Degeneration of Prym Varieties: A computational approach to the indeterminacy locus of the Prym map and degenerations of cubic threefolds

by

Josh Frinak

B.A., University of Wisconsin–Eau Claire, 2012

M.A., University of Colorado Boulder, 2016

A thesis submitted to the

Faculty of the Graduate School of the

University of Colorado in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Mathematics

2018

This thesis entitled:
Degeneration of Prym Varieties: A computational approach to the indeterminacy locus of the Prym map and degenerations of cubic threefolds
written by Josh Frinak
has been approved for the Department of Mathematics

Prof. Sebastian Casalaina–Martin

Prof. Mathieu Dutour Sikiric

Date

The final copy of this thesis has been examined by the signatories, and we find that both the content and the form meet acceptable presentation standards of scholarly work in the above mentioned discipline.

Frinak, Josh (Ph.D., Mathematics)

Degeneration of Prym Varieties: A computational approach to the indeterminacy locus of the Prym map and degenerations of cubic threefolds

Thesis directed by Prof. Sebastian Casalaina–Martin

In this paper we will explore the extension of the Prym period map from the moduli space of admissible double covers of stable curves to the perfect cone compactification of the moduli space of principally polarized abelian varieties. We will use the insight from Casalaina-Martin, Grushevsky, Hulek, Laza, and Dutour Sikirić to understand the indeterminacy locus of this extension of the Prym map. Using computational methods we characterize the indeterminacy locus up to codimension 10 in the case where the base curves have genus 5. The last section will be devoted to an application of the Prym period map in which we construct the necessary extension data needed to classify the intermediate Jacobian of a cubic threefold with $2A_1$ singularity type.

Acknowledgements

I would like to thank my advisor Sebastian Casalaina–Martin for all of the encouragement and help he has given me along the way. The amount of time he has invested in me is nothing short of immaculate. He has been an inspiration in my struggles with math research. I would also like to thank my second advisor Mathieu Dutour Sikiric. He provided a lot of programming insight and without his direction I would have been lost several times.

I would like to thank my mathematical colleagues in the CU math department. The many late nights I spent in the math department building were rarely spent alone. The support I received from my fellow students gave me the courage to complete my degree.

Lastly, I would like to thank my friends and family. These were the people who gave me an outlet to spend time away from my mathematics and help me with the non-academic struggles of attending graduate school. Without them I am nothing.

Contents

Figures

Figure

Chapter 1

Introduction

In this thesis we consider the period map for Prym varieties. For motivation, the Torelli map $M_g \to A_g$ is a morphism from the moduli space of smooth curves of genus $g \geq 2$ to the moduli space of principally polarized abelian varieties of dimension g. This map is defined by assigning to a smooth curve of genus g its Jacobian $\operatorname{Pic}^0 X$. A long standing question in algebraic geometry is whether the map extends to compactifications of the moduli spaces. The canonical choice of compactification for M_g is the Deligne-Mumford compactification. In this thesis we will be concerned with three standard toroidal compactifications of the moduli space principally polarized abelian varieties: the second Voronoi, perfect cone, and central cone compactifications.

In [Nam76], Namikawa credits Mumford with proving that the Torelli map extends to a morphism $\overline{M}_g \to \overline{A}_g$ from the Deligne-Mumford compactification of M_g to the Second Voronoi compactification of A_g. In [AB12], Alexeev and Brunyate show that the extended Torelli map is regular in the case of the perfect cone compactification for all g and regular in the case of the central cone compactification for $g \leq 6$ but not regular for $g \geq 9$.

After the Torelli map, a natural next case to consider is the Prym map. One can associate a principally polarized abelian variety to a connected étale double covers of curves, called a Prym variety. This association gives the Prym period map

$$P_g : R_g \to A_{g-1}$$

where R_g is the moduli space of connected étale double covers of curves of genus g. Prym varieties provide a geometric approach to understanding higher dimensional principally polarized abelian varieties as the Prym map is dominant for $g-1 \leq 5$ (whereas the Torelli map is dominant for $g \leq 3$).

A normal crossings compactification of R_g was constructed by Beauville using admissible double covers of stable curves. Similar to the Torelli map, the question is whether the Prym period map extends to a regular map in the case of each Toroidal compactification of A_{g-1}. In summary of [FS86], the Prym map does not extend to a regular map in any of the standard toroidal compactifications of A_{g-1}.

In Alexeev–Birkenhake–Hulek [ABH02] and Vologodsky [Vol02], they described the indeterminacy locus in the case of the second Voronoi compactification, $\overline{R}_{g+1} \dashrightarrow \overline{A}_g^V$. More precisely, in [FS86], Friedman and Smith found some explicit examples of admissible double covers where the Prym map does not extend to a regular map. In [ABH02] the indeterminacy locus of $\overline{R}_{g+1} \dashrightarrow \overline{A}_g^V$ is identified as the closure of the locus that Friedman-Smith identified.

In this thesis we study the indeterminacy locus of the Prym map in the case of the perfect cone compactification. We study the Friedman-Smith degenerations where the Prym map does not extend to a morphism. A result of [FS86] was a partial classification of the indeterminacy locus which follows,

$$\overline{FS_2} \cup \overline{FS_3} \subseteq \mathrm{Ind}(P_g^P).$$

The set $\overline{FS_n}$ denotes the set of Friedman-Smith degenerations with n nodes (see section 5.6). In [CMGHL17b, Thm. 7.1] the authors gave a further classification of the indeterminacy locus

$$\overline{FS_2} \cup \overline{FS_3} \subseteq \mathrm{Ind}(P_g^P) \subseteq \overline{FS_2} \cup \overline{FS_3} \cup \delta\overline{FS_4} \cup \cdots \cup \delta\overline{FS_g}$$

where $\delta\overline{FS_n} = \overline{FS_n} - FS_n$. The main question we address in this thesis is whether or not the indeterminacy locus of the Prym period map for the perfect cone compactification is actually equal to $\overline{FS_2} \cup \overline{FS_3}$ [CMGHL17b, Que. 7.4]. For $g < 5$ this is true [CMGHL17b, Rem. 7.2]. In $g \geq 5$ the answer to this question is unknown, but has been established in the affirmative up to codimension 6 [CMGHL17b, Thm. 7.1]. The main result of the thesis regarding this question the following:

Theorem 1.0.1. *The indeterminacy locus of the Prym map $P_5 : \overline{R}_5 \dashrightarrow \overline{A}_4^P$, from the moduli of* *admissible double covers of genus* 5 *curves to the perfect cone compactification of the moduli of principally polarized abelian varieties of dimension* 4*, is, up to codimension* 10*, equal to the locus of degenerations of Friedman-Smith covers with* 2 *or* 3 *nodes in the base curve; i.e.,* $\overline{FS_2} \cup \overline{FS_3}$*.*

The technique is computational, and in short, our goal was to write a computer program that could identify the indeterminacy locus. The strategy is as follows. In [CMGHL17b], the authors used Hodge theory to approach the problem of extending the Prym map. This technique allows us to determine the conditions for extending the map through the use of monodromy cones ([CMGHL17b][§4]). Calculating monodromy cones reduces the geometric aspects of the problem to purely combinatorial methods. In other words, we are able to determine whether or not an admissible cover lies in the indeterminacy locus entirely based on properties of its dual graph. The computer program implements this, and computes the monodromy cone for any given admissible cover. A program due to Mathieu Dutour Sikirić determines whether the monodromy cone lies in the perfect cone decomposition; this program was devoloped using similar techniques as [DSHS15] and [DSSV08]. Thus the question is reduced to enumerating all the dual graphs of admissible double covers of a given genus, and then implementing these programs. At this point, the main obstruction to obtaining a complete result in a given genus $g \geq 5$ is that there are too many admissible covers. We are able to reduce the number of admissible covers we need to consider in a few ways, but not enough to completely answer the question. Since we are only interested in the dual graph, and the dimension of a stratum in $\overline{\mathcal{R}}_g$ corresponding to covers of curves with dual graphs of a given type has codimension equal to the number of edges in the dual graph of the base curve, we choose to focus on enumerating the covers by the edges in the base curve. This also gives some partial results for larger g (see Theorem 12.2.2).

Chapter 2

Overview

Given a dual graph $\widetilde{\Gamma}$ of an admissible étale double cover of a curve $\widetilde{C}/C$ we first find $H_1(\widetilde{\Gamma}, \mathbb{Z})$ (section 3.3). The cover implicitly comes with an admissible involution ι and from this involution we can calculate the eigenspaces of the action of ι on $H_1(\widetilde{\Gamma}, \mathbb{Z})$. We denote these eigenspaces $H_1(\widetilde{\Gamma}, \mathbb{Z})^{\pm}$ respectively which follows the notation of [CMGHL17b]. From these eigenspaces we can calculate specific quotients of $H_1(\widetilde{\Gamma}, \mathbb{Z})$; mainly

$$H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} = H_1(\widetilde{\Gamma}, \mathbb{Z})/H_1(\widetilde{\Gamma}, \mathbb{Z})^{+}.$$

This quotient will be represented as a matrix whose rows correspond to linear forms, which upon squaring give quadratic forms that define the extremal rays of the monodromy cone (section 5.7).

In terms of implementing this we begin by enumerating all possible dual graphs of a base curve C. This means generating all graphs of fixed number of edges, vertices, and loops up to isomorphism (section 7). For each possible dual graph we enumerate all possible dual graphs of admissible étale double covers $\widetilde{C}/C$ up to isomorphism over the base dual graph (section 8). This involves checking potential double covers for admissibility (section 5). Next we select only the étale double covers which are Friedman-Smith degenerations of order 4 or higher (section 8.5). Finally, for each Friedman-smith degeneration dual graph of $\widetilde{C}/C$ with order greater than or equal to 4, we calculate the monodromy cone (section 6.3). Mathieu Dutour Sikirić inputs these monodromy cones into his program which determines whether or not each $\widetilde{C}/C \in \overline{FS}_n$, for $n \geq 4$, lies in the indeterminacy locus of the extension of the Prym period map in the perfect cone case.

The main difficulty in obtaining results in higher genus is twofold. As we increase the genus of the curve, the dimensions of the base graph grow linearly with respect to the genus (see section 10). This is problematic because the number of dual graphs of double covers over the base graph will grow exponentially with respect to the number of edges and vertices. When considering graphs with 10 edges we already have too many possible admissible covers to feasibly handle. One possible solution to this problem would be a way of finding reductions in the sample size of base graphs we are considering or further reducing the admissible covers for each base graph.

The second obstacle to handling higher genus curves is the isomorphism testing of each dual graph. We require the isomorphism testing of all graphs in order to limit the number of monodromy cone computations we perform and to significantly reduce the size of our output files. Without isomorphism testing larger dimensional cases would result in output files too large to be stored on a common laptop. There is a quasi-polynomial time algorithm to perform isomorphism tests (see [Bab15]) but the runtime expense of isomorphism testing becomes much worse in higher dimensions and is not feasible with the quantity of graphs we consider in such dimensions. The best isomorphism testing programs – Nauty, Bliss, or Trace – typically work very well but their worst runtime scenarios are exponential. The only solution to this beyond reducing the number of tests one needs to make would be to find a way of constructing unique members of isomorphism classes.

It is worth mentioning that a successful attempt was made to write this program in Java. Java could provide faster compilation times than Python but one major obstacle prevented us from getting comparatively high dimensional results like we obtained through Python – we only reached a complete calculation of 7 edges in Java vs 9 edges in Python. There are currently no good implementations of optimal graph isomorphism testing algorithms in Java. We were able to write our own algorithm in Java but it did not compare to the runtimes of the cutting edge techniques of isomorphism test implementations such as Nauty which is easily available through the SAGE library in Python. Python, via SAGE, also provides a lot of useful tools for handling graphs that allows us to outsource some of the tedious aspects of our code to a well tested library.

We do find partial results in higher dimensions – 10 and 11 edge base graphs – which will provide more intuition on making a conjecture for the full genus 5 case. We completely find the monodromy cones for all 9 edge base graphs and below. This will give us the result up to codimension 10.

We begin this thesis by discussing graph fundamentals in section 3. This section will include the definition of a graph, the incidence matrix of a graph, and other basic notations. In section 4 we go over the implementation of graphs in our program. This will be done by a `Python` class called `EGraph` . In section 5 we give the mathematics of an admissible cover of a dual graph. This section includes all the essential machinery to computing the monodromy cone along with examples. In section 6 we discuss the implementation of defining admissible cover graphs in our program. These objects will be stored in a class called `CoverGraph` which will resemble the `EGraph` class. Section 7 will be a step by step overview of the process of generating all base graphs of a specific dimension. There will be two methods of generating base graphs. The first will be a suboptimal approach used to generate all graphs of 3 edges. Then we will optimize our enumeration by recursively generating all graphs of higher dimension. In section 8 we enumerate all admissible cover graphs of a specific base graph. This complicated body of programming is carefully dissected to give the reader a thorough understanding of how to generate all possible covers of a base graph, test which covers are admissible, and finally test cover graph isomorphisms. In section 9 we carefully go over an example of calculating a monodromy cone of basic low dimensional dual graph and an étale double cover. Section 10 is an explanation of how we associate dimensions to a dual graph of a curve of fixed dimensions. This important section will help compute the necessary dimensions to completely answer the indeterminacy locus question over curves of a fixed genus. In section 11 we discuss a possible future direction for the project in which we only enumerate Friedman-Smith covers. In section 12 we give the conclusion of our calculations and explain some of the limiting factors. Finally, in section 13 we discuss an application of the Prym period map which we use to calculate the intermediate Jacobians of cubic threefolds with singularity type $2A_1$.

Chapter 3

Notation for graphs

3.1 Definition of a graph

Following Serre's notation in [Ser03, § 2.1], a **graph** Γ will consist of the data

$$(\overrightarrow{E} \underset{t}{\overset{s}{\rightrightarrows}} V, \overrightarrow{E} \xrightarrow{\tau} \overrightarrow{E}),$$

where V and $\overrightarrow{E}$ are sets, τ is a fixed-point free involution, and s and t are maps satisfying $s(\overrightarrow{e}) = t(\tau(\overrightarrow{e}))$ for all $\overrightarrow{e} \in \overrightarrow{E}$. The maps s and t are called the **source** and **target** maps respectively. We call $V =: V(\Gamma)$ the set of **vertices**. We call $\overrightarrow{E} =: \overrightarrow{E}(\Gamma)$ the set of **oriented edges**. We define the set of **(unoriented) edges** to be $E(\Gamma) = E := \overrightarrow{E}/\tau$. An **orientation of an edge** $e \in E$ is a representative for e in $\overrightarrow{E}$; we use the notation $\overrightarrow{e}$ and $\overleftarrow{e}$ for the two possible orientations of e. An **orientation of a graph** Γ is a section $\phi : E \to \overrightarrow{E}$ of the quotient map. An **oriented graph** consists of a pair (Γ, ϕ) where Γ is a graph and ϕ is an orientation. Given an oriented graph, we say that $\phi(e)$ is the **positive orientation** of the edge e. Given a subset $S \subseteq E$, we define $\overrightarrow{S} \subseteq \overrightarrow{E}$ to be the set of all orientations of the edges in S. A graph Γ is said to be **finite** if $\overrightarrow{E}$ and V are finite sets. Give an unoriented edge $e \in E(\Gamma)$, we define the set of endpoints of the edge e to be $\{s(\vec{e}), t(\vec{e})\}$ where $[\vec{e}] = e$, and each element of that set is called an endpoint of e.

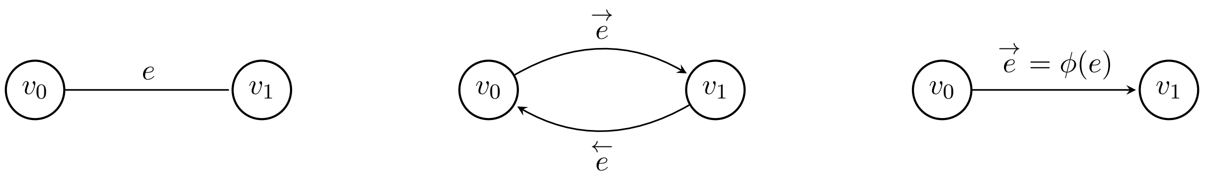

Figure 3.1: Converting unoriented graph into an oriented graph

The left and center figures give two depictions of the same graph Γ, one showing the unoriented edges $E(\Gamma)$ (left), and the other showing the oriented edges $\vec{E}(\Gamma)$ (center). The figure on the right depicts an oriented graph (Γ, ϕ), obtained by a choice of orientation of the graph Γ.

3.2 Morphism of graphs

Given two graphs Γ_1 and Γ_2 a **graph morphism** $f : \Gamma_1 \to \Gamma_2$ is the data of two maps, $f_{\vec{E}} : \vec{E}(\Gamma_1) \to \vec{E}(\Gamma_2)$ between the set of oriented edges and $f_V : V(\Gamma_1) \to V(\Gamma_2)$ between the set of vertices, such that the following two diagrams commute:

$$\begin{tikzcd} \vec{E}(\Gamma_1) \arrow[r, "f_{\vec{E}}"] \arrow[d, "s"'] & \vec{E}(\Gamma_2) \arrow[d, "s"'] \\ V(\Gamma_1) \arrow[r, "f_V"'] & V(\Gamma_2) \end{tikzcd} \qquad \begin{tikzcd} \vec{E}(\Gamma_1) \arrow[r, "f_{\vec{E}}"] \arrow[d, "t"'] & \vec{E}(\Gamma_2) \arrow[d, "t"'] \\ V(\Gamma_1) \arrow[r, "f_V"'] & V(\Gamma_2). \end{tikzcd}$$

To a morphism of graphs, one obtains an morphism on unoriented edges $f_E : E(\Gamma_1) \to E'(\Gamma_2)$ by $f_E([\vec{e}]) = [f_{\vec{E}}(\vec{e})]$.

A **morphism of oriented graphs** $f : (\Gamma_1, \phi_1) \to (\Gamma_2, \phi_2)$ is a morphism of graphs such that $f_{\vec{E}} \circ \phi_1 = \phi_2 \circ f_E$.

Example 3.2.1. Let Γ be the graph depicted in Figure 3.1. There is a morphism of graphs $f : \Gamma \to \Gamma$ given by $f_V(v_0) = v_1$, $f_V(v_1) = v_0$, $f_{\vec{E}}(\vec{e}) = \overleftarrow{e}$, and $f_{\vec{E}}(\overleftarrow{e}) = \vec{e}$. Note that at the level of unoriented edges, this fixes the edge e. This assignment does **not** define a morphism of oriented graphs $(\Gamma, \phi) \to (\Gamma, \phi)$. The only morphism of graphs $f : \Gamma \to \Gamma$ that induces a morphism of oriented graphs is the identity map.

3.3 Homology of a graph

Given a ring R, let $C_0(\Gamma, R) = \vec{C}_0(\Gamma, R)$ be the free R-module with basis $V(\Gamma)$ and $\vec{C}_1(\Gamma, R)$ be the R-module generated by $\vec{E}(\Gamma)$ with the relations $\overleftarrow{e} = -\vec{e}$ for every $e \in E(\Gamma)$. If we fix an orientation, then a basis for $\vec{C}_1(\Gamma, R)$ is given by the positively oriented edges; this induces an isomorphism with the usual group of 1-chains on the simplicial complex associated to Γ. These modules may be put into a chain complex. Define a boundary map ∂ by

$$\partial : \overrightarrow{C}_1(\Gamma, R) \longrightarrow \overrightarrow{C}_0(\Gamma, R) = C_0(\Gamma, R)$$
$$\overrightarrow{e} \mapsto t(\overrightarrow{e}) - s(\overrightarrow{e}).$$

We will denote by $H_\bullet(\Gamma, R)$ the groups obtained from the homology of $\overrightarrow{C}_\bullet(\Gamma, R)$. The homology groups $H_\bullet(\Gamma, R)$ coincide with the homology groups of the topological space associated to Γ.

Remark 3.3.1. Let $f : \Gamma_1 \to \Gamma_2$ be a morphism of graphs and let $\sigma : \triangle^n \to \Gamma_1$ be an n complex in Γ_1, for graphs $n = 0$ or $n = 1$. We define $f_\# : C_n(\Gamma_1) \to C_n(\Gamma_2)$ as follows,

$$f_\# \left(\sum_i \eta_i \sigma_i \right) = \sum_i \eta_i f \circ \sigma_i .$$

This will be a chain map; that is, $\partial f_\# = f_\# \partial$ and the following diagram commutes.

$$\begin{tikzcd}
0 \arrow[r] & C_0(\Gamma_1) \arrow[r, "\partial"] \arrow[d, "f_\#"'] & C_1(\Gamma_1) \arrow[r] \arrow[d, "f_\#"'] & 0 \\
0 \arrow[r] & C_0(\Gamma_2) \arrow[r, "\partial"'] & C_2(\Gamma_2) \arrow[r] & 0
\end{tikzcd}$$

Therefore $f_\#$ takes n-cycles to n-cycles and n-boundaries to n-boundaries. Furthermore, $f_\#$ induces maps $H_n(f) : H_n(\Gamma_1) \to H_n(\Gamma_2)$.

3.4 Incidence matrix of an oriented graph

Given a finite oriented graph (Γ, ϕ), and vertex and edge sets $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$, one obtains the associated $n \times m$ incidence matrix A, with entries a_{ij} as follows,

$$a_{ij} = \begin{cases} -1 & s(\phi(e_j)) = v_i \text{ and } e_j \text{ not a loop,} \\ 1 & t(\phi(e_j)) = v_i \text{ and } e_j \text{ not a loop,} \\ 0 & \text{else.} \end{cases}$$

In other words, one labels the rows of the matrix A by the vertices of the graph, and the columns by the oriented edges of the graph, and then for edges that are not loops, one enters $-1, 1, 0$ depending on whether an edge starts, ends, or does not contain a given vertex, respectively. For loops, we leave the column corresponding to the loop edge identically zero. This will be important for calculating homology. In implementing the graphs we will need to store information on which vertex each loop belongs to but this will be discussed later.

Example 3.4.1. Consider the following graph with 3 vertices and 3 edges with one edge being a loop.

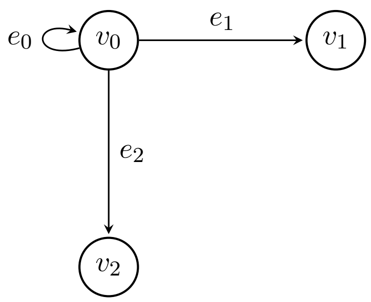

Figure 3.2: Basic Graph with Loop

This base graph has the following incidence matrix. Notice that the zero-column (first column but indexed at 0) has all zero entries. This indicates that e_0 is a loop.

$$A = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \end{array} \begin{array}{c} \begin{array}{ccc} e_0 & e_1 & e_2 \end{array} \\ \begin{bmatrix} 0 & -1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

An abstract incidence matrix is defined to be a matrix with all entries $0, 1, -1$, and such that in each column, the entries are either all 0, or exactly two entries are nonzero, with one equal to 1 and the other equal to -1.

$$IM : \{\text{finite oriented graphs with enumerated vertices and edges}\} \longrightarrow \{\text{incidence matrices}\}$$

For brevity, we will call these **finite oriented enumerated graphs**.

Remark 3.4.2. Note that a graph gives rise to an incidence matrix, but the incidence matrix forgets the data of the loops. For computing cohomology, this is actually useful for us. If we want to go the other direction, from incidence matrices to oriented graphs, we need the extra information which indicates where loops are located (see 7.1).

3.5 Connectedness

We will make use of the map

$$IC : \{\text{graphs}\} \longrightarrow \mathbb{Z}_2$$

that is 1 if the graph is connected, and 0 otherwise. We have $IC(\Gamma) = 0$ unless $\operatorname{rank} H_0(X, \mathbb{Z}) = 1$.

3.6 Loops in graphs

Given a graph Γ, consider the map

$$E(\Gamma) \to \mathbb{Z}_2$$

that is 1 if an edge is a loop, and 0 otherwise. For instance, given the incidence matrix A of a finite graph, $e \mapsto 1$ if and only if the corresponding column is a zero column. This identification will be useful when constructing admissible covering graphs. We will have to handle the covering of loops in a very specific way (see section 5.4).

3.7 Integral bases of homology

We will be interested in maps

$$H_1B : \{\text{finite oriented enumerated graphs}\} \longrightarrow \{\text{free } \mathbb{Z}\text{-modules of finite rank with a basis}\}$$

sending a graph to $H_1(\Gamma, \mathbb{Z})$ together with an integral basis. We will construct a particular map later.

Chapter 4

Implementing graphs: The EGraph class

We choose to write to program in Python primarily because of SageMath. SageMath is a mathematical software that builds upon many open sourced Python math packages. In our case we were primarily interested in the Sage packages dealing with Graphs. Sage has a graph class that has many useful methods. The one with the most upside for us was its Nauty based graph isomorphism testing [MP13]. We will be generating a lot of graphs and will only be interested in isomorphism classes of graphs.

Building off of the graph class in Sage we need to develop an extended graph class, `EGraph`. One of the reasons we need to extend the graph class is because we need to be able to impose a direction on our graph in order to calculate Homology (see 3.7). Another reason we need to extend the graph class in Sage is because we need a convenient way to store information about loops. The incidence matrix function in the sage graph class returns a matrix with columns containing exactly one -1 and one 1. This is acceptable for graphs with no loops but when calculating the homology of the graph it is crucial that we have zero columns corresponding to loops.

The `EGraph` class consist of two primary initial objects

- A DiGraph `G` (directed graph object from Sage)
- a 2-dimensional array of integers called `IM` representing the incidence matrix which was discussed in section 3.4.

To construct an instance of `EGraph` you would call the constructor which has three inputs:

The DiGraph `G`, the incidence matrix `IM`, and a dictionary called `Configs`. The dictionary stores the information on number of loops, edges and vertices in the graph. Upon initialization we calculate the homology of the base graph. This is done by taking a basis of the right kernel of the incidence matrix using methods from the Sage matrix classes.

```
self.Homology_Basis = Matrix(self.IM).right_kernel().basis()
```

To further explain the graph class we will consider the following example from above.

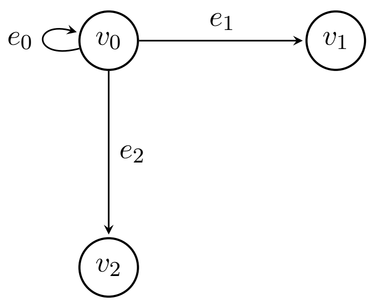

Figure 4.1: Basic Graph with Loop

This basic graph will have incidence matrix (§3.4) as follows,

$$\texttt{Incidence\_Matrix} = \begin{array}{c} \\ v_0 \\ v_1 \\ v_2 \end{array} \begin{array}{c} \begin{array}{ccc} e_0 & e_1 & e_2 \end{array} \\ \begin{bmatrix} 0 & -1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \end{array}$$

Throughout each iterative run of the program we will be fixing the values for loops, edges, and vertices. This information will be collected through command line arguments at the time of compiling and then stored into a dictionary named `congfigs`. In the above example we have three vertices, three edges, and one of the edges is a loop. Therefore the command to run the program would look like

```
sage -python GetBaseGraphs.py 3 3 1
```

and configs would look like,

```
configs = {‘‘verts’’ : 3, ‘‘edges’’ : 3, ‘‘loops’’ :1}
```

The reason for the sage portion of the run command is that we must run the Python program in a SAGE shell.

Remark 4.0.1. The reason for fixing the number of edges, vertices, and loops is not transparent at the moment. I will mention that for the purpose reducing the sample space of graph isomorphism checking it will be useful to restrict our attention to graphs of fixed dimensions.

Before we can create an object of EGraph we need to know how to create an object of the graph class in Sage. To do this we need to tell the compiler how many vertices we are working with, that we are allowing loops in the base graph, and that multiedges (multiple edges between two vertices) are allowed.

```
G = DiGraph(configs["verts"],loops=true, multiedges=true)
```

The above will initialize an instance of the Sage DiGraph class. Then we are free to add edges to the respective instance. Referring to the above example we need to add three edges: one edge is a loop on vertex 0, another edge is between vertex 0 and vertex 1, the final edge is between vertex 0 and vertex 2. In Python this would look like,

```
G.add_edge(0,0,0)
G.add_edge(0,1,1)
G.add_edge(0,2,2)
```

You can see that the first entry of the DiGraph class function add_edge is the starting vertex, the second entry is the terminal vertex, and the final entry is the label. In our case we will use the label to enumerate the edges of the graph.

To initialize an instance of the EGraph class with the above incidence matrix and loop information, we will call the EGraph class constructor as follows,

```
E = EGraph(G,IM,configs)
```

this will create a new instance of the `EGraph class` called `E` and implicitly calculate a first homology basis, `Homology_Basis`. To access the first homology basis we would make a reference to the `Homology_Basis` object of the `EGraph class` instance. In the above example we would have,

```
E.Homology_Basis = [(1,0,0)].
```

Observe that `Homology_Basis` is a list of tuples in Python. In the example we have one basis vector corresponding to the loop at vertex 0.

Chapter 5

Admissible covers of graphs

In this section we introduce the notions of admissible involutions of graphs, and admissible (double) covers of graphs. This is the graph theoretic analogue of admissible involutions and admissible double covers of stable curves.

5.1 Admissible involutions of graphs

Definition 5.1.1 (Admissible involution)**.** Let $\widetilde{\Gamma}$ be a finite graph; then an **admissible involution of $\widetilde{\Gamma}$** is a graph morphism $\iota : \widetilde{\Gamma} \to \widetilde{\Gamma}$ (see section 3.2) such that $\iota^2 = \mathrm{Id}$ with the stipulation that if ι_E fixes an unoriented edge $\tilde{e} \in E(\widetilde{\Gamma})$ then the endpoints of $\tilde{e}$ must be fixed by ι_V. An **admissible involution of an oriented, finite graph** $(\widetilde{\Gamma}, \phi)$ is a oriented graph morphism $\iota : (\widetilde{\Gamma}, \tilde{\phi}) \to (\widetilde{\Gamma}, \tilde{\phi})$ such that the induced morphism of graphs is an admissible involution.

See §5.3 for some examples.

Remark 5.1.2. An admissible involution of a stable curve gives rise to an admissible involution of the dual graph of the curve. Conversely, given an admissible involution of a connected finite graph, there exists an admissible involution of a stable curve whose dual graph is the original graph, and such that the induced involution of the dual graph is the initial admissible involution [CMGHL17b].

While dual graphs of curves do not come equipped with an orientation, it is frequently convenient to choose an orientation. We now translate the definition of admissible involution of a graph into the language of oriented graphs.

Lemma 5.1.3. *Let $(\widetilde{\Gamma}, \tilde{\phi})$ be a finite oriented graph. If $\iota : (\widetilde{\Gamma}, \tilde{\phi}) \to (\widetilde{\Gamma}, \tilde{\phi})$ is an oriented involution (a morphism of oriented graphs that is an involution), then the induced morphism of graphs $\iota : \widetilde{\Gamma} \to \widetilde{\Gamma}$ is an admissible involution; i.e., ι is an admissible involution of the oriented graph $(\widetilde{\Gamma}, \tilde{\phi})$.*

Conversely, given an admissible involution of graphs $\iota : \widetilde{\Gamma} \to \widetilde{\Gamma}$, there is an orientation $\tilde{\phi}$ of $\widetilde{\Gamma}$ such that ι induces an admissible involution of the oriented graph $(\widetilde{\Gamma}, \tilde{\phi})$.

Proof. Let $(\widetilde{\Gamma}, \tilde{\phi})$ be a finite oriented graph, and $\iota : (\widetilde{\Gamma}, \tilde{\phi}) \to (\widetilde{\Gamma}, \tilde{\phi})$ be an oriented involution (a morphism of oriented graphs that is an involution). To check that ι induces an admissible involution of the graph $\widetilde{\Gamma}$, we must check that if ι_E fixes an unoriented edge $\tilde{e} \in E(\widetilde{\Gamma})$ then the endpoints of $\tilde{e}$ must be fixed by ι_V. Let $\tilde{e} \in E(\widetilde{\Gamma})$ be such that $\iota_E(\tilde{e}) = \tilde{e}$. Then we have:

$$\iota_V(s(\tilde{\phi}(\tilde{e}))) = s(\iota_{\overrightarrow{E}}(\tilde{\phi}(\tilde{e}))) = s(\tilde{\phi}(\tilde{e}));$$

the first equality is from the definition of a morphism of graphs, and the second is from the definition of a morphism of oriented graphs. Similarly, $\iota_V(t(\tilde{\phi}(\tilde{e}))) = t(\tilde{\phi}(\tilde{e}))$. Therefore, ι defines an admissible involution of the graph $\widetilde{\Gamma}$.

Conversely, suppose we are given a graph $\widetilde{\Gamma}$ and an admissible involution $\iota : \widetilde{\Gamma} \to \widetilde{\Gamma}$. We can construct an orientation on $\widetilde{\Gamma}$ preserved by ι in the following way. Roughly speaking, we take the quotient graph Γ of $\widetilde{\Gamma}$ determined by the involution, choose an arbitrary orientation of Γ, and then lift that orientation back up to $\widetilde{\Gamma}$.

In more detail: we may construct a new graph $\Gamma = \widetilde{\Gamma}/\iota$ as follows: We set $V(\Gamma) = V(\widetilde{\Gamma})/\iota_V$ and $\overrightarrow{E}(\Gamma) = \overrightarrow{E}(\widetilde{\Gamma})/\iota_{\overrightarrow{E}}$. We define the source and target maps similarly. Pick an arbitrary orientation ϕ on Γ. We will essentially pull back the orientation ϕ along the quotient map $\pi_E : E(\widetilde{\Gamma}) \to E(\Gamma)$ to get an orientation $\tilde{\phi}$ on $\widetilde{\Gamma}$. To explain this, let $[\tilde{e}] \in E(\widetilde{\Gamma})$ then there exist an $[e] \in E(\Gamma)$ such that $\pi_E([\tilde{e}]) = [e]$, by the surjectivity of π_E. If $[\tilde{e}]$ has endpoints $\tilde{u}, \tilde{v} \in V(\widetilde{\Gamma})$ then $\tilde{u} \in \pi_V^{-1}(s(\phi(e)))$ or $\tilde{u} \in \pi_V^{-1}(t(\phi(e)))$. If $\tilde{u} \in \pi_V^{-1}(s(\phi(e)))$ then let $\tilde{u} = s(\tilde{\phi}(\tilde{e}))$ or if $\tilde{u} \in \pi_V^{-1}(t(\phi(e)))$ let $\tilde{u} = t(\tilde{\phi}(\tilde{e}))$. Do the same for $\tilde{v}$. This will assign a direction to $[\tilde{e}]$ and hence an orientation $\tilde{\phi}$ on $\widetilde{\Gamma}$.

Next we claim that ι preserves the orientation $\tilde{\phi}$ of $\widetilde{\Gamma}$. Let $\tilde{e} \in E(\widetilde{\Gamma})$ then $\pi(\iota_E(\tilde{e})) = \pi(\tilde{e})$. Therefore by the explicit construction of the orientation of $\widetilde{\Gamma}$ we have $\pi(s(\iota_{\overrightarrow{E}}(\tilde{\phi}(\tilde{e})))) = \pi(s(\tilde{\phi}(\tilde{e})))$

and therefore $\iota_V(s(\tilde{\phi}(\tilde{e}))) = s(\iota_E(\tilde{\phi}(\tilde{e})))$. By similar reasoning, $\iota_V(t(\tilde{\phi}(\tilde{e}))) = t(\iota_E(\tilde{\phi}(\tilde{e})))$. From section 3.2 we know that $\iota : \widetilde{\Gamma} \to \widetilde{\Gamma}$ is a graph morphism that preserves the orientation of $\widetilde{\Gamma}$. □

Remark 5.1.4. In light of Lemma 5.1.3, from now on, when discussing admissible involutions of graphs, we we always assume we have an oriented graph, in which case the admissible involution is equivalent to an involution of the oriented graph.

Remark 5.1.5. In the proof of lemma 5.1.3 we are allowed to choose an arbitrary orientation of the base graph Γ which will then make the inherited orientation of $\widetilde{\Gamma}$ lead to an orientation preserving involution ι. This will be extremely useful in programming the implementation of the cover graph. Namely, once we enumerate the vertices of the base graph Γ, this will induce an enumeration of the edges of Γ, as well as an orientation of Γ. We can then use this, as indicated above, to give a specific orientation of the cover graph.

Remark 5.1.6. We will now frequently drop the notation $\vec{e}$ for an oriented edge. For the rest of the thesis we will be working with oriented graphs and thus the context will be clear. This will also be convenient later, when we introduce cover graphs, so that we can adopt the notation $\tilde{e}$ to signify that an edge belongs to the cover graph $\widetilde{\Gamma}$ as opposed to edges e in the base graph Γ.

5.2 Admissible double covers of graphs

Definition 5.2.1. An **admissible (oriented) covering graph of degree** 2 **of** Γ (or just a covering graph of Γ, for short) is a triple $((\widetilde{\Gamma}, \iota), \Gamma)$ where,

- $\widetilde{\Gamma}$ is a finite (oriented) graph.
- ι is and admissible involution of (oriented) $\widetilde{\Gamma}$.
- Γ is a (oriented) graph such that $V(\Gamma) = V(\widetilde{\Gamma})/\iota_V$, $\vec{E}(\Gamma) = \vec{E}(\widetilde{\Gamma})/\iota_E$,
- The natural quotient map on vertices and edges induces a (oriented) graph morphism $\pi : \widetilde{\Gamma} \to \Gamma$.

We call $\widetilde{\Gamma}$ the cover graph and Γ the base graph.

Remark 5.2.2. An admissible cover of curves $\pi : \widetilde{C} \to C$ induces an admissible cover of dual graphs. Conversely, given an admissible cover of dual graphs, there exists an admissible cover of curves whose dual graphs are the original graphs. [CMGHL17b]

We introduce some notation that will be convenient later. Given an admissible cover graph, $v \in V(\Gamma)$ and $e \in E(\Gamma)$, we use the notation

$$\pi^{-1}(v) = \begin{cases} \{\tilde{v}^+, \tilde{v}^-\}, \\ \{\tilde{v}\} \end{cases} \qquad \pi^{-1}(e) = \begin{cases} \{\tilde{e}^+, \tilde{e}^-\}, \\ \{\tilde{e}\} \end{cases}$$

to indicate the various possibly vertices and edges in the cover graph lying over the vertices and edges in the base graph. The $\pm$ notation indicates that vertices or edges are interchanged under the involution; i.e., $\tilde{v}^+ = \iota(\tilde{v}^-)$.

5.3 Example of admissible involutions and covering graphs

First we will consider a very basic example of a graph.

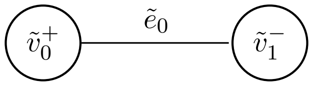

Figure 5.1: Non admissible involution

In this graph we have two vertices and one edge. The involution suggested by the notation, i.e., $\iota(\tilde{v}_0^+) = \tilde{v}_0^-$, and $\iota(\tilde{e}_0) = \tilde{e}_0$, fails to be admissible since the edge is fixed, but the endpoints are interchanged. Notice that there is no choice of orientation of the edge so that the involution becomes an orientation preserving graph morphism.

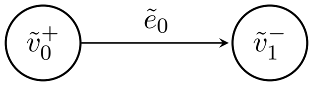

Figure 5.2: Non admissible involution (oriented)

If we consider the following base graph Γ,

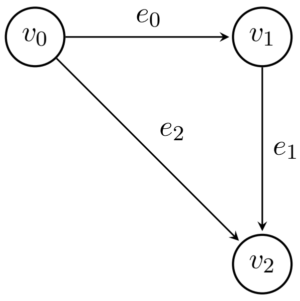

Figure 5.3: Base Graph Γ

Then the following is a dual graph of an admissible cover.

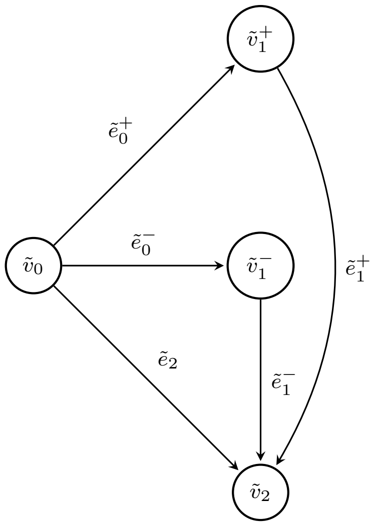

Figure 5.4: Admissible cover of Γ

The next dual graph is not admissible because ι is not an orientation preserving graph morphism of $\widetilde{\Gamma}$, notice

$$\tilde{v}_1^- = s(\iota_E(\tilde{e}_1)) \neq \iota_V(s(\tilde{e}_1)) = \tilde{v}_1^+.$$

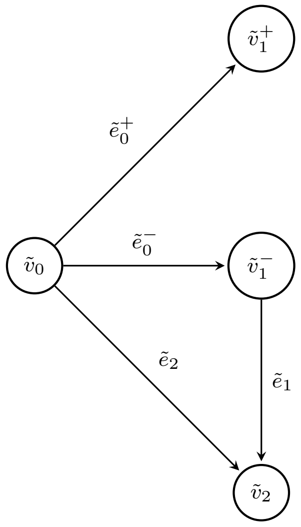

Figure 5.5: Not an admissible cover of Γ

5.4 Admissible covers of loops

In this section we will go through specific ways to admissibly cover a loop in the base graph. First, if the vertex is not fixed by the involution ι_V in the covering graph then the loop must also not be fixed by the involution ι_E in the covering graph. In this scenario there are two ways to cover a loop. Suppose we are given the following base graph.

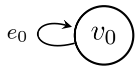

Figure 5.6: Base Graph Loop

The first way to cover the loop is to have a corresponding loop on the covered vertex as follows,

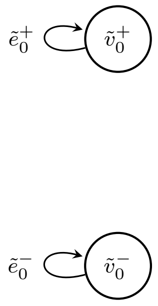

Figure 5.7: First Loop Cover Option

(Notice that this graph is not connected and would require more nodes and edges to be an admissible cover.) The second way to cover a loop is to have the loop become an edge that connects the two elements in the preimage of v_1. Then have the second edge be the same with the reverse orientation.

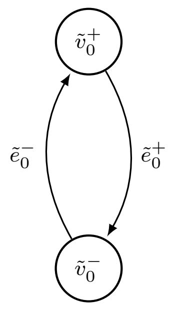

Figure 5.8: Second Loop Cover Option

Now suppose that the vertex of the graph is fixed by ι_V in the covering map. If the loop is not covered then we have the following situation.

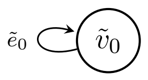

Figure 5.9: Covering a ramified loop for ramified vertex

If the loop e_0 is covered then we will have two loop on the admissible cover, $\tilde{e}_0^-$ and $\tilde{e}_0^+$, both starting and ending at the vertex $\tilde{v}_0$. The connection with moduli is reviewed in [CMGHL17b, §3.4]

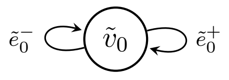

Figure 5.10: Covering a loop for ramified vertex

Remark 5.4.1. In the next section we will show how to reduce all admissible double covering dual graphs which contain loops to calculations on graphs of smaller dimensions. This indicates that we can assume that in the admissible covers the loop edges are covered and their corresponding vertices are covered. Also, in this case we only have to consider loop covering option number 2.

5.5 Reduction of loops in the cover graph

It has been mentioned before that there are too many possible covering graphs to check all examples therefore it would be helpful to make some reductions on the space of all cover graphs. The only reduction that we will be able to make is ability to reduce cover graphs containing a loop to lower dimensional computations following the methods from [CMGHL17b, Appen. D].

Propostion 5.5.1. Let $\widetilde{\Gamma}$ be an admissible cover of a base graph Γ and suppose that $\widetilde{\Gamma}$ contains a loop. We can reduce $\widetilde{\Gamma}$ to lower dimensional computations.

Proof. Let $\widetilde{\Gamma}$ be an admissible cover of a base graph Γ that contains a loop. If $e_0 \in E(\Gamma)$ is the loop in the base graph connected at the vertex v_0 – we can re-index to make this true – there are two cases to consider. First suppose that e_0 is fixed by the involution ι_E. Then $\tilde{e}_0 \in E(\widetilde{\Gamma})$ is the edge above e_0 in the cover graph. We can choose a basis for $H_1(\widetilde{\Gamma}, \mathbb{Z})$ such that it contains the basis vector $(1, 0, \ldots, 0)$ corresponding to the loop $\tilde{e}_0$ and so that $\tilde{e}_0$ is not a part of any other generating cycle in the basis. If we apply the map $\frac{1}{2}(\mathrm{id} - \iota)$ to $H_1(\widetilde{\Gamma}, \mathbb{Z})$ we still have the vector $(1, 0, \cdots, 0)$ in the basis of $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$. Next suppose that e_0 is not fixed by the involution ι_E then we can choose a basis for $H_1(\widetilde{\Gamma}, \mathbb{Z})$ such that it contains the two vectors $(1, 0, \ldots, 0)$ and $(0, 1, 0, \ldots, 0)$ corresponding to $\tilde{e}_0^-$ and $\tilde{e}_0^+$ respectively and no other generating cycle of $H_1(\widetilde{\Gamma}, \mathbb{Z})$ will contain the two edges $\tilde{e}_0^-$ and $\tilde{e}_0^+$. Now when we apply the map we will get the two vectors $(1, -1, 0, \ldots, 0)$ and $(-1, 1, 0, \ldots, 0)$ which are linearly dependent and thus will correspond to one basis vector in $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$.

Let $\Gamma_1 \subset \Gamma$ be the subgraph of containing the vertex v_0 and the loop e_0. Define $\widetilde{\Gamma}_1 \subset \widetilde{\Gamma}$ to be the cover graph of Γ_1 which contains vertices $\pi^{-1}(v_0)$ and the edges $\pi^{-1}(e_0)$ where $\pi : \widetilde{\Gamma} \to \Gamma$. Let $\Gamma_2 \subset \Gamma$ be the subgraph equal to Γ minus the loops $\pi^{-1}(e_0)$. That is $V(\Gamma_2) = V(\Gamma)$ and $E(\Gamma_2) = E(\Gamma) - \{e_0\}$. Define $\widetilde{\Gamma}_2 \subset \widetilde{\Gamma}$ to be the covering subgraph associated to Γ_2. Notice that $\iota\widetilde{\Gamma}_i = \widetilde{\Gamma}_i$ for $i = 1, 2$. We can decompose the monodromy cone as follows,

$$\mathrm{MC}(\widetilde{\Gamma}) = \left(\begin{array}{c|c} \mathrm{MC}(\widetilde{\Gamma}_1) & 0 \\ \hline 0 & \mathrm{MC}(\widetilde{\Gamma}_2) \end{array}\right)$$

Therefore we can reduce the question of whether $\mathrm{MC}(\widetilde{\Gamma})$ is in an admissible cone decomposition to a question of whether the lower dimensional monodromy cones $\mathrm{MC}(\widetilde{\Gamma}_1)$ and $\mathrm{MC}(\widetilde{\Gamma}_2)$ are contained in cones. □

5.6 Friedman-Smith Covers

We now discuss a class of examples of admissible double covers that Friedman and Smith used to show that the Prym map does not extend in [FS86].

Definition 5.6.1. A **Friedman-Smith** cover with $2n \geq 2$ nodes is an admissible cover $\pi : \tilde{C} \to C$ such that

(i) $\tilde{C} = \tilde{C}_1 \cup \tilde{C}_2$ with $\tilde{C}_1$ and $\tilde{C}_2$ smooth and irreducible, and

$$\tilde{C}_1 \cap C_2 = \left\{\tilde{p}_1^-, \tilde{p}_1^+, \ldots, \tilde{p}_n^-, \tilde{p}_n^+\right\}.$$

(ii) $\iota\tilde{C}_i = \iota C_i$ for $i = 1, 2$.

(iii) $\iota\tilde{p}_i^{\pm} = \tilde{p}_i^{\mp}$ for $i = 1, \ldots, n$.

A cover $\pi : \tilde{C} \to C$ is a **degeneration of a Friedman-Smith cover** if it can be obtained by further degenerations of a Friedman-Smith cover. In the language of dual graphs we have the following definition.

Definition 5.6.2. A graph $\widetilde{\Gamma}$ is a Friedman-Smith graph with $2n \geq 2$ edges with admissible involution ι if:

(i) $\widetilde{\Gamma}$ has two vertices $\{\tilde{v}_1, \tilde{v}_2\}$

(ii) $\widetilde{\Gamma}$ has $2n$ edges $\left\{\tilde{e}_1^-, \tilde{e}_1^+, \ldots, \tilde{e}_n^-, \tilde{e}_n^+\right\}$ with $s(\tilde{e}_i^{\pm}) = \tilde{v}_1$ and $t(\tilde{e}_i^{\pm}) = \tilde{v}_2$ for $i = 1, \ldots, n$.

(iii) $\iota\tilde{v}_i = \tilde{v}_i$ for $i = 1, 2$

(iv) $\iota\tilde{e}_i^{\pm} = \tilde{e}_i^{\mp}$ for $i = 1, \ldots, n$.

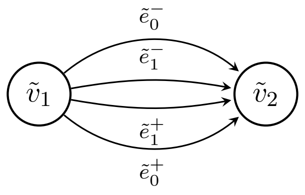

Figure 5.11: Friedman-Smith graph of order 2

A **degeneration of a Friedman-Smith graph** is a cover graph $\widetilde{\Gamma}$ with admissible involution ι with $2n \geq 2$ edges such that: $\widetilde{\Gamma}$ admits disjoint, connected subgraphs $\widetilde{\Gamma}_1$ and $\widetilde{\Gamma}_2$, with $\widetilde{\Gamma}_1$ and $\widetilde{\Gamma}_2$ connected by exactly $2n$ edges $\{\tilde{e}_1^-, \tilde{e}_1^+, \ldots, \tilde{e}_n^-, \tilde{e}_n^+\}$ and $\iota(\widetilde{\Gamma}_i) = \widetilde{\Gamma}_i$ for $i = 1, 2$ and $\iota(\tilde{e}_i^{\pm}) = \tilde{e}_i^{\mp}$ for $i = 1, \ldots, n$.

The closure of Friedman smith curves of order n is denoted $\overline{FS}_n$. This will be all the degenerations of Friedman-Smith graphs of order n. The set $\overline{FS}_n$ is codimension n in $\overline{R}_{g+1}$.

5.7 The cone of quadratic forms associated to an admissible double cover of graphs

As described in [CMGHL17b], associated to an admissible double cover of graphs $\pi : \widetilde{\Gamma} \to \Gamma$ is a cone of quadratic forms. We now recall this construction.

Let ι be the admissible involution of $\widetilde{\Gamma}$ associated to the admissible double cover, and fix an orientation $\tilde{\phi}$ of $\widetilde{\Gamma}$ so that ι is an involution of the oriented graph $(\widetilde{\Gamma}, \tilde{\phi})$. The involution ι determines an involution of $H_1(\widetilde{\Gamma}, \mathbb{Z})$ and of $H^1(\widetilde{\Gamma}, \mathbb{Z})$. We denote using $\pm$ superscripts the $\pm$ eigenspaces of the action of ι, and we set $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} := H_1(\widetilde{\Gamma}, \mathbb{Z})/H^1(\widetilde{\Gamma}, \mathbb{Z})^+$. It is often convenient to identify $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$ as the image of the map

$$\frac{1}{2}(\mathrm{Id} - \iota) : H_1(\widetilde{\Gamma}, \mathbb{Z}) \to H_1(\widetilde{\Gamma}, \frac{1}{2}\mathbb{Z}). \tag{5.7.1}$$

We have also that $H^1(\widetilde{\Gamma},\mathbb{Z})^- = \left(H_1(\widetilde{\Gamma},\mathbb{Z})^{[-]}\right)^\vee$. Finally, for each edge e of Γ (oriented with the given orientation of Γ), we fix a cocycle $\ell_e \in H^1(\widetilde{\Gamma},\mathbb{Z})^-$ by the rule

$$\ell_e := \begin{cases} \tilde{e}^\vee - \iota\tilde{e}^\vee & \text{if } \iota\tilde{e}^\vee \neq -\tilde{e}^\vee \in H^1(\widetilde{\Gamma},\mathbb{Z}), \\ \tilde{e}^\vee & \text{if } \iota\tilde{e}^\vee = -\tilde{e}^\vee \in H^1(\widetilde{\Gamma},\mathbb{Z}), \end{cases}$$

where we are taking $\tilde{e}$ to be an edge of $\widetilde{\Gamma}$ lying above e, with the canonical orientation.

Remark 5.7.1. We have the following possibly more elementary ways to parse the definition of ℓ_e. First, ℓ_e is the primitive element of $H^1(\widetilde{\Gamma},\mathbb{Z})$ in the real ray generated by $\tilde{e}^\vee - \iota\tilde{e}^\vee$ in $H^1(\widetilde{\Gamma},\mathbb{R})$. Alternatively, since $H^1(\widetilde{\Gamma},\mathbb{Z}) = H_1(\widetilde{\Gamma},\mathbb{Z})$, we can evaluate $\tilde{e}^\vee$ and $\iota\tilde{e}^\vee$ on cycles. We define $\ell_e = \tilde{e}^\vee - \iota\tilde{e}^\vee$, unless on every basic cycle γ of $\widetilde{\Gamma}$ (every edge appears with multiplicity $\pm1, 0$) we have $\ell_e(\gamma) = 0, 2$.

The cone of quadratic forms associated to $\pi : \widetilde{\Gamma} \to \Gamma$ is the cone:

$$\overline{\sigma}(\widetilde{\Gamma}/\Gamma) := \mathbb{R}_{\geq 0}\langle \ell_e^2\rangle_{e\in E(\Gamma)} \subseteq \left(\operatorname{Sym}^2 H_1(\widetilde{\Gamma},\mathbb{Z})^{[-]}\right)^\vee_{\mathbb{R}}.$$

See [CMGHL17b, §5.2] for more details.

5.7.1 Computing the cone of quadratic forms

In this subsection we describe a method of finding $\overline{\sigma}(\widetilde{\Gamma},\Gamma)$ computationally in examples. Specifically, one computes a basis $z_1,\dots,z_n$ of $H_1(\widetilde{\Gamma},\mathbb{Z})^{[-]}$. In light of (5.7.1), in practice, one can compute a basis of $H_1(\widetilde{\Gamma},\mathbb{Z})$, and then compute a basis for the image of $\frac{1}{2}(\mathrm{Id} - \iota)$ from this.

We then obtain the basis $z_1^\vee,\dots,z_n^\vee$ of $H_1(\widetilde{\Gamma},\mathbb{Z})^{[-]} = H^1(\widetilde{\Gamma},\mathbb{Z})^-$. If we enumerate the edges $e_1,\dots,e_m$ of the base graph Γ, then we can express the ℓ_{e_j} in terms of the basis $z_1^\vee,\dots,z_n^\vee$ as

$$\ell_{e_j} = \sum_i \ell_{e_j}(z_i) z_i^\vee.$$

The matrix $(\ell_{e_j}(z_i))_{i,j}$ then has columns that in the chosen bases represent linear forms whose squares are the extreme rays of the cone $\overline{\sigma}(\widetilde{\Gamma}/\Gamma)$.

5.7.2 Computing the monodromy cone in an example: Friedman–Smith covers

The following example is taken verbatim from [CMGHL17b, §6.2]. Let $\pi : \widetilde{C} \to C$ be a Friedman–Smith cover with $2n \geq 2$ nodes. The dual graph $\widetilde{\Gamma}$ of $\widetilde{C}$ has vertices $V(\widetilde{\Gamma}) = \{\tilde{v}_1, \tilde{v}_2\}$ and edges $E(\widetilde{\Gamma}) = \{\tilde{e}_1^+, \tilde{e}_1^-, \ldots, \tilde{e}_n^+, \tilde{e}_n^-\}$. The involution ι acts by $\iota(\tilde{v}_i) = \tilde{v}_i$ $(i = 1, 2)$ and $\iota(\tilde{e}_i^+) = \tilde{e}_i^-$ $(i = 1, \ldots, n)$. For simplicity, we will fix a compatible orientation on $\widetilde{\Gamma}$, as in Figure 11.1; i.e. for all i set $t(\tilde{e}_i^\pm) = \tilde{v}_2$ and $s(\tilde{e}_i^\pm) = \tilde{v}_1$.

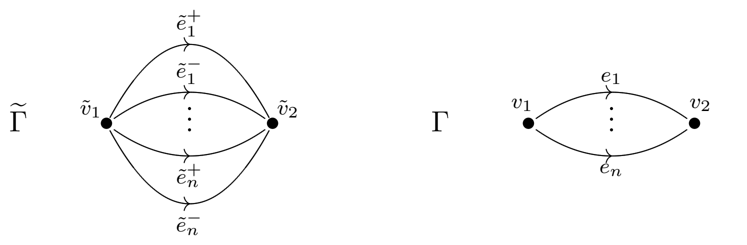

Figure 5.12: Dual graph of a Friedman–Smith example with $2n \geq 2$ nodes (FS_n).

One has

$$H_1(\widetilde{\Gamma}, \mathbb{Z}) = \mathbb{Z}\langle \tilde{e}_1^+ - \tilde{e}_1^-, \ldots, \tilde{e}_n^+ - \tilde{e}_n^-, \tilde{e}_1^+ - \tilde{e}_2^-, \ldots, \tilde{e}_{n-1}^+ - \tilde{e}_n^- \rangle. \tag{5.7.2}$$

Indeed, we have $b_1(\widetilde{\Gamma}) = \#E(\widetilde{\Gamma}) - \#V(\widetilde{\Gamma}) + b_0(\widetilde{\Gamma}) = 2n - 1$, since $\widetilde{\Gamma}$ is connected. The $2n-1$ elements listed above are in fact a generating set for $H_1(\widetilde{\Gamma}, \mathbb{Z})$, as can be easily detected from the associated matrix. For instance, if one takes the elements in the order $\tilde{e}_1^+ - \tilde{e}_1^-, \tilde{e}_1^+ - \tilde{e}_2^-, \ldots, \tilde{e}_n^+ - \tilde{e}_n^-, \tilde{e}_{n-1}^+ - \tilde{e}_n^-$ and constructs a matrix with rows expressing these elements with respect to the basis $\tilde{e}_1^-, \tilde{e}_1^+, \ldots, \tilde{e}_n^-, \tilde{e}_n^+$, one obtains a $(2n-1) \times (2n)$ matrix whose first $(2n-1) \times (2n-1)$ sub-matrix is upper triangular with all the diagonal entries equal to ± 1.

Recall that $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} = H_1(\widetilde{\Gamma}, \mathbb{Z})/H_1(\widetilde{\Gamma}, \mathbb{Z})^+$ and is isomorphic to the image of the map

$$\frac{1}{2}(\mathrm{Id} - \iota) : H_1(\widetilde{\Gamma}, \mathbb{Z}) \to H_1(\widetilde{\Gamma}, \mathbb{R}).$$

From (5.7.2), one has

$$H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} \cong \mathbb{Z}\langle \tilde{e}_1^+ - \tilde{e}_1^-, \frac{1}{2}(\tilde{e}_1^+ - \tilde{e}_1^-) + \frac{1}{2}(\tilde{e}_2^+ - \tilde{e}_2^-), \ldots, \frac{1}{2}(\tilde{e}_{n-1}^+ - \tilde{e}_{n-1}^-) + \frac{1}{2}(\tilde{e}_n^+ - \tilde{e}_n^-) \rangle.$$

For brevity, set

$$z_1 = \tilde{e}_1^+ - \tilde{e}_1^-,\ z_2 = \frac{1}{2}(\tilde{e}_1^+ - \tilde{e}_1^-) + \frac{1}{2}(\tilde{e}_2^+ - \tilde{e}_2^-), \ldots,\ z_n = \frac{1}{2}(\tilde{e}_{n-1}^+ - \tilde{e}_{n-1}^-) + \frac{1}{2}(\tilde{e}_n^+ - \tilde{e}_n^-)$$

so that $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} \cong \mathbb{Z}\langle z_1, \ldots, z_n\rangle$. Then $H^1(\widetilde{\Gamma}, \mathbb{Z})^- = \Big(H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}\Big)^\vee \cong \mathbb{Z}\langle z_1^\vee, \ldots, z_n^\vee\rangle$.

Now observe that

$$H^1(\widetilde{\Gamma}, \mathbb{Z}) = \mathbb{Z}\langle (\tilde{e}_1^+)^\vee, (\tilde{e}_1^-)^\vee, \ldots, (\tilde{e}_n^+)^\vee, (\tilde{e}_n^-)^\vee\rangle / \langle (\tilde{e}_1^+)^\vee + (\tilde{e}_1^-)^\vee + \ldots + (\tilde{e}_n^+)^\vee + (\tilde{e}_n^-)^\vee\rangle.$$

It follows that for $i = 1, \ldots, n$,

$$\begin{aligned} \iota(\tilde{e}_i^+)^\vee &= (\tilde{e}_i^-)^\vee = -(\tilde{e}_i^+)^\vee \quad \text{if } \ n = 1, \\ \iota(\tilde{e}_i^+)^\vee &= (\tilde{e}_i^-)^\vee \neq -(\tilde{e}_i^+)^\vee \quad \text{if } \ n \geq 2. \end{aligned}$$

Consequently, we may choose for $i = 1, \ldots, n$,

$$\ell_{e_i} := \begin{cases} (\tilde{e}_i^+)^\vee & \text{if } \ n = 1, \\ (\tilde{e}_i^+)^\vee - (\tilde{e}_i^-)^\vee & \text{if } \ n \geq 2. \end{cases}$$

For $n = 1$, ℓ_{e_1} is clearly a basis for $H^1(\widetilde{\Gamma}, \mathbb{Z})^-$, and so we note that condition (V) of Theorem 5.6 in [CMGHL17b] holds in this case. Now consider the case $n \geq 2$. Evaluating the ℓ_{e_i} on the basis $z_1, \ldots, z_n$, we obtain that

$$\begin{array}{ccccccccccc} \ell_{e_1} & = & 2z_1^\vee & + & z_2^\vee & & & & & & \\ \ell_{e_2} & = & & & z_2^\vee & + & z_3^\vee & & & & \\ \vdots & \vdots & & & & \ddots & & & & & \\ \vdots & \vdots & & & & & \ddots & & & & \\ \ell_{e_{n-1}} & = & & & & & & & z_{n-1}^\vee & + & z_n^\vee \\ \ell_{e_n} & = & & & & & & & & & z_n^\vee \end{array}$$

Thus, with respect to these bases, transposing the coefficients above, we have that $\overline{\sigma}(\widetilde{\Gamma}/\Gamma)$ is given by the matrix:

$$\begin{pmatrix} 2 & & & & & \\ 1 & 1 & & & & \\ & 1 & 1 & & & \\ & & \ddots & \ddots & & \\ & & & 1 & 1 & \\ & & & & 1 & 1 \end{pmatrix}. \tag{5.7.3}$$

5.8 Computing the cone of quadratic forms as matrix algebra

We now consider a matrix algorithm for computing the cone of quadratic forms. We explain the algorithm with an example.

5.8.1 Step 1: The graph and the associated incidence matrix

Suppose we are given $(\widetilde{\Gamma}, \tilde{\phi}, \iota)$. For instance, we will work with the example below:

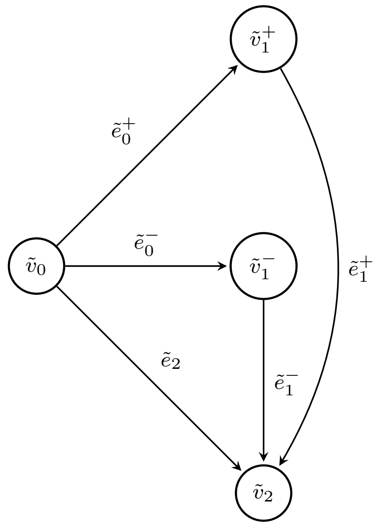

Figure 5.13: The graph $(\widetilde{\Gamma}, \tilde{\phi})$ with the involution ι indicated with the superscripts.

From the oriented graph we immediately obtain the incidence matrix:

$$A = \begin{array}{c} \\ \tilde{v}_0^- \\ \tilde{v}_0^+ \\ \tilde{v}_1^- \\ \tilde{v}_1^+ \\ \tilde{v}_2^- \\ \tilde{v}_2^+ \end{array} \begin{array}{c} \begin{array}{cccccc} \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ \end{array} \\ \begin{bmatrix} -1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

5.8.2 Step 2: A basis for the homology of the graph

Given the reduced incidence matrix, we explained before a matrix algorithm for computing $H_1(\widetilde{\Gamma}, \mathbb{Z}) \subseteq C_1(\widetilde{\Gamma}, \mathbb{Z})$. Namely, we first compute the reduced row echelon form of A:

$$\begin{array}{cccccc}
\tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ \\
\end{array}$$
$$\begin{bmatrix}
1 & 0 & 0 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & -1 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}$$

We then find a basis for the nullspace of the row reduced matrix ignoring the rows and columns corresponding to uncovered vertices and edges. In this case the rows corresponding to $\tilde{v}_0^+$ and $\tilde{v}_2^+$ would be ignored and the column corresponding to $\tilde{e}_2^+$ would be ignored.

$$B = \left[\begin{array}{cccccc}
\tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ \\
1 & -1 & 1 & -1 & 0 & 0 \\
1 & 0 & 1 & 0 & -1 & 0
\end{array}\right]$$

The rows give a basis for $H_1(\widetilde{\Gamma}, \mathbb{Z})$ inside of $C_1(\widetilde{\Gamma}, \mathbb{Z})$.

5.8.3 Step 3: Compute a generating set for $H_1(\widetilde{\Gamma},\mathbb{Z})^{[-]}$

We will use the identification $H_1(\widetilde{\Gamma},\mathbb{Z})^{[-]} = \frac{1}{2}(\mathrm{Id}-\iota)H_1(\widetilde{\Gamma},\mathbb{Z}) \subseteq C_1(\widetilde{\Gamma},\frac{1}{2}\mathbb{Z})$. For this we write down

$$\frac{1}{2}(\mathrm{Id}-\iota) : C_1(\widetilde{\Gamma},\mathbb{Z}) \to C_1(\widetilde{\Gamma},\frac{1}{2}\mathbb{Z})$$

in matrix form:

$$\frac{1}{2}(\mathrm{Id}-\iota) = \begin{array}{c} \begin{array}{cccccc} \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ \end{array} \\ \begin{bmatrix} \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & -\frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \end{array}$$

Recall, that $H_1(\Gamma,\mathbb{Z})^{[-]}$ can be naturally identified as the image of the map $H_1(\widetilde{\Gamma},\mathbb{Z}) \to H_1(\widetilde{\Gamma},\frac{1}{2}\mathbb{Z})$ given by $z \mapsto \frac{1}{2}(z-\iota z)$ (see equation 5.7.1). To obtain a generating set for $H_1(\widetilde{\Gamma},\mathbb{Z})^{[-]}$ inside of

$C_1(\widetilde{\Gamma}, \frac{1}{2}\mathbb{Z})$, we simply perform matrix multiplication $B \cdot \frac{1}{2}(\mathrm{Id} - \iota)$

$$\begin{array}{cccccc} \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ \\ \end{array} \begin{bmatrix} 1 & -1 & 1 & -1 & 0 & 0 \\ 1 & 0 & 1 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{2} & -\frac{1}{2} & 0 & 0 & 0 & 0 \\ -\frac{1}{2} & \frac{1}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \\ 0 & 0 & -\frac{1}{2} & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} = \begin{array}{cccccc} \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ \\ \end{array} \begin{bmatrix} 1 & -1 & 1 & -1 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \end{bmatrix}$$

The resulting matrix gives us a matrix C with rows that are a generating set for $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$ inside of $C_1(\widetilde{\Gamma}, \frac{1}{2}\mathbb{Z})$:

$$C = \begin{array}{cccccc} \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ \\ 1 & -1 & 1 & -1 & 0 & 0 \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \end{array}$$

5.8.4 Step 4: Compute a basis for $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$

Next we compute a basis for $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$ inside of $C_1(\widetilde{\Gamma}, \frac{1}{2}\mathbb{Z})$. For this we simply perform integral row reduction on the matrix C (i.e., with row operations that are given by integral matrices, with determinant ± 1). In our example, we obtain the matrix

$$D = \begin{array}{cccccc} \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ \\ \frac{1}{2} & -\frac{1}{2} & \frac{1}{2} & -\frac{1}{2} & 0 & 0 \end{array}$$

whose rows give a basis for $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$ inside of $C_1(\widetilde{\Gamma}, \frac{1}{2}\mathbb{Z})$.

5.8.5 Step 5: Compute the cone of quadratic forms

For this, we simply need to evaluate the linear forms ℓ_e on each of the basis elements of $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$. In fact, it is easy to see that we can simply evaluate $\tilde{e}^\vee - \iota\tilde{e}^\vee$ on the basis elements instead, and divide by powers of 2 if need be at the end. This makes the matrix form easier to describe. In other words, in our example we compute:

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} D^T = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{1}{2} \\ -\frac{1}{2} \\ \frac{1}{2} \\ -\frac{1}{2} \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}$$

Finally, we divide each nonzero row in the output matrix by powers of 2 until no entry is divisible by 2. The transpose Q of this output matrix has columns that define linear forms whose squares are the extreme rays of the cone of quadratic forms

$$Q = \begin{array}{ccc} \ell_{e_0} & \ell_{e_1} & \ell_{e_2} \\ \left[\,1\right. & 1 & \left.0\,\right] \end{array}.$$

Visually Q is easy to compute from D: one simply takes the difference of the subsequent entries in the rows, and then divides the resulting columns by 2 until they are primitive.

5.8.6 A remark about half integers

For computational purposes it can be useful to avoid half integers. This can be easily accomplished in the following way. In Step 3, we can use the matrix $(\mathrm{Id} - \iota)$, instead of $\frac{1}{2}(\mathrm{Id} - \iota)$. This will have the result of multiplying the matrix C by 2. Then in Step 4, since integral row operations commute with multiplication by 2, the integral row operations in Step 4 will end up giving $2D$. Then in Step 5, the output of the first matrix multiplication will differ by a factor of 2; but since in the end we are dividing each row by factors of 2, the final result is the same.

5.9 The Friedman–Smith computation revisited

We now do the Friedman–Smith computation using the matrix algorithm we just described. We have the incidence matrix:

$$A = \begin{array}{c} \\ \tilde{v}_1 \\ \tilde{v}_2 \end{array} \begin{array}{c} \begin{array}{ccccccc} \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \cdots & \tilde{e}_n^- & \tilde{e}_n^+ \end{array} \\ \begin{bmatrix} -1 & -1 & -1 & -1 & \cdots & -1 & -1 \\ 1 & 1 & 1 & 1 & \cdots & 1 & 1 \end{bmatrix} \end{array}$$

Next we compute the reduced row echelon form of A:

$$\begin{array}{c} \begin{array}{ccccccc} \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \cdots & \tilde{e}_n^- & \tilde{e}_n^+ \end{array} \\ \begin{bmatrix} 1 & 1 & 1 & 1 & \cdots & 1 & 1 \\ 0 & 0 & 0 & 0 & \cdots & 0 & 0 \end{bmatrix} \end{array}$$

We then augment with appropriate rows to obtain

$$\begin{array}{c}
\begin{array}{ccccccc} \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \cdots & \tilde{e}_n^- & \tilde{e}_n^+ \end{array} \\
\begin{bmatrix}
1 & 1 & 1 & 1 & \cdots & 1 & 1 \\
0 & -1 & 0 & 0 & \cdots & 0 & 0 \\
0 & & -1 & 0 & \cdots & 0 & 0 \\
\vdots & & & \ddots & & & \vdots \\
\vdots & & & & \ddots & & \vdots \\
0 & 0 & 0 & 0 & \cdots & -1 & 0 \\
0 & 0 & 0 & 0 & \cdots & 0 & -1
\end{bmatrix}
\end{array}$$

We then drop the first column (the column that is a basis vector), and transpose, to obtain the matrix

$$B = \begin{array}{c}
\begin{array}{ccccccc} \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \cdots & \tilde{e}_n^- & \tilde{e}_n^+ \end{array} \\
\begin{bmatrix}
1 & -1 & 0 & 0 & \cdots & 0 & 0 \\
1 & 0 & -1 & 0 & \cdots & 0 & 0 \\
1 & 0 & 0 & -1 & \cdots & 0 & 0 \\
\vdots & & & \ddots & & & \vdots \\
\vdots & & & & \ddots & & \vdots \\
1 & 0 & 0 & 0 & \cdots & -1 & 0 \\
1 & 0 & 0 & 0 & \cdots & 0 & -1
\end{bmatrix}
\end{array}$$

The rows give a basis for $H_1(\widetilde{\Gamma}, \mathbb{Z})$ inside of $C_1(\widetilde{\Gamma}, \mathbb{Z})$. To obtain a generating set for $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$ inside of $C_1(\widetilde{\Gamma}, \frac{1}{2}\mathbb{Z})$, we first perform matrix multiplication $(\mathrm{Id} - \iota)B^T$

$$\begin{bmatrix}
1 & -1 & 0 & 0 & \cdots & 0 & 0 \\
-1 & 1 & 0 & 0 & \cdots & 0 & 0 \\
0 & 0 & 1 & -1 & \cdots & 0 & 0 \\
0 & 0 & -1 & 1 & \cdots & 0 & 0 \\
\vdots & & & & \ddots & & \vdots \\
0 & 0 & 0 & 0 & \cdots & 1 & -1 \\
0 & 0 & 0 & 0 & \cdots & -1 & 1
\end{bmatrix}
\begin{bmatrix}
1 & 1 & 1 & \cdots & 1 & 1 \\
-1 & 0 & 0 & \cdots & 0 & 0 \\
0 & -1 & 0 & \cdots & 0 & 0 \\
\vdots & & \ddots & & & \vdots \\
\vdots & & & \ddots & & \vdots \\
0 & 0 & 0 & \cdots & -1 & 0 \\
0 & 0 & 0 & \cdots & 0 & -1
\end{bmatrix}
=
\begin{bmatrix}
2 & 1 & 1 & 1 & \cdots & 1 & 1 \\
-2 & -1 & -1 & -1 & \cdots & -1 & -1 \\
0 & 0 & 1 & -1 & \cdots & 0 & 0 \\
\vdots & & & & \ddots & & \vdots \\
0 & 0 & 0 & 0 & \cdots & -1 & 1 \\
0 & 0 & 0 & 0 & \cdots & 1 & -1
\end{bmatrix}$$

The transpose of the resulting matrix gives us a matrix C with rows that are a generating set for $2H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$ inside of $C_1(\widetilde{\Gamma}, \mathbb{Z})$:

$$C = \begin{array}{c} \begin{array}{ccccccc} \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \cdots & \tilde{e}_n^- & \tilde{e}_n^+ \end{array} \\ \begin{bmatrix} 2 & -2 & 0 & 0 & \cdots & 0 & 0 \\ 1 & -1 & -1 & 1 & \cdots & 0 & 0 \\ 1 & -1 & 1 & -1 & \cdots & 0 & 0 \\ \vdots & & & & \ddots & & \vdots \\ 1 & -1 & 0 & 0 & \cdots & -1 & 1 \\ 1 & -1 & 0 & 0 & \cdots & 1 & -1 \end{bmatrix} \end{array}$$

Next we compute a basis for $2H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$ inside of $C_1(\widetilde{\Gamma}, \mathbb{Z})$. For this we simply perform integral row reduction on the matrix C (i.e., with row operations that are given by integral matrices, with determinant ± 1). This is also known as the Hermite normal form of C

$$D = \begin{array}{c} \begin{array}{ccccccccc} \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \tilde{e}_3^- & \tilde{e}_3^+ & \cdots & \tilde{e}_n^- & \tilde{e}_n^+ \end{array} \\ \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & \cdots & 1 & -1 \\ 0 & 0 & 1 & -1 & 0 & 0 & \cdots & -1 & 1 \\ \vdots & & & & & & \ddots & & \vdots \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & -2 & 2 \end{bmatrix} \end{array}$$

whose rows give a basis for $2H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$ inside of $C_1(\widetilde{\Gamma}, \mathbb{Z})$. To compute the cone of quadratic forms, we consider

$$\begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & \cdots & 0 & 0 \\ & & & & & & \ddots & & \\ 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 1 & -1 \end{bmatrix} D^T = \begin{bmatrix} 2 & 0 & 0 & \cdots & 0 \\ 0 & 2 & 0 & \cdots & 0 \\ 0 & 0 & 2 & \cdots & 0 \\ & & & \ddots & \\ 2 & -2 & -2 & \cdots & -4 \end{bmatrix}$$

Finally, we divide each nonzero row in the output matrix by powers of 2 until no entry is divisible by 2. The transpose Q of this output matrix has columns that define linear forms whose squares are the extreme rays of the cone of quadratic forms

$$Q = \begin{array}{c} \begin{array}{ccccc} \ell_{e_0} & \ell_{e_1} & \ell_{e_2} & \cdots & \ell_{e_n} \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & \cdots & 1 \\ 0 & 1 & 0 & \cdots & -1 \\ 0 & 0 & 1 & \cdots & -1 \\ & & & \ddots & \\ 0 & 0 & 0 & \cdots & -2 \end{bmatrix} \end{array}.$$

Visually Q is easy to compute from D: one simply takes the difference of the subsequent entries in the rows, and then divides the resulting columns by 2 until they are primitive.

5.9.1 A few observations on row operations

Note that using row operations, we can put this matrix in the form

$$\begin{array}{c} \begin{array}{ccccc} \ell_{e_0} & \ell_{e_1} & \ell_{e_2} & \cdots & \ell_{e_n} \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & \cdots & -1 \\ 0 & 1 & 0 & \cdots & -1 \\ 0 & 0 & 1 & \cdots & -1 \\ & & & \ddots & \\ 0 & 0 & \cdots & 1 & -1 \\ 0 & 0 & 0 & \cdots & 2 \end{bmatrix} \end{array},$$

which is the form used in [CMGHL17b, App. A], and is easily derived from the first matrix we computed via row operations, and taking negatives of columns. Further row operations give the matrix

$$\begin{array}{ccccc} \ell_{e_0} & \ell_{e_1} & \ell_{e_2} & \cdots & \ell_{e_n} \\ \left[\begin{array}{ccccc} 1 & -1 & 0 & \cdots & 0 \\ 0 & 1 & -1 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ & & & \ddots & \\ 0 & 0 & \cdots & 1 & -1 \\ 0 & 0 & 0 & \cdots & 2 \end{array}\right] \end{array},$$

This version also seems to frequently appear in particular computations.

Chapter 6

Implementing the cover graph class

Given an oriented admissible cover $(\widetilde{\Gamma}, \iota, \Gamma, \phi)$ we are going to implement the data in code, called the `CoverGraph class`. The `CoverGraph class` contains five essential objects:

- A dictionary called `configs` which contains the specific information about the number of edges, vertices, and loops
- Two Boolean arrays `CV` for covered vertices and `CE` for covered edges. These Boolean arrays will store information about whether each edge or vertex is fixed by the corresponding involution in the covering graph. These will be discussed at the end of section 6.1
- Two incidence matrices, one called `IM` which will be used for calculating a basis of the homology and another called `sage_IM` which will be a slight modification of `IM` which will be appropriately formatted to implement the SAGE graph isomorphism checking.

6.1 The two incidence matrices

The cover incidence matrix, which we call `IM`, is similar to incidence matrix from the graph class. It is a matrix taking entries in $\{-1, 0, 1\}$. Similar to section 3.4, rows will correspond to vertices and columns will correspond to edges. A -1 entry will signify the start of an edge and a 1 will signify the terminal vertex for an edge. However, a difference between cover incidence matrix and an incidence matrix is that cover incidence matrix will always have the number of columns being twice the number of edges in the base graph and the number of rows being twice the number of vertices in the base graph. The idea is that we allow space for every possibility of vertices and edges in the cover graph, even if in the specific example, an option is not realized.

Before giving the precise definition, we begin with the following example.

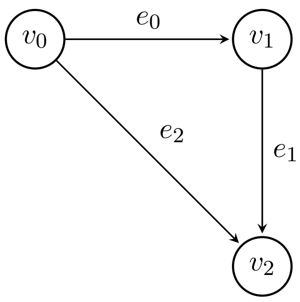

Figure 6.1: Oriented base graph (Γ, ϕ)

The oriented base graph (Γ, ϕ) has three vertices and three edges, and has the following `Incidence_Matrix` (see §3.4).

$$\texttt{Incidence\_Matrix}(\Gamma) = \begin{pmatrix} -1 & 0 & -1 \\ 1 & -1 & 0 \\ 0 & 1 & 1 \end{pmatrix}$$

The oriented covering graph $(\widetilde{\Gamma}, \tilde{\phi})$ is defined by the figure below:

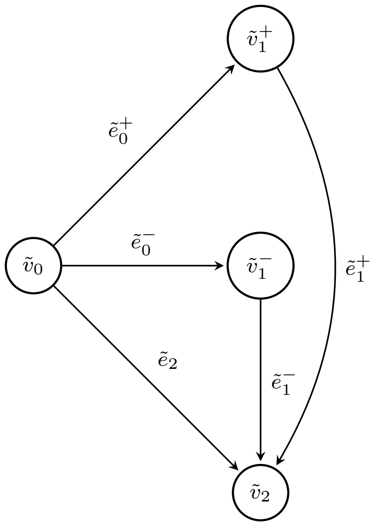

Figure 6.2: Oriented cover graph $(\widetilde{\Gamma}, \tilde{\phi})$ of (Γ, ϕ)

This dual graph to the cover will have the following cover incidence matrix,

$$\text{IM} = \begin{array}{c|cccccc|}
 & \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ \\
\tilde{v}_0^- & -1 & -1 & 0 & 0 & -1 & 0 \\
\tilde{v}_0^+ & 0 & 0 & 0 & 0 & 0 & 0 \\
\tilde{v}_1^- & 1 & 0 & -1 & 0 & 0 & 0 \\
\tilde{v}_1^+ & 0 & 1 & 0 & -1 & 0 & 0 \\
\tilde{v}_2^- & 0 & 0 & 1 & 1 & 1 & 0 \\
\tilde{v}_2^+ & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Observe:

- Row 1 corresponds to $\tilde{v}_0^-$, row 2 corresponds to $\tilde{v}_0^+$, row 3 corresponds to $\tilde{v}_1^-$, row 4 corresponds to $\tilde{v}_1^+$, and so on.
- Column 1 corresponds to $\tilde{e}_0^-$, column 2 corresponds to $\tilde{e}_0^+$, column 3 corresponds to $\tilde{e}_1^-$, column 4 corresponds to $\tilde{e}_1^+$, and so on.
- The vertices $\tilde{v}_0^-$ and $\tilde{v}_2^-$ are uncovered and thus row 2 and row 6 are zero rows.
- Edge $\tilde{e}_2^-$ is uncovered and thus column 6 is a zero row.
- Each edge comes with an orientation in this notation.

The reason why we are handling edges or vertices fixed by ι_E and ι_V in this manner is because when iterating through all possible covering graphs for a given base graph we must have the incidence matrix dimensions be consistent for each scenario. Due to the fact that every cover incidence matrix will have the same dimensions we need a good way to keep track of whether a specific edge (or vertex) in the base graph corresponds to an edge (or vertex) in the cover graph which is fixed by the involution. This information will be stored in the boolean arrays `CV` for covered vertices and `CE` for covered edges. We will see why this method of cover incidence matrix is useful when we discuss calculating a basis of homology.

The Boolean arrays `CV` and `CE` keep track of whether the edge (or vertex) in the base graph corresponds to an edge (or vertex) in the cover graph fixed by the involution; in other words, they keep track of zero columns (and zero rows). The ith entry of covered_edges (or covered_rows) will be a 1 if $\pi^{-1}(e_i) = \{\tilde{e}_i^-, \tilde{e}_i^+\}$ (if $\pi^{-1}(v_i) = \{\tilde{v}_i^-, \tilde{v}_i^+\}$) or a 0 if $\pi^{-1}(e_i) = \{\tilde{e}_i\}$ (if $\pi^{-1}(v_i) = \{\tilde{v}_i\}$). In the above example we have,

$$\text{covered\_edges} = \begin{array}{ccc} e_0 & e_1 & e_2 \\ \hline 1 & 1 & 0 \end{array}, \quad \text{covered\_verts} = \begin{array}{ccc} v_0 & v_1 & v_2 \\ \hline 0 & 1 & 0 \end{array}$$

The other kind of incidence matrix we have in the `CoverGraph class` is called `sage_IM`. This incidence matrix is constructed by taking a copy of the cover incidence matrix `IM` and deleting the zero rows and zero columns (the red rows and columns in the above example). Therefore in the above example we have,

$$\text{sage\_IM} = \begin{bmatrix} -1 & -1 & 0 & 0 & -1 \\ 1 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 1 & 1 \end{bmatrix}$$

Observe here that the rows still correspond to vertices and the columns still correspond to edges but there is not a direct link between the vertices and edges of this incidence matrix and the vertices and edges of the cover graph. The sole purpose of `sage_IM` is to create a new SAGE graph class instance and use it to check isomorphism. We will not be using the `sage_IM` to calculate anything along the lines of homology. Finally, this is a good place to recall that we have ruled out the case where there are loops in the cover graph (see section 5.5).

6.2 Basis of linear forms

Given an oriented, finite graph $(\widetilde{\Gamma}, \tilde{\phi})$ with an admissible oriented involution ι let $C_0(\widetilde{\Gamma}, \mathbb{Z})$ be the free $\mathbb{Z}$-module generated by $V(\widetilde{\Gamma})$ and $C_1(\widetilde{\Gamma}, \mathbb{Z})$ be the free $\mathbb{Z}$-module generated by $\vec{E}(\widetilde{\Gamma})$ (the oriented edges). Recall we define the boundary map as

$$\partial : C_1(\widetilde{\Gamma}, \mathbb{Z}) \to C_0(\widetilde{\Gamma}, \mathbb{Z})$$

given as $\partial(\tilde{e}) = t(\tilde{e}) - s(\tilde{e})$. Define $H_1(\widetilde{\Gamma}, \mathbb{Z})$ to be $\ker \partial$. The involution ι of $\widetilde{\Gamma}$ induces an involution of $C_\bullet(\widetilde{\Gamma}, \mathbb{Z})$ and $H_\bullet(\widetilde{\Gamma}, \mathbb{Z})$. Following the notation of [CMGHL17b, §3.2], define $H_1(\widetilde{\Gamma}, \mathbb{Z})^{\pm}$ to be the eigenspaces of the action of ι on $H_1(\widetilde{\Gamma}, \mathbb{Z})$.

We can consider $\left(\frac{1}{2}(\mathrm{Id} - \iota_E)\right) : H_1(\widetilde{\Gamma}, \mathbb{Z}) \to \frac{1}{2}H_1(\widetilde{\Gamma}, \mathbb{Z})$. Then we define

$$H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} = \mathrm{Im}\left(\frac{1}{2}(\mathrm{Id} - \iota_E)\right) \subseteq \frac{1}{2}H_1(\widetilde{\Gamma}, \mathbb{Z}).$$

Let $C^\bullet(\widetilde{\Gamma}, \mathbb{Z}) = \mathrm{Hom}(C_\bullet(\widetilde{\Gamma}, \mathbb{Z}))$ be the cochain complex associated to $C^\bullet(\widetilde{\Gamma}, \mathbb{Z})$ and then $H^\bullet(\widetilde{\Gamma}, \mathbb{Z})$ is the homology associated to the cochain complex. Notice that $H^i(\widetilde{\Gamma}, \mathbb{Z}) = H_i(\widetilde{\Gamma}, \mathbb{Z})^\vee$ and

$$H^i(\widetilde{\Gamma}, \mathbb{Z})^{[\pm]} = \left(H_i(\widetilde{\Gamma}, \mathbb{Z})^{\pm}\right)^\vee$$

for $i = 0, 1$. By definition $C^1(\widetilde{\Gamma}, \mathbb{Z}) = C_1(\widetilde{\Gamma}, \mathbb{Z})^\vee$ and thus if $\{\tilde{e}_i\}$ generates $C_1(\widetilde{\Gamma}, \mathbb{Z})$, denote $\{\tilde{e}_i^\vee\}$ the dual basis of $C^1(\widetilde{\Gamma}, \mathbb{Z})$. Elements $\tilde{e}^\vee$ are called co-edges.

The basis of linear forms will be by definition a choice of linearly independent generators of the $\mathbb{Z}$-module $H^1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$.

Example 6.2.1. Therefore if

$$H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} = \mathbb{Z}\left\langle \tilde{e}_i - \iota_E \tilde{e}_i \right\rangle$$

then the basis of linear forms is given as,

$$H^1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} = \mathbb{Z} \left\langle \tilde{e}_i^\vee - \iota_E \tilde{e}_i^\vee \right\rangle.$$

6.3 Functionality of the CoverGraph class

The purpose of the `CoverGraph` class is to compute a basis of linear forms for each cover graph. To better understand the way we calculate the basis of linear forms we break up the process into three steps.

(1) Find $H_1(\widetilde{\Gamma}, \mathbb{Z})$

(2) Use $H_1(\widetilde{\Gamma}, \mathbb{Z})$ to find $\mathrm{Im}\left(\frac{1}{2}(\mathrm{Id} - \iota_E)\right)$ called Image_CHB in the program and performed by calling find_image_chb().

(3) Use Image_CHB to find the basis of linear forms, performed by calling find_basis_linear_forms().

Step 1: To find a basis of $H_1(\widetilde{\Gamma}, \mathbb{Z})$ we call the `CoverGraph` class function `get_Homology_basis`. This class function computes the right kernel of the cover incidence matrix as follows,

```
self.Homology_Basis = list(Matrix(self.IM).right_kernel().basis()).
```

The above line of code computes the right kernel of the cover incidence matrix and then returns a basis chosen by SAGE and finally stores it in a list. Recall, the cover incidence matrix has some zero columns corresponding to uncovered edges. The last part of the `get_Homology_basis()` function is to remove the basis vectors which correspond to the zero columns of the cover incidence as these represent uncovered edges and not actual edges in the graph. After the first step we have a basis of $H_1(\widetilde{\Gamma}, \mathbb{Z})$ which we can use to find a corresponding dual basis of $H^1(\widetilde{\Gamma}, \mathbb{Z})$.

Step 2: We are trying to find generators for

$$H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} = \mathrm{Im}\left(\frac{1}{2}(\mathrm{Id} - \iota_E)\right).$$

This process begins by calling the `CoverGraph` class function find_image_chb, this function generates Image_CHB which represents $\mathrm{Im}\left(\frac{1}{2}(\mathrm{Id} - \iota_E)\right)$.

```
def get_Image_CHB(self):
  #get identity - involution matrix
  Identity_Minus_Edge_Involution =
      np.zeros((2*self.configs["edges"],2*self.configs["edges"]),
      dtype=np.int)
  for i in range(self.configs["edges"]):
    if(self.CE[i]):
      Identity_Minus_Edge_Involution[2*i+1][2*i]=-1
      Identity_Minus_Edge_Involution[2*i][2*i+1]=-1
      Identity_Minus_Edge_Involution[2*i][2*i]=1
      Identity_Minus_Edge_Involution[2*i+1][2*i+1]=1
  HB_matrix = np.matrix(self.Homology_Basis)
  Unreduced_Image = np.dot(HB_matrix,Identity_Minus_Edge_Involution)
  self.Image_CHB = Matrix(ZZ,Unreduced_Image).echelon_form()
  zero_rows = []
  for row in range(self.Image_CHB.nrows()):
    is_zero = True
    for col in range(2*self.configs["edges"]):
      if(self.Image_CHB[row,col]!=0):
        is_zero = False
        break
    if(is_zero):
      zero_rows.append(row)
  if(len(zero_rows)>0):
    self.Image_CHB = self.Image_CHB.delete_rows(zero_rows)
```

If we take a closer look at this function we can make the following observations:

- The first part of the function is generating a matrix called `Identity_Minus_Edge_Involution`. If n is the number of edges in the base graph than Identity_Minus_Edge_Involution is a $2n \times 2n$ matrix. We construct `Identity_Minus_Edge_Involution` in diagonal blocks. If the ith edge is covered the diagonal block will look like

$$\begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix}$$

  and if the ith edge is not covered the diagonal block will be the zero matrix.

- The function then defines `Unreduced_Image` to be the matrix product of `HB_matrix` and `Identity_Minus_Edge_Involution` where `HB_matrix` is the matrix whose rows are the cohomology basis.

- Finally the `CoverGraph` class object `Image_CHB` is found by taking the echelon form of `Unreduced_Image` over the ring $\mathbb{Z}$ and deleting the zero rows. This is done through the use of the `echelon_form()` function in the SAGE Matrix library. Notice that when we defined the matrix `Image_CHB` we declared its entries to be in the ring $\mathbb{Z}$ and therefore the echelon form function operations occur over the same ring.

Example 6.3.1. Let us consider the running example from section 6.1. There we found that,

$$\text{Self.IM} = \begin{pmatrix} -1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

and covered_edges $= [1, 1, 0]$. The array covered_edges tells us that only $\tilde{e}_2$ is fixed by the involution. Therefore the resulting permutation matrix representing ι_E and Identity_Minus_Edge_Involution matrix will be,

$$\iota_E = \begin{pmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}, \quad \text{Identity\_Minus\_Edge\_Involution} = \begin{pmatrix} 1 & -1 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Next the function multiplies the homology basis by Identity_Minus_Edge_Involution. That is

$$\text{Image\_CHB} = (\text{HB\_Matrix}) * (\text{Identity\_Minus\_Edge\_Involution})$$

with zero rows removed.

In the running example we have that dual graph to the cover has first homology generated by,

$$\text{HB\_Matrix} = \begin{pmatrix} 1 & -1 & 1 & -1 & 0 & 0 \\ 1 & 0 & 1 & 0 & -1 & 0 \end{pmatrix}.$$

After performing the echelon form function and deleting zero rows we calculate Image_CHB as,

$$\text{Image\_CHB} = \begin{pmatrix} 1 & -1 & 1 & -1 & 0 & 0 \end{pmatrix}.$$

Step 3: After finding `Image_CHB` we are ready to apply the class function `find_basis_Linear_Forms`. To construct the basis of linear forms we start with `Image_CHB`, for each vector (row) in `Image_CHB` we create a linear form. To do this we look at the array which stores the information about the covered edges, `CE`. If the ith edge is not fixed we subtract the $2i+1$ entry from the $2i$ entry to get the ith entry of the linear form vector. If the edge is fixed we take the $2i$ entry of `Image_CHB` to be the ith entry of Basis_Linear_Forms. Let us take a look at the code.

```
def find_basis_Linear_Forms(self):
  self.Basis_Linear_Forms = []
  for j in range(self.Image_CHB.nrows()):
```

```
  b_vector = [0]*self.configs["edges"]
    for i in range(self.configs["edges"]):
      if(self.CE[i]):
        b_vector[i]=self.Image_CHB[j][2*i]-self.Image_CHB[j][2*i+1]
      else:
        b_vector[i]=self.Image_CHB[j][2*i]
    self.Basis_Linear_Forms.append(b_vector)
#Make vectors primitive, Divide by two if possible
for e in range(self.configs["edges"]):
  while(self.is_divisible_by_two(e)):
    for z in self.Basis_Linear_Forms:
      z[e]/=2
self.Basis_Linear_Forms = Matrix(self.Basis_Linear_Forms)
```

we make the following observations:

- The first part of the code takes a row or vector of `Image_CHB` and cycles through each edge of the base graph. If the edge in the base graph is covered we subtract the $2i + 1$ entry from the $2i$ entry to get the ith entry of the linear form vector called `b_vector`. If the edge is not covered we take the $2i$ entry of `Image_CHB` to be the ith entry of `b_vector`

- Then we add each `b_vector` to `Basis_Linear_Forms`.

- Finally we make each basis vector primitive. To do this we take a basis vector or row of `Basis_Linear_Forms` and check if all entries are divisible by two with the row not identically zero. We do this with the class function `is_divisible_by_two`. If the row vector has all entries which are divisible by two and is not the zero vector we divide all entries by two.

Example 6.3.2. If we continue with our example, recall that e_1 is not fixed in the covering graph so we will take `Image_CHB[0]-Image_CHB[1]` and that will be `Basis_Linear_Forms[0]` (keep in mind that these are usually list of arrays but in our example there is only one array). Also e_2 is not fixed so `Image_CHB[3]-Image_CHB[3]` will be `Basis_Linear_Forms[1]`. Finally e_3 is fixed in the covering graph so `Image_CHB[4]=Basis_Linear_Forms[2]`. Hence if

$$\text{Image\_CHB} = \begin{pmatrix} 1 & -1 & 1 & -1 & 0 & 0 \end{pmatrix}$$

we would get the row $[2, 2, 0]$. There is a slight modification, if all the entries in one row are divisible by 2 then we divide the row by two as may times as possible. Thus,

$$\text{Basis\_Linear\_Forms} = \begin{pmatrix} 1 & 1 & 0 \end{pmatrix}.$$

Chapter 7

Generating all Base Graphs

The organization of the computational process can be summarized as follows:

- The first programs generates all base graphs of fixed dimensions (i.e., edges, vertices, and loops) up to isomorphism. The output is stored in text files.
- The second program loads all base graphs from the text files. It then computes all admissible 2:1 covers up to isomorphism, which are also Friedman-Smith graphs, and outputs the linear forms whose squares generate the monodromy cone to a text file.

There are two main reasons for this bifurcation in the code. First, the most time consuming aspect of the computation is graph isomorphism checking. If we separate the graph isomorphism checking process into two compilations – one for base graphs and one for cover graphs – we can better gauge the progress of the compile. The second reason for the separation of code is because under this organizational pattern we have built a database of all base graphs of fixed dimensions up to isomorphism. After the initial compile these can be easily loaded for multiple different types of computations.

Now is a good time to recall that we are really interested in unoriented graphs; the orientation merely provides us with a computational tool that we use to compute useful information. Any orientation chosen for our unoriented graph will provide us with equivalent information regarding homology and hence linear forms of our graph. Therefore we are free to choose a specific orientation which makes enumerating all base graphs easier. Once the vertices are index we can choose the canonical orientation which directs edges from smaller indexed vertices to large index vertices. For example, the following unoriented graph would become the following oriented graph.

v_0
e
v_1
v_0
$\vec{e}$
v_1

Figure 7.1: We give the unoriented graph on the left the canonical orientation which directs edges from smaller indexed vertices to larger indexed vertices. The graph on the right is the oriented version of the graph on the left.

7.1 Generating low dimensional base graphs

We start by describing the process of generating low-dimensional base graphs. This process will be used for generating all graphs with three edges. All higher dimensional graphs will be constructed recursively using lower dimensional graphs. Therefore we must have a process for generating our base case non-recursively.

We begin by constructing a list of all possible edges. An edge, in a very basic sense, stores the information of a terminal vertex and a starting vertex. Computationally we can think of an edge as a list that stores two integer values. The first value being the initial vertex and the second value being the terminal vertex of the edge. To construct all edges we construct all lists of length two taking values within the indexing set of our vertices. This is a rather basic exercise in for loops and the code is below.

```
All_Edges = []
for i in range(configs["verts"]):
  for j in range(i+1,configs["verts"]):
    A = [0]*2
    A[0] = i
    A[1] = j
    All_Edges.append(A)
```

```
max_value = len(All_Edges)-1
```

The information of the graph is completely stored in a list of integers called `incidence_matrix_array`. Recall that an edge is a column of the incidence matrix. Therefore each integer in the list of integers `incidence_matrix_array` will correspond to an edge of the graph and thus a column of the incidence matrix (see section 3.4). In the case where the base graph has 3 vertices the list `All_Edges` would look like,

$$\texttt{All\_Edges} = \left\{ \begin{bmatrix} 0 & 1 \end{bmatrix}, \begin{bmatrix} 0 & 2 \end{bmatrix}, \begin{bmatrix} 1 & 2 \end{bmatrix} \right\}.$$

In this example, `max_value` is the largest index in `All_Edges` and therefore is 2 (recall indexing starts at 0). The list `incidence_matrix_array` is a list of integers taking values between 0 and 2. To avoid generating graphs equivalent up to relabeling edges we require that `incidence_matrix_array` is increasing with each entry. For example, if we consider a graph with three edges and three vertices

$$\texttt{incidence\_matrix\_array} = \begin{bmatrix} 0 & 1 & 1 \end{bmatrix}$$

The 0 in `incidence_matrix_array` represents the edge starting at vertex 0 and ending at vertex 1. The 1 in `incidence_matrix_array` represents the edge starting at vertex 0 and ending at vertex 2. Therefore we get the following graph from this `incidence_matrix_array`.

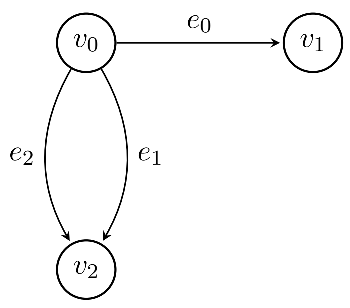

Figure 7.2: The directed graph corresponding to `incidence_matrix_array` = [0 1 1].

The objective is to loop through all possible combinations of `incidence_matrix_array` where the entries are increasing. We can also note that if we define an integer variable `max_repeat` to be

$$\texttt{max\_repeat} = \left\lfloor \frac{2\text{edges}}{\text{vertices}} \right\rfloor$$

this will be the maximum times an edge is allowed to repeat itself and have the base graph remain connected. For a graph to be connected it must have at least the number of vertices minus one distinct edges. We may still generate some disconnected graphs but those will be discarded later in the program.

There are two functions that help us generate the low-dimensional base graphs, `Iterate_vector` and `reset_vector`. The function `Iterate_vector` has two inputs, the list of integers `incidence_matrix_array` (the vector) and an integer m which will represent the `max_value` variable.

```
def Iterate_vector(vector,m):
  for i in range(0,len(vector)):
    if(vector[(len(vector)-1)-i]!=m-int(float(i)/max_repeat)):
      vector[(len(vector)-1)-i] += 1
      reset_vector(vector,len(vector)-1-i)
      return True
  return False
```

To explain this code let us consider the following example. If we have a graph with three vertices and three edges the maximum number of times we can repeat the same edge will be two as to make the graph connected (i.e., `max_repeat`=2). The iteration of `incidence_matrix_array` will go as follows.

$$\begin{bmatrix} 0 & 0 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 2 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & 2 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 1 & 2 \end{bmatrix} \rightarrow \begin{bmatrix} 1 & 2 & 2 \end{bmatrix}$$

Notice that we never repeat an integer more than two times. In the code we check

```
 if(vector[(len(vector)-1)-i]!=m-int(float(i)/max_repeat)):
```

what this is doing is checking that for each value in the array, (vector[len(vector) $-1-i$], is this value the maximum allowable value $\alpha(i)$, where

$$\alpha(i) = \texttt{max\_value} - \left\lfloor \frac{i}{\texttt{max\_repeat}} \right\rfloor .$$

To help understand this convoluted process let us take a look at our running example. Again, the number of vertices are 3 and the number of edges are 3. Therefore the `max_value` is 2 and `max_repeat` is 2. Also len(vector) = 3 which is always the number of edges.

i	vector[len(vector) $-1-i$]	$\alpha(i) = \texttt{max\_value} - \left\lfloor \frac{i}{\texttt{max\_repeat}} \right\rfloor$
0	vector[2]	$\alpha(0) = 2$
1	vector[1]	$\alpha(1) = 2$
0	vector[2]	$\alpha(2) = 1$

Notice that this is precisely the last iteration of `incidence_matrix_array` that we found. Getting back to the if statement in the `Iterate_vector` function, we see that we are looping through each value of the vector starting from the back and checking if that value is the maximum allowable value for that index. If it is we continue the loop and if it is not we add one to the value at that index and reset the vector after that index. We will discuss the `reset_vector` function next but first we need to mention what the return statements do. If the function was able to iterate the vector we return true; otherwise we return false. This will help us use a while loop to perform the iterations.

The `reset_vector` function takes in two inputs: the vector and an integer i which will be the index. The function can be summarized as follows, after incrementing a value in `incidence_matrix_array` we need to reset all of the preceding values to the minimum values that are allowed. Instead of going into the specifics of the code we will use a few examples. Consider when the `incidence_matrix_array` takes the value, `incidence_matrix_array` $= [\ 0 \quad 0 \quad 2\ ]$, if we apply the `Iterate_vector` function to `incidence_matrix_array` we would loop through the indices and at $i = 0$ we see that `incidence_matrix_array`[2] already takes on its maximum value. When $i = 1$ `incidence_matrix_array`[1] $= 0$ and $\alpha(1) = 2$ as we calculated in the table. Thus

`Iterate_vector` increments `incidence_matrix_array`[1] to 1 and we need to call

$$\texttt{reset\_vector}(\texttt{incidence\_matrix\_array}, 1)$$

this will reset all – in this case one – values after index 1. The minimum allowable value for `incidence_matrix_array`[2] is 1 because we haven't repeated the value 1 yet and `incidence_matrix_array` becomes $\begin{bmatrix} 0 & 1 & 1 \end{bmatrix}$. Next, consider when `incidence_matrix_array` equals $\begin{bmatrix} 0 & 2 & 2 \end{bmatrix}$, the `Iterate_vector` will increment the $i = 2$ index. Therefore `incidence_matrix_array`$[0] = 1$ and we call

$$\texttt{reset\_vector}(\texttt{incidence\_matrix\_array}, 0)$$

to reset the entries after index 0. The function `reset_vector` will change `incidence_matrix_array`$[1] = 1$ because we haven't repeated the value 1 in the array. Then when we get to index 2 we let `incidence_matrix_array`$[2] = 2$ and this is because we have now repeated the value 1 twice and `max_repeat` $= 2$.

At this point we have discussed how to iterate through all combinations of `incidence_matrix_array`. If the base graph contains loops, all of the information of the graph is not necessarily contained in the list of integers called `incidence_matrix_array`. There will be more information required to determine where the loops occur. To begin this section we discuss another object which contains the information about the loops called `LA`.

```
LA = Partitions(configs["loops"], max_length=configs["loops"]).list()
```

If we are considering graphs with ℓ loops we define `LA` to be the list of tuples, each of which is an integer partition of ℓ. We can index the vertices of a graph anyway we like and therefore we may choose to have the vertex with loops to be the lowest index vertices. By making this specification we are eliminating some isomorphic graphs that are just permutations of a graph with loops at higher indexed vertices.

Suppose we are looking at graphs with 2 loops ($\ell = 2$). There are two integer partitions of

2. Therefore,

$$\texttt{LA} = [(2), (\ 1 \quad 1\ )].$$

Consider the first integer partition given by the tuple (2) this implies that the base graph will have 2 loops on the vertex indexed by 0. The partition corresponding to the tuple $(1 \quad 1)$ implies that the base graph will have two loops, one on the vertex index 0 and another on the vertex indexed 1.

To construct a member of the `EGraph` we call the `Construct_EGraph` function. This function will have three inputs: `LIMA` which corresponds to a loop incidence matrix array (i.e., a tuple which is a partition of ℓ), `APE` which corresponds to all possible edges (i.e., the list `All_Edges`), and `IMA` which represents the incidence matrix array. The function will return a member of the `EGraph` class which we will use for isomorphism testing.

```
def Construct_EGraph(LIMA,APE,IMA):
  G = DiGraph(configs["verts"],loops=true, multiedges=true)
  edge_count = 0
  IM = np.zeros((configs["verts"],configs["edges"]),dtype=np.int)
  for k in range(len(LIMA)):
    for j in range(LIMA[k]):
      G.add_edge(k,k,edge_count)
      edge_count+=1
  for j in range(len(IMA)):
    G.add_edge(APE[IMA[j]][0],APE[IMA[j]][1],edge_count)
    IM[APE[IMA[j]][0]][edge_count] = -1
    IM[APE[IMA[j]][1]][edge_count] = 1
    edge_count+=1
  E = EGraph(G,IM,configs)
  return E
```

The function begins by defining G to be a digraph that has the required number of vertices, allows for loops, and also allows for multiple edges between two vertices. This DiGraph function is a member of the SAGE graph library. Next we add the edges to our DiGraph G. The first nested for loops will take the integer partition of the number of loops ℓ and add those loops to the DiGraph G using the `add_edge` function from the SAGE graph library. From section 4 we know that we require three things to initialize a member of `EGraph`: a DiGraph G, an incidence matrix `IM`, and a dictionary `configs`. After creating the DiGraph G and adding the loops, we add the non-loop edges and construct `IM`. Recall that the incidence matrix will record the information of the non-loop edges by marking a row -1 if the edge starts at the corresponding vertex and marking a row 1 if the edge terminates at the corresponding vertex (see section 3.4). Finally the function creates a member of the `EGraph` class by using the `EGraph` constructor.

We have now discussed all the preliminaries and are ready to construct all base graphs while checking for isomorphisms. We should mention that upon discovering a new non-isomorphic base graph we write the graph to our designated output file which will later be read by a different program doing specific calculations.

```
All_EGraphs = []
All_Graphs = []
Output_file = str(configs["verts"])+"V"+str(configs["edges"])+"E"
              +str(configs["loops"])+"LBaseGraphs.txt"
with open(Output_file,"w") as f:
  Incidence_Matrix_Array = [0]*(configs["edges"]-configs["loops"])
  reset_vector(Incidence_Matrix_Array,0)
  if(configs["loops"]>0):
    for la in LA:
      while(True):
        E = Construct_EGraph(la,All_Edges,Incidence_Matrix_Array)
        if(E.G.is_connected()):
```

```
        UG = E.G.to_undirected()
        is_new = True
        for g in All_Graphs:
          if(UG.is_isomorphic(g)):
            is_new = False
            break
        if(is_new):
          All_Graphs.append(UG)
          All_EGraphs.append(E)
          print(len(All_EGraphs))
          f.write(str(la)+"\n")
          f.write(str(Incidence_Matrix_Array)+"\n\n")
      if(not Iterate_vector(Incidence_Matrix_Array,max_value)):
        break
    Incidence_Matrix_Array = [0]*(configs["edges"]-configs["loops"])
    reset_vector(Incidence_Matrix_Array,0)

#No Loops
else:
  la = ()
  while(True):
    E = Construct_EGraph_no_loops(All_Edges,Incidence_Matrix_Array)
    if(E.G.is_connected() and not(2 in E.G.degree()
      or 1 in E.G.degree())):
      UG = E.G.to_undirected()
      is_new = True
      for g in All_Graphs:
        if(UG.is_isomorphic(g)):
          is_new = False
          break
      if(is_new):
        All_Graphs.append(UG)
        All_EGraphs.append(E)
        print(len(All_EGraphs))
        f.write(str(la)+"\n")
        f.write(str(Incidence_Matrix_Array)+"\n\n")
  if(not Iterate_vector(Incidence_Matrix_Array,max_value)):
    break
```

One can see that there are two blocks of code within the while loop, one block of code handles the case when we are dealing with loops in the base graph the other handles the case where there are no loops. Both blocks of code are very similar. The block that handles the no loops case is a simplification of the other. Therefore we will only summarize the block pertaining to loops.

After initializing the `incidence_matrix_array` to all zeros we reset it such that it does not have more than the maximum number of repeating integers. Next we check our dimensions for a positive number of loops. The first for loop loops through all possible partitions of the number of loops ℓ. Recall if $\ell = 2$ we have the following tuples in the list `LA`,

$$\texttt{LA} = \left[(2), (1 \quad 1)\right].$$

The local variable `la` will represent the loop array which is an integer partition of ℓ. The next while loop is essentially a do-while loop except `Python` does not permit do while loops. Therefore we are looping while `True` but at the end of the loop we check:

```
if(not Iterate_vector(Incidence_Matrix_Array,max_value)):
  break
```

This will break out of the loop when Incidence_Matrix_Array is done iterating. Next we construct a member of the `EGraph` class called E. This process was discussed in section 4. The next check

```
if(E.G.is_connected()):
```

uses the SAGE graph library to check if E is connected.

If E is connected and has no vertices of valency less than 3 we begin the isomorphism checking. The first step is to create an undirected graph from the DiGraph G member of `EGraph`. We only test graph isomorphisms on undirected graphs. This is to make the process more efficient and because we are only interested in unoriented dual graphs. We put a canonical choice of orientation on the graph solely for the purpose of calculations. We then loop through all previously tested graphs g in the `All_Graphs` list, implementing the `is_isomorphic` function in the SAGE graph library. If the graph is not isomorphic to any of the previously tested graphs we add the undirected graph to the list `All_Graphs` and the `EGraph` to the list `All_EGraphs`. The last three lines of code in the while loop are as follows.

```
print(len(All_EGraphs))
f.write(str(la)+"\n")
f.write(str(Incidence_Matrix_Array)+"\n\n")
```

The first line prints the length of `All_EGraphs` upon adding a new element. The purpose of this is to gauge the progress of the compilation. The next two lines of code write to the output file. We first write the loop array `la` which will tell us where the loops are. Next we write the `Incidence_Matrix_Array` which will tell us what the edges of the base graph are. The purpose of writing to an output file is we can easily load the already isomorphism tested graphs into another `Python` file which will do other computations (i.e., generate all 2:1 admissible covers of the base graphs).

7.2 Recursive base graph generation for all higher dimensional graphs

The process of graph generation is very complicated. The previous sections algorithm is very intuitive but not the most computationally efficient. We will use the previous section to generate low dimensional base graphs – specifically the cases where $e = 3$. After we have created the foundation for our database we will generate the rest of the base graphs recursively.

When generating the graphs recursively we start the same way we do in the low-dimensional context. We populate the list `All_Edges` in the same way as section 7.1. Suppose we are generating all graphs with v vertices, e edges, and ℓ loops. The first thing we must do is load all of the lower dimensional `GraphShells` .

A `GraphShell` is a new class that is the bare essentials of the graph: the loop array `LIMA` and the incidence matrix array `IMA` . This class serves the purpose of loading lower dimensional graphs. We read off the loop array and the incidence matrix array from the database and create all of the `GraphShells` of the lower dimensional graphs. This provides a fast way of reading and storing the old graphs.

```
class GraphShell:
  def __init__(self,LIMA,IMA,configs):
    self.LIMA = LIMA
    self.IMA = IMA
```

There are three cases to consider before loading the `GraphShells` . The first case is when the number of edges is less than the number of vertices. This only occurs when $e = v - 1$ because the graph must be connected. For this special the construction will be summarized as follows:

- Load all graphs with one less edge and one less vertex.
- Adjust the lower dimensional incidence matrix so that it reflects the new number of vertices.
- The higher dimensional graph will have an extra vertex. We create a list of all new edges that start at any vertex and terminate at the extra vertex.

- Create the new `EGraph` .
- Check the new graph for isomorphisms and add to the list if there are no isomorphisms.

```
if configs["edges"]<configs["verts"]:
  lowerDimensionAllEdges = []
  for i in range(configs["verts"]-1):
    for j in range(i+1,configs["verts"]-1):
      A = [0]*2
      A[0] = i
      A[1] = j
      lowerDimensionAllEdges.append(A)

  lowerDimensionGraphShells = LoadGraphs(configs["verts"]-1,
    configs["edges"]-1,0);
  for shell in lowerDimensionGraphShells:
    oldIMA = [x for x in shell.IMA]
    newIMA = [x + lowerDimensionAllEdges[x][0] for x in oldIMA]
    possibleNewEdges = []
    for x in range(1, configs["verts"]):
      possibleNewEdges.append(x * configs["verts"] - x * (x + 1) / 2 - 1)
    for k in range(len(possibleNewEdges)):
      IMA = [x for x in newIMA]
      IMA.append(possibleNewEdges[k])
      IMA.sort()
      E = Construct_EGraph_no_loops(All_Edges, IMA)
      UG = E.G.to_undirected()
      if E.G.is_connected():
```

```
    is_new = True
    for g in All_Graphs:
      if (UG.is_isomorphic(g)):
        is_new = False
        break
    if (is_new):
      All_Graphs.append(UG)
      All_EGraphs.append(E)
```

The second case to consider is when $\ell = 0$. Again let v be the number of vertices and e be the number of edges. First we need to load all `GraphShells` of dimension v vertices, $e - 1$ edges, and $\ell = 0$ loops, call these `lowerDimensionalGraphShells` . For each `shell` in `lowerDimensionalGraphShells` we will loop through ever edge in `All_Edges` and add it to the incidence matrix. Next we will sort the incidence matrix – this is for consistency. Finally we will construct the `EGraph` and do isomorphism checking in the same way from section 7.1.

```
elif configs["loops"] == 0:
  lowerDimensionGraphShells = LoadGraphs(configs["verts"],
                                         configs["edges"] - 1, configs["loops"])
  for shell in lowerDimensionGraphShells:
    for edge in range(len(All_Edges)):
      IMA = [x for x in shell.IMA]
      IMA.append(edge)
      IMA.sort()
      E = Construct_EGraph_no_loops(All_Edges, IMA)
      UG = E.G.to_undirected()
      if(E.G.degree().count(1)<3):
        is_new = True
```

```
for g in All_Graphs:
  if (UG.is_isomorphic(g)):
    is_new = False
    break
if (is_new):
  All_Graphs.append(UG)
  All_EGraphs.append(E)
```

The last case is when the base graph has loops. Again we will let v be the number of vertices, e be the number of edges, and $\ell \neq 0$ be the number of loops. In this case we will load the `GraphShells` having dimensions v vertices, e edges, and $\ell - 1$ loops. This time, for each `shell` in `lowerDimensionalGraphShells` we will loop through all the vertices and add a loop to each vertex. There is a slight complication here. In the program that generates all $2:1$ admissible covers, it is imperative that the loops occur on the lower indexed vertices. That is, we cannot have a loop on vertex zero and a loop on vertex four with no loops on vertex one, two, or three. To remedy this minor adversity we must re-index the vertices of the graph in certain situations. The question is, in which situations do we need to do this?

First of all if we are adding a loop to the vertex of index zero we do not need to re-index. Suppose that the loop list, a copy of `LIMA`, looks like $\texttt{loopList} = [\ell_0, \ell_1, \ell_2]$ and we are adding a loop at vertex ℓ_2. If $\ell_1 > \ell_2$ we do not need to re-index the graph. Finally, if we are adding a loop at vertex v_n and $\texttt{len}(\texttt{loopList}) = n$ then we do not need to re-index the graph.

In all other cases we will need to re-index the vertices. Suppose that we are adding a loop to a non-zero index vertex. If $\texttt{len}(\texttt{loopList}) = 0$. Then we need to transpose the zero vertex with the vertex we are adding a loop to so that the new vertex becomes the zero vertex. If the loop list looks like, $\texttt{loopList} = [\ell_0, \ell_1, \ell_2]$ and we are adding a loop to vertex v_2 with $\ell_1 = \ell_2$ then we need to transpose v_1 and v_2. Finally, if we are adding a loop at vertex v_m and $\texttt{len}(\texttt{loopList}) = n$ with $m > n$ then we need to transpose v_n and v_m.

```
else:
  lowerDimensionGraphShells = LoadGraphs(configs["verts"],
                              configs["edges"] − 1, configs["loops"] − 1)
  for shell in lowerDimensionGraphShells:
    IMA = [x for x in shell.IMA]
    loopList = [x for x in shell.LIMA]
    for v in range(configs["verts"]):
      needIMAChange = v>0 and (len(loopList)==0 or (v<len(loopList)
                    and loopList[v]==loopList[v−1]) or v>len(loopList))
      if needIMAChange:
        firstIndex = 0
        if len(loopList)==0:
          loopList.append(1)
          firstindex = 0
        elif v<len(loopList) and loopList[v] == loopList[v−1]:
          firstIndex = loopList.index(loopList[v])
          #firstIndex might not be v−1, i.e. lL = (1,1,1) and v=2
          loopList[firstIndex]+=1
        elif v>len(loopList):
          firstIndex = len(loopList)
          loopList.append(1)
        else:
          raise ValueError('We have not handled all IMA change cases')
        newIMA = swapVertices(All_Edges,IMA,firstIndex,v)
        E = Construct_EGraph(loopList, All_Edges, newIMA)
        UG = E.G.to_undirected()
        is_new = True
    for g in All_Graphs:
      if (UG.is_isomorphic(g)):
        is_new = False
        break
    if (is_new):
      All_Graphs.append(UG)
      All_EGraphs.append(E)

    loopList = [x for x in shell.LIMA]
    IMA = [x for x in shell.IMA]

  else:
    if v==0 and len(loopList)==0:
      loopList.append(1)
    elif v == 0:
      loopList[v] += 1
    elif v<len(loopList) and loopList[v]<loopList[v-1]:
      loopList[v] +=1
    elif v == len(loopList):
      loopList.append(1)
    else:
      raise ValueError('We have not handled all non-IMA change cases')
    E = Construct_EGraph(loopList, All_Edges, IMA)
    UG = E.G.to_undirected()
    is_new = True
    for g in All_Graphs:
      if (UG.is_isomorphic(g)):
```

```
    is_new = False
    break
if (is_new):
  All_Graphs.append(UG)
  All_EGraphs.append(E)
loopList = [x for x in shell.LIMA]
IMA = [x for x in shell.IMA]
```

Chapter 8

Generate All Possible Dual Graphs for 2:1 Coverings

We generate all admissible 2:1 covers of our base graphs in a separate `Python` program. The first step of the program is to re-populate the `All_EGraphs` list, the list of base graphs, by reading in the information from the output file produced using the techniques of section 7. We will use the information from the base graph and the two lists, covered edges abbreviated as `CE` and covered verts abbreviated as `CV` to produce and admissible cover. The two lists, `CE` and `CV`, are boolean list which will record information about whether the corresponding edge or vertex is fixed or unfixed by the involutions ι_V and ι_E. Essentially, for every base graph we will cycle through every combination of `CV` and `CE` and check whether the resulting covering incidence matrix is an admissible cover.

In the case where the base graph has loops there are normally two ways to cover the loop in the base graph (see section 5.4). We are able to reduce the case to 2:1 covers that contain no loops in the cover (see section 5.5). Therefore if there is a loop in the base graph at vertex v_i we can guarantee that the vertex v_i must be covered and thus $\texttt{CV}[i] = \texttt{True}$. Recall that we chose to label the vertices such that any loops would occur on the lowest indexed vertices. We also chose to label edges such that loops are the lowest indexed edges. Thus if there are ℓ loops in the base graph the first ℓ entries of `CE` must be `True`.

8.1 Generating all possible values of `CE` and `CV`

Before generating any cover graphs we must populate the lists `APEC` and `APVC` which stand for all possible edge covers and all possible vertex covers respectively. From the arguments directly above about loops we are able to initialize some entries of the `CE` list to always be `True`. Verifying that loop vertices are covered and thus correspond to `True` values in `CV` will be discussed in section 8.2. To generate all possibilities for `CE` and `CV` we need to discuss to functions called `reset_Boolean_List` and `iterate_Boolean_List`. These two functions will be very similar to `Iterate_vector` and `reset_vector` from section 7.1. Before we were iterating through a list of integers and now we are iterating through a list of Boolean values (i.e., `True` and `False`).

```
def reset_Boolean_List(B, col):
  if(col == len(B)-1):
    return
  for i in range(col+1,len(B)):
    B[i] = False
def iterate_Boolean_List(B):
  for i in range(len(B)):
    if(B[len(B)-1-i] == False):
      B[len(B)-1-i] = True
      reset_Boolean_List(B,len(B)-1-i)
      return True
  return False
```

Looking at the above code we see that `iterate_Boolean_List` takes a Boolean list B which will be `CE` or `CV` (and later `ccl_edges_isCrossed`) and starts from the last index in the list, `B[len(B)-1]` and checks if the entry is `False`. If we reach an entry of `B` which is `False` we set it to be `True` and reset the vector to be all `False` after the index we changed by calling the `reset_Boolean_List` function. If we consider a base graph with three edges and no loops the progression of `CE` is as follows.

$$\begin{bmatrix} \texttt{False} & \texttt{False} & \texttt{False} \end{bmatrix} \rightarrow \begin{bmatrix} \texttt{False} & \texttt{False} & \texttt{True} \end{bmatrix} \rightarrow \begin{bmatrix} \texttt{False} & \texttt{True} & \texttt{False} \end{bmatrix} \rightarrow$$
$$\begin{bmatrix} \texttt{False} & \texttt{True} & \texttt{True} \end{bmatrix} \rightarrow \begin{bmatrix} \texttt{True} & \texttt{False} & \texttt{False} \end{bmatrix} \rightarrow \begin{bmatrix} \texttt{True} & \texttt{False} & \texttt{True} \end{bmatrix} \rightarrow$$
$$\begin{bmatrix} \texttt{True} & \texttt{True} & \texttt{False} \end{bmatrix} \rightarrow \begin{bmatrix} \texttt{True} & \texttt{True} & \texttt{True} \end{bmatrix}$$

If we had loops in the base graph some of the initial values would be fixed to be true throughout all iterations. We are now ready to populate `APEC` and `APVC` .

```
APEC = []

CEdges = [false]*configs["edges"]

for l in range(configs["loops"]):

  CEdges[l] = True

while(True):

  CE = list(CEdges)

  APEC.append(CE)

  if(not iterate_Boolean_List(CEdges)):

    break

APVC = []

CVerts = [False]*configs["verts"]

APVC.append(FV)

while(True):

  CV = list(CVerts)

  APVC.append(CV)

  if(not iterate_Boolean_List(CVerts)):

    break
```

The two list `APEC` and `APVC` will give us all possible combinations of vertex and edge coverings. It turns out that many of these combinations will not lead to admissible covers but this gives us a way to check all possibilities. The next step is to talk about how to check whether a potential cover graph is admissible but before we can do this we need to discuss the scenario of edges becoming crossed in the cover graph.

8.2 Resolving Crossings in the Covering Graph

A very observant reader will notice that we are making a choice in the case where an edge e_i is not fixed by ι_E and both $s(e_i)$ and $t(e_i)$ are also not fixed by ι_V. There are two options for this covering.

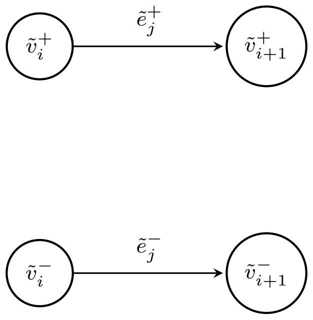

Figure 8.1: Covering Option 1

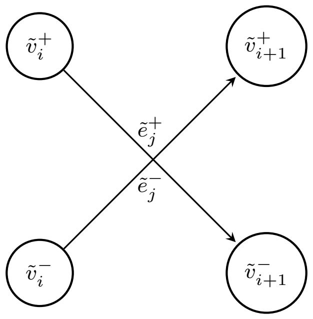

Figure 8.2: Covering Option 2

In the case of covering option 2 we say that the edge e_j lifts to crossing edges $\tilde{e}_j^-$ and $\tilde{e}_j^+$. Before we talk about handling multiple covering options we need a definition.

Definition 8.2.1. If e is an edge in Γ, covered by distinct edges $\tilde{e}^+, \tilde{e}^-$ such that $s(e)$ and $t(e)$ are both not fixed by the involution ι_V then we will call the edge e a **completely covered** edge (see Figures 8.1 and 8.2). Let $v_1 = s(e)$ and $v_2 = t(e)$. By assumption, there are distinct vertices $\tilde{v}_1^\pm$ (resp. $\tilde{v}_2^\pm$) lying over v_1 (resp. v_2). Now we say that in this case $\tilde{e}^+, \tilde{e}^-$ are **crossed** (with respect to our choice of labeling of $v_i^\pm$) if $s(\tilde{e}^\pm) = \tilde{v}_1^\pm$ and $t(\tilde{e}^\pm) = \tilde{v}_2^\mp$, or $s(\tilde{e}^\pm) = \tilde{v}_1^\mp$ and $t(\tilde{e}^\pm) = \tilde{v}_2^\pm$ (see Figure 8.2). Otherwise, we say $\tilde{e}^+, \tilde{e}^-$ are uncrossed (see Figure 8.1).

Remark 8.2.2. It is important to note that the notion of a crossed edge depends on the labeling of the vertices of the graph. As graphs, Figure 8.1 and Figure 8.2 are isomorphic; we simply relabel $v_{i+1}^{\pm} \mapsto v_{i+1}^{\mp}$. The importance of the notion of crossed edges comes in to play in the way we enumerate graphs, by providing a way to reduce the number of isomorphic copies of the same covering graphs that we construct.

Roughly speaking, the way that we will construct the admissible covers of a given base graph is to enumerate every possible edge and vertex cover type; e.g., for two distinct vertices joined by an edge, we can have one or two vertices over each vertex in the base, and one or two edges, meeting the vertices in the cover graph in various ways. Unfortunately, this is not very efficient. One way to help keep track of the various possibilities, and hopefully reduce the number of possibilities, is to consider how many edges are crossed with respect to our choice of labeling. In fact, it is natural to consider whether there is some choice of labeling that could "uncross" all of the edges. The answer is no, as demonstrated by the following example.

Example 8.2.3. If we start with the simple base graph Γ depicted below,

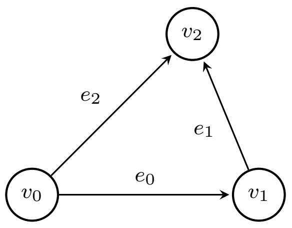

Figure 8.3: Base Graph Γ

we can have two potential admissible covers for Γ.

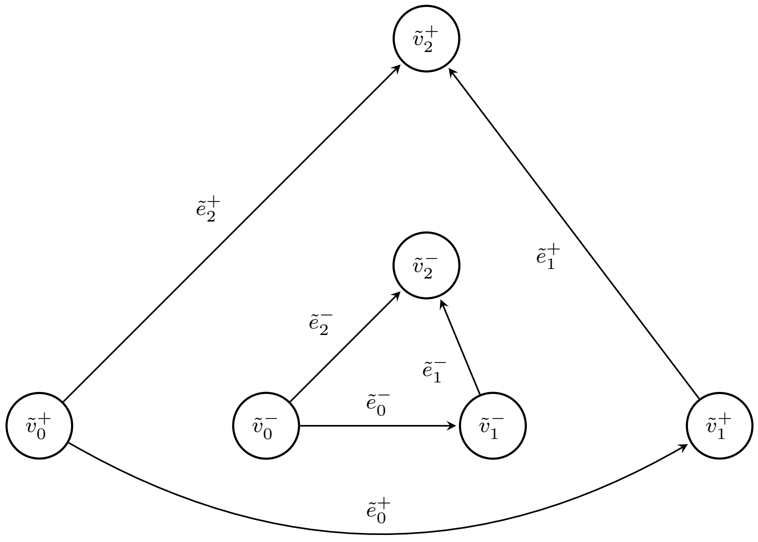

Figure 8.4: Covering e_2 using option 1

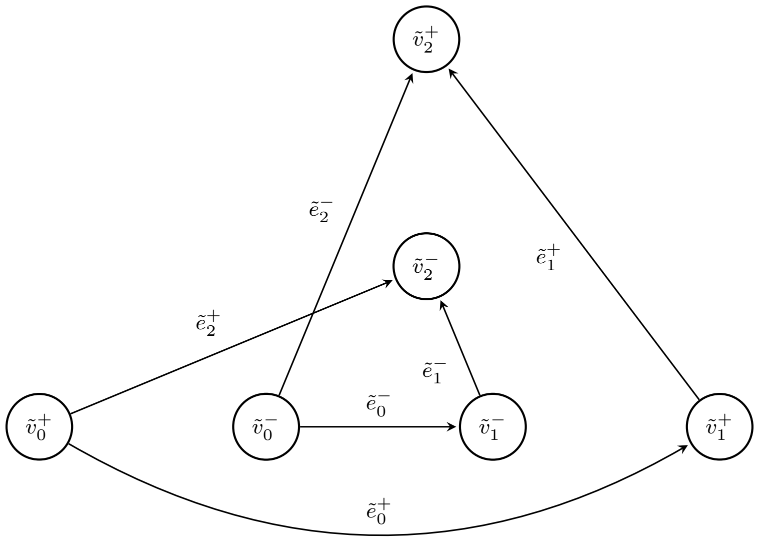

Figure 8.5: Covering e_2 using option 2

The two admissible covers are not isomorphic, one is connected and the other is not. This is a great illustration of why we cannot resolve all crossings in a covering graph.

Unfortunately we cannot resolve all crossings and thus our goal becomes to reduce the maximum number of edges that can lift to crossings up to isomorphism. This will lead to fewer possible admissible covers for each base graph with completely covered edges – more importantly less graph isomorphism testing.

The first reduction in this process is loops. Any loop in an admissible cover must be a completely covered edge. We can reduce any admissible covers with loops to graphs of smaller dimensions (see section 5.5). Furthermore any loop only has one choice of covering (see section 5.4) and we do not need to create multiple covering options for completely covered edges that are loops.

In making the next reduction we need to define some helpful vocabulary. We are working with completely covered edges, specifically completely covered edges that are crossed in the covering graph. The easiest way to resolve a crossing of an edge would be to perform an involution on the starting or terminal vertex of the edge in the cover graph (i.e., simply relabel the vertices). The problem with this is that any other completely covered edges sharing a valency with this vertex may become crossed. In identifying the right vertices to involute we need to consider a special kind of vertex degree.

Definition 8.2.4. Suppose $\widetilde{\Gamma}$ is an admissible covering graph of a dual graph Γ. Let $v \in V(\Gamma)$ such that v is not fixed by ι_V in $\widetilde{\Gamma}$. Consider all the non-loop edges in $E(\Gamma)$ that lift to completely covered edges in $\widetilde{\Gamma}$ which have a valency at v. Suppose that n non-loop edges in Γ lift to completely covered edges and have a valency at v, we say that v has completely covered edge degree of n_v. Also, assume that m of these edges also lift to crossings (thus $m \leq n$), we say that that v has completely covered crossed edge degree of m_v. If $m_v > \frac{n_v}{2}$ we say that the vertex v has **overloaded crossed edge degree**; motivation for this terminology will become clear shortly.

Example 8.2.5. Consider the following dual graph Γ and admissible covering graph $\widetilde{\Gamma}$.

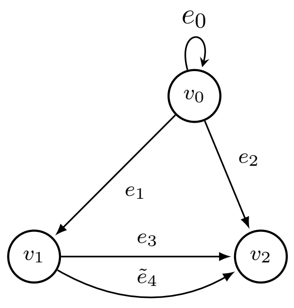

Figure 8.6: Base Graph Γ

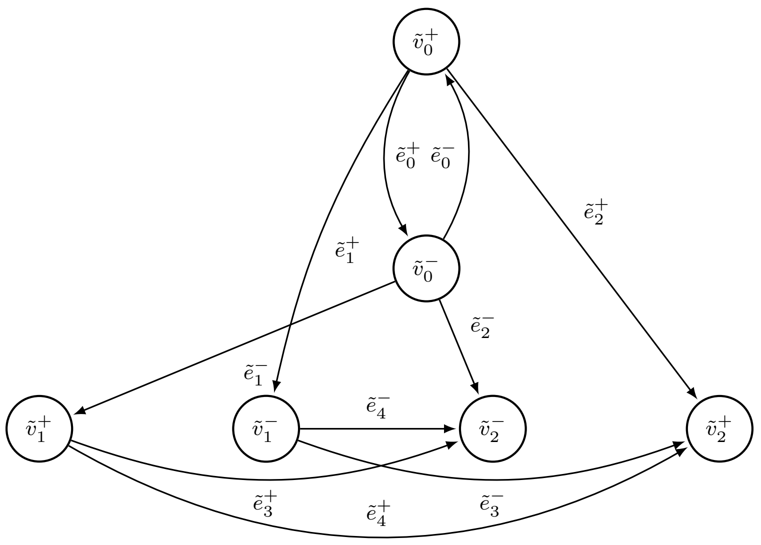

Figure 8.7: Admissible Cover Graph $\widetilde{\Gamma}$

In this example we see the vertex $v_1 \in V(\Gamma)$ has overloaded crossed edge degree because both e_1 and e_3 lift to crossings in $\widetilde{\Gamma}$; thus $m_{v_1} = 2$ and $n_{v_1} = 3$. The vertex $v_2 \in V(\Gamma)$ does not have overloaded crossed edge degree because only e_3 lifts to a crossed edge in $\widetilde{\Gamma}$; thus $m_{v_2} = 1$ and $n_{v_2} = 3$. Finally, $v_0 \in V(\Gamma)$ does not have overloaded crossed edge degree because we do not consider the loop $e_0 \in E(\Gamma)$ when determining crossed edge degree and e_1 lifts to a crossing but e_2 does not; therefore $n_{v_3} = 2$ and $m_{v_3} = 1$.

Lemma 8.2.6. *Let $\widetilde{\Gamma}$ be an admissible cover of a dual graph Γ which has $n > 0$ completely covered edges that are not loops. Suppose that m of these completely covered edges that are not loops lift to crossings. If $m > \frac{n}{2}$ then there exist a vertex in Γ that has overloaded crossed edge degree.*

Proof. If $n > 0$ there exist at least two vertices which are not fixed by ι_V in $\widetilde{\Gamma}$. Let S be the set of vertices that are not fixed by ι_V in $\widetilde{\Gamma}$. Using the fact that $\widetilde{\Gamma}$ is an admissible cover, on the set of vertices S we have $2n$ valencies coming from completely covered non-looped edges and $2m$ valencies coming from completely covered non-loop edges lifting to crossings. Let us assume that there does not exist a vertex $v \in S$ that has overloaded crossed edge degree. Then

$$2n = \sum_{v \in S} n_v \quad \text{and} \quad 2m = \sum_{v \in S} m_v$$

We are assume that there are no overloaded crossed edge degree vertices in S and so $2m_v < n_v$ for all $v \in S$. Then,

$$2m = \sum_{v \in S} m_v < \sum_{v \in S} \frac{n_V}{2} = \frac{\sum_{v \in S} n_v}{2} = n.$$

This contradiction and therefore there must be a vertex $v \in S$ that has overloaded crossed edge degree. □

Lemma 8.2.7. *Let $\widetilde{\Gamma}$ be an admissible cover of a dual graph Γ. If there exist a vertex $v \in V(\Gamma)$ that has overloaded crossed edge degree, then we can perform an isomorphism on $\widetilde{\Gamma}$ that reduces the number of edges that lift to crossings in $\widetilde{\Gamma}$.*

Proof. If $v \in V(\Gamma)$ has overloaded crossed edge degree then suppose that v has a valencies from edges that lift to crossings and b valencies from edges that do not lift to crossings. Then $a > b$. Since v has two distinct vertices $\tilde{v}^{\pm}$ lying over it, we can construct a new graph $\widetilde{\Gamma}'$ by relabeling $\tilde{v}^{\pm}$ by $\tilde{v}^{\mp}$. This is clearly an isomorphic graph since it is a relabeling of two vertices. After this isomorphism, the a edges having a valency at v that previously lifted to crossings will no longer lift to crossings and the b edges that previously did not lift to crossings having a valency at v will now lift to crossings. Given that $a > b$, we reduced the number of edges that lift to crossings in Γ. □

The previous two lemmas prove the following proposition.

Propostion 8.2.8. Let $\widetilde{\Gamma}$ be an admissible cover of a dual graph Γ which has $n > 0$ completely covered edges that are not loops. Suppose that m of these completely covered edges that are not loops lift to crossings. Through a series of isomorphisms we can always reduce m such that $m \leq \frac{n}{2}$.

Proposition 8.2.8 tells us that when the number of completely covered non-loop edges that lift to crossings in $\widetilde{\Gamma}$, m, exceeds the number of completely covered non-loop edges n, divided by two we can reduce this cover graph to a cover graph with less edges lifting to crossings. That is if

$$m > \frac{n}{2}$$

then we do not have to construct a possible cover for this set of `CVerts` and `CEdges`. We call $n - 2m$ the `crossingThreshold` and use it to reduce the number of possible admissible covers we need to check.

To implement this reduction we need the class function `setNumberCCEdges`. This function is member of the `EGraph` class and takes in three arguments: the `EGraph`, `CVerts`, and `CEdges`. The first thing this function does is set the Boolean value `are_LVs_covered` (are loop vertices covered). Recall that all loop vertices must be covered as there is only one option to cover a loop in the base graph such that the cover graph does not have any loops. If all the loop vertices correspond to `True` in `CVerts` then `are_LVs_covered` will be set to `True`. Conversely if one of the loop vertices is not covered then `are_LVs_covered` will be set to false. In generating all cover graphs over a base graph, `are_LVs_covered` will give a quick check to see if an admissible cover is possible.

The second step of `setNumberCCEdges` is to simultaneously set the number of completely covered edges in E, `numberCCEdges`, and to construct an array which indicates whether or not an edge is completely covered in E, `ccEdges`. To do this we first count all the loops and set their values to `True` in `ccEdges`. Then loop through the rest of the non-loop edges and check whether or not they are completely covered and act accordingly with respect to `numberCCEdges` and `ccEdges`.

```
def setNumberCCEdges(self,CV,CE):
  loop_verts = self.G.loop_vertices()
  self.are_LVs_covered = True
  for v in loop_verts:
    if(not CV[v]):
      self.are_LVs_covered = False
      break
  self.numberCCEdges = self.configs["loops"]
  #Number of completely covered edges including loops
  self.ccEdges = [False]*self.configs["edges"]
  #array of completely covered edges that are not loops
  #make sure loops are declared cc edges
  for e in range(self.configs["loops"]):
    self.ccEdges[e]=True
  for e in range(self.configs["loops"],self.configs["edges"]):
    if(CE[e] and CV[self.start(e)] and CV[self.end(e)]):
      self.numberCCEdges +=1
      self.ccEdges[e] = True
```

8.3 Get a possible cover

We first generate possible covers from a specific base graph and elements of the lists `APEC` and `APVC` . These possible covering graphs may not be admissible and the next step will be checking admissibility. Let us first examine the `get_Possible_Cover_IM` function in the context where the base graph will have no completely covered edges – note that this will implicitly base graphs with loops as loops are completely covered edges.

```
def get_Possible_Cover_IM_no_CCEdges(E, CV, CE):
  Possible_Cover_IM = np.zeros((2 * configs["verts"], 2 * configs["edges"]),
                          dtype=np.int)
  # copy base graph
  for v in range(configs["verts"]):
    for e in range(configs["edges"]):
      Possible_Cover_IM[2 * v][2 * e] = E.IM[v][e]


  # Covering Information
```

```
for e in range(configs["edges"]):
  # Edge is covered
  if (CE[e]):
    if (CV[E.start(e)]):
      Possible_Cover_IM[2 * E.start(e) + 1][2 * e + 1] = -1
    else:
      Possible_Cover_IM[2 * E.start(e)][2 * e + 1] = -1
    if (CV[E.end(e)]):
      Possible_Cover_IM[2 * E.end(e) + 1][2 * e + 1] = 1
    else:
      Possible_Cover_IM[2 * E.end(e)][2 * e + 1] = 1

# No loops in this base graph
return Possible_Cover_IM
```

This function has 3 inputs: an `EGraph` class object E, a Boolean list `CV` which will be an element of `APVC` , and a Boolean list `CE` which will be an element of `APEC` . The output is a 2-dimensional double list that will serve as the incidence matrix of a potentially admissible cover. The first step is to input the information of the base graph. Let us go back to the example from the section 5.3.

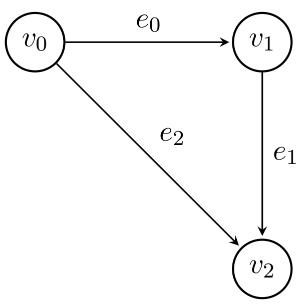

Figure 8.8: Base Graph E

This base graph will have the following Incidence_Matrix

$$\texttt{E.IM} = \begin{pmatrix} -1 & 0 & -1 \\ 1 & -1 & 0 \\ 0 & 1 & 1 \end{pmatrix}.$$

After `get_Possible_Cover_IM_no_CCEdges` copies the information from the incidence matrix of the base graph E to the new possible incidence matrix `Possible_Cover_IM` is as follows,

$$\texttt{Possible\_Cover\_IM} = \begin{pmatrix} -1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

where we have padded each row and column with zero rows and columns. Returning to the example suppose that

$$\texttt{CV} = [\texttt{False}, \texttt{True}, \texttt{False}] \quad \text{and} \quad \texttt{CE} = [\texttt{True}, \texttt{True}, \texttt{False}].$$

Recall from the definitions of `CE` and `CV` in the introduction to section 6 that this implies that v_0 and v_2 are not covered (fixed by ι_V) while v_1 is covered (not fixed by ι_V). Also e_0 and e_1 are covered (not fixed by ι_E), while e_2 is not covered (fixed by ι_E).

Then possible covering graph corresponding to the base graph E with these specific `CV` and `CE` will look like the following.

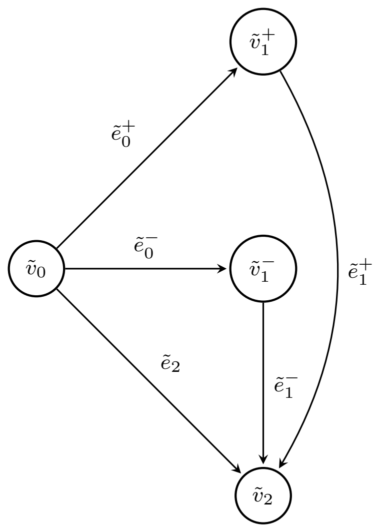

Figure 8.9: Possible Cover of E

The second part of the function deals with the covering information. We loop through `CE` and when we reach a covered edge, suppose it has index k, we check to see whether the vertex where e_k starts is also covered. If $s(e_k){=}v_\ell$ is covered then we give the corresponding $\tilde{v}_\ell^+$ a corresponding -1 for the edge $\tilde{e}_k^+$. If $s(e_k) = v_\ell$ is not covered then we give $\tilde{v}_\ell^-$ a corresponding -1 for the edge $\tilde{e}_k^+$. We treat $t(e_k)$ the same.

Let us look at our example. The list `CE` tells us that e_0 is covered, we find that $s(e_0) = v_0$ is not covered so the second column, corresponding to $\tilde{e}_0^+$, of `Possible_Cover_IM` will get a -1 in the corresponding $\tilde{v}_0^-$ entry. Also $t(e_0) = v_1$ and `CV` tells us that v_1 is covered. Therefore the column corresponding to $\tilde{e}_0^+$ will get a 1 in the $\tilde{v}_1^+$ row. Therefore after dealing with e_0 we have,

$$\texttt{Possible\_Cover\_IM} = \begin{pmatrix} -1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

After addressing e_1 we get,

$$\texttt{Possible\_Cover\_IM} = \begin{pmatrix} -1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

Now in the case where an edge, e_k, is not covered we leave the column corresponding to $\tilde{e}_k^+$ a zero column. In the example e_2 is not covered and so the last column is zero. That is the output of `Possible_Cover_IM` will be,

$$\texttt{Possible\_Cover\_IM} = \begin{pmatrix} -1 & -1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}.$$

In the context where the base graph has completely covered edges the process is very similar but there are more aspects to consider. In this situation we allow completely covered edges to lift to crossings. We will need more inputs in our `get_Possible_Cover_IM` function to accommodate this.

We will introduce three new lists. The first list is called `cc_edges` and it is a Boolean list having the same size as number of edges in the base graph. This list stores the information of whether or not an edge in the base graph is completely covered, notice that all loops are completely covered edges by construction. This list will be a member of the `EGraph` class and is populated with the function `setNumberCCEdges` from the `EGraph` class as discussed in section 8.2.

The second and third lists are called `cc_edge_locations` and `cc_edges_isCrossed`. The list `cc_edge_locations` will be a list having length equal to the number of completely covered edges and each entry is an index of a completely covered edge. The list `cc_edges_iscrossed` is a list with length also equal to the number of completely covered edges. This list represents whether each completely covered edge lifts to a crossing. The list `cc_edges_isCrossed` will be initially set to all `False` except for indices that represent loops. Loops must lift to crossings by construction. We will then iterated through all combinations of `cc_edges_isCrossed` . To see how `cc_edge_locations` is populated and how `cc_edges_isCrossed` is intitialized, consider the following code.

```python
# Determine which edges are completely covered
cc_edge_locations = []
# Get locations of completely covered edges
for e in range(configs["edges"]):
  if (E.ccEdges[e]):
    cc_edge_locations.append(e)
cc_edges_iscrossed = [False] * E.numberCCEdges
# make sure loops are always crossed
for i in range(len(cc_edge_locations)):
  if (cc_edge_locations[i] < configs["loops"]):
    #loops come first in cc_edge_locations
    cc_edges_iscrossed[i] = True
```

We are now ready to discuss the function `get_Possible_Cover_IM` in the context of E having completely covered edges. This function will have 6 inputs, three more then the context of no completely covered cycles, the three extra inputs will correspond to the lists `cc_edges` , `cc_edge_locations` , and `cc_edges_isCrossed` . As before this function will output a two-dimensional list `Possible_IM` which will represent the incidence matrix of a potential cover to the base graph E.

```python
def get_Possible_Cover_IM(E, CV, CE, cc_edges, cc_edges_iscrossed,
                        cc_edges_locations):
Possible_Cover_IM = np.zeros((2 * configs["verts"], 2 * configs["edges"]),
                        dtype=np.int)
# copy base graph
for v in range(configs["verts"]):
  for e in range(configs["edges"]):
    Possible_Cover_IM[2 * v][2 * e] = E.IM[v][e]
# e is a loop and must be crossed (no loops in cover graphs)
loop_num = 0
graph_loops = E.G.loop_edges()
for loop in graph_loops:
  vertex = loop[0]
  Possible_Cover_IM[2 * vertex][2 * loop_num] = -1
  Possible_Cover_IM[2 * vertex + 1][2 * loop_num + 1] = -1
  Possible_Cover_IM[2 * vertex][2 * loop_num + 1] = 1
  Possible_Cover_IM[2 * vertex + 1][2 * loop_num] = 1
  loop_num += 1
# Covering Information
for e in range(configs["loops"], configs["edges"]):
  edge_start = E.start(e)
  edge_end = E.end(e)
  # Edge is part of ccl and should be crossed
  if (cc_edges[e] and cc_edges_iscrossed[cc_edges_locations.index(e)]):
    Possible_Cover_IM[2 * edge_start][2 * e] = -1
    Possible_Cover_IM[2 * edge_start + 1][2 * e + 1] = -1
    Possible_Cover_IM[2 * edge_end][2 * e + 1] = 1
    Possible_Cover_IM[2 * edge_end + 1][2 * e] = 1
```

```
        Possible_Cover_IM[2 * edge_end][2 * e] = 0
      elif (CE[e]):
        if (CV[edge_start]):
          Possible_Cover_IM[2 * edge_start + 1][2 * e + 1] = -1
        else:
          Possible_Cover_IM[2 * edge_start][2 * e + 1] = -1
        if (CV[edge_end]):
          Possible_Cover_IM[2 * edge_end + 1][2 * e + 1] = 1
        else:
          Possible_Cover_IM[2 * edge_end][2 * e + 1] = 1
    return Possible_Cover_IM
```

The functions `start` and `end` are functions that we input an edge and the function returns the index of the starting or ending vertex respectively. The first step of `get_Possible_Cover_IM` is to input the information from `E_IM` into `Possible_IM`. The next step is to deal with the loops. Loops in the base graph must be completely covered and there is only one way to cover them which corresponds to crossing them in the cover graph. This corresponds to covering option 2 from the section on loop crossings, section 5.4. After dealing with the initial loop edges we deal with the non-loop edges. If e corresponds to a non-loop edge the first thing we do is get the indices of $s(e)$ and $t(e)$, this will be useful when deciding how to cover e. Next we check if e is a part of a completely covered loop and if e is crossed in the list `ccl_edges_isCrossed`. In this case we fill in the entries of `Possible_IM` corresponding to $\tilde{e}^+$ and $\tilde{e}^-$ using covering option 2 in section 8.2. Otherwise, if e is not a part of a completely covered loop and e is covered we fill in the entries of `Possible_IM` corresponding to $\tilde{e}^+$ and $\tilde{e}^-$ using covering option 1 in section 8.2.

Example 8.3.1. In order to better explain the `get_Possible_Cover_IM` function we will consider the following base graph.

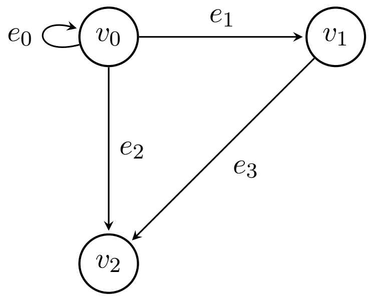

Figure 8.10: Base Graph G

Let `CV` and `CE` be as follows.

`CV` = [`True`, `True`, `True`] and `CE` = [`True`, `True`, `True`, `True`]

If we apply the class function `setNumberCCEdges` to the base graph G with `CV` and `CE` defined as such then it would look like,

`G.setNumberCCEdges(CV,CE).`

Notice that every edge and every vertex is covered; therefore, all edges are completely covered. That is

`cc_edges` = [`True`, `True`, `True`, `True`] and `cc_edge_locations` = [0, 1, 2, 3].

Suppose we are give the following iteration of `cc_edges_isCrossed` .

`cc_edges_isCrossed` = [`True`, `True`, `False`, `False`].

In the function `get_Possible_Cover_IM` we start by transferring the information from the incidence matrix of G into `Possible_IM.` After this step `Possible_IM` will look as follows.

$$\texttt{Possible\_IM} = \begin{bmatrix} 0 & 0 & -1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

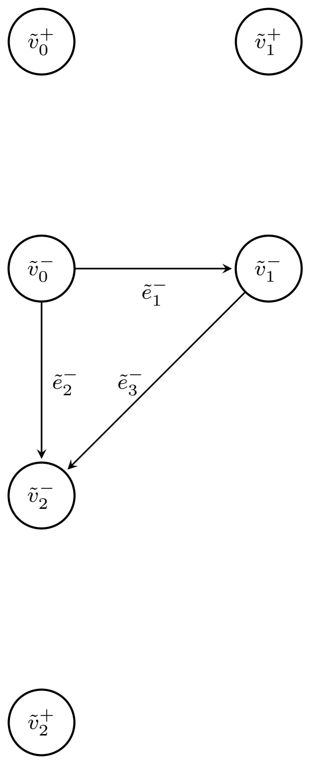

Figure 8.11: $\widetilde{\Gamma}$ after copying incidence matrix

The next step of `get_Possible_IM` is to insert the crossing information from the loops in the base graph. In our base graph G we have one loop on v_0. Therefore after this step `Possible_IM` will look as follows

.

$$\texttt{Possible\_IM} = \begin{bmatrix} -1 & 1 & -1 & 0 & -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

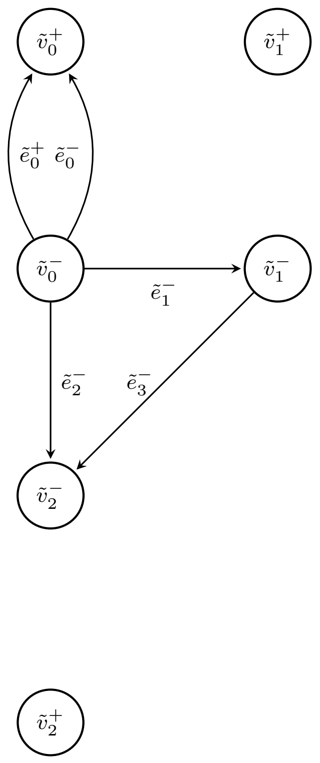

Figure 8.12: $\widetilde{\Gamma}$ after handling loops

Next the function will handle the covering information. We loop through all non-loop edges of the base graph G and interpret how to cover each edge. In the case of e_1 we find that $s(e_1) = v_0$ and $t(e_1) = v_1$. Then we check and see that e_1 is part of a completely covered cycle and corresponds to a `True` value in `cc_edges_isCrossed` therefore we cover e_1 as follows.

$$\texttt{Possible\_IM} = \begin{bmatrix} -1 & 1 & -1 & 0 & -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

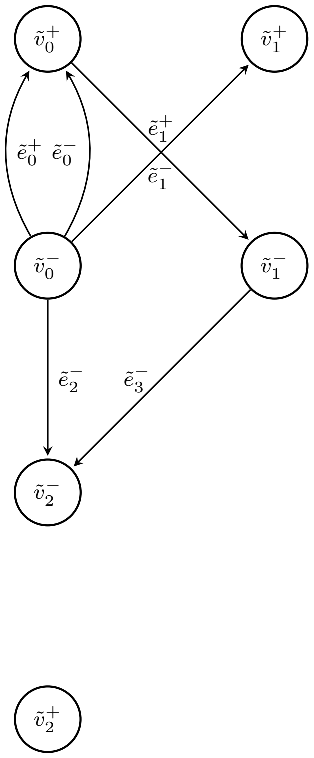

Figure 8.13: $\widetilde{\Gamma}$ after covering e_1

For the next two edges we check and find out that both e_2 and e_3 are a part of a completely covered cycle and both correspond to `False` values in `cc_edges_isCrossed.` After covering these final two edges we return the following `Possible_IM` which will then be tested for admissibility.

$$\texttt{Possible\_IM} = \begin{bmatrix} -1 & 1 & -1 & 0 & -1 & 0 & 0 & 0 \\ 1 & -1 & 0 & -1 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix}$$

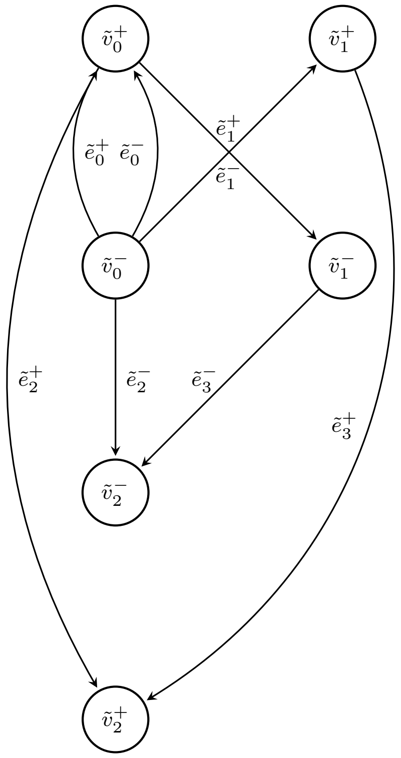

Figure 8.14: Possible Cover of G with `ccl_edges_isCrossed` = [`True`, `True`, `False`, `False`]

At this point in the program we can taken a base graph E from the list `All_EGraphs` , assign it two specific list `vc` and `ec` from `APVC` and `APEC` respectively, and constructed a possible covering incidence matrix called `Possible_IM` . We do not know whether this possible incidence matrix corresponds to an admissible cover. That is, we have not checked the admissibility conditions discussed in section 5.

8.4 Checking cover graphs for admissibility

After we generate a possible cover graph for a particular base graph the next vital step will be to check whether the possible incidence matrices actually represents an admissible cover. At the heart of this process is the `is_Cover` function. Similar to the `get_Possible_Cover_IM` function we will have two `is_Cover` functions, one for the situation where the base graph has no completely covered edges called `is_Cover_noCCEdges` and another for the situation where the base graph has completely covered edges called `is_Cover` . These are the functions which takes the possible incidence matrices outputted from the `get_Possible_Cover_IM_noCCEdges` or `get_Possible_Cover_IM` functions respectively and checks whether they satisfy the conditions required to be an admissible cover.

The first function, `is_Cover_noCCEdges` , is a simplification of `is_Cover` and therefore we will only describe `is_Cover` . This function will take in five inputs. The first input is the two-dimensional list of integers outputted from `get_Possible_Cover_IM` called `PCIM` which represents the possible cover incidence matrix. The second and third inputs will be the Boolean list `CV` and `CE` , as before these will represent the information about the covered vertices and covered edges. The fourth input is `cc_edge_locations` ; this is the same integer list described in section 8.3. The final input is `cc_edges_isCrossed` ; this is the same Boolean list as described in section 8.3. The function will output a Boolean value which will be `True` if the possible cover incidence matrix represents an admissible cover or `False` if the incidence matrix represents a non-admissible cover.

```
def is_Cover(PCIM, CVerts, CEdges, cc_edge_locations, cc_edges_isCrossed):
  #create isCrossed
  isCrossed = [false]*configs["edges"]
  for e in range(configs["edges"]):
    if(e in ccl_edge_locations):
      if(ccl_edges_isCrossed[ccl_edge_locations.index(e)]):
        isCrossed[e]=True
  for e in range(configs["edges"]):
    start1=-1
    start2=-1
    end1=-1
    end2=-1
    if(not isCrossed[e] and e>=configs["loops"]):
```

```
    if(CEdges[e]):
      start1=edge_start(PCIM,2*e+1)
      end1=edge_end(PCIM,2*e+1)
      if(CVerts[edge_start(PCIM,2*e)/2]):
        start2=edge_start(PCIM,2*e)+1
      else:
        start2=edge_start(PCIM,2*e)
      if(CVerts[edge_end(PCIM,2*e)/2]):
        end2=edge_end(PCIM,2*e)+1
      else:
        end2=edge_end(PCIM,2*e)
      if(start1!=start2 or end1!=end2):
        return false
    else:
      if(CVerts[edge_start(PCIM,2*e)/2]):
        return false
      if(CVerts[edge_end(PCIM,2*e)/2]):
        return false
  #We have constructed the loops such that the will always satisfy the cover
  #conditions. Crossed edges will also satisfy the criteria by construction
  return true
```

If you recall from section 5 we need to check that the involution ι associated to a possible cover graph $\widetilde{\Gamma}$ is admissible. Therefore we need to check each non-crossed edge to see if $\delta \circ i_e(e_i) = i_v \circ \delta(e_i)$ (this implicitly includes loops because all loops are constructed to be crossed). We have already constructed the crossed edges to specifically satisfy this condition.

The first thing `is_Cover` does is construct a Boolean list called `isCrossed` with the information of which edges are crossed in the cover graph and which edges are not. This Boolean list may easily be confused with the Boolean list `cc_edges_isCrossed` . The difference between these to Boolean list is that `isCrossed` is a boolean list that tells whether each edge in the base graph lifts to a crossing – the length of this Boolean list is the number of edges in the base graph. The list `cc_edges_isCrossed` is a Boolean list which tells whether only the edges which are completely covered lift to crossings – the length of this list is the number of completely covered edges.

The purpose of the `isCrossed` list is to dictate to the function which edges need to be checked for admissibility and which edges do not. After creating `isCrossed` we will loop through the edges in the base graph. If the edge is not a loop and the edge does not lift to a crossing we need to check it for admissibility. The other cases have been constructed to be admissible.

The check for admissibility proceeds as follows. If an edge e is covered in the covering graph (i.e., the edge is fixed by the involution ι_E), then we need to consider $s\left(\tilde{e}^+\right)$ and $t\left(\tilde{e}^+\right)$. This is because we are verifying that $\delta(\iota_E(e)) = \iota_V(\delta(e))$ for any edge e. Suppose that e lifts to $\tilde{e}^-$ and $\tilde{e}^+$ – $\pi^{-1}(e) = \{\tilde{e}^-, \tilde{e}^+\}$ – then we need to check that if $s(e)$ is covered in $\widetilde{\Gamma}$ then $s(\tilde{e}^+) = \widetilde{s(e)}^+$, otherwise $s(\tilde{e}^+) = \widetilde{s(e)}$. We also need to check that if $t(e)$ is covered then $t(\tilde{e}^+) = \widetilde{t(e))}^+$, otherwise $t(\tilde{e}^+) = \widetilde{t(e))}$. If either of these two conditions are false we return that the possible cover graph is not admissible.

Example 8.4.1. Let e be an edge of the base graph Γ and suppose that e is covered with $s(e) = v_0$ covered and $t(e) = v_1$ not covered. The part of the cover graph $\widetilde{\Gamma}$ above e needs to look like the following.

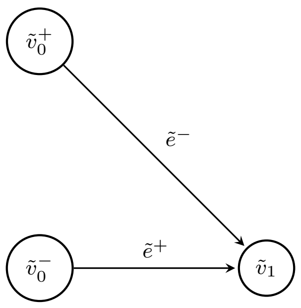

Figure 8.15: Only admissible cover of e in $\widetilde{\Gamma}$

In the case that the edge e is not covered, $\pi^{-1}(e) = \tilde{e}$, the program checks the covering information on $s(e)$ and $t(e)$. In this situation both $s(e)$ and $t(e)$ should be fixed by ι_V. If either $s(e)$ or $t(e)$ is not fixed by ι_V then the program returns that the possible cover graph is not admissible.

At this point we are able to generate all possible covers of a particular base graph and asses whether each of the possible covers represents an admissible $2:1$ cover of the base graph. The last part of this implementation we need to discuss is the isomorphism testing of the cover graphs. As discussed before, isomorphism testing is a runtime expensive process. It will be helpful to further select which specific admissible double covers we are interested in before isomorphism testing.

8.5 Friedman-Smith Testing

From [FS86] it is know that Friedman-Smith 2 and 3 degenerations will lead to members of the indeterminacy locus of the Prym period map under the perfect cone compactification. We would like to know if any higher order degenerations lead to members of the indeterminacy locus. Therefore, we need an effective way of determining when an admissible cover is a degeneration of a Friedman-Smith graph of order 4 or higher.

We define a class function `check_FS` of the `Cover Graph` class that will determine if each cover graph is a degree 4 or higher Friedman-Smith cover of a base graph. To check whether a cover graph is Friedman-Smith we consider the basis of linear forms calculated in subsection 6.3. We check the determinant of all maximal rank minors of the matrix of linear forms, if any of these maximal rank minors have a non-unit determinant we know that the cover graph is Friedman-Smith. The rank of the determinant tells us the degree of the Friedman-Smith cover.

```
def check_FS(self):
  FS = False
  rank = self.Basis_Linear_Forms.rank()
```

```
minors = self.Basis_Linear_Forms.minors(rank)
units = [0,−1,1] #units and zero of ZZ
for m in minors:
  if(m not in units):
    FS = true
    if(FS and rank>3):
      self.FS = True
    else:
      self.FS = False
```

8.6 Cover graph isomorphism testing and summary

The isomorphism testing of admissible covering graphs is a very important part of this program. Without testing for isomorphic cover graphs we would end up with more admissible cover graphs than we could computationally handle. Unfortunately checking for isomorphisms comes with a computational cost as well. Each cover graph has potentially twice the dimensions of the base graphs. The increase in dimension of graphs to be checked for isomorphism takes much more time for the compiler. As with the case of the base graphs we will be using the SAGE graph class `is_isomorphic` function which is a version of the Nauty graph isomorphism test.

We briefly mentioned at the beginning of section 8 that the entire program is split into two parts. One part generates the base graphs and the second part generates the covering information. We will now summarize the entire second part of the program which deals with getting the covering information.

(1) We start with all base graphs of fixed dimensions (loops, edges, vertices) and work with one base graph at a time (see section 7).

(2) For each particular base graph we generate all possible combinations of `CE` and `CV` (see subsection 8.1).

(3) For a given triple of a base graph E, a Boolean list of covered edges `CE` , and a Boolean list of covered vertices `CV` , we populate the list of completely covered edges called `cc_edges` using the `setNumberCCEdges` class function from the `EGraphs` class (section 8.2).

(4) Given the completely covered edges for the base graph E with specific choices of `CV` and `CE` we initialize a Boolean list called `cc_edges_isCrossed` . This Boolean list will dictate which completely covered edges of the base graph E lift to crossing (section 8.2).

(5) We will iterate though all possibilities of `cc_edges_isCrossed` making sure that `cc_edges_isCrossed` never has more crossings than the `crossingThreshold` (section 8.2).

(6) For a given base graph E, a Boolean list `CV` , a Boolean list `CE` , and a Boolean list `cc_edges_isCrossed` we generate a possible cover incidence matrix using the `get_Possible_Cover_IM` function (see section 8.3).

(7) We check the possible cover incidence matrix for admissibility using the function `is_Cover` (section 8.4).

(8) Finally, we first check if the cover graph is Friedman-Smith of order 4 or higher (section 8.5), is connected, and has a non-empty basis of linear forms. Next we check the Friedman-Smith admissible cover graphs over a particular base graph E for isomorphism. Unique members of isomorphism classes will have their basis of linear forms (see section 6) outputted to a text file which will be later checked for types of quadratic-cone-compactifications.

Remark 8.6.1. It should be noted that we are only comparing cover graphs over a specific base graph. This will result in generating some isomorphic cover graphs over different base graphs. The reason for not checking isomorphism over all base graphs is the computational cost would be too high.

```
with open(output_file, "w") as f:
```

```
with open(second_output, "w") as g:
  count = 0
  for E in All_EGraphs:
    All_CGS_For_G = []
    for vc in APVC:
      for ec in APEC:
        E.setNumberCCEdges(vc, ec)
        #There cannot be any loops in cover graph
        #All loops are already covered by construction of APEC.
        if (E.are_LVs_covered):
          if (E.numberCCEdges > 0):
            # Determine which edges are completely covered
            cc_edge_locations = []
            # Get locations of completely covered edges
            for e in range(configs["edges"]):
              if (E.ccEdges[e]):
                cc_edge_locations.append(e)
            cc_edges_iscrossed = [False] * E.numberCCEdges
            # make sure loops are always crossed
            for i in range(len(cc_edge_locations)):
              if (cc_edge_locations[i] < configs["loops"]):
                #loops come first in cc_edge_locations
                cc_edges_iscrossed[i] = True
            while (True):
              numberCrossedCCEdges = cc_edges_iscrossed.count(True)
              crossingThreshold = E.numberCCEdges+configs["loops"]-
                                    2*numberCrossedCCEdges
    if(crossingThreshold>=0):
      Possible_IM = get_Possible_Cover_IM(E, vc, ec, E.ccEdges,
                     cc_edges_iscrossed,cc_edge_locations)
      if (is_Cover(Possible_IM, vc, ec, cc_edge_locations,
          cc_edges_iscrossed)):
        CG = CoverGraph(Possible_IM, ec, vc, configs)
        if (CG.G.is_connected() and CG.FS and
            CG.Basis_Linear_Forms.nrows() > 0):
          is_new = True
            for cg in All_CGS_For_G:
              if (CG.G.is_isomorphic(cg.G)):
                is_new = False
                break
            if (is_new):
              f.write(str(CG.Basis_Linear_Forms))
              f.write("\n\n")
              g.write(CG.output())
    if (not iterate_Boolean_List(cc_edges_iscrossed)):
      break
# No CCEdges
else:
  Possible_IM = get_Possible_Cover_IM_no_CCEdges(E, vc, ec)
  if (is_Cover_no_CCEdges(Possible_IM, vc, ec)):
    CG = CoverGraph(Possible_IM, ec, vc, configs)
    if (CG.G.is_connected() and CG.FS and
        CG.Basis_Linear_Forms.nrows() > 0):
      is_new = True
```

```
        for cg in All_CGS_For_G:

          if (CG.G.is_isomorphic(cg.G)):

            is_new = False

            break

        if (is_new):

          f.write(str(CG.Basis_Linear_Forms))

          f.write("\n\n")

          g.write(CG.output())
```

Chapter 9

Example

Consider the following dual graph Γ with three vertices and five edges with one loop.

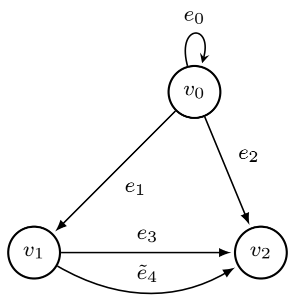

Figure 9.1: Base Graph Γ

We define `configs` to be the dictionary storing the values for the edges, vertices and loops. We initialize a memeber of the `EGraph` class called E by making a call to the `EGraph` constructor as follows,

$$E = \texttt{EGraph}(\Gamma, \texttt{IM}, \texttt{configs}).$$

Here `IM` will serve as the incidence matrix of Γ defined in 3.4. Let us pick the covered vertices array and covered edges array to be as follows,

$$\texttt{CV} = [\texttt{True}, \texttt{True}, \texttt{True}] \quad \text{and} \quad \texttt{CE} = [\texttt{True}, \texttt{True}, \texttt{True}, \texttt{True}, \texttt{True}].$$

Using `CV` and `CE` we call the `EGraph` class function `setNumberCCEdges` as follows,

$$E.\texttt{setNumberCCEdges}(\texttt{CV}, \texttt{CE}).$$

We notice that e_0 is a loop and there are four completely covered edges that are not loops: e_1, e_2, e_3, and e_4. This will set the E.`cc_Edges` array to be

$$\texttt{cc\_edges} = [\texttt{True}, \texttt{True}, \texttt{True}, \texttt{True}, \texttt{True}]$$

and therefore `cc_edge_locations` will be defined as,

$$\texttt{cc\_edge\_locations} = [0, 1, 2, 3, 4]$$

Let `cc_edges_isCrossed` be as follows

$$\texttt{cc\_edges\_isCrossed} = [\texttt{True}, \texttt{True}, \texttt{False}, \texttt{True}, \texttt{False}].$$

We pass in the parameters E, `CV` , `CE` , `cc_edges` , `cc_edge_locations,` and `cc_edges_isCrossed` into the function `get_Possible_Cover_IM` which will take the parameters and then return the following possible cover graph incidence matrix.

$$\texttt{Possible\_IM} = \begin{array}{c} \\ \tilde{v}_0^- \\ \tilde{v}_0^+ \\ \tilde{v}_1^- \\ \tilde{v}_1^+ \\ \tilde{v}_2^- \\ \tilde{v}_2^+ \end{array} \begin{array}{c} \begin{array}{cccccccccc} \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \tilde{e}_3^- & \tilde{e}_3^+ & \tilde{e}_4^- & \tilde{e}_4^+ \end{array} \\ \begin{bmatrix} -1 & 1 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & -1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 \end{bmatrix} \end{array}$$

Next we will pass in the parameters `Possible_IM` , `CV` , `CE` , `cc_edge_locations,` and `cc_edges_isCrossed` into the function `is_Cover` which will check whether given the following parameters `Possible_IM` is an admissible cover of Γ. Under these circumstances `is_Cover` will return `True` which signifies that `Possible_IM` is an admissible cover of Γ. Therefore we will create a member of the `CoverGraph` class with these specifications called `CG` which we will associate to $\widetilde{\Gamma}$. The cover dual graph $\widetilde{\Gamma}$ will be as follows,

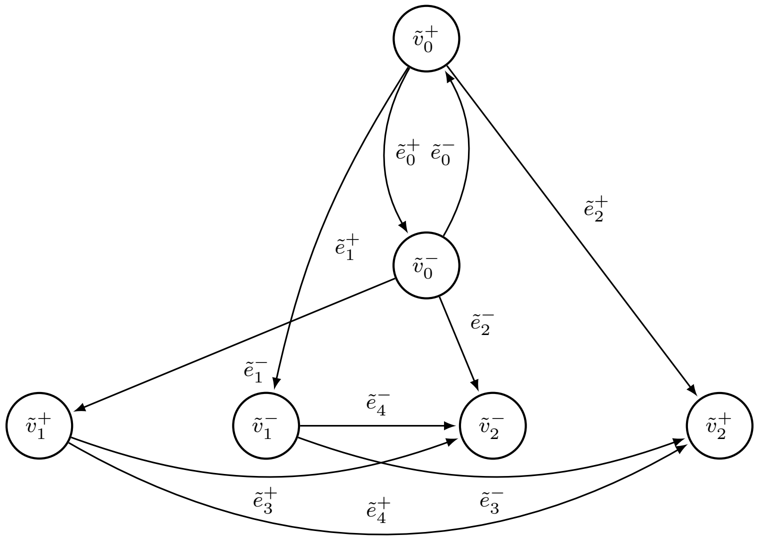

Figure 9.2: Admissible Cover Graph $\widetilde{\Gamma}$

Upon the creation of `CG` we will calculate the homology of $\widetilde{\Gamma}$ as follows,

$$H_1(\widetilde{\Gamma},\mathbb{Z}) = \begin{array}{c} \begin{array}{cccccccccc} \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \tilde{e}_3^- & \tilde{e}_3^+ & \tilde{e}_4^- & \tilde{e}_4^+ \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & -1 & 1 & 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 0 & 1 & -1 & 0 & -1 & 0 & 1 \\ 0 & 0 & 1 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & -1 & -1 \end{bmatrix} \end{array}$$

Next we use `CE` to construct $(\mathrm{id} - \iota)$ as follows,

$$(\mathrm{id}-\iota) = \begin{array}{c} \begin{array}{cccccccccc} \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \tilde{e}_3^- & \tilde{e}_3^+ & \tilde{e}_4^- & \tilde{e}_4^+ \end{array} \\ \begin{bmatrix} 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix} \end{array}$$

we will the perform the matrix multiplication $H_1(\widetilde{\Gamma}, \mathbb{Z}) \cdot (\mathrm{id} - \iota)$ to get a generating set for $2 \cdot H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$. Then we will row reduce over $\mathbb{Z}$ and delete zero rows to get a basis of $2 \cdot H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$. Typically one multiplies by $\frac{1}{2}(\mathrm{id}-\iota)$ but to avoid dealing with fractions we work with $2 \cdot H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$.

$$2 \cdot H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} = \begin{array}{c} \begin{array}{cccccccccc} \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \tilde{e}_3^- & \tilde{e}_3^+ & \tilde{e}_4^- & \tilde{e}_4^+ \end{array} \\ \begin{bmatrix} 1 & -1 & 0 & 0 & -2 & 2 & -1 & 1 & 1 & -1 \\ 0 & 0 & 1 & -1 & -1 & 1 & -1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

Finally we will compute a basis of linear forms on $\widetilde{\Gamma}$. To do this we evaluate $\tilde{e}^{\vee} - \iota\tilde{e}^{\vee}$ on the basis vectors of $2 \cdot H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$. Let us call this basis of linear forms Q.

$$Q^T = \begin{array}{cccccccccc} \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \tilde{e}_3^- & \tilde{e}_3^+ & \tilde{e}_4^- & \tilde{e}_4^+ \\ \left[\begin{array}{c} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{array} \right. & \begin{array}{c} -1 \\ 0 \\ 0 \\ 0 \\ 0 \end{array} & \begin{array}{c} 0 \\ 1 \\ 0 \\ 0 \\ 0 \end{array} & \begin{array}{c} 0 \\ -1 \\ 0 \\ 0 \\ 0 \end{array} & \begin{array}{c} 0 \\ 0 \\ 1 \\ 0 \\ 0 \end{array} & \begin{array}{c} 0 \\ 0 \\ -1 \\ 0 \\ 0 \end{array} & \begin{array}{c} 0 \\ 0 \\ 0 \\ 1 \\ 0 \end{array} & \begin{array}{c} 0 \\ 0 \\ 0 \\ -1 \\ 0 \end{array} & \begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{array} & \left. \begin{array}{c} 0 \\ 0 \\ 0 \\ 0 \\ -1 \end{array} \right] \end{array} \cdot \begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \\ -2 & -1 \\ 2 & 1 \\ -1 & -1 \\ 1 & 1 \\ 1 & 0 \\ -1 & 0 \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \\ -4 & -2 \\ -2 & -2 \\ 2 & 0 \end{bmatrix}$$

We will divide every entry in each row by two to accommodate for working with $2 \cdot H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$ and therefore we find the basis of linear forms Q as follows,

$$Q = \begin{array}{ccccc} \ell_{e_0} & \ell_{e_1} & \ell_{e_2} & \ell_{e_3} & \ell_{e_4} \\ \left[\begin{array}{c} 1 \\ 0 \end{array} \right. & \begin{array}{c} 0 \\ 1 \end{array} & \begin{array}{c} -2 \\ -1 \end{array} & \begin{array}{c} -1 \\ -1 \end{array} & \left. \begin{array}{c} 1 \\ 0 \end{array} \right] \end{array}$$

This program will output all of the bases of linear forms. From the basis of linear forms Q we may square the basis vectors to obtain a basis of quadratic forms. This will give us the monodromy cones for each cover. From the monodromy cones we then use another program to determine if the specific cover lies in the indeterminacy locus.

It should be noted that before we output the basis of linear forms we must check if $\widetilde{\Gamma}$ is a Friedman-Smith degeneration. To do this we may check the determinants of all the maximally ranked submatrices of Q. If any determinate is not a unit in $\mathbb{Z}$ or 0 then we know that $\widetilde{\Gamma}$ is not Friedman-Smith. The maximal rank will be the order of the potential Friedman-Smith degeneration.

In this case we see that the maximal rank of Q is 2 and the submatrix

$$\begin{bmatrix} 0 & -2 \\ 1 & -1 \end{bmatrix}$$

will have determinant 2 which implies that $\widetilde{\Gamma}$ is not a Friedman-Smith degeneration.

Chapter 10

A note on the maximum number of edges and vertices on the dual graph of a stable curve of a given genus

Lemma 10.0.1. *Let C be a stable curve of genus $g \geq 2$. Then the dual graph of C can have at most $2(g-1)$ vertices and $3(g-1)$ edges. Moreover, for every genus $g \geq 2$, there exists a stable curve C of genus g with dual graph having $2(g-1)$ vertices and $3(g-1)$ edges.*

Proof. Let Γ be the dual graph of a curve C. We start with the genus formula. Let $e = \#E(\Gamma)$, $v = \#V(\Gamma)$, and $g(v)$ be the genus of the normalization of the component of C corresponding to $v \in V(\Gamma)$. Then

$$g = e - v + 1 + \sum_{v \in V(\Gamma)} g(v).$$

If $g(v) > 0$, we can always degenerate the component of C corresponding to v to a rational nodal curve. This increases the number of edges of Γ, and preserves the number of vertices. Thus, to find a curve Γ that has the maximal number of edges or vertices, we can assume that all of the components are rational nodal curves. In particular, we have

$$g = e - v + 1.$$

Now since every component of C is rational, the dual graph must be at least trivalent at each vertex. If we cut each edge of the dual graph in half, and denote h to be the number of half-edges (i.e., $h = 2e$), then the valency condition implies that $h \geq 3v$. Thus $e \geq 3v/2$. This gives

$$g = e - v + 1 \geq \frac{3v}{2} - v + 1 = \frac{v}{2} + 1.$$

In other words,

$$v \leq 2(g-1).$$

Since we have $g = e - v + 1$, this gives

$$e = (g-1) + v \leq 3(g-1).$$

Note that this bound on the number of edges also follows from the fact that C is contained in a codimension e stratum in $\overline{M}_g$.

Finally we just need to construct a stable curve of genus g with this number of components and nodes. We do this as follows (I will call this the **It's-It curve**). The curve C has $2(g-1)$ components, which are all smooth rational curves. The curve has $3(g-1)$ nodes. The dual graph looks as follows. Draw a cylinder with circular base. Put $(g-1)$ vertices on the top circle, and $(g-1)$ vertices on the bottom circle. Then connect vertex 1 on the top circle with vertex 1 on the bottom circle, vertex 2 on the top circle with vertex 2 on the bottom circle, and so on. The end result looks something like an ice cream sandwich. □

Remark 10.0.2. Recall that in regards to the indeterminacy locus of the Prym map ([CMGHL17b, Thm. 7.1]), the first open case is to determine the indeterminacy of the Prym map $P_5^P : \overline{R}_5 \dashrightarrow \bar{A}_4^P$. Thus, one must consider the case where there are up to 8 vertices and up to 12 edges in the base graph.

Remark 10.0.3. In another direction, in any genus, one can try to compute up to a given codimension. This means fixing the number of edges e. Since the graphs are connected, we must have that $v \leq e + 1$. Thus, in light of [CMGHL17b, Thm. 7.1], the first open case is 7 edges and 8 vertices.

10.1 Counting vertices and edges for dual graphs in $\overline{M}_{g,n}$

For later reference, we also have:

Lemma 10.1.1. *Let $(C, p_1, \dots, p_n)$ be a stable curve of genus g with n marked points. Then the dual graph of C can have at most $2(g-1)+n$ vertices and $3(g-1)+n$ edges.*

Moreover, for every genus $g \geq 2$, and each $n \geq 0$, (or for $g = 1$ and each $n \geq 1$) there exists a stable curve C of genus g with n marked points with dual graph having $2(g-1)+n$ vertices and $3(g-1)+n$ edges.

For $g = 0$, and each $n \geq 3$, there can be at most $n-2$ vertices and $n-3$ edges. Moreover, there exists a stable curve C of genus 0 with n marked points with dual graph having $n-2$ vertices and $n-3$ edges.

Proof. Let Γ be the dual graph of a stable curve C with n marked points. We start with the genus formula. Let $e = \#E(\Gamma)$, $v = \#V(\Gamma)$, and $g(v)$ be the genus of the normalization of the component of C corresponding to $v \in V(\Gamma)$. Then

$$g = e - v + 1 + \sum_{v \in V(\Gamma)} g(v).$$

If $g(v) > 0$, we can always degenerate the component of C corresponding to v to a rational nodal curve. This increases the number of edges of Γ, and preserves the number of vertices. Thus, to find a curve so that Γ has the maximal number of edges or vertices, we can assume that all of the components are rational nodal curves. In particular, we have

$$g = e - v + 1.$$

Now since every component of C is rational, the dual graph must be at least trivalent at each vertex. If we cut each edge of the dual graph in half, and denote h to be the number of half-edges (i.e., $h = 2e + n$), then the valency condition implies that $h \geq 3v$. Thus $e \geq (3v - n)/2$. This gives

$$g = e - v + 1 \geq \frac{3v - n}{2} - v + 1 = \frac{v - n}{2} + 1.$$

In other words,

$$v \leq 2(g-1) + n$$

Since we have $g = e - v + 1$, this gives

$$e = (g-1) + v \leq 3(g-1) + n.$$

Note that this bound on the number of edges also follows from the fact that C is contained in a codimension e stratum in $\overline{M}_{g,n}$.

For $\overline{M}_{0,n}$ we simply use that the dimension is $n-3$, and so there can be at most $n-3$ edges. The fact that the first Betti number of the graph is 0, and it is connected, implies that it is a tree, so that it has $n-2$ vertices (if it has $n-3$ edges).

Finally we just need to construct a stable curve of genus g and n marked points, with this number of components and nodes. We do this as follows. For $g \geq 2$, the curve C has $2(g-1)+n$ components, which are all smooth rational curves. The curve has $3(g-1)+n$ nodes. The dual graph looks as follows. Draw a cylinder with circular base. Put $(g-1)$ vertices on the top circle, and $(g-1)$ vertices on the bottom circle. Then connect vertex 1 on the top circle with vertex 1 on the bottom circle, vertex 2 on the top circle with vertex 2 on the bottom circle, and so on. The end result looks something like an ice cream sandwich. Now along one edge in the top circle, insert n vertices. This increases the number of edges by n, as well. Now to make this curve stable, add n half edges (marked points) to these new vertices.

In the case $g = 1$, simply take a circle of n genus 0 vertices (and n edges) and then add n half edges to make it stable. In the case $g = 0$ consider the connected chain graph with $n-2$ genus 0 vertices and $n-3$ edges. Add $n-2$ half edges, one for each vertex, and then add one more half edge to the first and last vertices in the chain. □

Chapter 11

Another approach to enumerating degenerations of Friedman–Smith covers

The main obstacle to obtaining new results computationally, as described in the last section, is that there are too many graphs to check. Thus we need to focus our attention. The main point is that the indeterminacy locus of the rational Prym map ([CMGHL17b, Thm. 7.1])

$$P_g^P : \overline{R}_g \dashrightarrow \bar{A}_{g-1}^P$$

is understood, except for degenerations of Friedman–Smith covers, in $\partial\overline{FS}_4, \ldots, \partial\overline{FS}_g$. Thus we should focus on enumerating just these covers, not all admissible covers. This process would entail enumerating base graphs of lower dimensions and constructing Friedman–Smith graphs by attaching n edges between two lower dimensional base graphs. This would fix some of the covering information. For example, each of the n edges between the two lower dimensional base graphs would have to be unramified in the covering graph. We did not take this approach because the best way to enumerate FS degenerations would be to do it recursively. This would be the next thing to try. Using the a modification of the recursive base graph generation code this is a feasible approach to the problem of enumerating Friedman-Smith degenerations. This technique has the potential to compute up to the 12-edge case thus completing the $g = 5$ indeterminacy locus.

Let us recall the dual graphs of Friedman–Smith covers:

If C_1 is the smooth curve corresponding to v_1 and C_2 is the smooth curve corresponding to v_2, then the pairs of genera $(g(C_1), g(C_2))$ given by

$$(1, g-n+1), (2, g-n), \ldots (\lfloor \frac{g-n+2}{2} \rfloor, \lfloor \frac{g-n+3}{2} \rfloor).$$

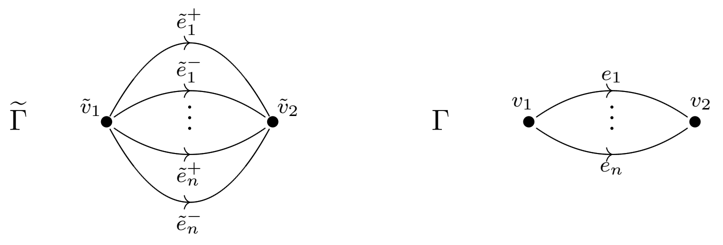

Figure 11.1: Dual graph of a Friedman–Smith example with $2n \geq 2$ nodes (FS_n).

In particular $FS_n = \emptyset$ in $\overline{\mathcal{R}}_{g+1}$ if $n \geq g+1$. Note also that the covers $\widetilde{C}_i \to C_i$ are étale, so that in particular, the curves $\widetilde{C}_i$ have odd genus $2g(C_i) - 1$.

The graphs we are interested in, namely dual graphs of degenerations of Friedman–Smith covers, are those admissible cover graphs that can be obtained from the graphs above by replacing the vertices with other graphs.

11.1 Enumeration in the special case $g = 5$

As mentioned above, when considering the map

$$P_5^P : \overline{R}_5 \dashrightarrow \bar{A}_4^P$$

which is the first open case, we only need to consider the cones of quadratic forms for graphs coming from $\partial\overline{FS}_4$. The only possible genera for the base curves associated to the vertices v_1 and v_2 are

$$(1, 1).$$

Thus, to parameterize the base curves C, we just need to enumerate all dual graphs in $\overline{M}_{1,4}$, and then consider all ways of inserting those into the Friedman–Smith base graphs above. The computation above says that each dual graph in $\overline{M}_{1,4}$ has at most 4 vertices and 4 edges (which seems within the realm of computable).

After enumerating all dual graphs in $\overline{M}_{1,4}$ we can then consider all admissible covers of those base graphs, (maybe with the given covering edges interchanged as given, but otherwise arbitrary covers). We can also ignore all loops in the cover graph $\widetilde{\Gamma}$ using the results of section 5.5.

Chapter 12

Results

12.1 Table of computation results

We will will start by giving the number of resulting Friedman–Smith degenerations of order four or higher in each dimension. The readers should recall that for covering graphs, we are not doing a complete isomorphism testing in each case. We are only isomorphism testing the covering graphs over a fixed base graphs. Therefore some covers over different base graphs may be isomorphic resulting in slightly inflated numbers.

Edges	Vertices	Loops	Number of Base Graphs	Number of $\overline{FS}_n$ Graphs ($n > 3$)
4	2	0	1	1
4	2	1	1	0
4	2	2	2	0
4	2	3	2	0
4	3	0	3	3
4	3	1	4	0
4	3	2	4	0
4	4	0	5	5
4	4	1	4	0
5	2	0	1	1
5	2	1	1	1

5	2	2	2	1
5	2	3	2	1
5	2	4	3	0
5	3	0	4	20
5	3	1	7	7
5	3	2	8	4
5	3	3	6	0
5	4	0	11	67
5	4	1	13	13
5	4	2	10	0
5	5	0	11	76
5	5	1	9	0
6	2	0	1	1
6	2	1	1	1
6	2	2	2	4
6	2	3	2	3
6	2	4	3	5
6	2	5	3	0
6	3	0	6	45
6	3	1	10	42
6	3	2	14	35
6	3	3	13	21
6	3	4	9	0
6	4	0	22	437
6	4	1	32	175
6	4	2	33	82
6	4	3	17	0

6	5	0	34	1023
6	5	1	41	248
6	5	2	24	0
6	6	0	29	936
6	6	1	20	0
7	2	0	1	1
7	2	1	1	1
7	2	2	2	4
7	2	3	2	4
7	2	4	3	5
7	2	5	3	5
7	2	6	4	0
7	3	0	7	69
7	3	1	14	91
7	3	2	20	105
7	3	3	22	84
7	3	4	19	50
7	3	5	12	0
7	4	0	37	1298
7	4	1	66	932
7	4	2	81	603
7	4	3	60	249
7	4	4	30	0
7	5	0	85	6816
7	5	1	136	2810
7	5	2	116	984
7	5	3	50	0

7	6	0	110	13672
7	6	1	125	3158
7	6	2	63	0
7	7	0	70	9140
7	7	1	48	0
8	2	0	1	1
8	2	1	1	1
8	2	2	2	5
8	2	3	2	4
8	2	4	3	7
8	2	5	3	5
8	2	6	4	7
8	2	7	4	0
8	3	0	9	103
8	3	1	18	146
8	3	2	28	186
8	3	3	32	163
8	3	4	33	122
8	3	5	26	66
8	3	6	16	0
8	4	0	61	2960
8	4	1	119	2594
8	4	2	165	2138
8	4	3	150	1225
8	4	4	106	487
8	4	5	44	0
8	5	0	193	26169

8	5	1	361	15364
8	5	2	390	7821
8	5	3	255	2438
8	5	4	96	0
8	6	0	348	99454
8	6	1	526	36834
8	6	2	408	10048
8	6	3	146	0
8	7	0	339	136618
8	7	1	378	30443
8	7	2	164	0
8	8	0	185	80801
8	8	1	115	0
9	2	0	1	1
9	2	1	1	1
9	2	2	2	5
9	2	3	2	5
9	2	4	3	7
9	2	5	3	7
9	2	6	4	7
9	2	7	4	7
9	2	8	5	0
9	3	0	11	142
9	3	1	23	225
9	3	2	36	297
9	3	3	45	289
9	3	4	48	237

9	3	5	45	160
9	3	6	35	87
9	3	7	20	0
9	4	0	95	5971
9	4	1	201	5725
9	4	2	299	5164
9	4	3	311	3476
9	4	4	264	1883
9	4	5	162	683
9	4	6	67	0
9	5	0	396	110985
9	5	1	841	70262
9	5	2	1044	40916
9	5	3	867	17303
9	5	4	497	4757
9	5	5	164	0
9	6	0	969	362476
9	6	1	1763	178455
9	6	2	1746	77298
9	6	3	1010	20331
9	6	4	315	0
9	7	0	1318	1026278
9	7	1	1961	359214
9	7	2	1372	85551
9	7	3	437	0
9	8	0	1067	1302916
9	8	1	1132	269473

9	8	2	444	0
9	9	0	479	664102
9	9	1	286	0

12.2 Interpretation of computations

Let $\overline{A}_g^P$ be the perfect cone compactification of the moduli space of principally polarized abelian varieties of dimension g. Let $\overline{R}_{g+1}$ be the normal crossings compactification of the moduli space of connected étale double covers of curves of genus $g+1$. Then the Prym period map $P_g^P : R_{g+1} \to A_g$ does not extend to a regular map $\overline{R}_{g+1} \to \overline{A}_g^P$. In [Thm. 5.6][CMGHL17b] the authors give a classification of when the Prym period map extend to a regular map in a neighborhood of a curve. Friedman and Smith discovered a class of admissible covers in [FS86] for which P_g^P does not extend. The main result being

$$\overline{FS_2} \cup \overline{FS_3} \subseteq \text{Ind}(P_g^P).$$

In [Thm. 0.1][Vol02], in the case of the second Voronoi compactification of A_g, being in the indeterminacy locus of the Prym map is equivalent to being a Friedman–Smith degeneration with at least 4 nodes. That is, if $P_g^V : \overline{R}_{g+1} \to \overline{A}_g^V$ is the extension of the Prym period map in the case of the second Voronoi compactification then

$$\text{Ind}(P_g^V) = \bigcup_{n\geq 2} \overline{FS_n}.$$

When considering the indeterminacy locus of P_g^P, the extension over the perfect cone compactification of A_g, the indeterminacy locus will be smaller. In [Thm. 7.1][CMGHL17b] it was found that

$$\overline{FS_2} \cup \overline{FS_3} \subseteq \text{Ind}(P_g^P) \subseteq \overline{FS_2} \cup \overline{FS_3} \cup \delta\overline{FS_4} \cup \cdots \cup \delta\overline{FS_g}$$

where $\delta\overline{FS_n} = \overline{FS_n} - FS_n$. Moreover,

$$\text{codim}_{\overline{R}_{g+1}} \text{Ind}(P_g^P) \smallsetminus \left(\overline{FS_2} \cup \overline{FS_3}\right) \geq 6.$$

By this result we have that if $g = 1$ we have that P_1^P extends to a morphism. If $g = 2$, $\mathrm{Ind}(P_2^P) = \overline{FS_2}$ and if $g = 3$, $\mathrm{Ind}(P_3^P) = \overline{FS_2} \cup \overline{FS_3}$. Also in [CMGHL17b] it is shown that for $g = 4$, $\mathrm{Ind}(P_4^P) = \overline{FS_2} \cup \overline{FS_3}$. This poses a question.

Question 12.2.1. For $g \geq 5$, is $\mathrm{Ind}(P_g^P) = \overline{FS_2} \cup \overline{FS_3}$?

In this thesis, through methods of sections 7 and 8 we were able to enumerate all Friedman–Smith degenerations of order 4 or higher with at most 9 edges in the base graph. This led to the following result.

Theorem 12.2.2. *If $\widetilde{C} \to C$ is an étale double cover of C such that $\widetilde{C}/C \in \overline{FS_n}$ for $n \geq 4$, the dual graph $\Gamma(C)$ of C has at most 9 edges, and $\widetilde{C}/C$ not in $\overline{FS_2}$ or $\overline{FS_3}$ then $\widetilde{C}/C$ is not in (Ind)(P_g^P).*

Corollary 12.2.3. *Define a subset of étale double covers as follows,*

$$Z_{10} = \left\{ \widetilde{C}/C \colon |E(\Gamma(C))| \geq 10 \right\}.$$

Here Z_{10} is a closed subset of $\overline{R}_6$ having codimension 10. Let $U_{10} = \overline{R}_6 \setminus Z_{10}$. Let $P_5^P : \overline{R}_6 \to \overline{A}_5$ be the Prym period map. Then P_5^P is regular on $U \setminus (\overline{FS_2} \cup \overline{FS_3})$. That is,

$$Ind\left(P_5^P|_{U_{10}} \right) = \overline{FS_2} \cup \overline{FS_3}.$$

This does not answer question 12.2.1. In order to completely classify the indeterminacy locus of the Prym period map in the case of $g = 5$ we need to allow for all dual graphs with up to 12 edges. Therefore this does not fully answer the question but it does give more insight on the description of the indeterminacy locus.

Chapter 13

Degeneration of cubic threefolds

13.1 Introduction

A cubic threefold is a smooth hypersurface of degree three in $\mathbb{P}^4$. Cubic threefolds have provided a lot of interesting results in algebraic geometry. For example, Lüroth's Theorem states that every unirational curve is rational and Castelnuovo's Theorem states every unirational surface is rational but a result from Clemens and Griffiths ([CG72]) shows that while cubic threefolds are unirational, they are not rational. Their proof uses the fact that the intermediate Jacobian of a cubic threefold is not the Jacobian of a curve. Here we will study degenerations of intermediate Jacobians of cubic threefolds, as the cubic threefolds degenerate to a singular cubic hypersurface.

In [Mum74], Mumford showed that the intermediate Jacobians of cubic threefolds are Prym varieties of connected étale double covers of plane quintics. Therefore singular cubic threefolds can be studied using the degeneration of intermediate Jacobians which can be computed as degenerations of Prym varieties. In this thesis we will be computing the degeneration data associated to a cubic threefold with two A_1-singularities. The interest in this special case is that while the degeneration data for this case was computed in [CMGHL17a] via theta functions, and partial degeneration data was computed via Pryms in [Hav16], here we would like to describe the degeneration more completely using Pryms. In principle, from this case, one can compute the degeneration data via Pryms for all degenerations to cubics with isolated AD singularities.

13.2 Intermediate jacobian of singular cubic threefolds

Let X be a cubic hypersurface in $\mathbb{P}^4$ containing a double point at P. Let $\pi : \mathbb{P}^4 \to \mathbb{P}^3$ be the projection through the point P. Define C to be the image of all the lines in X that pass through P. There is a well-know result about the image C in relation to X, see [Hav16, CMJL12] for the details and the proof.

Theorem 13.2.1. *If X is a cubic threefold with isolated singularities then C is a complete intersection curve of type $(2,3)$ such that*

$$Bl_C\mathbb{P}^3 \cong Bl_P X.$$

The previous theorem gives us a way of associating a curve C in $\mathbb{P}^3$ to a cubic threefold X with a chosen double point. We call this curve C the $(2,3)$-curve which is a complete intersection of a quadric and a cubic. The next result explains the relationship between the singularities of X and the singularities of C.

Theorem 13.2.2 ([CMJL12], Prop. 1.3)**.** *If X has isolated singularities only, the singularities of $Bl_P X$ are in bijection with the singularities of C. Furthermore, the singularites on $Bl_P X$ and C have the same singularity types.*

We will follow the work of Clemens and Griffiths, in [CG72], to characterize the intermediate Jacobian of the the cubic threefold X with a unique A_1 or A_2 singularity at $P \in X$. Let X be a cubic threefold with a single A_1 or A_2 singularity at P. By theorem 13.2.1, we know that $\mathrm{Bl}_P X \cong \mathrm{Bl}_C \mathbb{P}^3$ where C is as described above. By blowing up X at P we are resolving the A_1 or A_2 singularity. By theorem 13.2.2 this suggest that both $\mathrm{Bl}_P X$ and C are non-singular. The Hodge diamond of $\widetilde{X} = \mathrm{Bl}_P X \cong \mathrm{Bl}_C \mathbb{P}^3$ was computed in detail in [§2.3][Hav16] and the result is below.

$$\begin{array}{ccccccc}
 & & & \mathbb{C} & & & \\
 & & 0 & & 0 & & \\
 & 0 & & \mathbb{C}^2 & & 0 & \\
0 & & H^{1,0}(C) & & H^{0,1}(C) & & 0 \\
 & 0 & & \mathbb{C}^2 & & 0 & \\
 & & 0 & & 0 & & \\
 & & & \mathbb{C} & & &
\end{array}$$

From this we see that $H^{1,2}(\widetilde{X}) \oplus H^{0,3}(\widetilde{X}) \cong H^{0,1}(C)$, and $H^3(\widetilde{X}, \mathbb{C}) \cong H^1(C, \mathbb{C})$ which proves the following theorem.

Theorem 13.2.3 ([CG72]). *The intermediate Jacobian of the desingularization $\widetilde{X} = Bl_P X$ of a cubic threefold X with a single A_1 or A_2 singularity at P is isomorphic to the jacobian JC of the $(2,3)$-curve C.*

13.3 Construction of the plane quintic

We will define the discriminant curve associated to X following Mumford (see [CG72], [§3.2][Hav16]). Let $\ell \subset X$ be a general line in X not containing any singular points. We can parametrize all the two dimensional planes containing ℓ in $\mathbb{P}^4$ by the two dimensional projective plane $\mathbb{P}^2$. Define $\Pi = \mathbb{P}^2$ to be the parameterizing space. That is $V \in \Pi$ represents a plane in $\mathbb{P}^4$ containing ℓ. Consider the intersection $X \cap V$ in X; X is a cubic hypersurface in $\mathbb{P}^4$ and therefore this intersection has to have degree 3. We know the intersection contains the line ℓ and therefore $X \cap V$ is either the union of ℓ and a conic or the union of ℓ and two other lines. Denote $D \subset \Pi$ as the set of $V \in \Pi$ such that $X \cap V$ is the union of 3 lines. Then D is a quintic in the plane Π (see [§3.2][CMGHL17a]) with singularities in bijection to the singularities of X respecting singularity type. Let $R = \langle \ell, Q \rangle$ be the plane spanning ℓ and a line through Q, then viewing R as a point of D in Π, it has the same singularity type as Q in X.

Define $\widetilde{D}$ to be the curve in the Fano scheme of lines of X whose points represent lines of X

intersecting ℓ but $\ell \notin \widetilde{D}$. For $\ell_1 \in \widetilde{D}$ we have that $V' = \langle \ell, \ell_1 \rangle \in D \subset \Pi$. That is $V' \cap X = \{\ell, \ell_1, \ell_2\}$. Define a map $\pi : \widetilde{D} \to D$ where

$$\ell_1 \mapsto V' = \langle \ell, \ell_1 \rangle .$$

This is a double cover of D because $\pi^{-1}(V') = \{\ell_1, \ell_2\}$. If we normalize D and $\widetilde{D}$ we get the following commutative diagram

$$\begin{tikzcd}
N\widetilde{D} \arrow[r, "\hat{\pi}"] \arrow[d] & ND \arrow[d] \\
\widetilde{D} \arrow[r, "\pi"] & D
\end{tikzcd}$$

where $\hat{\pi}$ is an étale double cover of ND. For this thesis we will assume that D is irreducible and thus ND is connected.

Propostion 13.3.1 ([Hav16], Prop. 3.2.2)**.** For a cubic threefold X with an A_n singularity and an irreducible discriminant D, the curve ND is trigonal.

Using the previous proposition we may apply Recillas' theorem ([Rec74]) which states that the Prym variety associated to an étale double cover of a trigonal and non-hyperelliptic curve is the Jacobian of a tetragonal curve. We will construct the tetragonal curve.

Consider the composition $N\widetilde{D} \to ND \to \mathbb{P}^1$ where the preimage of a point in $\mathbb{P}^1$ is 6 points in $N\widetilde{D}$ because $ND \to \mathbb{P}^1$ is degree 3 and $N\widetilde{D} \to ND$ is degree 2. Let $p \in \mathbb{P}^1$ then above p we have $\{p_1, p_2, p_3\}$ in ND. Above p_i we have $p_i^{\pm}$ in $N\widetilde{D}$. Therefore a point in the tetragonal curve will be a triple of points in $N\widetilde{D}$ such that the 3 points are mapped to different points of ND but then mapped to the same point of $\mathbb{P}^1$. The triple $(p_1^{\pm}, p_2^{\pm}, p_3^{\pm})$ is a point on the tetragonal curve. There are 8 choices of points above $p \in \mathbb{P}^1$. The curve we have constructed will have 2 identical components and by picking one of the components we have the desired tetragonal curve.

Theorem 13.3.2 ([Hav16], Thm. 3.2.5)**.** *For a cubic threefold X having an A_n singularity, an irreducible discriminant D, and a non-hyperelliptic ND, the tetragonal curve constructed above is* *isomorphic to the normalization NC where C is the (2,3)-curve obtained from projection through an A_n singularity.*

The proof of theorem 13.3.2 is provided in detail in [Hav16]. We will need some of the details of this proof and thus we will provide a sketch of the proof.

A point $m \in C$ is a line on X going through the A_n singularity at $Q \in X$. The two lines m and ℓ – the line used for the projection to obtain D – will span a 3-dimensional space W in $\mathbb{P}^4$. The set of 2-dimensional subspaces in W which contain ℓ will form a line $w \subset \Pi$.

Consider the intersection $Y := X \cap W$, Y is a cubic surface with a singularity at Q. For a general choice of W the singularity of Y at Q is of type A_1. A cubic surface with an A_1 singularity contains 21 lines (an exposition of this along with the classification of other singular cubic surfaces is provide in Cayley's Memoir on Cubic Surfaces [Cay69]). There are six lines $\{\ell_1, \ldots, \ell_6\}$ which pass through the singular point Q on Y. The lines ℓ_i an ℓ_j, $1 \leq i < j \leq 6$, determines a plane which cuts out a third line on Y denoted ℓ_{ij}. The lines ℓ_{ij} account for the other 15 lines. The line ℓ_i intersects ℓ_{rs} if $i = r$ or $i = s$. The line ℓ_{ij} intersects ℓ_{rs} if i, j, r, and s are all disjoint.

We have already identified 2 lines on the cubic surface, ℓ does not meet at the singularity Q and m does. Denote $\ell = \ell_{12}$ and $m = \ell_6$. The line $w \subset \Pi$ intersects the plane quintic D at R and three additional points because $\deg(D) = 5$. The three additional points represent 2-dimensional subspaces of W which contain L and intersect Y at 3 lines. The point $R \in D$ corresponds to the plane which is the span of Q and ℓ_{12}, this plane intersects Y at ℓ_1, ℓ_2 and ℓ_{12}. The three other intersection points will correspond to three other triples of lines in Y, each containing ℓ_{12}.

$$(\ell_{12}, \ell_{34}, \ell_{56})$$

$$(\ell_{12}, \ell_{35}, \ell_{46})$$

$$(\ell_{12}, \ell_{36}, \ell_{45})$$

In D the lines $\{\ell_{12}, \ell_{34}, \ell_{56}\}$ all represent a plane intersecting X in three lines and so they correspond to a point $V \in D$. In the double cover $\pi : \widetilde{D} \to D$ we have $\pi^{-1}(V) = \{\ell_{34}, \ell_{56}\}$. In summary, picking a point on the tetragonal curve is equivalent to picking a line ℓ_{ij} from each of the 3 triples not equal to ℓ_{12}. We can do this by picking the line in the triple that intersects ℓ_6. This will give a map from an open subset of C to the tetragonal curve which is injective and thus birational. This will give a birational map from NC to the tetragonal curve and because both curves are smooth this will be an isomorphism.

13.4 Degenerations of Jacobians

The theorems 13.2.3 and 13.3.2 have shown that taking limits of 1-parameter degenerations of cubic threefolds is equivalent to taking degenerations of 1-parameter Prym variety of unramified double covers of plane quintics. Recillas' theorem gives a correspondence to these Prym varieties and Jacobians of tetragonal curves. Therefore it will be useful to construct the jacobian of nodal curves.

Given a family of curves $\mathcal{X} \to \Delta$ over the unit disk the Jacobian of the generic fiber C_0 can be described as the limit of Jacobians of smooth curves. We give this Jacobian as an extension,

$$0 \to H^1(\Gamma, \mathbb{Z})^{\otimes}\mathbb{C}^* \to JC_0 \to J(NC) \to 0$$

where Γ is the dual graph of C_0 and NC_0 is the normalization of C_0. For Jacobians the limit JC_0 only depends on the curve C_0. In [§3.1][ABH02], the authors give the data required to classify JC_0.

(J0) The abelian variety JN_0, which is called the compact part, where N_0 is the normalization of the curve C_0.

(J1) The lattice $H_1(\Gamma, \mathbb{Z})$ where Γ is the dual graph of C_0 and the semi-abelian variety which is the extension given by the sequence,

$$0 \to H^1(\Gamma, \mathbb{Z}) \otimes \mathbb{C}^* \to JC_0 \to JN_0 \to 0.$$

The torus $H^1(\Gamma, \mathbb{Z}) \otimes \mathbb{C}^* \cong (\mathbb{C}^*)^n$, n is the rank of $H^1(\Gamma, \mathbb{Z})$, is called the non-compact part.

(J2) A class in

$$\mathrm{Ext}^1(JN_0, H^1(\Gamma,\mathbb{Z})\otimes\mathbb{C}^*) = \mathrm{Hom}(H_1(\Gamma,\mathbb{Z}),\widehat{JN_0}).$$

(J3) The lift $\tau_0 : H_1(\Gamma,\mathbb{Z})\times H_1(\Gamma,\mathbb{Z}) \to (\mathcal{P}^{-1})^*$ where $\mathcal{P}$ is the Poincaré bundle over $JN_0\times\widehat{JN_0}$.

(J6) The monodromy cone for JN_0

The class in $\mathrm{Hom}(H_1(\Gamma,\mathbb{Z}),\widehat{JN_0})$ from (J2) will be called the **classifying map** $c : H_1(\Gamma,\mathbb{Z}) \to \widehat{JN_0}$. For every edge $e_j \in E(\Gamma)$ corresponding to a double point Q_j of C_0 we have two points, Q_j^- and Q_j^+ in N_0. We can associate a line bundle to each edge e_j as follows,

$$e_j \mapsto \mathcal{O}_{N_0}(Q_j^+ - Q_j^-) \in \mathrm{Pic}^0(N_0).$$

This extends to a linear map $C_1(\Gamma,\mathbb{Z}) \to \mathrm{Pic}^0(N_0)$ which descends to a map $H_1(\Gamma,\mathbb{Z}) \to \mathrm{Pic}^0(N_0) = JN_0$. Let Θ be the principal polarization of JN_0, then Θ induces an isomorphism $\phi_\Theta : JN_0 \to \mathrm{Pic}^0(JN_0) = \widehat{JN_0}$. The classifying map $c : H_1(\Gamma,\mathbb{Z}) \to \widehat{JN_0}$ can be defined as the following composition,

$$c : H_1(\Gamma, Z) \to JN_0 \xrightarrow{\phi_\Theta} \widehat{JN_0}.$$

This data will define a unique compactified Jacobian which will correspond to a point in the second Voronoi compactification of A_g. If C_0 is stable the image of C_0 under the Torelli map will correspond to this unique compactified Jacobian.

Let us now describe (J3) in a little more detail. It is convenient to view it as a lift:

$$\begin{tikzcd}
 & (\mathcal{P}^{-1}) \arrow[d] \\
H_1(\Gamma,\mathbb{Z})\times H_1(\Gamma,\mathbb{Z}) \arrow[r, "c\times c"] \arrow[ur, dashed, "\tau_0"] & JN_0\times JN_0
\end{tikzcd}$$

where we have identified $JN_0 = \widehat{JN}_0$ via the principal polarization. We have that τ_0 is given by the Deligne pairing [ABH02]. In a little more detail, recall that given degree 0 line bundles L_1, L_2 on N_0, and rational sections σ_1 and σ_2, respectively, with disjoints supports, the Deligne pairing gives an element $\langle\sigma_1,\sigma_2\rangle \in (\mathcal{P}^{-1}_{(L_1,L_2)})$. The Deligne pairing also only depends on the rational sections, up to scaling by a constant, so that it is actually defined on degree 0 divisors with disjoint supports. See [§13.5][ACG11] for more discussion. Since the classifying map c is actually defined in terms of degree 0 divisors, the Deligne pairing gives a lift as in the diagram above, so long as the supports of the cycles are disjoint. To deal with the case where the supports are not disjoint, one does the following. For each oriented edge $\vec{e}$ of Γ, one associates a rational section $\sigma_{\vec{e}}$ of $\mathcal{O}_{N_0}(t(\vec{e}) - s(\vec{e}))$. For distinct edges, one has the Deligne pairing $\langle \vec{e}_1, \vec{e}_2 \rangle$. One defines $\langle \vec{e}, \vec{e} \rangle$ **arbitrarily**. It is shown in [§5.5][Ale04] that with this arbitrary choice, the lift we will define will agree with τ_0, up to an equivalence that does not affect the period map to the second Voronoi compactification (or, therefore, the construction of Alexeev's limit stable semi-abelic pair). Now one simply defines the lift of a pair of cycles $\sum \vec{e}_i$ and $\sum \vec{f}_i$ as the product of the associated elements $\langle \vec{e}_i, \vec{f}_j \rangle$. This agrees with the Deligne pairing, where both are defined.

13.5 Degenerations of Prym varieties

Let $(\widetilde{C}, \iota)$ be a curve $\widetilde{C}$ with an admissible involution $\iota : \widetilde{C} \to \widetilde{C}$. Observe that we are keeping the notation consistent with the rest of the paper. Define $C = \widetilde{C}/\langle \iota \rangle$ with $\pi : \widetilde{C} \to C$ as the quotient map which will be an étale double cover of $\widetilde{C}$. Let $\mathrm{Nm} := (\pi_* : J\widetilde{C} \to JC)$ and define

$$P_{\widetilde{C}/C} := \ker(\mathrm{Nm} : J\widetilde{C} \to JC)_0$$

that is, $P_{\widetilde{C}/C}$ is the connected component of the kernel of the norm map containing the identity. Alternatively we can define $P_{\widetilde{C}/C}$ as,

$$P_{\widetilde{C}/C} = \ker((1+\iota) : J\widetilde{C} \to JC)_0 = \mathrm{im}((1-\iota) : J\widetilde{C} \to JC)$$

Given a family of curves with admissible involutions $\mathcal{X} \to \Delta$ over the unit disk such that the generic fiber $(\widetilde{C}_0, \iota)$ is a stable curve, the Prym variety associated to $(\widetilde{C}_0, \iota)$ can be described as the limit of Prym varieties of étale double covers of smooth curves. In [§3.2][ABH02], the authors give the combinatorial data required to classify the Prym variety for the generic fiber $P_{\widetilde{C}_0/C_0}$. We focus on the following degeneration data:

(PP0) The abelian variety $P_{\widetilde{N}_0/N_0}$, which is called the compact part, where $\widetilde{N}_0$ is the normalization of the curve $\widetilde{C}_0$. This can be described by the extension,

$$0 \to H^1(\widetilde{\Gamma}, \mathbb{Z})^- \otimes \mathbb{C}^* \to P_{\widetilde{C}_0/C_0} \to P_{\widetilde{N}_0/N_0} \to 0$$

where $\widetilde{\Gamma}$ is the dual graph of $\widetilde{C}$.

(PP1) The classifying map $c^{[-]} : H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} \to \widehat{P_{\widetilde{N}_0/N_0}}$ which is defined as: for any $z \in H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$ one has $2z \in H_1(\widetilde{\Gamma}, \mathbb{Z})^- \subset H_1(\widetilde{\Gamma}, \mathbb{Z})$ and we set $c^{[-]}(z) = c(2z)$ where c comes from the classifying map defined in 13.4.

(PP2) The lattice $H_1(\widetilde{\Gamma}, \mathbb{Z})^-$; the torus $H^1(\Gamma, \mathbb{Z})^- \otimes \mathbb{C}^* \cong (\mathbb{C}^*)^n$, where n is the rank of $H^1(\Gamma, \mathbb{Z})$, is called the non-compact part.

(PP3) The lift $\tau_0^- : H_1(\Gamma, \mathbb{Z})^- \times H_1(\Gamma, \mathbb{Z})^- \to (\mathcal{P}^{-1})^*$ where $\mathcal{P}$ is the Poincaré bundle over $P_{\widetilde{N}_0/N_0} \times \widehat{P_{\widetilde{N}_0/N_0}}$.

The data (PP4)–(PP5) essentially plays an auxiliary role, and the data (PP6) can essentially be determined by the monodromy cone, which we have discussed how to compute earlier. The degeneration data (PP1) and (PP2) for the $2A_1$ cubic was considered in [Hav16]; in short, our focus in this thesis is really on (PP3), which is obtained from restriction from the lift given by the degeneration of the covering curves.

13.6 $2A_1$ cubic threefold

Following the calculations of [§4.1.11][Hav16], if X has two nodes the plane quintic D has two nodes, p and q. The double cover $\widetilde{D}$ has four nodes p^+, p^-, q^+, and q^-. If $\widetilde{\Gamma}$ is the dual graph of $\widetilde{D}$ then in [Hav16] the following was computed,

$$H_1(\widetilde{\Gamma}, \mathbb{Z}) = \mathbb{Z}\left\langle e^+, e^-, f^+, f^- \right\rangle \tag{13.6.1}$$

$$H_1(\widetilde{\Gamma}, \mathbb{Z})^+ = \mathbb{Z}\left\langle e^+ + e^-, f^+ + f^- \right\rangle \tag{13.6.2}$$

$$H_1(\widetilde{\Gamma}, \mathbb{Z})^- = \mathbb{Z}\left\langle e^+ - e^-, f^+ - f^- \right\rangle \tag{13.6.3}$$

$$H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} = \mathbb{Z}\left\langle \frac{1}{2}(e^+ - e^-), \frac{1}{2}(f^+ - f^-) \right\rangle \tag{13.6.4}$$

where $e^{\pm}$ corresponds to $p^{\pm}$ and $f^{\pm}$ corresponds to $q^{\pm}$. The space $H_1(\widetilde{\Gamma}, \mathbb{Z})^-$ has rank 2 and thus the non-compact torus $H_1(\widetilde{\Gamma}, \mathbb{Z})^- \otimes \mathbb{C}^* \cong (\mathbb{C}^*)^2$ has rank 2. The intermediate Jacobian is given by

$$0 \to (\mathbb{C}^*)^2 \to IJ(X) \to P_{N\tilde{C}/NC} \to 0$$

with extension data

$$g_1 \mapsto \mathcal{O}_{N\widetilde{D}}(p_2^+ - p_1^+ - p_2^- + p_1^-) \tag{13.6.5}$$

$$g_2 \mapsto \mathcal{O}_{N\widetilde{D}}(q_2^+ - q_1^+ - q_2^- + q_1^-) \tag{13.6.6}$$

In [§4.1.11][Hav16] it is shown that ND is genus 4 and it is not hyperelliptic. This suggest that we may use theorem 13.3.2 which tells us that the Prym variety of ND is the Jacobian of NC where C is the (2,3)-curve obtained from projection through one of the A_1 singularities.

The complete intersection curve C has one node ξ which corresponds to the line in X passing through both nodes. Assume C was obtained with a projection through the node corresponding to p. We can get a g_3^1 on ND by considering a pencil of lines through the node p and lifting it to ND as a line through p will intersect D in three other points up to multiplicity. The line ℓ' going through both nodes p and q will correspond to $q_1 + q_2 + r$ where q_1 and q_2 come from the desingularization of q and r is the fifth point of intersection on D from ℓ'. From the discussion of the sketch of the proof of theorem 13.3.2 we know that triples of points in $N\widetilde{D}$ correspond to points on C and lift to points in NC. This gives us a way of expressing our extension data in terms of points on the Jacobian of the normalization of C, $J(NC)$. From the extension data in (13.6.5) we get

$$q_2^+ - q_1^+ - q_2^- + q_1^- = (q_1^- + q_2^+ + \tilde{r}) - (q_1^+ + q_2^- + \tilde{r}) = \xi_1 - \xi_2 \in J(NC).$$

The point $\tilde{r} \in N\tilde{D}$ is a point above $r \in ND$. The points ξ_1 and ξ_2 are obtained by the desingularization of ξ. For the extension data in (13.6.6) we can look at the g^1_3 divisors we get from pencils of lines through q. This will give us

$$p_2^+ - p_1^+ - p_2^- + p_1^- = (p_1^- + p_2^+ + \tilde{s}) - (p_1^+ + p_2^- + \tilde{s}) = \eta_1 - \eta_2 \in J(NC).$$

Here the $\tilde{s} \in N\tilde{D}$ is a point above the point $s \in ND$ which is the fifth point of intersection. The points η_1 and η_2 are obtained by the desingularization of $\eta \in C$ which is the node in C coming from the line in X passing through the 2 nodes after projecting from the other node. From theorem 13.2.3, the intermediate Jacobian is given by

$$1 \to (\mathbb{C}^*)^2 \to IJ(X) \to J(NC) \to 0$$

with classifying map given by the data:

$$g_1 \mapsto \mathcal{O}_{NC}(\eta_1 - \eta_2) \tag{13.6.7}$$

$$g_2 \mapsto \mathcal{O}_{NC}(\xi_1 - \xi_2) \tag{13.6.8}$$

where g_1 and g_2 are a choice of generators of the character lattice (which can be canonically identified with $H^1(\widetilde{\Gamma}, \mathbb{Z})^-$). So far this rehashes the results in [Hav16], and the data above corresponds to (PP1), and (PP2). Tacitly, this essentially also computes (PP6). Here we want to explain (PP3). From [ABH02], this is obtained from restriction from the Jacobian case; i.e., from considering the degeneration of the Jacobian of the covering curves. Translating, through the trigonal construction, we want a lift

$$\begin{tikzcd} & (\mathcal{P}^{-1})^* \arrow[d] \\ H_1(\Gamma,\mathbb{Z})^- \times H_1(\Gamma,\mathbb{Z})^- \arrow[r, "c\times c"] \arrow[ur, dashed, "\tau_0"] & JNC \times JNC \end{tikzcd}$$

Since again the classifying map is given by divisors, the Deligne pairing defines a lift, and tracing through the definitions, this defines τ_0.

We make one more observation here, which is that $\xi_1 = \eta_1$ and $\xi_2 = \eta_2$. This was overlooked in [Hav16].

We can also connect this to the work in [CMGHL17a, Thm. 7.2].

Corollary 13.6.1. *In the notation of [CMGHL17a, Thm. 7.2], the dimension* 8 *stratum* $\mathbf{A}11b$ *in the boundary of the intermediate Jacobian locus arises from degenerations to* $2A1$ *cubics.*

Technically this was already known in [CMGHL17a], but the arguments above provide a direct argument via degenerations of Pryms.

Bibliography

[AB12] Valery Alexeev and Adrian Brunyate, Extending the Torelli map to toroidal compactifications of Siegel space, Invent. Math. **188** (2012), no. 1, 175–196. MR 2897696

[ABH02] V. Alexeev, Ch. Birkenhake, and K. Hulek, Degenerations of Prym varieties, J. Reine Angew. Math. **553** (2002), 73–116. MR 1944808

[ACG11] Enrico Arbarello, Maurizio Cornalba, and Pillip A. Griffiths, Geometry of algebraic curves. Volume II, Grundlehren der Mathematischen Wissenschaften [Fundamental Principles of Mathematical Sciences], vol. 268, Springer, Heidelberg, 2011, With a contribution by Joseph Daniel Harris. MR 2807457

[Ale04] Valery Alexeev, Compactified Jacobians and Torelli map, Publ. Res. Inst. Math. Sci. **40** (2004), no. 4, 1241–1265. MR 2105707

[Bab15] László Babai, Graph isomorphism in quasipolynomial time, CoRR **abs/1512.03547** (2015).

[Cay69] Professor Cayley, A memoir on cubic surfaces, Philosophical Transactions of the Royal Society of London **159** (1869), 231–326.

[CG72] C. Herbert Clemens and Phillip A. Griffiths, The intermediate Jacobian of the cubic threefold, Ann. of Math. (2) **95** (1972), 281–356. MR 0302652

[CMGHL17a] Sebastian Casalaina-Martin, Samuel Grushevsky, Klaus Hulek, and Radu Laza, Complete moduli of cubic threefolds and their intermediate jacobians, preprint.

[CMGHL17b] ———, Extending the Prym map to toroidal compactifications of the moduli space of abelian varieties, J. Eur. Math. Soc. (JEMS) **19** (2017), no. 3, 659–723, With an appendix by Mathieu Dutour Sikirić. MR 3612866

[CMJL12] Sebastian Casalaina-Martin, David Jensen, and Radu Laza, The geometry of the ball quotient model of the moduli space of genus four curves, Compact moduli spaces and vector bundles, Contemp. Math., vol. 564, Amer. Math. Soc., Providence, RI, 2012, pp. 107–136. MR 2895186

[DSHS15] Mathieu Dutour Sikirić, Klaus Hulek, and Achill Schürmann, Smoothness and singularities of the perfect form and the second Voronoi compactification of $\mathcal{A}_g$, Algebr. Geom. **2** (2015), no. 5, 642–653. MR 3421785

[DSSV08] Mathieu Dutour Sikirić, Achill Schürmann, and Frank Vallentin, A generalization of Voronoi's reduction theory and its application, Duke Math. J. **142** (2008), no. 1, 127–164. MR 2397885

[FS86] Robert Friedman and Roy Smith, Degenerations of Prym varieties and intersections of three quadrics, Invent. Math. **85** (1986), no. 3, 615–635. MR 848686

[Hav16] Krisztian Havasi, Geometric realization of strata in the boundary of the intermediate Jacobian locus, ProQuest LLC, Ann Arbor, MI, 2016, Thesis (Ph.D.)–University of Colorado at Boulder. MR 3527174

[JK15] Tommi Junttila and Petteri Kaski, Bliss: A tool for computing automorphism groups and canonical labelings of graphs, http://www.tcs.hut.fi/Software/bliss/, 2015.

[KOr06] Petteri Kaski and Patric R. J. Östergå rd, Classification algorithms for codes and designs, Algorithms and Computation in Mathematics, vol. 15, Springer-Verlag, Berlin, 2006, With 1 DVD-ROM (Windows, Macintosh and UNIX). MR 2192256

[McK88] Brendan D. McKay, Isomorph-free exhaustive generation, Journal of Algorithms **26** (1988), 306–324.

[MP13] Brendan D. McKay and Adolfo Piperno, Practical graph isomorphism, II, CoRR **abs/1301.1493** (2013).

[Mum74] David Mumford, Prym varieties. I, 325–350. MR 0379510

[Nam76] Yukihiko Namikawa, A new compactification of the Siegel space and degeneration of polarized Abelian varieties, Sûgaku **28** (1976), no. 3, 214–225. MR 0498608

[Rea78] Ronald C. Read, Every one a winner or how to avoid isomorphism search when cataloguing combinatorial configurations, Ann. Discrete Math. **2** (1978), 107–120, Algorithmic aspects of combinatorics (Conf., Vancouver Island, B.C., 1976). MR 0491273

[Rec74] Sevin Recillas, Jacobians of curves with g^1_4's are the Prym's of trigonal curves, Bol. Soc. Mat. Mexicana (2) **19** (1974), no. 1, 9–13. MR 0480505

[Ser03] Jean-Pierre Serre, Trees, Springer Monographs in Mathematics, Springer-Verlag, Berlin, 2003, Translated from the French original by John Stillwell, Corrected 2nd printing of the 1980 English translation. MR 1954121 (2003m:20032)

[Vol02] Vitaly Vologodsky, The locus of indeterminacy of the Prym map, J. Reine Angew. Math. **553** (2002), 117–124. MR 1944809