BABEŞ-BOLYAI UNIVERSITY

Faculty of Economics and Business Administration

Business Modeling and Distributed Calculus

Dissertation

Studying strategic interactions in Axelrod's tournaments in the context of infinite horizon proxy setups

Graduate,

Mircea **Milencianu** ,

Supervisor,

Assoc. Prof. Cristian **Litan**, PhD

Abstract

Interactions between agents, either being humans or machines serving a purpose, frequently have unknown duration. Game theory represents it through the concept of infinite horizon for a repeated game. With the help of Axelrod's research and tournaments, since 1980, hundreds of strategies have been put against each other in the well-known Prisoner's Dilemma context. Simulations for a tournament with a finite horizon have yielded surprising results. In this thesis, I propose a new perspective for looking at these tournaments. Building on the Axelrod python library, I developed an extension to the original tournament which provides a new structure for visualizing the data. Tweaking parameters to simulate an approximate infinite horizon setup while looking at individual interactions between strategies provides an new perspective on the research. Almost all strategies act the same between each other, indifferent of a finite or infinite horizon, suggesting further research should be conducted in developing new types of strategies or sets of strategies if studied in the right context.

List of Figures

Contents

Chapter 1

Introduction

This thesis presents an extension to the Axelrod tournament along a new visualizing methond that puts the concept of infinite horizon at the center. Axelrod has sparked the interest of the scientific community with its proposal of holding a tournament for evolutionary purposes, built on the Repeated Prisoner's Dilemma game. His idea was to create a contest in which players represented by strategies would compete against each for payoffs. The winner would be decided by the highest average utility, calculated across all games within a tournament. The tournament type is round robin with a match consisting of 200 turns. Strategies were submitted by scientists across the world and across different fields. After repeating it 5 times to cancel out anomalies, the winner was TitForTat. Considered the simplest one, it also displayed characteristics which were defined with the help of this tournament. Much like real life, it has been demonstrated that niceness or forgiveness can be "qualities" that yield benefits for the beholder. These outcomes raised the interest for another tournament. Thus, later in September 1980 a second tournament was held with 64 strategies. Again, TitForTat was submitted and yet again, it won.

Years have passed since the two tournaments and still there is constant interest expressed in this direction. Within the Python programming language the entire tournament was reproduced within its library, the Axelrod library. Because of the capabilities that modern programming languages the library offers a multitude of analysis points and parameters configuration. Also, it has reached over 200+ strategies of different types. It has grown into a very complex and intriguing ecosystem.

Axelrod's library has evolved in many aspects but one has remained on the second place. The Axelrod library offers many configuration parameters that can alter a tournament but, besides a probability to end a turn there aren't many options related to infinite horizon. This thesis proposes an extension to the Axelrod library

that can simulate an infinite horizon. It uses well-known probabilistic functions to simulate a random or "unknown" number of turns which in turn yields an infinite horizon. Thus, I arrive at the motivation for such a thesis, an insightful way to study the concept of infinite horizon effect on strategies in a controlled setting.

This thesis is structured in 4 chapters starting with an overview of game theory concepts in chapter 1 where important theorems and definitions are given to guide the reader, along references for those interested in a deeper understanding. In chapter 2 the implementation of the Axelrod library is presented in sections 3.1, 3.2 and 3.3. In sections 3.4 and 3.5 the infinite extension created to support the idea of the thesis is detailed. Chapter 3 gives a simple view over the results and explains outcomes that are generated by this solution. This paper ends with conclusions about the current research and provides further research tracks that can be followed.

Chapter 2

Theoretical Approach

2.1 Overview of game theory concepts

Game theory can be viewed as the study of strategic interactions, cooperative or non-cooperative, between intelligent rational agents or decision-makers. It can be used in social science, as well as practical decision-making. The name may imply a fun or recreational approach to science but in fact its based on solid mathematical models. Some of these mathematical models give birth to simple hypothetical examples, or viceversa, which further help in understanding basic human interaction from a rational perspective. A renowned example is the Iterated Prisoners Dilemma which is at the core of the current thesis. But, in order to tackle this problem and its relevance to human behaviour, terms, concepts and theorems must be presented. All of the of the following explanations are given using [2] as source of truth. The reader should always assume [2] as a "global" reference for chapter 2, except explicitly told not to.

In game theory a **game** refers to any social situation involving two or more individuals, that may be called **players**. Above, a statement containing the adjectives rational and intelligent has been attributed to such individuals or players. Thus, a technical explanation is required. A player is **rational** if his decisions are consistently in pursuit of his own objectives, which represent a value of his own payoff. Payoffs are measured using utilities and players follow the *expected-utility maximization theorem* which will be explained in the next subsection. Analyzing a game, we say a player is **intelligent** if he knows everything we know about a game and he can make any inferences about the situation the we can make. These assumptions may not hold in real-life situations but are crucial for creating enclosed known environments from which we can draw conclusions.

2.1.1 Expected-utility maximization theorem

This theorem has been widely revisited in literature by different authors and its not in the scope of this theses to show its proof. It is based on decision theory which shows that any decision-maker who satisfies certain intuitive axioms should always behave in order to maximize the expected value of some utility function with respect to a probability distribution. For those interested in viewing the details of its proof, you are invited to look at [2] chapter 1, section 1.4 which contains an extensive demonstration. It is sufficient for this thesis to name the axioms which are used as basic properties that a rational decision-maker may be expected to satisfy:

- Completeness
- Transitivity
- Relevance
- Monotonicity
- Continuity
- Objective substitution and strict objective substitution
- Subjective substitution and strict subjective substitution
- Interest

All of the above are jointly satisfied if and only if there exists a utility function $u : X \times \Omega \to R$ and a conditional-probability function $p : \Xi \to \Delta(\Omega)$ such that it gives a set of three equations which mean:

1. Normalization: values of utility will range between 0 and 1;
2. A version of Bayes's formula which states that conditional probabilities addressed in one event must be related with conditionals probabilities addressed in other;
3. A decision-maker will always prefer a lottery/action with a higher expected utility;

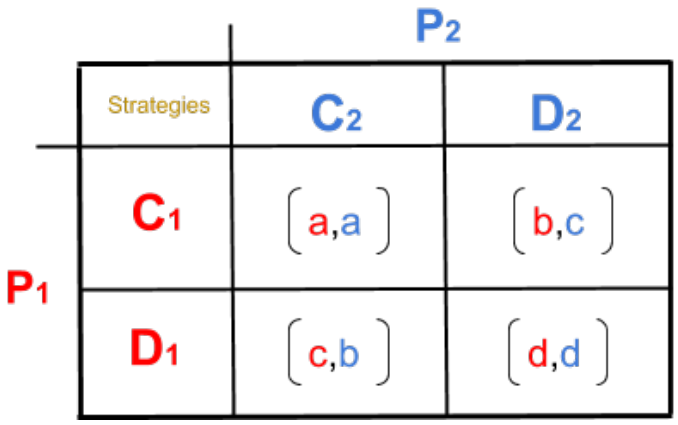
As a simple summary, any rational decision-maker behaviour can be described by a utility function and a subjective probability distribution(characterizing his beliefs about unknown factors); when new information is available his probabilities should be adjusted using Bayes's formula.

2.1.2 Game representation and models

In order to analyse games we need methods to describe them, called models. Those models need to yield enough information in order to facilitate a good study of a game. Throughout literature two methods of representing games have emerged which are the extensive form and the strategic(normal) form. The extensive form is the most complex in describing games while the strategic and its generalization, also called Bayesian form, are more simple for general analysis. An extensive form reveals more interesting concepts to its observer. One of those its the path of play which consists of the sequence of events when the game is played. The goal of game-theoretic analysis is to try and predict this path of play. Other interesting concepts that an extensive form presents are chance nodes or decision nodes for identifying influential points in a game. In this representative form labels are used to identify information states or movements which equate with decisions by players. Extensive form is not the desired representation because it can become hard to follow for many games, especially in the case of this thesis.

Strategic or normal form games will be the choice of modelling a game throughout this paper. Start by looking at the following figure 2.1, which presents the set of players in a game, the set of options available to each player and how payoffs depend on options that they choose:

Figure 2.1: Strategic representation of a game



The strategic-form representation is a static model because it excludes problems of timing and treats players as if they choose strategies simultaneously.

A comprehensive mathematical representation of strategic-form games is given in [2] chapter 2, section 2.2 from which I will extract only the essentials, details will

be left for those interested. A strategic-form game is any Γ of the form

$$\Gamma = (N, (C_i)_{i \in N}), (u_i)_{i \in N}) \tag{2.1}$$

where N is a nonempy set of players in the game Γ. For each player i, C_i is the set of strategies(or pure strategies) available to player i. When the game Γ is played, each player i must choose one of the strategies in the set C_i. A *strategy profile* is a combination for strategies that the players in N might choose. Let C denote the set of all possible strategy profiles so that

$$C = \bigtimes_{j \in N} C_j \tag{2.2}$$

For any strategy profile $c = (c_j)_{j \in N}$, the number $u_i(c)$ represents the expected utility payoff that player i would get in this game if c were the combination of strategies implemented by the players. According to [**1**] and the above mentioned properties a form for the utility function for an extensive-form game Γ^e is:

$$u_i(c) = \sum_{x \in \Omega^*} P(x|c) w_i(x). \tag{2.3}$$

where, simply put:

- Ω^* denotes the set of all terminal nodes in Γ^e;
- $P(x|c)$ is the probability that the path of play will go through node x and the path is determined by the player's relevant strategy in c;
- $w_i(x)$ is the utility payoff to player i at node x in Γ^e

When $\Gamma = (N, (C_i)_{i \in N}), (u_i)_{i \in N})$ is derived from Γ^e, Γ is called the normal representation of Γ^e.

2.2 Prisonners' Dilemma

Recalling figure 2.1 where all utilities are letters, represents only a blueprint of a normal-form game. In order to give meaning to it, some payoffs need to be inserted. In turn these payoffs will create the context for a simple but fascinating game.

Figure 2.2: Prisoner's Dilemma

		P_2	
	Strategies	C_2	D_2
P_1	C_1	(-1,-1)	(-3,0)
	D_1	(0,-3)	(-2,-2)

If this game is played once or N times is called a finite game. For this case we will see why two completely rational decision-makers might not cooperate in this setting, even if it appears that it is in their best interest from the outside. According to [3] the story around this game may be told as follows:

> Two members of a criminal gang are arrested and imprisoned. Each prisoner is in solitary confinement with no means of communicating with the other. The prosecutors lack sufficient evidence to convict the pair on the principal charge, but they have enough to convict both on a lesser charge. Simultaneously, the prosecutors offer each prisoner a bargain. Each prisoner is given the opportunity either to betray the other by testifying that the other committed the crime, or to cooperate with the other by remaining silent. If the payoffs are all nonpositive, their absolute values can be interpreted as the length of jail term each of prisoner will get in each scenario.

By putting ourselves in a prisoner's position we can deduct threes outcomes:

- If both betray(defect) the other, both play D; they both end up in prison for 2 years;
- If one betrays while the other remains silent, one plays C and the other D, the one who betrayed gets out of prison while gets 3 years in prison;
- If both stay silent(cooperate with each other), both play C, they both get 1 year in prison;

At this point two solution concepts need to be introduced in order to understand the thinking for a decision-maker:

1. **Domination** which builds upon the adjectives mentioned in 2.1, rational and intelligent. If those are satisfied both players know that the other understands the game at least as well as we do and they both play for maximizing their utility. In [3] section 3.3 a definition for dominant strategy is given in 3.3.2 which states:
A strategy is strictly (resp., weakly; very weakly) dominant for an agent if it strictly (weakly; very weakly) dominates any other strategy for that agent.

2. A strategy profile is a **Nash equilibrium** if for all decision-makers in a game there exists only one best response. More details can be found in [2] and [3].

Defection always results in a better payoff than cooperation, it is a dominant strategy. Mutual defection is the only strong Nash equilibrium in the game. Thus creating the dilemma that mutual cooperation yields a better outcome than mutual defection but is not the rational outcome because the choice to cooperate, from a self-interested perspective, is irrational. Using all the rationales enumerated throughout this section it can be said that any game that follows $c > a > d > b$ as a rule for payoffs, is a Prisoner's Dilemma.

Now, we have the basic theoretical aspects to understand this fascinating game it is time to move forward to see it in use, in the context of Axelrod tournaments.

2.3 Repeated Prisoner's Dilemma

2.3.1 Finite Prisoner's Dilemma

According to [3], chapter 6, section 6.1 a good way to exemplify finite horizon games, is through a twice-played Prisoner's dilemma. Two assumptions need to be made in this case:

1. Players choose their actions simultaneously;
2. Their utility function is additive, meaning the sum payoffs;

In this kind of game, an action played in the current round or turn can depend on the history of the game. By knowing "how long" the game is going to last, the phenomenon of backwards induction can be used (see [3], chapter 6, section 6.1 for more). By considering a finitely repeated Prisoner's Dilemma game along with backwards induction, it can be argued that in the last round (the second one for our situation) has a dominant strategy in "D"(defection). This being common knowledge, it can be applied to the oponnent as well. Thus, in the first round, the dominant strategy is also to defect, resulting in an equilibrium established on the act of defecting.

2.3.2 Infinite Prisoner's Dilemma

Until this point finite repeated games were discussed, where the end of the game known by all participants and is a finite number. But, what happens when the game is extended for an infinite amount of interactions? Also, how agents alter their behaviour when their interactions go for an unknown amount of time? This end-game concept is called horizon, until here only finite horizon was discussed. In [2] a repeated game is defined as a game that has an infinite horzion. Because no move is necessarily the last, a player must always consider the effect that his current move might have on the moves and information of the other players in the future. These conditions lead players to be more cooperative, or more belligerent, as [2] says. In [2] section 7, this type of games is comprehensively covered using extensive mathematical proofs and use cases. A simplistic view of this category of games is given in [3] which states that an infinitely repeated game is transformed into extensive form, the result is an infinite tree. Thus payoffs cannot be attached to any terminal nodes, nor can they be defined as. the sum of the payoffs in the stage games. In [3] section 6.2 two concepts are given for payoff in such a context:

Definition 6.2.1 - Average reward. Given an infinite sequence of payoffs r_i^1

, r_i^2 ,... for player i, the average reward of i is:

$$\lim_{k \to -\infty} \frac{\sum_{j=1}^{k} r_i^{(j)}}{k} \tag{2.4}$$

Definition 6.2.2 - Discounted reward. Given an infinite sequence of payoffs r_i^1 , r_i^2 ,... for player i, and a discount factor β with $0 \leq \beta \leq 1$

$$\sum_{j=1}^{\infty} \beta^j r_i^{(j)} k \tag{2.5}$$

If we consider strategy spaces in an infinitely repeated game, specifically Prisoner's Dilemma, a natural question is whether we can characterize all the Nash equilibria of the repeated game. As [3] states, if the discount factor is large enough, both players playing TfT(TitForTat) is a Nash equilibrium. But there is an infinite number of others. To understand this, *folk theorem* is used which does not characterize the equilibrium strategy profiles, but rather payoffs obtained in them. Roughly speaking, it states that in an infinitely repeated game the set of average rewards attainable in equilibrium are precisely those pairs attainable under mixed strategies in a single-stage game, with the constraint on the mixed strategies that each player's payoff is at least the amount he would receive if the other players adopted minmax strategies against him, according to [3]

2.4 Axelrod tournaments

Imagine you played this game for a couple of times and using rationales presented nothing interesting happened. Is that all there is to it? Apparently not, in 1981 Dr. Robert Axelrod and William D. Hamilton published a paper called "The Evolution of Cooperation", found at [1] where a finite version of Prisoner's Dilemma was used in an evolutionary problem. In the introduction he argues that cooperation has been disregarded from an evolutionary theory perspective although, in nature, we know of examples in which organisms are cooperating instinctively, such as, the fungus and the alga that create a lichen or, the fig wasps and the fig tree etc. Creating a formal theory for cooperation was his motivation and for this he used Prisoner's Dilemma which the paper states "is an elegant embodiment of the problem of achieving mutual cooperation, and therefore provides the basis for our analysis." The mutual defection solution is also present in biological evolution being the outcome of evolutionary trends for a set of conditions. While more reasons for creating a setting for studying cooperation are given in [1] lets focus on the rules of a tournament:

1. The type of the tournament is round robin;
2. Every strategy plays any other strategy including a copy of itself;
3. A match lasts for 200 iterations of Prisoner's Dilemma;
4. The payoffs are set as in figure 2.2;
5. The winner is the strategy with the best average utility per match;
6. The RANDOM strategy was also an entry;
7. The tournament was repeated 5 times to get a more stable estimate of the scores for each pair of players;

Once the rules are setup, strategies from people around the world and from different fields were submitted. The first tournament had 14 strategies submitted while the second 64. On both occasions Tit For Tat was declared the winner while being one of the most simplest submitted.

2.4.1 Strategies from the first tournament

In [11] we can find an analysis of the first tournament. A general conclusion is that 600 points equate to a good score by a player/strategy, which means that both sides always cooperate with each other. While, a benchmark for poor performance is 200 points. The range for scores starts with 0 ending with 1000, but most scores are capped at 600, while the winner(Tit For Tat) had an average of 504 per game. Before diving into strategy details lets see two of the properties of successful strategies:

- **Niceness** According to [11] this is the single property which distinguishes the relative high entries from the relatively low scoring entries. The property of being nice. A decision rule is nice if it will not be the first one to defect, or if at least it will not be the first to defect, or if at least it will not be the first to defect before the last few moves.

- **Forgiveness** This property refers to how "nice" strategies responded when defected. A response important to their overall success. A good definition for this property is given in [11] which states, forgiveness of a strategy is its propensity to cooperate in the moves after the other player has defected.

TitForTat

In [4] this strategy is described as follows: A player starts by cooperating and then mimics the previous action of the opponent. This strategy came first.

Tideman and Chieruzzi

From [4] this strategy is described as follows:

1. Every run of defections played by the opponent increases the number of defections that this strategy retaliates with by 1.

2. The opponent is given a 'fresh start' if:

    - it is 10 points behind this strategy
    - and it has not just started a run of defections
    - and it has been at least 20 rounds since the last 'fresh start'
    - and there are more than 10 rounds remaining in the match

- and the total number of defections differs from a 50-50 random sample by at least 3.0 standard deviations. A 'fresh start' is a sequence of two cooperations followed by an assumption that the game has just started (everything is forgotten).

3. The strategy defects on the last two moves.

This strategy came 2nd in Axelrod's original tournament.

Nydegger

Another interesting strategy which uses Tit For Tat to set its approach. According to [4] it is designed as follows:
The program begins with tit for tat for the first three moves, except that if it was the only one to cooperate on the first move and the only one to defect on the second move, it defects on the third move. After the third move, its choice is determined from the 3 preceding outcomes in the following manner:

$$A = 16 * a_1 + 4 * a_2 + a_3 \tag{2.6}$$

Where a_i is dependent on the outcome of the previous i^{th} round. If both strategies defect,$a_i = 3$, if the opponent only defects:$a_i = 2$ and finally if it is only this strategy that defects then$a_i = 1$.Finally this strategy defects if and only if:

$$A \in 1, 6, 7, 17, 22, 23, 26, 29, 30, 31, 33, 38, 39, 45, 49, 54, 55, 58, 61 \tag{2.7}$$

Thus if all three preceding moves are mutual defection, A = 63 and the rule cooperates. This rule was designed for use in laboratory experiments as a stooge which had a memory and appeared to be trustworthy, potentially cooperative, but not gullible. This strategy came 3rd in Axelrod's original tournament.

Alternator

As the name says, it alternates between cooperation and defection.

2.4.2 Strategies from the second tournament

In September 1980, Axelrod followed up with its second tournament with the scope of getting a deeper understanding of how to perform well in such a Prisoner's Dilemma setting, as stated in [12]. In this setting, again, the simplest solution Tit For

Tat was declared the winner. Because most of the rules were nice a new property was introduced during this tournament, **provocability**, which in [12] is defined as follows: a strategy is provocable if it immediately defects after an "uncalled for" defection from the other. The paper gives examples of "uncalled for" strategies as Tranquilizer or Tester, these are also called representatives. For more insightful details check [12] for now let's take the second and third strategy and check their implementation.

Champion

[4] describes it as follows: the player cooperates on the first 10 moves and plays Tit for Tat for the next 15 more moves. After 25 moves, the program cooperates unless all the following are true: the other player defected on the previous move, the other player cooperated less than 60% and the random number between 0 and 1 is greater that the other player's cooperation rate.

Borufsen

Based on [4] I will do a short overview of the strategy. This player keeps track of the the opponent's responses to own behavior:

- cd_count counts: Opponent cooperates as response to player defecting
- cc_count counts: Opponent cooperates as response to player cooperating.

The player has a defect mode and a normal mode. In defect mode, the player will always defect while in normal mode, the player obeys a list of ranked rules. More on the details of the rules can be found in [4].

Chapter 3

Implementation

3.1 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.

3.2 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. All of the above can be seen in detail in [9].

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), see [9] for more, 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 (see Python Standard Library for more)
- Argparse: makes it easy to write user-friendly command-line interfaces
- Pandas: provides data-structures and data analysis tools for Python (see [8]);
- Numpy: for complex mathematical operations (see [7]);
- Matplotlib: detailed in the following section (see [6]);
- Axelrod: detailed in the following section (see [4]);

3.3 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 [4]. 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 as presented in [4]:

- 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, also see [4] for details:

- 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, according to [4].

```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
```

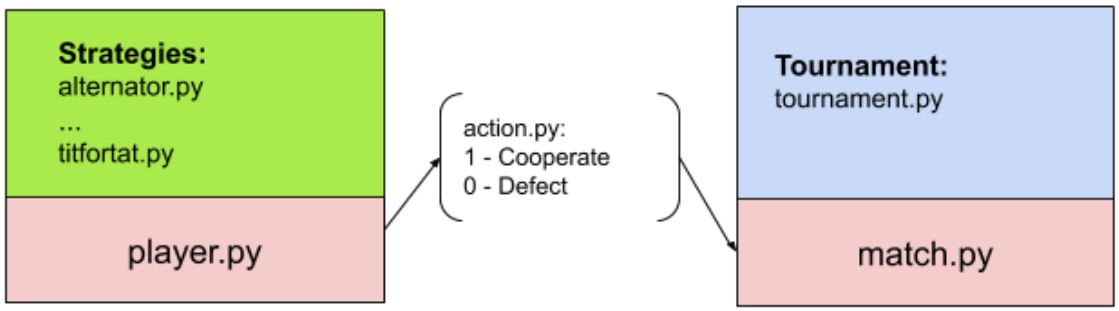
Listing 3.1: Alternator strategy implementation

Code in listing 3.1 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 3.1 which gives a good overview of what we want.

Figure 3.1: 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.

3.3.1 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.

```
@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 3.2: 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.

3.3.2 Action

```
@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 3.3: Trimmed action implementation

The code in listing 3.3 is trimming 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.

3.3.3 Strategy

Here, there is already the example of the alternator in listing 3.1. 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.

3.3.4 Match

The concept of a match is represented by a class in python. It holds within its scope different methods which help in putting strategies against each other within different contexts, for example: there are stochastic matches. Following we will touch some of main functions of the class that are used in playing a simple simultaneous match of Prisonners Dillema. A match starts with the initialization function, setting the parameters that characterize a match; some relevant parameters: players, turns, prob_end. All the above mentioned are explained within the docstrings of the code. Following lets see the method that pits to players against each other.

```
77 def simultaneous_play(self, player, coplayer, noise=0):
78     """This pits two players against each other."""
79     s1, s2 = player.strategy(coplayer), coplayer.strategy(player)
80     if noise:
81         s1 = self._random.random_flip(s1, noise)
82         s2 = self._random.random_flip(s2, noise)
83     player.update_history(s1, s2)
84     coplayer.update_history(s2, s1)
85     return s1, s2
```

Listing 3.4: Method for one simultaneous play

In listing 3.4 there is the essential code used for one turn in a match. This method is called further by the play method which returns a list with the results for a match. There are other methods that help in creating more complex or specific matches but, are not relevant for the current paper. If interested check the code and its docstrings for better understanding. For now, this is sufficient, thus we can move on to explaining a tournament.

3.3.5 Official tournament

This tournament is part of the Axelrod library and it follows the python's official standards, making it a standard for running different simulations. As previously presented classes in this library it has an initializations method and "playing" methods for carrying its purpose. There are also optimization methods which allow for running tournaments on different processes. This implementation its complex and again, its better explained in the code of the library. This are the main components of a tournament, most of those are used by my custom implementation which yields a better visualization for our purpose. Will move on to better describing this implementation and its benefits.

3.4 Custom tournament implementation

As presented in the previous subsection, there is already a tournament developed in the Axelrod library. The reason I dropped it and created a new tournament is mentioned previously in section ..insert section... To summarise, the results that the official tournament exposed did not contain my desired metric, the winner of a match. Thus, the goal of the new torunament was to create statistics and visualizations with respect winners of a match. Another important feature of the new solution was to introduce randomly generated numbers for key variables:

- Mean number of matches is represented by mean_m;
- Deviation of matches is representede by dev_m;
- Mean number of turns is represented by mean_t;
- Deviation of turns is represented by dev_t;
- Runs or repetitions of tournament represented buy runs;

These are partly supported by the official tournament but, combined with the missing metrics and no way to visualize the results in a easy manner a new tournament was developed. In the following subsections I will try to explain the new tournament that looks just for the winner of a match.

3.4.1 Implentation description

Setting the tournament

Everything starts with setting the tournament with its logical components, players or, its execution components argparse. Pythons modularity has a huge impact here because of its built-in and third party modules.

```
86 #1
87 import time, os, operator
88 import argparse
89 #2
90 import axelrod as axl
91 import numpy as np
92 import pandas as pd
93 #3
94 from players import players as strategy_type
95 from players import secondgen_str_text as strategy_text
```

Listing 3.5: Setting the tournament

Following the commented numbers in listing 3.5, 3 categories of imports can be distinguished that set up my tournament. Firstly, there are the native imports. These are modules supported inside python and do not require any installs or external dependencies. Usually these are very helpful as they take away a lot of complexity for minor tasks. Secondly, there are the third party imports which usually solve specific complex problems and are maintained by a community. The best example here is the Axelrod library which is thoroughly used in this paper. Thirdly, we have my personal imports. In this case we have the players that are imported from an external file and given names that are generally available for the rest of the program. This being really helpful in case we want to run different settings for a tournament

Playing N matches

A winner of a match between two players who employ strategies is the one who has a higher utility after a certain amount of turns. To achieve this a method that plays a match and saves the winner is needed. For achieving this purpose lets follow the code in listing 3.6. It starts by assigning players and other objects their corresponding values. The method receives a number of matches to be played and for every match executes the following with the help of a for loop:

1. Check if we have a deviation for turns;
2. If there is no deviation use the mean else pick a number from a normal probability distribution;
3. Create the match and play it;
4. Check and update the winner to a list;
5. Return the list with the results;

```
96  def custom_nrof_matches (p1, p2, matches):
97      '''
98          Plays N matches and M turns with random inputs.
99          Also gathers statistics.
100     '''
101     players = (p1, p2)
102     p1_wins, p2_wins, eq = 0, 0, 0
103     results = []
104     for m in range(matches):
105         if dev_t is None:
106             turns = mean_t
107         else:
```

```
108             turns = int(np.random.default_rng().normal(mean_t,
    dev_t, None))
109         match = axl.Match(players, turns=turns)
110         match.play()
111         if match.winner() == p1:
112             p1_wins += 1
113         if match.winner() == p2:
114             p2_wins += 1
115         elif match.winner() == False:
116             eq += 1
117     results = [p1_wins, p2_wins, eq]
118     return results
```

Listing 3.6: Method for playing N matches

At the end a list with the number of winners and equalities is returned.

Running a tournament

Observe how easy it is to play a match with the Axelrod library but, to centralize and normalize the data for good visualization more is needed. To achieve a repeatable tournament a certain logic has been put in place that starts with the "__main__" name trick in python in listing 3.7.

```
119 if __name__ == '__main__':
120     """
121     Main execution and preparation for tournaments.
122     """
123     parser = argparse.ArgumentParser(
124         description="Argumets for tournaments.")
125     parser.add_argument("--runs", "-R", ...
126     my_args = parser.parse_args()
127     runs = my_args.runs
128     main(random, runs)
```

Listing 3.7: Main execution of a tournamet

In listing 3.7 argparse is used to get arguments from the command-line into the program. Those are further passed while the rest of the variables are either hardcoded or randomly generated. Having all our variables set, check listing 3.8 where the main logic of a tournament resides. Each method has a number commented which corresponds to its explanation in the next list:

```
129 def main(random, runs):
130     if random == False:
131         print("Only random mode available.")
132     elif random == True: #1
```

```
133         print("Initializing pds for average results...")
134         init_avg_pds() #2
135         print("Running simulations...")
136         for el in range(runs):
137             if el > 0:
138                 playall_random(el) #3
139     print("Dividing elements...")
140     pd_elements_division(runs) #4
141     print("Writing to file...")
142     avg_to_file(runs) #5
```

Listing 3.8: Main logic of a tournamet

1. Check if random mode is true;
2. If random is true, initialize the data structures; achieved using the method *init_avg_pds()*
3. Run the tournament if the number is greater then 0; achieved using the method *playall_random(el)*
4. Normalize the results with respect to the number of runs; achieved using the method *pd_elements_division(runs)*
5. Writing the results to file; achieved using the method *avg_to_file(runs)*

Lets take each method starting with the second comment and try to explain it:

init_avg_pds()

With this method a new module is introduced, Pandas. It is used for creating data objects and manipulate them with ease. Within these objects, lists with the results from listing 3.6 are saved at the corresponding row and column. Here we only initialize all the elements with 0, other more complex operations are executed later.

playall_random(el)

```
143 for s1 in strategy_type:
144     fixed_s = s1
145     rc = 0
146     for s2 in strategy_type:
147        rotating_s = s2
148        results=list(map(
149                     operator.add,
150                     results,
151                     custom_nrof_mathches(s1, s2, matches)))
```

```
152         normed_res = normalisation(results, matches)
153         winners_DF.at[strategy_text[fc], strategy_text[rc]]=results
154         normed_DF.at[strategy_text[fc], strategy_text[rc]]=normed_res
155         rc+=1
156         results=[0, 0, 0]
157     fc+=1
158     results=[0, 0, 0]
```

Listing 3.9: Playing all matches

In listing 3.12 is the code block that does the heavy lifting takes place. Two for loops are used, that iterate over the same list of strategies and play all the matches. The sequence of steps of each execution starts with attributting the correct values to used variables. Next, a results list created by using the add operator on the current values of the list and a newly executed set of matches. The results are normalized and set at the correct position in the dataframes. Observe that we have 2 dataframes, one that holds the exact number of winned and equal matches and the second in which results are normalized with corresponding values between 0%-100%. Dataframes are also saved to files but this step will covered in the next section.

pd_elements_division(runs)

For this and next method details will be excluded because their use is for more complex simulations. This executes when we have multiple runs for a single set of matches and turns. For example: a tournament consisting of 1000 matches and 200 turns needs to be run 100 times. Division is applied on each element of the dataframe within this method.

avg_to_file(runs)

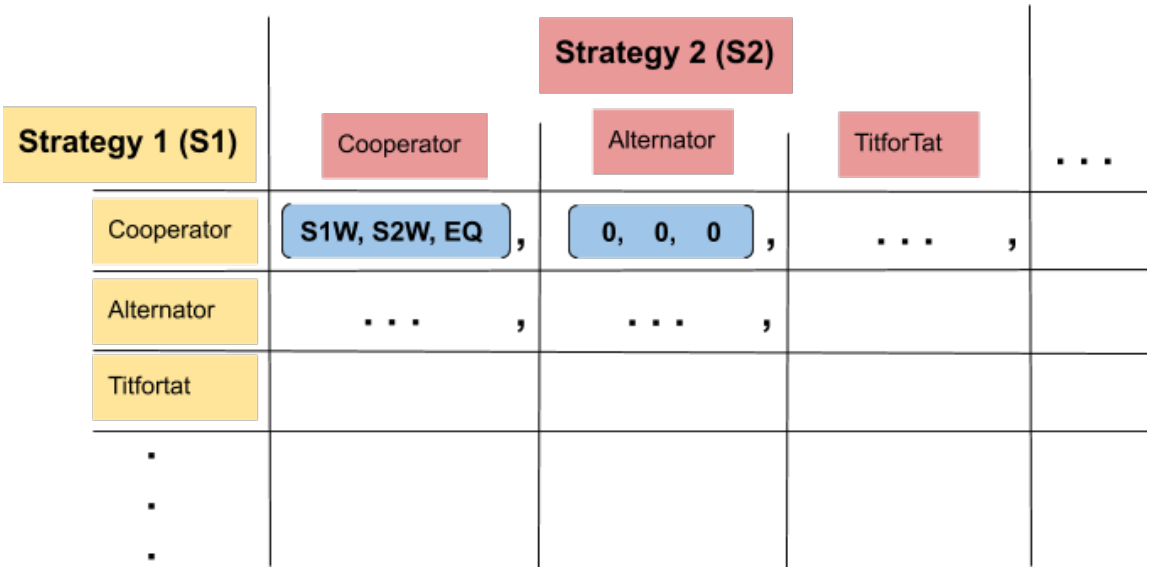
Here, I just save the dataframe objects to .csv files using built-in methods. More on how this is achieved in the next section.

3.4.2 Exporting results

After playing all the matches and saving the results at the right positions in a dataframe we need to export to a standard format which enables further processing. The chosen format is comma separated value or csv. This standard is a widespread format for saving data which is supported by a multitude of libraries in and outside of python's scope. All of the important libraries for handling data suppport this format, making it an obvious choice. A short definition for a csv file can be: a delimited text file that uses commas to separate values where each line of the file is a data record and a record consists of one or more fields, separated by commas.

In order to transform and process our data a good understanding of its structure is needed. To help achieve this, lets take a look at figure 3.2 which is the end result of a running program.

Figure 3.2: Data file structure



Both rows and columns contain the strategy names which are stored in a list of texts. The dataframe is a simple 2-dimensional matrix, NxN where N is the length of the list of strategies. On each position in the matrix there is a list containing:

- S1W: number of strategy 1 wins;
- S2W: number of strategy 2 wins;
- EQ: number of equalities between strategy 1 and strategy 2;

For example, on the first row and second column we have list which stores the wins of Cooperator on the first position, the wins of Alternator on the second and the number of equalities on the third. Having the structure set it is time to save it on disk in its corresponding file. As mentioned before the csv format is a standard supported by all libraries thus, saving it to disk is a trivial task solved by the following block code:

```
159 def avg_to_file(run):
160     """
161     Write pd objects containing the averages to csv file.
162     """
163     if not os.path.exists('results/avg_results_5000_200'):
164         os.makedirs('results/avg_results_5000_200')
165     avg_results_path = "results/avg_results_5000_200"
166 
167     avrg_win_df.to_csv("{}/winners_avg_{}.csv"
168     .format(avg_results_path, run))
169     avrg_norm_df.to_csv("{}/normed_avg_{}.csv"
170     .format(avg_results_path, run))
```

Listing 3.10: Save dataframe to file

Here, a check is performed to see if the file exists, if not the files are created empty. Then, using Pandas method *to_csv* the files are saved to disk.

3.5 Visualizing results

After saving the results they need to be viewed and interpreted. The csv format is not an easy format to read. For this case, it becomes harder as the number of strategies grows. Again, a solution that transforms data from csv to a human readble form must be created. To achieve this I relyed again on python modules with a focus on matplotlib. This module is a python adaptation of the plotting library used in Matlab, more can be found in its documentation ...insert doc ref here... Just like the tournament solution, all starts with the correct imports.

```
171 import os, re
172 import pandas as pd
173 
174 import matplotlib.pyplot as plt
175 import numpy as np
176 
177 from players import secondgen_str_text as strategy_text
```

Listing 3.11: Save dataframe to file

Again pandas is used for manipulating the data and we also need the same players list in order to match the tournament. Once we have our tools we need to instantiate and read the data produced by the tournament.

```
178 avrg_win_df = pd.DataFrame(index=strategy_text,
        columns=strategy_text)
179 avrg_norm_df = pd.DataFrame(index=strategy_text,
        columns=strategy_text)
180 avrg_win_df = pd.read_csv("path/to/file", index_col=0,
        converters={'column_name': eval})
181 avrg_norm_df = pd.read_csv("path/to/file", index_col=0,
        converters={'column_name': eval})
```

Listing 3.12: Save dataframe to file

With *read_csv* method in the above listing, the files are read and inserted into the new variables. For a better understanding and consistency between names is desired. Next, the data is transformed into dictionaries for each dataframe. Dictionaries are much easier to work with in python.

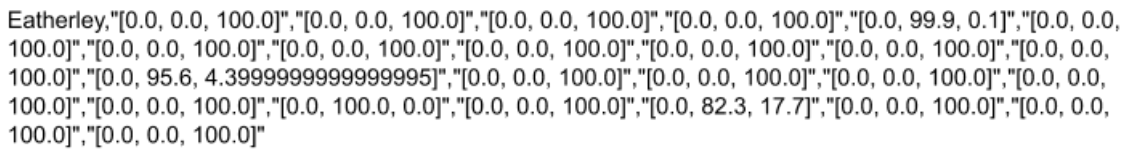


Figure 3.3: Dataframe row

At this point its not really necessary to understand the code but what it does. In figure 3.3 is an example of a row from a dataframe. That is not a good way to visualize data when there are another 23 rows that follow the same pattern. By transforming it to a dictionary the latter in figure 3.4 is obtained:

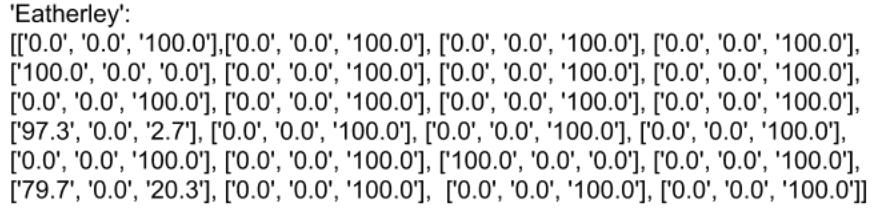


Figure 3.4: Dictionary for dataframe row

At this point we have a dictionary that has a strategy name as key. The value corresponding the key is a list of lists that contains all the results for the respective

strategy. This can be easily transformed into pie charts by the following code, in listing 3.13:

```
182 def pie_plot(data, strategy):
183     #1
184     path = "dev_run_{}/{}.jpeg".format(dev_t, strategy)
185     labels = 'P2W', 'P1W', 'EQs'
186     explode = (0, 0.2, 0)
187     fig = plt.figure(figsize=(20,10))
188 
189     plot_index = 1
190     list_index = 0
191     #2
192     while list_index < len(strategy_text):
193         #3
194        ax1 = plt.subplot(6, 4, plot_index)
195         plt.xlabel("P2:{}".format(strategy_text[list_index]))
196        #4
197         ax1.pie(data[list_index], explode=explode,
198                 autopct='%1.1f%%', shadow=True, startangle=90)
199         ax1.axis('equal')
200 
201         plot_index+=1
202         list_index+=1
203     #5
204     fig.text(0.06, 0.5, 'P1:{}'.format(strategy),fontsize=16,
205              ha='center', va='center', rotation='vertical')
206     plt.legend(labels=labels)
207     plt.savefig(path, format='jpeg')
208     plt.close('all')
```

Listing 3.13: Creating pie plots

Like before follow the commented numbers in listing 3.13 along the list below:

1. Preparing the variables for the method. The path, label and the plot are created and given proper values.
2. Looping over all the elements of the strategy list starting with 0.
3. Create a subplot with the corresponding parameters and set a label.
4. Create the pie plot with a list and some aspect related parameters.
5. Create a figure with all the pie charts, create a legend, save the figure and close all open figures.

Chapter 4

Analyzing Results

This chapter contains a dry overview on the results from the appendicies based on concepts, theorems and definitions presented in chapter 2 and data exported with the solution from chapter 3. For the finite games case, TfT has been declared a winner and put a "monopoly" on the second tournament having won the first one. While in the infinite case, we only know that there are an infinite number of equilibriums and there is the folk theorem which is about payoffs and not about strategies. By tweaking the conditions of the first and second tournament to partially simulate an infinite horizon within the original tournament it can be shown that TfT and other strategies based on it, can yield different results when interacting with simple deterministic strategies, like Alternator. Results exported using the solution presented in chapter 3 can be found in the appendices and they will be referenced when relevant. In section 2.4 the two tournaments were defined, while in chapter 3 a second approach and a new way of visualizing results is given. In both tournaments the individual interactions between strategies is emphasized, this being the main reason why an alternative tournament was developed on the basis of the first. Also, parameters were modified to create a model that relatively yields an infinite horizon game and consequently, a tournament. Lets discuss those parameters:

Repetitions - gives the number of times a tournament is repeated. It is equal to 1000 in all simulations because it gives consistency to our results. In the official tournaments, only 5 is used. It could be argued that results will flatten when using such a high number but, as we will see it, this is not the case.

Mean turns - gives the number of turns in a match. A normal distribution is used to create this value but for finite tournaments only the mean is used. **Deviation for turns** - in a normal distribution the deviation creates an interval for a random number to be extracted from.

4.1 First tournament results

Figure 1 from appendicies shows the results of a normal finite tournament repeated 1000 times with a match of 200 turns. No random distribution is used thus, the mean is used for the number of turns. In figure 2 from appendicies a normal distribution is used with a deviation of 40 and the same number of repeated tournaments, 1000. Here, an interesting result appears. The performance of the alternator is heavily changed in the infinite case vs. the finite case while interacting to Nydegger and TfT. While, in the finite case alternator wins all the matches with the two strategies, for the infinite case the other two start making some matches equal, close 50% of the 1000 matches. Important to point out that Nydegger uses TfT to decide its behaviour (see section 2.4.1).

4.2 Second tournament results

In this context the same parameters as previously described were used, but the strategy set was changed to match the second Axelrod tournament. In appendicies, figure 3 and figure 4 show the results of the two tournaments. By studying the 2 figures it can easily be observed that no difference between the individual "behaviour" of Champion in the situations. Because of the high number of strategies, I chose to show only "Champion" which came in second the official tournament. Tables 3 and 4 (from appendicies) display almost the same results for the finite and infinite case. The winner should be disregarded in this situation because the current tournament has only 23 of the 64 strategies thus, the average utility score is not relevant with respect to the official one. This being true for all the other strategies selected, see my personal github for full results, at [10].

4.3 Second tournament results with alternator

A variant of the second tournament with "Alternator" yielded most of the same results except the interaction with "Tester". Can be observed in figures 5 and 6 where "Tester" (from appendicies) has an increase win rate for the infinite case. For the rest of the interactions, the two cases remain mirrored, check [10] for full results.

Chapter 5

Conclusions

This thesis tries to analyze the Reapeted Prisoner's Dilemma by introducing an infinite horizon into the Axelrod tournament. A new metric for analysis is introduced, which is the individual interaction between each strategy participating in the tournament. This is achieved using Python's visualization, mathematical and Axelrod library. For the official tournament an average utility metric was considered conclusive. Here, I try to shift the focus on how the infinite horizon changes the "behaviour" of strategies in the same setting. From results in the appendicies along the interpretation in chapter 4 two main ideas arise.

1. Influence of a deterministic, simple strategy, as "Alternator," on some TitforTat based strategies with the infinite horizon;

2. Introducing an infinite horizon does not affect the outcomes of the individual interactions between stretegies;

The first idea can be interpreted simply by evaluating how the affected strategies start their interaction. TitforTat, Nydegger and Tester start by using TfT in the first moves. While TfT keeps using the same simple pattern, the other two adopt a different pattern based on the outcome of playing TfT. Thus, concluding that in a limited number of situations the importance of start can influence the rest of game when playing for an uknown/infinite amount of "time". For the second idea the discussion is broader. I can start by emphasizing how most of game theory books and papers present the infinite horizon in reality. Claiming that most agents change their behaviour in case they know they play for an infinite amount of "time" with another agent. Although this is true, we have no tournaments or settings which try to emulate an infinite horizon. Almost all of them use fixed parameters. This is understandable and mandatory in research because reproducibility and consistency for

data is required. But, it does not mean that some parameters cannot be tweaked to simulate partially an infinite case while, at the same having constants for reproducibility. This thesis has done exactly that by changing the turns at random but controlled by a well defined probability distribution. The results are interesting and hint at the fact that none of the strategies submitted is designed for the infinite case. Of course, none of those strategies was designed with such a thought but most of them are taken as models and used throughout different fields. At the same time our real life interactions are not determined by some specific parameters thus, asking the question if those encapsulated strategies are the solution for real interactions?

To start answering this question more research on the matter is necessary. A systematic approach is required to analyze this "infinite case". A first step may be the standardization of a tournament which has an infinite horizon as a main characteristic. This infinite horizon may imply a high computational cost which in turn may create long-time running simulations. Also, introducing a high number of strategies will raise that cost as well. But a big difference between the 1980s and now it is that we have that computational power at our disposal. Once set, strategies should be developed and pit against each other.

In [5] the authors argue and demonstrate that there is no winner for the infinite case. This, does not mean that we should not analyze performance of strategies in infinte settings. Whatever the results, they should be interpreted within the context of the setting that produced them.

Appendices

Table 1: matches=1000, turns=200, turns_dev=None

Rank	Name	Median_score	Wins
0	Revised Downing	2.891785714285714	2.0
1	First by Nydegger	2.7587499999999996	0.0
2	First by Grofman	2.7096428571428572	0.0
3	Cooperator	2.687142857142857	0.0
4	First by Stein and Rapoport: 0.05: (D, D)	2.6378571428571425	10.0
5	First by Shubik	2.6135714285714284	2.0
6	First by Tideman and Chieruzzi: (D, D)	2.6125000000000003	11.0
7	Grudger	2.611071428571429	3.0
8	First by Davis: 10	2.6048214285714284	3.0
9	Tit For Tat	2.555357142857143	0.0
10	First by Feld: 1.0, 0.5, 200	2.0742857142857143	10.0
11	First by Tullock	2.003214285714286	9.0
12	First by Joss: 0.9	1.924642857142857	11.0
13	Random: 0.5	1.837142857142857	4.0
14	Alternator	1.7432142857142856	5.0

Table 2: matches=1000, turns=200, turns_dev=40

Rank	Name	Median_score	Wins
0	Revised Downing	2.8896103896103895	2.0
1	First by Nydegger	2.7583732057416266	0.0
2	First by Grofman	2.7091592617908407	0.0
3	Cooperator	2.6852358168147643	0.0
4	First by Stein and Rapoport: 0.05: (D, D)	2.6373889268626107	10.0
5	First by Tideman and Chieruzzi: (D, D)	2.611073137388927	11.0
6	First by Shubik	2.610731373889269	2.0
7	Grudger	2.610047846889952	3.0
8	First by Davis: 10	2.6033834586466167	3.0
9	Tit For Tat	2.5538277511961724	0.0
10	First by Feld: 1.0, 0.5, 200	2.0727956254272044	10.0
11	First by Tullock	1.9969241285030759	9.0
12	First by Joss: 0.9	1.9224196855775806	11.0
13	Random: 0.5	1.832535885167464	4.0
14	Alternator	1.7334244702665755	2.0

Figure 1: matches=1000, turns=200, turns_dev=None

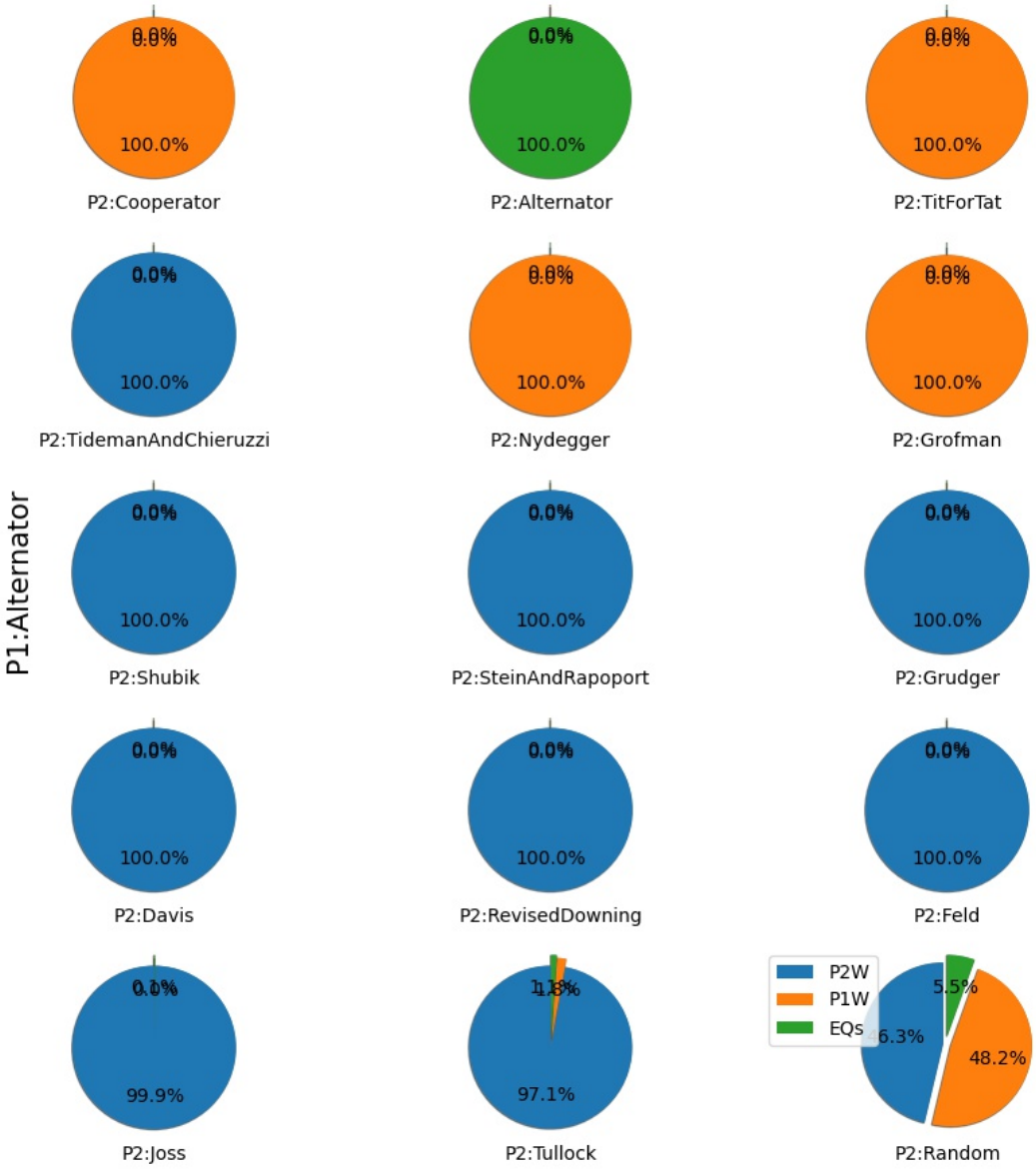


Figure 2: matches=1000, turns=200, turns_dev=40

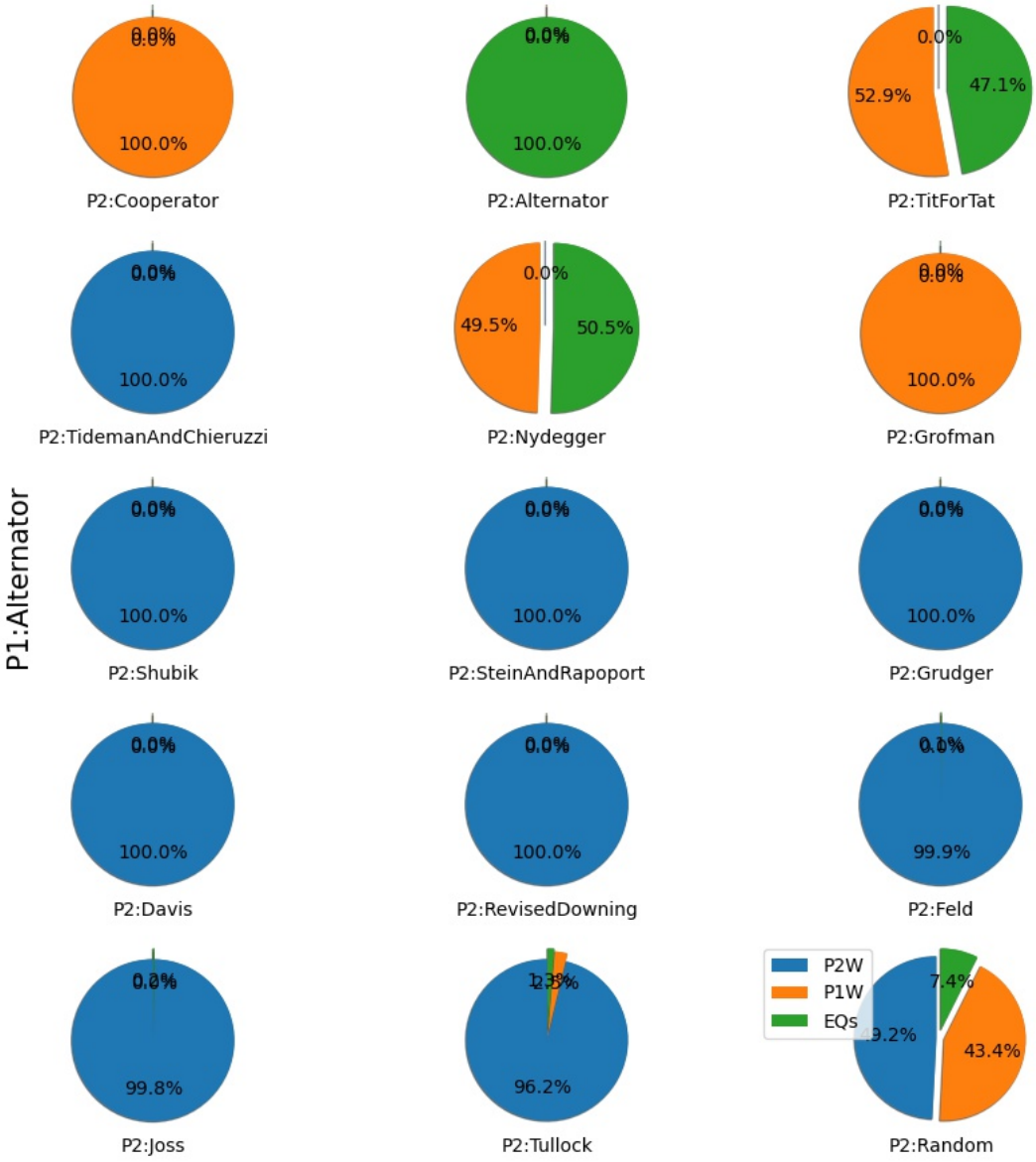


Table 3: matches=1000, turns=200, turns_dev=None

Rank	Name	Median_score	Wins
0	Second by RichardHufford	3.0359090909090907	14.0
1	Second by Harrington	3.015113636363636	11.0
2	Second by Tranquilizer	3.0113636363636362	14.0
3	Second by GraaskampKatzen	2.995681818181818	0.0
4	Second by Tideman and Chieruzzi	2.995	0.0
5	Second by Borufsen	2.9936363636363637	0.0
6	Second by Weiner	2.9931818181818177	0.0
7	Second by Kluepfel	2.9861363636363634	1.0
8	Second by Colbert	2.985227272727272	11.0
9	Second by Cave	2.984772727272727	0.0
10	Second by Getzler	2.9794318181818182	1.0
11	Second by WmAdams	2.979318181818182	0.0
12	Second by Eatherley	2.959090909090909	0.0
13	Second by Black	2.9495454545454547	0.0
14	Second by Tester	2.945340909090909	11.0
15	Second by Rowsam	2.940681818181818	0.0
16	Second by Leyvraz	2.9400000000000004	1.0
17	Second by Champion	2.9325	0.0
18	Second by White	2.9265909090909092	0.0
19	Second by Yamachi	2.9197727272727274	0.0
20	Second by Mikkelson	2.917272727272727	0.0
21	Second by Grofman	2.910227272727273	1.0
22	Second by Appold	2.9073863636363635	1.0

Table 4: matches=1000, turns=200, turns_dev=40

Rank	Name	Median_score	Wins
0	Second by Harrington	3.0426387771520513	11.0
1	Second by RichardHufford	3.0116653258246178	14.0
2	Second by Tranquilizer	3.0108608205953336	13.0
3	Second by GraaskampKatzen	2.9955752212389384	0.0
4	Second by Tideman and Chieruzzi	2.994770716009654	0.0
5	Second by Borufsen	2.993765084473049	0.0
6	Second by Weiner	2.992759452936444	0.0
7	Second by Kluepfel	2.985518905872888	1.0
8	Second by Colbert	2.984714400643605	11.0
9	Second by Cave	2.982703137570394	0.0
10	Second by Getzler	2.9794851166532585	1.0
11	Second by WmAdams	2.9678197908286403	0.0
12	Second by Eatherley	2.9605792437650846	0.0
13	Second by Tester	2.9593724859211585	11.0
14	Second by Black	2.9517296862429605	0.0
15	Second by Champion	2.947304907481899	0.0
16	Second by Grofman	2.9432823813354787	1.0
17	Second by Leyvraz	2.942880128720837	1.0
18	Second by Rowsam	2.9382542236524536	0.0
19	Second by Yamachi	2.937650844730491	0.0
20	Second by Mikkelson	2.932019308125503	0.0
21	Second by White	2.9283990345937254	0.0
22	Second by Appold	2.9199517296862427	0.0

Figure 3: matches=1000, turns=200, turns_dev=None

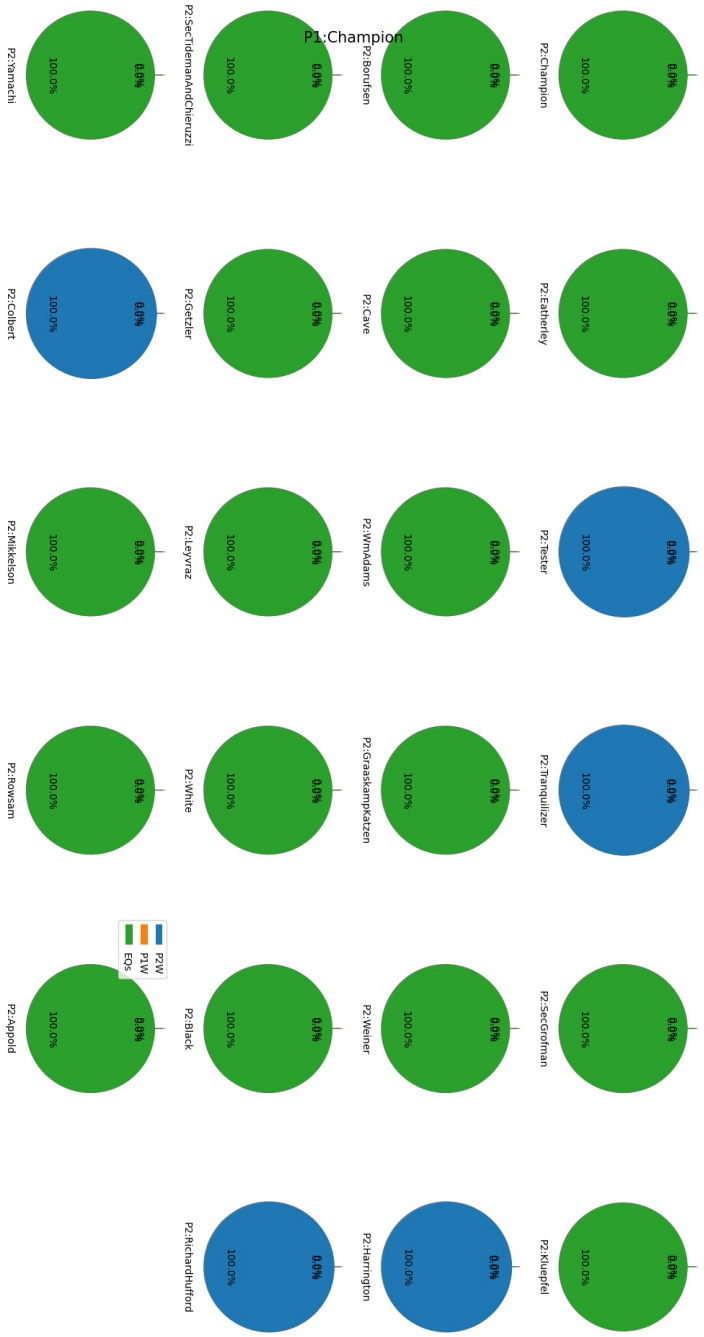


Figure 4: matches=1000, turns=200, turns_dev=40

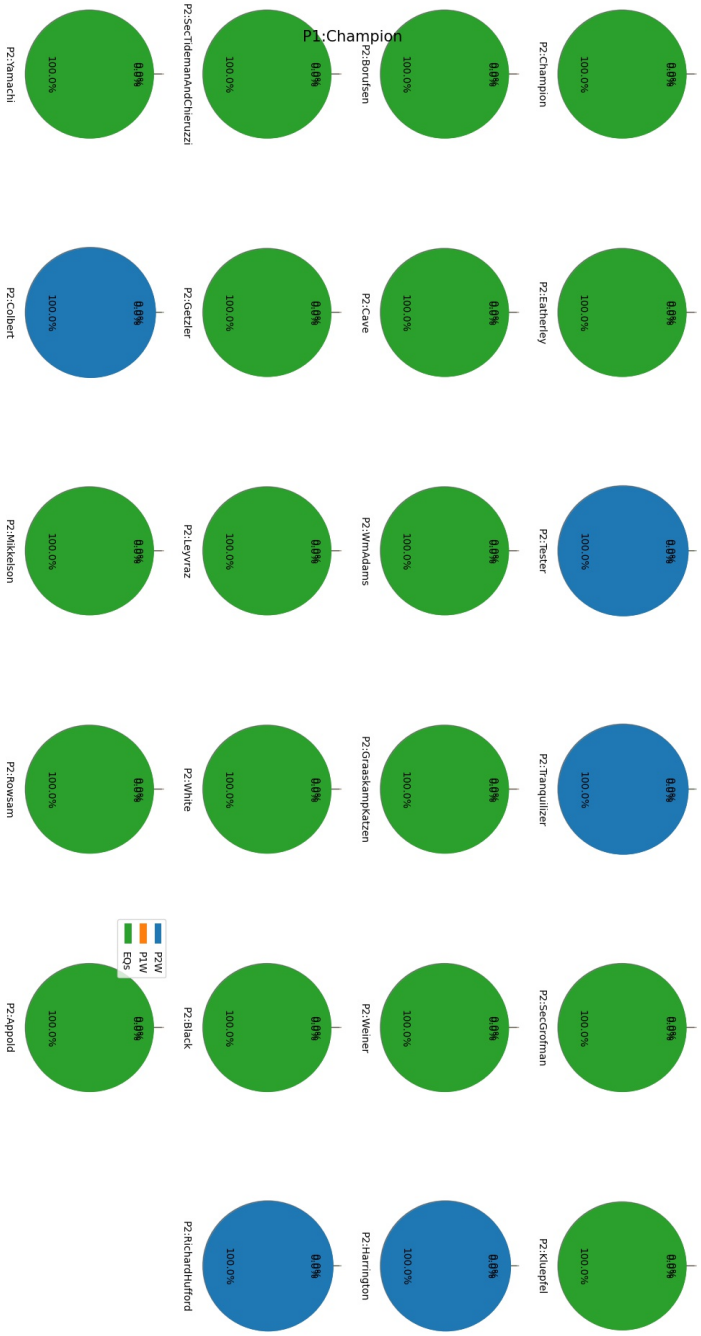


Table 5: matches=1000, turns=200, turns_dev=None

Rank	Name	Median_score	Wins
0	Second by RichardHufford	3.0031521739130436	14.0
1	Second by Tideman and Chieruzzi	2.994347826086957	1.0
2	Second by GraaskampKatzen	2.9932608695652174	1.0
3	Second by Weiner	2.9910869565217393	1.0
4	Second by Kluepfel	2.979347826086957	2.0
5	Second by Tranquilizer	2.9791304347826086	14.0
6	Second by Cave	2.9771739130434782	1.0
7	Second by Harrington	2.9730434782608692	11.0
8	Second by Borufsen	2.957608695652174	0.0
9	Second by WmAdams	2.9556521739130437	0.0
10	Second by Colbert	2.9526086956521733	11.0
11	Second by Getzler	2.9480434782608693	2.0
12	Second by Rowsam	2.936956521739131	1.0
13	Second by Tester	2.926086956521739	11.0
14	Second by Yamachi	2.917934782608696	1.0
15	Second by Eatherley	2.9174999999999995	0.0
16	Second by Mikkelson	2.9160869565217387	1.0
17	Second by Grofman	2.912934782608696	2.0
18	Second by White	2.903695652173913	0.0
19	Second by Black	2.9023913043478267	0.0
20	Second by Leyvraz	2.9008695652173913	1.0
21	Second by Champion	2.892608695652174	0.0
22	Second by Appold	2.878152173913044	1.0
23	Alternator	1.9591304347826088	13.0

Table 6: matches=1000, turns=200, turns_dev=40

Rank	Name	Median_score	Wins
0	Second by RichardHufford	2.996501749125437	14.0
1	Second by Tideman and Chieruzzi	2.9940029985007497	1.0
2	Second by GraaskampKatzen	2.9925037481259373	1.0
3	Second by Weiner	2.991004497751124	1.0
4	Second by Harrington	2.980259870064968	11.0
5	Second by Kluepfel	2.9787606196901555	2.0
6	Second by Tranquilizer	2.978385807096452	13.0
7	Second by Cave	2.975262368815592	1.0
8	Second by Borufsen	2.9575212393803096	0.0
9	Second by WmAdams	2.9532733633183406	0.0
10	Second by Colbert	2.952023988005997	11.0
11	Second by Getzler	2.949025487256372	2.0
12	Second by Rowsam	2.9361569215392302	1.0
13	Second by Tester	2.926786606696652	11.0
14	Second by Yamachi	2.92103948025987	1.0
15	Second by Grofman	2.920039980009995	2.0
16	Second by Mikkelson	2.918540729635182	1.0
17	Second by Eatherley	2.917541229385307	0.0
18	Second by Black	2.903298350824588	0.0
19	Second by White	2.9015492253873068	0.0
20	Second by Leyvraz	2.900299850074963	1.0
21	Second by Champion	2.896301849075462	0.0
22	Second by Appold	2.8821839080459775	1.0
23	Alternator	1.973638180909545	13.0

Figure 5: matches=1000, turns=200, turns_dev=None

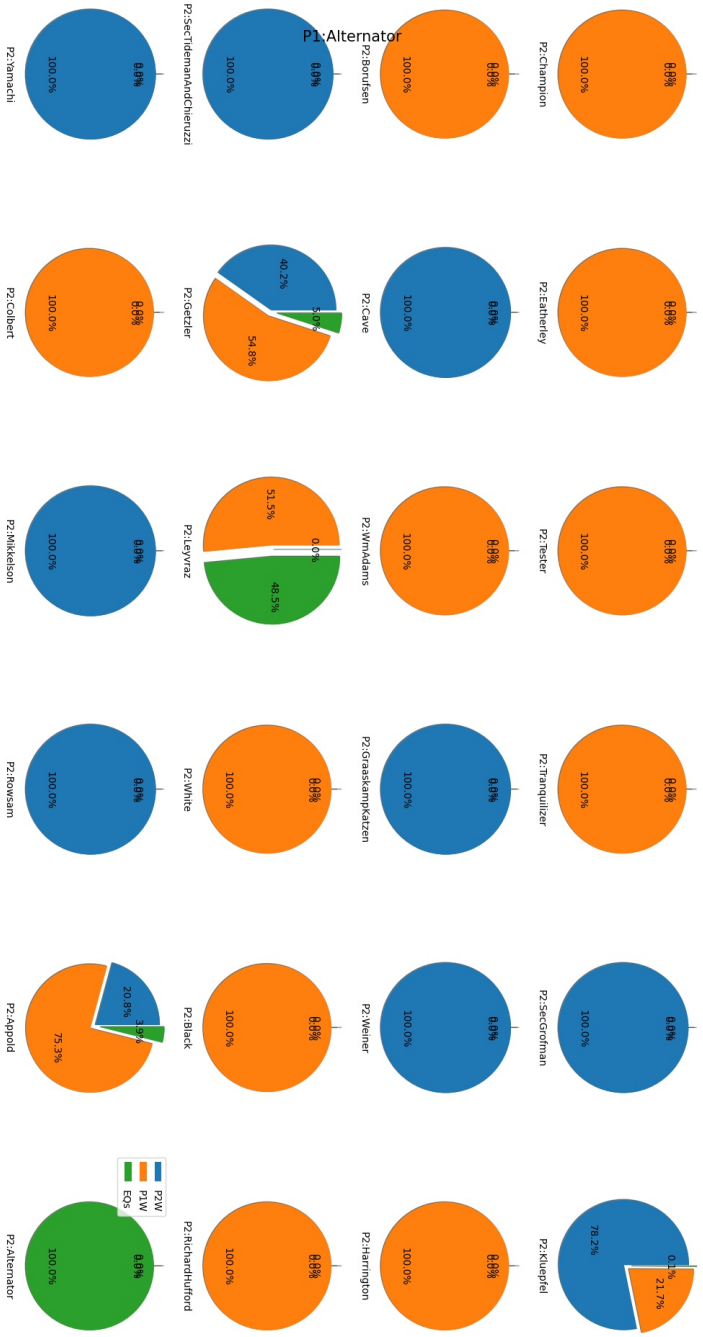
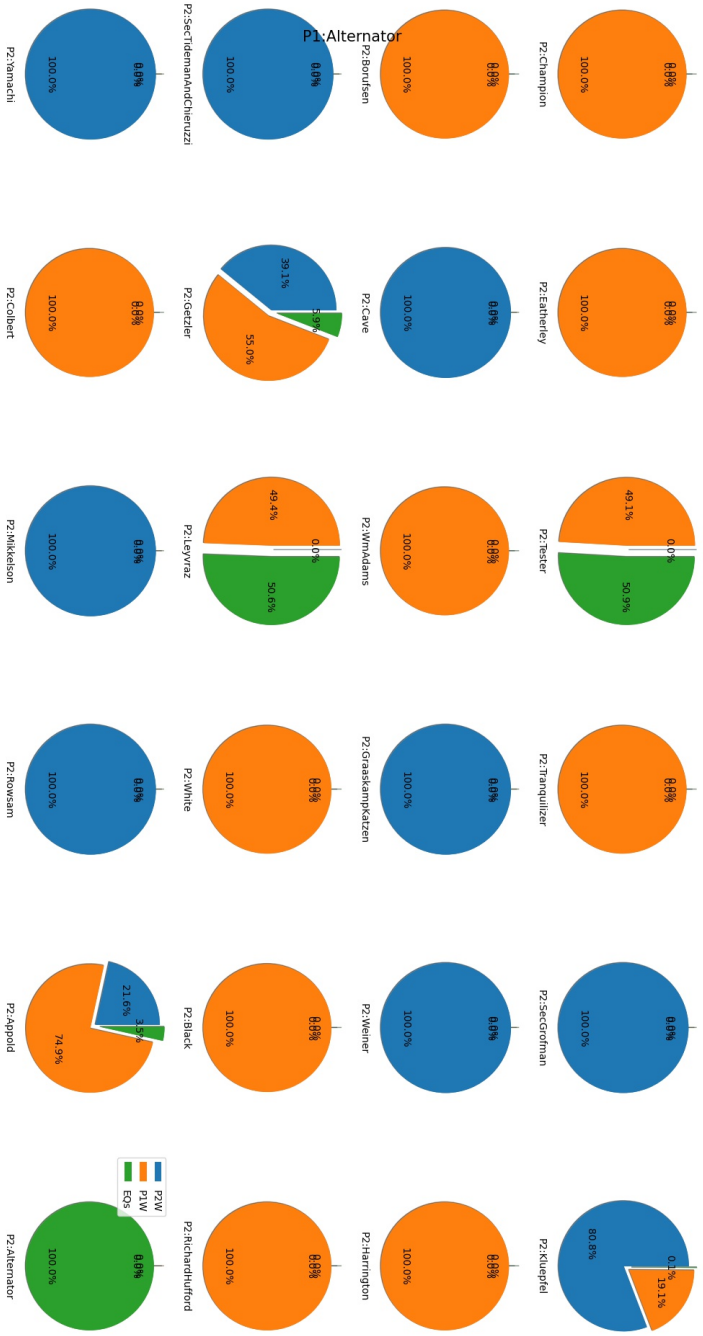


Figure 6: matches=1000, turns=200, turns_dev=40



Bibliography

[1] Robert Axelrod; William D. Hamilton. "The Evolution of Cooperation". In: *American Association for the Advancement of Science* 211.4489 (1981), pp. 1390–1396.

[2] Roger B. Myerson. *Game theory: analysis of conflict*. First Harvard University Press, 1997. ISBN: 0-674-34116-3.

[3] Yoav Shoham Kevin Leyton-Brown. *Essentials of game theory*. Morgan and Claypool Publishers, 2008. ISBN: 9781598295948.

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

[5] Matthijs van Veelen Julián García. *No Strategy Can Win in the Repeated Prisoner's Dilemma: Linking Game Theory and Computer Simulations*. 2018. URL: https://www.frontiersin.org/articles/10.3389/frobt.2018.00102/full.

[6] Matplotlib documentation. *Matplotlib: 3.3.3*. 2020. URL: https://matplotlib.org/3.3.3/contents.html.

[7] Numpy documentation. *Numpys: 1.19*. 2020. URL: https://https://numpy.org/doc/1.19/.pydata.org/docs.

[8] Pandas documentation. *Pandas: 1.2.0*. Dec. 2020. URL: https://pandas.pydata.org/docs/.

[9] PEP documentation. *PEP docs*. Dec. 2020. URL: https://www.python.org/dev/peps/.

[10] Mircea Milencianu. *Personal github for viewing full results for the simulation*. Feb. 2021. URL: https://github.com/MirceaM11/paper_implementation/tree/official_tour/vizualization.

[11] Robert Axelrod. "Effective Choice in the Prisoner's Dilemma". In: *The Journal of Conflict Resolution* 24.1 (Mar. 1980), pp. 3–25.

[12] Robert Axelrod. “More Effective Choice in the Prisoner’s Dilemma”. In: *The Journal of Conflict Resolution* 24.3 (Sep. 1980), pp. 379–403.