Lean4 Machine Assisted Proof Framework for Chip Firing Game & Graphical Riemann-Roch

Dhyey Dharmendrakumar Mavani

4th April 2025

Submitted to the
Department of Mathematics and Statistics
of Amherst College
in partial fulfillment of the requirements
for the degree of
Bachelor of Arts with Honors

Faculty Advisor(s): Professor Nathan Pflueger

Abstract

This thesis presents a novel, first-ever Lean4 formalization exploring the interplay between chip-firing games and algebraic geometry, culminating in the graph-theoretic analog of the Riemann–Roch theorem.

We focus on the *dollar game*, a variant of chip firing, for finite and connected graphs. Through a detailed study of chip firing moves and equivalence classes of chip distributions, we investigate key properties such as *winnability*—whether a given chip configuration can reach a debt-free state through a sequence of legal firing moves. To this end, we analyze and implement efficient algorithms for deciding winnability and characterizing winning strategies. We also examine the graph-theoretic Riemann–Roch theorem introduced by Baker and Norine, establishing a surprising bridge between discrete graph processes and algebraic geometry. We provide an accessible exposition of its statement and proof, leveraging divisor theory and linear equivalence in the context of graphs.

A key contribution of this work is developing a formalized proof framework for chip-firing games using the Lean4 proof assistant. By encoding multi-edged graph structures, firing rules, divisors, and related properties within Lean's theorem-proving environment, we produce verified machine-checked proofs of key results in chip firing and the Riemann-Roch setting assuming a modest set of prerequisites as axioms.

This work aims to enhance the Lean4's existing Mathlib4 library of proofs by providing modularized, formally verified domain content contributing to the growing body of computationally verified mathematics. We also briefly discuss the theoretical and practical implications of agentic-AI frameworks for advancing formal mathematics.

Acknowledgements

I want to take this opportunity to most sincerely thank my grandparents, sister, and parents for their unwavering support, sacrifice, and encouragement that have allowed me to continue my path at Amherst College for the last 4 years and beyond. Their unrelenting belief in me has been the wind that has propelled me forward, and I am forever grateful for their love and guidance.

I owe a deep debt of gratitude to my thesis advisor, Prof. Nathan Pflueger, for his insightful mentorship, thought-provoking discussions, and unshakable encouragement during this research and academics at Amherst. I am also immensely grateful to Prof. Emeritus Daniel Velleman for his prompt suggestions and comments on my work from his recent experience with Lean4. Their guidance and advice have been crucial in charting the research path for this thesis project.

I would also like to use this platform to thank my friends and colleagues at Amherst, specifically those at the Mathematics, Statistics, and Computer Science departments, for intellectual engagements and camaraderie. I want to thank Prof. Aamir Rasheed, Prof. Michael Ching, Prof. Rob Benedetto, Prof. Nicholas Horton, (late) Prof. Tanya Leise, Prof. Harris Daniels, Prof. Chris Elliott, and Prof. Joe Kraisler for feedback and for teaching through coursework, advising sessions, and cricket all around. I value all the friendships of the clubs, projects, and classes that made this experience enjoyable and engaging. A heartfelt thanks to the developers of Lean4 and Mathlib4, whose efforts to build formal proof systems have motivated and helped the computational material of this thesis. I owe my gratitude to (late) Prof. Lyle McGeoch for the original thesis.tex template.

Finally, thank you to all those who have helped me in small and large ways!

Contents

1 Introduction **1**

2 The Dollar Game **5**

2.1 Divisors & Linear Equivalence 6

2.2 Laplacian & Firing Script 18

3 Algorithms for Winnability **23**

3.1 Greedy Algorithm 23

3.2 “Benevolence” Algorithm 24

3.2.1 Preliminaries 24

3.2.2 Implementation & Setup 26

3.3 Configuration & Related Preliminaries 27

3.4 Dhar’s Algorithm 28

3.5 An Efficient Winnability Determination Algorithm 30

4 Riemann-Roch for Graphs & its Formalization **32**

4.1 Orientations and Genus 32

4.2 Rank 37

4.3 Riemann-Roch Theorem for Graphs 38

4.4 Application to Determination of Winnability 39

5 Formalization of Riemann-Roch for Chip Firing Graphs in Lean4 **41**

5.1 Introduction to Machine-Assisted Proving and Lean4 41

5.2 Riemann–Roch for Graphs in Lean4 43

5.3 Challenges in Formalization 45
5.4 Lessons Learned and Future Directions 48

Bibliography **51**

A Lean4 Implementation for Chip Firing & Graphical Riemann Roch **54**
A.1 Basic Properties for Chip Firing Graphs (Basic.lean) 54
A.2 Chip Firing Graphs Illustration (CFGraphExample.lean) 61
A.3 Configurations on Chip Firing Graphs (Config.lean) 65
A.4 Orientations and Directed Paths (Orientation.lean) 67
A.5 Rank and Genus (Rank.lean) 71
A.6 Helper Axioms, Lemmas and Theorems for Intermediate Results (Helpers.lean) . . 74
A.7 Intermediate Results for Riemann-Roch for Graphs (RRGHelpers.lean) 92
A.8 Riemann-Roch for Graphs and Relevant Corollaries (RiemannRochForGraphs.lean) 105
A.9 Package Main Index (ChipFiringWithLean.lean) 112
A.10 Lean CI/CD YAML Script for GitHub Actions (lean-action-ci.yml) 113
A.11 Lakefile Dependency and Library Management (lakefile.lean) 113
A.12 Streamlined Version Control (.gitignore) 114
A.13 Chip Firing Simulation with Algorithms for Winnability Determination in Python . 114
A.13.1 Laplacian Utility Class 114
A.13.2 Greedy Algorithm Class 115
A.13.3 Dhar's Algorithm Class 117
A.13.4 DollarGame Central Class Object 119
A.13.5 Main File for Initial Testing & Execution 120

B Additional Notes and Discussions **122**
B.1 Proof of Validity for Greedy Algorithm 122
B.1.1 Case 1: When a Solution Exists 122
B.1.2 Case 2: When No Solution Exists 123
B.2 Uniqueness Proof of Firing Script Produced by Greedy Algorithm 124
B.3 Proof of Validity for Dhar's Algorithm 124

B.4 A Peculiar Optimization of Winnability Determination Algorithm 125
B.5 Formalization of Riemann Roch Corollaries and Supporting Results 128
B.6 Parallels to Riemann-Roch for Riemann Surfaces 129

C PaperProof Generated Images for Proof Visualisation 131

List of Figures

2.1 Situational wealth distribution & relationship setup. 5

2.2 Application of set-firing moves leading to a win in the case of the divisor mentioned in Figure 2.1 13

3.1 Application of firing move by Charlie, for instance, in case of configurations w.r.t $q = Bob$ on the same graph as Figure 2.1 28

3.2 Application of Dhar's Algorithm on the same graph as Figure 3.1 29

C.1 PaperProof Visualisation for Loopless Boolean and Propositional Equivalence proof 132

C.2 PaperProof Visualisation for Linear Equivalence of Divisors is an Equivalence Relation proof 133

List of Notation

$G = (V, E)$ .... p. 6

$\mathrm{Div}(G)$ .... p. 7

$\deg(D)$ .... p. 7

$\mathrm{Div}^k(G)$ .... p. 8

$\mathrm{Div}_+(G)$ .... p. 8

$\mathrm{val}(v)$ .... p. 8

$D \xrightarrow{v} D'$ .... p. 11

$D \xrightarrow{W} D'$ .... p. 12

$D \sim D'$ .... p. 15

$D \geq 0$ .... p. 17

$[D]$ .... p. 17

$|D|$ .... p. 17

$\sigma \colon V \to \mathbb{Z}$ .... p. 18

$L \colon \mathbb{Z}^V \to \mathbb{Z}^V$ .... p. 19

$D \xrightarrow{\sigma} D'$ .... p. 19

$\widetilde{V} := V \setminus \{q\}$ .... p. 27

Config(G, q) .... p. 27

$\mathrm{outdeg}_S(v)$ .... p. 24

$\mathrm{indeg}_S(v)$ .... p. 24

q-reduced .... p. 24

$D' \prec D$ .... p. 25

$\mathcal{O}$ .... p. 32

$c(\mathcal{O})$ .... p. 33

K . . . . p. 33
g (genus) . . . . p. 35
$r(D)$ . . . . p. 37

Chapter 1

Introduction

In this thesis, we study *chip-firing games* on finite, connected graphs, focusing on a variant known as the *dollar game*. A chip-firing game is a discrete process in which vertices of a graph redistribute a commodity (often called chips, dollars, or sand) among their neighbors according to simple rules. In each firing move, one selects a vertex and has it send one chip along every edge to its adjacent vertices, thereby redistributing wealth in an "equitable" manner across the graph. Despite the simplicity of these rules, chip-firing games exhibit rich combinatorial structure and have been the subject of extensive study in graph theory and beyond [2]. One fundamental question is whether a given starting configuration of chips can reach a "stable" state (i.e. one where no vertex has debt) through a sequence of such firing moves. In the dollar game interpretation, vertices carry an integer representing money (positive for wealth and negative for debt), and the aim is to find a sequence of lending/borrowing moves that clears all debts. If such a sequence exists, the configuration is called **winnable**. This thesis examines the dollar game as a gateway to understanding the broader class of chip-firing games. It uses it as a running example to build up a more general theory.

Chip-firing games are not only entertaining puzzles but also carry significant mathematical interest. In combinatorics, they provide an elementary model for discrete dynamical systems on graphs and lead to deep invariants. Notably, any finite connected graph G gives rise to a finite abelian group associated with chip-firing often called the *sandpile group* or *critical group* of G [2, §1.3]. This group consists of all chip configurations modulo the "moves" of the game (intuitively, two configurations are equivalent if one can be reached from the other by a sequence of firings). These connections mean that chip-firing configurations can be studied with tools from algebraic graph theory and even draw on analogies with algebraic geometry. Indeed, Baker and Norine [1] famously established a graph-theoretic version of the Riemann–Roch theorem, showing that divisors on a finite graph (which correspond to chip configurations) satisfy a formula analogous to the classical Riemann–Roch formula for Riemann surfaces. This result revealed an unexpected bridge between combinatorial game moves and concepts like linear systems of divisors, genus, and rank in algebraic geometry. In a similar vein, chip-firing games have been employed to prove theorems in algebraic geometry. Cools, Draisma, Payne, and Robeva [14] utilized tropical geometry—a combinatorial shadow of algebraic geometry—to provide a novel proof of the Brill–Noether theorem, a fundamental result concerning special divisors on algebraic curves. Additionally, research from Pflueger [15] has explored the Brill–Noether varieties of k-gonal curves, further illustrating the deep connections between chip-firing dynamics and algebraic geometry. In this way, the dollar game and its relatives serve as a concrete illustration of how a seemingly simple graph process connects to advanced mathematical theories.

From an algorithmic perspective, chip-firing games pose interesting challenges and insights. There are efficient procedures to determine if a given configuration is winnable; for example, variants of Dhar's "burning" algorithm can test the reachability of a debt-free state in the dollar game. More generally, one can algorithmically search for a sequence of firing moves that leads to a desired configuration, and this thesis explores such methods in the context of the dollar game. However, not all questions about chip-firing are easy: computing certain invariants can be computationally demanding. In particular, the *rank* of a divisor (a chip configuration) on a graph—a central notion in the Riemann–Roch graph theory—is generally difficult to compute. It has been proven that determining the rank of a given divisor on a graph is an NP-hard problem [5]. This complexity result underscores that, while the firing rules are simple, the space of possible move sequences and outcomes can grow exponentially, making some problems in this domain intractable for large inputs. These considerations motivate the development of efficient algorithms for special cases and a careful theoretical understanding of chip-firing dynamics.

Another key aspect of our work is the incorporation of *computational proof verification* into the study of chip-firing games. Ensuring that mathematical proofs are correct is crucial, especially for intricate results like the graph Riemann–Roch theorem. In recent years, *interactive theorem provers* or *proof assistants* have emerged as powerful tools for checking the correctness of proofs by computer. Systems such as Lean4, Coq, and Isabelle allow one to formalize definitions, theorems, and proofs in a rigorous language that a computer can verify step by step [6, 7, 8]. Lean4, in particular, is a modern proof assistant and programming language designed for both expressiveness and efficiency in formalizing mathematics. Using such a system, we can develop a machine-checked theory of chip-firing games, eliminating ambiguity or gaps in informal proofs. This approach adds a layer of reliability to the results and aligns with a broader trend in mathematics to use software for verifying proofs. Notable recent successes of formal verification include the fully machine-checked proofs of profound mathematical results—such as the Prime Number Theorem, the Fundamental Theorem of Algebra, and Gödel's Incompleteness Theorems—all formalized in Lean as part of the community effort to verify Wiedijk's list of 100 Theorems [12]. These achievements demonstrate that proof assistants are now capable of handling sophisticated and historically challenging theories. Inspired by these advances, we build a rigorous machine-assisted proving framework for chip-firing games using Lean4. In particular, we encode graphs, divisors (chip distributions), and firing moves in Lean4's language and formalize key propositions and invariants of the dollar game. This includes laying the groundwork for a formal proof of the Baker–Norine Riemann–Roch theorem on graphs. All Lean4 code developed for this project thus far is included alongside the mathematical exposition; we employ a custom LaTeX listing style from Wang et al. [3] to seamlessly embed the code snippets for readability. Integrating formal proofs into the narrative validates our theoretical claims and showcases how computational proof assistants can be applied to combinatorics and algebra.

The remainder of this thesis is organized as follows. Chapter 2 introduces the dollar game in detail, providing definitions and examples that ground the abstract concept of chip-firing in a tangible scenario. We formally define what it means for a configuration to be winnable and illustrate the chip-firing process through an example walkthrough and initial Lean4 verification of simple cases. Chapter 3 focuses on algorithms for winnability: We present methods to decide if a given game configuration can reach a debt-free state, including a discussion of efficient algorithms and their correctness. This chapter connects the intuitive idea of "firing until done" with concrete procedures and touches on complexity considerations for more general settings. In Chapter 4, we develop the theoretical framework necessary to understand the Riemann–Roch theorem for graphs. We introduce the language of divisors on graphs, linear equivalence (chip-firing moves viewed as an equivalence relation), and important concepts like q-reduced divisors and the graph's genus. With these tools in hand, we present the statement of the Riemann-Roch theorem and outline its proof, highlighting how chip-firing games provide the combinatorial backbone of this result. Chapter 5 then turns to the complete formalization of the chip-firing theory and the Riemann–Roch theorem in Lean4. In this chapter, we describe the implementation of our definitions and theorems in the Lean4 proof assistant, detail the structure of the formal proofs, and report on the extent to which the Riemann–Roch theorem has been verified in our framework. We emphasize the correspondence between the informal mathematical arguments and the formal proof script, illustrating the successes and challenges encountered in the formalization process. Finally, Appendix A contains the complete Lean4 code developed under this initiative thus far. This appendix is a reference point for the formal developments and includes additional commentary in the code to help readers navigate the Lean definitions and proofs. The appendix is structured into multiple sections (A.1-A.12) covering various aspects of chip firing graphs, divisors, configurations, orientations, rank, genus, and ultimately the proof of the Riemann-Roch theorem for graphs. We also provide Python code in section A.13, which implements an initial version of the chip firing setup, "Greedy" Algorithm, and Dhar's Algorithm in an object-oriented manner for better visualization and understanding. Appendix B includes additional notes and discussions regarding proofs related to validity of algorithms, uniqueness properties, and some peculiar optimizations. Appendix C contains generated images for proof visualization to aid intuitive understanding of the theoretical concepts.

Chapter 2

The Dollar Game

Consider a graph $G = (V, E)$ with V as a set of vertices representing people and E as a set of edges representing relationships between them. The more edges between individuals, the stronger the relationship. At each vertex, we also record their wealth via an integer representing the number of dollars they have, with negative values indicating debt. The goal is to find a sequence of lending/borrowing moves so everyone becomes debt-free. However, in one move, a vertex can lend money (*fire*) or borrow money by taking or sending 1 unit of currency across each edge it shares in the graph. This is called the *dollar game on G*, and if such a **sequence exists**, the game is said to be **winnable**.

Let us walk through an example in Figure 2.1 to illustrate the setup better.

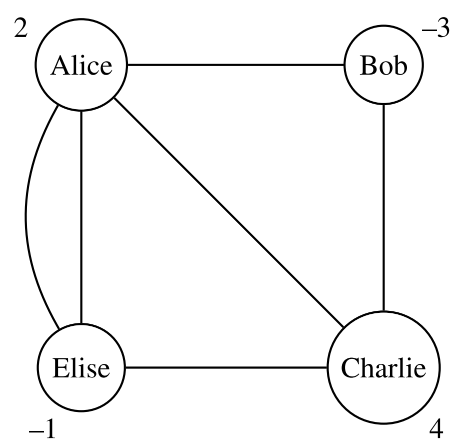

Figure 2.1: Situational wealth distribution & relationship setup.

Alice starts with a wealth of 2, Bob is in debt with -3, Charlie has 4, and Elise is slightly in debt with -1. The edges between vertices represent relationships; in each turn, a person can either lend, borrow, or do nothing with 1 through **all** the edges they are connected to. For instance, Charlie can lend 1 to each of Elise, Alice, and Bob, which takes Elise out of debt, leaving Alice with 2 and Bob with -2. The game continues until no one remains in debt, and if such a redistribution is possible, the game is considered **winnable**.

Before exploring solutions to this game, let us mathematically formalize our established setup.

2.1 Divisors & Linear Equivalence

When we mention a **graph**, we refer to a *finite, connected, undirected multigraph without loop edges*. Essentially, a **multigraph** $G = (V, E)$ consists of two components: a set of vertices V and a multiset of edges E, where each edge is an unordered pair $\{v, w\}$ representing connections between vertices. The prefix "multi" means that pairs like $\{v, w\}$ can appear multiple times in E. We often simplify notation by writing an edge as vw. A multigraph G is considered finite if both V and E are finite and connected if there is a path of edges between any two vertices.

Let us formalize this structure using Lean4—a functional programming language with a powerful type system explicitly designed for mathematical formalization. Lean4 enables us to define precise mathematical objects, such as sets and algebras, building upon existing axioms. It allows us to prove these objects' properties with machine-checked accuracy rigorously. Throughout this chapter, we will present commented Lean4 code snippets, each accompanied by detailed explanations in the text right after the code blocks.

```
-- Assume V is a finite type with decidable equality
variable {V : Type} [DecidableEq V] [Fintype V]

-- Define a set of edges to be loopless only if it doesn't have loops
def isLoopless (edges : Multiset (V × V)) : Bool :=
  Multiset.card (edges.filter (λ e => (e.1 = e.2))) = 0

def isLoopless_prop (edges : Multiset (V × V)) : Prop :=
  ∀ v, (v, v) ∉ edges

-- Define a set of edges to be undirected only if it doesn't have both (v, w) and (w,
    v)
def isUndirected (edges : Multiset (V × V)) : Bool :=
  Multiset.card (edges.filter (λ e => (e.2, e.1) ∈ edges)) = 0

def isUndirected_prop (edges : Multiset (V × V)) : Prop :=
  ∀ v1 v2, (v1, v2) ∈ edges → (v2, v1) ∉ edges
```

```
-- Multigraph with undirected and loopless edges
structure CFGraph (V : Type) [DecidableEq V] [Fintype V] :=
  (edges : Multiset (V × V))
  (loopless : isLoopless edges = true)
  (undirected: isUndirected edges = true)
```

Let us walk through this Lean4 code. We first declare a `variable V`, representing our vertex set, along with assumptions `DecidableEq V`, ensuring computable equality between vertices, and `Fintype V`, confirming the finiteness of `V`. The `isLoop` function checks whether a given edge connects a vertex to itself, returning a boolean (`Bool`), which is particularly useful for algorithmic implementations and examples as it allows efficient, conditional checks within code. Subsequently, the `isLoopless` function determines if the given Multiset of edges contains loops by filtering edges that connect vertices to themselves (using `.filter` method) and by requiring the cardinality of this filtered set (using `Multiset.card`) to be zero. Alongside this boolean function, a propositional logic counterpart, `isLoopless_prop`, explicitly states that no vertex can form a loop with itself, useful in formal proofs due to its precise logical formulation.

Similarly, we introduce `isUndirected`, another boolean function ensuring our edge set contains no reversed duplicates—edges appearing as both `(v, w)` and `(w, v)`. Its corresponding propositional form, `isUndirected_prop`, explicitly forbids such bidirectional duplications. [1] Finally, we integrate these properties into a cohesive structure, `CFGraph`, encapsulating a multiset of edges with built-in constraints of looplessness and undirectedness.

Definition 2.1.1. A **divisor** on the graph G is an element of the *free abelian group* on its vertices:

$$\mathrm{Div}(G) = \mathbb{Z}V = \left\{ \sum_{v \in V} D(v)v \colon D(v) \in \mathbb{Z} \right\}.$$

Divisors can be thought of as ways to describe the wealth distribution on G. If $D = \sum_{v \in V} D(v) \cdot v \in \mathrm{Div}(G)$, then $D(v)$ gives the amount of money at the vertex (or person) v, with negative values representing debt. The total money in the system is captured by the *degree* of the divisor as defined below.

[1] We present proof that the boolean and propositional logic declarations for looplessness and undirectedness conditions are equivalent statements in Appendix A.1.

Definition 2.1.2. The **degree** of a divisor $D = \sum_{v \in V} D(v) \cdot v \in \text{Div}(G)$ is defined as:

$$\deg(D) = \sum_{v \in V} D(v).$$

For instance, from the example presented earlier in Figure 2.1, the divisor D can be represented as $D = 2(A) - 3(B) + 4(C) - (E)$, and thus the $\deg(D) = 2 - 3 + 4 - 1 = 2$.

We use $\text{Div}^k(G)$ to denote all divisors with degree k, and $\text{Div}_+(G)$ for divisors with a non-negative degree. Note that the word "degree" can refer to two things in our sub-domain at the intersection of graph theory and combinatorics, so for clarity, we define $\text{val}(v)$ (valence of v) as the number of edges connected to v.

Now, we formalize the above definitions in Lean4 as follows:

```
-- Divisor as a function from vertices to integers
def CFDiv (V : Type) := V → ℤ

-- Number of edges between two vertices as an integer
def num_edges (G : CFGraph V) (v w : V) : ℤ :=
  ↑(Multiset.card (G.edges.filter (λ e => e = (v, w) ∨ e = (w, v))))

-- Degree (Valence) of a vertex as an integer
def vertex_degree (G : CFGraph V) (v : V) : ℤ :=
  ↑(Multiset.card (G.edges.filter (λ e => e.fst = v ∨ e.snd = v)))
```

In Lean4, `def CFDiv (V : Type) := V → ℤ` defines a divisor as a function from vertices to integers—simple yet elegant! The `num_edges` function counts edges between two vertices (considering both directions since the graph is undirected), and `vertex_degree` calculates how many edges connect to a vertex, matching our intuitive notion of "relationships" in the dollar game. The lambda notation (λ) in Lean4 succinctly defines anonymous functions inline, specifying their inputs and immediately describing their output behavior without needing a separate named declaration. Please note that ↑ is a casting symbol used to convert the resulting output to our preferred integer ($\mathbb{Z}$) type.

Now, after that mouthful of definitions, let us take a step back and establish the example we discussed in Figure 2.1 in Lean4 and verify the conditions of the graph along with the structural properties to ensure that our setup is functioning as mathematically intended.

```
inductive Person : Type
  | A | B | C | E
  deriving DecidableEq

instance : Fintype Person where
```

```
  elems := {Person.A, Person.B, Person.C, Person.E}
  complete := by {
    intro x
    cases x
    all_goals { simp }
  }
```

The line `inductive Person : Type` defines a new type called `Person` with four possible values: `A`, `B`, `C`, and `E`, representing Alice, Bob, Charlie, and Elise. One can think of `inductive` as a way to create a custom set of (finite) options. The `deriving DecidableEq` part ensures Lean4 can check if two `Person` values are equal (e.g., `Person.A = Person.B` is false).

Following this, we declare an `instance` of the type class `Fintype` (finite type) for `Person`. This instance explicitly enumerates all possible elements of `Person` and provides a completeness proof verifying that no other elements can exist beyond those listed. The proof is constructed using several Lean4 tactics. A tactic in Lean4 is essentially an instruction to automate routine proof steps. [6] The tactic `intro` introduces a new arbitrary element `x` of type `Person` into our proof context. Next, the tactic `cases` is applied to `x`, instructing Lean4 to systematically consider each possible case of `x` (`A`, `B`, `C`, or `E`). Finally, `all_goals` instructs Lean4 to apply the tactic `simp` (simplify) to all generated subgoals simultaneously. The `simp` tactic automatically resolves straightforward logical statements, confirming that the explicitly defined set of elements accounts for each introduced case.

```
-- Example usage for Person type in a loopless graph
def exampleEdges : Multiset (Person × Person) :=
  Multiset.ofList [
    (Person.A, Person.B),
    (Person.B, Person.C),
    (Person.C, Person.E)
  ]
theorem loopless_example_edges : isLoopless exampleEdges = true := by rfl
theorem undirected_example_edges : isUndirected exampleEdges = true := by rfl

-- Example usage for Person type in a graph with a loop
def edgesWithLoop : Multiset (Person × Person) :=
  Multiset.ofList [
    (Person.A, Person.B),
    (Person.A, Person.A),   -- This is a loop
    (Person.B, Person.C),
  ]
theorem loopless_test_edges_with_loop : isLoopless edgesWithLoop = false := by rfl

-- Example usage for Person type in a graph with a non-undirected edge
def edgesWithNonUndirected : Multiset (Person × Person) :=
  Multiset.ofList [
```

```
    (Person.A, Person.B),
    (Person.B, Person.C),
    (Person.C, Person.E),
    (Person.E, Person.C)  -- This is a non-undirected edge
  ]
theorem undirected_test_edges_with_non_undirected : isUndirected
    edgesWithNonUndirected = false := by rfl
```

Subsequently, we define specific multisets of edges representing various graph structures: a loopless, undirected example, a graph containing a loop, and another containing non-undirected edges. Each definition is accompanied by theorem declarations, such as `loopless_example_edges`, verified using the Lean4 tactic `rfl`. This tactic, `rfl`, instructs Lean4 to automatically verify straightforward equality proofs, simplifying the proof process by eliminating the need to write out repeating steps explicitly. Employing tactics like `rfl` significantly streamlines formal verification, highlighting one of Lean4's key strengths—automating mechanical aspects of proofs that would typically be glossed over in standard mathematical exposition.

```
def example_graph : CFGraph Person := {
  edges := Multiset.ofList [
    (Person.A, Person.B), (Person.B, Person.C),
    (Person.A, Person.C), (Person.A, Person.E),
    (Person.A, Person.E), (Person.E, Person.C)
  ],
  loopless := by rfl,
  undirected := by rfl
}

def initial_wealth : CFDiv Person :=
  fun v => match v with
  | Person.A => 2
  | Person.B => -3
  | Person.C => 4
  | Person.E => -1

-- Test vertex degrees
theorem vertex_degree_A : vertex_degree example_graph Person.A = 4 := by rfl
theorem vertex_degree_B : vertex_degree example_graph Person.B = 2 := by rfl
theorem vertex_degree_C : vertex_degree example_graph Person.C = 3 := by rfl
theorem vertex_degree_E : vertex_degree example_graph Person.E = 3 := by rfl

-- Test edge counts
theorem edge_count_AB : num_edges example_graph Person.A Person.B = 1 := by rfl
theorem edge_count_BA : num_edges example_graph Person.B Person.A = 1 := by rfl
theorem edge_count_BC : num_edges example_graph Person.B Person.C = 1 := by rfl
theorem edge_count_CB : num_edges example_graph Person.C Person.B = 1 := by rfl
theorem edge_count_AC : num_edges example_graph Person.A Person.C = 1 := by rfl
```

```
theorem edge_count_CA : num_edges example_graph Person.C Person.A = 1 := by rfl
theorem edge_count_AE : num_edges example_graph Person.A Person.E = 2 := by rfl
theorem edge_count_EA : num_edges example_graph Person.E Person.A = 2 := by rfl
theorem edge_count_EC : num_edges example_graph Person.E Person.C = 1 := by rfl
theorem edge_count_CE : num_edges example_graph Person.C Person.E = 1 := by rfl
theorem edge_count_BE : num_edges example_graph Person.B Person.E = 0 := by rfl
theorem edge_count_EB : num_edges example_graph Person.E Person.B = 0 := by rfl

-- Test No self-loops
theorem edge_count_AA : num_edges example_graph Person.A Person.A = 0 := by rfl
theorem edge_count_BB : num_edges example_graph Person.B Person.B = 0 := by rfl
theorem edge_count_CC : num_edges example_graph Person.C Person.C = 0 := by rfl
theorem edge_count_EE : num_edges example_graph Person.E Person.E = 0 := by rfl
```

Next, `def example_graph : CFGraph Person` creates our graph, specifying its edges as pairs like `(Person.A, Person.B)`. The `initial_wealth` function assigns starting dollar amounts to each Person, using a `match` expression to map each `Person` to an integer. Finally, theorem statements such as `theorem vertex_degree_A` prove that the graph structure is initialized as intended, with by `rfl` telling Lean4 to verify this by simple computation.

As one can see in this example, solving the game or, let alone even defining a game, can be and often is a non-trivial task in Lean4.

Now, let us define lending and borrowing moves for the game.

Definition 2.1.3. Given divisors $D, D' \in \mathrm{Div}(G)$ and a vertex $v \in V$, we say D' is obtained from D by a *lending move* at v, written as $D \xrightarrow{v} D'$, if:

$$D' = D - \sum_{vw \in E} (v - w) = D - \mathrm{val}(v) \cdot v + \sum_{vw \in E} w.$$

Similarly, D' is obtained from D by a *borrowing move* at v, written as $D \xleftarrow{v} D'$, if:

$$D' = D + \sum_{vw \in E} (v - w) = D + \mathrm{val}(v) \cdot v - \sum_{vw \in E} w.$$

We formalize the above definition in Lean4 as follows:

```
-- Firing move at a vertex
def firing_move (G : CFGraph V) (D : CFDiv V) (v : V) : CFDiv V :=
  λ w => if w = v then D v - vertex_degree G v else D w + num_edges G v w

-- Borrowing move at a vertex
def borrowing_move (G : CFGraph V) (D : CFDiv V) (v : V) : CFDiv V :=
  λ w => if w = v then D v + vertex_degree G v else D w - num_edges G v w
```

```
-- Define finset_sum using Finset.fold
def finset_sum {α β} [AddCommMonoid β] (s : Finset α) (f : α → β) : β :=
  s.fold (· + ·) 0 f

-- Define set_firing to use finset_sum with consistent types
def set_firing (G : CFGraph V) (D : CFDiv V) (S : Finset V) : CFDiv V :=
  λ w => D w + finset_sum S (λ v => if w = v then -vertex_degree G v else num_edges G
    v w)
```

Further extending our implementation, the definitions `firing_move` and `borrowing_move` model the precise mechanics of wealth redistribution in the dollar game. The definition `set_firing` employs `finset_sum`, which leverages `Finset.fold` to systematically apply an additive operation across a finite set of vertices. Here, `Finset.fold` is part of Lean4's Mathlib library, a comprehensive mathematics library that includes algebraic structures such as `AddCommMonoid`. The `AddCommMonoid` structure specifically provides an algebraic framework ensuring addition is associative, commutative, and has an identity element (zero), enabling seamless and mathematically rigorous summations over finite sets.

What is interesting here is that the order in which lending or borrowing happens does not matter. This gives the dollar game an **abelian property**, meaning the operations commute with each other.

Definition 2.1.4. Suppose D' is obtained from D by lending from all the vertices in some subset $W \subseteq V$. In this case, we call this a *set-lending* (or *set-firing*) move by W, denoted $D \xrightarrow{W} D'$.

Using these definitions, let us try to perform set-firing on the example divisor shown in Figure 2.1 earlier in this chapter.

As shown in Figure 2.2, we can have a firing-set $W_1 = \{A, E, C\}$. After this firing move, all the internal lending and borrowing between the members of the firing set cancels out, and effective lending of 2 happens to B from A & C lending 1 each. Now, to get B out of debt, we repeat the same set-firing move on this newly obtained divisor with $W_2 = W_1$. This gives us the divisor that can be represented as $0(A) + 1(B) + 2(C) - 1(E)$. Finally, to get E out of debt, we can carefully engineer our firing set to be $W_3 = \{B, C\}$. This ensures we take the minimum number of lenders out of debt (vaguely speaking). After this move, as shown in the figure, all the graph members come out of debt, signifying that we have won the game!

Let us walk through this example in Lean4 and ensure our code compiles. This will ensure that we have correctly defined the core firing-move-related properties of the chip-firing graphs.

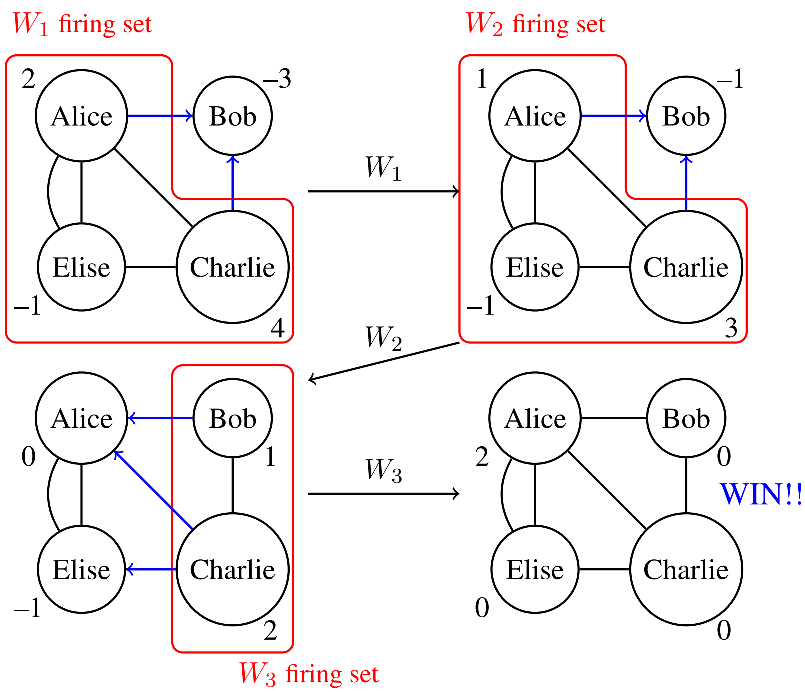

Figure 2.2: Application of set-firing moves leading to a win in the case of the divisor mentioned in Figure 2.1

```
-- Test Charlie lending through an individual firing move
def after_charlie_lends := firing_move example_graph initial_wealth Person.C
theorem charlie_wealth_after_lending : after_charlie_lends Person.C = 1 := by rfl
theorem bob_wealth_after_charlie_lends : after_charlie_lends Person.B = -2 := by rfl

-- Test set firing W₁ = {A,E,C}
def W₁ : Finset Person := {Person.A, Person.E, Person.C}
def after_W₁_firing := set_firing example_graph initial_wealth W₁
theorem alice_wealth_after_W₁ : after_W₁_firing Person.A = 1 := by rfl
theorem bob_wealth_after_W₁ : after_W₁_firing Person.B = -1 := by rfl
theorem charlie_wealth_after_W₁ : after_W₁_firing Person.C = 3 := by rfl
theorem elise_wealth_after_W₁ : after_W₁_firing Person.E = -1 := by rfl

-- Test set firing W₂ = {A,E,C}
def W₂ : Finset Person := W₁
def after_W₂_firing := set_firing example_graph after_W₁_firing W₂
theorem alice_wealth_after_W₂ : after_W₂_firing Person.A = 0 := by rfl
theorem bob_wealth_after_W₂ : after_W₂_firing Person.B = 1 := by rfl
theorem charlie_wealth_after_W₂ : after_W₂_firing Person.C = 2 := by rfl
theorem elise_wealth_after_W₂ : after_W₂_firing Person.E = -1 := by rfl

-- Test set firing W₃ = {B,C}
def W₃ : Finset Person := {Person.B, Person.C}
def after_W₃_firing := set_firing example_graph after_W₂_firing W₃
theorem alice_wealth_after_W₃ : after_W₃_firing Person.A = 2 := by rfl
```

```
theorem bob_wealth_after_W₃ : after_W₃_firing Person.B = 0 := by rfl
theorem charlie_wealth_after_W₃ : after_W₃_firing Person.C = 0 := by rfl
theorem elise_wealth_after_W₃ : after_W₃_firing Person.E = 0 := by rfl

-- Test borrowing moves
def after_bob_borrows := borrowing_move example_graph initial_wealth Person.B
theorem bob_wealth_after_borrowing : after_bob_borrows Person.B = -1 := by rfl
theorem alice_wealth_after_bob_borrows : after_bob_borrows Person.A = 1 := by rfl
theorem charlie_wealth_after_bob_borrows : after_bob_borrows Person.C = 3 := by rfl

-- Test degree of divisors
theorem initial_wealth_degree : deg initial_wealth = 2 := by rfl
theorem after_W₁_degree : deg after_W₁_firing = 2 := by rfl
theorem after_W₂_degree : deg after_W₂_firing = 2 := by rfl
theorem after_W₃_degree : deg after_W₃_firing = 2 := by rfl
```

Each theorem clearly corresponds to specific transformations observed in the illustrative figure. The definitions of firing sets W_1, W_2, and W_3 match their roles in the figure, confirming each vertex's wealth distribution precisely after each firing action. Additionally, the code rigorously checks properties such as individual vertex degrees and the effectiveness of divisors—ensuring that wealth conditions precisely align with our theoretical expectations.

Proposition 2.1.5. *Borrowing from a vertex $v \in V$ is just like lending from all vertices in $V \setminus v$, and if we perform set-lending from all vertices in V, the net effect is zero.*

Proof. First, let us consider the case of firing from all vertices $v \in V$. From definition 2.1.3 of individual firing moves, we have:

$$D' = D - \sum_{u \in V} \left[\sum_{uw \in E} (u - w) \right]$$

$$= D - \sum_{u \in V} \left[\text{val}(u)u - \sum_{uw \in E} w \right]$$

$$= D - \sum_{u \in V} \text{val}(u)u + \sum_{u \in V} \left[\sum_{uw \in E} w \right]$$

For the last step, we need to recognize that the sum $\sum_{u \in V} \left[\sum_{uw \in E} w \right]$ can be regrouped. Each edge uw appears exactly once in this sum when we enumerate from vertex u. However, each vertex w appears in this sum exactly $\text{val}(w)$ times – once for each neighbor. Therefore:

$$\sum_{u \in V} \left[\sum_{uw \in E} w \right] = \sum_{w \in V} \text{val}(w)w$$

Substituting this back:

$$D' = D - \sum_{u \in V} \text{val}(u)u + \sum_{w \in V} \text{val}(w)w = D$$

Since the vertices u and w both range over the same set V, the two sums cancel out, giving us $D' = D$. This shows that performing set-lending from all vertices in V leads to a zero net effect. We have thus proven that $\sum_{u \in V} \left[\sum_{uw \in E}(u - w)\right] = 0$.

Now, for the other part, a set-firing $D \xrightarrow{V \setminus \{v\}} D'$ can be represented as follows by definition 2.1.3 after breaking it down to each individual firing/lending move.

$$D' = D - \sum_{u \in V \setminus \{v\}} \left[\sum_{uw \in E} (u - w)\right]$$

$$= D - \sum_{u \in V} \left[\sum_{uw \in E} (u - w)\right] + \sum_{vw \in E} (v - w)$$

Now, using the above-mentioned (proven) result that $\sum_{u \in V} \left[\sum_{uw \in E}(u - w)\right] = 0$, we can say that:

$$D' = D + \sum_{vw \in E} (v - w).$$

This proves that borrowing from a vertex $v \in V$ has the same effect as a *set-lending* from all vertices in the set $V \setminus \{v\}$. □

Definition 2.1.6. A divisor D is said to be **linearly equivalent** to another divisor D' (denoted $D \sim D'$) if we can obtain D' from D by a sequence of lending moves.

We can formalize the linear equivalence of divisors in Lean4 as follows:

```
-- Define the group structure on CFDiv V
instance : AddGroup (CFDiv V) := Pi.addGroup

-- Define the firing vector for a single vertex
def firing_vector (G : CFGraph V) (v : V) : CFDiv V :=
  λ w => if w = v then -vertex_degree G v else num_edges G v w

-- Define the principal divisors generated by firing moves at vertices
def principal_divisors (G : CFGraph V) : AddSubgroup (CFDiv V) :=
  AddSubgroup.closure (Set.range (firing_vector G))

-- Define linear equivalence of divisors
def linear_equiv (G : CFGraph V) (D D' : CFDiv V) : Prop :=
  D' - D ∈ principal_divisors G
```

This Lean4 code introduces a couple of new ideas. The `instance: AddGroup (CFDiv V)` line tells Lean4 that divisors (functions from vertices to integers) can be added and subtracted like a mathematical group, using a built-in structure from Mathlib [10] called `Pi.addGroup`. This makes sense for the dollar game, where we combine wealth distributions. The `principal_divisors` definition uses `AddSubgroup.closure` to create a subgroup of divisors generated by firing moves—essentially, all possible outcomes of lending captured in one object, which we, in turn, use as a module to define linear equivalence (`linear_equiv`).

Proposition 2.1.7. *Linear equivalence is an equivalence relation on* $Div(G)$.

Proof: We proved this proposition in Lean4 by individually proving reflexivity, symmetry & transitivity pieces of the argument as follows:

```
-- [Proven] Proposition: Linear equivalence is an equivalence relation on Div(G)
theorem linear_equiv_is_equivalence (G : CFGraph V) : Equivalence (linear_equiv G) :=
    by
  apply Equivalence.mk
  -- Reflexivity
  · intro D
    unfold linear_equiv
    have h_zero : D - D = 0 := by simp
    rw [h_zero]
    exact AddSubgroup.zero_mem _

  -- Symmetry
  · intros D D' h
    unfold linear_equiv at *
    have h_symm : D - D' = -(D' - D) := by simp [sub_eq_add_neg, neg_sub]
    rw [h_symm]
    exact AddSubgroup.neg_mem _ h

  -- Transitivity
  · intros D D' D'' h1 h2
    unfold linear_equiv at *
    have h_trans : D'' - D = (D'' - D') + (D' - D) := by simp
    rw [h_trans]
    exact AddSubgroup.add_mem _ h2 h1
```

□

In this proof, Lean4's `theorem` syntax shines. The `apply Equivalence.mk` command sets up the three properties of an equivalence relation—reflexivity, symmetry, and transitivity—which we prove separately in blocks. Interestingly, for sub-modularization of proofs, have introduces intermediate steps that Lean4 checks, making the proof rigorous and readable. The tactic `intro` introduces a general element or assumption into the current proof context, allowing subsequent statements to directly reference and reason about it. The tactic `intros` generalizes `intro`, allowing the introduction of multiple assumptions or variables simultaneously. The keyword `unfold` explicitly expands the definition of `linear_equiv` here, for instance. The command `rw []` ("rewrite") explicitly applies previously proven equations or known definitions to transform expressions, guiding Lean4 to logically replace terms or sub-expressions to simplify or restructure the current proof state. Moreover, as mentioned before, the `simp` tactic simplifies expressions automatically (e.g., $D - D = 0$). Finally, the `exact` tactic concludes a proof by providing a precise, already established statement or lemma matching the goal exactly.

Definition 2.1.8. The **divisor class** determined by $D \in \mathrm{Div}(G)$ is:

$$[D] = \{D' \in \mathrm{Div}(G) \colon D' \sim D\}.$$

One can think of a divisor class as a self-contained economy where the total wealth does not change, but the distribution of wealth might shift around. In simpler terms, it represents all possible money distributions that can be achieved through lending.

Definition 2.1.9. A divisor D is **effective** if $D(v) \geq 0$ for all $v \in V$, meaning no one is in debt. The set of effective divisors on G is denoted by $\mathrm{Div}_+(G)$. [2] We write this as $D \geq 0$.

Thus, we can see that using the above definitions, the **objective** of the dollar game becomes: *Is a given divisor linearly equivalent to an effective divisor?*

Definition 2.1.10. A divisor D is **winnable** if D is linearly equivalent to an effective divisor. Otherwise, it is *unwinnable*.

Definition 2.1.11. A *complete linear system* of $D \in \mathrm{Div}(G)$ is:

$$|D| = \{E \in \mathrm{Div}(G) \colon E \sim D, E \geq 0\}.$$

Equivalently, a *complete linear system* is the set of all possible effective divisors on graph G that are linearly equivalent to D. We formalized the above definitions of divisor class, effective divisor, and winnability in Lean4 as follows:

[2] Since the set $\mathrm{Div}_+(G)$ does not have inverses, it is NOT a subgroup of $\mathrm{Div}(G)$, but rather a *commutative monoid*.

```
-- Define divisor class determined by a divisor
def divisor_class (G : CFGraph V) (D : CFDiv V) : Set (CFDiv V) :=
  {D' : CFDiv V | linear_equiv G D D'}

-- Define effective divisors (in terms of non-negativity, returning a Bool)
def effective_bool (D : CFDiv V) : Bool :=
  ↑((Finset.univ.filter (fun v => D v < 0)).card = 0)

-- Define effective divisors (in terms of equivalent Prop)
def effective (D : CFDiv V) : Prop :=
  ∀ v : V, D v ≥ 0

-- Define the set of effective divisors
-- Note: We use the Prop returned by `effective` in set comprehension
def Div_plus (G : CFGraph V) : Set (CFDiv V) :=
  {D : CFDiv V | effective D}

-- Define winnable divisor
-- Note: We use the Prop returned by `linear_equiv` in set comprehension
def winnable (G : CFGraph V) (D : CFDiv V) : Prop :=
  ∃ D' ∈ Div_plus G, linear_equiv G D D'

-- Define the complete linear system of divisors
def complete_linear_system (G: CFGraph V) (D: CFDiv V) : Set (CFDiv V) :=
  {D' : CFDiv V | linear_equiv G D D' ∧ effective D'}
```

The `Set (CFDiv V)` type represents a collection of divisors, defined using curly braces and a condition (e.g., `linear_equiv G D D'`). The `effective_bool` function uses `Finset.univ.filter` to check if any vertex has negative wealth—returning `true` if none do—while `effective` defines the same idea as a mathematical property (or `Prop`) that can be proven.

2.2 Laplacian & Firing Script

Laplacian aims to measure the "equitability" or evenness of a diffusive process in pure sciences. Similar to the continuous case, we define a discrete analog of Laplacian by drawing some parallels. Before we start defining Laplacian, let us generalize our notion of *set-firing* of V from the previous section into a **firing-script**, where we plan to compactly encode the essential information for the move in which some vertices in lending subset V lend/borrow multiple times, for instance.

Definition 2.2.1. A *firing script* is a function $\sigma\colon V \to \mathbb{Z}$, which denotes the number of times each vertex v lends (fires) if $\sigma(v) > 0$. If $\sigma(v) < 0$, it denotes the number of borrowing moves.

Moreover, if $\sigma(v) = 0$, the vertex v does not participate in the move. [3]

Furthermore, a *discrete Laplacian operator* is used to map a firing script to a divisor, and we define it as follows:

Definition 2.2.2. The *discrete Laplacian operator* on G is the linear mapping $L\colon \mathbb{Z}^V \to \mathbb{Z}^V$ defined by

$$L(f)(v) := \sum_{vw \in E} (f(v) - f(w)),$$

where the space $\mathbb{Z}^V := \{f\colon V \to \mathbb{Z}\}$ contains $\mathbb{Z}$-valued functions on the vertices of G.

In the context of chip firing games, we can think of the discrete Laplacian as a tool that maps a firing script in $M(G)$ to a resulting divisor in $\mathrm{Div}(G)$. If $\sigma\colon V \to \mathbb{Z}$ is a firing script, then the resulting divisor after firing is given by:

$$D' = D - \sum_{v \in V} \sigma(v) \left(\mathrm{val}(v)v - \sum_{wv \in E} w \right) = D - \sum_{v \in V} \sigma(v) \sum_{vw \in E} (v - w)$$

$$= D - \sum_{v \in V} \left(\mathrm{val}(v)\sigma(v) - \sum_{vw \in E} \sigma(w) \right) v = D - \sum_{v \in V} \sum_{vw \in E} (\sigma(v) - \sigma(w))v$$

Thus, any divisor can & should be reached by a firing script from any other divisor in the linearly equivalent set.

Definition 2.2.3. The *script-firing with firing script* σ is denoted by $D \xrightarrow{\sigma} D'$, and because degree is preserved under firing moves, we also have $\deg(L(\sigma)) = 0$.

We formalize the above definitions in Lean4 as follows:

```
-- Degree of a divisor
def deg (D : CFDiv V) : ℤ := Σ v, D v

-- Define a firing script as a function from vertices to integers
def firing_script (V : Type) := V → ℤ
```

In order to generalize this further and encode all the discrete Laplacians into one custom object, which we can use for building further sophisticated tools, we define the Laplacian matrix as follows:

Definition 2.2.4. The *Laplacian matrix*, denoted by $L\colon \mathbb{Z}^{|V|} \to \mathbb{Z}^V$, is an $|V| \times |V|$ integer matrix with ij entry given by:

---

[3] **Aside:** The collection of all firing scripts form an abelian group $\mathcal{M}(G)$ or $\mathbb{Z}^V$ [2, Definition 2.2].

$$L_{ij} = L(\chi_j)(v_i) = \begin{cases} \text{val}(v_i) & \text{if } i = j \\ -(\text{\# of edges between } v_j \text{ and } v_i) & \text{if } i \neq j. \end{cases}$$

Here $\chi_j(v_i) = \begin{cases} 1 & i = j \\ 0 & i \neq j. \end{cases}$ is the firing script of v_j making a single lending move (to v_i).

It is also evident that $L = \text{Deg}(G) - A^T$, where $\text{Deg}(G)$ is a diagonal matrix with vertex-degrees of G, and A is the adjacency matrix s.t. $A_{ij} = \#$ of edges between v_i and v_j.

Note: All lending moves are encoded in L because the lending move by v_j corresponds to subtracting the jth column of L from a divisor at hand. For instance, given a firing-script column vector $\vec{\sigma}$, we can say $D' = D - L\vec{\sigma}$.

We formalize the matrix form of Laplacian in Lean4 as follows:

```
-- Define Laplacian matrix as an |V| x |V| integer matrix
open Matrix
variable [Fintype V]

def laplacian_matrix (G : CFGraph V) : Matrix V V ℤ :=
  λ i j => if i = j then vertex_degree G i else - (num_edges G i j)

-- Apply the Laplacian matrix to a firing script, and current divisor to get a new
    divisor
def apply_laplacian (G : CFGraph V) (σ : firing_script V) (D: CFDiv V) : CFDiv V :=
  fun v => (D v) - (laplacian_matrix G).mulVec σ v
```

The `Matrix V V ℤ` type represents a square matrix with integer entries indexed by vertices. The `laplacian_matrix` definition uses a conditional expression to set diagonal entries to vertex degrees and off-diagonal entries to the negative number of edges, matching our mathematical definition. The `apply_laplacian` function then uses matrix multiplication (`mulVec`) to compute the new divisor after applying a firing script.

As an illustration, going back to the example we have discussed so far in Figure 2.2, we can see that the Laplacian matrix assumes the following form:

$$L = \begin{bmatrix} 4 & -1 & -1 & -2 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -2 & 0 & -1 & 3 \end{bmatrix}, \text{ which is filled in as } \begin{array}{c|cccc} & V_{Alice} & V_{Bob} & V_{Charlie} & V_{Elise} \\ \hline V_{Alice} & 4 & -1 & -1 & -2 \\ V_{Bob} & -1 & 2 & -1 & 0 \\ V_{Charlie} & -1 & -1 & 3 & -1 \\ V_{Elise} & -2 & 0 & -1 & 3 \end{array}$$

Moreover, from Figure 2.2, we can see that to win (reach an effective divisor), Bob borrowed twice, and then Bob and Charlie both can be considered to lend once. So, we can represent this in the form of a firing script (ordered column vector) as: $\vec{\sigma} = \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix}$. Thus,

$$D' = \begin{bmatrix} 2 \\ -3 \\ 4 \\ -1 \end{bmatrix} - \begin{bmatrix} 4 & -1 & -1 & -2 \\ -1 & 2 & -1 & 0 \\ -1 & -1 & 3 & -1 \\ -2 & 0 & -1 & 3 \end{bmatrix} \begin{bmatrix} 0 \\ -1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \\ -3 \\ 4 \\ -1 \end{bmatrix} - \begin{bmatrix} 0 \\ -3 \\ 4 \\ -1 \end{bmatrix} = \begin{bmatrix} 2 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

We can see that the final result D' matches the effective divisor we obtained in Figure 2.2 right when we hit the win condition. Let us formalize and verify this finding in Lean4 by continuing with the example we have been working on thus far.

```
-- Test Laplacian matrix values and symmetricity
def example_laplacian := laplacian_matrix example_graph
theorem laplacian_diagonal_A : example_laplacian Person.A Person.A = 4 := by rfl
theorem laplacian_diagonal_B : example_laplacian Person.B Person.B = 2 := by rfl
theorem laplacian_diagonal_C : example_laplacian Person.C Person.C = 3 := by rfl
theorem laplacian_diagonal_E : example_laplacian Person.E Person.E = 3 := by rfl
theorem laplacian_off_diagonal_AB : example_laplacian Person.A Person.B = -1 := by rfl
theorem laplacian_off_diagonal_AC : example_laplacian Person.A Person.C = -1 := by rfl
theorem laplacian_off_diagonal_AE : example_laplacian Person.A Person.E = -2 := by rfl
theorem laplacian_off_diagonal_BC : example_laplacian Person.B Person.C = -1 := by rfl
theorem laplacian_off_diagonal_BE : example_laplacian Person.B Person.E = 0 := by rfl
theorem laplacian_off_diagonal_CE : example_laplacian Person.C Person.E = -1 := by rfl
theorem check_example_laplacian_symmetry : Matrix.IsSymm example_laplacian := by {
  apply Matrix.IsSymm.ext
  intros i j
  cases i <;> cases j
  all_goals {
    rfl
  }
}

-- Test script firing through laplacians
def firing_script_example : firing_script Person := fun v => match v with
  | Person.A => 0
  | Person.B => -1
  | Person.C => 1
  | Person.E => 0
def res_div_post_lap_based_script_firing := apply_laplacian example_graph
    firing_script_example initial_wealth
theorem lap_based_script_firing_preserves_degree : deg
    res_div_post_lap_based_script_firing = 2 := by rfl
```

In this Lean4 snippet, we test specific entries of the Laplacian matrix (e.g., Alice's degree is 4, and there are two edges to Elise, so -2). The theorem check_example_laplacian_symmetry uses the notion of symmetry (pre-established in Mathlib [10] 's Matrix module) to check if the constructed Laplacian matrix from our example graph is symmetric by breaking down the goal into cases for all the pairs and then compiling them one-by-one through `rfl` tactic. Finally, the `firing_script_example` encodes our strategy—Bob borrows once (`-1`), Charlie lends once (`1`)—and `apply_laplacian` computes the result. The degree preservation theorem confirms that the total wealth stays at 2, a key property of firing moves. Fully tested and integrated Lean4 code for the relevant definitions presented above, along with example graph usage we covered in this chapter, can be found in Appendix A, sections A.1 and A.2.

Chapter 3

Algorithms for Winnability

Since we have defined the complete setup of the dollar game and our objective (winnability), we will dive into some possible algorithms for winnability determination in this chapter.

3.1 Greedy Algorithm

One way to play the dollar game is for each in-debt vertex to attempt to borrow its way out of debt. The problem is that borrowing once from each vertex is the same as not borrowing at all. Solving this gives an algorithm for the dollar game, which is essentially repeatedly choosing an in-debt vertex and making a borrowing move at that vertex until either the game is won or it becomes impossible to go on without reaching a state in which all of the vertices have made borrowing moves. Below, we first present an adaptation of the algorithm's pseudocode from [2, §3.1].

Note on Uniqueness of the greedy algorithm script: The greedy algorithm [1] can be modified to produce a firing script if its input is winnable. Initialize by setting $\sigma = 0$, and then each time step 6 of the algorithm below is invoked, replace σ by $\sigma - v$. It turns out that the resulting script is independent of the order in which vertices are added.

Proposition 3.1.1. *Suppose D is winnable, and let σ_1 and σ_2 be firing scripts produced by applying the greedy algorithm to D, so that firing these scripts from D produces effective divisors E_1 and E_2 respectively. Then $\sigma_1 = \sigma_2$, so that $E_1 = E_2$ as well.*

Proof. This can be proved by supposing a contradiction as presented in Appendix B's section B.2. □

[1] Proof of the validity of the greedy algorithm can be found in Appendix B's section B.1.

Algorithm 1 Greedy algorithm for the dollar game.

Require: $D \in \mathrm{Div}(G)$.

Ensure: TRUE if D is winnable; FALSE if not.

```
initialization: M = ∅ ⊆ V, the set of marked vertices.
while D not effective do
  if M ≠ V then
    choose any vertex in debt: v ∈ V such that D(v) < 0
    modify D by performing a borrowing move at v
    if v is not in M then
      add v to M
    end if
  else
    /* required to borrow from all vertices */
    return FALSE                                  /* unwinnable */
  end if
end while
return TRUE                                       /* winnable */
```

We now implement the greedy algorithm with all Python code presented in Appendix A.13 along with the outlined file structure. Upon running our implementation on the example from Figure 2.1, we can see that it outputs the following:

```
The game is winnable with the greedy algorithm.
Firing Script: {'A': -1, 'B': -2, 'C': 0, 'E': -1}
Resulting Divisor: {'A': 2, 'B': 0, 'C': 0, 'E': 0}
```

We can see that this resulting divisor and winnability conclusion agrees with our manual walk-through in Figure 2.2 we reached the same resulting effective divisor. We are currently working on a Python package at `https://pypi.org/project/chipfiring/` to aid researchers and educators in running chip-firing simulations correctly and efficiently.

3.2 "Benevolence" Algorithm

3.2.1 Preliminaries

Now that we have successfully defined the game and its rules, we will define more mathematical tools to help us define our objective function of **winnability**.

Definition 3.2.1. Let $q \in V$. A divisor $D \in Div(G)$ is called q**-reduced** if the following conditions hold:

1. $D(v) \geq 0$ for all $v \in V \setminus \{q\}$.

2. For every nonempty subset $S \subseteq V \setminus \{q\}$, there exists a vertex $v \in S$ such that $D(v) < \mathrm{outdeg}_S(v)$, where $\mathrm{outdeg}_S(v)$ denotes the number of edges vw such that $w \notin S$.

Definition 3.2.2. Given divisors $D, D' \in \mathrm{Div}(G)$ and a spanning tree (T, q) of G rooted at a vertex q, let $v_1 = q, v_2, \ldots, v_n$ be a tree ordering of the vertices, where:

- T is a connected, cycle-free subgraph of G that includes all vertices and contains exactly $n-1$ edges (i.e., a spanning tree),
- the ordering respects the structure of T, meaning that if v_i lies on the unique path from q to v_j in T, then $i < j$.

We say that $D' \prec D$ if either:

1. $\deg(D') < \deg(D)$, or
2. $\deg(D') = \deg(D)$ and there exists an index i such that $D'(v_i) > D(v_i)$, and for all $j < i$, $D'(v_j) = D(v_j)$. Equivalently, i is the smallest index for which $D'(v_i) > D(v_i)$.

Definition 3.2.3. Let $D \in \mathrm{Div}(G)$, and let $S \subseteq V$. Suppose D' is obtained from D by firing each of the vertices in S once. Then $D \xrightarrow{S} D'$ is a **legal set-firing** if $D'(v) \geq 0$ for all $v \in S$, i.e., after firing S, none of the vertices in S are in debt. In this case, we say it is **legal to fire** S. [Note: if it is legal to fire S, then the vertices in S must also be out of debt *before* firing.]

The Lean4 formalization of these definitions can be found in Appendix A.1.

Since we have defined q-reduced divisors, we would like to order them with respect to winnability. An essential property of this ordering should be that when a set of vertices other than q fires, some dollars move towards q, producing a $D' \prec D$. To make this idea more precise, we can see that our above definitions give us this in case of set-firing $D \xrightarrow{S \subseteq \widetilde{V}} D'$, we have $D' \prec D$. This is because in the case of a tree ordering of the spanning tree (T, q) where $v_1 = q$. Let v_j be the vertex with the smallest index such that v_j is incident to an edge connected to S. Upon firing S, the degree of D remains unchanged, and $D'(v_i) = D(v_i)$ for all $i = 2, \ldots, j-1$, but $D'(v_j) > D(v_j)$. Therefore, by definition, $D' \prec D$.

Furthermore, there exists a unique q-reduced divisor D_q linearly equivalent to D [2, Theorem 3.6], which leads to the following proposition:

Proposition 3.2.4. *Let $D \in \mathrm{Div}(G)$, and let D' be the q-reduced divisor linearly equivalent to D. Then $|D| \neq \emptyset$ if and only if $D' \geq 0$. In other words, D is winnable if and only if $D'(q) \geq 0$.*

Proof. Suppose D is winnable. Then, we know from definition 2.1.10 that an effective divisor $E \in |D|$. Say we perform all legal set-firings for vertices other than q from E as defined in definition 3.2.3, arriving at q-reduced divisor $E' \geq 0$. By uniqueness of q-reduced divisors, we have $D' = E'$. Moreover, by definition 2.1.9 we get $D' \geq 0$. The converse is immediate because if $D' \geq 0$, there is a winnable divisor, which means $|D|$ is non-empty. We present this version and alternative proof on lines 39-64 and 66-78, respectively, within Appendix A.7. □

3.2.2 Implementation & Setup

One might think that "benevolence", in the form of debt-free vertices lending to their in-debt neighbors, might also work, but it does not, in general [2]; we present a particular version of benevolence that does solve the dollar game and which will have theoretical significance for us later on.

Starting with $D \in \mathrm{Div}(G)$, we would like to find an effective divisor $E \sim D$. This can be done by following the steps below: [2]

1. Pick some "benevolent vertex" $q \in V$. Call q the source vertex, and let $V \setminus \{q\}$ be the set of non-source vertices.

2. Let q lend/fire so many chips that the non-source vertices, sharing among themselves, are out of debt. This is done to concentrate the debt at the source vertex.

3. At this stage, only q is in debt, and it makes no further lending or borrowing moves. It is now the job of the non-source vertices to try to relieve q of its debt. Look for $S \subseteq V \setminus \{q\}$ with the legal set-firing property as stated in definition 3.2.3. Having found such an S, make the corresponding set-lending move.

4. Repeat until no such S remains. The resulting divisor is said to be q-reduced. More importantly, if, in the end, q is no longer in debt, D is winnable. Otherwise, $|D| = \emptyset$, or equivalently D is unwinnable.

As mentioned by Corry and Perkinson [2], some naturally interesting questions about this strategy are: Is it always possible to complete step 2? Is step 3 guaranteed to terminate? If the strategy does not win, does this mean the game is unwinnable? (After all, the moves in step 3 are constrained.) Is the resulting q-reduced divisor unique? Can the strategy be efficiently implemented?

The main goal of the following sections is to gradually show that the answer to all of these questions is "yes".

3.3 Configuration & Related Preliminaries

Definition 3.3.1. Fix a vertex $q \in V$ and define $\widetilde{V} := V \setminus \{q\}$. Then a *configuration* c is an element of the subgroup

$$\text{Config}(G, q) = \mathbb{Z}\widetilde{V} \subseteq \mathbb{Z}V = \text{Div}(G).$$

A configuration is a divisor that omits a specific vertex q. Thus, we also write $c' \geq c$ for $c, c' \in \text{Config}(G)$ if $c(v) \geq c'(v)$ for all $v \in \widetilde{V}$.

Note: c is said to be *non-negative* ($c \geq 0$) is $c(v) \geq 0$ for all $v \in \widetilde{V}$. Furthermore, lending & borrowing operations can still occur at q like with divisors, but in the case of configurations by definition, we do not keep track of the number of chips present at q.

Definition 3.3.2. The *degree* of a configuration c is calculated as $\deg(c) = \sum_{v \in \widetilde{V}} c(v)$.

Definition 3.3.3. Configurations c and c' are said to be *linearly equivalent*, written as $c \sim c'$ if they can be transformed into one another through a sequence of lending and borrowing operations.

Note: Unlike divisors, linearly equivalent configurations are not necessarily required to have the same degree because, in the case of configurations, we are not keeping track of the number of chips at q, so through a given transformation, some chips might exit the configuration by definition.

For instance, let us consider a slight relabeling of the example in Figure 2.1, where Bob is labeled as q, and thus we consider configurations with respect to Bob. We can see configurations $c \sim c'$ as depicted in Figure 3.1.

Definition 3.3.4. Let $c \in \text{Config}(G)$, and let $S \subseteq \widetilde{V}$. Suppose c' is the configuration obtained from c by firing the vertices in S. Then $c \xrightarrow{S} c'$ is a **legal set-firing** if $c'(v) \geq 0$ for all $v \in S$.

Definition 3.3.5. The configuration $c \in \text{Config}(G)$ is **superstable** if $c \geq 0$ and has no legal nonempty set-firings. Equivalently, for all nonempty $S \subseteq \widetilde{V}$, there exists $v \in S$ such that $c(v) <$ outdeg$_S(v)$., which in case of firing on S leads to $c'(v) < 0$.

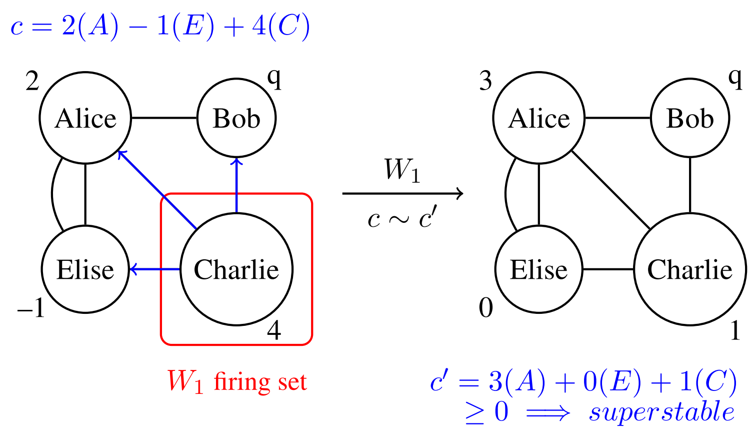

Figure 3.1: Application of firing move by Charlie, for instance, in case of configurations w.r.t $q = Bob$ on the same graph as Figure 2.1

We formalize these definitions in Lean4 within appendix A.3.

3.4 Dhar's Algorithm

With the above definitions in mind, one might wonder how we use these tools to quickly find non-empty legal set-firing for configuration c (if it exists) instead of having to search through the entire parameter space of $2^{|\widetilde{V}|} - 1$ plausible subsets. Dhar's algorithm, as mentioned in Corry and Perkinson [2], helps us answer just that exact question, and it can be described as follows:

Let $c \geq 0$ such that $c \in \text{Config}(G, q)$. If this is false, we fail early since we need this for the superstablity as in definition 3.3.5. Now, to find a legal set-firing for c, if it exists, imagine the edges of our graph are made of wood so that when vertex q is ignited, the fire spreads along its incident edges. Furthermore, think of the configuration c as $c(v)$ firefighters present at each $v \in \widetilde{V}$, given that each firefighter can only control the fire coming from a single edge. This tells us that a vertex is protected only if the number of burning incident edges is $\leq c(v)$. Otherwise, the firefighters fail, and the vertex is set on fire.[2] In the end, we conclude that the unburnt vertices constitute a set that may be legally fired from c and that if this set is empty, then by definition 3.3.5 c is superstable.

Below, we present an adaptation of the pseudocode version of the algorithm [3] from [2, §3.4.1]:

Let us run through Dhar's algorithm described above in Figure 3.2 while continuing from the

[2]Fun Note: No need to worry; firefighters are rescued by an underground tunnel built by Amherst College. [2]

[3]Proof of validity of this algorithm can be found in Appendix B's section B.3.

Algorithm 2 Dhar's algorithm.

Require: a nonnegative configuration c
Ensure: a legal firing set $S \subseteq \widetilde{V}$, empty iff c is superstable

initialization: $S = \widetilde{V}$
while $S \neq \emptyset$ **do**
  if $c(v) < \text{outdeg}_S(v)$ for some $v \in S$ **then**
    $S = S \setminus \{v\}$
  else
    return S /* c is not superstable */
  end if
end while
return S

starting linearly equivalent configuration (c'), which we obtained in Figure 3.1. After the vertex q is set on fire, it sets two edges on fire (one between Alice and Bob and one between Bob and Charlie). However, since there are $3 \geq 1$ firemen at Alice's vertex, and there are $1 \geq 1$ firefighters at Charlie's vertex, the algorithm terminates and outputs the legal set firing as the set $S = \{Alice, Elise, Charlie\}$ as expected.

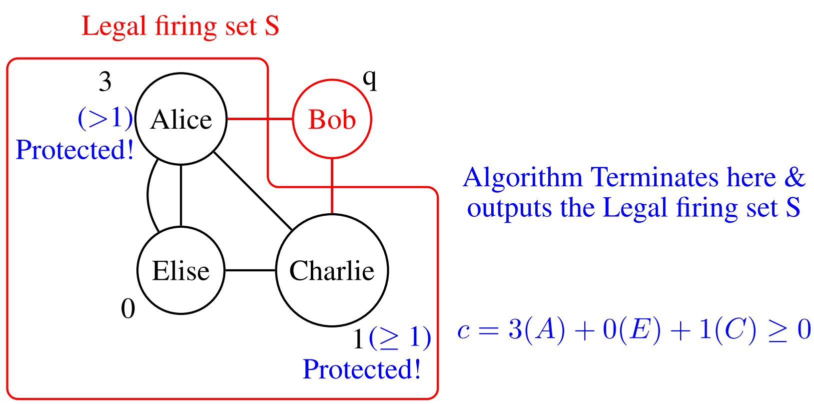

Figure 3.2: Application of Dhar's Algorithm on the same graph as Figure 3.1

Since we have Dhar's algorithm as a tool, let us revisit the procedure of finding q-reduced divisors (or "benevolence" algorithm) as described in section 3.2. Below, we present an adaptation of the pseudocode version of the updated algorithm from [2, §3.4]. This algorithm has some additional peculiar optimizations, which can help decrease the runtime [2]. We present an adaptation of the same in Appendix B's section B.4.

Algorithm 3 Find the linearly equivalent q-reduced divisor.

Require: $D \in \mathrm{Div}(G)$ and $q \in V$

Ensure: the unique q-reduced divisor linearly equivalent to D

Use a greedy algorithm to bring each vertex $v \neq q$ out of debt, so we may assume $D(v) \geq 0$ for all $v \neq q$.

Repeatedly apply Dhar's algorithm until D is q-reduced.

3.5 An Efficient Winnability Determination Algorithm

Now, using all the tools & algorithms we have developed so far, let's devise an efficient winnability determination algorithm. We formalize the algorithm in the form of pseudocode below.

It is important to note that $D_q(q) \geq 0$ directly translates to **winnability** by direct implication from proposition 3.2.4.

Algorithm 4 Efficient Winnability Determination Algorithm

Require: $D \in \mathrm{Div}(G)$

Ensure: TRUE if D is winnable; FALSE if not

1: Choose source vertex $q \in V$
2: Let $\widetilde{V} \leftarrow V \setminus \{q\}$ /* Set of non-source vertices */
3: Fire from q and share among vertices in $\widetilde{V}$ until only q is in debt
4: **while** Dhar's Algorithm returns a non-empty set **do**
5: Apply Dhar's Algorithm to current configuration c
6: Fire the returned set if non-empty
7: **end while**
8: $D_q(q) \leftarrow \deg(D) - \deg(c)$ /* Calculate chips on source vertex */
9: **if** $D_q(q) \geq 0$ **then**
10: **return** TRUE /* winnable */
11: **else**
12: **return** FALSE /* unwinnable */
13: **end if**

Furthermore, one of the strategies that can be used in step 3 of the algorithm below is to systematically concentrate all debt at our distinguished vertex q through a reverse-distance prioritized approach. The key insight is that we can move debt away from vertices furthest from q first, working our way inward toward q systematically.

This strategy works by ordering all non-source vertices based on their distance from q, which we can determine through a simple *breadth-first search (BFS) traversal*. Then, proceeding from the vertices furthest from q and moving inward, we perform borrowing operations on any vertex with a negative chip count (in debt). When a vertex borrows, it receives chips equal to its degree but must distribute one chip along each incident edge to its neighbors. This borrowing operation effectively pushes debt closer to q.

By processing vertices in reverse distance order (from furthest to closest to q), we ensure that once a vertex is out of debt, it remains out of debt throughout the process. This is because we only process vertices closer to q after all vertices further from q have been handled. The result is a configuration where only the source vertex q may be in debt, with all other vertices having non-negative chip counts. In practice, the number of Dhar iterations is typically small. This makes the algorithm more efficient than exhaustive approaches that might simulate all possible chip-firing sequences.

As an illustration, we implemented this BFS-based debt clustering strategy along with Dhar's algorithm in Python code, which can be referenced in Appendix A.13's section A.13.3. Upon running our implementation on the example from Figure 3.1, we can see that it outputs the following:

```
The game is winnable using Dhar's algorithm.
Legal firing set: {'A', 'C', 'E'}
```

Chapter 4

Riemann-Roch for Graphs & its Formalization

In this chapter, we will build on some final tools, mainly including orientations and ranks, which will, in turn, help us define and better understand the Riemann Roch theorem for graphs.

Definition 4.0.1. *Maximal unwinnable divisors* are those unwinnable divisors D such that for any unwinnable divisor D', if $D \leq D'$, then $D = D'$. Equivalently, D is maximal unwinnable if D is unwinnable, but $D + v$ is winnable for each $v \in V$.

Definition 4.0.2. *Maximal superstable configurations* are those superstable configurations c such that for any superstable configuration c', if $c \leq c'$, then $c = c'$.

4.1 Orientations and Genus

Definition 4.1.1. A graph orientation $\mathcal{O}$ assigns a specific direction to each edge of the graph within the multiset of edges. For any edge $e = uv \in E$, we define $e^- = u$ as the starting vertex (tail) and $e^+ = v$ as the ending vertex (head), meaning that the edge e is directed from u to v.

Definition 4.1.2. The *reverse orientation* of $\mathcal{O}$, denoted by $\overline{\mathcal{O}}$, swaps the roles of the head and the tail of each edge. For instance, e under $\overline{\mathcal{O}}$ would become $\overline{e}$ with $\overline{e}^- = v$ as the starting vertex (tail) and $\overline{e}^+ = u$ as the ending vertex (head).

Note: A vertex w is called a *source* for an orientation $\mathcal{O}$ if all edges incident to w are directed away from w. Equivalently, if for all edges $e \in \mathcal{O}$, $e^+ \neq w$. Similarly, a vertex v is called a *sink* for an orientation $\mathcal{O}$ if all edges incident to v are directed towards v. Equivalently, for all edges $e \in \mathcal{O}$, $e^- \neq v$.

Definition 4.1.3. For any vertex $v \in V$ under an orientation $\mathcal{O}$, the *outdegree* counts the edges that start from v (i.e. $\operatorname{outdeg}_{\mathcal{O}}(v) = |\{e \in \mathcal{O} \colon e^- = v\}|$), while the *indegree* is the number of edges directed towards v (i.e. $\operatorname{indeg}_{\mathcal{O}}(v) = |\{e \in \mathcal{O} \colon e^+ = v\}|$).

Definition 4.1.4. A *divisor* associated with a given orientation $\mathcal{O}$ on the graph G is defined as:

$$D(\mathcal{O}) = \sum_{v \in V} (\operatorname{indeg}_{\mathcal{O}}(v) - 1) \cdot v.$$

Definition 4.1.5. The *configuration* associated to a source vertex $q \in V$ under $\mathcal{O}$ is defined as:

$$c(\mathcal{O}) = \sum_{v \in \widetilde{V}} (\operatorname{indeg}_{\mathcal{O}}(v) - 1) \cdot v.$$

Definition 4.1.6. The *canonical divisor* K of a graph G is defined as: $K := D(\mathcal{O}) + D(\overline{\mathcal{O}})$. The canonical divisor only depends on graph G and is independent of orientation because for any vertex $v \in V$, we have:$K(v) = (\operatorname{indeg}_{\mathcal{O}}(v) - 1) + (\operatorname{outdeg}_{\mathcal{O}}(v) - 1) = \operatorname{val}(v) - 2.$, and thus, the canonical divisor can also be written as: $K := \sum_{v \in V} (\operatorname{val}(v) - 2) \cdot v$.

Definition 4.1.7. A *directed path* is a sequence of vertices connected by edges, where each vertex (except the first and last) acts as both the head of the previous edge and the tail of the next one.

Note: All vertices along a directed path are distinct, except possibly the start & end vertices.

Definition 4.1.8. A *directed cycle* is a directed path in which the start and end vertices are identical.

Definition 4.1.9. An orientation $\mathcal{O}$ is *acyclic* if it does not contain any cycle of directed edges.

Note: In the case of acyclic orientations, multiple edges between two vertices must be oriented in the same direction.

We have formalized these definitions and properties in Lean4 within appendix A.4.

Lemma 4.1.10. *An acyclic orientation can be determined by its indegree sequence: if $\mathcal{O}$ and $\mathcal{O}'$ are acyclic orientations of G and $\forall v \in V$, $\operatorname{indeg}_{\mathcal{O}}(v) = \operatorname{indeg}_{\mathcal{O}'}(v)$, then $\mathcal{O} = \mathcal{O}'$.*

Proof. Given an acyclic orientation $\mathcal{O}$ on G, let $V_1 \subset V$ be its set of source vertices. First, we claim that V_1 is necessarily non-empty in this case. In order to prove this, we will suppose a contradiction that $V_1 = \emptyset$. This means that there are no source vertices for orientation $\mathcal{O} \iff$ there are no sink vertices for the corresponding reverse orientation $\overline{\mathcal{O}} \implies$ there exists at least one cycle in the orientation $\overline{\mathcal{O}} \implies$ there exists at least one cycle in the orientation $\mathcal{O}$ (opposite direction of the one we saw in $\overline{\mathcal{O}}$). This contradicts the fact that the orientation $\mathcal{O}$ is acyclic; hence, we have proved our claim that V_1 is necessarily non-empty.

Now, there are finite source vertices (i.e. vertices v with $\mathrm{indeg}_{\mathcal{O}}(v) = 0$). Let us remove these vertices along with their incident edges from G and $\mathcal{O}$ to obtain an acyclic orientation $\mathcal{O}_1$ and a subgraph G_1. Repeating this until we run out of vertices in V (i.e. until G is empty) will leave us with a sequence $(V_1, V_2, \cdots)$ partitioning V. We can see that indegree sequence determines the sequence $(V_1, V_2, \cdots)$, which determines $\mathcal{O}$. Hence, an acyclic orientation can be determined by its indegree sequence. □

Let us now revisit Dhar's algorithm as we introduced in section 3.4 by incorporating our new viewpoint of orientations. We include the pseudocode for modified Dhar's algorithm below with a vital change that whenever a fire spreads from vertex u to v, we "record" this piece of information by placing a direction $u \to v$ on edge uv in the orientation.

Algorithm 5 Orientation-based Dhar's Algorithm

Require: a nonnegative configuration c and source vertex q
Ensure: a pair $(S, \mathcal{O})$ where $S \subseteq \widetilde{V}$ is a legal firing set (empty if and only if c is superstable) and $\mathcal{O}$ is the resulting orientation

initialization: $S \leftarrow \widetilde{V}$, $\mathcal{O} \leftarrow \emptyset$, $B \leftarrow \{q\}$ /* B is burning set */
while $B \neq V$ **do**
  for each $v \in S$ **do**
    $E_v \leftarrow$ edges between v and B /* potentially burning edges */
    if $|E_v| > c(v)$ **then**
      $S \leftarrow S \setminus \{v\}$
      $B \leftarrow B \cup \{v\}$
      for each $e \in E_v$ **do**
        Orient e towards v in $\mathcal{O}$ /* record burning direction */
      end for
    end if
  end for
  if no new vertices added to B **then**
    return $(S, \mathcal{O})$ /* c is not superstable */
  end if
end while
return $(\emptyset, \mathcal{O})$ /* c is superstable */

Proposition 4.1.11. *Fix $q \in V$. Then, the correspondence $\mathcal{O} \mapsto c(\mathcal{O})$. is a bijection between acyclic orientations $\mathcal{O}$ of G (with unique source q) and maximal superstable configurations $c(\mathcal{O}) \in$* $\mathrm{Config}(G, q)$.

Proof. In order to prove that the given map is a bijection, we need to show that the map is both one-one and onto.

To show that the given map is one-one, it suffices to show the following: Given acyclic orientations $\mathcal{O}$ and $\mathcal{O}'$, if $c(\mathcal{O}) = c(\mathcal{O}')$, then $\mathcal{O} = \mathcal{O}'$. Since we are given a fixed source vertex q, which is used for the determination of configurations for all the orientations in consideration, we have $\mathrm{indeg}_{\mathcal{O}}(q) = \mathrm{indeg}_{\mathcal{O}'}(q)$. So, we also have $\forall v \in V$, $\mathrm{indeg}_{\mathcal{O}}(v) = \mathrm{indeg}_{\mathcal{O}'}(v)$. Now, we satisfy all the hypotheses put forth by Lemma 4.1.10, and hence we can conclude that $\mathcal{O} = \mathcal{O}'$. This completes the proof for the given map being one-one.

To show that the given map is onto, it suffices to show the following: given a maximal superstable configuration $c \in \mathrm{Config}(G, q)$, there exists a corresponding acyclic orientation $\mathcal{O}$ of G (with unique source vertex q). Suppose we apply Dhar's modified algorithm for acyclic orientation tracking starting at the source vertex q. In that case, we are guaranteed to see by definition $S = \emptyset$ upon termination since we are given that c is superstable. Furthermore, we would also have a valid orientation $\mathcal{O}$ as a byproduct. The guarantees of uniqueness of q and acyclicity of $\mathcal{O}$ as hinted by Corry and Perkinson [2] comes from the fact that having another source vertex would make it unreachable by Dhar's algorithm, and hence will invalidate the superstability of c since $S = \{q\} \neq \emptyset$. Moreover, on the other hand, a potential directed cycle in $\mathcal{O}$ would lead to non-deterministic directional encodings within the orientation. Hence, the given map is onto.

We present a version of this proof in Lean4 on lines 98-173 within Appendix A.7. □

Definition 4.1.12. The *genus* of a graph is the quantity: $g = |E| - |V| + 1$.

Proposition 4.1.13. *Let c be a superstable configuration and D be a divisor. Then,*

1. *c is maximal if and only if* $\deg(c) = g$.
2. *D is maximal winnable if and only if its q-reduced form is c − q, with maximal superstable c.*

Proof. (1):

Given that c is superstable, and we know that there exists a maximal superstable $c(\mathcal{O})$ from Proposition 4.1.11, we get that $deg(c) \leq deg(c(\mathcal{O})) = \sum_{v \in \widetilde{V}}(\text{indeg}_{\mathcal{O}}(v) - 1) = \sum_{v \in \widetilde{V}} \text{indeg}_{\mathcal{O}}(v) - \sum_{v \in \widetilde{V}} 1 = |E| - (|V| - 1) = |E| - |V| + 1 = g$. The inequality turns into an equality $\iff c = c(\mathcal{O})$ by definition 4.0.2 of maximal superstable configuration.

(2):

($\implies$): By definition 3.3.1 of configurations, we can say that every divisor D can be represented as $c + kq$ where $c \in \text{Config}(G, q)$ and $k \in \mathbb{Z}$. Hence, D is q-reduced $\iff$ c is superstable. Finally, given that D is maximal unwinnable, its q-reduced form is $c - q$ because if q had a non-negative degree, then D would have been winnable. If q had a degree less than -1, then $D + q$ would be unwinnable. Hence, c must be maximal superstable.

($\impliedby$): Given that $D = c - q$ with c being maximal superstable. Then D is unwinnable since $D(q) < 0$. Let $v \in V$. If $v = q$, then $D + v$ is clearly winnable, otherwise when $v \neq q$, we have $D = (c + v) - q$. Since $(c + v) \in \text{Config}(G, q)$, in order to compute the q-reduced form of $D + v$, we need to superstabilize $c + v$. By part (1), we know that $deg(c + v) = g + 1$, and the degree of its superstabilization is at most g. Hence, at least one dollar is sent to q, showing that $D + v$ is winnable. This completes the proof that D is maximal unwinnable.

We present a version of this proof in Lean4 on lines 174-334 within Appendix A.7. □

Proposition 4.1.14. *Let D be a divisor on G.*

1. *The correspondence $\mathcal{O} \mapsto D(\mathcal{O})$. is a bijection between acyclic orientations of G with unique source q and maximal unwinnable q-reduced divisors of G.*

2. *If D is a maximal unwinnable divisor, then* $\deg(D) = g - 1$. *Thus,* $\deg(D) \geq g$ *implies D is winnable.*

Proof. (1): Using proposition 4.1.13 and proposition 4.1.11, we are done.

(2): Using proposition 4.1.13, we have that $D = c - q$ if D is maximal unwinnable, which is the case here, so this leads to $\deg(D) = \deg(c - q) = \deg(c) - 1 = g - 1$. Hence, we are done.

We present a version of this proof in Lean4 on lines 394-419 within Appendix A.7. □

We formalize these definitions of genus and a divisor being maximal unwinnable in Lean4 within appendix A.5.

4.2 Rank

One of our winnability questions pertained to *"Are some games more winnable than others?"* One way to answer this is by defining the *rank function*, which is central to proving the Riemann-Roch theorem for graphs and will lend us the ability to formalize further the answer to the question *"How many chips can be removed for the divisor to remain still winnable?"*

Definition 4.2.1. The *rank function* $r(D)\colon D \to \mathbb{Z}$ is defined as:

1. $r(D) = -1$ if and only if $|D| = \emptyset$.
2. $r(D) \geq k$ for $k \geq 0$ if and only if the dollar game is winnable starting from all divisors obtained from D by removing k dollars.
3. $r(D) = k$ if and only if $r(D) \geq k$ and there exists an effective divisor E such that $\deg(E) = k + 1$ and $D - E$ is unwinnable.

Continuing our remark from chapter 2, it is worth restating that the problem of computing the rank of a general divisor on a general graph is **NP**-hard [5], which means the time it takes for an algorithm to compute the rank is non-polynomial. Specifically, this time grows exponentially with the size of the graph [2, §5.1].

Corollary 4.2.2. *For divisors D, D' with $r(D), r(D') \geq 0$, we have $r(D + D') \geq r(D) + r(D')$.*

Proof. Let's say $r(D) \geq k_1, r(D') \geq k_2$ for some $k_1, k_2 \geq 0$, then by definition 4.2.1, we have that $\forall E \geq 0$ of degree k_1, $D - E$ is winnable, and that $\forall E' \geq 0$ of degree k_2, $D' - E'$ is winnable. Consider $E'' \geq 0$ of degree $k_1 + k_2$, and since we can break down $E'' = E_1 + E_2$ such that $\deg(E_1) = k_1, \deg(E_2) = k_2$. Since we have $(D - E_1)$ and $(D' - E_2)$ as winnable from above, we can say that $(D + D') - (E_1 + E_2) = (D + D') - E''$ is winnable too. Hence, by definition 4.2.1, we can say that $r(D + D') \geq k_1 + k_2$. Then using our definitions of k_1, k_2, we get the inequality $r(D + D') \geq r(D) + r(D')$. We present a version of this proof in Lean4 on lines 421-481 within Appendix A.7. □

Corollary 4.2.3. *For any graph G, with a canonical divisor K as defined in definition 4.1.6 $\deg(K) = 2g - 2$, where $g = |E| - |V| + 1$ is the genus of G.*

Proof. From definition 4.1.6 of a canonical divisor K, we have that $K := \sum_{v \in V}(\text{val}(v) - 2)v$. $\implies deg(K) = \sum_{v \in V}(\text{val}(v) - 2) = \sum_{v \in V}(\text{val}(v)) - 2\sum_{v \in V}(1) = 2|E| - 2|V| = 2(|E| - |V| + 1) - 2 = 2g - 2$. We present a version of this proof in Lean4 on lines 483-506 within Appendix A.7. □

4.3 Riemann-Roch Theorem for Graphs

Theorem 4.3.1. *(Riemann-Roch for graphs). Let D be a divisor on a (loopless, undirected) graph G of genus $g = |E| - |V| + 1$ with canonical divisor K. Then,*

$$r(D) - r(K - D) = 1 + \deg(D) - g.$$

Proof. By definition 4.2.1, there exists an effective divisor E such that $\deg(E) = r(D) + 1$ and $D - E$ is unwinnable. Using Dhar's algorithm from section 3.4 on $D - E$, we can find a q-reduced divisor $c + kq \sim D - E$, where c is superstable and $k < 0 \in \mathbb{Z}$ since $D - E$ is unwinnable.

Let us pick a maximal superstable $c' \geq c$. Consider the corresponding maximal unwinnable divisor $c' - q$. Let $\mathcal{O}$ be the corresponding acyclic orientation. Then, we have $D(\mathcal{O}) = c' - q \geq c + kq \sim D - E$ by proposition 4.1.13.

Let us define an effective divisor

$$H := (c' - c) - (k + 1)q \sim D(\mathcal{O}) - (D - E)$$

Adding $D(\overline{\mathcal{O}})$ on both sides gives us

$$D(\overline{\mathcal{O}}) + H \sim D(\overline{\mathcal{O}}) + D(\mathcal{O}) - (D + E)$$

Using definition 4.1.6 of K,

$$\implies K - H - D \sim D(\overline{\mathcal{O}}) - E$$

We know that $E \geq 0$, but $D(\overline{\mathcal{O}})$ is unwinnable, hence we get that $D(\overline{\mathcal{O}}) - E$ and $K - H - D$ are unwinnable. By definition 4.2.1, $r(K - D) < \deg(H)$.

$$\implies r(K - D) < \deg(D(\mathcal{O}) - (D - E)) \implies r(K - D) < \deg(D(\mathcal{O})) - \deg(D) + \deg(E)$$

Using proposition 4.1.14 and fact that $\deg(E) = r(D) + 1$,

$$\implies r(K - D) < g - 1 - \deg(D) + r(D) + 1 \implies \deg(D) - g < r(D) - r(K - D)$$

Now, since the above inequality is valid for any divisor $D \in Div(G)$, without loss of generality, we can substitute D with $K - D$ to get the following:

$$\implies \deg(K-D) - g < r(K-D) - r(D) \implies \deg(K) - \deg(D) - g < r(K-D) - r(D)$$

Using corollary 4.2.3, we have that $\deg(K) = 2g - 2$, which gives us:

$$\implies 2g - 2 - \deg(D) - g < r(K-D) - r(D) \implies g - 2 - \deg(D) < r(K-D) - r(D)$$

Thus, by using both the upper and lower bounds of the inequalities and the fact that by definition 4.2.1, rank is an integer, we can conclude that $r(K-D) - r(D) = 1 + \deg(D) - g$.

□

4.4 Application to Determination of Winnability

Finally, with all the tools we have developed so far, we will determine the "winnability" (rank) of a divisor D in this section.

Corollary 4.4.1. *A divisor D is maximal unwinnable if and only if the divisor $K - D$ is maximal unwinnable.*

Proof. By proposition 4.1.14, we have that $\deg(D) = g - 1$, and by definition 4.2.1, we have that $r(D) = -1$. Then, by using theorem 4.3.1, we have that $-1 - r(K-D) = \deg(D) + 1 - g = 0 \implies r(K-D) = -1 \implies (K-D)$ is unwinnable. Now, using corollary 4.2.3, consider $\deg(K-D) = \deg(K) - \deg(D) = 2g - 2 - g + 1 = g - 1$. Thus, by proposition 4.1.14, $K - D$ is maximal unwinnable. Similarly, replacing D with $(K - D)$ yields the other direction.

We present a version of this proof in Lean4 on lines 99-172 within Appendix A.8. □

Theorem 4.4.2 (Clifford's Theorem)**.** *Suppose $D \in Div(G)$ is a divisor with $r(D) \geq 0$ and $r(K - D) \geq 0$, then we have $r(D) \leq \frac{1}{2}\deg(D)$.*

Proof. By theorem 4.3.1, we have that $r(D) = r(K-D) + 1 + \deg(D) - g$. When $D = K$, using corollary 4.2.3, we have $r(K) = r(K-K) + 1 + \deg(K) - g = 0 + 1 + (2g-2) - g = g - 1$.

Then, by corollary 4.2.2, we have that $r(K) = r(D + K - D) \geq r(D) + r(K-D)$. Substituting the value of $r(K)$ into this inequality gives us $g - 1 \geq r(D) + r(K-D)$. Again, using theorem

4.3.1 to substitute in the value of $r(K-D)$ into this inequality gives us

$$g-1 \geq r(D)+r(D)-\deg(D)-1+g \implies r(D) \leq \frac{1}{2}\deg(G)$$

We present a version of this proof in Lean4 on lines 175-252 within Appendix A.8. □

Corollary 4.4.3. *Let $D \in Div(G)$.*

1. *If $\deg(D) < 0$, then $r(D) = -1$.*
2. *If $0 \leq \deg(D) \leq 2g-2$, then $r(D) \leq \frac{1}{2}\deg(D)$.*
3. *If $\deg(D) > 2g-2$, then $r(D) = \deg(D) - g$.*

Proof. (1): This follows from the definition 4.2.1 directly.

(2): To begin with, let us consider the case when D is unwinnable. This case is trivial because by definition 4.2.1 and by the given fact that $0 \leq \deg(D)$, we have $r(D) = -1 \leq 0 \leq \frac{1}{2}\deg(D)$.

Finally, we have two more sub-cases when $r(D) \geq 0$. Firstly, when $r(K-D) = -1$, we can use Riemann-Roch theorem 4.3.1 to state that $r(D) = r(K-D)+\deg(D)+1-g = -1+\deg(D)+1-g = \deg(D)-g$. We can transform the given condition to get $g \geq \frac{1}{2}\deg(D)+1$. Substituting this into the equality we got from Riemann-Roch, we obtain $r(D) \leq \deg(D) - \frac{1}{2}\deg(D) - 1 = \frac{1}{2}\deg(D) - 1 \leq \frac{1}{2}\deg(D)$. Secondly, in the last sub-case, when $r(K-D) \geq 0$, we are done directly using Clifford's theorem 4.4.2.

(3): Using Riemann-Roch theorem 4.3.1, we have $r(D) = r(K-D)+1+\deg(D)-g$, and since by definition 4.2.1, we have $r(K-D) \geq -1$, we can say that $r(D) \geq -1+1+\deg(D)-g = \deg(D)-g$. Given we have $\deg(D) > 2g-2$ in this part, then by corollary 4.2.3, we will have $\deg(D) > \deg(K) \implies \deg(K)-\deg(D) < 0 \implies \deg(K-D) < 0$. Hence, $K-D$ is unwinnable, so $r(K-D) = -1$ by definition 4.2.1, and thus $r(D) = \deg(D) - g$.

We present a version of this proof in Lean4 on lines 254-369 within Appendix A.8. □

Chapter 5

Formalization of Riemann-Roch for Chip Firing Graphs in Lean4

5.1 Introduction to Machine-Assisted Proving and Lean4

Proof assistants have become indispensable in modern mathematics for verifying complex proofs with absolute rigor. These tools, such as Lean4 [6], Coq [7], and Isabelle [8], allow mathematicians to encode definitions and proofs in a formal language that a computer can check step by step. This approach mitigates human errors and builds high-confidence proofs, especially as mathematical results grow in complexity. In recent years, the synergy between theorem proving and computer science has grown markedly [13]. Lean4, in particular, is a modern proof assistant designed as a verification system and a functional programming language. It introduces advanced features like metaprogramming [9] and an improved type-theoretic foundation, making it a powerful tool for researchers and educators. It has gained popularity among mathematicians (including prominent figures like Terence Tao [17, 18]) for pushing the boundaries of how theorems are proven [13]. By enforcing strict logical rules, Lean4 promotes precision and formality that is difficult to achieve in traditional pen-and-paper proofs.

The advantages of machine-assisted proving are numerous. Proof assistants provide an unparalleled level of rigor by systematically eliminating errors in proofs. They facilitate modularity by allowing substantive proofs to be broken into smaller, verifiable components, which can then be independently checked and reused. This modularity also fosters collaboration among mathematicians, enabling large teams to work on different parts of a proof without requiring every participant to understand the entire argument fully. Additionally, formalization efforts contribute to a growing library of reusable theorems and results such as Mathlib4 [10] for Lean4, which can serve as building blocks for future research. Beyond their use in research, proof assistants like Lean4 have proven to be valuable educational tools. Interactive platforms such as the "Natural Number Game" [11] introduce students to formal logic intuitively and engagingly, making mathematical proof-writing accessible and rigorous.

One exciting development is the integration of machine learning with formal theorem proving. Researchers are exploring AI to automate or assist in finding proofs, thereby reducing the manual effort required for complex theorems [13]. For example, large language models have been trained to suggest proof steps or complete proofs in systems like Lean. Recent work proposes *TheoremLlama*[3] and *MA-LoT*[4] frameworks, which train general-purpose language models to act as Lean4 proof producers and evaluators. These advances use neural networks to navigate the enormous search space of possible proofs and have shown promising results in automating parts of the proof process. The marriage of AI and proof assistants raises interesting questions: While machine learning can improve efficiency, one must ensure the reliability of the proofs generated. Nonetheless, the trend is clear—machine-assisted proving, augmented by AI, is becoming a crucial component of the mathematician's toolkit, enabling the formal verification of profound results that were once impractical to check exhaustively by hand. As mentioned before, Lean4 has been used in large-scale collaborative projects [12], demonstrating its capacity to handle theoretical and applied mathematical problems.

Lean4 provides a suitable platform for formalizing graph-theoretic results thanks to its expressive type system and supportive community libraries. Throughout this thesis, we have used Lean4 to formalize the chip-firing game and its associated theorems. In what follows, we focus on the pinnacle of these efforts: the formalization of the graph-theoretic Riemann–Roch theorem. We will see how Lean4 is employed to ensure every logical detail of the proof is correct and how the formalization process yields insights into the theorem's structure.

Additionally, recent innovations in VSCode extensions have led to the development of PaperProof [19], which allows us to visualize our Lean4 proofs through an intuitive user interface. Since the interface is interactive, as of now, it is hard to extract the images. Despite this, we have presented some lemmas with smaller proofs as figures in Appendix C.

Despite these advantages, machine-assisted proving still faces significant challenges. One of the primary obstacles is the time and effort required to formalize proofs. While human mathematicians often take shortcuts by relying on intuition or prior knowledge, proof assistants demand exhaustive detail, making the formalization process labor-intensive. Another challenge is scalability: verifying highly complex proofs often requires substantial computational resources, limiting the practicality of these tools for specific applications. Furthermore, while integrating proof assistants with machine learning and language models holds great promise, this area is still in its infancy. For example, combining the logical rigor of proof assistants with the generative capabilities of large language models could accelerate formalization and suggest innovative proof strategies. However, as of now, realizing this potential will require significant advancements in both fields.

5.2 Riemann–Roch for Graphs in Lean4

We now present the formalized version of the Riemann–Roch theorem for chip-firing graphs. Recall from earlier chapters that a *divisor* on a graph $G = (V, E)$ is an integer-valued function on the vertices, and its *degree* $\deg(D)$ is the sum of its values. The graph's *genus* g is given by $g = |E| - |V| + 1$, and the *canonical divisor* K is a special divisor defined by $K(v) = \mathrm{val}(v) - 2$ for each vertex v (Definition 4.1.6). The chip-firing "dollar-game" interpretation allows us to talk about winnability: a divisor D is *winnable* (or equivalent to an effective divisor) if one can redistribute chips (via legal vertex firings) so that no vertex has negative chips. The *rank* $r(D)$ (Definition 4.2.1) measures how far D is from being unwinnable: intuitively, $r(D) \geq 0$ if D is winnable, and generally $r(D)$ is the maximum number of chips one can continuously remove from D while keeping it winnable. We proved in Theorem 4.3.1 (graph Riemann–Roch) that for any divisor D on a loopless, undirected graph G of genus g,

$$r(D) - r(K - D) = \deg(D) + 1 - g.$$

This combinatorial Riemann–Roch theorem and our formal proof mirror the constructive approach via chip-firing and orientations.

In Lean4, we formalized all the necessary ingredients to state and prove this theorem as we walked along the previous chapters. We can state the Riemann–Roch theorem in Lean4 with these definitions. The formal statement introduces necessary hypotheses (for example, that G is loopless and undirected) and then asserts the equality of $r(D) - r(K - D)$ and $\deg(D) + 1 - g$. In code, the theorem is proven in appendix A.8. Let us step through the implementation. We ensure that each intermediate result (lemmas about orientations, degrees, etc.) is separately proven and invoked, mirroring the logical flow of the informal proof. The `rcases` tactic unpacks existential statements, like finding an effective divisor E or a maximal superstable configuration c', using helper lemmas from `RRGHelpers.lean`. The `linarith` tactic automates linear arithmetic reasoning, which is crucial for balancing the rank and degree terms. The proof establishes equality by proving both a lower and upper bound, leveraging the rank-degree inequality and properties of the canonical divisor.

Formalizing the Riemann–Roch theorem in Lean4 required a combination of graph-theoretic reasoning and careful encoding of combinatorial algorithms. One of the significant insights was leveraging the relationship between chip-firing game configurations and graph orientations. In the proof, we made crucial use of *acyclic orientations* of G with a specified source vertex. We formalized the concept of an orientation in Lean4 and proved that each acyclic orientation $\mathcal{O}$ (with a unique source) corresponds bijectively to $D(\mathcal{O})$ on the graph. Based on Dhar's burning algorithm, this correspondence allowed us to translate between combinatorial objects (orientations) and algebraic ones (divisors) within Lean. For example, we proved a Lean4 lemma capturing the fact that firing all vertices indicated by Dhar's algorithm yields an orientation $\mathcal{O}$ for which $D(\mathcal{O})$ is a maximal unwinnable divisor. Such lemmas were key stepping stones in the formal proof.

Divisor properties like linear equivalence and superstable configurations must also be carefully encoded. One important proposition we formalized states that any *maximal unwinnable* divisor has degree $g - 1$. This fact was used in the Riemann–Roch proof to relate the degree of a particular divisor to the genus g. The formal proof of this proposition in Lean4 followed the intuitive argument: if D is unwinnable but adding any chip to any vertex makes it winnable, then D must distribute $g-1$ chips in a specific way across the graph.

Our formalization in Lean4 builds on a modular codebase, organized as follows, with files gradually building on top of the previous ones:

1. **Basic.lean (section A.1)**: Defines core structures like `CFGraph` and `CFDiv`.

2. **CFGraphExample.lean (section A.2)**: Defines and computationally verifies `CFGraph`.

3. **Config.lean (section A.3)**: Handles configurations, subsets of divisors excluding a vertex q.

4. **Orientation.lean (section A.3)**: Formalizes graph orientations and its properties.

5. **Rank.lean (section A.5)**: Implements the rank function, critical to Riemann-Roch.

6. **Helpers.lean (section A.6)**: Auxiliary axioms, lemmas, and propositions for `RRGHelpers.lean`.

7. **RRGHelpers.lean (section A.7)**: Provides auxiliary theorems specific to the Riemann Roch.

8. **RiemannRochForGraphs.lean (section A.8)**: Contains the main theorem's proof.

Throughout the formalization, computational techniques complemented theoretical reasoning. For instance, we declared some helper theorems and lemmas as axioms for the sake of simplicity as we were facing induction issues with the types, and thus, in the interest of time since we knew of their validity from our written proofs in the earlier chapters, which are in turn inspired and backed by Corry and Perkinson [2].

It is worth noting that some computations in this domain are inherently complex. Determining the rank of a divisor on an arbitrary graph is an NP-hard problem [5]. No efficient general algorithm is known for computing large graphs' $r(D)$. In our Lean4 development, we did not attempt to *compute* ranks for general graphs, but rather to *prove* properties about rank symbolically. Lean4's strength is checking proofs for all cases simultaneously (through induction or contradiction) rather than brute-force search. We were careful to structure the proof to avoid heavy case analyses or explorations of exponentially many graph configurations. The formal proof remains efficient and abstract by relying on general lemmas and symmetry arguments (for example, swapping D with $K - D$ in inequality when needed). This showcases an important lesson: Formalization encourages a proof style that is often more general and conceptual, avoiding ad-hoc reasoning that might be infeasible to verify by brute force.

5.3 Challenges in Formalization

The journey of encoding the Riemann–Roch theorem in Lean4 had many challenges. A primary difficulty was translating high-level mathematical concepts into the lower-level objects that Lean understands. Mathematical arguments often skip trivial or repetitive steps or implicitly assume certain constructions are possible. In Lean4, every detail must be explicit. For example, our paper proof might say "orient the graph acyclically by choosing an arbitrary vertex order." In Lean, we had to construct this orientation step by step: we proved a lemma that given a finite graph, one can well-order its vertices and direct each edge from the lower-ranked to the higher-ranked vertex, yielding an acyclic orientation. This constructive proof was necessary to use such orientations later on since Lean4 cannot accept an argument that assumes existence without a method to obtain the object. This illustrates a trade-off between classical and constructive approaches. In a classical proof, one might invoke a non-constructive lemma (like Zorn's lemma or a counting argument) to assert the existence of a particular divisor or orientation. In the formalization, we often opted for a constructive route (such as explicitly using Dhar's algorithm or sorting vertices) to avoid using the axiom of choice or excluding the middle unless absolutely needed. Lean4 supports classical reasoning (which we used for convenience in some parts), but keeping proofs constructive when possible makes them more computationally meaningful and often more straightforward to check.

Managing the complexity of the proofs was another challenge. The Riemann–Roch proof involves multiple intermediate claims about divisors and orientations (for instance, establishing the inequalities that sandwich $r(D) - r(K - D)$ between $\deg(D) + 1 - g$ from above and below). Each of these had to be proven in Lean as a separate lemma or theorem. Organizing these lemmas in a coherent order required careful planning. We modularized the formalization: first came basic lemmas about firing and linear equivalence, then properties of ranks and degrees, then lemmas about orientations and their associated divisors, and finally, the main theorem and its corollaries. Ensuring that each lemma had all the necessary hypotheses (and no extra ones) was tedious. Often, a lemma failed to apply in the main proof because we assumed G was connected in one place but not elsewhere. We had to iterate on the statements to balance generality and usability. This process underscored how formalization forces meticulous clarity about assumptions easily overlooked in hand-written proof work.

The limitations of proof automation in Lean4 became apparent in some of the more involved combinatorial arguments. Lean4 has powerful tactics (such as 'simp' and 'rw' for rewriting), but these are not a substitute for human insight in a complex proof. For example, to prove the key inequality in Riemann–Roch (that $r(K-D) < \deg(D(\mathcal{O})) - \deg(D) + \deg(E)$ in our paper proof notation, which leads to one side of the bound), we had to guide Lean through a series of substitutions and logical deductions. No single built-in tactic could manage this automatically. We often broke goals into smaller sub-goals that the automation could handle or explicitly instructed Lean which prior lemma to use at each step. This is a common situation in formal proofs: human creativity is needed to identify the proper intermediate claims and the overall proof strategy, while the proof assistant reliably checks the mechanical steps and can automate only the straightforward parts. We gradually built a toolbox of custom tactics for recurring patterns in our proofs (for example, to handle inequalities of a specific shape or to automatically verify that a given divisor is unwinnable by trying a finite sequence of firings). These helped mitigate the grunt work but required initial effort to set up.

Another challenge was dealing with performance and complexity. As mentioned, rank computation is NP-hard in general [5], so we had to be careful not to accidentally write a definition or lemma that entailed an explosive search. Lean4's evaluator or simplifier could, in theory, get bogged down if we phrased something in a non-terminating way. We encountered this when first defining $r(D)$: a naive recursive definition might explore all subsets of vertices (exponential in number) to find effective divisors. We avoided this by using a non-algorithmic definition of rank (quantifying over degrees rather than over subsets directly), which is logically clear but not intended to be executed. During formalization, we learnt to separate the existence proofs from actual algorithms. We invoke algorithms like Dhar's when we need them for constructive proofs, but for something like rank, which is hard to compute, we only prove things about it without ever trying to compute it in general. Moreover, Lean4's handling of inductive types and recursion required that we prove certain functions terminate. For instance, in the case of Lemma 4.1.10 and Corollary 4.4.1, we had to show that each firing reduces a well-founded measure (like the lexicographic combination of "number of chips that can still fire" and "graph size") to convince Lean that the algorithm always ends. Crafting these termination arguments was an additional proof layer not evident in the traditional mathematical presentation.

One subtle issue was ensuring that our Lean development remained in sync with mathematical intuition despite the different nature of logic. For instance, in our paper proof, we used the phrase "without loss of generality, replace D by $K-D$" to obtain a symmetric inequality. In Lean, "without loss of generality" has to be replaced by an explicit invocation of a symmetry lemma or by proving a separate lemma for the swapped case. We ended up proving a symmetry result: since the statement of Riemann–Roch is symmetric under exchanging D with $K - D$ (up to the sign of the formula), it suffices to prove one inequality and then invoke this symmetry for the other. Lean4 made us explicit about such steps, providing a complete understanding of the proof's structure.

Lastly, a practical challenge was that Lean4 is a relatively new system, and its math library (at the time of formalization) was not as extensive as the Lean 3 mathlib. We often had to develop basic graph theory notions from scratch or port them from known results. For example, we defined our multi-edged graph structure 'CFGraph.' We proved basic properties like "loopless and undirected" for induction use because such lemmas were not yet in the standard library. Working with a cutting-edge proof assistant meant we were simultaneously preparing work that would be eventual direct contributions to its mathematics library. This slowed us down initially but paid off by giving us complete control over definitions.

The challenges in formalizing Riemann–Roch ranged from translating intuitive arguments into formal proofs to overcoming tooling and library limitations. Grappling with each difficulty we encountered left us with a corresponding solution or workaround: introducing constructive arguments, organizing the proof into many axioms and lemmas, writing custom tactics, and occasionally developing new adjacent generic content that can be contributed to the library directly. The result proves that even deep combinatorial theorems can be successfully captured in Lean4. However, it requires perseverance and a willingness to handle many minutiae that traditional proofs gloss over.

5.4 Lessons Learned and Future Directions

The formalization of the Riemann–Roch theorem for graphs in Lean4 has been a rich learning experience, highlighting both the rewards and the hurdles of machine-assisted proving. One key takeaway is the increased rigor and clarity that formalization enforces. The process also underscored the value of modular proofs: breaking down a complex theorem into lemmas made the formal proof possible and improved our understanding of the theorem's anatomy. Each lemma we proved in Lean corresponds to a tangible mathematical insight (like the behavior of maximal unwinnable divisors or the effect of adding two winnable divisors). The computer proof became a dialogue partner, forcing us to justify every claim and often suggesting alternative approaches when a direct formal translation of a human proof was difficult.

Another lesson learned is that formalizing combinatorial theorems can guide the development of better algorithms and structures. Moreover, the exercise of formalization often reveals which parts of a proof are canonical and which are ad hoc. For example, our formal proof of Riemann–Roch heavily used the concept of acyclic orientations with a unique source, which appears to be a fundamental combinatorial bridge between chip-firing and divisor theory. Any future attempt to generalize or extend Riemann–Roch (say, to other chip-firing-like games) will likely use a similar bridge. On the other hand, certain tricks in the informal proof (like a specific choice of a divisor H in the proof of 4.3.1) turned out not to be unique—there were many ways to formalize that step, and Lean4 let us explore variants. Thus, we learned which aspects of the proof are structurally important and which are merely choices of convenience.

Looking forward, this formalization opens up several avenues in both mathematics and computer-assisted proof. On the graph theory front, one immediate extension is to explore the full *Baker–Norine theory* on graphs. We have formalized Riemann–Roch and Clifford's theorem; a natural next step is to formalize the graph-theoretic analog of Brill–Noether theory, which concerns the existence of special divisors of given rank and degree on graphs. Another direction is to consider chip-firing on metric graphs (tropical curves) and attempt a formal comparison between the discrete and continuous cases. This would contribute to bridging combinatorial and algebraic geometry in a formal proof assistant setting.

There are also opportunities in the realm of combinatorial optimization and network theory. The chip-firing game is closely related to flows in networks, the dollar game being analogous to balancing a flow with supplies and demands at vertices. Our Lean4 development could be extended to formally verify algorithms that compute flows or cuts using chip-firing methods. For example, specific optimal chip-firing sequences solve the max-flow min-cut problem on planar graphs. By formalizing those connections, we could use Lean4 to verify classical network optimization algorithms in a new way, reducing them to chip-firing processes and leveraging our correctness proofs of those processes. This aligns with a broader goal of using formal methods to guarantee the correctness of algorithms in combinatorial optimization, where subtle bugs can sometimes go unnoticed in informal proofs.

From a machine-assisted proving perspective, the success of this project suggests that Lean4 (and similar tools) are now robust enough to handle nontrivial combinatorial theorems. As Lean4's math library grows, future formalizations will become easier. In a few years, one can imagine that much of the groundwork we had to develop (graph theory basics, etc.) manually will be readily available, allowing new users to jump directly into formalizing advanced chip-firing results without reinventing the wheel. We also anticipate improved automation and AI integration in proof assistants. Indeed, as mentioned in the introduction, there is ongoing work on AI-driven tools to assist Lean users. There is scope to experiment with a prototype "agentic" AI tool that takes a rough outline or "proof sketch" and attempts to fill in the formal details. The idea is to let mathematicians input their intuitive strategy (for example, "use Dhar's algorithm to get an orientation, then form divisor H and apply Riemann–Roch") and have the AI suggest the formal Lean tactics to realize that strategy. Our experience with TheoremLlama [3] has been encouraging: as demonstrated further in the case study in [4, Appendix D], even when the AI does not fully solve a problem, it frequently offers valuable insights or automates routine components of the proof process. In the future, such technology could dramatically speed up the formalization of results like Riemann–Roch by reducing the manual translation overhead. This approach can be refined further and potentially with tighter integration with Lean4, thus turning formal proof development into a more interactive, high-level process where humans provide insights and AI + Lean handles at least some of the tedious details.

Reflecting on the broader impact of formalizing a theorem like Riemann–Roch for graphs, this work contributes to the growing body of formally verified mathematics, which not only serves as a proof of concept that "it can be done" but also ensures that the results will stand the test of time. Anyone can now check our Lean4 code to see the exact assumptions (structurally and temporarily axiomatic) and the proof, leaving no ambiguity. As mathematics leans towards greater complexity, having theorems in a proof assistant means they can be reused as reliable building blocks for future theorems (much as we rely on a lemma in our proofs, future formal proofs can import our Riemann–Roch theorem directly). In the long run, we envision this work contributing to an even more robust and complete library of formalized combinatorial theorems that can be applied to problems in computer science (e.g., verification of network algorithms) and mathematics alike. Formalizing chip-firing games and graphical Riemann–Roch is one step in that direction. It paves the way for tackling even more ambitious results with confidence that combining human insight and machine precision will continue to scale up.

Bibliography

[1] M. Baker and S. Norine, Riemann-Roch and Abel-Jacobi theory on a finite graph, Advances in Mathematics, 215 (2007), 766–788. 1, 129

[2] S. Corry and D. Perkinson, Divisors and Sandpiles: An Introduction to Chip-Firing, American Mathematical Society, Providence, Rhode Island. 2018. 1, 19, 23, 25, 26, 28, 29, 35, 37, 45, 122, 124, 126, 129

[3] R. Wang, J. Zhang, Y. Jia, R. Pan, S. Diao, R. Pi, and T. Zhang, TheoremLlama: Transforming General-Purpose LLMs into Lean4 Experts, arXiv preprint, https://arxiv.org/abs/2407.03203, 2024. 3, 42, 50

[4] R. Wang, R. Pan, Y. Li, J. Zhang, Y. Jia, S. Diao, R. Pi, J. Hu, and T. Zhang, MA-LoT: Multi-Agent Lean-based Long Chain-of-Thought Reasoning enhances Formal Theorem Proving, arXiv preprint, https://arxiv.org/abs/2503.03205, 2025. 42, 50

[5] V. Kiss and L. Tóthmérész, "Chip-firing games on Eulerian digraphs and **NP**-hardness of computing the rank of a divisor on a graph," Discrete Applied Mathematics, vol. 193, pp. 48–56, Oct. 2015. DOI: http://dx.doi.org/10.1016/j.dam.2015.04.030. 2, 37, 45, 47

[6] de Moura, L., Ullrich, S.: The lean 4 theorem prover and programming language. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction – CADE 28. pp. 625–635. Springer International Publishing, Cham (2021) 3, 9, 41

[7] The Coq Development Team: The Coq reference manual – release 8.19.0. https://coq.inria.fr/doc/V8.19.0/refman (2024) 3, 41

[8] Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higherorder logic, vol. 2283. Springer Science Business Media (2002) 3, 41

[9] Gabriel Ebner, Sebastian Ullrich, Jared Roesch, Jeremy Avigad, and Leonardo de Moura. 2017. A metaprogramming framework for formal verification. PACMPL 1, ICFP (2017), 34:1–34:29. https://doi.org/10.1145/3110278 41

[10] mathlib: The lean mathematical library. CoRR abs/1910.09336 (2019), http://arxiv.org/abs/1910.09336 16, 21, 42

[11] Kevin Buzzard, Jon Eugster, et al. *Natural Number Game*. 2023. Available at: https://adam.math.hhu.de/#/g/leanprover-community/nng4. Accessed: 2025-01-31. 42

[12] Lean Community, *Wiedijk's 100 Theorems in Lean*, Lean Prover Community, https://leanprover-community.github.io/100.html. Accessed: April 2025. 3, 42

[13] I. Teng, "Theorem Proving using Machine Learning and Lean 4," *Isaac Teng's Blog*, April 15, 2024. Available at: https://isaacteng.co.uk/2024/04/15/theorem-proving-using-machine-learning-lean-4/. [Accessed: February 28, 2025]. 41, 42

[14] F. Cools, J. Draisma, S. Payne, and E. Robeva, A tropical proof of the Brill–Noether Theorem, *Advances in Mathematics*, 230 (2012), 759–776. DOI: https://doi.org/10.1016/j.aim.2012.02.019. 2

[15] N. Pflueger, Brill–Noether varieties of k-gonal curves, *Advances in Mathematics*, 312 (2017), 46–63. DOI: https://doi.org/10.1016/j.aim.2017.01.027. 2

[16] B. Osserman, Limit linear series and the Amini-Baker construction, arXiv preprint, https://arxiv.org/abs/1707.03845, 2017. 130

[17] T. Tao, "A slightly longer Lean 4 proof tour," *What's new*, December 5, 2023. Available at: https://terrytao.wordpress.com/2023/12/05/a-slightly-longer-lean-4-proof-tour/. [Accessed: March 20, 2025]. 41

[18] T. Tao, Y. Dillies, and B. Mehta, "Formalizing the proof of PFR in Lean 4 using Blueprint: A short tour," December 2023. [Accessed: March 20, 2025]. 41

[19] N. Weibel, A. Ispas, B. Signer, and M. C. Norrie, "PaperProof: a paper-digital proof-editing system," in *CHI '08 Extended Abstracts on Human Factors in Computing Systems*, Florence, Italy, pp. 2349–2354, ACM, 2008. DOI: https://doi.org/10.1145/1358628.1358682. 42, 131

Appendix A

Lean4 Implementation for Chip Firing & Graphical Riemann Roch

A.1 Basic Properties for Chip Firing Graphs (Basic.lean)

```
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Fold
import Mathlib.Data.Multiset.Basic
import Mathlib.Algebra.Group.Subgroup.Basic
import Mathlib.Tactic.Abel
import Mathlib.LinearAlgebra.Matrix.GeneralLinearGroup.Defs
import Mathlib.Algebra.BigOperators.Group.Finset

import Init.Core
import Init.NotationExtra

import Paperproof

set_option linter.unusedVariables false
set_option trace.split.failure true
set_option linter.unusedSectionVars false

open Multiset Finset

-- Assume V is a finite type with decidable equality
variable {V : Type} [DecidableEq V] [Fintype V]

-- Define a set of edges to be loopless only if it doesn't have loops
def isLoopless (edges : Multiset (V × V)) : Bool :=
  Multiset.card (edges.filter (λ e => (e.1 = e.2))) = 0

def isLoopless_prop (edges : Multiset (V × V)) : Prop :=
  ∀ v, (v, v) ∉ edges

lemma isLoopless_prop_bool_equiv (edges : Multiset (V × V)) :
    isLoopless_prop edges ↔ isLoopless edges = true := by
  unfold isLoopless_prop isLoopless
```

```
  constructor
  · intro h
    apply decide_eq_true
    rw [Multiset.card_eq_zero]
    simp only [Multiset.eq_zero_iff_forall_not_mem]
    intro e he
    have h_eq : e.1 = e.2 := by
      exact Multiset.mem_filter.mp he |>.2
    have he' : e ∈ edges := by
      exact Multiset.mem_filter.mp he |>.1
    cases e with
    | mk a b =>
      simp at h_eq
      have : (a, b) = (a, a) := by rw [h_eq]
      rw [this] at he'
      exact h a he'

  · intro h
    intro v
    intro hv
    apply False.elim
    have h_filter : (v, v) ∈ Multiset.filter (λ e => e.1 = e.2) edges := by
      apply Multiset.mem_filter.mpr
      constructor
      · exact hv
      · simp

    have h_card : Multiset.card (Multiset.filter (λ e => e.1 = e.2) edges) > 0 := by
      apply Multiset.card_pos_iff_exists_mem.mpr
      exists (v, v)

    have h_eq : Multiset.card (Multiset.filter (λ e => e.1 = e.2) edges) = 0 := by
      -- Use the fact that isLoopless edges = true means the cardinality is 0
      unfold isLoopless at h
      -- Since h : decide (...) = true, we extract the underlying proposition
      apply of_decide_eq_true h

    linarith

-- Define a set of edges to be undirected only if it doesn't have both (v, w) and (w, v)
def isUndirected (edges : Multiset (V × V)) : Bool :=
  Multiset.card (edges.filter (λ e => (e.2, e.1) ∈ edges)) = 0

def isUndirected_prop (edges : Multiset (V × V)) : Prop :=
  ∀ v1 v2, (v1, v2) ∈ edges → (v2, v1) ∉ edges

lemma isUndirected_prop_bool_equiv (edges : Multiset (V × V)) :
    isUndirected_prop edges ↔ isUndirected edges = true := by
  unfold isUndirected_prop isUndirected
  constructor
```

```
  · intro h_prop -- Assume isUndirected_prop edges
    apply decide_eq_true -- Goal: decide (...) = true
    rw [Multiset.card_eq_zero] -- Goal: filter (...) = 0
    simp only [Multiset.eq_zero_iff_forall_not_mem] -- Goal: ∀ (a : V × V), a ∉ filter (...) edges
    intro e he_filter -- Assume e ∈ filter (...) edges
    -- Unpack he_filter
    have h_mem_edges : e ∈ edges := Multiset.mem_filter.mp he_filter |>.1
    have h_rev_mem_edges : (e.2, e.1) ∈ edges := Multiset.mem_filter.mp he_filter |>.2
    -- Use h_prop to get a contradiction
    exact h_prop e.1 e.2 h_mem_edges h_rev_mem_edges
  · intro h_bool -- Assume isUndirected edges = true
    intro v1 v2 h_v1v2_mem h_v2v1_mem -- Assume v1, v2, (v1, v2) ∈ edges, (v2, v1) ∈ edges. Goal: False
    apply False.elim
    -- Show (v1, v2) is in the filtered multiset
    have h_filter_mem : (v1, v2) ∈ Multiset.filter (λ e => (e.2, e.1) ∈ edges) edges := by
      apply Multiset.mem_filter.mpr
      constructor
      · exact h_v1v2_mem -- (v1, v2) ∈ edges
      · simp -- Goal: ((v1, v2).2, (v1, v2).1) ∈ edges
        exact h_v2v1_mem -- (v2, v1) ∈ edges
    -- The card of the filtered multiset must be > 0
    have h_card_pos : Multiset.card (Multiset.filter (λ e => (e.2, e.1) ∈ edges) edges) > 0 := by
      apply Multiset.card_pos_iff_exists_mem.mpr
      exists (v1, v2)
    -- Get card = 0 from h_bool
    have h_card_zero : Multiset.card (Multiset.filter (λ e => (e.2, e.1) ∈ edges) edges) = 0 := by
      apply of_decide_eq_true h_bool
    -- Contradiction
    linarith -- h_card_pos contradicts h_card_zero


-- Multigraph with undirected and loopless edges
structure CFGraph (V : Type) [DecidableEq V] [Fintype V] :=
  (edges : Multiset (V × V))
  (loopless : isLoopless edges = true)
  (undirected: isUndirected edges = true)

-- Divisor as a function from vertices to integers
def CFDiv (V : Type) := V → ℤ

-- Divisor addition (pointwise)
instance : Add (CFDiv V) := ⟨λ D₁ D₂ => λ v => D₁ v + D₂ v⟩

-- Divisor subtraction (pointwise)
instance : Sub (CFDiv V) := ⟨λ D₁ D₂ => λ v => D₁ v - D₂ v⟩
```

```lean
-- Zero divisor
instance : Zero (CFDiv V) := ⟨λ _ => 0⟩

-- Neg for divisors
instance : Neg (CFDiv V) := ⟨λ D => λ v => -D v⟩

-- Add coercion from V → ℤ to CFDiv V
instance : Coe (V → ℤ) (CFDiv V) where
  coe f := f

-- Properties of divisor arithmetic
@[simp] lemma add_apply (D₁ D₂ : CFDiv V) (v : V) :
  (D₁ + D₂) v = D₁ v + D₂ v := rfl

@[simp] lemma sub_apply (D₁ D₂ : CFDiv V) (v : V) :
  (D₁ - D₂) v = D₁ v - D₂ v := rfl

@[simp] lemma zero_apply (v : V) :
  (0 : CFDiv V) v = 0 := rfl

@[simp] lemma neg_apply (D : CFDiv V) (v : V) :
  (-D) v = -(D v) := rfl

@[simp] lemma distrib_sub_add (D₁ D₂ D₃ : CFDiv V) :
  D₁ - (D₂ + D₃) = (D₁ - D₂) - D₃ := by
  funext x
  simp [sub_apply, add_apply]
  ring

@[simp] lemma add_sub_distrib (D₁ D₂ D₃ : CFDiv V) :
  (D₁ + D₂) - D₃ = (D₁ - D₃) + D₂ := by
  funext x
  simp [sub_apply, add_apply]
  ring

/-- Lemma: Lambda form of divisor subtraction equals standard form -/
lemma divisor_sub_eq_lambda (G : CFGraph V) (D₁ D₂ : CFDiv V) :
  (λ v => D₁ v - D₂ v) = D₁ - D₂ := by
  funext v
  simp [sub_apply]

-- Number of edges between two vertices as an integer
def num_edges (G : CFGraph V) (v w : V) : ℕ :=
  ↑(Multiset.card (G.edges.filter (λ e => e = (v, w) ∨ e = (w, v))))

-- Lemma: Number of edges is non-negative
lemma num_edges_nonneg (G : CFGraph V) (v w : V) :
  num_edges G v w ≥ 0 := by
  unfold num_edges
  apply Nat.cast_nonneg
```

```lean
-- Degree (Valence) of a vertex as an integer
def vertex_degree (G : CFGraph V) (v : V) : ℤ :=
  ↑(Multiset.card (G.edges.filter (λ e => e.fst = v ∨ e.snd = v)))

-- Lemma: Vertex degree is non-negative
lemma vertex_degree_nonneg (G : CFGraph V) (v : V) :
  vertex_degree G v ≥ 0 := by
  unfold vertex_degree
  apply Nat.cast_nonneg

-- Firing move at a vertex
def firing_move (G : CFGraph V) (D : CFDiv V) (v : V) : CFDiv V :=
  λ w => if w = v then D v - vertex_degree G v else D w + num_edges G v w

-- Borrowing move at a vertex
def borrowing_move (G : CFGraph V) (D : CFDiv V) (v : V) : CFDiv V :=
  λ w => if w = v then D v + vertex_degree G v else D w - num_edges G v w

-- Define finset_sum using Finset.fold
def finset_sum {α β} [AddCommMonoid β] (s : Finset α) (f : α → β) : β :=
  s.fold (· + ·) 0 f

-- Define set_firing to use finset_sum with consistent types
def set_firing (G : CFGraph V) (D : CFDiv V) (S : Finset V) : CFDiv V :=
  λ w => D w + finset_sum S (λ v => if w = v then -vertex_degree G v else num_edges G
    v w)

-- Define the group structure on CFDiv V
instance : AddGroup (CFDiv V) := Pi.addGroup

-- Define the firing vector for a single vertex
def firing_vector (G : CFGraph V) (v : V) : CFDiv V :=
  λ w => if w = v then -vertex_degree G v else num_edges G v w

-- Define the principal divisors generated by firing moves at vertices
def principal_divisors (G : CFGraph V) : AddSubgroup (CFDiv V) :=
  AddSubgroup.closure (Set.range (firing_vector G))

-- Lemma: Principal divisors contain the firing vector at a vertex
lemma mem_principal_divisors_firing_vector (G : CFGraph V) (v : V) :
  firing_vector G v ∈ principal_divisors G := by
  apply AddSubgroup.subset_closure
  apply Set.mem_range_self

-- Define linear equivalence of divisors
def linear_equiv (G : CFGraph V) (D D' : CFDiv V) : Prop :=
  D' - D ∈ principal_divisors G

-- [Proven] Proposition: Linear equivalence is an equivalence relation on Div(G)
theorem linear_equiv_is_equivalence (G : CFGraph V) : Equivalence (linear_equiv G) :=
    by
```

```
  apply Equivalence.mk
  -- Reflexivity
  · intro D
    unfold linear_equiv
    have h_zero : D - D = 0 := by simp
    rw [h_zero]
    exact AddSubgroup.zero_mem _

  -- Symmetry
  · intros D D' h
    unfold linear_equiv at *
    have h_symm : D - D' = -(D' - D) := by simp [sub_eq_add_neg, neg_sub]
    rw [h_symm]
    exact AddSubgroup.neg_mem _ h

  -- Transitivity
  · intros D D' D'' h1 h2
    unfold linear_equiv at *
    have h_trans : D'' - D = (D'' - D') + (D' - D) := by simp
    rw [h_trans]
    exact AddSubgroup.add_mem _ h2 h1

-- Define divisor class determined by a divisor
def divisor_class (G : CFGraph V) (D : CFDiv V) : Set (CFDiv V) :=
  {D' : CFDiv V | linear_equiv G D D'}

-- Define effective divisors (in terms of non-negativity, returning a Bool)
def effective_bool (D : CFDiv V) : Bool :=
  ↑((Finset.univ.filter (fun v => D v < 0)).card = 0)

-- Define effective divisors (in terms of equivalent Prop)
def effective (D : CFDiv V) : Prop :=
  ∀ v : V, D v ≥ 0

-- Define the set of effective divisors
-- Note: We use the Prop returned by `effective` in set comprehension
def Div_plus (G : CFGraph V) : Set (CFDiv V) :=
  {D : CFDiv V | effective D}

-- Define winnable divisor
-- Note: We use the Prop returned by `linear_equiv` in set comprehension
def winnable (G : CFGraph V) (D : CFDiv V) : Prop :=
  ∃ D' ∈ Div_plus G, linear_equiv G D D'

-- Define the complete linear system of divisors
def complete_linear_system (G: CFGraph V) (D: CFDiv V) : Set (CFDiv V) :=
  {D' : CFDiv V | linear_equiv G D D' ∧ effective D'}

-- Degree of a divisor
def deg (D : CFDiv V) : ℤ := Σ v, D v
def deg_prop (D : CFDiv V) : Prop := deg D = Σ v, D v
```

```lean

/-- Axiomatic Definition: Linear equivalence preserves degree of divisors -/
axiom linear_equiv_preserves_deg {V : Type} [DecidableEq V] [Fintype V]
  (G : CFGraph V) (D D' : CFDiv V) (h : linear_equiv G D D') : deg D = deg D'

-- Define a firing script as a function from vertices to integers
def firing_script (V : Type) := V → ℤ

-- Define Laplacian matrix as an |V| x |V| integer matrix
open Matrix
variable [Fintype V]

def laplacian_matrix (G : CFGraph V) : Matrix V V ℤ :=
  λ i j => if i = j then vertex_degree G i else - (num_edges G i j)

-- Note: The Laplacian matrix L is given by Deg(G) - A, where Deg(G) is the diagonal matrix of degrees and A is the adjacency matrix.
-- This matrix can be used to represent the effect of a firing script on a divisor.

-- Apply the Laplacian matrix to a firing script, and current divisor to get a new divisor
def apply_laplacian (G : CFGraph V) (σ : firing_script V) (D: CFDiv V) : CFDiv V :=
  fun v => (D v) - (laplacian_matrix G).mulVec σ v

-- Define q-reduced divisors
def q_reduced (G : CFGraph V) (q : V) (D : CFDiv V) : Prop :=
  (∀ v ∈ {v | v ≠ q}, D v ≥ 0) ∧
  (∀ S : Finset V, S ⊆ (Finset.univ.filter (λ v => v ≠ q)) → S.Nonempty →
    ∃ v ∈ S, D v < finset_sum (Finset.univ.filter (λ w => ¬(w ∈ S))) (λ w => num_edges G v w))

-- Define the ordering of divisors
def divisor_order (G : CFGraph V) (q : V) (D D' : CFDiv V) : Prop :=
  (∃ T : Finset V, T ⊆ (Finset.univ.filter (λ v => v ≠ q)) ∧ D' = set_firing G D T) ∧
  (∀ T' : Finset V, T' ⊆ (Finset.univ.filter (λ v => v ≠ q)) → D' ≠ set_firing G D T')

-- Define the ordering of divisors using the divisor_order relation
def divisor_ordering (G : CFGraph V) (q : V) (D D' : CFDiv V) : Prop :=
  divisor_order G q D' D

-- Legal set-firing: Ensure no vertex in S is in debt after firing
def legal_set_firing (G : CFGraph V) (D : CFDiv V) (S : Finset V) : Prop :=
  ∀ v ∈ S, set_firing G D S v ≥ 0

/-- Axiom: Q-reduced form uniquely determines divisor class in the following sense:
    If two divisors D1 and D2 are both q-reduced and linearly equivalent,
    then they must be equal. This is a key uniqueness property that shows
    every divisor class contains exactly one q-reduced representative.
    This was especially hard to prove in Lean4, so we are leaving it as an axiom for
```

```
    the time being. -/
axiom q_reduced_unique_class (G : CFGraph V) (q : V) (D₁ D₂ : CFDiv V) :
  q_reduced G q D₁ ∧ q_reduced G q D₂ ∧ linear_equiv G D₁ D₂ → D₁ = D₂
```

A.2 Chip Firing Graphs Illustration (CFGraphExample.lean)

```
import ChipFiringWithLean.Basic
import Mathlib.Data.Int.Order.Lemmas
import Mathlib.Data.Int.Order.Basic
import Mathlib.Tactic.NormNum
import Mathlib.LinearAlgebra.Matrix.Symmetric
import Paperproof

set_option linter.unusedVariables false
set_option trace.split.failure true

open Multiset Finset

inductive Person : Type
  | A | B | C | E
  deriving DecidableEq

instance : Fintype Person where
  elems := {Person.A, Person.B, Person.C, Person.E}
  complete := by {
    intro x
    cases x
    all_goals { simp }
  }

-- Example usage for Person type in a loopless graph
def exampleEdges : Multiset (Person × Person) :=
  Multiset.ofList [
    (Person.A, Person.B),
    (Person.B, Person.C),
    (Person.C, Person.E)
  ]
theorem loopless_example_edges : isLoopless exampleEdges = true := by rfl
theorem loopless_prop_example_edges : isLoopless_prop exampleEdges := by
  unfold isLoopless_prop
  decide
theorem undirected_example_edges : isUndirected exampleEdges = true := by rfl
theorem undirected_prop_example_edges : isUndirected_prop exampleEdges := by
  unfold isUndirected_prop
  decide

-- Example usage for Person type in a graph with a loop
def edgesWithLoop : Multiset (Person × Person) :=
  Multiset.ofList [
    (Person.A, Person.B),
```

```
    (Person.A, Person.A),   -- This is a loop
    (Person.B, Person.C),
  ]
theorem loopless_test_edges_with_loop : isLoopless edgesWithLoop = false := by rfl

-- Example usage for Person type in a graph with a non-undirected edge
def edgesWithNonUndirected : Multiset (Person × Person) :=
  Multiset.ofList [
    (Person.A, Person.B),
    (Person.B, Person.C),
    (Person.C, Person.E),
    (Person.E, Person.C)  -- This is a non-undirected edge
  ]
theorem undirected_test_edges_with_non_undirected : isUndirected
    edgesWithNonUndirected = false := by rfl

def example_graph : CFGraph Person := {
  edges := Multiset.ofList [
    (Person.A, Person.B), (Person.B, Person.C),
    (Person.A, Person.C), (Person.A, Person.E),
    (Person.A, Person.E), (Person.E, Person.C)
  ],
  loopless := by rfl,
  undirected := by rfl
}

def initial_wealth : CFDiv Person :=
  fun v => match v with
  | Person.A => 2
  | Person.B => -3
  | Person.C => 4
  | Person.E => -1

-- Test vertex degrees
theorem vertex_degree_A : vertex_degree example_graph Person.A = 4 := by rfl
theorem vertex_degree_B : vertex_degree example_graph Person.B = 2 := by rfl
theorem vertex_degree_C : vertex_degree example_graph Person.C = 3 := by rfl
theorem vertex_degree_E : vertex_degree example_graph Person.E = 3 := by rfl

-- Test edge counts
theorem edge_count_AB : num_edges example_graph Person.A Person.B = 1 := by rfl
theorem edge_count_BA : num_edges example_graph Person.B Person.A = 1 := by rfl
theorem edge_count_BC : num_edges example_graph Person.B Person.C = 1 := by rfl
theorem edge_count_CB : num_edges example_graph Person.C Person.B = 1 := by rfl
theorem edge_count_AC : num_edges example_graph Person.A Person.C = 1 := by rfl
theorem edge_count_CA : num_edges example_graph Person.C Person.A = 1 := by rfl
theorem edge_count_AE : num_edges example_graph Person.A Person.E = 2 := by rfl
theorem edge_count_EA : num_edges example_graph Person.E Person.A = 2 := by rfl
theorem edge_count_EC : num_edges example_graph Person.E Person.C = 1 := by rfl
theorem edge_count_CE : num_edges example_graph Person.C Person.E = 1 := by rfl
theorem edge_count_BE : num_edges example_graph Person.B Person.E = 0 := by rfl
```

```lean
theorem edge_count_EB : num_edges example_graph Person.E Person.B = 0 := by rfl

-- Test No self-loops
theorem edge_count_AA : num_edges example_graph Person.A Person.A = 0 := by rfl
theorem edge_count_BB : num_edges example_graph Person.B Person.B = 0 := by rfl
theorem edge_count_CC : num_edges example_graph Person.C Person.C = 0 := by rfl
theorem edge_count_EE : num_edges example_graph Person.E Person.E = 0 := by rfl

-- Test Charlie lending through an individual firing move
def after_charlie_lends := firing_move example_graph initial_wealth Person.C
theorem charlie_wealth_after_lending : after_charlie_lends Person.C = 1 := by rfl
theorem bob_wealth_after_charlie_lends : after_charlie_lends Person.B = -2 := by rfl

-- Test set firing W₁ = {A,E,C}
def W₁ : Finset Person := {Person.A, Person.E, Person.C}
def after_W₁_firing := set_firing example_graph initial_wealth W₁
theorem alice_wealth_after_W₁ : after_W₁_firing Person.A = 1 := by rfl
theorem bob_wealth_after_W₁ : after_W₁_firing Person.B = -1 := by rfl
theorem charlie_wealth_after_W₁ : after_W₁_firing Person.C = 3 := by rfl
theorem elise_wealth_after_W₁ : after_W₁_firing Person.E = -1 := by rfl

-- Test set firing W₂ = {A,E,C}
def W₂ : Finset Person := W₁
def after_W₂_firing := set_firing example_graph after_W₁_firing W₂
theorem alice_wealth_after_W₂ : after_W₂_firing Person.A = 0 := by rfl
theorem bob_wealth_after_W₂ : after_W₂_firing Person.B = 1 := by rfl
theorem charlie_wealth_after_W₂ : after_W₂_firing Person.C = 2 := by rfl
theorem elise_wealth_after_W₂ : after_W₂_firing Person.E = -1 := by rfl

-- Test set firing W₃ = {B,C}
def W₃ : Finset Person := {Person.B, Person.C}
def after_W₃_firing := set_firing example_graph after_W₂_firing W₃
theorem alice_wealth_after_W₃ : after_W₃_firing Person.A = 2 := by rfl
theorem bob_wealth_after_W₃ : after_W₃_firing Person.B = 0 := by rfl
theorem charlie_wealth_after_W₃ : after_W₃_firing Person.C = 0 := by rfl
theorem elise_wealth_after_W₃ : after_W₃_firing Person.E = 0 := by rfl

-- Test borrowing moves
def after_bob_borrows := borrowing_move example_graph initial_wealth Person.B
theorem bob_wealth_after_borrowing : after_bob_borrows Person.B = -1 := by rfl
theorem alice_wealth_after_bob_borrows : after_bob_borrows Person.A = 1 := by rfl
theorem charlie_wealth_after_bob_borrows : after_bob_borrows Person.C = 3 := by rfl

-- Test degree of divisors
theorem initial_wealth_degree : deg initial_wealth = 2 := by rfl
theorem after_W₁_degree : deg after_W₁_firing = 2 := by rfl
theorem after_W₂_degree : deg after_W₂_firing = 2 := by rfl
theorem after_W₃_degree : deg after_W₃_firing = 2 := by rfl

-- Test effectiveness of divisors
theorem initial_not_effective : ¬effective initial_wealth := by {
```

```
  intro h
  have hB := h Person.B
  have h_neg : initial_wealth Person.B = -3 := by rfl
  have h_lt : -3 < 0 := by norm_num
  exact not_le.mpr h_lt hB
}
theorem initial_not_effective_bool : effective_bool initial_wealth = false := by rfl
theorem after_W₃_firing_effective : effective_bool after_W₃_firing = true := by rfl

-- Test Laplacian matrix values and symmetricity
def example_laplacian := laplacian_matrix example_graph
theorem laplacian_diagonal_A : example_laplacian Person.A Person.A = 4 := by rfl
theorem laplacian_diagonal_B : example_laplacian Person.B Person.B = 2 := by rfl
theorem laplacian_diagonal_C : example_laplacian Person.C Person.C = 3 := by rfl
theorem laplacian_diagonal_E : example_laplacian Person.E Person.E = 3 := by rfl
theorem laplacian_off_diagonal_AB : example_laplacian Person.A Person.B = -1 := by rfl
theorem laplacian_off_diagonal_AC : example_laplacian Person.A Person.C = -1 := by rfl
theorem laplacian_off_diagonal_AE : example_laplacian Person.A Person.E = -2 := by rfl
theorem laplacian_off_diagonal_BC : example_laplacian Person.B Person.C = -1 := by rfl
theorem laplacian_off_diagonal_BE : example_laplacian Person.B Person.E = 0 := by rfl
theorem laplacian_off_diagonal_CE : example_laplacian Person.C Person.E = -1 := by rfl
theorem check_example_laplacian_symmetry : Matrix.IsSymm example_laplacian := by {
  apply Matrix.IsSymm.ext
  intros i j
  cases i <;> cases j
  all_goals {
    rfl
  }
}

-- Test script firing through laplacians
def firing_script_example : firing_script Person := fun v => match v with
  | Person.A => 0
  | Person.B => -1
  | Person.C => 1
  | Person.E => 0
def res_div_post_lap_based_script_firing := apply_laplacian example_graph
    firing_script_example initial_wealth
theorem lap_based_script_firing_preserves_degree : deg
    res_div_post_lap_based_script_firing = 2 := by rfl

-- Test divisor that is not q-reduced with respect to Person.A
def non_q_reduced_example : CFDiv Person := fun v => match v with
  | Person.A => 1
  | Person.B => -1  -- violates non-negativity condition for non-q vertices
  | Person.C => 2
  | Person.E => 1

theorem non_q_reduced_example_is_invalid : ¬q_reduced example_graph Person.A
    non_q_reduced_example := by {
  intro h
```

```
  cases h with
  | intro h1 h2 => {
    have hB := h1 Person.B (by simp)
    simp [non_q_reduced_example] at hB
  }
}
```

A.3 Configurations on Chip Firing Graphs (Config.lean)

```
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Fold
import Mathlib.Data.Multiset.Basic
import Mathlib.Algebra.Group.Subgroup.Basic
import Mathlib.Tactic.Abel
import Mathlib.LinearAlgebra.Matrix.GeneralLinearGroup.Defs
import Mathlib.Algebra.BigOperators.Group.Finset
import ChipFiringWithLean.Basic
import Paperproof

set_option linter.unusedVariables false
set_option trace.split.failure true
set_option linter.unusedSectionVars false

open Multiset Finset

-- Assume V is a finite type with decidable equality
variable {V : Type} [DecidableEq V] [Fintype V]

/-- A configuration on a graph G with respect to a distinguished vertex q.
    Represents an element of ℤ(V\{q}) ⊆ ℤV with non-negativity constraints on V\{q}.

    Fields:
    * vertex_degree - Assignment of integers to vertices
    * non_negative_except_q - Proof that all values except at q are non-negative -/
structure Config (V : Type) (q : V) :=
  /-- Assignment of integers to vertices representing the number of chips at each vertex -/
  (vertex_degree : V → ℤ)
  /-- Proof that all vertices except q have non-negative values -/
  (non_negative_except_q : ∀ v : V, v ≠ q → vertex_degree v ≥ 0)

/-- The degree of a configuration is the sum of all values except at q.
    deg(c) = Σ_{v ∈ V\{q}} c(v) -/
def config_degree {q : V} (c : Config V q) : ℤ :=
  Σ v in (univ.filter (λ v => v ≠ q)), c.vertex_degree v

/-- Ordering on configurations: c ≥ c' if c(v) ≥ c'(v) for all v ∈ V.
    This is a pointwise comparison of the number of chips at each vertex. -/
def config_ge {q : V} (c c' : Config V q) : Prop :=
  ∀ v : V, c.vertex_degree v ≥ c'.vertex_degree v
```

```
/-- A configuration is non-negative if all vertices (including q) have non-negative values.
    This is stronger than the basic Config constraint which only requires non-negativity on V\{q}. -/
def config_nonnegative {q : V} (c : Config V q) : Prop :=
  ∀ v : V, c.vertex_degree v ≥ 0

/-- Linear equivalence of configurations: c  c' if they can be transformed into one another
    through a sequence of lending and borrowing operations. The difference between configurations
    must be in the subgroup generated by firing moves. -/
def config_linear_equiv {q : V} (G : CFGraph V) (c c' : Config V q) : Prop :=
  let diff := λ v => c'.vertex_degree v - c.vertex_degree v
  diff ∈ AddSubgroup.closure (Set.range (λ v => λ w => if w = v then -vertex_degree G v else num_edges G v w))

-- Definition of the out-degree of a vertex v ∈ S with respect to a subset S ⊆ V \ {q}
-- This counts edges from v to vertices *outside* S (but not q).
-- outdeg_S(v) = |{ (v, w) ∈ E | w ∈ (V \ {q}) \ S }|
def outdeg_S (G : CFGraph V) (q : V) (S : Finset V) (v : V) : ℤ :=
  -- Sum num_edges from v to w, where w is not in S and not q.
  Σ w in (univ.filter (λ x => x ≠ q)).filter (λ x => x ∉ S), (num_edges G v w : ℤ)

-- Standard definition of Superstability:
-- A configuration c is superstable w.r.t. q if for every non-empty subset S of V \ {q},
-- there is at least one vertex v in S that cannot fire without borrowing,
-- meaning its chip count c(v) is strictly less than its out-degree w.r.t. S.
def superstable (G : CFGraph V) (q : V) (c : Config V q) : Prop :=
  ∀ S : Finset V, S ⊆ univ.filter (λ x => x ≠ q) → S.Nonempty →
    ∃ v ∈ S, c.vertex_degree v < outdeg_S G q S v

/-- A maximal superstable configuration has no legal firings and dominates all other superstable configs -/
def maximal_superstable {q : V} (G : CFGraph V) (c : Config V q) : Prop :=
  superstable G q c ∧ ∀ c' : Config V q, superstable G q c' → config_ge c' c

/-- Axiom: Defining winnability of configurations through linear equivalence and chip addition.
    Used to show that adding a chip at any non-q vertex results in a winnable configuration
    when starting from a linearly equivalent divisor to a maximal superstable configuration.
    Proving this inductively is a bit tricky at the moment, and we ran into infinite recursive loop,
    thus we are declaring this as an axiom. -/
axiom winnable_through_equiv_and_chip (G : CFGraph V) (q : V) (D : CFDiv V) (c : Config V q) :
```

```
  linear_equiv G D (λ v => c.vertex_degree v - if v = q then 1 else 0) →
  maximal_superstable G c →
  ∀ v : V, v ≠ q →
  winnable G (λ w => D w + if w = v then 1 else 0)
```

A.4 Orientations and Directed Paths (Orientation.lean)

```
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Fold
import Mathlib.Data.Multiset.Basic
import Mathlib.Algebra.Group.Subgroup.Basic
import Mathlib.Tactic.Abel
import Mathlib.LinearAlgebra.Matrix.GeneralLinearGroup.Defs
import Mathlib.Algebra.BigOperators.Group.Finset
import ChipFiringWithLean.Basic
import ChipFiringWithLean.Config
import Paperproof

set_option linter.unusedVariables false
set_option trace.split.failure true
set_option linter.unusedSectionVars false

open Multiset Finset

-- Assume V is a finite type with decidable equality
variable {V : Type} [DecidableEq V] [Fintype V]

/-- An orientation of a graph assigns a direction to each edge.
    The consistent field ensures each undirected edge corresponds to exactly
    one directed edge in the orientation. -/
structure Orientation (G : CFGraph V) :=
  /-- The set of directed edges in the orientation -/
  (directed_edges : Multiset (V × V))
  /-- Preserves edge counts between vertex pairs -/
  (count_preserving : ∀ v w,
    Multiset.count (v, w) G.edges + Multiset.count (w, v) G.edges =
    Multiset.count (v, w) directed_edges + Multiset.count (w, v) directed_edges)
  /-- No bidirectional edges in the orientation -/
  (no_bidirectional : ∀ v w,
    Multiset.count (v, w) directed_edges = 0 ∨
    Multiset.count (w, v) directed_edges = 0)

/-- Number of edges directed into a vertex under an orientation -/
def indeg (G : CFGraph V) (O : Orientation G) (v : V) : ℕ :=
  Multiset.card (O.directed_edges.filter (λ e => e.snd = v))

/-- Number of edges directed out of a vertex under an orientation -/
def outdeg (G : CFGraph V) (O : Orientation G) (v : V) : ℕ :=
  Multiset.card (O.directed_edges.filter (λ e => e.fst = v))

```

```
/-- A vertex is a source if it has no incoming edges (indegree = 0) -/
def is_source (G : CFGraph V) (O : Orientation G) (v : V) : Bool :=
  indeg G O v = 0

/-- A vertex is a sink if it has no outgoing edges (outdegree = 0) -/
def is_sink (G : CFGraph V) (O : Orientation G) (v : V) : Bool :=
  outdeg G O v = 0

/-- Helper function to check if two consecutive vertices form a directed edge -/
def is_directed_edge (G : CFGraph V) (O : Orientation G) (u v : V) : Bool :=
  (u, v) ∈ O.directed_edges

/-- Axiom: Well-foundedness of vertex levels
    This was especially hard to prove in Lean4, so we are leaving it as an axiom for the time being. -/
axiom vertex_measure_decreasing (G : CFGraph V) (O : Orientation G) (v : V) :
  is_source G O v = false →
  ∀ u, is_directed_edge G O u v = true →
  (univ.filter (λ w => is_directed_edge G O w u)).card <
  (univ.filter (λ w => is_directed_edge G O w v)).card

/-- Axiom: If u is in the filter set for vertex_level calculation of v,
    then there is a directed edge from u to v
    This was especially hard to prove in Lean4, so we are leaving it as an axiom for the time being. -/
axiom filter_implies_directed_edge (G : CFGraph V) (O : Orientation G) (v u : V) :
  u ∈ univ.filter (λ w => is_directed_edge G O w v) →
  is_directed_edge G O u v = true

/-- Axiom: Filter membership for vertex levels
    This was especially hard to prove in Lean4, so we are leaving it as an axiom for the time being. -/
axiom vertex_filter_membership (G : CFGraph V) (O : Orientation G) (v u : V) :
  u ∈ univ.filter (λ w => is_directed_edge G O w v)

/-- The level of a vertex is its position in the topological ordering -/
def vertex_level (G : CFGraph V) (O : Orientation G) (v : V) : ℕ :=
  if h : is_source G O v then 0
  else Nat.succ (Finset.sup (univ.filter (λ u => is_directed_edge G O u v))
                            (λ u => vertex_level G O u))
termination_by
  Finset.card (univ.filter (λ u => is_directed_edge G O u v))
decreasing_by {
  apply vertex_measure_decreasing G O v
  · exact eq_false_of_ne_true h
  · apply filter_implies_directed_edge G O v u
    exact vertex_filter_membership G O v u
}

/-- Vertices at a given level in the orientation -/
def vertices_at_level (G : CFGraph V) (O : Orientation G) (l : ℕ) : Finset V :=
```

```
  univ.filter (λ v => vertex_level G O v = l)

/-- Helper function for safe list access -/
def list_get_safe {α : Type} (l : List α) (i : Nat) : Option α :=
  l.get? i

/-- A directed path in a graph under an orientation -/
structure DirectedPath (G : CFGraph V) (O : Orientation G) where
  /-- The sequence of vertices in the path -/
  vertices : List V
  /-- Every consecutive pair forms a directed edge -/
  valid_edges : ∀ (i : Nat), i + 1 < vertices.length →
    match (vertices.get? i, vertices.get? (i + 1)) with
    | (some u, some v) => is_directed_edge G O u v
    | _ => False
  /-- All vertices in the path are distinct -/
  distinct_vertices : ∀ (i j : Nat), i < vertices.length → j < vertices.length → i ≠ j →
    match (vertices.get? i, vertices.get? j) with
    | (some u, some v) => u ≠ v
    | _ => True

/-- A directed cycle is a directed path whose first and last vertices coincide.
    Apart from the repetition of the first/last vertex, all other vertices in the cycle are distinct. -/
structure DirectedCycle (G : CFGraph V) (O : Orientation G) :=
  (vertices : List V)
  /-- Every consecutive pair of vertices forms a directed edge in the orientation. -/
  (valid_edges : ∀ (i : Nat), i + 1 < vertices.length →
    match (vertices.get? i, vertices.get? (i + 1)) with
    | (some u, some v) => is_directed_edge G O u v
    | _ => False)
  /-- The cycle condition: the first vertex equals the last, ensuring a closed loop. -/
  (cycle_condition : vertices.length > 0 ∧ vertices.get? 0 = vertices.get? (vertices.length - 1))
  /-- All internal vertices (ignoring the last vertex which is the same as the first)
      are distinct from each other. This ensures there are no other repeated vertices
      besides the repetition at the end forming the cycle. -/
  (distinct_internal_vertices : ∀ (i j : Nat),
    i < vertices.length - 1 →
    j < vertices.length - 1 →
    i ≠ j →
    match (vertices.get? i, vertices.get? j) with
    | (some u, some v) => u ≠ v
    | _ => True)

/-- Check if there are edges in both directions between two vertices -/
def has_bidirectional_edges (G : CFGraph V) (O : Orientation G) (u v : V) : Prop :=
  ∃ e₁ e₂, e₁ ∈ O.directed_edges ∧ e₂ ∈ O.directed_edges ∧ e₁ = (u, v) ∧ e₂ = (v, u)

```

```
/-- All multiple edges between same vertices point in same direction -/
def consistent_edge_directions (G : CFGraph V) (O : Orientation G) : Prop :=
  ∀ u v : V, ¬has_bidirectional_edges G O u v

/-- An orientation is acyclic if it has no directed cycles and
    maintains consistent edge directions between vertices -/
def is_acyclic (G : CFGraph V) (O : Orientation G) : Prop :=
  consistent_edge_directions G O ∧
  ¬∃ p : DirectedPath G O,
    p.vertices.length > 0 ∧
    match (p.vertices.get? 0, p.vertices.get? (p.vertices.length - 1)) with
    | (some u, some v) => u = v
    | _ => False


/-- Vertices that are not sources must have at least one incoming edge. -/
lemma indeg_ge_one_of_not_source (G : CFGraph V) (O : Orientation G) (v : V) :
    ¬ is_source G O v → indeg G O v ≥ 1 := by
  intro h_not_source -- h_not_source : is_source G O v = false
  unfold is_source at h_not_source -- h_not_source : (decide (indeg G O v = 0)) = false
  apply Nat.one_le_iff_ne_zero.mpr -- Goal is indeg G O v ≠ 0
  intro h_eq_zero -- Assume indeg G O v = 0
  have h_decide_true : decide (indeg G O v = 0) = true := by
    rw [h_eq_zero]
    exact decide_eq_true rfl
  rw [h_decide_true] at h_not_source
  simp at h_not_source

/-- For vertices that are not sources, indegree - 1 is non-negative. -/
lemma indeg_minus_one_nonneg_of_not_source (G : CFGraph V) (O : Orientation G) (v : V) :
    ¬ is_source G O v → 0 ≤ (indeg G O v : ℤ) - 1 := by
  intro h_not_source
  have h_indeg_ge_1 : indeg G O v ≥ 1 := indeg_ge_one_of_not_source G O v h_not_source
  apply Int.sub_nonneg_of_le
  exact Nat.cast_le.mpr h_indeg_ge_1

/-- Configuration associated with a source vertex q under orientation O.
    Requires O to be acyclic and q to be the unique source.
    For each vertex v ≠ q, assigns indegree(v) - 1 chips. Assumes q is the unique source. -/
def config_of_source {G : CFGraph V} {O : Orientation G} {q : V} -- Make G, O, q implicit
    (h_acyclic : is_acyclic G O) (h_unique_source : ∀ w, is_source G O w → w = q) : Config V q :=
  { vertex_degree := λ v => if v = q then 0 else (indeg G O v : ℤ) - 1,
    non_negative_except_q := λ v hv => by
      simp [vertex_degree]
      split_ifs with h_eq
      · contradiction
```

```
      · have h_not_source : ¬ is_source G O v := by
          intro hs_v
          exact hv (h_unique_source v hs_v)
        -- Need to provide implicit arguments G O v explicitly now
        exact indeg_minus_one_nonneg_of_not_source G O v h_not_source
  }

/-- The divisor associated with an orientation assigns indegree - 1 to each vertex -/
def divisor_of_orientation (G : CFGraph V) (O : Orientation G) : CFDiv V :=
  λ v => indeg G O v - 1

/-- The canonical divisor assigns degree - 2 to each vertex.
    This is independent of orientation and equals D(O) + D(reverse(O)) -/
def canonical_divisor (G : CFGraph V) : CFDiv V :=
  λ v => (vertex_degree G v) - 2

/-- Auxillary Lemma: Double canonical difference is identity -/
lemma canonical_double_diff (G : CFGraph V) (D : CFDiv V) :
  (λ v => canonical_divisor G v - (canonical_divisor G v - D v)) = D := by
  funext v; simp

/-- Definition (Axiomatic): Canonical divisor is sum of two acyclic orientations -/
axiom canonical_is_sum_orientations {V : Type} [DecidableEq V] [Fintype V] (G : CFGraph V) :
  ∃ (O₁ O₂ : Orientation G),
    is_acyclic G O₁ ∧ is_acyclic G O₂ ∧
    canonical_divisor G = λ v => divisor_of_orientation G O₁ v + divisor_of_orientation G O₂ v
```

A.5 Rank and Genus (Rank.lean)

```
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Fold
import Mathlib.Data.Multiset.Basic
import Mathlib.Algebra.Group.Subgroup.Basic
import Mathlib.Tactic.Abel
import Mathlib.LinearAlgebra.Matrix.GeneralLinearGroup.Defs
import Mathlib.Algebra.BigOperators.Group.Finset
import ChipFiringWithLean.Basic
import ChipFiringWithLean.Config
import ChipFiringWithLean.Orientation
import Paperproof

set_option linter.unusedVariables false
set_option trace.split.failure true
set_option linter.unusedSectionVars false

open Multiset Finset

-- Assume V is a finite type with decidable equality
```

```
variable {V : Type} [DecidableEq V] [Fintype V]

/-- Definition of maximal winnable divisor -/
def maximal_winnable (G : CFGraph V) (D : CFDiv V) : Prop :=
  winnable G D ∧ ∀ v : V, ¬winnable G (λ w => D w + if w = v then 1 else 0)

/-- A divisor is maximal unwinnable if it is unwinnable but adding
    a chip to any vertex makes it winnable -/
def maximal_unwinnable (G : CFGraph V) (D : CFDiv V) : Prop :=
  ¬winnable G D ∧ ∀ v : V, winnable G (λ w => D w + if w = v then 1 else 0)

/-- Given an acyclic orientation O with a unique source q, returns a configuration c(O) -/
def orientation_to_config (G : CFGraph V) (O : Orientation G) (q : V)
    (h_acyclic : is_acyclic G O) (h_unique_source : ∀ w, is_source G O w → w = q) : Config V q :=
  config_of_source h_acyclic h_unique_source

/-- The genus of a graph is its cycle rank: |E| - |V| + 1 -/
def genus (G : CFGraph V) : ℤ :=
  Multiset.card G.edges - Fintype.card V + 1

/-- A divisor has rank -1 if it is not winnable -/
def rank_eq_neg_one_wrt_winnability (G : CFGraph V) (D : CFDiv V) : Prop :=
  ¬(winnable G D)

/-- A divisor has rank -1 if its complete linear system is empty -/
def rank_eq_neg_one_wrt_complete_linear_system (G : CFGraph V) (D : CFDiv V) : Prop :=
  complete_linear_system G D = ∅

/-- Given a divisor D and amount k, returns all possible ways
    to remove k dollars from D (i.e. all divisors E where D-E has degree k) -/
def remove_k_dollars (D : CFDiv V) (k : ℕ) : Set (CFDiv V) :=
  {E | effective E ∧ deg E = k}

/-- A divisor D has rank ≥ k if the game is winnable after removing any k dollars -/
def rank_geq (G : CFGraph V) (D : CFDiv V) (k : ℕ) : Prop :=
  ∀ E ∈ remove_k_dollars D k, winnable G (λ v => D v - E v)

/-- Helper to check if a divisor has exactly k chips removed at some vertex -/
def has_k_chips_removed (D : CFDiv V) (E : CFDiv V) (k : ℕ) : Prop :=
  ∃ v : V, E = (λ w => D w - if w = v then k else 0)

/-- Helper to check if all k-chip removals are winnable -/
def all_k_removals_winnable (G : CFGraph V) (D : CFDiv V) (k : ℕ) : Prop :=
  ∀ E : CFDiv V, has_k_chips_removed D E k → winnable G E

/-- Helper to check if there exists an unwinnable configuration after removing k+1 chips -/
def exists_unwinnable_removal (G : CFGraph V) (D : CFDiv V) (k : ℕ) : Prop :=
  ∃ E : CFDiv V, effective E ∧ deg E = k + 1 ∧ ¬(winnable G (λ v => D v - E v))
```

```
/-- Lemma: If a divisor is winnable, there exists an effective divisor linearly equivalent to it -/
lemma winnable_iff_exists_effective (G : CFGraph V) (D : CFDiv V) :
  winnable G D ↔ ∃ D' : CFDiv V, effective D' ∧ linear_equiv G D D' := by
  unfold winnable Div_plus
  simp only [Set.mem_setOf_eq]

/-- Axiom: Rank existence and uniqueness -/
axiom rank_exists_unique (G : CFGraph V) (D : CFDiv V) :
  ∃! r : ℤ, (r = -1 ∧ rank_eq_neg_one_wrt_winnability G D) ∨
    (r ≥ 0 ∧ rank_geq G D r.toNat ∧ exists_unwinnable_removal G D r.toNat ∧
     ∀ k' : ℕ, k' > r.toNat → ¬(rank_geq G D k'))

/-- The rank function for divisors -/
noncomputable def rank (G : CFGraph V) (D : CFDiv V) : ℤ :=
  Classical.choose (rank_exists_unique G D)

/-- Definition: Properties of rank function with respect to effective divisors -/
def rank_effective_char {V : Type} [DecidableEq V] [Fintype V] (G : CFGraph V) (D : CFDiv V) (r : ℤ) :=
  rank G D = r ↔
  (∀ E : CFDiv V, effective E → deg E = r + 1 → ¬(winnable G (λ v => D v - E v))) ∧
  (∀ E : CFDiv V, effective E → deg E = r → winnable G (λ v => D v - E v))

/-- Definition (Axiomatic): Helper for rank characterization to get effective divisor -/
axiom rank_get_effective {V : Type} [DecidableEq V] [Fintype V] (G : CFGraph V) (D : CFDiv V) :
  ∃ E : CFDiv V, effective E ∧ deg E = rank G D + 1 ∧ ¬(winnable G (λ v => D v - E v))

/-- Rank satisfies the defining properties -/
axiom rank_spec (G : CFGraph V) (D : CFDiv V) :
  let r := rank G D
  (r = -1 ↔ rank_eq_neg_one_wrt_winnability G D) ∧
  (∀ k : ℕ, r ≥ k ↔ rank_geq G D k) ∧
  (∀ k : ℤ, k ≥ 0 → r = k ↔
    rank_geq G D k.toNat ∧
    exists_unwinnable_removal G D k.toNat ∧
    ∀ k' : ℕ, k' > k.toNat → ¬(rank_geq G D k'))

/-- Axiomatic Definition: The zero divisor has rank 0 -/
axiom zero_divisor_rank (G : CFGraph V) : rank G (λ _ => 0) = 0

/-- Helpful corollary: rank = -1 exactly when divisor is not winnable -/
theorem rank_neg_one_iff_unwinnable (G : CFGraph V) (D : CFDiv V) :
  rank G D = -1 ↔ ¬(winnable G D) := by
  exact (rank_spec G D).1

/-- Lemma: If rank is not non-negative, then it equals -1 -/
```

```
lemma rank_neg_one_of_not_nonneg {V : Type} [DecidableEq V] [Fintype V]
  (G : CFGraph V) (D : CFDiv V) (h_not_nonneg : ¬(rank G D ≥ 0)) : rank G D = -1 := by
  -- rank_exists_unique gives ∃! r, P r ∨ Q r
  -- Classical.choose_spec gives (P r ∨ Q r) ∧ ∀ y, (P y ∨ Q y) → y = r, where r = rank G D
  have h_exists_unique_spec := Classical.choose_spec (rank_exists_unique G D)
  -- We only need the existence part: P r ∨ Q r
  have h_disjunction := h_exists_unique_spec.1
  -- Now use Or.elim on the disjunction
  apply Or.elim h_disjunction
  · -- Case 1: rank G D = -1 ∧ rank_eq_neg_one_wrt_winnability G D
    intro h_case1
    -- The goal is rank G D = -1, which is the first part of h_case1
    exact h_case1.1
  · -- Case 2: rank G D ≥ 0 ∧ rank_geq G D (rank G D).toNat ∧ ...
    intro h_case2
    -- This case contradicts the hypothesis h_not_nonneg
    have h_nonneg : rank G D ≥ 0 := h_case2.1
    -- Derive contradiction using h_not_nonneg
    exact False.elim (h_not_nonneg h_nonneg)

/-- Axiom: Linear equivalence is preserved when adding chips, provided deg D = g - 1
    This makes sense because such a D is maximal unwinnable, and adding a chip to a maximal unwinnable divisor
    is equivalent to adding a chip to the canonical divisor.
    This was especially hard to prove in Lean4, so we are leaving it as an axiom for the time being. -/
axiom linear_equiv_add_chip {V : Type} [DecidableEq V] [Fintype V]
  (G : CFGraph V) (D : CFDiv V) (v : V)
  (h_deg : deg D = genus G - 1) :
  linear_equiv G
    (λ w => D w + if w = v then 1 else 0)
    (λ w => (canonical_divisor G w - D w) + if w = v then 1 else 0)
```

A.6 Helper Axioms, Lemmas and Theorems for Intermediate Results (Helpers.lean)

```
import ChipFiringWithLean.Basic
import ChipFiringWithLean.Config
import ChipFiringWithLean.Orientation
import ChipFiringWithLean.Rank
import Mathlib.Algebra.Ring.Int
import Paperproof
import Mathlib.Algebra.BigOperators.Group.Multiset
import Mathlib.Algebra.BigOperators.Group.Finset
import Mathlib.Data.Finset.Basic
import Mathlib.Data.Finset.Fold
import Mathlib.Data.Multiset.Basic
import Mathlib.Data.Nat.Cast.Basic
```

```lean
import Mathlib.Data.Finset.Card

set_option linter.unusedVariables false
set_option trace.split.failure true

open Multiset Finset

-- Assume V is a finite type with decidable equality
variable {V : Type} [DecidableEq V] [Fintype V]


/-
# Helpers for Proposition 3.2.4
-/

/- Axiom: Existence of a q-reduced representative for any divisor class
   This was especially hard to prove in Lean4, so we are leaving it as an axiom for the time being. -/
axiom exists_q_reduced_representative (G : CFGraph V) (q : V) (D : CFDiv V) :
  ∃ D' : CFDiv V, linear_equiv G D D' ∧ q_reduced G q D'

/- [Proven] Helper lemma: Uniqueness of the q-reduced representative within a divisor class -/
lemma uniqueness_of_q_reduced_representative (G : CFGraph V) (q : V) (D : CFDiv V)
  (D₁ D₂ : CFDiv V) (h₁ : linear_equiv G D D₁ ∧ q_reduced G q D₁)
  (h₂ : linear_equiv G D D₂ ∧ q_reduced G q D₂) : D₁ = D₂ := by
  -- Extract information from hypotheses
  have h_equiv_D_D1 : linear_equiv G D D₁ := h₁.1
  have h_qred_D1 : q_reduced G q D₁ := h₁.2
  have h_equiv_D_D2 : linear_equiv G D D₂ := h₂.1
  have h_qred_D2 : q_reduced G q D₂ := h₂.2

  -- Use properties of the equivalence relation linear_equiv
  let equiv_rel := linear_equiv_is_equivalence G
  -- Symmetry: linear_equiv G D D₁ → linear_equiv G D₁ D
  have h_equiv_D1_D : linear_equiv G D₁ D := equiv_rel.symm h_equiv_D_D1
  -- Transitivity: linear_equiv G D₁ D ∧ linear_equiv G D D₂ → linear_equiv G D₁ D₂
  have h_equiv_D1_D2 : linear_equiv G D₁ D₂ := equiv_rel.trans h_equiv_D1_D h_equiv_D_D2

  -- Apply the q_reduced_unique_class axiom from Basic.lean
  -- Needs: q_reduced G q D₁, q_reduced G q D₂, linear_equiv G D₁ D₂
  exact q_reduced_unique_class G q D₁ D₂ ⟨h_qred_D1, h_qred_D2, h_equiv_D1_D2⟩

/- [Proven] Helper lemma: Every divisor is linearly equivalent to exactly one q-reduced divisor -/
lemma helper_unique_q_reduced (G : CFGraph V) (q : V) (D : CFDiv V) :
  ∃! D' : CFDiv V, linear_equiv G D D' ∧ q_reduced G q D' := by
  -- Prove existence and uniqueness separately
  -- Existence comes from the axiom
  have h_exists : ∃ D' : CFDiv V, linear_equiv G D D' ∧ q_reduced G q D' := by
```

```
    exact exists_q_reduced_representative G q D

  -- Uniqueness comes from the lemma proven above
  have h_unique : ∀ (y₁ y₂ : CFDiv V),
    (linear_equiv G D y₁ ∧ q_reduced G q y₁) →
    (linear_equiv G D y₂ ∧ q_reduced G q y₂) → y₁ = y₂ := by
    intro y₁ y₂ h₁ h₂
    exact uniqueness_of_q_reduced_representative G q D y₁ y₂ h₁ h₂

  -- Combine existence and uniqueness using the standard constructor
  exact exists_unique_of_exists_of_unique h_exists h_unique

/-- Axiom: The q-reduced representative of an effective divisor is effective.
    This follows from the fact that the reduction process (like Dhar's algorithm or repeated
    legal firings) preserves effectiveness when starting with an effective divisor.
    This was especially hard to prove in Lean4, so we are leaving it as an axiom for the time being. -/
axiom helper_q_reduced_of_effective_is_effective (G : CFGraph V) (q : V) (E E' : CFDiv V) :
  effective E → linear_equiv G E E' → q_reduced G q E' → effective E'





/-
# Helpers for Lemma 4.1.10
-/

/-- Axiom: A non-empty graph with an acyclic orientation must have at least one source.
    Proving this inductively is a bit tricky at the moment, and we ran into infinite recursive loop,
    thus we are declaring this as an axiom for now. -/
axiom helper_acyclic_has_source (G : CFGraph V) (O : Orientation G) :
  is_acyclic G O → ∃ v : V, is_source G O v

/-- [Proven] Helper theorem: Two orientations are equal if they have the same directed edges -/
theorem helper_orientation_eq_of_directed_edges {G : CFGraph V}
  (O O' : Orientation G) :
  O.directed_edges = O'.directed_edges → O = O' := by
  intro h
  -- Use cases to construct the equality proof
  cases O with | mk edges consistent =>
  cases O' with | mk edges' consistent' =>
  -- Create congr_arg to show fields are equal
  congr

/-- Axiom: Given a list of disjoint vertex sets that form a partition of V,
```

```
    this axiom states that an acyclic orientation is uniquely determined
    by this partition where each set contains vertices with same indegree.
    Proving this inductively is a bit tricky at the moment, and we ran into infinite recursive loop,
    thus we are declaring this as an axiom for now. -/
axiom helper_orientation_determined_by_levels {G : CFGraph V}
  (O O' : Orientation G) :
  is_acyclic G O → is_acyclic G O' →
  (∀ v : V, indeg G O v = indeg G O' v) →
  O = O'





/-
# Helpers for Proposition 4.1.11
-/

/- Axiom: Defining a reusable block for a configuration from an acyclic orientation with source q being superstable
          Only to be used to define a superstable configuration from an acyclic orientation with source q as a Prop.
   This was especially hard to prove in Lean4, so we are leaving it as an axiom for now.
-/
axiom helper_orientation_config_superstable (G : CFGraph V) (O : Orientation G) (q : V)
    (h_acyc : is_acyclic G O) (h_unique_source : ∀ w, is_source G O w → w = q) :
    superstable G q (orientation_to_config G O q h_acyc h_unique_source)

/- Axiom: Defining a reusable block for a configuration from an acyclic orientation with source q being maximal superstable
          Only to be used to define a maximal superstable configuration from an acyclic orientation with source q as a Prop.
   This was especially hard to prove in Lean4, so we are leaving it as an axiom for now.
-/
axiom helper_orientation_config_maximal (G : CFGraph V) (O : Orientation G) (q : V)
    (h_acyc : is_acyclic G O) (h_unique_source : ∀ w, is_source G O w → w = q) :
    maximal_superstable G (orientation_to_config G O q h_acyc h_unique_source)

/-- [Proven] Helper lemma: Orientation to config preserves indegrees -/
lemma orientation_to_config_indeg (G : CFGraph V) (O : Orientation G) (q : V)
    (h_acyclic : is_acyclic G O) (h_unique_source : ∀ w, is_source G O w → w = q) (v : V) :
    (orientation_to_config G O q h_acyclic h_unique_source).vertex_degree v =
    if v = q then 0 else (indeg G O v : ℤ) - 1 := by
  -- This follows directly from the definition of config_of_source
  simp only [orientation_to_config] at *
  -- Use the definition of config_of_source
```

```
  exact rfl

/-- [Proven] Helper lemma: Two acyclic orientations with same indegrees are equal -/
lemma orientation_unique_by_indeg {G : CFGraph V} (O₁ O₂ : Orientation G)
    (h_acyc₁ : is_acyclic G O₁) (h_acyc₂ : is_acyclic G O₂)
    (h_indeg : ∀ v : V, indeg G O₁ v = indeg G O₂ v) : O₁ = O₂ := by
  -- Apply the helper statement directly since we have exactly matching hypotheses
  exact helper_orientation_determined_by_levels O₁ O₂ h_acyc₁ h_acyc₂ h_indeg

/-- [Proven] Helper lemma to show indegree of source is 0 -/
lemma source_indeg_zero {G : CFGraph V} (O : Orientation G) (v : V)
    (h_src : is_source G O v) : indeg G O v = 0 := by
  -- By definition of is_source in terms of indeg
  unfold is_source at h_src
  -- Convert from boolean equality to proposition
  exact of_decide_eq_true h_src

/-- [Proven] Helper theorem proving uniqueness of orientations giving same config -/
theorem helper_config_to_orientation_unique (G : CFGraph V) (q : V)
    (c : Config V q)
    (h_super : superstable G q c)
    (h_max : maximal_superstable G c)
    (O₁ O₂ : Orientation G)
    (h_acyc₁ : is_acyclic G O₁)
    (h_acyc₂ : is_acyclic G O₂)
    (h_src₁ : is_source G O₁ q)
    (h_src₂ : is_source G O₂ q)
    (h_unique_source₁ : ∀ w, is_source G O₁ w → w = q)
    (h_unique_source₂ : ∀ w, is_source G O₂ w → w = q)
    (h_eq₁ : orientation_to_config G O₁ q h_acyc₁ h_unique_source₁ = c)
    (h_eq₂ : orientation_to_config G O₂ q h_acyc₂ h_unique_source₂ = c) :
    O₁ = O₂ := by
  apply orientation_unique_by_indeg O₁ O₂ h_acyc₁ h_acyc₂
  intro v

  have h_deg₁ := orientation_to_config_indeg G O₁ q h_acyc₁ h_unique_source₁ v
  have h_deg₂ := orientation_to_config_indeg G O₂ q h_acyc₂ h_unique_source₂ v

  have h_config_eq : (orientation_to_config G O₁ q h_acyc₁
    h_unique_source₁).vertex_degree v =
                     (orientation_to_config G O₂ q h_acyc₂
    h_unique_source₂).vertex_degree v := by
    rw [h_eq₁, h_eq₂]

  by_cases hv : v = q
  · -- Case v = q: Both vertices are sources, so indegree is 0
    rw [hv]
    -- Use the explicit source assumptions h_src₁ and h_src₂
    have h_zero₁ := source_indeg_zero O₁ q h_src₁
    have h_zero₂ := source_indeg_zero O₂ q h_src₂
    rw [h_zero₁, h_zero₂]
```

```
  · -- Case v ≠ q: use vertex degree equality
    rw [h_deg₁, h_deg₂] at h_config_eq
    simp only [if_neg hv] at h_config_eq
    -- From config degrees being equal, show indegrees are equal
    have h := congr_arg (fun x => x + 1) h_config_eq
    simp only [sub_add_cancel] at h
    -- Use nat cast injection
    exact (Nat.cast_inj.mp h)

/-- [Proven] Helper lemma to convert between configuration equality forms -/
lemma helper_config_eq_of_subtype_eq {G : CFGraph V} {q : V}
    {O₁ O₂ : {O : Orientation G // is_acyclic G O ∧ (∀ w, is_source G O w → w = q)}}
    (h : orientation_to_config G O₁.val q O₁.prop.1 O₁.prop.2 =
         orientation_to_config G O₂.val q O₂.prop.1 O₂.prop.2) :
    orientation_to_config G O₂.val q O₂.prop.1 O₂.prop.2 =
    orientation_to_config G O₁.val q O₁.prop.1 O₁.prop.2 := by
  exact h.symm

/-- Axiom: Every superstable configuration extends to a maximal superstable configuration
    This was especially hard to prove in Lean4, so we are leaving it as an axiom for now. -/
axiom helper_maximal_superstable_exists (G : CFGraph V) (q : V) (c : Config V q)
    (h_super : superstable G q c) :
    ∃ c' : Config V q, maximal_superstable G c' ∧ config_ge c' c

/-- Axiom: Every maximal superstable configuration comes from an acyclic orientation
    This was especially hard to prove in Lean4, so we are leaving it as an axiom for now. -/
axiom helper_maximal_superstable_orientation (G : CFGraph V) (q : V) (c : Config V q)
    (h_max : maximal_superstable G c) :
    ∃ (O : Orientation G) (h_acyc : is_acyclic G O) (h_unique_source : ∀ w, is_source G O w → w = q),
      orientation_to_config G O q h_acyc h_unique_source = c





/-
# Helpers for Corollary 4.2.2
-/

/-- Axiom: A divisor can be decomposed into parts of specific degrees
    This was especially hard to prove in Lean4, so we are leaving it as an axiom for now. -/
axiom helper_divisor_decomposition (G : CFGraph V) (E'' : CFDiv V) (k₁ k₂ : ℕ)
  (h_effective : effective E'') (h_deg : deg E'' = k₁ + k₂) :
  ∃ (E₁ E₂ : CFDiv V),
    effective E₁ ∧ effective E₂ ∧
    deg E₁ = k₁ ∧ deg E₂ = k₂ ∧
```

```
    E'' = λ v => E₁ v + E₂ v

/- [Proven] Helper theorem: Winnability is preserved under addition -/
theorem helper_winnable_add (G : CFGraph V) (D₁ D₂ : CFDiv V) :
  winnable G D₁ → winnable G D₂ → winnable G (λ v => D₁ v + D₂ v) := by
  -- Assume D₁ and D₂ are winnable
  intro h1 h2

  -- Get the effective divisors that D₁ and D₂ are equivalent to
  rcases h1 with ⟨E₁, hE₁_eff, hE₁_equiv⟩
  rcases h2 with ⟨E₂, hE₂_eff, hE₂_equiv⟩

  -- Our goal is to show that D₁ + D₂ is winnable
  -- We'll show E₁ + E₂ is effective and linearly equivalent to D₁ + D₂

  -- Define our candidate effective divisor
  let E := E₁ + E₂

  -- Show E is effective
  have hE_eff : effective E := by
    intro v
    simp [effective] at hE₁_eff hE₂_eff ⊢
    have h1 := hE₁_eff v
    have h2 := hE₂_eff v
    exact add_nonneg h1 h2

  -- Show E is linearly equivalent to D₁ + D₂
  have hE_equiv : linear_equiv G (D₁ + D₂) E := by
    unfold linear_equiv
    -- Show (E₁ + E₂) - (D₁ + D₂) = (E₁ - D₁) + (E₂ - D₂)
    have h : E - (D₁ + D₂) = (E₁ - D₁) + (E₂ - D₂) := by
      funext w
      simp [sub_apply, add_apply]
      -- Expand E = E₁ + E₂
      have h1 : E w = E₁ w + E₂ w := rfl
      rw [h1]
      -- Use ring arithmetic to complete the proof
      ring

    rw [h]
    -- Use the fact that principal divisors form an additive subgroup
    exact AddSubgroup.add_mem _ hE₁_equiv hE₂_equiv

  -- Construct the witness for winnability
  exists E

/- [Alternative-Proof] Helper theorem: Winnability is preserved under addition -/
theorem helper_winnable_add_alternative (G : CFGraph V) (D₁ D₂ : CFDiv V) :
  winnable G D₁ → winnable G D₂ → winnable G (λ v => D₁ v + D₂ v) := by
  -- Introduce the winnability hypotheses
  intros h1 h2
```

```
  -- Unfold winnability definition for D₁ and D₂
  rcases h1 with ⟨E₁, hE₁_eff, hE₁_equiv⟩
  rcases h2 with ⟨E₂, hE₂_eff, hE₂_equiv⟩

  -- Our goal is to find an effective divisor linearly equivalent to D₁ + D₂
  use (E₁ + E₂)

  constructor
  -- Show E₁ + E₂ is effective
  {
    unfold Div_plus -- Note: Div_plus is defined using effective
    unfold effective at *
    intro v
    have h1 := hE₁_eff v
    have h2 := hE₂_eff v
    exact add_nonneg h1 h2
  }

  -- Show E₁ + E₂ is linearly equivalent to D₁ + D₂
  {
    unfold linear_equiv at *

    -- First convert the function to a CFDiv
    let D₁₂ : CFDiv V := (λ v => D₁ v + D₂ v)

    have h : (E₁ + E₂ - D₁₂) = (E₁ - D₁) + (E₂ - D₂) := by
      funext v
      simp [Pi.add_apply, sub_apply]
      ring

    rw [h]
    exact AddSubgroup.add_mem (principal_divisors G) hE₁_equiv hE₂_equiv
  }





/-
# Helpers for Corollary 4.2.3 + Handshaking Theorem
-/

/-- [Proved] Helper lemma: Every divisor can be decomposed into a principal divisor and an effective divisor -/
lemma eq_nil_of_card_eq_zero {α : Type _} {m : Multiset α}
    (h : Multiset.card m = 0) : m = ∅ := by
  induction m using Multiset.induction_on with
  | empty => rfl
  | cons a s ih =>
    simp only [Multiset.card_cons] at h
```

```
    -- card s + 1 = 0 is impossible for natural numbers
    have : ¬(Multiset.card s + 1 = 0) := Nat.succ_ne_zero (Multiset.card s)
    contradiction

/-- [Proven] Helper lemma: In a loopless graph, each edge has distinct endpoints -/
lemma edge_endpoints_distinct (G : CFGraph V) (e : V × V) (he : e ∈ G.edges) :
    e.1 ≠ e.2 := by
  by_contra eq_endpoints
  have : isLoopless G.edges = true := G.loopless
  unfold isLoopless at this
  have zero_loops : Multiset.card (G.edges.filter (λ e' => e'.1 = e'.2)) = 0 := by
    simp only [decide_eq_true_eq] at this
    exact this
  have e_loop_mem : e ∈ Multiset.filter (λ e' => e'.1 = e'.2) G.edges := by
    simp [he, eq_endpoints]
  have positive : 0 < Multiset.card (G.edges.filter (λ e' => e'.1 = e'.2)) := by
    exact Multiset.card_pos_iff_exists_mem.mpr ⟨e, e_loop_mem⟩
  have : Multiset.filter (fun e' => e'.1 = e'.2) G.edges = ∅ :=
    eq_nil_of_card_eq_zero zero_loops
  rw [this] at e_loop_mem
  cases e_loop_mem

/-- [Proven] Helper lemma: Each edge is incident to exactly two vertices -/
lemma edge_incident_vertices_count (G : CFGraph V) (e : V × V) (he : e ∈ G.edges) :
    (Finset.univ.filter (λ v => e.1 = v ∨ e.2 = v)).card = 2 := by
  rw [Finset.card_eq_two]
  exists e.1
  exists e.2
  constructor
  · exact edge_endpoints_distinct G e he
  · ext v
    simp only [Finset.mem_filter, Finset.mem_univ, true_and,
               Finset.mem_insert, Finset.mem_singleton]
    -- The proof here can be simplified using Iff.intro and cases
    apply Iff.intro
    · intro h_mem_filter -- Goal: v ∈ {e.1, e.2}
      cases h_mem_filter with
      | inl h1 => exact Or.inl (Eq.symm h1)
      | inr h2 => exact Or.inr (Eq.symm h2)
    · intro h_mem_set -- Goal: e.1 = v ∨ e.2 = v
      cases h_mem_set with
      | inl h1 => exact Or.inl (Eq.symm h1)
      | inr h2 => exact Or.inr (Eq.symm h2)

/-- [Proven] Helper lemma: Swapping sum order for incidence checking (Nat version). -/
lemma sum_filter_eq_map_inc_nat (G : CFGraph V) :
  Σ v : V, Multiset.card (G.edges.filter (λ e => e.fst = v ∨ e.snd = v))
    = Multiset.sum (G.edges.map (λ e => (Finset.univ.filter (λ v => e.1 = v ∨ e.2 =
    v)).card)) := by
  -- Define P and g using Prop for clarity in the proof - Available throughout
  let P : V → V × V → Prop := fun v e => e.fst = v ∨ e.snd = v
```

```
  let g : V × V → ℕ := fun e => (Finset.univ.filter (P · e)).card

  -- Rewrite the goal using P and g for proof readability
  suffices goal_rewritten : ∑ v : V, Multiset.card (G.edges.filter (P v)) = Multiset.sum (G.edges.map g) by
    exact goal_rewritten -- The goal is now exactly the statement `goal_rewritten`

  -- Prove the rewritten goal by induction on the multiset G.edges
  induction G.edges using Multiset.induction_on with
  -- Base case: s = ∅
  | empty =>
    simp only [Multiset.filter_zero, Multiset.card_zero, Finset.sum_const_zero,
               Multiset.map_zero, Multiset.sum_zero] -- Use _zero lemmas
  -- Inductive step: Assume holds for s, prove for a :: s
  | cons a s ih =>
    -- Rewrite RHS: sum(map(g, a::s)) = g a + sum(map(g, s))
    rw [Multiset.map_cons, Multiset.sum_cons]

    -- Rewrite LHS: ∑ v, card(filter(P v, a::s))
    -- card(filter) -> countP
    simp_rw [← Multiset.countP_eq_card_filter]

    -- Use countP_cons _ a s inside the sum. Assumes it simplifies
    -- to the form ∑ v, (countP (P v) s + ite (P v a) 1 0)
    simp only [Multiset.countP_cons]

    -- Distribute the sum
    rw [Finset.sum_add_distrib]

    -- Simplify the second sum (∑ v, ite (P v a) 1 0) to g a
    have h_sum_ite_eq_card : ∑ v : V, ite (P v a) 1 0 = g a := by
      -- Use Finset.card_filter: (s.filter p).card = ∑ x ∈ s, if p x then 1 else 0
      rw [← Finset.card_filter]
      -- Should hold by definition of sum over Fintype and definition of g
    rw [h_sum_ite_eq_card] -- Goal: ∑ v, countP (P v) s + g a = g a + sum (map g s)

    -- Rewrite the first sum's countP back to card(filter)
    simp_rw [Multiset.countP_eq_card_filter] -- Goal: ∑ v, card(filter (P v) s) + g a = g a + ...

    -- Apply IH and finish
    rw [add_comm] -- Goal: g a + ∑ v, card(filter (P v) s) = g a + ...
    rw [ih] -- Apply inductive hypothesis



/-- [Proven] Helper lemma: Summing mapped incidence counts equals summing constant 2 (Nat version). -/
lemma map_inc_eq_map_two_nat (G : CFGraph V) :
  Multiset.sum (G.edges.map (λ e => (Finset.univ.filter (λ v => e.1 = v ∨ e.2 = v)).card))
```

```
    = 2 * (Multiset.card G.edges) := by
  -- Define the function being mapped
  let f : V × V → ℕ := λ e => (Finset.univ.filter (λ v => e.1 = v ∨ e.2 = v)).card
  -- Define the constant function 2
  let g (_ : V × V) : ℕ := 2
  -- Show f equals g for all edges in G.edges
  have h_congr : ∀ e ∈ G.edges, f e = g e := by
    intro e he
    simp [f, g]
    exact edge_incident_vertices_count G e he
  -- Apply congruence to the map function itself first using map_congr with rfl
  rw [Multiset.map_congr rfl h_congr] -- Use map_congr with rfl
  -- Apply rewrites step-by-step
  rw [Multiset.map_const', Multiset.sum_replicate, Nat.nsmul_eq_mul, Nat.mul_comm]

/--
**Handshaking Theorem:** [Proven] In a loopless multigraph \(G\),
the sum of the degrees of all vertices is twice the number of edges:

\[
  \sum_{v \in V} \deg(v) = 2 \cdot \#(\text{edges of }G).
\]
-/
theorem helper_sum_vertex_degrees (G : CFGraph V) :
    Σ v, vertex_degree G v = 2 * ↑(Multiset.card G.edges) := by
  -- Unfold vertex degree definition
  unfold vertex_degree
  calc
    -- Start with the definition of sum of vertex degrees
    Σ v, vertex_degree G v
    -- Express vertex degree as Nat cast of card filter
    = Σ v, ↑(Multiset.card (G.edges.filter (λ e => e.1 = v ∨ e.2 = v))) := by rfl
    -- Pull the Nat cast outside the sum over vertices
    _ = ↑(Σ v, Multiset.card (G.edges.filter (λ e => e.1 = v ∨ e.2 = v))) := by rw [Nat.cast_sum]
    -- Apply the sum swapping lemma (Nat version)
    _ = ↑(Multiset.sum (G.edges.map (λ e => (Finset.univ.filter (λ v => e.1 = v ∨ e.2 = v)).card))) := by
      rw [sum_filter_eq_map_inc_nat G]
    -- Apply the lemma relating sum of incidences to 2 * |E| (Nat version)
    _ = ↑(2 * (Multiset.card G.edges)) := by
      rw [map_inc_eq_map_two_nat G]
    -- Pull the constant 2 outside the Nat cast
    _ = 2 * ↑(Multiset.card G.edges) := by
      rw [Nat.cast_mul, Nat.cast_ofNat] -- Use Nat.cast_ofNat for Nat.cast 2





/-
```

```
# Helpers for Proposition 4.1.13 Part (1)
-/

/-- Axiom: Correspondence between q-reduced divisors and superstable configurations
    A divisor is q-reduced if and only if it corresponds to a superstable configuration minus q
    This was especially hard to prove in Lean4, so I am leaving it as an axiom for the time being. -/
axiom q_reduced_superstable_correspondence (G : CFGraph V) (q : V) (D : CFDiv V) :
  q_reduced G q D ↔ ∃ c : Config V q, superstable G q c ∧
  D = λ v => c.vertex_degree v - if v = q then 1 else 0

/- Axiom: The degree of a q-reduced divisor is at most g-1.
    Proving this directly requires formalizing Dhar's burning algorithm or deeper results
    relating q-reduced divisors to acyclic orientations, which is beyond the current scope.
    Attempts to prove it here encounter difficulties due to interactions
    between `config_degree` and the value at `q`, or potential definition mismatches.
    Therefore, it remains an axiom for now. -/
axiom lemma_q_reduced_degree_bound (G : CFGraph V) (q : V) (D : CFDiv V) :
  q_reduced G q D → deg D ≤ genus G - 1

/-- Lemma: Superstable configuration degree is bounded by genus -/
lemma helper_superstable_degree_bound (G : CFGraph V) (q : V) (c : Config V q) :
  superstable G q c → config_degree c ≤ genus G := by
  intro h_super

  -- Define c₀ such that c₀(q) = 0 and c₀(v) = c(v) for v ≠ q.
  let c₀_deg_func := λ v => c.vertex_degree v - if v = q then c.vertex_degree q else 0
  have h_c₀_nonneg_except_q : ∀ v : V, v ≠ q → c₀_deg_func v ≥ 0 := by
    intro v hv
    simp [c₀_deg_func, hv] -- Simplify using v ≠ q
    exact c.non_negative_except_q v hv -- Use original property of c
  let c₀ := Config.mk c₀_deg_func h_c₀_nonneg_except_q

  -- Show c₀ is superstable if c is.
  have h_super₀ : superstable G q c₀ := by
    -- Unfold superstability for c₀
    unfold superstable at *
    intro S hS_subset hS_nonempty
    -- Use the fact that c is superstable
    rcases h_super S hS_subset hS_nonempty with ⟨v, hv_in_S, h_c_lt_outdeg⟩
    -- We need to show ∃ v' ∈ S, c₀.vertex_degree v' < outdeg_S G q S v'
    use v -- Use the same vertex v
    constructor
    · exact hv_in_S
    · -- Show c₀.vertex_degree v = c.vertex_degree v since v ∈ S implies v ≠ q
      have hv_ne_q : v ≠ q := by
        have h_v_in_V_minus_q := Finset.mem_filter.mp (hS_subset hv_in_S) -- Corrected parenthesis
```

```
     exact h_v_in_V_minus_q.right -- Extract the second part (v ≠ q)
    -- First show c₀_deg_func v = c.vertex_degree v
    have h_c0v_eq_cv : c₀_deg_func v = c.vertex_degree v := by simp [c₀_deg_func, hv_ne_q]
    -- Rewrite the goal using this equality
    simp [c₀] -- Unfold c₀ in the goal
    rw [h_c0v_eq_cv]
    -- The goal is now c.vertex_degree v < outdeg_S G q S v, which is h_c_lt_outdeg
    exact h_c_lt_outdeg

  -- Show config_degree c₀ = config_degree c.
  have h_config_deg_eq : config_degree c₀ = config_degree c := by
   unfold config_degree
   apply Finset.sum_congr rfl
   intro v hv_mem
   -- hv_mem implies v is in the filter {x | x ≠ q}
   have hv_ne_q : v ≠ q := by exact Finset.mem_filter.mp hv_mem |>.right
   simp [c₀_deg_func, hv_ne_q] -- Prove equality pointwise

  -- Define D' based on c₀ (which has c₀(q) = 0).
  let D' := λ v => c₀.vertex_degree v - if v = q then 1 else 0

  -- Show D' is q-reduced using the correspondence axiom.
  have h_D'_q_reduced : q_reduced G q D' := by
   apply (q_reduced_superstable_correspondence G q D').mpr
   -- Provide c₀ as the witness
   use c₀

  -- Apply the degree bound axiom for q-reduced divisors.
  have h_deg_D'_bound : deg D' ≤ genus G - 1 := by
   exact lemma_q_reduced_degree_bound G q D' h_D'_q_reduced

  -- Calculate the degree of D'.
  have h_deg_D'_calc : deg D' = config_degree c₀ - 1 := by
   calc
    deg D' = Σ v, D' v := rfl
    _ = (Σ v in (Finset.univ.filter (λ x => x ≠ q)), D' v) + D' q := by
      rw [← Finset.sum_filter_add_sum_filter_not (s := Finset.univ) (p := λ v' => v' ≠ q)]
      simp [Finset.filter_eq']
    _ = (Σ v in (Finset.univ.filter (λ x => x ≠ q)), (c₀.vertex_degree v - if v = q then 1 else 0)) +
      (c₀.vertex_degree q - if q = q then 1 else 0) := rfl
    _ = (Σ v in (Finset.univ.filter (λ x => x ≠ q)), c₀.vertex_degree v) +
      (c₀.vertex_degree q - 1) := by simp [Finset.sum_sub_distrib] -- Note: simp removes the 'if v=q then 1 else 0' part correctly
    _ = config_degree c₀ + (c₀.vertex_degree q - 1) := by rw [config_degree]
    -- Show c₀(q) = 0
    _ = config_degree c₀ + (0 - 1) := by
      have h_c₀_q_zero : c₀.vertex_degree q = 0 := by simp [c₀, c₀_deg_func]
      rw [h_c₀_q_zero]
```

```
      _ = config_degree c₀ - 1 := by ring

  -- Combine the bound and calculation.
  have h_ineq := h_deg_D'_bound
  rw [h_deg_D'_calc] at h_ineq -- Substitute calculated degree into bound
  -- h_ineq is now: config_degree c₀ - 1 ≤ genus G - 1

  -- Use linearity of ≤ over addition to get config_degree c₀ ≤ genus G
  have h_config_deg_c₀_bound : config_degree c₀ ≤ genus G := by linarith [h_ineq]

  -- Substitute back config_degree c.
  rw [← h_config_deg_eq] -- Rewrite goal using symmetry
  exact h_config_deg_c₀_bound

/-- Axiom: Every maximal superstable configuration has degree at least g
    This was especially hard to prove in Lean4, so I am leaving it as an axiom for the time being. -/
axiom helper_maximal_superstable_degree_lower_bound (G : CFGraph V) (q : V) (c : Config V q) :
  superstable G q c → maximal_superstable G c → config_degree c ≥ genus G

/-- Axiom: If a superstable configuration has degree equal to g, it is maximal
    This was especially hard to prove in Lean4, so I am leaving it as an axiom for the time being. -/
axiom helper_degree_g_implies_maximal (G : CFGraph V) (q : V) (c : Config V q) :
  superstable G q c → config_degree c = genus G → maximal_superstable G c





/-
# Helpers for Proposition 4.1.13 Part (2)
-/

/-- Axiom: Superstabilization of configuration with degree g+1 sends chip to q
    This was especially hard to prove in Lean4, so I am leaving it as an axiom for the time being. -/
axiom helper_superstabilize_sends_to_q (G : CFGraph V) (q : V) (c : Config V q) :
  maximal_superstable G c → config_degree c = genus G →
  ∀ v : V, v ≠ q → winnable G (λ w => c.vertex_degree w + if w = v then 1 else 0 - if w = q then 1 else 0)

-- Axiom (Based on Merino's Lemma / Properties of Superstable Configurations):
-- If c and c' are superstable (using the standard definition `superstable`)
-- and c' dominates c pointwise (config_ge c' c), then their difference (c' - c)
-- must be a principal divisor. This is a known result in chip-firing theory.
-- It implies deg(c') = deg(c) because non-zero principal divisors have degree 0.
-- This was especially hard to prove in Lean4, so I am leaving it as an axiom for the time being.
axiom superstable_dominance_implies_principal (G : CFGraph V) (q : V) (c c' : Config
```

```lean
    V q) :
  superstable G q c → superstable G q c' → config_ge c' c →
  (λ v => c'.vertex_degree v - c.vertex_degree v) ∈ principal_divisors G

/-- [Proven] Helper lemma: Difference between dominated configurations
    implies linear equivalence of corresponding q-reduced divisors.

    This proof relies on the standard definition of superstability (`superstable`)
    and an axiom (`superstable_dominance_implies_principal`) stating that the difference
    between dominated standard-superstable configurations is a principal divisor.
-/
lemma helper_q_reduced_linear_equiv_dominates (G : CFGraph V) (q : V) (c c' : Config V q) :
  superstable G q c → superstable G q c' → config_ge c' c →
  linear_equiv G
    (λ v => c.vertex_degree v - if v = q then 1 else 0)
    (λ v => c'.vertex_degree v - if v = q then 1 else 0) := by
  intros h_std_super_c h_std_super_c' h_ge

  -- Goal: Show linear_equiv G D₁ D₂
  -- By definition of linear_equiv, this means D₂ - D₁ ∈ principal_divisors G
  unfold linear_equiv -- Explicitly unfold the definition

  -- Prove the difference D₂ - D₁ equals c' - c pointwise
  have h_diff : (λ v => c'.vertex_degree v - if v = q then 1 else 0) - (λ v => c.vertex_degree v - if v = q then 1 else 0) =
                (λ v => c'.vertex_degree v - c.vertex_degree v) := by
    funext v
    rw [sub_apply] -- Explicitly apply pointwise subtraction definition
    -- Goal is now: (c' v - if..) - (c v - if..) = c' v - c v
    by_cases hv : v = q
    · -- Case v = q:
      simp only [hv, if_true] -- Simplify if clauses using v=q
      ring -- Goal is (c' q - 1) - (c q - 1) = c' q - c q
    · -- Case v ≠ q:
      simp only [hv, if_false] -- Simplify if clauses using v≠q
      ring -- Goal is (c' v - 0) - (c v - 0) = c' v - c v

  -- Rewrite the goal using the calculated difference D₂ - D₁ = c' - c
  rw [h_diff]

  -- Apply the axiom `superstable_dominance_implies_principal`.
  -- This axiom states that if c and c' are standard-superstable and c' dominates c,
  -- then their difference (c' - c) is indeed a principal divisor.
  exact superstable_dominance_implies_principal G q c c' h_std_super_c h_std_super_c' h_ge

/-- [Proven] Helper theorem: Linear equivalence preserves winnability -/
theorem helper_linear_equiv_preserves_winnability (G : CFGraph V) (D₁ D₂ : CFDiv V) :
  linear_equiv G D₁ D₂ → (winnable G D₁ ↔ winnable G D₂) := by
```

```
  intro h_equiv
  constructor
  -- Forward direction: D₁ winnable → D₂ winnable
  { intro h_win₁
    rcases h_win₁ with ⟨D₁', h_eff₁, h_equiv₁⟩
    exists D₁'
    constructor
    · exact h_eff₁
    · -- Use transitivity: D₂ ~ D₁ ~ D₁'
      exact linear_equiv_is_equivalence G |>.trans
        (linear_equiv_is_equivalence G |>.symm h_equiv) h_equiv₁ }
  -- Reverse direction: D₂ winnable → D₁ winnable
  { intro h_win₂
    rcases h_win₂ with ⟨D₂', h_eff₂, h_equiv₂⟩
    exists D₂'
    constructor
    · exact h_eff₂
    · -- Use transitivity: D₁ ~ D₂ ~ D₂'
      exact linear_equiv_is_equivalence G |>.trans h_equiv h_equiv₂ }

/-- Axiom: Existence of elements in finite types
    This is a technical axiom used to carry forward existence arguments we frequently use
    such as the fact that finite graphs have vertices. This axiom
    captures this in a way that can be used in formal lean4 proofs. -/
axiom Fintype.exists_elem (V : Type) [Fintype V] : ∃ x : V, True



/-
# Helpers for Proposition 4.1.14
-/

/-- [Proven] Helper lemma: Source vertices have equal indegree (zero) when v = q -/
lemma helper_source_indeg_eq_at_q {V : Type} [DecidableEq V] [Fintype V]
    (G : CFGraph V) (O₁ O₂ : Orientation G) (q v : V)
    (h_src₁ : is_source G O₁ q = true) (h_src₂ : is_source G O₂ q = true)
    (hv : v = q) :
    indeg G O₁ v = indeg G O₂ v := by
  rw [hv]
  rw [source_indeg_zero O₁ q h_src₁]
  rw [source_indeg_zero O₂ q h_src₂]





/-
# Helpers for Rank Degree Inequality used in RRG
-/

```

```lean
/-- Axiom: Dhar's algorithm produces q-reduced divisor from any divisor
    Given any divisor D, Dhar's algorithm produces a unique q-reduced divisor that is
    linearly equivalent to D. The algorithm outputs both a superstable configuration c
    and an integer k, where k represents the number of chips at q. This is a key result
    from Dhar (1990) proven in detail in Chapter 3 of Corry & Perkinson's "Divisors and
    Sandpiles" (AMS, 2018) -/
axiom helper_dhar_algorithm {V : Type} [DecidableEq V] [Fintype V] (G : CFGraph V) (q : V) (D : CFDiv V) :
  ∃ (c : Config V q) (k : ℤ),
    linear_equiv G D (λ v => c.vertex_degree v + (if v = q then k else 0)) ∧
    superstable G q c

/-- Axiom: Dhar's algorithm produces negative k for unwinnable divisors
    When applied to an unwinnable divisor D, Dhar's algorithm must produce a
    negative value for k (the number of chips at q). This is a crucial fact used
    in characterizing unwinnable divisors, proven in chapter 4 of Corry & Perkinson's
    "Divisors and Sandpiles" (AMS, 2018). The negativity of k is essential for
    showing the relationship between unwinnable divisors and q-reduced forms. -/
axiom helper_dhar_negative_k {V : Type} [DecidableEq V] [Fintype V] (G : CFGraph V) (q : V) (D : CFDiv V) :
  ¬(winnable G D) →
  ∀ (c : Config V q) (k : ℤ),
    linear_equiv G D (λ v => c.vertex_degree v + (if v = q then k else 0)) →
    superstable G q c →
    k < 0

/-- Axiom: Given a graph G and a vertex q, there exists a maximal superstable divisor
    c' that is greater than or equal to any superstable divisor c. This is a key
    result from Corry & Perkinson's "Divisors and Sandpiles" (AMS, 2018) that is
    used in proving the Riemann-Roch theorem for graphs.
    This was especially hard to prove in Lean4, so I am leaving it as an axiom for the time being. -/
axiom helper_superstable_to_unwinnable (G : CFGraph V) (q : V) (c : Config V q) :
  maximal_superstable G c →
  ¬winnable G (λ v => c.vertex_degree v - if v = q then 1 else 0)

/-- Axiom: Rank and degree bounds for canonical divisor
    This was especially hard to prove in Lean4, so I am leaving it as an axiom for the time being. -/
axiom helper_rank_deg_canonical_bound (G : CFGraph V) (q : V) (D : CFDiv V) (E H : CFDiv V) (c' : Config V q) :
  linear_equiv G (λ v => c'.vertex_degree v - if v = q then 1 else 0) (λ v => D v - E v + H v) →
  rank G (λ v => canonical_divisor G v - D v) + deg D - deg E + deg H ≤ rank G D

/-- Axiom: Degree of H relates to graph parameters when H comes from maximal superstable configs
    This was especially hard to prove in Lean4, so I am leaving it as an axiom for the time being. -/
```

```
axiom helper_H_degree_bound (G : CFGraph V) (q : V) (D : CFDiv V) (H : CFDiv V) (k : ℤ) (c : Config V q) (c' : Config V q) :
  effective H →
  H = (λ v => if v = q then -(k + 1) else c'.vertex_degree v - c.vertex_degree v) →
  rank G D + 1 - (Multiset.card G.edges - Fintype.card V + 1) < deg H

/-- Axiom: Linear equivalence between DO and D-E+H
    This was especially hard to prove in Lean4, so I am leaving it as an axiom for the time being. -/
axiom helper_DO_linear_equiv (G : CFGraph V) (q : V) (D E H : CFDiv V) (c' : Config V q) :
  linear_equiv G (λ v => c'.vertex_degree v - if v = q then 1 else 0)
                (λ v => D v - E v + H v)

/-- Axiom: Adding a chip anywhere to c'-q makes it winnable when c' is maximal superstable
    This was especially hard to prove in Lean4, so I am leaving it as an axiom for the time being. -/
axiom helper_maximal_superstable_chip_winnable_exact (G : CFGraph V) (q : V) (c' : Config V q) :
  maximal_superstable G c' →
  ∀ (v : V), winnable G (λ w => (λ v => c'.vertex_degree v - if v = q then 1 else 0) w + if w = v then 1 else 0)





/-
# Helpers for RRG's Corollary 4.4.1
-/

/-- Axiom: Rank decreases in K-D recursion for maximal unwinnable divisors
    This captures that when we apply canonical_divisor - D to a maximal unwinnable divisor,
    the rank measure decreases. This is used for termination of maximal_unwinnable_symmetry.
    This was especially hard to SETTLE in Lean4, so I am leaving it as an axiom for the time being. -/
axiom rank_decreases_for_KD {V : Type} [DecidableEq V] [Fintype V]
  (G : CFGraph V) (D : CFDiv V) :
  maximal_unwinnable G (λ v => canonical_divisor G v - D v) →
  ((rank G (λ v => canonical_divisor G v - D v) + 1).toNat < (rank G D + 1).toNat)





/-
# Helpers for RRG's Corollary 4.4.3
-/
```

```
/-- [Proven] Effective divisors have non-negative degree -/
lemma effective_nonneg_deg {V : Type} [DecidableEq V] [Fintype V]
  (D : CFDiv V) (h : effective D) : deg D ≥ 0 := by
  -- Definition of effective means all entries are non-negative
  unfold effective at h
  -- Definition of degree as sum of entries
  unfold deg
  -- Non-negative sum of non-negative numbers is non-negative
  exact sum_nonneg (λ v _ ↦ h v)
```

A.7 Intermediate Results for Riemann-Roch for Graphs (RRGHelpers.lean)

```
import ChipFiringWithLean.Basic
import ChipFiringWithLean.Config
import ChipFiringWithLean.Orientation
import ChipFiringWithLean.Rank
import ChipFiringWithLean.Helpers
import Mathlib.Algebra.Ring.Int
import Paperproof

set_option linter.unusedVariables false
set_option trace.split.failure true

open Multiset Finset

-- Assume V is a finite type with decidable equality
variable {V : Type} [DecidableEq V] [Fintype V]

-- [Proven] Lemma: effectiveness is preserved under legal firing (Additional)
lemma legal_firing_preserves_effective (G : CFGraph V) (D : CFDiv V) (S : Finset V) :
  legal_set_firing G D S → effective D → effective (set_firing G D S) := by
  intros h_legal h_eff v
  simp [set_firing]
  by_cases hv : v ∈ S
  -- Case 1: v ∈ S
  · exact h_legal v hv
  -- Case 2: v ∉ S
  · have h1 : D v ≥ 0 := h_eff v
    have h2 : finset_sum S (λ v' => if v = v' then -vertex_degree G v' else num_edges
    G v' v) ≥ 0 := by
      apply Finset.sum_nonneg
      intro x hx
      -- Split on whether v = x
      by_cases hveq : v = x
      · -- If v = x, contradiction with v ∉ S
        rw [hveq] at hv
        contradiction
      · -- If v ≠ x, then we get num_edges which is non-negative
        simp [hveq]
```

```
    linarith

-- [Proven] Proposition 3.2.4: q-reduced and effective implies winnable
theorem winnable_iff_q_reduced_effective (G : CFGraph V) (q : V) (D : CFDiv V) :
  winnable G D ↔ ∃ D' : CFDiv V, linear_equiv G D D' ∧ q_reduced G q D' ∧ effective D' := by
  constructor
  { -- Forward direction
    intro h_win
    rcases h_win with ⟨E, h_eff, h_equiv⟩
    rcases helper_unique_q_reduced G q D with ⟨D', h_D'⟩
    use D'
    constructor
    · exact h_D'.1.1  -- D is linearly equivalent to D'
    constructor
    · exact h_D'.1.2  -- D' is q-reduced
    · -- Show D' is effective using:
      -- First get E ~ D' by transitivity through D
      have h_equiv_symm : linear_equiv G E D := (linear_equiv_is_equivalence G).symm h_equiv -- E ~ D
      have h_equiv_E_D' : linear_equiv G E D' := (linear_equiv_is_equivalence G).trans h_equiv_symm h_D'.1.1 -- E ~ D ~ D' => E ~ D'
      -- Now use the axiom that q-reduced form of an effective divisor is effective
      exact helper_q_reduced_of_effective_is_effective G q E D' h_eff h_equiv_E_D' h_D'.1.2
  }
  { -- Reverse direction
    intro h
    rcases h with ⟨D', h_equiv, h_qred, h_eff⟩
    use D'
    exact ⟨h_eff, h_equiv⟩
  }

-- [Proven] Proposition 3.2.4 (Extension): q-reduced and effective implies winnable
theorem q_reduced_effective_implies_winnable (G : CFGraph V) (q : V) (D : CFDiv V) :
  q_reduced G q D → effective D → winnable G D := by
  intros h_qred h_eff
  -- Apply right direction of iff
  rw [winnable_iff_q_reduced_effective]
  -- Prove existence
  use D
  constructor
  · exact (linear_equiv_is_equivalence G).refl D  -- D is linearly equivalent to itself using proven reflexivity
  constructor
  · exact h_qred  -- D is q-reduced
  · exact h_eff   -- D is effective

/-- [Proven] Lemma 4.1.10: An acyclic orientation is uniquely determined by its indegree sequence -/
theorem acyclic_orientation_unique_by_indeg {G : CFGraph V}
```

```
  (O O' : Orientation G)
  (h_acyclic : is_acyclic G O)
  (h_acyclic' : is_acyclic G O')
  (h_indeg : ∀ v : V, indeg G O v = indeg G O' v) :
  O = O' := by
  -- Apply the helper_orientation_determined_by_levels axiom directly
  exact helper_orientation_determined_by_levels O O' h_acyclic h_acyclic' h_indeg

/-- [Proven] Lemma 4.1.10 (Alternative Form): Two acyclic orientations with same indegree sequences are equal -/
theorem acyclic_equal_of_same_indeg {G : CFGraph V} (O O' : Orientation G)
    (h_acyclic : is_acyclic G O) (h_acyclic' : is_acyclic G O')
    (h_indeg : ∀ v : V, indeg G O v = indeg G O' v) :
    O = O' := by
  -- Use previously defined theorem about uniqueness by indegree
  exact acyclic_orientation_unique_by_indeg O O' h_acyclic h_acyclic' h_indeg

/-- [Proven] Proposition 4.1.11: Bijection between acyclic orientations with source q
    and maximal superstable configurations -/
theorem stable_bijection (G : CFGraph V) (q : V) :
    let α := {O : Orientation G // is_acyclic G O ∧ (∀ w, is_source G O w → w = q)};
    let β := {c : Config V q // maximal_superstable G c};
    let f_raw : α → Config V q := λ O_sub => orientation_to_config G O_sub.val q O_sub.prop.1 O_sub.prop.2;
    let f : α → β := λ O_sub => ⟨f_raw O_sub, helper_orientation_config_maximal G O_sub.val q O_sub.prop.1 O_sub.prop.2⟩;
    Function.Bijective f := by
  -- Define the domain and codomain types explicitly (can be removed if using let like above)
  let α := {O : Orientation G // is_acyclic G O ∧ (∀ w, is_source G O w → w = q)}
  let β := {c : Config V q // maximal_superstable G c}
  -- Define the function f_raw : α → Config V q
  let f_raw : α → Config V q := λ O_sub => orientation_to_config G O_sub.val q O_sub.prop.1 O_sub.prop.2
  -- Define the function f : α → β, showing the result is maximal superstable
  let f : α → β := λ O_sub =>
    ⟨f_raw O_sub, helper_orientation_config_maximal G O_sub.val q O_sub.prop.1 O_sub.prop.2⟩

  constructor
  -- Injectivity
  { -- Prove injective f using injective f_raw
    intros O₁_sub O₂_sub h_f_eq -- h_f_eq : f O₁_sub = f O₂_sub
    have h_f_raw_eq : f_raw O₁_sub = f_raw O₂_sub := by simp [f] at h_f_eq; exact h_f_eq

    -- Reuse original injectivity proof structure, ensuring types match
    let ⟨O₁, h₁⟩ := O₁_sub
    let ⟨O₂, h₂⟩ := O₂_sub
    -- Define c, h_eq₁, h_eq₂ based on orientation_to_config directly
    let c := orientation_to_config G O₁ q h₁.1 h₁.2
```

```
    have h_eq₁ : orientation_to_config G O₁ q h₁.1 h₁.2 = c := rfl
    have h_eq₂ : orientation_to_config G O₂ q h₂.1 h₂.2 = c := by
      exact h_f_raw_eq.symm.trans h_eq₁ -- Use transitivity

    have h_src₁ : is_source G O₁ q := by
      rcases helper_acyclic_has_source G O₁ h₁.1 with ⟨s, hs⟩; have h_s_eq_q : s = q := h₁.2 s hs; rwa [h_s_eq_q] at hs
    have h_src₂ : is_source G O₂ q := by
      rcases helper_acyclic_has_source G O₂ h₂.1 with ⟨s, hs⟩; have h_s_eq_q : s = q := h₂.2 s hs; rwa [h_s_eq_q] at hs

    -- Define h_super and h_max in terms of c
    have h_super : superstable G q c := by
      rw [← h_eq₁]; exact helper_orientation_config_superstable G O₁ q h₁.1 h₁.2
    have h_max   : maximal_superstable G c := by
      rw [← h_eq₁]; exact helper_orientation_config_maximal G O₁ q h₁.1 h₁.2

    apply Subtype.eq
    -- Call helper_config_to_orientation_unique with the original h_eq₁ and h_eq₂
    exact (helper_config_to_orientation_unique G q c h_super h_max
      O₁ O₂ h₁.1 h₂.1 h_src₁ h_src₂ h₁.2 h₂.2 h_eq₁ h_eq₂)
  }

  -- Surjectivity
  { -- Prove Function.Surjective f
    unfold Function.Surjective
    intro y -- y should now have type β
    -- Access components using .val and .property
    let c_target : Config V q := y.val -- Explicitly type c_target
    let h_target_max_superstable := y.property

    -- Use the axiom that every maximal superstable config comes from an orientation.
    rcases helper_maximal_superstable_orientation G q c_target h_target_max_superstable with
      ⟨O, h_acyc, h_unique_source, h_config_eq_target⟩

    -- Construct the required subtype element x : α (the pre-image)
    let x : α := ⟨O, ⟨h_acyc, h_unique_source⟩⟩

    -- Show that this x exists
    use x

    -- Show f x = y using Subtype.eq
    apply Subtype.eq
    -- Goal: (f x).val = y.val
    -- Need to show: f_raw x = c_target
    -- This is exactly h_config_eq_target
    exact h_config_eq_target
    -- Proof irrelevance handles the equality of the property components.
  }

```

```
/-- [Proven] Proposition 4.1.13 (1): Characterization of maximal superstable configurations by their degree -/
theorem maximal_superstable_config_prop (G : CFGraph V) (q : V) (c : Config V q) :
  superstable G q c → (maximal_superstable G c ↔ config_degree c = genus G) := by
  intro h_super
  constructor
  { -- Forward direction: maximal_superstable → degree = g
    intro h_max
    -- Use degree bound and maximality
    have h_bound := helper_superstable_degree_bound G q c h_super
    have h_geq := helper_maximal_superstable_degree_lower_bound G q c h_super h_max
    -- Combine bounds to get equality
    exact le_antisymm h_bound h_geq }
  { -- Reverse direction: degree = g → maximal_superstable
    intro h_deg
    -- Apply the axiom that degree g implies maximality
    exact helper_degree_g_implies_maximal G q c h_super h_deg }

/-- [Proven] Proposition 4.1.13 (2): Characterization of maximal unwinnable divisors -/
theorem maximal_unwinnable_char (G : CFGraph V) (q : V) (D : CFDiv V) :
  maximal_unwinnable G D ↔
  ∃ c : Config V q, maximal_superstable G c ∧
  ∃ D' : CFDiv V, q_reduced G q D' ∧ linear_equiv G D D' ∧
  D' = λ v => c.vertex_degree v - if v = q then 1 else 0 := by
  constructor
  { -- Forward direction (⇒)
    intro h_max_unwinnable_D -- Assume D is maximal unwinnable
    -- Get the unique q-reduced representative D' for D
    rcases helper_unique_q_reduced G q D with ⟨D', ⟨h_D_equiv_D', h_qred_D'⟩, _⟩
    -- Show D' corresponds to some superstable c
    rcases (q_reduced_superstable_correspondence G q D').mp h_qred_D' with ⟨c, h_super_c, h_form_D'_eq_c⟩

    -- Prove that this c must be maximal superstable
    have h_max_super_c : maximal_superstable G c := by
      -- Prove by contradiction: Assume c is not maximal superstable
      by_contra h_not_max_c
      -- If c is superstable but not maximal, there exists a strictly dominating maximal c'
      rcases helper_maximal_superstable_exists G q c h_super_c with ⟨c', h_max_c', h_ge_c'_c⟩
      -- Define D'' based on the maximal superstable c'
      let D'' := λ v => c'.vertex_degree v - if v = q then (1 : ℤ) else (0 : ℤ)
      -- Show D'' is q-reduced (from correspondence with superstable c')
      have h_qred_D'' := (q_reduced_superstable_correspondence G q D'').mpr ⟨c', h_max_c'.1, rfl⟩

      -- Show D' is linearly equivalent to D''
      have h_D'_equiv_D'' : linear_equiv G D' D'' := by
        -- We know D' = form(c) (h_form_D'_eq_c)
```

```
      -- We know form(c) ~ form(c') = D'' (helper_q_reduced_linear_equiv_dominates)
      rw [h_form_D'_eq_c] -- Replace D' with form(c)
      exact helper_q_reduced_linear_equiv_dominates G q c c' h_super_c h_max_c'.1 h_ge_c'_c

    -- Since D' and D'' are both q-reduced and linearly equivalent, they must be equal
    have h_D'_eq_D'' : D' = D'' := by
      apply q_reduced_unique_class G q D' D''
      -- Provide the triple ⟨q_reduced D', q_reduced D'', D' ~ D''⟩
      exact ⟨h_qred_D', h_qred_D'', h_D'_equiv_D''⟩

    -- Now relate c and c' using D' = D''
    have h_lambda_eq : (λ v => c.vertex_degree v - if v = q then 1 else 0) = (λ v => c'.vertex_degree v - if v = q then 1 else 0) := by
      rw [← h_form_D'_eq_c] -- LHS = D'
      rw [h_D'_eq_D'']      -- Goal: D'' = RHS

    -- This pointwise equality implies c = c'
    have h_c_eq_c' : c = c' := by
      -- Prove equality by showing fields are equal
      cases c; cases c' -- Expose fields vertex_degree and non_negative_except_q
      -- Use simp [mk.injEq] to reduce goal to field equality (proves non_negative_except_q equality automatically)
      simp only [Config.mk.injEq]
      -- Prove vertex_degree functions are equal using h_lambda_eq
      funext v
      have h_pointwise_eq := congr_fun h_lambda_eq v
      -- Use Int.sub_right_inj to simplify the equality
      rw [Int.sub_right_inj] at h_pointwise_eq
      exact h_pointwise_eq -- The hypothesis now matches the goal

    -- Contradiction: helper_maximal_superstable_exists implies c' ≠ c if c wasn't maximal
    have h_c_ne_c' : c ≠ c' := by
      intro hc_eq_c' -- Assume c = c' for contradiction
      -- Rewrite c' to c in the hypothesis h_max_c' using the assumed equality
      rw [← hc_eq_c'] at h_max_c'
      -- h_max_c' now has type: maximal_superstable G c ∧ config_ge c c
      -- This contradicts the initial assumption h_not_max_c (¬ maximal_superstable G c)
      exact h_not_max_c h_max_c' -- Apply h_not_max_c to the full hypothesis

    -- We derived c = c' and c ≠ c', the final contradiction
    exact h_c_ne_c' h_c_eq_c'
  -- End of by_contra proof. We now know maximal_superstable G c holds.

  -- Construct the main existential result for the forward direction
  use c, h_max_super_c -- We found the required maximal superstable c
  use D', h_qred_D', h_D_equiv_D', h_form_D'_eq_c -- Use the q-reduced D' and its properties
```

```
  }
  { -- Reverse direction (⇐)
    intro h_exists -- Assume the existence of c, D' with the given properties
    rcases h_exists with ⟨c, h_max_c, D', h_qred_D', h_D_equiv_D', h_form_D'_eq_c⟩

    -- Goal: Prove maximal_unwinnable G D
    constructor
    { -- Part 1: Show D is unwinnable (¬winnable G D)
      intro h_win_D -- Assume 'winnable G D' for contradiction
      -- Use linear equivalence to transfer winnability from D to D'
      have h_win_D' : winnable G D' :=
        (helper_linear_equiv_preserves_winnability G D D' h_D_equiv_D').mp h_win_D

      -- The divisor form derived from a maximal superstable config is unwinnable
      have h_unwin_form : ¬(winnable G (λ v => c.vertex_degree v - if v = q then 1 else 0)) :=
        helper_superstable_to_unwinnable G q c h_max_c

      -- Since D' equals this form, D' is unwinnable
      have h_unwin_D' : ¬(winnable G D') := by
        rw [h_form_D'_eq_c] -- Rewrite goal to use the form
        exact h_unwin_form -- Apply the unwinnability of the form

      -- Contradiction between h_win_D' and h_unwin_D'
      exact h_unwin_D' h_win_D'
    }
    { -- Part 2: Show D + δ is winnable for any v (∀ v, winnable G (D + δ))
      intro v -- Take an arbitrary vertex v

      -- Define the divisor form associated with c and the form plus δ
      let D'_form : CFDiv V := λ w => c.vertex_degree w - if w = q then (1 : ℤ) else (0 : ℤ)
      let delta_v : CFDiv V := fun w => ite (w = v) 1 0
      let D'_form_plus_delta_v := D'_form + delta_v

      -- Adding a chip to the form derived from a maximal superstable config makes it winnable
      have h_win_D'_form_plus_delta_v : winnable G D'_form_plus_delta_v :=
        helper_maximal_superstable_chip_winnable_exact G q c h_max_c v

      -- Define D + δ
      let D_plus_delta_v := D + delta_v

      -- Show that D + δ is linearly equivalent to D' + δ (which equals D'_form + δ)
      have h_equiv_plus_delta : linear_equiv G D_plus_delta_v D'_form_plus_delta_v := by
        -- Goal: (D'_form + delta_v) - (D + delta_v) ∈ P
        unfold linear_equiv -- Explicitly unfold goal
        -- Simplify the difference using group properties
        have h_diff_simp : (D'_form + delta_v) - (D + delta_v) = D'_form - D := by
           funext w; simp only [add_apply, sub_apply]; ring -- Pointwise proof (use
```

```lean
    funext)
        rw [h_diff_simp] -- Apply simplification
        -- Goal: D'_form - D ∈ P
        -- We know D' - D ∈ P (from h_D_equiv_D')
        unfold linear_equiv at h_D_equiv_D'
        -- We know D' = D'_form (by h_form_D'_eq_c)
        rw [h_form_D'_eq_c] at h_D_equiv_D' -- Rewrite D' as D'_form in h_D_equiv_D'
        exact h_D_equiv_D' -- Use the rewritten hypothesis

      -- Since D + δ is linearly equivalent to a winnable divisor (D'_form + δ), it
    must be winnable.
      exact (helper_linear_equiv_preserves_winnability G D_plus_delta_v
    D'_form_plus_delta_v h_equiv_plus_delta).mpr h_win_D'_form_plus_delta_v
    }
  }

/-- [Proven] Proposition 4.1.13: Combined (1) and (2)-/
theorem superstable_and_maximal_unwinnable (G : CFGraph V) (q : V)
    (c : Config V q) (D : CFDiv V) :
    (superstable G q c →
     (maximal_superstable G c ↔ config_degree c = genus G)) ∧
    (maximal_unwinnable G D ↔
     ∃ c : Config V q, maximal_superstable G c ∧
     ∃ D' : CFDiv V, q_reduced G q D' ∧ linear_equiv G D D' ∧
     D' = λ v => c.vertex_degree v - if v = q then 1 else 0) := by
  -- This theorem now just wraps the two proven theorems above
  exact ⟨maximal_superstable_config_prop G q c,
         maximal_unwinnable_char G q D⟩

/-- Theorem: A maximal unwinnable divisor has degree g-1
    This theorem now proven based on the characterizations above. -/
theorem maximal_unwinnable_deg {V : Type} [DecidableEq V] [Fintype V]
  (G : CFGraph V) (D : CFDiv V) :
  maximal_unwinnable G D → deg D = genus G - 1 := by
  intro h_max_unwin

  rcases Fintype.exists_elem V with ⟨q, _⟩

  have h_equiv_max_unwin := maximal_unwinnable_char G q D
  rcases h_equiv_max_unwin.mp h_max_unwin with ⟨c, h_c_max_super, D', h_D'_qred,
    h_equiv_D_D', h_D'_eq⟩

  have h_c_super : superstable G q c := h_c_max_super.1

  -- Use the characterization theorem for config degree
  have h_config_deg : config_degree c = genus G := by
    have prop := maximal_superstable_config_prop G q c h_c_super -- Apply hypothesis
    first
    exact prop.mp h_c_max_super -- Use the forward direction of the iff

  have h_deg_D' : deg D' = genus G - 1 := calc
```

```
    deg D' = deg (λ v => c.vertex_degree v - if v = q then 1 else 0) := by rw [h_D'_eq]
    _ = (Σ v, c.vertex_degree v) - (Σ v, if v = q then 1 else 0) := by {unfold deg; rw [Finset.sum_sub_distrib]}
    _ = (Σ v, c.vertex_degree v) - 1 := by {rw [Finset.sum_ite_eq']; simp}
    _ = (config_degree c + c.vertex_degree q) - 1 := by
        have h_sum_c : Σ v : V, c.vertex_degree v = config_degree c + c.vertex_degree q := by
          unfold config_degree
          rw [← Finset.sum_filter_add_sum_filter_not (s := Finset.univ) (p := λ v' => v' ≠ q)] -- Split sum based on v ≠ q
          simp [Finset.sum_singleton, Finset.filter_eq'] -- Add Finset.filter_eq' hint
        rw [h_sum_c]
    _ = genus G - 1 := by
        -- Since c is maximal superstable, it corresponds to an orientation
        rcases helper_maximal_superstable_orientation G q c h_c_max_super with
          ⟨O, h_acyc, h_unique_source, h_c_eq_orient_config⟩

        -- The configuration derived from an orientation has 0 at q
        have h_orient_config_q_zero : (orientation_to_config G O q h_acyc h_unique_source).vertex_degree q = 0 := by
          unfold orientation_to_config config_of_source
          simp

        -- Thus, c must have 0 at q
        have h_c_q_zero : c.vertex_degree q = 0 := by
          -- First establish equality of the vertex_degree functions using structure equality
          have h_vertex_degree_eq : c.vertex_degree = (orientation_to_config G O q h_acyc h_unique_source).vertex_degree := by
            rw [h_c_eq_orient_config] -- This leaves the goal c.vertex_degree = c.vertex_degree which is true by reflexivity
          -- Apply the function equality at vertex q
          have h_eq_at_q := congr_fun h_vertex_degree_eq q
          -- Rewrite the RHS of h_eq_at_q using the known value (0)
          rw [h_orient_config_q_zero] at h_eq_at_q
          -- The result is the desired equality
          exact h_eq_at_q

        -- Now substitute known values into the expression
        rw [h_config_deg, h_c_q_zero] -- config_degree c = genus G and c.vertex_degree q = 0
        simp -- genus G + 0 - 1 = genus G - 1

  have h_deg_eq : deg D = deg D' := linear_equiv_preserves_deg G D D' h_equiv_D_D'
  rw [h_deg_eq, h_deg_D']

/-- [Proven] Proposition 4.1.14: Key results about maximal unwinnable divisors:
    1) There is an injection from acyclic orientations with source q to maximal unwinnable q-reduced divisors,
       where an orientation O maps to the divisor D(O) - q where D(O) assigns
```

```
    indegree to each vertex. (Surjection proof deferred)
    2) Any maximal unwinnable divisor has degree equal to genus - 1. -/
theorem acyclic_orientation_maximal_unwinnable_correspondence_and_degree
    {V : Type} [DecidableEq V] [Fintype V] (G : CFGraph V) (q : V) :
    (Function.Injective (λ (O : {O : Orientation G // is_acyclic G O ∧ is_source G O
    q}) =>
      λ v => (indeg G O.val v) - if v = q then 1 else 0)) ∧
    (∀ D : CFDiv V, maximal_unwinnable G D → deg D = genus G - 1) := by
  constructor
  { -- Part 1: Injection proof
    intros O₁ O₂ h_eq
    have h_indeg : ∀ v : V, indeg G O₁.val v = indeg G O₂.val v := by
      intro v
      have := congr_fun h_eq v
      by_cases hv : v = q
      · exact helper_source_indeg_eq_at_q G O₁.val O₂.val q v O₁.prop.2 O₂.prop.2 hv
      · simp [hv] at this
        exact this
    exact Subtype.ext (acyclic_orientation_unique_by_indeg O₁.val O₂.val O₁.prop.1
    O₂.prop.1 h_indeg)
  }
  { -- Part 2: Degree characterization
    -- This now correctly refers to the theorem defined above
    intro D hD
    exact maximal_unwinnable_deg G D hD
  }

/-- [Proven] Corollary 4.2.2: Rank inequality for divisors -/
theorem rank_subadditive (G : CFGraph V) (D D' : CFDiv V)
    (h_D : rank G D ≥ 0) (h_D' : rank G D' ≥ 0) :
    rank G (λ v => D v + D' v) ≥ rank G D + rank G D' := by
  -- Convert ranks to natural numbers
  let k₁ := (rank G D).toNat
  let k₂ := (rank G D').toNat

  -- Show rank is ≥ k₁ + k₂ by proving rank_geq
  have h_rank_geq : rank_geq G (λ v => D v + D' v) (k₁ + k₂) := by
    -- Take any effective divisor E'' of degree k₁ + k₂
    intro E'' h_E''
    have ⟨h_eff, h_deg⟩ := h_E''

    -- Decompose E'' into E₁ and E₂ of degrees k₁ and k₂
    have ⟨E₁, E₂, h_E₁_eff, h_E₂_eff, h_E₁_deg, h_E₂_deg, h_sum⟩ :=
      helper_divisor_decomposition G E'' k₁ k₂ h_eff h_deg

    -- Convert our nat-based hypotheses to ℤ-based ones for rank_spec
    have h_D_nat : rank G D ≥ ↑k₁ := by
      have h_conv : ↑((rank G D).toNat) = rank G D := Int.toNat_of_nonneg h_D
      rw [←h_conv]

    have h_D'_nat : rank G D' ≥ ↑k₂ := by
```

```
      have h_conv : ↑((rank G D').toNat) = rank G D' := Int.toNat_of_nonneg h_D'
      rw [←h_conv]

    -- Get rank_geq properties from rank_spec
    have h_D_rank_geq := ((rank_spec G D).2.1 k₁).mp h_D_nat
    have h_D'_rank_geq := ((rank_spec G D').2.1 k₂).mp h_D'_nat

    -- Apply rank_geq to get winnability for both parts
    have h_D_win := h_D_rank_geq E₁ (by exact ⟨h_E₁_eff, h_E₁_deg⟩)
    have h_D'_win := h_D'_rank_geq E₂ (by exact ⟨h_E₂_eff, h_E₂_deg⟩)

    -- Show that (D + D') - (E₁ + E₂) = (D - E₁) + (D' - E₂)
    have h_rearrange : (λ v => (D v + D' v) - (E₁ v + E₂ v)) =
                      (λ v => (D v - E₁ v) + (D' v - E₂ v)) := by
      funext v
      ring

    -- Show winnability of sum using helper_winnable_add and rearrangement
    rw [h_sum, h_rearrange]
    exact helper_winnable_add G (λ v => D v - E₁ v) (λ v => D' v - E₂ v) h_D_win
    h_D'_win

  -- Connect k₁, k₂ back to original ranks
  have h_k₁ : ↑k₁ = rank G D := by
    exact Int.toNat_of_nonneg h_D

  have h_k₂ : ↑k₂ = rank G D' := by
    exact Int.toNat_of_nonneg h_D'

  -- Show final inequality using transitivity
  have h_final := ((rank_spec G (λ v => D v + D' v)).2.1 (k₁ + k₂)).mpr h_rank_geq

  have h_sum : ↑(k₁ + k₂) = rank G D + rank G D' := by
    simp only [Nat.cast_add]  -- Use Nat.cast_add instead of Int.coe_add
    rw [h_k₁, h_k₂]

  rw [h_sum] at h_final
  exact h_final

-- [Proven] Corollary 4.2.3: Degree of canonical divisor equals 2g - 2
theorem degree_of_canonical_divisor (G : CFGraph V) :
    deg (canonical_divisor G) = 2 * genus G - 2 := by
  -- First unfold definitions
  unfold deg canonical_divisor

  -- Use sum_sub_distrib to split the sum
  have h1 : Σ v, (vertex_degree G v - 2) =
            Σ v, vertex_degree G v - 2 * Fintype.card V := by
    rw [sum_sub_distrib]
    simp [sum_const, nsmul_eq_mul]
    ring
```

```
  rw [h1]

  -- Use the fact that sum of vertex degrees = 2|E|
  have h2 : Σ v, vertex_degree G v = 2 * Multiset.card G.edges := by
    exact helper_sum_vertex_degrees G
  rw [h2]

  -- Use genus definition: g = |E| - |V| + 1
  rw [genus]

  ring

/-- [Proven] Rank Degree Inequality -/
theorem rank_degree_inequality {V : Type} [DecidableEq V] [Fintype V]
    (G : CFGraph V) (D : CFDiv V) :
    deg D - genus G < rank G D - rank G (λ v => canonical_divisor G v - D v) := by
  -- Get rank value for D
  let r := rank G D

  -- Get effective divisor E using rank characterization
  rcases rank_get_effective G D with ⟨E, h_E_eff, h_E_deg, h_D_E_unwin⟩

  -- Fix a vertex q
  rcases Fintype.exists_elem V with ⟨q, _⟩

  -- Apply Dhar's algorithm to D - E to get q-reduced form
  rcases helper_dhar_algorithm G q (λ v => D v - E v) with ⟨c, k, h_equiv, h_super⟩

  -- k must be negative since D - E is unwinnable
  have h_k_neg := helper_dhar_negative_k G q (λ v => D v - E v) h_D_E_unwin c k
    h_equiv h_super

  -- Get maximal superstable c' ≥ c
  rcases helper_maximal_superstable_exists G q c h_super with ⟨c', h_max', h_ge⟩

  -- Let O be corresponding acyclic orientation using the bijection
  rcases stable_bijection G q with ⟨h_inj, h_surj⟩
  -- Apply h_surj to the subtype element ⟨c', h_max'⟩
  rcases h_surj ⟨c', h_max'⟩ with ⟨O_subtype, h_eq_c'⟩ -- O_subtype is {O // acyclic ∧
    unique_source}

  -- Get configuration c' from orientation O_subtype
  -- O_subtype.val is the Orientation, O_subtype.prop.1 is acyclicity,
    O_subtype.prop.2 is uniqueness
  let c'_config := orientation_to_config G O_subtype.val q O_subtype.prop.1
    O_subtype.prop.2

  -- Check consistency: h_eq_c' implies c'_config = c'
  have h_orient_eq_c' : c'_config = c' := by exact Subtype.mk.inj h_eq_c'
```

```
-- Check consistency (assuming h_eq_c' implies c' = c'_config)
-- Define H := (c' - c) - (k + 1)q as a divisor (using original c')
let H : CFDiv V := λ v =>
  if v = q then -(k + 1)
  else c'.vertex_degree v - c.vertex_degree v

have h_H_eff : effective H := by
  intro v
  by_cases h_v : v = q
  · -- Case v = q
    rw [h_v]
    simp [H]
    -- Since k < 0, k + 1 ≤ 0, so -(k + 1) ≥ 0
    have h_k_plus_one_nonpos : k + 1 ≤ 0 := by
      linarith [h_k_neg]
    linarith

  · -- Case v ≠ q
    simp [H, h_v]
    -- h_ge shows c' ≥ c for maximal superstable c'
    have h_ge_at_v : c'.vertex_degree v ≥ c.vertex_degree v := by
      exact h_ge v
    -- Therefore difference is non-negative
    linarith

-- Complete h_DO_unwin
have h_DO_unwin : maximal_unwinnable G (λ v => c'.vertex_degree v - if v = q then 1 else 0) := by
  constructor
  · -- First show it's unwinnable
    exact helper_superstable_to_unwinnable G q c' h_max'

  · -- Then show adding a chip anywhere makes it winnable
    exact helper_maximal_superstable_chip_winnable_exact G q c' h_max'

-- Use degree property of maximal unwinnable divisors
have h_DO_deg : deg (λ v => c'.vertex_degree v - if v = q then 1 else 0) = genus G - 1 :=
  maximal_unwinnable_deg G _ h_DO_unwin

calc deg D - genus G
  _ = deg D - (Multiset.card G.edges - Fintype.card V + 1) := by rw [genus]
  _ < deg D - deg E + deg H := by
      -- Substitute deg E = rank G D + 1
      rw [h_E_deg]
      -- Goal simplifies to: rank G D + 1 - genus G < deg H
      -- Apply the axiom to get this inequality as a hypothesis
      have h_bound := helper_H_degree_bound G q D H k c c' h_H_eff (by simp [H]) -- Provide proof for H form
      -- Show the original goal follows algebraically
      linarith [h_bound]
```

```
    _ ≤ rank G D - rank G (λ v => canonical_divisor G v - D v) := by
      -- This inequality is helper_rank_deg_canonical_bound rearranged
      apply le_sub_iff_add_le.mpr
      -- Provide the axiom conclusion and let linarith handle rearrangement
      have h_bound_orig := helper_rank_deg_canonical_bound G q D E H c' (helper_DO_linear_equiv G q D E H c')
      linarith [h_bound_orig]
```

A.8 Riemann-Roch for Graphs and Relevant Corollaries (RiemannRochForGraphs.lean)

```
import ChipFiringWithLean.Basic
import ChipFiringWithLean.Config
import ChipFiringWithLean.Orientation
import ChipFiringWithLean.Rank
import ChipFiringWithLean.RRGHelpers
import Mathlib.Algebra.Ring.Int
import Paperproof

set_option linter.unusedVariables false
set_option trace.split.failure true

open Multiset Finset

-- Assume V is a finite type with decidable equality
variable {V : Type} [DecidableEq V] [Fintype V]

/-- [Proven] The main Riemann-Roch theorem for graphs -/
theorem riemann_roch_for_graphs {V : Type} [DecidableEq V] [Fintype V] (G : CFGraph V) (D : CFDiv V) :
  rank G D - rank G (λ v => canonical_divisor G v - D v) = deg D - genus G + 1 := by
  -- Get rank value for D
  let r := rank G D

  -- Get effective divisor E using rank characterization
  rcases rank_get_effective G D with ⟨E, h_E_eff, h_E_deg, h_D_E_unwin⟩

  -- Fix a vertex q
  rcases Fintype.exists_elem V with ⟨q, _⟩

  -- Apply Dhar's algorithm to D - E to get q-reduced form
  rcases helper_dhar_algorithm G q (λ v => D v - E v) with ⟨c, k, h_equiv, h_super⟩

  -- k must be negative since D - E is unwinnable
  have h_k_neg := helper_dhar_negative_k G q (λ v => D v - E v) h_D_E_unwin c k h_equiv h_super

  -- Get maximal superstable c' ≥ c
  rcases helper_maximal_superstable_exists G q c h_super with ⟨c', h_max', h_ge⟩

```

```lean
-- Let O be corresponding acyclic orientation with unique source q (from bijection)
rcases stable_bijection G q with ⟨_, h_surj⟩
-- O_subtype has type {O // is_acyclic G O ∧ (∀ w, is_source G O w → w = q)}
rcases h_surj ⟨c', h_max'⟩ with ⟨O_subtype, h_f_eq_c'⟩

-- From h_f_eq_c' : f O_subtype = ⟨c', h_max'⟩, we get that the configuration part is equal
have h_orient_config_eq_c' : orientation_to_config G O_subtype.val q O_subtype.prop.1 O_subtype.prop.2 = c' := by
  exact Subtype.mk.inj h_f_eq_c'

-- Define H := (c' - c) - (k + 1)q as a divisor
let H : CFDiv V := λ v =>
  if v = q then -(k + 1)
  else c'.vertex_degree v - c.vertex_degree v

-- Get canonical divisor decomposition
rcases canonical_is_sum_orientations G with ⟨O₁, O₂, h_O₁_acyc, h_O₂_acyc, h_K⟩

-- Get key inequality from axiom
have h_ineq := rank_degree_inequality G D

-- Get reverse inequality by applying to K-D
have h_ineq_rev := rank_degree_inequality G (λ v => canonical_divisor G v - D v)

-- Get degree of canonical divisor
have h_deg_K : deg (canonical_divisor G) = 2 * genus G - 2 :=
  degree_of_canonical_divisor G

-- Since rank is an integer and we have bounds, equality must hold
suffices rank G D - rank G (λ v => canonical_divisor G v - D v) ≥ deg D - genus G + 1 ∧
       rank G D - rank G (λ v => canonical_divisor G v - D v) ≤ deg D - genus G + 1 from
  le_antisymm (this.2) (this.1)

constructor
· -- Lower bound
  linarith [h_ineq]
· -- Upper bound
  have h_swap := rank_degree_inequality G (λ v => canonical_divisor G v - D v)
  -- Simplify double subtraction in h_swap
  have h_sub_simplify : (λ (v : V) => canonical_divisor G v - (canonical_divisor G v - D v)) = D := by
    funext v
    ring

  rw [h_sub_simplify] at h_swap

  have h_deg_sub : deg (λ v => canonical_divisor G v - D v) = deg (canonical_divisor G) - deg D := by
```

```
      unfold deg
      -- Split the sum over subtraction
      rw [Finset.sum_sub_distrib]

    -- Substitute the degree of canonical divisor
    rw [h_deg_K] at h_deg_sub

    -- Simplify inequality
    have h_ineq_sub : deg (λ v => canonical_divisor G v - D v) - genus G <
      rank G (λ v => canonical_divisor G v - D v) - rank G D := h_swap

    rw [h_deg_sub] at h_ineq_sub

    -- Final inequality using linarith
    linarith [h_ineq_sub]

/-- [Proven] Corollary 4.4.1: A divisor D is maximal unwinnable if and only if K-D is maximal unwinnable -/
theorem maximal_unwinnable_symmetry {V : Type} [DecidableEq V] [Fintype V]
    (G : CFGraph V) (D : CFDiv V) :
  maximal_unwinnable G D ↔ maximal_unwinnable G (λ v => canonical_divisor G v - D v) := by
  constructor
  -- Forward direction
  { intro h_max_unwin
    -- Get rank = -1 from maximal unwinnable
    have h_rank_neg : rank G D = -1 := by
      rw [rank_neg_one_iff_unwinnable]
      exact h_max_unwin.1

    -- Get degree = g-1 from maximal unwinnable
    have h_deg : deg D = genus G - 1 := maximal_unwinnable_deg G D h_max_unwin

    -- Use Riemann-Roch
    have h_RR := riemann_roch_for_graphs G D
    rw [h_rank_neg] at h_RR

    -- Get degree of K-D
    have h_deg_K := degree_of_canonical_divisor G
    have h_deg_KD : deg (λ v => canonical_divisor G v - D v) = genus G - 1 := by
      -- Get general distributive property for deg over subtraction
      have h_deg_sub : ∀ D₁ D₂ : CFDiv V, deg (D₁ - D₂) = deg D₁ - deg D₂ := by
        intro D₁ D₂
        unfold deg
        simp [sub_apply]

      -- Convert lambda form to standard subtraction
      rw [divisor_sub_eq_lambda G (canonical_divisor G) D]

      -- Apply distributive property
      rw [h_deg_sub (canonical_divisor G) D]
```

```

   -- Use known values
   rw [h_deg_K, h_deg]

   -- Arithmetic: (2g-2) - (g-1) = g-1
   ring

  constructor
  · -- K-D is unwinnable
   rw [←rank_neg_one_iff_unwinnable]
   linarith
  · -- Adding chip makes K-D winnable
   intro v
   have h_win := h_max_unwin.2 v

   -- Define the divisors explicitly to avoid type confusion
   let D₁ : CFDiv V := λ w => D w + if w = v then 1 else 0
   let D₂ : CFDiv V := λ w => canonical_divisor G w - D w + if w = v then 1 else 0

   -- Show goal matches D₂
   have h_goal : (λ w => (canonical_divisor G w - D w) + if w = v then 1 else 0) = D₂ := by
    funext w
    simp [D₂]

   -- Use linear equivalence to transfer winnability
   have h_equiv := linear_equiv_add_chip G D v h_deg
   have h_win_transfer := (helper_linear_equiv_preserves_winnability G D₁ D₂ h_equiv).mp h_win

   -- Apply the result
   rw [h_goal]
   exact h_win_transfer
 }
 -- Reverse direction
 { intro h_max_unwin_K
  -- Apply canonical double difference
  rw [←canonical_double_diff G D]
  -- Mirror forward direction's proof
  exact maximal_unwinnable_symmetry G (λ v => canonical_divisor G v - D v) |>.mp h_max_unwin_K
 }
 termination_by (rank G D + 1).toNat
 decreasing_by { exact rank_decreases_for_KD G D h_max_unwin_K }


/-- [Proven] Clifford's Theorem (4.4.2): For a divisor D with non-negative rank
      and K-D also having non-negative rank, the rank of D is at most half its degree. -/
theorem clifford_theorem {V : Type} [DecidableEq V] [Fintype V]
  (G : CFGraph V) (D : CFDiv V)
```

```
    (h_D : rank G D ≥ 0)
    (h_KD : rank G (λ v => canonical_divisor G v - D v) ≥ 0) :
    (rank G D : ℚ) ≤ (deg D : ℚ) / 2 := by
  -- Get canonical divisor K's rank using Riemann-Roch
  have h_K_rank : rank G (canonical_divisor G) = genus G - 1 := by
    -- Apply Riemann-Roch with D = K
    have h_rr := riemann_roch_for_graphs G (canonical_divisor G)
    -- For K-K = 0, rank is 0
    have h_K_minus_K : rank G (λ v => canonical_divisor G v - canonical_divisor G v) = 0 := by
      -- Show that this divisor is the zero divisor
      have h1 : (λ v => canonical_divisor G v - canonical_divisor G v) = (λ _ => 0) := by
        funext v
        simp [sub_self]

      -- Show that the zero divisor has rank 0
      have h2 : rank G (λ _ => 0) = 0 := zero_divisor_rank G

      -- Substitute back
      rw [h1, h2]
    -- Substitute into Riemann-Roch
    rw [h_K_minus_K] at h_rr
    -- Use degree_of_canonical_divisor
    rw [degree_of_canonical_divisor] at h_rr
    -- Solve for rank G K
    linarith

  -- Apply rank subadditivity
  have h_subadd := rank_subadditive G D (λ v => canonical_divisor G v - D v) h_D h_KD
  -- The sum D + (K-D) = K
  have h_sum : (λ v => D v + (canonical_divisor G v - D v)) = canonical_divisor G := by
    funext v
    simp
  rw [h_sum] at h_subadd
  rw [h_K_rank] at h_subadd

  -- Use Riemann-Roch to get r(K-D) in terms of r(D)
  have h_rr := riemann_roch_for_graphs G D

  -- Explicit algebraic manipulation
  have h1 : rank G (λ v => canonical_divisor G v - D v) =
           rank G D - (deg D - genus G + 1) := by
    linarith

  -- Substitute this into the subadditivity inequality
  have h2 : genus G - 1 ≥ rank G D + (rank G D - (deg D - genus G + 1)) := by
    rw [h1] at h_subadd
    exact h_subadd

```

```
  -- Solve for rank G D
  have h3 : 2 * rank G D - (deg D - genus G + 1) ≤ genus G - 1 := by
    linarith

  have h4 : 2 * rank G D ≤ deg D := by
    linarith

  have h5 : (rank G D : ℚ) ≤ (deg D : ℚ) / 2 := by
    -- Convert to rational numbers and use algebraic properties
    have h_cast : (2 : ℚ) * (rank G D : ℚ) ≤ (deg D : ℚ) := by
      -- Cast integer inequality to rational
      exact_mod_cast h4

    -- Divide both sides by 2 directly using algebra
    have h_two_pos : (0 : ℚ) < 2 := by norm_num

    calc (rank G D : ℚ)
      _  = (rank G D : ℚ) * (1 : ℚ) := by ring
      _  = (rank G D : ℚ) * (2 / 2 : ℚ) := by norm_num
      _  = (2 : ℚ) * (rank G D : ℚ) / 2 := by field_simp
      _  ≤ (deg D : ℚ) / 2 := by
          -- Use the fact that division by positive number preserves inequality
          apply (div_le_div_right h_two_pos).mpr
          exact h_cast

  exact h5

/-- [Proven] RRG's Corollary 4.4.3 establishing divisor degree to rank correspondence
     -/
theorem riemann_roch_deg_to_rank_corollary {V : Type} [DecidableEq V] [Fintype V]
  (G : CFGraph V) (D : CFDiv V) :
  -- Part 1
  (deg D < 0 → rank G D = -1) ∧
  -- Part 2
  (0 ≤ (deg D : ℚ) ∧ (deg D : ℚ) ≤ 2 * (genus G : ℚ) - 2 → (rank G D : ℚ) ≤ (deg
    D : ℚ) / 2) ∧
  -- Part 3
  (deg D > 2 * genus G - 2 → rank G D = deg D - genus G) := by
  constructor
  · -- Part 1: deg(D) < 0 implies r(D) = -1
    intro h_deg_neg
    rw [rank_neg_one_iff_unwinnable]
    intro h_winnable
    -- Use winnable_iff_exists_effective
    obtain ⟨D', h_eff, h_equiv⟩ := winnable_iff_exists_effective G D |>.mp h_winnable
    -- Linear equivalence preserves degree
    have h_deg_eq : deg D = deg D' := by
      exact linear_equiv_preserves_deg G D D' h_equiv
    -- Effective divisors have non-negative degree
    have h_D'_nonneg : deg D' ≥ 0 := by
      exact effective_nonneg_deg D' h_eff
```

```
    -- Contradiction: D has negative degree but is equivalent to non-negative degree divisor
    rw [←h_deg_eq] at h_D'_nonneg
    exact not_le_of_gt h_deg_neg h_D'_nonneg

  constructor
  · -- Part 2: 0 ≤ deg(D) ≤ 2g-2 implies r(D) ≤ deg(D)/2
    intro ⟨h_deg_nonneg, h_deg_upper⟩
    by_cases h_rank : rank G D ≥ 0
    · -- Case where r(D) ≥ 0
      let K := canonical_divisor G
      by_cases h_rankKD : rank G (λ v => K v - D v) ≥ 0
      · -- Case where r(K-D) ≥ 0: use Clifford's theorem
        exact clifford_theorem G D h_rank h_rankKD
      · -- Case where r(K-D) = -1: use Riemann-Roch
        have h_rr := riemann_roch_for_graphs G D
        have h_rankKD_eq : rank G (λ v => K v - D v) = -1 :=
          rank_neg_one_of_not_nonneg G (λ v => K v - D v) h_rankKD

        rw [h_rankKD_eq] at h_rr

        -- Arithmetic manipulation to get r(D) equality
        have this : rank G D = deg D - genus G := by
          -- Convert h_rr from (rank G D - (-1)) to (rank G D + 1)
          rw [sub_neg_eq_add] at h_rr
          have := calc
            rank G D = rank G D + 1 - 1 := by ring
            _ = deg D - genus G + 1 - 1 := by rw [h_rr]
            _ = deg D - genus G := by ring
          exact this

        -- Apply the result
        rw [this]

        -- Show that deg D - genus G ≤ deg D / 2 using rational numbers
        have h_bound : (deg D - genus G : ℚ) ≤ (deg D : ℚ) / 2 := by
          linarith [h_deg_upper]

        -- Make sure types match with explicit cast
        have h_cast : (deg D - genus G : ℚ) = (↑(deg D - genus G) : ℚ) := by
          exact_mod_cast rfl
        rw [← h_cast]
        exact h_bound

    · -- Case where r(D) < 0
      have h_rank_eq := rank_neg_one_of_not_nonneg G D h_rank
      have h_bound : -1 ≤ deg D / 2 := by
        -- The division by 2 preserves non-negativity for deg D
        have h_div_nonneg : deg D / 2 ≥ 0 := by
          have h_two_pos : (2 : ℤ) > 0 := by norm_num
          rw [Int.div_nonneg_iff_of_pos h_two_pos]
```

```
     -- Convert explicitly to the right type
     have h : deg D ≥ 0 := by exact_mod_cast h_deg_nonneg
     exact h

    linarith
   rw [h_rank_eq]

   -- Convert to rational numbers
   have h_bound_rat : ((-1) : ℚ) ≤ (deg D : ℚ) / 2 := by linarith [h_bound]

   exact h_bound_rat

 · -- Part 3: deg(D) > 2g-2 implies r(D) = deg(D) - g
  intro h_deg_large
  have h_canon := degree_of_canonical_divisor G
  -- Show K-D has negative degree
  have h_KD_neg : deg (λ v => canonical_divisor G v - D v) < 0 := by
   -- Calculate deg(K-D)
   calc
    deg (λ v => canonical_divisor G v - D v)
    _ = deg (canonical_divisor G) - deg D := by
     unfold deg
     simp [sub_apply]
    _ = 2 * genus G - 2 - deg D := by rw [h_canon]
    _ < 0 := by linarith

  -- Show K-D is unwinnable, so rank = -1
  have h_rankKD : rank G (λ v => canonical_divisor G v - D v) = -1 := by
   rw [rank_neg_one_iff_unwinnable]
   intro h_win
   -- If winnable, would be linearly equivalent to effective divisor
   obtain ⟨E, h_eff, h_equiv⟩ := winnable_iff_exists_effective G _ |>.mp h_win
   have h_deg_eq := linear_equiv_preserves_deg G _ E h_equiv
   -- But effective divisors have non-negative degree
   have h_E_nonneg := effective_nonneg_deg E h_eff
   rw [←h_deg_eq] at h_E_nonneg
   -- Contradiction: K-D has negative degree
   exact not_le_of_gt h_KD_neg h_E_nonneg

  -- Apply Riemann-Roch to get r(D) = deg(D) - g
  have h_rr := riemann_roch_for_graphs G D
  rw [h_rankKD] at h_rr
  rw [sub_neg_eq_add] at h_rr
  linarith
```

A.9 Package Main Index (ChipFiringWithLean.lean)

```
-- This module serves as the root of the `ChipFiringWithLean` library.
-- Import modules here that should be built as part of the library.
import ChipFiringWithLean.Basic
```

```
import ChipFiringWithLean.CFGraphExample
import ChipFiringWithLean.Config
import ChipFiringWithLean.Orientation
import ChipFiringWithLean.Rank
import ChipFiringWithLean.Helpers
import ChipFiringWithLean.RRGHelpers
import ChipFiringWithLean.RiemannRochForGraphs
```

A.10 Lean CI/CD YAML Script for GitHub Actions (lean-action-ci.yml)

We used the following YAML configuration file for Continuous Integration (CI) Testing with Lean4 and Mathlib4. This allowed us to use runners (temporary virtual machines) sanctioned by GitHub via GitHub Actions, ensuring that our work compiles and is integrable with the broader work in the Lean4 landscape.

```
name: Lean Action CI

on:
  push:
  pull_request:
  workflow_dispatch:

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - uses: actions/checkout@v4
      - uses: leanprover/lean-action@v1
```

A.11 Lakefile Dependency and Library Management (lakefile.lean)

We used the following Lakefile (Lean4's wrapper over Makefile) to specify our Lean4 project's metadata, dependencies, and build configuration. This allowed us to seamlessly integrate Mathlib, Paperproof, and native C++ libraries while enabling Unicode-friendly pretty-printing for improved readability.

```
import Lake
open Lake DSL

package "chip-firing-with-lean" where
  version := v!"0.1.0"
  keywords := #["math"]
  leanOptions := #[
    ⟨`pp.unicode.fun, true⟩ -- pretty-prints `fun a ↦ b`
```

```
  ]
  moreLinkArgs := #[
    "-L./.lake/packages/LeanCopilot/.lake/build/lib",
    "-lctranslate2"
  ]

require "leanprover-community" / "mathlib"
require Paperproof from git "https://github.com/Paper-Proof/paperproof.git"@"main"/"lean"

@[default_target]
lean_lib ChipFiringWithLean where
  -- add any library configuration options here
```

A.12 Streamlined Version Control (.gitignore)

We used the following ‘.gitignore‘ file to exclude local development artifacts (mainly cached libraries and dependencies), editor-specific settings, and system files from version control. This ensures a clean and portable repository for collaborators and CI systems.

```
/.lake
/.vscode
*.DS_Store
```

A.13 Chip Firing Simulation with Algorithms for Winnability Determination in Python

Below, we present an initial implementation we wrote to understand and efficiently implement the dollar game and various related algorithms. This further motivated us and led to the ideation of a Python package that we are currently working on: chipfiring (`https://pypi.org/project/chipfiring/`).

A.13.1 Laplacian Utility Class

```python
# chip-firing-simulation/utils/laplacian.py
from collections import defaultdict

class Laplacian:
    def __init__(self, graph):
        """
        Initialize the Laplacian with a graph.

        :param graph: A dictionary representing the adjacency list of the graph.
        """
```

```python
        self.graph = graph

    def construct_matrix(self):
        """
        Construct the Laplacian matrix for the graph.

        :return laplacian: A dictionary where each key is a vertex, and the value is a dictionary representing the row of the Laplacian matrix.
        """
        laplacian = defaultdict(lambda: defaultdict(int))
        for v in self.graph:
            degree = sum(self.graph[v].values())
            laplacian[v][v] = degree
            for w, edge_count in self.graph[v].items():
                laplacian[v][w] -= edge_count
        return laplacian

    def apply(self, divisor, firing_script):
        """
        Apply the Laplacian matrix to a firing script to calculate the resulting divisor.

        :param divisor: Initial divisor dictionary representing wealth at each vertex.
        :param firing_script: The firing script dictionary where keys are vertices and values are the number of times they fired.
        :return resulting_divisor: A dictionary representing the resulting divisor after applying the Laplacian.
        """
        laplacian = self.construct_matrix()
        resulting_divisor = defaultdict(int, divisor)

        for v in self.graph:
            for w in self.graph:
                resulting_divisor[v] -= laplacian[v][w] * firing_script[w]

        return resulting_divisor

```

A.13.2 Greedy Algorithm Class

```python
# chip-firing-simulation/algorithms/greedy_algorithm.py
class GreedyAlgorithm:
    def __init__(self, graph, divisor):
        """
        Initialize the greedy algorithm for the dollar game.

        :param graph: A dictionary representing the adjacency list of the graph.
        :param divisor: A dictionary representing the wealth at each vertex.
        """
```

```python
        self.graph = graph
        self.divisor = divisor.copy()  # Make a copy to avoid modifying original
        self.marked_vertices = set()
        self.firing_script = {v: 0 for v in graph} # Initialize firing script with all vertices

    def is_effective(self):
        """
        Check if all vertices have non-negative wealth.

        :return: True if effective, otherwise False.
        """
        return all(wealth >= 0 for wealth in self.divisor.values())

    def borrowing_move(self, vertex):
        """
        Perform a borrowing move at the specified vertex.

        :param vertex: The vertex at which to perform the borrowing move.
        """
        # Decrement the borrowing vertex's firing script since it's receiving
        self.firing_script[vertex] -= 1

        # Update wealth based on the borrowing move
        for neighbor, edge_count in self.graph[vertex].items():
            total_borrowed = edge_count
            self.divisor[neighbor] -= total_borrowed
            self.divisor[vertex] += total_borrowed

    def play(self):
        """
        Execute the greedy algorithm to determine winnability.

        :return: Tuple (True, firing_script) if the game is winnable; otherwise (False, None).
        """
        moves = 0
        # Enforcing a Scalable and Reasonable upper bound
        max_moves = len(self.graph) * 10

        while not self.is_effective():
            moves += 1
            if moves > max_moves:
                return False, None

            in_debt_vertex = next((v for v in self.divisor if self.divisor[v] < 0), None)
            if in_debt_vertex is None:
                break

```

```python
            self.borrowing_move(in_debt_vertex)

        return True, dict(self.firing_script)

```

A.13.3 Dhar's Algorithm Class

```python
# chip-firing-simulation/algorithms/dhar_algorithm.py
class DharAlgorithm:
    def __init__(self, graph, configuration, q):
        """
        Initialize Dhar's Algorithm for finding a maximal legal firing set.

        Args:
            graph: A dictionary representing the adjacency list of the graph
            configuration: A dictionary representing the chip configuration
            q: The distinguished vertex (fire source)
        """
        self.graph = graph
        # Store a copy of the full configuration
        self.full_configuration = configuration.copy()
        # For convenience, store a separate configuration excluding q
        self.configuration = {v: configuration[v] for v in graph if v != q}
        self.q = q
        self.unburnt_vertices = set(self.configuration.keys())

    def outdegree_S(self, vertex, S):
        """
        Calculate the number of edges from a vertex to vertices in set S.

        Args:
            vertex: The vertex to calculate outdegree for
            S: Set of vertices to count edges to

        Returns:
            Sum of edge weights from vertex to vertices in S
        """
        return sum(self.graph[vertex][neighbor] for neighbor in self.graph[vertex] if neighbor in S)

    def send_debt_to_q(self):
        """
        Concentrate all debt at the distinguished vertex q, making all non-q vertices out of debt.
        This method modifies self.configuration so all non-q vertices have non-negative values.

        The algorithm works by performing borrowing moves at vertices in debt,
        working in reverse order of distance from q (approximated by BFS).
        """
```

```
        # Sort vertices by distance from q (approximation using BFS)
        queue = [self.q]
        visited = {self.q}
        distance_ordering = [self.q]

        while queue:
            current = queue.pop(0)
            for neighbor in self.graph[current]:
                if neighbor not in visited and neighbor in self.unburnt_vertices:
                    visited.add(neighbor)
                    queue.append(neighbor)
                    distance_ordering.append(neighbor)

        # Process vertices in reverse order of distance (excluding q)
        vertices_to_process = [v for v in reversed(distance_ordering) if v in self.unburnt_vertices]

        for v in vertices_to_process:
            # While v is in debt, borrow
            while self.configuration[v] < 0:
                # Perform a borrowing move at v
                vertex_degree = sum(self.graph[v].values())
                self.configuration[v] += vertex_degree

                # Update neighbors based on edge counts
                for neighbor, edge_count in self.graph[v].items():
                    if neighbor in self.configuration:
                        self.configuration[neighbor] -= edge_count

    def run(self):
        """
        Run Dhar's Algorithm to find a maximal legal firing set.

        This implementation uses the "burning process" metaphor:
        1. Start a fire at the distinguished vertex q
        2. A vertex burns if it has fewer chips than edges to burnt vertices
        3. Vertices that never burn form a legal firing set

        Returns:
            A set of vertices that form a maximal legal firing set
        """
        # First, ensure all non-q vertices are out of debt
        self.send_debt_to_q()

        # Initialize burnt set with the distinguished vertex q
        burnt = {self.q}
        unburnt = set(self.graph.keys()) - burnt

        # Continue until no new vertices burn
        changed = True
```

```python
        while changed:
            changed = False

            # Check each unburnt vertex to see if it should burn
            for v in list(unburnt):
                # Count edges from v to burnt vertices
                edges_to_burnt = sum(self.graph[v][neighbor]
                                     for neighbor in self.graph[v]
                                     if neighbor in burnt)

                # A vertex burns if it has fewer chips than edges to burnt vertices
                if v in self.configuration and self.configuration[v] < edges_to_burnt:
                    burnt.add(v)
                    unburnt.remove(v)
                    changed = True

        # Return unburnt vertices (excluding q) as the maximal firing set
        return unburnt - {self.q}
```

A.13.4 DollarGame Central Class Object

```python
# chip-firing-simulation/dollar_game.py
from algorithms.greedy_algorithm import GreedyAlgorithm
from algorithms.dhar_algorithm import DharAlgorithm
from utils.laplacian import Laplacian

class DollarGame:
    def __init__(self, graph, divisor):
        """
        Initialize the dollar game with a choice of algorithms.

        :param graph: A dictionary representing the adjacency list of the graph.
        :param divisor: A dictionary representing the wealth at each vertex.
        """
        self.graph = graph
        self.divisor = divisor
        self.laplacian = Laplacian(graph)

    def play_game(self, strategy="greedy", q=None):
        """
        Play the game using the specified strategy.

        :param strategy: "greedy" or "dhar" to choose the algorithm.
        :param q: The distinguished vertex for Dhar's algorithm.
        :return: Tuple (True, result) if the game is winnable; otherwise (False, None).
        """
        if strategy == "greedy":
            greedy_algo = GreedyAlgorithm(self.graph, self.divisor)
```

```
            return greedy_algo.play()

        elif strategy == "dhar":
            dhar_algo = DharAlgorithm(self.graph, self.divisor, q)
            legal_firing_set = dhar_algo.run()
            if legal_firing_set:
                return True, legal_firing_set
            else:
                return False, None

    def apply_laplacian(self, firing_script):
        """
        Apply the Laplacian matrix to the firing script to get the resulting
        ↪  divisor.

        :param firing_script: The firing script dictionary.
        :return: The resulting divisor after applying the Laplacian.
        """
        return self.laplacian.apply(self.divisor, firing_script)

```

A.13.5 Main File for Initial Testing & Execution

```
# chip-firing-simulation/main.py
from dollar_game import DollarGame

def main():
    graph = {
        'A': {'B': 1, 'C': 1, 'E': 2},
        'B': {'A': 1, 'C': 1},
        'C': {'A': 1, 'B': 1, 'E': 1},
        'E': {'A': 2, 'C': 1}
    }
    divisor = {'A': 2, 'B': -3, 'C': 4, 'E': -1}

    # Create a DollarGame instance
    game = DollarGame(graph, divisor)

    # Play with the greedy algorithm
    winnable, result = game.play_game(strategy="greedy")
    if winnable:
        print("The game is winnable with the greedy algorithm.")
        print("Firing Script:", dict(result))

        # Apply the Laplacian matrix to verify the result
        resulting_divisor = game.apply_laplacian(result)
        print("Resulting Divisor:", dict(resulting_divisor))
    else:
        print("The game is not winnable with the greedy algorithm.")

```

```python
    # Example of using Dhar's algorithm (consistent with example in write-up)
    divisor = {'A': 3, 'B': -2, 'C': 1, 'E': 0}
    q = 'B'  # Distinguished vertex for Dhar's algorithm

    # Create a DollarGame instance
    game = DollarGame(graph, divisor)

    # Play with Dhar's algorithm
    winnable, result = game.play_game(strategy="dhar", q=q)
    if winnable:
        print("The game is winnable with Dhar's algorithm.")
        print("Legal firing set:", result)
    else:
        print("The game is superstable with Dhar's algorithm.")

if __name__ == "__main__":
    main()
```

Appendix B

Additional Notes and Discussions

B.1 Proof of Validity for Greedy Algorithm

(Adapted from [2, §3.1])

B.1.1 Case 1: When a Solution Exists

Let us begin by examining scenarios where $D \in \mathrm{Div}(G)$ possesses a viable solution. Consider any non-negative divisor D' with the relationship $D \sim D'$, and identify a lending/borrowing pattern σ that accomplishes: $D \xrightarrow{\sigma} D'$.

We can simplify our analysis by shifting σ to ensure all its values are non-positive and that at least some vertices remain untouched by borrowing. Specifically, we define a set: $Z := \{v \in V : \sigma(v) = 0\}$. This means σ successfully transforms D into a non-negative state purely through strategic lending without requiring any vertex in Z to take on debt.

Armed with this insight about D, we act upon our greedy strategy. When D already has no debt, nothing needs to be done. Otherwise, we identify a vertex u where $D(u) < 0$. Since u has debt that needs clearing, and our transformation σ works, we know that $\sigma(u) < 0$ (indicating borrowing must occur at u). Our algorithm addresses this by performing a borrowing operation at u, which we track by incrementing $\sigma(u)$ by 1. When this brings $\sigma(u)$ to zero, we include u in our set Z. We continue this process iteratively.

Since all values in σ begin non-positive and each step increases their sum by exactly one unit, this process cannot continue indefinitely. It must eventually convert D into an equivalent non-negative divisor.

B.1.2 Case 2: When No Solution Exists

Consider when $D \in \text{Div}(G)$ lacks any viable solution. Examine any sequence $D_1, D_2, D_3, \ldots$ created by starting with $D = D_1$ and performing successive borrowing operations at vertices with debt. We need to demonstrate that, eventually, every vertex must participate in borrowing.

We first establish that for any vertex v and any step i in our process: $D_i(v) \leq \max\{D(v), \text{val}(v)-1\} := B_v$. This boundary exists because a vertex only borrows when in debt, and borrowing increases its value by exactly $\text{val}(v)$. By defining B as the maximum of all B_v across the graph, we universally ensure $D_i(v) \leq B$. With n representing the total vertex count, and knowing that $\deg(D_i) = \deg(D)$ remains constant throughout, we can derive for any vertex v:

$$\deg(D) = D_i(v) + \sum_{w \neq v} D_i(w) \leq D_i(v) + (n-1)B$$

This establishes both upper and lower boundaries for all divisor values: $\deg(D) - (n-1)B \leq D_i(v) \leq B$. These finite bounds mean the sequence of divisors D_i cannot produce infinitely many distinct states. Consequently, some divisors must appear twice in our sequence (i.e., $D_j = D_k$ for some $j < k$).

To complete our proof, we need a fundamental result about graph Laplacians. For any undirected, connected multigraph $G = (V, E)$, the kernel of its Laplacian matrix L is generated solely by the all-ones vector $\vec{1} \in \mathbb{Z}^n$. Equivalently, the kernel of the divisor homomorphism div (which equals the kernel of the Laplacian operator) consists precisely of constant functions $c : V \to \mathbb{Z}$.

We can prove this by showing that any function $\sigma : V \to \mathbb{Z}$ satisfying $\text{div}(\sigma) = 0$ must be constant. Let us choose a vertex $v \in V$ where σ reaches its maximum value $k \in \mathbb{Z}$. Since the divisor of σ is zero, we must have: $\text{val}(v)k = \sum_{vw \in E} \sigma(w)$.

However, since $\sigma(w) \leq k$ for all vertices w, this equality can only hold if $\sigma(w) = k$ for all vertices w adjacent to v. Since G is connected, we can extend this argument vertex by vertex throughout the graph, showing that σ must equal k at every vertex. Thus, $\sigma = k$ is a constant function.

Returning to our sequence where $D_j = D_k$, we know that the changes made between these two identical states must form a function in the kernel of the Laplacian. Based on the above result on Laplacians, we can conclude that the sequence of borrowing operations performed between steps j

and k must include every vertex in the graph.

B.2 Uniqueness Proof of Firing Script Produced by Greedy Algorithm

(Adapted from [2, §3.1])

We will use proof by contradiction. Assume that $\sigma_1 \neq \sigma_2$. Without loss of generality, we can assume there exists a vertex that borrows more times according to σ_2 than according to σ_1.

Let $\{w_1, \ldots, w_m\}$ be the sequence of borrowings corresponding to script σ_2. By our assumption, as we execute this borrowing sequence, there must be a first step k where vertex w_k borrows more than $|\sigma_1(w_k)|$ times.

Let $\tilde{\sigma}_2$ represent the firing script associated with the first $k-1$ steps of the σ_2-borrowing sequence, i.e., with $\{w_1, \ldots, w_{k-1}\}$.

Since $\sigma_1, \sigma_2 \leq 0$ (firing scripts have non-positive values), we have $\tilde{\sigma}_2 \geq \sigma_1$ and $\tilde{\sigma}_2(w_k) = \sigma_1(w_k)$.

Therefore, after performing the first $k-1$ steps of the σ_2-borrowing sequence, the amount of money at vertex w_k is:

$$
\begin{aligned}
(D - \text{div}(\tilde{\sigma}_2))(w_k) &= D(w_k) - \text{val}(w_k)\tilde{\sigma}_2(w_k) + \sum_{w_k u \in E} \tilde{\sigma}_2(u) && \text{(B.1)}\\
&\geq D(w_k) - \text{val}(w_k)\sigma_1(w_k) + \sum_{w_k u \in E} \sigma_1(u) && \text{(B.2)}\\
&= (D - \text{div}(\sigma_1))(w_k) && \text{(B.3)}\\
&= E_1(w_k) && \text{(B.4)}\\
&\geq 0 && \text{(B.5)}
\end{aligned}
$$

This shows that w_k has no debt after the first $k-1$ steps of the σ_2-borrowing sequence, which contradicts our assumption that w_k needs to borrow at the k-th step.

B.3 Proof of Validity for Dhar's Algorithm

When Dhar's Algorithm returns the set S, we can verify that $c(v) \geq \text{outdeg}_S(v)$ holds for every vertex $v \in S$. This condition confirms that S constitutes a valid legal firing set. In the special case where a configuration c is superstable, we know by definition that no non-empty subset of vertices can legally fire together, which means the algorithm must return an empty set S.

Conversely, we must prove that when the algorithm returns an empty set S, the configuration c is superstable. To establish this, we will demonstrate that any arbitrary non-empty subset U of vertices cannot form a legal firing set for c.

At initialization, the algorithm sets $S = \tilde{V}$ (the complete set of vertices). During the execution of the while-loop, vertices failing the firing condition are systematically removed one at a time, given our assumption that S becomes empty upon termination, and since U is non-empty, a moment must exist during the algorithm's execution when the first vertex from U is removed from S. Let us denote this vertex as u.

At the precise moment just before u is removed from S, two crucial conditions hold: first, U remains entirely contained within S (as u is the first element of U to be removed), and second, $c(u) < \mathrm{outdeg}_S(u)$ (the exact condition that triggers u's removal). The key insight here is that since $U \subseteq S$ at this point, we can establish that $\mathrm{outdeg}_S(u) \geq \mathrm{outdeg}_U(u)$.

Combining these observations, we arrive at the inequality $c(u) < \mathrm{outdeg}_S(u) \geq \mathrm{outdeg}_U(u)$, which simplifies to $c(u) < \mathrm{outdeg}_U(u)$. This inequality demonstrates that u lacks sufficient chips to fire along all its outgoing edges to vertices within U. Since u is an element of U, we have proven that U cannot function as a legal firing set.

As U was chosen arbitrarily, we have established that no non-empty subset of vertices can form a legal firing set when the algorithm returns an empty set. By definition, this confirms that the configuration c is superstable, thus completing our proof of the algorithm's validity.

B.4 A Peculiar Optimization of Winnability Determination Algorithm

When determining whether a divisor $D \in \mathrm{Div}(G)$ is winnable, we can employ a mathematical optimization known as the debt-reduction trick, which significantly improves computational efficiency. This approach leverages properties of the reduced Laplacian matrix to transform divisors into more manageable forms before applying standard algorithms.

Before detailing the optimization itself, we need to establish a critical mathematical property that forms its foundation:

Lemma B.4.1 (Invertibility of the Reduced Laplacian)**.** *If $G = (V, E)$ is an undirected, connected multigraph, then the kernel of its reduced Laplacian matrix $\tilde{L}$ is zero. Consequently, $\tilde{L}$ is invertible as a linear operator over $\mathbb{Q}^{n-1}$. [2, Corollary 2.15]*

Proof. The proof relies on understanding how the reduced Laplacian relates to the full Laplacian matrix. From the property of the kernel of the Laplacian matrix proven in section B.1, we know that the kernel of the full Laplacian L is one-dimensional, generated solely by the all-ones vector $\vec{1} \in \mathbb{Z}^n$. This fundamental property has an important structural implication: each column in L sums to zero, meaning the final column must be the negative sum of all other columns.

Since L is symmetric (a property of undirected graphs), this column relationship also translates to rows—the final row of L must be the negative sum of all previous rows. This symmetry creates an important relationship: any vector orthogonal to the first $n-1$ rows of L must also be orthogonal to the last row.

Now, let us consider what happens if $\tilde{a} \in \mathbb{Z}^{n-1}$ is in the kernel of the reduced Laplacian $\tilde{L}$ (which is formed by removing the row and column corresponding to our designated source vertex). By definition, this means $\tilde{L}\tilde{a} = 0$.

We can extend $\tilde{a}$ to a vector in the original space by appending a zero, giving $(\tilde{a}, 0) \in \mathbb{Z}^n$. This extended vector has the property that its dot product with each of the first $n - 1$ rows of L is zero. According to our earlier observation about row relationships, the dot product with the final row must also be zero.

Therefore, $(\tilde{a}, 0)$ is orthogonal to all rows of L, meaning it belongs to the kernel of L. However, we know that (from the property of the kernel of Laplacian matrix proven in section B.1) the only vectors in the kernel of L are scalar multiples of the all-ones vector. Since $(\tilde{a}, 0)$ has a zero in its last component, it can only equal the zero vector. This forces $\tilde{a} = 0$.

We have thus shown that the only vector in the kernel of $\tilde{L}$ is the zero vector, making the kernel trivial since we know that for a linear operator, having a trivial kernel is equivalent to being injective. Because $\tilde{L}$ is a square matrix operating on a finite-dimensional space $\mathbb{Q}^{n-1}$, injectivity implies surjectivity and, therefore, invertibility over the rational numbers. □

The fundamental insight for our optimization begins with decomposing any divisor D into the form $D = c + kq$, where $c \in \text{Config}(G)$ represents the configuration on non-source vertices, and

q is a designated source vertex. By fixing an ordering of vertices $v_1, \ldots, v_n$ with $q = v_1$, we can identify c with a vector in $\mathbb{Z}^{n-1}$. Our objective becomes finding a firing script that transforms c into an equivalent configuration with smaller coefficient values.

This transformation relies on the invertibility of the reduced Laplacian matrix $\tilde{L}$ that we just established. The optimization proceeds by computing $\tilde{L}^{-1}c$, which would theoretically yield a configuration with zeros at all non-source vertices. However, since firing scripts must be integer vectors, we approximate this ideal solution by taking the floor function of each component, defining $\sigma := \lfloor \tilde{L}^{-1}c \rfloor \in \mathbb{Z}^{n-1}$.

Applying this firing script produces an equivalent configuration $c' := c - \tilde{L}\sigma$ with remarkably small coefficients. This transformation ensures that $|c'(v)| < \deg_G(v)$ for all non-source vertices v, meaning any vertex in debt can be brought out with a single borrowing move. The original divisor D can then be replaced with the linearly equivalent divisor $D' := c' + (\deg(D) - \deg(c'))q$, which has substantially smaller coefficients.

This debt-reduction trick provides an optional but powerful pre-processing step before applying greedy algorithms or Dhar's algorithm for winnability determination. In practice, it reduces the number of iterations required in subsequent steps by transforming highly indebted configurations into ones where debt can be eliminated with minimal operations. The complete procedure first applies this debt-reduction optimization, then uses a greedy algorithm to eliminate debt at non-source vertices, and finally applies Dhar's algorithm repeatedly until the divisor is q-reduced.

The elegance of this approach lies in its mathematical foundation—leveraging linear algebra to optimize a graph-theoretic problem. By exploiting the invertibility of the reduced Laplacian, we transform what might otherwise require many iterative steps into a single matrix operation followed by minor adjustments, dramatically improving algorithmic efficiency. Our current Python implementation of Dhar's Algorithm A.13 does not implement this optimization for now. However, we are actively working on incorporating it in our chipfiring Python package (`https://pypi.org/project/chipfiring/`).

B.5 Formalization of Riemann Roch Corollaries and Supporting Results

The formal proof of Riemann–Roch in Lean4 opens the door to several corollaries and related theorems. Perhaps the most significant among these is the graph-theoretic version of *Clifford's theorem*. In algebraic geometry, Clifford's theorem gives a bound on the dimension of special linear systems on a curve. In the language of chip-firing on graphs, it translates to a bound on $r(D)$ when both D and its complement with respect to the canonical divisor have non-negative rank (4.4.2). The proof is an elegant application of Riemann–Roch and a convexity argument on ranks.

In Lean4, we proved Clifford's theorem by starting from the Riemann–Roch equality and the superadditivity of rank. The formal proof begins by substituting the Riemann–Roch formula for $r(D)$ in terms of $r(K - D)$, $\deg(D)$, and g. It then uses the fact that the canonical divisor K has rank $g - 1$ (Corollary 4.2.3) and the earlier proven inequality $r(D + D') \geq r(D) + r(D')$. The Lean proof proceeds as follows: we know $r(K) = g - 1$, and since $K = D + (K - D)$, we have $g - 1 = r(K) \geq r(D) + r(K - D)$ by rank superadditivity. Using the Riemann–Roch relation to express $r(K - D)$ in terms of $r(D)$, this inequality rearranges to $2r(D) \leq \deg(D)$ (details are given in the formal proof), which is precisely the desired $r(D) \leq \frac{1}{2}\deg(D)$. Thus, Lean4 confirms Clifford's theorem for graphs with the same clarity as the paper proof detailed in appendix A.8.

This result, now machine-checked, strengthens our chip-firing framework by delineating the limitations of how large $r(D)$ can be relative to $\deg(D)$ when D is special (in the sense that both D and $K - D$ are effective or at least equivalent to effective divisors). It mirrors the classical Clifford inequality for algebraic curves, thus further validating Baker and Norine's dictionary between algebraic divisors on curves and chip-firing divisors on graphs.

Beyond Clifford's theorem, we derived several corollaries that give a fuller picture of divisor theory on graphs. For example, we formalized the piecewise-linear behavior of the rank function (Corollary 4.4.3), which we verified in Lean as a corollary, classify divisors into three regimes: low degree (always special in the sense $r(D) = -1$ or small), mid-range degree (special divisors obeying Clifford's inequality), and high degree (non-special divisors for which Riemann–Roch is equality). The formal proofs of these corollaries in Lean4 are case analyses using the definitions and theorems we established. They serve as a consistency check for our framework: the extremal cases of Riemann–Roch match intuition and known results from classical theory.

B.6 Parallels to Riemann-Roch for Riemann Surfaces

The Riemann-Roch theorem for graphs shares a striking structural similarity with its classical counterpart in algebraic geometry, a foundational result in the study of Riemann surfaces. On a compact Riemann surface X of genus g, a divisor D is a formal integer linear combination of points on X, denoted as $D = \sum_{a \in X} n_a \cdot a$. The degree of D, $\deg(D)$, is the sum of these integers, while the genus g reflects the surface's topological complexity, often visualized as the number of "handles" on the surface. For a deeper exploration of this analogy, Baker and Norine's work [1] provides valuable insights.

In graphs, a divisor is an integer assignment across vertices, and the degree is the total sum of these integers. The genus g in graph theory is defined combinatorially as $g = |E| - |V| + 1$, equivalent to the cyclomatic number or the rank of the graph's cycle space. This measure of genus differs from the topological genus used in surface embeddings but serves a similar role in quantifying structural complexity. As noted by Corry and Perkinson [2]; graph theorists often refer to the cyclomatic number as the "genus" because it counts the number of independent cycles in the graph. However, in the context of this work, the term "genus" follows Baker and Norine's usage, aligning with the combinatorial definition central to the Riemann-Roch formula for graphs (Theorem 4.3.1).

The graph-theoretic Riemann-Roch theorem mirrors its classical counterpart in form but interprets concepts in a combinatorial framework. For any divisor D and the canonical configuration K, the theorem states that $r(D) - r(K - D) = \deg(D) - g + 1$, where $r(D)$ is the rank of D. This rank is defined in terms of chip-firing dynamics, which parallels the dimension of function spaces in algebraic geometry. The analogy extends beyond formal similarity: both settings involve a dualizing role for K, and both use the genus to capture structural complexity, albeit in different ways.

The distinction in genus interpretation highlights the theorem's adaptability across disciplines. While algebraic geometry defines genus topologically, the graph-theoretic genus quantifies the dimension of the cycle space—a purely combinatorial invariant. For instance, planar graphs have a topological genus of 0 since they can be embedded on a sphere without crossings, but their combinatorial genus may vary.

The structural parallels between these theorems have significant implications for interdisciplinary research. Researchers can generalize results from algebraic geometry by leveraging graph-theoretic Riemann-Roch, such as Osserman's work on Amini-Baker construction and the limit linear series for curves [16]. Conversely, insights from algebraic geometry can inform graph theory. This analogy enriches graph theory and provides algebraic geometry with novel computational tools, underscoring the power of mathematical abstraction to reveal profound symmetries across seemingly disparate fields.

Appendix C

PaperProof Generated Images for Proof Visualisation

Visualization is a key component of any proof that can help uncover some hidden insights. PaperProof [19] is a fantastic tool engineered to allow effective and interactive visualization of Lean4 proofs.

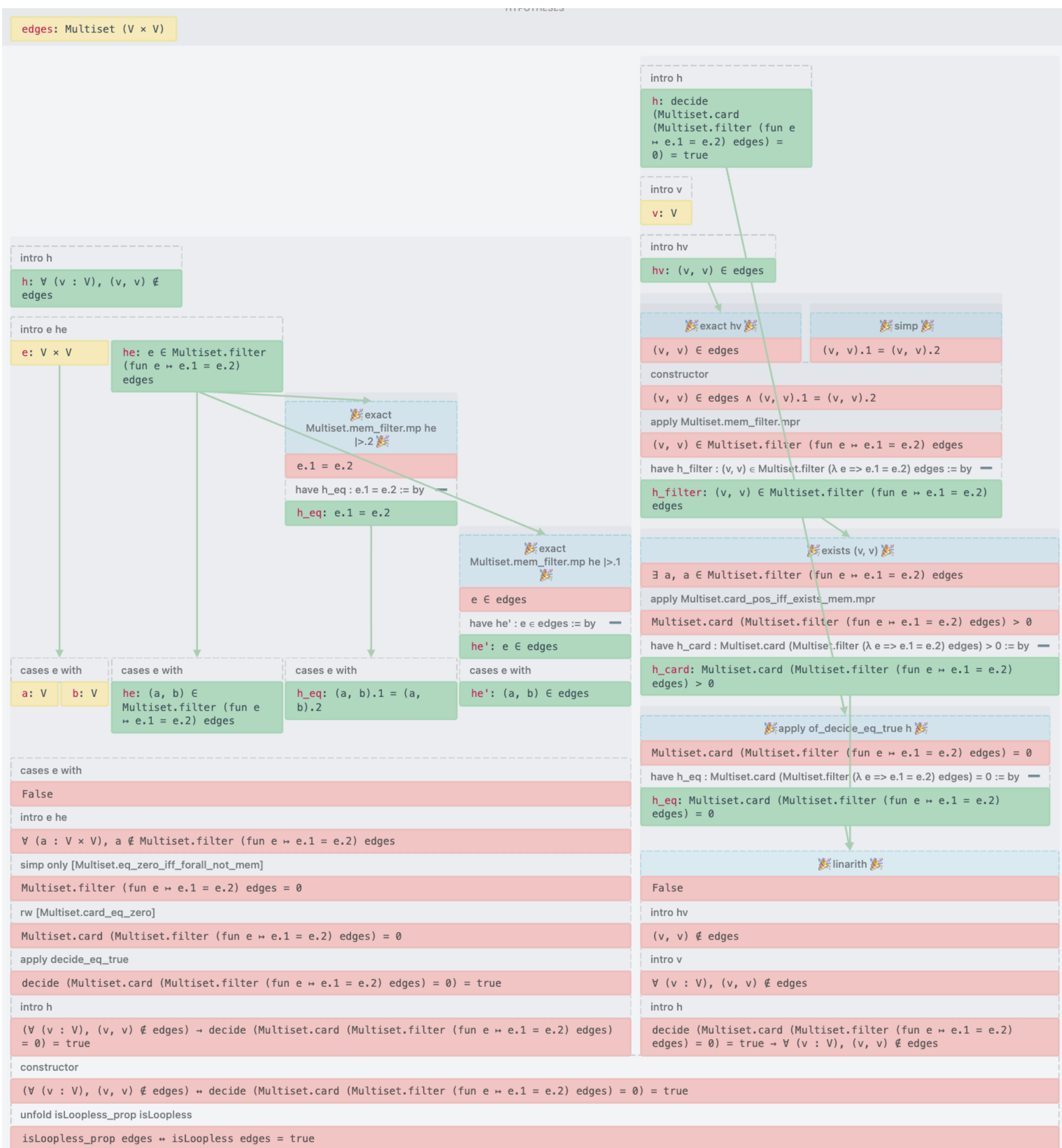

Figure C.1: PaperProof Visualisation for Loopless Boolean and Propositional Equivalence proof

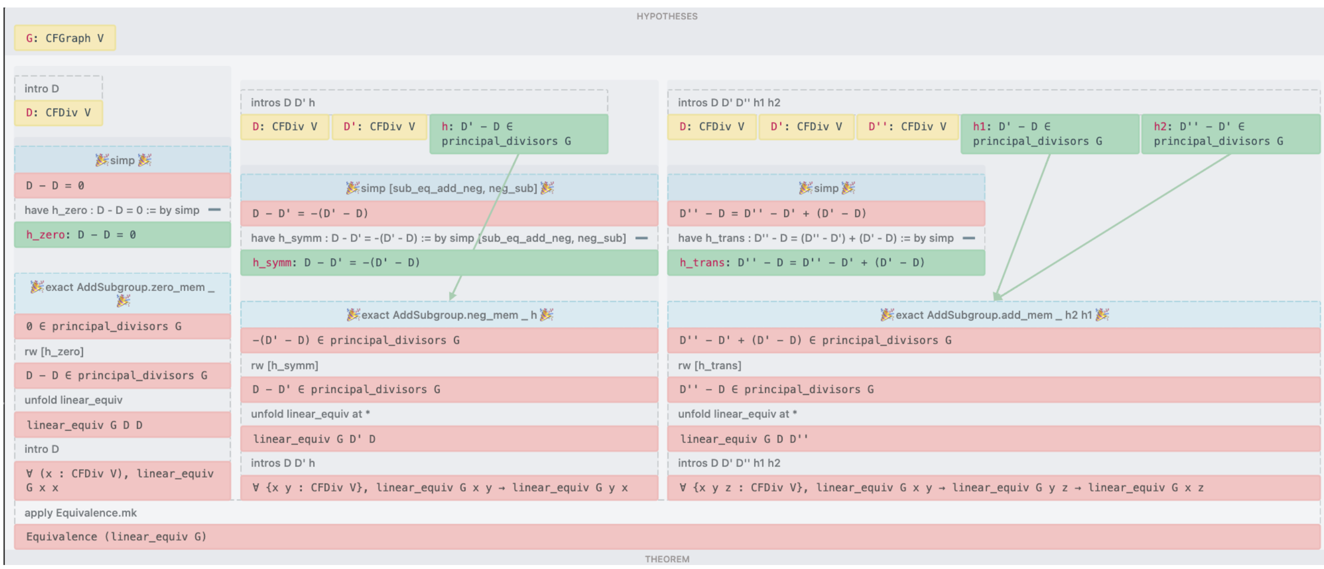

Figure C.2: PaperProof Visualisation for Linear Equivalence of Divisors is an Equivalence Relation proof

Corrections

When originally submitted, this honors thesis contained some errors which have been corrected in the current version. Here is a list of the errors that were corrected.

- In Figure 2.2, the final divisor mistakenly showed all vertices with wealth 0. This typographical error has been fixed: Alice's wealth is now correctly shown as 2.
- To improve clarity and consistency, the notation for vertex degree was changed from $\deg_G$ to val in approximately five instances.
- Approximately 4 minor corrections (p.20, p.23, p.26) to grammar and LaTeX typography were made throughout the thesis.