Generating the Superstable Configurations of a Graph via its Acyclic Orientations

---

A Thesis
Presented to
The Division of Mathematics and Natural Sciences
Reed College

In Partial Fulfillment
of the Requirements for the Degree
Bachelor of Arts

Sofia D. Wright

May 2014

Approved for the Division
(Mathematics)

Dave Perkinson

Acknowledgements

Thank you Dave Perkinson for being a source of motivation, knowledge and wisdom throughout this process. Thank you Mom and Dad for supporting me always. Thank you to my best friends back home for being there when I needed you, and to the friends I have made here for contributing to my education more than any book ever has. You are all super, and you keep me stable.

Table of Contents

Introduction **1**

Chapter 1: Background Math **3**
- 1.1 Superstable Configurations 3
  - 1.1.1 Divisors, Firing Scripts, and the Jacobian Group 3
  - 1.1.2 Firing Scripts and the Laplacian Matrix 4
  - 1.1.3 Fixing a Sink 5
  - 1.1.4 Introducing Superstables 7
- 1.2 Acyclic Orientations 8
- 1.3 Converting Acyclic Orientations to Superstables 9
  - 1.3.1 Dhar's Algorithm 9
  - 1.3.2 Proof of the Bijection 9

Chapter 2: The Algorithm **13**
- 2.1 Generating Acyclic Orientations 13
  - 2.1.1 Toy Example 13
  - 2.1.2 Defining a Poset 15
  - 2.1.3 The First Piece of Code: Compute_AO 16
  - 2.1.4 Expanding the Code for the Entire Graph 19
  - 2.1.5 Unique Source Acyclic Orientations 20
- 2.2 Conversion to Superstables 21
  - 2.2.1 Maximal Superstables 21
  - 2.2.2 Find All Superstables 21

Chapter 3: Testing for Time **23**
- 3.1 The Pre-Existing Code 23
- 3.2 The Results 24

Conclusion **27**

Appendix A: The Code (in Sage programming language) **29**

References **39**

Abstract

Superstable configurations on a graph G are the elements of a finite abelian group associated with G through a certain chip-firing game. Benson [2] defines a bijection between these superstable configurations and the group of unique-source acyclic orientations on G. In this thesis we expand upon an algorithm created by Squire [5] to generate the acyclic orientations of a graph and use the bijection to generate the superstable configurations.

Introduction

Let G be a connected, finite, simple graph with vertex set V and edge set E (thus, G is undirected, unweighted, and has no loops). Imagine this graph as a representation of a community, where the vertices V represent people in the community and the edges E represent social connections between these people: two vertices that share an edge are acquainted with each other, while two vertices that are unconnected by an edge likewise do not have any social connection. We will call acquaintances of v the *neighbors* of v.

Each person-vertex in this community may possess some amount of money (a positive integer value), or may owe some amount of money (a negative integer value). They may change their monetary possession by lending and borrowing from their neighbors. But in the interest of being fair, they follow these rules: if a person-vertex v borrows some number of dollars from one neighbor, she must borrow an equal amount from all of her neighbors. Likewise, people-vertices must lend to all of their neighbors in equal amounts as well (see Figure 1).

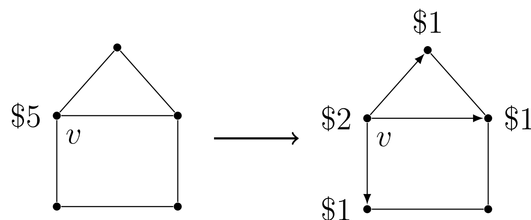


Figure 1: In a lending move, vertex v lends an equal amount of dollars to each of its neighbors.

Say the wealth in some community is distributed in a certain way, call this distribution D, and person v borrows one dollar from each of her neighbors; this creates a new distribution, call it D'. But there is some relation between these two distributions, in that D' can be achieved from D by a simple borrowing, and easily revert back to D if v were to lend a dollar to each of her neighbors. We call the set of all distributions that can be achieved from D, by any number of borrowings and lendings, the *equivalence class* of D.

Since we know how the rules of borrowing and lending work, given any wealth distribution D we can find all elements in its equivalence class. Thus, we can represent an entire equivalence class by just a single distribution from the class. We can imagine that, for a larger community, it would then be convenient to look at just the set of

representatives of the equivalence classes.

To describe a nice set of representatives for these equivalence classes, imagine that the goal of our community is to get all community members out of debt. Naturally this might not be possible, so to this end we could imagine that one benevolent individual, call her person-vertex q, might volunteer as tribute and take on the only debt in the community. All other members could borrow and trade dollars amongst themselves, and borrow from q, until only q would have a negative number of dollars.

Then we could imagine that the non-q vertices of our community, because they are fairly benevolent themselves, would cooperate amongst themselves, borrowing and lending, to try and bring q as much out of debt as possible without going into debt themselves. They would eventually achieve a distribution of wealth such that there was no further possible single-vertex or teamwork lending or borrowing move that could be made without sending some non-q vertex into debt. This type of distribution has a name: *superstable configuration*.

The set of all superstable configurations, or *superstables*, of the graph make up a representative group of the equivalence classes [4]. In this paper we are concerned with calculating the superstables of a given graph in as efficient a process as possible.

A computer algorithm already exists in the Sage mathematics software system [6] for finding the superstable configurations of a graph. But this algorithm seems inefficient; through a laborious process of repeated stabilizing (adding dollars to the system and then lending and borrowing between vertices until a certain balance is achieved) it computes a "dual" set to the superstables, then converts them to the desired superstable configurations.

There exists a simple bijection [2] between the set of superstables of a graph G and another important set in graph theory, the set of acyclic orientations on G (acyclic orientations will be defined in Section 1.2). Then, an algorithm that may be more efficient than what exists could be to generate all acyclic orientations on a graph and then convert those, via the bijection, to superstables. In this paper we present and code this algorithm. Our hope is that this will be a more efficient—specifically, a quicker—process than the existing algorithm for generating superstables.

In Chapter 1 we present the background math necessary for understanding the process and the motivation behind this algorithm. We introduce divisors and firing scripts, specify those concepts for configurations, and look at the structure of the superstable group. Then we introduce acyclic orientations and go on to describe and prove the bijection between acyclic orientations and the maximal superstable configurations.

In Chapter 2 we first present a small-scale example of how our algorithm works. Then we present the pseudocode for the multiple steps in the algorithm, from generating the acyclic orientations to converting those to maximal superstables.

Finally, in Chapter 3 we test the runtime of our algorithm on different graphs, alongside the runtime for the pre-existing Sage code, and discuss the results.

Chapter 1

Background Math

1.1 Superstable Configurations

1.1.1 Divisors, Firing Scripts, and the Jacobian Group

Let $G = G(V, E)$ be a graph with vertex set V and edge set E. We now convert the analogy of our borrowing and trading community into more formal language.

The distribution of wealth amongst our vertex-people is called a *divisor* of G. Formally, a divisor of G is an element of the free abelian group on the vertices of G. So a divisor takes the form $D = \sum_{v \in V} d_v v$, for $d_v \in \mathbb{Z}$. This value, d_v, represents the number of "dollars" currently held by vertex v. We define the *degree* of a graph as $\deg(D) = \sum_{v \in V} d_v$. The set of divisors on G is denoted by $\text{Div}(G)$.

Like the distribution of dollars between vertices, the divisor on a graph can be changed by vertices lending to and borrowing from their neighbors. When v lends to its neighbors, we call this a *vertex firing* on v or say that we have fired v. When v borrows from its neighbors, we talk about a *reverse-firing* on v, though for simplicity we may continue to say v borrows.

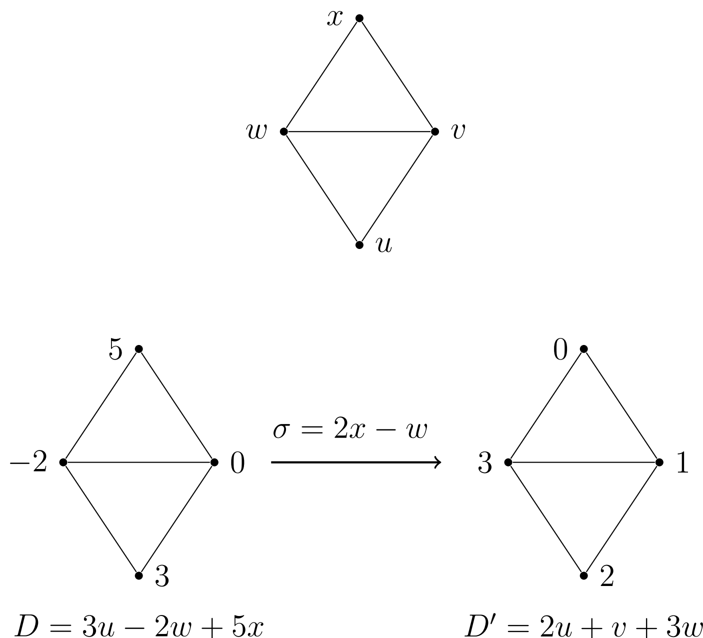
We can represent a combination of vertex firings by an expression called a *firing script*. A firing script σ, like a divisor, is an element of the free abelian group on the vertices of G and can be expressed as $\sigma = \sum_{v \in V} s_v v$, for $s_v \in \mathbb{Z}$. The coefficient s_v represents the number of times that v is fired.

If a divisor $D' \in \text{Div}(G)$ can be obtained by a series of vertex firings or reverse-firings, represented by firing script σ, from a divisor $D \in \text{Div}(G)$, we say that D and D' are *linearly equivalent* and write $D \sim D'$. We represent this transformation by the notation $D \xrightarrow{\sigma} D'$. If $D \sim D'$, then $\deg(D) = \deg(D')$. For $D \in \text{Div}(G)$, the set of all $D' \in \text{Div}(G)$ such that $D \sim D'$ is called the *equivalence class* of D.

The set of representatives of this equivalence class can be discussed as the group $\text{Div}(G)$ modulo the equivalence $\sim$. This group is known as the *Picard group* and is denoted $\text{Pic}(G) = \text{Div}(G)/\!\sim$.

Example 1.1.1. We have a simple graph G on vertices $\{u, v, w, x\}$, pictured below. We assign to G the divisor $D = 3u - 2w + 5x$. Suppose we apply firing script $\sigma = 2x - w$, so we fire x twice and borrow at w once. The resulting divisor

is $D' = 2u + v + 3w$.



1.1.2 Firing Scripts and the Laplacian Matrix

It is simple to observe a single vertex firing on a small graph, but we need a tool for calculating multiple firing moves on any size of graph. The tool we use is the *Laplacian matrix.*

A divisor D is written as a linear equation. However, if we impose an ordering on the vertices, say $(v_1, ..., v_n)$, and let d_i be the coefficient of v_i in D, then we can represent D as simply an ordered list, or a vector, of the d_i values. This suggests an isomorphism with $\mathbb{Z}^n$:

$$\begin{aligned} \mathrm{Div}(G) &\xrightarrow{\sim} \mathbb{Z}^n \\ D = \sum_{i=1}^{n} d_i v_i &\longmapsto (d_1, ..., d_n). \end{aligned}$$

Assume an ordering $(v_1, ..., v_n)$ on the vertices of G. Define the degree of a vertex v, denoted $\deg_G(v)$, as the number of edges in E incident to v. Let $\mathbf{M}$ be the $n \times n$ diagonal matrix, with $\mathbf{M}_{ii} = d_i$ for $i \in (1, ..., n)$. Let $\mathbf{A}$ be the $n \times n$ adjacency matrix for G,

$$\mathbf{A}_{ij} = \begin{cases} \deg_G(v) & \text{for } i \neq j \\ 0 & \text{for } i = j. \end{cases}$$

Then the Laplacian matrix is defined as

$$L = \mathbf{M} - \mathbf{A}.$$

Note that the ith column of L represents the rules for borrowing at vertex v_i, and conversely the ith column of $-L$ represents the rules for firing vertex v_i. If $D \xrightarrow{v_i} D'$, then $D - L(v_i) = D'$.

Like divisors, firing scripts can be represented as elements of $\mathbb{Z}^n$. So if for firing script σ we have $D \xrightarrow{\sigma} D'$, then $D - L(\sigma) = D'$ (see Example 1.1.2).

Since the image of L encodes the borrowing moves that distinguish the equivalence classes of D, then the Picard group is isomorphic to $\mathbb{Z}^n$ modulo $\mathrm{im}(L)$:

$$\mathrm{Pic}(G) = \mathrm{Div}(G)/\sim \quad \xrightarrow{\sim} \quad \mathbb{Z}^n/\mathrm{im}(L). \tag{1.1}$$

Example 1.1.2. The graph from Example 1.1.1 with vertex ordering (u, v, w, x) has Laplacian matrix

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

We can view the divisor from Example 1.1.1 as a vector, $D = (3, 0, -2, 5)$, and the same for the firing script, $\sigma = (0, 0, -1, 2)$. Then we can apply σ to D and determine the results using the graph's Laplacian matrix:

$$D - L(\sigma) = \begin{pmatrix} 3 \\ 0 \\ -2 \\ 5 \end{pmatrix} - \begin{pmatrix} 2 & -1 & -1 & 0 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ 0 & -1 & -1 & 2 \end{pmatrix} \begin{pmatrix} 0 \\ 0 \\ -1 \\ 2 \end{pmatrix} = \begin{pmatrix} 2 \\ 1 \\ 3 \\ 0 \end{pmatrix} = D'.$$

1.1.3 Fixing a Sink

We now return to another component of our introductory analogy: the benevolent person-vertex who volunteers to take on the entire debt of the community so that all the others can be out of debt. On our graph $G(V, E)$, we choose such a vertex $q \in V$ and call it a *sink*.

First, we define a subset of $\mathrm{Div}(G)$. Let $\mathrm{Div}_0(G)$ be the set of divisors of degree zero, i.e,

$$\mathrm{Div}_0(G) \;=\; \{D = \textstyle\sum_{v \in V} d_v v \in \mathrm{Div}(G) \mid \sum_{v \in V} d_v = 0\}.$$

An isomorphism can be defined between $\mathrm{Div}(G)$ and $\mathrm{Div}_0(G)$ by fixing a sink vertex q. Let $\widetilde{V} = V \backslash \{q\}$. Then for $D \in \mathrm{Div}(G)$, $D' \in \mathrm{Div}_0(G)$, and $k \in \mathbb{Z}$, we have the isomorphism of groups,

$$\begin{aligned} \mathrm{Div}(G) &\xrightarrow{\sim} \mathbb{Z} \times \mathrm{Div}_0(G) \\ D &\longmapsto (\deg(D), D - \deg(D)q) \\ D' + kq &\longleftarrow\!\shortmid (k, D'). \end{aligned}$$

The same equivalence relation that applies to $\mathrm{Div}(G)$ also applies to $\mathrm{Div}_0(G)$, and we can likewise define classes in $\mathrm{Div}_0(G)$: the class of a divisor $D \in \mathrm{Div}_0(G)$ is the set of all $D' \in \mathrm{Div}_0(G)$ such that $D \sim D'$.

If $D \sim D'$, then $\deg(D) = \deg(D')$, hence,

$$D - \deg(D)q \sim D' - \deg(D')q.$$

From this, the previous isomorphism induces an isomorphism for the Picard group:

$$\begin{aligned} \mathrm{Pic}(G) = \mathrm{Div}(G)/\!\sim \; &\xrightarrow{\sim} \mathbb{Z} \times \mathrm{Div}_0(G)/\!\sim \\ D &\longmapsto (\deg(D), D - \deg(D)q) \\ D' + kq &\longleftarrow (k, D'). \end{aligned} \tag{1.2}$$

We call $\mathrm{Div}_0(G)$ the *Jacobian* of G and denote it $\mathrm{Jac}(G)$.

For $D = \sum_{v \in V} d_v v \in \mathrm{Div}_0(G)$, we know $d_q = -\sum_{v \in \widetilde{V}} d_v$, hence, we can forget the value on q. With this adjustment in thinking, we consider the distribution of wealth in a graph on just the vertices in $\widetilde{V} = V \backslash \{q\}$. A *configuration*, c, of G with respect to q is an element of the free abelian group on $\widetilde{V}$ and so takes the form $c = \sum_{v \in \widetilde{V}} c_v v$, where c_v is the number of dollars on v. We denote the set of configurations on G with fixed sink q by $\mathrm{Config}(G, q)$.

There is a clear mapping from $\mathrm{Div}(G)$ onto $\mathrm{Config}(G, q)$, since for any divisor D, we can set the value $d_q = 0$ and then ignore q to produce a configuration c. Likewise there is a mapping from $\mathrm{Config}(G, q)$ onto $\mathrm{Div}(G)$. For $c = \sum_{v \in \widetilde{V}} c_v v$, then $\deg(c) = \sum_{v \in \widetilde{V}} c_v$. Then, for any configuration $c \in \mathrm{Config}(G, q)$, we have $c - \deg(c)q \in \mathrm{Div}(G)$. These mappings induce the isomorphism

$$\begin{aligned} \mathrm{Div}_0(G) &\xrightarrow{\sim} \mathrm{Config}(G, q)) \\ D &\longmapsto D_{q=0} \\ c - \deg(c)q &\longleftarrow c. \end{aligned}$$

Thus, we can specify most concepts that we looked at for divisors to apply to configurations. A firing script σ that does not involve q (i.e, $s_q = 0$) can be applied to a configuration c in the same way that it is applied to a divisor. For $c, c' \in \mathrm{Config}(G, q)$ and for some firing script σ, if $c \xrightarrow{\sigma} c'$ we say c and c' are linearly equivalent and write $c \sim c'$. We also have equivalence classes of configurations, and the configurations modulo this equivalence form a group, $\mathrm{Config}(G, q)/\!\sim$.

Suppose, for some configuration c, we have $c_v \geq 0$ for all $v \in V$. Then we write $c \geq 0$. Given some $c \geq 0$ and a transformation $c \xrightarrow{\sigma} c'$, we say σ is a *legal* firing move if $c' \geq 0$.

As with divisors, we can calculate firing script transformations for configurations using matrices and vectors. The *reduced Laplacian*, $\widetilde{L}$, with respect to q is the Laplacian matrix with the column and row corresponding to q removed. Like with the Laplacian and divisors, the columns of $-\widetilde{L}$ represent the rules for firing at those corresponding vertices on a configuration. Thus, if $c \xrightarrow{\sigma} c'$, then $c - \widetilde{L}(\sigma) = c'$.

The previous isomorphisms (1.1) and (1.2) induce the following chain of isomorphisms:

$$\mathrm{Jac}(G) = \mathrm{Div}_0(G)/\!\sim \;\xrightarrow{\sim}\; \mathrm{Config}(G, q)/\!\sim \;\xrightarrow{\sim}\; \mathbb{Z}^{n-1}/\mathrm{im}(\widetilde{L}).$$

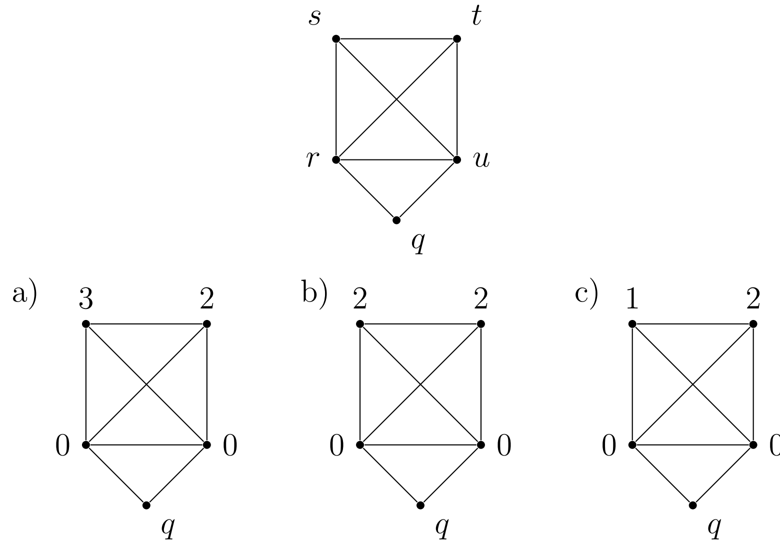


Figure 1.1: Three possible configurations on the vertex set $\{r, s, t, u\}$ with sink vertex q: (a) depicts a non-stable configuration, since vertex s can be legally fired; (b) depicts a stable, but not superstable, configuration, since no single vertex can be fired but the vertex set $\{s, t\}$ can be legally fired; (c) depicts a superstable configuration, as it has no legal vertex firings.

1.1.4 Introducing Superstables

Suppose we have firing script $\sigma = v$, a single vertex in $\widetilde{V}$. This is our basic vertex firing. Recall, for $c, c' \in \text{Config}(G)$, $c \xrightarrow{\sigma} c'$ is a legal firing if $c' \geq 0$. If a configuration $c \geq 0$ and for all $v \in \widetilde{V}$ there does not exist a legal vertex firing, we say c is a *stable configuration*.

For some configurations, no legal single vertex firings are possible, but it might be possible to make a legal *set firing*. In a set firing, we take some subset W of $\widetilde{V}$ and fire all vertices in W at once. As a script firing, this would be expressed as $\sigma = \sum_{v \in W} v$. (Note that, if W is a set of just a single vertex, then this firing is equivalent to a vertex firing). If a configuration c has no legal set firings, we call c a *superstable configuration* (Figure 1.1).

A graph usually has multiple superstable configurations. Denote the set of superstable configurations on a graph G with respect to sink q by $\text{SS}(G, q)$. For configurations c and c', we say $c \geq c'$ if and only if $c_v \geq c'_v$ for all v in $\widetilde{V}$. If c is a superstable configuration and $c \geq c'$ for all superstables $c' \in \text{SS}(G)$, then we say c is a *maximal superstable* configuration.

We *stabilize* a non-negative configuration c by continually firing any vertex v that has $c_v \geq \deg(v)$ until no such vertex exists, i.e, until there are no more legal vertex firing moves possible; the resulting configuration is stable.

The set of superstables $\text{SS}(G, q)$ forms a group with the operation $*$ defined as follows: for $c, c' \in \text{Config}(G, q)$, we let $c * c'$ represent the configuration formed by stabilizing $c + c'$. It is well-known that this operation is well-defined [4].

Every configuration of G is uniquely equivalent to an element in $\mathrm{SS}(G,q)$ [4]. This means that every equivalence class of configurations can be uniquely represented by a superstable configuration. Thus, $\mathrm{SS}(G,q)$ is isomorphic to $\mathrm{Config}(G,q)/\!\sim$, so we have the chain of isomorphisms,

$$\mathrm{Jac}(G) \xrightarrow{\sim} \mathrm{Config}(G,q)/\!\sim \xrightarrow{\sim} \mathrm{SS}(G,q).$$

Since $\mathrm{Config}(G,q)/\!\sim \xrightarrow{\sim} \mathbb{Z}^{n-1}/\mathrm{im}(\widetilde{L})$, it follows (Holroyd, Lemma 2.8 [4]) that

$$|\mathrm{SS}(G,q)| = |\mathrm{Jac}(G)| = \det(\widetilde{L}).$$

1.2 Acyclic Orientations

We now introduce a new concept: the *orientation* of a graph. Consider traversing a graph G by moving between neighbor vertices along their shared edge. If an edge uv between vertex u and vertex v is assigned a direction of traversal, say the edge can be traversed from u to v but not from v to u, then we call this a *directed edge* and denote it (u,v). A graph in which all edges are assigned a direction is called a *directed graph*, or *digraph*. We can also call this an *oriented graph*, and the set of directions that correspond to each edge an *orientation*.

If one can traverse a series of directed edges to get from u to v, we say v is *reachable* from u and call this series of edges a *directed path*. A directed path that ends on the vertex it started at is a *directed cycle*. For this paper we are concerned with the orientations on a graph that do not contain any cycles—the *acyclic orientations*. We denote the set of all acyclic orientations on a graph G by $\mathrm{Ao}(G)$.

The *indegree* of a vertex v is the number of its incident edges that are directed towards v under a given configuration. For a graph with orientation α, we denote this by $\mathrm{indeg}_\alpha(v)$. A vertex with indegree equal to zero, in other words a vertex that only has incident edges directed away from it, is called a *source vertex*. An acyclic orientation must always have at least one source. We call an orientation that has only a single source vertex a *unique source orientation* (Figure 1.2). If we choose to fix vertex q as the unique source, then the set of q-unique source orientations, $\mathrm{Ao}(G,q)$, is a subset of $\mathrm{Ao}(G)$.

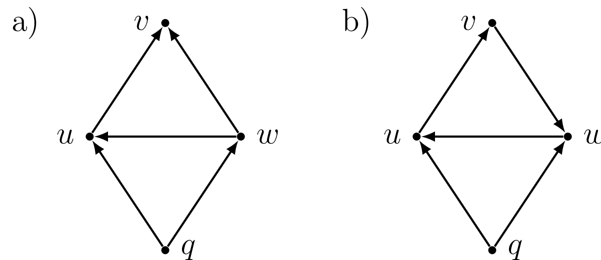


Figure 1.2: (a) An acyclic orientation with unique source q. (b) This orientation is not acyclic, since the set of edges $\{(u,v),(v,w),(w,u)\}$ forms a cycle.

1.3 Converting Acyclic Orientations to Superstables

Now we finally arrive at what was promised in our introduction: a bijection between the unique-source acyclic orientations on a graph and that graph's maximal superstable configurations. We prove that such a bijection exists largely with the help of Dhar's algorithm [3].

1.3.1 Dhar's Algorithm

For a subset W of the vertices V of a graph G and a vertex $v \in V$, let $\mathrm{outdeg}_W(v)$ be the number of edges $\{u, v\}$ in G such that $u \in V \setminus W$. We now describe Dhar's algorithm, as it applies to configurations.

INPUT: A configuration c on the graph G.
OUTPUT: A subset W of vertices of G, possibly empty.
Step 1. Let $W = \widetilde{V} = V \setminus q$.
Step 2. If there exists $v \in W$ such that $c_v < \mathrm{outdeg}_W(v)$, remove v from W.
Step 3. Repeat step 2 until it does not apply anymore (i.e. there is no such v in W).

In the end, W is empty if and only if c is a superstable configuration.

To verify the conclusion, we recall the definition of a superstable configuration: if there exists a subset of vertices of c which can be legally fired, then it is not superstable. At the end of Dhar's algorithm, W represents that very subset—a set of vertices that can be legally fired. Thus, if c is superstable, then W must be empty.

To prove the other direction, suppose that W is empty at the end of the algorithm. Let U be a nonempty subset of $\widetilde{V}$, so at the start of the algorithm $U \subseteq W = \widetilde{V}$. Since W is empty at the end, then through the course of the algorithm, all vertices in U will be removed from W. Let u be the first vertex of U removed from W. Then right before u is removed, we have $c_u < \mathrm{outdeg}_W(u)$. But since we still have $U \subseteq W$, then $\mathrm{outdeg}_W(u) < \mathrm{outdeg}_U(u)$, so $c_u < \mathrm{outdeg}_U(u)$, so U is not a legal firing set. We conclude that c is superstable.

1.3.2 Proof of the Bijection

We define a mapping from $\mathrm{Ao}(G, q)$ to $\mathrm{SS}_{\max}(G)$ as follows: given an orientation $\alpha \in \mathrm{Ao}(G, q)$, we get the configuration

$$c = \mathbf{c}(\alpha) = \sum_{v \in \widetilde{V}} (\mathrm{indeg}_\alpha(v) - 1)v. \tag{1.3}$$

First we verify that, for any unique-source acyclic orientation α, the configuration $c = \mathbf{c}(\alpha)$ is superstable. We run Dhar's Algorithm on c. Start with $W = \widetilde{V}$. There will be at least one vertex v in the neighbors of q that is a "source" in the subgraph on W, i.e, the only edges directed towards v come from outside W (from q).

In this case, $\text{indeg}_\alpha(v) = \text{outdeg}_W(v)$, so clearly $c_v = \text{indeg}_\alpha(v) - 1 < \text{outdeg}_W(v)$; by Step 2 of Dhar's Algorithm, we remove v from W. Then repeat for another "source" vertex of the new W. By construction, this inequality will hold for all vertices, so all vertices will be eventually removed, leaving empty W. Which, by the conclusion of Step 3 of Dhar's algorithm, confirms that $\mathbf{c}(\alpha)$ is superstable.

We will prove that $\mathbf{c}(\alpha)$ is maximal once we have defined some helpful lemmas (Lemma 1.3.1 and Lemma 1.3.2). These lemmas will also be used to prove the existence of the bijection, stated in the following theorem:

Theorem 1.3.1. The mapping $\mathbf{c}\colon \text{Ao}(G, q) \to \text{SS}_{\max}(G)$ is a bijection.

Before proving this, we define an inverse mapping, $\mathbf{a}\colon \text{SS}(G, q) \to \text{Ao}(G, q)$. Again we turn to Dhar.

INPUT: A configuration $c \in \text{SS}(G, q)$.
OUTPUT: An acyclic orientation with unique source q .

Step 1. Let $W = \widetilde{V}$.

Step 2. Choose v in W such that $c_v < \text{outdeg}_W(v)$. Orient all edges accounted for by $\text{outdeg}_W(v)$ in towards v. Remove v from W.

Step 3. Repeat Step 2 until all vertices have been removed from W.

We verify that Step 3 is in fact valid: Since c is superstable, there will always be a v in W that meets the criterion of Step 2, otherwise that subset W could be legally fired, contradicting the definition of superstable. So all vertices will eventually be removed.

We can also verify that the orientation on G produced by this process will be unique-source acyclic. Since at Step 2 we are always orienting the edges in towards W, and since vertices are never added to W, only removed, then we know there will be no cycles in the produced orientation. Moreover, since every vertex in V save for q will be addressed at Step 2 and have at least one edge oriented towards it, then q is the unique source in the final orientation. So we have a mapping $\text{SS}(G, q) \xrightarrow{\mathbf{a}} \text{Ao}(G, q)$.

Now we introduce our lemmas.

Lemma 1.3.1. $c \leq \mathbf{c}(\mathbf{a}(c))$, for all configurations $c \in \text{SS}(G, q)$.

Proof. If we apply the configuration mapping to the orientation $\mathbf{a}(c) \in \text{Ao}(G, q)$, we get $\mathbf{c}(\mathbf{a}(c)) = \sum_{v \in \tilde{V}} (\text{indeg}_{\mathbf{a}(c)}(v) - 1)v$. From Step 2 defining $\mathbf{a}\colon \text{SS}(G, q) \to \text{Ao}(G, q)$, we have that

$$c_v < \text{outdeg}_W(v) \leq \text{indeg}_{\mathbf{a}(c)}(v) = \mathbf{c}(\mathbf{a}(c))_v + 1$$

for all $v \in \widetilde{V}$, so $c \leq \mathbf{c}(\mathbf{a}(c))$.

□

Lemma 1.3.2. $\mathbf{c}(\alpha) \leq \mathbf{c}(\alpha')$ iff $\alpha = \alpha'$, for any $\alpha, \alpha' \in \text{Ao}(G, q)$.

Proof. If $\alpha = \alpha'$, then $\mathbf{c}(\alpha) = \mathbf{c}(\alpha')$. Now suppose $\mathbf{c}(\alpha) \leq \mathbf{c}(\alpha')$; we want to prove $\alpha = \alpha'$.

Consider, $\mathbf{c}(\alpha) \leq \mathbf{c}(\alpha')$ implies $\text{indeg}_\alpha(v) \leq \text{indeg}_{\alpha'}(v)$, for all $v \in V$. But

$$\sum_{v \in \widetilde{V}} (\text{indeg}_\alpha v) = \sum_{v \in \widetilde{V}} (\text{indeg}_{\alpha'} v) = |E|,$$

where $|E|$ is the number of edges of G. This is only possible if $\text{indeg}_\alpha(v) = \text{indeg}_{\alpha'}(v)$. We claim that this is sufficient to show that $\alpha = \alpha'$. To verify this, we describe yet another implementation of Dhar's Algorithm.

INPUT: A unique source acyclic orientation, α.
OUTPUT: A sequence of subsets of vertices, $\text{seq}(\alpha)$.

Step 1. Let $W = V$, $\widehat{W} = [\;]$, $\text{seq}(\alpha) = [\;]$.

Step 2. Let $\widehat{W}$ = all source vertices of W, with respect to α on the subgraph $G(W, E_W)$. Let $W = W \backslash \widehat{W}$. Add $\widehat{W}$ to $\text{seq}(\alpha)$.

Step 3. Repeat Step 2 until all vertices have been removed from W.

The result is a sequence of sets $\widehat{W_i}$ of sources corresponding to a sequence of shrinking subsets W_i of the vertices. This sequence uniquely determines α.

Suppose that this process is run on two orientations, α and α'. To finish the proof, consider that at each iteration of Step 2 for α and α',

$$\text{outdeg}_W(v) = \text{indeg}_\alpha(v) = \text{indeg}_{\alpha'}(v)$$

for all v. But $\text{outdeg}_W(v)$ determines W at each iteration, and thus determines the sequence. So $\text{indeg}_\alpha(v) = \text{indeg}_{\alpha'}(v)$ implies $\text{seq}(\alpha) = \text{seq}(\alpha')$.

Thus, $\text{indeg}_\alpha(v) = \text{indeg}_{\alpha'}(v)$ implies $\alpha = \alpha'$.

□

We are now equipped to verify that $\mathbf{c}(\alpha)$ is maximal, for all α. Suppose $\mathbf{c}(\alpha) \leq c'$ for some $c' \in \text{SS}(G, q)$. By Lemma 1.3.1,

$$\mathbf{c}(\alpha) \leq c' \leq \mathbf{c}(\mathbf{a}(c')).$$

Then, by Lemma 1.3.2, $\alpha = \mathbf{a}(c')$. But then $\mathbf{c}(\alpha) = \mathbf{c}(\mathbf{a}(c'))$, so we must have $\mathbf{c}(\alpha) = c'$. Hence, $\mathbf{c}(\alpha)$ is maximal.

Proof of Theorem 1.3.1. To prove that $\mathbf{c}$ is a bijection, we show that it is onto and one-to-one.

Let $c \in \text{SS}_{\max}(G, q)$. We apply to c the mapping $\mathbf{a}$, then $\mathbf{c}$, to get $\mathbf{c}(\mathbf{a}(c))$. By Lemma 1.3.1 we know $c \leq \mathbf{c}(\mathbf{a}(c))$. But c is maximal, so we must have equality, $c = \mathbf{c}(\mathbf{a}(c))$. Thus, when mapping from $\text{SS}_{\max}(G, q)$, the composition $\mathbf{c} \circ \mathbf{a}$ is the identity mapping. This requires that $\mathbf{a}$: $\text{SS}_{\max}(G, q) \to \text{Ao}(G, q)$ is one-to-one and $\mathbf{c}$: $\text{Ao}(G, q) \to \text{SS}_{\max}(G, q)$ is onto.

Conversely, suppose that $\mathbf{c}(\alpha) = \mathbf{c}(\alpha')$ for some $\alpha, \alpha' \in \text{Ao}(G, q)$. Then by Lemma 1.3.2, we must have $\alpha = \alpha'$. So $\mathbf{c}$ is one-to-one. Thus $\mathbf{c}$ is a bijection.

□

Chapter 2

The Algorithm

2.1 Generating Acyclic Orientations

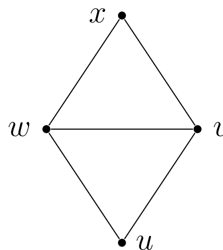
In this chapter we present an algorithm for finding all unique source acyclic orientations of a given graph. We adapt and build on an algorithm developed by Squire [5]. We assume that the graph is simple (i.e, it has no loops or multiple edges) and connected.

To outline the overall process to find the acyclic orientations on a graph with unique-source q: we will remove q from the graph and generate the acyclic orientations on the remaining graph $G\backslash q$. In order for the final output to have q as the only source, we will only keep orientations from this process that have no other sources except for q. Then we will add q back into these graphs as a source the only source.

The core of this process is the generation of the acyclic orientations on the subgraph $G\backslash q$. To develop this process, we will first look at a simple example.

2.1.1 Toy Example

Say we have the graph G defined on vertices $\{u, v, w, x\}$:



Consider the subgraph H that consists of just the edge uw. Then there are exactly two orientations that can be assigned to H, orienting the edge uw either u to w or w to u, and neither of these are cycles; so we say the acyclic orientations of H are $\text{Ao}(H) = \{\{(u, w)\}, \{(w, u)\}\}$. For general notation, let an oriented graph on underlying graph H be denoted by H_α.

Take the edge vx, the only edge in G that is not incident to H. Then the edges

between H and vx are $E_{\text{cross}} = \{uv, vw, wx\}$; let a set of orientations on these edges be denoted $\vec{E}_{\text{cross}}$. Choose one orientation of H, say $H_\alpha = \{(u, w)\}$. Assign an orientation to the edge vx, say (v, x). Since there are so few edges in E_{cross}, we can easily orient these edges so that the ultimate digraph, $G_\alpha = H_\alpha \cup (v, x) \cup \vec{E}_{\text{cross}}$, is acyclic (Figure 2.1).

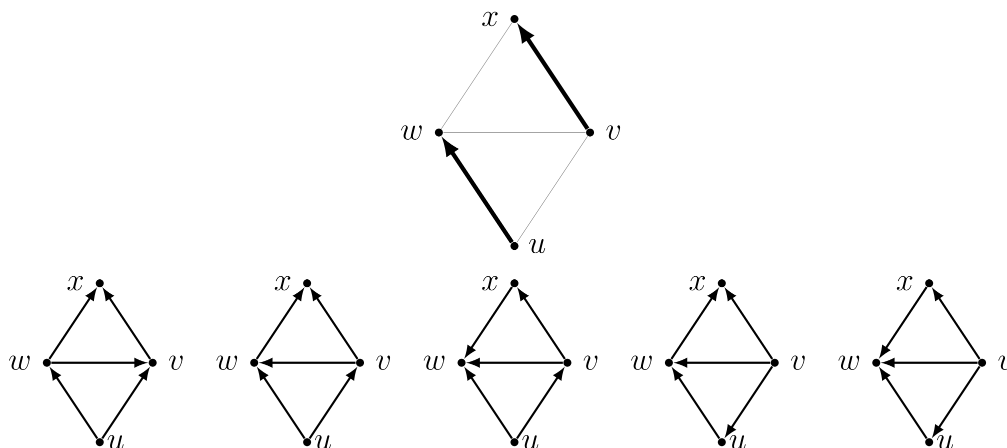


Figure 2.1: Fixing the edge orientations (u, w) and (v, x) gives five possible assignments of orientations on $E_{\text{cross}} = \{uv, vw, wx\}$ such that the orientation on the entire graph is acyclic.

Orienting vx the other way, as (x, v), would give us another set of acyclic orientations on G. Likewise, if we went back to H and chose the other orientation, $H_\alpha = \{(w, u)\}$, we would have two new sets of acyclic orientations, one for each orientation of vx. All these orientations together would make up the complete set $\text{Ao}(G)$ of acyclic orientations on G.

Note that on a more complicated graph with more connecting edges, it would not be so easy to choose directions to assign to E_{cross} so that the resulting orientation is acyclic. For this reason, we codify the selection by defining a relation between edges and using posets, as described in Sections 2.1.2 and 2.1.3.

Returning to our example, now suppose that G is a subgraph of some larger graph F. Then we could repeat the process for directed subgraph G_α, some new edge yz in $F \backslash G$ that is disjoint from uw and vx (i.e, $y, z \notin \{u, v, w, x\}$), and the edges connecting those. In this way a recursive process is established that we can use to determine all acyclic orientations on a set of disjoint edges and the edges that connect them. This process is formally presented in Section 2.1.3.

However, this process is not sufficient for producing the acyclic orientations over the entire graph for every graph. Some graphs do comprise solely a set of disjoint edges and their connecting edges (in which case the process described up to this point would be sufficient) but many graphs do not fit such requirements (the most basic example of this is the complete graph on three vertices). In Section 2.1.4 we expand this process to be sufficient for any simple graph.

2.1.2 Defining a Poset

A *partially ordered set*, or *poset*, comprises a base set S and a relation R on the elements of this set, such that this relation is reflexive, transitive, and antisymmetric. We denote a poset by $\mathcal{P}(S, R)$. The relation imposes an order to the elements of S. If $a \in S$ precedes $b \in S$ under this ordering, we write $a \preceq b$. An upset U is a subset of S such that if a is in U and $a \preceq b$, then b is in U.

Let H_α be an oriented subgraph of G and let J_α be an oriented *complete* subgraph of G. In our case we will be looking at a poset with base set E_{cross}, the unoriented edges between H_α and J_α. Let uv and wx be edges in E_{cross}, where $\{u, w\}$ are vertices of H and $\{v, x\}$ are vertices of J. Then we define the relation $\preceq$ as follows: we will say $uv \preceq wx$ if w is reachable from u and v is reachable from x (Figure 2.2). This defines our poset $\mathcal{P}(E_{\text{cross}}, R)$.

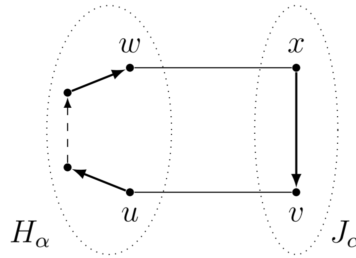


Figure 2.2: We define our poset relation: $uv \preceq wx$ if w is reachable from u in H_α and v is reachable from x in J_α. In this case, J_α comprises just a single directed edge.

Note that in the case where J_α comprises just a single edge vx (as in our toy example, and what we will see in Algorithm 2), then the requirement for v to be reachable from x is equivalent to saying that vx is oriented (x, v). In the case where J_α is a single vertex v (which we will see in Algorithm 3), it is clear that v is always reachable from itself, so the only requirement of consequence for the relation is for w to be reachable from u.

Let $\mathcal{U}$ be the set of all upsets on the poset defined above. Let $\text{Ao}(E_{\text{cross}})$ be all sets of orientations on the edges of E_{cross} such that, for $\vec{E}_{\text{cross}} \in \text{Ao}(E_{\text{cross}})$ the directed graph $G_\alpha = H_\alpha \cup J_\alpha \cup \vec{E}_{\text{cross}}$ is acyclic. We claim that there is a bijection between $\mathcal{U}$ and $\text{Ao}(E_{\text{cross}})$.

Squire [5] provides a proof of this claim for any J_α, but for the purposes of this paper we present a simpler proof for only the case when J_α consists of a single directed edge, say $J = (x, v)$ for vertices x and v of graph G.

First we show that there is an injection from the acyclic orientations $\text{Ao}(G)$ on the graph $G = H \cup \{v, x\} \cup E_{\text{cross}}$ to the upsets $\mathcal{U}$ of our poset $\mathcal{P}(E_{\text{cross}}, R)$. Define the mapping as follows: for some orientation $\alpha \in \text{Ao}(G)$ define U_α to be the set of edges $e \in E_{\text{cross}}$ that are oriented from $\{x, v\}$ to vertices of H. Since our graph is acyclic, by our definition for the poset relation R we must have that U_α is an upset. The mapping is clearly one-to-one.

Now we show an injection exists from the upsets $\mathcal{U}$ to the acyclic orientations $\text{Ao}(G)$. Let $J_\alpha = (x, v)$. We define a mapping from $\mathcal{U}$ to the orientations on G. For $U \in \mathcal{U}$,

we consider each $e \in E_{\text{cross}}$. If $e \in U$, we orient e from (x, v) to H_α. If $e \in E_{\text{cross}} \backslash U$, we orient it from H_α to (x, v). Let G_α be the resulting oriented graph. We show that G_α must be acyclic by contradiction.

Suppose G_α contains a cycle that includes oriented edges e_1 and e_2 in $\vec{E}_{\text{cross}}$. Let u and w be vertices in H such that $e_1 = (v, u)$ and $e_2 = (w, x)$. Then to complete the cycle, w must be reachable from u in H_α. So $uv \preceq wx$. The edge uv is directed (v, u), from $\{x, v\}$ to H, so by our definition of the upset U above $uv \in U$. Then by the general definition of an upset, $wx \in U$. But then wx would also have to be oriented as (x, w), from $\{x, v\}$ to H, which contradicts our original assumption of the cycle. So we conclude G_α is acyclic, giving us our injection. This proves the bijection between $\text{Ao}(G)$ and $\mathcal{U}$.

The proof for when J_α is a single vertex is similar.

2.1.3 The First Piece of Code: Compute_AO

The recursive process illustrated in the toy example forms the cornerstone of the acyclic orientation generation; we describe it in the algorithm, COMPUTE_AO (Algorithm 1). The algorithm takes as input the graph $G = G(E, V)$, a set of disjoint edges $E_{\text{disj}} \subseteq E$, the set of all acyclic orientations on a subgraph H of G, denoted by $\text{Ao}(H)$, and some subset of vertices $N \subseteq V$. Let E' denote the set of all edges between separate elements of E_{disj} and all edges between elements of E_{disj} and H. The algorithm returns as output the set of all acyclic orientations on the subgraph $G' = H \cup E_{\text{disj}} \cup E'$.

We will explain the purpose of the the subset of vertices N shortly. COMPUTE_AO does not perform any operations directly on N, merely carries it through the recursion and feeds it to the helper function AO_EDGE.

For each edge uv in E_{disj}, we take some acyclic orientation $H_\alpha \in \text{Ao}(H)$ and an orientation (u, v) on uv, then generate all the orientations on the edges between uv and H_α. To do this we write the helper function AO_EDGE (Algorithm 2). Then we repeat this with the same orientation H_α but with uv oriented (v, u). If we do this for every orientation in $\text{Ao}(H)$, we produce the set of acyclic orientations on the subgraph $H' = H \cup \{uv\} \cup E_{\text{cross}}$, where $E_{\text{cross}} \subseteq E'$ is the set of all edges between H and $\{uv\}$. Then we recursively perform COMPUTE_AO again with this new set of acyclic orientations $\text{Ao}(H')$ and with $E_{\text{disj}} \backslash \{uv\}$. (Note: in Algorithm 1, $\text{Ao}(H')$ is denoted by next_Ao). The recursive process ends when all edges have been removed from E_{disj}.

The helper function AO_EDGE (Algorithm 2) takes the oriented subgraph H_α of G and an edge in $G \backslash H$ with a given orientation (x, v), and returns all orientations on the cross edges, E_{cross}, between the vertices of (x, v) and of H_α.

The first part of the code (Lines 3-12) assigns the relations on the edges E_{cross} that will be used in the edge poset $\mathcal{P}(E_{\text{cross}}, R)$. It is the relation R described in the previous section, with the subgraph J_α being the single edge (x, v): for directed edges (u, v) and (w, x), we will say $(u, v) \preceq (w, x)$ if w is reachable from u and v is reachable from x. We then define a poset on the edges with this relation (Line 13). Each upset of this poset corresponds to a unique acyclic orientation, as we described

Algorithm 1 COMPUTE_AO $(G, E_{\text{disj}}, \text{Ao}(H), N)$.

```
1: if E_disj ≠ [ ] then
2:     choose some uv ∈ E_disj
3:     next_Ao := [ ]
4:     for H_α ∈ Ao(H) do
5:         Ao1 := AO_EDGE(G, (u, v), H_α, N)
6:         Ao2 := AO_EDGE(G, (v, u), H_α, N)
7:         next_Ao := next_Ao ∪ Ao1 ∪ Ao2
8:     Ao(G) := COMPUTE_AO(G, E_disj\{uv}, next_Ao, N)
9: return Ao(G)
```

and proved in Section 2.1.2: for a given upset U, all edges in U orient from (x, v) to H_α, while all edges in E_{cross} not in U orient from H_α to (x, v) (Lines 21-24).

Here is where the subset of vertices N becomes relevant. The ultimate purpose of this algorithm is to generate all *unique-source* acyclic orientations. Recall that the process COMPUTE_AO will be performed on a graph with the intended source removed: $G \backslash q$. Then q will be added as a source to each acyclic graph at the end. To ensure that q is the only source in the final product, we want COMPUTE_AO to return only graphs which have no sources, save for the neighbors of q which will have edges directed to them from q and so will not be sources when q is added back in.

We define the subset $N \subseteq V$ to be the set of neighbors of q. Let $\deg_G(x)$ be the number of edges incident to a vertex x on an undirected graph G and let $\deg_\alpha(x)$ be the number of oriented edges incident to a vertex x under a (maybe partial) orientation α. For an arbitrary directed edge $e = (x, v)$, let $e^- = x$.

Then, when we add a directed edge e to an orientation, we check the following (lines 17-22, 32-36): if e^- is not in N and if all of its incident edges have been assigned a direction (i.e, $\deg_G(e^-) = \deg_\alpha(e^-)$) then we check that e^- is not a source (i.e, that $\text{indeg}_\alpha(e^-)$ is greater than zero). If this condition is not met, we do not keep that given orientation in our results.

Algorithm 2 AO_EDGE $(G, (x, v), H_\alpha, N)$.

```
1: E_cross := all edges between vertices {x, v} and the vertices of H_α
2: R := [ ]    // The relations for the poset.
3: for all vertices u in H_α that are neighbors to v do
4:     for all vertices w reachable from u do
5:         if vw ∈ E_cross then
6:             add vu ⪯ vw to R
7:         if xw ∈ E_cross then
8:             add vu ⪯ xw to R
9: for all vertices u in H_α that are neighbors to x do
10:    for all vertices w reachable from u do
11:        if xw ∈ E_cross then
12:            add xu ⪯ xw to R
13: 𝒰 := all upsets of the poset 𝒫(E_cross, R)
14: result := [ ]
15: add edge (x, v) to H_α
16: if E_cross = [ ] then
17:     keep := True
18:     if x ∉ N and deg_G(x) = deg_α(x) then
19:         if indeg_α(x) = 0 then
20:             keep := False
21:     if keep = True then
22:         append H_α to result
23: else
24:     for S ∈ 𝒰 do
25:         keep := True
26:         for e ∈ E_cross do
27:             if e ∈ S then
28:                 orient e from (x, v) to H_α
29:             else
30:                 orient e from H_α to (x, v)
31:             add edge e to H_α
32:             if e⁻ ∉ N and deg_G(e⁻) = deg_α(e⁻) then
33:                 if indeg_α(e⁻) = 0 then
34:                     keep := False
35:         if keep = True then
36:             append H_α to result
37: return result
```

2.1.4 Expanding the Code for the Entire Graph

As we noted previously, many graphs comprise more than just a set of disjoint edges and their connecting edges. We can run COMPUTE_AO on such a graph and a disjoint subset of its edges, but we will be left with some extra vertices and their incident edges that are not assigned an orientation. To address this problem, we write another code for these left-over vertices.

AO_VERTEX (Algorithm 3) works similarly to AO_EDGE, but instead of taking as input an edge it takes a single vertex. The methods for establishing relations for the edge poset, for assigning orientation from the upsets of this poset, and for removing orientations with multiple sources are the same.

Algorithm 3 AO_VERTEX (G, v, H_α, N).

1: E_{cross} := all edges between vertices $\{x, v\}$ and the vertices of H_α
2: R := [ ]
3: **for** all vertices u in H_α that are neighbors to v **do**
4:     $e := (v, u)$
5:     **for** all vertices w reachable from u **do**
6:         **if** $(v, w) \in E_{\text{cross}}$ **then**
7:             add $(v, u) \preceq (v, w)$ to R
8: $\mathcal{U}$:= all upsets of the poset $\mathcal{P}(E_{\text{cross}}, R)$
9: result := [ ]
10: **for** S $\in \mathcal{U}$ **do**
11:     keep := True
12:     **for** $e \in E_{\text{cross}}$ **do**
13:         **if** $e \in$ S **then**
14:             orient e from v to H_α
15:         **else**
16:             orient e from H_α to v
17:         add edge e to H_α
18:     **if** $e^- \notin N$ and $\deg_G(e^-) = \deg_\alpha(e^-)$ **then**
19:         **if** $\text{indeg}_\alpha(e^-) = 0$ **then**
20:             keep := False
21:     **if** keep = True **then**
22:         append H_α to result
23: **return** result

In order to use AO_EDGE and AO_VERTEX, we need to identify a set of disjoint edges and the set of edges of remaining vertices. This process (codified in Algorithm 4: STARTSETS) works as follows: remove an edge e of a graph G and its corresponding vertices (so any edges incident to e are also removed from G), and add e to a set E_{disj}. Repeat this process until there are no edges left. Then whatever vertices remain form their own set V_o.

The next step is to combine our separate algorithm components – COMPUTE_AO, AO_EDGE, AO_VERTEX, and STARTSETS – into one process.

Algorithm 4 STARTSETS($G(V, E)$).

1: $E_{\text{disj}} := [\,]$
2: $V_o := V$
3: **for** $uv \in E$ **do**
4: remove uv from E
5: append uv to E_{disj}
6: delete vertices $\{u, v\}$ from V_o
7: **return** $[E_{\text{disj}}, V_o]$

2.1.5 Unique Source Acyclic Orientations

We assemble our algorithm components in Algorithm 5. Given a graph and source q as input, we remove q from G (line 2). Working with this subgraph, $G' = G\backslash q$, we run STARTSETS to determine the set of disjoint edges E_{disj} and the remaining vertices V_o.

Then COMPUTE_AO performs its recursive function, building from an empty digraph as its first directed subgraph, to generate all acyclic orientations on the subgraph covered by edges of E_{disj}. Next we add the vertices of V_o by applying AO_VERTEX to each vertex v in V_o for each acyclic orientation generated by COMPUTE_AO.

Finally, we add back in the vertex q with all its incident edges directed away from it, making it a source. This gives us our generator for all acyclic orientation with a fixed unique source of any simple, connected graph.

Algorithm 5 ACYCLIC_ORIENTATIONS_WITH_SOURCE(G, q).

1: source_neighbors := neighbors of q in G
2: $G' = G\backslash q$
3: $E_{\text{disj}} :=$ STARTSETS$(G')[0]$
4: $V_o :=$ STARTSETS$(G')[1]$
5: $D_\alpha :=$ an empty digraph
6: Ao(G') := COMPUTE_AO$(G', E_{\text{disj}}, \{D_\alpha\}$,source_neighbors)
7: **for** $v \in V_o$ **do**
8: Ao_helper := [ ]
9: **for** $H_\alpha \in$ Ao(G') **do**
10: Ao(v) := AO_VERTEX$(G', v, H_\alpha$,source_neighbors)
11: Ao_helper := Ao_helper ∪ Ao(v)
12: Ao(G') := Ao_helper
13: **for** $G_\alpha \in$ Ao(G') **do**
14: **for** $u \in$ source_neighbors **do**
15: add oriented edge (q, u) to Ao(G)
16: **return** Ao(G)

2.2 Conversion to Superstables

2.2.1 Maximal Superstables

We now return to the bijection between acyclic orientations and maximal superstables introduced in Section 1.3.1: $c_v = \text{indeg}(v) - 1$, for all $v \in V$. Our function FIND_MAX_SUPERSTABLES (Algorithm 6) applies this mapping, taking in a graph and a source and returning all maximal superstable configurations (as vectors of coefficients of the configurations).

Algorithm 6 FIND_MAX_SUPERSTABLES$(G(V, E), q)$.

```
1: Ao(G, q) := ACYCLIC_ORIENTATIONS_WITH_SOURCE(G, q)
2: SS_max := [ ]
3: for G_α ∈ Ao(G, q) do
4:     c := [ ]
5:     for v ∈ V do
6:         c_v := indeg(v) − 1
7:         append c_v to c
8:     append c to SS_max
9: return SS_max
```

2.2.2 Find All Superstables

Having produced the maximal superstables of G, it is a simple task to find all superstable configurations of G from there. Let DOWN be a function that, when given an ordered list c of integers, produces all lists of integers that are component-wise less than or equal to c, i.e, $c_i \geq c'_i$ for all i. The final piece of our code is then as follows:

Algorithm 7 FIND_SUPERSTABLES(G, q).

```
1: SS_max := FIND_MAX_SUPERSTABLES(G, q)
2: SS(G, q) := DOWN(SS_max)
3: return SS(G, q)
```

Chapter 3

Testing for Time

Recall, one of our motivations was to create an algorithm that will produce the superstable configurations on a graph faster than the algorithm already implemented in Sage. To see if we achieved this, we run our new algorithm and the pre-existing algorithm on a variety of graphs and compare the run times between the two.

3.1 The Pre-Existing Code

Before comparing our code to the pre-existing Sage code, we briefly look at how the pre-existing code works [6].

The pre-existing algorithm produces the superstables via the *recurrent configurations*. To understand recurrent configurations, we consider our graph and the distribution of dollars amongst vertices as a system that can have dollars added to it from the outside. If we continually add dollars to non-sink vertices, and then stabilize the resulting configuration, there are a finite number of stable configurations we can get. If we did this many times, there would be some stable configurations that would appear again and again—these are the recurrent configurations.

Formally, a stable configuration c is *recurrent* if, for any configuration c', there exists some nonnegative configuration c'' such that the stabilization of $c' + c''$ (the component-wise addition of the two configurations) is c.

There is a duality between the recurrent configurations and the superstable configurations. Let $c_{\max}$ be the maximal stable configuration. Then some configuration c is superstable if and only if $c_{\max} - c$ is recurrent [4]. Since the zero configuration ($c_v = 0$ for all v) is superstable, then $c_{\max}$ is itself recurrent.

The pre-existing code computes the recurrent configurations as follows: start with $c_{\max}$. For all $v_i \in \widetilde{V} = V \backslash q$, stabilize $c_{\max} + v_i$. The resulting stable configurations form a set R of recurrent configurations. For each $c \in R$, we stabilize $c + v_i$ for all $v_i \in \widetilde{V}$, which gives us a new set of recurrent configurations. We repeat this process until a round of stabilization produces no new recurrent configurations.

Once the code has this set of recurrent configurations, it uses the above described duality to produce all superstable configurations from the recurrents.

We will refer to that pre-existing code as the recurrents code (RC), and our new

code as the acyclic orientations code (AOC).

3.2 The Results

We run the recurrents code and the acyclic orientations code on four types of graphs and compare the runtimes. The basic graphs we look at are the complete graph, the wheel graph with the center vertex as the source, the wheel graph with an outside vertex as the source (see Figure 3.1), and the cycle graph—each over a range of different numbers of vertices $n = |V|$.

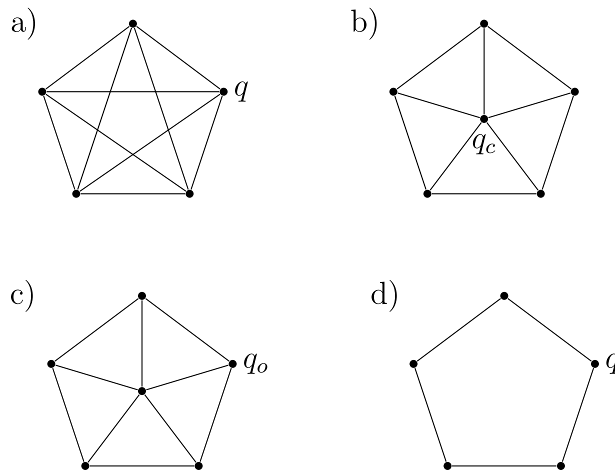


Figure 3.1: a) A complete graph on $n = 5$ vertices with source q at an arbitrary vertex; b) a wheel graph on $n = 6$ vertices with source q_c at center vertex; c) a wheel graph on $n = 6$ vertices with source q_o at an arbitrary outer vertex; d) a cycle graph on $n = 5$ vertices with source q at an arbitrary vertex.

The runtime results are presented in Table 3.1. We record the number of vertices ($|V| = n$), the number of edges ($|E|$), and the runtimes in seconds for each code (AOC for the acyclic orientations code and RC for the recurrents code). We also record the number of edges on the graph when the sink is removed, since our code works by generating all acyclic orientations on $G \backslash q$ and then adding q as the source.

We see that our acyclic orientations algorithm consistently runs faster for $n > 4$ on the complete graph and wheel graph. For the cycle graph, the recurrents algorithm is faster for smaller n, but at some number of vertices $n > 30$, the acyclic orientations code achieves the faster runtime.

Table 3.1: Acyclic Orientations Code Run Times (AOC) vs. Recurrents Code Run Times (RC) in Seconds, on Different Types of Graphs

Graph Type	n	$\|E\|$	$\|E_{\setminus\{q\}}\|$	AOC (s)	RC (s)
Complete	3	3	1	0.01	0.00
	4	6	3	0.01	0.00
	5	10	6	0.04	0.05
	6	15	10	0.53	1.41
	7	21	15	65.87	152.83
Wheel (q = center)	3	3	1	0.01	0.00
	5	8	4	0.02	0.02
	7	12	6	0.10	0.25
	9	16	8	1.92	5.27
	10	18	9	8.40	30.64
Wheel (q = outside)	3	3	1	0.01	0.00
	5	8	5	0.03	0.02
	7	12	9	0.17	0.31
	9	16	13	2.15	6.81
	10	18	15	7.73	39.79
Cycle	3	3	1	0.01	0.00
	5	5	3	0.01	0.00
	7	7	5	0.04	0.01
	9	9	7	0.09	0.02
	11	11	9	0.16	0.04
	13	13	11	0.24	0.07
	20	20	18	0.66	0.37
	30	30	28	1.85	1.71
	40	40	38	3.44	5.41
	50	50	48	5.80	12.26
	60	60	58	9.84	28.56

Conclusion

We produced an algorithm to generate the superstable configurations of a graph via its unique-source acyclic orientations. One of our objectives was for this algorithm to operate faster than the already existing algorithm that generates the superstables via the recurrent configurations in Sage. Our runtime results show that we were successful in this endeavor.

However, there may be ways to improve our algorithm to make it run faster yet. We propose some possible inefficiencies with our algorithm and suggested improvements:

- Our code relies on certain functions that are already built into Sage—notably, in Algorithm 2, to find all vertices reachable from a given vertex (Lines 4 and 10) we rely on the breadth-first-search function in Sage, and for calculating the upsets (Line 13) we rely on the Sage function that produces the order ideals of a poset. The process that happens in these functions could be picked apart and incorporated into the process of the code to be more efficient. For instance, instead of performing a breadth-first-search from every vertex, it is possible that we could build on the reachability we have already determined, i.e, if u is reachable from v and w is reachable from u, then we know that w is reachable from v.

- The code may run faster if written in a different programming language, such as C, instead of the Python-based Sage.

Appendix A

The Code (in Sage programming language)

Here is the exact code for our algorithm.

```
def _compute_ao(G,Edisj,Ao_H,N):
    r"""
    Given Edisj, a set of disjoint edges of G, and Ao_H, a set of digraphs
    on some subset of G, recursively find all acyclic orientations of G
    that contain some element of Ao_H.

    INPUT:

        G -- Graph
        Edisj -- Set of non-adjacent edges
        Ao_H -- set of digraphs
        N -- a subset of vertices

    OUTPUT:

        Array of acyclic orientations on the vertices

    EXAMPLES::

        sage: g = graphs.CompleteGraph(4)
        sage: h = DiGraph()
        sage: Aos = [h]
        sage: e = [(0,1),(2,3)]
        sage: result = compute_ao(g,e,Aos,N)
        sage: for i in range(len(result)):
                      result[i].show(layout='circular',figsize=2)

    """
    if Edisj != []:
        q1 = list(Edisj[0])[0]
        q2 = list(Edisj[0])[1]
        next_Ao = []
        for Ha in Ao_H:
```

```
            Ao1 = _ao_edge(G,(q1,q2),Ha,N)
            Ao2 = _ao_edge(G,(q2,q1),Ha,N)
            next_Ao += Ao1 + Ao2
        Edisj.remove(Edisj[0])
        Ao_H = _compute_ao(G,copy(Edisj),copy(next_Ao),N)
    return Ao_H

-----

def _ao_edge(G,(x,v),Ha,N):
    r"""
    Given an acyclic orientation Ha on a subgraph of G and the oriented
    edge (x,v) of G, find all the ways of orienting the edges between Gp
    and (x,v) so that the resulting subgraph is acyclic.

    INPUT:

        G -- Graph
        Ha -- DiGraph (with underlying undirected graph a subgraph of G)
        (x,v) -- vertices of G, not in Gp, sharing an edge oriented from x to v
        N -- a subset of vertices

    OUTPUT:

        list of DiGraphs

    EXAMPLES::

        sage: G = graphs.CompleteGraph(4)
        sage: h = DiGraph()
        sage: h.add_edge((0,1))
        sage: aos = ao_edge(G,(2,3),h,N)
        sage: for i in range(len(aos)):
                    aos[i].show(layout='circular',figsize=2)
    """
    E = [set(e) for e in G.edges(false)]
    # We want to create a poset of all the edges connecting Gp and (x,v)
    cross_edges = []
    relations = [] # relations for the edge poset
    verts = [u for u in G.neighbor_iterator(v) if Ha.has_vertex(u)]
    # We define the relation: (v,u) <= (x,w) if x is reachable from v and u is
    #    reachable from w.
    for u in verts: # for all vertices u that are neighbors to v
        e = E.index(set((v,u)))
        for w in Ha.breadth_first_search(u):  # for all vertices reachable from u
            if G.has_edge((v,w)):
                relations.append([e,E.index(set((v,w)))])
                if not set((v,w)) in cross_edges:
                    cross_edges.append(set((v,w)))
            if G.has_edge((x,w)):
                relations.append([e,E.index(set((x,w)))])
                if not set((x,w)) in cross_edges:
                    cross_edges.append(set((x,w)))
    verts = [u for u in G.neighbor_iterator(x) if Ha.has_vertex(u)]
```

```
for u in verts:  # for all vertices u that are neighbors to x
    e = E.index(set((x,u)))
    for w in Ha.breadth_first_search(u):  # for all vertices reachable from u
        if G.has_edge((x,w)):
            relations.append([e,E.index(set((x,w)))])
            if not set((x,w)) in cross_edges:
                cross_edges.append(set((x,w)))
elts = [E.index(i) for i in cross_edges]
P = Poset([elts, relations])
upsets = P.dual().order_ideals_lattice().list()
result = []     # set of DiGraphs - the orientations corresponding
 #  to the upsets
if cross_edges==[]:
    keep = True
    Ho = copy(Ha)
    Ho.add_edge((x,v))
    if x not in N and G.degree(x) == Ho.degree(x):
        if Go.in_degree(x) == 0:
            keep = False
    if keep == True:
        result.append(Ho)
# If there are edges between (x,v) and Ha, we need to assign orientations
# to those. Each upset in the edge poset corresponds to a unique acyclic
# orientation.
else:
    for up in upsets:
        Ho = copy(Ha)
        Ho.add_edge((x,v))
        for i in range(len(cross_edges)):
            q1 = list(cross_edges[i])[0]
            q2 = list(cross_edges[i])[1]
            if elts[i] in up:   # i is a set of edges; cross_edges[i] is
             #   a set of vertices.
                if q1 in (x,v):    # Appropriate orientation for edges in
                 #   upset is from {x,v} to Ha.
                    Ho.add_edge((q1,q2))
                    if q1 not in N and G.degree(q1) == Ho.degree(q1):
                        if Ho.in_degree(q1) == 0:
                            keep = False
                else:
                    Ho.add_edge((q2,q1))
                    if q2 not in N and G.degree(q2) == Ho.degree(q2):
                        if Ho.in_degree(q2) == 0:
                            keep = False
            else:
                if q1 in (x,v):    # Appropriate orientation for edges not
                 #   in upset is from Ha to {x,v}
                    Ho.add_edge((q2,q1))
                    if q2 not in N and G.degree(q2) == Ho.degree(q2):
                        if Ho.in_degree(q2) == 0:
                            keep = False
                else:
                    Ho.add_edge((q1,q2))
                    if q1 not in N and G.degree(q1) == Ho.degree(q1):
```

```
                            if Ho.in_degree(q1) == 0:
                                keep = False
            if keep == True:
             result.append(Ho)
    return result

-----

def _ao_vertex(G,v,Ha,N):
    r"""
    Given an acyclic orientation Ha on a subgraph of G and a vertex v in G,
    find all the ways of orienting the edges between Ha and v so that the
    resulting subgraph is acyclic.

    INPUT:

        G -- Graph
        Ha -- DiGraph (with underlying undirected graph a subgraph of G)
        v -- vertices of G, not in Gp
        N -- subset of vertices

    OUTPUT:

        list of DiGraphs

    EXAMPLES::

        sage: G = graphs.CompleteGraph(3)
        sage: h = DiGraph()
        sage: h.add_edge((0,1))
        sage: aovs = ao_vertex(G,2,h,N)
        sage: for i in range(len(aovs)):
                  aovs[i].show(layout='circular',figsize=2)
    """
    E = [set(e) for e in G.edges(false)]
    cross_edges = []
    relations = [] # relations for the edge poset
    verts = [u for u in G.neighbor_iterator(v) if Ha.has_vertex(u)]
    for u in verts:
        e = E.index(set((v,u)))
        for w in Ha.breadth_first_search(u):
            if G.has_edge((v,w)):
                relations.append([e,E.index(set((v,w)))])
                if not set((v,w)) in cross_edges:
                    cross_edges.append(set((v,w)))
    elts = [E.index(i) for i in cross_edges]
    P = Poset([elts, relations])
    upsets = P.dual().order_ideals_lattice().list()
    result = []        # set of DiGraphs - the orientations corresponding
     #   to the upsets.
    # Each upset in the edge poset corresponds to a unique acyclic orientation.
    for up in upsets:
        Ho = copy(Ha)
        for i in range(len(cross_edges)):
```

```
            q1 = list(cross_edges[i])[0]
            q2 = list(cross_edges[i])[1]
            if elts[i] in up:  # i is a set of edges; cross_edges[i] is a
             #   set of vertices
                if q1==v:    # Appropriate orientation for edges in upset
                 # is from v to Ha.
                    Ho.add_edge((q1,q2))
                    if q1 not in N and G.degree(q1) == Ho.degree(q1):
                            if Ho.in_degree(q1) == 0:
                                keep = False
                else:
                    Ho.add_edge((q2,q1))
                    if q2 not in N and G.degree(q2) == Ho.degree(q2):
                            if Ho.in_degree(q2) == 0:
                                keep = False
            else:
                if q1==v:    # Appropriate orientation for edges not in upset
                 #   is from Ha to v.
                    Ho.add_edge((q2,q1))
                    if q2 not in N and G.degree(q2) == Ho.degree(q2):
                            if Ho.in_degree(q2) == 0:
                                keep = False
                else:
                    Ho.add_edge((q1,q2))
                    if q1 not in N and G.degree(q1) == Ho.degree(q1):
                            if Ho.in_degree(q1) == 0:
                                keep = False
        if keep == True:
            result.append(Ho)
    return result

-----

def _startsets(G):
    r"""
    For a graph G, find a set of disjoint edges (edge_list) and all
    remaining vertices V when these edges are removed from G.

    INPUT:

        G -- Graph

    OUTPUT:

        [edge_list,V] -- pair: set of edges, list of integers

    Examples::

        sage: G = graphs.CycleGraph(5)
        sage: s = startsets(G)
        sage: s[0]
        [set([0, 1]), set([2, 3])]
        sage: s[1]
        [4]
```

```
    """
    H = copy(G)
    E = [set(e) for e in H.edges(false)]
    edge_list = []
    while not E == []:
        edge_list.append(E[0])
        q1 = list(E[0])[0]
        q2 = list(E[0])[1]
        H.delete_vertices([q1,q2])
        E = [set(e) for e in H.edges(false)]
    V = H.vertices()
    return [edge_list,V]

-----

def acyclic_orientations_with_source(G,s):
    r"""

    Find all acyclic orientations on G with a given unique source s.

    INPUT:

        G -- graph
        s -- vertex of G

    OUTPUT:

        set of DiGraphs

    EXAMPLES::

        sage: g = graphs.CycleGraph(4)
        sage: g.add_edge((1,3))
        sage: s = 0
        sage: result = acyclic_orientation_with_source(g,s)
        sage: for i in range(len(result)):
                      result[i].show(layout='circular',figsize=2)

    """
    N = G.neighbors(s)
    # Remove the source vertex.
    G_s = copy(G)
    G_s.delete_vertex(s)
    # Find perfect matching edges and remaining, unmatched vertices.
    Start = _startsets(G_s)
    Edisj = Start[0]
    Vo = Start[1]
    Ha = DiGraph()
    aos = [Ha]
    # Compute all acyclic orientations on perfect matching edges (on G w/o s).
    Ao = _compute_ao(G_s,Edisj,aos,N)
    # Add in the lone vertices with appropriate orientations
    for v in Vo:
```

```
        Ao2 = []
        while Ao != []:
            aovs = _ao_vertex(G,v,Ao[0],N)
            for a in aovs:
                Ao2.append(a)
            Ao.remove(Ao[0])
        Ao = copy(Ao2)
    result = Ao
    # Now add back in the source vertex.
    for i in range(len(result)):
        for u in source_neighbors:
            result[i].add_edge((s,u))
    return result

-----

def find_max_superstables(G,s):
    r"""
    Return the coefficients of all maximal superstable configurations.

    INPUT:

        G - Graph
        s - source

    OUTPUT:

        list of list of integers -- each list is the coefficients of a max
        superstable

    EXAMPLES::

        sage: g = graphs.CycleGraph(4)
        sage: g.add_edge((1,3))
        sage: s = 0
        sage: find_max_superstables(g,s)
        [[-1, 0, 0, 2], [-1, 0, 1, 1], [-1, 1, 1, 0], [-1, 2, 0, 0]]

    """
    select_aos = acyclic_orientations_with_source(G,s)
    result = []
    for Ha in select_aos:
        config = []
        for v in Ha.vertices():
            cv = Ha.in_degree(v) - 1
            config.append(cv)
        result.append(config)
    return result

-----

def _down(M):
    r"""
    Create a list of all integer lists that are component-wise less than
```

```
    some element in the list M.

    NOTE: all integer lists need to be same length.

    INPUT:

        M -- list of lists of integers

    OUTPUT:

        list of list of integers.

    EXAMPLES::

        sage: m = [[-1, 0, 2], [0, 1, 1]]
        sage: down(m)
        [[-1, 0, 2], [0, 1, 1], [0, 0, 1], [0, 1, 0], [0, 0, 0], [-1, 0, 1],
        [-1, 0, 0]]

    """
    result = copy(M)
    active = copy(M)
    if M == []:
        return None
    else:
        n = len(M[0])
    while active != []:
        v = active.pop()
        for i in range(n):
            if v[i] > 0:
                w = copy(v)
                w[i] -= 1
                if not w in result:
                    result.append(w)
                    active.append(w)
    return result

-----

def find_superstables(G,s):
    r"""
    Return the coefficients of all superstable configurations.

    INPUT:

        G - Graph
        s - source

    OUTPUT:

        list of list of integers

    EXAMPLES::
```

```
        sage: g = graphs.CycleGraph(4)
        sage: g.add_edge((1,3))
        sage: s = 0
        sage: find_superstables(g,s)
        [[-1, 0, 0, 2], [-1, 0, 1, 1], [-1, 1, 1, 0], [-1, 2, 0, 0],
        [-1, 1, 0, 0], [-1, 0, 0, 0], [-1, 0, 1, 0], [-1, 0, 0, 1]]

    """
    mss = find_max_superstables(G,s)
    result = _down(mss)
    return result
```

References

[1] Matthew Baker and Farbod Shokrieh. Chip-firing games, potential theory on graphs, and spanning trees. *J. Combin. Theory Ser. A*, 120(1):164–182, 2013.

[2] Brian Benson, Deeparnab Chakrabarty, and Prasad Tetali. G-parking functions, acyclic orientations and spanning trees. *Discrete Math.*, 310(8):1340–1353, 2010.

[3] Deepak Dhar. Theoretical studies of self-organized criticality. *Phys. A*, 369(1):29–70, 2006.

[4] Alexander E. Holroyd, Lionel Levine, Karola Mészáros, Yuval Peres, James Propp, and David B. Wilson. Chip-firing and rotor-routing on directed graphs. In *In and out of equilibrium. 2*, volume 60 of *Progr. Probab.*, pages 331–364. Birkhäuser, Basel, 2008.

[5] Matthew B. Squire. Generating the Acyclic Orientations of a Graph. *J. Algorithms*, 26(2):275–290, 1998.

[6] W. A. Stein et al. *Sage Mathematics Software (Version 6.1.1)*. The Sage Development Team, 2014. `http://www.sagemath.org`.