A COMPUTATIONAL APPROACH TO DETERMINING THE INDETERMINACY LOCUS OF THE PRYM MAP WHEN $g = 5$

JOSH FRINAK

ABSTRACT. abstract

CONTENTS

Date: January 17, 2018.

1. Introduction

In this paper, we consider the period map for Prym varieties. For motivation, the he Torelli map $M_g \to A_g$ is a rational map from the moduli space of smooth curves of genus g to the moduli space of principally polarize abelian varieties of dimension g. This map is defined by assigning a smooth curve of genus g its Jacobian $\operatorname{Pic}^0 X$. A fundamental 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 paper we will be concerned with three standard toroidal compactifications of the moduli space principally polarized abelian varieties: second Voronoi, and perfect cone, and central cone compactifications.

In [Nam], 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 [AB], 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. Associating a principally polarized abelian variety to connected étale double covers of curves gives us a Prym variety. This association gives the Prym period map $R_g \to A_{g-1}$ 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 dimension principally polarized abelian varieties.

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
1 $\Longrightarrow$ map in the case of each Toroidal compactification of A_g. In summary of [CMGH$^+$], [1] the Prym map does not extend to a regular map in any of the standard toroidal compactifications of A_{g-1}.

In Alexeev–Birkenhake–Lange and Vologodsky, they described the indeterminancy locus in the case of the second Voronoi compactification, $\overline{R}_{g+1} \dashrightarrow \overline{A}_g^V$. In [RS] Friedman and Smith found some explicit examples of admissible double covers where the Prym map does not extend to a regular map. In [ABH], Alexeev, Birkenhake, and Hulek identify the indeterminacy locus of $\overline{R}_{g+1} \to \overline{A}_g$ as the closure of the Friedman-Smith locus with 4 nodes. In [CMGH$^+$], examples where calculated by hand.

In this paper we study the indeterminacy locus of the Prym map in the case of the perfect cone compactification. In Casalaina-Martin–Gr–H–L they gave a partial description in Theorem .... (Include a description of the result. In that paper they provide a partial description, showing the indeterminacy locus is contained in ... The first case where this is not known is in genus 5. Here we try to answer that question in genus 5.) The contribution of this paper will be writing a computer program capable of determining whether or not any Fiedman-Smith covers with 4 or more nodes belongs to the indeterminacy locus of the Prym map in the case of $g = 5$. We will generate all possible dual graphs in the genus 5 situation and then generate all possible admissible double covers for those dual graphs up to isomorphism. For each isomorphism class of admissble double covers we will find a basis of quadratic forms and determine wheter or not the fan generated by the quadratic forms lies within the fan determining the second Voronoi compactification. The main result of this paper is as follows.

[1](Yano) Actually this is the Friedman–Smith paper.

Theorem 1.1. *In the case of $g = 5$, the only admissible double covers in the indeterminacy locus of the Prym map, $\overline{R}_{g+1} \to \overline{A}_g$, are degenerations of Friedman-Smith graphs with 2 or 3 nodes.*

The strategy is:

The question is entirely combinatorial due to CM-G ... , depending on the dual graphs of the admissible cover.

The dual graphs give rise to associated matrices, whose rows give linear forms, whose squares give the quadratic forms defining the rays of the monodromy cone.

We enumerate all of the possible dual graphs, removing those that we can for simple reasons, and then enumerate all the associated monodromy cones.

We then use a program to determine if these cones lie in perfect cones.

This gives the result.

The main difficulty in obtaining results in higher genus is that the number of dual graphs gets to be too large.

We do get partial results in higher genus, which we describe later (or we leave to the reader); results on the indeterminacy locus in examples, which describe the indeterminacy locus up to codimension ... If we do examples for every base curve with n edges, then we have obtained the result up to codimension n.

OUTLINE:

In section 2, we do ... in section 3 we do ...

2. Notation for graphs

ec: Defgraph

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

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

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.

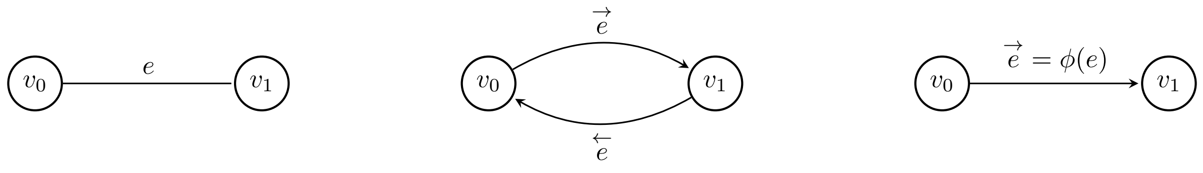

FIGURE 1. The left and center figures give two depictions of the same graph Γ, one showing the unoriented edges $E(\Gamma)$ (left), and the other showing the oriented edges $\overrightarrow{E}(\Gamma)$ (center). The figure on the right depicts an oriented graph (Γ, ϕ), obtained by a choice of orientation of the graph Γ.

F:graphEx

Homomorphism

2.2. **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, $f_{\overrightarrow{E}} : \overrightarrow{E}(\Gamma_1) \to \overrightarrow{E}(\Gamma_2)$ 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 2.1. Let Γ be the graph depicted in Figure 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.

Sec: HomGra

2.3. **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

$$\begin{aligned}\partial : \vec{C}_1(\Gamma, R) &\longrightarrow \vec{C}_0(\Gamma, R) = C_0(\Gamma, R)\\ \vec{e} &\mapsto t(\vec{e}) - s(\vec{e}).\end{aligned}$$

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

Remark 2.2. 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 $H_n(f) : H_n(\Gamma_1) \to H_n(\Gamma_2)$.[2]

2 $\Longrightarrow$

idencematrix

2.4. **Incidence matrix of an oriented graph.** Given a finite oriented graph (Γ, ϕ), and enumerations $V = \{v_1, \ldots, v_n\}$ and $E = \{e_1, \ldots, e_m\}$, one obtains the associated $n \times m$ incidence matrix A, with entries a_{ij} as follows,

$$a_{ij} = \begin{cases} -1 & s(\phi(e_j)) = v_i \neq t(\phi(e_j)), \\ 1 & t(\phi(e_j)) = v_i, \\ 0 & \text{else.} \end{cases}$$

[2](Josh) Maybe cite Hatcher §2 pg 110

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, one enters 1.

For instance, the following graph,

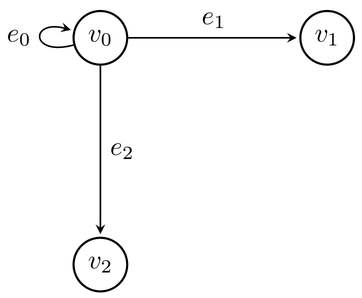

FIGURE 2. Basic Graph with Loop

has incidence matrix,

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

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 one entry is nonzero and is equal to 1, or exactly two entries are nonzero, with one equal to 1 and the other equal to -1. There is an obvious identification between finite oriented graphs with enumerated vertices and edges, and abstract incidence matrices.

$$IM : \{\text{finite oriented graphs with enumerated vertices and edges}\} \xrightarrow{1:1} \{\text{incidence matrices}\}$$

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

onnectedness

2.5. **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 rank $H_0(X, \mathbb{Z}) = 1$.

S:Loop

2.6. **Loops in graphs.** Given a graph Γ, will make use of 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 basis vector (has a unique 1 and all else 0).

ec: HomBases

2.7. **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.

S:GrCl

3. 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 primarliy 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. 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 caclulate Homology (see 2.7). Another reason we need to extend the graph class in Sage is because we need a convienient 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 2.4.

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.

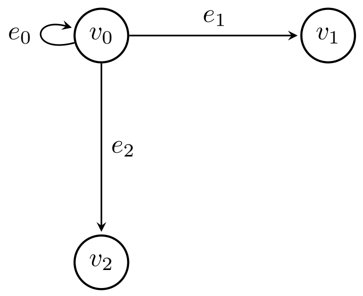

FIGURE 3. Basic Graph with Loop

This basic graph will have incidence matrix (§2.4) 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 compile 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 compile command would look like

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

and `configs` would look like,

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

Remark 3.1. 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. Refering 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 lable. In our case we will use the lable 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.

dmissibility

4. 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.

4.1. Admissible involutions of graphs.

Definition 4.1 (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 2.2) 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 §4.3 for some examples.

Remark 4.2. 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 [CMGH+].

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.

lma: lma1

Lemma 4.3. *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:

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

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. Similarily, $\iota_V(t(\tilde{\phi}(\tilde{e}))) = t(\tilde{\phi}(\tilde{e}))$. Therefore, ι defines an admissible involution of the graph $\widetilde{\Gamma}$.

Conversely, suppose we are given a graph $\widetilde{\Gamma}$ and an admissible involuiton $\iota : \widetilde{\Gamma} \to \widetilde{\Gamma}$. We can construct a on 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 $\vec{E}(\Gamma) = \vec{E}(\widetilde{\Gamma})/\iota_{\vec{E}}$. 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 $\pi(s(\iota_{\vec{E}}(\tilde{\phi}(\tilde{e})))) = \pi(s(\tilde{\phi}(\tilde{e})))$ 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 2.2 we know that $\iota : \widetilde{\Gamma} \to \widetilde{\Gamma}$ is a graph morphism that preserves the orientation of $\widetilde{\Gamma}$. $\square$

Remark 4.4. In light of Lemma 4.3, 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 4.5. In the proof of lemma 4.3 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 implimentation 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 4.6. We will now frequently drop the notation $\vec{e}$ for an oriented edge. For the rest of the paper 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.

4.2. Admissible double covers of graphs.

dfn: dfn1

Definition 4.7. 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 4.8. 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. [CMGH+]

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., $\tilde{v}^+ = \iota(\tilde{v}^-)$.

xa-cov-graph

4.3. **Example of admissible involutions and covering graphs.** First we will consider a very basic example of a graph.

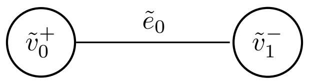

FIGURE 4. Non admissible involution

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.

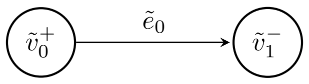

FIGURE 5. Non admissible involution (oriented)

If we consider the following base graph Γ,

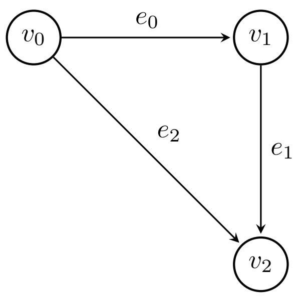

FIGURE 6. Base Graph Γ

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

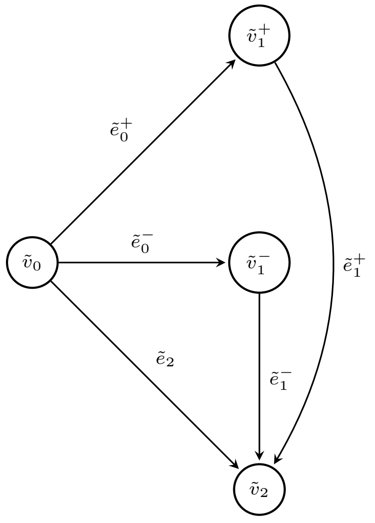

FIGURE 7. Admissible cover of Γ

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

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

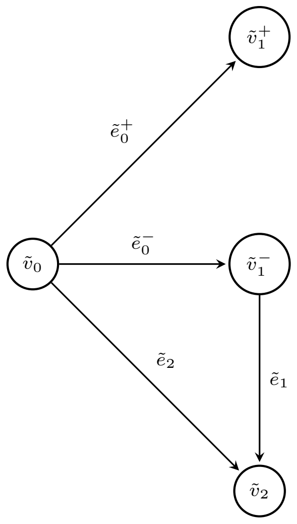

FIGURE 8. Not an admissible cover of Γ

loopCovers

4.4. **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 be 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.

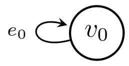

FIGURE 9. Base Graph Loop

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

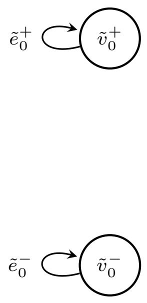

FIGURE 10. First Loop Cover Option

(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.

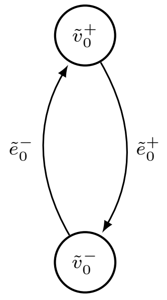

FIGURE 11. Second Loop Cover Option

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

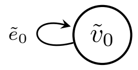

FIGURE 12. Covering a ramified loop for ramified vertex

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 [CMGH$^+$, §3.4]

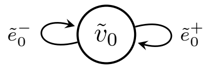

FIGURE 13. Covering a loop for ramified vertex

Remark 4.9. We will be able to reduce all admissible double covering dual graphs which contain loops to calculations on graphs of smaller dimensions (see section 11.1). This indicates that we only need to consider admissible covers where the loops are covered and where the loop vertices are covered. Also, in this case we only have to consider loop covering option number 2.

4.5. **The cone of quadratic forms associated to an admissible double cover of graphs.** As described in [CMGH$^+$], 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 $\pm$ eigen spaces 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

E:H1[-] (4.1)
$$\frac{1}{2}(\mathrm{Id} - \iota) : H_1(\widetilde{\Gamma}, \mathbb{Z}) \to H_1(\widetilde{\Gamma}, \frac{1}{2}\mathbb{Z}).$$

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 4.10. 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 $\pm 1, 0$) 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 [CMGH+, §5.2] for more details.

4.5.1. *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 (4.1), in practice, one can compute a basis of $H_1(\widetilde{\Gamma},\mathbb{Z})$, and then compute a basis for the image of $\frac{1}{2}(\operatorname{Id} - \iota)$ 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 extremal rays of the cone $\overline{\sigma}(\widetilde{\Gamma}/\Gamma)$.

S:FSMonCone

4.5.2. *Computing the monodromy cone in an example: Friedman–Smith covers.* The following example is taken verbatim from [CMGH+, §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 $E(\widetilde{\Gamma}) = \{\tilde{e}_1^+, \tilde{e}_1^-, \dots, \tilde{e}_n^+, \tilde{e}_n^-\}$. The involution ι acts by $\iota(\tilde{v}_i) = \tilde{v}_i$ $(i = 1,2)$ and $\iota(\tilde{e}_i^+) = \tilde{e}_i^-$ $(i = 1,\dots,n)$. For simplicity, we will fix a compatible orientation on $\widetilde{\Gamma}$, as in Figure 34; i.e. for all i set $t(\tilde{e}_i^\pm) = \tilde{v}_2$ and $s(\tilde{e}_i^\pm) = \tilde{v}_1$.

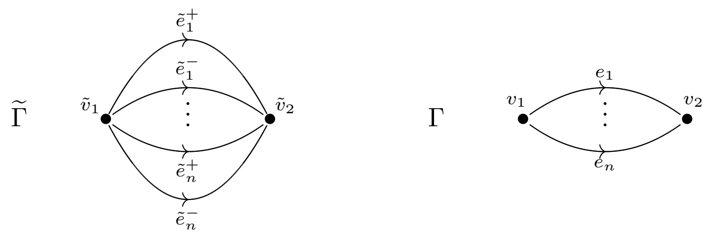

Fig:dgFSG

FIGURE 14. Dual graph of a Friedman–Smith example with $2n \geq 2$ nodes (FS_n).

One has

eqnh1fs

$$H_1(\widetilde{\Gamma},\mathbb{Z}) = \mathbb{Z}\langle \tilde{e}_1^+ - \tilde{e}_1^-, \dots, \tilde{e}_n^+ - \tilde{e}_n^-, \tilde{e}_1^+ - \tilde{e}_2^-, \dots, \tilde{e}_{n-1}^+ - \tilde{e}_n^- \rangle. \tag{4.2}$$

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 $\tilde{e}_1^+ - \tilde{e}_1^-, \tilde{e}_1^+ - \tilde{e}_2^-, \dots, \tilde{e}_n^+ - \tilde{e}_n^-, \tilde{e}_{n-1}^+ - \tilde{e}_n^-$ and constructs a matrix with rows expressing these elements with respect to the basis $\tilde{e}_1^-, \tilde{e}_1^+, \dots, \tilde{e}_n^-, \tilde{e}_n^+$, 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 (4.2), one has

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

For brevity, set

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

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

Now observe that

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

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

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

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

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

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 **??** holds in this case. Now consider the case $n \geq 2$. Evaluating the ℓ_{e_i} on the basis $z_1, \dots, z_n$, we obtain that

$$\begin{array}{lllllllll} \ell_{e_1} & = & 2z_1^\vee & + & z_2^\vee & & & & \\ \ell_{e_2} & = & & & z_2^\vee & + & z_3^\vee & & \\ \vdots & \vdots & & & & \ddots & & & \\ \vdots & \vdots & & & & & \ddots & & \\ \ell_{e_{n-1}} & = & & & & & z_{n-1}^\vee & + & z_n^\vee \\ \ell_{e_n} & = & & & & & & & z_n^\vee \end{array} \tag{4.3}$$

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

eqnFSMCo

$$\begin{pmatrix} 2 & & & & & \\ 1 & 1 & & & & \\ & 1 & 1 & & & \\ & & \ddots & \ddots & & \\ & & & 1 & 1 & \\ & & & & 1 & 1 \\ & & & & & 1\ 1 \end{pmatrix}. \tag{4.4}$$

4.6. **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.

4.6.1. *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:

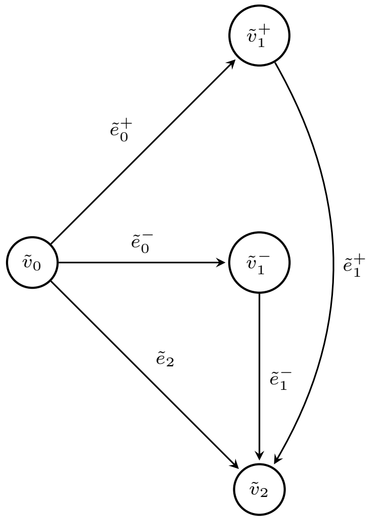

FIGURE 15. The graph $(\widetilde{\Gamma}, \tilde{\phi})$ with the involution ι indicated with the superscripts.

From the oriented graph we immediately obtain the incidence matrix:

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

Since there are no loops, this incidence matrix is equal to the reduced incidence matrix

$$A' = \begin{array}{c|ccccc} & \tilde{e}_0^- & \tilde{e}_0^+ & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2 \\ \hline \tilde{v}_0 & -1 & -1 & 0 & 0 & -1 \\ \tilde{v}_1^- & 1 & 0 & -1 & 0 & 0 \\ \tilde{v}_1^+ & 0 & 1 & 0 & -1 & 0 \\ \tilde{v}_2 & 0 & 0 & 1 & 1 & 1 \end{array}$$

4.6.2. *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':

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

We then augment with appropriate rows to obtain

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

We then drop the first three columns (the columns that are basis vectors), and transpose, to obtain the matrix

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

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

4.6.3. *Step 3: Compute a generating set for $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$.* Next we compute a generating set for $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]}$. To do this, 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:

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

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 $\frac{1}{2}(\mathrm{Id} - \iota)B^T$

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

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

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

4.6.4. *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

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

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

4.6.5. *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:

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

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 extremal rays of the cone of quadratic forms

$$Q = \begin{array}{c} \begin{array}{ccc} \ell_{e_0} & \ell_{e_1} & \ell_{e_2} \end{array} \\ \begin{bmatrix} 1 & 1 & 0 \end{bmatrix} \end{array}.$$

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.

4.6.6. *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.

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

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

Since there are no loops, this incidence matrix is equal to the reduced incidence matrix

$$A' = \begin{array}{c|cccccccc} & \tilde{e}_1^- & \tilde{e}_1^+ & \tilde{e}_2^- & \tilde{e}_2^+ & \cdots & \tilde{e}_n^- & \tilde{e}_n^+ \\ \hline \tilde{v}_1 & -1 & -1 & -1 & -1 & \cdots & -1 & -1 \\ \tilde{v}_2 & 1 & 1 & 1 & 1 & \cdots & 1 & 1 \end{array}$$

Next we compute the reduced row echelon form of A':

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

We then augment with appropriate rows to obtain

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

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

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

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).

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

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 extremal 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.

4.7.1. *A few observations on row operations.* Note that using row operations, we can put this matrix in the form

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

which is the form used in [CMGH+, 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

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

This version also seems to frequently appear in particular computations.

S: CGCl

5. 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 5.1
- 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.

SubSec: CIM

5.1. **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 2.4, rows will corresond to vertices and columns will corresond 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.

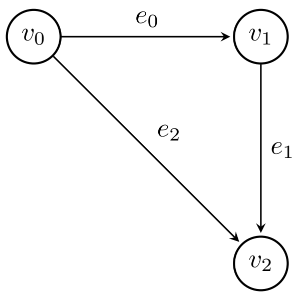

FIGURE 16. Oriented base graph (Γ, ϕ)

The oriented base graph (Γ, ϕ) has three vertices and three edges, and has the following `Incidence_Matrix` (see §2.4).

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

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

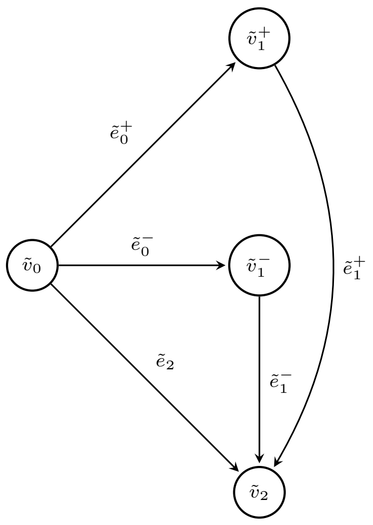

FIGURE 17. Oriented cover graph $(\widetilde{\Gamma}, \tilde{\phi})$ of (Γ, ϕ)

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

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

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 memory allotment 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 corresonds to an edge (or vertex) in the cover graph which is fixed by the involuiton. 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 calcuating 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,

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

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 isomorphims. 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 11).

`sLinearForms`

5.2. **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 [CMGH+, §3.2], define $H_1(\widetilde{\Gamma}, \mathbb{Z})^\pm$ 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 generators of the $\mathbb{Z}$-module $H^1(\widetilde{\Gamma},\mathbb{Z})^{[-]}$.

Example 5.1. 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,

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

ec: BasisLFs

5.3. **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 the process up 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 preformed by calling find_image_chb().
(3) Use Image_CHB to find the basis of linear forms, preformed 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 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 correspoding 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_Involu`
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 throught 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 fuction operations occur over the same ring.

Example 5.2. Let us consider the running example from section 5.1. 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 preforming the echolon 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 equal to the zero vector. We have concealed this code in the very basis 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 5.3. 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 modificaiton, if all the entries in one row are divisible by 2 then we divide the row by two as may times as possible. Thus,

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

6. Generating all Base Graphs

enBaseGraphs

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

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

There are two main reasons for this bifurcation in the code. First, the most time consuming aspect of the compilation is graph isomorphism checking. If we seperate the graph isomorphism checking process into two compilations – one for base graphs and one for cover graphs – we can better gage the progress of the compile. The second reason for the seperation 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 mulitple different types of computations.

Now is a good time to recall that we are really interested in unoriented graphs; the orientation merely provides us a computational tool with which 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 18. 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.

imBaseGraphs

6.1. **Generating low dimensional base graphs.** We start by discribing 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 dimenional 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 list of length two taking values within the indexing set of our vertices. This is a rather basis 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 2.4). 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. Two avoid generating graphs equivalent up to relabling 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`.

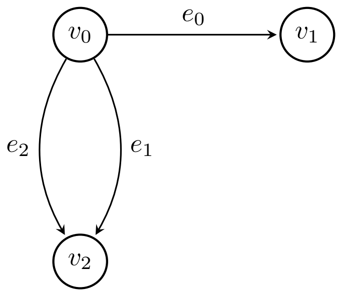

FIGURE 19. The directed graph corresponding to `incidence_matrix_array` = [0 1 1].

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 weeded out 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.

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

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 interation 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 preform 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 summerized as follows after incrementing a value in `incidence_matrix_array` we need to reset all of the preceeding 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, $\texttt{incidence\_matrix\_array} = [\begin{matrix} 0 & 0 & 2 \end{matrix}]$, if we apply the `Iterate_vector` function to `incidence_matrix_array` we would loop through the indices and at $i = 0$ we see that $\texttt{incidence\_matrix\_array}[2]$ already takes on its maximum value. When $i = 1$ $\texttt{incidence\_matrix\_array}[1] = 0$ and $\alpha(1) = 2$ as we calculated in the table. Thus `Iterate_vector` incraments $\texttt{incidence\_matrix\_array}[1]$ 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 $\texttt{incidence\_matrix\_array}[2]$ is 1 because we haven't repeated the value 1 yet and `incidence_matrix_array` becomes $[\begin{matrix} 0 & 1 & 1 \end{matrix}]$. Next, consider when `incidence_matrix_array` equals $[\begin{matrix} 0 & 2 & 2 \end{matrix}]$, the `Iterate_vector` will increment the $i = 2$ index. Therefore $\texttt{incidence\_matrix\_array}[0] = 1$ and we call

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

to reset the entries after index 0. The function `reset_vector` will change $\texttt{incidence\_matrix\_array}[1] = 1$ because we haven't reapeated the value 1 in the array. Then when we get to index 2 we let $\texttt{incidence\_matrix\_array}[2] = 2$ and this is because we have now repeated the value 1 twice and $\texttt{max\_repeat} = 2$.

At this point we have discussed how to iterate through all combinations of `incidence_matrix_array`. Next we will discuss the how to generate a member of the `EGraph` class from the `incidence_matrix_array`.

siveGraphGen

6.2. Recursive base graph generation for all higher dimensional graphs.

ConsEGraph

6.3. **Creating a member of the `EGraph` class from `incidence_matrix_array`.** In the previous sections 6.1 and **??** we discussed how all of the information of the graph is contained in the list of integers called `incidence_matrix_array`. This is not entirely true in the cases where the base graph contains loops. To begin this section we must discuss another object which contains 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 partion 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,

$$\texttt{LA} = [(2), (\begin{matrix} 1 & 1 \end{matrix})].$$

Consider the first integer partion given by the tuple (2) this implies that the base graph will have 2 loops on the vertex indexed by 0. The partion correponding to the tuple $(\begin{matrix} 1 & 1 \end{matrix})$ implies that the base graph will have two loops, one on the vertex index 0 and another on the verex 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 partion 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 mulitple 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 3 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 correponding vertex and marking a row 1 if the edge terminates at the corresonding vertex (see section 2.4). Finally the function creates a member of the `EGraph` class by using the `EGraph` constructor.

6.4. Constructing all base graphs and isomorphism checking.

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 right the graph to our designated ouput 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() 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
      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 simplication 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`,

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

The local variable `la` will represent the loop array which is an integer partion 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 6.3. The next check

```
if(E.G.is_connected() and not(2 in E.G.degree() or 1 in E.G.degree())):
```

uses the SAGE graph library to check if E is connected and rules out graphs with a one-valency vertex or a two-valency vertex. The elimination of graph with low valency vertices will be discussed in sections 11.2 and 11.3.

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 ouput 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).

S: Coverings

7. Generate All Possible Dual Graphs for 2:1 Coverings

We generate all admissible 2:1 covers of our base graphs in a seperate `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 6. 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 correponding 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 4.4). We are able to reduce the case to 2:1 covers that contain no loops in the cover (see section 11.1). Therefore if there is a loop in the base graph at vertex v_i we can gaurantee that the vertex v_i must be covered and thus `CV`$[i]$ = `True`. Recall that we chose to label the vertices such that any loops would occur on the lowest indexed vetices. 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`.

genCVandCE

7.1. **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 7.2. 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 ??. 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.

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

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.

lveCrossings

7.2. **Resolving Crossings in the Covering Graph.** A very observant reader will notice that we are making a choice in the case were 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.

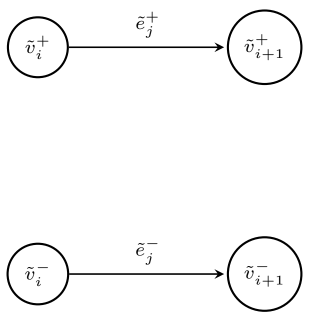

FIGURE 20. Covering Option 1

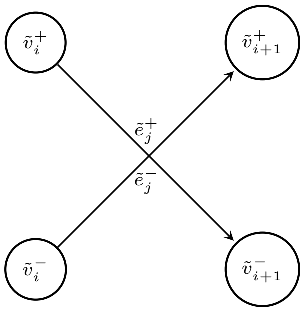

FIGURE 21. Covering Option 2

In the case of covering option 2 we say that the edge e_j lifts to crossing edges $\tilde{e}_j^-$ and $\tilde{e}_j^+$.

Definition 7.1. If e is a non loop edge in Γ, covered by distinct edges $\tilde{e}^+, \tilde{e}^-$, then we will call e *completely covered* (see Figures above ...). In particular, loops cannot be completely covered edges. 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 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 ....). Otherwise, we say $\tilde{e}^+, \tilde{e}^-$ are uncrossed (see Figure ...).

A very brute force way to deal with this multiple covering option problem is to create two possile covering graphs for each completely covered edge in the base graph. This is solution to our problem but it is not very efficient. In some cases we can apply a graph isomrophism to untwist crossing edges in the covering graph but resolving one crossing may result in others. It is a natural question to ask whether we can always resolve all crossings in a covring graph. The answer to that is no, consider the following example.

Example 7.2. If we start with the simple base graph Γ depicted below,

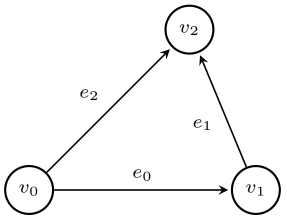

FIGURE 22. Base Graph Γ

we can have two potential admissible covers for Γ.

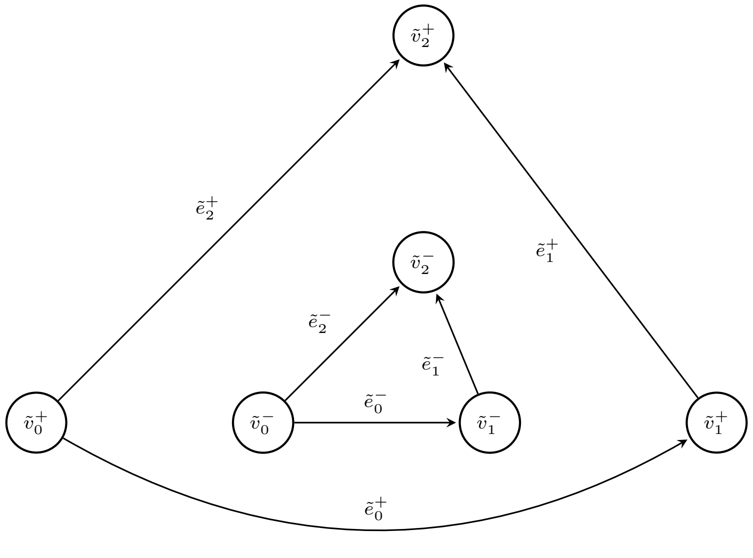

Figure 23. Covering e_2 using option 1

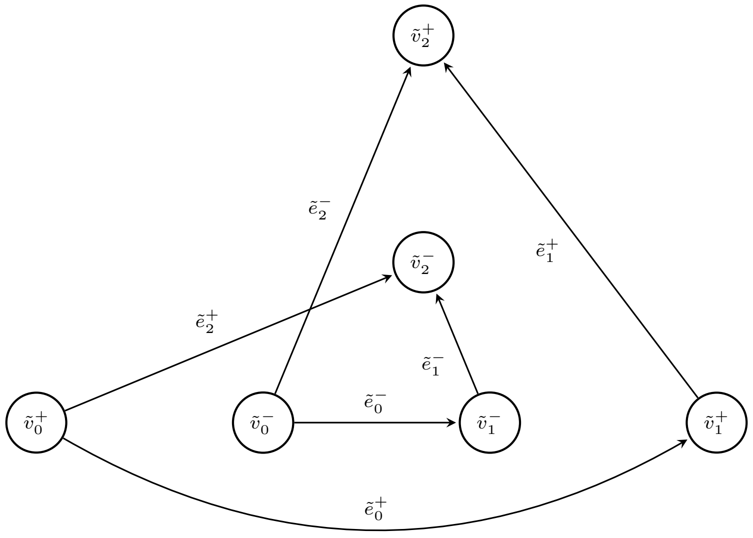

Figure 24. Covering e_2 using option 2

The two addmissible 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 admissble 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 11.1). Furthermore any loop only has one choice of covering (see section 4.4) 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 commpletely covered edges that are crossed in the covering graph. The easiest way to resolve a crossing of an edge would be to involute the starting or terminal vertex of the edge in the cover graph. The problem with this is that any other completely covered edges sharing a valency with this vertex may become crossed. In identifying the right verices to involute we need to consider a special kind of vertex degree.

Definition 7.3. 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. Also, assume that m of these edges also lift to crossings (thus $m \leq n$). If $m > \frac{n}{2}$ we say that the vertex v has *overloaded crossed edge degree.*

Example 7.4. Consider the following dual graph Γ and admissble covering graph $\widetilde{\Gamma}$.

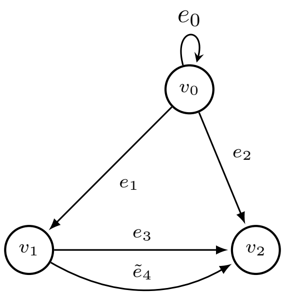

FIGURE 25. Base Graph Γ

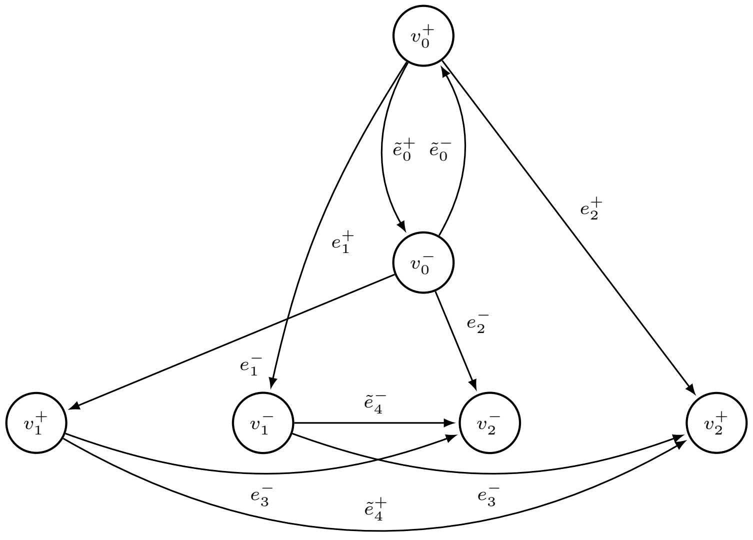

FIGURE 26. Admissible Cover Graph $\widetilde{\Gamma}$

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 $\widetilde{\Gamma}$. The vertex $v_2 \in V(\Gamma)$ does not have overloaded crossed edge degree because only e_3 lifts to a crossed edge in $\widetilde{\Gamma}$. Finally, $v_0 \in V(\Gamma)$ does not have overloaded crossed edge degree becuase 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.

Lemma 7.5. *Let $\widetilde{\Gamma}$ be an admissible cover of a dual graph Γ which has $n > 0$ completely coverd 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 vetices 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. Given that $m > \frac{n}{2}$, $2m > \frac{2n}{2}$. Therefore there must be at least
$3 \Longrightarrow$ one vertex S that has overloaded crossed edge degree. [3] $\square$

Lemma 7.6. *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 than we can preform 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. $\square$

The previous two lemmas prove the following proposition.

prop: prop1 **Propostion 7.7.** Let $\widetilde{\Gamma}$ be an admissible cover of a dual graph Γ which has $n > 0$ completely coverd 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 $m \leq \frac{n}{2}$.

[3](Yano) Add the proof of this. $m = \sum_v m_v/2,\ n = \sum_v n_v/2$, thus if $m_v \leq n_v/2$ for all v, then $m \leq \sum \leq \sum /2 = n/2$.

Proposition 7.7 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

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

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 nee 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` . Coversely if one of the loop vetices 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
  self.ccEdges = [False]*self.configs["edges"] #array of completely covered edges that are
  #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
```

ossibleCover

7.3. Get a possible cover. [4] 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 loops. ⟸ 4

```
def get_Possible_Cover_IM_no_loops(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"]):
```

[4](Josh) Update this section

```
    # 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 4.3.

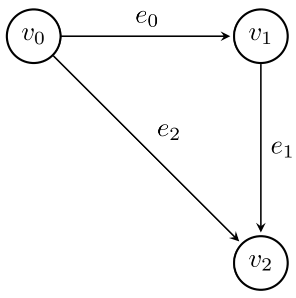

FIGURE 27. Base Graph E

This base graph will have the following Incidence_Matrix

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

The first step of get_Possible_Cover_IM is to copy the information from the incidence matrix of the base graph E to the new possible incidence matrix. Therefore, if `Possible_Cover_IM` is the 2-dimensional list output then after copying the information from the incidence matrix of E, `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 5 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.

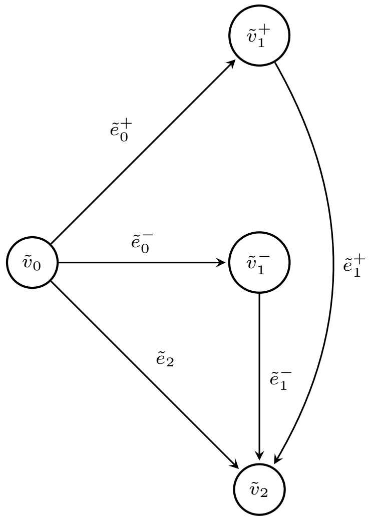

FIGURE 28. Possible Cover of E

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_ℓ 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 cycles the process is very similar but there are more aspects to consider. For example in the case where we have completely covered loops we are also allowed crossed edges in the covering graph. For this reason we will need more inputs in our `get_Possible_Cover_IM` function in this context.

We will introduce three new lists. The first list is called `ccl_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 is part of a completely covered loop. In populating this and the following list we need to explain the `generate_CC_loops` function from the `EGraph` class.

The `generate_CC_loops` function from the `EGraph` class will keep a list of each of the basis vectors of $H_1(\Gamma, \mathbb{Z})$ that are completely covered in terms of the list `CV` and `CE` . The function has three inputs: `self` which represents that it is a class function and must be associated with an `EGraph` object, `CV` which is the list storing the information about covered vertices, and `CE` which is the list storing information about covered edges. The function will return a list of tuples which will be a subset of `Homology_Basis` called `CC_loops` .

```python
def generate_CC_loops(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
    flag = True
    #For each basis vector in the homology basis
    for bv in self.Homology_Basis:
        flag = True
        #For each edge in the basis vector
        for e in range(self.configs["edges"]):
            #if the edge is actually a part of the cycle
            if(bv[e]!= 0):
                #verify that the edge is covered
                condition_1 = not CE[e]
                #verify that if the edge is a loop then the vertex is covered
                condition_2 = (e<self.configs["loops"]) and not self.are_LVs_covered
                #verify that if edge not a loop then the starting vertex of the edge is covered
                condition_3 = (e>=self.configs["loops"]) and not CV[self.start(e)]
                #verify that if edge not a loop then the ending vertex of the edge is covered
                condition_4 = (e>=self.configs["loops"]) and not CV[self.end(e)]
                if(condition_1 or condition_2 or condition_3 or condition_4):
                    flag = False
                    break;
        if(flag):
            self.CC_loops.append(bv)
```

We first check whether loops in the base graph will correspond to completely covered cycles given the specific `CV` and `CE` . To do this we construct a list called `loop_verts` which uses the SAGE graph library function `loop_vertices` to create a list of all vertices which have a loop valency. We then loop through all vertices in `loop_verts` and check whether that index in `CV` corresponds to a `true` value. If all vetices in `loop_verts` are covered than the boolean flag value `are_LVs_covered` remains true if one of the loop vertices is not covered than we set `are_LVs_covered` to false.

Next we loop through all the basis vectors in `Homology_Basis` . For each cycle in `Homology_Basis` we loop through all the edges (nonzero values) in the cycle. If the edge is a part of the cycle we have four conditions to check.

(1) We verify that the edge is covered
(2) We verify that if the cycle is a loop (a cycle of length 1 then the corresponding vertex is covered.
(3) We verify that if the cycle is not a loop then $s(e)$ is covered.

(4) We verify that if the cycle is not a loop then $t(e)$ is covered.

These four criteria correspond to conditions 1-4 in the code. If any of these conditions are true, meaning that the edge does not satisfy the condition then we break out of the cycle check and change our `flag` to false. If the `flag` is `false` this implies that the corresponding cycle is not completely covered and will not be added to `CC_Loops` .

Now we are ready to generate `ccl_edges` and `ccl_edge_locations` , recall that `ccl_edges` is a list stores whether a particular edge is a part of a completely covered cycle in $H_1(\Gamma, \mathbb{Z})$ given the choices of `CV` and `CE` . The list `ccl_edge_locations` is a list of integers such that for each true value in `ccl_edges` stores the index of the edge. Let us examine the following code.

```
E.generate_CC_loops(vc,ec)
#There cannot be any loops in cover graph therefore all loop vertices must be covered.
#Recall all loops are already covered by construction of APEC
if(E.are_LVs_covered):
  if(len(E.CC_loops)>0):
    #Determine which edges are a part of CC_loops
    ccl_edges = [False]*configs["edges"]
    ccl_edge_locations =[]
    for bv in E.CC_loops:
      for e in range(configs["edges"]):
        if(bv[e]!=0 and not ccl_edges[e]):
        ccl_edges[e] = True
        ccl_edge_locations.append(e)
```

We are give an `EGraph` E from the list `All_EGraphs` and lists `vc` and `ec` from `APVC` and `APEC` respectively. First we call the `EGraph` function `generate_CC_loops` which was discussed above. Next we verify that all vertices with loop valencies are covered, if they are not all covered we will move on to the next combination of E, `vc` , and `ec` due to the arguments from section 11.1. If all loop vertices are covered we check if we have any completely covered cycles in $H_1(\Gamma, \mathbb{Z})$. If we have completely covered cycles we create the lists `ccl_edges` and `ccl_edge_locations` . In this construction we loop through all members of `CC_loops` and make all edges in completely covered cycles as `true` in `ccl_edges` . Also if an edge is a part of a completely covered cycle we add its index to the list `ccl_edge_locations` .

Finally we need to create the last essential list called `ccl_edges_isCrossed` . This will be a boolean list which will tell us whether or not to cross the corresponding edge in the potential cover graph incidence matrix. This construction will be as follows,

```
ccl_edges_iscrossed = [false]*len(ccl_edge_locations)
#make sure loops are always crossed
for i in range(len(ccl_edge_locations)):
  if(ccl_edge_locations[i]<configs["loops"]):
    #loops come first in ccl_edge_locations
    ccl_edges_iscrossed[i] = True
```

The list `ccl_edges_isCrossed` has the same length as `ccl_edge_locations` which is the number of edges which occure in a completely covered cycle. We initialize `ccl_edges_isCrossed` to have `true` values for each edge that corresponds to a loop, the edges will occur first in `ccl_edge_locations` becuase they occur first in `Homology_Basis` of E. The rest of the edges in `ccl_edge_locations` will be initialized to `false` . In constructing all possible cover incidence matrices we will iterate `ccl_edges_isCrossed` in the same way that we iterated `CV` and `CE` which we detailed in section 7.1. For each iteration of `ccl_edge_locations` we will create a possible cover graph and check for admissibility.

We are now ready to discuss the function `get_Possible_Cover_IM` in the context of E having completely covered cycles. This function will have 6 inputs, three more then the contex of no completely covered cycles, the three extra inputs will correspond to the lists `ccl_edges` , `ccl_edge_locations` , and `ccl_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.

```
def get_Possible_Cover_IM(E, CV, CE, ccl_edges, ccl_edges_iscrossed,
                          ccl_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(ccl_edges[e] and ccl_edges_iscrossed[ccl_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 first step 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 4.4. 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 7.2. 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 7.2.

Example 7.8. In order to better explain the `get_Possible_Cover_IM` function in the context of completely covered cycles we will consider the following base graph.

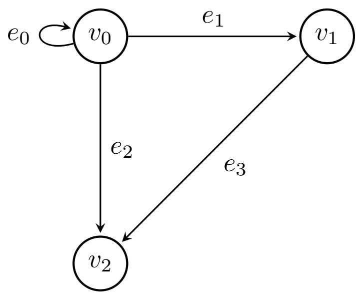

FIGURE 29. Base Graph G

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

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

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

$$\texttt{G.generate\_CC\_loops(CV,CE)}.$$

Notice that the homology basis for G is given by: `G.Homology_Basis` $= [(1,0,0,0),(0,1,1,1)]$. The function would find that both cycles are completely covered loops. That is `G.CC_loops` $= [(1,0,0,0),(0,1,1,1)]$. This information will tell us that

$$\texttt{ccl\_edges} = [\texttt{true}, \texttt{true}, \texttt{true}, \texttt{true}] \quad \text{and} \quad \texttt{ccl\_edge\_locations} = [0,\ 1,\ 2,\ 3].$$

Suppose we are working with the iteration of

$$\texttt{ccl\_edges\_iscrossed} = [\texttt{true}, \quad \texttt{true}, \quad \texttt{false}, \quad \texttt{false}].$$

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

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

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.

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

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 `ccl_edges_isCrossed` therefore we cover e_1 as follows.

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

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 correpond to `false` values in `ccl_edges_isCrossed.` After covering these final two edges we return the following `Possible_IM` which will then be tested for admissibility.

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

This possible incidence matrix will lead to the following possible covering graph of G.

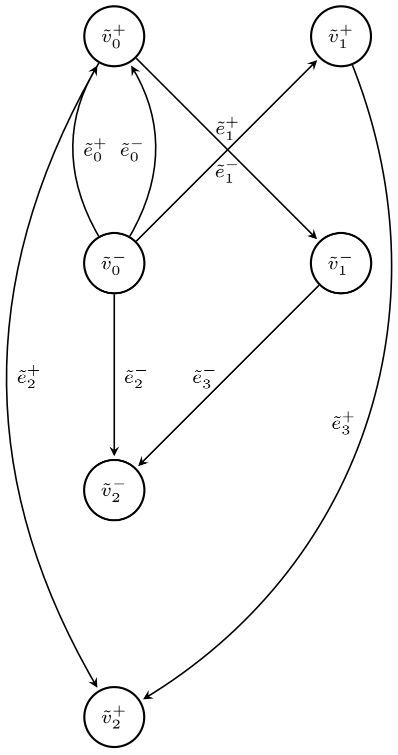

FIGURE 30. Possible Cover of G with `ccl_edges_isCrossed` = [`true`, `true`, `false`, `false`]

At this point in the program we have taken a base graph E from the list `All_EGraphs` , assigned 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 4.

7.4. **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. This is the function which takes the possible incidence matrices outputted from the `get_Possible_Cover_IM` functions and checks whether they satisfy the conditions required to be an admissible 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 `ccl_edge_locations` ; this is the same integer list described in section 7.3. The final input is `ccl_edges_isCrossed` ; this is the same boolean list as described in section 7.3. 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, ccl_edge_locations, ccl_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 cover criteria by construction
  return true
```

If you recall from section 4 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 `ccl_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 `ccl_edges_isCrossed` is a boolean list which tells whether only the edges which are a part of completely covered cycles lift to crossings – the length of this list is the number of edges that are a part of completely covered crossings.

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 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 $s(\tilde{e}^+)$ and $t(\tilde{e}^+)$. 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 addmissible.

Example 7.9. 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.

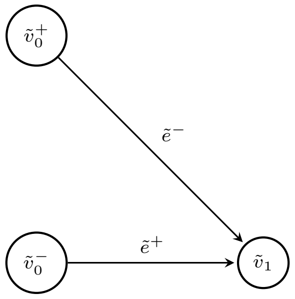

FIGURE 31. Only admissible cover of e in $\widetilde{\Gamma}$

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 base graph. The last part of this implementation we need to discuss is the isomorphism testing of the cover graphs.

SubSec: FS

7.5. **Friedman-Smith Testing.** It is helpful to detemine when we are dealing with Friedman-Smith cover of a base graph. [5]

5 ⟹

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 5.3. 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):
```

[5](Josh) Need a better explination here

```
      self.FS = True
    else:
      self.FS = False
```

7.6. **Cover graph isomorphism testing and summary.** Testing 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 then 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 7 that the entire program is scaffolded into two parts. One part generates the base graphs and the second part generates the covering information. We will now summerize 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 6).
(2) For each particular base graph we generate all possible combinations of `CE` and `CV` (see subsection 7.1).
(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 generate of list of completely covered cycles in $H^1(\Gamma, \mathbb{Z})$ using the class function `generate_CC_loops` (see subsection 7.3).
(4) Given the completly covered cycles for the base graph E with specific choices of `CV` and `CE` we generate a boolean list called `ccl_edges_isCrossed` . This boolean list will dictate which edges of the base graph E belonging to completely covered cycles will lift to crossing. Recall that only edges of completly covered loops will need to be considered for crossed liftings (see subsection 7.2).
(5) For a given base graph E, a boolean list `CV` , a boolean list `CE` , and a boolean list `ccl_edges_isCrossed` we generate a possible cover incidence matrix (see section 7.3).
(6) We check the possible cover incidence matrix for admissibility using the techniques from section 4. The implemention is in subsection ??.
(7) Finally, we first check if the cover graph is Friedman-Smith (see subsection 7.5, then we check the Friedman-Smith admissible cover graphs over a particular base graph E for isomorphims. Unique members of isomorphism classes will have their basis of linear forms (see section 5) outputted to a text file which will be later checked for types of quadratic-cone-compactifications.

Remark 7.10. 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 computation cost of the compile time was too high.

```
for E in All_EGraphs:
  All_CGS_For_G=[]
  for vc in APVC:
    for ec in APEC:
      E.generate_CC_loops(vc,ec)
      #There cannot be any loops in cover graph therefore all loop vertices
      #must be covered.
      #Recall all loops are already covered by construction of APEC
      if(E.are_LVs_covered):
        if(len(E.CC_loops)>0):
          #Determine which edges are a part of CC_loops
          ccl_edges = [False]*configs["edges"]
          ccl_edge_locations =[]
          for bv in E.CC_loops:
            for e in range(configs["edges"]):
```

```
      if(bv[e]!=0 and not ccl_edges[e]):
        ccl_edges[e] = True
        ccl_edge_locations.append(e)

  ccl_edges_iscrossed = [false]*len(ccl_edge_locations)
  #make sure loops are always crossed
  for i in range(len(ccl_edge_locations)):
    if(ccl_edge_locations[i]<configs["loops"]):
      #We are assuming that loops come first in ccl_edge_locations
      ccl_edges_iscrossed[i] = True
  while(True):
    Possible_IM = get_Possible_Cover_IM(E, vc,ec, ccl_edges,
                      ccl_edges_iscrossed, ccl_edge_locations)
    if(is_Cover(Possible_IM,vc,ec,ccl_edge_locations,ccl_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):
          All_CGS_For_G.append(CG)
          f.write(str(CG.Basis_Linear_Forms))
          f.write("\n\n")
          g.write(CG.output())

    if(not iterate_Boolean_List(ccl_edges_iscrossed)):
      break

#No Loops
else:
  Possible_IM = get_Possible_Cover_IM_no_loops(E,vc,ec)
  if(is_Cover_no_loops(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):
        All_CGS_For_G.append(CG)
        f.write(str(CG.Basis_Linear_Forms))
        f.write("\n\n")
        g.write(CG.output())
```

8. Example

We will compute a low-dimensional example to test the program and further understand how it works. For this example let us consider a base graph of 5 vertices and 6 edges given below.

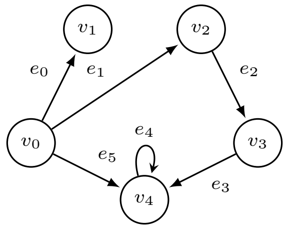

FIGURE 32. Base Graph G

Notice that the above base graph G has a loop at the vertex v_4. We can represent the information of the base graph G in the form of an incidence matrix as follows,

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

This graph will also have the is_loop array $[0, 0, 0, 0, 1, 0]$. With the information from the incidence matrix and the is_loop array we can create a graph class object. The graph class will compute the homology basis as such,

$$\text{Homology\_Basis} = \begin{pmatrix} 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & 1 & 1 & 0 & -1 \end{pmatrix}.$$

Clearly these correspond to the two loops in the graph created by e_4 and the loop created by e_1 to e_2 to e_3 to $-e_5$.

Now, we will choose an admissible cover for G. Let the following graph represent the admissible cover for G.

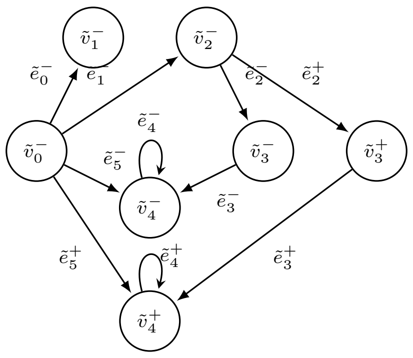

FIGURE 33. Admissible Cover $\tilde{G}$

This admissible cover will have CVerts $= [0, 0, 0, 1, 1,]$ and CEdges $= [0, 0, 1, 1, 1, 1]$. The Cover incidence matrix will be,

$$\text{Cover\_Incidence\_matrix} = \begin{pmatrix} -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 1 \end{pmatrix}.$$

This information is sufficient in giving us a Cover Graph Object. The resulting cover graph object will have Cover_Homology_Basis,

$$\text{Cover\_Homology\_Basis} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & -1 \end{pmatrix}.$$

From this we can calculate Image_CHB as,

$$\text{Image\_CHB} = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 & -1 & 1 & -1 & 0 & 0 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2 & -2 & 2 & -2 \end{pmatrix}.$$

Finally, we can compute the basis of linear forms as,

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

This implies for the dual graph of the cover $\tilde{G}$ we have a basis of two linear forms written in terms of the original edges, $z_1 = e_2 + e_3 - e_5$ and $z_2 = e_4 + e_5$.

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

Lemma 9.1. *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 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., $h = 2e$), 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 9.2. Recall that in regards to the indeterminacy locus of the Prym map ([CMGH+, Thm. 7.1]), the first open case is to determine the indeterminacy of the Prym map $P_4^P : \overline{R}_5 \dashrightarrow \bar{A}_4^P$. Thus, one must consider the case where there are up to 8 vertices and up to 12 edges.

I can still think of some ways to improve the algorithm (we can almost certainly use [CMGH$^+$, Appen. D] to avoid loops in the cover graph). But this still may be out of reach computationally.

Remark 9.3. 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 the first open case would be something like 7 edges and 8 vertices. This may be more feasible.

9.1. **Counting vertices and edges for dual graphs in $\overline{M}_{g,n}$.** For later reference, we also have:

Lemma 9.4. *Let $(C, p_1, \ldots, p_n)$ 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 Beti 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. ☐

10. 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 ([CMGH+, Thm. 7.1])

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

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.

First let us recall the dual graphs of Friedman–Smith covers:

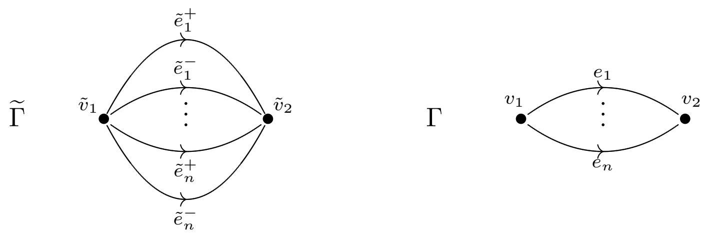

Fig:dgFSG

Figure 34. Dual graph of a Friedman–Smith example with $2n \geq 2$ nodes (FS_n).

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

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

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.

10.1. Enumeration in the special case $g = 4$.

As mentioned above, when considering the map

$$P_4^P : \overline{R}_5 \dashrightarrow \bar{A}_4^P$$

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).

Once we have done that, then as a first step, one can then just consider all admissible covers of those base graphs, (maybe with the given covering edges interchanged as given, but otherwise arbitrary covers).

Or, possibly, one can get more clever about enumerating the degenerate covers. But that should be a good start.

Also, one can ignore all loops in the cover graph $\widetilde{\Gamma}$. I will write out the argument later. This is essentially clear from [CMGH+, Appen. D]. This should also reduce the number of cases a great deal.

One can also ignore all graphs such that rank $H_1(\widetilde{\Gamma}, \mathbb{Z})^{[-]} = 1$. (Once one computes a basis for this space, if the basis consists of 1 element, one can simply throw away that graph.)

11. Some ideas to implement/prove to reduce the number of graphs

laceNumGraphs

11.1. **Loops in the cover graph.** We can almost certainly use [CMGH+, Appen. D] to avoid loops in the cover graph.

Sec: 1valent

11.2. **Tree branches in the base graph (one valent vertices in the base graph).** If there is a disconnecting node, and one of the disconnected graphs obtained from removing the node is a tree, then one can remove that tree from the base graph (and the cover curve). In other words, one can avoid base graphs with such subtrees.

A quick inductive algorithm is as follows. If there is a vertex in the base graph with only one edge (valency 1) one can remove that vertex and edge. (Iterating this will remove all the unneeded trees.) But since we can remove an edge and vertex, whatever we would have obtained would have been treated already in a case with fewer edges and vertices, so we can just ignore these graphs.

This can be quickly done at the level of matrices. If there is a row with all zeros, except for a single 1, or a single -1 (which is not indicating a loop–this would only be possible if there was only 1 vertex in the graph, since otherwise this vertex would be disconnected from the other vertices), then the graph can be ignored (assuming one has done all relevant graphs with fewer edges and vertices).

Sec: 2valent

11.3. **Two-valent vertices in the base graph.** ERROR.

I think we should be able to ignore base graphs with 2-valent vertices. The idea is the following. If the valency is coming from a loop, then the 2-valent vertex is the only vertex of the graph, and we know this case already. Otherwise, let's say the vertex in question is v and there are vertices v_ℓ and v_r with edges e_ℓ and e_r between v_ℓ and v, and v and v_r respectively.

We want to say that if we take the graph where we remove v and replace the edges e_ℓ and e_r with an edge e from v_ℓ to v_r, we get the same cone of quadratic forms.

This is pretty obvious if v is covered by $\tilde{v}^+$ and $\tilde{v}^-$. I think it is also OK in the other case, but this should be proven carefully.

NOT QUITE: MISTAKE.

Can rule out everything except the case where v_ℓ and v_r are are covered, but v just has one point covering it.

11.4. **Disconnecting vertices in the cover.** This is in appendix D [CMGH+, Appen. D]. This also happens to slightly reduce further the case where there is a valence two vertex in the base graph – but doesn't quite rule out the valence two graphs in the base.

References

[AB] V. Alexeev and A. Brunyate, *Extending the torelli map to toroidal compactifications of siegel space*, Inventiones Mathematicae.

[ABH] V. Alexeev, Ch. Birkenhake, and K. Hulek, *Degenerations of prym varieties*, Crelle Journal.

[CMGH+] Sebastian Casalaina-Martin, Samuel Grushevsky, Klaus Hulek, Radu Laza, and Mathieu Dutour Sikirić, *Extending the prym map to toroidal compactifications of the moduli space of abelian varieties*, Journal of European Mathematics Society (JEMS).

[Nam] Y. Namikawa, *A new compactification of the siegel space and degenerations of abelian varieties*, Annals of Mathematics.

[RS] R.Fiedman and R. C. Smith, *Degenerations of prym varieties and intersections of three quadrics*, Inventiones Mathematicae.

[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)

Josh Frinak, University of Colorado-Boulder, Boulder, CO 80303
E-mail address: Joshua.Frinak@Colorado.edu