlo tournament

Chapter 4

Implementation

4.1 Short description

Having a theoretical base in mind, set in the previous chapter, I will advance in explaining the program developed for supporting those ideas. Starting with foundational building block of this solution, the Python Programming Language. Python has many advantages and it has become the de facto programming language for different fields and purposes. For this solution it helped me by giving an important jumpstart using the Axelrod library, being the second building block on which this entire solution is built. In the following pages I will explain why and how Python, Axelrod library and different other third party modules have helped shaping a program that will offer a statistical and visual representations of what this type of tournament might represent, its patterns and limitations.

4.2 Python Programming language

Python is a strong, procedural, object-oriented, functional language crafted in the late 1980s by Guido Van Rossum. The language is named after Monty Python, a comedy group. The language is currently being used in diverse application domains. These include software development, web development, Desktop GUI development, education, and scientific applications. So, it spans almost all the facets of development. Its popularity is primarily owing to its simplicity and robustness. There are many third party modules for accomplishing the above tasks. Forexample Django, an immensely popular Web framework dedicated to clean and fast development, is] developed on Python. This, along with the support for HTML, E-mails, FTP, etc.,

makes it a good choice for web development. Python also finds its applications in scientific analysis. SciPy is used for Engineering and Mathematics, and IPython is used for parallel computing. SciPy provides MATLAB like features and can be used for processing multidimensional arrays. Working in statistics would find some of these libraries extremely useful and easy to use.
It's design offers some support for functional programming in the Lisp tradition. It has filter, map, and reduce functions; list comprehensions, dictionaries, sets, and generator expressions. The standard library has two modules (itertools and functools) that implement functional tools borrowed from Haskell and Standard ML. The language's core philosophy is summarized in the document The Zen of Python (PEP 20), which includes aphorisms such as:

- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
- Complex is better than complicated.
- Readability counts.

Rather than having all of its functionality built into its core, Python was designed to be highly extensible. This compact modularity has made it particularly popular as a means of adding programmable interfaces to existing applications. Following you will find a list of full featured modules for their purposes which helped in shaping this solution (alog their reference for further reading):

- Built-in modules: datetime, os, operator, random;
- Argparse: makes it easy to write user-friendly command-line interfaces;
- Pandas: provides data-structures and data analysis tools for Python;
- Matplotlib: detailed in the following section;
- Axelrod: detailed in the following section;

4.3 Axelrod library description

The Axelrod python library builds on the python programming language, thus giving researchers a comprehensive, flexible and stable tool to study the Iterated Prisoner's Dilemma game[1]. The goal of the library is to provide facilities for the design of new strategies and interactions between them, as well as conducting tournaments and simulations. Usually a strategy is studied in isolation with opponents chosen by the creator, other times strategies are revised without updating the source code. As such, most results cannot reliably be replicated thus, not getting scientific credibility. Some of the objectives of this library are:

- To enable the reproduction of Iterated Prisoner's Dilemma research as easily as possible;
- To produce the de-facto tool for any future Iterated Prisoner's Dilemma research;
- To provide as simple a means as possible for anyone to define and contribute new and original Iterated Prisoner's Dilemma strategies;

Reproductible research is another motivation for this library along the following:

- Open: all code is released under an MIT license;
- Reproducible and well-tested;
- Well-documented: all features of the library are documented for ease of use andmodification;
- Extensive: 135 strategies are included, with infinitely-many available in the caseof parametrised strategies;
- Extensible: easy to modify to include new strategies and to run new tournaments;

Unit, property and integration tests are written and automatically ran, making any new strategy added already compatible with the rest of the code with some certainity. Most trategies inherit from abstract a class and only the specifics must be developed in the actual class. These being some major reasons why this library is suits the principles for reproductible, reliable and easy research.

```python
from axelrod.action import Action
from axelrod.player import Player

C, D = Action.C, Action.D

class Alternator(Player):
    """
    A player who alternates between cooperating and defecting.
    Names:
    - Alternator: [Axelrod1984]_
    - Periodic player CD: [Mittal2009]_
    """
    name = "Alternator"
    classifier = {
        "memory_depth": 1,
        "stochastic": False,
        "long_run_time": False,
        "inspects_source": False,
        "manipulates_source": False,
        "manipulates_state": False,
    }
    def strategy(self, opponent: Player) -> Action:
        """
        Actual strategy definition that determines player's action.
        """
        if len(self.history) == 0:
            return C
        if self.history[-1] == C:
            return D
        return C
```

Listing 4.1: Alternator strategy implementation

Code in listing 4.1 represents a simple strategy taken from the library's github repository. To better understand it, we could divide the listing in 3 sections, showing us some python best practices used in the library. The first section is at the top, where other modules are imported, in this case, the Player and Action modules which will be explained later. For now, it is enough to know that we need the python implementation for a player and its actions which are defined with simple names C and D(Cooperate and Defect). The second section will be the actual class for the strategy, in our case, its the Alternator. The body of the class starts with a docstring (documents the class) which describes the strategy and has no affect on the logic or the execution of the code. Under the name variable is a dictionary with common characteristic for different strategies. These characteristics are specific to game the-

ory concepts and do not make the subject of this chapter. The logic is in the strategy function which is part of the class. For the alternator it is very simple, it takes into account the start and its own history, alternating between actions.
Concepts briefly mentioned above must be exemplified and explained within the context of the library because they represent the starting point for this implementation. I presented above how the main concept, a strategy, is created within the Axelrod library using the Pyhton language. There are many concepts from Game Theory implemented throughout this library which may be of intereset with different purposes but, further i will start highlighting the ones of intereset for this context. Will start with figure 4.3 which gives a good overview of what we want.

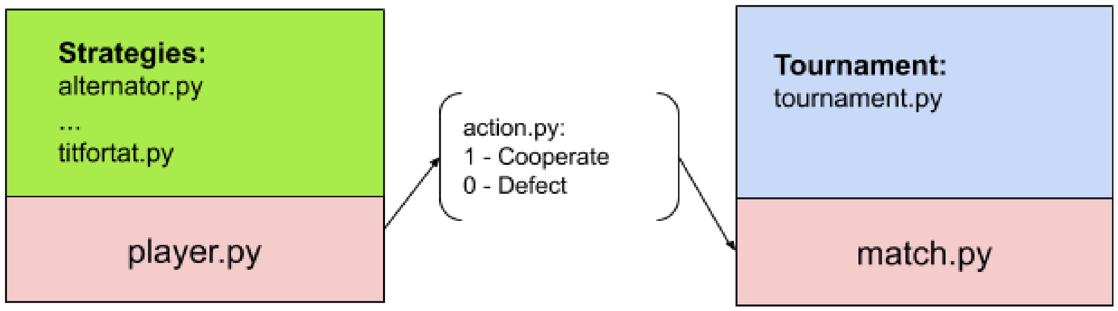

Figure 4.1: Components of interest in the library structure

Simply, this can be explained in the following phrase: A **Player** employs an **Action** according to a **Strategy** within a **Match** played in the context of a **Tournament**. The corresponding file(having the extension .py) is mentioned for those who are interested in viewing the source code, for now lets take each notion and detail a part of its code.

4.3.1 Player

Every game has a player or many players, they are the starting point for any game. Within the Axelrod library the player.py class is called an abstract base class, not intended for direct use. Abstracting away the player is good reasoning. We want players to be the same with regards to the implementation. The only thing that should differ is how they employ the strategies that are assigned to them. Thus, most of the functions are used to set different parameters, verify or delete the state of a player and many more complex requirements. Also property decorator is used for operations like determining a player's history. Check the following snnipet of code which shows an initialization function for parameters comming from a strategy and an example for property decorators.

```python
@classmethod
def init_params(cls, *args, **kwargs):
    """
    Return a dictionary containing the init parameters of a strategy
    (without 'self').
    Use *args and **kwargs as value if specified
    and complete the rest with the default values.
    """
    sig = inspect.signature(cls.__init__)
    self_param = sig.parameters.get("self")
    new_params = list(sig.parameters.values())
    new_params.remove(self_param)
    sig = sig.replace(parameters=new_params)
    boundargs = sig.bind_partial(*args, **kwargs)
    boundargs.apply_defaults()
    return boundargs.arguments

def __init__(self):
    """Initial class setup."""
    self._history = History()
    self.classifier = copy.deepcopy(self.classifier)
    self.set_match_attributes()

@property
    def cooperations(self):
        return self._history.cooperations
```

Listing 4.2: Parameters init and property decorator

Going further into details is not necessary here because we could lose ourselves in different python specifics which are not relevant.

4.3.2 Action

```python
@total_ordering
class Action(Enum):

    C = 0  # Cooperate
    D = 1  # Defect

    @classmethod
    def from_char(cls, character):
        if character == "C":
            return cls.C
        if character == "D":
            return cls.D
        raise UnknownActionError('Character must be "C" or "D".')

def str_to_actions(actions: str) -> Tuple[Action, ...]:
    return tuple(Action.from_char(element) for element in actions)

def actions_to_str(actions: Iterable[Action]) -> str:
    return "".join(map(str, actions))
```

Listing 4.3: Trimmed action implementation

The code in listing 4.3 is trimmed for removing non-essential functions like flip. What remains are the representation of Cooperation and Defection within programming language. Important to mention, do not confuse these numbers with the concept of utility. The method from_char gets the character and converts it into an action object through the cls variable which takes the values 0 and 1. The rest of the methods are simple converters from string to action and vice versa for ease of use.

4.3.3 Strategy

Here, there is already the example of the alternator in listing 4.1. For more and detailed information please refer to titfortat.py and check the TitforTat strategy which has been explicitly made in a verbose way to serve as a model for any subsequent strategies. An important mention would be the classifier dictionary that is part of the parameters set using the class player and its methods.

4.3.4 Match

4.3.5 Tournament

4.4 Design and implementation

4.5 Outputs

Chapter 5

Results

Chapter 6

Conclusions

Bibliography

[1] Axelrod project developers. *Axelrod: 4.10.0*. Apr. 2016. DOI: `doiinformation`. URL: `http://dx.doi.org/10.5281/zenodo.3980654`.