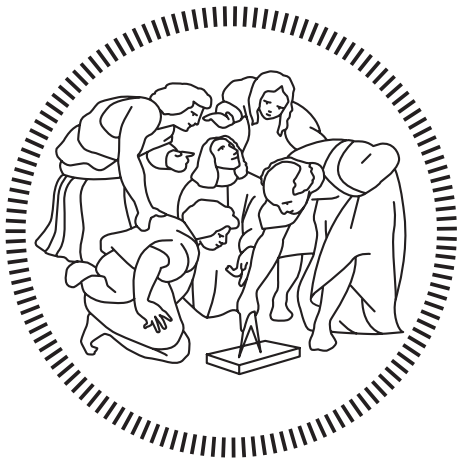
Politecnico di Milano

SCHOOL OF INDUSTRIAL AND INFORMATION ENGINEERING
Master of Science – Computer Science & Engineering



Efficient Algorithms for Computing Semivalue Single Point Solution Concepts of TU Cooperative Games

Supervisor
Prof. Roberto LUCCHETTI

Candidate
Alessandro AIMI – 899642

Academic Year 2019 – 2020

... correndo spensierati, si rotolavano a terra
per la sola gioia d'esser vivi ...

— Rudyard Kipling, *Il libro della giungla*

Acknowledgements

First of all, I thank Professor Roberto Lucchetti because, despite my background, he welcomed my desire of working on mathematical subjects, he granted me the freedom of becoming passionate about what I was doing and he helped me navigate around the hurdles I could not foresee alone.

I thank my parents because, when I gave them my worst, they always decided to give me their best in return.

With all my heart, I thank my brother Francesco. For all the time you spent with me, looking for rhythm in the strangest of times.

I thank all my friends, near and far, for the precious people you are.

Finally, I thank Chiara, as she made this time fly in the sweetest and most puzzling of ways.

Abstract

English

Game Theory has gained a lot of importance in recent days due to the several topics to which it can be efficiently applied. In particular, Cooperative Game Theory deals with either an efficient way to allocate utilities among cooperating players, or to describe, as carefully as possible, the relative power of the players. In this last case, players need not to be human beings, as they usually are objects interacting. Thus, for instance, we could be interested in measuring the (relative) power of parties in a Parliament, or in establishing how crucial is a node in a network. In order to do this, semivalues are an important tool. The most famous one is the Shapley Value, and this is due to the fact that it can serve as a general solution of cooperative games, while other semivalues, like, for instance, the Banzhaf semivalue, can be used only to measure power, so their use is limited to the so called simple games. Unfortunately, the definition of semivalues entails exponential computation on the number of players, so the problem of exactly calculating them is hard in the general case. However, in many instances this issue can be tackled, or at least mitigated, by exploiting the structural properties of specific cooperative game classes.
In this work I will survey over algorithms of tractable complexity for computing semivalues of some useful game classes. At first I will revise a framework to model a general class of cooperative games, recently introduced in (Tarkowski et al. 2018). In this class, the authors give conditions to check if the semivalues for a game can be obtained polynomially and, when this is true, they provide an algorithm to compute them. Then, I will consider another algorithm, introduced in (T. Matsui and Y. Matsui 2000), to compute semivalues for weighted voting games, a widely employed subclass of simple games. This algorithm is only pseudopolynomial, in the sense that it is polynomial at the condition to put an upper bound on the size of the weights attached to the players. Finally, I will analyze a recent polynomial algorithm to compute the Shapley Value for games defined for Boolean Conjunctive Queries on relational databases. This algorithm was introduced in (Livshits et al. 2020). Studing the algorithm in detail, I will argue by examples that there is a problem in the computation of a subcase. I will explain why, I will propose some possible corrections and provide an improved version which correctly computes all semivalues.

Italian

Negli ultimi anni la Teoria dei Giochi ha ottenuto molta rilevanza, dato il gran numero di situazioni in cui può essere applicata efficientemente. In particolare, la Teoria dei Giochi Cooperativi si occupa sia di allocare utilità in modo efficiente tra i giocatori cooperanti, sia di descrivere, nel modo più accurato possibile, il potere relativo dei giocatori. In quest'ultimo caso, non è necessario che i giocatori rappresentino esseri umani, in quanto spesso il problema riguarda oggetti che interagiscono. Quindi, per esempio, potremmo voler misurare il potere (relativo) dei partiti in un Parlamento o stabilire quanto sia cruciale un nodo all'interno di una rete. Per fare questo, i semivalori sono uno strumento importante. Il più famoso di essi è il Valore di Shapley e ciò è dovuto al fatto che può servire come soluzione generale dei giochi cooperativi, mentre altri semivalori, come, per esempio, il semivalore di Banzhaf, possono essere usati solo per misurare il potere dei giocatori, perciò il loro impiego è limitato ai cosidetti giochi semplici. Sfortunatamente, la definizione di semivalore comporta un calcolo esponenziale sul numero di giocatori, quindi, nel caso generale, ottenere tali valori è un problema di complessità intrattabile. Spesso questo problema può essere affrontato, o perlomeno mitigato, sfruttando le proprietà strutturali presenti in specifiche classi di giochi cooperativi.
In questo lavoro esaminerò diversi algoritmi di complessità trattabile per il calcolo dei semivalori in alcune utili classi di giochi. Per prima cosa presenterò una cornice teorica usata per rappresentare una classe molto generale di giochi cooperativi, introdotta recentemente in (Tarkowski et al. 2018). In tale classe, gli autori forniscono delle condizioni per controllare se i semivalori per un gioco possono essere calcolati polinomialmente e, se questo è il caso, presentano un algoritmo per ottenerli. Successivamente, considererò un altro algoritmo, introdotto in (T. Matsui and Y. Matsui 2000), per calcolare i semivalori dei giochi che modellano le votazioni pesate, essendo una sottoclasse di giochi semplici ampiamente utilizzata. Questo algoritmo è solo pseudopolinomiale, nel senso che risulta polinomiale una volta stabilito un limite massimo dei pesi associati ai giocatori. Infine, analizzerò un recente algoritmo polinomiale per calcolare il Valore di Shapley nei giochi definiti su Interrogazioni Congiuntive Booleane nelle basi di dati relazionali. Tale algoritmo è stato introdotto in (Livshits et al. 2020). Studiando dettagliatamente l'algoritmo, argomenterò con esempi che esso presenta un problema nel calcolo di un sottocaso. Spiegerò perché, proporrò alcune possibili correzioni e ne fornirò una versione migliorata che calcola correttamente tutti i semivalori.

Contents

Acknowledgements **v**

Abstract **vii**
English . . . vii
Italian . . . ix

Contents **xii**

List of Figures **xiii**

1 Introduction **1**

2 Preliminaries **5**
2.1 Cooperative Games . . . 5
2.2 Semivalues . . . 7

3 A Generic Framework for Transferable Utility Games **11**
3.1 The class $\mathscr{M}$ of Models for Characteristic Functions . . . 11
3.2 The Polynomial Subclass $\mathscr{M}^*$ of $\mathscr{M}$ . . . 15
3.3 Further Example of the Framework Application: Maintenance Games 18
3.4 Faster Algorithms for Argumentation and Maintenance Games . . . 20
3.4.1 Subclass of Argumentation Games . . . 20
3.4.2 Maintenance Games . . . 21
3.5 Comparing the class $\mathscr{M}$ to Basic Generalized Additive Games . . . 21
3.5.1 Basic Generalized Additive Games . . . 21
3.5.2 Basic GAGs as elements of $\mathscr{M}$ . . . 22
3.5.3 Elements of $\mathscr{M}$ as Basic GAGs . . . 23

4 Computing Semivalues of Weighted Voting Games **25**
4.1 Weighted Voting Games . . . 25
4.2 Pseudopolynomial Algorithm . . . 26

5 Query Answering Games on Database Tuples **29**
5.1 Quantifying Data Relevance in Databases . . . 30
5.2 Relational Databases and Queries . . . 32
5.3 Query Answering Games . . . 35
5.4 An Efficient Algorithm for the Shapley Value of Boolean Query Answering Games . . . 37
5.4.1 Identified Issues in the Algorithm . . . 40

5.4.2 Proposed Changes 45

6 Conclusions **47**

Bibliography **51**

A Glossary of Notation **53**

B Qsemivalue Python Implementation **59**
B.1 Code 59
B.2 Example of Json Database Tables in Input 70

List of Figures

3 A Generic Framework for Transferable Utility Games

Figure 3.1 Running example of an argumentation game: graph and characteristic function of the game $\mathcal{A}$ 11
Figure 3.2 Running example of an argumentation game: model $M_{\mathcal{A}}$ of the characteristic function of $\mathcal{A}$ 13
Figure 3.3 Example of a maintenance game: tree and characteristic function 19

5 Query Answering Games on Database Tuples

Figure 5.1 Portion of a database storing information about car owners . . 29
Figure 5.2 A simple database storing information about some beautiful songs 33
Figure 5.3 The database D of example 5.4.1 41
Figure 5.4 The database D of Example 5.4.2 42

B Qsemivalue Python Implementation

Figure B.1 File `Author.json`: the Author table 70
Figure B.2 File `Citations.json`: the Citations table 70
Figure B.3 File `Inst.json`: the Inst table 70
Figure B.4 File `Pub.json`: the Pub table 70

Chapter 1

Introduction

Coalitional (or cooperative) games study how groups of self interested agents, called players, interact to accomplish more together than they could achieve individually. These interactions are modeled using a set of players, a set of actions, and a preference profile over the joint outcomes. Thus, the final result for players does not depend only from the choices they individually make, but also from the choices of other agents. While the players can cooperate to achieve the outcomes that they want, their reasoning is always self-interested and they seek the actions most likely to bring the highest possible payoff (utility) for themselves. The study of cooperative games is a branch of Game Theory, which, generally speaking, can be described as the study of taking optimal decisions in presence of multiple interactive agents.
"Thus, a game is a simplified, yet very efficient, model of real life every day situations. Though the first, and probably more intuitive, applications of the theory were in an economical setting, theoretical models and tools of this theory nowadays are spread on various disciplines. To quote some of them, we can start from psychology: a more modern approach than classical psychoanalysis takes into account that the human being is mainly an interactive agent. So to speak, we play everyday with our professors/students, with our parents/children, with our lover, when bargaining with somebody. Also the Law and the Social Sciences are obviously interested in Game Theory, since the rules play a crucial role in inducing the behaviour of the agents. Not many years after the first systematic studies in Game Theory, interesting applications appeared in studying animals, starting with the analysis of competing species. It is much more recent and probably a little surprising to know that recent applications of the theory deal with genes in microbiology, or computers in telecommunication problems." (Lucchetti 2011)

Semivalues are a class of solution concepts for cooperative games. *"Solution concepts describe, in each game, the set of outcomes believed to arise based on the type of reasoning employed by the players. Coalitional solution concepts are often universal, i.e. applicable to all types of coalitional games, and attempt to ensure some form of stability. No matter how good an outcome is for the society at large, it may not be enforced if several players can become rich by deviating. In general, there exists a tension between outcomes that are beneficial for social welfare and outcomes that are stable."* (Branzei 2010)
Given the high number of applications of Cooperative Game Theory, many thought it would be interesting to study how much power each player holds in a game, or, in

other words, the degree of influence he has on the final outcome(s). In order to do this, the family of solution concepts called power indices was introduced, together with its widely used subfamily denoted as semivalues.
Semivalues have a number of applications in a variety of fields. For example, the courts in New York State have accepted the Banzhaf index (also called the Coleman value or Chow parameters) as an appropriate measure for weighted voting systems. Moreover, the Shapley value has been applied in various areas and fields beyond Cooperative Game Theory (e.g., (Aumann and Myerson 2003; Aziz and Keijzer 2013)), such as bargaining foundations in economics (Gul 1989), takeover corporate rights in law (Nenova 2003), pollution responsibility in environmental management (Liao, Zhu, and Shi 2015; Petrosjan and Zaccour 2003), influence measurement in social network analysis (Narayanam and Narahari 2010), and utilization of multiple Internet Service Providers (ISPs) in networks (Ma et al. 2010). Interestingly, in the database field the Shapley value has been used to attribute a level of inconsistency to a statement in an inconsistent knowledge base (Hunter and Konieczny 2010; Yun et al. 2018); the idea is natural: as wealth, adopt a measure of inconsistency for a set of logical sentences (Grant and Hunter 2006), and then associate to each sentence its Shapley value. So, in many cases, computing semivalues for a cooperative game is very relevant because, even if some of them do not represent a payoff distribution between the players, they model the responsibility of players towards the final outcome(s) of the game.

Ideally we would like to be able to compute semivalues in an efficient manner, but this is not an easy task. We will see that the computation of semivalues depends on the number of coalitions that can form between players of a cooperative game, which is exponential on the size of the set of players. Luckily, the structure of some games allows us to exploit specific properties in order to reduce the number of steps needed to compute a semivalue to a quantity polynomial to the size of the players set. The scope of this work is to survey some of the most relevant techniques in the literature for efficiently calculating semivalues.
To do this, I will begin by using the results explained in (Tarkowski et al. 2018) for network centrality measures and generalizing them so that they can be applied to the characteristic function of any generic cooperative game. This means that now we have a generic framework that provides us with a number of constraints on the structure of games, basically creating a subclass of cooperative games. As a consequence, if a cooperative game can be rewritten as a member of this class, we can easily check the polynomiality of its semivalue computation. Furthermore, the authors provide a polynomial algorithm to do so. I will give some examples of application of the framework and a brief comparison of it with the class of Basic Generalized Additive Games, as they share many similarities. Unfortunately, the aforementioned framework seems to be more suitable to model certain kinds of games than others. Specifically, for some simple games it may be impossible to find a suitable model. One example are weighted voting games, for which has been proved the non polynomiality of the computation of specific semivalues. Given the usefulness and relevance of this kind of games, I felt it was important to include a pseudopolynomial algorithm to compute their semivalues. Another interesting class of simple games I elaborated on is the one built on the problem of answering queries on a relational database. Once we define such class of games, using semivalues, we are able to measure the contribution of a database entry to a query answer. In (Livshits et al. 2020), it is proposed a

polynomial algorithm to compute the Shapley Value for games built on Boolean Conjunctive Queries, under some constraints. Furthermore, this algorithm is used to prove a dichotomy on the complexity of such computation. The algorithm, as presented by its original authors, has some slight inaccuracies and it does not compute correctly in a specific case. I will analyse these issues and propose some corrections, as well as providing a corrected version of the algorithm generalized to any semivalues. In general, the problem of deciding if semivalues for a generic class of games can be computed polynomially or not is still an open question that has yet to be fully answered.
The remainder of this work is organized as follows. The following chapter is devoted to the exposition of the basic formal concepts of Cooperative Game Theory needed for the comprehension of the rest of the work. In chapter 3, the generic framework to model cooperative games will be presented, together with a couple of examples of application and some considerations on its relation with Basic Generalized Additive Games. Chapter 4 contains a pseudopolynomial algorithm to compute semivalues for the relevant class of weighted voting games. In chapter 5 it is explained ho we can build games on databases and queries. Then, it is presented the algorithm for the computation of the Shapley Value in answering Boolean Conjunctive Queries, as well as its issues and some proposed corrections. Finally, appendix A is a glossary of the many symbols and notations used in this work and appendix B contains a Python implementation of the corrected algorithm of chapter 5.

Chapter 2

Preliminaries

In this section, I will introduce the key concepts pertaining to cooperative game theory necessary for the understanding of this work. The exposition is mainly based on (Lucchetti 2011). Throughout this document I will assume acquaintance of the reader with the basic concepts of complexity theory, graph theory, probability theory, set theory, relational algebra and, obviously, cooperative game theory. Nonetheless, a brief introduction to specific concepts will be present at the beginning of each section they are used in.

2.1 Cooperative Games

In game theory, a *cooperative game* (or *coalitional game*) is a game with competition between groups of players, called *coalitions*, due to the possibility of external enforcement of cooperative behavior. This enforcement of cooperative behavior is encoded in the payoff the game assigns to coalitions: if players know that they may have an advantage - namely, an higher payoff - in acting cooperatively compared to acting individually, they will choose to form a coalition. In this context the payoff is called *utility*.
Utility is transferable if one player can losslessly transfer part of its utility to another player. Avoiding unneeded formalisms, this means that we can describe a *transferable utility* (TU) cooperative game by just specifying a single utility value for a coalition, that represents the total utility of the players in the coalition. Transferable Utility is assumed in many cooperative games, because they become much more tractable. All the games presented in this work are TU cooperative games, so I will refer to them as TU games, cooperative games or coalitional games regardless.

Definition 1 (TU Cooperative Game). *Given a finite set of players N and a function $\nu\colon 2^N \to \mathbb{R}$, such that $\nu(\emptyset) = 0$ and 2^N is the power set of N, a Transferable Utility Cooperative Game is a couple (N, ν).*

Two important coalition are the empty set $\emptyset$, and the one composed by the set all players N, named *grand coalition*. Also, the utility function ν is called *characteristic function* of the game. In most cases, a TU game can just be described by its characteristic function, but in this work I will always include the set of the players in definitions, as we will often need to analyse how ν is built in relation to the context

players come from, or try to remap the set of players entirely.
Some of the most relevant properties a cooperative game can satisfy are listed below. A game (N, ν) is said to be

- *monotonic* if it holds $\nu(S) \leq \nu(T)$ for all $S, T \subseteq N$ such that $S \subseteq T$;
- *superadditive* if it holds that $\nu(S \cup T) \geq \nu(S) + \nu(T)$ for all $S, T \subseteq N$ such that $S \cap T = \emptyset$;
- *convex* or *supermodular* if it holds that $\nu(S \cup T) + \nu(S \cap T) \geq \nu(S) + \nu(T)$ for all $S, T \subseteq N$.

Obviously, all convex games are also superadditive.

Example 2.1.1

Consider the game (N, ν), where $N = \{a, b, c\}$ and the characteristic function is as follows.

S	$\nu(S)$	S	$\nu(S)$
$\emptyset$	0	$\{a, b\}$	7
$\{a\}$	1	$\{a, c\}$	3
$\{b\}$	2	$\{b, c\}$	11
$\{c\}$	0	N	15

As we can see, this game is monotonic and superadditive, but it is not convex. In fact $\nu(\{a, b, c\}) + \nu(\{b\}) < \nu(\{a, b\}) + \nu(\{b, c\})$.

Another interesting subclass of cooperative games is the one of *simple games*.

Definition 2 (Simple Game). *A TU game (N, ν) is called simple provided that ν assumes values in $\{0, 1\}$, it is monotonic and $\nu(N) = 1$.*

Typically, simple games model majority situations, in which a coalition with utility 1 is called a *winning coalition*.

Now it remains to define what are the rational outcomes of a game, namely, how players will share the utility they obtained by forming coalitions. This can be modeled by any *solution vector* $\mathbf{x} = (x_1, \ldots, x_n)$, where $n = |N|$ and each x_i represents the amount assigned to player i. This does not mean that any solution vector is an admissible outcome.
Because in most games there is not a single admissible outcome or way to share the coalitions payoff, there exists many *solution concepts* for a game, which are (possibly empty) sets of solution vectors. In fact, different solution concepts may be better in different situations the game is trying to model.
That said, there are two minimal conditions necessary for any solution vector $\mathbf{x}$ to represent an outcome: no solution should assign to a player less than he is able to do by himself ($x_i \geq \nu(\{i\})$ for all i), otherwise he wouldn't participate in the game, and, in case of the grand coalition forming, that the division is feasible and the whole amount is distributed ($\sum_{i=1}^{n} x_i = \nu(N)$). The set of solution vectors which respect

these properties is called the set of *imputations* and any solution vector that represents an outcome must be inside it.

Example 2.1.2

Let's take the game in Example 2.1.1. An imputation for this game is the solution vector $\{5, 5, 5\}$, but will the players accept it? Of course not, because players b and c know that they can do better if they form a coalition without a. An accepted solution concept (also called *core* of the game) could be the set of imputations that, for every coalition, the sum of what is given to its players is greater or equal than the utility assigned by ν to the coalition. Some examples of such imputations are $\{3, 6, 6\}, \{1, 10, 4\}, \{4, 3, 8\}, \{4, 11, 0\}, \{4, 5, 6\}$.

Another important family of solution concepts are *power indices*. In most cases they do not model an outcome, but they give a quantitative idea of the power each player holds in a game. The most renown power indices are the Shapley value and the Banzhaf index. The subclass of power indices called *semivalues* is presented in depth in the following section, as they are central to the focus of this work.

2.2 Semivalues

In order to define semivalues, I will give a brief explanation of *probabilistic indices*, a subclass of power indices. Probabilistic indices are a *single value* solution concepts, as they define a single solution vector for each game. To ease the exposition, I will proceed to explain how a single component of this vector (i.e., the probabilistic index of a player) is calculated.
First we need to define the concept of *marginal contribution* of a player to a coalition he joins.

Definition 3 (Marginal Contribution). *Given a cooperative game (N, ν), a player $i \in N$ and a coalition $S \subseteq N \setminus \{i\}$, the marginal contribution $\mathrm{MC}(S, i)$ of player i to coalition $S \cup \{i\}$ is*

$$\mathrm{MC}(S, i) = \nu(S \cup \{i\}) - \nu(S).$$

Less formally, the marginal contribution is the utility a player adds to the value of a coalition when he joins it.

Example 2.2.1

The marginal contributions of the players in Example 2.1.1 are

S	$\mathrm{MC}(S, a)$
$\emptyset$	1
$\{b\}$	5
$\{c\}$	3
$\{b, c\}$	4

S	$\mathrm{MC}(S, b)$
$\emptyset$	2
$\{a\}$	6
$\{c\}$	11
$\{a, c\}$	12

S	$\mathrm{MC}(S, c)$
$\emptyset$	0
$\{a\}$	2
$\{b\}$	9
$\{a, b\}$	8

By looking at marginal contributions we can already have a glance at which players contribute the most to the value of coalitions.

This concept is central, as the definition of probabilistic index of a player is a weighted sum of the marginal contributions of the player to every coalition. The weights of this sum are, for every player i, a probability measure p_i on $2^{N\setminus\{i\}}$, set of the coalitions not containing him, such that $\sum_{S\in 2^{N\setminus\{i\}}} p_i(S) = 1$. Basically, $p_i(S)$ represents the probability of the coalition $S \cup \{i\}$ forming.

Semivalues are the subfamily of probabilistic indices in which the coefficients p_k of the weighted sum do not depend from the player i, but only from the size $k = |S|$ of the coalition.

Definition 4 (Semivalue of a player). *Given a cooperative game (N, ν) and a player $i \in N$, the semivalue $\pi_i^{\mathbf{p}}(\nu)$ of the player i is*

$$\begin{aligned}\pi_i^{\mathbf{p}}(\nu) &= \sum_{S\subseteq N\setminus\{i\}} p_k \cdot \mathrm{MC}(S, i) \\ &= \sum_{S\subseteq N\setminus\{i\}} p_k \cdot \big[\nu(S \cup \{i\}) - \nu(S)\big], \end{aligned} \tag{2.1}$$

where, given $n = |N|$, $\mathbf{p} = \{p_0, \ldots, p_{n-1}\}$ is a vector of probabilities on the sizes of coalitions not containing player i, such that $p_k \geq 0$ for all $k \in \{0, \ldots, n-1\}$ and $\sum_{k=0}^{n-1} \binom{n-1}{k} p_k = 1$.

So a semivalue is just a vector $\pi^{\mathbf{p}}(\nu) = \{\pi_1^{\mathbf{p}}(\nu), \ldots, \pi_n^{\mathbf{p}}(\nu)\}$. Moreover, a semivalue is said to be *regular* if $p_k > 0$ for all k.
From this definition a number of different semivalues arise, of which the most renown are

- the Shapley Value (1953), shortly denoted by σ, having $p_k = \frac{1}{n\binom{n-1}{k}}$;
- the Banzhaf Index (1965), shortly denoted by β, having $p_k = \frac{1}{2^{n-1}}$.

The probabilistic interpretation of the coefficients of the Shapley Value is that p_k is the probability that, between all possible permutations over N, the player is in the $(k+1)$-th position. Instead, the Banzhaf Index sets p_k to a uniform probability over the 2^n coalitions, so it is independent from the coalition size.

The Shapley Value has four interesting properties, defined for any TU game (N, ν):

1. *Efficiency*: $\sum_{i\in N} \sigma_i(\nu) = \nu(N)$;
2. *Symmetry*: for all players $i, j \in N$ and every coalition $S \subset N$ not containing them, if $\nu(S \cup \{i\}) = \nu(S \cup \{j\})$, then $\sigma_i(\nu) = \sigma_j(\nu)$;
3. *Null Player*: for $i \in N$, if $\nu(S) = \nu(S \cup \{i\})$ for all coalitions $S \subseteq N$, then $\sigma_i(\nu) = 0$;
4. *Additivity*: for any other TU game (N', ν'), $\sigma(\nu + \nu') = \sigma(\nu) + \sigma(\nu')$.

The first one tells us that the Shapley vale is also an imputation, thus, an outcome of the game. Usually, two players that verify the first part of property 2 are called *symmetric players*, so the Shapley Value is coherent with this definition. The same holds for players that verify the condition of property 3, as they are called *null players*. Moreover, the Shapley Value is the only semivalue satisfying this four properties. The Banzhaf Index, instead, only satisfies the last three.

Example 2.2.2

For the game in Example 2.1.1, the vector of probabilities for the Shapley Value is $\{\frac{1}{3}, \frac{1}{6}, \frac{1}{3}\}$ and for the Banzhaf Index is $\{\frac{1}{4}, \frac{1}{4}, \frac{1}{4}\}$. The computation follows from Formula 2.1: the Shapley Value of game (N, ν) is $\sigma(\nu) = \{3, \frac{15}{2}, \frac{9}{2}\} = \{3, 7.5, 4.5\}$ and the Banzhaf Index is $\beta(\nu) = \{\frac{13}{4}, \frac{31}{4}, \frac{19}{4}\}$. As we can see, the Shapley Value is efficient and a possible outcome of the game, while the Banzhaf Index is not.

The main focus of this work is to explore the computational complexity of semivalues of some relevant classes of TU games. By taking a look at the formula for computing semivalues we see that it is a sum on the possible coalitions over the set of players N. This means that if we want to compute any semivalue $\pi_i^{\mathbf{P}}(\nu)$ using the definition we will need to list all possible $2^{|N|}$ coalitions, ending up with a computation of exponential complexity on the size of N. So, would it be possible to obtain a polynomial computation of semivalues for, at least, some classes of games? The answer is not obvious in most cases, but yes, it is possible for some classes of games.
Throughout the course of this work I will assume that each element p_k of the vector of probabilities of a generic semivalue can be precomputed in $O(|N|)$ time or less. This can be easily generalized to polynomial time, but for the sake of simplicity it is better to keep this assumption as it is.

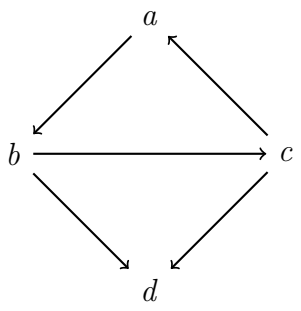
Chapter 3

A Generic Framework for Transferable Utility Games

Given a network, we can define functions called centrality measures that associate with each node a real value representing its importance in the network. This measures can be extended to groups of nodes, giving us the possibility to model group centrality measures as games over the set of nodes in a graph, and thus to use semivalues to compute the centrality of single nodes considering their synergies with any group of nodes in the network. In (Tarkowski et al. 2018) it is explained a framework to define a class $\mathscr{M}$ of models for group centrality measures with the ultimate goal of defining a list of constraints that allow polynomial computation of semivalues for a subclass $\mathscr{M}^* \subseteq \mathscr{M}$ of models satisfying such properties. These results are not exclusive to games based on centrality measures over networks so I will synthesize them referring to the class of generic cooperative TU games.

3.1 The class $\mathscr{M}$ of Models for Characteristic Functions

In order to provide a clear explanation of the framework I will use a running example of a class of games the model of which will be gradually built as I provide more definitions, paired with a simple example of a game.



S	$\nu(S)$	S	$\nu(S)$	S	$\nu(S)$
$\emptyset$	0	$\{ab\}$	1	$\{abc\}$	0
$\{a\}$	1	$\{ac\}$	1	$\{abd\}$	1
$\{b\}$	1	$\{ad\}$	2	$\{acd\}$	1
$\{c\}$	1	$\{bc\}$	1	$\{bcd\}$	1
$\{d\}$	1	$\{bd\}$	1	$\{abcd\}$	0
		$\{cd\}$	1		

Figure 3.1: Running example of an argumentation game: graph and characteristic function of the game $\mathcal{A}$

Example 3.1.1

In Figure 3.1 is shown a simple argumentation game $\mathcal{A} = (N, \nu)$, by providing its graph and utility function. Abstract argumentation games (Bonzon, Maudet, and Moretti 2014) are defined on a directed graph $\langle N, \mathcal{R} \rangle$, where $\mathcal{R} \subseteq N \times N$. Nodes represent a set of *arguments* N, while an edge $(i, j) \in \mathcal{R}$ between node i and node j means that argument i is *attacking* argument j. Given that, we can define the set $\mathcal{P}(i) = \{j \in N \colon (j, i) \in \mathcal{R}\}$ of arguments attacking j, called *attackers* of i. Vice versa we call $\widetilde{\mathcal{P}}(i) = \{j \in N \colon (i, j) \in \mathcal{R}\}$ the set of arguments attacked by i. For example in Figure 1, $\mathcal{P}(c) = \{b\}$ and $\widetilde{\mathcal{P}}(c) = \{a, d\}$. In argumentation games a coalitions of arguments represent an *opinion*. In our case we will measure the value of an opinion S as the number of arguments in it that are not attacked by an other argument of the same opinion, that is, $\nu(S) = |\{i \in S \colon \mathcal{P}(i) \cap S = \emptyset\}|$.

The main objective of this framework is to break up how value is assigned to coalitions, in order to group things as much as possible and to avoid listing all $2^{|N|}$ elements of the power set of players. To do this, it is introduced a new set of objects, called *items*, that will be used as an intermediary layer between coalitions and the characteristic function of the game. On one side, items act as bricks used to build the value of the characteristic function for a coalition. On the other, every player is connected to all items, but for every item this mapping can be in one of three fashions: positive, negative or neutral. These mappings are called *relations* and, once are defined for players, are automatically built for coalitions too. Also, items will be regrouped as well if they have similar behaviour in the model. In this way we move the focus of the problem from coalitions to the contribution of every player to the characteristic function. In fact there are always $2^{|N|}$ coalitions, but there may be significantly less ways in which all players contribute to value of the characteristic function.

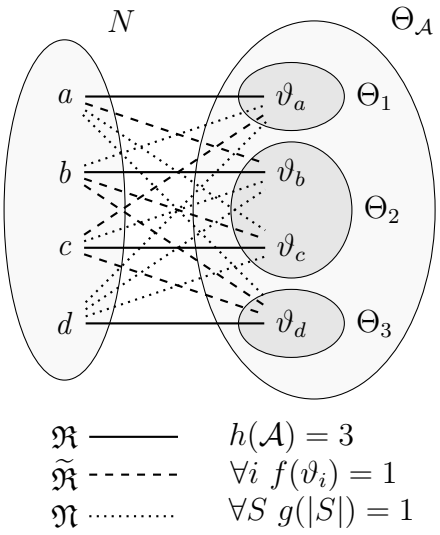
Items refer to entities that influence the value of a coalition in a specific game. When we want to model a game, it is up to us to define which is a suitable set of items. They can be players or other structural entities of the game linked to the way the value of a coalition is obtained. We assume that the characteristic function ν of the game can be computed by counting these *items*. Each item ϑ has his own real value $f(\vartheta) \in \mathbb{R}$, impacting the value of players and coalitions. Intuitively, we say that a player i contributes to a coalition $S \subseteq N \setminus \{i\}$ *through an item* ϑ if, when i is added to S, the value of the coalition increases by $f(\vartheta)$.
Given a game $\mathcal{G} = (N, \nu)$, the set of all items is denoted by $\Theta_{\mathcal{G}}$. We will partition $\Theta_{\mathcal{G}}$ into pairwise disjoint *groups* $\Theta_1, \Theta_2, \ldots, \Theta_{h(\mathcal{G})}$ of items according to $f(\vartheta)$. In principle, items of the same group should have the same value, whereas items from different groups should have different value; formal constraints on this partition will be defined later. $h(\mathcal{G})$ is the size of the partition, namely, the number of groups. This is a key concept because even if the number of items is exponential, $h(\mathcal{G})$ can be polynomial.

Relations control how items affect the value of players and coalitions. As we did for items, it is up to us to define them accordingly. There are three types of relations: neutral $\mathfrak{N}$, positive $\mathfrak{R}$ and negative $\widetilde{\mathfrak{R}}$. We will see that relations need to be chosen in a way such that the sum of the values of items in a positive relation to a coalition equals the utility of the coalition. Let's start by defining what is a relation between a player and an item, and then we will build the relation between a coalition and an

item from that.
A relation between players and items is simply a subset of $N \times \Theta_{\mathcal{G}}$. For example, player i is in a positive relation with item ϑ if $(i, \vartheta) \in \mathfrak{R}$, but for convenience we will denote it with $\mathfrak{R}(i, \vartheta)$. The three relations $\mathfrak{N}$, $\mathfrak{R}$ and $\widetilde{\mathfrak{R}}$, as defined between single players and items, are an exact partition of the set $N \times \Theta_{\mathcal{G}}$, so every couple (i, ϑ) is in one and only one of the three relations.
Now we can define when a coalition S is in a neutral, positive or negative relation with an item ϑ using the already mentioned relations between players and items:

$$\begin{aligned} \mathfrak{N}(S, \vartheta) &\iff \forall_{i \in S} \mathfrak{N}(i, \vartheta) \\ \mathfrak{R}(S, \vartheta) &\iff \exists_{i \in S} \mathfrak{R}(i, \vartheta) \wedge \nexists_{j \in S} \widetilde{\mathfrak{R}}(j, \vartheta) \\ \widetilde{\mathfrak{R}}(S, \vartheta) &\iff \exists_{i \in S} \widetilde{\mathfrak{R}}(i, \vartheta) \end{aligned} \tag{3.1}$$

Note that relations as defined for coalitions are pairwise mutually exclusive, so we just need two of them in order to obtain all three. The following notation is adopted: $\mathfrak{R}(S)$ is the set of items that is positively related to coalition S, $\mathfrak{R}_{\Theta_l}(S)$ is the set of such items in group Θ_l, $\mathfrak{R}^{-1}(\vartheta)$ is the set of coalitions positively related to ϑ and $\mathfrak{R}^{-1}_{\#k}(\vartheta)$ is the set of such coalitions of size k. In the case of a single player coalition we write $\mathfrak{R}(i)$ instead of $\mathfrak{R}(\{i\})$ to avoid cluttering notation.
Note that how we choose to assign relations between players and items is fundamental to obtain relations between coalitions and items: the complexity needed to handle this second set of relations will greatly impact the complexity of the final computation.



S	$\mathfrak{R}(S)$	$\widetilde{\mathfrak{R}}(S)$	$\mathfrak{N}(S)$
$\emptyset$	$\emptyset$	$\emptyset$	$\Theta_{\mathcal{A}}$
$\{ab\}$	$\{\vartheta_a\}$	$\{\vartheta_b\vartheta_c\vartheta_d\}$	$\emptyset$
$\{ac\}$	$\{\vartheta_c\}$	$\{\vartheta_a\vartheta_b\vartheta_d\}$	$\emptyset$
$\{ad\}$	$\{\vartheta_a\vartheta_d\}$	$\{\vartheta_b\}$	$\{\vartheta_c\}$
$\{bc\}$	$\{\vartheta_b\}$	$\{\vartheta_a\vartheta_c\vartheta_d\}$	$\emptyset$
$\{bd\}$	$\{\vartheta_b\}$	$\{\vartheta_c\vartheta_d\}$	$\{\vartheta_a\}$
$\{cd\}$	$\{\vartheta_c\}$	$\{\vartheta_a\vartheta_d\}$	$\{\vartheta_b\}$
$\{abc\}$	$\emptyset$	$\Theta_{\mathcal{A}}$	$\emptyset$
$\{abd\}$	$\{\vartheta_a\}$	$\{\vartheta_b\vartheta_c\vartheta_d\}$	$\emptyset$
$\{acd\}$	$\{\vartheta_c\}$	$\{\vartheta_a\vartheta_b\vartheta_d\}$	$\emptyset$
$\{bcd\}$	$\{\vartheta_b\}$	$\{\vartheta_c\vartheta_d\}$	$\{\vartheta_a\}$
$\{abcd\}$	$\emptyset$	$\Theta_{\mathcal{A}}$	$\emptyset$

$$M_{\mathcal{A}} = (\Theta_{\mathcal{A}}, \{\Theta_1, \Theta_2, \Theta_3\}, \mathfrak{R}, \widetilde{\mathfrak{R}}, \mathfrak{N}, f, g)$$

Figure 3.2: Running example of an argumentation game: model $M_{\mathcal{A}}$ of the characteristic function of $\mathcal{A}$

Example 3.1.2

In our running example on argumentation games we can define an item for every argument, so $\Theta_{\mathcal{A}} = \{\vartheta_i \colon i \in N\}$. The value of every item can be set to $f(\vartheta_i) = 1$ for all $i \in N$, as each one of them represents an argument and they contribute 1 to the value of a coalition if such argument is not attacked by another one in the opinion/coalition.
The partition $\{\Theta_1, \ldots, \Theta_{h(\mathcal{A})}\}$ of $\Theta_{\mathcal{A}}$ can be set to the one made of singletons, as we are not sure of the presence of elements that behave exactly in the same way in the general case, despite the fact that they all have the same value. The relevance of this choice will be justified in the next paragraph, as the constraints for the class $\mathscr{M}$ are introduced. So $h(\mathcal{A}) = |N|$ and $\Theta_l = \{\vartheta_l\}$ for all $l \in \{1, \ldots, h(\mathcal{A})\}$.
We will define relations for every argument such that the items positively related to a coalition correspond to the arguments not attacked by another one in the opinion/coalition. Thus, for every $i \in N$

$$\begin{aligned}
\mathfrak{R}(i) &= \begin{cases} \{\vartheta_i\} & \text{if } \{i\} \cap \mathcal{P}(i) = \emptyset \\ \emptyset & \text{otherwise} \end{cases}, \\
\widetilde{\mathfrak{R}}(i) &= \{\vartheta_j \colon j \in \widetilde{\mathcal{P}}(i)\}, \\
\mathfrak{N}(i) &= \{\vartheta_j \colon j \neq i \wedge j \notin \widetilde{\mathcal{P}}(i)\},
\end{aligned}$$

which is equivalent to saying that every argument is in a positive relation with itself if it does not attack itself, it is in a negative relation with all arguments it attacks and in a neutral relation with all arguments different from itself and that it does not attack. Consequently, relations between coalitions and items form as defined by Equations 3.1:

$$\begin{aligned}
\mathfrak{R}(S) &= \{\vartheta_i \colon i \in S \text{ and is not attacked by any } j \in S\}; \\
\widetilde{\mathfrak{R}}(S) &= \{\vartheta_i \colon i \in N \text{ and is attacked by some } j \in S\}; \\
\mathfrak{N}(S) &= \{\vartheta_i \colon i \notin S \text{ and is not attacked by any } j \in S\}.
\end{aligned}$$

All these elements are shown in Figure 3.2 for the argumentation game in Figure 3.1. Note that in this specific example the partition of $\Theta_{\mathcal{A}}$ isn't made of singletons. This will be discussed after the next two definitions, together with the meaning of $g(|S|)$ and of the model $M_{\mathcal{A}}$.

Definition 5 (Model of a characteristic function). *Given a coalitional game $\mathcal{G} = (N, \nu)$, where N is the set of players and $\nu \colon 2^N \to \mathbb{R}$ is the characteristic function, a model of ν is a tuple*

$$M_{\mathcal{G}} = (\Theta_{\mathcal{G}}, \{\Theta_1, \ldots, \Theta_{h(\mathcal{G})}\}, \mathfrak{R}, \widetilde{\mathfrak{R}}, \mathfrak{N}, f, g),$$

where $\{\Theta_1, \ldots, \Theta_{h(\mathcal{G})}\}$ is a partition of a set $\Theta_{\mathcal{G}}$ of items, $\mathfrak{R}$, $\widetilde{\mathfrak{R}}$ and $\mathfrak{N}$ are the positive, negative and neutral association relations, respectively, and $f \colon \Theta_{\mathcal{G}} \to \mathbb{R}$ and $g \colon \mathbb{N} \to \mathbb{R}$ are real-valued function, such that for all coalitions $S \subseteq N$ the following property holds:

$(M1)$ $\nu(S) = g(|S|) \sum_{\vartheta \in \mathfrak{R}(S)} f(\vartheta)$.

For simplicity, we assume that, if $\mathfrak{R}(S) = \emptyset$, $\nu(S) = 0$. Because we want to group similar items together using the partition, we focus on models for which a number of structural regularity constraints hold:

$(M2)$ $\forall_{1\leq l\leq h}\forall_{\vartheta_i,\vartheta_j\in\Theta_l}$: $f(\vartheta_i) = f(\vartheta_j)$;

$(M3)$ $\forall_{1\leq l\leq h}\forall_{\vartheta_i,\vartheta_j\in\Theta_l}$: $|\mathfrak{N}^{-1}_{\#k}(\vartheta_i)| = |\mathfrak{N}^{-1}_{\#k}(\vartheta_j)|$;

$(M4)$ $\forall_{1\leq l\leq h}\forall_{\vartheta_i,\vartheta_j\in\Theta_l}$: $|\mathfrak{R}^{-1}_{\#k}(\vartheta_i)| = |\mathfrak{R}^{-1}_{\#k}(\vartheta_j)|$.

If these properties hold, with a slight abuse of notation we will use the part symbol Θ_l in place of its elements $\vartheta_i \in \Theta_l$ in $f(\cdot)$, $\mathfrak{R}^{-1}_{\#k}(\cdot)$, $\mathfrak{N}^{-1}_{\#k}(\cdot)$ and $\widetilde{\mathfrak{R}}^{-1}_{\#k}(\cdot)$. Note that these properties do not limit the expressiveness of the model, they are a convention to group similar items together. However the criterion by which value is assigned to coalitions in the original game will affect the number of groups in the partition of items, impacting the computational complexity of semivalue solution concepts. For example a partition that contains only singletons will always satisfy these properties,but it may be prohibitively large.

Definition 6 (The class $\mathscr{M}$). *$\mathscr{M}$ is the class of all characteristic functions of TU games such that for all $\nu \in \mathscr{M}$, and for every game $\mathcal{G} = (N, \nu)$, there exist a model $M_{\mathcal{G}}$ of ν such that $M_{\mathcal{G}}$ has the properties $(M1)$, $(M2)$, $(M3)$ and $(M4)$.*

Example 3.1.3

The model for a generic argumentation game of the type we considered can be immediately built from the elements we introduced in the last part of the example. As we can see now, we intentionally defined the partition and the relations in order to submit to the constraints for $M_{\mathcal{A}}$ to be a model of $\mathscr{M}$. It can be easily seen that the way we defined relations allows us to verify property $(M1)$, while properties $(M2)$, $(M3)$ and $(M4)$ are always verified by partitions made of singletons.
In the example in Figure 3.2, we can group ϑ_2 and ϑ_3 because for all k it holds $|\mathfrak{N}^{-1}_{\#k}(\vartheta_2)| = |\mathfrak{N}^{-1}_{\#k}(\vartheta_3)|$ and $|\mathfrak{R}^{-1}_{\#k}(\vartheta_2)| = |\mathfrak{R}^{-1}_{\#k}(\vartheta_3)|$, which are the constraints needed for the characteristic function of a game to be in the model class $\mathscr{M}$. This is not strictly necessary, but it has the purpose of showing an example of a model not having the partition just made of singletons.

3.2 The Polynomial Subclass $\mathscr{M}^*$ of $\mathscr{M}$

For any characteristic function ν and semivalue $\pi^{\mathbf{p}}$, we define the problem SEMIVALUE$(\nu, \pi^{\mathbf{p}})$ to compute the semivalue $\pi^{\mathbf{p}}$ of a player i:

SEMIVALUE$(\nu, \pi^{\mathbf{p}})$

Given: Game $\mathcal{G} = (N, \nu)$, player $i \in N$

Problem: Compute $\pi^{\mathbf{p}}_i(\nu)$

Definition 7 (The subclass $\mathscr{M}^*$ of $\mathscr{M}$). *$\mathscr{M}^*$ is the subclass of characteristic functions in $\mathscr{M}$ for which additionally the following properties hold:*

$(M1^*)$ $\exists_{n\in\mathbb{N}}\forall_{\mathcal{G}}\exists_{M_{\mathcal{G}}}$: *$M_{\mathcal{G}}$ models ν, and $h(\mathcal{G}) \leq O(|N|^n)$;*

$(M2^*)$ $\exists_{n\in\mathbb{N}}\forall_{\mathcal{G}}$ *exists a model* $(\Theta_{\mathcal{G}}, \{\Theta_1, \ldots, \Theta_{h(\mathcal{G})}\}, \mathfrak{R}, \widetilde{\mathfrak{R}}, f, g)$ *of* ν *such that the function that for each* $i \in N$, k, l *returns the values* $f(\Theta_l)$, $g(k)$, $|\mathfrak{R}_{\Theta_l}(i)|$, $|\widetilde{\mathfrak{R}}_{\Theta_l}(i)|$, $|\mathfrak{N}^{-1}_{\#k}(\Theta_l)|$ *and* $|\mathfrak{R}^{-1}_{\#k}(\Theta_l)|$ *is computable in time* $O(|N|^n)$.

Theorem 1. *For all* $\nu \in \mathscr{M}^*$ *and semivalues* $\pi^{\boldsymbol{p}}$, SEMIVALUE$(\nu, \pi^{\boldsymbol{p}})$ *is solvable in polynomial time.*

Algorithm 1: (SEMI) The semivalue single point solution

Input: Game $G = (N, \nu)$, player $i \in N$, vector of probabilities $\mathbf{p}$, functions $c^{\mathfrak{N}}_{\mathcal{G}}$ and $c^{\mathfrak{R}}_{\mathcal{G}}$, precomputed vectors $\{|\mathfrak{R}_{\Theta_l}(\{i\})| \colon 1 \leq l \leq h(\mathcal{G}) \wedge i \in N\}$, and $\{|\widetilde{\mathfrak{R}}_{\Theta_l}(\{i\})| \colon 1 \leq l \leq h(\mathcal{G}) \wedge i \in N\}$

Output: semivalue, $\pi^{\mathbf{p}}_i(\nu)$, for the player i

$\pi^{\mathbf{p}}_i \leftarrow 0$;
for $k \leftarrow 0$ **to** $|N| - 1$ **do**
  $\mathrm{MC}_k \leftarrow 0$;
  for $l \leftarrow 0$ **to** $h(\mathcal{G})$ **do**
    // ⋆⋆⋆⋆⋆⋆⋆⋆⋆[$\mathfrak{R}(i, \vartheta)$ and $\mathfrak{N}(S, \vartheta)$]⋆⋆⋆⋆⋆⋆⋆⋆⋆
    $|\mathfrak{N}^{-1}_{\#k}(\Theta_l)| \leftarrow c^{\mathfrak{N}}_{\mathcal{G}}(\vartheta \in \Theta_l, k)$;
    $\mathrm{MC}^{[1]} \leftarrow g(k+1) f(\Theta_l) |\mathfrak{N}^{-1}_{\#k}(\Theta_l)|$;
    $\mathrm{MC}_k \leftarrow \mathrm{MC}_k + |\mathfrak{R}_{\Theta_l}(\{i\})| \mathrm{MC}^{[1]}$;
    // ⋆⋆⋆⋆⋆⋆⋆⋆⋆[$\widetilde{\mathfrak{R}}(i, \vartheta)$ and $\mathfrak{R}(S, \vartheta)$]⋆⋆⋆⋆⋆⋆⋆⋆⋆
    $|\mathfrak{R}^{-1}_{\#k}(\Theta_l)| \leftarrow c^{\mathfrak{R}}_{\mathcal{G}}(\vartheta \in \Theta_l, k)$;
    $\mathrm{MC}^{[2]} \leftarrow g(k) f(\Theta_l) |\mathfrak{R}^{-1}_{\#k}(\Theta_l)|$;
    $\mathrm{MC}_k \leftarrow \mathrm{MC}_k - |\widetilde{\mathfrak{R}}_{\Theta_l}(\{i\})| \mathrm{MC}^{[2]}$;
    // ⋆⋆⋆⋆⋆[$\mathfrak{R}(i, \vartheta)$ or $\mathfrak{N}(i, \vartheta)$, and $\mathfrak{R}(S, \vartheta)$]⋆⋆⋆⋆⋆
    $\mathrm{MC}^{[3]} \leftarrow \big(g(k+1) - g(k)\big) f(\Theta_l) |\mathfrak{R}^{-1}_{\#k}(\Theta_l)|$;
    $\mathrm{MC}_k \leftarrow \mathrm{MC}_k + |\mathfrak{R}_{\Theta_l}(\{i\}) \cup \mathfrak{N}_{\Theta_l}(\{i\})| \mathrm{MC}^{[3]}$;
  $\pi^{\mathbf{p}}_i \leftarrow \pi^{\mathbf{p}}_i + p_k \mathrm{MC}_k$;

Algorithm 1 is a direct implementation of the algorithm used to compute SEMIVALUE$(\nu, \pi^{\mathbf{p}})$ in the proof of Theorem 1. Here $c^{\mathfrak{N}}_{\mathcal{G}}$ and $c^{\mathfrak{R}}_{\mathcal{G}}$ are the functions that respectively compute $|\mathfrak{N}^{-1}_{\#k}(\Theta_l)|$ and $|\mathfrak{R}^{-1}_{\#k}(\Theta_l)|$ in polynomial time. $\pi^{\mathbf{p}}_i(\nu)$ is computed using

$$\begin{aligned}\pi^{\mathbf{p}}_i(\nu) &= \sum_{0\leq k<|N|} p_k \binom{|N|-1}{k} \mathbb{E}[\mathrm{MC}(S_k, i)] \\ &= \sum_{0\leq k<|N|} p_k \sum_{1\leq l\leq h(\mathcal{G})} \mathrm{MC}(k, i, \Theta_l),\end{aligned}$$

where S_k is the random variable of all possible coalitions of size k drawn with uniform probability from the set $N \setminus \{i\}$, $\mathbb{E}[\cdot]$ is the expected value operator and $\mathrm{MC}(k, i, \Theta_l)$ is the sum of marginal contributions of i through all $\vartheta \in \Theta_l$ to all coalitions of size k.

In the central part of the computation are pointed out three sections corresponding to the three mutually exclusive ways in which a node i can contribute a (possibly negative) value to a k-sized coalition S through $\vartheta \in \Theta_l$.

Corollary 1. *If $\nu \in \mathscr{M}^*$, then there exist constants m_1, m_2 and m_3 such that:*

1. *f can be computed in time $O(|N|^{m_1})$;*
2. *g can be computed in time $O(|N|^{m_2})$;*
3. *$|\mathfrak{N}^{-1}_{\#k}(\Theta_l)|$, $|\mathfrak{R}^{-1}_{\#k}(\Theta_l)|$, $|\mathfrak{R}_{\Theta_l}(i)|$ and $|\widetilde{\mathfrak{R}}_{\Theta_l}(i)|$ can be computed in time $O(|N|^{m_3})$.*

For a given TU game $\mathcal{G} = (N, \nu)$ and player i, Algorithm 1 runs in time

$$O(h(\mathcal{G})|N|^{m_1} + |N|^{m_2+1} + h(\mathcal{G})|N|^{m_3+1}).$$

Example 3.2.1

Finally we can discuss of the complexity of the elements of the generic model for the argumentation games we considered, in order to see if it is in the class $\mathscr{M}^*$. We will do this analysis considering the more complex case of a partition of $\Theta_{\mathcal{A}}$ not exclusively made of singletons. Obviously $h(\mathcal{A}) \leq |N|^n$, as $h(\mathcal{A})$ is less than $|N|$. So property $(M1)^*$ is satisfied. Moreover, f and g can be computed in $O(1)$, being constant functions.
Note that $|\mathcal{P}(i)| = |\mathcal{P}(j)|$ for all $\vartheta_i, \vartheta_j \in \Theta_l$ and all $l \in \{1, \ldots, h(\mathcal{A})\}$, as it holds $|\widetilde{\mathfrak{R}}^{-1}_{\#1}(\vartheta_i)| = |\widetilde{\mathfrak{R}}^{-1}_{\#1}(\vartheta_j)|$, because of properties $(M3)$ and $(M4)$, and, for definition, $|\widetilde{\mathfrak{R}}^{-1}_{\#1}(\vartheta_i)|$ is exactly $|\mathcal{P}(i)|$. Introducing a slight abuse of notation, we will use $|\mathcal{P}(\Theta_l)|$ to indicate the number of arguments attacking any argument corresponding to an item of Θ_l. Moreover, because of $(M4)$, either all arguments corresponding to items in any Θ_l attack themselves or none does.
To keep the complexity as low as possible, we need to precompute the incidence matrix of the graph $\langle N, \mathcal{R}\rangle$ and $|\mathcal{P}(\Theta_l)|$ for all $l \in \{1, \ldots, h(\mathcal{A})\}$ so that they can be accessed in constant time. They can be computed in $O(|N|^2)$ time each. Finally, we can mark which Θ_l contain only items corresponding to arguments that attack themselves. Then, for all $i \in N$

$$|\mathfrak{R}_{\Theta_l}(i)| = \begin{cases} 1 & \text{if } \vartheta_i \in \Theta_l \wedge (i,i) \notin \mathcal{R} \\ 0 & \text{otherwise} \end{cases},$$

$$|\widetilde{\mathfrak{R}}_{\Theta_l}(i)| = |\{j \colon \vartheta_j \in \Theta_l \wedge (i,j) \in \mathcal{R}\}|,$$

and for all $l \in \{1, \ldots, h(\mathcal{A})\}$

$$|\mathfrak{R}^{-1}_{\#k}(\Theta_l)| = \begin{cases} \binom{|N|-1-|\mathcal{P}(\Theta_l)|}{k-1} & \text{if } k > 0 \wedge \forall \vartheta_i \in \Theta_l, (i,i) \notin \mathcal{R} \\ 0 & \text{otherwise} \end{cases},$$

$$|\mathfrak{N}^{-1}_{\#k}(\Theta_l)| = \begin{cases} \binom{|N|-1-|\mathcal{P}(\Theta_l)|}{k} & \text{if } \forall \vartheta_i \in \Theta_l, (i,i) \notin \mathcal{R} \\ \binom{|N|-|\mathcal{P}(\Theta_l)|}{k} & \text{otherwise} \end{cases}.$$

It is easy to see that $|\mathfrak{R}_{\Theta_l}(i)|$ can be computed in $O(1)$, while $|\widetilde{\mathfrak{R}}_{\Theta_l}(i)|$, $|\mathfrak{R}^{-1}_{\#k}(\Theta_l)|$ and $|\mathfrak{N}^{-1}_{\#k}(\Theta_l)|$ in $O(|N|)$. This verifies property $(M2^*)$ and tells us that this model is in the class $\mathscr{M}^*$. Thanks to Theorem 1 we can say that computing semivalues for it can be done in polynomial time. Using Corollary 1, we can state that we can compute semivalues for argumentation games in $O(|N|^3)$ using Algorithm 1.

3.3 Further Example of the Framework Application: Maintenance Games

Now I will provide an example of application of the aforementioned framework to a different class of games. Let's take maintenance games (Koster 1999), which generalize the simpler airport games.

Maintenance games are built upon the class of maintenance problems. This exposition is mainly based on the one given in (Borm, Hamers, and Hendrickx 2001). The main idea of maintenance problems is the following. A group of players is connected by a fixed network to a certain service provider, for example, by a road network to a community centre. This network is a tree in which the service provider is situated at the root. Each road in this network has some maintenance costs associated with it and the question is how the maintenance cost of the entire network should be divided in a fair way among all users.

Formally, a *maintenance problem* is a triple $(\mathcal{T}, t, N)$ where

- $\mathcal{T} = \langle V, E\rangle$ is a tree with vertex set V and edge set E; the root r has only one adjacent edge;
- $t\colon E \to \mathbb{R}_+$ is a nonnegative cost function on the edges of the tree;
- $N = \{1, \ldots, n\}$ is a finite player set; each player $i \in N$ is located at some vertex $v(i) \in V$ and every vertex in V except the root corresponds exactly to one player.

In order to analyse maintenance problems, I will introduce some more notation. First note that every vertex $v \in V$ is connected to the root of the tree by a unique path P_v (including v itself). We denote by e_v the edge in P_v that is incident on v. The precedence relation $\preccurlyeq$ on V is defined by

$$v' \preccurlyeq v \iff v' \text{ is on the path } P_v.$$

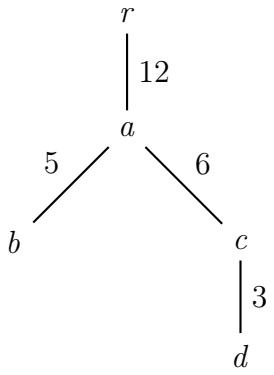
A *trunk* of $\mathcal{T}$ is a set of vertices $R \subset V$ which is closed under the relation $\preccurlyeq$, that is, if $v \in R$ and $v' \preccurlyeq v$, then $v' \in R$. The set of *followers* of player $i \in N$ is given by $F(i) = \{j \in N | v(i) \preccurlyeq v(j)\}$ and the set of *predecessors* by $P(i) = \{j \in N | v(j) \preccurlyeq v(i)\}$. The total cost of a trunk R equals

$$T(R) = \sum_{v \in R \setminus \{r\}} t(e_v).$$

With each maintenance problem $\Gamma = (\mathcal{T}, t, N)$ we associate a *maintenance game* $\mathcal{M} = (N, \nu_\Gamma)$ defined by

$$\nu_\Gamma(S) = \min\{T(R) | v(i) \in R \text{ for all } i \in S \text{ and } R \text{ is a trunk}\}$$

for all $S \subset N, S \neq \emptyset$ and $\nu_\Gamma(\emptyset) = 0$. An example of a very simple maintenance game is shown in Figure 3.3. As I mentioned before, maintenance games are a generalization of airport games, in fact a maintenance game is called airport game when the weighted graph is linear.



S	$\nu_\Gamma(S)$	S	$\nu_\Gamma(S)$	S	$\nu_\Gamma(S)$
$\emptyset$	0	$\{ab\}$	17	$\{abc\}$	23
$\{a\}$	12	$\{ac\}$	18	$\{abd\}$	26
$\{b\}$	17	$\{ad\}$	21	$\{acd\}$	21
$\{c\}$	18	$\{bc\}$	23	$\{bcd\}$	26
$\{d\}$	21	$\{bd\}$	26	$\{abcd\}$	26
		$\{cd\}$	21		

Figure 3.3: Example of a maintenance game: tree and characteristic function

Given this definition we can go on and build a suitable model of the characteristic function in the framework defined in the former section. It is immediate to see that the characteristic function can be rewritten as the sum of the weights of the e_v of all the nodes in the smallest trunk containing the nodes corresponding to the players in S. Therefore we can define $\Theta_\mathcal{M} = \{e_{v(i)} : i \in N\}$ and $f = t$, where the items of $\Theta_\mathcal{M}$ are the edges of the tree graph $\mathcal{T}$. Note that to every edge in the tree $\mathcal{T}$ corresponds exactly one player and vice versa, so $|E| = |N|$. Then we need to select a partition of $\Theta_\mathcal{M}$ and we pick the one made of singletons, so, for all $j \in N$, $\Theta_j = \{e_{v(j)}\}$ and $h(\mathcal{M}) = |N|$. In this way we can immediately state that conditions $(M2)$, $(M3)$ and $(M4)$ are satisfied. After we need to build the neutral, positive and negative relations between players and items and we establish them for all $i \in N$ as such:

$$\begin{aligned}
&\mathfrak{R}(i) = \{e_{v(j)} : j \in P(i)\};\\
&\mathfrak{N}(i) = N \setminus \mathfrak{R}(i);\\
&\widetilde{\mathfrak{R}}(i) = \emptyset.
\end{aligned}$$

Relations between coalitions and items are now easily built using Equations 3.1. In this way each coalition is in a positive relation with every edge that connects its nodes to the root and in a neutral relation with the other edges. Note that the negative relation is empty. These considerations, joint with $g(|S|) = 1$ for all $S \subseteq N$, allows us to rewrite

$$\nu_\Gamma(S) = g(|S|) \sum_{e_v \in \mathfrak{R}(S)} f(e_v).$$

Now the model $M_\mathcal{M} = (\{\Theta_j\}_{j=1}^{h(\mathcal{M})}, \mathfrak{R}, \widetilde{\mathfrak{R}}, \mathfrak{N}, f, g)$ satisfies property $(M1)$ and therefore it puts ν_Γ into the class $\mathscr{M}$. To see if ν_Γ is in $\mathscr{M}^*$ we need to check the computational

complexity of some elements of $M_{\mathcal{M}}$. It's trivial to see that property $(M1^*)$ is satisfied because $h(\mathcal{M}) = |N| \leq O(|N|^n)$. To ease the complexity of the subsequent computations we need to build a navigable tree structure from the tree graph $\mathcal{T}$, in which every node links to its children and to its father. This can be done in $O(|N|^2)$. Then, we can precompute the number of followers of every player in $O(|N|^2)$, by doing a depth-first exploration of the structure we just built and incrementing the counters on the way. Obviously the complexity of computing g is $O(1)$ and the same holds for f, assuming that t is a hash indexed by i. Proceeding to $|\widetilde{\mathfrak{R}}_{\Theta_j}(i)|$ and $|\mathfrak{R}_{\Theta_j}(i)|$, the former can be computed in $O(1)$ because it is always 0, and the latter is 1 only if j is a predecessor of i, 0 otherwise, so it can be decided in $O(|N|)$ by doing a depth-first search of j in the tree and than going on looking for i. Finally $|\mathfrak{N}^{-1}_{\#k}(\Theta_j)| = \binom{|N|-|F(j)|}{k}$ and $|\mathfrak{R}^{-1}_{\#k}(\Theta_j)| = \binom{|N|}{k} - |\mathfrak{N}^{-1}_{\#k}(\Theta_j)|$ need $O(|N|)$ time due to the binomial coefficient. So $(M2^*)$ is satisfied too, and we have proven that ν_Γ belongs to the class $\mathscr{M}^*$, thus we can compute semivalues for the game $\mathcal{M}$ in polynomial time using Algorithm 1. Moreover, using Corollary 1 we can state that it runs in $O(|N|^3)$.

3.4 Faster Algorithms for Argumentation and Maintenance Games

In the previous sections I presented a subclass of argumentation games and the class of maintenance games in order to show examples of application of the generic framework for TU games. Often, the computation of specific semivalues in specific classes of games can be further simplified. So, for completeness, I will present a faster algorithm to compute the Shapley value in both classes.

3.4.1 Subclass of Argumentation Games

In the context of argumentation games, let's consider $\nu(S)$ the characteristic function that measures the value of an opinion $S \subseteq N$ in a graph $\langle N, \mathcal{R}\rangle$, as the number of arguments in it that are not attacked by another argument of the same opinion.

Proposition 1. *Consider an argumentation game $\mathcal{A} = (N, \nu)$ associated with a graph $\langle N, \mathcal{R}\rangle$. Then the Shapley value of player $i \in N$ in the game $\mathcal{A}$ is*

$$\sigma_i(\nu) = \frac{1}{|P(i)| + 1} - \sum_{j \in \widetilde{P}(i)} \frac{1}{|P(j)|(|P(j)| + 1)},$$

where $P(\cdot)$ is the set of the attackers of an argument and $\widetilde{P}(\cdot)$ is the set of arguments it attacks.

It is immediate to see that the complexity of this formula, taken from (Bonzon, Maudet, and Moretti 2014), is bounded by the time needed to count the attackers for every argument, which can be precomputed in $O(|N|^2)$ time by scanning the set of edges $\mathcal{R}$.

3.4.2 Maintenance Games

The formula to compute the Shapley value of a maintenance game is presented in the following theorem (Borm, Hamers, and Hendrickx 2001).

Theorem 2. *Let $\mathcal{M} = (N, \nu_\Gamma)$ be the maintenance game associated with the maintenance problem $\Gamma = (\mathcal{T}, t, N)$. Then, for every $i \in N$, the Shapley value of player i in the game $\mathcal{M}$ is*

$$\sigma_i(\nu_\Gamma) = \sum_{j \in P(i)} \frac{1}{|F(j)|} t(e_{v(j)}),$$

where $t(e_{v(\cdot)})$ is the cost of the edge incident with vertex $v(\cdot)$ in the path from $v(\cdot)$ to the root, and $P(\cdot)$ and $F(\cdot)$ are the sets of predecessors and followers of a player, respectively.

Given that every maintenance game is associated with a tree $\mathcal{T}$, here the computation is bounded by the time needed to count the number of followers of every predecessor of i. To do this we can build a navigable tree structure and then scan it. These computations take $O(|N|^2)$ time each.

3.5 Comparing the class $\mathscr{M}$ to Basic Generalized Additive Games

There are many similarities between the constraints for a characteristic function to be in the class $\mathscr{M}$ and the way a *Basic Generalized Additive Game* (GAG) is defined (Cesari, Lucchetti, and Moretti 2017). I will proceed to briefly explain how Basic GAGs work in order to see if we can compare them with the characteristic functions in $\mathscr{M}$.

3.5.1 Basic Generalized Additive Games

We call *Basic Generalized Additive Situation* (GAS) a triple $(N, w, \mathfrak{C})$ where

- N is a set of players,
- $w\colon N \to \mathbb{R}$ is a map that assigns to each player a real value,
- $\mathfrak{C} = \{\mathfrak{C}_i\}_{i \in N}$ is a collection. Each $\mathfrak{C}_i = \{F_i^1, \dots, F_i^m, E_i\}$ is, in turn, a collection of subsets of N such that $F_i^k \cap E_i = \emptyset$ for all $i \in N$ and for all $k \in \{1, \dots, m\}$.

We call each F_i^k the k-th set of *friends* of i, while E_i is the set of *enemies* of i. Every Basic GAS is associated with coalitional map defined, for each $S \subseteq N$, as:

$$\mathfrak{M}(S) = \{i \in N\colon S \cap F_i^1 \neq \emptyset, \dots, S \cap F_i^m \neq \emptyset, S \cap E_i = \emptyset\}$$

Given a Basic GAS, the associated Basic GAG is defined as the TU-game (N, ν), where

$$\nu(S) = \begin{cases} \sum_{i \in \mathfrak{M}(S)} w(i) & \text{if } \mathfrak{M}(S) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}.$$

A particularly simple case of Basic GAG is when every player has a unique set of friends, that we shall denote by F_i.
So, can we model some characteristic functions of Basic GAGs such that they are in the class $\mathscr{M}$? Vice versa, are games with characteristic function in $\mathscr{M}$, Basic GAGs?

3.5.2 Basic GAGs as elements of $\mathscr{M}$

The idea here is to represent Basic GAGs with the tools provided in the framework used to define $\mathscr{M}$. By looking at the characteristic functions of the two game classes, we see that, ideally, we would like to define a coalitional map using items and relations such that items represent players, and $\mathfrak{R}(S)$ are the items corresponding to the players $\mathfrak{M}(S)$, for all $S \subseteq N$
Let's begin by defining the set of items $\Theta_{\mathcal{G}} = \{\vartheta_1, \dots, \vartheta_n\}$ such that $n = |N|$. In this way, there is a bijective correspondence between items and players. W.l.o.g, we also set the partition of $\Theta_{\mathcal{G}}$ to the one made of singletons. This will ensure that constraints $(M2)$, $(M3)$ and $(M4)$ are satisfied. Next we will set $g(|S|) = 1$ for all $S \subseteq N$ and $f(\vartheta_i) = w(i)$ for all $i \in N$. Now, constraint $(M1)$ is satisfied too, and the last thing we need to do to show which Basic GAGs are in $\mathscr{M}$, is to define relations such that the items $\mathfrak{R}(S)$ correspond to the players $\mathfrak{M}(S)$.
For now, let's consider the simple case of Basic GAG having a single group of friends. We can define relations for all $i \in N$ as

$$\mathfrak{R}(i) = \{\vartheta_j \colon i \in F_j\},$$
$$\widetilde{\mathfrak{R}}(i) = \{\vartheta_j \colon i \in E_j\},$$

where F_j and E_j are the set of friends and the set of enemies of player $j \in N$, respectively. From this, using Equations 3.1, we can build the relations for coalitions. It is easy to see that items in a positive relation with a coalition S correspond to the players mapped by $\mathfrak{M}(S)$. From this directly follows that all Basic GAGs with a single set of friends are in the class $\mathscr{M}$. Provided that, for all $i \in N$, $w(i)$, F_i and E_i can be computed in polynomial time, it can be easily shown that these Basic GAGs are also in $\mathscr{M}^*$.
Now, let's try to generalize that for multiple sets of friends. Are we able to model $\mathfrak{M}(S)$ again using $\mathfrak{R}(S)$ and not changing anything else? Let's take a simple case where there are only two sets of friends and let's assume that we can define relations between players and items such that

$$\mathfrak{R}(S) = \{\vartheta_j \colon \exists i_1 \in S, i_1 \in F_j^1 \wedge \exists i_2 \in S, i_2 \in F_j^2 \wedge \nexists i_3 \in S, i_3 \in E_j\}.$$

This is impossible because, by looking at Equations 3.1, we can see that to apply such constraints for items to be in a positive relation with a coalition, we would need to have something like multiple positive relations between single players and items, one for every set of friends. Then, we could change how the positive relation between coalitions and items is defined to take it into account.
Of course this does not mean that there can't be another way to describe Basic GAGs with multiple sets of friends as elements of $\mathscr{M}$. It would be relevant to find an example of Basic GAG that cannot be described by any means as an element of $\mathscr{M}$, to prove that Basic GAGs are not exactly contained in it.

3.5.3 Elements of $\mathscr{M}$ as Basic GAGs

This is much more difficult to do because the map $\mathfrak{M}(\cdot)$ goes from sets of players to sets of players, while items could represent anything. Moreover we cannot directly represent $g(\cdot)$ or the partition of Θ_G in the context of Basic GAGs. This is an open topic, it would be interesting to find an example to prove that there are elements of $\mathscr{M}$ that cannot be represented as a Basic GAG.

Chapter 4

Computing Semivalues of Weighted Voting Games

Weighted voting games, also called weighted majority games, are a very simple class of coalitional games. They model settings where players cooperate to reach a goal, but every one of them holds a different amount of a given resource. This goal can be reached by any coalition that possesses a sufficient amount of this resource. They are typically used to model voting, for example a parliamentary voting, where players are the parties, and the resource of each party is the number of votes they control. First I will provide some formal definitions on weighted voting games and some examples. Then I will explain an algorithm to compute semivalues for this class of games in pseudopolynomial time, provided that the weights are small. The exposition is mainly based on the one given in (Chalkiadakis, Elkind, and Wooldridge 2011).

4.1 Weighted Voting Games

Let's start by giving by defining a of weighted voting game. Given a set of players $N = \{1, \dots, n\}$, a list of *weights* $\mathbf{w} = (w_1, \dots, w_n) \in \mathbb{R}^n$ and a *quota* $q \in \mathbb{R}$, we call a *weighted voting problem* any triple $(N, \mathbf{w}, q)$.
With each weighted voting problem $\Delta = (N, \mathbf{w}, q)$ we associate a *weighted voting game* $\mathcal{W} = (N, \nu_\Delta)$, defined by

$$\nu_\Delta(S) = \begin{cases} 1 & \text{if } \sum_{i \in S} w_i \geq q \\ 0 & \text{otherwise} \end{cases}$$

It is usually assumed that all weights and the quota are non-negative; in what follows we will make this assumption as well. Also it is standard to assume that $0 < q \leq \sum_{i \in N} w_i$; this condition ensures that the empty coalition is losing and that the grand coalition is winning.
Finally, a player i is said to be *pivotal* to a coalition S if $\nu_\Delta(S) = 0$ and $\nu_\Delta(S \cup \{i\}) = 1$; in other words, a player is pivotal to any coalition that is losing, if, when it is added to the coalition, the coalition is now winning.

Example 4.1.1

Weighted voting games can describe many real world situations. In particular they are very well suited to model coalition formation in legislative bodies, where groups of people take decisions by means of a vote. An example of such setting is when shareholders of a small company need to take a decision on a company matter, typically to approve or reject a proposal. Their influence of each shareholder on the result of this decision corresponds to the number of shares of the company he owns. In this case the quota q can be set to half the total shares of the company plus one, and a proposal will pass if the sum of the number of shares of the people positively voting is equal or more than q.

Example 4.1.2

Another real world example of a weighted majority game is the United States Electoral College, if we consider every state as a player in the election of the President of the United States. The Electoral College consists of 538 electors and an absolute majority of 270 votes is needed to win the election. Every state is entitled to a certain number of electors, which is (almost) proportional to the number of inhabitants of the state. Now, for the sake of this example, let's assume that every elector will vote accordingly to the result of the elections in his state. Then, the weight of every player (i.e., of every state) is the number of electors he has in the Electoral College and the quota of the game is 270. For example California has 55 electors, while Alaska has only 3.

4.2 Pseudopolynomial Algorithm

Computing the Shapley value and the Banzhaf index for weighted voting games is known to be #P-complete (Deng and Papadimitriou 1994; Faliszewski and Hemaspaandra 2009; Y. Matsui and T. Matsui 2001; Prasad and Kelly 1990). However, for small weights, it exists an algorithm (T. Matsui and Y. Matsui 2000) to compute them in pseudopolynomial time. This algorithm is used to prove the following theorem.

Theorem 3. *Given an n-player weighted voting game $\mathcal{W} = (N, \nu_\Delta)$, associated with a weighted voting problem $\Delta = (N, \mathbf{w}, q)$, and a player $i \in N$, we can compute $\beta_i(\nu_\Delta)$ and $\sigma_i(\nu_\Delta)$ in time $O(n^2 w_{\max})$ and $O(n^3 w_{\max})$, respectively.*

where $w_{\max} = \max_{i \in N} w_i$ is the maximum weight between the weights of the players and it's used to bound the complexity, hence classifying the algorithm as pseudopolynomial. The complexity is different because the Shapley value's probability coefficient depends on the coalition size, while the Banzhaf one does not. In fact, in the formula of the Banzhaf index of a generic player i we do not need to sum over the coalition sizes, and we can just compute once the number of coalitions of size at most $|N| - 1$ for which i is pivotal. This heavily depends on how this algorithm work and cannot be generalized to other algorithms. Instead the Shapley value version of the algorithm can be extended to any other semivalue having a probability coefficient that can be computed for every coalition size. For this reason I will present this second version of the algorithm, and then briefly explain how to simplify it for the Banzhaf index.

Algorithm 2: VSemivalue$(n, \mathbf{w}, \mathbf{p})$

```
1  W_sum ← 0
2  for j ← 1 to n do
3      W_sum ← W_sum + w_j
4  for j ← 1 to n − 1 do
5      for W ← 0 to W_sum do
6          if W = 0 then
7              X[j, W, 0] ← 1
8          else
9              X[j, W, 0] ← 0
10         for k ← 1 to n − 1 do
11             if j = 1 then
12                 if W = w_1 ∧ k = 1 then
13                     X[1, W, k] ← 1
14                 else
15                     X[1, W, k] ← 0
16             else
17                 X[j, W, k] ← X[j − 1, W, k] − X[j − 1, W − w_j, k − 1]
18 smv ← 0 for k ← 0 to n − 1 do
19     N_k ← 0
20     for W ← 1 to w_n do
21         N_k ← N_k + X[n − 1, q − W, k]
22     smv ← smv + p_k · N_k
23 return smv
```

The algorithm is VSemivalue$(n, \mathbf{w}, \mathbf{p})$ shown in Algorithm 2, which takes in input $n = |N|$, the vector $\mathbf{p}$ of the semivalue probabilities and the vector $\mathbf{w}$ of the players weights, rearranged in a way such that the last weight w_n is the one of the player i we want to compute the semivalue of. Indirectly, the weight ordering will also give us an ordering of the players such that player i is the last one.
The main idea behind this algorithm is that the computation is bounded by the dimensions of the three-dimensional matrix X, which are $n \times nw_{\max} \times n$. Each $X[j, W, k]$ represents the number of k-elements subsets in the first j players that have weight W. The elements of X are then used to compute every $\mathcal{N}_k$, which is the number of k-elements subsets of $N \setminus \{i\}$ that have weight $W \in \{q - w_n, \ldots, q - 1\}$, that is to say, for which the last player is pivotal.
Finally, VSemivalue$(n, \mathbf{w}, \mathbf{p})$ can be adapted to a faster one for the Banzhaf index if we omit the third index of X. This is due to the fact that to compute the Banzhaf index we only need to know how many subsets of $N \setminus \{i\}$ have weight that is at least $q - w_n$ and at most $q - 1$.

Chapter 5

Query Answering Games on Database Tuples

Databases are an incredible technology, that is central in our modern information-permeated world. For instance, the whole concept of Big Data is built around them and their convenient data manipulation capabilities. The most abstract description we can give of a real world database is that it is a set of tables. Each table is an abstract representation of an object or a concept, for example a person, a car or, in a more abstract fashion, the concept of possessing something. Then, each table has a fixed number of columns and an undefined number of rows. Columns, called attributes, represent properties of the concept represented by the table. For example, a person has a name, an age, a favorite color and the list goes on. Rows, called tuples, are instead instances of that concept. So, the table PERSON will be populated by a list of people and their characteristics.

PERSON

	name	*favorite color*	...
p_1	Aldo	Blue	...
p_2	Benny	Yellow	...
p_3	Carlo	Red	...
p_4	Daria	Blue	...

HAS

	person	*car brand*
h_1	Aldo	Fiat
h_2	Aldo	Mini
h_3	Benny	Bmw
h_4	Carlo	Peugeot

CAR MODEL

	brand	*color*	...
c_1	Bmw	Grey	...
c_2	Fiat	Blue	...
c_3	Lamborghini	Yellow	...
c_4	Mini	Grey	...
c_5	Peugeot	White	...

Figure 5.1: Portion of a database storing information about car owners

In Figure 5.1, we can see an example of a very simple database: it consists of three tables storing people, possible car models and the information of owning one of them. The tables PERSON and CAR MODEL may have more attributes, but they are omitted for simplicity. The tools that allow us to extract meaningful data from databases are the so called query languages: with them we can write queries, also called interrogations, and evaluate them on a database; so, a query can be seen as an actual question we can ask to a database. In practice, queries work as a collection of constraints that trims down the database in order to selectively keep what we are interested in. For example, using the most common query languages we could formulate queries asking which people in the database of Figure 5.1 own a car, if it exists a white car, how many models are not owned by anyone or if it exists someone

who bought a car of his favorite color. In our every day life, every time we use our bank or our university online services, our social networks or, indeed, when we visit any web page, a number of queries is being evaluated in the process. They can be seen as asking 'What is the balance of my account?' or 'Which are my friends on Facebook?' to the respective databases. Of course, nowadays databases are massive, their table structure is really complex and intertwined, and each table contains millions of entries.

Thus, queries can get really articulated too and it is not easy to use them to infer general properties on the information stored in a database, while it would be desirable to do so in many contexts employing Big Data. For instance, we may want to know how relevant is a piece of information, as a table entry, in the context of a database. But, what does it mean to say that a tuple is relevant in the context of a database?

5.1 Quantifying Data Relevance in Databases

Databases, per se, are used to represent and give structure to data, so we could say that they operate more on the syntax of the data and less on the semantics of it. For example, we could look for how connected is a tuple to other ones, but we would be measuring its importance with respect to the structure of the data, which could not reflect its meaning. Queries, on the other hand, are much more semantically charged, as they are usually built from questions expressed in natural language.
So, it makes much more sense to ask ourself 'How relevant is a tuple in the process of answering a query on its database?', which can be rephrased in a less passive way as 'How much does a tuple contribute to the answer of a query?' For instance, considering again the database in Figure 5.1, we could be interested in the contribution of entry p_1 (the person named `Aldo`) in answering the query 'How many people are stored in the database?'[1] To do this, it is reasonable to take the answer to the query - in this case, the number 4 - and to subdivide it between between all tuples, proportionally to their contribution to the answer. In this way, we will represent contribution as a coefficient that is exactly the slice of the query answer that each entry is accountable for. Following this criterion, we will give the quantity $\frac{1}{4} \cdot 4 = 1$ to `Aldo`, as there are four people in the database. Obviously, the same holds for the other three entries of table PERSON. To all the other tuples in the database we will assign contribution 0, as they did not take part by any means in answering the query.
So, this approach can be used to quantify contribution to a query answer and to give credit to entities involved. In a more complex and realistic example, if we take a database storing detailed data about a football tournament, we may want to measure, considering a team, the contribution of each player of the team to the total number of successful ball passes for that team in the whole tournament. In general, given the complexity of the world we live in and the astonishing amount of data we use to try and represent it, this would be a way to shade some light on responsibility matters or cause-consequence relations that we may be unaware of because deeply hidden behind an interweaving of data relations.
For more complex queries, we may want restrict the evaluation of the contribution to a subset of tuples and only share between them the total value. We do this because often we need to consider only some entities' contribution towards the answer of the

[1] In query language this question would be formulated as 'Count all entries of table PERSON'.

query and considering all tuples would give a more complex and less clear result. Thus, we decide which tuples will have active responsibility and give all the contribution credit to them.

Differently from what it can seem, measuring the contribution to a query answer is not a matter of simply counting specific entries in a table and assigning a coefficient proportional to this number. Let's take the query on the database of Figure 5.1 that counts the number of possible groups of three tuples, one for each table, where attribute *name* of PERSON equals attribute *person* of HAS, and attribute *car brand* of HAS equals attribute *brand* of CAR MODEL. It answers the question 'How many cars are owned in total in this database?', and we can easily see that the answer is 4. But let's say that we want to measure how the contribution to the answer is shared between all entries of both table PERSON and table CAR MODEL, considering all of them as separate entities. The answer is much more complex than before and we may not have systematic way to determine it for some tuples yet.
As we did before, we set the quantity to be shared between tuples to the query answer, namely, the number 4. We can immediately see that the contribution of p_4 (`Daria`) and c_3 (`Yellow Lamborghini`) can be set to 0, as they add nothing to the answer of the query. So, we still need to divide the value 4 between p_1, p_2, p_3, c_1, c_2, c_4 and c_5. For example, we could say that p_2 (`Benny`) and p_3 (`Carlo`) should have the same coefficient because they both contribute with one car, and that p_1 (`Aldo`) should be assigned a coefficient which is double the one assigned to p_2 and p_3, because he owns two cars, but not much more. The answer heavily depends by the structure of the database and of the query, and becomes more and more complex as we try to apply this approach to realistic databases.

A solution to this problem was proposed in *The Shapley Value of Tuples in Query Answering* (Livshits et al. 2020). First of all, what they do is to adopt a formal mathematical model to represent databases. One of the most important mathematical foundations of databases is relational algebra, which was explicitly designed to model databases and queries. Then, their approach to the problem is to apply Cooperative Game Theory to relational databases and queries. Given a database and a query, they define a cooperative game such that:

- the set of players is the set of tuples of which we want to measure the contribution to the query answer;
- the value of each coalition is somehow related to the answer of the query evaluated on the tuples constituting the coalition.

By doing that, the contribution of a tuple to the query answer can be computed as its semivalue in such game, as it is a coefficient modeling the power (contribution, in this case) of players (tuples) on the game outcomes (to the answer of the query).
Unluckily, the problem now shifts to the complexity of computing semivalues, which is notoriously high. In (Livshits et al. 2020) it is proposed a polynomial algorithm the Shapley value of a tuple in the aforementioned game, under a number of constraints on the query. As it will be explained later, the query must be a Boolean[2] Conjunctive

[2]A query is called boolean when it returns true or false, yes or no: that is, when it tests the presence in the database of a set of tuples complying with its constraints.

Query which is self join less and hierarchical. The algorithm is fundamental, as its correctness is also used to prove the tractability part of a theorem (here reported as Theorem 4), which is the main result of the paper. In fact, such theorem allows to define a dichotomy in (i.e., a full classification of) complexity on the computation of the Shapley value with regard to a Boolean Conjunctive Query without self joins, and is later used to define a similar dichotomy with regard to the more elaborate aggregate-relational queries.

In the following section I will give a brief introduction on the formalities of relational databases and queries. Then I will show how we can build a cooperative game on them, together with a presentation of the algorithm proposed in (Livshits et al. 2020). I will proceed to explain why the algorithm they proposed will not correctly compute the Shapley value in some cases by providing an example. Subsequently, I will propose two possible changes to make the algorithm work and correctly validate Theorem 4. Then, I will rewrite the algorithm accordingly, generalizing it to any semivalue.

5.2 Relational Databases and Queries

A relational database D is a collection of $facts$. Facts $R(c_1, \ldots, c_k)$ represent the tuples of the database: each fact contains a vector of constants $\vec{c}$, which is the data of the represented tuple, and the information of which table the tuple belongs to, in the form of a relation symbol R. We assume a countably infinite set Const of constants that are used as database values. The vector $\vec{c} \in \mathsf{Const}^k$ holds its constants in the same order of the data inside the tuple it represents, so it is the closest thing to the tuple itself. The only lacking information is the table it belongs to, together with the number of attributes of such table. This is supplied by the relation symbol, which has an associated arity (i.e., a cardinality) k too, that constraints the arity of its vectors of constants, in the same way a table would set the number of attributes for its tuples. Relational databases are defined over a relational schema $\mathcal{S}$, which is a collection of relation symbols determining which ones can be present in the facts of the database: that is to say, which tables are in the database. If we look at the database of Figure 5.2, the constant vectors are all the rows of the tables and the relation symbols are the table names, while the notation f_j^{i} is used to refer more easily to facts. For example, f_2^{a} denotes the fact Album(`At Last!`,1960), where Album is the relation symbol, which has arity 2, and (`At Last!`,1960) is the vector of constants, which has 2 elements due to the arity.
Now, as we seen before, we need to define which facts, that we previously called tuples, we will consider to have active responsibility towards query answers and, therefore, we will limit the computation to. To do this, we partition D into two sets: D_{n} is the set of *endogenous* facts - namely, the players - and D_{x} is the set of *exogenous* facts. This means that we are interested in - i.e., we will only consider - the responsibility of endogenous facts in the answer of a query on the database D. Keep in mind that we can choose endogenous facts to be any arbitrary subset of all facts in the database, as they depend on what we are looking for: for example, we may only want to know if either f_1^{m} or f_5^{a} is more influential towards a query answer, and so set only these two facts to endogenous. For the sake of simplicity and without loss of generality, in our examples we will define as endogenous, for each table (i.e., each relation), all facts in

SONG (endo)

	name	*artist*	*album*
f_1^{s}	A Big Hunk O' Love	Presley, E.	A Big Hunk O' Love
f_2^{s}	A Sunday Kind of Love	James, E.	At Last!
f_3^{s}	Blue Rondo à la Turk	Brubeck, D.	Time Out
f_4^{s}	C Jam Blues	Peterson, O.	Night Train
f_5^{s}	Compared to What	Harris, E.	Swiss Movement
f_6^{s}	Giant Steps	Coltrane, J.	Giant Steps
f_7^{s}	Take Five	Brubeck, D.	Time Out

MUSICIAN (exo)

	name	*instrument*
f_1^{m}	Brubeck, D.	Piano
f_2^{m}	Coltrane, J.	Saxophone
f_3^{m}	Harris, E.	Saxophone
f_4^{m}	James, E.	Voice
f_5^{m}	Peterson, O.	Piano
f_6^{m}	Presley, E.	Voice

ALBUM (exo)

	name	*year*
f_1^{a}	A Big Hunk O' Love	1959
f_2^{a}	At Last!	1960
f_3^{a}	Giant Steps	1960
f_4^{a}	Night Train	1963
f_5^{a}	Time Out	1959

Figure 5.2: A simple database storing information about some beautiful songs

it, or none of them. For instance, in Figure 5.2 the endogenous facts D_{n} are all the elements of table SONG.

Our analysis will focus on Boolean Conjunctive Queries. They are a special case of relational queries that take the form

$$q() \text{ :- } R_1(\vec{t_1}), \ldots, R_n(\vec{t_n}),$$

where q is the query, R_i are relation symbols taken from the schema $\mathcal{S}$ and each $\vec{t_i}$ is a vector of variables and constants of the same arity of R_i. They are called boolean because they can only model questions that imply a yes/no, or a true/false answer. So, when the answer is positive, we denote with $D \models q$ that the boolean query q is satisfied by the database D. We call $q()$ the *head* of q and $R_1(\vec{t_1}), \ldots, R_n(\vec{t_n})$ the *body* of q. In general, the head of the query would contain some variables that represent which data we want to extract from the database. In Boolean CQ, the head is empty to indicate that the query just checks some constraints on the database and returns true if they are verified. These constrains are enforced by atoms.
Each $R_i(\vec{t_i})$ is an *atom* and it is used to evaluate some constraints on facts that have the same relation symbol R_i. When it is evaluated on a fact, an atom is true if they have the same relation symbol and if the vector of variables and constants $\vec{t_i}$ has the same structure of the vector of constants $\vec{c}$ of the fact. Obviously, $\vec{t_i}$ and $\vec{c}$ have the same cardinality, as they are both constrained by R_i. They also have the same structure if, for each constant $c \in \mathsf{Const}$ of $\vec{t_i}$, c appears in $\vec{c}$ in the same place it appears in $\vec{t_i}$, and, for each variable x of $\vec{t_i}$, $\vec{c}$ has the same constant in each place x appears in $\vec{t_i}$. Bear in mind that different variables can correspond to the same constant and verify an atom, as they are not limited in this sense. For example, atom $\text{SONG}(x, y, z)$ is satisfied by every fact in table SONG of the database in Figure 5.2,

atom $\textsc{Song}(x, y, x)$ is satisfied by f_1^{s} and f_6^{s}, and atom $\textsc{Song}(x, \texttt{Peterson, O.}, y)$ is only satisfied by f_4^{s}.
Every variable that occurs in the body of the query is existentially quantified and is called *existential variable*. This type of relational query is called conjunctive because the atoms in the body are in conjunctive form, so the query it is true if and only if all atoms are. The answer to q on a database D is true if there exists a homomorphism from q to D, false otherwise. A homomorphism from q to D is a mapping of the variables in q to the constants of D, such that every atom in q is mapped to a fact in D. Note that an atom with no variables can be considered a fact, as they have exactly the same structure. So, less formally, a Boolean CQ is true if we can assign a constant to each variable in its body and the set of facts we can obtain from its new atoms is contained in the database.

Example 5.2.1

Let us consider the following boolean queries on the database of Figure 5.2.

$$q_1() \text{ :- } \textsc{Song}(x, y, x), \textsc{Album}(x, w)$$
$$q_2() \text{ :- } \textsc{Song}(x, y, w), \textsc{Musician}(y, \texttt{Piano})$$
$$q_3() \text{ :- } \textsc{Song}(\texttt{C Jam Blues}, x, y), \textsc{Song}(\texttt{Take Five}, z, y)$$
$$q_4() \text{ :- } \textsc{Song}(x, y, w), \textsc{Musician}(y, z), \textsc{Album}(w, k)$$

Query q_1 is true if in the database there exists a song that gives the name to the album it comes from and such album is present in the database. For example, it is satisfied by the couple of facts f_6^{s} and f_3^{a}, as they have the same value `Giant Steps` in all the positions corresponding to variable x in the atoms of the query. Note that in this case y and w could assume any value, as they appear only once and, therefore, they are not constrained. Query q_2 checks if in the database there is at least one song of which the artist plays piano. Query q_3 tests if songs `C Jam Blues` and `Take Five` come from the same album and it is not verified. Finally, query q_4 is satisfied if there is at least a song with complete information about its album and its artist.

We will see that it can exist a polynomial algorithm to compute the Shapley value of a tuple in a game built on Boolean Conjunctive Query answering if the query is self join free and hierarchical. A Conjunctive query q is *self join free* if every relation symbol occurs at most once in the body, i.e., there are not two atoms referring to the same table. A Conjunctive Query q is *hierarchical* if, for all couples of existential variables in the body of q, either one of the two is contained in all the atoms containing the other, or they are not contained in a single atom at the same time. More formally, let A_x be the set of all atoms of q containing variable x, q is hierarchical if, for all its existential variables x and x' it holds either $A_x \subseteq A_{x'}$, or $A_{x'} \subseteq A_x$, or $A_x \cap A_{x'} = \emptyset$. In Example 5.2.1 queries q_1, q_2 and q_4 are self join less, while q_3 has a self join on the relation Song. Moreover, all of them are hierarchical but q_4, because not all the atoms that contain variable y contain w too or vice versa, and there exists one atom which contains both.

5.3 Query Answering Games

Until now we have only formally defined Boolean Conjunctive Queries, which are just a subset of all possible Conjunctive Queries. So, before giving a formal definition to the cooperative game we mentioned in the introduction of this chapter, we need to make a brief premise on query answers in general, as they can be of two types: a list of tuples or a number. In the first case, queries are used to selectively extract information from databases, in the form of a list of tuples created with the data selected by the query. With the database in Figure 5.2, this happens if we ask for the names of all musicians who play saxophone: the query answer will be a list of two one-dimensional tuples containing `Coltrane, J.` and `Harris, E.` In the second case, they map a number to a database, in order to extract a piece of information not explicitly written in it. For instance this is done by counting facts with, in the position of an attribute, a certain value we may be interested in, but in general it could be any function on the database that outputs a number. These queries are called *numerical* and we will denote them with the symbol α.
Many numerical queries are built on top of non-numerical ones, as they first extract a list of tuples and then they map it to a number. For example, we can ask how many musicians play the saxophone in the database of Figure 5.2: this query will be obtained by counting the tuples in the answer of the last one. Now that we made this distinction, it is easy to see that measuring the contribution of a fact to a query answer makes only sense for numerical queries. As a consequence, we can only build games on them, as they provide us with a quantitative measure that we will use as characteristic function. Moreover, all Conjunctive Queries are monotonic, in the sense that the answer can only grow in size or stay the same if we add facts to the database. Therefore, numerical queries are monotonic too.
Finally, it is important to specify that boolean queries are usually not considered numerical. Formally, they are non-numerical queries that return an empty tuple () if they are satisfied and nothing if they are not. Without loss of generality, we can always identify them with the numerical queries that associate 1 to the empty tuple, and 0 otherwise. For the sake of our study, from now on we will always do so.

Now, in order to quantify the contribution of facts to numerical query answers by using semivalues, we need to define a suitable cooperative game.

Definition 8 (Query Answering Game). *Given a relational schema $\mathcal{S}$, let D be a relational database over $\mathcal{S}$ partitioned into endogenous facts D_n and exogenous facts D_x, and let α be a numerical query over $\mathcal{S}$. Then we call query game the couple $\mathcal{D} = (D_\mathsf{n}, \nu)$, where D_n is the set of players of $\mathcal{D}$ and ν is its characteristic function, defined, for all $S \subseteq D_\mathsf{n}$, as*

$$\nu(S) = \alpha(S \cup D_\mathsf{x}) - \alpha(D_\mathsf{x}).$$

Thus, the value of a coalition in a query answering game $\mathcal{D}$ is the partial contribution of the endogenous facts in such coalition to the result of the query. In this way the value of the grand coalition equals the contribution of all endogenous facts and the value of the empty coalition is always 0, even in the special case where the query is satisfied by exogenous facts only. Note that a query answering game defined on

boolean query is a simple game if and only if the query is satisfied at least by the whole database.

As it can be easily seen, query answering games are TU cooperative games and so we can compute, for example, their Shapley Value. Specifically, we will use $\text{Shapley}(D, \alpha, f)$ to denote the Shapley Value of fact f in the query answering game defined on query α and database D.

Example 5.3.1

Take the database D in Figure 5.2 and suppose that we want to count the number of songs that appeared in an album released in 1959. So, even if we did not formally define numerical queries, the body of the one corresponding to this interrogation of the database will look something like $\text{Song}(x, y, w)$, $\text{Album}(w, \texttt{1959})$. We will denote it with α_1. The answer to this query is obviously 3, as it is satisfied by songs `Blue Rondo à la Turk` and `Take Five`, paired with the fact corresponding to their album `Time Out`, and by `A Big Hunk O' Love` together with its homonymous album. We appointed as endogenous the facts of table Song, so the query answering game on α_1 will assign a value ranging from 0 to 3 to each coalition according to the number of songs satisfying α_1 contained in it. In this case the Shapley Value of f_1^{s}, f_3^{s} and f_7^{s} will be 1 and it will be 0 for every other fact of table Song, as their addition to any coalition will never change the characteristic function.
Things change if we say that the facts of the table Album are endogenous too. We may want to do this if we are interested in measuring the contribution of albums to the number of songs released in 1959, together with the contribution of songs themselves. The main difference from before is that the weight of an album will be proportional to the number of its songs. Now for a coalition to have higher utility it is necessary to include both one of the three aforementioned songs together with its album. First of all, the Shapley Value of all endogenous facts different from f_1^{s}, f_3^{s}, f_7^{s}, f_1^{a} and f_5^{a} equals zero, as they can be considered null players. Let's begin by computing the Shapley Value of f_5^{a}. When it is added to a coalition, f_5^{a} can change the value of the query, and therefore the utility of the coalition, in two distinct cases: only one of f_3^{s} or f_7^{s} is already present, or both of them already are. In the first case the marginal contribution of f_5^{a} to the utility will be 1 and in the second it will be 2. By rigorous counting, we can see that the number of such coalitions of size 0 is 0 in both cases, of size 1 is 2 and 0, of size $2 \leq k \leq 10$ is $2 \cdot \binom{9}{k-1}$ and $\binom{9}{k-2}$, and of size 11 is 0 and 1, respectively. Then, for each size, we can merge these results in the formula for the Shapley Value, by multiplying the number of coalitions to their marginal contribution, which is 1 in the first case and 2 in the second. Now, every component of the sum is multiplied by 2, so we can bring it to the front.

$$\begin{aligned}&\text{Shapley}(D, \alpha_1, f_5^{\text{a}}) = \\ &= 2 \cdot \left(1 \cdot \frac{10!1!}{12!} + \sum_{k=2}^{10} \left[\binom{9}{k-1} + \binom{9}{k-2} \right] \frac{(12-k-1)!k!}{12!} + 1 \cdot \frac{0!11!}{12!} \right).\end{aligned}$$

Therefore, if we join the binomial coefficients using the Pascal's Rule, we can compute the Shapley Value of f_5^{a} using the following formula.

$$\text{Shapley}(D, \alpha_1, f_5^{\mathrm{a}}) = 2 \cdot \sum_{k=1}^{11} \binom{10}{k-1} \frac{(12-k-1)!k!}{12!} = 1.$$

With a similar reasoning, we can also show that the Shapley Value of f_3^{s}, f_7^{s}, f_1^{a} and f_1^{s} is $\frac{1}{2}$. These results were foreseeable as this is a pretty easy scenario, but things gets much more complex when facts are more intertwined and, thus, it is difficult to count all the coalitions with a specific value.

Example 5.3.2

Now, suppose we want to compute the Shapley Value of facts for query the q_1, one of the boolean queries introduced in Example 5.2.1. This query checks if in the database in Figure 5.2 is present at least one song that gives the name to its album. So, q_1 is satisfied by the endogenous facts f_1^{s} and f_6^{s}, coupled with f_1^{a} and f_3^{a}, respectively. So, the Shapley Value of every other fact inside table SONG equals 0. Similarly to what we did in the last example, their Shapley Value can be computed as

$$\sum_{k=0}^{5} \binom{5}{k} \frac{(7-k-1)!k!}{7!} = \frac{1}{2}.$$

In general, the only difference with boolean queries is that the Shapley Values of facts will sum to 1, as it is the utility of the grand coalition.

From now on we will go back to focusing our analysis on Boolean Conjunctive Queries, as the complexity results that will follow are achieved for them. These result can be then generalized to few specific classes of numerical queries, but we will not elaborate further on this matter.

5.4 An Efficient Algorithm for the Shapley Value of Boolean Query Answering Games

At this point we are able to compute the contribution of a fact to a query answer using semivalues of query answering games. But, as we already know, this computation is exponential if we use the definition of semivalue, which is considered intractable for non trivial cases. So, which factors can lower the complexity of this computation? This problem was studied in (Livshits et al. 2020) with relevant results. The main one is that they were able to define dichotomy in the *data complexity* of the problem of computing the Shapley Value with regard to a Boolean Conjunctive Query without self joins, as it is stated in the following theorem.

Theorem 4. *Let q be a Boolean CQ without self joins. If q is hierarchical, then* $\text{Shapley}(D, q, f)$ *can be computed in polynomial time, given D and f. Otherwise the problem is* $\mathrm{FP}^{\#\mathrm{P}}$*-complete.*

Evaluating the data complexity in the context of relational databases means that we are assuming the dimension of the query to be constant and much smaller than the dimension of the database, therefore we can only consider the latter. In the case of self join less queries, the number of atoms of the query is limited by the number of relation symbols of the schema the database is defined on, making the assumption true in most cases.

To prove the tractability side of Theorem 4, the authors reduce the computation of $\text{Shapley}(D, q, f)$ to the problem of counting the coalitions of size k that, along with the exogenous facts, satisfy q. Specifically, the reduction is to the problem of computing $|\text{Sat}(D, q, k)|$ where $\text{Sat}(D, q, k)$ is the set of all coalitions S of D_n such that $|S| = k$ and $D_\mathsf{x} \cup S \models q$. The reduction is given by the following formula, where $n = |D_\mathsf{n}|$ and D' is the same set as D, except that f is viewed as exogenous instead of endogenous.

$$\begin{aligned}
&\text{Shapley}(D, q, f) = \\
&= \sum_{S \subseteq (D_\mathsf{n} \setminus \{f\})} \frac{|S|!(n - |S| - 1)!}{n!} \Big(q(D_\mathsf{x} \cup S \cup \{f\}) - q(D_\mathsf{x} \cup S) \Big) \\
&= \sum_{k=0}^{n-1} \frac{k!(n-k-1)!}{n!} \Big(|\text{Sat}(D', q, k)| - |\text{Sat}(D \setminus \{f\}, q, k)| \Big)
\end{aligned}$$

In practice, we are grouping the possible coalitions not containing f, i.e., $S \subseteq (D_\mathsf{n} \setminus \{f\})$, depending on their size. Then, in each of these groups, we are computing the number of coalitions of fixed size k for which the utility of the query answering game associated with q changes from 0 to 1 with the addition of f. Note that this is similar to how we computed the Shapley Value in the examples of the last section. Finally, to compute this number we are counting all the coalitions of size k which are satisfied after we add f, and to them we are subtracting the number of such coalitions that were already satisfied without f.
So, in order to prove Theorem 4 it is only left to prove the following theorem.

Theorem 5. *Let q be a hierarchical Boolean CQ without self joins. There is a polynomial-time algorithm for computing the number $|\text{Sat}(D, q, k)|$ of subsets S of D_n such that $|S| = k$ and $D_\mathsf{x} \cup S \models q$, given D and k.*

To establish Theorem 5 the authors provide the polynomial algorithm depicted in Subroutine 1, named $\mathsf{CntSat}(D, q, k)$. Here $\mathsf{Vars}(q)$ indicates the existential variables in the query q and $\mathsf{Atoms}(q)$ its atoms. This subroutine is recursive and it has three main parts that correspond to the only three mutually exclusive scenarios that can happen with hierarchical queries and their variables (Dalvi and Suciu 2007).
The computation reaches lines 1-7 only if there are no variables in the query q and therefore there are only constants. It directly computes and returns the number of coalitions of size k in the input database D that satisfy q.
From line 9 to line 22, it is computed the case that in the query is present a root variable, namely, a variable that occurs in every atom. Here, all values that the variable can assume are extracted from the database and, for each one of them, it is created a new database D^{a_i} containing all facts of the original database D that map the value to the variable. Then, for each value a_i of the root variable, it is recursively

Subroutine 1: $\mathsf{CntSat}(D, q, k)$

1 **if** $\mathsf{Vars}(q) = \emptyset$ **then**
2 **if** $\exists a \in \mathsf{Atoms}(q)$ s.t. $a \notin D$ **then**
3 **return** 0
4 $A = \mathsf{Atoms}(q) \cap D_{\mathsf{n}}$
5 **if** $k < |A|$ or $k > |D_{\mathsf{n}}|$ **then**
6 **return** 0
7 **return** $\binom{|D_{\mathsf{n}}|-|A|}{k-|A|}$
8 result $\leftarrow 0$
9 **if** q has a root variable that occurs in all atoms **then**
10 $x \leftarrow$ a root variable of q
11 $V_x \leftarrow$ the set $\{a_1, \ldots, a_n\}$ of values for x
12 $D^{a_i} \leftarrow D_{[x \to a_i]}$ for all $i \in \{1, \ldots, n\}$
13 **for all** $i \in \{1, \ldots, |V_x|\}$ **do**
14 **for all** $j \in \{0, \ldots, k\}$ **do**
15 $f_{i,j} \leftarrow \mathsf{CntSat}(D^{a_i}, q_{[x \to a_i]}, j)$
16 $P_1^l = f_{1,l}$ for all $l \in \{0, \ldots, k\}$
17 **for all** $i \in \{2, \ldots, |V_x|\}$ **do**
18 **for all** $l \in \{0, \ldots, k\}$ **do**
19 $P_i^l \leftarrow 0$
20 **for all** $j \in \{0, \ldots, l\}$ **do**
21 $P_i^l \leftarrow P_i^l + P_{i-1}^{l-j} \cdot f_{i,j} + \left[\binom{\sum_{r=1}^{i-1} |D_{\mathsf{n}}^{a_r}|}{l-j} + P_{i-1}^{l-j}\right] \cdot f_{i,j} + P_{i-1}^{l-j} \cdot \left[\binom{|D_{\mathsf{n}}^{a_i}|}{j} - f_{i,j}\right]$
22 result $\leftarrow P_n^k$
23 **else**
24 let $q = q_1 \wedge q_2$ where $\mathsf{Vars}(q_1) \cap \mathsf{Vars}(q_2) = \emptyset$
25 let D^1 and D^2 be the restrictions of D to the relations of q_1 and q_2, respectively
26 **for all** k_1, k_2 s.t. $k_1 + k_2 = k$ **do**
27 result $\leftarrow$ result $+ \mathsf{CntSat}(D^1, q_1, k_1) \cdot \mathsf{CntSat}(D^2, q_2, k_2)$
28 **return** result

computed the number of coalitions in D^{a_i} satisfying $q_{[x \to a_i]}$, that is the query obtained by substituting a_i in place of the root variable. Once this is done for each value, the results are joined with dynamic programming. Here each $f_{i,j}$ is the number of j-sized coalitions in $D_{\mathsf{n}}^{a_i}$ that satisfy q when joint with $D_{\mathsf{x}}^{a_i}$. Each P_i^l is the number of l-sized coalitions in $\bigcup_{r=1}^{i} D_{\mathsf{n}}^{a_r}$ that satisfy q when joint with $\bigcup_{r=1}^{i} D_{\mathsf{x}}^{a_r}$. This second part will be analyzed in depth in the following sections.
Finally, in the last part going from line 23 to line 27, it is computed the case of having two distinct subqueries with no shared variables. In this case the query and the database are both split into two subsets. Then, the procedure is recursively called on each half for every coalition size and the results are joined. The correctness and

Algorithm 3: $\mathsf{QShapley}(D, q, f)$

1 $shv \leftarrow 0$
2 **for all** $k \in \{1, \ldots, |D_\mathsf{n}| - 1\}$ **do**
3 $\quad shv \leftarrow shv + k! \cdot (|D_\mathsf{n}| - k - 1)! \cdot [\mathsf{CntSat}(D', q, k) - \mathsf{CntSat}(D \setminus \{f\}, q, k)]$
4 **return** $\frac{1}{|D_\mathsf{n}|!} \cdot shv$

efficiency of CntSat is stated in the following lemma.

Lemma 1. *Let q be a hierarchical Boolean CQ without self joins. Then, $\mathsf{CntSat}(D, q, k)$ returns the number $|\mathrm{Sat}(D, q, k)|$ of subsets S of D_n such that $|S| = k$ and $D_\mathsf{x} \cup S \models q$, given D and k. Moreover, $\mathsf{CntSat}(D, q, k)$ terminates in polynomial time in k and $|D|$.*

For completeness, the reduction part of the polynomial computation of the Shapley value is presented by the authors in Algorithm 3, called $\mathsf{QShapley}(D, q, f)$.

As we already discussed at the beginning of this chapter, this result is very important because it allows us to quantify the contribution of each fact/tuple in the answer of a query on the database. Furthermore, we can easily generalize it for any semivalue as we only need to change the vector of probabilities by multiplying another coefficient other than the one of the Shapley Value for every coalition size in Algorithm 3. Unfortunately, the procedure of Subroutine 1 will not correctly compute in a specific class of cases. It is very important to fix these issues, as an unaware person trusting the algorithm may happen to use these indices of responsibility to support his decisions. If this is the case, wrong computations may have heavy consequences on the external world.

5.4.1 Identified Issues in the Algorithm

Fist of all, the loop on k in line 2 of $\mathsf{QShapley}(D, q, f)$ should start from zero instead of one, because otherwise the empty coalitions in which the query is satisfied by exogenous facts will not be accounted in the computation. For example this happens when f satisfies the query in coalition with some exogenous facts and we compute $\mathsf{CntSat}(D', q, k)$ at line 3. Remember that D' is the same database as D but f is considered exogenous instead of endogenous. This could just be a distraction error because in the mathematical formula the algorithm $\mathsf{QShapley}(D, q, f)$ is based upon, the loop is written correctly.

The main issue of the algorithm lies in the subroutine $\mathsf{CntSat}(D, q, k)$, specifically in the lines 11-12. This recursive subroutine computes these lines when there is a root variable in the query: that is, when there is a variable in the query that occurs at least once in all atoms. In line 11 it is built V_x, defined by the authors as the set $\{a_1, \ldots, a_n\}$ of values the root variable x is mapped to in some homomorphism from an atom in q to the database D. An atom in q is mapped to a fact $f \in D$ in an homomorphism if they both have the same relation symbol - namely, the table name - and the tuple of values of f comply with the structure of the tuple of variables and constants in the atom. This means that, if in a position of the atom there is a constant, f must have the same value of the constant in such position, and that f must have

the same value in all the positions corresponding to the positions of a same variable in the atom (e.g., atom $S(x, y, x, \texttt{Mario})$ is mapped to fact $S(\texttt{Alice}, \texttt{41}, \texttt{Alice}, \texttt{Mario})$). In line 12 it is built a new database D^{a_i} for every element of V_x: D^{a_i} is defined as the database that contains every fact $f \in D$ that maps the variable x to the value a_i in a homomorphism from the atom corresponding to the relation of f to the fact f. In other words, given a value a_i of V_x, for every atom of the query we search the corresponding table of the database for facts that have a_i in the same positions in which, in the atom, there is the root variable x

The issue in question arises if the root variable occurs more than once in an atom and in the database D is present a fact of the relation corresponding to that atom in which two or more different values are in the same positions of the root variable in the atom (e.g. given the atom $R(x, y, x)$, the fact $R(\texttt{a}, \texttt{b}, \texttt{c})$ has the different values `a` and `c` in the positions of x). If this is the case, the aforementioned fact won't provide any value to be added to V_x because there is no homomorphism mapping the atom to the fact, and the fact itself won't be added to any database D^{a_i}. This will lead to an incorrect computation of the number of coalitions satisfying the query, because coalitions of the input database D containing such fact may satisfy the query as well, and not be counted towards the result. For example this happens when a coalition contains a minimal sub-coalition satisfying q together with some other facts just needed to reach the current coalition size.

I will use three examples to show this second issue in practice, so from now on I will assume that the sum in QShapley starts from 0. To understand them it is crucial to keep in mind the meaning of each $f_{i,j}$ and P_i^l in the algorithm:

- $f_{i,j}$ is the number of j-sized coalitions in $D_{\mathsf{n}}^{a_i}$ that satisfy q when joint with $D_{\mathsf{x}}^{a_i}$;
- P_i^l is the number of l-sized coalitions in $\bigcup_{r=1}^{i} D_{\mathsf{n}}^{a_r}$ that satisfy q when joint with $\bigcup_{r=1}^{i} D_{\mathsf{x}}^{a_r}$.

T

A	B
1	1
5	6

U

A
1

Figure 5.3: The database D of example 5.4.1

Example 5.4.1

This is a minimal example in which the mentioned issue arises. Let's take query $q() \text{ :- } T(w, w), U(w)$ and database D of Figure 5.3, where all of its three facts are considered endogenous. We can immediately see that the query is satisfied by the couple $T(1, 1), U(1)$ alone. The consequence of this is that the Shapley Value of a game defined on D and q will be $\frac{1}{2}$ for $T(1, 1)$ and $U(1)$, and 0 for $T(5, 6)$.

Now, let's compute $\mathsf{QShapley}(D, q, f)$, with $f = T(1,1)$. For each $k \in \{0,1,2\}$, it will call the procedure $\mathsf{CntSat}(D', q, k)$, where D' is the same as D but $T(1,1)$ is exogenous, and the procedure $\mathsf{CntSat}(D \setminus \{f\}, q, k)$, where $T(1,1)$ is removed from D. As the only variable w is also a root variable, the subroutine always executes from line 10 in these calls.
The only value for w that will be put into V_x in line 11 is 1. In fact, in each one of the calls there is an homomorphism mapping $U(w)$ to $U(1)$ and in this mapping w assumes value 1. There are no other values because there is no homomorphism mapping $T(w,w)$ to $T(5,6)$, as there is not the same value in each position of the fact corresponding to a position of w in the atom. As a consequence, in line 12, the algorithm will create a single sub-database D^1, which contains both $T(1,1)$ and $U(1)$ in the calls where $T(1,1)$ is exogenous, or just $U(1)$ in the calls where $T(1,1)$ is removed from the database D.

Now we can skip to line 22, where it will set to P_1^k the variable to be returned. We do this because we know the meaning of the computation in lines 13-21, and so we can just infer the value of P_1^k for every k by looking at D^1.

k	0	1	2	the coalition size
P_1^k	0	1	0	if $T(1,1)$ is exogenous
P_1^k	0	0	0	if $T(1,1)$ is removed

Returning these values to $\mathsf{QShapley}$, we see that the whole algorithm will return $1 \cdot \frac{1!1!}{3!} = \frac{1}{6}$, which is not the Shapley Value of $T(1,1)$.

T

A	B
1	1
2	2
3	3
5	6

U

A
1
2
3
4

Figure 5.4: The database D of Example 5.4.2

Example 5.4.2

I will now show the issue again by running just CntSat on a more complex example. Here D is the database in Figure 5.4, where all facts are endogenous, and, as before, q is the query $q() \text{ :- } T(w,w), U(w)$. I will illustrate the subroutine wrongly computing the value of $\mathsf{CntSat}(D, q, 3)$: that is, the number of coalitions of size 3 of the database D that satisfy the query q. Here D is the database in Figure 5.4, where all facts are endogenous, and q is the query $q() \text{ :- } T(w,w), U(w)$.

I choose this example because it the same provided by the authors[3] to explain the first part of the algorithm execution, from line 1 to line 7. To do this they had to do all the steps in lines 9-12, as these are the first lines executed when the subroutine $\mathsf{CntSat}(D, q, 3)$ is called. The authors state that there is no homomorphism from $T(w, w)$ to the fact $T(\mathtt{5}, \mathtt{6})$; hence, the values $\mathtt{5}$ and $\mathtt{6}$ are not in V_w. The set of values built at line 11 is $V_w = \{\mathtt{1}, \mathtt{2}, \mathtt{3}, \mathtt{4}\}$, thus four new databases are created in line 12:

$$D^1 = \{T(\mathtt{1}, \mathtt{1}), U(\mathtt{1})\} \quad D^3 = \{T(\mathtt{3}, \mathtt{3}), U(\mathtt{3})\}$$
$$D^2 = \{T(\mathtt{2}, \mathtt{2}), U(\mathtt{2})\} \quad D^4 = \{U(\mathtt{4})\}$$

The execution proceeds computing, for each one of the four values in V_w, and for each $j \in \{0, \ldots, 3\}$, the number $f_{i,j}$ using the recursive call $\mathsf{CntSat}(D^{a_i}, q_{[w \to a_i]}, j)$, where $q_{[w \to a_i]}$ is the query obtained from q by substituting a_i for w.

$$f_{1,0} = 0 \quad f_{1,1} = 0 \quad f_{1,2} = 1 \quad f_{1,3} = 0$$
$$f_{2,0} = 0 \quad f_{2,1} = 0 \quad f_{2,2} = 1 \quad f_{2,3} = 0$$
$$f_{3,0} = 0 \quad f_{3,1} = 0 \quad f_{3,2} = 1 \quad f_{3,3} = 0$$
$$f_{4,0} = 0 \quad f_{4,1} = 0 \quad f_{4,2} = 0 \quad f_{4,3} = 0$$

Next, it is computed, for each $i \in \{1, \ldots, 4\}$ and for every $l \in \{0, \ldots, 3\}$, the number P_i^l of coalitions of size l in $\bigcup_{r=1}^{i} D_{\mathsf{n}}^{a_r}$ that satisfy q.

$$P_1^0 = 0 \quad P_1^1 = 0 \quad P_1^2 = 1 \quad P_1^3 = 0$$
$$P_2^0 = 0 \quad P_2^1 = 0 \quad P_2^2 = 2 \quad P_2^3 = 4$$
$$P_3^0 = 0 \quad P_3^1 = 0 \quad P_3^2 = 3 \quad P_3^3 = 12$$
$$P_4^0 = 0 \quad P_4^1 = 0 \quad P_4^2 = 3 \quad P_4^3 = 15$$

In the algorithm they are computed using dynamic programming, but we can obtain them trivially in simple examples. For instance, P_3^3 is the number of coalitions of size 3 taken from $D^1 \cup D^2 \cup D^3$ that satisfy the query q. This can be checked by seeing that in this joint database there are three couples of facts satisfying q (e.g., one of them is $T(1, 1), U(1)$) and that for each one of them we can add one of the other four facts to reach coalition size 3. Therefore, $P_3^3 = 3 \cdot 4 = 12$.
So $\mathsf{CntSat}(D, q, 3)$ returns P_4^3, which is equal to 15, but the number of subsets of size 3 of D that satisfy the query is equal to 18. This is due to the fact $T(\mathtt{5}, \mathtt{6})$ not being considered in the computation of the number of possible subsets of size 3 that satisfy q.

[3]For reference, see the third section of Example 26 in *The Shapley Value of Tuples in Query Answering* (Livshits et al. 2020), at the end of page 22.

Example 5.4.3

Let's run the algorithm on query $q_1() :\text{-} \textsc{Song}(x, y, x), \textsc{Album}(x, w)$ of Example 5.2.1, as it falls into the case of the main issue and we already know the Shapley Value of every fact. For instance, let's compute $\mathsf{QShapley}(D, q_1, f_6^{\mathrm{s}})$. As the number of recursive calls is very high and the issue lies in the central part of CntSat only when the root variable is x, we will analyze these cases alone.
Every time CntSat is called by QShapley, which is twice for every coalition size $k \in \{0, \ldots, 6\}$, x is the root variable. The set V_x will contain all the names of the albums as they are the values a_i that x is mapped to in the second atom of the query. So, the algorithm will produce a database for each value a_i. Most of these databases will contain only the fact corresponding to the album. The exceptions are the database corresponding to the value `A Big Hunk O' Love`, that will always contain the song too, and the one corresponding to the value `Giant Steps`, that will contain the song fact f_6^{s}, considered exogenous, only in the first call for every coalition size k.
Now, for each k, we need to compute P_n^k both when f_6^{s} is an exogenous fact of the database and when it is removed from it. Bear in mind that P_n^k is the number of k-sized subsets of the database obtained from joining all databases obtained from the values a_i. Such database will consist of all facts corresponding to albums together with f_6^{s}, if we are in the case in which it is exogenous, and the fact corresponding to the song `A Big Hunk O' Love`. This last fact will always be the only endogenous one, so P_n^k will always be 0 for $k \in \{2, \ldots, 6\}$. Furthermore, if f_6^{s} is exogenous, P_n^k will be 1 for both $k = 0$ and $k = 1$, as in the first case the query is verified by the empty set of endogenous facts and in the second it is verified by the single endogenous fact we mentioned. For the same reasons, in the cases where f_6^{s} is removed from the database, P_n^k will be 1 when $k = 1$, and 0 if $k = 0$.
The subroutine will now return these last values and the computation will proceed back from QShapley. The only time that the difference $\mathsf{CntSat}(D', q, k) - \mathsf{CntSat}(D \setminus \{f\}, q, k)$ is not 0 is for $k = 0$. So the algorithm will return the value

$$\frac{0! \cdot 6!}{7!} \cdot 1 = \frac{1}{7}.$$

This is clearly wrong, as we already established that the Shapley Value of f_6^{s} in this game is $\frac{1}{2}$, and the cause is the same issue I pointed out before.

These examples show that the subroutine $\mathsf{CntSat}(D, q, k)$ does not always return the number of subsets of size k of the endogenous part of D that, joined with all the exogenous facts, satisfy q, differently from what the authors claim in Lemma 1. Furthermore, this leads to an incorrect computation of the Shapley value in the algorithm $\mathsf{QShapley}(D, q, f)$, because these "phantom" endogenous facts are still counted in the total number of endogenous facts, that influences the coefficient $(k! \cdot (|D_{\mathsf{n}}| - k - 1)!)$ and the possible values that k can assume in the sum.
This invalidates the proof of Theorem 5 and, thus, the proof of tractability side of Theorem 4. Note that, if we counted beforehand the number of "phantom" endogenous facts, we could adjust $|D_{\mathsf{n}}|$ accordingly and $\mathsf{QShapley}(D, q, f)$ would correctly compute. This is due to the efficiency property of the Shapley value joint with the null player

status of the "phantom" facts, that allows us to remove a null player without changing the Shapley value of the other players.

5.4.2 Proposed Changes

As I already explained the loop in $\mathsf{QShapley}(D, q, f)$ should start from 0. Another simple quality of life change I propose is to include a check for the emptiness of V_x that directly returns result after line 11, because the indexing of the following loops has no meaning in this case. This is just because people who want to implement the algorithm do not need to find it out the hard way. For example V_x can be empty when the procedure is called from line 27 and one of the two databases is empty.

Now I will explain two possible modifications that can be either applied to make the algorithm correctly compute the Shapley value:

Correction 1 (Incorrect Homomorphisms). My first proposal of correction would be to change the way V_x is built by using maps that are not a perfect homomorphism from an atom in q to the database D. Here every atom of q is mapped to every fact with a relation corresponding to that atom, and none is excluded. In this map the root variable is only tied to values in the same position of its first occurrence in the atom. This means that in a fact with the relation corresponding to the atom in question, we will only add to V_x values in the position of the first time the root variable appears in the atom. For example if the atom is $T(w, w)$ and in the database there are the facts $T(1, 1)$ and $T(5, 6)$ we would add the values 1 and 5. This map gives the useful property $\bigcup_{i=1}^{|V_x|} D^{a_i} = D$ to the databases that are created in line 12, allowing the subroutine to consider every coalition and thus compute correctly. Moreover, the correct evaluation of the query on these new databases is guaranteed, because facts that are not exactly mapped by the atom of q in the original homomorphism, as $T(5, 6)$, are never in a minimal coalition satisfying the query, as $q_{[x \to a_i]}$ cannot be satisfied by them.

Correction 2 (Leftovers Set). A second way to rectify the algorithm, keeping the way the homomorphism between an atom in q and D is defined in origin, would be to append a set L of leftover facts to the list of the D^{a_i} databases. In this set we need to put facts that have the relation symbol corresponding to an atom of q but do not match with the same value every occurrence of the root variable in the atom (e.g. $T(5, 6)$). In this way $L \cup \bigcup_{i=1}^{|V_x|} D^{a_i} = D$. L is treated as a further database with $f_{|V_x|+1,j} = 0$ for every $j \in \{1, \ldots, k\}$. Now to make the procedure work we only need to change $|V_x|$ to $|V_x| + 1$ in line 16 and P_n^k to P_{n+1}^k in line 22.

Algorithm 4: $\mathsf{QSemivalue}(D, q, f, \mathbf{p})$

```
1 smv ← 0
2 for all k ∈ {0, ..., |D_n| − 1} do
3     smv ← smv + p_k · [CntSatLeftovers(D', q, k) − CntSatLeftovers(D \ {f}, q, k)]
4 return smv
```

$\mathsf{CntSatLeftovers}(D, q, k)$ written in Subroutine 2 is a direct implementation of the Leftovers correction of the last paragraph. I choose it because I find it more adherent

to the original idea of the subroutine, as it uses the homomorphism as defined by the authors. Finally, $\mathsf{QSemivalue}(D, q, f, \mathbf{p})$ in Algorithm 4 is the original reduction to $\mathsf{CntSatLeftovers}(D, q, k)$ adapted to any semivalue solution concept. Here p_k is a component of the vector of probabilities $\mathbf{p}$ associated with the semivalue. A Python implementation of this algorithm is written in Appendix B.

Subroutine 2: $\mathsf{CntSatLeftovers}(D, q, k)$

if $\mathsf{Vars}(q) = \emptyset$ **then**
  if $\exists a \in \mathsf{Atoms}(q)$ s.t. $a \notin D$ **then**
    return 0
  $A = \mathsf{Atoms}(q) \cap D_{\mathsf{n}}$
  if $k < |A|$ or $k > |D_{\mathsf{n}}|$ **then**
    return 0
  return $\binom{|D_{\mathsf{n}}|-|A|}{k-|A|}$
result $\leftarrow 0$
if q has a root variable that occurs in all atoms **then**
  $x \leftarrow$ a root variable of q
  $V_x \leftarrow$ the set $\{a_1, \ldots, a_n\}$ of values for x
  if $|V_x| = 0$ **then**
    return result
  $D^{a_i} \leftarrow D_{[x \to a_i]}$ for all $i \in \{1, \ldots, n\}$
  $D^{a_{n+1}} \leftarrow D \setminus \bigcup_{i=1}^{n} D^{a_i}$
  for all $i \in \{1, \ldots, |V_x|\}$ **do**
    for all $j \in \{0, \ldots, k\}$ **do**
      $f_{i,j} \leftarrow \mathsf{CntSat}(D^{a_i}, q_{[x \to a_i]}, j)$
  for all $j \in \{0, \ldots, k\}$ **do**
    $f_{n+1,j} \leftarrow 0$
  $P_1^l = f_{1,l}$ for all $l \in \{0, \ldots, k\}$
  for all $i \in \{2, \ldots, |V_x| + 1\}$ **do**
    for all $l \in \{0, \ldots, k\}$ **do**
      $P_i^l \leftarrow 0$
      for all $j \in \{0, \ldots, l\}$ **do**
        $P_i^l \leftarrow P_i^l + \binom{\sum_{r=1}^{i-1} |D_{\mathsf{n}}^{a_r}|}{l-j} \cdot f_{i,j} + P_{i-1}^{l-j} \cdot \left[\binom{|D_{\mathsf{n}}^{a_i}|}{j} - f_{i,j}\right]$
  result $\leftarrow P_{n+1}^k$
else
  let $q = q_1 \wedge q_2$ where $\mathsf{Vars}(q_1) \cap \mathsf{Vars}(q_2) = \emptyset$
  let D^1 and D^2 be the restrictions of D to the relations of q_1 and q_2, respectively
  for all k_1, k_2 s.t. $k_1 + k_2 = k$ **do**
    result $\leftarrow$ result $+ \mathsf{CntSat}(D^1, q_1, k_1) \cdot \mathsf{CntSat}(D^2, q_2, k_2)$
return result

Chapter 6

Conclusions

In this work, a generic framework to check the computational complexity of semivalues for a variety of TU cooperative games is reviewed. This framework was first proposed in (Tarkowski et al. 2018): their work was focused on relevant network centrality measures that can be obtained as semivalues of cooperative games defined on graphs. But such framework is surprisingly general and it can be used to model other cooperative games too. Therefore, I explained how it can be used in the context of the whole class of TU cooperative games. This is done in two steps: first we have to show that a game can be modeled in such framework and, only then, we can use the game model to check if the preconditions for the polynomiality of the computation are satisfied. If we are able to prove this for a game, we are also provided with a polynomial algorithm ready to be used to compute semivalues for such game. Despite the expressiveness of this framework, it may not model every possible game and it is generally more suitable to be used with games defined on graphs. Furthermore, the algorithm has the nice property of being very general, but in some specific games it may not be the fastest way to compute semivalues. That said, this framework remains very useful in a wide number of situations thanks to its generality. Subsequently, the class of Basic Generalized Additive Games is considered and compared to the aforementioned framework, as they share many similarities. I showed that we can model the simplest case of Basic GAG in the framework and then I briefly argued that, beyond doing that, comparing the two classes is a much more complex problem.
Next, an interesting pseudopolynomial algorithm is presented for weighted voting games: by putting an upper bound on the maximum weight a player is allowed to have, we can polynomially compute semivalues for this important game class, a problem that is known to be intractable in the general case.
Lastly, it is reviewed a recent polynomial algorithm to compute the Shapley Value of games defined on hierarchical self-join less Boolean Conjunctive Queries on relational databases. I showed that the algorithm as presented by its original authors in (Livshits et al. 2020) does not correctly compute the Shapley Value in a specific case. Then I proposed some corrections and I presented a working version of the algorithm, generalizing it to any semivalue.

On the topic of tractable computation of semivalues, many questions remain for future investigations. Limiting to the contents of this work, it would be interesting to further study which classes of games can be represented in the framework of Chapter 3, with the aim of obtaining a full classification of such games. To do this, it will be

relevant to compare the framework with similarly general game classes, such as Basic GAGs. For example, it may be useful to look for examples of games in the Basic GAGs class that cannot be modeled in the above framework and vice versa.
Regarding weighted voting games, they could be used to study the complexity of the problem for the entire class of simple games. As shown in (Chalkiadakis, Elkind, and Wooldridge 2011), any simple game can be represented by a vector weighted voting game, which is basically a list of weighted voting games. So, if we are to find a tractable way to compute semivalues for a vector weighted voting game from the semivalues of its sub-games, we could use the results on weighted voting games to define a general pseudopolynomial algorithm for any simple game.
Finally, while we have a thorough understanding of the complexity for Boolean Conjunctive Queries without self joins, very little is known in the presence of self joins. It will be important to study the complexity of the problem for this kind of Conjunctive Queries, and which constraints we can use to bound classes of queries for which the problem of computing semivalues is tractable.

Bibliography

Aumann, Robert J and Roger B Myerson (2003). "Endogenous formation of links between players and of coalitions: An application of the Shapley value". In: *Networks and Groups*. Springer, pp. 207–220 (cit. on p. 2).

Aziz, Haris and Bart de Keijzer (2013). "Shapley meets shapley". In: *arXiv preprint arXiv:1307.0332* (cit. on p. 2).

Bonzon, Elise, Nicolas Maudet, and Stefano Moretti (2014). "Coalitional games for abstract argumentation." In: *COMMA*, pp. 161–172 (cit. on pp. 12, 20).

Borm, Peter, Herbert Hamers, and Ruud Hendrickx (2001). "Operations research games: A survey". In: *Top* 9.2, p. 139 (cit. on pp. 18, 21).

Branzei, Simina (2010). "State of the Art: Solution Concepts for Coalitional Games". In: *BRAIN. Broad Research in Artificial Intelligence and Neuroscience* 1.2, pp. 89–101 (cit. on p. 1).

Cesari, Giulia, Roberto Lucchetti, and Stefano Moretti (2017). "Generalized additive games". In: *International Journal of Game Theory* 46.4, pp. 919–939 (cit. on p. 21).

Chalkiadakis, Georgios, Edith Elkind, and Michael Wooldridge (2011). *Computational aspects of cooperative game theory*. Vol. 5. Synthesis Lectures on Artificial Intelligence and Machine Learning 6. Morgan & Claypool Publishers, pp. 1–168 (cit. on pp. 25, 48).

Dalvi, Nilesh and Dan Suciu (2007). "The dichotomy of conjunctive queries on probabilistic structures". In: *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pp. 293–302 (cit. on p. 38).

Deng, Xiaotie and Christos H Papadimitriou (1994). "On the complexity of cooperative solution concepts". In: *Mathematics of operations research* 19.2, pp. 257–266 (cit. on p. 26).

Faliszewski, Piotr and Lane Hemaspaandra (2009). "The complexity of power-index comparison". In: *Theoretical Computer Science* 410.1, pp. 101–107 (cit. on p. 26).

Grant, John and Anthony Hunter (2006). "Measuring inconsistency in knowledgebases". In: *Journal of Intelligent Information Systems* 27.2, pp. 159–184 (cit. on p. 2).

Gul, Faruk (1989). "Bargaining foundations of Shapley value". In: *Econometrica: Journal of the Econometric Society*, pp. 81–95 (cit. on p. 2).

Hunter, Anthony and Sébastien Konieczny (2010). "On the measure of conflicts: Shapley inconsistency values". In: *Artificial Intelligence* 174.14, pp. 1007–1026 (cit. on p. 2).

Koster, Maurice (1999). "Cost sharing in production situations and network exploitation". Ph.D. thesis. Tilburg University, School of Economics and Management (cit. on p. 18).

Liao, Zhenliang, Xiaolong Zhu, and Jiaorong Shi (2015). "Case study on initial allocation of Shanghai carbon emission trading based on Shapley value". In: *Journal of Cleaner Production* 103, pp. 338–344 (cit. on p. 2).

Livshits, Ester et al. (2020). "The Shapley Value of Tuples in Query Answering". In: *23rd International Conference on Database Theory (ICDT 2020)*. Ed. by Carsten Lutz and Jean Christoph Jung. Vol. 155. Leibniz International Proceedings in Informatics (LIPIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 20:1–20:19 (cit. on pp. vii, ix, 2, 31, 32, 37, 43, 47).

Lucchetti, Roberto (2011). *A Primer in Game Theory*. Società Editrice Esculapio (cit. on pp. 1, 5).

Ma, Richard TB et al. (2010). "Internet Economics: The use of Shapley value for ISP settlement". In: *IEEE/ACM Transactions on networking* 18.3, pp. 775–787 (cit. on p. 2).

Matsui, Tomomi and Yasuko Matsui (2000). "A survey of algorithms for calculating power indices of weighted majority games". In: *Journal of the Operations Research Society of Japan* 43.1, pp. 71–86 (cit. on pp. vii, ix, 26).

Matsui, Yasuko and Tomomi Matsui (2001). "NP-completeness for calculating power indices of weighted majority games". In: *Theoretical Computer Science* 263.1-2, pp. 305–310 (cit. on p. 26).

Narayanam, Ramasuri and Yadati Narahari (2010). "A shapley value-based approach to discover influential nodes in social networks". In: *IEEE Transactions on Automation Science and Engineering* 8.1, pp. 130–147 (cit. on p. 2).

Nenova, Tatiana (2003). "The value of corporate voting rights and control: A cross-country analysis". In: *Journal of financial economics* 68.3, pp. 325–351 (cit. on p. 2).

Petrosjan, Leon and Georges Zaccour (2003). "Time-consistent Shapley value allocation of pollution cost reduction". In: *Journal of economic dynamics and control* 27.3, pp. 381–398 (cit. on p. 2).

Prasad, Kislaya and Jerry S Kelly (1990). "NP-completeness of some problems concerning voting games". In: *International Journal of Game Theory* 19.1, pp. 1–9 (cit. on p. 26).

Tarkowski, Mateusz K. et al. (2018). "Efficient computation of semivalues for game-theoretic network centrality". In: *Journal of Artificial Intelligence Research* 63, pp. 145–189 (cit. on pp. vii, ix, 2, 11, 47).

Yun, Bruno et al. (2018). “Inconsistency measures for repair semantics in OBDA”. In: (cit. on p. 2).

Appendix A

Glossary of Notation

Preliminaries

$\mathcal{G} = (N, \nu)$	Game consisting of the set of players N and of the characteristic function ν.
N	A set of players.
S	A coalition.
$i, j \in N$	A player from the set N.
n	The number of players in a game.
k	Size of a coalition.
$\nu \colon 2^N \to \mathbb{R}$	A characteristic function associating coalitions to utility values.
$\nu(S)$	Utility value associated with the coalition S by the characteristic function ν.
$\mathbf{x}$	A vector of some cardinality (denoted by a lowercase bold letter of the Latin alphabet).
$MC(S, i)$	Marginal contribution of a player i when added to a coalition S.
$\pi_i^{\mathbf{p}}(\nu)$	Semivalue of a player i in a game with characteristic function ν.
$\mathbf{p}$	The vector of probabilities of a semivalue.
p_k	k-th component of the vector of probabilities.
$\sigma_i(\nu)$	The Shapley Value of a player i in a game with characteristic function ν.
$\beta_i(\nu)$	The Banzhaf Index of a player i in a game with characteristic function ν.

A Generic Framework for TU Games

$\mathscr{M}$	A class of characteristic functions.
$\mathscr{M}^*$	A subclass of the class $\mathscr{M}$.
$\Theta_{\mathcal{G}}$	A set of items for a game $\mathcal{G}$.
$\vartheta \in \Theta_{\mathcal{G}}$	An item of the set $\Theta_{\mathcal{G}}$.
$\Theta_l \subseteq \Theta_{\mathcal{G}}$	A group of items from the set $\Theta_{\mathcal{G}}$.
$f\colon \Theta_{\mathcal{G}} \to \mathbb{R}$	Function associating a real value to each item.
$f(\vartheta)$	The value of an item ϑ.
$h(\mathcal{G})$	The number of groups in a partition of $\Theta_{\mathcal{G}}$.
$\mathfrak{R}(S, \vartheta)$	The positive association relation between coalition S and item ϑ.
$\widetilde{\mathfrak{R}}(S, \vartheta)$	The negative association relation between coalition S and item ϑ.
$\mathfrak{N}(S, \vartheta)$	The neutral association relation between coalition S and item ϑ.
$\mathfrak{R}_{\Theta_l}(S)$	Set of items in group Θ_l that is positively related to coalition S. It is similarly defined for the negative and the neutral relation.
$\mathfrak{R}^{-1}_{\#k}(\Theta_l)$	Set of coalitions of size k to which Θ_l is positively related. It is similarly defined for the negative and the neutral relation.
$M_{\mathcal{G}}$	Model of the characteristic function of the game $\mathcal{G}$.
$g\colon \mathbb{N} \to \mathbb{R}$	Function associating a real value to each coalition size.
$g(\lvert S \rvert)$	Real value associated with the size of coalition S.
$c^{\mathfrak{R}}_{\mathcal{G}}(\vartheta \in \Theta_l, k)$	The function computing $\lvert\mathfrak{R}^{-1}_{\#k}(\Theta_l)\rvert$ in polynomial time. It is similarly defined for the neutral relation.
S_k	Random variable of all possible coalitions of size k drawn with uniform probability from a set of players.
$\mathrm{MC}(k, i, \Theta_l)$	Sum of the marginal contributions of player i through all items in Θ_l to all coalition of size k.

Abstract Argumentation Games

$\mathcal{A} = (N, \nu)$	An abstract argumentation game.
$\langle N, \mathcal{R}\rangle$	Graph of an argumentation game, consisting of the set of nodes (the players) N and edges $\mathcal{R}$.
$\mathcal{R}$	The set of edges of the graph.
$\mathcal{P}(i)$	The set of attackers of player i in an argumentation game.
$\widetilde{\mathcal{P}}(i)$	The set of players attacked by i in an argumentation game.

Maintenance Games

$\Gamma = (\mathcal{T}, t, N)$	Maintenance problem consisting of a tree graph $\mathcal{T}$, a nonnegative cost function on the edges t and a set of players N.
$\mathcal{T} = \langle V, E \rangle$	The tree graph of a maintenance problem, consisting of the set of vertices V and edges E.
V	The set of vertices of the tree graph.
E	The set of edges of the tree graph.
$t \colon E \to \mathbb{R}_+$	The nonnegative cost function on the edges of the tree.
$v, v' \in V$	A vertex from the set V.
$v(i)$	The vertex corresponding to player i.
r	Root vertex of the tree.
P_v	The unique path connecting vertex v to r (including v itself).
e_v	The edge in P_v that is incident on v.
$\preccurlyeq$	Precedence relation on the set of vertices.
$R \subset V$	Trunk of the tree closed under the precedence relation.
$P(i)$	Set of players which are predecessors of a player i.
$F(i)$	Set of players which are followers of a player i.
$T(R)$	Total cost of a trunk R.
$\mathcal{M} = (N, \nu_\Gamma)$	The maintenance game associated to the maintenance problem Γ.

Basic Generalized Additive Games

$(N, w, \mathfrak{C})$	Basic Generalized Additive Situation (GAS) consisting of a set of players N, a map w and a collection $\mathfrak{C}$.
$w \colon N \to \mathbb{R}$	The map assigning each player a rel value.
$\mathfrak{C}$	Collection of other collections, one for each player.
$\mathfrak{C}_i$	Collection containing the sets of friends and the set of enemies of player i.
F_i^k	k-th set of friends of player i.
E_i	the set of enemies of player i.
m	Number of sets of friends of any player.
$\mathfrak{M}(S)$	Coalitional map associating every coalition to another coalition.

Weighted Voting Games

$\Delta = (N, \mathbf{w}, q)$	Weighted voting problem consisting of a set of players N, a vector of weights $\mathbf{w}$ and a quota q.
$\mathbf{w}$	The vector of weights.
w_i	Component of the weights vector associated with player i.
q	The quota.
$\mathcal{W} = (N, \nu_\Delta)$	The weighted voting game associated with the weighted voting problem Δ.
$w_{\max}$	Maximum weight between the weights of the players.
$X[j, W, k]$	Number of k-elements subsets between the first j players that have weight W.
$\mathcal{N}_k$	Number of k-elements subsets in the set of players without the last player, that have weight between the quota minus one and the quota minus the weight of the last player.

Query Answering Games on Databases and Tuples

D	A relational database.
$R(\vec{c})$	A fact of the database, consisting of a relation symbol R and a vector of constants $\vec{c}$.
f	A fact of the database.
R	The relation symbol.
$\vec{c}$	The vector of constants.
$\mathcal{S}$	The schema over which is defined the database D.
f_j^i	Fact j of relation i in the database.
D_{n}	The set of endogenous facts.
D_{x}	The set of exogenous facts.
$q(D)$	A query on D.
$\vec{t}$	A vector of variables and constants.
x, x', y, z, w	The existential variable.
A_x	Set of all atoms of q containing variable x.
$D \models q$	The Boolean query q is satisfied by database D.
$\alpha(D)$	A numerical query on D

$\mathcal{D} = (D_{\mathsf{n}}, \nu)$	The game associated with query q on database D.
$\text{Shapley}(D, q, f)$	The Shapley Value of fact f in the game $\mathcal{D}$.
V_x	Set of values variable x is mapped to in some homomorphism from an atom in q to the database D.
a_i	Values of the set V_x
D^{a_i}	The database containing every fact of D that maps the variable x to the value a_i in a homomorphism from the atom corresponding to the relation of the fact to the fact itself.
$q_{[x \to a_i]}$	Query obtained from q by substituting a_i for q.
$f_{i,j}$	Number of subsets of size j of the database D^{a_i} satisfying query $q_{[x \to a_i]}$.
P_i^l	Number of subsets of size l satisfying the database $\bigcup_{r=1}^{i} D^{a_r}$.

Appendix B

Qsemivalue Python Implementation

The implementation of Algoritm 4 follows. The first section contains the actual Pyhton code. At the beginning of the code are declared five strings that serve as program inputs. The documentation is in the comments preceding them. Input databases are represented using a directory containg a `.json` file for every table. Pay attention to the fact that the name of each file will be used as the name of the corresponding table in the code. In the second section there is an example database, which serves as reference for the formatting onvention of its files. Furthermore, all the examples of other inputs in the code are based on this example database too.

Disclaimer. The following code it is not optimized by any means, as its purpose has been to test the correctness of Algorithm 3 and Subroutine 1, and to help design the corrections proposed in Section 5.4.2. Moreover, I strongly suggest any reader to study in depth Chapter 5 before trying to understand this Python implementation, as the terminology used in its comments is the same used in the theoretical treatment.

B.1 Code

```
import json
import math
import os
import re
import itertools
import scipy.special as sp
import numpy as np


######################### Input variables ########################

# Here goes the path to the database directory.
database_path = "/data/database1/"

# Here are listed the names of the tables containing endogenous
# facts, separated by a comma followed by a space. By adding a
```

```
# table name to this list, all facts in that table are considered
# endogenous in the computation.
end_string = "Author, Pub"

# Here goes the query. It has the following syntax:
# "():- R1(x,y,...), R2(w,z,x,...), R3(z,x,'constant',...), ...".
# Constants are written between this '' kind of quotation mark.
# Variables can be a single lowercase letter, constants can be
# numbers or words with more than one letter. Pay attention to
# spacing, it matters!
query_str = "():- Author(x,y), Pub(x,z)"

# Here goes the list of facts of which we want to compute the
# semivalue; use the input below as reference of the syntax.
fact_str = "Author(Alice,UCLA), Author(Bob,NYU), " \
           "Author(Cathy,UCSD), Author(David,MIT), " \
           "Author(Ellen,UCSD)"

# This represents the vector of probabilities of the semivalue.
# Options are: "shapley", "banzhaf", "unweighted" or any vector
# of probabilities with the same size as the set of endogenous
# facts and such that the sum of each component multiplied by
# the binomial coefficient (n-1,k) equals 1. In the binomial
# coefficient 'n' length of the vector and 'k' is the index of
# the component, which goes from 0 to 'n-1'.
# E.g., for five endogenous facts: [0.2, 0.05, 1/30, 0.05, 0.2].
# In fact 0.2*1 + 0.05*4 + 1/30*6 + 0.05*4 + 0.2*1 = 1.
semi_p = "shapley"


############################## Main ################################

# Initialize data, call QSemivalue algorithm for every fact in
# fact_str and print output.
def main():
    database, query, facts, prob_v = get_data()
    for fact in facts:
        print(str(fact[0])
              + str(fact[1]).replace(' ', '').replace('\'', '')
                            .replace(',)', ')')
              + ': '
              + str(q_semivalue(database, query, fact, prob_v)))


###################### Data Initialization #####################

# Prepare input data by manipulating input strings or calling
```

```
64  # other functions. Return the database, the query, a list of
65  # facts to compute the semivalue of, and the vector of
66  # probabilities of the semivalue.
67  def get_data():
68      endogenous_facts = list(end_string.split(", "))
69      db = load_database(os.path.dirname(os.path
70                                         .abspath(__file__))
71                         + database_path, endogenous_facts)
72      # Each fact is structured as a tuple of size two containing
73      # the relation symbol and another tuple of constants; for
74      # example, (Author, (Alice, UCLA)) is a fact.
75      fact_list = [(f.split('(')[0],
76                    tuple([parse_val(s)
77                           for s in f.split('(')[1]
78                                    .strip(')')
79                                    .split(',')]))
80                   for f in fact_str.split(", ")]
81      # Check if all facts in the list of facts to compute the
82      # semivalue of exist and are endogenous.
83      for f in fact_list:
84          if f[0] not in endogenous_facts:
85              print('Bad choice of facts: Shapley value computation '
86                    'makes sense only on endogenous facts and '
87                    + str(f[0]) + str(f[1]).replace(' ', '')
88                                           .replace('\'', '')
89                                           .replace(',)', ')')
90                    + ' is not one of them.')
91              exit()
92      q = query_transform(query_str)
93      prob_v = get_weights(len(db[1]))
94      return db, q, fact_list, prob_v
95
96
97  # Extract the database from the directory containing .json files.
98  # Databases are structured as tuple of size two containing a list
99  # of all facts and another list consisting of endogenous facts only.
100 def load_database(path, e_names):
101     directory = os.fsencode(path)
102     db = list()
103     end_facts = list()
104     for file in os.listdir(directory):
105         filename = os.fsdecode(file)
106         if filename.endswith(".json"):
107             fp = open(path + filename)
108             table_list = json.load(fp)
109             fp.close()
110             name = os.path.splitext(filename)[0]
```

```
111             for d in table_list:
112                 # Check if all tuples of constants in the table
113                 # have the same cardinality.
114                 if len(d.values()) != len(table_list[0].values()):
115                     print('Bad database formatting. Error in the '
116                           'cardinality of values in table ' + name)
117                     exit()
118                 db.append((name, tuple(d.values())))
119                 if name in e_names:
120                     end_facts.append((name, tuple(d.values())))
121     return db, end_facts
122
123
124 # Manipulate the query input string to be used later and return it.
125 # Queries are structured as a tuple of size two containing a list
126 # of atoms and a list of the variables of the query. Atoms have
127 # exactly the same structure as facts.
128 def query_transform(q):
129     # Check that the input string of the query is well formatted.
130     if not bool(re.fullmatch(
131             "[(][)]:- [A-Za-z0-9]{1,15}"
132             "[()([a-z]|'[A-Za-z][A-Za-z0-9]{1,14}'|[0-9]+)"
133             "(,([a-z]|'[A-Za-z][A-Za-z0-9]{1,14}'|[0-9]+))*[)]"
134             "(, [A-Za-z0-9]{1,15}[(]"
135             "([a-z]|'[A-Za-z][A-Za-z0-9]{1,14}'|[0-9]+)"
136             "(,([a-z]|'[A-Za-z][A-Za-z0-9]{1,14}'|[0-9]+))*[)])*", q)):
137         print('\nBad query, ill formatted. \nPlease enter again a '
138               'query in this format "():- R1(x,y,...), '
139               'R2(\'constant\',z,x,...), ...". \nConstants are written'
140               ' between this '' kind of quotation mark. \nVariables '
141               'can be a single lowercase letter, constants can be '
142               'numbers or words with more than one letter. \nPay '
143               'attention to spacing, it matters!')
144         exit()
145     body = q.split(':- ')[1].split(', ')
146     atoms = list()
147     var = set()
148     a_names = list()
149     for a in body:
150         var_list = a.split("(")[1].strip(')').split(',')
151         for v in var_list:
152             if v[0] != '\'':
153                 var.add(v)
154             else:
155                 var_list[var_list.index(v)] = v[1:-1]
156         atoms.append((a.split("(")[0], tuple(var_list)))
157         a_names.append(a.split("(")[0])
```

```
158     # Check that the query doesn't have two or more atoms with the same
159     # relation symbol.
160     if len(a_names) > len(set(a_names)):
161         print('Bad query, has self joins.')
162         exit()
163     # Check that the query is hierarchical.
164     elif not check_hierarchy(atoms, list(var)):
165         print('Bad query, not hierarchical.')
166         exit()
167     return atoms, list(var)
168 
169 
170 # Check if the input query is hierarchical.
171 def check_hierarchy(atoms, variables):
172     # List consisting of, for each variable in the query, the list of
173     # all atoms containing that variable.
174     lists_of_atoms = [set([a for a in atoms if v in a[1]])
175                       for v in variables]
176     # For every couple of variables check hierarchy conditions.
177     for v1, v2 in itertools.combinations(variables, 2):
178         if not (lists_of_atoms[variables.index(v1)]
179                 <= lists_of_atoms[variables.index(v2)] or
180                 lists_of_atoms[variables.index(v2)]
181                 <= lists_of_atoms[variables.index(v1)] or
182                 lists_of_atoms[variables.index(v1)]
183                 .isdisjoint(lists_of_atoms[variables.index(v2)])):
184             return False
185     return True
186 
187 
188 # Transform input string to a int or float, if it contains an int or
189 # a float.
190 def parse_val(s):
191     try:
192         return int(s)
193     except ValueError:
194         pass
195     try:
196         return float(s)
197     except ValueError:
198         pass
199     return s
200 
201 
202 # Check that input variable semi_p has an admissible value,
203 # compute the vector of probabilities and return it.
204 def get_weights(n):
```

```
205     prob_v = []
206     if semi_p == "unweighted":
207         for k in range(n):
208             prob_v.append(1)
209         return prob_v
210     elif semi_p == "shapley":
211         for k in range(n):
212             prob_v.append(math.factorial(k)
213                           * math.factorial(n - k - 1)
214                           / math.factorial(n))
215         return prob_v
216     elif semi_p == "banzhaf":
217         for k in range(n):
218             prob_v.append(1 / (2 ** (n - 1)))
219         return prob_v
220     coef = []
221     for k in range(n):
222         coef.append(sp.binom(n - 1, k))
223     if len(semi_p) != n or np.dot(semi_p, coef) != 1:
224         print("Invalid vector of probabilities. "
225               "Check comment over declaration of "
226               "input semi_p.")
227         exit()
228     else:
229         prob_v = semi_p
230     return prob_v
231
232
233 ########################## QSemivalue ############################
234
235 # Prepare inputs for cnt_sat_l and execute lines 1 to 4 of the
236 # QSemivalue pseudocode.
237 def q_semivalue(db, q, f, prob_v):
238     semi = 0
239     db_all_no_f = db[0].copy()
240     db_all_no_f.remove(f)
241     db_end_no_f = db[1].copy()
242     db_end_no_f.remove(f)
243     for k in range(len(db[1])):
244         semi += prob_v[k] \
245                 * (cnt_sat_l((db[0], db_end_no_f), q, k)
246                    - cnt_sat_l((db_all_no_f, db_end_no_f), q, k))
247     return semi
248
249
250 ######################## CntSatLeftovers #########################
251
```

```
252 # Main execution of CntSatLeftovers subroutine.
253 def cnt_sat_l(db, q, k):
254     # Pseudocode lines 1-8.
255     if len(q[1]) == 0:
256         for a in q[0]:
257             if a not in db[0]:
258                 return 0
259         dim_a = len(set(db[1]).intersection(q[0]))
260         if k < dim_a or k > len(db[1]):
261             return 0
262         return sp.binom(len(db[1]) - dim_a, k - dim_a)
263     result = 0
264     # Pseudocode line 9.
265     # For all variables in the query write down if they are a
266     # root variable. Then, if at least one of them is, proceed.
267     root_check = [all([var_name in q[0][atom_index][1]
268                        for atom_index in range(len(q[0]))])
269                    for var_name in q[1]]
270     if any(root_check):
271         # Pseudocode line 10.
272         # Identify the first root variable.
273         var_name = q[1][root_check.index(True)]
274         # Pseudocode lines 11-27.
275         # Extract values for the variable and create new databases
276         # and queries from them. Then, if the set of values is not
277         # empty, call function compute_p to execute lines 16-26 of
278         # the pseudocode.
279         data = get_dbs_queries_from_values(db, q, var_name)
280         if len(data[0]) != 0:
281             result = compute_p(data, k)
282     # Pseudocode lines 28-33.
283     # Split database and query, and recursively call the
284     # subroutine.
285     else:
286         q1, q2, db1, db2 = split_q(db, q)
287         for i in range(k + 1):
288             result += cnt_sat_l(db1, q1, i) \
289                       * cnt_sat_l(db2, q2, k - i)
290     return result
291
292
293 # Execute lines 16-26 of the pseudocode.
294 def compute_p(data, k):
295     databases_list, check_leftovers, queries_list = data
296     f = np.zeros(shape=(len(databases_list), k + 1),
297                  dtype=float)
298     for i in range(len(databases_list) - int(check_leftovers)):
```

```
299         for j in range(k + 1):
300             f[i][j] = cnt_sat_l(databases_list[i],
301                                 queries_list[i], j)
302     p = np.zeros(shape=(len(databases_list), k + 1),
303                  dtype=float)
304     for l1 in range(k + 1):
305         p[0][l1] = f[0][l1]
306     for i in range(1, len(databases_list)):
307         for l2 in range(k + 1):
308             p[i][l2] = 0
309             for j in range(l2 + 1):
310                 p[i][l2] += \
311                     sp.binom(sum([len(databases_list[r][1])
312                                   for r in range(i)]),
313                              l2 - j) \
314                     * f[i][j] \
315                     + p[i - 1][l2 - j] \
316                     * (sp.binom(len(databases_list[i][1]), j)
317                        - f[i][j])
318     return p[len(databases_list) - 1][k]
319
320
321 # Given a root variable, cycle through all facts. If a fact
322 # contains a suitable value for the root variable, call function
323 # update_dbs_queries to add it to a database corresponding to
324 # that value. Otherwise, add the fact to the leftovers set.
325 # Then, return two lists containing, for each value, a
326 # sub-database of the original consisting of the facts that have
327 # that value in place of the root variable and a query with the
328 # value substituted to the variable, respectively. Also, include
329 # a flag to signal if leftovers set was added to the list of
330 # databases.
331 def get_dbs_queries_from_values(db, q, var_name):
332     check_leftovers = False
333     leftover_facts = ([], [])
334     databases_list = list()
335     queries_list = list()
336     values_list = list()
337     # Cycle through all atoms of the query.
338     for q_atom in q[0]:
339         # Cycle through facts with the same relation symbol of
340         # the atom.
341         for f in (fact for fact in db[0] if fact[0] == q_atom[0]):
342             # From the fact, get the value in the same position of
343             # the first occurrence of the root variable in the atom.
344             value = f[1][q_atom[1].index(var_name)]
345             # Check if the fact is endogenous.
```

```
346             endo = f in db[1]
347             # Check if, in the fact, in all the positions of the
348             # root variable in the atom there is the value (namely,
349             # the fact maps the value to the root variable in a
350             # homomorphism from the atom to the fact). If not, add
351             # the fact to the leftovers set.
352             if all([value == f[1][i] for i, x in
353                     enumerate(q_atom[1]) if x == var_name]):
354                 update_dbs_queries(f, value, endo, (q, var_name),
355                                    (values_list,
356                                     databases_list,
357                                     queries_list)
358                                    )
359             else:
360                 check_leftovers = True
361                 append_fact_to_db(f, endo, leftover_facts)
362     if check_leftovers:
363         databases_list.append(leftover_facts)
364     return databases_list, check_leftovers, queries_list
365
366
367 # Given a value for a root variable, and a fact containing it, if
368 # the value is new, add it to the list of values for the variable,
369 # create a new query with that value in place of the variable and
370 # create a new database containing the fact. Otherwise, if the
371 # fact is not in the database corresponding to the value, add it
372 # to such database.
373 def update_dbs_queries(f, value, endo, q_info, lists):
374     values_list, databases_list, queries_list = lists
375     if value not in values_list:
376         values_list.append(value)
377         queries_list.append(obtain_new_query(q_info, value))
378         databases_list.append(append_fact_to_db(f, endo))
379     elif f not in databases_list[values_list.index(value)][0]:
380         append_fact_to_db(f, endo, databases_list[values_list
381                                                   .index(value)])
382
383
384 # Append a fact to a (possibly new) database, adding it to
385 # endogenous facts if necessary.
386 def append_fact_to_db(f, endo, db=None):
387     if db is None:
388         db = ([], [])
389     db[0].append(f)
390     if endo:
391         db[1].append(f)
392     return db
```

```
393
394
395 # Given a query with a root variable, get a new one from
396 # substituting a constant to a specified variable in all its
397 # atoms.
398 def obtain_new_query(q_info, value):
399     q, var_name = q_info
400     new_vars = q[1].copy()
401     new_vars.remove(var_name)
402     new_query = [], new_vars
403     for atom in q[0]:
404         new_const = list()
405         for v in atom[1]:
406             if v == var_name:
407                 new_const.append(value)
408             else:
409                 new_const.append(v)
410         new_query[0].append((atom[0], tuple(new_const)))
411     return new_query
412
413
414 # Split the query into two subqueries having a root variable,
415 # then, call a function to split the database accordingly.
416 def split_q(db, q):
417     occurrences = [[a for a in q[0] if v in a[1]]
418                    for v in q[1]]
419     occ_len = [len(o) for o in occurrences]
420     q1_atoms = occurrences[occ_len.index(max(occ_len))]
421     variables = set()
422     for a in q1_atoms:
423         variables.update([v for v in q[1] if v in a[1]])
424     q1 = q1_atoms, list(variables)
425     q2 = [a for a in q[0] if a not in q1_atoms], \
426         list(set(q[1]).difference(variables))
427     db1, db2 = split_db(db, q1_atoms)
428     return q1, q2, db1, db2
429
430
431 # Given a subquery, create two new databases separating facts
432 # having relation symbol in the subquery from the others.
433 def split_db(db, q1_atoms):
434     db1 = [], []
435     db2 = [], []
436     for f in db[0]:
437         if f[0] in [a_name[0] for a_name in q1_atoms]:
438             db1[0].append(f)
439             if f in db[1]:
```

```
440                 db1[1].append(f)
441         else:
442             db2[0].append(f)
443             if f in db[1]:
444                 db2[1].append(f)
445     return db1, db2
446 
447 
448 if __name__ == '__main__':
449     main()
```

B.2 Example of Json Database Tables in Input

```
[{
  "name": "Alice",
  "affil": "UCLA"
}, {
  "name": "Bob",
  "affil": "NYU"
}, {
  "name": "Cathy",
  "affil": "UCSD"
}, {
  "name": "David",
  "affil": "MIT"
}, {
  "name": "Ellen",
  "affil": "UCSD"
}]
```

Figure B.1: File `Author.json`: the Author table

```
[{
  "paper": "A",
  "cits": 18
}, {
  "paper": "B",
  "cits": 2
}, {
  "paper": "C",
  "cits": 8
}, {
  "paper": "D",
  "cits": 12
}]
```

Figure B.2: File `Citations.json`: the Citations table

```
[{
  "name": "UCLA",
  "state": "CA"
}, {
  "name": "UCSD",
  "state": "CA"
}, {
  "name": "NYU",
  "state": "NY"
}, {
  "name": "MIT",
  "state": "MA"
}]
```

Figure B.3: File `Inst.json`: the Inst table

```
[{
  "author": "Alice",
  "pub": "A"
}, {
  "author": "Alice",
  "pub": "B"
}, {
  "author": "Bob",
  "pub": "C"
}, {
  "author": "Cathy",
  "pub": "C"
}, {
  "author": "Cathy",
  "pub": "D"
}, {
  "author": "David",
  "pub": "C"
}]
```

Figure B.4: File `Pub.json`: the Pub table