2017

Classifying the Jacobian Groups of Adinkras

Aaron R. Bagheri
Harvey Mudd College

Classifying the Jacobian Groups of Adinkras

Aaron Bagheri

Dagan Karp, Advisor

Stefan Mendez-Diez, Reader

Department of Mathematics

May, 2017

Abstract

Supersymmetry is a theoretical model of particle physics that posits a symmetry between bosons and fermions. Supersymmetry proposes the existence of particles that we have not yet observed and through them, offers a more unified view of the universe. In the same way Feynman Diagrams represent Feynman Integrals describing subatomic particle behaviour, supersymmetry algebras can be represented by graphs called adinkras. In addition to being motivated by physics, these graphs are highly structured and mathematically interesting. No one has looked at the Jacobians of these graphs before, so we attempt to characterize them in this thesis. We compute Jacobians through the 11-cube, but do not discover any significant discernible patterns. We then dedicate the rest of our work to generalizing the notion of the Jacobian, specifically to be sensitive to edge directions. We conclude with a conjecture describing the form of the directed Jacobian of the directed n-topology. We hope for this work to be useful for theoretical particle physics and for graph theory in general.

Contents

List of Figures

Acknowledgments

I want to express my appreciation to Harvey Mudd College and the Department of Mathematics for supporting this thesis. In particular, I would like to thank Professor Dagan Karp for all of his help and direction as my advisor. Finally, I thank Professor Stefan Mendez-Diez for reading my thesis, Professor Lisette de Pillis for organizing the thesis class, and my thesis classmates for providing feedback on several occasions over the year.

Chapter 1

Adinkras

Adinkras are graphs that represent supersymmetry algebras. The physics behind these graphs will not be addressed in this thesis and can be found in Faux and Gates (2005). We devote this chapter to defining an adinkra. We follow the definitions of Zhang (2013).

Definition 1.1. An *n-dimensional adinkra topology* is a finite connected simple graph that is bipartite and n-regular.

Since we will be working only with adinkra topologies, we will henceforth simply refer to them as n-topologies.

Definition 1.2. We call a colored graph *quadrilateral* if for any distinct i, j, the edges with colors i and j form a disjoint union of 4-cycles.

Definition 1.3. An *n-chromotopology* is an n-topology with a quadrilateral coloring of n colors assigned to it such that every vertex is incident to exactly 1 edge of each color.

Now to define an adinkra, we need two more properties.

Definition 1.4. We will call a quadrilateral graph *odd-dashed* if its edges are dashed in such a way that every 4-cycle contains an odd number of dashed edges.

Lastly, we would like to define a height assignment, but need another definition first.

Definition 1.5. A *Hasse diagram*, as defined by Weisstein (b), is a graphical representation of a partially ordered set, or poset that illustrates the covering

relationship between elements of the set. An element z of a poset *covers* another element x if there exists no third element y in the poset for which $x \leq y \leq z$. Weisstein (a). A point is drawn for each element of the poset, and line segments are drawn between these points according to the following two rules:

(a) If $x < y$ in the poset, then the point corresponding to x appears lower in the drawing than the point corresponding to y.

(b) The line segment between the points corresponding to any two elements x and y of the poset is included in the drawing if and only if x covers y or y covers x.

For example, the Hasse diagram for the power set of a two element set, $\{a, b\}$ with $R \leq S$ if $R \subseteq S$ would be the following.

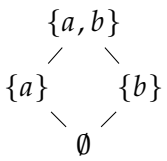


Figure 1.1 A simple Hasse diagram of the power set of a two element set.

Definition 1.6. We give a graph a *height assignment* by identifying it with the Hasse diagram of cube vertices. Each vertex of an n-cube can be defined by an n-digit binary string, and for two strings s and t, we have $s \leq t$ if s has a 1 in the same positions as t. For a 3-cube, this looks like the following.

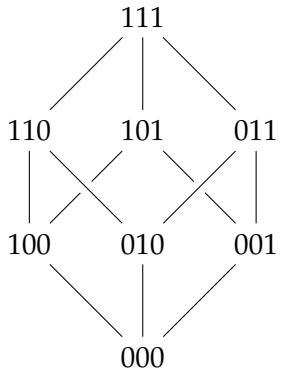


Figure 1.2 A Hasse diagram for the vertices of a 3-cube.

It is intuitive to draw the heights assigned to each vertex by placing them physically higher on the page.

Definition 1.7. We finally define an *n-adinkra* as a quadrilateral dashed n-chromotopology with a height assignment.

All of these properties can be seen in the following examples of 2,3, and 4-adinkras, produced by Adinkramat (2012).

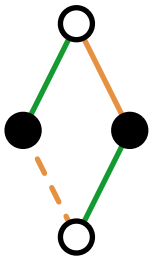


Figure 1.3 A 2-adinkra.

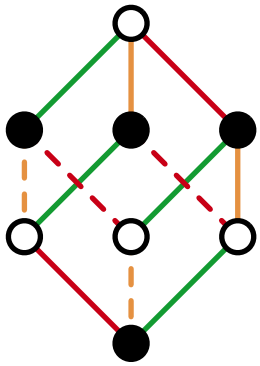


Figure 1.4 A 3-adinkra.

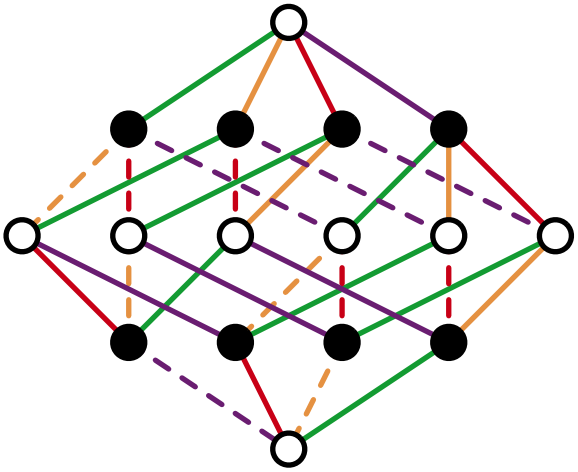


Figure 1.5 A 4-adinkra.

Note that, in general, we have n, k-adinkras. The graphs we have defined as n-adinkras are equivalent to what would more generally be known as $n, 0$-adinkras. We proceed with our definition of the n-adinkra because it is simpler than that of the general n, k-adinkra and because we limit our investigations in this thesis to the case where $k = 0$. In this case, the topology of the n-adinkra is that of an n-cube. For the more general case of the n, k-adinkra, consult Zhang (2013).

Chapter 2

The Jacobian Group of a Graph

We now define the Jacobian group of a graph and present several useful theorems discussed in Hoppenfeld (2014).

2.1 Defining the Jacobian

To begin defining the Jacobian of a graph, consider a finite graph topology G with labeled vertices, as below.

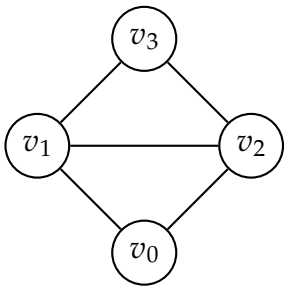


Figure 2.1 A graph, G, with labeled vertices.

On our graph G, we can fire chips as follows. An integer chip value is assigned to each vertex. For example, we could assign chip values to the vertices of G as follows.

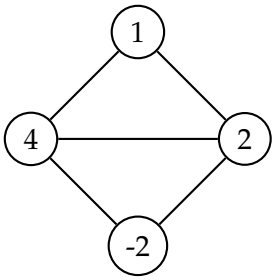


Figure 2.2 The graph G with integers assigned to each vertex.

With each firing of chips, a single vertex either takes or fires 1 chip along every edge it is incident to. For example, legal moves starting from G could be as follows.

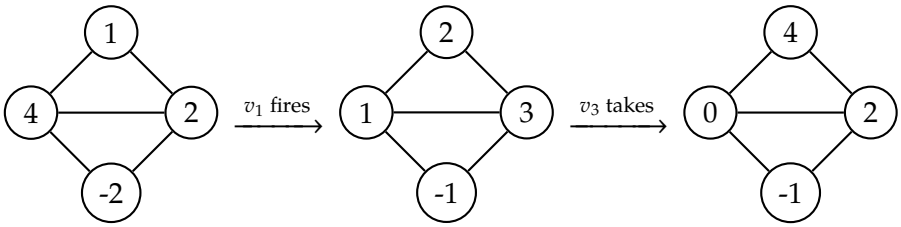


Figure 2.3 The graph G undergoing two chip firing moves.

Definition 2.1. Every assignment of numbers is called a *divisor* of the graph, and is represented

$$D = \sum_{v \in V(G)} a_v v,$$

where $V(G)$ is the vertex set of a graph G and a_v is the number assigned to vertex v.

Thus, the first divisor in Figure 2.3 might be referred to as $D = -2v_0 + 4v_1 + 2v_2 + v_3$. Note that "+" is being used here as a notational tool rather than regular addition.

Definition 2.2. The *degree* of a divisor, D, is

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

Definition 2.3. A *zero divisor* of a graph is a divisor, D, with

$$\deg(D) = 0.$$

Definition 2.4. We define an equivalence class on divisors with $D \sim D'$ if D' can be obtained from D through some chip firing sequence. We write $[D]$ for the equivalence class of the divisor D and $D' \in [D]$.

We can verify that this is indeed an equivalence relation:

- $D \sim D$ because $D = D$ with 0 moves.
- If $D \sim D'$, reverse move order to get $D' \sim D$.
- If $D \sim D'$ and $D' \sim D''$, concatenate sequences of moves to get $D \sim D''$.

Note: all divisors in an equivalence class have the same degree because every firing of chips preserves degree.

As an example, writing out the divisors of the three graphs above shows us that the divisors

$$D = v_1 + 2v_2 - 2v_3 + 4v_4,$$
$$D' = 2v_1 + 3v_2 - v_3 + v_4,$$
$$D'' = 4v_1 + 2v_2 - v_3$$

of G are all equivalent.

Definition 2.5. The collection of all divisors on a graph G is

$$\mathrm{Div}(G) = \left\{ \sum_{v \in V(G)} a_v v : v \in V \right\}.$$

This collection of divisors forms a group under componentwise addition, i.e. $\sum a_v v + \sum b_v v = \sum (a_v + b_v) v$. We can quickly verify group properties:

- **Closure:** Let $A = \sum_{v \in V(G)} a_v v$ and $B = \sum_{v \in V(G)} b_v v$ be divisors of G. We see that the sum

  $$A + B = \sum_{v \in V(G)} (a_v + b_v) v \in \mathrm{Div}(G)$$

  because the integers are closed under addition.

- **Associativity:** Given divisors $A = \sum a_v v$, $B = \sum b_v v$, and $C = \sum c_v v$, we see that the sums $A+(B+C)$ and $(A+B)+C$ are equal by associativity of integer addition:

  $$\sum_{v \in V(G)} [a_v + (b_v + c_v)]\, v = \sum_{v \in V(G)} [(a_v + b_v) + c_v]\, v.$$

- **Identity:** We have an identity element $0 = \sum 0v$ since

$$A + 0 = \sum_{v \in V(G)} (a_v + 0)v = \sum_{v \in V(G)} (0 + a_v)v = \sum_{v \in V(G)} a_v v = A.$$

- **Inverses:** For a divisor $A = \sum a_v v$, we have an inverse divisor $-A = \sum -a_v v$:

$$A + (-A) = \sum_{v \in V(G)} (a_v - a_v)v = 0.$$

Definition 2.6. The collection of all zero divisors on a graph G is

$$\mathrm{Div}^0(G) = \left\{ \sum_{v \in V(G)} a_v v : v \in V, \sum_{v \in V(G)} a_v = 0 \right\}.$$

We can verify that $\mathrm{Div}^0(G) \leq \mathrm{Div}(G)$:

- **Closure:** We verify that the sum of two zero divisors, $A = \sum a_v v$, $B = \sum b_v v$, is also a zero divisor by looking at the degree of the sum:

$$\deg(A + B) = \sum_{v \in V(G)} a_v + b_v = \sum_{v \in V(G)} a_v + \sum_{v \in V(G)} b_v = 0 + 0 = 0.$$

- **Inverses:** Taking the same inverse element as before, we see that the inverse of a zero divisor, $A = \sum a_v v$, also has degree zero:

$$\deg(-A) = \sum_{v \in V(G)} -a_v = -\sum_{v \in V(G)} a_v = 0,$$

as desired.

Definition 2.7. For a graph G, the collection of zero divisors under the equivalence class defined above is called the *Jacobian group*:

$$\mathrm{Jac}(G) = \mathrm{Div}^0(G)/\sim$$

or

$$\mathrm{Jac}(G) = \left\{ [D] : D \in \mathrm{Div}^0(G) \right\}.$$

This set is also commonly known as the sandpile group of G.

Again, we can verify that the Jacobian group is indeed a group, where addition is componentwise as before, i.e. $[\sum a_v v] + [\sum b_v v] = [\sum (a_v + b_v)v]$. This follows closely the verification that $\mathrm{Div}(G)$ is a group.

- **Closure:** Let $A = [\sum_{v \in V(G)} a_v v]$ and $B = [\sum_{v \in V(G)} b_v v]$ be equivalence classes of divisors of G. We see that the sum

$$A + B = \left[\sum_{v \in V(G)} (a_v + b_v)v \right] \in \mathrm{Jac}(G)$$

  because integers are closed under addition and we get the equivalence class of another divisor.

- **Associativity:** Given divisor classes $A = [\sum a_v v]$, $B = [\sum b_v v]$, and $C = [\sum c_v v]$, we see that the sums $A + (B + C)$ and $(A + B) + C$ are equal by associativity of integer addition:

$$\left[\sum_{v \in V(G)} [a_v + (b_v + c_v)]\, v \right] = \left[\sum_{v \in V(G)} [(a_v + b_v) + c_v]\, v \right].$$

- **Identity:** We have an identity element $0 = [\sum 0v]$ since

$$A + 0 = \left[\sum_{v \in V(G)} (a_v + 0)v \right] = \left[\sum_{v \in V(G)} (0 + a_v)v \right] = \left[\sum_{v \in V(G)} a_v v \right] = A.$$

- **Inverses:** For a divisor $A = [\sum a_v v]$, we have an inverse divisor $-A = [\sum -a_v v]$:

$$A + (-A) = \left[\sum_{v \in V(G)} (a_v - a_v)v \right] = 0.$$

2.2 Finding a Graph Jacobian

Now that we have defined the Jacobian, we would like to find some. However, doing so from the definition is nontrivial. To demonstrate, we will attempt to find the Jacobian group of K_3, the complete graph on 3 vertices. To do so, we first draw and label K_3.

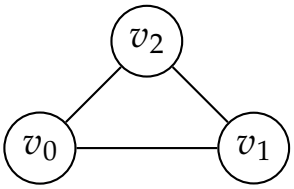


Figure 2.4 K_3, the complete graph on three vertices, with labeled vertices.

We can assign integers to this graph. Let us begin by looking at some divisors equivalent to 0.

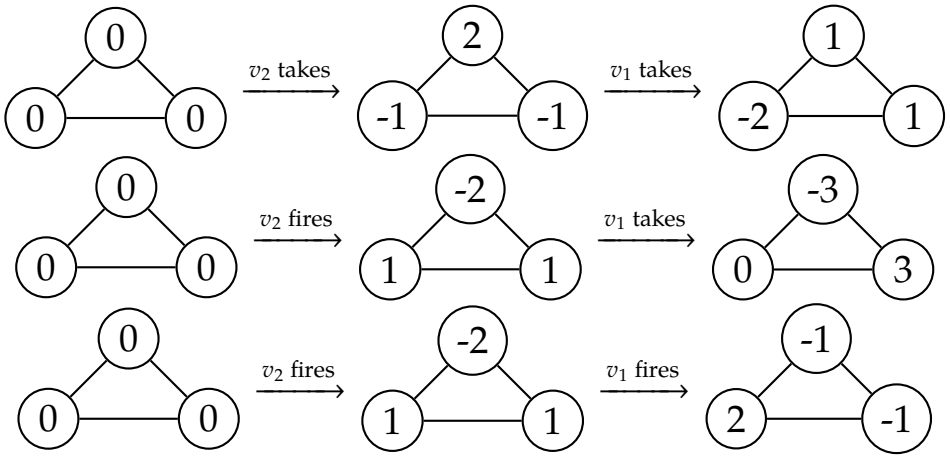


Figure 2.5 A sampling of divisors equivalent to 0 in K_3.

It does not seem like there is an easy way to get the divisor $v_0 - v_1$, so let us propose that as being in a different equivalent class and find divisors equivalent to it.

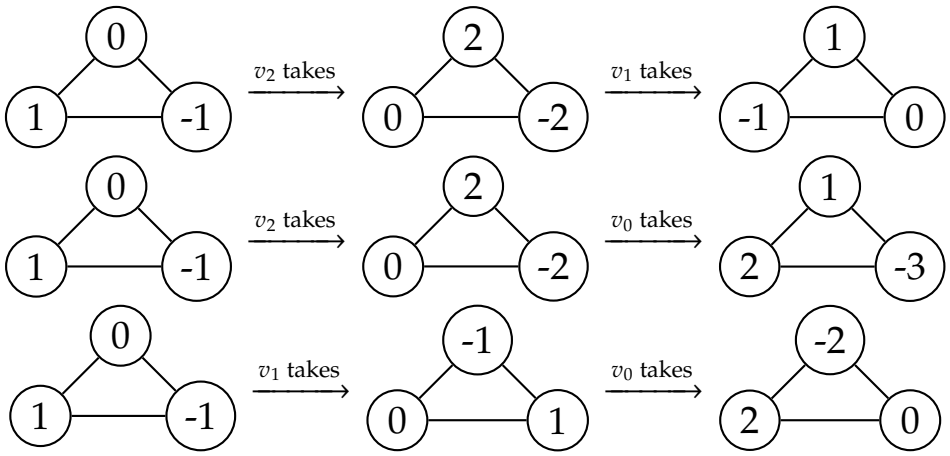


Figure 2.6 A sampling of divisors equivalent to $v_0 - v_1$ in K_3.

We see here that we can permute the vertices so that the ±1 rotate around the graph, but we cannot switch them. That is, we were able to find divisors $v_0 - v_1$, $v_1 - v_2$, and $v_2 - v_0$, but not divisors $-v_0 + v_1$, $-v_1 + v_2$, and $-v_2 + v_0$. Perhaps these live in their own equivalence class.

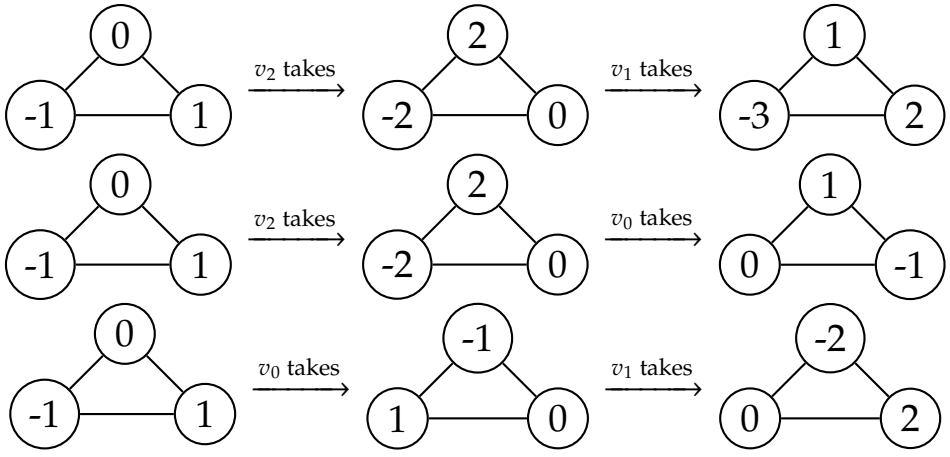


Figure 2.7 A sampling of divisors equivalent to $-v_0 + v_1$ in K_3.

We have only taken in the last two samplings, but one can check that giving will reach the same divisors as sequences of taking. It does seem from these examples that $[v_0 - v_1]$ is a different (and opposite) equivalence class from $[-v_0 + v_1]$, but this would require proof. It turns out that $\mathrm{Jac}(K_3) \cong \mathbb{Z}/3\mathbb{Z}$, the group of order 3, so we are done, but this is unclear

from the equivalence classes and would require further study. This example illustrates the need for an easier way to find the Jacobian group of a graph. A very helpful theorem will give us this (and could be used to verify that $\text{Jac}(K_3) = \mathbb{Z}/3\mathbb{Z}$).

2.3 An Easier Approach to Jacobians

In this section, we present a theorem that will be incredibly useful in our computation of Jacobians. Before we can though, we need a few more definitions.

Definition 2.8. The *adjacency matrix* of a graph G on n vertices is the $n \times n$ matrix whose i, jth entry is 1 if there is an edge from vertex i to vertex j and 0 otherwise.

Notice that the adjacency matrices for our undirected graphs will be symmetric since every edge is bidirectional. This will not be true later when we consider Jacobians of directed graphs.

Definition 2.9. The *Laplacian* of a graph G on n vertices is defined to be an $n \times n$ matrix

$$\Delta = D - A,$$

where D is the diagonal matrix with the degrees of the vertices along the diagonal and A is the adjacency matrix of G.

Definition 2.10. The *reduced Laplacian*, denoted $\tilde{\Delta}$, with respect to some vertex v of a graph G on n vertices is the minor of G with respect to v. That is, $\tilde{\Delta}$ is the $n-1 \times n-1$ matrix obtained by removing the row and column of the Laplacian matrix that correspond to vertex v.

We can now present a useful theorem:

Theorem 2.11. *For any graph G on n vertices, we have*

$$\text{Jac}(G) \cong \mathbb{Z}^{n-1}/\text{im}\left(\tilde{\Delta}\right).$$

To use this theorem, it is convenient to compute the invariant factors of $\mathbb{Z}^{n-1}/\text{im}\left(\tilde{\Delta}\right)$ by finding the Smith-Normal form of $\tilde{\Delta}$.

Definition 2.12. We say that an $m \times n$ integer matrix is in *Smith-Normal form* if it is a diagonal matrix with diagonal entries $a_1, a_2, \ldots, a_n$ and if $a_i \mid a_{i+1}$ for all i.

As discussed in Hoppenfeld (2014), any integer matrix can be reduced to Smith-Normal form through a series of the following row and column operations:

(a) Swapping any two rows (or any two columns)

(b) Negating any row (or any column)

(c) Adding any row (or any column) to another.

Though not directly relevant to our work, something that will prove useful to us is the relationship between the Jacobian group and the spanning trees of a graph. From Theorem 2.11 and Kirchhoff's matrix tree theorem, we get the following theorem.

Theorem 2.13. *For a graph G, $|\mathrm{Jac}(G)|$ = the number of spanning trees of G.*

We will keep this fact in mind going forward to inform our generalization of the Jacobian.

Chapter 3

Jacobians of Adinkras

Now that we have seen both adinkras and Jacobians, we may begin finding Jacobians of adinkras. Notice that our definition of the Jacobian does not consider coloring, dashing, or height-assignment. A generalization of the notion of the Jacobian is necessary in order to find the Jacobian of an adinkra. As a first step, we begin our discussion with Jacobians of adinkra topologies.

Let us begin with an example of a 2-topology, as below.

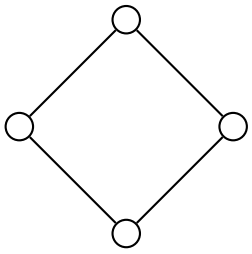
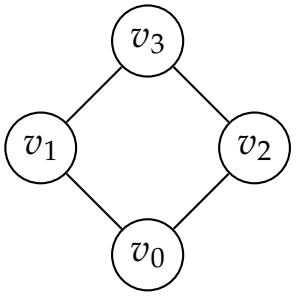


Figure 3.1 A 2-topology, simply a diamond.

We recall Theorem 2.11 from our discussion of Jacobians that for a graph G on n vertices,

$$\mathrm{Jac}(G) \cong \mathbb{Z}^{n-1}/\mathrm{im}(\tilde{\Delta}),$$

where $\tilde{\Delta}$ is the reduced Laplacian of G, and $\Delta = D - A$, the adjacency matrix, A, subtracted from the digonal matrix of vertex degrees, D. As discussed earlier, our approach for finding the Jacobian of the 2-topology will be to find the reduced Laplacian, reduce it to Smith-Normal form, and use the Theorem 2.11. To do this, we first number the vertices and find the adjacency matrix:



$$A = \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix}.$$

Figure 3.2 A labeled 2-topology with its adjacency matrix.

Note that the 2-topology is 2-regular, so our diagonal matrix is simply

$$D = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix},$$

and our Laplacian is

$$\begin{aligned} \Delta &= D - A \\ &= \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{bmatrix} - \begin{bmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{bmatrix} \\ &= \begin{bmatrix} 2 & -1 & -1 & 0 \\ -1 & 2 & 0 & -1 \\ -1 & 0 & 2 & -1 \\ 0 & -1 & -1 & 2 \end{bmatrix}. \end{aligned}$$

Removing the last row and column gives us the reduced Laplacian

$$\tilde{\Delta} = \begin{bmatrix} 2 & -1 & -1 \\ -1 & 2 & 0 \\ -1 & 0 & 2 \end{bmatrix}.$$

Reduced to Smith-Normal form, we are left with

$$\tilde{\Delta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 4 \end{bmatrix}.$$

We find an image

$$\text{im}\left(\tilde{\Delta}\right) = \left\{ \begin{bmatrix} a \\ b \\ 4c \end{bmatrix} : a, b, c \in \mathbb{Z} \right\} = \mathbb{Z} \times \mathbb{Z} \times 4\mathbb{Z},$$

and a Jacobian

$$\mathbb{Z}^{4-1} / \left(\mathbb{Z} \times \mathbb{Z} \times 4\mathbb{Z}\right) \cong \mathbb{Z}/4\mathbb{Z}.$$

Thus, the Jacobian group of the 2-topology is isomorphic to $\mathbb{Z}/4\mathbb{Z}$.

While this process is not difficult, it is tedious since an n-topology contains 2^n vertices. We have, therefore, written a computer algorithm to automate it. Our code can be found in Appendix A. Given an integer n, we return the entries along the diagonal of the Smith-normal form of the reduced Laplacian for the n-cube. Our algorithm for doing so comes from Hoppenfeld (2014) and consists of

(a) computing the adjacency matrix for the n-cube, which requires:

- enumerating binary strings of length n. Each string with k 1's represents a vertex in the k-th level of our height assignment.
- creating a $2^n \times 2^n$ matrix of 0's.
- for each vertex i and each vertex j in a higher level than i, changing the i, j-th entry to a 1 if the binary string associated with vertex j has 1's in the same locations as does the binary string associated with vertex i.
- reflecting the resulting upper triangular matrix across the diagonal, leaving the diagonal with 0's.

(b) computing the Laplacian of the n-cube by subtracting the above adjacency matrix from a diagonal matrix with n's on the diagonal.

(c) acquiring the reduced Laplacian by removing the last row and column of the Laplacian.

(d) computing the Smith-normal form of the reduced Laplacian by

- swapping rows and columns so that the entry with the smallest nonzero absolute value is in the 1,1 position.

- adding multiples of the first row/column to each other row/column to reduce the entire first row and column (except for the 1,1 entry) as far as possible. After each addition, we check to see if there are new smallest nonzero absolute values in the matrix. If so, we move them to the 1,1 position. At the end of this process, the entire first row and column are 0 except for the 1,1 entry.
- repeating this process, treating each entry on the diagonal as the 1,1 entry of the smaller matrix acquired by removing the rows and columns that have already undergone the process.
- swapping rows and columns to rearrange the diagonal if it is not sorted by divisibility.

While the Smith-normal form algorithm above works, Stein et al. (YYYY) has a more efficient Smith-normal form function that we have used instead. Our original function can be found in Appendix B. The first few Smith-normal form diagonals we have found are

$$
\begin{aligned}
D_2 &= [1,1,4], \text{ as expected,}\\
D_3 &= [1,1,1,1,2,8,24],\\
D_4 &= [(1,8),(2,2),(8,1),(24,3),(96,1)],\\
D_5 &= [(1,16),(2,5),(6,1),(24,4),(48,1),(192,3),(960,1)],\\
D_6 &= [(1,32),(2,10),(6,2),(12,4),(24,1),(96,4),(192,4),(960,6)],\\
D_7 &= [(1,64),(2,21),(6,7),(12,1),(48,8),(96,5),(480,1),(960,14),(1920,5),\\
&\qquad (13440,1)],\\
D_8 &= [(1,128),(2,43),(6,13),(12,2),(48,13),(240,3),(480,12),(960,28),\\
&\qquad (1920,5),(13440,7),(107520,1)],\\
D_9 &= [(1,256),(2,86),(6,34),(12,9),(60,1),(240,16),(480,26),(960,47),\\
&\qquad (6720,1),(13440,26),(53760,1),(215040,7),(645120,1)],\\
D_{10} &= [(1,512),(2,171),(6,69),(12,18),(60,18),(120,26),(480,16),(960,73),\\
&\qquad (6720,75),(26880,1),(107520,26),(215040,8),(645120,10)],\\
D_{11} &= [(1,1024),(2,341),(6,155),(12,54),(60,12),(120,32),(240,99),\\
&\qquad (1680,1),(6720,164),(13440,1),(53760,100),(215040,9),(645120,54),\\
&\qquad (7096320,1)].
\end{aligned}
$$

where each tuple gives the diagonal entry followed by the number of times it appears, e.g. D_4 is the diagonal matrix with diagonal

$[1,1,1,1,1,1,1,1,2,2,8,24,24,24,96]$. These diagonals give us Jacobians of

$$J_2 \cong \mathbb{Z}^3/(\mathbb{Z}\times\mathbb{Z}\times 4\mathbb{Z}) \cong \mathbb{Z}/4\mathbb{Z}$$
$$J_3 \cong \mathbb{Z}^7/\left(\mathbb{Z}^4\times 2\mathbb{Z}\times 8\mathbb{Z}\times 24\mathbb{Z}\right) \cong (\mathbb{Z}/2\mathbb{Z})\times(\mathbb{Z}/8\mathbb{Z})\times(\mathbb{Z}/24\mathbb{Z})$$
$$\vdots$$

To check these Jacobians, we can find the order of each group:

$|J_2| = |\mathbb{Z}/4\mathbb{Z}| = 4;$

$|J_3| = |(\mathbb{Z}/2\mathbb{Z})\times(\mathbb{Z}/8\mathbb{Z})\times(\mathbb{Z}/24\mathbb{Z})| = 384;$

$|J_4| = 2^2\cdot 8\cdot 24^3\cdot 96 = 42,467,328;$

$|J_5| = 2^5\cdot 6\cdot 24^4\cdot 48\cdot 192^3\cdot 960 = 20,776,019,874,734,407,680;$

$|J_6| = 2^{10}\cdot 6^2\cdot 12^4\cdot 24\cdot 96^4\cdot 192^4\cdot 960^6 =$
$1,657,509,127,047,778,993,870,601,546,036,901,052,416,000,000;$

$|J_7| = 2^{21}\cdot 6^7\cdot 12\cdot 48^8\cdot 96^5\cdot 480\cdot 960^{14}\cdot 1920^5\cdot 13440 =$
$153,850,844,349,814,660,487,100,539,994,381,178,281,567,942,393,055,$
$761,257,560,677,644,718,869,248,475,136,000,000,000,000,000,000,000;$

$|J_8| = 2^{43}\cdot 6^{13}\cdot 12^2\cdot 48^{13}\cdot 240^3\cdot 480^{12}\cdot 960^{28}\cdot 1920^5\cdot 13440^7\cdot 107520 =$
$17,404,759,458,462,474,728,830,233,927,402,010,830,990,825,809,132,$
$498,874,513,212,082,249,039,756,858,999,576,329,081,673,343,128,923,$
$696,522,065,077,911,775,269,987,383,478,545,223,764,454,192,971,651,$
$307,006,223,974,400,000,000,000,000,000,000,000,000,000,000,000,000,$
$000,000,000,000,000,000;$

$|J_9| = 2^{86}\cdot 6^{34}\cdot 12^9\cdot 60\cdot 240^{16}\cdot 480^{26}\cdot 960^{47}\cdot 6720\cdot 13440^{26}\cdot 53760\cdot 215040^7\cdot$
$645120;$

$|J_{10}| = 2^{171}\cdot 6^{69}\cdot 12^{18}\cdot 60^{18}\cdot 120^{26}\cdot 480^{16}\cdot 960^{73}\cdot 6720^{75}\cdot 26880\cdot 107520^{26}\cdot$
$215040^8\cdot 645120^{10};$

$|J_{11}| = 2^{341}\cdot 6^{155}\cdot 12^{54}\cdot 60^{12}\cdot 120^{32}\cdot 240^{99}\cdot 1680\cdot 6720^{164}\cdot 13440\cdot 53760^{100}\cdot$
$215040^9\cdot 645120^{54}\cdot 7096320.$

This sequence matches the number of spanning trees of an n-cube (Sloane and Knuth (1995)), which is encouraging since this is exactly what we expect from the order of the Jacobian. While this is already a well known sequence, the Jacobians are not, and there seems to be very little discernible pattern in

the ones we have found. The general form of the Jacobian of an adinkra is an open problem.

Chapter 4

Generalizing the Jacobian

4.1 Directed Topologies

We now generalize the notion of the Jacobian group of a graph to be sensitive to graph properties other than topology. The first property we address is the adinkra height assignment. Assigning heights to an adinkra is equivalent to removing height considerations and assigning each edge to be directed upward. That is, we can draw the same graph in the following two ways:

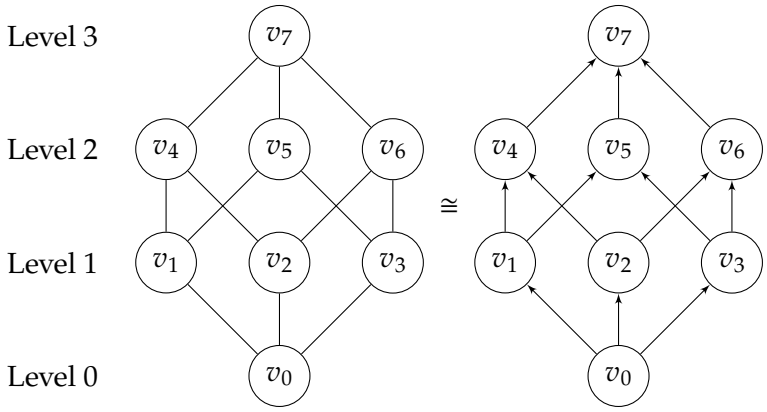


Figure 4.1 An $n = 3$ adinkra topology. Assigning heights is equivalent to directing all edges upward.

Since edge direction is more general and has been investigated to a greater degree than level assignment, we begin our investigations by defining a directed Jacobian. While doing so, it will be helpful to continue to visualize

the graph with levels. We can continue to think of a vertex as being in level n if a path to them from the root consists of n edges.

Back in Definition 2.7, we defined the Jacobian of an undirected graph to be the group of zero divisors under equivalence defined by chip firing sequences. We define the directed Jacobian in the same way as we did the undirected Jacobian, but we alter the chip firing scheme so that chips are taken and fired only along outgoing edges. This convention comes from Gaslowitz (2013) and appears to be standard. For example, we have equivalent divisors of the directed 2-topology:

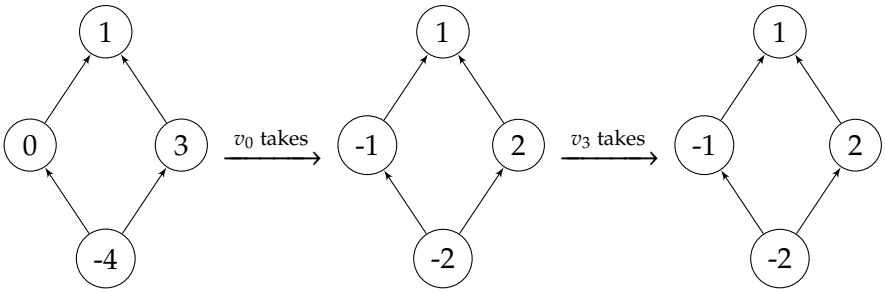


Figure 4.2 An $n = 2$ directed adinkra topology undergoing two moves of this new chip firing scheme.

We should verify that this newly defined object is still a group under the same addition as before. We can appeal to the same arguments as with the undirected Jacobian because divisors do not depend on edge directions. The following arguments are the same as before, but the equivalence classes are different.

- **Closure:** Let $A = [\sum_{v \in V(G)} a_v v]$ and $B = [\sum_{v \in V(G)} b_v v]$ be equivalence classes of divisors of G. We see that the sum

$$A + B = \left[\sum_{v \in V(G)} (a_v + b_v) v \right] \in \mathrm{Jac}(G)$$

  because integers are closed under addition and we get the equivalence class of another divisor.

- **Associativity:** Given divisor classes $A = [\sum a_v v]$, $B = [\sum b_v v]$, and $C = [\sum c_v v]$, we see that the sums $A + (B + C)$ and $(A + B) + C$ are

equal by associativity of integer addition:

$$\left[\sum_{v\in V(G)} [a_v + (b_v + c_v)]\, v\right] = \left[\sum_{v\in V(G)} [(a_v + b_v) + c_v]\, v\right].$$

- **Identity:** We have an identity element $0 = [\sum 0v]$ since

$$A + 0 = \left[\sum_{v\in V(G)} (a_v + 0)v\right] = \left[\sum_{v\in V(G)} (0 + a_v)v\right] = \left[\sum_{v\in V(G)} a_v v\right] = A.$$

- **Inverses:** For a divisor $A = [\sum a_v v]$, we have an inverse divisor $-A = [\sum -a_v v]$:

$$A + (-A) = \left[\sum_{v\in V(G)} (a_v - a_v)v\right] = 0.$$

With group structure verified, we now ask what directed Jacobians look like if defined this way and whether they seem reasonable. We begin by considering the $n = 2$ directed adinkra topology through some examples:

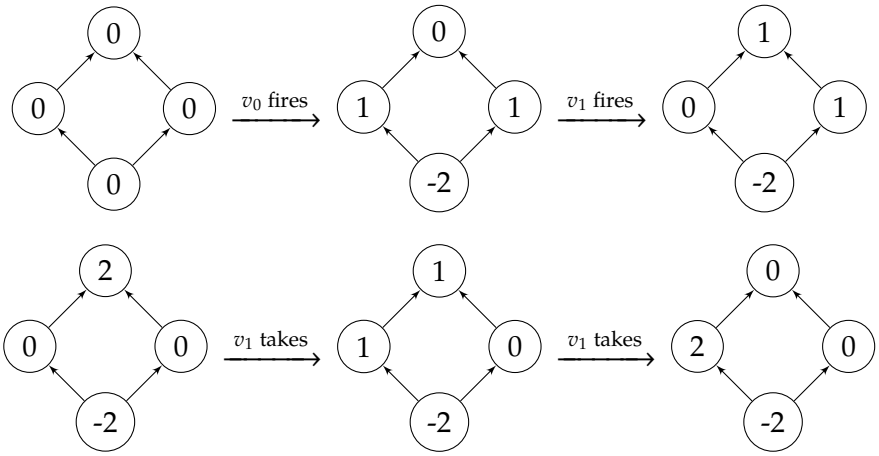


Figure 4.3 An $n = 2$ directed adinkra topology with an even number of chips at its root undergoing chip-firing moves along outgoing (upward) edges only.

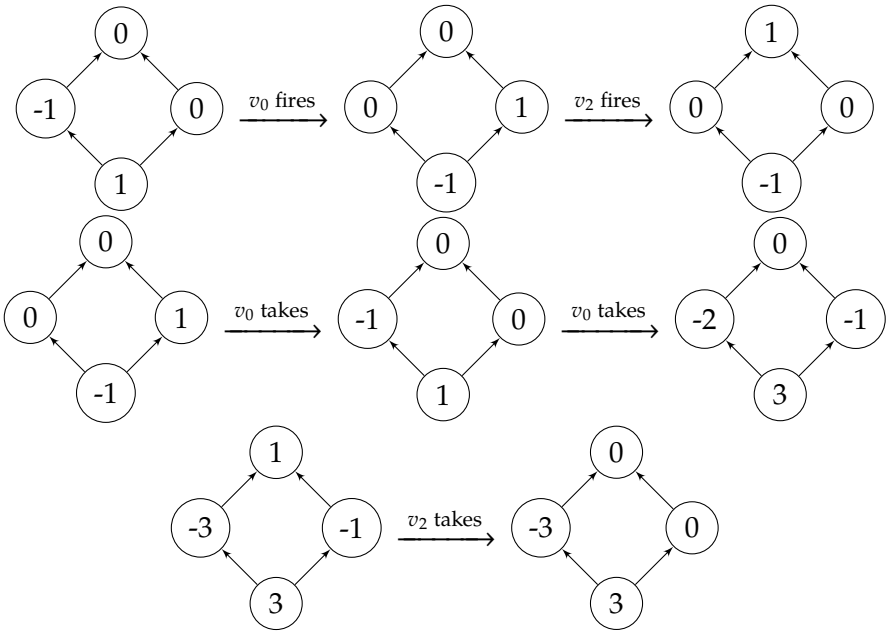
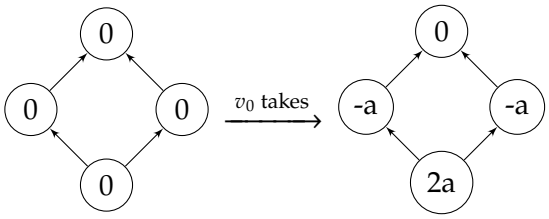


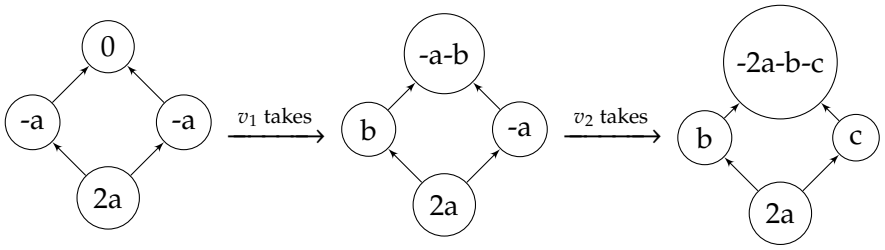
Figure 4.4 An $n = 2$ directed adinkra topology with an odd number of chips at its root undergoing chip-firing moves along outgoing (upward) edges only.

Conjecture 4.1. *All zero divisors with the same parity at the root are equivalent for the $n = 2$ directed adinkra topology.*

Proof. To prove our conjecture for the $n = 2$ directed adinkra topology, we first wish to show that on the directed 2-topology, we can get from divisor $D = 0$ to any divisor $D' = 2av_0 + bv_1 + cv_2 + (-2a - b - c)v_3$, for $a, b, c \in \mathbb{Z}$. Notice that we must have $d = -2a - b - c$ because the degree of the divisor must be 0. We begin by having the root vertex, v_0, take chips a times:



We now have vertex 1 take $a + b$ chips and vertex 3 take $a + c$ chips:



We have now constructed any arbitrary zero divisor with an even degree root. Repeating this argument with an odd degree root gives us any zero divisor with an odd degree root. Thus, all zero divisors with the same parity at the root are equivalent. □

Now that we know the unique zero divisors for the directed 2-topology, we can talk about the group structure. If we let $0 = [0v_0 + 0v_1 + 0v_2 + 0v_3]$ and $1 = [1v_0 + 0v_1 + 0v_2 - 1v_3]$, we see that the Jacobian under componentwise addition of graph divisors is isomorphic to $\mathbb{Z}/2\mathbb{Z}$.

4.2 Arborescences

We would like to conduct a similar analysis for a general graph, but this is a nontrivial task. In order to learn more about this directed Jacobian, we recall what is known about the regular Jacobian: the order of the Jacobian group of a graph is equal to the number of spanning trees of the graph. We continue, then, by considering the relationship between the directed Jacobian and the number of directed spanning trees.

Definition 4.2. An *arborescence* of a directed graph G is a subgraph, H, of G such that for some root vertex $u \in V(H)$ and any vertex $v \in V(H)$, there is exactly one directed path uv.

Definition 4.3. A *spanning arborescence* of a graph G is an arborescence of G that includes every vertex of G.

As the order of the Jacobian is equal to the number of spanning trees of a graph, so too, we hope, is the order of the directed Jacobian equal to the number of spanning arborescences. We can check this for the $n = 2$ case we just considered. Given that an adinkra always begins from a single lowest level vertex, our arborescences are limited to trees that begin there, and we have two:

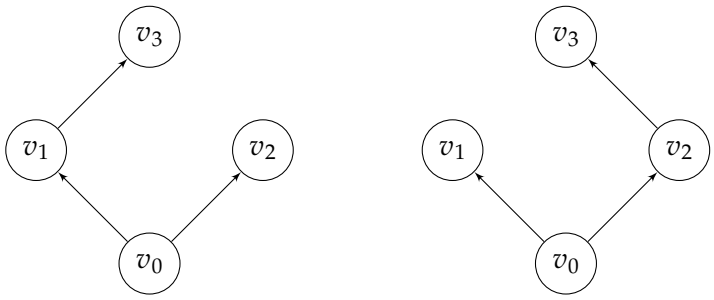


Figure 4.5 The two arborescences of the directed 2-topology.

For completeness, and because the insight will be of use to us later, we may find the number of spanning arborescences in a general directed n-topology. The proof for this formula will fall out of the following example of the 3-adinkra. Since there is only one edge from the root to any of the level 1 vertices and all edges are directed upward, we see that any spanning arborescence must include all three of the lowest edges:

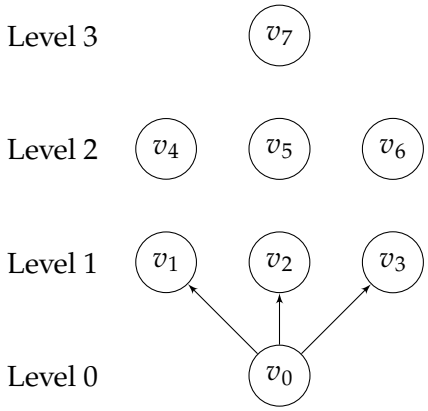


Figure 4.6 The lowest three edges of the directed 3-topology are required for any arborescence.

Now, a spanning arborescence can get to each of the three vertices in level 2 in two ways, giving us $2^3 = 8$ ways to fill in the next set of edges:

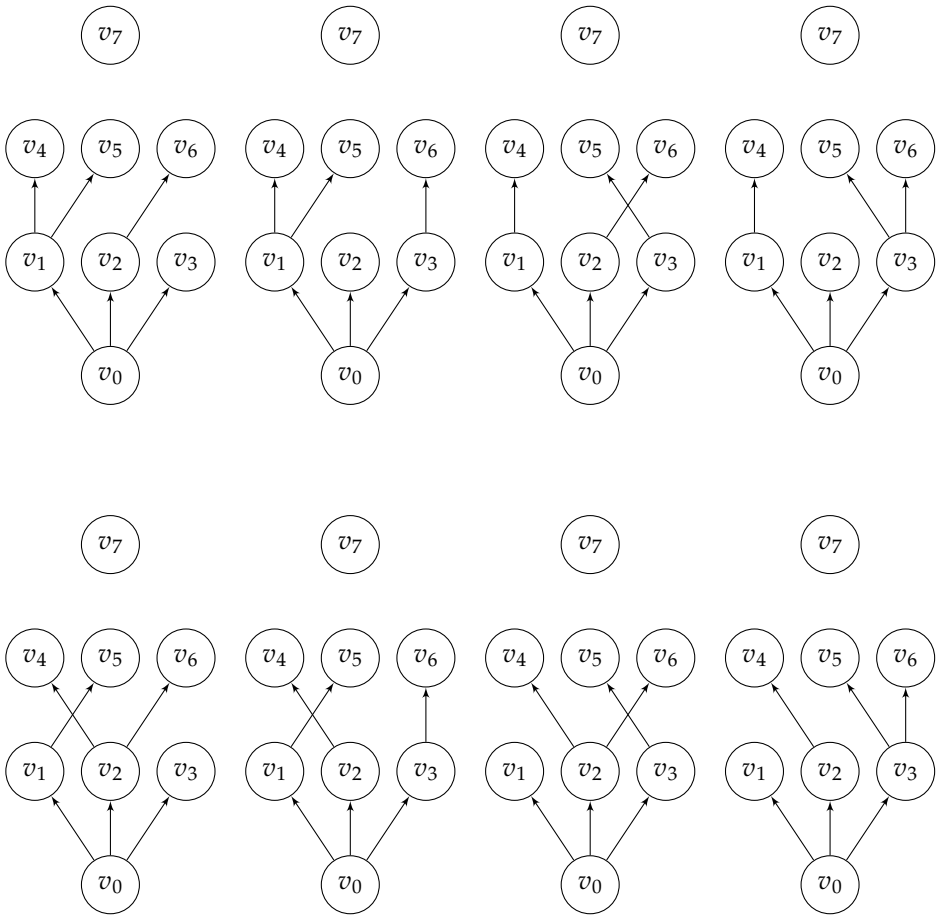


Figure 4.7 There are eight different ways to have an arborescence reach vertices 4, 5, and 6.

Of course, there are three vertices in level 2 from which to go to the single vertex in the last level, giving us 3 options. This is a total of $1 \cdot 8 \cdot 3 = 24$ arborescences for $n = 3$. Similarly, for $n = 4$, there are 1,4,6,4,1 vertices in levels 0,1,2,3,4, respectively. There is 1 configuration of edges from level 0 to level 1. From level 1 to level 2, there are $2^6 = 64$ configurations since we have to get to 6 vertices and each can be achieved in two ways (see figure 1.3). From level 2 to level 3, there are another $3^4 = 81$ configurations because there are four vertices in level 4 and each can come from three places. Of course, we then have 4 possibilities for the last level. This gives us a total of $1 \cdot 64 \cdot 81 \cdot 4 = 20\,736$ arborescences for $n = 4$. From this reasoning, we arrive at a result.

Theorem 4.4. *The number, N, of spanning arborescences of a directed n-topology is given by*

$$N = \prod_{i=1}^{n} i^{\binom{n}{i}}.$$

Proof. Consider some directed n-cube adinkra topology, A. To count the number of arborescences, we first note that there are no edges directed downward. So, a path from the root (the only vertex of height 0) can go directly from a vertex of height k only to a vertex of height $k+1$ and can only get to a vertex of height $k+1$ directly from a vertex of height k.

Now, consider some height $i \leq n$. Our graph A contains $\binom{n}{i}$ vertices at height i, each of which has i distinct incoming edges. So, since our paths can only ascend in the way described above, there are i ways to connect each vertex at height i to a path below. This gives us $i^{\binom{n}{i}}$ ways to extend paths from level $i-1$ to level i. We need to do this path extension process from levels 1 through n, and each level is independent of the previous one. This gives us the desired formula,

$$N = \prod_{i=1}^{n} i^{\binom{n}{i}}.$$

□

This expression gives us a sequence 2, 24, 20 736, 309 586 821 120, 11 501 279 977 342 425 366 528 000 000, . . . for the number of spanning arborescences, and so, the order of the directed Jacobian of the n-adinkra. This sequence appears in the Online Encyclopedia of Integer Sequences as the product of sizes of all the nonempty subsets of an n-element set.

Before we continue with our main result, it will be helpful to consider Tutte's directed matrix tree theorem, an analog to Kirchhoff's matrix tree theorem for directed graphs.

Theorem 4.5 (Tutte, 1948)**.** *If G is a directed graph with edge set $E(G)$ and vertex set $V(G) = \{v_1, \ldots, v_n\}$ and L is an $n \times n$ matrix whose entries are given by*

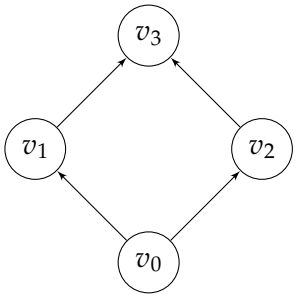
$$L_{ij} = \begin{cases} \deg_{in}(v_j) & \text{If } i = j \\ -1 & \text{If } i \neq j \text{ and } (v_i, v_j) \in E(G) \\ 0 & \text{Otherwise} \end{cases}$$

then the number N_j of spanning arborescences with root at v_j is

$$N_j = \det\left(\hat{L}_j\right)$$

where $\hat{L}_j$ is the matrix produced by deleting the j-th row and column from L.

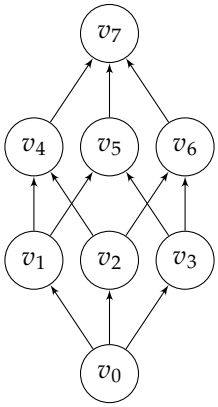
We can consider a couple examples as they relate to adinkras. Note that directed adinkra topologies are particularly well behaved because they have only one option for a root vertex. So, the total number of spanning arborescences is just the number of spanning arborescences with root v_0. For $n = 2$, we have



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

$$N_0 = \det\left(\hat{L}_0\right) = \det\left(\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \\ 0 & 0 & 2 \end{bmatrix}\right) = 2.$$

As a comforting check, we see that this is consistent with the number of arborescences we get from our earlier formula. For $n = 3$, we have



$$L = \begin{bmatrix} 0 & -1 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & -1 & 0 & 0 & -1 \\ 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3 \end{bmatrix}$$

$$N_1 = \det\left(\hat{L}_0\right) = \det\left(\begin{bmatrix} 1 & 0 & 0 & -1 & 0 & 0 & -1 \\ 0 & 1 & 0 & -1 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 & -1 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 2 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 2 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 3 \end{bmatrix}\right) = 24.$$

This is again consistent with the number of arborescences we get from our earlier formula. In fact, this will always be consistent because the matrix will always be upper triangular, and the determinant will be the product of the entries on the diagonal. Since the diagonal is the in-degree of each vertex, we get exactly the product expression in Theorem 4.5.

4.3 Finding a General Directed Jacobian

Recall our useful Theorem 2.11, which allows us to more easily find isomorphic groups to desired graph Jacobians. Our work with spanning arborescences motivates us to propose the following analog.

Conjecture 4.6. *For a directed adinkra topology G on n vertices, we have*

$$\mathrm{Jac}(G) \cong \mathbb{Z}^{n-1}/\mathrm{im}\left(\hat{L}_0\right).$$

To begin to prove this conjecture, we consider a group homomorphism

$$\begin{aligned} \varphi : \mathbb{Z}^{n-1} &\to \mathrm{Jac}(G) \\ d &\mapsto [d_0 v_0 + d_1 v_1 + \cdots d_{n-2} v_{n-2}], \end{aligned}$$

where the d_i are the $n-1$ integers of d. Note that we have assigned $n-1$ vertices, which leaves the last vertex, v_{n-1} to be the sum of the negatives of the integers in d. Let us determine the kernel of φ.

In the case of the directed 2-topology, we have $n = 4$ vertices, and we want to find $d = (a, b, c) \in \mathbb{Z}^3$ such that $\varphi(d) = [0v_0 + 0v_1 + 0v_2 + 0v_3] \in \mathrm{Jac}(G)$. Recall that for the directed 2-topology, a divisor is equivalent to this $0_{\mathrm{Jac}(G)}$ if it has an even degree v_0 and any degree v_1, v_2. Of course, v_3 is then determined by these values. So, the kernel of φ for the directed 2-topology is $\{(2a, b, c) \mid a, b, c \in \mathbb{Z}\}$, which is exactly $\mathrm{im}(\hat{L}_j)$, and the theorem holds by the first isomorphism theorem.

This seems to offer some support for our conjecture, but it remains to be proven in general.

Note that the original Theorem 2.11 comes as a corollary to our new Conjecture 4.6. Given an undirected graph, we may treat each undirected edge as a directed edge in both directions. If we do this, our matrix $\hat{L}_j$ becomes the reduced Laplacian, $\hat{\Delta}$ from before, and we are left with the earlier result.

With this new conjecture, we may find directed adinkra Jacobians as we found undirected cube Jacobians earlier. Our Sage algorithm in Appendix A gives us the Smith-normal forms of $\hat{L}_0$. As before, we find the first few Smith-normal form diagonals:

$$\begin{aligned}
D_2 &= [1,1,2],\\
D_3 &= [1,1,1,1,2,2,6],\\
D_4 &= [(1,8),(2,3),(6,3),(12,1)],\\
D_5 &= [(1,16),(2,5),(6,5),(12,4),(60,1)],\\
D_6 &= [(1,32),(2,10),(6,6),(12,9),(60,6)],\\
D_7 &= [(1,64),(2,21),(6,7),(12,14),(60,20),(420,1)],
\end{aligned}$$

where each tuple gives a diagonal entry followed by the number of times it appears. These diagonals give us Jacobians of

$$\begin{aligned}
J_2 &\cong \mathbb{Z}^3/\left(\mathbb{Z}\times\mathbb{Z}\times 2\mathbb{Z}\right) \cong \mathbb{Z}/2\mathbb{Z},\\
J_3 &\cong \mathbb{Z}^7/\left(\mathbb{Z}^4\times 2\mathbb{Z}\times 2\mathbb{Z}\times 6\mathbb{Z}\right) \cong (\mathbb{Z}/2\mathbb{Z})^2\times(\mathbb{Z}/6\mathbb{Z}),\\
&\vdots
\end{aligned}$$

and orders of

$$\begin{aligned}
|J_2| &= |\mathbb{Z}/2\mathbb{Z}| = 2,\\
|J_3| &= |(\mathbb{Z}/2\mathbb{Z})\times(\mathbb{Z}/2\mathbb{Z})\times(\mathbb{Z}/6\mathbb{Z})| = 24,\\
|J_4| &= 2^3\cdot 6^3\cdot 12 = 20\,736,\\
|J_5| &= 2^5\cdot 6^5\cdot 12^4\cdot 60 = 309\,586\,821\,120,\\
|J_6| &= 2^{10}\cdot 6^6\cdot 12^9\cdot 60^6 = 11\,501\,279\,977\,342\,425\,366\,528\,000\,000,\\
|J_7| &= 2^{21}\cdot 6^7\cdot 12^{14}\cdot 60^{20}\cdot 420 = 115\,744\,510\,977\,565\,557\,983\,391\,999\,957\,434\\
&\qquad 605\,749\,927\,936\,000\,000\,000\,000\,000\,000\,000.
\end{aligned}$$

These directed Jacobian orders match the number of spanning arborescences we found earlier, as desired. As we have defined it then, the order of the

directed Jacobian of the directed n-topology is the product of sizes of all the nonempty subsets of an n-element set, a sequence published as Krizek (2013). As before, it is unclear what patterns the groups themselves follow.

Chapter 5

Further Work

There are many branches that we have considered briefly, but that we have not yet explored. Further work could be done in these directions.

5.1 Patterns among Jacobians

It would be ideal to give a form for the Jacobian group (not just the order) of a given adinkra without having to compute the Laplacian and Smith-normal form. Unfortunately, there does not seem to be any easily noticeable pattern among the Jacobians we have found. The most significant patterns we have found are listed below:

- The number of 1's in the diagonal of the Smith Normal Form of the reduced Laplacian of an n-cube is 2^{n-1}.
- The number of 2's does not match any sequences on the online encyclopedia of integer sequences, but it is very close to some. Sequences A000975 (Bernstein et al. (1996)) and A215410 (Lagneau (2012)) are examples. In both of these, all the terms are off by either 0 or 1.

$$A000975 : a(2n) = 2 * a(2n-1), a(2n+1) = 2 * a(2n) + 1$$
(also n-th number without consecutive equal binary digits),
$$A215410 : a(0) = 0; a(n+1) = 2 * a(n) + k,$$
where $k = 0$ if $\text{prime}(n+2)/\text{prime}(n+1) < \text{prime}(n+1)/\text{prime}(n)$,
otherwise $k = 1$.

- The number 4 appears only in the $n = 2$ case.

- The number of 6's briefly follows the number of character table entries of the symmetric group S_n which are < 0, sequence A051748.
- The number 8 does not seem to appear often enough for pattern matching.
- The number 12 (and possibly all following numbers) does not match anything.

The directed Jacobians seem to be very slightly better behaved than the undirected ones, in that some sequences appear in the Online Encyclopedia of Integer Sequences. For example, the appearances of the number 2 give us a sequence $1, 2, 3, 5, 10, 21, \ldots$, which matches several sequences. More terms could narrow down possibilities.

5.2 A Different Way to Find Jacobians

As we have done it in this thesis, the Jacobian group is not very easy to find. Finding the Smith-normal form of a $2^n \times 2^n$ matrix quickly becomes computationally expensive. Our investigations seem to indicate that this is the most efficient way to find Jacobians. Even if a more efficient way cannot be found, any variation may give us more insight into the form of the Jacobian.

5.3 Adinkras from Sub-Adinkras

Something we could investigate is sub-adinkras of adinkras. Notice that as we go up an adinkra, we encounter smaller adinkras. For example, each vertex in the second level on a 4-adinkra is the root of a 3-adinkra. This could be potentially helpful for analyzing adinkras recursively. This is not entirely trivial, however, because the separate subadinkras share vertices. If we wished to, say, count the number, A_n, of spanning arborescences of a directed n-adinkra, we would have nA_{n-1} plus the number of arborescences that go through multiple sub-adinkras.

We should check to see if Zhang (2013) shows that adinkras are composed of sub-adinkras and show it if it does not. We could do this by removing the root and considering the codes that represent the vertices. Vertices share an edge iff their codes differ in exactly one digit. Since all vertices in a path from the root have 1s in the same place, we can remove that place, and the

topological structure would be the same in the smaller graph. need to check colouring, dashing.

5.4 Codes and n, k-Adinkras

We have done some investigation of the Jacobian groups and the number of spanning trees of adinkras whose topologies are n-cubes. Adinkras are also possible as quotients of cubes. A general n, k-adinkra is defined in Zhang (2013) in terms of doubly even error-correcting codes. We could also investigate the Jacobian groups and numbers of spanning trees of general n, k-adinkras.

5.5 General Directed Jacobian

Our Conjecture 4.6 still needs to be proven. One approach we have not investigated yet is treating each firing of chips as multiplication by the matrix of some linear transformation. Once that conjecture is proven, we can generalize the result to any directed graph, not just the directed adinkra topology. In our graphs, there was only one candidate for the root of an arborescence. In general, there may be many. Recall that Theorem 4.5 tells us the number of arborescences given a root vertex. The sum of this number over all root vertices gives us the total number of spanning arborescences. A generalized Conjecture 4.6 would need to account for this.

5.6 Other Jacobian Generalizations

In this thesis, we have investigated a notion of Jacobian that is sensitive to edge direction and have produced a new theorem to generalize our previous helpful theorem. But adinkras have properties that we have not addressed yet. A good property to consider next might be colouring. We could alter the chip firing scheme on a graph to consider edge colouring in its moves as well as direction. This could be generally useful as coloured graphs are widely interesting in mathematics and computer science.

5.7 Physical Significance of Adinkra Jacobians

Recall that our journey was originally suggested by theoretical particle physics. Adinkras represent supersymmetry algebras. While our work has been motivated by mathematical interest, it would be useful and interesting to investigate the physical meaning of the Jacobian of an adinkra.

Appendix A

Code for Adinkra Jacobian Computation

The following is our code for computing the Smith-normal forms of the reduced Laplacians of adinkras. It takes a boolean argument that determines whether it computes directed or undirected Jacobians. It was written in SageMathCloud, Stein et al. (YYYY).

```
import sys
import itertools

def Adjacency(n, directed):
    ''' create the same n-cube adjacency matrices we have been exploring by hand
        input:  n, the dimension of the cube
                directed, a boolean indicating whether we want the adjacency
                    matrix of a directed graph
        output: the adjacency matrix of the n-cube
    '''

    # Set the number of vertices to be 2^n for an n-adinkra.
    V = 2**n;

    # We begin by creating an array of all binary strings of length n to
    # represent the vertices and sort them by the number of 1's.  This gives us
    # an array of vertices with array index i corresponding to v_i.
    allCoords = ["".join(seq) for seq in itertools.product("01", repeat=n)];
    allCoords.sort(key=lambda s: s.count('1'));

    # Create a new array with pairs of coordinates and their indeces.  Since we
    # know there are n choose i vertices in level i, we can just go forward the
    # correct amount in this array to find the vertices in the level we are
    # working with.
    verts = [(i,allCoords[i]) for i in range(V)];

    # Create an empty 2^n x 2^n matrix to be populated.
```

```
        A = Matrix(V,V);

        # We know v_0 will always be adjacent to the n vertices in level 1.
        for i in range(1,n+1):
            A[0,i] = 1;

        # Now we will populate the rest of the matrix A.  We loop through each level
        # of the n-cube (up to n-1) and determine where we need edges by looking at
        # the next level.
        for i in range(1,n):
            # There are n choose i vertices in level i, so to find the vertices we
            # want, we add up n choose i for every level up to the one we are
            # currently working with.
            nCksAll = [binomial(n,k) for k in range(i)];
            nCk = sum(nCksAll);                         # n choose k, k < i
            # So that we know where to stop, we need to know where the vertices in
            # the next level begin.
            nCi = nCk+binomial(n,i)                     # n choose k, k <= i
            # Now, we take the list of vertices in the i-th level only.
            iVerts = verts[nCk:nCi];                    # i-th level vertices
            # We need to compare the i-th level vertices with the i+1-th level
            # vertices, so we need to know where the i+1-th level vertices end.
            nCi1 = nCi + binomial(n,i+1)
            # We can now make a list of the i+1-th level vertices since we know the
            # indeces at which they begin and end.
            i1Verts = verts[nCi:nCi1]                   # i+1-th level vertices
            # For each i-th level vertex, we need to figure out where it has edges.
            for j in iVerts:
                # We need to compare each i-th level vertex to every i+1-th level
                # vertex to see whether an edge should exist between them.
                for k in i1Verts:
                    edge = True;      # a bool representing the existence of an edge
                    # We loop through the binary strings of j, the i-th level
                    # vertex, and k, each i+1-th level vertex, to see if j has 1's
                    # only where k has 1's.  If so, add an edge.  If there are any
                    # positions in which j has an edge and k does not, we do not
                    # have an edge.
                    for l in range(n):
                        if (j[1][l] == '1' and k[1][l] != '1'):
                            edge = False;
                        A[j[0],k[0]] = int(edge);

        # We have only populated the upper triangle of A.  For an undirected graph,
        # we now copy all of these values to the lower triangle to make A symmetric.
        # If we consider edge directions, then the upper triangle matrix is the
        # correct adjacency matrix, and we do not need these lines.
        if (directed == False):
            for i in range(1,V):
                for j in range(i):
                    A[i,j] = A[j][i];

        return A;


    def nDiag(n, directed):
```

```
    ''' create D, the diagonal matrix of vertex degrees
        input:  n, the dimension of the cube
                directed, a boolean indicating whether we want the D matrix of a
                    directed graph
        output: the matrix with vertex degrees of the n-cube on the diagonal
    '''

    # Set the number of vertices to be 2^n for an n-adinkra.
    V = 2**n;

    # An undirected n-cube is n-regular, so we know all the vertex degrees are n
    if (directed == False):
        return Matrix(V,V,n);

    # For a directed n-topology, the in-degrees of the vertices increase as we
    # go up the graph.  Each graph has levels 0 through n, and level i contains
    # n choose i vertices.  Each of these n choose i vertices has in-degree i.
    # This gives us a D matrix with a diagonal that begins with 0 and increases
    # by 1 every n choose i entries.  For example, the diagonal for the directed
    # 3-topology is [0, 1, 1, 1, 2, 2, 2, 3].  That is, the entry 0 appears 3
    # choose 0 times, the entry 1 appears 3 choose 1 times, the entry 2 appears
    # 3 choose 2 times, and the entry 3 appears 3 choose 3 times.

    # We first create an empty matrix to populate.
    M = Matrix(V,V);

    # We now need a coordinate counter to keep track of where we are along the
    # diagonal of the matrix.
    coord = 0;

    # We loop through levels 0 to n of the height-assigned cube.
    for i in range(n+1):
        # Level i of the n-topology has n choose i vertices.
        for j in range(binomial(n,i)):
            # Each vertex in level i has in-degree i.
            M[coord,coord] = i;
            # We increment the coordinate so that we continue to move down the
            # diagonal.
            coord += 1;

    return M;


def Laplacian(n, directed):
    ''' create the Laplacian matrix for an n-cube
        input:  n, the dimension of the cube
                directed, a boolean indicating whether we want the Laplacian
                    matrix of a directed graph
        output: the Laplacian matrix of the n-cube
    '''

    D = nDiag(n, directed);     # construct diagonal matrix with n's
    A = Adjacency(n, directed); # construct adjacency matrix

    # Sage has a cube graph function that we can use to construct the
```

```
    # adjacency matrix for the n-cube.  This creates a rearranged matrix that
    # does not number vertices the way we have, but still gives the same
    # Smith-normal form.  This also does not give us the directed adjacency
    # matrix we want.
    # A = graphs.CubeGraph(n).adjacency_matrix()

    return D - A;               # subtract A from D to construct Laplacian


def reducedLaplacian(n, directed):
    ''' create the reduced Laplacian matrix for an n-cube
        input:  n, the dimension of the cube
                directed, a boolean indicating whether we want the reduced
                    Laplacian matrix of a directed graph
        output: the reduced Laplacian matrix produced by taking the (0,0)-minor
                    of the Laplacian matrix of the n-cube
    '''

    L = Laplacian(n, directed);

    # For the directed n-topology, we must remove the 0-th row and column since
    # v_0 is the root from which spanning arborescences can be produced.  For
    # the undirected graph, it does not matter which row and column we remove,
    # so we just remove the same ones.
    L = L.delete_columns([0]);
    L = L.delete_rows([0]);

    return L;


def cube(n, directed):
    ''' compute the Smith-normal form of the reduced Laplacian matrix for an
                    n-cube
        input:  n, the dimension of the cube
                directed, a boolean indicating whether we want the Smith-normal
                    form of the reduced Laplacian matrix of a directed graph
        output: the Smith-normal form of the reduced Laplacian matrix of the
                    n-cube.  This is returned as an array representing the
                    diagonal of the Smith-normal form matrix.  The elements of
                    this array are pairs giving the diagonal entry followed by
                    the number of times it appears.  For example, the directed
                    3-cube gives an output of [(1, 4), (2, 2), (6, 1)], which
                    should be interpreted as the diagonal
                    [1, 1, 1, 1, 2, 2, 6].
    '''

    RL = reducedLaplacian(n, directed);
    diagRL = RL.elementary_divisors();

    # We realized very late that Sage's elementary_divisors() function gives the
    # diagonal of the Smith-normal form and is far faster than the Smith-form
    # function.  We have included the original method below as well.

    # The first element of the smith_form() function output is the matrix.
    # RL = RL.smith_form()[0];
```

```
# We create an array of the diagonal entries of the Smith-normal form of the
# reduced Laplacian.  This is done for us by the elementary_divisors
# function.
# diag = [];
# for i in range(RL.nrows()):
#     diag += [RL[i,i]];

# The list we actually want is a shorter one that includes the entries of
# the Smith-normal form and the number of times each appears.
printList = [];
# We convert the diagonal to a set to remove duplicates.
diagSet = set(diagRL);
# For each unique entry in the diagonal, we add a pair containing the entry
# and the number of times it appears in the diagonal.
for i in diagSet:
    printList += [(i,diagRL.count(i))];
# Sets are not ordered, so we re-sort the diagonal entries to be in
# ascending order.
printList.sort();

return printList;
```

Appendix B

Code for Smith-Normal Form Computation

The following is our code for computing the Smith-normal form of a given matrix. It utilizes a brute force approach and has been replaced by Sage's more efficient `elementary_divisors()`. It was written in Python, Rossum (1995). Note that it depends on Python's default arbitrary-precision integers since the matrix entries get quite large through the calculation.

```
def swapRow(M,m,n):
    ''' swaps row m with row n of matrix M
        input:  M, the matrix to be modified
                m, the index of the first row to be swapped
                n, the index of the second row to be swapped
        output: M, with rows m and n swapped
    '''

    temp = M[m];
    M[m] = M[n];
    M[n] = temp;

    return M;


def swapCol(M,m,n):
    ''' swaps column m with column n of matrix M
        input:  M, the matrix to be modified
                m, the index of the first column to be swapped
                n, the index of the second column to be swapped
        output: M, with columns m and n swapped
    '''

    temp = []
    for i in range(len(M)):
        temp += [M[i][m]];
```

```
    for i in range(len(M)):
        M[i][m] = M[i][n];
        M[i][n] = temp[i];

    return M;


def negateRow(M,n):
    ''' negates row n of matrix M
        input:  M, the matrix to be modified
                n, the index of the row to be swapped
        output: M, with row n negated
    '''

    for i in range(len(M)):
        M[n][i] = -M[n][i];

    return M;


def negateCol(M,n):
    ''' negates column n of matrix M
        input:  M, the matrix to be modified
                n, the index of the column to be swapped
        output: M, with column n negated
    '''

    for i in range(len(M)):
        M[i][n] = -M[i][n];

    return M;


def addRow(M,m,n,add):
    ''' adds row m add times to row n of matrix M
        input:  M, the matrix to be modified
                m, the index of the row to add to another
                n, the index of the row to be added to
                add, the number of times to add m to n
        output: M, with row n modified
    '''

    for i in range(len(M)):
        M[n][i] = add*M[m][i] + M[n][i];

    return M;


def addCol(M,m,n,add):
    ''' adds column m add times to column n of matrix M
        input:  M, the matrix to be modified
                m, the index of the column to add to another
                n, the index of the column to be added to
                add, the number of times to add m to n
```

```
        output: M, with column n modified
    '''

    for i in range(len(M)):
        M[i][n] = add*M[i][m] + M[i][n];

    return M;


def sameSign(x,y):
    ''' compares the signs of two numbers, x and y
        input:  x, first number
                y, second number
        output: True if x and y have the same sign.  False otherwise.
    '''

    return ((x<0) == (y<0));


# We call the (0,0) entry of the matrix the head of the matrix.  Our
# Smith-normal form computation will involve iterating over smaller and smaller
# minors that we treat as smaller Smith-normal form problems.  Each such step
# places the new head of the matrix one diagonal entry farther down.  Within
# each of these large steps, we will need to use the head of the matrix to
# reduce all the entries in the row and column of the head to 0.  By the time
# the head is at the last diagonal entry, the matrix will be in Smith-normal
# form.
def SNF(M):
    ''' computes the Smith-normal form of a matrix M
        input:  M, the matrix we want the Smith-normal form of
        output: the matrix M in Smith-normal form
    '''

    # We will be using j to keep track of how far we have recursed.  Within each
    # iteration of j, we use row and column operations to get all 0's in the
    # outermost (j-th) row and column.  We advance j once we have gotten all
    # entries in the j-th row and j-th column except the diagonal entry, (j,j),
    # to be 0.  Advancing j repeats this process on the (0,0)-minor of the
    # matrix we were working with before so that we have smaller and smaller
    # "outermost" rows and columns until we have gotten through the entire
    # matrix and made every entry except the diagonal 0.
    for j in range(len(M)):
        # In order to use the head to reduce the other entries in the head's row
        # and column, we must search for the smallest nonzero entry absolute
        # value in the matrix and move it to the head by swapping rows and
        # columns.
        # We first assume by default that the smallest entry we are looking for
        # is already the head.  We can then compare other entries to it to see
        # if any are smaller.
        minIndex = (j,j);
        minValue = abs(M[j][j]);

        # If the head is zero, we will not be able to reduce anything through
        # our row operations, and we will end up with a zero in our
        # Smith-normal form.  We also cannot easily find another element to
```

```
# replace it since it is smaller than all the other absolute values.
# So, if we see a 0 at the head, we replace it with maxint so that it is
# sure to be replaced by something better.
if minValue == 0:
    minValue = sys.maxint;

# We now loop through the entire matrix after row and column j to see if
# we can find anything smaller than the current head.  Of course, we do
# not need to search if the head is already 1 since there is nothing
# smaller in the matrix.  In these cases, we just break.
for k in range(j,len(M)):
    if minValue != 1:
        for l in range(j,len(M)):
            if minValue != 1:
                # If we find something a nonzero entry that is smaller
                # than the head, we record it as the most recent
                # smallest value and keep looping.
                if (abs(M[k][l]) < minValue) and (abs(M[k][l]) > 0):
                    minIndex = (k,l);
                    minValue = abs(M[k][l]);
            else:
                break;
    else:
        break;
# If we found a smaller entry fitting our above description, we use row
# and column operations to move it to the head before proceeding.  If
# not, we proceed knowing the head is already the smallest entry.
if minIndex != (j,j):
    M = swapRow(M, j, minIndex[0]);
    M = swapCol(M, j, minIndex[1]);

# We create the variable i to loop through the j-th row and column.  At
# this point, j represents the the index along the diagonal where the
# current head is.  If the matrix is n x n, we are effectively trying to
# find the Smith-normal form of an n-j x n-j matrix because we have
# already done the first j rows.  At each step of j, we define i and
# loop from j+1 to n.  At each step of i, we look at matrix entries
# (j,i) and (i,j) and subtract a multiple of the head entry to get them
# as close to 0 as possible.  At each step of this process, we check to
# see if we have produced any matrix entries smaller than the head
# anywhere in the matrix.  If so, we swap rows and cloumns to make it
# the new head and restart the whole process for the current j.  At the
# end of this while loop, the j-th row and column should be all 0,
# except for the diagonal entry (j,j).
i = j + 1
while i < len(M):
    # A restart boolean keeps track of whether we have moved another
    # matrix entry to the head so we know if we need to restart the
    # process for this j.  Every time we restart, we have a smaller
    # head.  Eventually, we should end up with a head that divides all
    # entries of the j-th row and column and makes them all 0.
    restart = False;
    # label the head
    head = M[j][j];
```

```
# We move to the i-th entry of the j-th row.  Integer division of
# this entry by the head gives us add, the number of times we can
# subtract the head from this entry (j,i) to minimize it (or get
# close.  it shouldn't matter that we could possibly get smaller
# absolute value by subtracting one more because at each step, we
# look for smaller entries to become the head).  We then subtract
# the j-th column from the i-th column add times.
add = M[j][i]/head;
M = addCol(M,j,i,add*-1);

# This is our minimum entry finding algorithm again.  This should
# perhaps be its own function so that we can just call it within
# this function, but the SNF function did not work properly when
# that was tried, so it currently exists in this repetitive form.
minIndex = (j,j);
ixminValue = abs(M[j][j]);
if minValue == 0:
    minValue = sys.maxint;
for k in range(j,len(M)):
    if minValue != 1:
        for l in range(j,len(M)):
            if minValue != 1:
                if (abs(M[k][l]) < minValue) and (abs(M[k][l]) > 0):
                    minIndex = (k,l);
                    minValue = abs(M[k][l]);
            else:
                break;
    else:
        break;
# If we do move things around in the matrix, we set restart to True
# so that we start i at j+1 again.
if minIndex != (j,j):
    M = swapRow(M, j, minIndex[0]);
    M = swapCol(M, j, minIndex[1]);
    restart = True;

# We repeat everythnig for the i-th entry of the j-th column.
# Together with the last piece of code, each iteration of the while
# loop with respect to i reduces one additional entry along the j-th
# row and j-th column.
add = M[i][j]/head;
M = addRow(M,j,i,add*-1);

# Again, check for smaller matrix entries.
minIndex = (j,j);
minValue = abs(M[j][j]);
if minValue == 0:
    minValue = sys.maxint;
for k in range(j,len(M)):
    if minValue != 1:
        for l in range(j,len(M)):
            if minValue != 1:
                if (abs(M[k][l]) < minValue) and (abs(M[k][l]) > 0):
                    minIndex = (k,l);
                    minValue = abs(M[k][l]);
```

```
                    else:
                        break;
                else:
                    break;
            # Again, update the head if a smaller entry was found and set
            # restart to True.
            if minIndex != (j,j):
                M = swapRow(M, j, minIndex[0]);
                M = swapCol(M, j, minIndex[1]);
                restart = True;

            # If either of our reductions made it so that our matrix had an
            # entry smaller than the head, we will have moved it to the head.
            # In this case, we want to set i = j+1 so that we can start reducing
            # the j-th row and column again.  If not, we just increment i and
            # continue down the j-th row and column.
            if restart:
                i = j+1;
            else:
                i += 1;

    # At the end of our giant for loop, we will have gone through every level of
    # the matrix and we should have reduced all but the diagonal to 0.  At this
    # point, we may have negatives along the diagonal.  We negate the rows
    # that are negative by taking the absolute value of every diagonal entry.
    for i in range(len(M)):
        M[i][i] = abs(M[i][i]);

    return M;
```

Bibliography

Adinkramat. 2012. Visual adinkra creation software. https://github.com/chelleorc/adinkramat.

Baker, Matthew, and Serguei Norine. 2006. Riemann-Roch and Abel-Jacobi theory on a finite graph. *arXiv:math/0608360* URL http://arxiv.org/abs/math/0608360. ArXiv: math/0608360.

Bernstein, Mira, N. J. A. Sloane, and Robert G. Wilson. 1996. A000975 - OEIS. URL https://oeis.org/A000975.

Faux, Michael, and S. J. Gates. 2005. Adinkras: A graphical technology for supersymmetric representation theory. *Phys Rev D* 71:065,002. doi:10.1103/PhysRevD.71.065002.

Gaslowitz, Joshua. 2013. Chip Firing Games and Riemann-Roch Properties for Directed Graphs. *HMC Senior Theses* URL http://scholarship.claremont.edu/hmc_theses/42.

Hoppenfeld, Tessa. 2014. The structure of the jacobian group of a graph. Http://people.reed.edu/ davidp/homepage/students/hoppenfeld.pdf.

Krizek, Jaroslav. 2013. A229333 - OEIS. URL http://oeis.org/search?q=2%2C+24%2C+20736%2C+309586821120%2C+1150127997734242536652800000&sort=&language=english&go=Search.

Lagneau, Michel. 2012. A215410 - OEIS. URL https://oeis.org/A215410.

Rossum, Guido. 1995. Python reference manual. Tech. rep., Amsterdam, The Netherlands, The Netherlands.

Sloane, N. J. A., and Don Knuth. 1995. A006237 - OEIS. URL https://oeis.org/A006237.

Stein, W. A., et al. YYYY. *Sage Mathematics Software (Version x.y.z)*. The Sage Development Team. `http://www.sagemath.org`.

Weisstein, Eric W. ????a. Cover Relation. From MathWorld—A Wolfram Web Resource. URL http://mathworld.wolfram.com/CoverRelation.html.

———. ????b. Hasse Diagram. From MathWorld—A Wolfram Web Resource. URL http://mathworld.wolfram.com/HasseDiagram.html.

Zhang, Yan X. 2013. Adinkras for Mathematicians. In *25th International Conference on Formal Power Series and Algebraic Combinatorics (FPSAC 2013)*, eds. Alain Goupil and Gilles Schaeffer, *DMTCS Proceedings*, vol. AS, 457–468. Paris, France: Discrete Mathematics and Theoretical Computer Science. URL https://hal.inria.fr/hal-01229733.