Politecnico di Torino

Politecnico di Torino - Karlsruhe Institute of Technology

Master Degree course in Computer Engineering
A.a. 2023/2024

Formalization and Verification of Proportional Representation Voting Systems using Isabelle/HOL

A Study of D'Hondt and Sainte-Laguë Methods

Supervisors
Prof. Paolo Enrico Camurati
Prof. Dr. Bernhard Beckert
Dr. Michael Kirsten

Candidate
Salvatore Francesco ROSSETTA

Abstract

This master thesis delves into the multifaceted realm of Social Choice Theory, aspiring to not only deepen but also broaden our understanding of collective decision-making in the face of diverse and conflicting individual preferences.

At the core of this exploration is the critical examination of the formalization, verification, and customization of voting systems—foundational mechanisms that significantly impact pivotal decision-making scenarios, including elections and resource allocation.

Leveraging the advanced capabilities of Isabelle/HOL, a sophisticated platform for formal mathematical reasoning, this project meticulously investigates the intricacies of the D'Hondt and Sainte-Laguë voting systems.

Social Choice Theory, the theoretical backdrop of this study, encompasses more than mere preference aggregation—it accentuates the complex interplay of rationality and fairness criteria essential for fostering democratic governance.

Within this theoretical framework, voting systems assume various forms, including plurality, ranked preference, and proportional representation systems, each guided by distinct rules that collectively shape the integrity of the decision-making process. Emphasizing the unique features of Isabelle/HOL, particularly its application of Higher-Order Logic, this study constructs rigorous mathematical models that ensure precision and consistency in the analysis of these intricate systems.

This endeavor represents a pioneering effort to synthesize the principles of Social Choice Theory with the formalization capabilities of Isabelle/HOL. The application of this logical framework to D'Hondt and Sainte-Laguë voting systems serves as a crucial bridge, shedding light on the underlying mechanisms of democratic governance.

Beyond theoretical exploration, the study undertakes a practical mission—to provide a robust foundation for electoral reform and decision-making processes. This involves aligning theoretical insights with the formidable formalization and verification tools offered by Isabelle/HOL, thereby contributing to the advancement of informed and equitable democratic practices.

Central to this research are the properties of anonymity and monotonicity, which are systematically examined within the context of voting systems. Anonymity ensures that individual preferences remain confidential, safeguarding the democratic process against undue influence, while monotonicity scrutinizes the consistency of outcomes when preferences are altered. By probing these properties, the study not only deepens our theoretical understanding but also presents practical implications for the design and implementation of voting systems, adding a layer of insight crucial for the improvement of democratic decision-making.

Contents

1 Introduction

1.1 Motivation

This master thesis will delve into Social Choice Theory, with the aim to better understand collective decision-making, taking into account conflicting individual preferences.

Formal methods are the base for what concerns the fairness of voting systems, which are a core part of scenarios such as elections and resource allocation.

In this work, some properties will be analyzed, formalized and discussed in order to provide a better comprehension of the decision-making and to make sure that the voting system formalized follows this properties.

Among all, three properties have been chosen as milestones in the fairness of voting systems: monotonicity, anonymity and concordance. Furthermore, a formalization of the voting system will help to build a foundation to study its components and its fairness.

This study tries to achieve the following goals:

G1. Formalize the mathematical models of divisor method (D'Hondt and Sainte-Laguë) voting system in Isabelle/HOL.

G2. Prove Anonymity, Monotonicity and Concordance for the divisor method using Isabelle/HOL.

G3. Explore the modularity and extensibility of the framework for encoding voting systems.

G4. Analyze similarities and differences of the two methods, discussing whether one of the methods is "fairer" than the other one.

The goals above lead to the following research questions:

Q1. What considerations arise in the formalization process and how can these challenges be addressed? (G1)

Q2. What are the key properties of the D'Hondt and Sainte-Laguë voting systems that need to be verified for their integrity? (G2)

Q3. How can Isabelle/HOL be utilized to rigorously verify these properties and what insights does this verification process provide? (G2)

Q4. Can the proving of these properties be executed through proving of its components? (G2)

Q5. How can modularity be leveraged to handle variations or extensions of the D'Hondt and Sainte-Laguë methods? (G1 and G3)

Q6. What are the steps to adapt the existing framework to proportional representation? (G3)

Q7. Is there a way to define "fairness" and are the voting systems biased somehow? (G4)

1.2 Contribution

The core of this research is a study into formalization, verification, and customization of voting systems, putting focus towards D'Hondt and Sainte-Laguë methods.

An integral aspect of this investigation lies in the analysis of key properties, notably concordance, anonymity and monotonicity, whose importance influences both theoretical formalization and practical implementation of voting systems.

By leveraging the advanced capabilities of Isabelle/HOL, a sophisticated tool for formal mathematical reasoning, this project tries to build a bridge between Social Choice Theory and formalization capacities of Isabelle/HOL. The contribution of this study is not only defined to the theoretical bases of democracy but also aims to offer practical insights and tools for electoral reform and decision-making processes.

In the following sections, an exploration of Social Choice Theory will follow, showing the complexities inherent in aggregation of preference by voters and the formulation of fairness criteria. The thesis navigates into the properties of concordance, anonymity and monotonicity, revealing their pivotal role in shaping the efficacy and fairness of voting systems.

This exploration is supported by the using of Isabelle/HOL, emphasizing Higher-Order Logic to construct rigorous mathematical models. The core of the thesis involves the application of Isabelle/HOL to formalize D'Hondt and Sainte-Laguë voting systems, with meticulous attention to the evaluation and verification of properties as mentioned above. The ultimate objective is to furnish a robust foundation for electoral reform and decision-making processes, synthesizing theoretical insights with the precision and verification capabilities of Isabelle/HOL to foster informed and equitable democratic practices.

1.2.1 Theoretical Contributions

It would be unnecessary to formalize fundamental definitions pertaining to ballots, voting, and preferences, as accurately formalized by Diekhoff, Kirsten, and Krämer.

Subsequently, this thesis will incorporate some of their established definitions. Besides the use of already estabilished definitions, some new definitions, submodules and modules will be implemented.

Mostly, they will primarily refer to the domain of proportional representation, a facet not comprehensively addressed in the aforementioned work by Diekhoff, Kirsten, and Krämer.

The target is to comprehensively investigate and formally articulate these properties within the Isabelle logical framework, subsequently substantiating their adherence within proportional representation models.

Through this analytical overture, the thesis aims to contribute to the broader topic on the theoretical foundation of proportional representation and establish a formal foundation for subsequent proofs and analyses.

As said above, in this work some definitions from Social Choice theory will be adapted to our case of study.

Parties Definition: introducing an accurate and formal definition of political parties in the context of the voting system, stating a clear understanding of the entities taking part in the electoral process.

Seats Definition: formally defining seats as the units of representation allocated to parties within the legislative body, providing a pivotal idea for the analysis of proportional representation.

Votes Definition: defining votes as expressions of individual preferences, putting effort on the importance of a formalized understanding to find the dangers of voter choice within the voting system.

Fractional Votes Definition: introducing fractional votes, a new theoretical contribution for an accurate representation of voter preferences, particularly relevant in proportional representation systems. Fractional votes will be pivotal in the module, as they will be the metric to assign seats during the decision-making process.

Divisor Method Definition: proposing a formal definition of the divisor method within the voting system, pointing out the approach to allocating seats based on the distribution of votes among the list of the parties, contributing to the theoretical framework of proportional representation.

These theoretical contributions extend the foundational aspects of Social Choice Theory, offering precise and formalized definitions that enhance the analytical toolkit for the study of voting systems. They start to build a base for a deeper understanding of the dynamics at play in electoral processes, aligning theoretical insights with the practical intricacies of democratic governance.

1.2.2 Practical Contributions

The existing framework established by Diekhoff, Kirsten, and Krämer will go under refinement through the implementation of additional modules and submodules designed to abstract the concept of proportional representation.

While certain pre-existing modules will be incorporated, as previously suggested in this document, the primary emphasis lies in the development of modules dedicated to constructing the Divisor Module. This module serves as a foundational element, presenting a generalized methodology applicable to both the D'Hondt and Sainte-Laguë methods for seat allocation.

The strategic extension of the framework aims to improve its versatility and comprehensiveness, allowing for an accurate representation of proportional representation dynamics within the Isabelle logical framework.

Ranking to Votes Module: the ranking to votes module is a useful component, serving as the first step to both extend the existing framework and to introduce the proportional representation.

It is a completely new addition within the framework, converting profiles lists (rankings of alternatives) into votes for parties. Thanks to this module, other methods based on resource allocation or with multiple-winners voting systems can be implemented in the future additions of this framework, since the module has been developed with a generic approach such that different variants of alternatives can be analyzed, resulting in an outcome with the number of preferences for every alternative.

The module has been developed by starting from the already existing Profile, a list of preference relations. In this module, there is an aggregation of the first positions of the rankings, thus generating a corresponding set of votes, linking the preferences of voters expressed as rankings with the parties they prioritize.

Core Divisor Module: Climbing the hierarchy of the modules, the core divisor module constitutes the fundamental mechanism for the assignment of a single seat within the voting system.

This module encapsulates the essential logic and rules governing the allocation of individual seats, serving as the foundational building block for the broader divisor method.

In this module, Fractional Votes of all the alternatives will be compared, thus resulting in a set of winning alternatives. At this point, the modules implemented a safe mechanism to approach the eventuality of a tie.

Studying the divisor method, this tie case emerged as a problem in the formalization and development of the method itself in Isabelle/HOL. According to the method, it's pivotal to note that two different scenarios can happen.

In the first case, only one alternative will be the winner of the seat. In this case, the seat will be assigned and the algorithm can proceed without further complications.

The second case manifests when more than one alternative appear as winner, therefore when there is a tie between more parties. This case has been split in two sub-cases, based on the remaining seats.

In fact, it has been noted that if there are enough resources (so enough seats) with respect to the number of winners, the tie is only apparent. In this case, one seat to each winner alternative can be assigned, resulting in a fair allocation of resources. This brings as a consequence to the other sub-case, namely the one in which there are not enough seats for all the winners.

This is the only tie that has to be handled with more caution in the module, since an erroneous allocation of the seats could result with an unfair outcome, thus not providing a fair voting system.

Furthermore in the work, the implementation of the two cases will be introduced with a clear explanation of their implementation in Isabelle.

Divisor Module: the divisor module extends the core divisor module, implementing the complete divisor method within the voting system.

This module overviews the iterative seat allocation process, ensuring an accurate and systematic implementation of the divisor method. The outcome of this method provides the assignment of all the resources, the seats.

These additions collectively extends the capabilities of the existing framework, providing a generic and adaptable infrastructure for the formalization and analysis of voting systems.

Proofs: the core part of thesis brings an important addition to the framework with proofs of some properties of the method.

1.3 Structure of the Thesis

Chapter 2 will comprehensively address fundamental aspects of voting theory, including pertinent definitions and an exploration of proportional representation, and defines the practical foundation on which the work of this thesis is built. Chapter 3 entails an extensive analysis and formalization of the divisor method. Additionally, this chapter will delineate supplementary modules developed within the ambit of this thesis, contributing to the expansion of the framework. Chapter 4 will delve into the implementation in the tool Isabelle/HOL, starting from the theoretical groundwork provided in chapter 3. In Chapter 5, there will be a short overview of the implementation of properties described in Chapter 3. Chapter 6 is an evaluation of the divisor method and its properties and focuses on the conclusion and related work. The focal point of this expansion is the incorporation of the divisor method, serving as the nucleus of this scholarly inquiry.

2 Foundations

This chapter introduces the whole foundation work on which this thesis is built. As already written in the introduction, this work focuses on two methods that belong to Social Choice Theory and some properties that belong to them.

Therefore, the first foundation section is completely dedicated to introducing the basis of Social Choice Theory that are required to understand the context of the thesis.

This method will be implemented into the tool Isabelle/HOL, into an already existing framework that has already been mentioned above.

As a consequence, the other milestone concerning the foundation of this thesis is the tool Isabelle/HOL, along with the framework that is planned to be extended with the additions of this work.

2.1 Social Choice Theory

In the study of voting systems and their mechanisms in the realm of Social Choice Theory, a foundational understanding of key definitions and properties is needed before going further in this investigation.

These underpinnings serve as the bases upon which the complexities of collective decision-making are studied. In this chapter, we delve in a study through fundamental definitions and properties that encapsulate the essence of social choice, offering insights into the challenges and complications inherent in designing fair, stable and sound voting systems.

From the starting of the theoretical bases of the method analyzed in this work to the various considerations concerning the properties of the method, each concept discovers as a critical lens through which the working of social decision-making are pursued.

As we delve into the core components of Social Choice Theory, the importance of these definitions and properties becomes pivotal, developing the foundation for a comprehensive deep analysis of the formalization and discussion of voting systems within the boundaries of the thesis.

Definition 1 (Proportionality). Proportional representation aims to allocate seats in a legislative body in proportion to the votes each political party receives.

Analyzing the effort to which the voting system achieves proportional representation is fundamental, especially in systems with multiple parties.

Differently from all the methods already developed in the framework, the divisor method provides a new case, not involving a single winner but the allocation of resources, resulting in multiple winners.

As a consequences, all the properties have to be introduced, whether they are newly formalized in this work or not.

The properties analyzed in this work are three pivotal characteristics that every method should hold in order to ensure fairness over the election. The first property, Monotonicity, ensures that receiving stronger support from voters cannot be a loss for the candidate or reduce that candidate's chances of winning seats.

Definition 2 (Monotonicity). The voting system is monotonic if an increase in the votes of a candidates does not cause the candidate to lose seats (Pukelsheim, 2017c). Mathematically, this can be expressed as follows.

Let v_i denote the number of votes received by candidate i and s_i denote the number of seats allocated to candidate i. The voting system is monotonic if for any candidate i, increasing their votes v_i results in either maintaining or increasing the number of seats s_i they receive:

$$v_i' > v_i, \longrightarrow s_i' \geq s_i$$

where v_i' and s_i' represent the increased number of votes and the corresponding number of seats, respectively.

The second property on which this thesis is focused, Anonymity, will be introduced in the same way as done in Pukelsheim, 2017b. It ensures that individual candidates are treated equally regardless of their order in the list of the parties.

Definition 3 (Anonymity). Anonymity in voting means that the order of the parties in the list of the alternatives should not affect the outcome of the election.

The third and final property will be also directly inherited by Pukelsheim, 2017b. This characteristic can be also interpreted as a variant of monotonicity and it is possible to encounter it under this name in other works.

Despite this, to differentiate it from the basic monotonicity described above, during this thesis this property will be addressed as "Concordance".

Definition 4 (Concordance). A divisor method is "concordant" if, for two parties, the stronger party gets at least as many seats as the weaker party. Formally, let $v_i \in V$ be the votes belonging to party i and let $s_i \in S$ be the seats belonging to party i. Then, if the following condition holds,

$$v_j > v_k \Rightarrow s_j \geq s_k, \quad \text{for all parties } j, k = 1, \ldots, n$$

the method is said to be concordant.

In extension to the foundational definitions defined earlier, this thesis is intricately connected to the work of Diekhoff, Kirsten, and Krämer, 2020.

As a consequence, specific definitions from their original framework have been assimilated into my research methodology. It is important to write that some of these definitions could be adapted to better suit the needs of my study, resulting in similar but different definitions or components.

Definition 5 (Ballots). Within the framework of interest for this work (Diekhoff, Kirsten, and Krämer, 2020), it is possible to define a *ballot* as the expression of a preference of a voter within a predetermined and finite set A of eligible alternatives.

The ensemble of voters, designated as V, is likewise finite. Each voter, identified by the index i, provides a *ballot* denoted as $\mathcal{B}_i$. This ballot encapsulates a linear order over the set A, indicating the voter's prioritization or preference among the alternatives.

The linear order expresses the systematic arrangement of alternatives according to the voter's individual preference.

$$\mathcal{B}_i = [pr_1, pr_2, \ldots, pr_n] \tag{2.1}$$

Definition 6 (Profile). As defined in the original framework provided by Diekhoff, Kirsten, and Krämer, 2020, in this context, a *profile* $\mathcal{P}$ is defined as a sequence of k ballots, denoted as

$$\mathcal{P} = (\mathcal{B}_1, \ldots, \mathcal{B}_k) \tag{2.2}$$

where each $\mathcal{B}_i$ represents the ballot cast by voter i. Profiles are the start of the method, as they will be used to count preferences and aggregated to define the number of votes of each alternative.

Definition 7 (Result). The tuple Result in the framework provided by Diekhoff, Kirsten, and Krämer, 2020 is defined at the conclusion of the algorithm, followed by the fulfillment of the termination condition, which is either the assignment of all the available seats or a tie.

The three alternatives depicts the different condition of the alternative during the decision process. In the first set, there are the elected alternatives, while in the second the rejected. In the third set, deferred alternatives are simply the alternatives that still haven't been neither elected nor rejected.

Although one single set, which is the Elected set, could sufficiently encapsulate the election outcome, the inclusion of three sets has the only purpose to maintain coherence with the original framework upon which the extension developed in this work is built.

Moreover, as it can be read in Chapter 4, in the later implementation this choice has been useful to handle the case of a tie.

Let

$$\text{Result} = (\text{Elected}, \text{Rejected}, \text{Deferred})$$

be a tuple of three sets:

$$\begin{aligned} \text{Elected} &= \{(\mathrm{p}_i, s_i) \mid \mathrm{p}_i \in P \text{ and } s_i \in S\} \\ \text{Rejected} &= \{(\mathrm{p}_i, s_i) \mid \mathrm{p}_i \in P \text{ and } s_i \in S\} \\ \text{Deferred} &= \{(\mathrm{p}_i, s_i) \mid \mathrm{p}_i \in P \text{ and } s_i \in S\} \end{aligned}$$

Also, it's important to notice that the three sets have to be disjointed:

$$\text{Elected} \cap \text{Rejected} = \text{Elected} \cap \text{Deferred} = \text{Rejected} \cap \text{Deferred} = \varnothing$$

Each record in the sets represents a party (p_i) and the corresponding number of seats (s_i). Despite the definition, since the algorithm has ended, there will be no "Deferred" parties, so the last set will always be empty. As said above, the tuple is in this structure for consistency with the basic framework.

In the implementation of this work, the Result has been adapted to fit the consideration involving the assignment of seat, resulting in a tuple involving assigned and disputed seats, leaving the second set, originally aimed to describe the rejected alternatives, empty.

The methods that will be analyzed and implemented into Isabelle during this work are well known in Social Choice theory and serve as a base for many voting systems in Europe.

Despite a later formalization will be given in Chapter 3, it is still worth to briefly mention these methods in the foundation, since they are already existing methods.

Definition 8 (Divisor Method). The divisor method (Pukelsheim, 2017a) is an apportionment method, mostly used in the context of proportional representation. It aims to allocate a number of seats between different candidates according to the representation of these alternatives, calculated through preferences of votes.

The divisor method states that the allocation of a seat is decided searching for the maximum value between quotients, calculated for each candidate as

$$q = \frac{Votes}{Divisor}$$

The scaling of the votes through these divisors ensures that also parties with lower votes have chances to gain seats, since a greater number of seats corresponds to a greater divisor, thus decreasing the quotient.

The different variants of this method, as the ones chosen for this thesis, are developed with the aim to reach a fair apportionment changing the divisors.

In fact, the choice of these divisors influence the outcome of the election, as it can be seen in the example provided in chapter 6.

The two variants chosen to be analyzed are the most widely used, even if not in their purest form.

In fact, as it is predictable, the new apportionment method are *mixed*, which means that they have been developed starting from the divisor method implementing other rules to try to reach a fairer result.

Despite this, it is pivotal to always start from the basic methods, in this case D'Hondt and Sainte-Laguë variants. With this addition, it will be also possible to further implement some of the *mixed* methods mentioned earlier.

Definition 9 (D'Hondt Method). The D'Hondt method, as described by Gallagher, 1991, is a divisor method where the divisors belonging to one candidate are calculated simply as the number of seats of that candidate increased by one.

$$q = \frac{Votes}{Seats + 1}$$

The straightforwardness of this method, given by the simple formula to calculate the quotient related to each candidate, brings as collateral effect the fact that, among all the variants, is one of the least proportional in terms of representing the preferences into the assignment of seats.

As a result, different variants were developed, with Sainte-Laguë being one of the most common.

Definition 10 (Sainte-Laguë). The Sainte-Laguë method, as described by Lijphart, 1990, is a divisor method in which the formulas to calculate the quotient related to each candidate is

$$q = \frac{Votes}{2 * Seats + 1}$$

The Sainte-Laguë method, therefore, consists of only odd divisors (1, 3, 5...). As a consequence, the quotient related to each candidate will decrease faster compared to D'Hondt quotient.

Therefore, as a first insight, it is quite intuitive to say that Sainte-Laguë variant gives to smaller parties higher chances to gain seats.

Different variants have been proposed with each of them having a different approach on the formula to create the divisors.

Besides the choice of divisors, it is straightforward to think that these variants all share the same properties. The implementation of these variants in Isabelle will further prove this sharing of properties.

2.2 Formal frameworks

This section will describe the practical foundation on which this work is built. As a matter of fact, this thesis has been developed keeping the focus on two milestones.

The first one is the existing framework that has been already briefly introduced, which gave the possibility to start from already built types and has been useful both as a starting point and as a reference.

The second pivotal practical foundation is the tool Isabelle/HOL itself, which allowed to prove the aforementioned properties for the given method with efficiency.

The proofs have first been developed on a theoretical level but after interacting with Isabelle it was possible, due to a better understanding of the proofs and structures as a consequence of the exploration of the tool, to rewrite the theoretical part with more accuracy and clearness.

2.2.1 Verified Construction of Fair Voting Rules

In this research attempt, the foundation of our analytical framework rests upon the existing work of Diekhoff, Kirsten, and Krämer, 2020. Their comprehensive framework provides a robust infrastructure for the formalization and analysis of voting systems.

Specifically, we leverage key modules from their framework, each contributing crucial functionalities. In the starting phase, there is an exploitation of a set of type synonyms to instantiate our own definitions.

These declarations are directly adopted from the original work without modification, since they will be widely used in the new additional modules.

Preference List. Preference List is a list of a non defined (polymorphic) type *'a*. It derives from preference relations, showing a ranking of alternatives starting from the most up to the least preferred.

Also, it is trivial but it is worth saying that it is required for the Preference List to be well-formed to ensure clarity and consistency in representing individual preferences.

Each element in the list has to correspond to a distinct alternative, since having duplicates might lead to ambiguities or unintended interpretations.

```
type_synonym 'a Preference_List = "'a list"
```

To maintain things clear and draw a comparison with the theoretical foundation, this preference list corresponds to a single ballot.

Profile List. A Profile List contains one ballot for each voter. It is a list of Preference List of unspecified type 'a.

```
type_synonym 'a Profile_List = "'a Preference_List list"
```

An electoral result, a key outcome in the modular voting framework, extends the idea of winning alternatives in social choice modules. These results split the received alternatives into three distinct sets: elected, rejected, and deferred alternatives.

It is important to note that any of these sets, such as the set of winning alternatives, can be empty, as long as collectively, they sum up to all received alternatives.

Result. A result contains three sets of alternatives: elected, rejected, and deferred alternatives.

```
type_synonym 'a Result = "'a set * 'a set * 'a set"
```

As mentioned earlier, the Result will be newly adapted to better describe the context of proportional representation, giving primary importance to the seats, which will be either Assigned or Disputed. Thanks to the generic implementation, it was possible to use this same type synonym adapting it without the need to change anything.

Following the definition of type synonyms, the subsequent step involves the incorporation of modules and submodules sourced from the original framework.

Loop Composition. The Loop Composition module extends the framework by enabling the seamless composition of iterative processes, fostering a more dynamic and adaptive analytical environment. A loop composition repeatedly executes an electoral module sequentially with itself until either a stable point is reached or a termination condition is verified.

As navigation through complexities of this established framework proceeds, the study of the module extends beyond the existing modules. The enrichment of the framework with additional modules focused to the specific demands of our study helps to split the whole divisor module in smaller components.

These modules, crafted to enhance the analytical capabilities of the framework, will contribute novel perspectives to the formalization and verification processes.

The Loop Composition has been used to developed another completely new function for the module, formalized in this work, as the complexity of the Divisor Method required some additional parameters and effort, since the outcome is not a binary elected/rejected choice but rather an allocation of the seats, requiring more structures to be described and executed.

In essence, this research stands at the meeting point of foundational work built by Diekhoff, Kirsten, and Krämer and the pioneering efforts to extend and customize the existing framework.

Throughout this symbiotic unification of established modules and personal additions, the aspiration is to enrich the analytical framework available, thereby advancing our understanding of the intricacies inherent in the formalization and analysis of voting systems.

2.2.2 Isabelle/HOL: A Tool for Formal Verification

Isabelle, a powerful formal verification tool and interactive theorem prover, plays an important role in ensuring the rigor and correctness of mathematical logic and the formal verification of various systems, including software and hardware.

It provides a robust extensive framework for the meticulous validation of mathematical theorems and the properties of complex systems.

Within the Isabelle ecosystem, Isabelle/HOL stands out as a formidable system renowned for its prowess in rigorous mathematical reasoning and formal verification, as described in Makarius, Paulson, and Nipkow, 2020. Isabelle/HOL emphasizes Higher-Order Logic (HOL), a formalism that facilitates precise reasoning about higher-order functions and expressions.

Widely applied in various topics extending from software and hardware verification to mathematical theorem proving, Isabelle/HOL stands as a beacon of reliability and precision in the realm of formal verification.

Applying Isabelle/HOL to analyze voting systems, particularly the D'Hondt and Sainte-Laguë methods, brings forth numerous advantages.

By encoding the rules and algorithms of these voting systems in Isabelle/HOL, it is possible to prove essential properties, including proportionality, fairness, and resistance to manipulation. This formal approach also provides a deep comprehension of the intricate mechanisms governing these systems.

Moreover, using Isabelle/HOL facilitates the development of reliable software implementations of these voting systems, furnishing a stable underpinning for analysis and decision-making in electoral governance.

Theory Files. In Isabelle, theory files (Wenzel et al., 2023) are essential documents for formalizing mathematical reasoning.

Written in the Isabelle/Isar language, these files comprehend definitions, theorems and proofs, divided and ordered into structured sections. Isar provides a human-readable syntax, facilitating clear expression of logical concepts.

Theory files support modularity by allowing the import of other theories, advertising reuse of established developments. They integrate proof scripts, guiding the Isabelle proof assistant with either Isar or tactic-based scripts. This flexibility accommodates different proof styles.

Isabelle's integration with other tools enhances the capabilities within theory files, offering features like proof automation and model checking. As a consequence, theory files in Isabelle provide a structured but flexible platform for rigorous mathematical reasoning, combining logical precision, modularity, and practical tool integration.

Theorems. Theorems in Isabelle, as defined by Wenzel et al., 2023, play a central role in formalized mathematical reasoning, serving as fundamental compositional blocks for logical assertions and conclusions.

In Isabelle, theorems are rigorously defined and proven statements that contribute to the overall structure of theory files. These files organize theorems alongside definitions and proofs, creating a cohesive environment for mathematical development.

The modularity of theory files in Isabelle is only the basis for what it's then enriched to theorems, enabling their reuse across different projects and environments. The importation of theorems from other theories promotes efficiency and consistency in formalizing mathematical concepts.

Theorems in Isabelle embody the rigor and precision of formalized mathematical reasoning. They contribute to the logical structure of theory files, supporting modularity and enabling the reuse of mathematical developments.

Isabelle's integration with proof scripts highlights the practicality and flexibility of working with theorems, making it a powerful tool for formal reasoning in mathematics.

In this work, theorems will be used to define the properties analyzed, so monotonicity, concordance and anonymity. Furthermore, the goal is to proof these theorems by using simpler lemmas and logical reasoning.

Functions. Functions (Wenzel et al., 2023) constitute an important component of mathematical modeling and reasoning in the Isabelle proof assistant. In Isabelle, functions are formalized as mappings between sets, adhering to the foundational principles of the underlying logical framework.

This structure are defined using a syntax that aligns with the precision required for formal mathematical reasoning.

The modular nature of functions in Isabelle facilitates their integration into larger mathematical environments.

Functions defined in a theory can be without any effort imported and exploited in other theories, promoting code reuse and ensuring consistency across different topics.

Isabelle provides various and powerful sets of tools for reasoning about functions, including powerful simplification and automation tactics. Theorems about functions can be proven using these tactics, contributing to the establishment of mathematical properties and relationships.

Moreover, the Isar proof language in Isabelle allows for clear and structured explanations of proofs related to the functions. The integration of functions into the wider logical framework, combined with Isabelle's support for automation and proof scripting, makes it a versatile platform for formalizing mathematical concepts involving functions.

Record. In Isabelle, a record (Wenzel et al., 2023) is a composite data structure that contains named fields. It's similar to a struct in languages like C or a record in languages like Ada or Pascal.

Records provide a way to group related pieces of data together into a single unit. In Isabelle, records are typically defined using the record keyword followed by the record name and a list of field names along with their types.

As an example, a record could be named *Point* and have two fields, namely x and y, that are fields of the record.

Using records is good practice when the aim is to encapsulate various fields belonging to a more complex structure, because it allows an efficient and comprehensible parameter definition.

In the development of this thesis, records have been used to formalize the parameters belonging to the Divisor Method. Through this technique, the different functions executing the method are cleaner and only pass one parameter to each other, indeed the record itself.

Then, every function is changing only the fields of the record it is required to and passes the updated record to the next function.

As a consequence for this approach, it was possible to reduce the amount of redundant variables and always have an accurate idea of the state of the record and, therefore, of the algorithm.

3 Divisor Method

Proportional representation (PR) stands as a cornerstone in the architecture of democratic electoral systems, designed to ensure a fair and equitable translation of voters' preferences into legislative representation. Unlike majoritarian systems, Proportional Representation seeks to allocate seats to political parties proportionally to their share of the total vote, fostering a more accurate reflection of diverse voter choices.

At the heart of many PR systems lies the divisor method, a mathematical mechanism employed to distribute seats among competing parties. This method aims to standardize the distribution of seats by dividing the total votes cast for each party by a set of divisors. This process, in turn, determines the number of seats a party is entitled to in a proportional manner.

Two main variants of the divisor method are the D'Hondt method and the Sainte-Laguë method. The D'Hondt method, named after Belgian mathematician Victor D'Hondt, utilizes a sequence of divisors that are incrementally increased for each round of seat allocation. This method tends to favor larger parties, making it especially relevant in political landscapes dominated by established entities.

On the other hand, the Sainte-Laguë method, named after French mathematician André Sainte-Laguë, deploys a sequence of divisors that follows an odd-number pattern. This approach introduces a higher level of proportionality, providing smaller parties with a greater opportunity to secure seats.

The first step in this work is to analyze and formalize the Divisor Method, delving into theoretical foundations of Social Choice Theory in order to give a legitimate and accurate description of the proportional representation method on which the thesis is focused, always keeping an eye on the importance to persist in the aim of being uncomplicated, making the formalization of the method clean and understandable.

3.1 Basic Definitions

This chapter lays the groundwork by unpacking the essentials of voting theory. We'll dive into the basics — parties, seats, fractional votes, and winners. Drawing on the groundwork of Diekhoff, Kirsten, and Krämer, we'll explore the fundamental concepts that underpin democratic decision-making. These basic definitions create a shared understanding, forming the basis for delving deeper into the complexities of voting theory. This section is the entry point to grasp theoretical foundations that steer collective decision-making.

The first definition is the one about Parties, a vector in which each element represents a party.

Definition 1 (Parties). Let P be the list of parties, where p_i represents the name of the i-th party.

$$P = [p_1, p_2, \ldots, p_k] \tag{3.1}$$

In the realm of proportional representation, political entities compete for parliamentary seats. Each party is assigned a specific number of seats in accordance with the votes it accrues. This allocation process is integral to ensuring that the composition of the legislature aligns proportionately with the preferences expressed by the electorate.

Definition 2 (Seats). Let s be the set of seats, where s_i denotes the number of seats allocated to party $p_i \in P$.

$$s(n) = [s_1(n), s_2(n), \ldots, s_k(n)] \tag{3.2}$$

assuming that the following condition holds true

$$s_i(j) \geq 0 \qquad \forall i \geq 0, j \geq 0$$

Formally, s_i is the number of seats allocated to party p_i.

The preferences of the voters are expressed through ballots, with every voter's preference being expressed through one single ballot. Recalling from voting theory that one ballot is defined as a list of preferences

$$\mathcal{B}_i = [pr_1, pr_2, \ldots, pr_n] \tag{3.3}$$

with ever pr_i being a preference of a candidate over another, in the form

$$pr_i : p_j > p_k \quad \text{where } p_j, p_k \in P$$

In the current framework, we employ a methodology centered around "ballots", representing a sequential order of candidate preferences. These individual ballots are subsequently organized into a Profile.

However, a subsequent adjustment is needed for this specific scenario, as the Profile inherently signifies the complete ordering of all candidates based on voter preference, ranging from the foremost to the last candidate. On the contrary, in the divisor method the number of votes for every party is required. In the following paragraph, the approach used will be described.

In the context of proportional representation methods like D'Hondt and Sainte-Laguë, divisor parameters play a crucial role in determining the allocation of seats to political parties. These divisors are values used in the seat distribution formula to standardize the party's vote count and calculate their normalized votes.

Definition 3 (Divisors). Let Λ be the vector of divisors, where λ_i denotes the i-th parameter.

$$\Lambda = [\lambda_0, \lambda_1, \ldots, \lambda_k, \ldots] \tag{3.4}$$

It is pivotal to establish that Λ is monotone.

$$\forall i \geq 1 \implies \lambda_i \geq \lambda_{i-1} \tag{3.5}$$

In other words, each value in the sequence is greater than or equal to its predecessor. The choice of these parameters can significantly impact the outcome of an election, influencing the representation of both large and small political entities.

It is a key consideration in the design of electoral systems, balancing the interests of different-sized parties and contributing to the overall democratic integrity of the voting process.

3.2 Formalization of the Divisor Method

After having formalized the basic definition which will be the foundation of the work, it is possible to start to give a proper and clear formalization of the divisor method. Therefore, recalling (3.1), (3.2) and (3.3), the starting point is a list of candidates, namely Parties

$$P = [p_1, p_2, \ldots, p_k],$$

a list of N seats to be allocated

$$s(n) = [s_1(n), s_2(n), \ldots, s_N(n)] \qquad 0 < n \leq N$$

where n is indicating the current *round* and a Profile is defined as a list of ballots

$$\mathcal{P} = [\mathcal{B}_1, \mathcal{B}_2, \ldots, \mathcal{B}_k].$$

As already mentioned earlier, in the divisor method, though, preference is not expressed through Profile; instead, every candidate (party) has a number of votes, used to determine its number of seats.

Due to this reason, the following approach has been adopted: starting from $\mathcal{P}$, for a party p_i, its number of votes will be determined retrieving all the ballots having p_i as first preference.

Definition 4 (Votes). Let $\mathcal{P}$ be the Profile containing all the ballots and let V be the set of votes, where $v_i \in V$ denotes the number of votes that the party $p_i \in P$ has received.

$$V = [v_1, v_2, \ldots, v_n] \tag{3.6}$$

The total number of votes v_i, computed for each party p_i, is derived from the sum of all ballots wherein p_i occupies the first position in the preference list.

$$v_i = \sum_{\mathcal{P}} \delta(p_i \text{ is the first preference}) \tag{3.7}$$

It is indisputable that the number of votes must be one of the core parameter for the assignment of the seat. Despite this, assigning seats only over the number of seats could lead to unfairness assignment problems.

As a consequence, Divisors can be introduced.

Definition 5 (Divisors). Let $s_i \in s$ be the number of seats of $p_i \in P$. Then, λ is a function that associates a natural number to every s_i, called divisor.

$$\lambda : \mathbb{N} \to \mathbb{N}$$

These divisors have been defined because they play the most important role in the module, because changing this can lead to a change in the outcome, since the assignment of a seat will not depend on the pure votes but on another type of parameter, that will be named *fractional vote*.

Definition 6 (Fractional Vote). Let $v_i \in V$ be the votes for the party $p_i \in P$ and let $s_i(n) \in s$ be the number of seats of p_i. Finally, let $\lambda(s_i(n))$ be the divisor associated to $s_i(n)$. Therefore, the function f can be defined as

$$f(p_i, n) = \frac{v_i}{\lambda(s_i(n))} \quad \forall i \geq 0, n \in N. \tag{3.8}$$

Since, according to the divisor method, the winner is searched through the maximum fractional vote, there can be multiple winners in the same time, in other words the parties with the same value of *fractional vote*.

Definition 7 (Winners). Let P be the list of candidates and let f be the function defined in (3.8). Then w is the vector of k parties with the maximum current *fractional vote*,

$$w(n) = [w_1(n), ..., w_k(n)] = \arg\max_{p_i \in P}\{f(i, n)\}. \tag{3.9}$$

Since all the winners have the same right of gaining a seat, it is pivotal to define a strategy to choose the winner for the next seat. At this point, the theoretical formalization meets the practical implementation on Isabelle/HOL.

The following approach has been chosen: the first element in the vector of winners, w_1 will receive the next seat.

As a consequence, this rule can be inherited

$$\frac{p_i = w_1(n)}{s_i(n+1) = s_i(n) + 1} \tag{3.10}$$

Definition 8 (Gamma). Let $s(n)$ be the seats at round n and let $w(n)$ be the list of winners of round n. The function gamma is defined as

$$\gamma(s, n) = \begin{cases} s_i(n+1) = s_i(n) + 1 & p_i = w_1(n) \\ s_i(n+1) = s_i(n) & p_i \neq w_1(n) \end{cases} \tag{3.11}$$

It is worth considering the case in which at some point during the algorithm the list of winners is longer than the remaining seats. This would lead the last parties not to have the seat they deserve. This case has to be correctly analyzed and handled to avoid errors in the practical implementation.

In case of a tie, the seat is going to remain disputed and the module will proceed without assigning the next seat or updating values of *fractional vote*.

Definition 8 (Core Module). Let $w(n)$ be the list of winners competing for the n-th seat, with N being the total number of seats. Then, the core module function is defined as

$$C(s,n) = \begin{cases} \gamma(s,n) & |w(n)| \leq \text{N-n} \\ s & |w(n)| > \text{N-n} \end{cases} \tag{3.12}$$

with $N - n$ being the remaining number of seats.

As a consequence, when there is a tie the seats s will be returned without changing any value. Through this strategy, the method will not update any values and the seats values will remain the same until the end of the method.

At the end of the algorithm, if there has not been a tie, then all the seats will be assigned. Otherwise, in a more rare case, a lack of available seats will leave the remaining seats disputed.

Definition 9 (Divisor Method). Let P be the candidates with votes V and let N be the number of seats. Then, the divisor method is a function

$$D(s,N) = \sum_{0<n\leq N} C(s,n) \tag{3.13}$$

The goal of this paragraph was to write a formalization of the method acknowledging different sub-goals, including the usage of an understandable syntax to keep the method clear.

Moreover, it has been taken into account that the method should consider all the possibilities, for example also the one regarding a tie, to avoid leaving borderline cases undefined in this formalization.

Finally, it was also taken in consideration that this formalization has to be useful to write the method into the tool Isabelle/HOL, therefore the iterative approach, along with the exploit of logical operators and implications.

3.2.1 D'Hondt Method

Suppose there are three political parties (Party A, Party B, and Party C) and a total of five seats to be allocated based on their share of the vote.

Party	Votes	Seats
Party A	5000	-
Party B	4000	-
Party C	3000	-

We will use the D'Hondt method to allocate the five seats.

1. Initialize the array of allocated seats: $S = [0, 0, 0]$.

2. For each seat to be allocated, we calculate the allocation ratio for each party:

Party	Allocation Ratio	New Seat
Party A	$\frac{5000}{0+1} = 5000$	-
Party B	$\frac{4000}{0+1} = 4000$	-
Party C	$\frac{3000}{0+1} = 3000$	-

3. We allocate the first seat to Party A since it has the highest allocation ratio.

Party	Votes	Seats
Party A	5000	1
Party B	4000	-
Party C	3000	-

4. For the second seat, we recalculate the allocation ratios:

Party	Allocation Ratio	New Seat
Party A	$\frac{5000}{1+1} = 2500$	-
Party B	$\frac{4000}{0+1} = 4000$	-
Party C	$\frac{3000}{0+1} = 3000$	-

5. The second seat goes to Party B.

Party	Allocation Ratio	Seats
Party A	$\frac{5000}{1+1} = 2500$	1
Party B	$\frac{4000}{1+1} = 2000$	1
Party C	3000	-

   The third seat goes now to Party C, since it has the biggest allocation ratio.

6. We repeat the process to allocate the remaining seats. After allocating all five seats, the distribution is as follows:

Party	Votes	Seats
Party A	5000	2
Party B	4000	2
Party C	3000	1

   The D'Hondt method has been used to allocate seats proportionally based on the vote counts.

3.2.2 Sainte-Laguë Method

Suppose there are three political parties (Party A, Party B, and Party C) and a total of five seats to be allocated based on their share of the vote.

Party	Votes	Seats
Party A	5000	-
Party B	4000	-
Party C	3000	-

We will use the Sainte-Laguë method to allocate the five seats.

1. Initialize the array of allocated seats: $S = [0, 0, 0]$.

2. For each seat to be allocated, we calculate the allocation ratio for each party:

Party	Allocation Ratio	New Seat
Party A	$\frac{5000}{1}$	-
Party B	$\frac{4000}{1}$	-
Party C	$\frac{3000}{1}$	-

3. We allocate the first seat to the party with the highest allocation ratio.

Party	Votes	Seats
Party A	5000	1
Party B	4000	-
Party C	3000	-

4. For the second seat, we recalculate the allocation ratios:

Party	Allocation Ratio	New Seat
Party A	$\frac{5000}{3} = 1667$	-
Party B	$\frac{4000}{1}$	-
Party C	$\frac{3000}{1}$	-

5. The second seat goes to the party with the highest allocation ratio, in this case Party B.

6. We repeat the process to allocate the remaining seats.

After allocating all five seats, the distribution is as follows:

Party	Votes	Seats
Party A	5000	2
Party B	4000	2
Party C	3000	1

The Sainte-Laguë method has been used to allocate seats proportionally based on the vote counts.

3.3 Properties in Proportional Representation

Proceeding into the study of the proportional representation, the next step into this work is to analyze some properties that characterize these voting systems.

As a matter of fact, the method itself would not be useful or efficient if it does not hold some properties. Therefore, it is pivotal to prove that these systems have some properties that can assure some level of "fairness" in the outcome of the elections. Without these properties, the method would not be advantageous to use.

Various characteristics have been defined for the proportional representation methods and some of these have been chosen to be analyzed in the realm of this work: anonymity, monotonicity and concordance.

In the following paragraphs, the three properties will be briefly introduced and analyzed. Moreover, the proofs of these properties will be demonstrated, showing that the method itself respects these three features.

As the same approach has been followed for the divisor method, the aim of this chapter is to both give a transparent idea of these properties and to prepare the theoretical foundation for the work that will be done in the tool Isabelle/HOL.

3.3.1 Anonymity

The anonymity property in voting asserts that the outcome of an election remains unchanged when the order of parties is altered, provided the votes for each party remain constant.

Before proving anonymity property, it is important to define a lemma that will simplify part of the proof of the theorem.

Lemma 1. *Let $s(n) \in s$ and let $s'(n) \in s'$ be the number of seats at round n.*

Let $w(n)$ be the winners associated to $s(n)$ and $w'(n)$ the winners associated to $s'(n)$, with $w'(n) = \pi(w(n))$ and $k = |w(n)| = |w'(n)|$. If

$$s(n) = s'(n)$$

holds true, then

$$s(n+k) = s'(n+k). \tag{3.14}$$

In other words, this lemma is assuring that even if the list of winners may be in a different order, after k rounds every winner will have the number of seats updated or, in case of a tie, they will stay with the same seats.

Proof. After the first assignment, the new list of winners will be something like

$$w(n+1) = [w_2, ..., w_k]$$

after the second assignment,

$$w(n+2) = [w_3, ..., w_k]$$

and so on, after k, all the parties in $w(n)$ will have gained one seat.

The same applies for $w'(n)$: all the parties in this vector receive one seat. Since $w'(n) = \pi(w(n))$ the same parties will have the seats increased in the two occasions and therefore

$$s(n+k) = s'(n+k).$$

□

Theorem 1. ***(Anonymity).*** *Let N be the number of seats and let P be the list of parties, with $P' = \pi(P)$ being its permutation. Then, let V' and S' be the number of votes and number of seats of P'. Moreover, let f be*

$$f(i,n) = \frac{v_i}{\lambda(s_i(n))}$$

and let f' be

$$f'(i,n) = \frac{v_i'}{\lambda(s_i'(n))}.$$

The divisor method is said to be anonymous if

$$s_i(N) = s_j'(N) \quad \forall i,j \in P : p_i = p_j'.$$

The proof of the anonymity property for the divisor method can appear trivial but it actually suffers the consequences of the formalization above.

In fact, it is straight-forward that the only change of the order of party does not affect the search for the maximum, as long as the number of votes stays the same.

The formalization of the method has been built in a way that assigns the seat to the first winner in the list. This brings to the consequence that if the order of parties changes, then also the first party in the list of winners changes.

This means that under permutation the seats will not be assigned in the same order but the outcome, the final number of seats for each party, is the same.

Proof. For this proof, the starting point is the first assignment of seat. I have the assumption that $s(1) = s'(1)$ since all the seats are zero.

Since $s(1) = s'(1)$, it is possible to write

$$f(p_i, 1) = f'(p_j, 1) \quad \forall p_i = p_j.$$

As a consequence, this means that $w(1)$ and $w'(1)$, the functions finding the maximum values for f, have the same parties, just in a different order, so

$$w'(1) = \pi(w(1)).$$

Since the two assumptions holds, the lemma described above and therefore

$$s(1+k) = s'(1+k).$$

The next step is to assign the $k+2$-th seat. Since the seats are the same in both occasions for both parties, consequently again the list of winners $w(k+2)$ has the same parties of $w'(k+2)$, so again $w'(k+2) = \pi(w(k+2))$. us notice that the assumptions holds. $s(1+k) = s'(1+k)$ holds.

Therefore, also for the case $k+2$, the theorem holds because the k' winners are gonna be the same.

At this point it is straight-forward that the same strategy can be applied again until the total number of seats N is reached.

□

The study of the anonymity property is crucial for several reasons. Firstly, it upholds the principle of fairness by ensuring that all parties are treated equally in the electoral process. If altering the order of parties led to different outcomes, it could introduce bias into the system, favoring or disadvantaging certain parties based solely on their position in the list.

Secondly, the anonymity property promotes the integrity and transparency of the electoral system. Voters can have confidence that their choices will not be unduly influenced by the arrangement of parties on the ballot.

In the context of the divisor method, adherence to the anonymity property is paramount. As the method allocates seats based on the votes received by each party, the order in which parties are presented should not impact the distribution of seats.

Ensuring the anonymity property is followed in the implementation of the divisor method contributes to a more equitable and unbiased electoral process. It reinforces the notion that the method treats all parties fairly and impartially, irrespective of their position in the sequence.

On a theoretical level, it can seem trivial that anonymity holds for the divisor method. Nevertheless, it is important to give a correct and formal proof for this property, both to assure the correctness of the method and to provide a starting point for the later practical implementation in Isabelle/HOL.

3.3.2 Monotonicity

Monotonicity in voting systems asserts that an increase in voter support for a candidate should not result in a decrease in the candidate's standing in the final outcome.

Theorem 2. ***(Monotonicity)****. Let N be the total number of seats, let $v_i \in V$ be the votes of party $p_i \in P$ and let $s_i(n) \in D(s, N)$ be the number of seats of p_i during round n. Finally, let v_i' be an increased value of v_i*

$$v_i' > v_i \tag{3.15}$$

Then D(s, N) is said to be monotone if

$$s_i'(n) \geq s_i(n) \quad 1 \leq n \leq N.$$

The monotonicity property is a crucial criterion for evaluating the fairness and consistency of a voting system. Violations of monotonicity can lead to unexpected and counterintuitive outcomes, raising concerns about the democratic integrity of the decision-making process.

Proof. To prove this theorem, the strategy adopted starts from the number of seats as core parameter. Due to this reason, the following proof is executed by induction over the number of seats n.

The number of seats has been chosen as the focus of induction because the divisor method has been formalized with one round allocating one seat, meaning that the induction process easily works on this interpretation.

Moreover, iterating over the number of seats makes the proof more clear. Another strategy could have been, as an example, to induct over the number of parties or votes. These ideas were not fitting the method and the theorem, therefore the iteration over the seats is the best choice.

As already known, the proof by induction consists in proving two different scenarios: the base case, when $n = 1$ and the step case, in which assuming that the theorem holds for the step n, it also holds for the next step, $n + 1$.

Base case. The base case is focused on the case

$$n = 1.$$

Therefore, the context is the first *round* of assignment of seats and therefore, no seats have been assigned yet.

$$s_i(1) = 0,$$

$$s_i'(1) = 0.$$

Then, the conclusion follows

$$s_i'(1) \geq s_i(1).$$

Step case. In this second case of the proof, the following inductive hypothesis holds

$$s_i'(n) \geq s_i(n)$$

and the sub-goal to prove is

$$s_i'(n+1) \geq s_i(n+1).$$

The pivotal element of the induction is the number of seats and the step case examined delves into the assignment of the $n + 1$-th seat.

As a consequence, it is quite hard to define a general proof without knowing the outcome of the next round and it is needed to investigate every possible case separately.

Due to this reason, different scenarios can happen. In the first one, the party wins the seat both before and after the increasing of the votes (1), while in the second the party already wins a seat before increasing its votes (2) and in the third case the party wins a seat only after having increased its votes (3).

In the last case, the party does not win the seat nor before neither after increasing the votes (4).

Sub-case (1). In this context the party wins the seat both before and after the increase of the votes and as a consequence

$$s_i(n+1) = s_i(n) + 1,$$

$$s_i(n+1)' = s_i(n)' + 1.$$

At this point, using the inductive hypothesis

$$s_i(n) \geq s_i(n)' \implies s_i(n) + 1 \geq s_i(n)' + 1$$

and by substitution the thesis is proved

$$s_i(n+1) \geq s_i(n+1)'.$$

Sub-case (2). In this scenario, the party wins the seat before the increase of the votes but recalling eq. (3.8) and since eq. (3.15), it is allowed to write

$$f(p_i, n)' > f(p_i, n).$$

Since the win of the seat is given by the maximum value in f, it is trivial that if the party is winning before the increase, it will necessarily win also after the aforementioned increase. Due to this reason, this case falls back in the sub-case (1).

Sub-case (3). In the third case, p_i wins the seat only after having increased its number of votes. As a consequence,

$$s_i(n+1) = s_i(n)$$

$$s_i(n+1)' = s_i(n)' + 1$$

and starting from the inductive hypothesis the conclusion easily follows

$$s_i(n+1)' \geq s_i(n+1).$$

Sub-case (4). The last case is also the most trivial: the party is incapable to win a seat both before and after the increase of the votes. Consequently,

$$s_i(n+1) = s_i(n),$$

$$s_i(n+1)' = s_i(n)'$$

and the thesis holds

$$s_i(n+1)' \geq s_i(n+1).$$

□

In essence, the monotonicity property ensures that heightened voter support for a candidate translates into a more favorable or, at the very least, an unchanged position within the final outcome.

3.3.3 Concordance

The concordance property is one of the pivotal properties considered in the voting theory and in proportional representation it assures that in the outcome of the results a stronger party will not receive less seats than the weaker party.

Theorem 3. ***(Concordance).*** *Let $p_j \in P$ and $p_k \in P$ be two parties, such that*

$$p_j \neq p_k \quad \forall j \geq 0, k \geq 0,$$

competing for N number of seats. Let $v_j \in V$ and $v_k \in V$ be the corresponding number of votes of p_j and p_k, with

$$v_j > v_k \quad , \quad v_j \geq 0, v_k \geq 0. \tag{3.16}$$

Moreover, let $s_j(n) \in D(s, N)$ be the seats assigned to p_j and $s_k(n) \in D(s, N)$ the seats assigned to p_k.

Then

$$s_j(n) \geq s_k(n) \qquad 0 < n \leq N. \tag{3.17}$$

Proof. The above theorem can be proved through induction over the number of seats n. The proof has been developed inducting over the number of seats because it is the pivotal element in the theorem, while induction over other elements would not have given the same cleanliness over the proof.

Instead, having the seats as the central element of the proof helps to understand why it works and makes the proof more understandable and clear.

As it is known, the induction proof works over two different cases: the base case and the step case.

The first one is the base case. The goal of this case is to prove that the theorem holds starting from

$$n = 1.$$

This means that the current round is the first one. Trivially, at the beginning of the method no seats have been assigned yet. Therefore

$$s_j(1) = 0,$$

$$s_k(1) = 0,$$

and, regardless of the votes, it is possible to write

$$s_j(1) \geq s_k(1).$$

Therefore, the thesis holds for the base case.

The next part of the proof involves the step case, which takes into consideration the inductive assumption as true and proves that the theorem holds also for the next step of the function. More in detail, let the condition

$$s_j(n) \geq s_k(n) \tag{3.18}$$

for the step n be true. If

$$s_j(n+1) \geq s_k(n+1) \tag{3.19}$$

then the theorem holds.

The correct approach to do it is to consider all the three sub-cases: the next seat is assigned to p_j (1), the next seat is assigned to p_k (2), the next seat is not assigned to any of the two parties (3).

Sub-case (1). The stronger party, p_j, gets the seat and as a consequence

$$s_j(n+1) = s_j(n) + 1.$$

While p_j gained one seat, on the contrary p_k did not, therefore

$$s_k(n+1) = s_k(n).$$

and it is allowed to inherit

$$s_j(n+1) > s_j(n) \geq s_k(n).$$

therefore, since $s_k(n+1) = s_k(n)$,

$$s_j(n+1) \geq s_k(n+1).$$

This proves that for this sub-case the theorem holds.

Sub-case (2). In this sub-case, p_k wins a seat. This is the most tangled case to prove, because the win of a seat by p_k could apparently bring a negation of the thesis. Therefore, only for this case, it is needed to split it into two sub-cases, namely (2a) and (2b), dividing the inductive hypothesis.

$$s_j(n) > s_k(n) \quad \wedge \quad s_j(n) = s_k(n)$$

Sub-case (2a). In this sub-case, it is possible to assume that $s_j(n)$ is strictly greater than $s_k(n)$

$$s_j(n) > s_k(n)$$

but, since the context is still case (2), then p_k is winning a seat,

$$s_k(n+1) = s_k(n) + 1 \tag{3.20}$$

while p_j is not,

$$s_j(n+1) = s_j(n) \tag{3.21}$$

Since s is a vector of natural numbers, I can write

$$s_j(n) > s_k(n) \implies s_j(n) \geq s_k(n) + 1 \tag{3.22}$$

Therefore, since (3.29) and (3.30), by substitution

$$s_j(n+1) \geq s_k(n+1)$$

Sub-case (2b). This case takes the part of the inductive hypothesis in which

$$s_j(n) = s_k(n).$$

Theoretically, it would make sense to say that p_k would end up with more seats than p_j but if $s_j(n) = s_k(n)$, then

$$\lambda(s_j(n)) = \lambda(s_k(n))$$

According to the assumption of case (2), p_k won. This means that

$$f(p_k, n+1) \geq f(p_j, n+1)$$

but recalling eq. (3.8), it is possible to substitute and have

$$\frac{v_k}{\lambda(s_k(n))} \geq \frac{v_j}{\lambda(s_j(n))} \implies v_k \geq v_j.$$

This is going against of the hypothesis of the theorem itself, eq. (3.16). As a consequence, this whole case is contradictory and therefore the thesis is proven.

Sub-case (3). Neither p_j nor p_k win a seat. This case is the most trivial, since both parties will stay with their number of seats, so

$$s_j(n+1) = s_j(n),$$

$$s_k(n+1) = s_k(n).$$

Therefore, using (3.18) and the equivalences above, it is proven that

$$s_j(n+1) \geq s_k(n+1).$$

□

Concordance property, in summary, provides a strong and important characteristic in the context of divisor method, ensuring one of the bases of the Social Choice Theory: the candidate with more preferences wins.

4 Implementation in Isabelle

This chapter completely encompasses the step from theory to practical implementation, shifting our focus to the actual realization of the divisor method within the Isabelle/HOL framework.

This crucial step links the abstract concepts explored until this point, mostly in Chapter 3, with concrete application, with an in-depth examination of the encoding of the divisor method, which is a pivotal component in the voting theory. Using Isabelle, an important step is to adapt theoretical principles into executable code. This way, it is possible to give a detailed exploration of the implementation process.

The sections in the chapter try to provide a comprehensive analysis of the Isabelle/HOL implementation of the divisor method. With this analysis, the main focus is the intersection of theoretical foundations and practical application in the domain of the Social Choice Theory.

Moreover, this chapter will also show the obstacles that have been encountered in developing this implementation with respect to the theoretical counterpart, demonstrating the differences that emerged through this advancement due to the Isabelle/HOL tool and the need to adapt a theoretical concept to practical groundwork.

Along with these difficulties, the corresponding solutions will be presented and explained, trying to keep the resolution simple and understandable without any experience of the topic.

4.1 Definitions and Basic Functions

In the first section of this chapter, a first approach within the Isabelle/HOL framework is made, bringing the foundational concepts of our theoretical framework into a rigorous and executable form.

Parties, votes and seat, among other components, are the basic entities that constitute the backbone of our analysis, so it's important to give precise definition within the expressive confines of Isabelle.

The importance of the development of these components is crucial, since all the next functions, submodules and modules will be built above these elements, so an erroneous analysis of this basis would bring disorderly and disjointed modules, making the work much more hard.

Putting these fundamental parts in a formal representation shapes the groundwork for a correct exploration of voting systems and their properties.

The first basic implementation presented is the list of the parties. Indeed, this list is necessary in order to keep track of all the parties in the election, also the ones that don't have any seats allocated.

The aim of this thesis is to keep things as simple as it is possible, therefore it is implemented just as a list of polymorphic type. This way, it is possible to adapt this list to different cases and it is not strictly related to the case of study of this work but can be used by further works in different modules.

```
type_synonym 'b Parties = "'b list"
```

After the parties, a pivotal component is the number of votes belonging to each party. This component required a transformation from the original framework's Profile. Using new functions, the preferences for each candidate (in this case, each party) is aggregated and brought into a list of rational numbers.

The same list will be used for what it are named Fractional Votes, which is the list containing the number of votes divided by the factors related to the number of seats of each candidate.

```
type_synonym 'b Votes = "'b list"
```

On the other hand, Seats has been implemented as a function to underline the importance of the seats, that are unique.

At the beginning of this analysis, another formalization was developed: the first implementation provided a function with an aggregation of seats, a value associated to every party. As a consequence, this means that for every candidate, it was possible to retrieve only the total number of seats, therefore, seats were not individual elements, rather than an aggregation with respect to the party.

Due to this reason, another implementation, which works in the opposite way, has been preferred. In this case, every seat is assigned to one or more candidates (parties). According to this choice, every seat remains defined as an individual element, not losing its uniqueness by getting aggregated.

Through the function, it is possible to easily check which is the winner of which round, checking the *n-th* seat.

A set is not the optimal choice for associating specific values with distinct keys, such as mapping party names to votes or seats. Functions, on the other hand, provide a more natural and efficient solution for key-value mappings, allowing direct querying and retrieval based on keys.

This makes functions better suited for representing the correspondence between party names and their associated seats.

```
type_synonym ('a, 'b) Seats = "'a ⇒ 'b list"
```

Another definition useful for the module is the final Result. As stated in the Chapter 2, in the original framework the result is indeed a tuple of three sets of polymorphic type, mainly used to define the elected, rejected and deferred choices.

In this implementation, the type Result has been used to describe the state of the seats. The first set is designed for the seats that have been assigned, while the third one is used to identify seats that have not been allocated yet.

These different type synonyms are useful along with other parameter that will be used throughout the whole module. To group these different parameters, a record structure is defined to better handle these parameters.

Having briefly defined some of the fields of this record, it is further expanded with a set of indexes, to help the implementation of functions and the number of seats, that will be the focus of the termination condition in the method.

Furthermore, there is a list that indicates how many seats every part has and it is used to keep track of the number of seats.

Finally, the divisors have been implemented as a list, while on a theoretical plane these were introduced as a function.

The reason is purely practical: handling a list is easier and provides better code generation for Isabelle/HOL, with better functions to handle and manipulate it.

```
record ('a::linorder, 'b) Divisor_Module =
  res :: "'a::linorder Result"
  p :: "'b Parties"
  i :: "'a::linorder set"
  s :: "('a::linorder, 'b) Seats"
  ns :: nat
  v :: "nat Votes"
  fv :: "rat Votes"
  sl :: "nat list"
  d :: "nat list"
```

Abbreviation to retrieve the assigned seats from the Result type.

This record will be passed throughout the functions in higher level. During this process, every function will use and/or change the fields of the record according to its role.

It would be unnecessary to have the above record also in the final output, since most of these fields are used within the function but are useless for the complete comprehension of the outcome of the election.

That is why another record is introduced and used as final output. This record serves as an extension of the current *Result* type, providing additional but needed information.

```
record ('a::linorder, 'b) Divisor_Result =
  seats :: "'a::linorder Result"
  parties :: "'b Parties"
  seats_fun :: "('a::linorder, 'b) Seats"
  seats_list :: "nat list"
```

The above record only contains the basic information about the result of the election: the first field, namely *seats*, has the utility to show if there are any disputed seats, while the *seats_list* contains the number of seats for every party, according to the order in the *parties* list.

Finally, also the *Seats* function is provided. This would not be strictly necessary, since there is already *seats_list* but it has been kept in order to give the chance to manually check the winner of every round, in case of a double check or in case a comparison between variants will be evaluated.

4.2 Aggregating Preferences into Votes

As already mentioned above, the first obstacle encountered was about the preferences of the voters: currently, in the framework every voter expresses its preferences through a list of preference, while in the divisor method it is needed to have a single preference.

Therefore, the first function to be implemented is recursively iterating through all the ballots contained in the Profile, counting the number of votes for one candidate.

```
fun cnt_votes :: "'a ⇒ 'a Profile ⇒ nat ⇒ nat" where
  "cnt_votes p [] n = n" |
  "cnt_votes p (px # pl) n =
     (case (count_above px p) of
        0 ⇒ cnt_votes p pl (n + 1)
      | _ ⇒ cnt_votes p pl n)"
```

It would be pointless to analyze in detail how every function has been structured. However, it is important to briefly explain the above function since it is the first step towards the method described in this work.

The function, for a given party is iterating through the ballots, counting how many parties are above the aforementioned party in the preference list.

If the output is zero, it means that the party is the highest in the preference list (i.e., the ballot) and therefore its number of votes gets increased. Otherwise, the number of votes remains unchanged and the next ballot is evaluated.

The following step is then to define a function that is iterating through the list of all Parties and, for every party, calling the function above.

```
fun calc_votes :: "'a Parties ⇒ 'a Parties ⇒ 'a Profile ⇒ nat Votes ⇒ nat Votes"
where
  "calc_votes [] fp pl votes = votes" |
  "calc_votes (px # ps) fp pl votes =
      (let n = cnt_votes px pl 0;
       i = index fp px in
      calc_votes ps fp pl (list_update votes i n))"
```

After this function, the basic components needed for the module, parties and votes, are available and it is possible to start to implement the divisor module.

4.3 Assignment of a seat

In this section, the implementation of a single round, so the assignment of a seat and the subsequent adjustment of values, will be described. The theoretical formalization provided in Chapter 3 states that the first step in the round is retrieving the list of winners through all the parties, retrieving the one with maximum value of fractional votes, as described in * add def winners ref *.

The function *get_winners* is helped through two other functions, *max_p* and *max_v*.

```
fun max_v:: "'a::linorder Votes ⇒ 'a::linorder" where
"max_v v = Max (set v)"

fun max_p:: "'a::linorder ⇒'a::linorder Votes ⇒ 'b Parties ⇒ 'b Parties"
  where
"max_p m v ps = filter (λx. v ! (index ps x) = m) ps"

fun get_winners :: "'a::linorder Votes ⇒ 'b Parties ⇒ 'b Parties" where
  "get_winners v p = (let m = max_v v in max_p m v p)"
```

As already formalized in theory, the function *get_winners* has in output a list of Parties, of polymorphic type to keep the implementation generic, which represents the list of parties that have their fractional votes equal to the maximum, calculated in *max_v*.

According to the formalization of the divisor method given in Chapter 3, it consists of two cases: in the first one, there are enough seats for all the winners, while in the second one there are not, bringing to a tie.

Two different functions have been developed separately handling the two cases.

```
fun upd_round :: "'b Parties ⇒ ('a::linorder, 'b) Divisor_Module ⇒
                      ('a::linorder, 'b) Divisor_Module"
```

The function *upd_round* is handling whatever concerns within one single round. This function will be executed in the case there are enough seats, so one winner will be chosen and will have a new seat assigned.

As already mentioned above, the strategy is the following: the first element of the list of winners will be the one that will get the next seat. Since there are enough seats for everyone, it is not a concern if the other will get their seat later.

Moreover, this function will update the *Result* type moving the next seat from the disputed to the assigned set and calculate the new *fractional vote* related to the winner, since its number of seats have been increased. By doing so, on the next round when the winners will be calculated again, this party will not be in the list anymore and it will be possible to assign the seat to another party. This process will last until the list ended and, after some rounds, another list of winners will start the process.

```
fun break_tie :: "'b Parties ⇒ ('a::linorder, 'b) Divisor_Module ⇒
                      ('a::linorder, 'b) Divisor_Module"
```

This second function will be called when there are not enough seats for all the winners.

As an example, let us assume that there are three remaining seats but there are five winners with the same value of *fractional vote*. If the first function would be called, the last two parties will be left without the seat they deserve, making the method not anonymous, since it would be influenced by the other of parties in the list.

As a consequence, the final seats will be disputed between the parties, leaving them in the third set of the *Result* type.

Therefore, a more generic function calculates the list of winners and confronts its length with the remaining seats, calling *upd_round* or *break_tie* according to the case.

```
fun assign_seat :: "('a::linorder, 'b) Divisor_Module
                        ⇒ ('a::linorder, 'b) Divisor_Module"
  where
"assign_seat rec = (
      let ws = get_winners (fv rec) (p rec) in
      if length ws ≤ ns rec then
        let rec' =  (upd_round [hd ws] rec) in
                    (|res = (res rec'),
                         p = (p rec'),
                         i = (i rec'),
                         s = (s rec'),
                         ns = ((ns rec') - 1),
                         v = (v rec'),
                         fv = (fv rec'),
                         sl = (sl rec'),
                         d = (d rec')
                        |)
      else
         let rec'' = (break_tie ws rec) in
                       (|res = (res rec''),
                         p = (p rec''),
                         i = (i rec''),
                         s = (s rec''),
                         ns = 0,
                         v = (v rec''),
                         fv = (fv rec''),
                         sl = (sl rec''),
                         d = (d rec'')
```

It is possible to note that in case there are enough seats available, the function *upd_round* will be executed and the number of seats is decreased by one.

Nevertheless, differently from the theoretical formalization, the function handling the tie is not decreasing the number of seats by one and deferring the current state: instead, the number of seats is directly set to zero, bringing the algorithm to an end.

In fact, the algorithm will end when the seats are zero and through this strategy it is possible to save other fruitless calculations. This approach will be clearer later, when the whole module will be presented.

4.4 Divisor Module

After implementing the foundational groundwork and all the submodules concerning one single round, encompassing essential definitions and the structural framework of our method—defining parties, votes, and seats, the next step is the implementation of a function that executed more rounds, building the core of the module.

The process iterates, scrutinizing a termination condition that guides the decision to continue assigning seats or exit the allocation loop.

```
function loop_div ::
  "('a::linorder, 'b) Divisor_Module ⇒ ('a::linorder, 'b) Divisor_Module"
  where
  "ns r = 0  ⟹ loop_div r = r" |
  "ns r > 0 ⟹ loop_div r = loop_div (assign_seat r)"
```

As it can be retrieved from the code, the loop is recursive and is calling the function *assign_seat* until the number of seats reaches zero.

Without delving into the details of the implementation, it is worth pointing out that in order to write and subsequently use this loop in other functions, it was necessary to develop the proof of its termination. Since *assign_seat* is decreasing the number of seats by one, it was then proved that the loop will eventually end.

Finally, this loop can be encapsulated in a cleaner function as it follows.

```
fun divisor_method:: "('a::linorder, 'b) Divisor_Module ⇒ 'b Profile ⇒
                  ('a::linorder, 'b) Divisor_Module"
  where
"divisor_method rec pl = (
    let sv = calc_votes (p rec) (p rec) pl (v rec);
    sfv = start_fract_votes sv
    in loop_div (|
             res = res rec,
             p = p rec,
             i = i rec,
             s = s rec,
             ns = ns rec,
             v = sv,
             fv = sfv,
             sl = sl rec,
             d = d rec
            |))"
```

In this function, a record with all the fields concerning the algorithm is passed as a parameter, along with the list of ballots in the Profile.

Then, the starting votes are calculated through the Profile and the loop is executed.

4.5 D'Hondt and Sainte-Laguë

At the end of this practical implementation, it is possible to define the two methods analyzed in the theoretical formalization.

As already said in Chapter 3, the two variants are similar. In fact, the only difference between D'Hondt and Sainte-Laguë is in the choice of divisors. Due to this reason, it was possible to write only one implementation up to this level keeping the functions independent of the values of the divisors.

At this point, it is needed to write two different functions for the two modules.

```
fun dhondt_method :: "'b Parties ⇒ nat ⇒ 'b Profile ⇒
                  (nat, 'b) Divisor_Result"
  where
"dhondt_method partiti nseats pr =
    (let rec = build_record partiti nseats;
      result = divisor_method (rec(|d := upt 1 (ns rec)|)) pr in
        (| seats = res result,
           parties = p result,
           seats_fun = s result,
           seats_list = sl result|))"
```

Making this function more clear, the *divisors* field, namely *d*, is created with natural numbers starting from one up to the total number of seats. With this adjustment, even if a party would gain all the available seats, the function will have enough divisors to calculate all the different *fractional vote*.

In example, for ten seats, it will provide ten divisors as follows

$$[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]$$

On the other hand, the Sainte-Laguë method is slightly different.

```
fun saintelague_method:: "'b Parties ⇒ nat ⇒ 'b Profile ⇒
                  (nat, 'b) Divisor_Result"
  where
"saintelague_method partiti nseats pr =
  (let rec = build_record partiti nseats;
      result = divisor_method (rec(|d := filter (λx. x mod 2 = 1) (upt 1 (2*ns rec))|))
pr in
        (| seats = res result,
           parties = p result,
           seats_fun = s result,
           seats_list = sl result|))"
```

In this case, the *divisors* will be created taking only the odd numbers and, as above, there will be a number of divisors equal to the total number of seats.

Therefore, for ten seats, the *divisors* will be

$$[1, 3, 5, 7, 9, 11, 13, 15, 17, 19]$$

Finally, in these two functions only the starting parameters will be provided: the list of parties, the number of seats to allocate and ballots, where the function *build_record* initializes the record with all the needed parameters.

It is worth noticing that the implementation has been developed in such a way that if a party is in the list of candidates but has no votes, it will get zero votes.

Viceversa, if a party is not in the candidates and appears in the ballots it will not be considered in the election.

As the culmination of this comprehensive process, a Result emerges as the output of the module, encapsulating the culmination of seat assignments based on the principles encoded within the divisor method.

Along with the Result output, which describes the current situation of the seats, whether they are assigned or not, additional information will be provided.

As a matter of fact, also the list of parties, along with another list indicating how many seats every party has, will be in the final output.

Moreover, in the outcome there is a function, namely *seats_fun*. Through this function, it is possible to inspect every round, checking which party is the winner.

In case of a tie, the seats disputed will be displayed in the third set of the *Result* type. Inspecting these seats through *seats_fun*, it is possible to check the tied parties that are contending that seat.

5 Proof of properties

This chapter represents the other core point of this work, after the implementation of the divisor method. In fact, it embarks on the practical implementation of the properties analyzed in Chapter 3 into the tool Isabelle/HOL.

These properties, as described earlier, are three different pivotal components of Social Choice Theory that play an important role in defining the fairness of an electoral module.

Some of these properties, such as anonymity and monotonicity, have already been implemented in the current framework. Nonetheless, the additional work provided in this thesis develops a new branch of Social Choice Theory, since the only methods already implemented in the framework are the ones concerning a single winner.

On the contrary, in the context of this work the properties have been adapted to methods regarding allocations of seats and multiple winners.

Due to this reason, it has not been possible to use the common properties already present in the framework. Nevertheless, these were the starting point for the properties developed for the divisor method.

Moreover, the third property analyzed is a completely new addition to this framework. In fact, concordance property is crucial in the context of proportional representation, ensuring that voting for a candidate is not damaging it in any way.

This property could seem trivial for an electoral method but there have been different cases in the past in which gaining votes was not bringing benefits to the candidate but on the contrary, it was damaging them. Some of these cases will be presented in Chapter 7, dedicated to conclusions.

Instead, focusing again on the proofs, before entering into the single proofs of the properties, it is worth to give an explanation of the approach followed.

In fact, these properties could not merely be written and proved as it has been done in Chapter 3. When entering Isabelle/HOL domain, it is not possible to skip any step, relying on the reader's knowledge for some passages of the proof. Instead, it is mandatory to accurately define all the components and ensure the semantic and syntactic correctness of every step.

Due to this reason, lower level lemmas were needed in order to fully prove the theorem and to write the proof in a modular and cleaner way.

These sections will not contain all the lemmas and cases actually developed during the work concerning this thesis, since they will be in the appendix.

Nevertheless, it is reasonable to face some of these lemmas, with the merely intention to deliver a general idea of the approach used and discuss about important steps in the proofs.

As an example, the first lemma chosen to be analyzed in this section is regarding the winners of the round.

```
lemma max_eqI_helper:
  assumes
"x < length l" and
"length l = length l'" and
"l ! x = Max(set l)" and
"l' ! x > l ! x" and
"⋀y. y ≠ x ⟹ y < length l ⟹ l' ! y ≤ l ! y"
shows "l' ! x = Max (set l')"
proof(rule ccontr)
  assume "l' ! x ≠ Max (set l')"
  then have "∃y. y ≠ x ⟶ y < length l ⟶ Max(set l') = l' ! y"
    by auto
  then have "⋀y. y < length l ⟹ l ! y ≤ Max(set l)"
    by simp
  then have "⋀y. y ≠ x ⟶ y < length l ⟶ l' ! y ≤ Max(set l)"
    by (meson assms(5) dual_order.trans)
  then have "Max(set l') > Max(set l)"
    using assms dual_order.strict_trans1
          get_winners_weak_winner_implies_helper by metis
  then have "⋀y. y ≠ x ⟶ y < length l ⟶ l' ! y < Max(set l')"
    using ‹⋀y. y ≠ x ⟶ y < length l ⟶ l' ! y ≤ Max (set l)› by fastforce
  then show False
    using ‹∃y. y ≠ x ⟶ y < length l ⟶ Max(set l') = l' ! y›
          ‹⋀y. y ≠ x ⟶ y < length l ⟶ l' ! y < Max(set l')› assms
          ‹⋀y. y ≠ x ⟶ y < length l ⟶ l' ! y ≤ Max (set l)›
          ‹l' ! x ≠ Max (set l')› in_set_conv_nth leD linorder_le_cases
          max_val_wrap_eqI_helper order.strict_trans2
  by (metis (no_types, opaque_lifting))
qed
```

This lemma states that if the votes of a candidate (informally called x) gets increased and, for every other party, the number of votes remains the same or decreases, then this party is still a winner of the round.

The above proof is important because of various reasons. First, differently from the most of the lemmas, this one has been approached and developed by contradiction.

Furthermore, this lemma is of pivotal importance for the proof of different cases encountered lately when proving the monotonicity.

Moreover, another lemma, not present in this section but that can be read in the appendix, states that this party is not only a winner but it is indeed the only one, after increasing its votes.

Instead, the following lemma proofs that regardless of the result of the round, the assignment of a seat cannot bring the number of seats of any party to be decreased.

```
lemma assign_one_seat_mon:
assumes "ind < length (sl rec)"
  shows "sl (upd_round winner rec) ! ind ≥ (sl rec) ! ind"
proof -
```

```
    define seat new_s new_as new_fv ind curr_ns new_sl new_di
    where "seat = Min (disp_r (res rec))" and
          "new_s = update_seat seat winner (s rec)" and
          "new_as = ass_r (res rec) ∪ {seat}" and
          "new_fv = upd_votes winner (rec⦇s:= new_s⦈)" and
          "ind =  index (p rec) (hd winner)" and
          "curr_ns = (sl rec) ! ind" and
          "new_sl = list_update (sl rec) ind (curr_ns + 1)" and
          "new_di =  disp_r (res rec) - {seat}"
  have "(upd_round winner rec) =  (⦇res = (new_as, {}, new_di),
             p = (p rec),
             i = (i rec),
             s = new_s,
             ns = (ns rec),
             v = (v rec),
             fv = new_fv,
             sl = new_sl,
             d = (d rec)
            ⦈)"
    unfolding upd_round.simps new_sl_def Let_def
    using nth_list_update_eq curr_ns_def ind_def new_as_def new_di_def
          new_fv_def new_s_def seat_def by fastforce
  then have "sl (upd_round winner rec) = new_sl"
    by simp
  then have "sl (upd_round winner rec) ! ind = new_sl ! ind"
    by simp
  then have "... = (list_update (sl rec) ind (curr_ns + 1)) ! ind"
    using new_sl_def nth_list_update_eq by blast
  then show ?thesis
    using ‹sl (upd_round winner rec) = new_sl› assms(1) curr_ns_def le_add1 new_sl_def

          nth_list_update_eq nth_list_update_neq order_refl by metis
qed
```

This lemma, which could appear almost trivial on a theoretical plane, had instead to been proved by hand in order to be used in the following lemmas.

All of these lower level lemmas are the main difference between the theoretical approach and the practical work, since they had to be proved individually in the context of Isabelle/HOL, while on paper they were trivial and they have been assumed as ground truth.

In addition, they have been of fundamental importance in the development of the proofs, to better understand how to theoretically write the properties, in a loop process that brought cleanness to the proofs and the theorems.

5.1 Monotonicity

Having formalized the monotonicity property in Chapter 3, the current focus pivots towards concrete implementation within the Isabelle and theoretical constructs, meticulously outlined earlier, with the realm of executable Isabelle code.

The implementation tried to follow the theoretical formalization of the property, developing the different cases independently and then combining them together in order to define the whole theorem concerning the property.

Despite this, it was almost inevitable that due to differences between theoretical definition and practical implementation of the method, also the properties would have been affected and therefore differ in some parts.

The same cases structure has been preserved but in practice more cases had to be unfolded and developed. In fact, four different variables had to be taken into account developing this property.

The first two conditions were related to check whether there was or not a tie in both the first election and in the second, the one concerning the candidate with the increased votes.

The other two conditions were instead aimed to determine whether the party is indeed the winner or not in both contexts, before and after the increase of the votes.

The final theorem, that can be found in the appendix, proves monotonicity regarding the assignment of a seat, assuring that the process is monotone.

While the full proof will not be shown here, fig. 5.1 abstracts the structure of the proof and how the different cases have been handled.

The proof has been built with a modular approach, as mentioned earlier, based on different cases that can happen. To help the reading of the flowchart of fig. 5.1, the various conditions of the decision paths will be briefly explained.

Nevertheless, before analyzing the flowchart in detail, it is important to briefly describe the approach that has been followed.

In the development of this proof, the hypothesis made is that the increase of the votes of the party is treated as a different election which is giving another output of winners. Therefore, in the original election, before the increase of the votes of the party, the list of winners is defined as w.

Next, the votes of the party are increased, therefore bringing another outcome to the function that is searching the maximum factor between the parties.

As a consequence, the list of winners obtained after the increase of votes differs from the previous one (w) and therefore needs a different notation, namely w'. Finally, R expresses the remaining seats of the election.

The first case encountered is $|w'| \leq R$, which can be translated in checking if there are enough seats for the list of winners obtained after increasing the votes of the party in exam.

The second case encountered, $|w| \leq R$ is instead considering whether there are enough seats for the winners obtained before the votes have been increased.

The choice to check this conditions is the following: in case these conditions are not respected, therefore when a tie occurs, the number of seats of the party is not increased and therefore the proofs of these cases are straightforward.

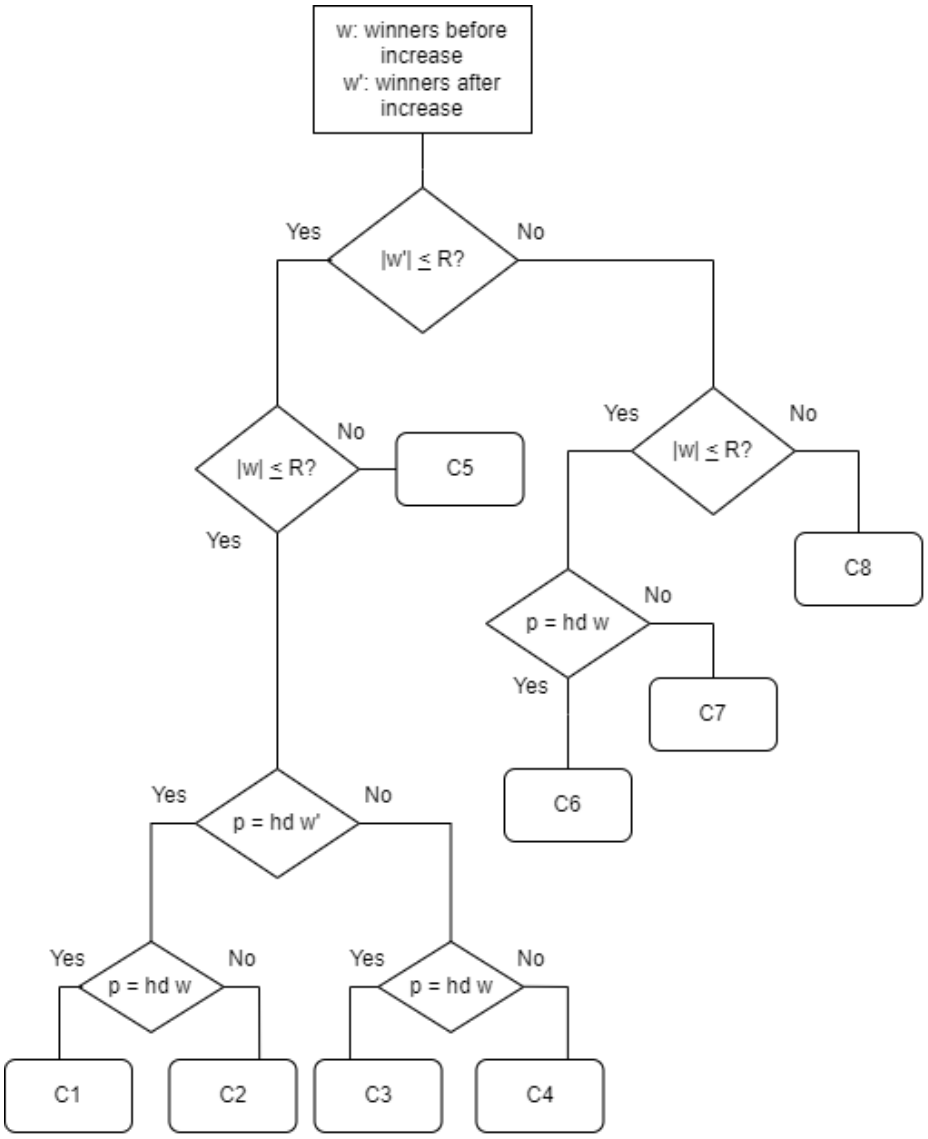

Figure 5.1: structure of monotonicity proof

In fact, if monotonicity is respected before the assignment of the new seat and the number of seats does not change (since a tie occurred), it is trivial to conclude that monotonicity property holds for this cases.

The above reasoning is reflected in the complexity of the proofs in Isabelle/HOL and can be observed in the flowcharts.

In fact, in the cases of a tie (as example C5, C8) the proof of monotonicity is not delving more into the other decision paths and can be soon demonstrated.

On the contrary, when there is not a tie, more variables have to be taken into consideration to correctly and fully prove that monotonicity holds for the assignment of a seat.

Next, examining the aforementioned cases in which a tie does not occurs, more cases have to been handled. In fact, the module has been built to assign the next seat to the first party in the list of winners and iterating the process.

As an example, if the party in consideration is the second in the list of winners, it will not get assigned the next seat *n* but the seat of the round after, indeed the *n+1*-th.

In other words, if the party has the maximum factor of this round and there are enough seats for every party in the list of winners, the party will not surely receive the very next seat but *eventually* it will receive the seat it deserves.

Going down on the left side of the flowchart, the next conditions met are the ones described above: *p = hd w'* refers to the party being the first in the list of *w'*, which was defined as the winners after the increase of votes.

Shortly after, the next condition met is *p = hd w*, which is instead checking if the party is the head of *w*, which is the list of winners before the increase.

The flowchart stops at the level of detail of these four decision paths but in some cases (as C3 and C6) other sub-cases had to be considered and complex and intricate proofs written through contradiction have been provided.

In summary, these different cases have all been proved individually and only after put together in intermediate level lemmas.

Without delving into the details of the implementation of every single case described above, it is worth analyzing only one to give an example of the work developed in this work.

```
lemma assign_seat_incre_case_FTqT:
  fixes
  v::"rat"
assumes
  "m' = Max(set (fv rec'))" and
  "winners = get_winners (fv rec) (p rec)" and
  "winners' = get_winners (fv rec') (p rec')" and
  "winners ≠ []" and "winners' ≠ []" and
  "p rec ≠ []" and
  "size (fv rec) = size (p rec)" and
  "size (fv rec') = size (p rec)" and
  "length (fv rec) = length (fv rec')" and
  "party ∈ set (p rec')" and
  "v' > v" and
```

```
  "fv rec ! index (p rec) party = v / of_int(d rec ! (sl rec ! index (p rec) party))"
and
  "fv rec' ! index (p rec) party = v' / of_int(d rec ! (sl rec' ! index (p rec) party))"
and
  "sl rec' ! index (p rec) party ≥ sl rec ! index  (p rec) party" and
  "d rec ! (sl rec ! index (p rec) party) ≠ 0" and
  "d rec ! (sl rec' ! index (p rec) party) ≠ 0" and
  "index (p rec) party < length (sl rec')" and
  "length (get_winners (fv rec') (p rec')) =
   length (filter (λx. (fv rec') ! x = m') [0..<length (fv rec')])"
  "index (p rec') party < length (fv rec')" and
  "index (p rec) party < length (sl rec)" and
  "index (p rec) party < length (fv rec)" and
  "⋀x. x ≠ index (p rec) party ⟹ (fv rec') ! x ≤ (fv rec) ! x" and
  "ns rec = ns rec'" and
  "length winners' > ns rec'" and
  "length winners ≤ ns rec" and
  "party = hd winners" and
  "p rec = p rec'"
shows "sl (assign_seat rec') ! index  (p rec) party ≥
       sl (assign_seat rec) ! index  (p rec) party"
proof -
  have "sl (assign_seat rec') = sl rec'"
    using assign_seat_break_tie_case assms(24) assms(3) by metis
  then show ?thesis
  proof(cases "sl rec' ! index  (p rec) party > sl rec ! index  (p rec) party")
   case True
   then have "sl rec' ! index (p rec) party = sl (assign_seat rec') ! index (p rec) party"
     using ‹sl (assign_seat rec') = sl rec'› by presburger
   then have "sl (assign_seat rec) ! index (p rec) party = sl rec ! index (p rec) party
+ 1"
     using assign_seat_seats_increased assms(2) assms(20) assms(25) assms(26) by blast
   then show ?thesis
     using Suc_eq_plus1 True ‹sl (assign_seat rec') = sl rec'› less_eq_Suc_le by metis
  next
   case False
   then have "(fv rec) ! index (p rec) party = Max(set (fv rec))"
     using assms(2) assms(26) assms(4) get_winners_not_in_win max_v.elims by metis
   then have "⋀y. y ≠ index (p rec) party ⟹ y < length (fv rec) ⟹
              fv rec ! y ≤ Max(set (fv rec))"
     by auto
   then have "Max(set (fv rec')) = (fv rec') ! index (p rec) party"
     using ‹fv rec ! index (p rec) party = Max(set (fv rec))› max_eqI False
           Fract_of_int_eq Fract_of_nat_eq divide_less_cancel int_ops(1)
           nat_less_le negative_eq_positive of_nat_less_0_iff of_nat_less_of_int_iff
           assms(11) assms(12) assms(13) assms(14) assms(16) assms(21) assms(22) assms(9)
     by (smt (verit, ccfv_threshold))
```

```
  then have "fv rec' ! index (p rec') party = m'"
    using assms(1) assms(27) by presburger
  then have "Max(set (fv rec')) > Max(set (fv rec))"
    using False ‹fv rec ! index (p rec) party = Max (set (fv rec))› assms(1) assms(11)

      assms(12) assms(13) assms(14) assms(16) assms(27) divide_strict_right_mono
      of_nat_le_0_iff by force
  then have "⋀y. y ≠ index (p rec) party ⟹ y < length (fv rec)
            ⟹ (fv rec) ! y < Max(set (fv rec'))"
    using ‹Max(set (fv rec')) > Max(set (fv rec))› assign_seat_incre_case_helper
     ‹⋀y. y ≠ index (p rec) party ⟹ y < length (fv rec)
            ⟹ (fv rec) ! y ≤ Max(set (fv rec))›
    by metis
  then have "⋀x. x ≠ index (p rec') party ⟹ x < length (fv rec') ⟹ (fv rec') !
x < m'"
    using assms(1) assms(22) assms(27) assms(9) leD linorder_le_less_linear
     max_v.simps max_val_wrap_lemma order_antisym_conv by metis
 then have "length (winners') = 1"
   using assms ‹Max(set (fv rec')) = (fv rec') ! index (p rec) party›
     ‹fv rec' ! index (p rec') party = m'›
     ‹⋀y. y ≠ index (p rec') party ⟹ y < length (fv rec') ⟹ (fv rec') ! y < m'›
     filter_size_is_one_helper_my_case_3 by metis
 then have "length winners' ≤ length winners"
   using assms(4) leI length_0_conv less_one by metis
 then show ?thesis
   using linorder_not_less assms(10) assms(23) assms(24) assms(25) by linarith
qed
qed
```

The above case, shown as an example, is proving the monotonicity property in the case where the party in consideration belongs to w but after increasing the votes a tie appears (in fig. 5.1, it is represented by C6).

On paper, it might seem almost trivial to prove the theorem through contradiction in this case. However, engaging in the practical work reveals new challenges and necessitates proving every step that might be skipped in a paper-based proof.

This necessity highlights the true aim of the thesis: crafting the proofs and translating them into a practical application, navigating the various obstacles inherent in this process.

This meticulous work underscores the difference between theoretical proofs and their implementation within a formal verification system like Isabelle/HOL.

The example discussed above represents only one sub-case among the many that have been individually explored and proved to construct the comprehensive theorem.

Each of these sub-cases involves detailed scrutiny and validation of steps that are often considered trivial in theoretical discussions.

The overall Divisor Method, encompassing the full rule as defined in both theory and practical implementation, essentially involves iterating a function that assigns seats, a process that has been rigorously proved to be monotone. This iterative function underpins the broader method, ensuring its consistency and reliability.

Through this iterative proof process, we can deduce the monotonicity of the entire Divisor Method. This conclusion is not merely theoretical; it has been substantiated through rigorous formal verification, adding a layer of robustness to the theoretical claims.

In conclusion, while theoretical proofs might simplify or skip certain steps, the practical implementation within a formal system demands comprehensive validation of each step.

This rigorous approach not only confirms the theoretical properties of methods like the Divisor Method but also ensures their applicability in real-world scenarios, where precision and reliability are paramount.

Therefore, it is possible to conclude that the Divisor Method is not only theoretically but also practically verified to be monotone.

5.2 Concordance

This property, differently from the other two analyzed in this thesis, is a completely new addition to the framework.

Concordance, described in section 3.3.3, states that if a party has more votes than another, it should not end with less seats assigned than the weaker party.

This property is of fundamental importance in the process of defining the fairness of a voting method, being one of the pivotal milestones in the Social Choice Theory.

In a similar approach that has been successful for the monotonicity property, the cases defined in the concordance proof have been translated into Isabelle/HOL.

Therefore, the theorem has been broken down in different cases and sub-cases, in a tree structure shown in fig. 5.2 that finally brought to the assembly of the final proof.

The different cases have been proved individually in the theorem, which can be found in its entirety in the appendix.

The initial aim for the implementation of this theorem into Isabelle/HOL was to be the closest to its theoretical part, trying to minimize the differences of the two sides.

Nevertheless, as it had already happened for the monotonicity property, the theoretical and practical parts diverge because of the differences in the implementation of the Divisor Method and its formalization given in Chapter 3.

To navigate the structure of the proof shown in fig. 5.2, it is important to notice that R indicates the remaining number of seats, while sp indicates the stronger party and wp the weaker party.

The first condition met in the decision tree checks whether a tie occurred in the searching for the winners. If there is a tie, it is quite trivial to conclude that, assuming the concordance property was respected before the assignment of this seat, it will continue to be true after the execution of the function.

In fact, in case of a tie, no new seats will be assigned and the concordance property will be respected.

On the contrary, when there are enough seats, the cases to be considered are two: the first one, $sp = hd\ w$ indicates whether the stronger party is the first in the winners and therefore will get the seat assigned.

If not, the other condition will be evaluated, indeed $wp = hd\ w$, to check if the weaker party is instead the winner of the round.

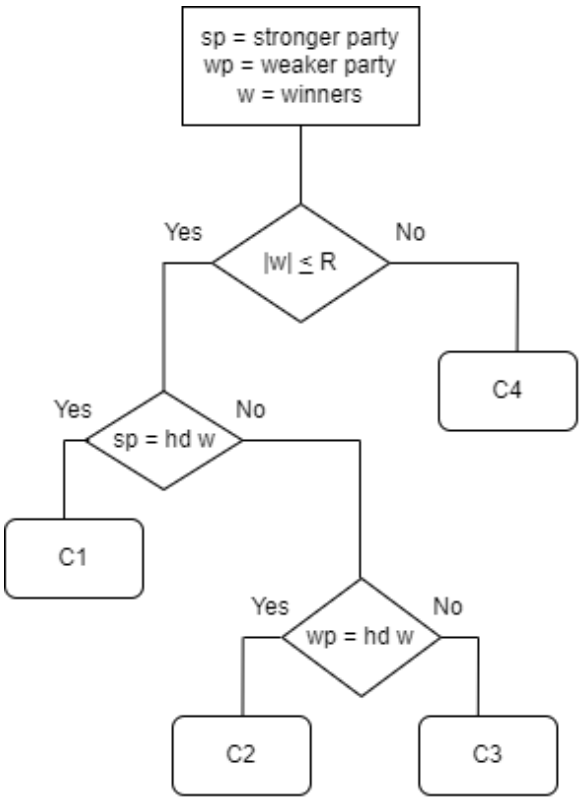

Figure 5.2: structure of concordance proof

In addition, even if the tree shown in this paragraph entails only the conditions described above, in the most complicated and problematic cases, other sub-cases had to be considered to better handle the proof, as in the case of C2.

In this paragraph, only one lemma will be explained and shown in its entirety as an example, while the whole set of proofs that helped to prove and form the final theorem are in the appendix.

The following lemma proves that, in case of the weaker party winning the next round, then the stronger party will in every case have more seats than the weaker after the aforementioned round. To keep things clear, it reflects the case C2 of the decision tree.

It strictly follows one of the cases of concordance proof written in section 3.3.3. In fact, the proof in Isabelle is split into two "cases", that correspond to the cases **(2a)** and **(2b)**.

```
lemma assign_seat_ccontr:
  fixes
  v1::"rat" and v2::"rat"
assumes
  "index (p rec) party1 < length (fv rec)" and
  "v1 > v2" and
  "party2 = hd ( get_winners (fv rec) (p rec))" and
  "fv rec ! index (p rec) party1 = v1 / of_int(d rec ! (sl rec ! index (p rec) party1))"
and
  "fv rec ! index (p rec) party2 = v2 / of_int(d rec ! (sl rec ! index (p rec) party2))"
and
  "d rec ! (sl rec ! index (p rec) party2) ≠ 0" and
  "sl rec ! index (p rec) party1 ≥ sl rec ! index (p rec) party2" and
  "get_winners (fv rec) (p rec) ≠ []" and
  "index (p rec) party1 < length (sl rec)" and
  "index (p rec) party2 < length (sl rec)"
shows "sl (assign_seat rec) ! index (p rec) party1 ≥
       sl (assign_seat rec) ! index (p rec) party2"
proof(cases "length (get_winners (fv rec) (p rec)) ≤ ns rec")
  case True
  then show ?thesis
    using True assign_seat_helper_lemma_cond_ver assms(1) assms(8) assms(9)
          assms(10) assms(2) assms(3) assms(4) assms(5) assms(6) assms(7) by metis
next
  case False
   define rec''
    where
     "rec''= break_tie ( get_winners (fv rec) (p rec)) rec"
   have "sl (assign_seat rec) = sl rec''"
     using False rec''_def by simp
   then show ?thesis
     using assms(7) break_tie_lemma rec''_def by metis
qed
```

This lemma has been chosen as an example to show that in some cases, the distance between the theoretical formalization of the proof and its practical translation is almost not perceivable.

In this case, mostly, there is a complete correspondence of the theoretical and practical components.

Furthermore, it is important to recognize that the lemmas underpinning the concordance theorem extend beyond the primary ones, namely C1, C2, C3, and C4.

In reality, a comprehensive set of lemmas supports the foundational proofs, addressing low-level functions, such as the identification of maximal factor between the parties. These auxiliary lemmas invoke properties that might seem trivial and thus were not explicitly stated on the theoretical proofs.

Conversely, in Isabelle/HOL, every logical step, no matter how elementary, must be rigorously documented and proven. This meticulous approach ensures the accuracy and reliability of the formal proofs.

Isabelle/HOL offers built-in tools that streamline the proof development process by automating certain straightforward steps.

Despite these aids, the level of detail required in the lemmas within the appendix, and generally throughout Isabelle, far exceeds that of the formal proofs presented in Chapter 4.

The theorem discussed herein comprehensively addresses all scenarios articulated in the theoretical framework of the proof, thereby affirming the concordance of the divisor method.

The definitive Concordance theorem is detailed in the appendix, focusing on a single round, akin to the monotonicity theorem mentioned in the preceding section. However, as indicated in the formalization, the Divisor Method essentially involves iterating this function.

Thus, if concordance is established for a single round, it follows logically that the entire divisor module can be inferred to be concordant. This deductive process ensures that the broader application of the divisor method maintains the same properties of concordance demonstrated in the individual iterations.

5.3 Anonymity

The anonymity property, as described in the previous theoretical chapter, states that if the order of parties is changing, then the outcome of the election should not change.

While this property can appear easy to prove on a theoretical level and it is trivial to be true, it has been quite challenging to prove it in Isabelle/HOL.

This happened because in the implementation of the divisor method the parties have been built as a list of polymorphic type. As a consequence, without delving into the specific types of Isabelle/HOL, different obstacles started to rise because to work on permutations the best choice would have been to build the list of parties as a multiset (a set where multiplicity of elements matters), rather than list.

Due to restricted time, the following solution has only been developed but not only implemented: the approach would be to make the polymorphic type a narrower linearly ordered one.

This way, it would be possible to agilely convert the list into a multiset and vice-versa without encountering problems and, therefore, proving anonymity.

Despite this, it is worth to briefly analyze the function counting the votes.

```
fun calc_votes :: "'a Parties ⇒ 'a Parties ⇒ 'a Profile ⇒ nat Votes ⇒ nat Votes"
where
  "calc_votes [] fp pl votes = votes" |
  "calc_votes (px # ps) fp pl votes =
      (let n = cnt_votes px pl 0;
       i = index fp px in
      calc_votes ps fp pl (list_update votes i n))"
```

For every party, the function above is executing the following function to calculate the number of votes over the preference list.

```
fun cnt_votes :: "'a ⇒ 'a Profile ⇒ nat ⇒ nat" where
  "cnt_votes p [] n = n" |
  "cnt_votes p (px # pl) n =
     (case (count_above px p) of
        0 ⇒ cnt_votes p pl (n + 1)
      | _ ⇒ cnt_votes p pl n)"
```

As it can be seen, the process of calculating the votes is completely independent of the order of parties, that will end with the same number of votes regardless of the order they are listed.

Therefore, even without a formal proof written, it is not hard to explain that the Divisor Method is indeed anonymous.

Nevertheless, a formal proof could be added, as said before, adapting the type of the list of parties. This update would certainly help the proof of anonymity.

6 Evaluation

This chapter will embark on a detailed analysis of the methodology that brought to the development of the implementation of the theoretical work into the tool Isabelle/HOL, elaborating on the approach used within Isabelle/HOL and the results obtained through the approaches and the different choices that have been made.

Isabelle is a powerful tool that allows to define reusable components and modules, giving the chance to prove theorems and properties over some functions through different types of proofs.

The implementation in Isabelle has been the focus of this work and it has been the point where the most obstacles have been met, due to different reasons.

First of all, the learning of a new tool always brings some obstacles, given from the syntax and the differences that start to arise from the theoretical formalization and the planned execution.

Furthermore, the essence of Isabelle as a tool for theorem proofs requires the most accurate definition of the components, their types and the definition of the properties that need to be proved.

In other words, with Isabelle every meticulous step has to be correct and precisely defined in order to correctly step into the proofs.

This meticulousness has been the first important obstacle during this work, since a high level of specification and details was required. Mostly at the beginning of this thesis, various steps, type definitions and assumptions were missing in the code due to lack of scrupulousness.

As a consequence, the practical implementation into Isabelle was not fully representative of the study case which had been formalized in the theoretical part and, as a result of this lack of specification, the process of writing proofs was really complex.

This demanded a better study of the tool and a subsequent substantial update of the practical implementation. Different additions were included and after this update the process of developing the proofs was smoother and cleaner.

As a second point, the work provided in this thesis lays on the foundation of the already existing framework provided by Diekhoff, Kirsten, and Krämer, 2020.

Despite this, the current state of the framework does not have any method similar to the one analyzed in this document, indeed the case of proportional representation.

In fact, until this moment, the framework only embodies cases of a single winner outcomes, while the Divisor Method consists of multiple rounds, where each round has a winner which gains a seat.

Due to this state of the framework, almost all the code had to be fully implement from scratch, not having the chance to rely on existing modules.

This state of the framework, although, has not been perceived as a limitation, rather than as an opportunity.

In fact, the addition of the Divisor Method started an almost new branch of the framework and in the future, starting from this work, new implementations based on election with multiple winners can be easily added.

Without showing the whole session graph which would be only confusing, fig. 6.1 is showing the new additions of the framework.

After analyzing how the framework had to been adapted to fit the case developed in this work, it is worth to briefly analyze the proofs that have been written.

Another obstacle that has been complex to overcome was the one regarding the modular structure of the method and how this affected the proofs.

As a consequence of the modular approach, also the proofs had to be modular and, before proving the properties that are the focus of this work, a set of different properties and lemmas had to be proved to facilitate the process of proving the aforementioned properties.

Various lemmas have been proved and helped to simplify the most important theorems, avoiding to have a monolithic proof rather than smaller proofs that work together.

The strategic use of smaller lemmas to aid in proving the main lemmas and theorems creates a network of dependencies among these smaller lemmas.

This interconnected set of dependencies will be detailed in this section, illustrating how they contribute to the overall proof structure. By breaking down complex proofs into smaller, more manageable lemmas, the process becomes more systematic and easier to follow.

These dependencies are crucial because they simplify the higher-level lemmas and theorems, making the proofs more accessible and comprehensible.

Each smaller lemma builds upon the previous ones, ensuring that every step is logically sound and meticulously verified. This methodical approach not only streamlines the proof process but also enhances the clarity and reliability of the results.

$$\frac{P \text{ is in the winners} \qquad P \text{ gets its votes increased}}{P \text{ becomes the only winner}}$$

As an example, the above inference has been used in the monotonicity property. Nevertheless, even if the above property is trivial on a theoretical plane, it has been proved in Isabelle/HOL.

Another low level inference that was deeply used in the proofs of the theorems is the following.

$$\frac{S \text{ has more votes than } W \qquad S \text{ and } W \text{ have the same seats}}{W \text{ is not in the list of winners}}$$

In the addition of the framework, various smaller lemmas regarding the winners or the increment of the votes have been developed, since they refer to smaller functions that have been kept to a generic level through polymorphic types.

In case of a future work, these smaller modules can be taken and reused adapting them for another case without having any obstacles. In addition to this, these lemmas will still be respected and therefore will it will be possible to use this lemmas in later proofs.

Later, moving towards an higher level of the implementation and delving closer to the theorems, the dependencies of the proofs resemble the cases described in Chapter 5.

$$\frac{S \text{ has more votes than } W \qquad W \text{ is the winner} \qquad S \text{ has at least the same seats as } W}{\text{After the assignment of the seat, } S \text{ still has at least the same seats as } W}$$

The development of the implementation of the divisor method provided in the appendix went under numerous changes. In fact, the process was not as straightforward as expected and different choices had to be made.

Throughout the whole thesis, both the implementation and the theoretical part have been heavily affected by the characteristics of Isabelle/HOL, first of all the syntax and the possibilities.

The whole work was indeed an iterative process, with the implementation, starting from the most basic definitions to the higher level components. The core elements of the algorithm, such as the assignment of a seat or the loop function, have been under refinement after understanding the possibilities and limitations of Isabelle/HOL.

Moreover, while writing the divisor method and the proof, these also affected the way the divisor method was built, since Isabelle prefers some structures over others.

As an example, the modular structure of the method had to be refined to be able to write the induction proofs with a correct technique.

In summary, the comprehensive structure of this work, consisting of various interconnected modules and submodules, provided significant advantages in terms of flexibility and efficiency.

This modular approach allowed for individual parts of the project to be updated or modified without necessitating a complete rewrite of the entire system. Each module and sub-module could be developed, tested, and refined independently, ensuring that changes in one area did not adversely impact the others.

This significantly streamlined the development process and facilitated a more manageable and scalable framework.

Moreover, the lemmas and proofs written for this project closely follow the structure of the implementation of the functions that constitute the divisor method.

This alignment between the theoretical and practical aspects of the project ensured that the proofs were not only rigorous but also directly applicable to the implementation.

By mirroring the functional architecture in the proof structure, the project achieved a high degree of coherence and clarity.

Each lemma served as a building block, directly corresponding to specific functions within the divisor method, thus simplifying the verification process and enhancing the overall comprehensibility of the work.

The success of this modular structure is evident in several key areas. Firstly, it facilitated a clear understanding of each module's role and function within the larger system.

This clarity is crucial for both the initial development and for future maintenance and extension of the project. The ability to isolate and address specific components independently means that new variants of the methods can be incorporated with minimal disruption.

This modularity also supports scalability, as additional modules can be integrated into the existing framework with relative ease.

Furthermore, this approach has significant implications for the extension and adaptation of the framework. By maintaining a modular architecture, the system is inherently flexible and can be adapted to accommodate various extensions or modifications.

This adaptability ensures that the framework remains relevant and useful even as new requirements or methods emerge. It also makes the system more robust, as individual modules can be improved or replaced without compromising the integrity of the entire project.

In conclusion, the modular and structured approach employed in this work provided a robust and flexible framework for both the development and verification of the divisor method.

The alignment of lemmas with the functional implementation not only ensured rigorous proof standards but also enhanced the clarity and manageability of the project. This structure supports ongoing adaptability and scalability, ensuring that the framework can evolve and remain effective in the face of new challenges and requirements.

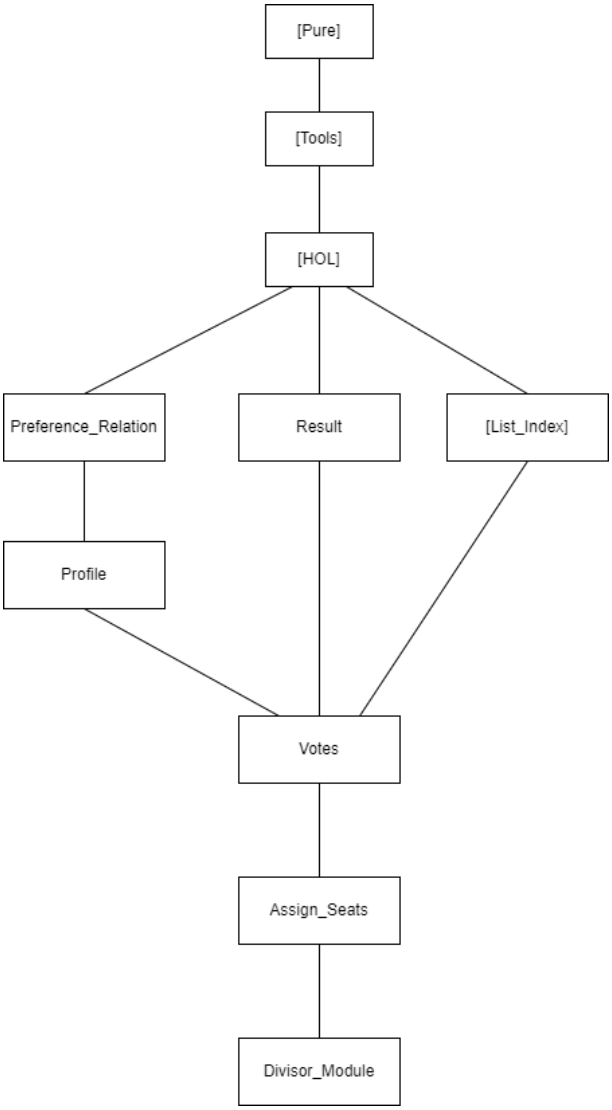

Figure 6.1: session graph

7 Conclusion

This final chapter wraps up the work developed in the thesis, trying to deliver an idea of which core ideas were achieved. Furthermore, it will explain which theoretical contributions and practical additions to the original framework were made.

7.1 Summary

This thesis embarked on the formalization of the Divisor Method, a core proportional representation method in Social Choice Theory.

As starting point of this thesis, the first step was to give an idea of the foundational groundwork the work is built on, therefore resuming some definitions and conceptualizations that were useful to develop new theory. Most of this foundations are taken by Pukelsheim, 2017b and Diekhoff, Kirsten, and Krämer, 2020, with some changes made in order to better handle the case approached in this thesis.

After retrieving the foundations, in Chapter 3 a theoretical formalization of the topic has been introduced, starting from the most basic definitions up to the whole Divisor Method, while defining some intermediate level functions that were of primary importance for the following building of the code.

During the first process of theoretical writing of the method, obstacles regarding the formalization were successfully overcame and some of the insights gained by this formalization have been exploited later in the implementation of the code, leading the research question Q1 in the introduction to be answered.

Moreover, after the method, the properties taken in consideration are formally introduced and formalized. These properties, described through theorems, have been correctly proved and have been used as a reference and starting point for building the proof in Isabelle/HOL.

Chapter 4 establishes the practical side of the work of this thesis, bringing an implementation into Isabelle/HOL with the aim to present a finished set of functions on which theorems can be proved.

In this section all functions have been written with the goal to stay as generic as possible and provide a modular structure, so that in a potential future work they can be used individually.

Throughout Chapter 5 the proofs developed earlier only on a theoretical plane are concretely implemented in Isabelle/HOL, acknowledging that some adaptations from the formalized proofs had to be done, in order to accurately translate the properties into the realm of Isabelle/HOL.

During this phase, two of the three properties, Monotonicity and Concordance, have been developed almost completely. In fact, they have been proved to hold for the core

function of the Divisor Method, which represents the assignment of one single seat. This is considered to be the most important component of the algorithm, since the decision rule defined in the method is applied in this phase.

Moreover, this function also updates the factors according to the divisors, which appear to be pivotal in the method, since variants like D'Hondt and Sainte-Laguë are built bringing changes on these parameter.

This means that this modular component identifies the pivotal steps of the method and on this function, the two properties have been correctly written. What is currently missing is a proof to step from the assignment of one seat to the assignment of all the available seats.

Nevertheless, as already described earlier in this work, the method is simply an iterative application of the function described above.

Therefore, it is straightforward to say that if a property holds for a given function, then it will also hold for an iteration of the same function.

On the contrary, Anonymity has not been implemented due to shortcoming of time. In fact, the proof of the above properties required more time and effort than estimated, both for implementing them and for the obstacles encountered in approaching the new tool, Isabelle/HOL, resulting in only an unfruitful attempt to implement Anonymity.

What is present in Chapter 5 about this property, however, is a simple explanation that, even with a missing concrete proof, it would not be hard to imagine the Divisor Method to be anonymous and a proposed solution is introduced.

Chapter 4 and Chapter 5 are without any doubt the most important part of the thesis. The implementation in Isabelle is the part the took most of the time and, through the implementation given, it has been possible to prove the theorems by proving other properties or lemmas regarding smaller modules.

This modular approach is pivotal for the work of the thesis, since it influences more than one research questions brought in the introduction (Q3, Q4 and Q5).

In fact, this thesis analyzed how Isabelle can be used to prove properties related to the divisor method (Q3). Moreover, these proofs have been executed also through proofs related to the components (Q4), an approach that helps also for future extensions and variations of the divisor method (Q5).

Finally, in Chapter 6 a comparison between the two methods is presented, underlining similarities and differences of the two variants and the possible different outcomes they can give with respect to the same starting election material.

This final chapter briefly introduces and addresses the last two research questions.

As a matter of fact, in the aforementioned Chapter an analysis of how the framework was lacking multiple winners methods was developed, therefore pointing out that existing framework had to be extended in a radical way with new modules (Q6).

Moreover, in the brief comparison section of Chapter 6, it has been shown how the two methods both respect the same properties but, on another perspective, hold some differences due to the dissimilarity of the choice of divisors.

Therefore, to answer Q7, if the *fairness* is only described as the holding of some properties, it can be said that the two methods are indeed *fair* on the same level.

On the other hand, it was shown that D'Hondt method tends to be in favor of larger parties, while Sainte-Laguë gives more possibilities to smaller parties.

At this point, if the definition of *fairness* takes into consideration also this aspect, it could be inherited that indeed Sainte-Laguë method is more *fair* than D'Hondt.

In summary, the research questions proposed in the introduction have been answered almost fully, giving an important contribution to the framework and respecting the goals planned during the first preliminary study of this thesis.

7.2 Comparison

This section aims to elaborate on the results developed by this work, comparing the two methods in the similarities and differences and bringing up examples of how the properties on which this work is focused on are not always respected in some methods and can bring to unfairness in the outcome of the election.

It is quite trivial that the two methods are almost identical, with the only difference, as said previously, is the choice of divisors. Despite this, it is enough for the methods to be considered different and to have different outcomes.

The two methods have been developed in the framework and, as it can be seen in the code provided in the appendix, they share almost the same code. This means that the two methods also share Concordance and Monotonicity properties that have been proved for a function common to both variants.

Besides these properties, it is possible to inherit that every property that can be proved to the common functions of the two methods that it holds for both variants.

More in general, on a theoretical plane, whatever property does not depend on some condition of the divisors, than it is shared between all variants of the method provided.

Beyond the amount or similarities, it is quite easy to give an idea of how the methods are different with a simple example.

Suppose there are two political parties (Party A and Party B) and a total of five seats to be allocated based on their share of the vote.

Party	Votes	Seats
Party A	4500	-
Party B	2000	-

We will use the D'Hondt method to allocate the five seats.

1. For every seat to be assigned, the allocation ratio for each party is calculated:

Party	Allocation Ratio
Party A	$\frac{4500}{0+1} = 4500$
Party B	$\frac{2000}{0+1} = 2000$

2. We allocate the first seat to Party A since it has the highest allocation ratio.

Party	Seats
Party A	1
Party B	0

3. For the second seat, we recalculate the allocation ratios:

Party	Allocation Ratio
Party A	$\frac{4500}{1+1} = 2250$
Party B	$\frac{2000}{0+1} = 2000$

4. The second seat goes again to Party A.

Party	Seats
Party A	2
Party B	0

   The third seat goes now to Party B, since it has the biggest allocation ratio.

5. We repeat the process to allocate the remaining seats. After allocating all five seats, the distribution is as follows:

Party	Votes	Seats
Party A	4500	4
Party B	2000	1

So in this case, Party A has the 80% of the representation in the seats, while Party B has only the 20%.

Instead, in the following procedure the same seats will be assigned according to the Sainte-Laguë variant.

1. For every seat to be assigned, the allocation ratio for each party is calculated:

Party	Allocation Ratio
Party A	$\frac{4500}{0+1} = 4500$
Party B	$\frac{2000}{0+1} = 2000$

2. The first seat is assigned to Party A due to the highest allocation ratio.

Party	Seats
Party A	1
Party B	0

3. For the second seat, we recalculate the allocation ratios:

Party	Allocation Ratio
Party A	$\frac{4500}{3} = 1500$
Party B	$\frac{2000}{1} = 2000$

4. The second seat goes to Party B.

Party	Seats
Party A	1
Party B	1

   The third seat goes now to Party A, since it has the biggest allocation ratio.

5. The process is repeated to allocate the remaining seats. After allocating all five seats, the distribution is as follows:

Party	Votes	Seats
Party A	4500	3
Party B	2000	2

In this case, Party A has a representation of 60% and Party B has 40% of the total seats.

As it is possible to retrieve from this simple example, the two methods despite the only change produce very different output.

In fact, in the second case Party B has gained 40% of the seats, while in the first one it had only 20%, meaning that it doubled its representation.

7.3 Future work

While most of the work that had been initially planned for the thesis has been fully developed, there are still missing components that have not been added, mostly due to shortcoming of time.

7.3.1 Theoretical work

The theoretical part of this thesis is giving a full and comprehensive explanation of the topics needed to implement the code.

Nevertheless, a possible theoretical future work would be possible, adding new theorems and properties of the method developed in this work. As a result of the outcome of the practical part, future additions to the framework would not have to formalize again the method and it would be possible to completely focus only on the theorems, using the implementation of the Divisor Method that has been built in this thesis.

Furthermore, every variant of the Divisor Method can be easily added to the framework by simply following the pattern that led to the actual implementation of the D'Hondt and Sainte-Laguë methods.

Also, it is known that the current voting systems rather than a "pure" method use an ensemble of different methods that origin from the basic ones, as the case of Germany. Therefore, it would be possible to formalize a variation of this method which could resemble the ones used in reality more accurately.

7.3.2 Extending the framework

As already described earlier, the practical work of the thesis covered the development of the Divisor Method and some proofs related to some properties it holds.

While most of the proofs have been written and correctly work, it could be possible to give a proper proof for the Anonymity property, which has not been developed.

Moreover, it could be possible to finally wrap Monotonicity and Concordance property in the full method function, since at the moment it is missing in the framework.

The method itself has been fully developed but some additions can be made. In fact, it would be possible to implement variants of the Divisor Method by simply changing the divisors, following the same structure that has been adopted to develop D'Hondt and Sainte-Laguë rules.

Also, as suggested in the previous subsection, a future additional could be to build more complex methods that better reflect reality, starting from the Divisor Method developed in this work or using only some parts of this method, since it was developed in a modular way.

In conclusion, this new branch of the framework enables to analyze a whole new branch of properties that were never considered until this moment in the existing work.

With this implementation, it would be also possible to bring new multiple winners methods and draw some comparisons with the one developed in this thesis, to search for similarities and differences between the various methods of Social Choice Theory.

Bibliography

Diekhoff, Karsten, Michael Kirsten, and Jonas Krämer (2020). "Verified Construction of Fair Voting Rules". In: *Logic-Based Program Synthesis and Transformation*. Ed. by Maurizio Gabbrielli. Cham: Springer International Publishing, pp. 90–104. ISBN: 978-3-030-45260-5.

Gallagher, Michael (1991). "Proprtionality, Disproportionality and Electoral Systems". In: *Electoral Studies*, pp. 33–51.

Lijphart, Arend (1990). "The Political Consequences of Electoral Laws, 1945–85". In: *American Political Science Review* 84.2, pp. 481–496. DOI: 10.2307/1963530.

Makarius, Wenzel, Lawrence C. Paulson, and Tobias Nipkow (2020). *Isabelle/HOL - A Proof Assistant for Higher Order Logic*. Springer Berlin, Heidelberg.

Pukelsheim, Friedrich (2017a). "Divisor Methods of Apportionment: Divide and Round". In: *Proportional Representation: Apportionment Methods and Their Applications*. Cham: Springer International Publishing, pp. 71–93. ISBN: 978-3-319-64707-4. DOI: 10.1007/978-3-319-64707-4_4. URL: https://doi.org/10.1007/978-3-319-64707-4_4.

– (2017b). *Proportional Representation*. Springer Cham. ISBN: 978-3-319-64706-7.

– (2017c). "Securing System Consistency: Coherence and Paradoxes". In: *Proportional Representation: Apportionment Methods and Their Applications*. Cham: Springer International Publishing, pp. 159–183. ISBN: 978-3-319-64707-4. DOI: 10.1007/978-3-319-64707-4_9. URL: https://doi.org/10.1007/978-3-319-64707-4_9.

Wenzel et al., Makarius (2023). *The Isabelle/Isar reference manual*.

A Appendix

A.1 Votes

theory *Votes*
  imports *Complex_Main*
HOL.List
"HOL-Combinatorics.Multiset_Permutations"
"HOL-Combinatorics.List_Permutation"
Preference_Relation
Profile
Result
"List-Index.List_Index"

type_synonym *'b Parties = "'b list"*

type_synonym *'b Votes = "'b list"*

type_synonym *('a, 'b) Seats = "'a ⇒ 'b list"*

type_synonym *Params = "nat list"*

Auxiliary Code

definition *above_set :: "_ ⇒ 'a ⇒ 'a set"*
  where *"above_set r a ≡ above (set r) a"*

lemmas *[code] = above_set_def[symmetric]*

lemma *[code]:*
  ‹above_set [] a = {}›
  ‹above_set ((x,y)#xs) a = (if x=a then {y} else {}) ∪ above_set xs a›
  by *(auto simp: above_set_def above_def)*

definition *remove_some :: "'a multiset ⇒ 'a"* **where**
"remove_some M = (SOME x. x ∈ set_mset M)"

fun *empty_votes :: "char list ⇒ rat"* **where**
  "empty_votes p = 0"

fun *start_votes :: "'a list ⇒ nat Votes"* **where**
  "start_votes [] = []" |
  "start_votes (x # xs) = 0 # start_votes xs"

```
fun start_seats :: "'a list ⇒ nat list" where
  "start_seats [] = []" |
  "start_seats (x # xs) = 0 # start_seats xs"
```

This function returns the rank of one candidate over a ballot.

```
fun count_above :: "('a rel) ⇒ 'a ⇒ nat" where
  "count_above r a = card (above r a)"

fun count_above_mset :: "('a multiset rel) ⇒'a multiset ⇒ nat" where
  "count_above_mset r a = card (above r a)"

fun create_empty_seats :: "'a::linorder set ⇒ 'b Parties ⇒ ('a::linorder, 'b) Seats"

  where
"create_empty_seats indexes parties =
    (λi. if i ∈ indexes then parties else [])"

fun start_fract_votes :: "nat Votes ⇒ rat Votes" where
  "start_fract_votes [] = []" |
  "start_fract_votes (nn # nns) = (of_nat nn) # start_fract_votes nns"
```

This function retrieves the votes for the specified party.

```
fun get_votes :: "'b ⇒ 'b Parties ⇒ nat Votes ⇒ nat" where
"get_votes px ps v = v ! index ps px"
```

This function counts votes for one party and add returns the number of votes.

```
fun cnt_votes :: "'a ⇒ 'a Profile ⇒ nat ⇒ nat" where
  "cnt_votes p [] n = n" |
  "cnt_votes p (px # pl) n =
     (case (count_above px p) of
        0 ⇒ cnt_votes p pl (n + 1)
      | _ ⇒ cnt_votes p pl n)"

fun cnt_votes_mset :: "'a multiset ⇒ (('a multiset × 'a multiset) set) list ⇒ nat
⇒ nat"
  where
  "cnt_votes_mset p [] n = n" |
  "cnt_votes_mset p (px # profil) n =
     (case (count_above_mset px p) of
        0 ⇒ cnt_votes_mset p profil (n + 1)
      | _ ⇒ cnt_votes_mset p profil n)"
```

This function receives in input the list of parties and the list of preferation relations. The output is a list in which every party has the correspondent number of votes.

```
fun calc_votes :: "'a Parties ⇒ 'a Parties ⇒ 'a Profile ⇒ nat Votes ⇒ nat Votes"
where
  "calc_votes [] fp pl votes = votes" |
  "calc_votes (px # ps) fp pl votes =
      (let n = cnt_votes px pl 0;
       i = index fp px in
      calc_votes ps fp pl (list_update votes i n))"

fun max_v:: "'a::linorder Votes ⇒ 'a::linorder" where
"max_v v = Max (set v)"

fun max_p:: "'a::linorder ⇒'a::linorder Votes ⇒ 'b Parties ⇒ 'b Parties"
  where
"max_p m v ps = filter (λx. v ! (index ps x) = m) ps"

fun empty_v :: "'b ⇒ rat" where
  "empty_v p = 0"
```

This function calculates the winners between candidates according to the votes.

```
fun get_winners :: "'a::linorder Votes ⇒ 'b Parties ⇒ 'b Parties" where
  "get_winners v p = (let m = max_v v in max_p m v p)"

fun update_seat :: "'a::linorder ⇒ 'b Parties ⇒ ('a::linorder, 'b) Seats
                    ⇒ ('a::linorder, 'b) Seats" where
  "update_seat seat w seats = seats(seat := w)"
```

This function counts seats of a given party.

```
fun cnt_seats :: "'b Parties ⇒ ('a::linorder, 'b) Seats ⇒
                    'a::linorder set => nat" where
  "cnt_seats p s i = card {ix. ix ∈ i ∧ s ix = p}"
```

Auxiliary Lemmas

A.1.1 Proofs

```
lemma perm_induct:
  assumes "P [] []"
  assumes "⋀x xs ys. P (x # xs) (x # ys)"
  assumes "⋀x y xs ys. P (x # y # xs) (y # x # ys)"
  assumes "⋀xs ys zs. P xs ys ⟹ P ys zs ⟹ P xs zs"
  shows "mset xs = mset ys ⟹ P xs ys"
proof (induction xs arbitrary: ys)
  case Nil
  then show ?case
    using assms by auto
next
```

```
  case (Cons x xs)
  then show ?case
    using assms
    by (metis neq_Nil_conv perm_empty2)
qed

lemma index_correct:
  fixes
  p::"'a list"
assumes "index p px < size p"
  shows "p ! (index p px) = px"
  by (meson assms index_eq_iff)

lemma index_diff_elements:
  assumes
    "p1 ∈ set p" and
    "p1 ≠ p2"
  shows "index p p1 ≠ index p p2"
proof (rule ccontr)
  assume "¬ (index p p1 ≠ index p p2)"
  then have "index p p1 = index p p2"
    by simp
  then obtain n1 n2 where "index p p1 = n1" and "index p p2 = n2"
    by blast
  hence "n1 = n2"
    by (meson ‹¬ index p p1 ≠ index p p2›)
  hence "p ! n1 = p1"
    using assms index_correct ‹index p p1 = n1› by fastforce
  hence "p ! n2 = p2"
    using assms index_correct ‹index p p2 = n2›
          ‹index p p1 = index p p2› by force
  hence "p1 = p2"
    using assms ‹¬ index p p1 ≠ index p p2› ‹n1 = n2› ‹p ! n1 = p1› by blast
  with assms show False by simp
qed

lemma minus_max:
 assumes "ip = index parties party"
  "size v = size parties" and
  "max_v v ≠ v ! ip" and
  "finite (set v)"
shows "Max(set v) = Max ((set v) - {v ! ip})"
  by (metis Diff_empty Diff_insert0 Max.remove assms(3) assms(4) max_def max_v.simps)

lemma max_val_wrap:
  fixes
  v::"rat list" and
  v'::"rat list"
```

```
assumes "set v = set v'"
shows "max_v v = max_v v'"
  using assms by simp

lemma max_val_wrap_lemma:
  fixes fv::"rat list"
  assumes "fvx = fv ! i" and "i < length fv"
  shows "max_v fv ≥ fvx"
  by (simp add: assms)

lemma max_p_ne:
  assumes "p ∈ set ps" and
    "p ∉ set (max_p m v ps)" and
    "m>0" and
    "ps ≠ []"
  shows "v ! index ps p ≠ m"
  using assms by simp
```

This lemma shows that max p function cannot be empty.

```
lemma max_p_no_empty:
  assumes "m = Max (set v)" and
  "(set v) ≠ {}" and
  "length v = length ps" and
  "⋀i j. i ≠ j ⟹ ps ! i ≠ ps ! j"
  shows "max_p m v ps ≠ []"
proof -
  have "Max(set v) ∈ (set v)"
    using assms(2) by auto
  then show ?thesis
    using ‹Max (set v) ∈ set v› assms empty_filter_conv in_set_conv_nth max_p.simps
          nth_index nth_mem by (smt (verit))
qed

lemma P_no:
  assumes
    "x ∈ set xs" and
    "¬(P x)"
  shows "x ∉ set (filter(λx. P x) xs)"
  using assms by simp

lemma filter_loss:
  assumes
    "p ∈ set ps" and
    "m = Max(set v)" and
    "¬ (v ! (index ps p) = m)"
  shows "p ∉ set (filter (λx. v ! (index ps x) = m) ps)"
  using assms by simp
```

```
lemma max_p_loss:
  assumes
    "p ∈ set ps" and
    "m = Max(set v)" and
    "¬ (v ! (index ps p) = m)"
  shows "p ∉ set (max_p m v ps)"
  using assms by simp

lemma max_p_in_win:
  assumes
"v ! (index ps px) = m" and
"px ∈ set ps"
  shows "px ∈ set (max_p m v ps)"
proof -
  have "max_p m v ps = filter (λx. v ! (index ps x) = m) ps"
    using assms by simp
  then have "px ∈ set (filter (λx. v ! (index ps x) = m) ps)"
    using assms by auto
  then show ?thesis
    by simp
qed

lemma get_winners_no_empty:
  assumes "∀i j. p ! i ≠ p ! j"
  shows "get_winners v ps ≠ []"
  using assms by blast

lemma get_winners_loss:
  assumes
    "p ∈ set ps" and
    "(v ! (index ps p) ≠ Max(set v))"
  shows "p ∉ set (get_winners v ps)"
  using assms by simp
```

This theorem proves that if a party has the max votes, then it is in the list of winners.

```
theorem get_winners_in_win:
  assumes
    "fv ! index ps px = max_v fv" and
    "px ∈ set ps"
  shows "px ∈ set (get_winners fv ps)"
proof -
  have "get_winners fv ps = (let m = max_v fv in max_p m fv ps)"
    using get_winners.simps by blast
  then have "... = max_p (max_v fv) fv ps"
    by simp
  then show ?thesis
    using assms max_p_in_win ‹get_winners fv ps = (let m = max_v fv in max_p m fv ps)›
```

```
    by metis
qed

lemma get_winners_not_in_win:
  assumes
    "fv ! (index ps px) ≠ max_v fv" and
    "get_winners fv ps ≠ []"
  shows "px ≠ hd (get_winners fv ps)"
proof -
  have "get_winners fv ps = (let m = max_v fv in max_p m fv ps)"
    using get_winners.simps by blast
  then have "px ∉ set (max_p (max_v fv) fv ps)"
    using assms by simp
  then show ?thesis
    using ‹get_winners fv ps = (let m = max_v fv in max_p m fv ps)›
          hd_in_set assms by metis
qed

lemma get_winners_only_winner:
  assumes "fv ! (index ps px) = max_v fv" and
    "∀x ≠ index ps px. fv ! (index ps x) < max_v fv" and
    "get_winners fv ps ≠ []"
shows "px = hd (get_winners fv ps)"
  using get_winners_not_in_win verit_comp_simplify1(1) assms
  by metis

lemma get_winners_weak_winner_implies_helper:
  assumes "x < size fv"
  shows "Max(set fv) ≥ fv ! x"
  using assms by simp

lemma max_val_wrap_eqI_helper:
  assumes
"⋀y. y ∈ (set fv) ⟹ y ≤ fv ! x" and
"fv ! x ∈ (set fv)"
  shows "Max(set fv) = fv ! x"
  using Max_eqI assms by blast

lemma max_eqI_helper:
  assumes
"x < length l" and
"length l = length l'" and
"l ! x = Max(set l)" and
"l' ! x > l ! x" and
"⋀y. y ≠ x ⟹ y < length l ⟹ l' ! y ≤ l ! y"
shows "l' ! x = Max (set l')"
proof(rule ccontr)
  assume "l' ! x ≠ Max (set l')"
```

```
  then have "∃y. y ≠ x ⟶ y < length l ⟶ Max(set l') = l' ! y"
    by auto
  then have "⋀y. y < length l ⟹ l ! y ≤ Max(set l)"
    by simp
  then have "⋀y. y ≠ x ⟶ y < length l ⟶ l' ! y ≤ Max(set l)"
    by (meson assms(5) dual_order.trans)
  then have "Max(set l') > Max(set l)"
    using assms dual_order.strict_trans1
          get_winners_weak_winner_implies_helper by metis
  then have "⋀y. y ≠ x ⟶ y < length l ⟶ l' ! y < Max(set l')"
    using ‹⋀y. y ≠ x ⟶ y < length l ⟶ l' ! y ≤ Max (set l)› by fastforce
  then show False
    using ‹∃y. y ≠ x ⟶ y < length l ⟶ Max(set l') = l' ! y›
          ‹⋀y. y ≠ x ⟶ y < length l ⟶ l' ! y < Max(set l')› assms
          ‹⋀y. y ≠ x ⟶ y < length l ⟶ l' ! y ≤ Max (set l)›
          ‹l' ! x ≠ Max (set l')› in_set_conv_nth leD linorder_le_cases
          max_val_wrap_eqI_helper order.strict_trans2
  by (metis (no_types, opaque_lifting))
qed

lemma max_eqI:
assumes
"index ps px < length fv" and
"length fv = length fv'" and
"fv ! index ps px = Max(set fv)" and
"fv' ! index ps px > fv ! index ps px" and
"⋀y. y ≠ index ps px ⟹ y < length fv ⟹ fv' ! y ≤ fv ! y"
shows "Max(set fv') = fv' ! index ps px"
  using assms max_eqI_helper by metis
```

This lemma proves that if a party wins, then if its votes increases, it wins again.

```
lemma get_winners_weak_winner_implies:
assumes
"index ps px < length fv" and
"length fv = length fv'" and
"px ∈ set ps" and
"fv ! index ps px = Max(set fv)" and
"fv' ! index ps px > fv ! index ps px" and
"⋀y. y ≠ index ps px ⟹ y < length fv ⟹ fv' ! y ≤ fv ! y"
shows "px ∈ set (get_winners fv' ps)"
  using assms get_winners_in_win max_eqI_helper max_v.simps by metis

lemma filter_size_is_one_helper:
  fixes
fv::"'a :: linorder list"
assumes
"x < length fv" and
```

```
"fv ! x = m" and
strict_le: "⋀y. y ≠ x ⟹ y < length fv ⟹ fv ! y < m"
shows "length (filter (λx. fv ! x =  m) [0..<length fv]) = 1"
proof -
  have le: ‹[0..<length fv] = ([0..<x] @ [x] @ [Suc x ..< length fv])›
    using ‹fv ! x = m› append_Cons assms(1) leI le_add_diff_inverse less_imp_le_nat
        not_less_zero self_append_conv2 upt_add_eq_append upt_rec
    by metis
  show ‹length (filter (λx. fv ! x =  m) [0..<length fv]) = 1›
    using strict_le assms unfolding le by (force simp: filter_empty_conv)
qed

lemma filter_size_is_one_helper_my_case:
  fixes
fv::"rat list"
assumes
"index ps x < length fv" and
"fv ! index ps x = m" and
strict_le: "⋀y. y ≠ index ps x ⟹ y < length fv ⟹ fv ! y < m" and
"length (filter (λx. fv ! (index ps x) = m) ps)
  = length (filter (λx. fv ! x =  m) [0..<length fv])"
shows "length (filter (λx. fv ! (index ps x) = m) ps) = 1"
  by (metis assms filter_size_is_one_helper)

lemma filter_size_is_one_helper_my_case_3:
  fixes
fv::"'a::linorder list"
assumes
"index ps x < length fv" and
"fv ! index ps x = m" and
strict_le: "⋀y. y ≠ index ps x ⟹ y < length fv ⟹ fv ! y < m" and
"length (get_winners fv ps) = length (filter (λx. fv ! x =  m) [0..<length fv])"
shows "length (get_winners fv ps) = 1"
  using assms(1) assms(2) assms(4) filter_size_is_one_helper strict_le by auto
```

This theorem states that if one the parties in the winners list gets its votes increased, then it will become the only winner.

```
theorem get_winners_size_one:
  fixes
fv::"'a::linorder list"
assumes
"party ∈ set parties" and
"index parties party < length fv" and
"m = Max(set fv)" and
"fv ! index parties party = m" and
strict_le: "⋀y. y ≠ index parties party ⟹ y < length fv ⟹ fv ! y < m" and
"length (get_winners fv parties) = length (filter (λx. fv ! x =  m) [0..<length fv])"
```

```
shows "party = hd (get_winners fv parties)"
proof -
  have "length (get_winners fv parties) = 1"
    using filter_size_is_one_helper_my_case_3
  by (metis assms(2) assms(4) assms(6) strict_le)
  then have "party ∈ set (get_winners fv parties)"
    using assms(1) assms(3) assms(4) get_winners_in_win max_v.elims by metis
  then show ?thesis
    using ‹length (get_winners fv parties) = 1› ‹party ∈ set (get_winners fv parties)›

        hd_conv_nth in_set_conv_nth length_greater_0_conv less_one by metis
qed
```

This lemma states that if a party is in the head of winners, then it has the maximum votes.

```
lemma get_winners_rev:
  assumes
    "party = hd (get_winners v ps)" and
    "get_winners v ps ≠ []"
  shows "v ! index ps party = max_v v"
proof(rule ccontr)
  assume "v ! index ps party ≠ max_v v"
  then have "party ∉ set (get_winners v ps)"
    by simp
  then show False
    using assms list.set_sel(1) by blast
qed
```

A.2 Assign Seats

```
theory Assign_Seats
  imports Complex_Main
Main
"Social_Choice_Types/Preference_Relation"
"Social_Choice_Types/Profile"
"Social_Choice_Types/Result"
"Electoral_Module"
"Social_Choice_Types/Votes"
"Termination_Condition"
"HOL-Combinatorics.Multiset_Permutations"
"Consensus_Class"
"Distance"
```

begin

This record contains the list of parameters used in the whole divisor module.

```
record ('a::linorder, 'b) Divisor_Module =
  res :: "'a::linorder Result"
  p :: "'b Parties"
  i :: "'a::linorder set"
  s :: "('a::linorder, 'b) Seats"
  ns :: nat
  v :: "nat Votes"
  fv :: "rat Votes"
  sl :: "nat list"
  d :: "nat list"
```

Abbreviation to retrieve the assigned seats from the Result type.

```
abbreviation ass_r :: "'a Result ⇒ 'a set" where
  "ass_r r ≡ fst r"
```

Abbreviation to retrieve the disputed seats from the Result type.

```
abbreviation disp_r :: "'a Result ⇒ 'a set" where
  "disp_r r ≡ snd (snd r)" fun upd_votes ::  "'b Parties ⇒ ('a::linorder, 'b) Divisor_Module
⇒
                      rat Votes"
  where
"upd_votes w r =
    (let ix = index (p r) (hd w);
     n_seats = cnt_seats w (s r) (i r);
     n_v = of_nat(get_votes (hd w) (p r) (v r)) / of_int(d r ! n_seats) in
     list_update (fv r) ix n_v)"
```

This function moves one seat from the disputed set to the assigned set. Moreover, returns the record with updated Seats function and "fractional" Votes entry for the winning party.

```
fun upd_round :: "'b Parties ⇒ ('a::linorder, 'b) Divisor_Module ⇒
                      ('a::linorder, 'b) Divisor_Module"
  where
"upd_round w rec =
  (let
    seat = Min (disp_r (res rec));
    new_s = update_seat seat w (s rec);
    new_as = ass_r (res rec) ∪ {seat};
    new_fv = upd_votes w (rec(|s:= new_s|));
    ix = index (p rec) (hd w);
    curr_ns = (sl rec) ! ix;
    new_sl = list_update (sl rec) ix ( (sl rec) ! ix + 1);
    new_di =  disp_r (res rec) - {seat}
     in (|res = (new_as, {}, new_di),
            p = (p rec),
            i = (i rec),
            s = new_s,
            ns = (ns rec),
            v = (v rec),
```

```
            fv = new_fv,
            sl = new_sl,
            d = (d rec)
           ⦈)"
```

This function is handling the case of a tie. It will just defer the record assigning the disputed seats to ALL winners.

```
fun break_tie :: "'b Parties ⇒ ('a::linorder, 'b) Divisor_Module ⇒
                     ('a::linorder, 'b) Divisor_Module"
  where
"break_tie ws rec =
         (⦇res = (res rec),
            p = (p rec),
            i = (i rec),
            s = update_seat (Min (disp_r (res rec))) ws (s rec),
            ns = (ns rec),
            v = (v rec),
            fv = (fv rec),
            sl = (sl rec),
            d = (d rec)
           ⦈)"
```

This function checks whether there are enough seats for all the winning parties. - If yes, assign the seat to the first party in the list. - If not, assign the seat to the winning parties, making these seats "disputed".

```
fun assign_seat :: "('a::linorder, 'b) Divisor_Module
                       ⇒ ('a::linorder, 'b) Divisor_Module"
  where
"assign_seat rec = (
      let ws = get_winners (fv rec) (p rec) in
      if length ws ≤ ns rec then
        let rec' =  (upd_round [hd ws] rec) in
                    (⦇res = (res rec'),
                         p = (p rec'),
                         i = (i rec'),
                         s = (s rec'),
                         ns = ((ns rec') - 1),
                         v = (v rec'),
                         fv = (fv rec'),
                         sl = (sl rec'),
                         d = (d rec')
                        ⦈)
      else
         let rec'' = (break_tie ws rec) in
                      (⦇res = (res rec''),
                        p = (p rec''),
                        i = (i rec''),
                        s = (s rec''),
```

```
                    ns = 0,
                    v = (v rec''),
                    fv = (fv rec''),
                    sl = (sl rec''),
                    d = (d rec'')
```

A.2.1 Proofs

```
lemma assign_seat_mantain_seats_lemma:
  assumes
"ix ≠ index (p rec) (hd w)"
shows "sl (upd_round w rec) ! ix = sl rec ! ix"
proof -
   define seat new_s new_as new_fv ind curr_ns new_sl new_di
    where "seat = Min (disp_r (res rec))" and
          "new_s = update_seat seat w (s rec)" and
          "new_as = ass_r (res rec) ∪ {seat}" and
          "new_fv = upd_votes w (rec(|s:= new_s|))" and
          "ind =  index (p rec) (hd w)" and
          "curr_ns = (sl rec) ! ind" and
          "new_sl = list_update (sl rec) ind (curr_ns + 1)" and
          "new_di =  disp_r (res rec) - {seat}"
  have "(upd_round w rec) =  (|res = (new_as, {}, new_di),
             p = (p rec),
             i = (i rec),
             s = new_s,
             ns = (ns rec),
             v = (v rec),
             fv = new_fv,
             sl = new_sl,
             d = (d rec)
            |)"
    unfolding upd_round.simps new_sl_def Let_def
    using nth_list_update_eq curr_ns_def ind_def new_as_def new_di_def new_fv_def
          new_s_def seat_def by fastforce
  then have "sl (upd_round w rec) = new_sl"
    by simp
  then have "... =  list_update (sl rec) ind (curr_ns + 1)"
    using new_sl_def by simp
  then have "sl (upd_round w rec) ! ix = list_update (sl rec) ind (curr_ns + 1) ! ix"
    using ‹sl (upd_round w rec) = new_sl› by presburger
  then have "... = (sl rec) ! ix"
    using nth_list_update_neq ind_def assms(1) by metis
  then show ?thesis
  using ‹sl (upd_round w rec) ! ix = (sl rec)[ind := curr_ns + 1] ! ix› by presburger
qed
```

```
lemma assign_seat_sl_update:
  fixes
winner :: "'b list" and
rec :: "('a::linorder, 'b) Divisor_Module"
defines i_def: "ind ≡ index (p rec) (hd winner)"
shows "sl (upd_round winner rec) = list_update (sl rec) ind (sl rec ! ind + 1)"
proof -
  define seat new_s new_as new_fv ind curr_ns new_sl new_di
    where "seat = Min (disp_r (res rec))" and
          "new_s = update_seat seat winner (s rec)" and
          "new_as = ass_r (res rec) ∪ {seat}" and
          "new_fv = upd_votes winner (rec⦇s:= new_s⦈)" and
          "ind =  index (p rec) (hd winner)" and
          "curr_ns = (sl rec) ! ind" and
          "new_sl = list_update (sl rec) ind (curr_ns + 1)" and
          "new_di =  disp_r (res rec) - {seat}"
  have "(upd_round winner rec) =  ⦇res = (new_as, {}, new_di),
             p = (p rec),
             i = (i rec),
             s = new_s,
             ns = (ns rec),
             v = (v rec),
             fv = new_fv,
             sl = new_sl,
             d = (d rec)
            ⦈"
    unfolding upd_round.simps new_sl_def Let_def
    using assms nth_list_update_eq curr_ns_def ind_def
          new_as_def new_di_def new_fv_def new_s_def seat_def by fastforce
  then have "sl(upd_round winner rec) = new_sl"
    by simp
  also have "... = list_update (sl rec) ind (curr_ns + 1)"
    using new_sl_def by simp
  also have "... =  list_update (sl rec) ind ((sl rec) ! ind + 1)"
    using new_sl_def curr_ns_def by simp
  finally show ?thesis
    using ind_def nth_list_update_eq i_def by blast
qed
```

This lemma states that the winner gets its seats increased.

```
lemma assign_seat_increase_seats:
  fixes
  rec::"('a::linorder, 'b) Divisor_Module" and
  winner::"'b list"
defines i_def: "inde ≡ index (p rec) (hd winner)"
assumes "inde < length (sl rec)"
shows "sl (upd_round winner rec) ! inde = sl rec ! inde + 1"
```

```
proof -
  have "sl (upd_round winner rec) =  list_update (sl rec) inde ((sl rec) ! inde + 1)"

    using assign_seat_sl_update local.i_def by blast
  then have "sl (upd_round winner rec) ! inde =
             list_update (sl rec) inde (sl rec ! inde + 1) ! inde"
    using local.i_def by simp
  then have "... = ((sl rec) ! inde + 1)"
    using nth_list_update_eq assms by simp
  then show ?thesis
    using ‹sl (upd_round winner rec) ! inde =
          list_update (sl rec) inde (sl rec ! inde + 1) ! inde› by auto
qed
```

This lemma states that with assignment of a seat the number of seats won't decrease.

```
lemma assign_one_seat_mon:
assumes "ind < length (sl rec)"
  shows "sl (upd_round winner rec) ! ind ≥ (sl rec) ! ind"
proof -
    define seat new_s new_as new_fv ind curr_ns new_sl new_di
    where "seat = Min (disp_r (res rec))" and
          "new_s = update_seat seat winner (s rec)" and
          "new_as = ass_r (res rec) ∪ {seat}" and
          "new_fv = upd_votes winner (rec(|s:= new_s|))" and
          "ind =  index (p rec) (hd winner)" and
          "curr_ns = (sl rec) ! ind" and
          "new_sl = list_update (sl rec) ind (curr_ns + 1)" and
          "new_di =  disp_r (res rec) - {seat}"
  have "(upd_round winner rec) =  (|res = (new_as, {}, new_di),
             p = (p rec),
             i = (i rec),
             s = new_s,
             ns = (ns rec),
             v = (v rec),
             fv = new_fv,
             sl = new_sl,
             d = (d rec)
            |)"
    unfolding upd_round.simps new_sl_def Let_def
    using nth_list_update_eq curr_ns_def ind_def new_as_def new_di_def
          new_fv_def new_s_def seat_def by fastforce
  then have "sl (upd_round winner rec) = new_sl"
    by simp
  then have "sl (upd_round winner rec) ! ind = new_sl ! ind"
    by simp
  then have "... = (list_update (sl rec) ind (curr_ns + 1)) ! ind"
    using new_sl_def nth_list_update_eq by blast
```

```
  then show ?thesis
    using ‹sl (upd_round winner rec) = new_sl› assms(1) curr_ns_def le_add1 new_sl_def

          nth_list_update_eq nth_list_update_neq order_refl by metis
qed

lemma break_tie_lemma:
  fixes
  rec::"('a::linorder, 'b) Divisor_Module" and
  winner::"'b list"
shows "sl (break_tie winner rec) = sl rec" by simp

lemma break_tie_lemma_party:
  fixes
  rec::"('a::linorder, 'b) Divisor_Module" and
  winner::"'b list"
shows "sl (break_tie winner rec) ! ind ≥ sl rec ! ind" by simp

lemma assign_seat_break_tie_case:
  assumes
    "winners = get_winners (fv rec) (p rec)" and
    "length winners > ns rec"
  shows "sl rec = sl (assign_seat rec)" using assms by simp
```

This lemma states that all the parties besides the winner keep their number of seats.

```
lemma assign_seat_not_winner_mantains_seats:
assumes "winners = get_winners (fv rec) (p rec)" and
  "i1 = index (p rec) (hd winners)" and
  "i1 ≠ i2" and
  "i2 < length (sl rec)"
  shows "sl rec ! i2 = sl (assign_seat rec) ! i2"
proof (cases "(length winners) ≤ ns rec")
  case True
  define rec'
   where
     "rec'= (upd_round [hd winners] rec)"
  have "assign_seat rec = (|res = (res rec'),
                           p = (p rec'),
                           i = (i rec'),
                           s = (s rec'),
                           ns = ((ns rec') - 1),
                           v = (v rec'),
                           fv = (fv rec'),
                           sl = (sl rec'),
                           d = (d rec')
                          |)"
    using rec'_def assms True assign_seat.simps by (smt (verit, best))
```

```
  then have "sl (assign_seat rec) = sl (rec')"
    by simp
  then have "sl (assign_seat rec) ! i2
             = sl (upd_round [hd winners] rec) ! i2"
    using ‹sl (assign_seat rec) = sl rec'› rec'_def by presburger
  then have "... = (sl rec) ! i2"
    using assign_seat_mantain_seats_lemma list.sel(1) assms(2) assms(3) by metis
  then show ?thesis
    using ‹sl (assign_seat rec) ! i2 = sl (upd_round [hd winners] rec) ! i2› by linarith
next
  case False
 define rec''
    where
     "rec''= break_tie winners rec"
   then have "assign_seat rec = (|res = (res rec''),
                          p = (p rec''),
                          i = (i rec''),
                          s = (s rec''),
                          ns = 0,
                          v = (v rec''),
                          fv = (fv rec''),
                          sl = (sl rec''),
                          d = (d rec'')
                         |)"
     using False rec''_def assms assign_seat.simps by auto
  then have "sl (assign_seat rec) ! i2 = sl (rec'') ! i2"
    by simp
  then have "... = sl (break_tie winners rec) ! i2"
    using break_tie_lemma rec''_def by simp
  then have "... = (sl rec) ! i2"
    by simp
  then show ?thesis
    using ‹sl (assign_seat rec) ! i2 = sl rec'' ! i2›
          ‹sl rec'' ! i2 = sl (break_tie winners rec) ! i2› by force
qed
```

This lemma states that if a party gets its votes increase (v' > v) and it is not winning a round with v' votes, then it is not winning also with the first number of votes v.

```
lemma get_winners_mon:
  fixes
    v::"rat"
  assumes
  "party ∈ set (p rec)" and
  "ip = index (p rec) party" and
  "v' > v" and
  "fv rec ! ip = v / of_nat dp" and
  "fv rec' ! ip = v' / of_nat dp" and
```

```
  "max_v (fv rec) = max_v (fv rec')" and
  "max_v (fv rec') > (fv rec') ! ip"
shows "party ∉ set (get_winners (fv rec) (p rec))"
proof -
  have "max_v (fv rec') > (fv rec') ! ip"
    using assms(7) by auto
  then have "... > (fv rec) ! ip"
    using  divide_le_cancel dual_order.strict_trans2 nless_le of_nat_less_0_iff
     assms(3) assms(4) assms(5) by metis
  then have "max_v (fv rec) = max_v (fv rec')"
    using assms(6) by blast
  then have "max_v (fv rec) > (fv rec) ! ip"
    using ‹fv rec ! ip < max_v (fv rec')› by linarith
  then show ?thesis
    using add.comm_neutral add_le_same_cancel1 get_winners_loss le_numeral_extra(3)
          max_v.simps verit_comp_simplify1(3) assms(1) assms(2) by metis
qed
```

This lemma states that the number of seats is not decreasing after assign seat function.

```
lemma assign_seat_mon:
  fixes
  rec::"('a::linorder, 'b) Divisor_Module" and
  winners::"'b Parties"
assumes
"inde < length (sl rec)" and
"winners = get_winners (fv rec) (p rec)"
shows "sl (assign_seat rec) ! inde ≥ sl rec ! inde"
proof -
  define rec' rec''
    where
     "rec'= (upd_round [hd winners] rec)" and
     "rec''= (break_tie winners rec)"
have "assign_seat rec =  (
      let winners = get_winners (fv rec) (p rec) in
      if length winners ≤ ns rec then
        let rec' =  (upd_round [hd winners] rec) in
                    (|res = (res rec'),
                          p = (p rec'),
                          i = (i rec'),
                          s = (s rec'),
                          ns = ((ns rec') - 1),
                          v = (v rec'),
                          fv = (fv rec'),
                          sl = (sl rec'),
                          d = (d rec')
                     |)
      else
```

```
        let rec'' = (break_tie winners rec) in
                    (|res = (res rec''),
                      p = (p rec''),
                      i = (i rec''),
                      s = (s rec''),
                      ns = 0,
                      v = (v rec''),
                      fv = (fv rec''),
                      sl = (sl rec''),
                      d = (d rec'')
                     |)" using rec''_def by simp
then show ?thesis proof(cases "length winners ≤ ns rec")
  case True
  then have "assign_seat rec = (|res = (res rec'),
                      p = (p rec'),
                      i = (i rec'),
                      s = (s rec'),
                      ns = ((ns rec') - 1),
                      v = (v rec'),
                      fv = (fv rec'),
                      sl = (sl rec'),
                      d = (d rec')
                     |)" using rec'_def assms
  by (smt (verit) assign_seat.simps)
  then have "sl (assign_seat rec) ! inde= sl (upd_round [hd winners] rec) ! inde"
    using rec'_def by simp
  then have "... ≥ (sl rec) ! inde"
    using assms assign_one_seat_mon by blast
  then show ?thesis
    using assms rec'_def
  by (metis ‹sl (assign_seat rec) ! inde = sl (upd_round [hd winners] rec) ! inde›)
  next
  case False
  define rec''
    where
     "rec''= (break_tie winners rec)"
  then have "assign_seat rec = (|res = (res rec''),
                      p = (p rec''),
                      i = (i rec''),
                      s = (s rec''),
                      ns = 0,
                      v = (v rec''),
                      fv = (fv rec''),
                      sl = (sl rec''),
                      d = (d rec'')
                     |)"
    using rec''_def assms False by force
```

```
  then have "sl (assign_seat rec) ! inde = sl (break_tie winners rec) ! inde"
    using rec''_def by simp
  then show ?thesis
    using ‹sl (assign_seat rec) ! inde = sl (break_tie winners rec) ! inde› by fastforce
qed
qed

lemma assign_seat_update:
  fixes
  rec::"('a::linorder, 'b) Divisor_Module"
  defines winners_def: "winners ≡ get_winners (fv rec) (p rec)"
  defines rec'_def: "rec' ≡ upd_round [hd winners] rec"
  assumes "length winners ≤ ns rec"
  shows "assign_seat rec = (|res = (res rec'),
                             p = (p rec'),
                             i = (i rec'),
                             s = (s rec'),
                             ns = ((ns rec') - 1),
                             v = (v rec'),
                             fv = (fv rec'),
                             sl = (sl rec'),
                             d = (d rec')|)" using assms
  by (smt (verit) assign_seat.simps)
```

This lemma states that the winner gets its seats increased

```
lemma assign_seat_seats_increased:
   fixes
  rec::"('a::linorder, 'b) Divisor_Module"
assumes
"winners = get_winners (fv rec) (p rec)" and
"length winners ≤ ns rec" and
"index (p rec) (hd winners) < length (sl rec)"
shows "sl (assign_seat rec) ! index (p rec) (hd winners) =
       sl rec ! index (p rec) (hd winners) + 1"
proof -
  define rec' where  "rec' ≡ (upd_round [hd winners] rec)"
  have "sl (assign_seat rec) =  sl ((|res = (res rec'),
                          p = (p rec'),
                          i = (i rec'),
                          s = (s rec'),
                          ns = ((ns rec') - 1),
                          v = (v rec'),
                          fv = (fv rec'),
                          sl = (sl rec'),
                          d = (d rec')
                         |))"
    using rec'_def assign_seat_update assms(1) assms(2) by metis
```

```
  then have "sl (assign_seat rec) ! ( index (p rec) (hd winners))
            = (sl rec') ! index (p rec) (hd winners)"
    by simp
  also have "... = sl (upd_round [hd winners] rec) ! ( index (p rec) (hd winners))"
    using rec'_def by simp
  also have "... = sl rec ! ( index  (p rec) (hd winners)) + 1"
    using assign_seat_increase_seats assms(3) list.sel(1) by metis
  finally have "sl (assign_seat rec) ! index (p rec) (hd winners)
                = sl rec ! index (p rec) (hd winners) + 1" by simp
  then show ?thesis
    using ‹sl (assign_seat rec) ! index  (p rec) (hd winners) =
          sl rec ! index  (p rec) (hd winners) + 1› by simp
qed
```

This lemma states that in case of two parties with one stronger than the other, then the weaker will not be in the list of winners.

```
lemma assign_seat_helper_lemma_helper:
  fixes
  v1::"rat"
assumes
  "m ≡ max_v (fv rec)" and
  "i2 ≡ index (p rec) party2" and
  "fv1 ≡ fv rec ! i1" and
  "fv2 ≡ fv rec ! i2" and
  "winners ≡ get_winners (fv rec) (p rec)" and
  "winners ≠ []" and
  "i1 < length (fv rec)" and
  "v1 > v2" and
  "fv1 ≡ v1 / (of_int (d rec ! (sl rec ! i1)))" and
  "fv2 ≡ v2 / (of_int (d rec ! (sl rec ! i2)))" and
  "(d rec) ! ((sl rec) ! i2) ≠ 0" and
  "sl rec ! i1 = sl rec ! i2" and
  "i1 ≠ i2" and
  "i1 < length (sl rec)" and
  "i2 < length (sl rec)"
  shows "party2 ≠ hd (winners)"
proof (cases "fv1 = max_v (fv rec)")
  case True
  then have "fv1 > fv2"
    using divide_strict_right_mono assms(10) assms(11) assms(12) assms(8) assms(9) by
force
  then have "fv2 ≠ max_v (fv rec)"
    using True dual_order.irrefl by simp
  then have "fv rec ! index (p rec) party2 ≠ m"
  using assms(1) assms(2) assms(4) by blast
  then have "fv2 ≠ m"
  using assms(2) assms(4) by blast
```

```
  then have "party2 ≠ hd winners"
    using ‹fv rec ! index (p rec) party2 ≠ m› get_winners_not_in_win assms(1) assms(6)

      assms(5) by metis
  then show ?thesis
    by simp
next
  case False
  then have "max_v (fv rec) > fv1"
    using False less_eq_rat_def max_val_wrap_lemma assms(3) assms(7) by blast
  then have "max_v (fv rec) ≠ fv2"
    using divide_right_mono less_le_not_le negative_zle of_int_nonneg assms(10) assms(12)

      assms(8) assms(9) by (smt (z3))
  then show ?thesis
    using assms(6) assms(2) assms(4) assms(5) get_winners_not_in_win by metis
qed


lemma assign_seat_helper_lemma_cond_ver:
  fixes
  v1::"rat"
assumes
  "index (p rec) party1 < length (fv rec)" and
  "v1 > v2" and
  "party2 = hd (get_winners (fv rec) (p rec))" and
  "fv rec ! index (p rec) party1 ≡ v1 / of_int(d rec ! (sl rec ! index (p rec) party1))"
and
  "fv rec ! index (p rec) party2 ≡ v2 / of_int(d rec ! (sl rec ! index (p rec) party2))"
and
  "(d rec) ! (sl rec !  index (p rec) party2) ≠ 0" and
  "sl rec ! index (p rec) party1 ≥ sl rec ! index (p rec) party2" and
  "length (get_winners (fv rec) (p rec)) ≤ ns rec" and
  "get_winners (fv rec) (p rec) ≠ []" and
  "index (p rec) party1 < length (sl rec)" and
  "index (p rec) party2 < length (sl rec)"
shows "sl (assign_seat rec) ! index (p rec) party1 ≥
       sl (assign_seat rec) ! index (p rec) party2"
proof(cases "sl rec ! index (p rec) party1 = sl rec ! index (p rec) party2")
  case True
 then have "fv rec ! index (p rec) party1 = v1 / (d rec) ! (sl rec ! index (p rec) party2)"

   using True of_int_of_nat_eq of_rat_divide of_rat_of_nat_eq assms(4)
   by (metis (no_types, lifting))
  also have "fv rec ! index (p rec) party1 = v2 / (d rec) ! (sl rec ! index (p rec) party2)"

    using True divide_le_cancel
          get_winners_not_in_win max_val_wrap_lemma of_int_of_nat_eq
```

```
        of_nat_le_0_iff verit_comp_simplify1(3) assms(1) assms(9) assms(2) assms(3)

        assms(4) assms(5) assms(6) by metis
  also have "party2 ≠ hd ( get_winners (fv rec) (p rec))"
    using True divide_cancel_right of_nat_0 of_nat_eq_iff of_rat_eq_iff
        verit_comp_simplify1(1) max_val_wrap_lemma assms(2) assms(6) calculation by
auto
  then show ?thesis
    using assms(3) by blast
next
  case False
  have "sl rec ! index (p rec) party1 > sl rec ! index (p rec) party2"
    using False le_neq_implies_less assms(7) leI le_antisym by linarith
  then have "index (p rec) party2 = index (p rec) (hd ( get_winners (fv rec) (p rec)))"

    using assms(3) by blast
  then have "sl (assign_seat rec) ! index (p rec) party2 =
           sl rec ! index (p rec) party2 + 1"
    using ‹index (p rec) party2 = index (p rec) (hd (get_winners (fv rec) (p rec)))›
        assign_seat_seats_increased False assms(8) assms(11) by metis
  then have "sl (assign_seat rec) ! index (p rec) party1 = sl rec ! index (p rec) party1"

    using assign_seat_not_winner_mantains_seats False assms(10) assms(3) by metis
  then show ?thesis
    using ‹sl (assign_seat rec) ! index (p rec) party2 = sl rec ! index (p rec) party2
+ 1›
        ‹sl rec ! index (p rec) party2 < sl rec ! index (p rec) party1› by linarith
qed

lemma assign_seat_ccontr:
  fixes
  v1::"rat" and v2::"rat"
assumes
  "index (p rec) party1 < length (fv rec)" and
  "v1 > v2" and
  "party2 = hd ( get_winners (fv rec) (p rec))" and
  "fv rec ! index (p rec) party1 = v1 / of_int(d rec ! (sl rec ! index (p rec) party1))"
and
  "fv rec ! index (p rec) party2 = v2 / of_int(d rec ! (sl rec ! index (p rec) party2))"
and
  "d rec ! (sl rec ! index (p rec) party2) ≠ 0" and
  "sl rec ! index (p rec) party1 ≥ sl rec ! index (p rec) party2" and
  "get_winners (fv rec) (p rec) ≠ []" and
  "index (p rec) party1 < length (sl rec)" and
  "index (p rec) party2 < length (sl rec)"
shows "sl (assign_seat rec) ! index (p rec) party1 ≥
      sl (assign_seat rec) ! index (p rec) party2"
proof(cases "length (get_winners (fv rec) (p rec)) ≤ ns rec")
```

```
  case True
  then show ?thesis
    using True assign_seat_helper_lemma_cond_ver assms(1) assms(8) assms(9)
          assms(10) assms(2) assms(3) assms(4) assms(5) assms(6) assms(7) by metis
next
  case False
   define rec''
    where
     "rec''= break_tie ( get_winners (fv rec) (p rec)) rec"
   have "sl (assign_seat rec) = sl rec''"
     using False rec''_def by simp
   then show ?thesis
     using assms(7) break_tie_lemma rec''_def by metis
qed

theorem assign_seat_concordant:
  fixes
  v1::"rat"
assumes
  "index (p rec) party1 < length (fv rec)" and
  "index (p rec) party2 < length (fv rec)" and
  "v1 > v2" and
  "party1 ∈ set (p rec)" and
  "party2 ∈ set (p rec)" and
  "get_winners (fv rec) (p rec) ≠ []" and
  "fv rec ! index (p rec) party1 =
    v1 / of_int (d rec ! (sl rec ! index (p rec) party1))" and
  "fv rec ! index (p rec) party2 =
    v2 / of_int (d rec ! (sl rec ! index (p rec) party2))" and
  "sl rec ! (index  (p rec) party1) ≥ sl rec ! (index  (p rec) party2)" and
  "d rec ! (sl rec ! (index  (p rec) party2)) ≠ 0" and
  "index (p rec) party1 < length (sl rec)" and
  "index (p rec) party2 < length (sl rec)"
shows "sl (assign_seat rec) ! index (p rec) party1 ≥
       sl (assign_seat rec) ! index (p rec) party2"
proof(cases "length (get_winners (fv rec) (p rec)) ≤ ns rec")
  case True
  then show ?thesis
      proof(cases "party1 = hd (get_winners (fv rec)  (p rec))")
        case True
        have "sl (assign_seat rec) ! ( index (p rec) party1) =
              (sl rec) ! (index (p rec) party1) + 1"
          using  ‹length ( get_winners (fv rec)  (p rec)) ≤ ns rec›
                 True assign_seat_seats_increased assms(11) by blast
        also have "sl (assign_seat  rec) ! index (p rec) party2 =
                   sl rec ! index (p rec) party2"
          using True assign_seat_not_winner_mantains_seats add.right_neutral
                add_less_same_cancel1 divide_cancel_right divide_less_cancel
```

```
            of_int_of_nat_eq of_nat_eq_0_iff assms(10) assms(12) assms(3)
            assms(6) assms(7) assms(8) get_winners_not_in_win
      by (metis (no_types, lifting))
      then show ?thesis
        using calculation trans_le_add1 assms(9) by presburger
    next
      case False
      then show ?thesis
      proof(cases "party2 = hd ( get_winners (fv rec) (p rec))")
        case True
        then show ?thesis
          using True assign_seat_ccontr assms(1) assms(10) assms(11) assms(12)
                assms(2) assms(3) assms(6) assms(7) assms(8) assms(9) by metis
      next
        case False
        have "party2 ≠ hd ( get_winners (fv rec) (p rec))"
          using False by simp
        then have "index (p rec) (hd ( get_winners (fv rec) (p rec))) ≠
                   index (p rec) party2"
          using False index_diff_elements assms(5) by metis
        then have "sl (assign_seat rec) ! ( index (p rec) party2) =
                   (sl rec) ! ( index (p rec) party2)"
          using False assms(12)
                assign_seat_not_winner_mantains_seats by metis
        have "party1 ≠ hd (get_winners (fv rec) (p rec))"
          using ‹party1 ≠ hd (get_winners (fv rec) (p rec))› by simp
        then have "index (p rec) (hd (get_winners (fv rec) (p rec))) ≠
                    index (p rec) party1"
          using index_diff_elements False assms(4) by metis
        then have "sl (assign_seat rec) ! ( index (p rec) party1) =
                    (sl rec) ! ( index (p rec) party1)"
          using ‹party1 ≠ hd ( get_winners (fv rec)  (p rec))›
                assign_seat_not_winner_mantains_seats assms(11) by metis
        then show ?thesis
          using ‹sl (assign_seat rec) ! index (p rec) party2 =
                 sl rec ! index (p rec) party2› assms(9) by presburger
        qed
      qed
next
  case False
  define rec''
    where
     "rec''= break_tie ( get_winners (fv rec) (p rec)) rec"
  have "assign_seat rec = (|res = (res rec''),
                        p = (p rec''),
                        i = (i rec''),
                        s = (s rec''),
```

```
                        ns = 0,
                        v = (v rec''),
                        fv = (fv rec''),
                        sl = (sl rec''),
                        d = (d rec'')
                       ⦈"
    using rec''_def assign_seat.simps False by auto
  then have "sl (assign_seat rec) = sl rec''"
    using False rec''_def by simp
  then have "sl (assign_seat rec) ! index (p rec) party1 =
             sl rec'' ! index (p rec) party1"
    by simp
  then have "... = sl rec ! (index (p rec) party1)"
    using break_tie_lemma rec''_def by metis
  also have "sl (assign_seat rec) ! (index (p rec) party2) =
             sl rec ! index (p rec) party2"
     using ‹sl (assign_seat rec) = sl rec''› break_tie_lemma rec''_def by metis
  then show ?thesis
  using False assms ‹sl (assign_seat rec) ! index (p rec) party1 =
         sl rec'' ! index (p rec) party1› calculation by metis
qed

lemma assign_seat_incre_case_helper:
assumes
  "⋀y. y ≠ index (p rec) party ⟹ y < length (fv rec) ⟹ fv rec ! y ≤ Max(set (fv
rec))"
  and
  "Max(set (fv rec')) > Max(set (fv rec))"
shows
  "⋀y. y ≠ index (p rec) party ⟹ y < length (fv rec) ⟹ fv rec ! y < Max(set (fv
rec'))"
  using assms dual_order.trans leD linorder_le_less_linear by meson
```

This lemma is one specific case of the monotonicity property written later. It is proving monotonicity property holds in the case the party after increasing votes wins the round.

```
lemma assign_seat_incre_case_TTTq:
assumes
  "p rec = p rec'" and
  "winners = get_winners (fv rec) (p rec)" and
  "winners' = get_winners (fv rec') (p rec')" and
  "party ∈ set (p rec)" and
  "sl rec' ! index (p rec) party ≥ sl rec ! index (p rec) party" and
  "index (p rec) party < length (sl rec)" and
  "index (p rec) party < length (sl rec')" and
  "length winners' ≤ ns rec'" and
  "length winners ≤ ns rec" and
  "party = hd (winners')"
```

```
shows "sl (assign_seat rec') ! (index (p rec) party) ≥
       sl (assign_seat rec) ! (index (p rec) party)"
proof(cases "party = hd winners")
  case True
  have "sl (assign_seat rec') ! index (p rec) party = sl rec' ! index (p rec) party +
1"
    using assign_seat_seats_increased assms(1) assms(10) assms(3) assms(7) assms(8)
    by metis
  have "sl (assign_seat rec) ! index (p rec) party = sl rec ! index (p rec) party + 1"

    using True assign_seat_seats_increased assms(2) assms(6) assms(9) by blast
  then show ?thesis
    using ‹sl (assign_seat rec') ! index (p rec) party = sl rec' ! index (p rec) party
+ 1›
    assms(5) by linarith
next
  case False
  have "sl (assign_seat rec') ! index (p rec) party = sl rec' ! index (p rec) party +
1"
  using assign_seat_seats_increased assms(1) assms(10) assms(3) assms(7) assms(8)
  by metis
  have "sl (assign_seat  rec) ! index (p rec) party = sl rec ! index (p rec) party"
    using False assign_seat_not_winner_mantains_seats assms(2) assms(4) assms(6)
          index_eq_index_conv by metis
  then show ?thesis
    using ‹sl (assign_seat rec') ! index (p rec) party = sl rec' ! index (p rec) party
+ 1›
        using assms(5) by linarith
qed
```

This lemma is one specific case of the monotonicity property written later. It is proving monotonicity property holds in the case there is a tie in both between the cases, before and after increasing votes of the party.

```
lemma assign_seat_incre_case_FFqq:
assumes
  "p rec = p rec'" and
  "winners = get_winners (fv rec) (p rec)" and
  "winners' = get_winners (fv rec') (p rec)" and
  "sl rec' ! index (p rec) party ≥ sl rec ! index (p rec) party" and
  "length winners' > ns rec'" and
  "length winners > ns rec"
shows "sl (assign_seat rec') ! index (p rec) party ≥
       sl (assign_seat rec) ! index (p rec) party"
  using assign_seat_break_tie_case assms by metis
```

This lemma is one specific case of the monotonicity property written later. It is proving monotonicity property holds in the case the party wins the round both before and after increasing its votes.

```
lemma assign_seat_incre_case_TqTT:
  fixes
    v::"rat"
assumes
  "p rec = p rec'" and
  "winners = get_winners (fv rec) (p rec)" and
  "winners' = get_winners (fv rec') (p rec)" and
  "sl rec = sl rec'" and
   "index (p rec) party < length (sl rec')" and
  "length winners' ≤ ns rec'" and
  "party = hd (winners')" and
  "party = hd (winners)"
shows "sl (assign_seat rec') ! (index (p rec) party) ≥
       sl (assign_seat rec) ! (index (p rec) party)"
proof(cases "length winners ≤ ns rec")
  case True
  have "sl (assign_seat rec') ! index (p rec) party = sl rec' ! index (p rec) party +
1"
    using assign_seat_seats_increased assms(1) assms(3) assms(5) assms(6) assms(7)
    by metis
  then have "sl (assign_seat rec) ! index (p rec) party = sl rec ! index (p rec) party
+ 1"
    using True assign_seat_seats_increased assms(2) assms(4) assms(5) assms(8) by metis

  then show ?thesis
    using ‹sl (assign_seat rec') ! index (p rec) party = sl rec' ! index (p rec) party
+ 1›
    order_refl assms(4) by presburger
  next
  case False
  have "sl (assign_seat rec') ! index (p rec) party = sl rec' ! index (p rec) party +
1"
    using assign_seat_seats_increased assms(1) assms(3) assms(5) assms(6) assms(7)
    by metis
  then have "sl (assign_seat rec) ! index (p rec) party = sl rec ! index (p rec) party"
    using False assign_seat_break_tie_case assms(2) less_or_eq_imp_le linorder_neqE_nat

    by metis
  then show ?thesis
    using ‹sl (assign_seat rec') ! index (p rec) party = sl rec' ! index (p rec) party
+ 1›
    le_add1 assms(4) by presburger
qed
```

This lemma is one specific case of the monotonicity property written later. It is proving monotonicity property holds in the case the party is winning the round after increasing votes.

```
lemma assign_seat_incre_case_TqTq:
assumes
  "p rec = p rec'" and
  "winners' = get_winners (fv rec') (p rec)" and
  "party ∈ set (p rec)" and
  "sl rec = sl rec'" and
  "sl rec' ! (index (p rec) party) ≥ sl rec ! (index (p rec) party)" and
  "index (p rec) party < length (sl rec')" and
  "length winners' ≤ ns rec'" and
  "party = hd (winners')"
shows "sl (assign_seat rec') ! (index (p rec) party) ≥
       sl (assign_seat rec) ! (index (p rec) party)"
  using assign_seat_break_tie_case assign_seat_incre_case_TTTq assign_seat_mon
        linorder_le_less_linear assms by metis
```

This lemma is one specific case of the monotonicity property written later. It is proving monotonicity property holds in the case the party before increasing votes wins the round and after increasing votes ends in a tie. This is a particular case of the following lemma.

```
lemma assign_seat_incre_case_FTqT:
  fixes
  v::"rat"
assumes
  "m' = Max(set (fv rec'))" and
  "winners = get_winners (fv rec) (p rec)" and
  "winners' = get_winners (fv rec') (p rec')" and
  "winners ≠ []" and "winners' ≠ []" and
  "p rec ≠ []" and
  "size (fv rec) = size (p rec)" and
  "size (fv rec') = size (p rec)" and
  "length (fv rec) = length (fv rec')" and
  "party ∈ set (p rec')" and
  "v' > v" and
  "fv rec ! index (p rec) party = v / of_int(d rec ! (sl rec ! index (p rec) party))"
and
  "fv rec' ! index (p rec) party = v' / of_int(d rec ! (sl rec' ! index (p rec) party))"
and
  "sl rec' ! index (p rec) party ≥ sl rec ! index  (p rec) party" and
  "d rec ! (sl rec ! index (p rec) party) ≠ 0" and
  "d rec ! (sl rec' ! index (p rec) party) ≠ 0" and
  "index (p rec) party < length (sl rec')" and
  "length (get_winners (fv rec') (p rec')) =
   length (filter (λx. (fv rec') ! x = m') [0..<length (fv rec')])"
  "index (p rec') party < length (fv rec')" and
  "index (p rec) party < length (sl rec)" and
  "index (p rec) party < length (fv rec)" and
  "⋀x. x ≠ index (p rec) party ⟹ (fv rec') ! x ≤ (fv rec) ! x" and
  "ns rec = ns rec'" and
```

```
  "length winners' > ns rec'" and
  "length winners ≤ ns rec" and
  "party = hd winners" and
  "p rec = p rec'"
shows "sl (assign_seat rec') ! index  (p rec) party ≥
       sl (assign_seat rec) ! index  (p rec) party"
proof -
  have "sl (assign_seat rec') = sl rec'"
    using assign_seat_break_tie_case assms(24) assms(3) by metis
  then show ?thesis
  proof(cases "sl rec' ! index  (p rec) party > sl rec ! index  (p rec) party")
   case True
   then have "sl rec' ! index (p rec) party = sl (assign_seat rec') ! index (p rec) party"
     using ‹sl (assign_seat rec') = sl rec'› by presburger
   then have "sl (assign_seat rec) ! index (p rec) party = sl rec ! index (p rec) party
+ 1"
     using assign_seat_seats_increased assms(2) assms(20) assms(25) assms(26) by blast
   then show ?thesis
     using Suc_eq_plus1 True ‹sl (assign_seat rec') = sl rec'› less_eq_Suc_le by metis
  next
   case False
   then have "(fv rec) ! index (p rec) party = Max(set (fv rec))"
     using assms(2) assms(26) assms(4) get_winners_not_in_win max_v.elims by metis
   then have "⋀y. y ≠ index (p rec) party ⟹ y < length (fv rec) ⟹
              fv rec ! y ≤ Max(set (fv rec))"
     by auto
   then have "Max(set (fv rec')) = (fv rec') ! index (p rec) party"
     using ‹fv rec ! index (p rec) party = Max(set (fv rec))› max_eqI False
           Fract_of_int_eq Fract_of_nat_eq divide_less_cancel int_ops(1)
           nat_less_le negative_eq_positive of_nat_less_0_iff of_nat_less_of_int_iff
           assms(11) assms(12) assms(13) assms(14) assms(16) assms(21) assms(22) assms(9)
     by (smt (verit, ccfv_threshold))
   then have "fv rec' ! index (p rec') party = m'"
     using assms(1) assms(27) by presburger
   then have "Max(set (fv rec')) > Max(set (fv rec))"
     using False ‹fv rec ! index (p rec) party = Max (set (fv rec))› assms(1) assms(11)

       assms(12) assms(13) assms(14) assms(16) assms(27) divide_strict_right_mono
       of_nat_le_0_iff by force
   then have "⋀y. y ≠ index (p rec) party ⟹ y < length (fv rec)
              ⟹ (fv rec) ! y < Max(set (fv rec'))"
     using ‹Max(set (fv rec')) > Max(set (fv rec))› assign_seat_incre_case_helper
      ‹⋀y. y ≠ index (p rec) party ⟹ y < length (fv rec)
              ⟹ (fv rec) ! y ≤ Max(set (fv rec))›
     by metis
   then have "⋀x. x ≠ index (p rec') party ⟹ x < length (fv rec') ⟹ (fv rec') !
x < m'"
```

```
    using assms(1) assms(22) assms(27) assms(9) leD linorder_le_less_linear
     max_v.simps max_val_wrap_lemma order_antisym_conv by metis
  then have "length (winners') = 1"
    using assms ‹Max(set (fv rec')) = (fv rec') ! index (p rec) party›
      ‹fv rec' ! index (p rec') party = m'›
      ‹⋀y. y ≠ index (p rec') party ⟹ y < length (fv rec') ⟹ (fv rec') ! y < m'›
      filter_size_is_one_helper_my_case_3 by metis
  then have "length winners' ≤ length winners"
    using assms(4) leI length_0_conv less_one by metis
  then show ?thesis
    using linorder_not_less assms(10) assms(23) assms(24) assms(25) by linarith
 qed
qed

lemma assign_seat_incre_case_FT:
  fixes
  v::"rat"
assumes
  "m' = Max(set (fv rec'))" and
  "winners = get_winners (fv rec) (p rec)" and
  "winners' = get_winners (fv rec') (p rec')" and
  "winners ≠ []" and "winners' ≠ []" and
  "p rec ≠ []" and
  "size (fv rec) = size (p rec)" and
  "size (fv rec') = size (p rec)" and
  "length (fv rec) = length (fv rec')" and
  "party ∈ set (p rec')" and
  "v' > v" and
  "fv rec ! index (p rec) party = v / of_int(d rec ! (sl rec ! index (p rec) party))"
and
  "fv rec' ! index (p rec) party = v' / of_int(d rec ! (sl rec' ! index (p rec) party))"
and
  "sl rec' ! index (p rec) party ≥ sl rec ! index (p rec) party" and
  "d rec ! (sl rec ! index (p rec) party) ≠ 0" and
  "d rec ! (sl rec' ! index (p rec) party) ≠ 0" and
  "index (p rec) party < length (sl rec')" and
  "length (get_winners (fv rec') (p rec'))
   = length (filter (λx. (fv rec') ! x = m') [0..<length (fv rec')])"
  "index (p rec') party < length (fv rec')" and
  "index (p rec) party < length (sl rec)" and
  "index (p rec) party < length (fv rec)" and
  "⋀x. x ≠ index (p rec) party ⟹ fv rec' ! x ≤ fv rec ! x" and
  "ns rec = ns rec'" and
  "length winners' > ns rec'" and
  "length winners ≤ ns rec" and
  "p rec = p rec'"
shows "sl (assign_seat rec') ! index (p rec) party ≥
       sl (assign_seat rec) ! index (p rec) party"
```

```
proof(cases "party = hd winners")
  case True

lemma assign_seat_case_TFqq:
assumes
  "winners = get_winners (fv rec) (p rec)" and
  "sl rec' ! index (p rec) party ≥ sl rec ! index (p rec) party" and
  "index (p rec) party < length (sl rec')" and
  "length winners > ns rec"
shows "sl (assign_seat rec') ! index (p rec) party ≥
       sl (assign_seat rec) ! index (p rec) party"
proof -
  have "sl (assign_seat rec) ! index (p rec) party = sl rec ! index (p rec) party"
    using assign_seat_break_tie_case assms(1) assms(4) by metis
  then have "sl (assign_seat rec') ! index (p rec) party ≥ sl rec' ! index (p rec) party"
    using assign_seat_mon assms(3) by blast
  then show ?thesis
    using ‹sl (assign_seat rec) ! index (p rec) party = sl rec ! index (p rec) party›
          assms(2) by linarith
qed
```

This lemma is one specific case of the monotonicity property written later. It is proving monotonicity property holds in the case the party before increasing votes wins but it does not after increasing votes.

```
lemma assign_seat_incre_case_T:
  fixes
  v::"rat"
assumes
  "m' = Max(set (fv rec'))" and
  "winners = get_winners (fv rec) (p rec)" and
  "winners' = get_winners (fv rec') (p rec)" and
  "winners ≠ []" and
  "size (fv rec) = size (p rec)" and
  "size (fv rec') = size (p rec)" and
  "party ∈ set (p rec')" and
  "v' > v" and
  "fv rec ! index (p rec) party = v / of_int(d rec ! (sl rec ! index (p rec) party))"
and
  "fv rec' ! index (p rec) party = v' / of_int(d rec ! (sl rec' ! index (p rec) party))"
and
  "sl rec' ! index (p rec) party ≥ sl rec ! index (p rec) party" and
  "d rec ! (sl rec' !  index (p rec) party) ≠ 0" and
  "index (p rec) party < length (sl rec')" and
  "length (get_winners (fv rec') (p rec'))
   = length (filter (λx. (fv rec') ! x = m') [0..<length (fv rec')])"
  "index (p rec') party < length (fv rec')" and
  "index (p rec) party < length (sl rec)" and
```

```
  "⋀x. x ≠ index (p rec) party ⟹fv rec' ! x ≤ fv rec ! x" and
  "length winners ≤ ns rec" and
  "party ≠ hd (winners')" and
  "party = hd winners" and
  "p rec = p rec'"
shows "sl (assign_seat rec') ! index (p rec) party ≥
       sl (assign_seat rec) ! index (p rec) party"
proof(cases "(sl rec) ! index (p rec) party = (sl rec') ! index (p rec) party")
  case True
  have "⋀y. y ≠ index (p rec) party ⟹ y < length (fv rec)
        ⟹ (fv rec) ! y ≤ Max(set (fv rec))"
    by simp
    then have "fv rec' ! index (p rec) party > fv rec ! index (p rec) party"
      using divide_strict_right_mono True of_int_0_less_iff of_nat_le_0_iff
           assms(8) assms(9) assms(10) assms(12) by fastforce
    then have "fv rec ! index (p rec) party = Max(set (fv rec))"
      using ‹party = hd winners› assms(2)assms(21) assms(7) assms(20) assms(4) assms(8)

        get_winners_loss hd_in_set by metis
     then have "Max(set (fv rec')) > Max(set (fv rec))"
      using ‹fv rec ! index (p rec) party < fv rec' ! index (p rec) party› assms(15)

            assms(21) max_v.elims dual_order.strict_trans1
            get_winners_weak_winner_implies_helper
     by metis
     then have "⋀y. y ≠ index (p rec) party ⟹ y < length (fv rec)
                ⟹ fv rec ! y < Max(set (fv rec'))"
    using ‹Max(set (fv rec')) > Max(set (fv rec))›
      ‹⋀y. y ≠ index (p rec) party ⟹ y < length (fv rec)
           ⟹ (fv rec) ! y ≤ Max(set (fv rec))›
    assign_seat_incre_case_helper by metis
  then have "⋀x. x ≠ index (p rec') party ⟹ x < length (fv rec') ⟹ (fv rec') !
x < m'"
    using assms(1) assms(17) assms(21) assms(5) assms(6) leD linorder_le_less_linear

      max_v.simps max_val_wrap_lemma order_antisym_conv by metis
  then have "(fv rec') ! index (p rec') party = m'"
    using ‹fv rec ! index (p rec) party < fv rec' ! index (p rec) party›
          ‹fv rec ! index (p rec) party = Max (set (fv rec))› assms(1) assms(15) assms(17)

          assms(21) assms(5) assms(6) max_eqI by metis
  then have "party = hd (get_winners (fv rec') (p rec))"
    using assms(7) assms(15) assms(1) assms(14) ‹(fv rec') ! index (p rec') party = m'›
          ‹⋀y. y ≠ index (p rec') party ⟹ y < length (fv rec') ⟹ (fv rec') ! y <
m'›
          get_winners_size_one assms(21) by metis
  then show ?thesis
```

```
  using assms(19) assms(3) by blast
next
  case False
  then have "sl rec' ! (index  (p rec) party) > sl rec ! index (p rec) party"
    using False assms(11) le_neq_implies_less by blast
  then have "sl (assign_seat rec') ! index  (p rec) party = sl rec' ! index (p rec) party"
    using assign_seat_not_winner_mantains_seats assms(13) assms(19)
      assms(21) assms(3) assms(7) index_eq_index_conv by metis
  then have "sl (assign_seat rec) ! index  (p rec) party = sl rec ! index (p rec) party
+ 1"
    using assign_seat_seats_increased assms(16) assms(2) assms(18) ‹party = hd winners›

    by blast
  then show ?thesis
    using ‹sl (assign_seat rec') ! index (p rec) party = sl rec' ! index (p rec) party›

        ‹sl rec ! index (p rec) party < sl rec' ! index (p rec) party› by linarith
  qed
```

This lemma is one specific case of the monotonicity property written later. It is proving monotonicity property holds in the case the party does not win in both cases.

```
lemma assign_seat_incre_case_F:
  fixes
  v::"rat"
assumes
  "winners = get_winners (fv rec) (p rec)" and
  "winners' = get_winners (fv rec') (p rec)" and
  "party ∈ set (p rec')" and
  "sl rec' ! (index  (p rec) party) ≥ sl rec ! (index  (p rec) party)" and
  "index (p rec) party < length (sl rec')" and
  "index (p rec) party < length (sl rec)" and
  "party ≠ hd (winners')" and
  "party ≠ hd (winners)" and
  "p rec = p rec'"
shows "sl (assign_seat rec') ! (index  (p rec) party) ≥
       sl (assign_seat rec) ! (index  (p rec) party)"
proof -
 have "sl (assign_seat rec') ! (index  (p rec) party) = sl rec' ! index (p rec) party"

   using assign_seat_not_winner_mantains_seats assms(7) assms(9) assms(2) assms(3)
        assms(5) index_diff_elements by metis
  then have "sl (assign_seat rec) ! (index  (p rec) party) = sl rec ! index (p rec) party"

    using assign_seat_not_winner_mantains_seats assms(1) assms(8) assms(9) assms(3)
       assms(6) index_diff_elements by metis
  then show ?thesis
```

```
    using ‹sl (assign_seat rec') ! index (p rec) party = sl rec' ! index (p rec) party›

      assms(4) by presburger
qed
```

This lemma is one specific case of the monotonicity property written later. It is proving monotonicity property holds in the case the party is not the winner after increasing the votes.

```
lemma assign_seat_incre_case_TTFq:
  fixes
  v::"rat"
assumes
  "m' = Max(set (fv rec'))" and
  "winners = get_winners (fv rec) (p rec)" and
  "winners' = get_winners (fv rec') (p rec)" and
  "winners ≠ []" and "winners' ≠ []" and
  "p rec ≠ []" and
  "size (fv rec) = size (p rec)" and
  "size (fv rec') = size (p rec)" and
  "party ∈ set (p rec')" and
  "v' > v" and
  "fv rec ! index (p rec) party = v / of_int(d rec ! (sl rec ! index (p rec) party))"
and
  "fv rec' ! index (p rec) party = v' / of_int(d rec ! (sl rec' ! index (p rec) party))"
and
  "sl rec' ! index  (p rec) party ≥ sl rec ! index  (p rec) party" and
  "d rec ! (sl rec !  index (p rec) party) ≠ 0" and
  "d rec ! (sl rec' !  index (p rec) party) ≠ 0" and
  "index (p rec) party < length (sl rec')" and
  "length (get_winners (fv rec') (p rec'))
   = length (filter (λx. (fv rec') ! x = m') [0..<length (fv rec')])"
  "index (p rec') party < length (fv rec')" and
  "index (p rec) party < length (sl rec)" and
  "index (p rec) party < length (fv rec)" and
  "⋀x. x ≠ index (p rec) party ⟹(fv rec') ! x ≤ (fv rec) ! x" and
  "length winners' ≤ ns rec'" and
  "length winners ≤ ns rec" and
  "party ≠ hd (winners')" and
  "p rec = p rec'"
shows "sl (assign_seat rec') ! index (p rec) party ≥
       sl (assign_seat rec) ! index (p rec) party"
proof(cases "party = hd winners")
  case True

lemma assign_seat_incre_case_TTqq:
  fixes
  v::"rat"
```

```
assumes
  "m' = Max(set (fv rec'))" and
  "winners = get_winners (fv rec) (p rec)" and
  "winners' = get_winners (fv rec') (p rec)" and
  "winners ≠ []" and "winners' ≠ []" and
  "p rec ≠ []" and
  "size (fv rec) = size (p rec)" and
  "size (fv rec') = size (p rec)" and
  "party ∈ set (p rec')" and
  "v' > v" and
  "fv rec ! index (p rec) party = v / of_int(d rec ! (sl rec ! index (p rec) party))"
and
  "fv rec' ! index (p rec) party = v' / of_int(d rec ! (sl rec' ! index (p rec) party))"
and
  "sl rec' ! index  (p rec) party ≥ sl rec ! index (p rec) party" and
  "d rec ! (sl rec ! index (p rec) party) ≠ 0" and
  "d rec ! (sl rec' ! index (p rec) party) ≠ 0" and
  "index (p rec) party < length (sl rec')" and
  "length (get_winners (fv rec') (p rec'))
   = length (filter (λx. (fv rec') ! x = m') [0..<length (fv rec')])"
  "index (p rec') party < length (fv rec')" and
  "index (p rec) party < length (sl rec)" and
  "index (p rec) party < length (fv rec)" and
  "⋀x. x ≠ index (p rec) party ⟹(fv rec') ! x ≤ (fv rec) ! x" and
  "length winners' ≤ ns rec'" and
  "length winners ≤ ns rec" and
  "p rec = p rec'"
shows "sl (assign_seat rec') ! index (p rec) party ≥
       sl (assign_seat rec) ! index (p rec) party"
proof(cases "party = hd winners'")
  case True
  then show ?thesis
    using True assign_seat_incre_case_TTTq assms(13) assms(16) assms(19) assms(2)
          assms(22) assms(23) assms(24) assms(3) assms(9) by metis
next
  case False
  then show ?thesis
    using ‹party ≠ hd winners'› assms assign_seat_incre_case_TTFq by metis
qed
```

This lemma is one specific case of the monotonicity property written later, it is a more generic case that will use some lemmas above and will be used in the most generic lemma later.

```
lemma assign_seat_monotone_case_T:
  fixes
  v::"rat"
assumes
```

```
  "m' = Max(set (fv rec'))" and
  "winners = get_winners (fv rec) (p rec)" and
  "winners' = get_winners (fv rec') (p rec)" and
  "winners ≠ []" and "winners' ≠ []" and
  "p rec ≠ []" and
  "size (fv rec) = size (p rec)" and
  "size (fv rec') = size (p rec)" and
  "party ∈ set (p rec')" and
  "v' > v" and
  "fv rec ! index (p rec) party = v / of_int(d rec ! (sl rec ! index (p rec) party))"
and
  "fv rec' ! index (p rec) party = v' / of_int(d rec ! (sl rec' ! index (p rec) party))"
and
  "sl rec' ! index (p rec) party ≥ sl rec ! index (p rec) party" and
  "d rec ! (sl rec ! index (p rec) party) ≠ 0" and
  "d rec ! (sl rec' ! index (p rec) party) ≠ 0" and
  "index (p rec) party < length (sl rec')" and
  "length (get_winners (fv rec') (p rec')) =
   length (filter (λx. (fv rec') ! x = m') [0..<length (fv rec')])" and
  "index (p rec') party < length (fv rec')" and
  "index (p rec) party < length (sl rec)" and
  "index (p rec) party < length (fv rec)" and
  "⋀x. x ≠ index (p rec) party ⟹(fv rec') ! x ≤ (fv rec) ! x" and
  "length winners' ≤ ns rec'" and
  "p rec = p rec'"
shows "sl (assign_seat rec') ! index (p rec) party ≥
       sl (assign_seat rec) ! index (p rec) party"
proof(cases "length winners ≤ ns rec")
  case True
  then show ?thesis using ‹length winners ≤ ns rec› assms
                    assign_seat_incre_case_TTqq by metis
next
  case False
  then show ?thesis
    using assign_seat_case_TFqq assms(13) assms(16) assms(2) linorder_le_less_linear

    by metis
qed
```

Monotonicity Theorem. The monotonicity property states that if one party gets the votes increased, then it cannot end with less number of seats it had before the increase.

```
theorem assign_seat_monotone:
  fixes
  v::"rat"
assumes
  "m' = Max(set (fv rec'))" and
  "winners = get_winners (fv rec) (p rec)" and
```

```
  "winners' = get_winners (fv rec') (p rec')" and
  "winners ≠ []" and "winners' ≠ []" and
  "p rec ≠ []" and
  "size (fv rec) = size (p rec)" and
  "size (fv rec') = size (p rec)" and
  "length (fv rec) = length (fv rec')" and
  "party ∈ set (p rec')" and
  "v' > v" and
  "fv rec ! index (p rec) party = v / of_int(d rec ! (sl rec ! index (p rec) party))"
and
  "fv rec' ! index (p rec) party = v' / of_int(d rec ! (sl rec' ! index (p rec) party))"
and
  "sl rec' ! index (p rec) party ≥ sl rec ! index (p rec) party" and
  "(d rec) ! ((sl rec) !  index (p rec) party) ≠ 0" and
  "(d rec) ! ((sl rec') !  index (p rec) party) ≠ 0" and
  "index (p rec) party < length (sl rec')" and
  "length (get_winners (fv rec') (p rec'))
    = length (filter (λx. (fv rec') ! x = m') [0..<length (fv rec')])" and
  "index (p rec') party < length (fv rec')" and
  "index (p rec) party < length (sl rec)" and
  "index (p rec) party < length (fv rec)" and
  "⋀x. x ≠ index (p rec) party ⟹ fv rec' ! x ≤ fv rec ! x" and
  "ns rec = ns rec'" and
  "p rec = p rec'"
shows "sl (assign_seat rec') ! index (p rec) party ≥
       sl (assign_seat rec) ! index (p rec) party"
proof(cases "length winners' ≤ ns rec'")
  case True
  then show ?thesis using ‹length winners' ≤ ns rec'› assms
                          assign_seat_monotone_case_T by metis
  next
  case False
  then have "length winners' > ns rec'"
    using False by simp
  then show ?thesis
  proof(cases "length winners ≤ ns rec")
    case True
    then show ?thesis
      using ‹length winners'>ns rec'› ‹length winners ≤ ns rec› assms
        assign_seat_incre_case_FT by metis
  next
    case False
    then show ?thesis
      using False assign_seat_break_tie_case assms
            linorder_le_less_linear assign_seat_case_TFqq
      by metis
  qed
```

```
qed

lemma nseats_decreasing:
  assumes
   non_empty_parties: "p rec ≠ []" and
   n_positive: "(ns rec) > 0"
  shows "ns (assign_seat rec) < ns rec"
proof (cases "length (get_winners (fv rec) (p rec)) ≤ ns rec")
  case True
  then have "ns (assign_seat rec) = ns rec - 1"
    by (auto simp add: Let_def)
  also have "... < ns rec"
    using True n_positive non_empty_parties by simp
  finally show ?thesis .
next
  case False
  then have "ns (assign_seat rec) = 0"
    by (auto simp add: Let_def)
  also have "... < ns rec" using n_positive
    by simp
  finally show ?thesis .
qed
```

A.3 Divisor Module

```
theory Divisor_Module
  imports Complex_Main
Main
"Component_Types/Social_Choice_Types/Preference_Relation"
"Component_Types/Social_Choice_Types/Profile"
"Component_Types/Social_Choice_Types/Result"
"Component_Types/Electoral_Module"
"Component_Types/Social_Choice_Types/Votes"
"Component_Types/Termination_Condition"
"HOL-Combinatorics.Multiset_Permutations"
"Component_Types/Consensus_Class"
"Component_Types/Distance"
"Component_Types/Assign_Seats"
begin
```

This record contains the list of parameters used in the whole divisor module.

```
record ('a::linorder, 'b) Divisor_Result =
  seats :: "'a::linorder Result"
  parties :: "'b Parties"
  seats_fun :: "('a::linorder, 'b) Seats"
  seats_list :: "nat list"
```

```
function loop_div ::
  "('a::linorder, 'b) Divisor_Module ⇒ ('a::linorder, 'b) Divisor_Module"
  where
  "ns r = 0  ⟹ loop_div r = r" |
  "ns r > 0 ⟹ loop_div r = loop_div (assign_seat r)"
  by auto

termination by (relation "measure (λr. ns r)")
               (auto simp add: Let_def nseats_decreasing)

lemma loop_div_lemma[code]: ‹loop_div r = (if ns r = 0 then r else loop_div (assign_seat
r))›
  by (cases r) auto
```

This function is executing the whole divisor method, taking in input the already initialized record and calculates the final output of the election. The Divisor Method is allocating a number of seats based on the number of votes of the candidates. Moreover, it is scaling the number of votes of every party with a factor proportional to the number of seats already assigned to that given party, to avoid unfairness and predominance of bigger parties. If there is a tie, the remaining seats will not be assigned and will remain "disputed".

```
fun divisor_method:: "('a::linorder, 'b) Divisor_Module ⇒ 'b Profile ⇒
                  ('a::linorder, 'b) Divisor_Module"
  where
"divisor_method rec pl = (
    let sv = calc_votes (p rec) (p rec) pl (v rec);
    sfv = start_fract_votes sv
    in loop_div (|
             res = res rec,
             p = p rec,
             i = i rec,
             s = s rec,
             ns = ns rec,
             v = sv,
             fv = sfv,
             sl = sl rec,
             d = d rec
            |))"

fun build_record :: "'b Parties ⇒ nat ⇒ (nat, 'b) Divisor_Module"
  where
  "build_record cp cns = (| res =  ({}, {}, {1..cns}), p = cp, i = {1..cns},
                            s = create_empty_seats {1..cns} cp, ns = cns,
                            v = start_votes cp, fv = [],
                            sl = start_seats cp, d = [] |)"

fun build_record_generic :: "'a::linorder list ⇒ 'b Parties ⇒ nat ⇒
                           ('a::linorder, 'b) Divisor_Module"
```

where
```
"build_record_generic l cp cns = (| res =  ({}, {}, (set l)), p = cp, i = (set l),
                                s = create_empty_seats (set l) cp, ns = cns,
                                v = start_votes cp, fv = [],
                                sl = start_seats cp, d = [] |)"
```

The D'Hondt method is the most classic variant of the Divisor Method, in which the factors used to scale the votes are natural number (1, 2, 3, ...). The function starts from a list of candidates, a list of ballots and the number of seats and fully executes the D'Hondt method. In this function, seats are identified by natural numbers.

```
fun dhondt_method :: "'b Parties ⇒ nat ⇒ 'b Profile ⇒
                  (nat, 'b) Divisor_Result"
  where
"dhondt_method partiti nseats pr =
    (let rec = build_record partiti nseats;
      result = divisor_method (rec(|d := upt 1 (ns rec)|)) pr in
        (| seats = res result,
           parties = p result,
           seats_fun = s result,
           seats_list = sl result|))"

fun dhondt_method_generic :: "'a::linorder list ⇒ 'b Parties ⇒ nat ⇒ 'b Profile ⇒
                  ('a::linorder, 'b) Divisor_Result"
  where
"dhondt_method_generic l partiti nseats pr =
    (let rec = build_record_generic l partiti nseats;
      result = divisor_method (rec(|d := upt 1 (ns rec)|)) pr in
        (| seats = res result,
           parties = p result,
           seats_fun = s result,
           seats_list = sl result|))"
```

The Sainte-Laguë method is a variant of the Divisor Method, in which the factors used to scale the votes are odd natural numbers (1, 3, 5, ...). The function starts from a list of candidates, a list of ballots and the number of seats and fully executes the Sainte-Laguë method. In this function, the seats are identified with natural numbers.

```
fun saintelague_method:: "'b Parties ⇒ nat ⇒ 'b Profile ⇒
                  (nat, 'b) Divisor_Result"
  where
"saintelague_method partiti nseats pr =
  (let rec = build_record partiti nseats;
      result = divisor_method (rec(|d := filter (λx. x mod 2 = 1) (upt 1 (2*ns rec))|))
pr in
        (| seats = res result,
           parties = p result,
           seats_fun = s result,
           seats_list = sl result|))"
```

```
fun saintelague_method_generic:: "'a::linorder list ⇒ 'b Parties ⇒ nat ⇒ 'b Profile
⇒
                  ('a::linorder, 'b) Divisor_Result"
  where
"saintelague_method_generic l partiti nseats pr =
  (let rec = build_record_generic l partiti nseats;
      result = divisor_method (rec⦇d := filter (λx. x mod 2 = 1) (upt 1 (2*ns rec))⦈)
pr in
        ⦇ seats = res result,
           parties = p result,
           seats_fun = s result,
           seats_list = sl result⦈)"
```